UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Image Recognition with Deep Learning Techniques and TensorFlow

Maurici Yagües Gomà

Master in Innovation and Research in Informatics

Data Mining and Business Intelligence

Advisor: Jordi Torres Viñals
Universitat Politècnica de Catalunya (UPC)
Department of Computer Architecture (DAC)

Co-Advisor: Ruben Tous Liesa
Universitat Politècnica de Catalunya (UPC)
Department of Computer Architecture (DAC)

October 20, 2016

Abstract

Deep neural networks have gained popularity in recent years, obtaining outstanding results in a wide range of application, but most notoriously in computer vision and natural language processing tasks. Despite the newly found interest, research in neural networks span many decades back, and some of today's most used network architectures where invented many years ago. Nevertheless, the progress made during this period cannot be understood without taking into account the technological advancements seen in key contiguous domains such as massive data storage and computing systems, more specifically in the Graphic Processing Unit (GPU) domain. These two components are responsible for the enormous performance gains in neural networks, that have made what we call Deep Learning a common word among the Artificial Intelligence and Machine Learning community.

These kind of networks need massive amounts of data to effectively train the millions of parameters they contain, and this training can take up to days or weeks depending on the computer architecture we are using. The size of new published datasets keeps growing, and the tendency of creating deeper networks that outperforms shallower architectures means that on the medium and long term the computer hardware to undertake these kind of training processes can only be found in high performance computing facilities, where they have enormous clusters of computers. However, using these machines is not straightforward, as both the framework and the code need to be appropriately tuned for effectively taking advantage of these distributed environments.

For this reason, we test TensorFlow, an open-sourced framework for Deep Learning from Google that has built-in distributed support, on top of the GPU cluster, called MinoTauro, at Barcelona Supercomputing Center (BSC). We aim to implement a defined workload using the distributed features the framework offers, to speed up the training process, acquire knowledge of the inner workings of the framework and understand the similarities and differences with respect to a classic single node training.

Acknowledgements

I would like to express my gratitude to my advisor, Jordi Torres, for introducing me to the Deep Learning universe, his endless enthusiasm and guiding me through the making of this thesis. Also, to my co-advisor, Ruben Tous, for the wise comments and encouragement. To the Barcelona Supercomputing Center for the opportunity and to their Support Team for the prompt answers whenever a problem occurred.

To my colleagues at the Autonomic Systems and e-Business Platforms group, for the interesting talks at lunch and their constant support whenever there is a problem. Special thanks to Francesc Sastre, without whom this thesis will not have been finished on time.

Also, to all people I have met during these years at the master, for introducing me to new areas of informatics I did not know.

Thanks to my family and friends, for supporting me and being always there, in the good, the not so good, and the funny times.

Contents

1. Problem statement

1.1 Goals of this thesis

The aim of this master thesis is to test the scalability of the TensorFlow framework in the MinoTauro machine (GPU cluster) at Barcelona Supercomputing Center (BSC). This framework is oriented, but not constrained, to develop Deep Learning applications, and has built-in distributed computing for using multiple nodes of both CPUs and GPUs. A series of machine configurations will be tested with a defined workload, in order to understand the behaviour of both the framework and the machines.

Distributed training in deep learning is quite a recent tendency given the massive sizes of datasets coupled with the always increasing depth of the neural networks. As of today, mainly the big software companies use these systems internally for their own products, but out of the box versions were difficult to implement or were non-existent. TensorFlow is in ongoing development, and the amount of documentation for their distributed version is scarce, given that it is not the most used functionality, as not many users have access to clusters of GPUs.

1.2 Introduction to distributed machine learning systems

During the last decade machine learning has seen an increased interest from many different areas, becoming a key element in multiple industries. Although many of today's algorithms rely on decades, or even centuries, old mathematics, their increased performance is a consequence of both more complex models and bigger volumes of data. The need for storing these large amounts of data (both structured and unstructured) has encouraged the research in new database formats, circumventing the constraints of old systems. The rise of distributed storage systems led to the need of machine learning algorithms that could benefit from that type of architecture, and that could cope with more complex models, even though that meant designing new algorithms or adapting old ones.

Part of this section takes the main ideas of the great introduction in Bekkerman, Bilenko, and Langford [5], presenting some general concepts of applying machine learning in distributed environments. Furthermore, an overview of deep learning and the current state of distributed machine learning is presented.

1.2.1 Scaling up machine learning

Distributed computing has emerged as an efficient solution for tackling growing machine learning challenges. These platforms allow the scalability of machine learning tasks, enabling some properties that could not be achieved with single-machine processing:

- **Larger number of data instances**: combining the distributed storage and bandwidth of a cluster of machines the amount of data that can be processed is increased.
- **Higher input dimensionality**: some tasks involving natural language, video or biological data are prone to a high number of features. Parallelizing the computation across features can be used for scaling up the process.
- **Model and algorithm complexity**: running complex non-linear models that are computationally expensive can be alleviated with distributed computing by means of parallel multicore or multinode, and the use of coprocessors such as GPUs.
- **Model selection and parameter tuning**: for these kind of tasks parallelization is straightforward as they consist of independent executions of the same dataset, with different combinations of hyper-parameters.

However, scaling up the computation to a distributed system is also a more difficult task than working on a single machine. Depending on the efficiency of the algorithm, communication between nodes can be a bottleneck, and the system has to be robust in order to overcome possible failures.

1.2.2 Types of parallelism

The main idea behind this computing paradigm is to run tasks concurrently instead of serially, as it would happen in a single machine. To achieve this, there are two principal implementations, and it will depend on the needs of the application to know which one will perform better, or even if a mix of both approaches can increase the performance.

Data parallelism

In this mode, the training data is divided into multiple subsets, and each one of them is run on the same replicated model in a different node (worker nodes). These will need to synchronize the model parameters (or its gradients) at the end of the batch computation to ensure they are training a consistent model. This is straightforward for machine learning applications that use input data as a batch, and the dataset can be partitioned both rowwise (instances) and columnwise (features).

Some interesting properties of this setting is that it will scale with the amount of data available and it speeds up the rate at which the entire dataset contributes to the optimization [51]. Also, it requires less communication between nodes, as it benefits from high amount of computations per weight [38]. On the other hand, the model has to entirely fit on each node [15], and it is mainly used for speeding computation of convolutional neural networks with large datasets.

Model parallelism

In this case, the model will be segmented into different parts that can run concurrently, and each one will run on the same data in different nodes. The scalability of this method depends on the degree of task parallelization of the algorithm, and it is more complex to implement than the previous one. It may decrease the communication needs, as workers need only to synchronize the shared parameters (usually once for each forward or backward-propagation step) and works well for GPUs in a single server that share a high speed bus [14]. It can be used with larger models as hardware constraints per node are no more a limitation, but is highly vulnerable to worker failures.

1.2.3 Performance metrics

The term performance in these systems has a double interpretation. On one hand it refers to the predictive accuracy of the model, and on the other to the computational speed of the process. The first metric is independent of the platform and is the performance metric to compare multiple models, whereas the second depends on the platform on which the model is deployed and is mainly measured by metrics such as:

- **Speedup**: ratio of solution time for the sequential algorithms versus its parallel counterpart
- **Efficiency**: ratio of speedup to the number of processors
- **Scalability**: efficiency as a function of an increasing number of processors

Some of this metrics will be highly dependent on the cluster configuration, the type of network used and the efficiency of the framework using the libraries and managing resources.

1.3 Deep learning overview

Despite having gained a lot of notoriety in the last years, research in multilayer neural networks span many decades [57]. Conventional machine learning techniques had difficulty in processing natural data in their raw form, so to make classifiers more powerful it was common to use generic non-linear features such as kernel methods and create multi-stage hand-tuned pipelines of extracted features and discriminative classifiers [55]. However, those generic features did not allow the learner to generalize well from the training examples, so one of the main advantages of these new methods is the fact that good features can be learned automatically using a general purpose learning procedure and without human engineering [41]. That is why deep learning techniques are also referenced as representation learning methods that are fed with raw data and have multiple levels of representation, obtained by composing simple but nonlinear modules that each transform the representation at one level into a representation at a higher, slightly more abstract level [41, 7]. These methods have dramatically improved the state of the art in recent years in multiple areas such as speech recognition, visual object recognition and detection, drug discovery and genomics.

1.3.1 Early tendencies and evolution

Early works of multilayer neural networks date back to Ivakhnenko [34] for Feedforward Multilayer Perceptron type networks and to Fukushima [20] for today's version of convolutional neural networks, inspired by the structure of the nervous visual system, having lower order hypercomplex cells in the early layers and higher order hypercomplex cells in the latest ones. Although this works mainly established the structure for deep neural networks that was later popularised, they still lacked a good learning algorithm for weight updates during the training phase. It was not until the mid 80s that backpropagation was popularised for neural networks by Rumelhart, Hinton, and Williams [54]. With backpropagation the error values at the output are back propagated using the chain rule to compute the gradient of the error with respect to each weight [26]. This way the weights get updated at each step with the objective to minimize a loss function. LeCun et al. [42] were the first to apply backpropagation to the network architecture presented by Fukushima [20] to recognize handwritten zip code digits (MNIST dataset). Backpropagation is still today the dominant approach for training deep networks.

However, backpropagation for feedforward neural networks and recurrent neural networks seemed to work only for shallow structures and suffered from the vanishing or exploding gradient problem. That means cumulative backpropagated error signals either shrink rapidly, or grow out of bounds, making deep structures impossible to train [57]. This led to the Long Short-Term Memory (LSTM) cell [31] that outperformed the performance of recurrent neural networks on tasks that require learning the rules of regular languages, and tasks involving context free languages that could not be represented by Hidden Markov Models. In addition, this new method did not suffer from the vanishing or exploding gradients. In recent years these kind of structures have improved the state of the art in applications such as protein analysis, handwriting recognition, voice activity detection, optical character recognition, language identification, audio onset detection, text-to-speech synthesis, social signal classification and machine translation [57].

All in all, ventures based on neural networks and other AI technologies began to make unrealistically ambitious claims while seeking investments. When AI research did not fulfill these unreasonable expectations, investors were disappointed, which led to a decline of popularity of neural networks until the mid 2000s [24]. That gap was filled with other methods such as Support Vector Machines, kernel methods and graphical models that dominated many practical and commercial pattern recognition applications during that time [57].

The expression Deep Learning was coined around 2006 with the publication of structures such as a stack of Restricted Boltzmann Machines called Deep Belief Network that arouse interest for the good results obtained [30]. At the same time, early tests on GPU-based convolutional neural networks made training times several times faster, and by 2010 a GPU implementation of backpropagation was up to 50 times faster than its CPU version [57].

1.3.2 Recent years

Certainly advancements in computer hardware are of main importance in developing deeper architectures that can build more complex models. Neural networks have doubled in size roughly every 2.4 years with the availability of faster CPUs at first and the advent of general purpose GPUs later, in addition to better software infrastructure for distributed computing [24]. Also, the digitization of society has made possible the availability of gigantic datasets that can be used to successfully train neural networks. ImageNet is an image database with 1.2 million images with 1000 different categories organized according to the WordNet hierarchy and has, at least, 732 images per class, making it one of the biggest datasets available. Different tracks are available, image classification, single and multiple-object localization, or object detection in videos, with classifications for models using the provided data and another for those that use additional resources. Since 2012, deep convolutional networks have won the ImageNet competition for object classification, and by 2014 all top contestants were relying on convolutional neural networks [55].

Table 1.1 and Table 1.2 show the progress convolutional neural networks have achieved in image classification and single-object localization, challenges started at 2010 and 2011, respectively. For those years, the winners used methods related to support vector machines, but since then increasingly deeper neural networks have been topping the results, with notable gains year after year.

Table 1.1: Summary of winning teams in the ImageNet image classification challenge. (Source: Russakovsky et al. [55] with some additions)

Year	Top-5 Error	# of layers	Institution	Name	Ref
2010	28.2	-	NEC Labs America, University of Illinois at Urbana-Champaign, Rutger	NEC	[45]
2011	25.8	-	Xerox Research Center Europe	XRCE	[56]
2012	16.4	8	University of Toronto	SuperVision	[38]
2013	11.7	8	Clarifai	Clarifai	[72]
2014	6.7	22	Google	GoogLeNet	[65]
2015	3.5	152	Microsoft Research Asia	MSRA	[28]

Table 1.2: Summary of winning teams in the ImageNet single-object localization. (Source: Russakovsky et al. [55] with some additions)

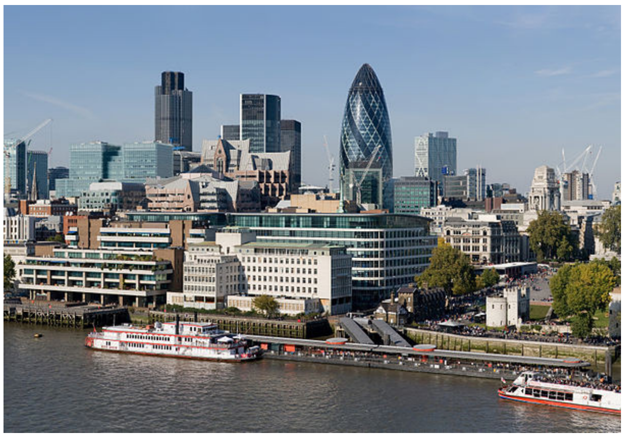
Year	Error	Institution	Name	Ref
2011	42.5	University of Amsterdam, University of Trento	UvA	[70]
2012	34.2	University of Toronto	SuperVision	[38]
2013	30.0	New York University	OverFeat	[58]
2014	25.3	University of Oxford	VGG	[61]
2015	9.0	Microsoft Research Asia	MSRA	[28]

1.3.3 Current uses and tendencies

The domain of application and research of deep learning techniques has progressed a lot in a small span of time. Image classification was the first application where deep neural networks began showing incredible results. The ImageNet challenge pushed the research of many groups in this area, with incredible results, as seen in the previous section, in the domains of image classification, single and multiple-object localization, and object detection in videos as of recently.

However, convolutional neural networks have been used in some varied applications such as in Gatys, Ecker, and Bethge [22], where these kind of networks are able to modify the content of a photo to match the style of a picture. Using the pre-trained network in Simonyan and Zisserman [61] the new image is approximated to both, the activations at some layer of the original content, and the Gram matrix of the activations of the style image. The user can tune the weight of the content and the style in the final picture adjusting some hyperparameters in the loss function, and choose the layers to take into account in the optimization process. This approach was later applied to video by Ruder, Dosovitskiy, and Brox [53] and Anderson et al. [4], and some alternative implementations of the original neural art appeared during the last year [11, 21, 35, 48, 69].

Other neural network architectures have also benefited from deeper models, more computational power and bigger datasets. Recurrent neural networks and LSTM have been used for some years related to natural language processing fields, improving the state of the art in


(a) City of London skyline (Photo by DAVID ILIFF)[1]

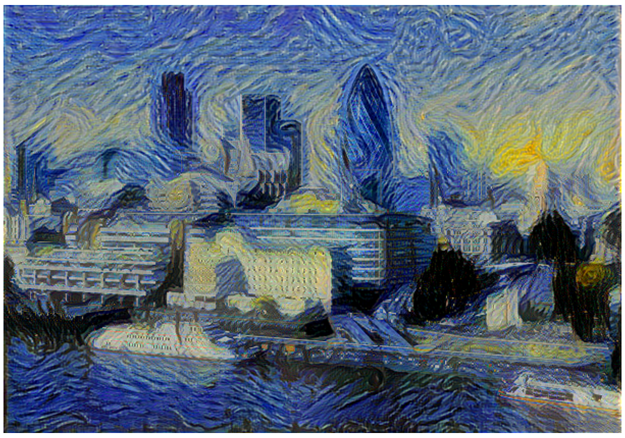

(b) Style from Vincent van Gogh's *The Starry Night*

Figure 1.1: Example of neural art with own implementation.

applications such as language translation [13, 64], speech-to-text recognition [25] and sentiment analysis [63]. Vinyals et al. [71] coupled the powerful architecture of convolutional neural networks for image recognition to a recurrent neural networks, to create a generative model that can output a description of an image given as input. Recently, a group of people from the Google Brain has been developing algorithms for music generation with recurrent neural networks [52].

Another successful application that had a lot of attention in the media was the win of the Google DeepMind project AlphaGo[2] to Go professional player Lee Sedol. Go had been till then a much more difficult feat for artificial intelligence, due to its enormous search space and the difficulty of evaluating board position and moves. Prior to this match, the program was paired to another Go player, Fan Hui, and the details of the match can be found at Silver et al. [60]. The paper also explains the methods used for building AlphaGo, which uses: convolutional networks for creating a representation of the position, supervised learning for training a policy network from human expert moves, reinforcement learning for optimizing the policy network in games of self-play, value networks for evaluating the board positions and predicting the winner of the game, and paired with the Monte Carlo tree search that estimates the value of each state in the tree of all possible sequence of moves.

Given the costs of computation of the networks, they used asynchronous multi-threaded searches, executing simulations on CPUs, and computing policy and value networks in parallel on GPUs. The final version of AlphaGo used 40 search threads, 48 CPUs, and 8 GPUs, but a distributed version was tested with 40 search threads, 1,202 CPUs and 176 GPUs. Their version was tested against other programs available, winning most of the matches even when free moves where allowed for the opponents. During the match against Lee Sedol a custom ASIC, called TPU was used [16, 36], which is more energy efficient than GPUs, and is specially tailored for machine learning tasks, as it is optimized for low precision computations that have proved to work well in inference.

[1]License: CC-BY-SA 3.0 (Wikipedia)
[2]AlphaGo webpage

1.3.4 Framework comparison

Given the amount of interest in these algorithms, many frameworks specialized in this area have appeared in the last years[3]. However, most of them where designed for efficient single-node computation with multiple devices (CPUs and GPUs), and their distributed applications were dependent on other frameworks of distributed processing (Spark, Hadoop, etc.).

The appearance of new programming models, such as MapReduce [17], that built on old approaches like Message Passing Interface, has shifted the computation of many companies to distributed configurations. This new models make it easy to parallelize tasks on many servers, taking advantage of the distributed storage system for efficient computation. However, state-of-the-art deep learning systems rely on custom implementations to facilitate their asynchronous, communication-intensive workloads [47]. As a consequence, some deep learning frameworks with only single-node computing capabilities have been used on top of those parallelizing systems to obtain a satisfactory degree of performance.

With the growing expansion of deep learning, some frameworks with different properties and options have been appearing during the last years. Table 1.3 resumes some properties of a reduced selection of frameworks available for deep learning.

Table 1.3: Framework comparison. (Source: Tokui et al. [68] with some additions)

	Caffe	CNTK	MXNet	TensorFlow	Theano	Torch
Started from*	2013	2014[4]	2015	2015	2008	2012
Main developers	BVLC[5]	Microsoft	DMLC[6]	Google	Université de Montréal	Facebook, Twitter, etc.[7]
Core languages	C++	C++	C++	C++, Python	C, Python	C, Lua
Supported languages	C++, Python MATLAB	CLI	C++, Python R, Julia, Go...	C++, Python	C, Python	C, Lua
Parallel computation	Multi GPU	Multi GPU Multi-node	Multi GPU Multi-node	Multi GPU Multi-node[8]	Multi GPU	Multi GPU

*Year of GitHub repository creation

Most of the broadly used frameworks enable multi-device computation and support both CPUs and GPUs. Also, most of them enable by now parallel computing in single node environments (multi GPUs), and some of the newest begin supporting multi-node architectures, although this is a much more scarce feature, given the complexity of the implementation and that not many users have access to clusters of GPUs.

Performance on single-node execution is similar across frameworks, as they rely on similar implementations for running on GPUs [1]. Differences may arise in the way each framework handles memory allocation or other specific design choices of the developers. As Adolf et al. [3] summarize, there is a convergent evolution amongst the more popular libraries. For example,

[3]At least 40 according to the following list (Deep Learning Software links)
[4]Released on January 2016 (announcment)
[5]Berkeley Vision and Learning Center (BVLC)
[6]Distributed (Deep) Machine Learning Community (dmlc.ml)
[7]Ronan Collobert, Clement Farabet, Koray Kavukcuoglu, Soumith Chintala (list of contributors)
[8]Since version 0.8.0 RC0 in April 2016 (release notes)

they all use a simple front-end specification language, optimized for productivity, and highly-tuned native backend libraries for the actual computation.

As said before, prior to the appearance of native distributed versions, some experiments were done combining single-node frameworks with parallel engines. For instance, Caffe is a framework that can be used on multiple GPUs in a single-node machine, and has been applied to a distributed environment with SparkNet [47] and CaffeOnSpark [19], developed by the Yahoo Big ML Team. Prior to open sourcing the native distributed implementation of TensorFlow, its single-node version was scaled with Spark by Databricks [32], for hyperparameter tuning, and Arimo [62], for scalability tests.

Other trends include the addition of extensions and libraries to those frameworks that enable multi-node capabilities without the need of a middle layer as Spark. Theano-MPI is a framework, built on Theano, that supports both synchronous and asynchronous training of models in data parallelism setting on multiple nodes of GPUs [46]. In the case of Caffe, FireCaffe offers similar properties for scaling the training of complex models [33], and for Torch, the Twitter Cortex team developed a distributed version [66].

1.4 Framework used: TensorFlow

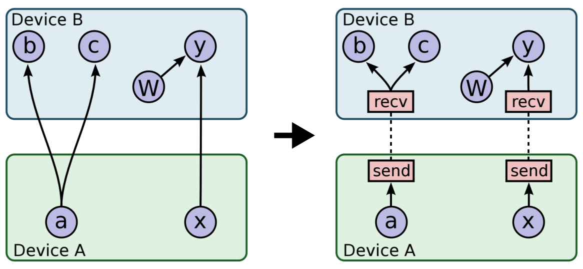
TensorFlow [9] was open-sourced on November 2015, as a software library for numerical computation using data flow graphs [2]. Despite its extended library for deep learning functions, the system is general enough to be used for many other domains. This system is the evolution of the previous scalable framework DistBelief, supports large-scale training and inference in multiple platforms and architectures, and aims to be a flexible framework for both experimenting and running in production environments. This is achieved by using dataflow programming models (Spark, MapReduce) which provide a number of useful abstractions that insulate from low-level details of distributed processing. Then, communication between subcomputations is explicit and makes it easy to partition and run independent sub-graphs on multiple distributed devices.

The node placement algorithm uses a greedy heuristic for examining the completion time of every node on each possible device, including communication costs across devices. The graph is then partitioned into a set of subgraphs, one per device, and *Receive* and *Send* nodes are added to the incoming and outgoing connections, as Figure 1.2a shows. These will coordinate the transfer data across devices, making the scheduling of the nodes decentralized into the workers.

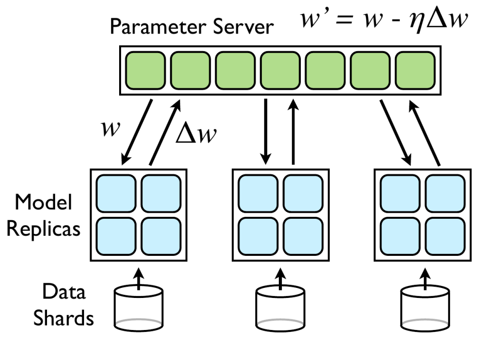
The native distributed version of TensorFlow builds up from DistBelief, which could utilize computing clusters with thousands of machines [18]. The parameter server architecture presented in Li et al. [43] manages asynchronous data communication between nodes, and supports flexible consistency models, elastic scalability, and continuous fault tolerance. This architecture uses a set of servers to manage shared state variables that are updated by a set of data-parallel workers. In essence, variables of the model are stored in the parameter servers, and are loaded to each worker for the computation, with the batch of data. The workers return the gradients obtained to the parameter servers, where the variables are updated and sent again for the next computation (Figure 1.2b).

Initial implementations of the parameter server were (key, value) pairs, where for LDA the pair is a combination of the word ID and the topic ID. The implementation in [43] treats these

[9]Homepage: https://www.tensorflow.org



(a) Placement of Send/Receive nodes. (Source: Abadi et al. [2])



(b) Parameter server architecture. (Source: Dean et al. [18])

Figure 1.2: Architecture details of TensorFlow.

objects as sparse linear algebra objects, by assuming that keys are ordered, and those missing are treated as zeros, which enables optimized operations for vectors.

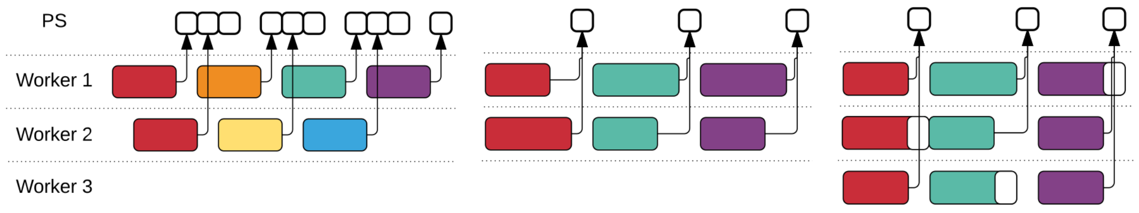
TensorFlow implements a user-level checkpointing for fault-tolerance, so that a trained model can be restored in the event of a failure. Those checkpoints need not to be consistent, unless strictly specified, and is a choice of the user to define the checkpoint retention scheme. This enables easy reutilization of models for fine-tuning or unsupervised pre-training [1].

Another interesting feature is the visualization tool named TensorBoard, that enables computation graph and summary data visualizations, for better understanding of the model defined and the training process. Some internal features made it possible to build a tracing profile of the model, so that the user can track the device placement of every node, node dependencies and computation time.

For this project, the version used is the latest stable one available 0.10.0, and all the scripts and models have been built with the Python API. Further details of the implementation will be specified when necesseary in the following sections.

1.4.1 Types of data parallelism

TensorFlow enables multiple strategies for aggregating the results in multinode settings when working with data parallelism. Chen et al. [12] compare different methods, with varying results in model accuracy and training speeds for up to 200 different nodes. The workers will have a similar task in both cases, but it is the way the model is updated at the parameter servers that makes the key difference between these two approaches.



(a) Asynchronous replication (b) Synchronous replication (c) Synchronous with backup workers

Figure 1.3: Strategies for data parallelism. (Source: Abadi et al. [1])

Asynchronous mode

In this mode, each worker replica will be computed independently of the computations in other nodes, so no problems will arise if one of them crashes. They will follow these steps for every batch iteration:

1. The worker fetches from the parameter servers the most up-to-date model needed to process the current batch
2. Computes the gradients of the loss with respect to these parameters
3. Sends back the gradients to the parameter servers

The parameters servers will then update the model accordingly, and sent it to the worker for the next computation. The idea is the model on the parameter servers will be updated each time a worker sends the gradients (Figure 1.3a). This makes it easy to implement large scale training processes in architectures that may have different types of hardware, as each one will behave independently, and they maintain high throughput in the presence of stragglers.

The problem for this strategy is that individual steps use stale information, making each step less effective [1], so when a model is trained with N workers, each update will be $N-1$ steps old on average. It may happen that by increasing the number of workers, the training results worsen with the noise introduced by the discrepancy between the model used to compute gradients and the model that will be updated in the parameter server [12].

Synchronous mode

With this setting, the workers will behave as before, as they only need to make gradient computation, but the parameter servers will wait for all the workers to send the gradients before proceeding to the model update, using queues for accumulating and applying them atomically. It means that at the start of the batch iteration, each worker will have the most up-to-date model, and the gradients will be computed from the same parameter values (Figure 1.3b).

Given that, for this case, the model in each worker is the same, the batch contains different items and the gradient computation is done all at once, it can be seen as a single step with a batch size N times bigger. According to results in Chen et al. [12] synchronous training can achieve better accuracy, needs fewer epochs to converge and has less accuracy loss when increasing the number of workers. However, the update time of the model depends on the slowest worker, as the parameter servers need to wait until the end of the execution in every one of them, so time per batch may be slower than in the asynchronous case. In order to mitigate this effect, a possible workaround is to implement *backup* workers, so that the parameter servers update the model when they have collected the gradients from m of n workers (in Figure 1.3c $m = 2$ and $n = 3$).

Despite the several publications from Google demonstrating the different results between synchronous and asynchronous modes [1, 12], the public version of TensorFlow has not a working implementation for the synchronous training [10] [11] in multinode.

[10] GitHub issue #3458
[11] Synchronous example (link)

1.5 Platform

BSC-CNS (Barcelona Supercomputing Center – Centro Nacional de Supercomputación) is the National Supercomputing Facility in Spain and was officially constituted in April 2005. BSC-CNS manages MareNostrum, one of the most powerful supercomputers in Europe. The mission of BSC-CNS is to investigate, develop and manage information technology in order to facilitate scientific progress. With this aim, special dedication has been taken to areas such as Computer Sciences, Life Sciences, Earth Sciences and Computational Applications in Science and Engineering.

All these activities are complementary to each other and very tightly related. In this way, a multidisciplinary loop is set up: "our exposure to industrial and non-computer science academic practices improves our understanding of the needs and helps us focusing our basic research towards improving those practices. The result is very positive both for our research work as well as for improving the way we service our society" [9].

BSC-CNS also has other High Performance Computing (HPC) facilities, as the NVIDIA GPU Cluster MinoTauro that is the one used for this project.

1.5.1 MinoTauro overview

MinoTauro is a heterogeneous cluster with 2 configurations [10]:

- 61 Bull B505 blades, each blade with the following technical characteristics:
  - 2 Intel E5649 (6-Core) processor at 2.53 GHz
  - 2 M2090 NVIDIA GPU Cards
  - 24 GB of main memory
  - Peak Performance: 88.60 TFlops
  - 250 GB SSD (Solid State Disk) as local storage
  - 2 Infiniband QDR (40 Gbit each) to a non-blocking network
  - 14 links of 10 GbitEth to connect to BSC GPFS Storage
- 39 bullx R421-E4 servers, each server with:
  - 2 Intel Xeon E5-2630 v3 (Haswell) 8-core processors, (each core at 2.4 GHz,and with 20MB L3 cache)
  - 2 K80 NVIDIA GPU Cards
  - 128 GB of Main memory, distributed in 8 DIMMs of 16 GB - DDR4 @ 2133 MHz - ECC SDRAM
  - Peak Performance: 250.94 TFlops
  - 120 GB SSD (Solid State Disk) as local storage
  - 1 PCIe 3.0 x8 8GT/s, Mellanox ConnectXR - 3FDR 56 Gbit
  - 4 Gigabit Ethernet ports

The operating system is RedHat Linux 6.7 for both configurations. The TensorFlow installation is working only on the servers with the NVIDIA K80 because the M2090 NVIDIA GPU cards do not have the minimum required software specifications TensorFlow demands.

Each one of the NVIDIA K80s contain two K40 chips in a single PCB with 12 GB GDDR5, so we physically have only two GPUs in each server, but are seen as 4 different cards by the software.

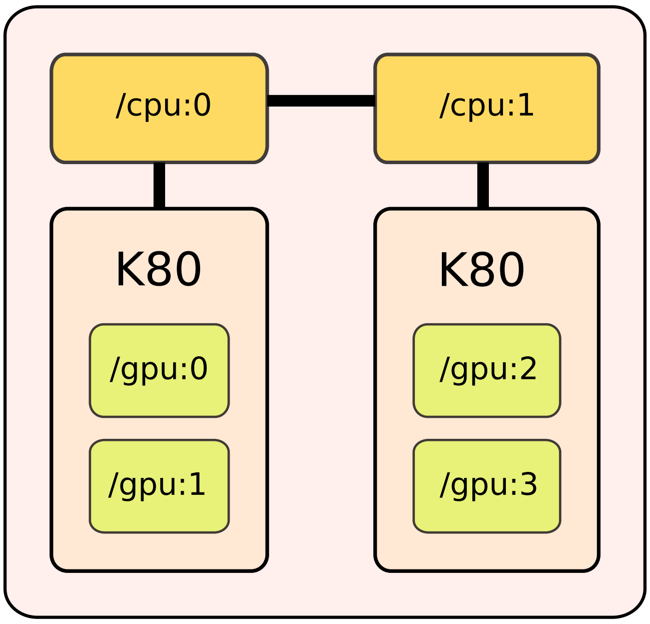


Figure 1.4: Schema of the hardware in a MinoTauro node

1.5.2 Software stack and queue system

MinoTauro uses the GPFS file system, which is shared with MareNostrum. Thanks to this, all nodes of MinoTauro have access to the same files. All the software is allocated in the GPFS system and is also common in all nodes, there are two software folders, the general folder and the K80 folder, the last folder contains all the software compiled for the NVIDIA K80 GPUs [10].

MinoTauro comes with a software stack, this software is static and the users can't modify it. If a user needs a new software, a modification or an update of an existent software has to request a petition to the BSC-CNS support center. This is the used part of the software stack in this project:

- **TensorFlow 0.8[12]:** The first version of TensorFlow used in this project. This version implements the distributed features and it was used on the first half of the project.
- **TensorFlow 0.10[13]:** The second release of TensorFlow used in this project. This version has a lot of bug fixes, improvements and new abstraction layer, TF-Slim, that allows to do some operations more easily.
- **Python 3.5.2:** The Python version installed in the K80 nodes. It comes with all required packages by TensorFlow.

[12] GitHub release 0.8
[13] GitHub release 0.10

- **CUDA 7.5:** CUDA is a parallel computing platform and programming model created by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the GPU [49]. This is required to use GPUs with TensorFlow.
- **cuDNN 5.1.3:** The NVIDIA CUDA Deep Neural Network library is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers [50].

MinoTauro has a queue system to manage all the jobs sent by all users in the platform. To send a job to the system is necessary to do a job file with the job directives:

`job_name`	The job's name
`initialdir`	Initial directory to run the scripts
`output`	The output file's name
`error`	The error file's name
`gpus_per_node`	Number of needed GPUs for each assigned node
`cpus_per_task`	Number of needed CPUs for each task
`total_tasks`	Number of tasks to do
`wall_clock` limit	Maximum job time
`features`	To request other features like K80 nodes

In order to obtain the maximum number of GPU cards, it is mandatory to set the parameter `cpus_per_task` to 16, all cores in a node, and set `gpus_per_node` to 4. With this configuration the queue system will give a full node with 4 GPUs, two K80 cards, for each task. The fact that we ask for the maximum number of CPUs is for the system to assign different nodes, so we can effectively send the distributed jobs. Despite the fact that we are mainly interested in the computational capabilities of GPUs, powerful CPUs are needed in order to feed the GPUs as fast as possible, so as to not have any bottlenecks on the device.

This is an example of a job file:

```
#!/bin/bash
# @ job_name= job_tf
# @ initialdir= .
# @ output= tf_%j.out
# @ error= tf_%j.err
# @ total_tasks= 3
# @ gpus_per_node= 4
# @ cpus_per_task= 16
# @ wall_clock_limit = 00:15:00
# @ features = k80
module load K80 cuda/7.5 mkl/2017.0.098 CUDNN/5.1.3 python/3.5.2
python script_tf.py
```

This file defines a job with 15 minutes of maximum duration on 3 nodes with 4 GPUs on each. Once the job is submitted the queue system will give the job ID number to do tracking on the task:

```
Submitted batch job 87869
```

At the time that the queue system assigns the nodes for the task, the user can see which nodes have been assigned and connect to them to see their GPU an CPU usage among other things. In the following job summary, we have been given all CPUs and GPUs in nodes 16, 19 and 25, during a maximum time of 15 minutes.

```
JOBID NAME   USER  STATE    TIME TIME_LIMI CPUS  NODES NODELIST(REASON)
87869 job_tf -     RUNNING  0:02     15:00   48      3 nvb[16,19,25]
```

All the parameters and configurations for the MinoTauro queue system are available on the MinoTauro User's Guide [10].

1.5.3 Distributed TensorFlow with Greasy

To run distributed TensorFlow is needed to run the workers in different nodes and know the IP address of the node which is running a TensorFlow process. With a simple MinoTauro task is not possible to this, as the user cannot control what to execute in each node, so we will use a tool called Greasy [67].

Greasy is a tool designed to make easier the deployment of embarrassingly parallel simulations in any environment. It is able to run in parallel a list of different tasks, schedule them and run them using the available resources.

We developed a script for obtaining the list of nodes the job had assigned and building the corresponding Greasy task. The file will contain as many lines as nodes available, and each one will execute in a different one. The following example uses the nodes assigned to define a parameter server in node 16 (`nvb16`) and workers at node 19 and 25 (`nvb19, nvb25`). We use the flag `task_index` to number the different workers, and the same process will be done with parameter servers, in case we have more than one.

```
    # In node 16
    python script_tf.py --ps_hosts=nvb16-ib0:2220
        --worker_hosts=nvb19-ib0:2220,nvb25-ib0:2220
        --job_name=ps --task_index=0

    # In node 19
    python script_tf.py --ps_hosts=nvb16-ib0:2220
        --worker_hosts=nvb19-ib0:2220,nvb25-ib0:2220
        --job_name=worker --task_index=0

    # In node 25
    python script_tf.py --ps_hosts=nvb16-ib0:2220
        --worker_hosts=nvb19-ib0:2220,nvb25-ib0:2220
        --job_name=worker --task_index=1
```

To configure the working modes of the GPUs in the servers we will use environment variables. For example, we will use the variable `CUDA_VISIBLE_DEVICES` to enforce externally which GPUs the program will use. When we launch a TensorFlow job, all 4 GPUs will be available by default, with indexes ranging from 0 to 3. However, with `CUDA_VISIBLE_DEVICES` we can enforce externally which GPU we want the job to see. For instance, with:

```
CUDA_VISIBLE_DEVICES="0" python script_tf.py
```

our program will only see a single GPU available, specifically the one with index 0. If we want the program to be able to use two GPUs we can then use:

```
CUDA_VISIBLE_DEVICES="0,3" python script_tf.py
```

and then the only GPUs the program will have available will be the ones with index 0 and 3. We can even define a job that cannot use any GPU by giving an empty parameter, so it will have to rely only on CPU:

```
CUDA_VISIBLE_DEVICES="" python script_tf.py
```

This little trick will be very useful for defining the models we intend to test and that are described in the following chapter. These options enable us to have a more flexible infrastructure to play with.

2. Testing methodology

The following chapter aims to describe the methodology used in the tests, and discuss the different decisions that were taken to conduct them. We review the dataset and network used, describing the peculiarities of the architecture with the different training implementations that we aim to test. We also include some comments on the tools we used to assess each implementation was working as intended.

2.1 Dataset

The CIFAR-10 dataset consists of 60,000 32x32 colour images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images [39]. The amount of images of each class is equally distributed among the training and testing datasets, with 5000 and 1000 images per class respectively. Classes are mutually exclusive, meaning that each image cannot have more than one class. The different classes with some examples are shown in Figure 2.1.

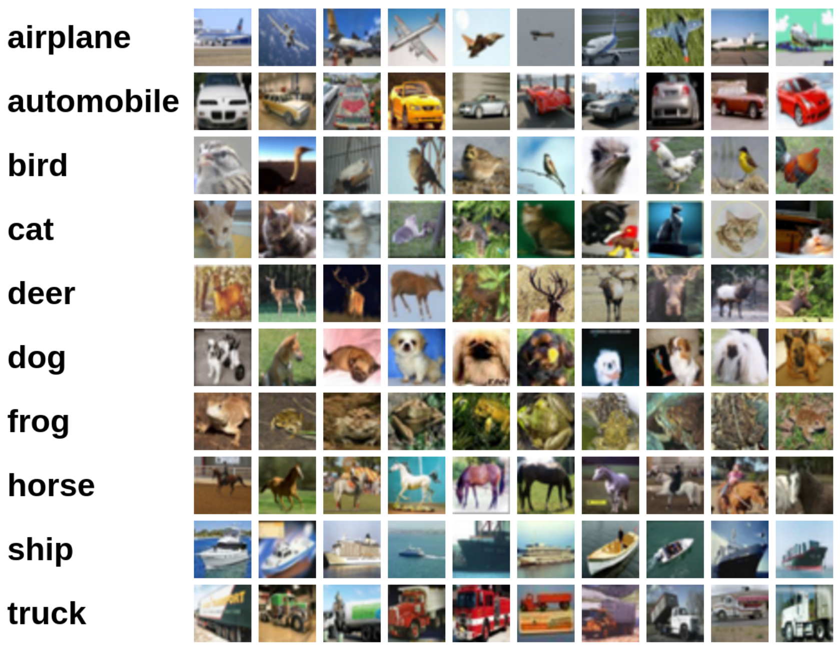


Figure 2.1: Example images in each class (Source: The CIFAR-10 dataset)

We used the binary version available on the webpage, and converted it to the standard TensorFlow format called TFRecords. This way, it is easier to mix and match data sets and network architectures. The binary format is converted to an Example protocol buffer, then it is serialized to a string, and finally writes the string to a TFRecords file. This slightly reduces the amount of memory used by the dataset, but this conversion greatly benefits larger datasets.

By today's standards, CIFAR-10 is a small dataset, compared to other ones available. MSCOCO has 2,500,000 labeled instances in 328,000 images for 91 different categories [44], ImageNet has 1.2 million training images with 1000 different classes [55] and Open Images has nearly 9 million images with over 6000 categories [37]. However, CIFAR-10 is a good enough dataset for proof of concept tasks, and is still commonly used as a benchmark. In this case, we are not aiming at state of the art results for this dataset, nor are we testing a new model architecture, but looking at the performance given different training methods. Thus, we consider this dataset to be good enough for our needs, as its size and training times should be generalizable to other cases.

2.2 Network architecture

For training our dataset, we use a relatively novel architecture presented by He et al. [28], called Residual Neural Networks (ResNets). As shown in Table 1.1, this architecture won the ImageNet classification and localization challenge in 2015, and enables to build much deeper networks than the ones trained until then. The reason to choose this architecture was that, in addition to be an interesting structure given the novelty, the original paper test the results on multiple datasets, including CIFAR-10, so we will have a base point for reference.

2.2.1 Residual Neural Networks

ResNets were created as an answer to the degradation problem observed when training very deep convolutional neural networks. As seen in section 1.3.2, deeper networks are crucial for better accuracy models, but some problems arises when trying to build deeper plain networks. Figure 2.2 shows how a plain deeper network performs worse than its shallower version.

Given that we just enlarge the network by adding layers that are identity mappings, there is no reason the deeper model should not perform as good as the shallow one. However, multiple tests demonstrate that there is an undesired behaviour, that generates a worse model. He et al. [28] argue that the problem seems not to be related to the notorious vanishing/exploding gradients, as this problem seems to have been largely addressed by normalized initialization of the layers, and intermediate normalization layers. They name the problematic as degradation, and found the main problem in the higher optimization difficulty of the more complex networks.

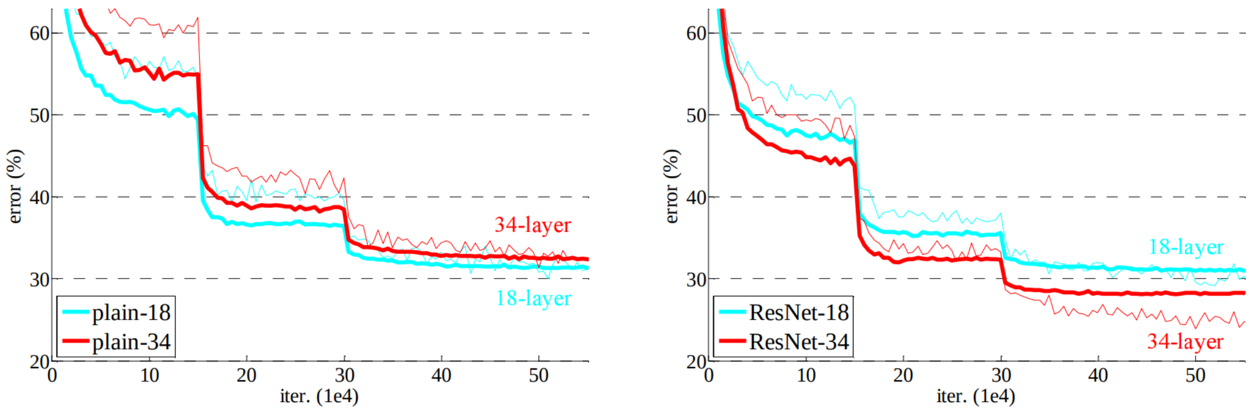


Figure 2.2: Comparison of plain networks and their residual counterparts in ImageNet. Thin lines denote training error and bold lines validation error. (Source: He et al. [28])

To solve this problematic they propose the addition of "shortcut connections", as shown in Figure 2.3, that skip one or more layers, and do not add any extra parameter or computation complexity.

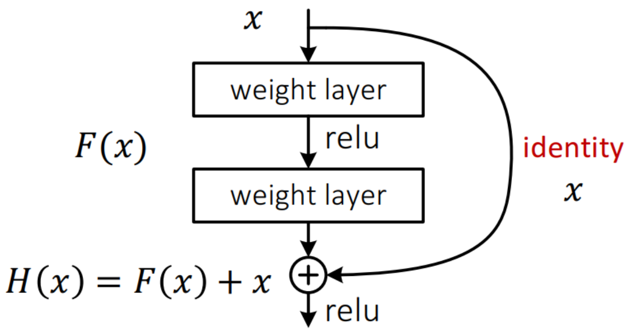


Figure 2.3: Residual learning building block. (Source: He [27])

They want to learn an underlying mapping $\mathcal{H}(\mathbf{x})$ by a few stacked layers, with $\mathbf{x}$ denoting the inputs to the first of these layers. With the hypothesis that multiple nonlinear layers can asymptotically approximate complicated functions, they assume that it will also asymptotically approximate residual functions. Then instead of approximating $\mathcal{H}(\mathbf{x})$, they approximate $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$, with the original function becoming $\mathcal{F}(\mathbf{x}) + \mathbf{x}$.

The idea is that given the case the identity is optimal, the solver may drive the weights of those layers to zero to approach the identity mappings. Although this may be a rare case, another case may be the optimal function is closer to an identity mapping than to a zero mapping, in which case it should be easier for the solver to find small fluctuations with reference to an identity mapping, than to learn the function as a new one.

The building block in Figure 2.3 can be defined as:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x} \tag{2.1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are the inputs and output vectors of the layers considered. For the case where a shortcut connection is done between two layers with different dimensions, a linear projection W_s is done to match them:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x} \tag{2.2}$$

With this approach, He et al. [28] won the classification, localization and detection tracks on the ImageNet challenge, setting new state of the art results. In the classification track, they used a 152-layer network, with a block architecture that had lower complexity than VGG network with 16 or 19 layers. The final result of 3.57% top-5 error on the test set was obtained with the ensemble of six models of different depth (two of those six were 152-layer models). Unfortunately, no word on training times is provided in the original publication.

2.2.2 ResNet-56 with bottlenecks

The network we will implement for our tests will be one of the networks presented in He et al. [28], specifically built for testing CIFAR-10 dataset with ResNets. They test ResNets with different depths, but follow the same pattern of blocks, just adding more repetitions of convolutions at multiple levels of depth.

They work with inputs of 32×32, and feed them to a convolution with kernel height and width of size 3. Then they use a stack of $6n$ convolutions of the same kernel width and height as before, on feature maps of sizes $\{32, 16, 8\}$, with $2n$ layers in each feature map size and $\{16, 32, 64\}$ filters respectively. The reduction of the map sizes is performed using convolutions with stride 2 at the end of each block, and the network ends with a global average pooling, a 10-way fully-connected layer, and a softmax, having a total of $6n + 2$ stacked weighted layers. The summary can be seen under the ResNet-56 column in Table 2.1, where the 56 layers are obtained with $n = 9$.

Table 2.1: ResNets summary

Layer name	Output size	ResNet-56	ResNet-83	# layers
conv_1	32×32	3×3, 16 stride 1		1
block_1	32×32	$\begin{bmatrix} 3 \times 3, 16 \\ 3 \times 3, 16 \end{bmatrix}$	$\begin{bmatrix} 1 \times 1, 16 \\ 3 \times 3, 16 \\ 1 \times 1, 64 \end{bmatrix}$	$n = 9$
block_2	16×16	$\begin{bmatrix} 3 \times 3, 32 \\ 3 \times 3, 32 \end{bmatrix}$	$\begin{bmatrix} 1 \times 1, 32 \\ 3 \times 3, 32 \\ 1 \times 1, 128 \end{bmatrix}$	$n = 9$
block_3	8×8	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix}$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix}$	$n = 9$
—	1×1	average pool, 10-d fc, softmax		1

In the original paper they present an alternative building block with the name of "bottleneck" that they use in the ImageNet training networks, that can be seen at the right in Figure 2.4. This alternative architecture uses the 1×1 layers for reducing and increasing dimensionality of the network, while avoiding expensive computations of convolutions with higher kernel size and large number of filters

The final structure we are going to test in our servers is the one shown in the ResNet-83 column of the Table 2.1. We will use $n = 9$ with the bottlenecks blocks for a total of 83 layers.

In order to built the model in TensorFlow we used a library, from the program itself, called TF-Slim [59]. This high level library contains the most common network architectures of the last years (AlexNet, VGG, Inception and ResNets), and also provides pretrained models for fine-tuning. The purpose of the library is to alleviate some of the tiring nomenclature TensorFlow uses when defining big models, and provide an easier programming model. However, it is a double-edged sword, because even though code can be simpler and easier to write, the extra layer can prevent full control of the model and customization the way it can be achieved with base TensorFlow.

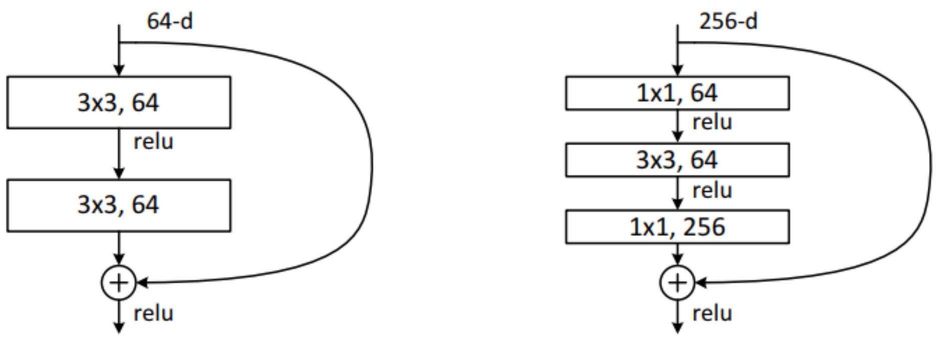


Figure 2.4: “Bottleneck” building block used in ImageNet architectures. (Source: He et al. [28])

Training parameters

In the paper they specify the training configuration, that we will mostly replicate in our experiments. They use weight decay of $\lambda = 0.0001$, for regularizing the convolution layers, so that the regularizing term $\Omega(\theta) = \frac{1}{2}\|\mathbf{w}\|_2^2$ is added to the objective function. TF-Slim is flexible enough for defining a general configuration that will be applied to all layers in our network, so we actually do not need to specify the parameters for each convolution, nor include the extra loss in our formula as this will be done automatically.

The optimizer will be gradient descent with Nesterov momentum [6] of 0.9, which is a first-order optimization method with better convergence rate guarantee than gradient descent in certain situations.

We will use the same layer initialization as they do in the original paper and that is described in He et al. [29]. It receives the name `variance_scaling_initializer` in TF-Slim, and improves the more common Xavier initialization [23] by taking into account the effect of rectifier nonlinearities. This enables them to train deeper models than using other more common initialization methods.

They use a batch size of 128 images, and begin with a learning rate of 0.1, dividing it by 10 at 32k and 48k iterations. It is possible to replicate this learning rate behavior in TensorFlow by using `tf.train.piecewise_constant`, and defining the interval steps for each learning rate value. These values are not exactly the same as we used in our tests, as some of them were not working for our distributed configuration. They will be specified in the corresponding tests results in sections 3.2 and 3.3.

Our loss function will be the common cross entropy loss, which measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). This is a common metric for classification problems in neural networks that works better for training purposes than plain accuracy or other formulas. Equation 2.3 computes the cross entropy between a true distribution p and an estimated distribution q.

$$H(p,q) = -\sum_{x} p(x)\log q(x) \tag{2.3}$$

The output of the softmax layer in our network would represent the q distribution, whereas the true labels will be one hot encoded vectors representing p. The final loss will be the mean value of the computed batch of images.

For data augmentation, we follow the same procedure for training: 4 pixel pad on each side of the image, and a 32×32 random crop, in addition of random horizontal flip. The testing dataset will not have any of those modifications.

2.3 Implemented architectures

We use the available implementation of ResNets in TF-Slim, and modify them to our particular network, as the models published are specifically targeted at ImageNet data. We now explain the properties and the considerations of the two modes implemented in this project and whose results will be shown in the next chapter.

2.3.1 Asynchronous mode

The first mode we implemented is the fully asynchronous mode. As explained in section 1.4.1, the neural network will be replicated to each one of the workers that, after computing the gradients of the given batch, will update the model in the parameter servers asynchronously. The neural network will be replicated to each one of the workers, using the built-in function `tf.train.replica_device_setter`. This function is fed with the cluster definition, that is, the IP directions for the workers and the parameter servers as seen in section 1.5.3.

The function will take care to allocate the proper graph nodes to the correct device, thus model variables will be saved in parameter servers and sent to workers where they will be used for computing the whole model. The cluster definition will set which nodes are parameter servers and which are workers, so the framework efficiently lets the user focus on the model definition, as the correct assignment of variables will be done under the hood.

This method is straightforward for servers that only have a single GPU, as we can define a cluster inside the code with two parameter servers and three workers with the following parameters:

```
cluster_spec = {
    "ps": ["nvb19-ib0:2222", "nvb20-ib0:2222"],
    "worker": ["nvb21-ib0:2222", "nvb22-ib0:2222", "nvb23-ib0:2222"]}
```

However, using this definition in our servers would make the program to use a single GPU in each node, wasting big amounts of computational power. Therefore, we will use the environment variable `CUDA_VISIBLE_DEVICES` as a workaround to allocate 4 workers in a single server. Also, server direction can have different port numbers, so they are seen by the program as two different locations in the network. For instance, `nvb21-ib0:2222` and `nvb21-ib0:2223` are considered different directions on the same device, thus we can use it to our advantage to put multiple workers in a single server. The following example shows how we could manage to launch 2 workers (with indexes 0 to 1) and a parameter server on the same node:

```
CUDA_VISIBLE_DEVICES=""  python script.py --ps_hosts=nvb19-ib0:2220 \
    --worker_hosts=nvb19-ib0:2221,nvb19-ib0:2222 --job_name=ps     --task_index=0 & \
CUDA_VISIBLE_DEVICES="0" python script.py --ps_hosts=nvb19-ib0:2220 \
    --worker_hosts=nvb19-ib0:2221,nvb19-ib0:2222 --job_name=worker --task_index=0 & \
CUDA_VISIBLE_DEVICES="1" python script.py --ps_hosts=nvb19-ib0:2220 \
    --worker_hosts=nvb19-ib0:2221,nvb19-ib0:2222 --job_name=worker --task_index=1
```

The parameter server has no available GPUs and all operations will be done using the CPUs. On the other hand, `/job:worker/task:0` will only see the GPU with index 0 as the available device, so it can only allocate operations in the CPU and that specific GPU. Finally, `/job:worker/task:1` will only see the GPU with index 1, as it is the index given to the `CUDA_VISIBLE_DEVICES`. As for the network directions, all of them point to node `nvb19-ib0` (the `-ib0` suffix is mandatory for all nodes given the platform protocol), but have different ports, thus will be seen as different nodes in the network. Table 2.2 summarizes the result of the call made above.

Table 2.2: Properties summary

Node	Port	Job name	Task Index	Available GPUs	GPU index
nvb19-ib0	2220	ps	0	None	—
nvb19-ib0	2221	worker	0	1	0
nvb19-ib0	2222	worker	1	1	1

With this configuration we can put up to 4 workers in a single server, each with a different GPU, ensuring that all GPUs in the server are being used. If we want to apply this procedure in more than one node, we can use:

```
# In node nvb19
CUDA_VISIBLE_DEVICES=""  python script.py --ps_hosts=nvb19-ib0:2220 \
    --worker_hosts=nvb19-ib0:2221,nvb19-ib0:2222,nvb20-ib0:2221,nvb20-ib0:2222 \
    --job_name=ps     --task_index=0 & \
CUDA_VISIBLE_DEVICES="0" python script.py --ps_hosts=nvb19-ib0:2220 \
    --worker_hosts=nvb19-ib0:2221,nvb19-ib0:2222,nvb20-ib0:2221,nvb20-ib0:2222 \
    --job_name=worker --task_index=0 & \
CUDA_VISIBLE_DEVICES="1" python script.py --ps_hosts=nvb19-ib0:2220 \
    --worker_hosts=nvb19-ib0:2221,nvb19-ib0:2222,nvb20-ib0:2221,nvb20-ib0:2222 \
    --job_name=worker --task_index=1

# In node nvb20
CUDA_VISIBLE_DEVICES="0" python script.py --ps_hosts=nvb19-ib0:2220 \
    --worker_hosts=nvb19-ib0:2221,nvb19-ib0:2222,nvb20-ib0:2221,nvb20-ib0:2222 \
    --job_name=worker --task_index=2 & \
CUDA_VISIBLE_DEVICES="2" python script.py --ps_hosts=nvb19-ib0:2220 \
    --worker_hosts=nvb19-ib0:2221,nvb19-ib0:2222nvb20-ib0:2221,nvb20-ib0:2222 \
    --job_name=worker --task_index=3
```

The application Greasy will be the one in charge to run each line in the corresponding node, to make it work properly. This configuration generates 2 workers in each node with only one parameter server in the first one. Table 2.3 summarizes the properties for this case, and Figure 2.5 shows the schema.

Table 2.3: Properties summary

Node	Port	Job name	Task Index	Available GPUs	GPU index
nvb19-ib0	2220	ps	0	None	—
nvb19-ib0	2221	worker	0	1	0
nvb19-ib0	2222	worker	1	1	1
nvb20-ib0	2221	worker	2	1	0
nvb20-ib0	2222	worker	3	1	2

This is a straightforward method for easily distributing workers and parameter servers among multiple nodes, making the replication of the models in a way we can fully use all the

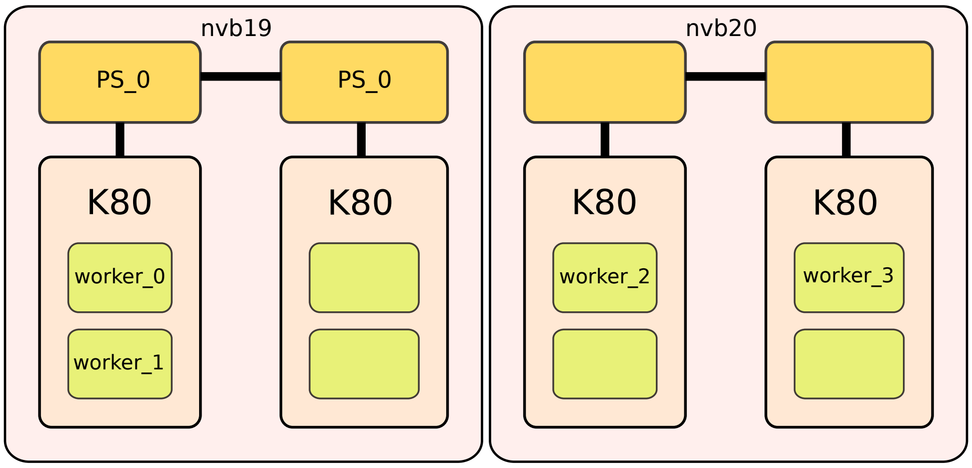


Figure 2.5: Schema for 2 nodes and 2 workers in each

available GPUs in the server for computing. The extent to which a parameter server in the same node can become a bottleneck is something that we will study more in depth in the next chapter. However, given the architecture of the MinoTauro this is the best way optimize the use of the servers, as allocating a full server only for this purpose will mean wasting a big amount of GPU resources.

2.3.2 Mixed asynchronous mode

As seen in the previous case, having multiple GPUs on the same server implies some constraints when building our distributed models. The mixed asynchronous mode is more suited for our architecture, as we can use synchronous mode between GPUs on the same node, and asynchronously update the model between servers. In other words, we are using synchronous replicas intra-node and asynchronous replicas inter-node. However, deploying the model and gathering the statistics in each GPU is more complex than the previous mode, because device placement needs to be done at the code level manually which adds complexity to the overall process.

The cluster definition is similar as the one before, but in this case each worker will have 4 available GPUs, so we will only have a worker for each node, instead of the 4 we had in the fully asynchronous. This is how the call will look like, for allocating a single parameter server and a single worker in a unique node:

```
CUDA_VISIBLE_DEVICES="" python scipt.py \
    --ps_hosts=nvb19-ib0:2220 --worker_hosts=nvb19-ib0:2221 \
    --job_name=ps --task_index=0 & \
CUDA_VISIBLE_DEVICES="0,1,2,3" python scipt.py \
    --ps_hosts=nvb19-ib0:2220 --worker_hosts=nvb19-ib0:2221 \
    --job_name=worker --task_index=0
```

This worker will have all 4 GPUs available, and the model will be responsible for placing the clones in the corresponding GPU. The function for enforcing device placement constraints in TensorFlow is `tf.device()`, and we can specify a loop to place the model to all available GPUs. This process will be again replicated to multiple nodes using the `tf.train.replica_device_setter`, and the total number of GPUs used will be the same. The key difference in this method is the way gradients are aggregated among all the clones in the node, which will make the iteration with a bigger effective size batch, improving the model.

The pseudocode in algorithm 1 aims to provide a general understanding of the main part of the mixed asynchronous procedure that places and computes the model in the 4 GPUs available and the collects the losses and gradients in each one and reports it back to the parameter server. The Figure 2.6 provides a general schema of a node with all the clones allocated.

```
for $i$ in $num\_gpus$ do
    with tf.device('/job:worker/gpu:%d' % i) do
        clone_images, clone_labels ← get batch of data
        predictions = model(clone_images)
        clone_loss = cross_entropy(predictions, clone_labels)
        clone_accuracy = accuracy(predictions, clone_labels)
    end
end
grad_op ← create optimizer
grads_and_vars ← create empty list
clone_losses ← create empty list
for $i$ in $num\_gpus$ do
    with tf.device('/job:worker/gpu:%d' % i) do
        clone_loss = get loss for clone $i$
        scaled_clone_loss = clone_loss / num_gpus
        clone_grad = compute gradients wrt trainable variables
        append clone_grad to grads_and_vars
        append scaled_clone_loss to clone_losses
    end
end
total_loss = sum clone_losses
total_gradients = sum grads_and_vars
make the train_op apply the gradients in total_gradients
```

Algorithm 1: Clone allocation and gradient computation

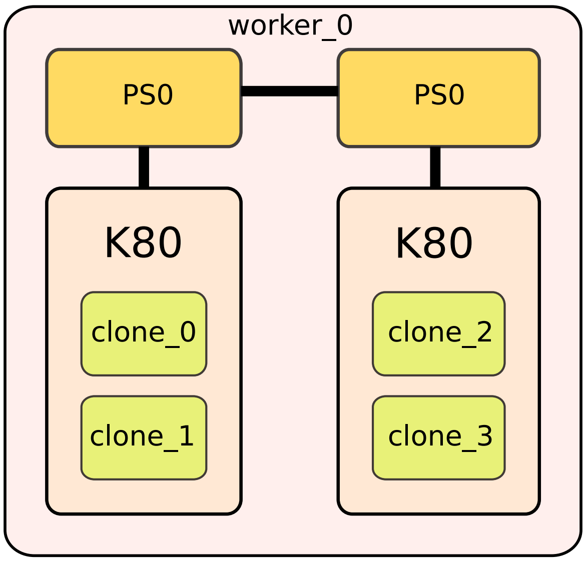


Figure 2.6: Node schema of the mixed asynchronous mode

2.4 Tools for assessing the model

As described in section 1.4, TensorFlow offers some tools to help during the training of the models. TensorBoard enables real-time monitoring of the running model, with a dashboard for following the evolution of the weights in the different layers during the training. Another interesting feature is the visualization of the operations defined in our model, as a full graph with interconnected nodes that represent each one of the operations declared in the code. It has different coloring options, so that nodes are colored according to their structure, or according to the device they are placed. For our case, this last option is the most interesting one, as we can assess if the device placement constraints we enforced in the code are correctly mapped.

Figure 2.7 shows the model build for the multi-GPU implementation. Inside the `get_data` node are all the operations related to loading and pre-processing the dataset. This node is then connected to the multiple clones, each one in a different GPU, as the colors in Figure 2.8a show. All the nodes that have not been given a specific device constraint will be placed in the fastest device available of either `/job:worker/task:0` or `/job:ps/task:0`, as the `get_data` node. The top row of clones contains the operations for the backpropagation loop, with the gradient computations and nodes related to the optimization process that are built by the program.

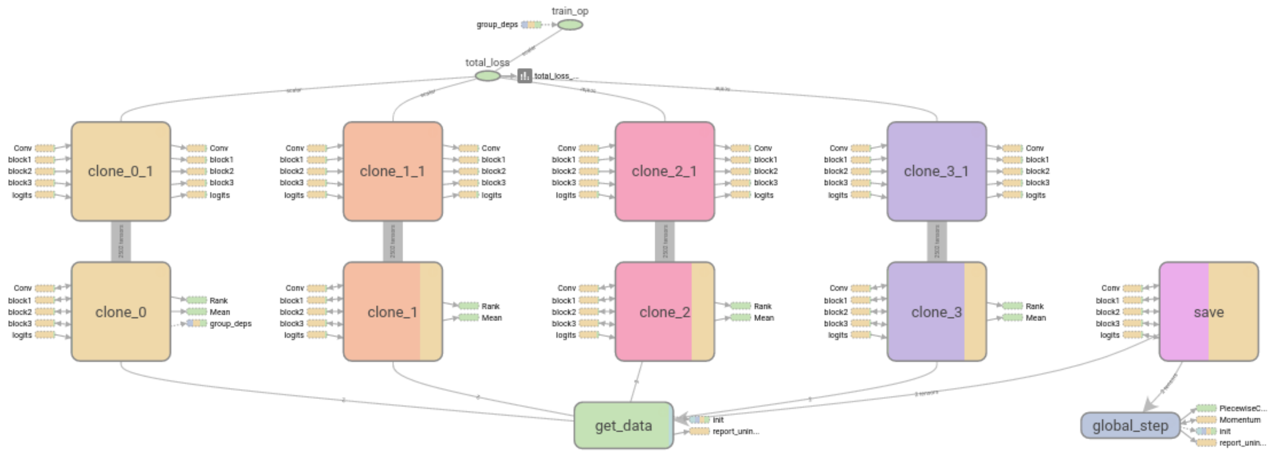
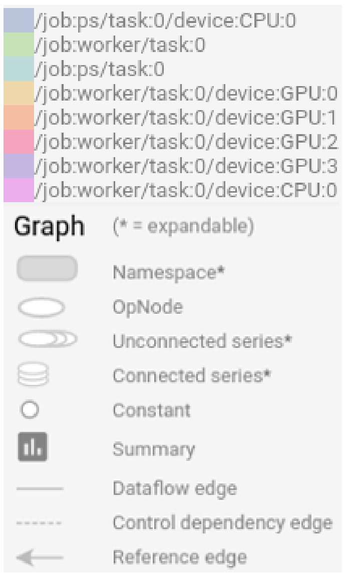


Figure 2.7: Multi-GPU version ResNet-56 ops with device placement coloring

According to the color representation, each clone is placed on a different GPU, but parts of the model will be executed on `/job:worker/task:0/gpu:0`. If we navigate to the corresponding nodes, the batch normalization layers are responsible for this misconfiguration. Despite having enforced the correct GPU, the creation of those nodes in the code is beyond our working layer, so it is difficult to know if it is the intended configuration or not. However, we are able to trace another device misconfiguration related to the computation of accuracy and loss metrics, as shown in Figure 2.9a.

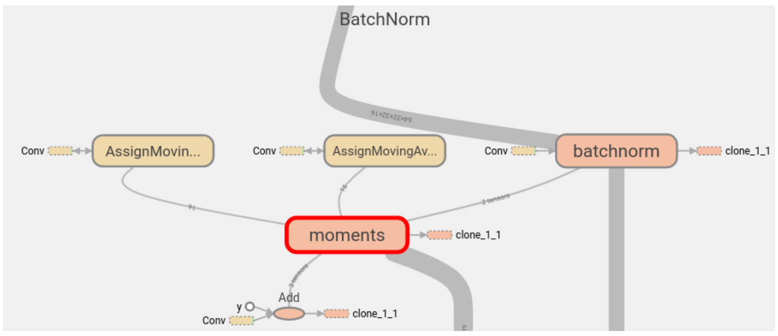
Those nodes are effectively placed by us, but a coding error in the indentation kept them out of the device placement constraint. The device they fall back by default is `/job:worker/task:0`, and more precisely `/gpu:0`, as it is the default device where TensorFlow ops will be allocated. Correcting the indentation and declaring the functions inside the corresponding device constraint solved the placement problem, as seen in Figure 2.9b.

Other more in-depth tools for supervising device placement and node computation times is the Timeline tool. At a given step of the computation, we can output a JSON file that contains all the information of the executed operations for that step. It can help detect bottlenecks in



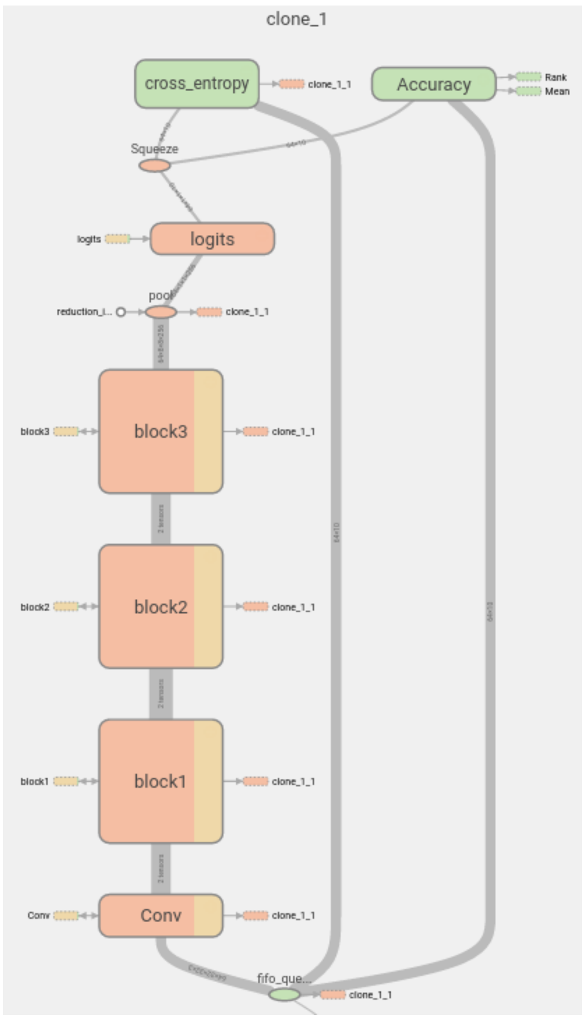
/job:ps/task:0/device:CPU:0
/job:worker/task:0
/job:ps/task:0
/job:worker/task:0/device:GPU:0
/job:worker/task:0/device:GPU:1
/job:worker/task:0/device:GPU:2
/job:worker/task:0/device:GPU:3
/job:worker/task:0/device:CPU:0
Graph
(* = expandable)
Namespace*
OpNode
Unconnected series*
Connected series*
Constant
Summary
Dataflow edge
Control dependency edge
Reference edge

(a) Graph and color device legends

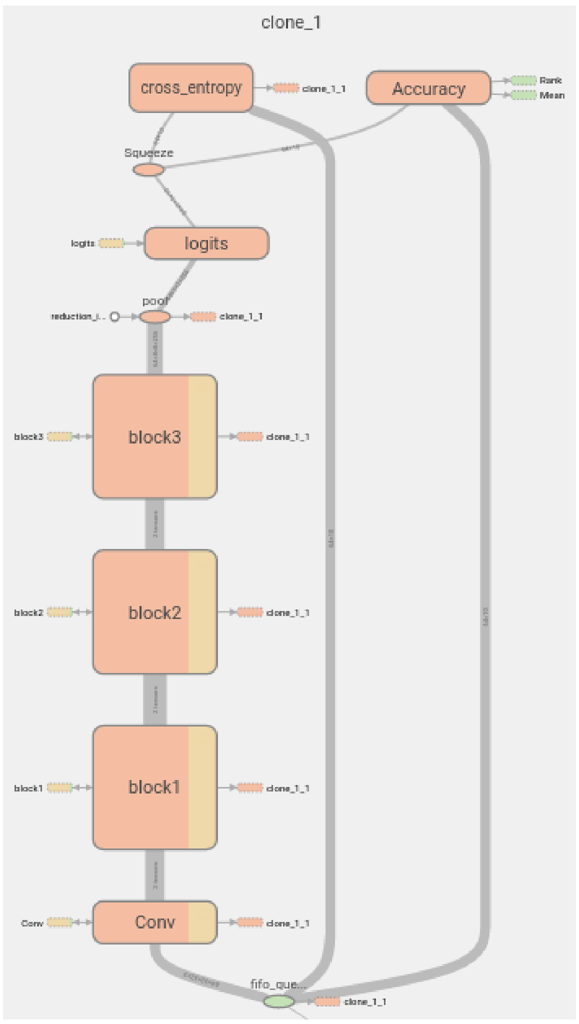


(b) Example of batch normalization layer device placement

Figure 2.8: TensorBoard screenshots



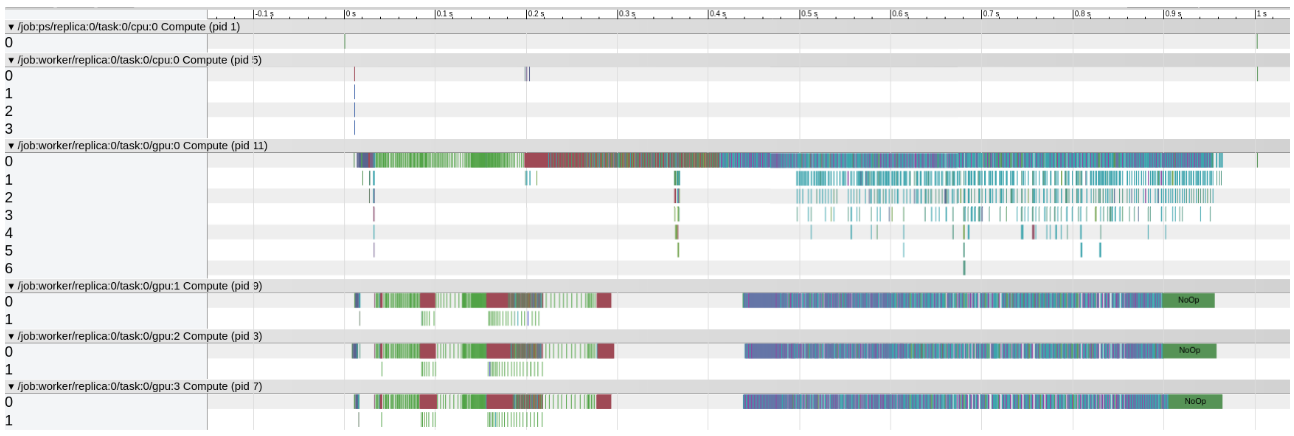
(a) Clone ops with unconstrained placement



(b) Clone ops with constrained placement

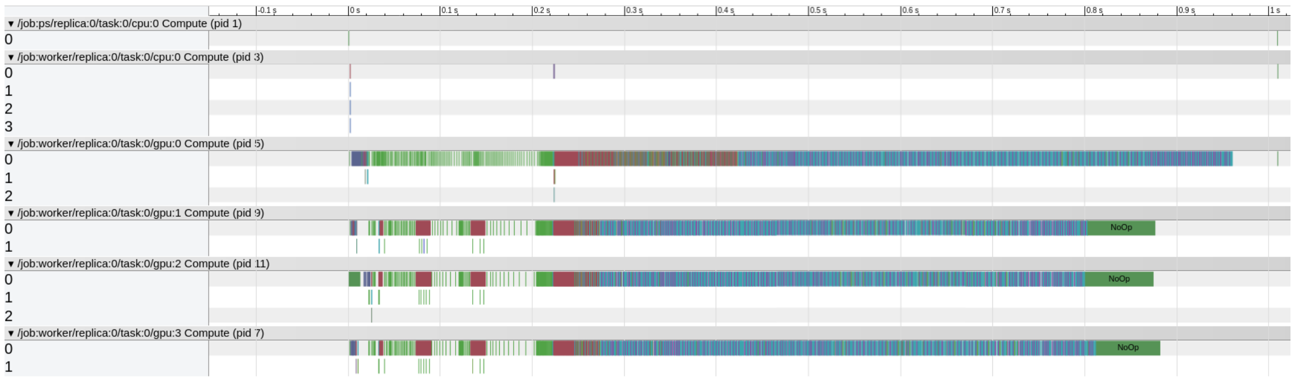
Figure 2.9: Deeper inspection of each clone ops

operations, or again check whether the device placement is correct. Returning to the previous case, Figure 2.10 shows the Timeline for the case with the bad indentation and the misconfigured device placement for the metrics. `/gpu:0` is working during all period, but the other GPUs are idle for some time between the 0.3s and 0.45s marks. The problem was the computation falling back to `/gpu:0` made the models in other GPUs wait until they received the results. Solving the indentation problem solved the issue as seen in Figure 2.11, and idling times on GPUs is solved. However, the whole computation time has not been reduced, and `/gpu:0` is still the bottleneck, mainly because it needs to compute extra operation for collecting all the gradients.



-0.1 s
0 s
0.1 s
0.2 s
0.3 s
0.4 s
0.5 s
0.6 s
0.7 s
0.8 s
0.9 s
1 s
▾ /job:ps/replica:0/task:0/cpu:0 Compute (pid 1)
▾ /job:worker/replica:0/task:0/cpu:0 Compute (pid 5)
▾ /job:worker/replica:0/task:0/gpu:0 Compute (pid 11)
▾ /job:worker/replica:0/task:0/gpu:1 Compute (pid 9)
▾ /job:worker/replica:0/task:0/gpu:2 Compute (pid 3)
▾ /job:worker/replica:0/task:0/gpu:3 Compute (pid 7)
NoOp

Figure 2.10: Tracing of the multi-GPU version ResNet-56 with unconstrained placement ops



-0.1 s
0 s
0.1 s
0.2 s
0.3 s
0.4 s
0.5 s
0.6 s
0.7 s
0.8 s
0.9 s
1 s
▾ /job:ps/replica:0/task:0/cpu:0 Compute (pid 1)
▾ /job:worker/replica:0/task:0/cpu:0 Compute (pid 3)
▾ /job:worker/replica:0/task:0/gpu:0 Compute (pid 5)
▾ /job:worker/replica:0/task:0/gpu:1 Compute (pid 9)
▾ /job:worker/replica:0/task:0/gpu:2 Compute (pid 11)
▾ /job:worker/replica:0/task:0/gpu:3 Compute (pid 7)
NoOp

Figure 2.11: Tracing of the multi-GPU version ResNet-56 with constrained placement ops

The Timelines showed above were obtained from a run in a single node, as the distributed version was still in development while writing this thesis, so no results of Timelines for distributed versions could be generated. It was still possible to obtain the Timeline for a single node, as if it were in a local machine, but no stats of the global process could be gathered.

3. Results

This chapter shows the different kind of tests conducted in the platform. In the first section, short tests for depicting the scalability are shown, and in the second section the actual training runs are evaluated. All multinode tests are done using the asynchronous methods, as it is the only one fully working on the open source version of TensorFlow as the writing of this thesis.

3.1 Testing scalability

These tests were done using the workload explained in the previous section, with a ResNet-56 with bottleneck blocks (Figure 2.4) which increases its depth to 83 layers, and CIFAR-10 dataset. We evaluated each implementation with the same batch size, as we were testing the throughput of the system. The idea is that for distributed systems, larger batch sizes are more beneficial, as this requires less iterations per epoch, and we try to avoid network overheads as much as possible. According to Bengio [8], changing this hyperparameter will have a bigger impact on training times than on testing performance, although having fewer model updates per epoch will require visiting more examples in order to achieve the same error.

We also test different configurations of workers and parameter servers (PS), in order to be sure that the latter does not overcome a bottleneck when we increase the number of models that it receives and needs to update.

For reference, some scalability tests done in TensorFlow can be found at Abadi et al. [1], albeit they probably use the internal version deployed at Google servers. They test the scalability of training Inception-v3 using ImageNet dataset, with multiple replicas in both synchronous and asynchronous settings. In their server configurations each worker has 5 CPUs and 1 GPU (NVIDIA K40), whereas their parameter servers have 8 CPUs with no GPUs. Instead of showing multiple working configurations of number of workers versus number of PS, they enable 17 PS for all tests, and modify the number of workers from 25 up to 200.

Our configuration is somewhat different, in the sense that all our servers have 4 NVIDIA K40 GPUs, so if we use a full server as a PS, we will be wasting 4 GPUs. Given the constraints of the architecture, we decided to allocate PS in the same nodes that contain the workers, but limiting their hardware use to only CPUs. As we did not know if that would be a limiting factor, we tried several configurations for both asynchronous and mixed-asynchronous workers, with the results shown in Figure 3.1.

The results are somewhat expected, as the fully asynchronous implementation is faster and scales better than the mixed-asynchronous. In the latter, the fact that losses and gradients need to be aggregated and processed generates some overhead that becomes more notorious as we increase the number of nodes. Take into account that in the asynchronous results we use each GPU as an independent worker, so total number of replicas will be 4, 8, 12, 16, 20, 24,

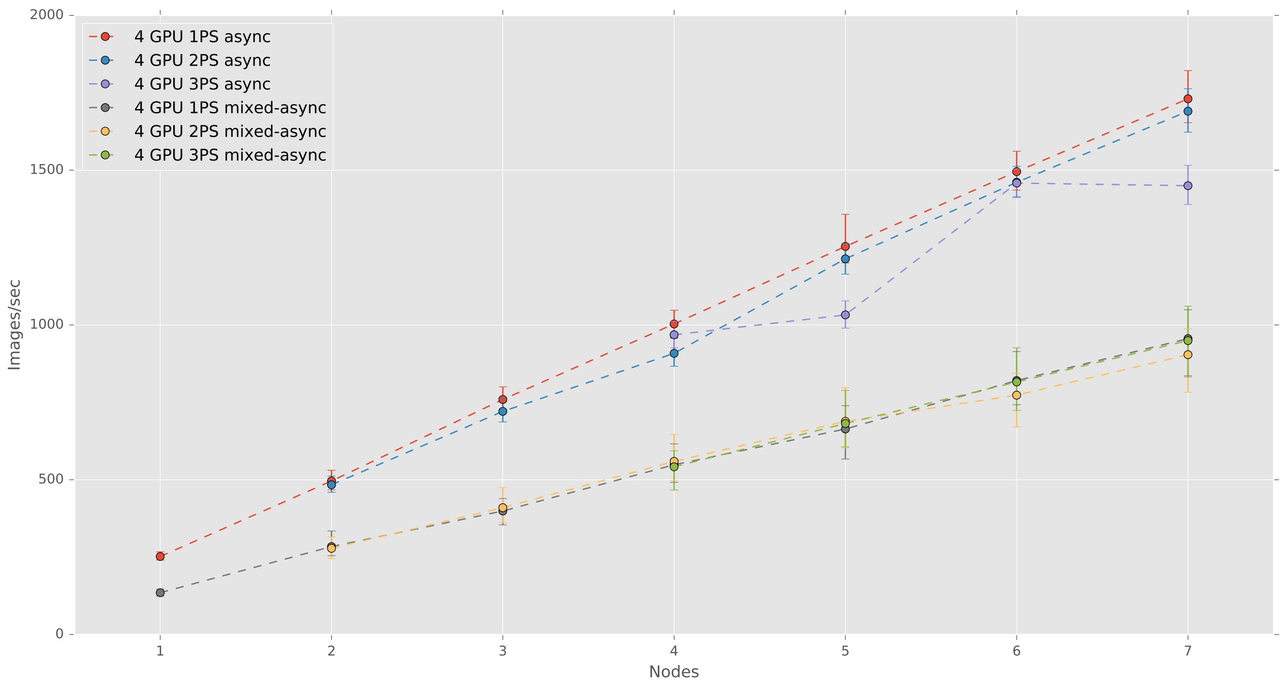


Figure 3.1: Tests of the workload in multiple server configurations.

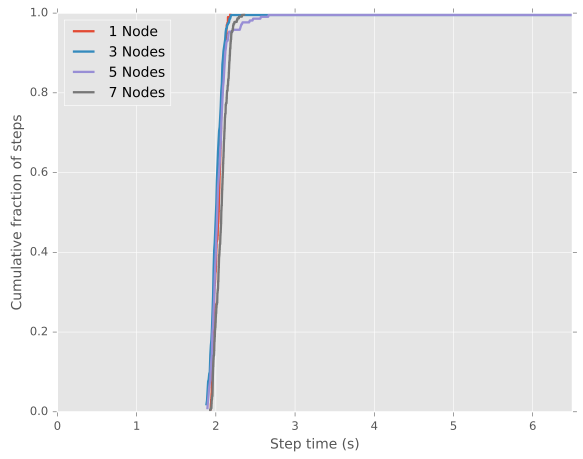
28. In the mixed asynchronous case, there will be as much workers as number of nodes, and each will have 4 GPUs inside working in synchronous mode, but to the PS it would look as having 1, 2, 3, 4, 5, 6, 7 model replicas. Figures 2.5 and 2.6 in the previous section depict the hardware allocation of each mode.

If we look at the tests with different combinations of PS, there is no gain in using more than 1 PS, at least for the scale tested. The are some inconsistencies in the results when using more than 1 PS, and although we ran the tests at least two times, it is difficult to guess if these anomalies could be generated by a congested network, as other groups running jobs at MinoTauro use the same network. All in all, we did not observe any case in which the 1 PS server configuration was much slower than any other, so we concluded that, at least for these configuration levels, 1 PS server was enough.

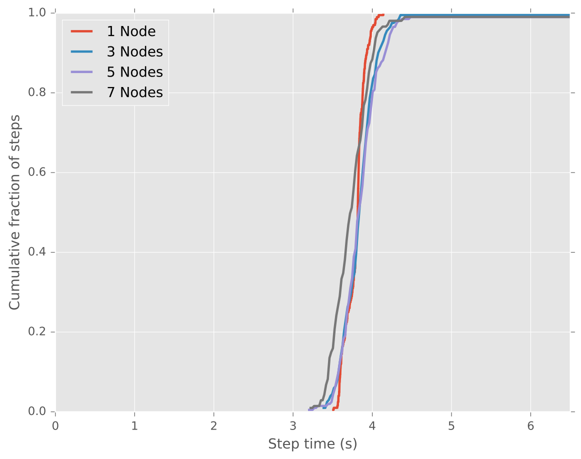
We included error bars in Figure 3.1 for portraying best and worst case scenarios registered, but a better study of step times is displayed in Figure 3.2. We plot the distribution of the 500 step times for both modes, with fully asynchronous workers having a faster speed time than mixed asynchronous ones. Step times at different scalability sizes are very consistent for both methods, but there is more variability for the mixed asynchronous for the reasons explained before. Those figures show that for the fully asynchronous case 70% of the steps take less than 2 seconds to complete, and overall most of them are below 2.5 seconds. For the mixed asynchronous, 80% of the steps take less than 4 seconds, and the range of step times is higher, from 3.5 to almost 4.5 seconds.

3.2 Asynchronous training

For the asynchronous training analysis we trained 3 different configurations: 1, 4 and 8 workers. As said before, each worker will have be independent GPU, thus we will be using 1 node for the first two cases and 2 nodes when using 8 workers. We tried to use the same training parameters as the original publication for our network, but the model did not converge



(a) Step times in asynchronous mode with 1 GPU as 1 worker



(b) Step times in asynchronous mode with 4 GPUs as 1 worker

Figure 3.2: Step times for both modes of parallelism and 1 PS

and was impossible to successfully train the network. Keeping the batch size of 128 images, we reduced the learning rate to have an initial value of 0.01 instead of 0.1. We used the same parameters for all the configurations, whose results can be seen in Figure 3.3.

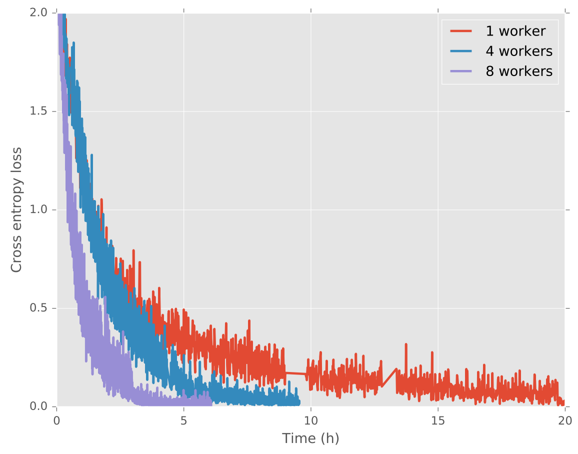
The plotted curves have some noise, which may indicate that the batch size used is too small, and the results in successive iterations is not too consistent. We observe some expected results, as loss and train error as a function of images seen by the model decrease faster for the single worker than for the multiple worker versions (Figure 3.3b). The fact that multiple models are being updated, with gradients updating slightly old versions of the model, make the single worker version more effective. However, when looking at the time scale, training time and convergence are much quicker in the multiworker versions (Figure 3.3a). The small periods of unavailable data in the single worker version where due to model crashes, that needed to be restarted from the last checkpoint. For the single worker version we need 22 hours of training time until convergence, for a testing error of 10.37%. The 4 worker version converges after 7.5 hours, but with a test error of 15.59% and, finally, the 8 worker version converges in just 3.4 hours with a similar test error of 15.54%.

Despite the quicker convergence, there is a big regression in testing error, making the multiworker models much worse than the single worker. Even this configuration is far away of the results showed in the original publication, with a test error of 6.97% for the ResNet-56. All in all, it is interesting that the two multiworker versions perform almost equally, in terms of train and test accuracy, but the one with 8 workers converges two times faster than the other.

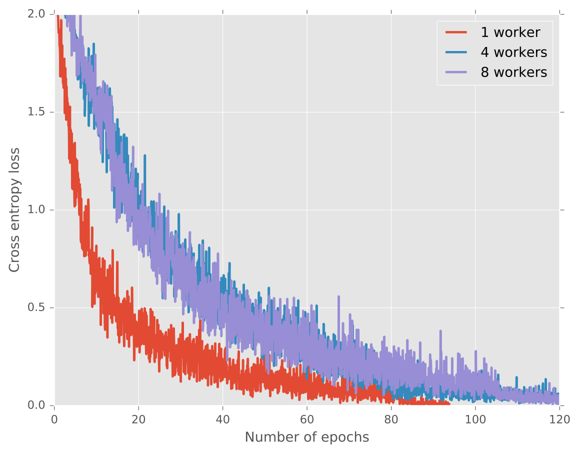
3.3 Mixed asynchronous training

Similarly to the results in the asynchronous case, we tested the mixed asynchronous with 1 and 2 nodes. In this case, the number of GPUs in each training will be 4 and 8, as before, but they will work synchronously in groups of 4. Plots in Figure 3.4 show the results obtained with this configuration. We will use a batch of 64 images in each GPU, meaning that the effective batch at every iteration will be of 256 images. We used a learning rate of 0.1, and divided it by 10 after 100 epochs.

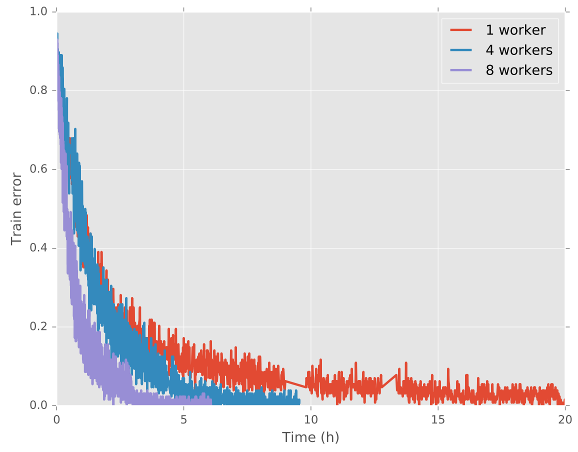
We can quickly see that lines for these plots are smoother and less noisy than the ones



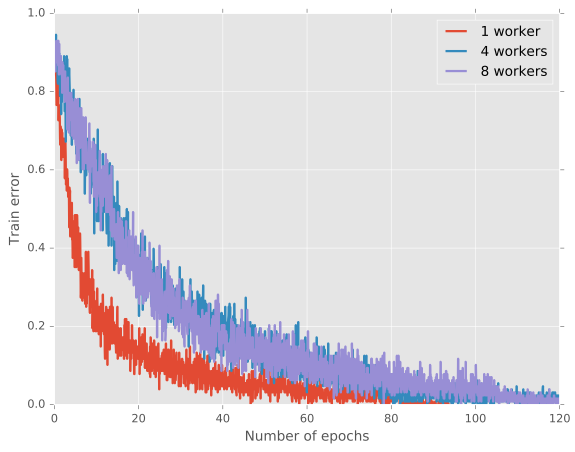
(a) Loss as a function of time



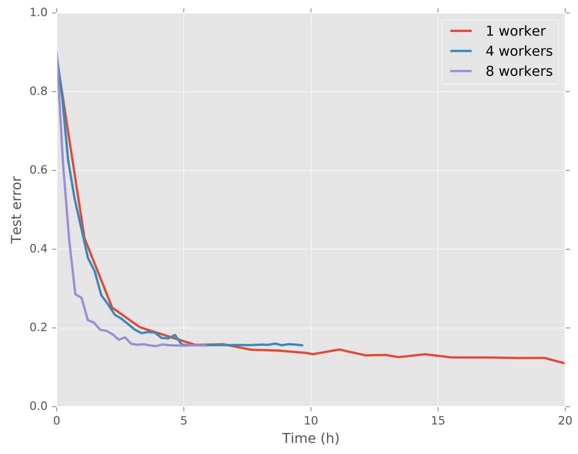
(b) Loss as a function of iterations



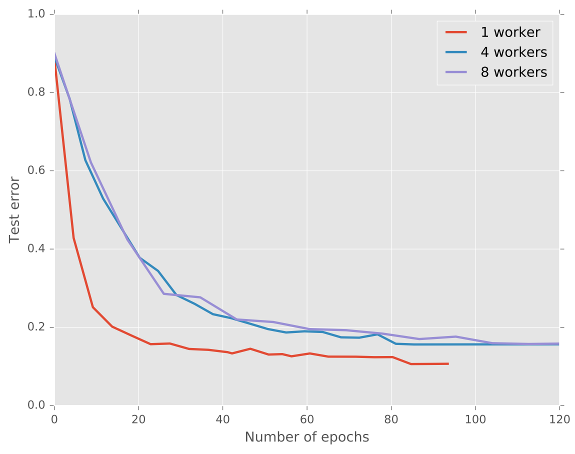
(c) Train error as a function of time



(d) Train error as a function of iterations



(e) Test error as a function of time



(f) Test error as a function of iterations

Figure 3.3: Asynchronous results

obtained in Figure 3.3, even though the plotting frequency is the same. The explanation is the bigger effective batch size used in this test makes the results at every step less variable. From Figure 3.4b we can conclude that there is a regression in both the loss and error as a function of images processed, indicating that the 2 workers need to see more images to obtain the same error than the 1 worker version. This is probably caused by the inconsistencies when updating the gradients on models slightly different that occurs in the asynchronous case. However, this regression is not enough to make the model perform worse when looking at the metrics as a function of time.

Figures 3.4a and 3.4c show that despite the regression caused by the asynchronous updates, the throughput of the 2 workers is good enough to make the model converge in less time than the 1 worker version. While the 2 workers need between 3 and 4 hours till convergence, the 1 worker version takes little more than 6 hours. This behaviour is consistent with the results seen in the scalability test, as the scalability is almost linear.

The behaviour for the test errors is similar as the one seen in the previous cases, where 2 workers outperform 1 worker in terms of iterations but not in clock time. The final error for the 2 workers is 11.15% whereas the error for the 1 worker run is 11.48%, still far from the 6.97% error reported in He et al. [28] with the ResNet-56 using basic bottlenecks.

3.4 Model comparison

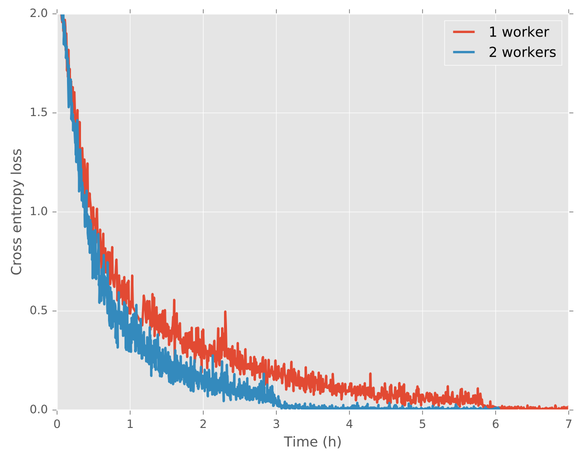
We present the aggregated results of all the test summarized in Table 3.1. We compute the speedup with respect to a single worker, for the fully asynchronous methods, and with respect to a single node for the mixed asynchronous version. The fact that we used different hyperparameters, and the training modes are different makes it easier to evaluate them separately.

The speedups obtained are similar to the ones seen in the proof of concept test done in section 3.1. The model that best performed is the single worker version of the fully asynchronous, followed by the mixed asynchronous implementations. The network we implemented, performs worse in all the configurations tested than the simpler version presented in He et al. [28] with basic bottlenecks.

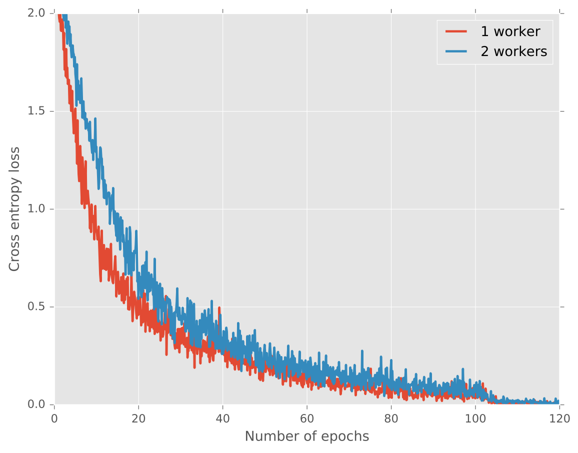
Table 3.1: Model comparison summary

Mode	Number workers	Number GPUs	Steps per second	Batch size	Initial learning rate	Test error	Train time (h)	Speedup
Async	1	1	0.47	128	0.01	10.37%	22	1x
	4	4	1.73	128	0.01	15.59%	7.5	3x
	8	8	3.73	128	0.01	15.54%	3.4	6.5x
Mixed async	1	4	0.99	256	0.1	11.48%	6.8	1x
	2	8	1.85	256	0.1	11.15%	3.5	2x

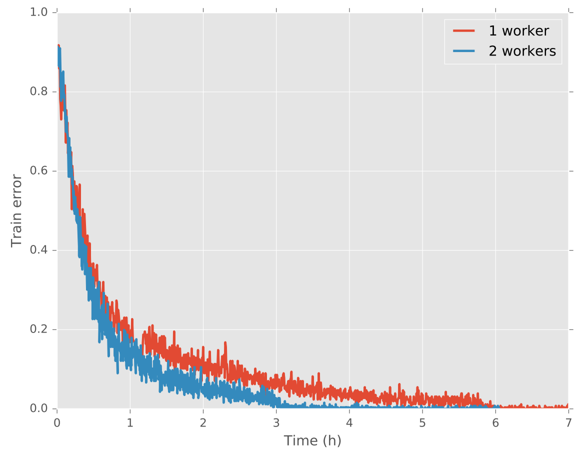
We present the confusion matrix for the best performing model, asynchronous single worker, in Table 3.2. Columns represent true classes, whereas rows show the predicted values. The class dictionary is the following: {0: airplane, 1: automobile, 2: bird, 3: cat, 4: deer, 5: dog, 6: frog, 7: horse, 8: ship, 9: truck }. The best predicted class is 'automobile', while the worst is 'cat', that gets often classified as 'dog', the second worst performing class.



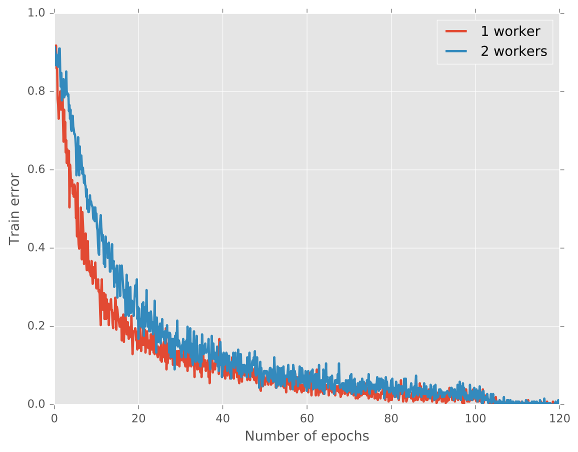
(a) Loss as a function of time



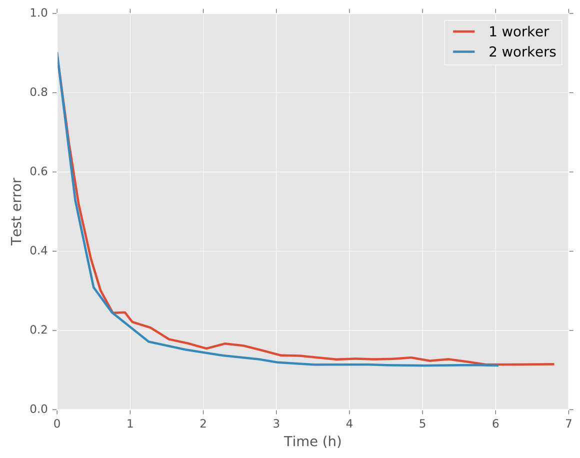
(b) Loss as a function of iterations



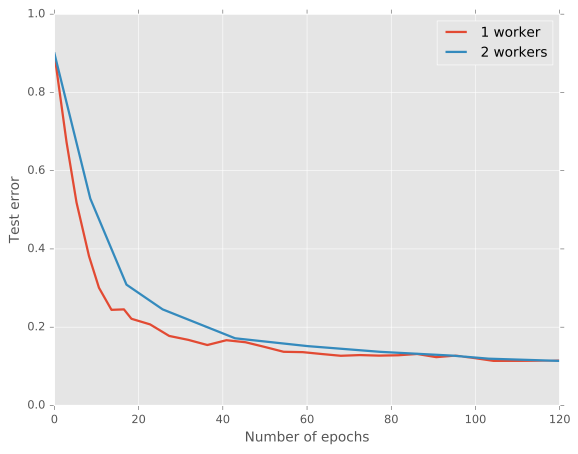
(c) Train error as a function of time



(d) Train error as a function of iterations



(e) Test error as a function of time



(f) Test error as a function of iterations

Figure 3.4: Asynchronous results

Table 3.2: Confusion matrix of the best performing model

	0	1	2	3	4	5	6	7	8	9	Error
0	903	5	13	7	6	5	7	10	22	9	9.7%
1	4	957	1	3	1	2	1	3	16	35	4.3%
2	25	0	853	19	17	25	23	5	2	0	14.7%
3	13	2	27	808	23	89	21	18	6	6	19.2%
4	12	3	45	31	890	17	10	16	1	1	11.0%
5	0	1	21	88	17	828	11	19	1	1	17.2%
6	2	0	28	17	21	8	920	2	1	3	8.0%
7	5	0	8	19	22	20	4	925	1	1	7.5%
8	26	8	3	6	3	4	2	2	945	11	5.5%
9	10	24	1	2	0	2	1	0	5	933	6.7%

4. Conclusions

Distributed training strategies for deep learning architectures will become more important as the size of datasets increases. Then, it is important to understand which are the most efficient ways to perform distributed training, in order to maximize the throughput of the system, while minimizing the accuracy and model regression. For this purpose, we tested two possible implementations that work with asynchronous replicas, and evaluated the results.

We named this two modes of parallelism as (fully) asynchronous and mixed asynchronous. The first one is somewhat simple to implement, as it relies on placing replicas of a model in each GPU available, that we have specified using the environment tools MinoTauro provides. The other method uses a combined strategy of intra-node synchronisation and inter-node asynchronous replicas. These are the only strategies that we have been able to implement in MinoTauro, as the fully synchronous version is still not available in the open-source version of TensorFlow.

We were interested in observing the performance of both the facility and the framework, under these two implementations with multiple server configurations, testing the scalability of the platform. We observed that under the fully asynchronous mode, the cluster of servers had a higher throughput than the mixed mode, and we concluded that it could be caused by the more complex implementation of the latter mode. We also tried to parametrize the amount of parameter servers needed to perform our following experiments, given that we used a very specific configuration that we had to enforce given the cluster constraints. The main reason to put the parameter servers inside the same worker nodes was to optimize the total number of different nodes used, trying to reduce the footprint of the long training jobs in the cluster. We concluded that when using up to 7 nodes, the use of a single parameter server was enough for a correct working process.

We then tested some training workload to compare the final accuracies of the model with respect to the time of training, as our main objective is to reduce training times without losing final model accuracy. To that end, we implemented a variant of the ResNet-56 with bottleneck blocks specially built for the CIFAR-10 dataset, and trained it with the different parallel modes. We found the best performing model was the single GPU worker implementation, with an error of 10.37%, worse than the one presented in the original paper implementation, although we were testing a different configuration. This model was by far the slowest to train, but the multiple worker trainings with 4 and 8 workers in the fully asynchronous mode could not achieve similar results in terms of model accuracy, even though trainings time were up to 6 times faster.

When we tested the alternative mode (mixed asynchronous), model accuracy was up to 11.15% for 2 workers, which was not as good as the single worker, but was good enough compared to previous tests. This mode, despite its added complexity, seems to perform particularly good, as it gets a good trade-off between training time and model accuracy.

All in all, our model configuration did not obtain good accuracy results, and probably a deeper study of hyperparameter optimization should be done to improve that part of the project. However, we successfully implemented our distributed modes despite the scarce documentation available and obtained meaningful results.

From a personal point of view, this project began as an implementation of the Deep Art algorithm [22] in recently open-sourced TensorFlow, an alternative and curious application of deep neural networks. This meant learning many new concepts on this field, and understand how those concepts could be translated to the new framework. This was a very useful introduction for both the theory behind these algorithms and to settle the basics of TensorFlow inner workings. With the publication of the distributed version, a whole new area of exploration and research was available, and using the knowledge on the framework from previous months, we decided it would be interesting to have a working distribution at MinoTauro for future projects to use. Therefore, we began investigating and experimenting with distributed workloads and their implementations, and ended up being the content of this thesis.

4.1 Future work

There are still many areas for research in this domain, as this project has just been an introductory implementation to build upon, now that a basic implementation is fully working. First, when the synchronous mode is fully implemented in the public version of TensorFlow it should be straightforward to implement, as basic tests were already done, but the inner functions of TensorFlow were not working as expected. Synchronous replicas open a new window of test, with the addition of backup workers, that should alleviate some of the drawbacks of synchronisation.

Second, it would be interesting to test some application that exploits model parallelization instead of data parallelization. Although it is said that it does not scale as well as the latter and the configuration is pretty hard because operations and layers need to be placed manually, it could be an interesting experiment.

Finally, it could be interesting to evaluate the effect of network congestion to training times. As the network of the cluster is shared among all the jobs, it is unclear if slowdowns during training could be caused by a congested network, and to which extend it affects the process.

Bibliography

[1] M. Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *ArXiv e-prints* (2016). arXiv: 1605.08695 [cs.DC].

[2] M. Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *ArXiv e-prints* (2016). arXiv: 1603.04467 [cs.DC].

[3] R. Adolf et al. "Fathom: Reference Workloads for Modern Deep Learning Methods". In: *ArXiv e-prints* (2016). arXiv: 1608.06581 [cs.LG].

[4] A. G. Anderson et al. "DeepMovie: Using Optical Flow and Deep Neural Networks to Stylize Movies". In: *ArXiv e-prints* (2016). arXiv: 1605.08153 [cs.CV].

[5] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling Up Machine Learning: Parallel and Distributed Approaches*. New York, NY, USA: Cambridge University Press, 2011.

[6] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. "Advances in Optimizing Recurrent Networks". In: *ArXiv e-prints* (Dec. 2012). arXiv: 1212.0901 [cs.LG].

[7] Yoshua Bengio. "Learning deep architectures for AI". In: *Foundations and Trends in Machine Learning* 2.1 (2009), pp. 1–127. DOI: 10.1561/2200000006.

[8] Yoshua Bengio. "Practical Recommendations for Gradient-Based Training of Deep Architectures". In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoie Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 437–478. DOI: 10.1007/978-3-642-35289-8_26. arXiv: 1206.5533 [cs.LG].

[9] Barcelona Supercomputing Center BSC. *About BSC*. 2016. URL: http://www.bsc.es/about-bsc.

[10] Barcelona Supercomputing Center BSC. *MinoTauro User's Guide*. 2016. URL: http://www.bsc.es/support/MinoTauro-ug.pdf.

[11] A. J. Champandard. "Semantic Style Transfer and Turning Two-Bit Doodles into Fine Artworks". In: *ArXiv e-prints* (2016). arXiv: 1603.01768 [cs.CV].

[12] Jianmin Chen et al. "Revisiting Distributed Synchronous SGD". In: *International Conference on Learning Representations Workshop Track*. 2016. arXiv: 1604.00981 [cs.LG].

[13] K. Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *ArXiv e-prints* (2014). arXiv: 1406.1078 [cs.CL].

[14] Adam Coates et al. "Deep learning with COTS HPC systems". In: *Proceedings of the 30th International Conference on Machine learning (ICML-13)*. Ed. by Sanjoy Dasgupta and David Mcallester. Vol. 28. 3. JMLR Workshop and Conference Proceedings, 2013, pp. 1337–1345. URL: http://jmlr.org/proceedings/papers/v28/coates13.pdf.

[15] Henggang Cui et al. "GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: ACM, 2016, 4:1–4:16. DOI: 10.1145/2901318.2901323.

[16] Jeff Dean. *Large-Scale Deep Learning with TensorFlow*. Association for Computing Machinery (ACM). 2016. URL: https://youtu.be/vzoe2G5g-w4?t=51m26s.

[17] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. 2004, pp. 137–150. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

[18] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1223–1231. URL: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf.

[19] Andy Feng, Jun Shi, and Mridul Jain. *CaffeOnSpark Open Sourced for Distributed Deep Learning on Big Data Clusters*. Yahoo. 2016. URL: http://yahoohadoop.tumblr.com/post/139916563586/caffeonspark-open-sourced-for-distributed-deep.

[20] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4 (1980), pp. 193–202. DOI: 10.1007/BF00344251.

[21] L. A. Gatys et al. "Preserving Color in Neural Artistic Style Transfer". In: *ArXiv e-prints* (2016). arXiv: 1606.05897 [cs.CV].

[22] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. "A Neural Algorithm of Artistic Style". In: *ArXiv e-prints* (2015). arXiv: 1508.06576 [cs.CV].

[23] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: vol. 9. 2010, pp. 249–256. URL: http://www.jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning". Book in preparation for MIT Press. 2016. URL: http://www.deeplearningbook.org.

[25] A. Graves, A.-r. Mohamed, and G. Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In: *ArXiv e-prints* (2013). arXiv: 1303.5778 [cs.NE].

[26] A. Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". In: *ArXiv e-prints* (Feb. 2015). arXiv: 1502.05767 [cs.SC].

[27] Kaiming He. "Deep Residual Networks. Deep Learning Gets Way Deeper". In: *ICML 2016 tutorial*. 2016. URL: http://icml.cc/2016/tutorials/icml2016_tutorial_deep_residual_networks_kaiminghe.pdf.

[28] K. He et al. "Deep Residual Learning for Image Recognition". In: *ArXiv e-prints* (2015). arXiv: 1512.03385 [cs.CV].

[29] K. He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *ArXiv e-prints* (Feb. 2015). arXiv: 1502.01852 [cs.CV].

[30] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural Comput.* 18.7 (July 2006), pp. 1527–1554. DOI: 10.1162/neco.2006.18.7.1527.

[31] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[32] Tim Hunter. *Deep Learning with Apache Spark and TensorFlow*. Databricks. 2016. URL: https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html.

[33] F. N. Iandola et al. “FireCaffe: near-linear acceleration of deep neural network training on compute clusters”. In: *ArXiv e-prints* (2015). arXiv: 1511.00175 [cs.CV].

[34] A. G. Ivakhnenko. “Polynomial Theory of Complex Systems”. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-1.4 (Oct. 1971), pp. 364–378. DOI: 10.1109/TSMC.1971.4308320.

[35] J. Johnson, A. Alahi, and L. Fei-Fei. “Perceptual Losses for Real-Time Style Transfer and Super-Resolution”. In: *ArXiv e-prints* (2016). arXiv: 1603.08155 [cs.CV].

[36] Norm Jouppi. *Google supercharges machine learning tasks with TPU custom chip*. Google Cloud Platform Blog. 2016. URL: https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[37] Ivan Krasin and Tom Duerig. *Introducing the Open Images Dataset*. 2016. URL: https://research.googleblog.com/2016/09/introducing-open-images-dataset.html.

[38] A. Krizhevsky. “One weird trick for parallelizing convolutional neural networks”. In: *ArXiv e-prints* (2014). arXiv: 1404.5997 [cs.DC].

[39] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[41] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (May 2015), pp. 436–444. DOI: 10.1038/nature14539.

[42] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Comput.* 1.4 (Dec. 1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[43] Mu Li et al. “Scaling Distributed Machine Learning with the Parameter Server”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[44] T.-Y. Lin et al. “Microsoft COCO: Common Objects in Context”. In: *ArXiv e-prints* (May 2014). arXiv: 1405.0312 [cs.CV].

[45] Y. Lin et al. “Large-scale image classification: Fast feature extraction and SVM training”. In: *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. 2011, pp. 1689–1696. DOI: 10.1109/CVPR.2011.5995477.

[46] H. Ma, F. Mao, and G. W. Taylor. “Theano-MPI: a Theano-based Distributed Training Framework”. In: *ArXiv e-prints* (2016). arXiv: 1605.08325 [cs.LG].

[47] P. Moritz et al. “SparkNet: Training Deep Networks in Spark”. In: *ArXiv e-prints* (2015). arXiv: 1511.06051 [stat.ML].

[48] Y. Nikulin and R. Novak. “Exploring the Neural Algorithm of Artistic Style”. In: *ArXiv e-prints* (2016). arXiv: 1602.07188 [cs.CV].

[49] NVIDIA. *CUDA Parallel Computing Platform*. 2016. URL: http://www.nvidia.com/object/cuda_home_new.html.

[50] NVIDIA. *NVIDIA CUDNN, GPU Accelerated Deep Learning*. 2016. URL: https://developer.nvidia.com/cudnn.

[51] T. Paine et al. “GPU Asynchronous Stochastic Gradient Descent to Speed Up Neural Network Training”. In: *ArXiv e-prints* (2013). arXiv: 1312.6186 [cs.CV].

[52] Adam Roberts et al. *Magenta*. 2016. URL: https://magenta.tensorflow.org/about/.

[53] M. Ruder, A. Dosovitskiy, and T. Brox. “Artistic style transfer for videos”. In: *ArXiv e-prints* (2016). arXiv: 1604.08610 [cs.CV].

[54] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1”. In: Cambridge, MA, USA: MIT Press, 1986. Chap. Learning Internal Representations by Error Propagation, pp. 318–362. URL: http://dl.acm.org/citation.cfm?id=104279.104293.

[55] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y. arXiv: 1409.0575 [cs.CV].

[56] J. Sanchez and F. Perronnin. “High-dimensional signature compression for large-scale image classification”. In: *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. June 2011, pp. 1665–1672. DOI: 10.1109/CVPR.2011.5995504.

[57] Jürgen Schmidhuber. “Deep Learning in Neural Networks: An Overview”. In: *Neural Networks* 61 (2015), pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003. arXiv: 1404.7828 [cs.NE].

[58] P. Sermanet et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks”. In: *ArXiv e-prints* (2013). arXiv: 1312.6229 [cs.CV].

[59] Nathan Silberman and Sergio Guadarrama. *TF-Slim: A high level library to define complex models in TensorFlow*. 2016. URL: https://research.googleblog.com/2016/08/tf-slim-high-level-library-to-define.html.

[60] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (Jan. 2016). Article, pp. 484–489. URL: http://dx.doi.org/10.1038/nature16961.

[61] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *ArXiv e-prints* (2014). arXiv: 1409.1556 [cs.CV].

[62] Christopher Smith, Christopher Nguyen, and Ushnish De. *Distributed TensorFlow: Scaling Google’s Deep Learning Library on Spark*. Arimo. 2016. URL: https://arimo.com/machine-learning/deep-learning/2016/arimo-distributed-tensorflow-on-spark.

[63] Richard Socher et al. “Recursive deep models for semantic compositionality over a sentiment treebank”. In: *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*. Vol. 1631. 2013, p. 1642. eprint: http://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf.

[64] I. Sutskever, O. Vinyals, and Q. V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *ArXiv e-prints* (2014). arXiv: 1409.3215 [cs.CL].

[65] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: http://arxiv.org/abs/1409.4842.

[66] Zak Taylor. *Distributed learning in Torch*. Twitter. 2016. URL: https://blog.twitter.com/2016/distributed-learning-in-torch.

[67] BSC Support Team. *Greasy User Guide*. 2012. URL: https://github.com/jonarbo/GREASY/blob/master/doc/greasy_userguide.pdf.

[68] Seiya Tokui et al. "Tutorial: Deep Learning Implementations and Frameworks". In: 2016. URL: http://www.slideshare.net/beam2d/differences-of-deep-learning-frameworks.

[69] D. Ulyanov et al. "Texture Networks: Feed-forward Synthesis of Textures and Stylized Images". In: *ArXiv e-prints* (2016). arXiv: 1603.03417 [cs.CV].

[70] Koen EA Van de Sande et al. "Segmentation as selective search for object recognition". In: *2011 International Conference on Computer Vision*. IEEE. 2011, pp. 1879–1886.

[71] O. Vinyals et al. "Show and Tell: A Neural Image Caption Generator". In: *ArXiv e-prints* (2014). arXiv: 1411.4555 [cs.CV].

[72] M. D Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *ArXiv e-prints* (2013). arXiv: 1311.2901 [cs.CV].