Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Barcelona Supercomputing Center (BSC-CNS)

Scalability Study of Deep Learning Algorithms in High Performance Computer Infrastructures

Francesc Sastre Cabot

Master in Informatics Engineering

Advisor: Jordi Torres i Viñals

Universitat Politècnica de Catalunya (UPC)

Department of Computer Architecture (DAC)

Co-Advisor: Maurici Yagües Gomà

Barcelona Supercomputing Center (BSC)

Computer Sciences - Autonomic Systems and e-Business Platforms

April 28, 2017

Abstract

Deep learning algorithms base their success on building high learning capacity models with millions of parameters that are tuned in a data-driven fashion. These models are trained by processing millions of examples, so that the development of more accurate algorithms is usually limited by the throughput of the computing devices on which they are trained.

This project show how the training of a state-of-the-art neural network for computer vision can be parallelized on a distributed GPU cluster, Minotauro GPU cluster from Barcelona Supercomputing Center with the TensorFlow framework.

In this project, two approaches for distributed training are used, the synchronous and the mixed-asynchronous. The effect of distributing the training process is addressed from two different points of view. First, the scalability of the task and its performance in the distributed setting are analyzed. Second, the impact of distributed training methods on the final accuracy of the models is studied.

The results show an improvement for both focused areas. On one hand, the experiments show promising results in order to train a neural network faster. The training time is decreased from 106 hours to 16 hours in mixed-asynchronous and 12 hours in synchronous. On the other hand we can observe how increasing the numbers of GPUs in one node rises the throughput, images per second, in a near-linear way. Moreover the accuracy can be maintained, like the one node training, in the synchronous methods.

Acknowledgements

I would like to thank my advisor, Jordi Torres, for the great opportunity he has offered me and for introducing me to the exciting High-Performance Computer and Deep Learning worlds, as well his encouraging and enthusiasm. Also, thanks to my co-advisor, Maurici Yagües, for his help, advice, and patience. I also would want to highlight the great job done by the BSC Support Team. Thanks to all because without you this project would not have been possible.

Thanks to all my colleagues at the Autonomic Systems and e-Business Platforms group (BSC) for making an enjoyable working environment, especially to Víctor Campos and Míriam Bellver for all things that I have learned from them.

I also would render thanks to Alberto, my friend during the degree, the master, and now at work which without him, none of this would have started.

Last but not least, my special thanks to my family and friends, for their support and comprehension when I have needed it.

Contents

1. Problem Statement

1.1 Goal 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) continuing the work started in the thesis of Maurici Yagües [104]. The contents of this thesis may be valuable for the Barcelona Supercomputing Center in order to define new strategies, new research fields and to prove that Deep Learning problems can take advantage of high performance computers in order to obtain better results in less time.

This document will be focused in three areas: (1) Choose the best configuration to distribute the Deep Learning algorithms, (2) study the scalability in terms of throughput and (3) the scalability from the learning standpoint.

The introduction of this thesis will introduce the ideas of Machine Learning and Deep Learning, explore different applications of Neural Networks, and show a brief detail about the most popular Deep Learning frameworks. Also, it will explain the concepts of distribution in Deep Learning , the infrastructure to be used, and the Neural Network use case of this thesis.

1.2 Machine learning

Machine learning (ML) is an Artificial Intelligence subfield that provides computers with the ability to learn without being explicitly programmed. A Machine Learning algorithm can change its output depending of the data that it has learned. This kind of algorithms learn data in order to find patterns and build a model to predict and classify items. Machine learning performs well in some scenarios, although there are some cases where it cannot create an accurate model due to the large amount of variations that the problem has.

For example, a classifier of animal pictures. Pictures of same animal can be much different, different positions, different daylight, different atmospheric conditions, obstructions. A traditional Machine learning algorithm cannot

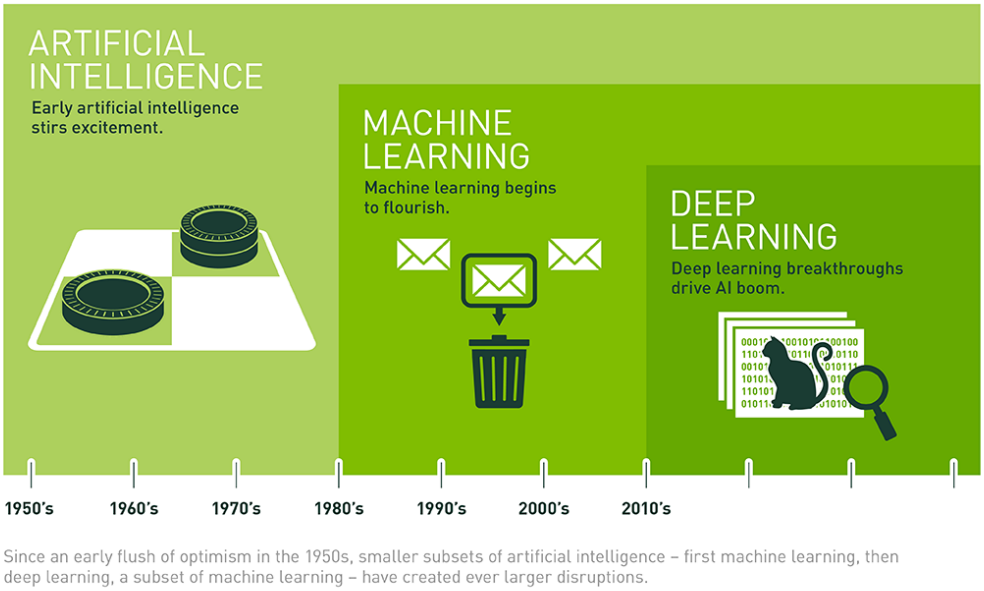


Figure 1.1: IA evolution. (Source: Nvidia Blog [26])

achieve this goal whereas there is a kind of ML algorithms, Deep Learning, that can perform well in this kind of problems.

Figure 1.1 shows a graphic explanation about the Artificial Intelligence and the disciplines contained in it.

1.3 Deep Learning

Deep Learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep Learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a model should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech [53].

1.3.1 Trends in Deep Learning for Computer Vision

Recently, the fields covered by Deep Learning have increased exponentially. Here, several trends will be explained in order to offer a global vision about Deep Learning and how far it reaches. These Deep Learning applications work in a very similar way as the application of this thesis and they could be scaled up as the model of this thesis.

ImageNet classification

ImageNet[31] is an image database organized according to the WordNet[1] hierarchy. ImageNet aims to populate the majority of the 80,000 synsets of WordNet with an average of 500-1000 clean and full resolution images. This will result in tens of millions of annotated images organized by the semantic hierarchy of WordNet. Figure 1.2 shows an example of Imagenet classification.

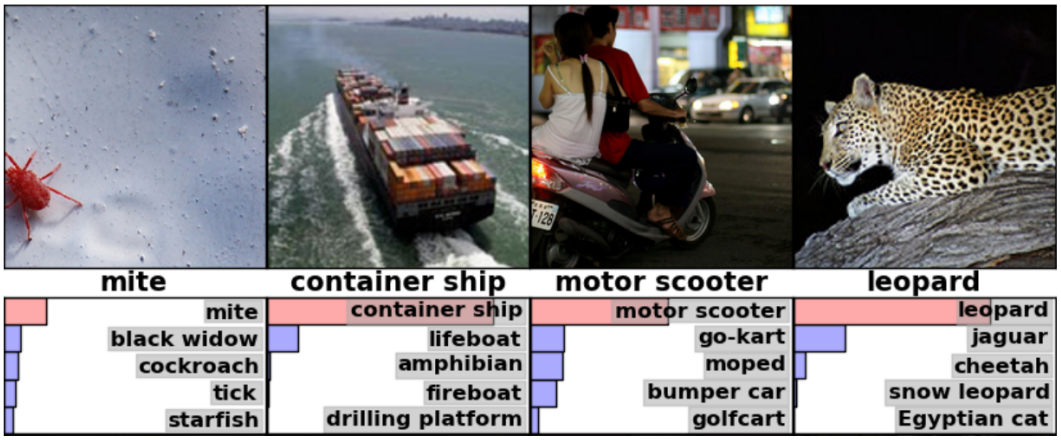


Figure 1.2: ImageNet classification. (Source: Alex Krizhevsky et al. [51])

There are several publications about ImageNet classification. AlexNet [51], GoogleNet [91], VGG-Net [86] and Resnet [38] are some of the most used deep models to do ImageNet classification. Most Deep Learning frameworks come with this pre-trained models and they can be used to create new models saving much training time.

Object Detection

A simple image classification problem assumes that there is only one target to classify in the image. Detection problems are about to detect different objects in a image and then classify them individually. There are several methods based on deep neural nets to realize the classification.

[1] WordNet website (https://wordnet.princeton.edu/)

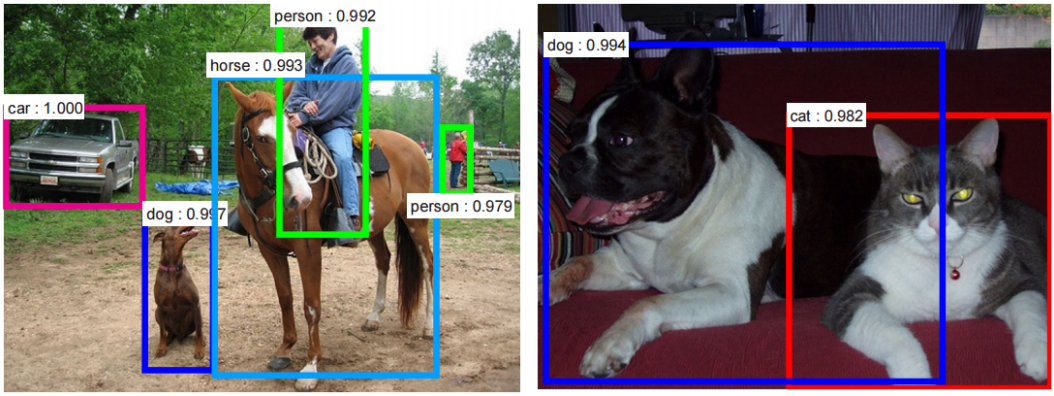


Figure 1.3: Object detection and classification. Images with regions and classifications. (Source: Shaoqing Ren et al. [79])

Some of these methods use a two stage procedure to detect and classify as R-CNN [36], Fast R-CNN [35], Faster R-CNN [79] or R-FCN [28]. In the two stage methods, the image goes through a network in order extract features and at some point (layer), the detector finds the regions on the image where there are objects. These regions are cut in the image and sent to classify using a neural net. The network can be another one, the same used for the object detection, or even the classification can continue at the same layer of the detection. Figure 1.3 shows examples of object detection.

On the other hand there are methods that can do all in one stage, they can detect and classify going through the network once. Some examples are YOLO [78] or SSD [56].

An important point for object detection is performance for real-time detection, for mobile or embedded applications. Jonathan Huang et al. in "Speed/accuracy trade-offs for modern convolutional object detectors" [40] compares the different object detectors using different neural nets in terms of speed, memory or accuracy.

Edge Detection

The edge detection problem with deep neural networks is strongly related to the object detection problem. Edge detection is a way for segmentation, image classification and object tracking. Saining Xie et al. [102], Gedas Bertasius et al. with DeepEdge [9] and Wei Shen et al. with DeepContour [83] try to address the problem with different strategies. Figure 1.4 shows an example of edge detection using deep neural networks.

Figure 1.4: Edge detection using DeepContour. (Source: Wei Shen et al. [83])

Low-Level Vision

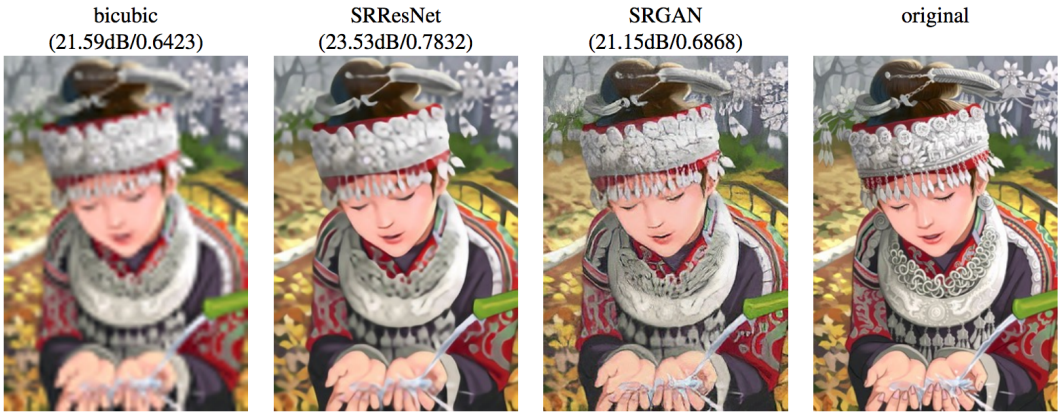
There are different techniques to face the problem of image reconstruction. Image reconstruction with Deep Learning, image super-resolution, can recover a pixelated image in a sharp well defined image and can upscale it, Figure 1.5a show an example. Several publications and authors propose different manners to address this problem as Chao Dong et al. [33] or Christian Ledig et al. [55].

Other applications of low-level vision are image colorization [110], like Figure 1.5b, blur removal [90, 81] or image quality recovering after compression [32].

Image Captioning

Automatically describing the content of an image is a fundamental problem in Artificial Intelligence that connects Computer Vision and Natural Language Processing. Deep Learning models can generate image descriptions with natural language, see Figure 1.6. This is useful to automate descriptions of images for disabled people or to improve image searching using words.

Most of this models are combinations of deep vision neural nets for feature extraction and recurrent neural nets for natural language generation [97, 59]. Others models like [47] use object detection models combined with recurrent neural networks.



(a) Super Resolution application. (Source: Christian Ledig et al. [55])

(b) Image Colorization. (Source: Richard Zhang et al. [110])

Figure 1.5: Low-level vision applications.

Figure 1.6: Images with captions. (Source: Andrej Karpathy et al. [47])

Image Generation

A generative model is one that captures the observable data distribution. The objective of deep neural generative models is to disentangle different factors of variation in data and be able to generate new or similar. Figure 1.7 is an example of generated images.

Most of these type of models uses generative adversarial networks [37, 89, 29]. They have two agents, the generator and the discriminator. The generator generates new images and sends them to the discriminator. The discriminator analyzes the image and classifies it between real or generated. The goal of the discriminator is to reduce its own classification errors and the achievement of the generator is maximize the failures of the discriminator. There are models combining convolutional networks with generative adversarial networks [76], or others using recurrent networks [69].

Figure 1.7: Generated images of bedrooms. (Source: Alec Radford et al. [76])

1.3.2 Deep Learning frameworks

Recent years, as a result of the increase of the popularity of Deep Learning, many frameworks have surged in order to ease the task of create and train models. Frameworks use different programing languages, strategies to train the models or compute the data and different characteristics as distribution or GPU acceleration support. Most of these frameworks are open sourced and their popularity can be measured using their number of followers, contributors and updates as seen in Figure 1.8, from François Chollet Twitter[2]. Figure

[2]François Chollet Tweet (https://twitter.com/fchollet/status/852194634470223873)

1.9 shows the temporal evolution of the popularity of the more active Deep Learning frameworks.

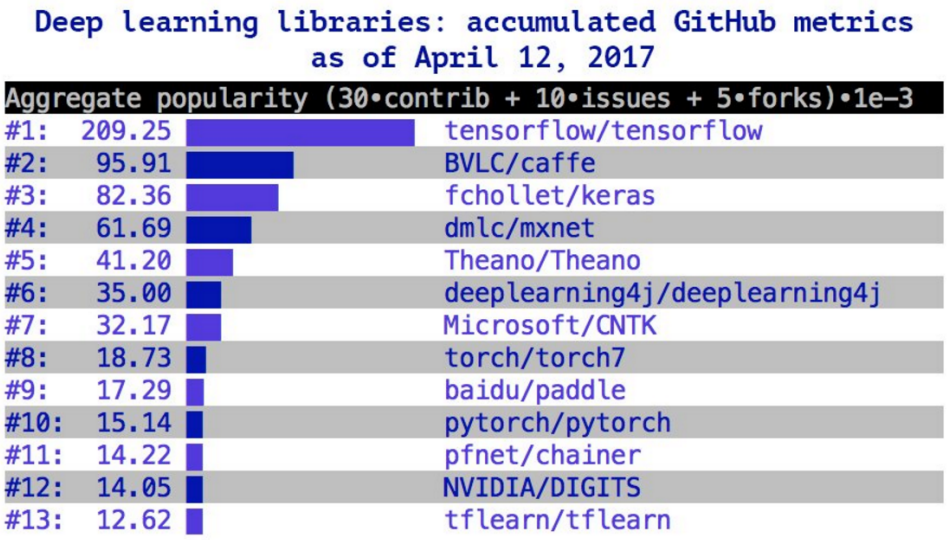


Figure 1.8: Aggregate framework popularity in 04/2017

Here is a brief description of more active and popular frameworks for Deep Learning with their characteristics:

Torch and PyTorch

Torch is a flexible Machine Learning framework with LuaJIT interface launched in 2012. Its core is based in C and it has OpenMP and CUDA implementations for high performance requirements [25]. It has modules for distributed training[92].

Recently, PyTorch[3] was launched. It is based in Python and is more focused on Deep Learning. It maintains the functions of Torch and adds new tools about Deep Learning such as networks, data loading, optimizers and training.

Theano

Theano started as a CPU and GPU math compiler in Python similar as Numpy but with faster evaluation and GPU support. It has a Python core [6] and it generates C++ and CUDA code for running [8]. It implements experimental support for multi-GPU model parallelism but for data parallelism it needs external frameworks.

[3]PyTorch website (http://pytorch.org)

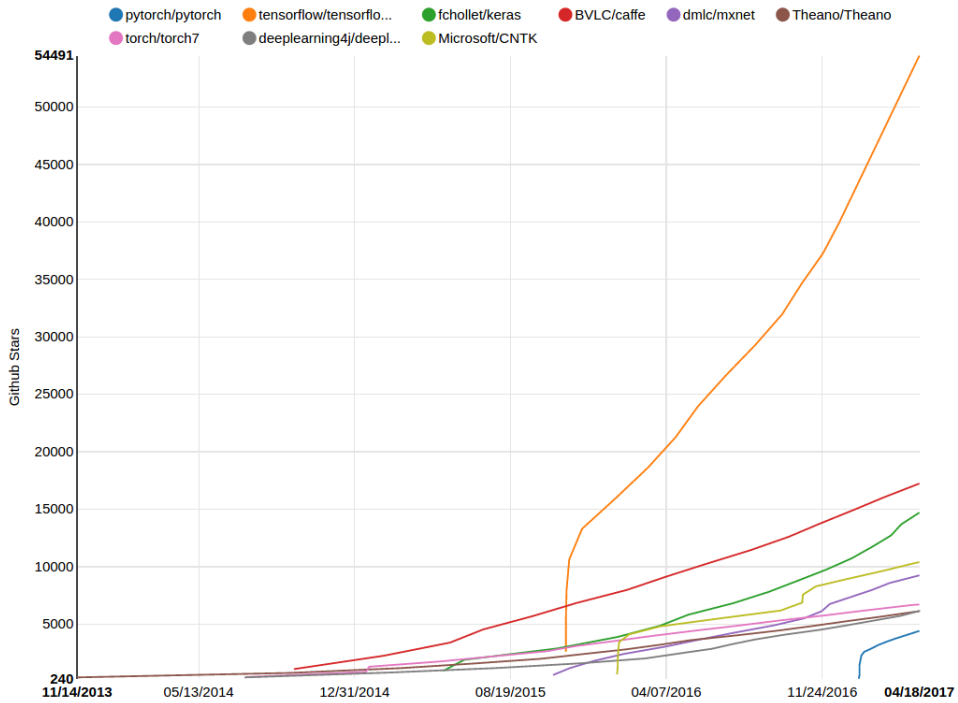


Figure 1.9: Frameworks popularity evolution in GitHub

Caffe

Caffe is a Deep Learning framework based in C++ with Python and MATLAB bindings. Caffe is maintained and developed by the Berkeley Vision and Learning Center (BVLC). Caffe allows experimentation and seamless switching among platforms for ease of development and deployment from prototyping machines to cloud environments [41]. It has not a distributed implentation and needs external tools like CaffeOnSpark[4] to run in a cluster.

Nevertheless, Caffe2[5] has been recently released directly by Facebook Research. Caffe2 GitHub page describes it as "an experimental refactoring of Caffe that allows a more flexible way to organize computation". It includes multi-GPU and multi-machine distribution and it's more focused in production environments being able to work in different devices such mobiles or servers.

CNTK

Microsoft Computational Network Toolkit (CNTK)[109] or Cognitive Toolkit is a Deep Learning framework for Windows and Linux. Its core is based on C++/CUDA and it has APIs for C++, C# and Python. It has GPU, multi-

[4]CaffeOnSpark repository (https://github.com/yahoo/CaffeOnSpark)
[5]Caffe2 Website (https://caffe2.ai/)

GPU and distributed training support. CNTK uses graph computation and allows to easily realize and combine popular model types such as feed-forward DNNs, convolutional nets (CNNs), and recurrent networks (RNNs/LSTMs). It implements stochastic gradient descent (SGD, error backpropagation) and learning with automatic differentiation. CNTK has been available under an open-source license since April 2015.

Microsoft offers CNTK as a service in the Azure Platform[6] with GPU instances.

MXNET

MXNET is a multi-language framework for Machine Learning. It has a C++ core and Python, R, C++ and Julia bindings. It is compatible with CPU and GPUs and natively supports GPU parallelism and distributed execution. MXNET supports declarative and imperative execution and permits the user to choose which want to use. MXNet is computation and memory efficient and runs on various heterogeneous systems, ranging from mobile devices to distributed GPU clusters [22].

deeplearning4j

Deeplearning4j[7] is an open-source, distributed deep-learning library written for Java and Scala. Integrated with Hadoop and Spark, DL4J is designed to be used in business environments on distributed GPUs and CPUs. Its core is written in C/C++/CUDA and has a Python API using Keras. It uses Spark/Hadoop for the distribution, it is compatible with multi-GPU and it can run in Amazon AWS Elastic Map Reduce[8].

Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of either TensorFlow[1, 2] or Theano[8]. The user can change the backend (TF/Theano) without changing the code. The Keras code can be mixed with the code of the backend. Keras uses the different characteristics offered by the available backends and it is compatible with multi-GPU and distributed run.

[6] Azure Deep Learning toolkit for Data Science VM website (https://azuremarketplace.microsoft.com/en-us/marketplace/apps/microsoft-ads.dsvm-deep-learning)

[7] deeplearning4j website (https://deeplearning4j.org/)

[8] Amazon AWS Elastic Map Reduce website (https://aws.amazon.com/emr/)

1.4 Distribution

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 [71]. Also, it requires less communication between nodes, as it benefits from high amount of computations per weight [50]. On the other hand, the model has to entirely fit on each node [27], 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 [24]. It can be used with larger models as hardware constraints per node are no more a limitation, but is highly vulnerable to worker failures.

1.4.1 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 CPUs / GPUs or nodes
- **Scalability**: efficiency as a function of an increasing number of CPUs / GPUs or nodes

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.

1.5 TensorFlow

TensorFlow is a Machine Learning system that operates at large scale and in heterogeneous environments. TensorFlow uses dataflow graphs to represent computation, shared state, and the operations that mutate that state. It maps the nodes of a dataflow graph across many machines in a cluster, and within a machine across multiple computational devices, including multicore CPUs, general purpose GPUs, and custom designed ASICs known as Tensor Processing Units (TPUs [4]). This architecture gives flexibility to the application developer: whereas in previous "parameter server" designs the management of shared state is built into the system, TensorFlow enables developers to experiment with novel optimizations and training algorithms.

The system is flexible and can be used to express a wide variety of algorithms, including training and inference algorithms for deep neural network models, and it has been used for conducting research and for deploying Machine Learning systems into production [1, 2].

TensorFlow has Python, Java, Go and C++ bindings and it supports distributed traning using model parallelism and data parallelism. This framework is maintained by Google and it has a great support from the community, see Figures 1.8 and 1.9. Tensorflow has been chosen for this project due to the flexibility that it can offer and the awesome support showed on recent years. Lots of projects and companies have started to migrate from another frameworks such Caffe or Theano to Tensorflow.

TensorFlow also includes Tensorboard, see Figure 1.10. Tensorboard is used to visualize the TensorFlow graph, plot quantitative metrics about the

execution of the graph like accuracy or loss, and show additional data like images, videos or audios.

The first stable release of Tensorflow was released in February 2017. This new version starts a long term support with the promise of maintaining the compatibility among the different minor versions of this major version. TensorFlow will include the Keras library, see Section 1.3.2, inside the core of the framework. With Keras, Tensorflow will decrease its learning curve and attract more developers to the platform.

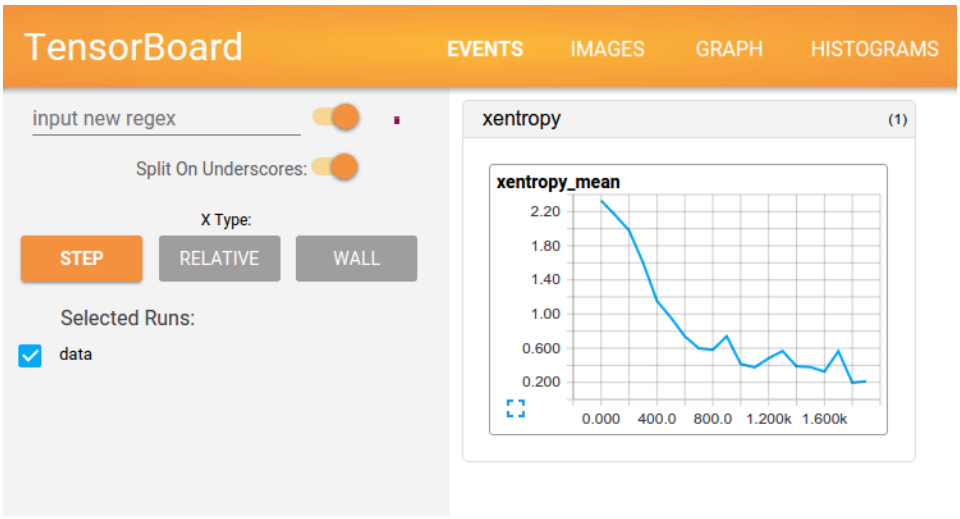


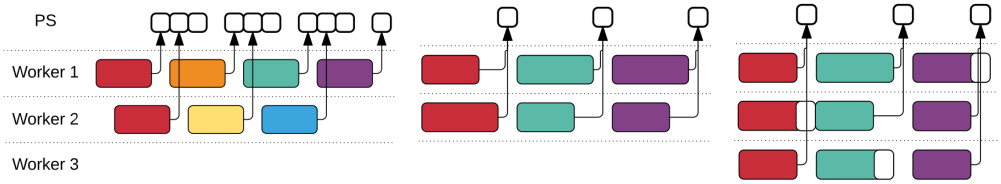
Figure 1.10: Tensorboard example. (Source: tensorflow.org)

1.5.1 Strategies for data parallelism

There are two types of processes involved in distributed TensorFlow. First, the worker processes, in charge of the computation using GPUs, CPUs and/or TPUs. On the other hand there is the parameter server (PS), this kind of process receives the model updates from the workers, processes them and sends a new model to the workers.

There are different strategies in order to do data parallelism in TensorFlow. The distinctness is focused in the way to merge the gradients between different workers. Chen et al. [19, 20] compare the strategies using a very large number of workers without focusing in efficiency terms.

Synchronous mode: In this case, the PS waits until all worker nodes have computed the gradients with respect to their data batches. Once the gradients are received by the PS, they are applied to the current weights and the updated model is sent back to all the worker nodes. This method is as fast as the slowest node, as no updates are performed until all worker nodes finish the computation, and may suffer from unbalanced network speeds when



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

Figure 1.11: Strategies for data parallelism. (Source: Abadi et al. [2])

the cluster is shared with other users. However, faster convergence is achieved as more accurate gradient estimations are obtained. Authors in [20] present an alternative strategy for alleviating the slowest worker update problem by using backup workers, as Figure 1.11c.

Asynchronous mode: every time the PS receives the gradients from a worker, the model parameters are updated. Despite delivering an enhanced throughput when compared to its synchronous counterpart, every worker may be operating on a slightly different version of the model, thus providing poorer gradient estimations. As a result, more iterations are required until convergence due to the stale gradient updates. Increasing the number of workers may result in a throughput bottleneck by the communication with the PS, in which case more PS need to be added.

Mixed mode: mixed mode appears as a trade-off between adequate batch size and throughput by performing asynchronous updates on the model parameters, but using synchronously averaged gradients coming from subgroups of workers. Larger learning rates can be used thanks to the increased batch size, leading to a faster convergence, while reaching throughput rates close to those in the asynchronous mode. This strategy also reduces communication as compared to the pure asynchronous mode.

1.6 Infrastructure

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 re-

lated. 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" [12].

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

1.6.1 Minotauro overview

Minotauro is a heterogeneous cluster with 2 configurations [13]:

- 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, see Figure 1.12.

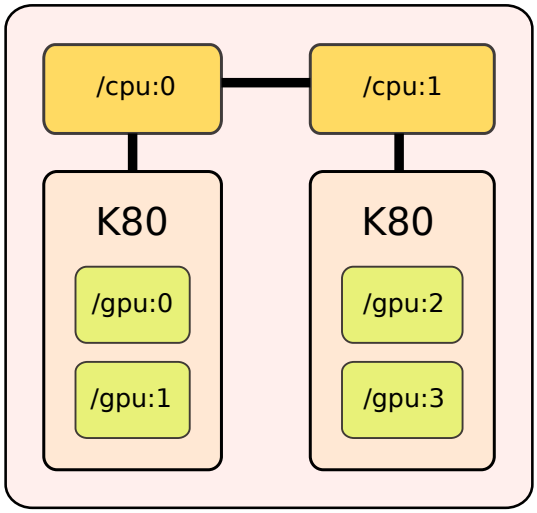


Figure 1.12: Schema of the hardware in a Minotauro node. (Source: Maurici Yagües [104])

1.7 Neural Network use case

Affective Computing [74] has recently garnered much research attention. Machines that are able to understand and convey subjectivity and affect would lead to a better human-computer interaction that is key in some fields such as robotics or medicine. Despite the success in some constrained environments such as emotional understanding of facial expressions [60], automated affect understanding in unconstrained domains remains an open challenge which is still far from other tasks where machines are approaching or have even surpassed human performance. In this work, we will focus on the detection of Adjective Noun Pairs (ANPs) using a large-scale image dataset collected from Flickr [46]. These mid-level representations, which are a rising approach for overcoming the *affective gap* between low level image features and high level affect semantics, can then be used to train accurate models for visual sentiment analysis or visual emotion prediction. Figure 1.13 shows a brief detail how the proposed structure works.

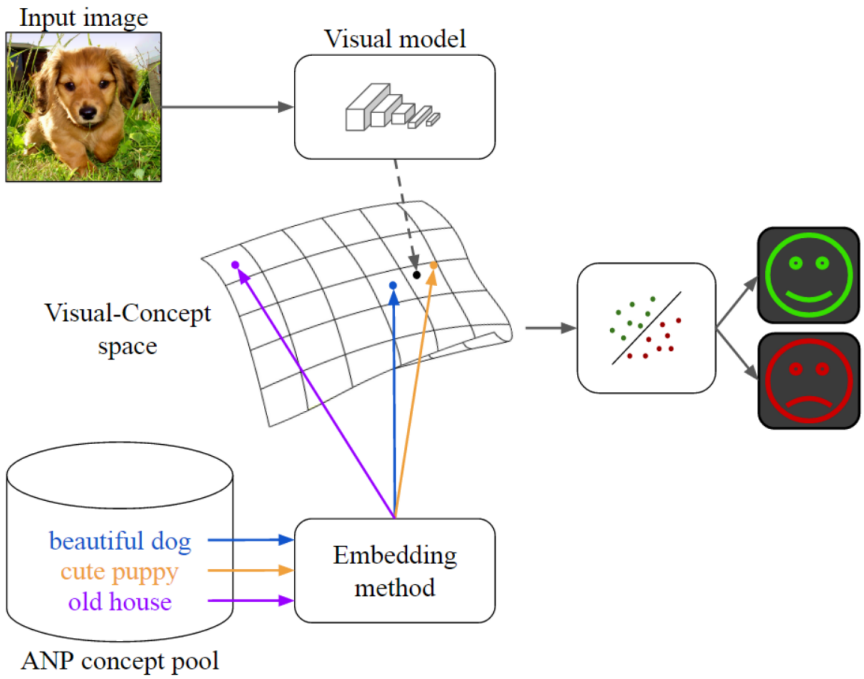


Figure 1.13: First, an output embedding is generated by passing all adjectives and nouns in the Adjective Noun Pair (ANP) pool through a language model and combining their outputs. Then, a visual model learns to map the image to the output space while leveraging the fine-grained interclass relationships through the embedded representations. Finally, the learned visual model is used as a mid-level representation for affect recognition. (Source: Víctor Campos et al. [5])

2. Testing Methodology

2.1 Dataset

Adjective Noun Pairs (ANPs) are powerful mid-level representations [11] that can be used for affect related tasks such as visual sentiment analysis or emotion recognition. A large-scale ANP ontology for 12 different languages, namely Multilingual Visual Sentiment Ontology (MVSO), was collected by Jou et al. [46] following a model derived from psychology studies, *Plutchick's Wheel of Emotions* [75]. The noun component in an ANP can be understood to ground the visual appearance of the entity, whereas the adjective polarizes the content towards a positive or negative sentiment, or emotion [46]. These properties try to bridge the *affective gap* between low level image features and high level affective semantics, which goes far beyond recognizing the main object in an image. Whereas a traditional object classification algorithm may recognize a *baby* in an image, a finer-grained classification such as *happy baby* or *crying baby* is usually needed to fully understand the affective content conveyed in the image. Capturing the sophisticated differences between ANPs poses a challenging task that requires from high capacity models and large-scale annotated datasets.

The experiments with the subset of the English partition of MVSO, the tag-restricted subset, which contains over 1.2M samples covering 1,200 different ANPs. Since images in MVSO were downloaded from Flickr and automatically annotated using their metadata, such annotations have to be considered as *weak labels*, i.e. some labels may not match the real content of the images. The tag-pool subset contains those samples for which the annotation was obtained from the tags in Flickr instead of other metadata, so that annotations are more likely to match the real ground truth.

2.2 Network

Since the first successful application of Convolutional Neural Nets (CNNs) to large-scale visual recognition, the design of improved architectures for im-

proved classification performance has focused on increasing the depth, i.e. the number of layers, while keeping or even reducing the number of trainable parameters. This trend can be seen when comparing the 8 layers in AlexNet [51], the first CNN-based method to win the Image Large Scale Visual Recognition Challenge (ILSVRC) in 2012, with the dozens, or even hundreds, of layers in Residual Nets (ResNets) [38]. Despite the huge increase in the overall depth, a ResNet with 50 layers has roughly half the parameters in AlexNet. However, the impact of an increased depth is more notorious in the memory footprint of deeper architectures, which store more intermediate results coming from the output of each single layer, thus benefiting from multi-GPU setups that allow the use of larger batch sizes[1].

The ResNet50 CNN [38] is used on the experiments, an architecture with 50 layers that maps a $224 \times 224 \times 3$ input image to a 1,200-dimensional vector representing a probability distribution over the ANP classes in the dataset. Overall, the model contains over 25×10^6 single-precision floating-point parameters involved in over 4×10^9 floating-point operations that are tuned during training. It is important to notice that the more computationally demanding a CNN is, the larger the gains of a distributed training due to the amount of time spent doing parallel computations with respect to the added communication overhead.

Cross-entropy[2] between the output of the CNN and the ground truth, i.e. the real class distribution, is used as loss function together with an L2 regularization[3] term with a weight decay rate of 10^{-4}. The parameters in the model are tuned to minimize the former cost objective using batch gradient descent.

2.3 Experimental setup

The experiments are evaluated in a GPU cluster, where each node is equipped with 2 NVIDIA Kepler K80 dual GPU cards, 2 Intel Xeon E5-2630 8-core processors and 128GB of RAM. Inter-node communication is performed trough a 56Gb/s InfiniBand network. The CNN architectures and their training are implemented with TensorFlow[4], running on CUDA and using cuDNN primitives for improved performance. Since the training process needs to be

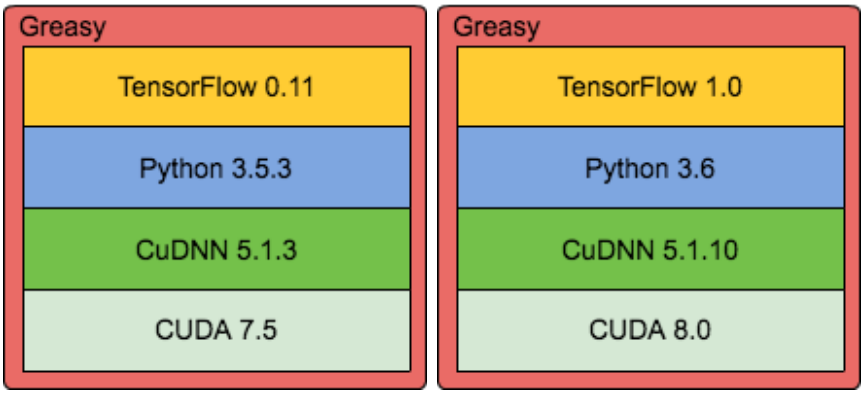
[1] The batch size is the number of elements that are processed in one iteration of training. The more deep is the network, the more memory is needed and the batch size has to decrease to fit in the GPU memory, but, if several GPUs are used the batch size can be smaller in each GPU but in total more elements are processed.

[2] CS231n Convolutional Neural Networks for Visual Recognition. Stanford Course (http://cs231n.github.io/linear-classify/)

[3] See footnote 2

[4] TensorFlow website (https://www.tensorflow.org/)

submitted through Slurm Workload Manager, task distribution and communication between nodes is achieved with Greasy [93].



(a) Software stack used in the asynchronous setup

(b) Software stack used in the synchronous setup

Figure 2.1: Software stack used in the experimental setup.

Unlike other works where each worker is defined as a single GPU [20, 43], all available GPUs are used in each node to define a single worker. Given the dual nature of the NVIDIA K80 cards, four model replicas are placed in each node. A mixed approach is used to synchronously average the gradients for all model replicas in the same node before communicating with the parameter server, which then performs asynchronous or synchronous model updates. This setup offers two main advantages: (1) communication overhead is reduced, as only a single collection of gradients needs to be exchanged through the network for each set of four model replicas, and (2) each worker has a larger effective batch size, providing better gradient estimations and allowing the use of larger learning rates for faster convergence. The software stack used in the asynchronous tests can be seen in Figure 2.1a.

Later a pure synchronous method is used. Inside a node everything is done as mixed approach, however the PS will wait until all the workers have sent their gradients. This will cause a lower images per second throughput but a better quality of learning because all workers will use updated gradients. Figure 2.1b shows the software stack for the synchronous tests.

There are two different stacks due to the large amount of needed time and resources to execute the experiments. TensorFlow 0.11 was enough to perform the asynchronous tests but the synchronous stack 2.1b needs TensorFlow 1.0 because it has the synchronous methods working. There was not enough time to perform all the tests with the most recent stack and the most recent compatible version of Tensorflow in that moment was used, besides, the cluster was shared with other departments and there were problems in order to obtain slot times to execute the experiments.

The loss function[5] is minimized using RMSProp [95] per-parameter adaptive learning rate as optimization method with a learning rate of 0.1, decay of 0.9 and $\epsilon = 1.0$. Each worker has an effective batch size of 128 samples, i.e. 32 images are processed at a time by each GPU. To prevent overfitting, data augmentation consisting in random crops and/or horizontal flips is asynchronously performed on CPU while previous batches are processed by the GPUs. The CNN weights are initialized using a model pre-trained[6] on ILSVRC [31], practice that has been proven beneficial even when training on large-scale datasets [105].

Previous publications on distributed training with TensorFlow [2, 20] tend to use different server configurations for worker and PS tasks. Given that a PS only stores and updates the model and there is no need for GPU computations, CPU-only servers are used for this task. On the other hand, most of the worker job involves matrix computations, for which servers equipped with GPUs are used. All nodes in the cluster used in these experiments are GPU equipped nodes, which means that placing PS and workers in different nodes would result in under-utilization of GPU resources. The impact of sharing resources between PS and workers as compared to the former configuration and whether this setup is suitable for a homogeneous cluster where all nodes are equipped with GPU cards will be studied.

2.4 Minotauro Deploy

2.4.1 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 [13].

MinoTauro comes with a software stack, this software is static and the users cannot 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 0.11[7]:** The first version of TensorFlow used in this project. This version has a lot of bug fixes, improvements and new abstraction

[5] The loss function calculates the difference between the input training example and its expected output.

[6] Pre-training a model is a habitual technique. A pre-trained model, is a model initialized with the weights resulted from the training with another dataset.

[7] GitHub release 0.11 https://github.com/tensorflow/tensorflow/releases/tag/v0.11.0rc0

layer, TF-Slim, that allows to do some operations more easily.

- **TensorFlow 1.0**[8]**:** The second release of TensorFlow used in this project. This version is the first major version of Tensorflow, it establishes a consistent API and ensures that the new changes will not break backwards compatibility unlike the 0.x versions. It provides new abstractions for distributed training and an improved synchronous training.
- **Python 3.5.2/3.6:** The Python version installed in the K80 nodes. It comes with all required packages by TensorFlow. The 0.11 version uses Python 3.5.2 and the 1.0 version uses Python 3.6.
- **CUDA 7.5/8.0:** 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 [67]. This is required to use GPUs with TensorFlow.
- **cuDNN 5.1.3/5.1.10:** 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 [23].

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 shown in Table 2.1.

`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

Table 2.1: Slurm job directives

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

[8]GitHub release 1.0 https://github.com/tensorflow/tensorflow/releases/tag/v1.0.0

```
#!/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
```

Figure 2.2: Example of a job file

```
Submitted batch job 87869
```

Figure 2.3: Output of submitting a job

are needed in order to feed the GPUs as fast as possible, so as to not have any bottlenecks on the device. Figure 2.2 shows an example of a job file.

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, as shown in Figure 2.3, to do tracking on the task.

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. The Figure 2.4 shows the queue status, we have been given all CPUs and GPUs in nodes 16, 19 and 25, during a maximum time of 15 minutes.

All the parameters and configurations for the MinoTauro queue system are available on the MinoTauro User's Guide [13].

2.4.2 PS and Worker allocation

As discussed before, in order to make use of all available resources, the parameter servers will be allocated in the same node as a worker. A script was

```
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]
```

Figure 2.4: Queue status

```
#params example:
1. python mt_cluster.py file_tasks.txt script.py num_ps num_gpus \
    mixed_async ps_alongside

#run example:
1. python mt_cluster.py file_tasks.txt "train_net.py --logdir=train" 1 4 \
    1 1
```

Figure 2.5: Parameter order and call example

developed to identify the assigned nodes for the Minotauro task and control where allocate the PS processes[80]. Table 2.2 explains the needed parameters to run the script. The number of PS depends on the `ps_alongisde` parameter, if it is activated, the number of PS can be the same as nodes, but if it is not activated, the number of PS can be the number of nodes less one.

Name	Description	Correct values
`file_tasks`	Name of the ouput greasy tasks file	-
`script`	Name of Tensorflow python file script, it can have parameters	-
`num_ps`	Number of parameter servers	1 to max_nodes
`num_gpus`	Number of GPUs to use	1 to 4
`mixed_async`	Use the mixed-async approach or launch an independent worker for each gpu	0 or 1
`ps_alongside`	Put the PS with the workers or use full nodes only for PS	0 or 1

Table 2.2: Configuration parameters to run the script to create a greasy file tasks to execute distributed Tensorflow

Figure 2.5 shows the parameter order and how to execute the script in order to obtain a task file as the figure 2.6. Each output line will be executed in a different node. An important point is the PS allocation, the script will avoid, if it is possible, the first node, this is due to prevent an overhead in the chief worker. The chief worker is always in charge to save the checkpoints and summaries and needs more CPU ussage.

The `nvidia-smi` program shows all available GPUs and the identification number, as seen in Figure 2.7, the nodes of Minotauro have 2 Nvidia Tesla K80 but they can work as 4 GPU cards. With the `CUDA_VISIBLE_DEVICES` directive, the number of GPUs used by Tensorflow can be controlled and the PS processes can not use the GPUs as shown in the second line of Figure 2.6.

```
1. CUDA_VISIBLE_DEVICES="0,1,2,3" python train_net.py --logdir=train \
    --ps_hosts=nvb2-ib0:2219 --worker_hosts=nvb1-ib0:2220,nvb2-ib0:2220 \
    --job_name=worker --task_index=0

2. CUDA_VISIBLE_DEVICES="" python train_net.py --logdir=train \
    --ps_hosts=nvb2-ib0:2219 --worker_hosts=nvb1-ib0:2220,nvb2-ib0:2220 \
    --job_name=ps --task_index=0 & CUDA_VISIBLE_DEVICES="0,1,2,3" \
    python train_net.py --logdir=train --ps_hosts=nvb2-ib0:2219 \
    --worker_hosts=nvb1-ib0:2220,nvb2-ib0:2220 --job_name=worker \
    --task_index=1
```

Figure 2.6: Content of ouput file with 2 available nodes (nvb1, nvb2)

```
+-----------------------------------------------------+
| NVIDIA-SMI 352.39     Driver Version: 352.39        |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           On   | 0000:04:00.0     Off |                    0 |
| N/A   38C    P0    59W / 149W |     22MiB / 11519MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K80           On   | 0000:05:00.0     Off |                    0 |
| N/A   31C    P0    72W / 149W |     22MiB / 11519MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K80           On   | 0000:85:00.0     Off |                    0 |
| N/A   31C    P0    58W / 149W |     22MiB / 11519MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K80           On   | 0000:86:00.0     Off |                    0 |
| N/A   43C    P0    74W / 149W |     22MiB / 11519MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Figure 2.7: Ouput of `nvidia-smi`

2.4.3 Distributed run with Greasy

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.

In order to run Greasy, it is mandatory to load the modules needed by the Greasy tasks and finally, run Greasy passing the task file as parameter:

```
module load K80 cuda/8.0 mkl/2017.1 CUDNN/5.1.10-cuda_8.0 intel-opencl/2016 python/3.6.0+_ML
/apps/GREASY/latest/bin/greasy file_task.txt
```

Table 2.3 shows how the tasks of file 2.6 should be allocated with 2 available nodes.

With a basic usage, Greasy executes each line of the task file in a different node, but it can be more flexible and restrictions and dependencies between tasks are available but they are not required in this project. All information is available in the Greasy User's Guide [93].

Minotauro K80 node nvb1	Minotauro K80 node nvb2
Worker process 0 (GPUs 0-3)	PS process 0 (No GPU) Worker process 1 (GPUs 0-3)

Table 2.3: Process allocation by Greasy using the task file 2.6

3. Results

3.1 Throughput and resource usage

3.1.1 Parameter server number

Available publications on distributed training with TensorFlow [2, 20] tend to use different server configurations for worker and parameter server (PS) tasks. Given the PS only stores and updates the model and there is no need for GPU computations, only CPU servers are used for this task. On the other hand, most of the worker job involves matrix computations, for which servers equipped with GPUs are used. The Minotauro configuration imposes some constraints, as the system only has GPU equipped nodes, which means that placing PS and workers in different nodes will result in under-utilization of GPU resources. We decided to study the impact of placing the PS in the same nodes as the workers, and restricting the resource usage in each task. Odd values of PS are used, because variables are sharded using a round-robin strategy. Since the weights of the layers take much of the model size, and layer weights and bias are initialized consecutively, we want to have them as much distributed as possible, hence the use of an odd value.

Figure 3.1 shows different configurations of worker and PS setups with their respective throughput. With these results, it seems that the cluster configuration can cope with workers and PS in the same node, as throughputs are quite similar when placing PS inside or outside the worker nodes. However, a more important task is to correctly tune the amount of PS needed so as to not have a network or server bottleneck. In addition, when placing PS with workers, they are placed on workers that do not need to run auxiliary tasks of the training, such as checkpoint saving or summary writing, in order to not overload the CPU.

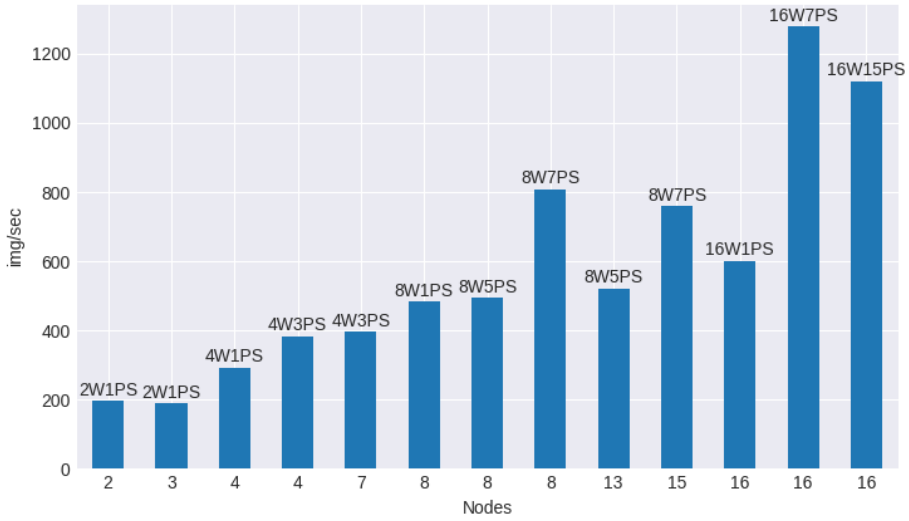


Figure 3.1: Throughput comparison between different distributed setups

Configuration	Throughput	Speedup	Efficiency
1 Node	124.18 img/sec	-	-
7 Nodes (4 Workers + 3 PS)	396.62 img/sec	3.19	0.46
4 Nodes (4 Workers + 3 PS)	374.73 img/sec	3.02	**0.76**
4 Nodes (4 Workers + 1 PS)	292.22 img/sec	2.35	0.58

Table 3.1: Comparison between different configurations when using 4 workers. Using dedicated nodes for the parameter servers slightly improves the throughput, but involves a much larger resource utilization. The efficiency is the relation between the speedup and the number of used nodes, it shows more clearly the grade of the resources exploitation.

3.1.2 Node parallelism

In order to determine the best strategy inside a node and make use of all available resources, there are two possibilities, (1) launch one process of Tensorflow managing the 4 GPU cards merging the results synchronously (GPU clones), see Figure 3.3a, transmitting only one gradient update per node or (2) launch one process Tensorflow per each GPU card and transmit four gradient update per node to the PS, see Figure 3.3b. Figure 3.2 shows how the TensorFlow operations are distributed among the available GPUs.

Figure 3.4 shows the speedup using those two strategies. Theoretically the asynchronous approach should have more throughput than the synchronous due to the asynchronous gradient updates but in this case the synchronous strategy evidences a better throughput and a more stable scaling.

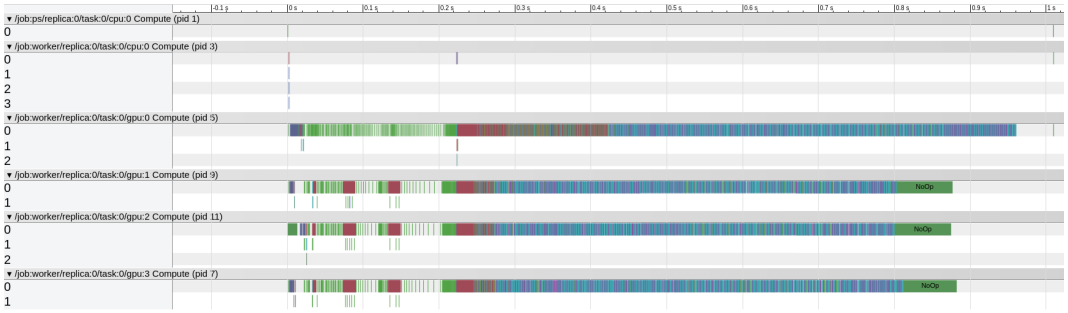
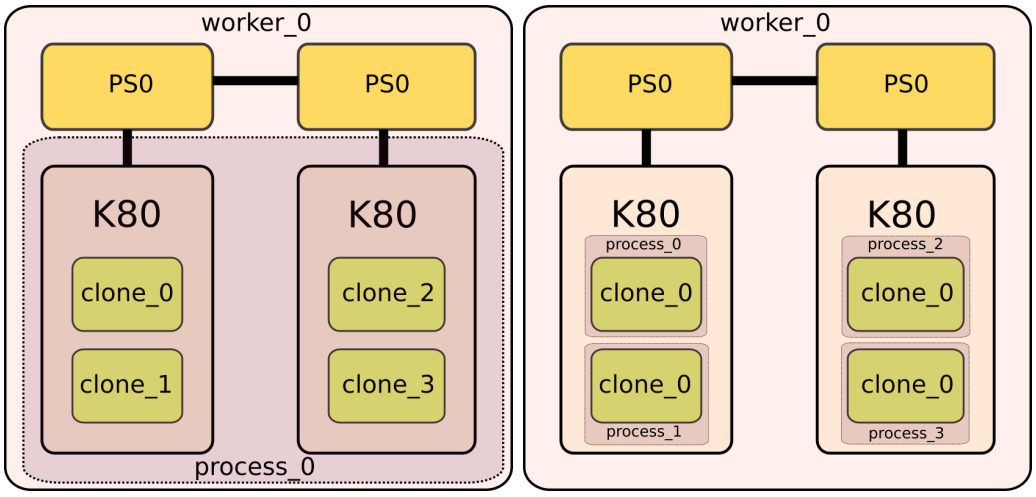


Figure 3.2: Tracing of GPU clones execution. (Source: Maurici Yagües [104])



(a) GPU allocation using clone strategy with one process.

(b) GPU allocation using four isolated processes.

Figure 3.3: Node parallelism strategies.

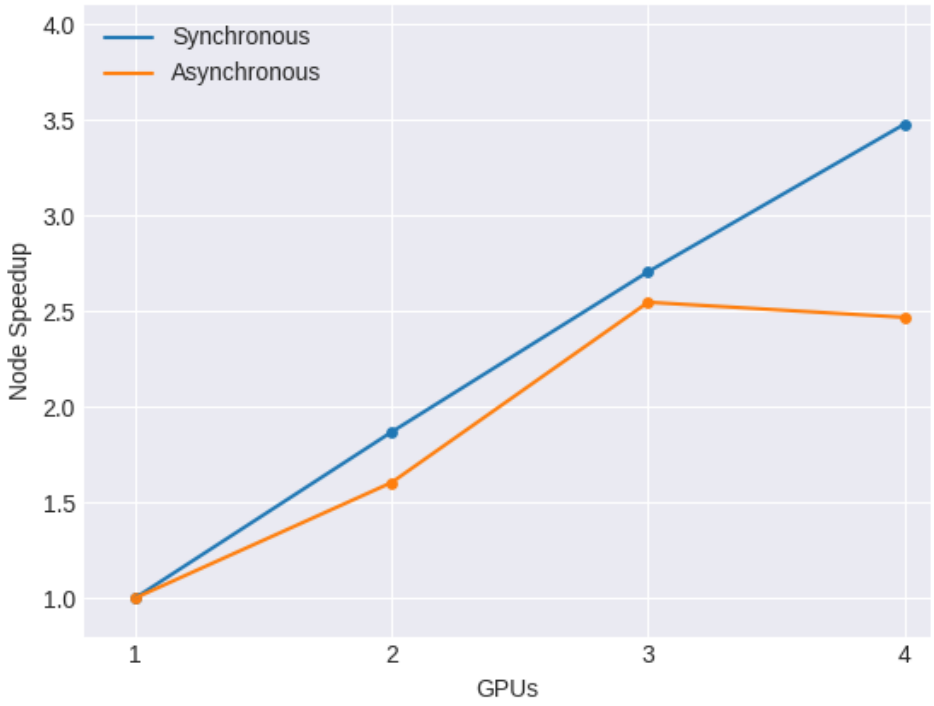


Figure 3.4: Parallelism speedup inside a node using different number of GPUs

3.1.3 Mixed-Asynchronous

In this case, the mixed-asynchronous approach is done. The gradients of each GPU are combined at the node, and are later sent to the parameter servers asynchronously.

For example, with eight nodes, we dispose of 32 GPUs. In a pure asynchronous strategy, the training process would need to send 32 model gradients through the network to the parameters servers, and then retrieve the updated model that can have around 400 MB. The whole process could need 12 GB for each iteration, which might cause overhead in the network and penalize the overall throughput. In the mixed approach, we try to reduce the network overhead reducing by four the data transfer doing only one sending of data per node.

Table 3.2 shows a brief detail about the throughput and the speedup among different setups. It is important to note the decrease in efficiency as the number of nodes grows, that is also related in diminishing return at training times. In Figure 3.5 there is a throughput comparison between all the tested configurations. We discern an improvement, although it is not linear, because there are a higher amount of resources to process and send through the network.

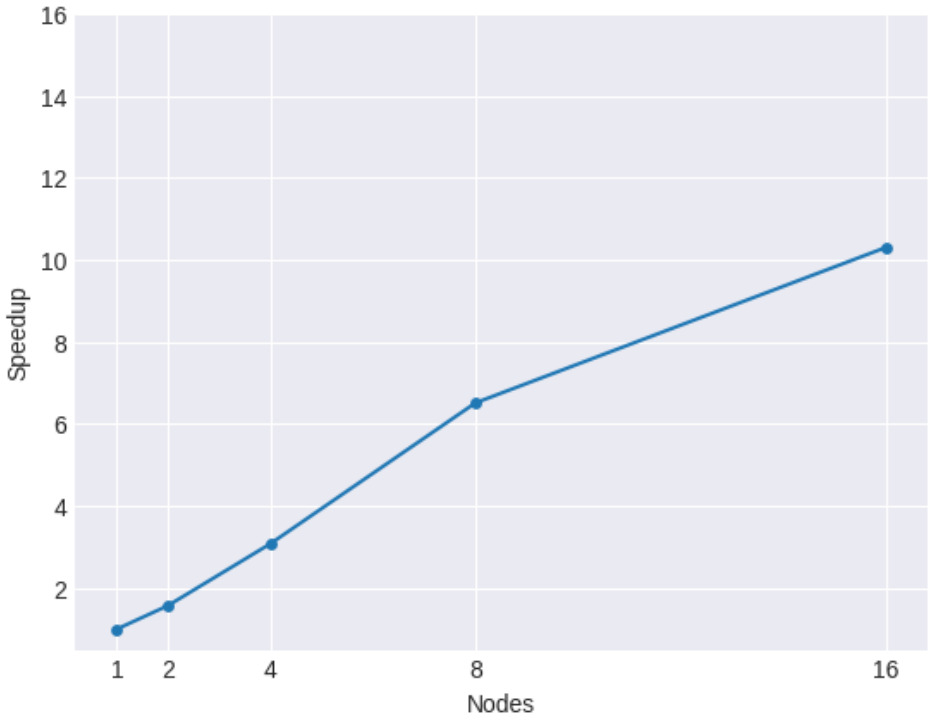


Figure 3.5: Distribution speedup using several nodes, with 4 GPUs, with the optimal asynchronous configuration. The baseline is one node with 4 GPUs.

3.1.4 Synchronous

As the same way in the mixed-asynchronous mode. The gradients of each GPU are combined at the node, and are later sent to the parameter servers, but in the synchronous approach, the parameters servers will wait until all the workers send their updates and then the parameter servers will perform the update.

There is a very similar speedup compared to the asynchronous mode, see Table 3.3. This may be surprising because theoretically the synchronous mode has to be slower due to the speed is defined by the slower worker. As seen in other publications [19, 20], synchronous and asynchronous have a similar throughput until 40 workers approximately. In these experiments, constrained by the limitations of the available hardware, the synchronous approach has more throughput than the asynchronous.

Besides, there are two different software stacks, see Figure 2.1. The synchronous tests were done with the newest stack, Tensorflow 1.0 and CUDA 8 instead of Tensorflow 0.11 and CUDA 7.5. The new versions may have influenced the final results due to the improvements done in these technologies.

In addition, like the asynchronous approach, there is a progressive loss of

Setup	Throughput	Speedup	Efficiency
1 Node (No distribution)	124.18 img/sec	-	-
2 Nodes (2 Workers + 1 PS)	195.60 img/sec	1.58	0.79
4 Nodes (4 Workers + 3 PS)	383.09 img/sec	3.09	0.77
8 Nodes (8 Workers + 7 PS)	809.10 img/sec	6.52	0.82
16 Nodes (16 Workers + 7 PS)	1278.53 img/sec	10.30	0.64

Table 3.2: Speedup comparing a single node between different asynchronous distributed configurations and their speedup-resources relation.

Setup	Throughput	Speedup	Efficiency
1 Node (No distribution)	124.18 img/sec	-	-
2 Nodes (2 Workers + 1 PS)	216.32 img/sec	1.74	0.87
4 Nodes (4 Workers + 3 PS)	401.09 img/sec	3.23	0.80
8 Nodes (8 Workers + 7 PS)	758.64 img/sec	6.11	0.76
16 Nodes (16 Workers + 7 PS)	1311.37 img/sec	10.56	0.66

Table 3.3: Speedup comparing a single node between different synchronous distributed configurations and their speedup-resources relation.

throughput. The more nodes, the more loss but in this case, the synchronous achieves better efficiency and speedup results than mixed-asynchronous.

3.2 Learning time

When studying the impact of distribution on the training process, there are two main factors to take into account. First, the time required for the model to reach a target loss value, which is the target function being optimized, determines which is the speedup on the training process. Second, the final accuracy determines how the asynchronous/syncrhonous updates affect the optimization of the cost function.

3.2.1 Mixed-Asynchronous

Despite the throughput increase is close to linear with respect to the number of nodes, Figure 3.7 shows how the time required by each setup to reach a target loss value does not benefit linearly from the additional nodes. This result is expected, since for an asynchronous gradient descent method with N workers each model update is performed with respect to weights wich are $N-1$ steps old on average.

The final accuracies on the test set reached by each configuration are de-

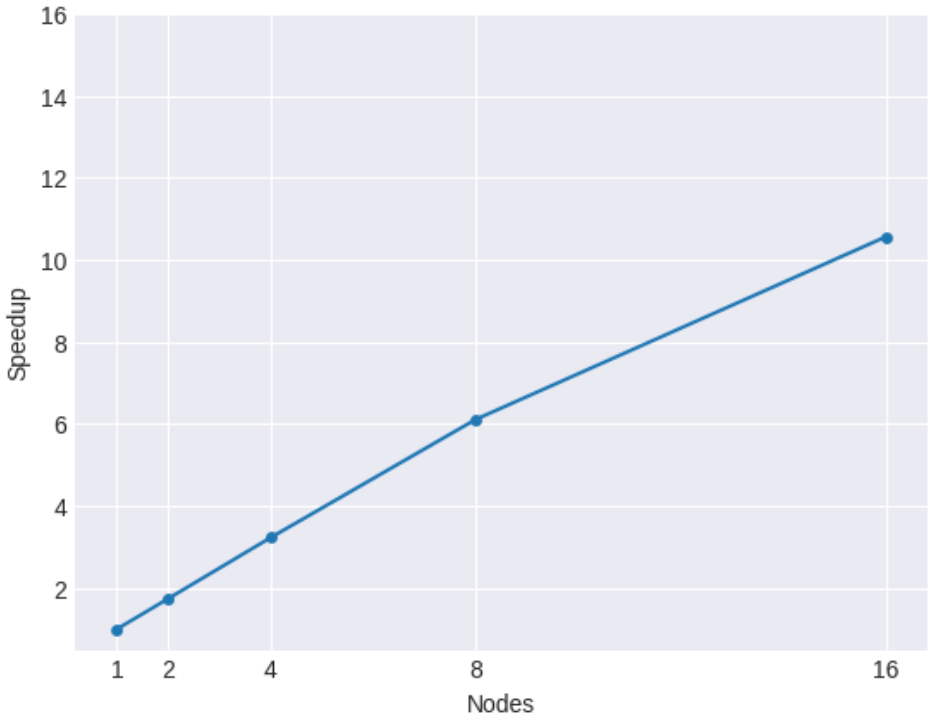


Figure 3.6: Distribution speedup using several nodes, with 4 GPUs, with the optimal synchronous configuration. The baseline is one node with 4 GPUs.

tailed in Table 3.4. None of the distributed setups is able to reach the final accuracy of the single node model, confirming that the stale gradients have a negative impact on the final minima reached at which the model converges. Moreover, we found keeping a similar throughput between worker nodes to be a critical factor for a successful learning process, since workers that are constantly behind the rest of nodes do nothing but aggravate the stale gradients problem.

Workers (GPUs)	Test Accuracy	Time (h)	Improvement
1 Node (4)	0.228	106.43	1.00
2 Nodes (8)	0.217	62.78	1.69
4 Nodes (16)	0.202	37.99	2.80
8 Nodes (32)	0.217	22.50	4.73
16 Nodes (64)	0.185	16.49	6.45

Table 3.4: Reached accuracy, spent time and improvement range among the optimal asynchronous setups.

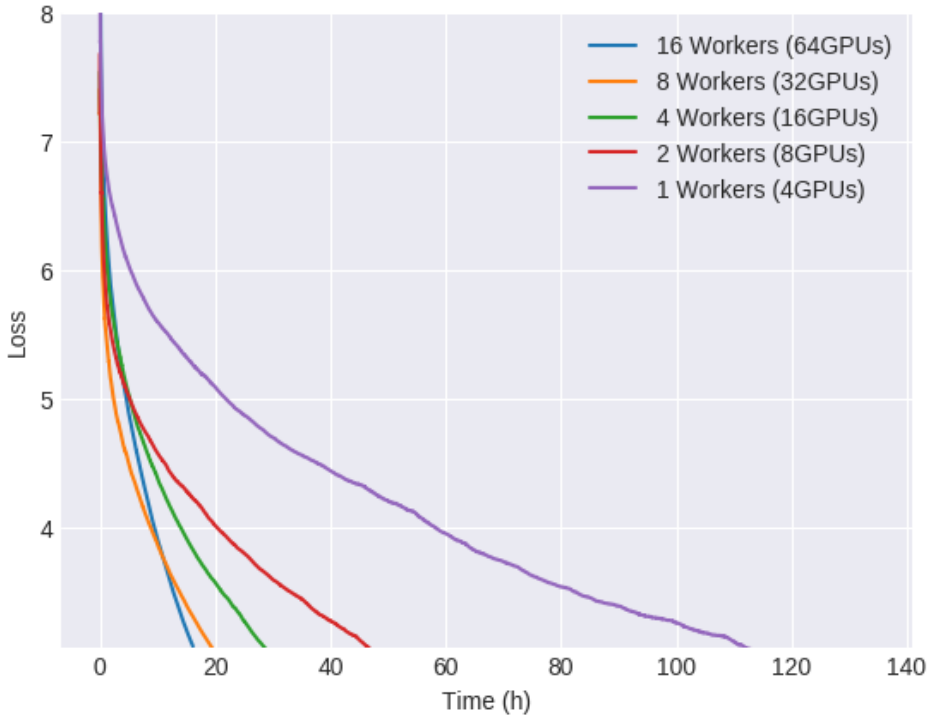


Figure 3.7: Train loss evolution for the different asynchronous distributed configurations. The more nodes, the faster a target loss value is reached.

Figure 3.8: Test accuracy evolution until convergence in different asynchronous distributed configurations.

3.2.2 Synchronous

The synchronous implementation shows an impressive enhancement, from 106 hours to 12 hours without losing accuracy, see Table 3.5. Unlike the near linear throughput increase, there is a positive learning time improvement in the first experiments, with 2 and 4 nodes, and a slight enhancement with the others experiments. Figures 3.9 and 3.10 show the biggest difference is between one and two nodes, then the improvement decreases. The more nodes used, the more network and the more risk to suffer bottlenecks and parameter server saturation.

Besides the increment of throughput, the synchronous method achieves a better accuracy in less time than the mixed-asynchronous approach. Thanks to the synchronous distributed strategy, the gradient staleness problem is avoided and it does not harm the overall accuracy contrary to the mixed-asynchronous. Table 3.5 shows the reached accuracies and they are better than the single node.

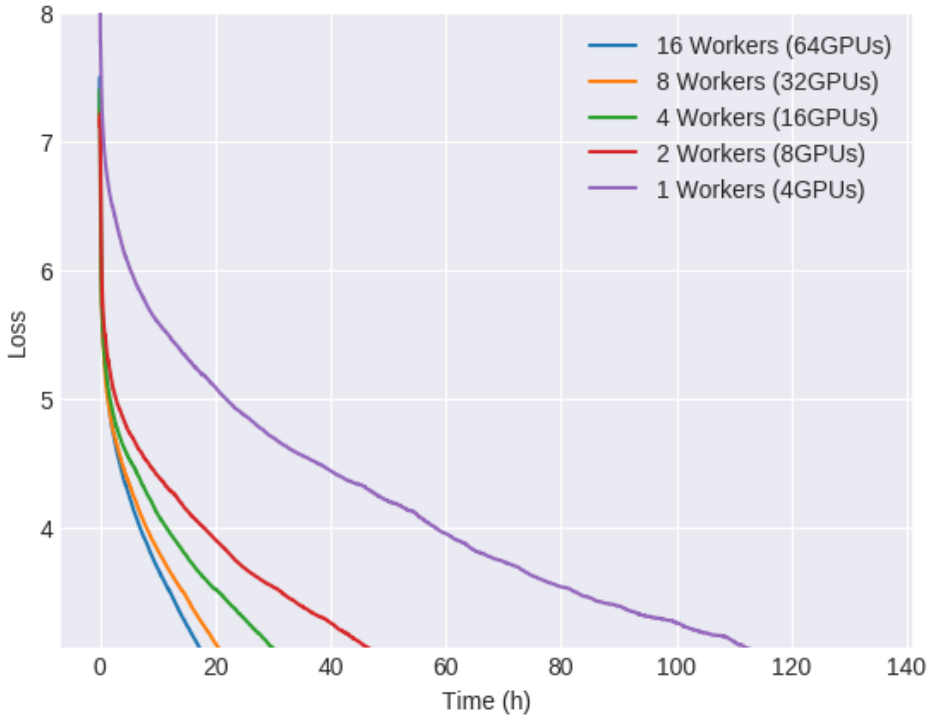


Figure 3.9: Train loss evolution for the different synchronous distributed configurations. The more nodes, the faster a target loss value is reached.

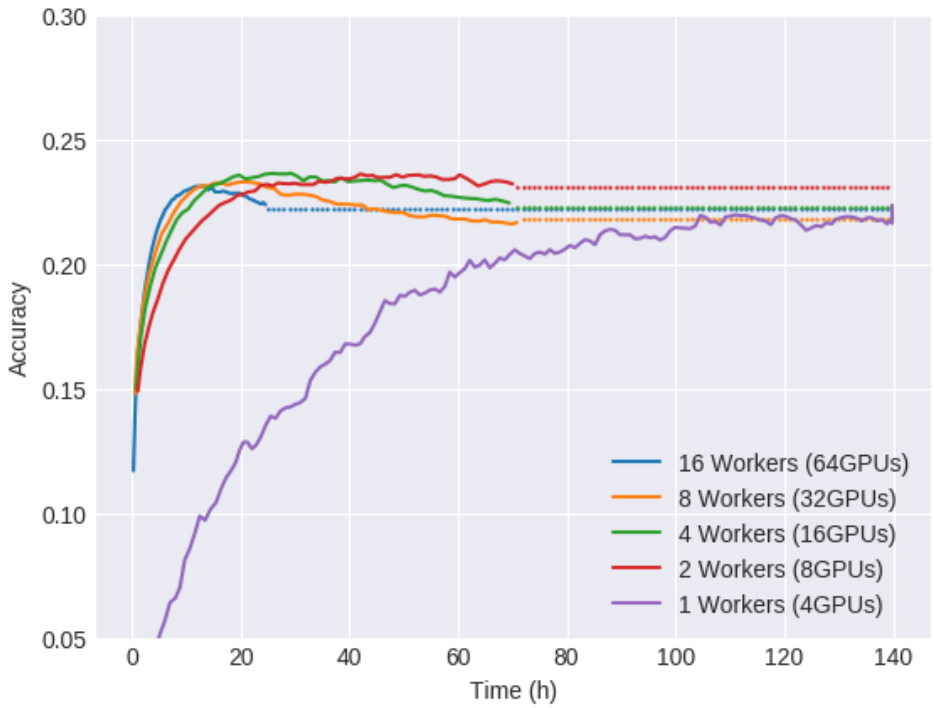


Figure 3.10: Train loss evolution for the different synchronous distributed configurations. The more nodes, the faster a target loss value is reached.

Workers (GPUs)	Test Accuracy	Time (h)	Improvement
1 Node (4)	0.228	106.43	1.00
2 Nodes (8)	0.239	44.21	2.40
4 Nodes (16)	0.238	21.01	5.06
8 Nodes (32)	0.235	14.19	7.50
16 Nodes (64)	0.232	12.33	8.63

Table 3.5: Reached accuracy, spent time and improvement range among the optimal synchronous setups.

3.2.3 Comparing strategies

The learning results in both strategies are different, especially with the reached accuracy. Figure 3.11 shows that meanwhile the synchronous method reaches the same or higher accuracy than the baseline and it is stable independent of the number of nodes, the asynchronous method achieve lower accuracies with an irregular pattern. This is, as commented before, due to the staleness problem in the asynchronous approach.

An important result to point is the accuracy result of 8 nodes in mixed-asynchronous training, it should have been lower than the 4 nodes and it is at same level as 2 nodes result. This might be caused by several factors, the stale gradients and the random queues of data. The workers are fed with different data in each experiment run and it might be possible that the results will not be the same in others executions.

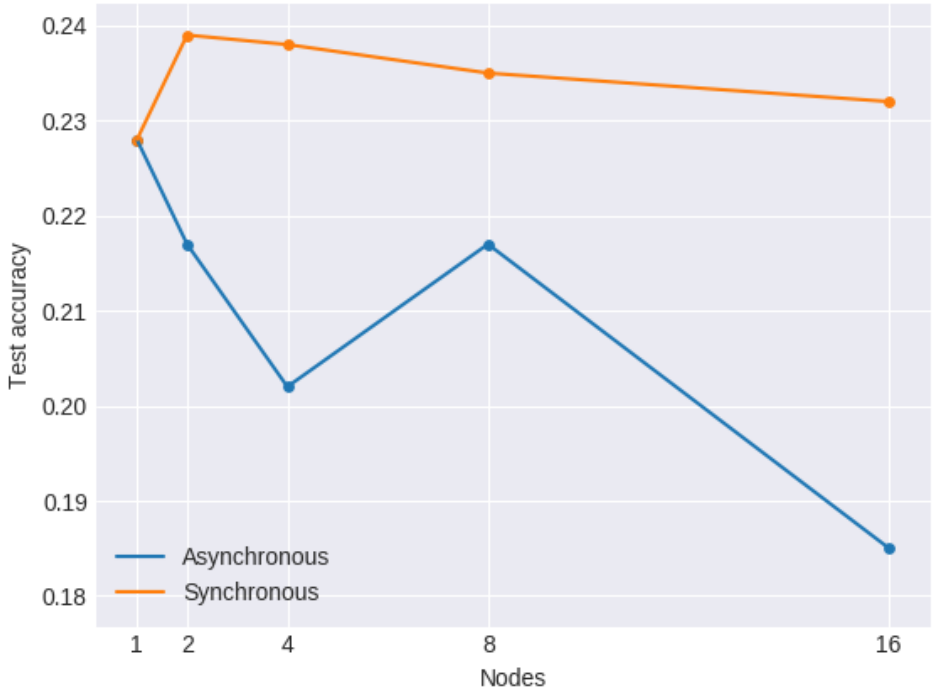


Figure 3.11: Accuracy reached by both strategies using different number of nodes.

In contrast to the accuracy results, both show a decreasing learning time as more nodes are used. Figure 3.12 shows how the time to reach the maximum accuracy decreases similarly in both implementations. The best improvements are in the 2 and 4 nodes experiments.

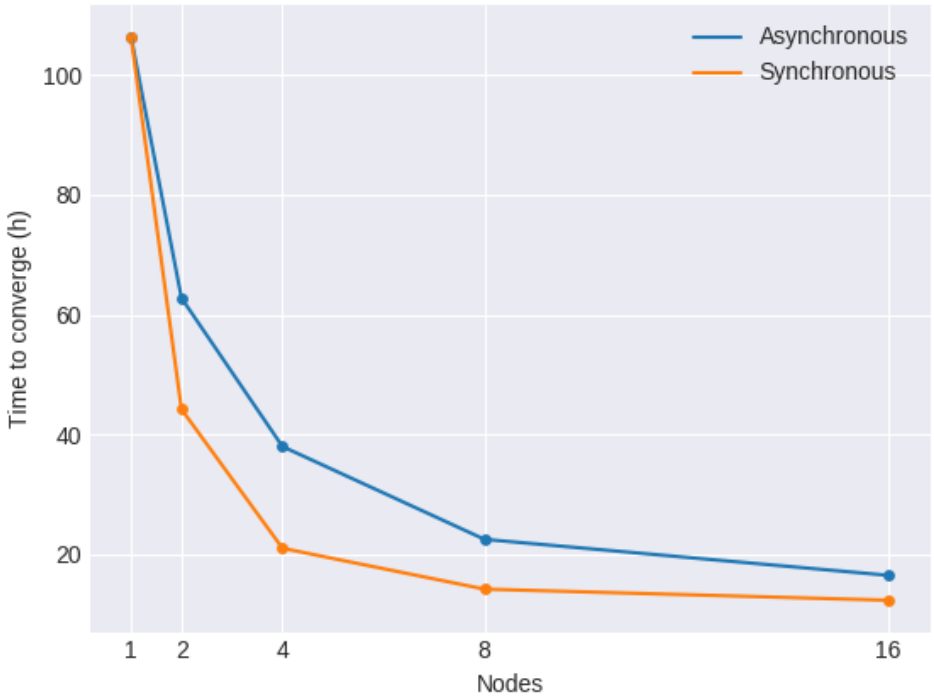


Figure 3.12: Time spent to converge by both strategies using different number of nodes.

4. Conclusions

Next years, Deep Learning will become more important everyday. The Deep Learning processes will have to be done in the shortest possible time, besides the amount of data to be processed does not stop increasing everyday. This boost of requirements will allow Deep Learning to encompass more fields and face more complex achievements. High performance computers have to help Deep Learning to go further and to do it faster.

4.1 Project outcomes

Barcelona Supercomputing Center strategic vision

This work shows that Deep Learning can be accelerated using high performance computers and applied in ones that were not created for this purpose. With these results, Barcelona Supercomputing Center may change its strategic vision about Deep Learning and may start new research fields. Besides, this work explains how to deploy a Deep Learning cluster inside its infrastructures.

Resource usage optimization

In this project, different modes to optimize the available resources have been evaluated. First, we observed that placing the parameter servers in the same machine as the worker processes does not affect in a significant way compared to the poor improvement putting the same parameters servers alone in others machines. Additionally, we checked different approaches for using all the GPUs in a node, launching a different process per GPU or using only one process an merge the results from each GPU before sending them to the parameter servers.

Results and costs

Distributing Deep Learning inside Minotauro we achieved a great improvement compared to one GPU. The throughput rises from 31 images per second using one GPU to 1311 images per second using 64 GPUs in 16 nodes, the learning time decreases from 420 hours to 12 hours and we have been able to maintain the accuracy. There are two distribution strategies in this project, the mixed-asynchronous and the synchronous. The synchronous approach overcame the mixed-asynchronous method in throughput, learning time and obtained accuracy. It may seem that the synchronous technique is better in all cases, nevertheless, [1, 2] show that the asynchronous approach is better, in throughput terms, than the synchronous when a large number of workers is used.

However, any experiment has not been done at once. Most of them showed problems at the execution moment and lots of adjustments had to have done. The cluster was shared with other research groups and the experiment jobs had to wait in a queue for their execution, this increased the time between experiment runs. In addition, some nodes died during the execution of distributed training without possibility of restore and these jobs had to be cancelled. Besides, this project is done using different versions of TensorFlow due to the lack of the synchronous methods in the earlier versions, during this changes of version, the code had to be adapted in order to work in the new version. Table 4.1 shows a brief detail about the spent computation hours in Minotauro in order to do the successful experiments. Table 4.2 details an estimated cost analysis[1] using Amazon EC2 GPU instances.

			CPU hours		Experiment hours	
Nodes	Cores	GPUs	Mixed-async	Sync	Mixed-async	Sync
1	16	4	2240	-	140	-
2	32	8	3840	2304	120	72
4	64	16	2880	4800	45	75
8	128	32	3840	9600	30	75
16	256	64	5120	6400	20	25
Total			41024		602 $\approx$ 25 days	

Table 4.1: Hour detail of Minotauro key experiments. One CPU hour represents one CPU core computing during one hour. The number of hours are approximate.

[1] Amazon AWS Prices (https://aws.amazon.com/ec2/pricing/on-demand/)

Nodes (GPUs)	Hours	Cost (€)	Amazon instance (€ Price)
Successful experiments			
1 (4)	140	544.32	4 x p2.xlarge (3.888/hour)
2 (8)	192	1492.99	1 x p2.8xlarge (7.776/hour)
4 (16)	120	1866.24	1 x p2.16xlarge (15.552/hour)
8 (32)	105	3265.92	2 x p2.16xlarge (31.104/hour)
16 (64)	45	2799.36	4 x p2.16xlarge (62.208/hour)
Failed experiments			
4 (16)	152	2363.90	1 x p2.16xlarge (15.552/hour)
8 (32)	14	435.45	2 x p2.16xlarge (31.104/hour)
16 (64)	70	4354.56	4 x p2.16xlarge (62.208/hour)
Total hours: 838 ≈ 35 days		Total cost ≈ € 17,000.00	

Table 4.2: Estimated cost analysis of all experiments using Amazon Web Services instances.

Publications

Furthermore, this research generated two papers that have been accepted in two conferences:

- 17th IEEE/ACM International Symposium on Cluster [18]
- Cloud and Grid Computing and in the International Conference of Computational Science (ICCS 2017) [17]

This publications show the implementation of the mixed-asynchronous approach, one focused in the implementation of distribution and the other focused in the learning results and how the distribution affects to the accuracy. This two publications prove that there is a growing interest in high performance computing fields for Deep Learning applications.

4.2 Personal outcome

Thanks to this project I was introduced into Deep Learning and high performance computing research fields. I have had the opportunity to learn a new and exciting field as Deep Learning combined with the experience of working with high performance computers at Barcelona Supercomputing Center.

In addition, being between these two fields, Deep Learning and supercomputing, offered to me a global vision about how these two fields can be combined and adapted to work together and get better results in less time. In addition, working with supercomputers helped me to achieve a new level of understanding about this kind of architectures and have the opportunity

of using technologies such CUDA or MPI that I could not use in traditional environments. On the other hand, acquiring knowledge about Deep Learning has been very valuable in order to increase my background and be able to apply this expertise in the future.

To finish, during this project I faced different difficulties. Starting with a new framework such TensorFlow is not easy, it has a high learning curve due to the graph paradigm that it follows. Another point was the Deep Learning world, I was familiar with some DL concepts but they were just the tip of the iceberg, I had to immerse in a exciting but complex universe. On the other hand I have learned that working with big computers entails bigger problems and more convoluted solutions.

4.3 Future work

This job may embrace several research areas. Firstly, we would test this solution in bigger heterogeneous machines with more nodes and different structures in order to test if this solution is scalable in bigger environments and prove which strategy is better for different clusters. The MareNostrum 4 supercomputer will offer new Intel Knights Landing and IBM POWER9 processors with new Nvidia GPUs clusters[2], with these new technologies available for research, it would be intriguing to try this solution on these new clusters. Besides, a proper implementation of TensorFlow using PyCOMPSs [94] instead gRPC and the Tensorflow distributed routines may be valubale for showing if PyCOMPSs implementation might be an alternative for the BSC infrastructures instead of the generic implementation of TensorFlow. Also, TensorFlow will support RDMA[3] in a near future and comparing the results of this thesis with ones using RDMA will be intriguing. In addition, compare these results with the results from other frameworks such as Caffe2 or PyTorch may be valuable.

Owing to the lack of nodes, there is another strategy that we could not test, the synchronous gradient descent with backup workers. Prove this strategy and compare it to the asynchronous in larger machines would be interesting.

Nowadays, there are some projects in order to create new tools to manage a TensorFlow cluster, one of this tools is TensorFlowOnSpark[4], developed by the Yahoo Hadoop Team, that permits to use Spark[5] to manage a cluster of TensorFlow and combine Spark and Hadoop data sources with TensorFlow.

[2]BSC press release (https://www.bsc.es/news/bsc-news/marenostrum-4-supercomputer-be-12-times-more-powerful-marenostrum-3)

[3]A Tutorial of the RDMA Model (https://www.hpcwire.com/2006/09/15/a_tutorial_of_the_rdma_model-1/)

[4]Yahoo Hadoop Blog (http://yahoohadoop.tumblr.com/post/157196317141/open-sourcing-tensorflowonspark-distributed-deep)

[5]Apache Spark Website (http://spark.apache.org/)

This will be useful for implementing TensorFlow on existing Big Data clusters that operate with Spark. This kind of initiatives help to expand the domains of TensorFlow to production environments.

To finish, at the moment there are tools to use TensorFlow as a service like Floyd Hub[6], this service allows to upload a TensorFlow code and run it without caring about the system setup. Nevertheless it does not support distribution and it is constrained to the number of GPUs available on the virtual machines. Creating a tool for researchers may be valuable, a tool that would allow researchers to run distributed TensorFlow porcesses without regard about network parameters, GPU configurations or parallelism management. Simply sending the code to the tool. A tool of this kind will prevent researchers to waste time dealing with problems about distribution.

[6]Floyd Hub Website (https://www.floydhub.com/)

Bibliography

[1] Martın Abadi et al. "TensorFlow: Large-scale machine learning on heterogeneous systems". In: (2015).

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

[3] Zeynep Akata, Florent Perronnin, Zaid Harchaoui, and Cordelia Schmid. "Label-embedding for image classification". In: *IEEE transactions on pattern analysis and machine intelligence* (2016).

[4] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *ISCA* (2017).

[5] Víctor Campos et al. "Sentiment Concept Embedding for Visual Affect Recognition". 2017.

[6] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. "Comparative Study of Caffe, Neon, Theano, and for Deep Learning". In: *CoRR* abs/1511.06435 (2015). URL: http://arxiv.org/abs/1511.06435.

[7] Marco Baroni and Roberto Zamparelli. "Nouns are vectors, adjectives are matrices: Representing adjective-noun constructions in semantic space". In: *EMNLP*. 2010.

[8] James Bergstra et al. "Theano: Deep learning on gpus with python". In: *NIPS Workshops*. 2011.

[9] Gedas Bertasius, Jianbo Shi, and Lorenzo Torresani. "DeepEdge: A Multi-Scale Bifurcated Deep Network for Top-Down Contour Detection". In: *CoRR* abs/1412.1123 (2014). URL: http://arxiv.org/abs/1412.1123.

[10] Damian Borth, Tao Chen, Rongrong Ji, and Shih-Fu Chang. "Sentibank: large-scale ontology and classifiers for detecting sentiment and emotions in visual content". In: *ACM MM*. 2013.

[11] Damian Borth, Rongrong Ji, Tao Chen, Thomas Breuel, and Shih-Fu Chang. "Large-scale visual sentiment ontology and detectors using adjective noun pairs". In: *ACM MM*. 2013.

[12] Barcelona Supercomputing Center BSC. *About BSC*. 2016. URL: https://www.bsc.es/discover-bsc/the-centre/what-we-do.

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

[14] Fabian Caba Heilbron, Victor Escorcia, Bernard Ghanem, and Juan Carlos Niebles. "Activitynet: A large-scale video benchmark for human activity understanding". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 961–970.

[15] Michel Cabanac. "What is emotion?" In: *Behavioural processes* 60.2 (2002).

[16] Victor Campos, Amaia Salvador, Brendan Jou, and Xavier Giro-i-Nieto. "Diving Deep into Sentiment: Understanding Fine-tuned CNNs for Visual Sentiment Prediction". In: *ASM, ACM MM Workshops*. 2015.

[17] Víctor Campos, Francesc Sastre, Maurici Yagües, Míriam Bellver, Xavier Giró-i-Nieto, and Jordi Torres. "Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster". 2017.

[18] Víctor Campos, Francesc Sastre, Maurici Yagües, Xavier Giró-i-Nieto, and Jordi Torres. "Scaling a Convolutional Neural Network for classification of Adjective Noun Pairs with TensorFlow on GPU Clusters". 2017.

[19] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. "Revisiting Distributed Synchronous SGD". In: *ICLR Workshops*. 2016.

[20] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. "Revisiting Distributed Synchronous SGD". ICLR 2017 conference submission. 2016.

[21] Tao Chen, Damian Borth, Trevor Darrell, and Shih-Fu Chang. "DeepSentiBank: Visual sentiment concept classification with deep convolutional neural networks". In: *arXiv:1410.8586* (2014).

[22] Tianqi Chen et al. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274* (2015).

[23] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759* (2014).

[24] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan catanzaro, and Ng Andrew. "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.

[25] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. "7: A Matlab-like Environment for Machine Learning". In: *BigLearn, NIPS Workshop*. 2011.

[26] Michael Copeland. *The Difference Between AI, Machine Learning, and Deep Learning? | NVIDIA Blog*. 2016. URL: https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/ (visited on 04/25/2017).

[27] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. "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.

[28] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. "R-FCN: Object Detection via Region-based Fully Convolutional Networks". In: *CoRR* abs/1605.06409 (2016). URL: http://arxiv.org/abs/1605.06409.

[29] Zhenwen Dai, Andreas C. Damianou, Javier González, and Neil D. Lawrence. "Variational Auto-encoded Deep Gaussian Processes". In: *CoRR* abs/1511.06455 (2015). URL: http://arxiv.org/abs/1511.06455.

[30] Vaidehi Dalmia, Hongyi Liu, and Shih-Fu Chang. "Columbia MVSO Image Sentiment Dataset". In: *arXiv preprint arXiv:1611.04455* (2016).

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "ImageNet: A large-scale hierarchical image database". In: *CVPR*. 2009.

[32] Chao Dong, Yubin Deng, Chen Change Loy, and Xiaoou Tang. "Compression Artifacts Reduction by a Deep Convolutional Network". In: *CoRR* abs/1504.06993 (2015). URL: http://arxiv.org/abs/1504.06993.

[33] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. "Image Super-Resolution Using Deep Convolutional Networks". In: *CoRR* abs/1501.00092 (2015). URL: http://arxiv.org/abs/1501.00092.

[34] Andrea Frome, Greg S Corrado, Jon Shlens, Samy Bengio, Jeff Dean, Tomas Mikolov, et al. "Devise: A deep visual-semantic embedding model". In: *NIPS*. 2013.

[35] Ross B. Girshick. "Fast R-CNN". In: *CoRR* abs/1504.08083 (2015). URL: http://arxiv.org/abs/1504.08083.

[36] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). URL: http://arxiv.org/abs/1311.2524.

[37] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative Adversarial Networks". In: *ArXiv e-prints* (June 2014). arXiv: 1406.2661 [stat.ML].

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *CVPR*. 2016.

[39] Eduard H Hovy. "What are Sentiment, Affect, and Emotion? Applying the Methodology of Michael Zock to Sentiment Analysis". In: *Language Production, Cognition, and the Lexicon* 48 (2014).

[40] Jonathan Huang et al. "Speed/accuracy trade-offs for modern convolutional object detectors". In: *CoRR* abs/1611.10012 (2016). URL: http://arxiv.org/abs/1611.10012.

[41] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional architecture for fast feature embedding". In: *ACM MM*. 2014.

[42] Yu-Gang Jiang, Baohan Xu, and Xiangyang Xue. "Predicting Emotions in User-Generated Videos." In: *AAAI*. 2014.

[43] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. "How to scale distributed deep learning?" In: *arXiv preprint arXiv:1611.04581* (2016).

[44] Brendan Jou and Shih-Fu Chang. "Deep Cross Residual Learning for Multitask Visual Recognition". In: *ACM MM*. 2016.

[45] Brendan Jou and Shih-Fu Chang. "Going Deeper for Multilingual Visual Sentiment Detection". In: *arXiv preprint arXiv:1605.09211* (2016).

[46] Brendan Jou, Tao Chen, Nikolaos Pappas, Miriam Redi, Mercan Topkara, and Shih-Fu Chang. "Visual affect around the world: A large-scale multilingual visual sentiment ontology". In: *ACM MM*. 2015.

[47] Andrej Karpathy and Fei-Fei Li. "Deep Visual-Semantic Alignments for Generating Image Descriptions". In: *CoRR* abs/1412.2306 (2014). URL: http://arxiv.org/abs/1412.2306.

[48] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. "Large-scale video classification with convolutional neural networks". In: *CVPR*. 2014.

[49] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. "On large-batch training for deep learning: Generalization gap and sharp minima". In: *arXiv preprint arXiv:1609.04836* (2016).

[50] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *arXiv preprint arXiv:1404.5997* (2014).

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *NIPS*. 2012.

[52] Richard D Lane and Lynn Nadel. *Cognitive neuroscience of emotion*. Oxford University Press, USA, 2002.

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

[54] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998).

[55] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew P. Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network". In: *CoRR* abs/1609.04802 (2016). URL: http://arxiv.org/abs/1609.04802.

[56] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. "SSD: Single Shot MultiBox Detector". In: *CoRR* abs/1512.02325 (2015). URL: http://arxiv.org/abs/1512.02325.

[57] He Ma, Fei Mao, and Graham W Taylor. "Theano-MPI: a Theano-based Distributed Training Framework". In: *arXiv preprint arXiv:1605.08325* (2016).

[58] Jana Machajdik and Allan Hanbury. "Affective Image Classification Using Features Inspired by Psychology and Art Theory". In: *ACM MM*. 2010.

[59] Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, and Alan L. Yuille. "Explain Images with Multimodal Recurrent Neural Networks". In: *CoRR* abs/1410.1090 (2014). URL: http://arxiv.org/abs/1410.1090.

[60] Daniel McDuff, Rana El Kaliouby, Jeffrey F Cohn, and Rosalind W Picard. "Predicting ad liking and purchase intent: Large-scale analysis of facial responses to ads". In: *IEEE Transactions on Affective Computing* 6.3 (2015).

[61] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space". In: *ICLR*. 2013.

[62] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality". In: *NIPS*. 2013.

[63] Frank F Montalvo. "Bridging the Affective Gap". In: *Affective Education: Methods and Techniques* (1989).

[64] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. "SparkNet: Training Deep Networks in Spark". In: *arXiv preprint arXiv:1511.06051* (2015).

[65] Takuya Narihira, Damian Borth, Stella X Yu, Karl Ni, and Trevor Darrell. "Mapping Images to Sentiment Adjective Noun Pairs with Factorized Neural Nets". In: *arXiv preprint arXiv:1511.06838* (2015).

[66] Mohammad Norouzi, Tomas Mikolov, Samy Bengio, Yoram Singer, Jonathon Shlens, Andrea Frome, Greg S Corrado, and Jeffrey Dean. "Zero-shot learning by convex combination of semantic embeddings". In: *arXiv preprint arXiv:1312.5650* (2013).

[67] CUDA Nvidia. "Compute unified device architecture programming guide". In: (2007).

[68] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. "Wavenet: A generative model for raw audio". In: *arXiv preprint arXiv:1609.03499* (2016).

[69] Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. "Conditional Image Generation with PixelCNN Decoders". In: *CoRR* abs/1606.05328 (2016). URL: http://arxiv.org/abs/1606.05328.

[70] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. "Learning and transferring mid-level image representations using convolutional neural networks". In: *CVPR*. 2014.

[71] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang. "GPU Asynchronous Stochastic Gradient Descent to Speed Up Neural Network Training". In: *ArXiv e-prints* (2013). arXiv: 1312.6186 [cs.CV].

[72] Bo Pang and Lillian Lee. "Opinion Mining and Sentiment Analysis". In: *Information Retrieval* 2.1-2 (2008).

[73] Nikolaos Pappas, Miriam Redi, Mercan Topkara, Brendan Jou, Hongyi Liu, Tao Chen, and Shih-Fu Chang. "Multilingual Visual Sentiment Concept Matching". In: *ICMR*. 2016.

[74] Rosalind W. Picard. *Affective Computing*. Vol. 252. MIT Press Cambridge, 1997.

[75] Robert Plutchik. *Emotion: A Psychoevolutionary Synthesis*. Harper & Row, 1980.

[76] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *CoRR* abs/1511.06434 (2015). URL: http://arxiv.org/abs/1511.06434.

[77] S Sundhar Ram, Angelia Nedic, and Venugopal V Veeravalli. "Asynchronous gossip algorithms for stochastic optimization". In: *GameNets*. 2009.

[78] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection". In: *CoRR* abs/1506.02640 (2015). URL: http://arxiv.org/abs/1506.02640.

[79] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). URL: http://arxiv.org/abs/1506.01497.

[80] Francesc Sastre. *Cluster creator for MT*. https://github.com/xiscosc/cluster_mt_creator. 2017.

[81] Christian J. Schuler, Michael Hirsch, Stefan Harmeling, and Bernhard Schölkopf. "Learning to Deblur". In: *CoRR* abs/1406.7444 (2014). URL: http://arxiv.org/abs/1406.7444.

[82] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs". In: *Interspeech 2014*. Sept. 2014.

[83] Wei Shen, Xinggang Wang, Yan Wang, Xiang Bai, and Zhijiang Zhang. "DeepContour: A Deep Convolutional Feature Learned by Positive-Sharing Loss for Contour Detection". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[84] Stefan Siersdorfer, Enrico Minack, Fan Deng, and Jonathon Hare. "Analyzing and predicting sentiment of images on the social web". In: *ACM MM*. 2010.

[85] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016).

[86] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556.

[87] Richard Socher, Milind Ganjoo, Christopher D Manning, and Andrew Ng. "Zero-shot learning through cross-modal transfer". In: *NIPS*. 2013.

[88] Richard Socher, Brody Huval, Christopher D Manning, and Andrew Y Ng. "Semantic compositionality through recursive matrix-vector spaces". In: *EMNLP-CoNLL*. 2012.

[89] J. T. Springenberg. "Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks". In: *ArXiv e-prints* (Nov. 2015). arXiv: 1511.06390 [stat.ML].

[90] Jian Sun, Wenfei Cao, Zongben Xu, and Jean Ponce. "Learning a Convolutional Neural Network for Non-uniform Motion Blur Removal". In: *CoRR* abs/1503.00593 (2015). URL: http://arxiv.org/abs/1503.00593.

[91] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions". In: *CVPR*. 2015.

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

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

[94] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. "PyCOMPSs: Parallel computational workflows in Python". In: *The International Journal of High Performance Computing Applications* 31.1 (2017), pp. 66–82. DOI: 10.1177/1094342015594678. eprint: http://dx.doi.org/10.1177/1094342015594678. URL: http://dx.doi.org/10.1177/1094342015594678.

[95] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4 (2012).

[96] Antonio Torralba and Alexei A Efros. "Unbiased look at dataset bias". In: *CVPR*. 2011.

[97] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. "Show and Tell: A Neural Image Caption Generator". In: *CoRR* abs/1411.4555 (2014). URL: http://arxiv.org/abs/1411.4555.

[98] Hee Lin Wang and Loong-Fah Cheong. "Affective understanding in film". In: *IEEE* (2006).

[99] Jingwen Wang, Jianlong Fu, Yong Xu, and Tao Mei. "Beyond Object Recognition: Visual Sentiment Analysis with Deep Coupled Adjective and Noun Neural Networks". In: *IJCAI*. 2016.

[100] Jason Weston, Samy Bengio, and Nicolas Usunier. "Large scale image annotation: learning to rank with joint word-image embeddings". In: *Machine learning* (2010).

[101] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[102] Saining Xie and Zhuowen Tu. "Holistically-Nested Edge Detection". In: *CoRR* abs/1504.06375 (2015). URL: http://arxiv.org/abs/1504.06375.

[103] Can Xu, Suleyman Cetintas, Kuang-Chih Lee, and Li-Jia Li. "Visual sentiment prediction with deep convolutional neural networks". In: *arXiv preprint arXiv:1411.5731* (2014).

[104] Maurici Yagües. "Image recognition with Deep Learning techniques and Tensorflow". 2016.

[105] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. "How transferable are features in deep neural networks?" In: *NIPS*. 2014.

[106] Quanzeng You, Liangliang Cao, Hailin Jin, and Jiebo Luo. "Robust Visual-Textual Sentiment Analysis: When Attention meets Tree-structured Recursive Neural Networks". In: *ACM MM*. 2016.

[107] Quanzeng You, Jiebo Luo, Hailin Jin, and Jianchao Yang. "Building a Large Scale Dataset for Image Emotion Recognition: The Fine Print and the Benchmark". In: *AAAI*. 2016.

[108] Quanzeng You, Jiebo Luo, Hailin Jin, and Jianchao Yang. "Robust Image Sentiment Analysis using Progressively Trained and Domain Transferred Deep Networks". In: *AAAI*. 2015.

[109] Dong Yu and Xuedong Huang. "Microsoft Computational Network Toolkit (CNTK)". In: A Tutorial Given at NIPS 2015 Workshops. 2015. URL: http://research.microsoft.com/en-us/um/people/dongyu/CNTK-Tutorial-NIPS2015.pdf.

[110] Richard Zhang, Phillip Isola, and Alexei A. Efros. "Colorful Image Colorization". In: *CoRR* abs/1603.08511 (2016). URL: http://arxiv.org/abs/1603.08511.

[111] Sixin Zhang, Anna E Choromanska, and Yann LeCun. "Deep learning with elastic averaging SGD". In: *NIPS*. 2015.

[112] Ziming Zhang and Venkatesh Saligrama. "Zero-shot learning via semantic similarity embedding". In: *CVPR*. 2015.