Deployment Service for Scalable Distributed Deep Learning Training on Multiple Clouds

Javier Jorge[1][a], Germán Moltó[1][b], Damian Segrelles[1][c], João Pedro Fontes[2] and Miguel Angel Guevara[2]

[1]*Instituto de Instrumentación para Imagen Molecular (I3M), Centro mixto CSIC - Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia*
{jjorge, gmolto, dquilis}@dsic.upv.es

[2]*Computer Graphics Center, University of Minho, Campus de Azurém Guimarães, Portugal*
{joao.fontes, miguel.guevara}@ccg.pt

Keywords: Cloud computing, Deep Learning, Multi-Cloud

Abstract: This paper introduces a platform based on open-source tools to automatically deploy and provision a distributed set of nodes that conduct the training of a deep learning model. To this end, the deep learning framework TensorFlow will be used, as well as the Infrastructure Manager service to deploy complex infrastructures programmatically. The provisioned infrastructure addresses: data handling, model training using these data, and the persistence of the trained model. For this purpose, public Cloud platforms such as Amazon Web Services (AWS) and General-Purpose Computing on Graphics Processing Units (GPGPU) are employed to dynamically and efficiently perform the workflow of tasks related to training deep learning models. This approach has been applied to real-world use cases to compare local training versus distributed training on the Cloud. The results indicate that the dynamic provisioning of GPU-enabled distributed virtual clusters in the Cloud introduces great flexibility to cost-effectively train deep learning models.

1 INTRODUCTION

Artificial Intelligence (AI), mainly through Deep Learning (DL) models, is currently dominating the field of classification tasks, especially after the disruptive results obtained in 2012 in the most challenging image classification task at that moment (Krizhevsky et al., 2012), defeating the SVM (Support Vector Machine) (Vapnik, 1998) approaches that were leading those contests previously. DL models are trained using Backpropagation (Rumelhart et al., 1985), an iterative procedure that could take days or weeks depending on the volume of the data or the complexity of the model. Among the options to reduce the computational cost, the most common technique to parallelize this process is the data parallelism approach (Dean et al., 2012) as that is the most straightforward and easy technique to implement.

Big tech companies and relevant research groups have open-sourced many of their algorithms and methods as AI frameworks for the community (i.e: TensorFlow (TF) (Abadi et al., 2016), PyTorch (PyTorch, 2018) or MXNet (MXNet, 2018)) enabling a quicker and easier evaluation of academic DL models and the deployment in production environments. These frameworks provide a uniform set of functions, such as parallelization on multiple nodes, flexibility to code using different interfaces, automatic differentiation, similar abstractions regarding the structure of the network, and “model zoos” to deploy an out-of-the-box model in production quickly.

Considering the interest of the academia and industry to use DL models in their pipelines, it is of vital importance to provide easier, faster and better tools to deploy their algorithms and applications. To this end, research groups and companies are using Cloud services (such as Google’s Cloud AI (Cloud AI, 2018) or Amazon SageMaker (Amazon SageMaker, 2018)) to develop their DL solutions, as long as they are

[a] https://orcid.org/0000-0002-9279-6768
[b] https://orcid.org/0000-0002-8049-253X
[c] https://orcid.org/0000-0001-5698-7965

computationally (storage and processing) demanding and require expensive and dedicated hardware such as General-Purpose Computing on Graphics Processing Units (GPGPUs), as it is the case of the DL techniques. However, these Cloud solutions result in the user being locked-in to a particular Cloud and, therefore, loosing the ability to perform infrastructure deployment on different or multiple Cloud providers. This Cloud approach is known as “Infrastructure As Code” (IaC) (Wittig and Wittig, 2016) and it arises to overcome these gaps and define instead the customized computational environment required without interaction with the Cloud provider’s interfaces and enabling non-expert users to define, in a Cloud-agnostic way, the virtual infrastructure as well as the process of managing and provisioning the computational resources from multiple Clouds for different trainings. Notice that a single training is performed within the boundaries of a single cloud to achieve reduced latency.

This paper introduces a Cloud Service that provides an automated DL pipeline, from the deployment and provision of the required infrastructure on Cloud providers, through an IaC approach, to its execution in production. It is based on open-source tools to deploy a customised computational environment based on TF, to perform the distributed DL model training based on data parallelism, on Cloud infrastructures involving accelerated hardware (GPUs) for increased efficiency. As use case to validate the service, it is presented the deployment, configuration and execution of a distributed cluster for training deep neural networks, on a public Cloud provider. This uses resources that the user may not be able to afford otherwise, such as several nodes connected using one or more expensive GPUs. This approach will be conveniently assessed by means of training several DL models on distributed clusters provisioned on Amazon Web Services (AWS). In addition, an actual research problem has been used to validate the proposed service in order to test the system in a real use case documented in the work by Brito at al. (Brito et al., 2018).

2 RELATED WORK

Recently, approaches for efficient DL have been developed *ad-hoc* for specific use cases, such as DeepCell (Bannon et al., 2018), a tool used to perform large-scale cellular image analysis, tuned for these particular problems. For a general purpose, the alternative is using proprietary tools that the Cloud providers offer, tailored for a specific framework that usually depends on the provider, or a high-level API to access these services easily. A thorough review of this can be found in (Luckow et al., 2016), where the authors studied distributed DL tools to use in the automotive industry.

There are other approaches related to the use of multi-GPUs beyond using different nodes or more than one GPU per node. For instance, GPU virtualization provides the interface to use a variable number of GPUs, intercepting local calls and sending them to a GPU server farm to be processed, with thousands of GPUs working in parallel. An example of these proposals is shown in rCUDA (Reaño et al., 2015). However, these approaches are either experimental or more expensive than public Clouds, and they do not provide additional services required for this kind of problems such as storage resources.

The topic of distributed DL has been addressed in the past, as it is the case of the work by Lee et al. (Lee et al., 2018) where a framework for DL across multiple machines using Apache Spark is introduced. There are also approaches that try to speed up this distributed computation providing another layer of abstraction over TF, such as Horovod (Sergeev and Del Balso, 2018), using MPI[1], the commonly used (M)essage (P)assing (I)nterface to intercommunicate nodes.

As opposed to previous works, we introduce an approach for distributed learning that relies exclusively on the capabilities of TF, thus minimizing external dependencies. Also, these distributed clusters can be dynamically deployed on different Cloud back-ends through an Infrastructure-as-Code approach thus fostering reproducibility of the distributed learning platform and the ability to easily profit from GPU resources provided by public Cloud platforms to accelerate training.

3 DEPLOYMENT SERVICE FOR DISTRIBUTED TENSORFLOW

This section identifies the components to design and implement the proposed service. The goal is to train a deep neural network in a distributed way, fostering data parallelism, and considering that the training process can be split according to different shards of data. This scales very easily, and this is how some of the big companies such as Google achieved the amazing breakthroughs these last years, such as AlphaGO (Silver et al., 2017). Among different approaches for doing this, we have followed the scheme

[1]https://www.mpi-forum.org/docs/

from (Dean et al., 2012), that is reproduced in Figure 1. This architecture shows different nodes that can be identified as follows:

- Parameter server (PS): Node(s) that initializes parameters, distributes them to the workers, gathers the modifications of the model's parameters and then applies them, and scatters them back to the workers.
- Model replicas (MR): Node(s) that stores the model and that carries out the backpropagation's forward and backward steps, and finally sends back the updates to the PS. There will be a master among them in charge of persisting the model. These nodes are the ones that require more computational power, i.e: GPU units.
- Data shards: Nodes that provide data for MRs.

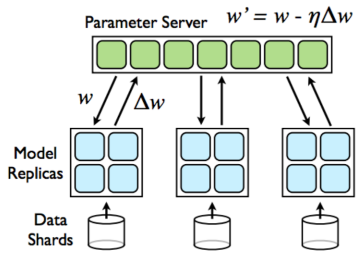

Figure 1: Distributed training scheme: Parameter server and model replicas, in a master-slave setup (Dean et al., 2012).

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

Regarding the data shards, we propose the use of a distributed file system among nodes, in order to share data and model checkpoints as well. In particular, we have chosen the Hadoop Distributed File System (HDFS) (Shvachko et al., 2010), a distributed and scalable file system that is focused on storing massive amounts of data. It has been used mainly on Java's framework Hadoop (Hadoop, 2018) to implement the MapReduce paradigm (Dean and Ghemawat, 2008).

The proposed service deploys the Hadoop cluster in charge of data storage (HDFS) and the TF cluster in charge of processing data, in the same resources and nodes, coexisting along with each other. Concerning training, each node will run a TF script that implements the aforementioned training scheme. As a summary, Figure 2 shows the proposed steps to prepare the architecture required to perform the training.

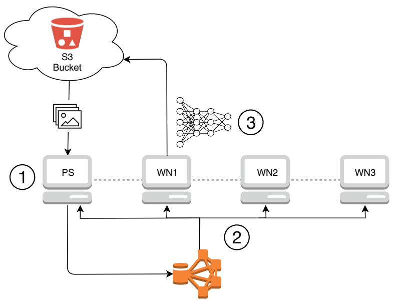

Figure 2: Step 1: Getting the data and uploading it to the HDFS storage. Step 2: Training starts, the PS node and MR nodes collaborate to train the model, using TF and the HDFS file system. Step 3: Once training is completed, a PS node stores the model in a persistent Cloud storage, such as an S3 bucket.

3.1 Infrastructure Manager

The Cloud service, in order to deploy the nodes commented above on Cloud uses an IaC approach relying on the Infrastructure Manager (IM) (Infrastructure Manager, 2018; Caballer et al., 2015)[2]. It is an open-source tool to deploy on-demand customized virtual computing infrastructures. It allows the user to deploy complex infrastructures choosing among multiple Cloud providers, such as AWS, Microsoft Azure or Google Cloud Platform. The IM is the orchestration tool adopted by the European Grid Infrastructure (EGI) (European Grid Infrastructure, 2018) in order to deploy Virtual Machines through the VMOps Dashboard on the EGI Federated Cloud that spans across Europe. This paves the way for easy adoption of the approach described in this work by the European scientific community. It uses "Resource and Application Description Language" (RADL)(Infrastructure Manager, 2018; Caballer et al., 2015), a high-level language to define virtual infrastructures and VM requirements.

The IM processes the RADL file and interacts with the Cloud providers to launch the Master VM that will orchestrate the deployment and configuration. A specific module called Configuration Manager configures this Master node using Ansible (Ansible, 2018), providing the contextualization and configuration files and installing the Contextualization Agent. This module will deploy, configure and provision main roles, in our case, the PS and MR Nodes, including TF and the Hadoop cluster.

The following subsections present the tools employed by the proposed service, in order to prepare the infrastructure required to support the architecture shown in Figure 2. After that, it presents how distributed TF training is achieved through the service.

[2]IM - https://www.grycap.upv.es/im

3.2 Resources' and Operations' definition

The RADL file and the Ansible roles to deploy the whole infrastructure's specifics required to achieve distributed training of a deep neural network describe the node's characteristics, the operations to perform upon them and the system architecture.

The execution of the RADL through the IM prepares the whole environment (PS and MR nodes) to execute the training script. The complete version of the resource allocation configuration files and code are available on GitHub[3], and thoroughly described in (Jorge Cano, 2019). We have relied on an Ansible role to deploy the Hadoop cluster in order to use HDFS as a shared data storage among the nodes. Once this step is completed, the Hadoop cluster is completely deployed, and then we can proceed with data handling, as well as configuring and installing TF. Amazon S3 is used as the initial storage for the data, and the dataset is retrieved from S3 and staged into HDFS upon deployment of the Hadoop cluster, but the data can come from any external source. After deploying the Hadoop cluster, the distributed TF script is prepared, installed and executed using another Ansible role. The set of parameters for the PS node is similar to the MR nodes, with some changes accordingly to their function, such as the `node_type` or whether a GPU is used or not. The role involves four parts: first, some environment preparation with paths and additional variables. Second, the conditional installation of a specific TF version depending on whether the node has a GPU or not. Third, cloning the actual training scripts and running them, depending on the role of the node in the cluster. Finally, we have included the code to upload the resulting model to an S3 bucket to persist it beyond the lifecycle of the dynamically provisioned Hadoop cluster, but we can select any external platform or service to store the model.

3.3 Distributed TensorFlow training

This section shows how we performed the adaptation of the distributed TF code to be used in our pipeline. TF distributed training implements a scheme where one or more nodes act as PS nodes and the rest as MR nodes. The code is available in Github[4].

This work uses the Estimator API in TF, which provides high-level functions that encapsulate the different parts of the machine learning pipeline: model, input, training, validation and evaluation. Providing common headers for these steps they can manage training and inference better, as the whole pipeline is decomposed into isolated functions. By defining the pipeline in these terms, it is entirely managed by TF, without worrying about running iterations, evaluation steps, logs or saving the model manually. Another significant advantage is to have a program that can run in a single node with CPU or GPU, with multiple GPUs or in a distributed way, just indicating these variations as an environment variable called `TF_CONFIG`.

4 Experimentation

In this section, we will use the configuration files, codes and scripts that we have presented in previous sections in order to evaluate our deployment and execution in terms of flexibility and timing. For doing this, we have selected a public Cloud provider to deploy our infrastructure and perform the evaluation. First, we consider executing the training using single nodes, in physical and virtual machines, to get the baseline that we can achieve with the resources at reach. Second, we study the use of a Cloud provider to deploy Virtual Machines with and without specialized hardware, to compare the results to both single and multiple node configurations.

To perform the local experimentation, we used a physical node with Ubuntu 16.04, NVIDIA CUDA 9 - CUDNN 7, TF r1.10. Regarding the hardware, the node includes an Intel(R) Xeon(R) CPU E5-1620 v3 4 cores @ 3.50GHz with 128 GB of RAM along with the GPU GeForce GTX 1080Ti (11GB, 3.5K cores @ 1.6Ghz).

We selected AWS as the public Cloud provider to use during the evaluation. An analysis of the instance types and pricing was carried out, in terms of choosing cost-effective computing resources. To reduce costs, we have selected an Amazon Machine Image (AMI) to accelerate the deployment and avoid some time-consuming tasks such as update packages when we use a fresh Ubuntu 16.04 VM. The AMI used is called Deep Learning Base AMI (Ubuntu) Version 10.0 AMI[5], and it is available in most AWS regions. This AMI comes with NVIDIA CUDA 9 and NVIDIA cuDNN 7 and can be used with several instance types, from a small CPU-only instance to the latest high-powered multi-GPU instances. The information of these instances' families is available on the

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

[4]https://github.com/JJorgeDSIC/DistributedTensorFlowCodeForIM

[5]https://aws.amazon.com/marketplace/pp/B077GCZ4GR

AWS website[6]. We have selected p2 instances, the p2.xlarge in particular, that are composed of an Intel Xeon E5-2686 v4 (Broadwell) CPU and NVIDIA K80 GPUs (12GiB, 2.5K cores). It is important to remark the years of development between the virtualized GPU (K80, year 2014) the physical GPU (RTX1080, year 2017), as the performance is not directly comparable.

For the CPU-based resources, we chose instance types t2.micro and t2.small, These are very low-cost computing instances that support burstable CPU performance. Concerning software, we have used the same versions as the aforementioned physical equipment in all the VM.

4.1 Datasets

To perform the experiments we selected two tasks: a well-known benchmark and a real research problem. As benchmark dataset, we selected the CIFAR-10 (Krizhevsky, 2009) dataset that comprises 60k 32x32 colour images, with 10 different categories (6k images per category). The official split defines 50k samples to train the model and 10K to evaluate its accuracy predicting the correct category among the 10 possible. Figure 3 shows some examples from this dataset with their categories.

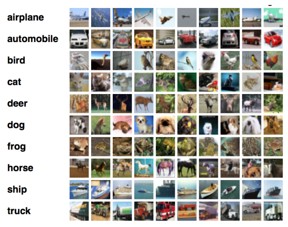

Figure 3: Images from CIFAR-10 with their categories (Krizhevsky, 2009).

We have selected this dataset since it is handy and complicated enough to require a non-trivial neural network model. As this is a dataset of images, we can use a convolutional neural network where GPUs, with their specialized hardware to process graphics, can take full advantage of the accelerated hardware.

Along with this well-known dataset, we have used an internal dataset provided by the "*Centro de Computação Gráfica* (CCG)"[7] from Portugal, in order to evaluate our software approach with an actual research problem. Our proposed deployment service was tested in a real use case documented in (Brito et al., 2018). This work is part of the AGATHA project[8] that consists of a solution providing image/video analysis for crime control and surveillance. For this collaboration, we have used a proprietary dataset provided by CCG, composed of pictures of faces, labelled with child, young, adult or elder classes, aiming to identify this features in a security surveillance system. Regarding the dataset, it is composed of around 65k RGB images with a resolution of 224x224. We train the model to predict the label among the set $\{child, young, adult, elder\}$. Some examples of these pictures are shown in Figure 4. It is composed of different sources, the subset that is shown was extracted from the website *Internet Movie Database* (IMDB)[9]. The 4.5GB dataset takes an average of 13 minutes to be retrieved from S3 to HDFS inside the Hadoop cluster.

Figure 4: Images from the provided dataset.

4.2 TensorFlow Code

We chosen an implementation provided in the TF official repository[10], using the version r1.10. We executed it without any change thanks to the Estimator API, introduced in the previous section, to illustrate that the code that follows this scheme can be trained inside this infrastructure with ease. For the same reason, we only included the required changes, just modifying the input pipeline, to adapt the age recognition model to be trained under this API. The model that these codes use is the convolutional neural network model known as Inception-Resnet-V1 (He et al., 2016). This model has 44 layers, millions of parameters, and it is based on one of the first huge deep learning models called Inception (Szegedy et al., 2015).

4.3 Results with CIFAR-10

First, we evaluated to decide whether to conduct asynchronous or synchronous training. Using a configuration with three GPU nodes, one parameter server and three worker nodes. We carried out both experiments and we show the results in Table 1, evaluating several

[6]https://aws.amazon.com/ec2/instance-types/
[7]http://www.ccg.pt
[8]http://www.agatha-osi.com/
[9]https://www.imdb.com
[10]https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10_estimator

iterations of the process. Regarding measuring, for these and the following experiments we have evaluated the number of global steps per second, that is, the number of batches processed per second and the average examples per second: number of instances that the model can process per second.

Table 1: Comparison of both training schemes: synchronous and asynchronous.

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

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

Regarding the single node experiment, we executed the training with different hardware configurations. We refer to the physical machines with the prefix `phys`, and `virt` to the virtual ones, while denoting the use of CPU/GPU adding the suffix `_cpu` or `_gpu`, respectively. In addition to global steps and average samples per second, we evaluated also the total time, that is, how long the training takes to converge. This is not always plausible considering that CPU based nodes can take a considerable amount of time. Table 2 shows the results for the comparison among single nodes. The * means that training did not fully converge and then an estimation of time to convergence is provided. We noticed the obvious performance boost that GPU provides with this kind of data. These problems are extremely resource-consuming for CPUs, discouraging any possible training using this hardware. It is important to remark again the difference between the physical GPU and the virtual one. In the next experiment we explore the options to parallelize the problem.

Next experiments involve the use of a distributed configuration. For doing this, we have deployed the infrastructure in AWS and run the experiments, taking the same measures than before. Table 3 summarizes the results. We divided accordingly the number of steps among the workers to add up to 60k, and values shown are related to the architecture as a whole.

Table 2: Single node results with different hardware

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

Table 3: Multiple node results with different hardware (G = G.step/sec, A = Avg.ex/sec.)

Machine	G	A	T.time
virt_cpu + virt_gpu x 2 (1 PS - 2 WN)	7,68±2,95	1259±395	˜92
virt_gpu x 3 (1 PS - 2 WN)	13,04±0,48	1687±80	˜41
virt_cpu + virt_gpu x 3 (1 PS - 3 WN)	10,04±5,59	2755±965	˜43

Some results show very high variance, this was the side-effect of using a lower-end instance type for the PS, such as `t2.small`. During the experiments with this kind of node, the global steps per second were decreasing gradually along iterations until they reach lower values than in the `virt_gpu` single training. Even if it seems counter-intuitive, using this fourth node as PS caused a bottleneck in the system. Figure 5 illustrates this fact, that we discovered after suspecting and discarding that HDFS latency to access to the data or the model was the cause. It seems that at a certain point, the resources of the CPU-based PS were exhausted, and then the performance got worse. Changing the type of instance solved the problem, as it can be seen in the improvement in the results using `virt_gpu x 3`, providing a stable global step/sec rate, at the expense of losing a GPU node to do the computationally intensive work.

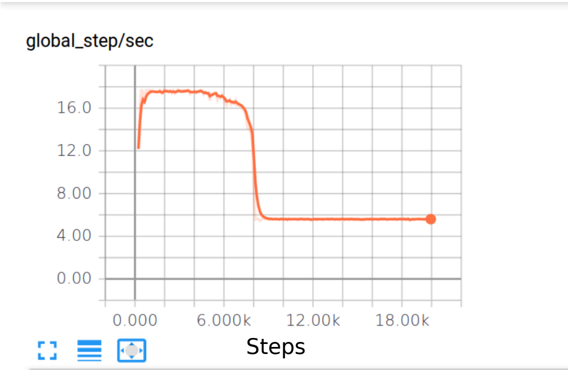

Figure 5: TensorBoard screenshot, showing the bottleneck in the middle of the training due to the use of low CPU-based instance.

Regarding flexibility, we performed the previous experiments again with the distributed configuration, using our deployment system with the IM and Ansible, changing parameters easily and interacting just with the RADL file, that define the architecture. Considering this, we found that the best combination is the use of three nodes (a PS and two WNs) that are GPU-based. In the following section we explore some of these setups with the computer vision research problem provided by the CCG.

4.4 Results with Age Recognition Task

Table 4 shows the results of the best systems after the evaluation considering `phys_gpu`, `virt_gpu x 3` and `virt_cpu + virt_gpu x 3`, the setups that provided the best result among virtualization systems.

Table 4: Single physical node and multiple node results on the Age dataset, with different hardware.

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

Concerning the storage, we did not perceive any important latency caused by the HDFS system neither in CIFAR-10 nor in the faces dataset. Once the dataset is uploaded, we have performed experiments with and without distributing data and model and not noticing any remarkable difference. This may happen with massive datasets since they require more complicated management. In terms of the introduced latency, the dynamic deployment's latency of the virtual infrastructure, although unavoidable, will be absorbed by the training when a real time-consuming task is carried out, that could imply hours or days of training. The advantages of performing all the steps without human intervention represent a significant step forward. It usually took between 15-20 minutes to deploy the configuration with a `t2.small` and three `p2.xlarge`. In summary, the results indicate that using automated deployment on the Cloud to take advantage of GPU-based nodes is a convenient approach for cost-effective training, as you can easily match your needs with your budget, enabling the access to both basic units and high performance GPUs.

4.5 Discussion

Increasing the throughput of the training of deep neural networks have become mandatory to solve the challenging tasks nowadays. Parallelizing training using GPUs seems the way to go, since the proposal in (Krizhevsky et al., 2012), where the process of sharing a model between GPUs in order to accelerate the training was done manually with a lot of human effort. This has changed and now it is very easy, as shown in this work, to code the training using TF to distribute the training effort among different distributed GPUs. Regarding this, we have shown how to deploy the required infrastructure in a public Cloud provider to get the most out of the functionality that this framework provides.

The parallel training scheme suits well in several problems, as it is shown in (Dean et al., 2012), but there are more approaches, such as Elastic Averaging SGD (Zhang et al., 2015). This work proposes a distributed optimization scheme designed to reduce communication overhead with the parameter server. Ideas in that direction are concerned with the function that the parameter server computes in order to favour the convergence of the training. This work aimed to develop the architecture and the deployment, as well as implementing a generic distributed training, so these kind of evaluations regarding training convergence are out of the scope of this work.

We have decided to focus on data parallelism as our motivation was to provide the DL pipeline for problems that leverage this kind of scheme, i.e.: several instances that are distributed in a data cluster, but there are tasks where this is not enough. Among these tasks, we can consider machine translation where models, or the composition of them, are huge. This is the problem that authors faced in (Wu et al., 2016), where they used model parallelism as well as data parallelism. For these kinds of models, there is no alternative for training them in a reasonable time.

5 CONCLUSIONS

In this work we provided and evaluated a deployment system to provision, configure and execute a distributed TF training pipeline in an unattended way, based on cluster computing in the Cloud. Using the IM and Ansible, we can modify our architecture and train easily just by modifying a configuration file and being able to deploy on different Cloud providers.

Regarding software, we have used the state-of-the-art tools, such as TF and CUDA, performing experiments with two different problems, CIFAR-10 and the Age recognition problem. We have shown that TF scripts can be adapted easily if they are using the Estimator API, a high-level API that eases the development of deep learning models.

Concerning the results, we are close to the baseline with the physical GPU, even considering that it is two generations ahead of the virtual GPUs that we used. However, we achieved a good performance according to the price of these instances. The present approach democratizes the access to cost-effective distributed computing based on TF on the Cloud to perform training of deep learning models.

Future work includes using massive datasets to identify the bottlenecks of the proposed approach. Also, more performant instance types were not used due to budget constraints. This goes inline with using the `p3` family of instances types that feature more than one GPU per node, to evaluate in-node parallelization and between-nodes parallelization jointly.

ACKNOWLEDGEMENTS

The authors would like to thank the Spanish "Ministerio de Economía, Industria y Competitividad" for the project "BigCLOE" with reference number TIN2016-79951-R. The development of this work has been partially supported by the project "AGATHA: Intelligent Analysis System for surveillance/crime control on open information sources" (project No. 010822), financed by the European Regional Development Fund (ERDF) through COMPETE 2020 - Operational Program for Competitiveness and Internationalisation and Lisbon 2020 - Lisbon Regional Operational Program 2014-2020.

REFERENCES

Abadi, M. et al. (2016). Tensorflow: a system for large-scale machine learning. In *Proc. of OSDI, 2016*, pages 265–283.

Amazon SageMaker (2018). [online-Feb.2021] https://aws.amazon.com/sagemaker/.

Ansible (2018). [online-Feb.2021] https://www.ansible.com/.

Bannon, D. et al. (2018). Deepcell 2.0: Automated cloud deployment of deep learning models for large-scale cellular image analysis. *bioRxiv*, page 505032.

Brito, P. et al. (2018). Agatha: Face benchmarking dataset for exploring criminal surveillance methods on open source data. In *Proc. of ICGI, 2018*, pages 1–8. IEEE.

Caballer, M. et al. (2015). Dynamic management of virtual infrastructures. *Journal of Grid Computing*, 13(1):53–70.

Cloud AI (2018). [online-Feb.2021] https://cloud.google.com/products/ai/.

Dean, J. et al. (2012). Large scale distributed deep networks. In *Proc. of NIPS, 2012*, pages 1223–1231.

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

European Grid Infrastructure (2018). [online-Feb.2021] https://www.egi.eu.

Hadoop (2018). [online-Feb.2021] http://hadoop.apache.org.

He, K. et al. (2016). Deep residual learning for image recognition. In *Proc. of CVPR, 2016*, pages 770–778.

Infrastructure Manager (2018). [online-Feb.2021] http://www.grycap.upv.es/im/.

Jorge Cano, J. (2019). Entrenamiento escalable de modelos de deep learning sobre infraestructuras cloud. [online-Feb.2021] https://riunet.upv.es/handle/10251/115454.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

Krizhevsky, A. et al. (2012). Imagenet classification with deep convolutional neural networks. In *Proc. of NIPS, 2012*, pages 1097–1105.

Lee, S. et al. (2018). TensorLightning: A Traffic-Efficient Distributed Deep Learning on Commodity Spark Clusters. *IEEE Access*, 6:27671–27680.

Luckow, A. et al. (2016). Deep learning in the automotive industry: Applications and tools. In *Proc. of Big Data2016, 2016*, pages 3759–3768. IEEE.

MXNet (2018). [online-Feb.2021] https://mxnet.apache.org.

PyTorch (2018). [online-Feb.2021] https://pytorch.org.

Reaño, C. et al. (2015). Local and remote gpus perform similar with edr 100g infiniband. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*, page 4. ACM.

Rumelhart, D. E. et al. (1985). Learning internal representations by error propagation. Technical report, Institute for Cognitive Science, California.

Sergeev, A. and Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.

Shvachko, K. et al. (2010). The hadoop distributed file system. In *Proc. of MSST, 2010*, pages 1–10. Ieee.

Silver, D. et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

Szegedy, C. et al. (2015). Going deeper with convolutions. In *Proc. of CVPR, 2015*, pages 1–9.

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

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

Wu, Y. et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

Zhang, S. et al. (2015). Deep learning with elastic averaging sgd. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Proc. of NIPS, 2015*, pages 685–693. Curran Associates, Inc.