

UNIVERSIDAD POLITECNICA DE MADRID



PROYECTO DE FIN DE GRADO
GRADO EN INGENIERÍA DE SOFTWARE

PARALLELIZED OPTIMIZATION FOR GRADIENT-BASED MACHINE LEARNING
ALGORITHMS IN DEEP NETWORKS

AUTOR: IGNACIO BASARTE FERNÁNDEZ

ESCUELA TECNICA SUPERIOR DE INGENIERIA DE SISTEMAS
INFORMATICOS

UNIVERSIDAD POLITECNICA DE MADRID



PROYECTO DE FIN DE GRADO
GRADO EN INGENIERÍA DE SOFTWARE

PARALLELIZED OPTIMIZATION FOR GRADIENT-BASED MACHINE LEARNING
ALGORITHMS IN DEEP NETWORKS

AUTOR: IGNACIO BASARTE FERNÁNDEZ

TUTORES:

ALBERTO MOZO VELASCO

SANDRA GÓMEZ CANAVAL

ESCUELA TECNICA SUPERIOR DE INGENIERIA DE SISTEMAS
INFORMATICOS

Resumen

Los algoritmos de Machine Learning se benefician de la gran cantidad de datos disponible. Cuanto mayor sea el conjunto de datos que se utiliza, mejor será el entrenamiento. Sin embargo, esto implica que se requieran cada vez más tiempo y recursos para obtener resultados.

Una forma de paliar esta limitación es buscar formas alternativas para optimizar algunas tareas realizadas por algoritmos de Machine Learning. Otra forma de optimizar estos procesos consiste en recurrir a las plataformas de computación distribuida que ofrecen la posibilidad de escalar recursos para afrontar la necesidad de un alto consumo de recursos computacionales.

Sin embargo, ante esta última posibilidad surge un problema que está relacionado con los algoritmos que se utilizan para la fase de entrenamiento de los datos. Estos algoritmos son de naturaleza iterativa, es decir, cada paso depende del anterior y por lo tanto no hay una forma natural o directa de paralelizar estos pasos.

En este Trabajo de Fin de Grado se abordará el problema de la paralelización de algunas tareas dentro de los algoritmos de Machine Learning. En particular, se hará un estudio del arte sobre el problema abordando las distintas aproximaciones y soluciones que se han planteado en la literatura, estudiando su viabilidad y probando las más prometedoras para paliar las limitaciones existentes. Adicionalmente, en este Trabajo se implementó una aplicación con Keras sobre TensorFlow Distribuido, con el fin de codificar las soluciones seleccionadas y comprobar, de forma práctica, la viabilidad de un enfoque paralelo y distribuido para solucionar las limitaciones antes mencionadas. Finalmente, se introduce un análisis sobre los resultados obtenidos, las soluciones implementadas y las conclusiones obtenidas.

Abstract

Machine Learning algorithms benefit from the large amount of data available. The larger the datasets used, the better the training. However, this implies a growth in the time and resources required to obtain results.

One way to alleviate this limitation is to look for alternative ways to optimize some tasks performed by Machine Learning algorithms. Another way to optimize these processes is to resort to distributed computing platforms that offer the possibility of scaling up resources to meet the need for high consumption of computational resources.

However, against this last possibility, a problem related to the algorithms that are used for the training phase of the data arises. These algorithms are iterative in nature, that is, each step depends on the previous one and therefore there is no natural or direct way to parallelize these steps.

In this Final Project, the problem of the parallelization of some tasks within the Machine Learning algorithms will be addressed. In particular, a study of the art on the problem will be made by addressing the different approaches and solutions that have been raised in the literature, studying their feasibility and testing the most promising in mitigating the existing limitations. Additionally, an application with Keras over Distributed TensorFlow was implemented in this work, in order to test the selected solutions and check, in a practical way, the viability of a parallel and distributed approach to solve the aforementioned limitations. Finally, an analysis on the results obtained, the implemented solutions and the conclusions obtained is introduced.

Index

Resumen	III
Abstract	v
Index	vii
Index of Figures	x
1. Introduction	1
1.1. Specific Goals	2
1.2. Motivation and justification	3
1.3. Document structure	4
2. Background	5
2.1. Gradient Descent	6
2.1.1. Stochastic Gradient Descent	8
2.1.2. Mini-batch Gradient Descent	9
2.2. Distributed and parallel computational frameworks	10
2.2.1. TensorFlow	10
2.2.2. Keras	12
2.2.3. Apache Spark	12
3. State of the art	14

4. Design and development	21
4.1. Problem Analysis	21
4.1.1. First approach: to address the problem with Spark	22
4.1.2. Second approach: to address the problem with Distributed TensorFlow	25
4.1.3. Third Approach: to address the problem with Keras over Distributed TensorFlow	26
4.2. Selected approach	28
5. Implementation	30
5.1. Implementation of the selected option	30
5.1.1. Environment setup	30
5.1.2. Running a Distributed TensorFlow script	32
5.1.3. Dataset specifications	34
5.1.4. Network Model and Prediction Model	36
5.1.5. Code specification and explanation	38
6. Experiments and results	44
7. Conclusions and future work	49
7.1. Conclusions	49
7.2. Social and Environmental Impact and Ethical and Professional Responsibility	50
7.3. Future work	51
Bibliography	52

Index of Figures

2.1. Machine Learning Process	5
2.2. TensorFlow graph example.	10
2.3. Single machine and distributed system structure [Abadi et al., 2015]	11
2.4. Send/Receive nodes insertion [Abadi et al., 2015]	12
2.5. Spark Architecture	13
3.1. Example of model parallelism from [Dean et al., 2012]	15
3.2. Left: Downpour SGD. Right: Sandblaster L-BFGS. From [Dean et al., 2012]	16
4.1. Approach representation.	22
4.2. Standalone cluster with 3 workers.	23
4.3. Error traceback.	23
4.4. Application register.	24
4.5. Asynchronous data parallelism from [Abadi et al., 2015]	26
4.6. Keras deep neural network model.	27
5.1. Python 3.6.5 installation working.	31
5.2. TensorFlow 1.8.0 installation working.	31
5.3. Keras 2.1.6 installation working.	32
5.4. Parameter server running.	33
5.5. Worker node waiting for the rest of workers.	34
5.6. Deep network representation.	37
5.7. Dropout technique.	37

INDEX OF FIGURES

XI

6.1. First epochs of centralized training.	45
6.2. Last epochs of centralized training.	46
6.3. First 200 iterations in worker 1.	47
6.4. First iterations in worker 2.	47
6.5. Last iterations on one of the workers.	48

Chapter 1

Introduction

In recent years, technology that creates and collects data has become cheap and accessible, and in consequence it is spreading everywhere. Devices like computers, smartphones, cameras, RFID (radio-frequency identification), sensors, etc., are capable to collect a huge amount of different kind of data. This huge amount of data, known as “Big Data” is characterized by not having a defined structure, it is being generated fast and it is considered with a great value among others. Big Data is an inexhaustible source of knowledge for scientists, industries and governments.

The growth of the technology supporting the storage and the extraction of value from Big Data is already available. However, specific and more accurate tools and algorithms to extract value of Big Data in an efficient way remain still a challenge. In particular, an important factor is that the existing algorithms mostly have an iterative nature finding trouble to take advantage of the massive and distributed parallel computing platforms and frameworks.

Machine Learning techniques are, indeed, being applied in a variety of fields, and data scientists are being sought after in many different industries. With Machine learning, we identify the processes through which we gain knowledge that is not readily apparent from data, in order to be able to make decisions. Applications of Machine Learning techniques may vary greatly and are applicable in disciplines as diverse as medicine, finance, and advertising and also, in many scientific fields as Computer Vision, Natural Language Processing, etc.

The Gradient Descent Algorithm (GDA) is one of the most important optimization techniques used in many Machine Learning applications. Specifically, GDA is an algorithm used to perform multidimensional optimization. The objective is to reach the global minimum. It is used to improve or optimize the model prediction included in the Machine Learning applications. Optimization involves calculating the error value and changing the weights of the parameters to achieve that minimal error. The direction of finding the minimum is the negative of the gradient of the loss function. The GDA implementation is iterative and its performance results are very good. Currently, there are not many parallelized adaptations of this algorithm that are able to work over parallel and distributed computational platforms.

In this work, we study and analyze the GDA and its performance. We go over most used GD algorithms focusing on the advances made on distributed implementations. We also study and test the better frameworks for Deep Learning and distributed computation. With the global view obtained we design and test a use case with the objective of training a deep neural network over a given data set using distributed computation.

1.1. Specific Goals

In order to reach this main goal, we have defined a set of specific goals supporting it, namely:

1. Research and study gradient-based algorithms used to train deep networks.
2. Study the logic behind distributed computing.
3. Investigate previous work on the field of distributed optimizations for deep network training algorithms.
4. Parallelize an iterative algorithm over a distributed computing framework.
5. Evaluate the parallelized implementation running over simulated cluster of machines.
6. Compare results between iterative and parallelized implementations.

7. Become familiar with the Python programming language, TensorFlow library and Apache Spark cluster-computing framework.

1.2. Motivation and justification

It is common to hear about Machine Learning in any recent publication related with information technology field. This is in consequence of the wide field of application that this technology covers. Within its varied uses we can find some that play a very important role in modern society. In a fully connected world where new data is being generated every minute, including sensitive personal data, data security gains vital importance. Machine learning is used to predict if certain files contain malware software or even to detect anomalies when accessing sensitive information that can become security breaches. There are many other uses apart from cybersecurity, some of them are computer vision, speech recognition, anomaly detection in network traffic, it is also used in healthcare to detect patterns that can facilitate diagnosis of severe diseases like cancer, natural language processing or even smart vehicles.

One specific field inside Machine Learning is Deep Learning, where complex computing systems called Neural Networks (NN) are used to reach the goal of the problem. NN is a model offering excellent results in classification problems, but it is computationally expensive. Therefore, it is important to find more efficient versions of each component within this kind of models.

The problem that motivates this work comes with optimization. With modern hardware, software and networks the size of data is growing exponentially every minute. This has a good point because having more data to train our systems will produce more accurate predictions, but it also shows a bad point when talking about performance. When Machine Learning systems try to process that huge quantity of data, performance falls taking long execution times that could result in late predictions, that in other words are useless predictions.

A possible solution to this problem comes with modern distributed computing frameworks. This kind of platform convey great computing power that allows managing this enormous data quantity.

Summarizing, in this work we are studying the way algorithmic components of Deep Learning applications are optimized over distributed computing frameworks in order to work with larger datasets that will lead to better accuracy when finding the solution to a given problem.

We are going to focus on the algorithms behind Deep Learning, more specially on the algorithms used to train deep networks. This is because since these algorithms are usually iterative, they take very long when working with large scale data, so there is need for optimization and in addition Deep Learning covers some of the most promising uses of Machine Learning, as for example, real time computer vision. In particular, we address the Gradient Descent Algorithm.

1.3. Document structure

This Final Project is structured as follows. In the Background chapter, the theoretical framework supporting this work is introduced. State of Art chapter introduces the previous work on the topic. It will be studied over different publications in order to set a complete and current framework about the solutions proposed. Everything related with the design and implementation of the algorithms involved and every framework and technology that plays an important role in the solution of the problem will be covered in this chapter.

Then, the next chapters introduce the development of the use case proposed in this work. First, we will analyze the problem and choose the adaptation of the algorithm that fits better supported by its design and the reasons for its choice. Next, our implementation is presented together with the details about its deployment into a computational architecture used. A set of experiments are going to defined in order to prove the evaluation of the results obtained by the solution. These experiments also can be able to show the performance results and its suitability to be adapted into Deep Learning applications.

The document will end with the conclusions and future work chapter based on experiment results.

Chapter 2

Background

Machine learning systems automatically learn models from examples known as training data. Typically, these systems consist of three components, feature extraction, the objective function and learning.



Figure 2.1: Machine Learning Process

Feature extraction processes the raw training data to obtain the feature vector, where each feature captures an attribute of the training data. The objective function is the expression of Machine Learning algorithms goal, and it captures the properties of the learned model. The learning algorithm minimizes this objective function to obtain the model. This kind of algorithm iteratively refines the model by processing training data until an optimal solution is found, considering that the model has converged.

The two main variants of Machine Learning problems are those related with risk minimization and those referred as unsupervised learning. The first main group, risk minimization, works with labeled data, meaning each training example is associated with a label. Models generated with this kind of data, try to predict the value of the label for a future example, with the prediction depending on the parameters.

In any learning algorithm there is an important relation between the amount of data and the model size. Unbalanced situations may result in overfitted models that fail predictions or underfitted models that will fail to capture relevant attributes of the training data.

Regularized risk minimization is a method to find a model that balances model complexity and training error. The risk, that is the prediction error, is used to penalize model complexity in order to find a better balance that fits the problem in a way that it increases prediction accuracy.

In the second major class of Machine Learning algorithms, the label to be applied to the training examples is unknown, and for that they are called unsupervised algorithms. These procedures attempt to find the underlying structure in the data, with different approaches such as clustering or topic modeling.

The Gradient Descent Algorithm (GDA) is one of the most important optimization techniques used in many Machine Learning applications. Specifically, GDA is an algorithm used to perform multidimensional optimization. The objective is to reach the global minimum. It is used to improve or optimize the model prediction included in the Machine Learning applications. Optimization involves calculating the error value and changing the weights of the parameters to achieve that minimal error. The direction of finding the minimum is the negative of the gradient of the loss function. The GDA implementation is iterative and its performance results are very good. Currently, there are not many parallelized adaptations of this algorithm that are able to work over parallel and distributed computational platforms.

2.1. Gradient Descent

This goal of this algorithm [Ng,] is to find the global minimum of a function using a given set of examples. Each example comes labeled with a value. The function tries to predict values for each example. The objective is to find the parameters of the function that minimize the error, which is the difference between predicted and actual values.

So we have an hypothesis function $h_{\theta}(x^{(i)})$ that gives the predicted value for an example $x^{(i)}$, being $x^{(i)}$ the i th example. Using this function and the actual value for

the example $y^{(i)}$ we can write down the function to minimize as the summed squared error. The total number of examples is noted as m .

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

Gradient descent will iteratively update parameters θ given a learning rate α and computing the partial derivative term for the function, being j the n number of parameters.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Putting everything together as an algorithm implementation we have the following pseudocode.

Algorithm 1 Gradient Descent

repeat

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad (\text{for every } j=0, 1, 2, \dots, n)$$

until converged

It is worth mentioning the convenience to add some gradient checking implementation that tell if the algorithm is converging to the minimum or if there is a need to set a better learning rate α . Also vectorized implementations are recommended to reduce the workload of every iteration.

The iterative nature of the algorithm leads to very long execution times when working with large sets of training data m because it has to go over every single example to take one step forward. This limitation lead to some optimizations that reduce the workload, being Stochastic Gradient Descent the most used one.

2.1.1. Stochastic Gradient Descent

This algorithm shares the same goal as the previous one but with a main difference in the way of reaching that goal. Instead of going through the entire training set in order to take every step, we just need to look at a single training example to start making progress towards moving the parameters to the global minimum.

To clarify, what this algorithm is going to do, is to compute the first example and modify the parameters a little bit to fit just the first example a bit better. Then it will do the same for the second example and so on until going over the full training set and in case of need, starting all over again from the first one. Because of this procedure, it is recommended to shuffle the training examples.

We have to set a mathematical formulation for the cost of modifying the parameters for a single example.

$$Cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

And the partial derivative term for this cost.

$$\frac{\partial}{\partial \theta_j} Cost(\theta, (x^{(i)}, y^{(i)})) = (h_{\theta}(x^{(i)}) - y^{(i)}) x^{(i)}$$

We can now write the algorithm as pseudo-code, having in mind that two loops are needed. An outer loop that relates with the times that the algorithm has to go over the entire training set. It is observed that for many situations the algorithm can converge when computing every example just once. It usually takes between one to ten times. The second loop is used to compute every single example in the training set.

SGD algorithm may take a longer path to find the global minimum of the function, but each step will be considerably faster resulting on notable shorter times.

Algorithm 2 Stochastic Gradient Descent

```

repeat
  for  $i = 1$  to  $m$  do
     $\theta_j := \theta_j - \alpha (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}$  (for every  $j=0, 1, 2, \dots, n$ )
  end for
until 1 to 10 repetitions (usually 1 if  $m$  is large)

```

2.1.2. Mini-batch Gradient Descent

A third variation worth noting is Mini-batch Gradient Descent. It takes something from each previous algorithms. The main point is to work with reduced batches of training examples. We can say that GD works with batches that include the whole training set and SGD works with batches of only one training example. Working with slightly larger batches brings the best of both implementations, reducing the time needed to take each step and reducing the number of steps needed to converge.

Say that $b = 10$ is the batch size and $m = 1000$ we can write the following pseudo-code for this algorithm.

Algorithm 3 Stochastic Gradient Descent

```

repeat
  for  $i = 1, 11, 21, 31, \dots$  to 991 do
     $\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{i=1}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$  (for every  $j=0, 1, 2, \dots, n$ )
  end for
until 1 to 10 repetitions (usually 1 if  $m$  is large)

```

This third variation of the Gradient Descent algorithm is very promising for parallelization. Using vectorization, meaning you use a vector that contains the examples of each mini-batch, you can try to parallelize and compute them at the same time reducing the overall time for the algorithm.

2.2. Distributed and parallel computational frameworks

This section covers some of the most suitable tools to work with distributed deep learning and test different optimizations of gradient based algorithms. TensorFlow to implement and deploy Machine Learning models and Keras to build deep neural networks. We are also covering Apache Spark because it is one of the most used distributed computation frameworks in Big Data environments and some of the latest works in the field are related with attempts of deploying TensorFlow over Spark.

2.2.1. TensorFlow

TensorFlow [Abadi et al., 2015] is an open-source library published by Google for expressing and executing Machine Learning algorithms. The focus of the project is to allow and simplify the real-world use of Machine Learning by providing the tools for implementation and deployment of large scale models over different hardware platforms such as mobile systems, single machines or large scale clusters running specialized machines.

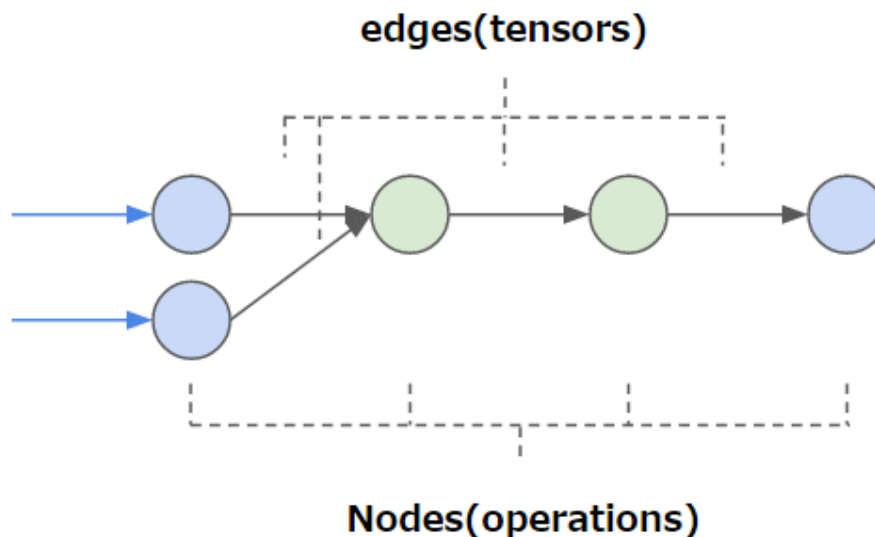


Figure 2.2: TensorFlow graph example.

To give some basis on how TensorFlow works, we are giving a brief description of the basic concepts. A TensorFlow computation is described by a graph with several nodes like in the figure 2.2. The graph represents data-flow computations. Each node instances an operation and can have various inputs and outputs. Tensors are multidimensional arrays that flow along normal edges of the graph. Special edges are used to control dependencies between nodes. Variables handle persistent mutable tensors that survive across executions of a graph. TensorFlow Session is used to run the whole graph or some parts allowing to repeat some computations.

Both local and distributed implementations are allowed. The main components of a system are the client that uses Session interface to communicate with the master and one or more worker processes responsible for executing graphs on one or more computational devices.

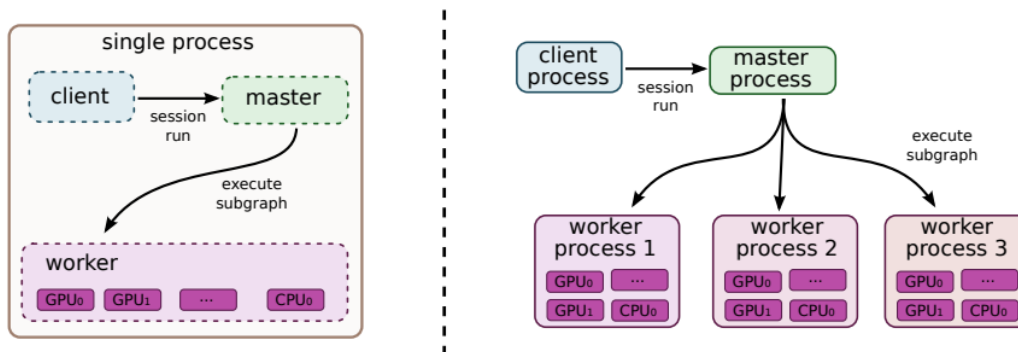


Figure 2.3: Single machine and distributed system structure [Abadi et al., 2015]

For multi-device and distributed execution, two steps are necessary. First, is node placement, deciding where to put each node of the graph, and then managing communication of data across devices or workers with send/receive node pairs that replace any cross-device edge of the resulting distributed graph.

TensorFlow includes built-in support for automatic gradient computation using many optimization algorithms like SGD. It also allows data parallel training. Assuming that a model is being trained using SGD with mini-batches, we can speed up training by parallelizing the computation for the gradient for a mini-batch. We can use several replicas of the model to each compute the gradients for each mini-batch and

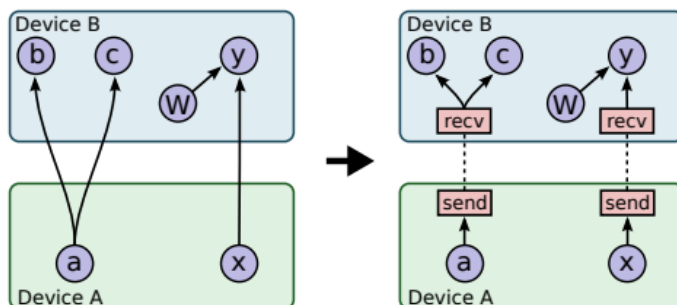


Figure 2.4: Send/Receive nodes insertion [Abadi et al., 2015]

then combine the gradients and apply updates. The update step can be made both synchronously and asynchronously.

2.2.2. Keras

Keras [Chollet et al., 2015] is a high-level open-source neural networks API developed with a focus on enabling fast experimentation. It is written in Python and is capable of running on top of TensorFlow simplifying the user experience when designing and working with neural networks.

A Keras model is understood as a sequence of modules that can be joined together almost without restrictions. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions and regularization schemes are standalone modules that can be combined to create new models.

2.2.3. Apache Spark

Spark [Zaharia et al., 2010] [Zaharia et al., 2012] is a distributed computation framework that supports applications that reuse a working set across multiple parallel operations while retaining scalability and fault tolerance of MapReduce.

Spark introduced Resilient Distributed Datasets (RDD) that are read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Once an RDD is generated with the working data, two kinds of operations can be made, transformations and actions. When an RDD is transformed, a new modified RDD based

on the original one is generated. Actions consist on operations over an RDD to obtain a result value that depends on the kind of action.

Since RDD can only be transformed and the resultant RDD depends on the previous one, if a partition is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition.

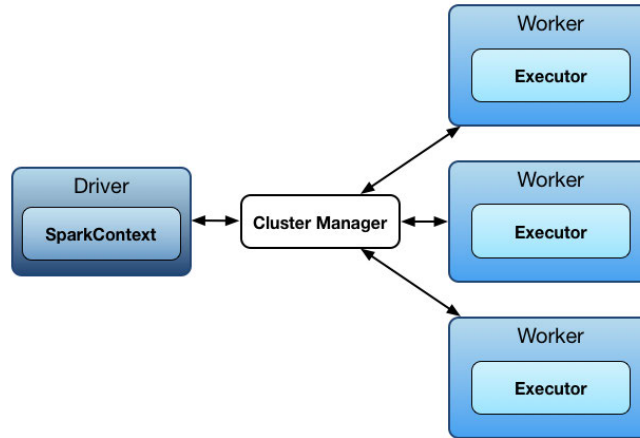


Figure 2.5: Spark Architecture

Spark architecture [Karau et al., 2015] automatically manages the distribution of the computation allowing users to only worry about the transformations that need to be done to datasets. Developers write a driver program using the SparkContext class. This driver is the master node and splits the application into tasks and schedules them to run on executors. It runs the cluster manager that communicates with the workers coordinating them for the execution of tasks. The workers are the compute nodes in Spark, they are Spark instances that run the executors. The executors are responsible for running the parallelized tasks.

Spark provides a standalone mode that allows developers to replicate this architecture on a single machine to test the basic features of Spark distributed computation having in mind that Spark relies on hardware scalability.

The use of RDD result in Spark is outperforming Hadoop MapReduce 10x in iterative Machine Learning jobs and can be used to interactively query a large dataset with sub-second response time.

Chapter 3

State of the art

To follow the better path, we research the previous work on the field and we summarize the most promising works that introduces key aspects to solve the problem.

Deep learning refers to Machine Learning algorithms that work with artificial neural deep networks. The general idea presented in the brief introduction to Machine Learning, can also be applied. Algorithms implement an objective function that has to be minimized in order to make predictions based on the learning data. There is a need to optimize the training algorithms used in this networks and a promising path comes with distributed computing.

The motivation for the search of distributed optimization solutions that scale up the training of deep networks is the observation that the scale of deep learning, according to the number of training examples, the number of model parameters, or both, can drastically improve ultimate classification accuracy.

The use of GPUs was a significant advance but has some limitations such as the small training speed-up when working with training sets that are larger than GPU memory or the need to reduce data and parameters in order to avoid bottlenecking in CPU-to-GPU transfers. These constrains make this option not optimal for large scale problems, with large number of examples and dimensions.

Another approach to the problem revolves around distributed computing using large-scale clusters of machines. One of the early references was presented in [Dean et al., 2012] as DistBelief, a software framework that enables model and data parallelism

within a machine via multithreading and across machines via message passing. With this framework two main findings were reported. The distributed optimization approach implemented can greatly accelerate the training of modestly sized models, and on the other hand it can train models that are larger than could be contemplated otherwise. Each statement was supported with a use case, for the first one, a cluster of machines was used to train a modestly sized speech model to the same classification accuracy in 1/10th of the time required with a GPU. On the other hand, the framework was used to train a large network of more than 1 billion parameters that allowed to drastically improve performance in computer vision.

This approach looks for a way of implementing distributed optimization that allows the use of a cluster of machines asynchronously without requiring that the problem be either convex or sparse. The focus is to scale deep learning techniques to train very large models, combining model parallelism with clever distributed optimization techniques that leverage data parallelism.

Model parallelism is supported in the way that the user may partition large models for neural networks across several machines, while the framework automatically manages communication, synchronization and data transfer between machines. Models with large number of parameters or high computational costs tend to benefit from access to more CPUs or memory to the point that communication costs dominate.

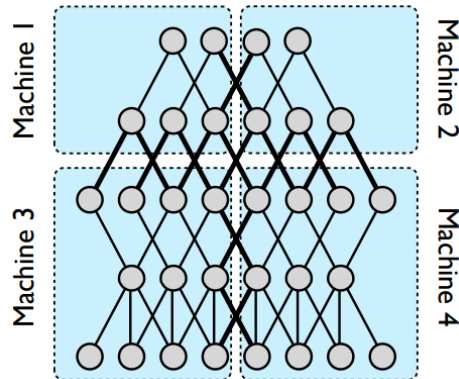


Figure 3.1: Example of model parallelism from [Dean et al., 2012]

Models with local connectivity structures as the one shown in the figure 3.1 benefit more from extensive distribution given their lower communication requirements. Only

those nodes with edges that cross partitions will need to have their state transmitted between machines and even if a node has multiples edges crossing, the state is only sent once. On each machine, the computation for individual nodes will be parallelized across all available CPU cores. Combining it with distributed optimization algorithms that use multiple replicas of the entire model is possible to achieve significant reductions in overall training times.

In order to train such large models in a reasonable amount of time, another level of parallelism that distributes training across multiple model instances is introduced. Two large scale distributed optimizations are compared. Both leverage the concept of a centralized parameter server sharded across many machines and take advantage of distributed computing. Each shard of the parameter server is responsible for storing and updating a reduced amount of the parameters. But most importantly, both methods tolerate variance in the processing speed of different replicas avoiding idle waiting times in the faster ones. The general idea is to simultaneously process distinct training examples and periodically combine results.

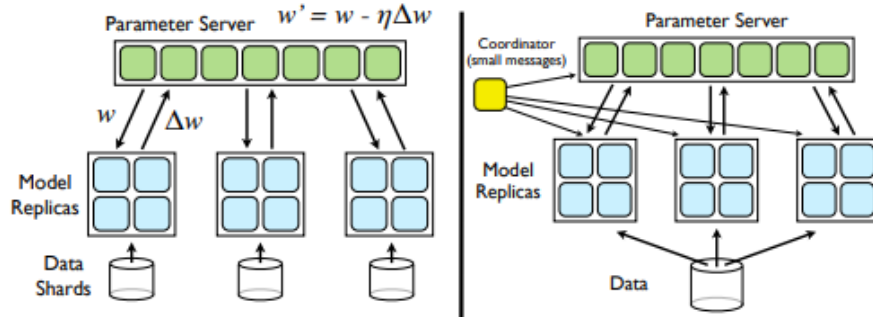


Figure 3.2: Left: Downpour SGD. Right: Sandblaster L-BFGS. From [Dean et al., 2012]

The first optimization procedure, named Downpour SGD, works with Stochastic Gradient Descent (SGD), probably the most common algorithm used to train deep neural networks. The traditional formulation for this algorithm is sequential making it impractical when working with large amount of data. The variation developed for this algorithm that allows it to be applied to large data sets, consists on dividing training data into subsets and running a copy of the model on each of them as it can be seen in 3.2. Each model communicates updates through a centralized parameter server which

keeps the current state of all parameters for the model. Each model replica asks the parameter server service for an updated copy of its parameters, then it processes a batch of data to compute the gradient and send it back to the parameter server which applies the gradient to the current value of the parameters.

The second procedure looks forward to apply batch methods to large models and large datasets. This procedure uses a coordinator process, represented in the figure 3.2, where the core of the L-BFGS optimization algorithm resides, and for that the procedure gets the name Sandblaster L-BFGS. This mentioned coordinator process issues commands from a small set of operations that can be performed on each parameter server shard independently, with the results being stored locally on the same shard. This allows running large models with billions of parameters without incurring the overhead of sending all the parameters and gradients to a single server. To avoid waiting for the slower machines, the load is balanced in the following way. The coordinator assigns each model a small portion of the work, much smaller than the batch, whenever they are free, letting faster model replicas do more work than slower ones, without the need to wait for them. In this procedure workers only fetch parameters at the beginning of the batch, when they have been updated by the coordinator, and only send gradients every few completed portions of the batch.

These optimization procedures were evaluated by applying them to the use cases mentioned before, speech recognition and visual object recognition. After studying the results, the conclusion reached was that Downpour SDG was dominant when working with a computational budget meanwhile Sandblaster L-BFGS and its more efficient use of network bandwidth enables it to scale to a larger number of concurrent cores for training a single model.

This work used the concept of the parameter server to manage communication of parameters. This concept has been developed further in later works. For example, in [Li et al., 2014] another parameter server framework for Machine Learning problems is proposed. In this elaboration, both data and workloads are distributed over worker nodes, while the server nodes maintain globally shared parameters, represented as dense or sparse vectors and matrices. When large models are shared globally by all worker nodes which must frequently access the shared parameters as they perform computation

to refine the model, three challenges are imposed. Accessing the parameter requires an enormous amount of network bandwidth. Many Machine Learning algorithms are sequential, resulting barriers that hurt performance when the cost of synchronization and machine latency is high. And third, at scale, fault tolerance is critical.

To overcome those challenges an open source implementation of a parameter server that focus on the systems aspects of distributed inference is presented. It provides five key features. Efficient communication optimized for Machine Learning tasks to reduce network traffic and overhead. Flexible consistency models that allow the designer to balance algorithmic convergence rate and system efficiency to reduce synchronization cost and latency. Elastic stability by adding new nodes without having to restart the running framework. Fault tolerance and durability by ensuring well defined behavior after network partition and failure. And finally, ease of use, representing shared parameters as vectors and matrices to facilitate development of Machine Learning applications.

Several algorithms were used to evaluate the work. The main findings confirm some essential aspects of this framework. The efficacy of reduced network traffic and the relaxed consistency model permitted the parameter server to outperform other solutions when running Sparse Logistic Regression. For Latent Dirichlet Allocation, a topic modeling algorithm, the parameter server showed a significant speedup in convergence that also scales well when increasing the number of machines.

The idea of working with several replicas hosted on different workers that share parameters in a parameter server was further developed until TensorFlow [Abadi et al., 2015] was introduced. This open-source framework is the current state of Google's development and simplifies distributed Machine Learning to the public. It supplies with prebuilt functions, algorithms and optimizers based on the ideas previously exposed that provide a higher level working framework that is easier to use for less experimented users.

Previously cited works rely on custom implementations. It is worth noting that before this framework was released, there were attempts of solving the problem using most common distributed computation frameworks.

Batch processing frameworks such as MapReduce or Spark have been gaining popularity because of the great simplification they bring to large scale data analytics tasks.

And even though they are not designed to support the workloads of existing deep learning systems, some implementations, as for example [Moritz et al., 2015], introduce a way for training deep networks using these frameworks, in this case, Spark.

The formulation comes with the name SparkNet and implements a scalable, distributed algorithm for training deep networks that lends itself to batch computation frameworks and works well in bandwidth limited environments.

Since much of the difficulty of applying Machine Learning has to do with obtaining, cleaning and processing data, training models with batch frameworks benefits from the existing data processing pipelines that have been engineered in today's distributed computational environments. Moreover, this approach allows data to be kept in memory from start to finish avoiding writing to disk between operations. In addition, hardware requirements are minimal, the framework gracefully handles bandwidth limited settings while also taking advantage of clusters with low latency communication. This is achieved by providing a simple algorithm for parallelizing SGD that involves minimal communication and permits straightforward implementation in batch frameworks. The goal is set in suggesting a system that can be easily implemented and performs nearly as well as custom frameworks, instead of attempting to outperform them. In SparkNet, training a deep network on the output of a SQL query, or a graph computation, or a streaming data source is direct due to its general purpose nature.

To perform well in bandwidth limited environments, a parallelization scheme for SGD that requires minimal communication is presented. Spark consists of a single master node and a number of worker nodes. The data is split among Spark workers. In every iteration the master node broadcasts the model parameters to each worker, that runs SGD on the model with its subset of data for a fixed number of iterations. Then the resulting parameters on each worker are sent to the master and averaged to form the new model parameters.

Although it works on a different platform, the idea behind the distribution of the computation is based on the same concepts. Several replicas of the model, train with different batches of data and submit gradients to a server where parameters are averaged and hosted. Currently the focus is diverting to deploying TensorFlow over a Spark cluster, but this path is on its first steps of development.

Since both works rely on the same ideas and TensorFlow is in a higher state of development due to the extensive use it gained since release, we think it provides better tools and a much robust framework to distribute the training of a deep neural network.

Chapter 4

Design and development

4.1. Problem Analysis

The problem we are addressing is the parallelization of the computation needed for training deep neural networks in order to reduce the time it takes to achieve the expected results. The main challenges that appear are related with the gradient based algorithms used to train this kind of networks. These algorithms have an iterative nature meaning that the previous step is needed to compute the next one. This conception collides with distributed computation and since redesigning them is way complicated, we are looking for a way to parallelize the computation.

Deep neural networks usually use very large datasets for training. Training consists on going over the full dataset to predict results, compare them with the expected ones and tuning the weights of the parameters of network depending on the quality of the prediction. This job is dependent of the size of the dataset.

So, we have a fixed algorithm, a model and a large dataset. We are going to try to replicate the training using slices of the dataset. The idea represented in the figure 4.1 is that if the network takes a certain time to train over a dataset, it will take a third of the time to train with a dataset a third of the size. This approximation is made keeping away of the equation every non dataset size dependent factors.

With this idea we need a distributed computation system that allows us to parallelize the training with model replication and data parallelism. If we can divide the training

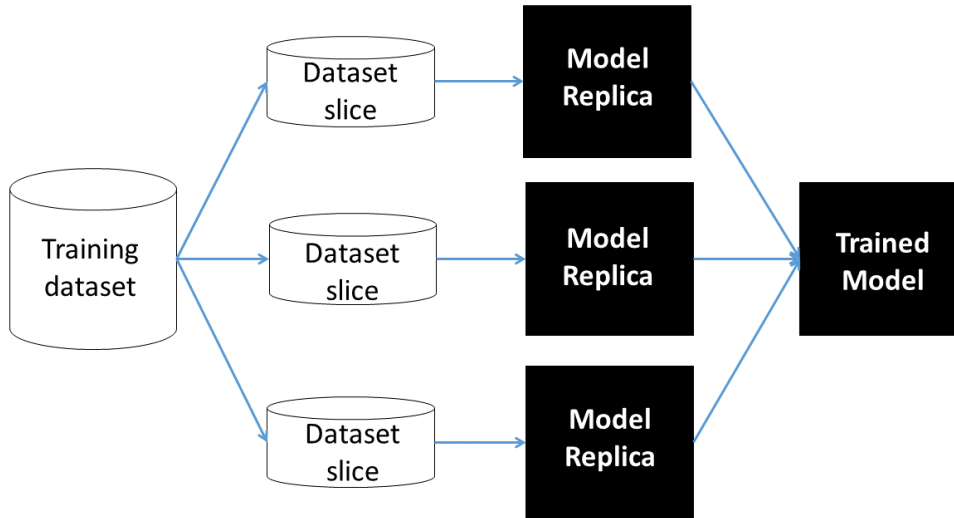


Figure 4.1: Approach representation.

between three model replicas that run at the same time and update the gradients together, we can reduce the time it takes to achieve the same precision.


After researching we decided to use Keras over TensorFlow to build our network but we need to test how this works on the existing distributed computation frameworks.

4.1.1. First approach: to address the problem with Spark

We decided to start with Spark because it greatly simplifies working with large datasets and it automatically manages distribution. Also, it is one of the most extended frameworks of distributed computation in Big Data and Machine Learning projects, consequently a solution built on this framework can be easier to add.

We used an ongoing project of adapting Keras library for Spark. This library is restricted to data parallelization in account of the difficulties to split up models with Spark.

We build a Spark standalone cluster with three workers to replicate a real cluster on a single machine. This permits us to test our applications before tuning and scaling to a high workload.


Spark Master at spark://192.168.1.47:7077

URL: spark://192.168.1.47:7077
REST URL: spark://192.168.1.47:6066 (*cluster mode*)
Alive Workers: 3
Cores in use: 3 Total, 0 Used
Memory in use: 9.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (3)

Worker Id	Address	State	Cores	Memory
worker-20180623233454-192.168.1.47-50520	192.168.1.47:50520	ALIVE	1 (0 Used)	3.0 GB (0.0 B Used)
worker-20180623233455-192.168.1.47-50542	192.168.1.47:50542	ALIVE	1 (0 Used)	3.0 GB (0.0 B Used)
worker-20180623233456-192.168.1.47-50564	192.168.1.47:50564	ALIVE	1 (0 Used)	3.0 GB (0.0 B Used)

Figure 4.2: Standalone cluster with 3 workers.

Datasets were prepared using Spark DataFrames and the model was created in the same way as any other Keras model. For the computation we used the customized Spark Estimator to call the methods from Keras library but after some testing we discard this approach due to the problems we found during execution.

```

File "C:\Python\lib\site-packages\elephas\spark_model.py", line 125, in start_server
    self.server.start()
File "C:\Python\lib\multiprocessing\process.py", line 105, in start
    self._popen = self._Popen(self)
File "C:\Python\lib\multiprocessing\context.py", line 223, in _Popen
    return _default_context.get_context().Process._Popen(process_obj)
File "C:\Python\lib\multiprocessing\context.py", line 322, in _Popen
    return Popen(process_obj)
File "C:\Python\lib\multiprocessing\popen_spawn_win32.py", line 65, in __init__
    reduction.dump(process_obj, to_child)
File "C:\Python\lib\multiprocessing\reduction.py", line 60, in dump
    ForkingPickler(file, protocol).dump(obj)
File "C:\Spark\spark-2.3.0-bin-hadoop2.7\python\lib\pyspark.zip\pyspark\context.py", line 303, in __getnewargs__
Exception: It appears that you are attempting to reference SparkContext from a broadcast variable, action, or transformation. SparkContext can only be used on the driver, not in code that it run on workers. For more information, see SPARK-5063.
  
```

Figure 4.3: Error traceback.

We run the code with a spark-submit command using as parameters the address of the master node and the path where the script is located. The specific bug that stopped us was a known bug that is pending to solve. Our execution encounters the following

exception: *Exception: It appears that you are attempting to reference SparkContext from a broadcast variable, action, or transformation. SparkContext can only be used on the driver, not in code that it run on workers. For more information, see SPARK-5063.*

In short, what this means is that the code is referencing the SparkContext from a worker and this cannot be done because Spark serializes objects to send them to the workers and due to SparkContext is not serializable it can only be accessed from the master node.



Application: TFG_IBF

ID: app-20180623234349-0000
Name: TFG_IBF
User: NachBas
Cores: Unlimited (3 granted)
Executor Limit: Unlimited (3 granted)
Executor Memory: 1024.0 MB
Submit Date: 2018/06/23 23:43:49
State: FINISHED

Executor Summary (3)

ExecutorID	Worker	Cores	Memory	State	Logs
------------	--------	-------	--------	-------	------

Removed Executors (3)

ExecutorID	Worker	Cores	Memory	State	Logs
2	worker-20180623233456-192.168.1.47-50564	1	1024	KILLED	stdout stderr
1	worker-20180623233454-192.168.1.47-50520	1	1024	KILLED	stdout stderr
0	worker-20180623233455-192.168.1.47-50542	1	1024	KILLED	stdout stderr

Figure 4.4: Application register.

After delving into this bug, we found that it was pending to solve and since this is not in the scope of our problem we considered trying other approaches.

Summarizing, Spark simplifies work with large datasets and hides distribution to the end user but the inconvenient of keeping the entire model on the master node without distribution may be incompatible with the solution to our problem. This approach needs further development to the point that different working solutions can be built enabling testing and benchmarking to conclude if this is a promising path to solve our problem.

4.1.2. Second approach: to address the problem with Distributed TensorFlow

Considering TensorFlow was born as a result of the development of the idea of optimizing deep network training and that almost every article we found relates with it on each or other way, naturally, the next step is moving to TensorFlow.

We want to deploy our solution using Distributed TensorFlow. This framework allows the user to distribute computation by assigning operations to different workers and variables to different parameter servers while automatically handling the process of communication between workers and parameter servers. It is extensible to any previously working cluster in virtue of it only needs the IP address of the nodes where we want to assign tasks. It also allows the user to use different ports from the local host to replicate a distributed execution. We leverage the local deployment to test our solution.

The focus is the same, to build a model, and replicate it on different workers in order to train with smaller pieces of the dataset at the same time.

The first step is to set the cluster specifications. The program needs to know the number of nodes from the beginning and it will not start until every node is operative. Then we must build the model. Although TensorFlow brings great tools for building Machine Learning models, these methods are still complex for the inexperienced user, hence this was the most problematic step of this approach.

At this point, we considered using Keras for the model even if we obtain a working solution only with TensorFlow since it adds higher level tools that are easier to use.

After building the model, we have to distribute computation between the nodes. We are using one parameter server and three workers. Because we only have one parameter server, variables are assigned straightaway. We divide the training dataset in three slices and assign each of them and the training operations for the entire model to each of the three workers achieving the desired model replication and data parallelism.

We have now three replicas of the model that go over a different training set. Each worker takes the values of the parameters to compute gradients, then it updates the parameters on the parameter server by averaging the values existing on the server and the worker results as seen in the figure 4.5. Then it repeats for the next iteration. The parameters are upgraded by averaging to allow asynchronous training between the three

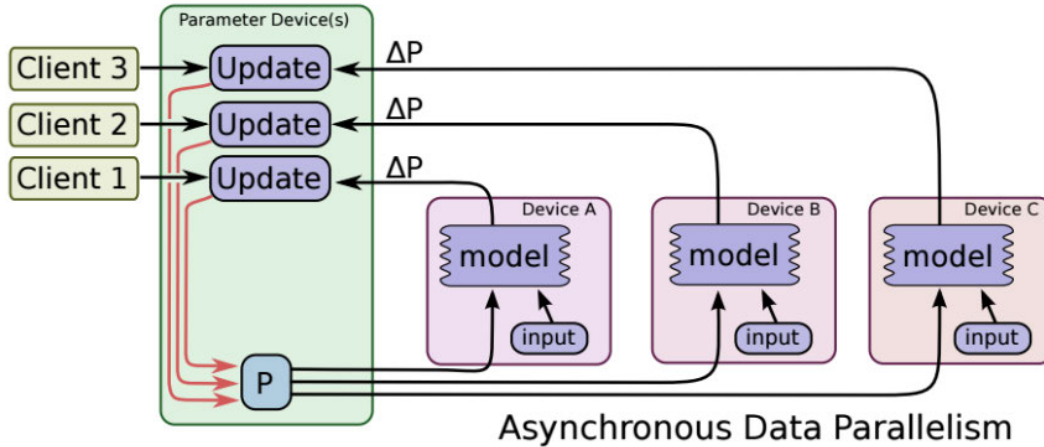


Figure 4.5: Asynchronous data parallelism from [Abadi et al., 2015]

workers.

To run Distributed TensorFlow programs the script must be run on every node that is included in the cluster specification done in the code. Each execution waits until every node is running and then, workers start computation asynchronously.

With this approach we managed to build a working solution. Distributed TensorFlow handles communication and most of the work behind distributing the computation and gives the user the chance of dividing operations at will. The downside comes in the complexity of the instructions needed to build the model, set the cost function and the optimizer algorithm. They are hard to use for non-expert users and increase mistake probability and difficulty for debugging when working with large models.

Although it is a working solution, before tuning and benchmarking the implementation, we are attempting to achieve the same solution but using Keras to build the model.

4.1.3. Third Approach: to address the problem with Keras over Distributed TensorFlow

Our third approach is based on the previous one. The difference is that this time we are using Keras to build our model.

Keras does not support Distributed TensorFlow so we cannot use model compile and fit methods. Model compile method is used to set the cost function, the optimization algorithm for the model and the metrics to monitoring the training. It is a straightforward method where the user only has to choose from a list of options the one that fits better with the model. Model fit is another easy to use method where the user only has to tell the training datasets and the number of epochs, then Keras starts and manages training while showing real time state of the metrics chosen.

The reason we are using Keras is the methods that simplify the process of building the model. This library includes methods that automatically create different kind of layers for neural networks models by only telling the number of outputs and other optional parameters as the number of inputs and activation. We are using the Sequential model that automatically builds the model by joining together the layers created.

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 512)	25600
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 512)	262656
dropout_2 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 43)	22059
Total params: 310,315		
Trainable params: 310,315		
Non-trainable params: 0		

Figure 4.6: Keras deep neural network model.

Keras also has method that summarizes the model and shows it in an easy to understand way. Our model can be seen in the figure 4.6. The information shown is the type and number of layers, the output shape of each layer and the number of parameters for each layer and the complete model.

Once the model is built, we distribute computation by assigning operations to the

workers as we did on the previous example. And we run the script following the same process.

In this approach we managed to get a working program that uses Keras layers to build the model and Distributed TensorFlow to distribute the training over different workers. It would be nice if in a future update of Keras they add support to Distributed TensorFlow allowing compile and fit methods to be used for distributed training letting the user just to choose between the kind of parallelism and the cluster specifications.

4.2. Selected approach

After researching and testing different approaches with different distributed computation frameworks we conclude that the most promising path to follow in our context is Keras on top of Distributed TensorFlow.

Spark and Keras are not fully compatible yet and is not even proved that this combination will be considerably faster. Spark nature collides with model parallelization therefore some important ways of distributing training seem to be hard to achieve with this framework. This approach is only recommended to use if the project is already based on Spark and there is a real need to add distributed training using the same framework.

On the other hand, Distributed TensorFlow was created to solve this problem and allows the user to create almost any Machine Learning model and distribute it on a cluster of machines. Distributed TensorFlow is also a project in development. Although it already provides with the tools for distributing computation, to work with complex models it requires a really deep knowledge about how TensorFlow graphs work and how to distribute operations on the workers to achieve the better solutions. Keras simplifies the process of building the model but there is still some work needed on developing Distributed TensorFlow to a more robust state that allows to easily work with most complex Machine Learning models in a parallelized way.

To summarize, we are using Keras to build our model in the most simple and direct way. And we are using Distributed TensorFlow to distribute the training using three model replicas that work with different slices of the training dataset. This is the most

adequate path to follow given the reduced experience with TensorFlow and the resources available to test our project.

Chapter 5

Implementation

5.1. Implementation of the selected option

This section will cover the implementation of our approach to the problem and every step needed to set up the environment and the tools to run Distributed TensorFlow scripts.

5.1.1. Environment setup

TensorFlow recommended programming language is Python. Although it supports other languages as Java or C, libraries for them are not as extensive as Python APIs. Consequently, the first step to set up our environment is to install Python. We chose to install latest versions of everything to make sure we are using the supported implementations to approach distributed computation.

Installing Python is straightforward with the installer. This installer also includes pip, a package management system that allows to install and manage software packages written in Python. Once it is finished installing, it is recommended to test the installation and double check the version installed.

On a Windows CMD window just by typing python, we get access to the information related with our installation and the Python shell to test if it is working correctly. In the Figure 5.1 we can see that we installed Python 3.6.5.

Once Python and pip are installed, downloading and deploying TensorFlow and

```
Microsoft Windows [Versión 10.0.17134.112]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\NachBas>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello World!")
hello World!
>>>
```

Figure 5.1: Python 3.6.5 installation working.

Keras libraries is simplified to just a single command. This automatically downloads the latest version and installs the package. Once again it is recommended to test the installation and check the version installed.

For TensorFlow we decided to use latest version for CPU-only. The GPU-version allows the user to assign operations to GPU resulting in faster computation. The system used for testing only holds one GPU and we are looking forward to using different workers at the same time. CPU allows us to simulate a worker for each of the CPU cores.

To install TensorFlow:

```
pip install --upgrade tensorflow
```

On the Python shell we are writing some basic TensorFlow instructions to check the installation was correct.

```
>>> import tensorflow as tf
>>> tf.__version__
'1.8.0'
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
2018-06-24 14:12:45.795357: I T:\src\github\tensorflow\tensorflow\core\platform\cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
>>> print(sess.run(hello))
b'Hello, TensorFlow!'
>>>
```

Figure 5.2: TensorFlow 1.8.0 installation working.

For Keras, we install the latest version and use the Python shell to confirm that the package is recognized.

To install Keras:

```
pip install --upgrade keras
```

Since Keras can be used over different Machine Learning frameworks, we have to set its configuration parameters to use TensorFlow backend.

Keras configuration parameters in Keras.json file:

```
{  
  "floatx": "float32",  
  "epsilon": 1e-07,  
  "backend": "tensorflow",  
  "image_data_format": "channels_last"  
}
```

```
>>> import keras as k  
Using TensorFlow backend.  
>>> k.__version__  
'2.1.6'  
>>>
```

Figure 5.3: Keras 2.1.6 installation working.

At this point, we have installed everything we need to test our solution. To write the scripts any code editor can be used.

5.1.2. Running a Distributed TensorFlow script

When writing a Distributed TensorFlow script, the cluster specification must be set from the beginning. It can be coded on the script or it can be passed as parameters in order to make the script more independent from the cluster.

For our example, we decide to set the cluster configuration on the code, so to run the script we have to use two parameters. The first one tells TensorFlow which kind of node is running the script and the second one is the task index.

So, we have the following command:

```
python script.py --job_name="node" --task_index=(int value)
```

With the cluster specification we set, the program already knows the address of each node and the number of workers and parameter servers, and it will not start running until every node in the cluster is active. We are using local implementation so, for our example the addresses are just local host ports. The configuration is one parameter server and three worker, so we assign task index 0 to the parameter server, and task index 0, 1 and 2 to the workers. Our script is called `keras_distributed.py`

To run the script in local mode, we need four command prompt windows, one for each node, and we will run each the following commands on each terminal:

To start the parameter server:

```
python keras_distributed.py --job_name="ps" --task_index=0
```

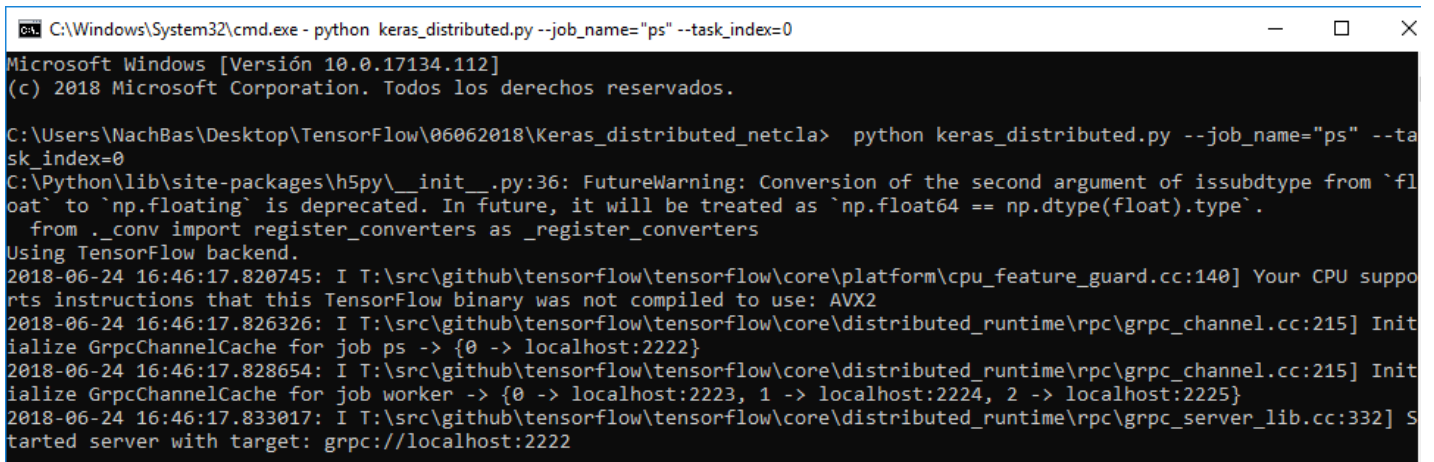
To start the three workers:

```
python keras_distributed.py --job_name="worker" --task_index=0
```

```
python keras_distributed.py --job_name="worker" --task_index=1
```

```
python keras_distributed.py --job_name="worker" --task_index=2
```

The expected behavior for the parameter server is to stand idle. This process runs on the background hosting the variables and does not show any screen information aside form successful start confirmation.



```

C:\Windows\System32\cmd.exe - python keras_distributed.py --job_name="ps" --task_index=0
Microsoft Windows [Versi3n 10.0.17134.112]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\NachBas\Desktop\TensorFlow\06062018\Keras_distributed_netcla> python keras_distributed.py --job_name="ps" --task_index=0
C:\Python\lib\site-packages\h5py\__init__.py:36: FutureWarning: Conversion of the second argument of 'issubdtype' from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
2018-06-24 16:46:17.820745: I T:\src\github\tensorflow\tensorflow\core\platform\cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
2018-06-24 16:46:17.826326: I T:\src\github\tensorflow\tensorflow\core\distributed_runtime\rpc\grpc_channel.cc:215] Initialize GrpcChannelCache for job ps -> {0 -> localhost:2222}
2018-06-24 16:46:17.828654: I T:\src\github\tensorflow\tensorflow\core\distributed_runtime\rpc\grpc_channel.cc:215] Initialize GrpcChannelCache for job worker -> {0 -> localhost:2223, 1 -> localhost:2224, 2 -> localhost:2225}
2018-06-24 16:46:17.833017: I T:\src\github\tensorflow\tensorflow\core\distributed_runtime\rpc\grpc_server_lib.cc:332] Started server with target: grpc://localhost:2222

```

Figure 5.4: Parameter server running.

In the Figure 5.4 we can see the cluster specification with a parameter server in

localhost:2222 and three workers, in localhost:2223/2224/2225. The parameter server was correctly started and is now waiting of the operations.

When running the workers nodes, we can see in the Figure 5.5 that once we ran the first one, it tells it is waiting for response from the other workers specified on the cluster.

```
Waiting for other servers
2018-06-24 16:52:16.832008: I T:\src\github\tensorflow\tensorflow\core\distributed_runtime\master.cc:221] CreateSession
still waiting for response from worker: /job:worker/replica:0/task:1
2018-06-24 16:52:16.835421: I T:\src\github\tensorflow\tensorflow\core\distributed_runtime\master.cc:221] CreateSession
still waiting for response from worker: /job:worker/replica:0/task:2
```

Figure 5.5: Worker node waiting for the rest of workers.

When the script is run on every worker, each of them start to compute operations asynchronously. This implementation is fault tolerant, we need every node to start running but if during the execution any worker is stopped, the remaining workers will continue with their jobs.

This kind of implementation is easy to scale up to a larger or real cluster. We only have to change the addresses and add desired new ones to the cluster specification. When working with real clusters, TensorFlow must be equally installed on ever node of the cluster, and the script has to be run on each server in the same way we did on our command prompt windows.

5.1.3. Dataset specifications

The training dataset that we used to test our implementation contains network information that identifies the app from which the activity was generated. It is a labeled dataset, this means we have features and the corresponding labels. The labels are the targets related to each register of the dataset. Having the labels, the predictions can be compared to the targets in order to compute the gradients and train the algorithm for better accuracy.

The dataset has 761.179 different examples for training and another 761.179 examples for validation. Since obtaining the best prediction accuracy is not the goal for this implementation we are going to reduce the number of training examples to 300K

and then we are dividing the dataset in 3 sections giving 100K to each worker. We do not have information about if the dataset is ordered, so to prevent this affecting our predictions we are selecting 300K random examples and not the first ones.

This dataset includes the following 49 features:

- cli_pl_header
- cli_pl_body
- cli_cont_len
- srv_pl_header
- srv_pl_body
- srv_cont_len
- aggregated_sessions
- bytes
- net_samples
- tcp_frag
- tcp_pkts
- tcp_retr
- tcp_ooo
- cli_tcp_pkts
- cli_tcp_ooo
- cli_tcp_retr
- cli_tcp_frag
- cli_tcp_empty
- cli_win_change
- cli_win_zero
- cli_tcp_full
- cli_tcp_tot_bytes
- cli_pl_tot
- cli_pl_change
- srv_tcp_pkts
- srv_tcp_ooo
- srv_tcp_retr
- srv_tcp_frag
- srv_tcp_empty
- srv_win_change
- srv_win_zero
- srv_tcp_full
- srv_tcp_tot_bytes
- srv_pl_tot
- srv_pl_change
- srv_tcp_win
- srv_tx_time
- cli_tcp_win
- client_latency
- application_latency
- cli_tx_time
- load_time
- server_latency
- proxy
- sp_healthscore
- sp_req_duration
- sp_is_lat
- sp_error
- throughput

The targets are 20 different types of applications identified with numbers from 0 to 42. Usually class 0 is used for unknown examples.

For classification we need to transform the targets to a binary array. This is an array of zeros and a single one to identify the value. Since the labels go from 0 to 42 we set the length of the array to 43.

$$[3] \implies [0, 0, 0, 1, 0, 0 \dots 0, 0]$$

We already have a working installation and a dataset that is ready to use, now we have to implement our model to start training.

5.1.4. Network Model and Prediction Model

The objective of this work is to distribute the training for a Machine Learning model to reduce execution times. Since our focus is not to improve predictions we are using a simple model that lets us test Distributed TensorFlow implementations.

From the dataset we know we are facing a classification problem where the focus is to identify to which of a set of categories a new observation belongs. To achieve this, we are building a deep neural network.

The network built has an input layer of 49 neurons, one for each feature, and an output layer of 43 neurons, one for each of the possible categories in our classification. We also include two hidden layers of 512 neurons with rectified linear unit activation function. This is a mathematical function used to choose and output from a number of inputs. It is one of the most common activation functions used in Deep Networks. The activation function for the output layer is *softmax function*, also known as normalized exponential function, that reduces a K-dimensional vector of real values to a K-dimensional vector of values in the range (0, 1) which all entries add up to 1. The values stored in the softmax output are the predictions in the form of the probabilities that each index is the correct value.

We used a fully connected model meaning every neuron is directly connected with all the neurons in the next layer. We also add two dropout layers to avoid overfitting.

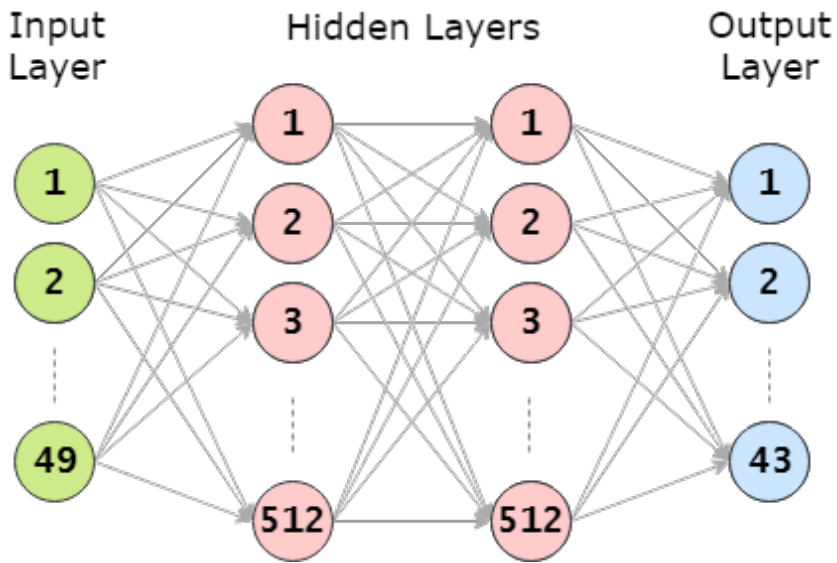


Figure 5.6: Deep network representation.

Overfitting in Machine Learning models refers to those with great accuracy on the training that fail on the validation set. Dropout technique deactivates a number of neurons from a layer for during an iteration of training forcing the model to learn the same concept with different neurons.

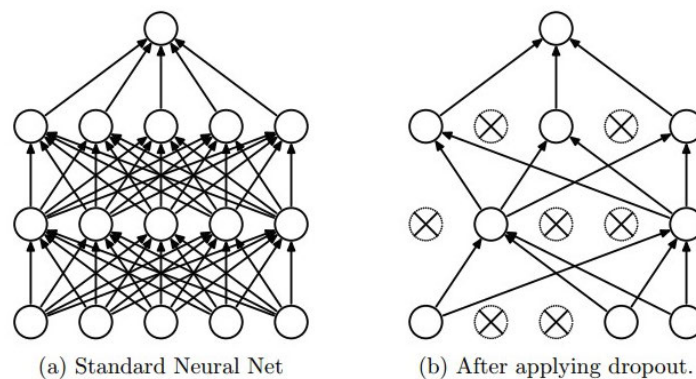


Figure 5.7: Dropout technique.

With the model already built, we have to set the loss function and the optimization algorithm. After researching classification models, we set the recommended loss function for this kind of problem which is categorical cross entropy function. The optimization

algorithm chosen was RMS prop. This is an optimized adaptation for Gradient Descent algorithm. It is mainly the same algorithm but with an adaptive learning rate that becomes useful when working with large numbers of parameters.

5.1.5. Code specification and explanation

Once the solution is designed the only step left is to write the script. This section covers our implementation explaining the structure of our code and focusing on the most important parts related with distributed implementations.

In the script we defined four functions and the main program. The functions are built to implement specific functionality making the code easier to maintain and debug. We will start by explaining these functions and then we will go over the main program to see how the operations are distributed.

Function: *load_data*

This function is used to read the csv files that contain the dataset. We created three files with 100K examples from the main dataset and each of the three slices is loaded on each of the workers. We have a separate file for the features and the labels. As we explained before in 5.1.3 we apply a transformation to the labels dataset to get the values as binary vectors. To achieve this we used the following Keras function:

```
from keras.utils import np_utils
labels = np_utils.to_categorical(labels, 43)
```

The datasets are loaded as global variables, accessible from every point of the program.

Function: *create_model*

This is the function used to generate the deep neural network. Keras sequential model allows to add the desired layers to the model with simplified methods as follows:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
```

```

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(49,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(43, activation='softmax'))

```

This code creates the model that is shown in the Figure 4.6 and explained in the previous Section 5.1.4. This function returns the model that can be used later to create operations that depend on its structure.

Function: *create_optimizer*

This function creates the operations needed to compute the gradients. It depends on a model and a targets tensor that are passed as parameters. A *tensor* is a TensorFlow unit with a defined type, size and dimensions. Operations are built using tensors and they are executed later when they receive data that fits the tensor's shape. For the following instructions predictions is the model output tensor and targets is the labels tensor.

First, we set the loss function and optimizer algorithm that were defined in the previous section. We set categorical cross-entropy using Keras package. Since we are implementing mini-batch gradient descent, each step of training computes the algorithm using a batch of examples. To obtain the loss function for the batch we add a reduce mean operation that obtains the average of a value over all the dimensions of a given tensor. We used TensorFlow implementation of RMSprop algorithm.

```

loss = tf.reduce_mean(
    keras.losses.categorical_crossentropy(
        targets,
        predictions))

optimizer = tf.train.RMSPropOptimizer(learning_rate)

```

The learning rate used for the algorithm was 0.001 as it is recommended for RMS-prop.

To make sure that the gradients are computed before the loss function is calculated we set control dependencies. These dependencies generate special edges on the TensorFlow graph that prevent an operation to be executed without meeting the correct conditions. In our code, train operation is dependent on the gradient updates that depend on a barrier operation that is only executed when the previous model update is already done.

```
with tf.control_dependencies(model.updates):
    barrier = tf.no_op(name="update_barrier")

with tf.control_dependencies([barrier]):
    grads = optimizer.compute_gradients(
        loss,
        model.trainable_weights)
    grad_updates = optimizer.apply_gradients(grads)

with tf.control_dependencies([grad_updates]):
    train_op = tf.identity(
        loss,
        name="train")
```

The barrier is a no op, this kind of operation just activates itself to unlock dependencies when the conditions are met. The optimizer implements the methods to compute and apply gradients. The compute gradients method runs the algorithm to minimize the previous value of the loss function that depends on the trainable weights of the parameters of the model. The apply gradients method update the parameters with the computed gradients. Finally, train op is an identity operation, that just returns a tensor with the same shape and contents as the input, so when train op is executed, loss function is calculated.

The last operation created in this function is the accuracy metric. This metric from the TensorFlow metrics package calculates how often predictions matches labels. Since

we have a Softmax layer as output of our model, we need to do some transformations. This is because Softmax layers return a vector of values of the probability for each index to be the predicted value. Comparing it with a binary vector result on wrong values for the metric since predictions never match labels. To obtain the right values of accuracy we used the argmax function on both targets and predictions. This function returns the index of the higher value in a vector, hence we will be comparing the label with the most probable value of our prediction.

```
targs = tf.argmax(targets, 1)
preds = tf.argmax(predictions, 1)
accuracy = tf.metrics.accuracy(targs, preds)
```

Function: *train*

This function gives the values to run the operations implemented in the previous one. It takes a batch from the training set and calls the operations using TensorFlow Session. This function depends on the step of the training epoch. The step is used to calculate the next batch used to train. This function also logs useful information related to the training to monitor execution.

Each worker takes the next batch from their training set. In our example we set 128 as batch size.

```
batch_x = features [batch_size*step:batch_size*(step+1)]
batch_y = labels [batch_size*step:batch_size*(step+1)]
```

The the operations are executed using batch x as model inputs and batch y as targets.

```
loss_value, accuracy_value = sess.run(
    [train_op, accuracy],
    feed_dict={
        model.inputs[0]: batch_x,
        targets: batch_y})
```

Main program:

At this point we have already implemented the operations needed for training. Now we have to assign the operations to each worker in order perform parallelized training.

The first step is to set the cluster specification and start the nodes. We used the `train` package from TensorFlow and set a cluster with one parameter server and three workers. We are using local host ports to simulate a parallelized execution on the same machine. As we explained earlier in Section 5.1.2, the same script is run on each node of the cluster and we use two parameters to tell the job name and the task index for each node.

```
cluster = tf.train.ClusterSpec({
    "ps": ["localhost:2222"],
    "worker": ["localhost:2223",
               "localhost:2224",
               "localhost:2225"]})

server = tf.train.Server(cluster,
    job_name=FLAGS.job_name,
    task_index=FLAGS.task_index)
```

The `train Server` method uses the cluster specification, the job name and the task index to start each node on the corresponding address.

All that remains is to assign operations to each worker. Distributed TensorFlow manages the communications between workers and parameter servers. The parameter server (`ps`) host the variables and stays listening to updates on the variables. The operations are executed on the workers, so we have to use the device setter methods to tell the program where to execute the following operations.

Since we are only using one parameter server, all variables will be hosted on it, but it is possible to work with multiple parameter servers and distribute variables across them. As we stated earlier, task index for workers goes from 0 to 2. In our implementation we replicate all operations on every worker. TensorFlow device method is used to assign operations to a given device. To identify the device, we used the replica device setter

method from the `train` package. With this method we can choose to which worker are the operations assigned and in which parameter server are the variables hosted. We also have to pass the cluster specification for the program to manage communications.

```

if FLAGS.job_name == "ps":
    server.join()
elif FLAGS.job_name == "worker":
    with tf.device(
        tf.train.replica_device_setter(
            worker_device="/job:worker/task:%d" % FLAGS.task_index,
            ps_device="/job:ps/task:0",
            cluster=cluster)):
        #operations

```

When assigning operations, a TensorFlow graph is built on each worker. In our implementation, the operations assigned are the operations created on the functions. Each worker calls the `create_model` function and once the model is created the `create_optimizer` function is called. Now that the graph is built, the variables are initialized. We used a TensorFlow supervisor to manage the session and save check points.

The last step is to create two loops, one to go over the entire training set calling the `train` function on every iteration and an outer loop to set the number of times the training will go over the training set.

Chapter 6

Experiments and results

The focus of our work was to deploy a distributed solution for deep network training. To test our implementation, we are going to set an experiment in which we compare the results of distributed training with the results of centralized training. Having in mind that since our distributed implementation is deployed on a single machine, results like time are not those that can be achieved with a real cluster. So, we set the focus on training the same network for the same number of epochs over the same dataset and see if the accuracy achieved is similar.

Summarizing what we explained in previous sections, we are training with a dataset that contains features for 300K examples and their correspondent labels. We are running mini-batch implementation of the optimized version of gradient descent using a batch size of 128 examples. We set the number of epochs to 100. The algorithm used is RMSProp optimizer with recommended learning rate of 0.001 and the loss function for our classification problem is categorical cross entropy.

To deploy the centralized solution for training we leverage Keras packages and methods. We used the same dataset and build the same model that we did earlier on the distributed implementation. To run the training, we used the Keras model compile and fit methods that could not be used for distributed training.

```

model.compile(
    optimizer='RMSprop',
    loss='categorical_crossentropy',
    metrics=['accuracy'])

model.fit(
    features_a,
    labels_a,
    batch_size=128,
    epochs=100,
    verbose=1)

```

Model compile is used to set the loss function, the optimizer algorithm and the metrics. Model fit is used to set the training dataset, the batch size and the number of epochs and then run the training. Vervose option is used to show the state of the training.

```

Layer (type)                 Output Shape              Param #
-----
dense_1 (Dense)              (None, 512)              25600
-----
dropout_1 (Dropout)         (None, 512)              0
-----
dense_2 (Dense)              (None, 512)              262656
-----
dropout_2 (Dropout)         (None, 512)              0
-----
dense_3 (Dense)              (None, 43)               22059
-----
Total params: 310,315
Trainable params: 310,315
Non-trainable params: 0
-----
Epoch 1/100
2018-06-30 09:12:29.586600: I T:\src\github\tensorflow\tensorflow\core\platform\cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
300000/300000 [=====] - 18s 59us/step - loss: 4.2368 - acc: 0.7371
Epoch 2/100
300000/300000 [=====] - 18s 59us/step - loss: 3.9173 - acc: 0.7570
Epoch 3/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570
Epoch 4/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570
Epoch 5/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570

```

Figure 6.1: First epochs of centralized training.

We run the centralized training of our network using the dataset to set a baseline of expected results. We can see that from the first few epochs the accuracy is already fixed at 0.7570. This is not a good value for a classification network, but our focus is to replicate training with distributed computation, not to obtain the best network for our dataset, so tuning the network to get better results is out of the scope of our work.

```

Epoch 87/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570
Epoch 88/100
300000/300000 [=====] - 18s 58us/step - loss: 3.9173 - acc: 0.7570
Epoch 89/100
300000/300000 [=====] - 17s 57us/step - loss: 3.9173 - acc: 0.7570
Epoch 90/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570
Epoch 91/100
300000/300000 [=====] - 17s 57us/step - loss: 3.9173 - acc: 0.7570
Epoch 92/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570
Epoch 93/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570
Epoch 94/100
300000/300000 [=====] - 17s 57us/step - loss: 3.9173 - acc: 0.7570
Epoch 95/100
300000/300000 [=====] - 17s 57us/step - loss: 3.9173 - acc: 0.7570
Epoch 96/100
300000/300000 [=====] - 17s 57us/step - loss: 3.9173 - acc: 0.7570
Epoch 97/100
300000/300000 [=====] - 17s 57us/step - loss: 3.9173 - acc: 0.7570
Epoch 98/100
300000/300000 [=====] - 17s 57us/step - loss: 3.9173 - acc: 0.7570
Epoch 99/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570
Epoch 100/100
300000/300000 [=====] - 17s 58us/step - loss: 3.9173 - acc: 0.7570

```

Figure 6.2: Last epochs of centralized training.

In the Figure 6.2 we can see that the last epochs of centralized training obtained the same 0.7570 accuracy. So now, we have a target accuracy to achieve with the distributed implementation.

Now we run our distributed implementation by running the script on each of the nodes as we explained in the Section 5.1.2. We can observe that the nodes work asynchronously by looking at the first iterations on each worker. The first worker starts with zero accuracy, but the following workers start the training with the current situation set by the already running workers. This is because we have three replicas of the model asynchronously updating the same parameters as represented in the Figure 4.5. The following images show the first iterations on each of the workers.

```

Epoch: 1, Iteration: 0, Cost: 13.2219, Accuracy: 0.0000 AvgTime: 12.50ms
Epoch: 1, Iteration: 10, Cost: 5.9932, Accuracy: 0.4844 AvgTime: 2.00ms
Epoch: 1, Iteration: 20, Cost: 4.9110, Accuracy: 0.6012 AvgTime: 2.00ms
Epoch: 1, Iteration: 30, Cost: 4.1554, Accuracy: 0.6484 AvgTime: 1.60ms
Epoch: 1, Iteration: 40, Cost: 3.1481, Accuracy: 0.6689 AvgTime: 1.80ms
Epoch: 1, Iteration: 50, Cost: 3.2740, Accuracy: 0.6845 AvgTime: 1.90ms
Epoch: 1, Iteration: 60, Cost: 3.6518, Accuracy: 0.6962 AvgTime: 2.20ms
Epoch: 1, Iteration: 70, Cost: 4.7851, Accuracy: 0.7059 AvgTime: 2.00ms
Epoch: 1, Iteration: 80, Cost: 4.4073, Accuracy: 0.7131 AvgTime: 1.90ms
Epoch: 1, Iteration: 90, Cost: 5.0369, Accuracy: 0.7185 AvgTime: 2.00ms
Epoch: 1, Iteration: 100, Cost: 3.7777, Accuracy: 0.7685 AvgTime: 3.00ms
Epoch: 1, Iteration: 110, Cost: 3.5258, Accuracy: 0.7588 AvgTime: 2.50ms
Epoch: 1, Iteration: 120, Cost: 2.6444, Accuracy: 0.7567 AvgTime: 2.60ms
Epoch: 1, Iteration: 130, Cost: 5.0369, Accuracy: 0.7574 AvgTime: 2.40ms
Epoch: 1, Iteration: 140, Cost: 3.2740, Accuracy: 0.7546 AvgTime: 2.60ms
Epoch: 1, Iteration: 150, Cost: 3.6518, Accuracy: 0.7554 AvgTime: 2.70ms
Epoch: 1, Iteration: 160, Cost: 4.5332, Accuracy: 0.7550 AvgTime: 2.60ms
Epoch: 1, Iteration: 170, Cost: 3.7777, Accuracy: 0.7548 AvgTime: 2.50ms
Epoch: 1, Iteration: 180, Cost: 4.5332, Accuracy: 0.7555 AvgTime: 2.60ms
Epoch: 1, Iteration: 190, Cost: 4.5332, Accuracy: 0.7535 AvgTime: 2.70ms
Epoch: 1, Iteration: 200, Cost: 4.1554, Accuracy: 0.7539 AvgTime: 2.20ms

```

Figure 6.3: First 200 iterations in worker 1.

```

Epoch: 1, Iteration: 0, Cost: 2.5185, Accuracy: 0.7489 AvgTime: 15.10ms
Epoch: 1, Iteration: 10, Cost: 3.7777, Accuracy: 0.7604 AvgTime: 2.30ms
Epoch: 1, Iteration: 20, Cost: 4.0295, Accuracy: 0.7562 AvgTime: 3.00ms
Epoch: 1, Iteration: 30, Cost: 4.5332, Accuracy: 0.7583 AvgTime: 2.40ms
Epoch: 1, Iteration: 40, Cost: 4.0295, Accuracy: 0.7548 AvgTime: 2.60ms
Epoch: 1, Iteration: 50, Cost: 3.3999, Accuracy: 0.7548 AvgTime: 2.90ms
Epoch: 1, Iteration: 60, Cost: 4.1554, Accuracy: 0.7549 AvgTime: 2.90ms
Epoch: 1, Iteration: 70, Cost: 3.6518, Accuracy: 0.7547 AvgTime: 2.90ms

```

Figure 6.4: First iterations in worker 2.

Once each of the workers finished running the 100 epochs over each dataset slice, we can see that the obtained results with our distributed implementation are the same we obtained with the centralized training. We achieved the target accuracy of 0.7570 with a similar behaviour.

In the Figure 6.5, we can see differences in the loss value obtained earlier, this is because in the centralized solution, what Keras shows is the average of the value of the loss function over the whole training set. In our distributed implementation we print the value of the loss function within each batch. If we average the values across all the iterations of one epoch we obtain very close results.

With these results we can conclude that our distributed implementation is correctly working. We achieved the same results as centralized training and with a similar behavior. The future work we propose is to test the same solution using better networks

```
Epoch: 100, Iteration: 650, Cost: 3.2740, Accuracy: 0.7570 AvgTime: 3.12ms
Epoch: 100, Iteration: 660, Cost: 2.8962, Accuracy: 0.7570 AvgTime: 3.12ms
Epoch: 100, Iteration: 670, Cost: 5.0369, Accuracy: 0.7570 AvgTime: 4.69ms
Epoch: 100, Iteration: 680, Cost: 3.7777, Accuracy: 0.7570 AvgTime: 3.12ms
Epoch: 100, Iteration: 690, Cost: 4.2814, Accuracy: 0.7570 AvgTime: 4.69ms
Epoch: 100, Iteration: 700, Cost: 4.6591, Accuracy: 0.7570 AvgTime: 1.56ms
Epoch: 100, Iteration: 710, Cost: 3.7777, Accuracy: 0.7570 AvgTime: 3.12ms
Epoch: 100, Iteration: 720, Cost: 3.5258, Accuracy: 0.7570 AvgTime: 3.12ms
Epoch: 100, Iteration: 730, Cost: 3.6518, Accuracy: 0.7570 AvgTime: 3.12ms
Epoch: 100, Iteration: 740, Cost: 4.0295, Accuracy: 0.7570 AvgTime: 3.12ms
Epoch: 100, Iteration: 750, Cost: 3.7777, Accuracy: 0.7570 AvgTime: 1.56ms
Epoch: 100, Iteration: 760, Cost: 3.5258, Accuracy: 0.7570 AvgTime: 1.56ms
Epoch: 100, Iteration: 770, Cost: 3.9036, Accuracy: 0.7570 AvgTime: 1.56ms
done
```

Figure 6.5: Last iterations on one of the workers.

with larger datasets and real cluster with different machines. With that configuration we could leverage the true benefit of parallelized computation that is a considerable reduction in training times.

Chapter 7

Conclusions and future work

7.1. Conclusions

During the development of this work we achieved our main goal that was to design and test a use case of parallelized computation to train a deep neural network. To achieve this, we have to study the basic concepts of Machine Learning and the most used versions of the gradient descent algorithm. Then, we researched different distributed computation frameworks to build our solution and deployed a working implementation. So, we can say that we successfully achieved our main goal by completing our specific goals, such as it is mentioned as following.

We studied Machine Learning, Deep Learning, and different optimizations of gradient descent. In addition, we studied distributed computation frameworks to decide the better one for our solution. Spark and Distributed TensorFlow also was studied in order to identify the best option to implement our solution.

Our solution have implemented a parallelized version of mini-batch gradient descent which run on a simulated cluster on a single machine which was built using Python programming language and and Keras in the top f TensorFlows. The results obtained with our solution show that it is able to obtain similar results to the centralized iterative implementations.

Other conclusions extracted from this work are all related with the early state of development of the frameworks that allow to parallelize Machine Learning optimization

algorithms.

The most important constraint we found is the design of the algorithm. Since it is iterative it can not be parallelized allowing the user only to distribute the workload. There are three main ways to achieve it, data parallelization, model parallelization or both combined. Being limited from the algorithm design reduces the possibilities of finding a successful solution.

The solutions found rely on Distributed TensorFlow. A high complexity application programming interface that require high experience to distribute workload over workers in an efficient way. Also, we found this solution is not compatible with most used distributed computation frameworks such as Spark. This slows the extension of use because it stops everyone that already has project using Spark or other machine learning platform and wants to add TensorFlow to distribute the workload of their models. In addition, the high learning curve requires long time to be familiar with the framework.

We reached a working solution that points at the correct path to follow when trying to implement a distributed solution that relies on these algorithms. As nowadays the size of available data is growing rapidly, this problem will grow when single machine implementations extend the times required for training to a point that can not be handled. Then this problem will be extended and the existing solutions will be improved, resolving the limitations we found.

7.2. Social and Environmental Impact and Ethical and Professional Responsibility

Actually, the environmental impact of this Project is non-existent because the computing application resulting does not involve the use of high consumption machines. It is a well-known that high energy consumption is associated with Data Centers. This Project does not make any use of them and therefore, we can state that the our application does not require high energy consumption.

With regard to the social impact, the development of this Project has involved a first exploration phase to address a set of challenges related with the migration of the sequential algorithm to the parallel one and migrates it to the a distributed compu-

ting platform. Since our project requires a couple of software process iterations to be considered a commercial software project, we consider that the social impact only is related with the preliminary results obtained that might allow a future development with interesting results to attack real problems in several domains of application.

Finally, with regard the Ethical and Professional Responsibility, we state that the development of this Project is not related with no one factor that contravenes with these two aspects. On the other hand, we state that this Project was developed with the highest respect for the pursuit of the profession and therefore have been considered both Ethics and Professional Responsibility in a personal way.

7.3. Future work

During the development of this work we get the feeling that the best way to approach the problem involves redesigning the algorithms used. This task needs to be approached by mathematicians with high knowledge on the Machine Learning field and may involve changes that impact more than just the algorithms.

Since this is not the focus of our work, we are going to expose what based on our experience are the most promising ways if they are developed in the correct direction.

First, we think that a higher level library that allows less experienced users access these frameworks is needed. This can be covered with Keras supporting distributed executions of TensorFlow.

Second, another important limitation that we think is important to break, is to allow Distributed TensorFlow to be deployed on most used distributed computation platforms. We tested an unfinished implementation using Keras and Spark, and we think that there is a need of development and improvement from both sides for this to become an actual option.

Last but maybe most important pending work is to thoroughly test and benchmark the optimizations made on the algorithms. These optimizations are approximations made to the algorithm behavior, and there is a need to prove that the results are actually the same as the ones obtained on single machine training.

Bibliography

- [Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
- [Chollet et al., 2015] Chollet, F. et al. (2015). Keras. <https://keras.io>.
- [Dean et al., 2012] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., aurelio Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., and Ng, A. Y. (2012). Large scale distributed deep networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc.
- [Karau et al., 2015] Karau, H., Konwinski, A., Wendell, P., and Zaharia, M. (2015). *Learning Spark: Lightning-Fast Big Data Analytics*. O’Reilly Media, Inc., 1st edition.
- [Li et al., 2014] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598.
- [Moritz et al., 2015] Moritz, P., Nishihara, R., Stoica, I., and Jordan, M. I. (2015). SparkNet: Training Deep Networks in Spark. *ArXiv e-prints*.

- [Ng,] Ng, A. Machine learning by coursera. <https://www.coursera.org/learn/machine-learning>.
- [Zaharia et al., 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association.
- [Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.