Distributed Deep Learning Architectures for Commodity Clusters

Maria Aspri

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Professor *Panagiotis Tsakalides*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Distributed Deep Learning Architectures for Commodity Clusters

Thesis submitted by
Maria Aspri
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Maria Aspri

Committee approvals: _____
Panagiotis Tsakalides
Professor, Thesis Supervisor

_____
Maria Papadopouli
Professor, Committee Member

_____
George Tzagkarakis
Principal Researcher, Committee Member

Departmental approval: _____
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, May 2019

Architectures of Distributed Deep Learning on Commodity Clusters

Abstract

For the last few years, Deep Learning, is becoming an important tool in many computational applications, having trivialized the whole pipeline of feature extraction and, as a result, replacing other popular Machine Learning algorithms. For Deep Learning to be effective though, it needs not only access to vast amounts of data, but also devices with high computational performance. At the same time, commodity computers, with their high availability and low cost, are a popular choice of hardware and thus widely used, both in industry as well as in academia. However, they lack not only the required space for storing large volume datasets, but also the computational capacity to make Deep Learning a viable choice. In order to address this issue, the solution of commodity clusters was proposed.

The main goal of the present thesis is the study and application of Distributed Deep Learning techniques, through the scope of both data and model parallelization, aiming to effectively migrate Deep Learning on commodity hardware. Our objective is the best possible management of the available Cluster resources, as well as to study and exploit the impact of distributed environments on the performance of Deep Learning algorithms. We conducted experiments on a five node CPU commodity cluster, and present the results of our research in the form of two case studies on the major research fields of cosmology and remote sensing.

In the first case study, we address the problem of spectroscopic redshift estimation in astronomy, through a distributed perspective. We perform data distribution techniques in order to study the performance of a Convolutional Neural Network, considering both the number of training nodes and data distribution, while quantifying their effects via the metrics of training accuracy and training loss.

In the second case study, we examine a research topic in the field of remote sensing. Our aim is to effectively split a multimodal Convolutional Neural Network used for multi-class land cover classification, that has a high number of parameters. Through model splits, we succeeded in effectively sharing the load of a Neural Network between the workers of our cluster and thus optimize CPU usage. We also managed to decrease the network traffic that happens due to frequent data transfers among the machines.

Αρχιτεκτονικές Κατανεμημένης Εμβριθούς Μάθησης για Συστάδες Μηχανημάτων Περιορισμένων Πόρων

Περίληψη

Τα τελευταία χρόνια, η Εμβριθής Μάθηση έχει αναχθεί σε δομικό συστατικό για πάρα πολλές υπολογιστικές εφαρμογές, έχοντας καταφέρει να αυτοματοποιήσει την διαδικασία παραγωγής γνωρισμάτων και συνεπώς να εκτοπίσει άλλες, πιο συμβατικές τεχνικές Μηχανικής Μάθησης. Για να είναι αποδοτική, η Εμβριθής Μάθηση πρέπει να έχει πρόσβαση τόσο σε δεδομένα μεγάλου όγκου, όσο και σε μηχανήματα με μεγάλα αποθέματα υπολογιστικής ισχύς και μνήμης. Παράλληλα, τα μηχανήματα χαμηλών υπολογιστικών πόρων είναι ευρέως διαδεδομένα τόσο στην έρευνα όσο και στη βιομηχανία, κυρίως λόγο της άμεσης διαθεσιμότητας τους αλλά και του χαμηλού κόστους τους. Όμως τέτοια υπολογιστικά συστήματα δεν έχουν ούτε τον χώρο που απαιτείται για την αποθήκευση δεδομένων μεγάλου όγκου, ούτε τα αποθέματα μνήμης για να εκτελέσουν αποτελεσματικά εμβριθή μοντέλα. Μια πρόταση για την αντιμετώπιση αυτού του προβλήματος, είναι η μεταφορά αυτών των μοντέλων σε κατανεμημένα περιβάλλοντα, με την ομαδοποίηση μηχανημάτων περιορισμένων πόρων σε Συστάδες.
Ο βασικός στόχος αυτής της μεταπτυχιακής εργασίας είναι η παρουσίαση και η εφαρμογή τεχνικών Κατανεμημένης Εμβριθούς Μάθησης, σε επίπεδο τόσο δεδομένων όσο και αρχιτεκτονικών, με σκοπό να να εκμεταλλευτούμε στο μέγιστο βαθμό τους διαθέσιμους πόρους μιας Συστάδας, αλλά και να αξιοποιήσουμε την επίδραση που μπορεί να έχει ένα τέτοιο κατανεμημένο περιβάλλον στην συμπεριφορά συνελικτικών μοντέλων.
Παρουσιάζουμε τα αποτελέσματά της μελέτης μας με την μορφή περιπτωσιολογικών μελετών σε δύο σημαντικά ερευνητικά πεδία, εκείνο της κοσμολογίας και αυτό της τηλεπισκόπησης, χρησιμοποιώντας μια Συστάδα πέντε μηχανημάτων με περιορισμένους πόρους. Στην πρώτη μελέτη, χρησιμοποιούμε τεχνικές κατανομής δεδομένων ώστε να εξετάσουμε την απόδοση ενός συνελικτικού νευρωνικού δικτύου που χρησιμοποιείται για εκτίμηση ερυθρής μετατόπισης, ως προς των αριθμό των μηχανημάτων μιας Συστάδας αλλά και ως προς τον τρόπο που μοιράζονται σε αυτά τα δεδομένα. Στην δεύτερη μελέτη, προσπαθούμε να σπάσουμε ένα διακλαδωμένο συνελικτικό μοντέλο με πολλές παραμέτρους μεταξύ των μηχανημάτων της Συστάδας, με σκοπό την ταξινόμηση κάλυψης γης. Στόχος μας μέσω της εφαρμογής αυτής είναι να πετύχουμε βέλτιστο διαμοιρασμό φόρτου υπολογιστικής ισχύς, αλλά και να ελαττώσουμε την κίνηση του δικτύου που προκύπτει λόγω της συχνής μεταφοράς δεδομένων μεταξύ των υπολογιστικών συστημάτων.

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επόπτη μου, καθηγητή του τμήματος και αντιπρύτανη κύριο Παναγιώτη Τσακαλίδη, του οποίου οι συμβουλές και η καθοδήγηση με βοήθησαν να φτάσω στο σημείο που βρίσκομαι σήμερα. Σας ευχαριστώ πολύ που πιστέψατε στις ικανότητές μου και μου δώσατε μια θέση στην την ερευνητική σας ομάδα, ακόμα και ως προπτυχιακή φοιτήτρια!

Καμία λέξη δεν είναι αρκετή για να μπορέσω να ευχαριστήσω τον άμεσο επιβλέποντα μου, δόκτωρ Γρηγόριο Τσαγκατάκη, που χωρίς την καθοδήγησή του αυτή η εργασία δεν θα είχε ολοκληρωθεί. Η συνεχής ενθάρρυνση του, οι εποικοδομητικές του ιδέες, η βοήθειά που μου προσέφερε και κυρίως η υπομονή που έδειξε όλον αυτό τον καιρό, ήταν πηγές έμπνευσης για εμένα και με οδήγησαν στο να καταβάλλω το μέγιστο των δυνατοτήτων μου. Γρηγόρη, ήταν χαρά μου να συνεργαστώ μαζί σου. Ευχαριστώ πολύ για όλα. Να είσαι καλα!

Δεν θα μπορώ να μην αναφερθώ στην Αθανασία Πανουσοπούλου, οι συμβουλές της οποίας με βοήθησαν αρκετά καθ΄ όλη τη διάρκεια του μεταπτυχιακού μου. Θα ήθελα επίσης να ευχαριστήσω το ίδρυμα Τεχνολογίας και Έρευνας ($FORTH - ICS$) για όλους τους πόρους που μου παρείχε τα τέσσερα αυτά χρόνια που με φιλοξένησε.

Θα ήθελα επίσης να πω ένα μεγάλο ευχαριστώ, στους συναδέλφους και στους φίλους που απέκτησα στη σχολή και στο εργαστήριο. Παιδιά, ευχαριστώ πολύ που μου σταθήκατε, στα εύκολα και στα δύσκολα, και που πάντα είσασταν εκεί όταν σας χρειαζόμουν. Μου χαρίσατε μερικά από τα πιο όμορφά μου χρόνια.

Τέλος θα ήθελα να πω ένα απέραντο ευχαριστώ στους γονείς μου, Παναγιώτη και Κατερίνα καθώς και στην πιο γλυκιά γιαγιά του κόσμου τη Φανή, για την ανιδιοτελή αγάπη και υποστήριξη που μου προσέφεραν σε όλη τη μέχρι τώρα ζωή μου. Σας αγαπώ πολύ!

Στην οικογένειά μου

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep Learning [1] is a field slowly taking over a variety of aspects in our everyday lives. It can be summed up as a sub field of Machine Learning, studying models named Deep Neural Networks (DNNs). These networks are able to learn complex and hierarchical representations from raw data, unlike other conventional hand crafted models, such as Support Vector Machines (SVMs), which require extracted features as data preprocessing. When trained properly, DNNs are able to provide extremely accurate results for difficult problems with high computational complexity, simply by observing large amounts of data. As a result, Deep Learning development tools, libraries and languages have been found their way into a plethora of applications, ranging from image classification [2] through speech recognition [3] and health-care [4], to autonomous driving [5] and finance predictions [6]. Even though Neural Networks have been gaining attention since 1980s with the invention of backpropagation [7], their rise into prominence was tightly coupled to the available computational power, which allowed to exploit their inherent parallelism. With the arrival of the current decade, rapid technological advancements in processing power, memory, and storage, as well as the drastic increase in the amount of available data gave a significant boost in the performance of various deep learning architectures, going as far as establishing DNNs as state-of-the-art in many research fields.
However, it has been observed that as datasets increase in size and DNNs become more deep and complex, the computational intensity and memory demands of Deep Learning increase proportionally. These computational resources most of the time are not available on commodity setups, therefore, in order to address this issue, distributed approaches have been proposed. A lot of research has been conducted, and it was observed that training a DNN to competitive accuracy nowadays essentially requires a cluster of machines, preferably with high-performance computing architectures. To harness the computational power available in such systems, different aspects of training and inference of DNNs have been modified to increase their underlying concurrency.

In this work, we study methodologies of Distributed Deep Learning on commodity clusters and explore their performance on case studies on the research fields of cosmology and remote sensing. More specifically, we consider the problem of spectroscopic redshift estimation through a distributed data approach, and we also address the problem of a very deep multimodal neural network used for land cover classification.

1.1 Need for Parallel and Distributed Deep Learning

Deep Learning is considered state-of-the-art in many fields and has a plethora of practical applications in many industries. However training a Deep Artificial Neural Network can be a fairly demanding task, as there exist several issues to address.
First of all, there are many industry datasets consisted of many terrabytes or even petabytes of data that need to be processed by a learning model in order to achieve better performance and be more accurate. Naturally, these datasets are too large to be stored on commodity machines, and even in cases that storage is not an issue, those amounts of data will exponentially increase training times.
Another issue emerges in the form of parameter storage. Very deep models are made of hundreds of layers, and thus there are millions of parameters which define them. Consequently, it requires several GB to store these models. This would normally not be an issue if a model could be stored on hard drives, but when an Deep Learning application is running, models need to reside in memory which is usually much smaller in size. This renders many deep models as too big to fit into most single machines, and need to be split across multiple devices.
A final issue to be addressed is the high computational power required to train a Neural Network. The huge amounts of model parameters along with large volumes of data, make training a Neural Network a computationally intensive process which takes a lot of time. Typically, it takes hours or even days to train a Deep Neural Network on a single machine, even when multiple Central Proceccing Units(CPUs) or Graphical Processing Units(GPUs) are available. For example, if we wanted to train a VGG [8] model on a single machine it would take up to 10 hours(roughly), assuming that we have a CPU of 8 cores [9].
In conclusion, deep learning can be elevated as a high performance computing problem. We need high computational power along with efficient data processing and storage. Therefore it is important to come up with parallel and distributed algorithms which can run much faster and can drastically reduce training times on commodity clusters.

1.2 Distributed Deep Learning Challenges

Despite the fact that Deep Neural Networks excel at solving optimization problems with large datasets and millions of variables, only the past few years researchers

have succeeded in migrating them in distributed computing. As a result, the corresponding field is subjected to rapid development, with contributions from diverse research communities. At the same time, we see a number of open source DL frameworks and orchestration systems emerging. However, Distributed Deep Learning is by no means a trivial task , as it is often performed in a distributed infrastructure of multiple compute nodes. This introduces a number of challenges. Parameter synchronization is required for a distributed Neural Network to function properly, therefore challenges on how and when to synchronize parameters are a hot research topic. Another issue regarding parameter synchronization, is how to minimize communication overhead for synchronization. One more issue emerges from the fact that both training and model data need to be handled in a suitable manner, while taking into account the available distributed infrastructure, the running training processes and the resource scheduling in the data center. Finally, an additional challenge in distributed Deep Learning is known as the scheduling problem, that is how to map the parallel training processes to the processing nodes in a distributed infrastructure.

1.3 Motivation

Although there are works on literature that report on parallel optimization algorithms and distributed framework comparisons, there is limited research that explores how a distributed environment affects the performance of a Neural Network and in which way those environments can be exploited in order to make faster and more accurate predictions. Furthermore, most distributed deep learning applications omit the part of parameter storage and focus only on Big Data scenarios and data distribution schemes, ignoring the fact that there are many instances where model split can be tremendously useful, such as Edge Computing and Cloud Computing applications. Moreover, there is a significant lack of use case analyses on deep learning for distributed environments.
In this work we aim to fill this gap by introducing two use cases on distributed deep learning on the fields of astrophysics and urban land classification. Each use case executes a different parallelization scheme, and tries to improve cluster utilization and network convergence without compensating on the quality of the predictions.

1.4 Contributions

This thesis contributes to the Deep Learning community by exploring different aspects of Distributed Deep Learning schemes. Furthermore we explore the performance of Distributed Deep Learning in an constrained environment that confronts severe bottlenecks regarding execution times, such as network traffic, limited memory and total absence of Graphical Processing Units.
Our first contribution is a use case study on Astrophysics. We use a state-of-the-art

deep learning model attuned to perform redshift estimation and explore its performance on a cluster of machines. For this use case we adopt a distributed data approach in order to address the cluster's behavior. We report on the performance of a CPU cluster, considering both the number of training nodes and data distribution, while quantifying their effects via the metrics of training accuracy and loss. This thesis further contributes to the Deep Learning domain with a second case study on multi-class land cover classification through a multimodal disrtibuted neural network. In this analysis we explore the characteristics and performance of various model paralelization schemes on a CPU commodity cluster. We design several distributed model architectures of the initial Neural Network and deploy them on our cluster. We compare the performance of our proposed architectures and report on their properties, in terms of network traffic, memory consumption and CPU usage.

1.5 Scope

The remainder of this thesis is structured as follows. Chapter 2, presents a brief outline of the relevant literature concerning the state-of-the-art of Distributed Deep Learning. A detailed overview of the existing theoretical background, models and techniques as well as the distributed frameworks adopted in this work, is provided in Chapter 3. Chapter 4 presents the setup of our Cluster as well as our adopted learning schemes. In Chapters 5 and 6, we focus on the two examined case studies, regarding the spectroscopic redshift estimation and land cover multi-class classification. We present the datasets used, analyze the proposed frameworks, demonstrate and evaluate our experimental results. Finally, in Chapter 6 we give concluding remarks and extrapolate potential future work.

Chapter 2

Related Work

Graphical Processing Units are the ideal commodity hardware to do Deep Learning on, since they were developed to deal with parallel computations and have large memory bandwidth. Despite the significant advances in terms of hardware capabilities, the fact remains that single machine setups do not posses sufficient computational resources to efficiently train complex DNNs. Therefore, a sufficient amount of research has been conducted on enabling distributed training on neural networks[10, 11, 12]. To address this limitation, different philosophies regarding the distributed training of DNNs have been considered, the most popular being model parallelism [13] and data parallelism [14].

In *model parallelism*, different machines are responsible for the computations in

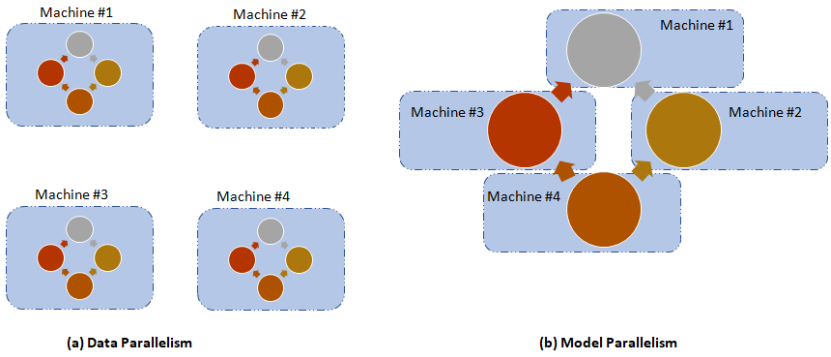


Figure 2.1: Neural Network Parallelism Schemes

different layers of the network. In this case, the weights are split equally among the machines, and all of them train on the same mini batch.

Model parallelism requires the generated output of each layer to be stacked, in order to provide the input for the next layers. As a result, the architecture of a DNN creates layer interdependencies, which, in turn, generate communication overheads that highly affect overall performance. In order to rectify that, the authors in [15] propose a method that introduces redundant computations to neural networks in

which each processor will be responsible for twice the neurons, and thus would require more computations but less communications. An other way to reduce communication overheads is with the use of Cannon's matrix multiplication algorithm [16], but it only produces better efficiency and speedups over simple partitioning on small-scale fully connected networks.
A second form of model parallelism can be seen as the replication of the elements of a neural network. The authors in [17] introduce Treenets, groups of separately trained networks called ensembles, whose results are averaged rather than their parameters. They propose ensemble-aware loss functions and backpropagation techniques, and the training process is parallelized across the network copies, assigning each copy to a different processor.
In *data parallelism* every machine receives a complete copy of the model and its parameters, called a replica. The dataset is distributed among the machines through different partitions, which train the replicas locally. Each replica must use the same weights but trains on different batches, which means that in the weight update phase the results of the partitions have to be averaged to obtain the global gradient. One of the earliest occurrences of mapping DNN computations to data parallel architectures were performed by the authors in [18], where they mapped the unsupervised training procedure to GPUs by running minibatch SGD. Today, data parallelism is supported by the vast majority of deep learning frameworks, through multiple GPUs, or a cluster of nodes [19].

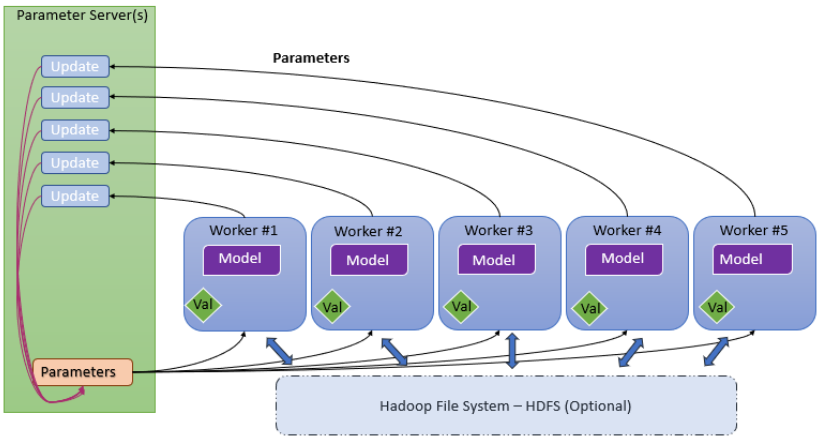


Figure 2.2: Asynchronous Scheme Architecture in Data Parallelism.

The most straightforward strategy for data parallelism is to partition the work of the dataset samples among multiple computational resources, either cores or different devices. Therefore, datasets too large to be stored on a single machine, use this method to achieve faster training [20, 21]. As stated before, Data parallel approaches keep a copy of the entire model on each machine, while processing different subsets of the training dataset, and require some method of combining results

and synchronizing the model parameters between workers. The prominent examples of parameter synchronization involve synchronous and asynchronous training, while coordination is facilitated by a worker amply named Parameter Server (PS) [22]. Synchronous methods [23] require all replicas to update their parameters at the same timestamp. By doing so, the batch size is effectively multiplied by the number of replicas. Many synchronous approaches for data parallelism were proposed in literature. In ParallelSGD [24], SGD is run k times in parallel, dividing the dataset among k processors. After the convergence of every SGD instance, the resulting weights are aggregated to obtain a global averaged weight vector. MapReduce [25] has also been used in many distributed deep learning implementations [26, 27], due to its easy to schedule parallel tasks. However, despite the overall successful first results, it generality hindered DNN-specific optimizations. More recent implementations however, make use of high-performance communication interfaces to implement parallelism features, such as reducing latency via Asynchronous methods [28]. Asynchronous executions can potentially provide higher throughput, since machines spend more time performing computations, instead of waiting for the parameter averaging step to be completed [29].

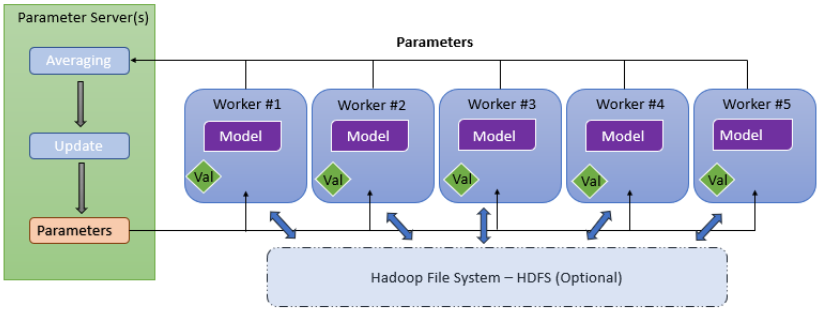


Figure 2.3: Synchronous Scheme Architecture in Data Parallelism.

It should also be noted that there works in literature that propose hybrid paralellism approaches. The general idea behind hybrid parallelism is to combine different parallelization schemes in order to overcome the drawbacks of each other. The authors in [30] perform a hybrid scheme on AlexNet [31], in which they apply data parallelism to convolutional layers, and model parallelismn to fully connected parts. The work presented in [28] proposed an asynchronous implementation of DNN training on CPUs, which uses an intermediate representation to implement fine-grained hybrid parallelism. Finally, in [32] distributed training is performed simultaneously on multiple model replicas, while each replica is trained on different data samples.

Chapter 3

Terminology and Theoretical Background

This chapter first describes the anatomy of a Convolutional Neural Network. It presents popular layer types and their properties, followed by a section dedicated to data fusion and multimodal Neural Networks. In the next section, we present the distributed frameworks adopted in this work. Firstly, we introduce Apache Spark, a popular platform for Big Data analysis that favors iterative computations. Then we present TensorFlowOnSpark, a distributed framework that combines the salient features from TensorFlow with Apache Spark clusters, and enables distributed deep learning on a cluster of GPU and CPU servers.

3.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a class of deep, feed-forward neural networks, most commonly applied on image classification [33]. A notable contribution, CNNs were initially designed to recognize relatively simple visual patterns, such as handwritten characters. Since then, CNNs have become extremely popular with applications on image and video recognition, recommended systems [35], and natural language processing[36]. A typical architecture of a CNN is provided in Figure 3.1. CNNs differ from the other Neural Networks in that they are structured in a locally connected manner, based on the assumption that neighboring regions of each observation have a higher chance to be correlated than regions located farther away. As a result, the number of trainable parameters is significantly reduced, thus rendering the network less prone to overfitting.
In the following subsections we present the basic components of a typical CNN, which demonstrate its structural and functional properties.

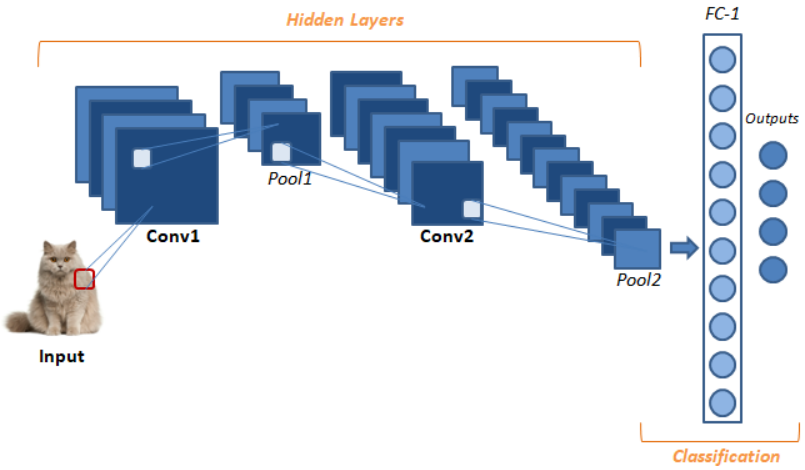


Figure 3.1: Example Architecture of a 2D CNN.

3.1.1 Feature Extraction

3.1.1.1 Convolutional Layers

Convolutions constitute the majority of computations involved in training and inference of Convolutional Neural Networks. As a result, both academia and industry have invested a considerable amount of effort for their optimization. A spatial convolution is defined by the number of filters (e.g. number of output channels), the properties of the filters (e.g. filter width) and the properties of the convolution itself (e.g. stride). Convolutional Layers take advantage of the fact that their inputs exhibit many spatial relationships. As a result, a two dimensional convolutional layer learns a set of N_k filters, convolved spatially with an input x, to produce a set of N_k 2D features maps z:

$$z_k = f_k * x$$

When a filter correlates well with a region of the corresponding input, the resulting response in the feature map will be strong. In these layers, the weights are shared over the entire input, thus reducing the number of parameters per response.

3.1.1.2 Activation Layers

CNN models offer limited capacity of forming more complex representations of input data, therefore a non-linearity needs to be introduced, enabling the network to act as a universal function approximator. Activation Layers are the ones responsible for filling this role, and are usually applied directly after each convolutional layer. An activation function takes as input a vector A and performs a fixed pointwise operation on it. The most popular activation functions are the ones described below:

- **Sigmoid**
  This activation function takes a real value and presses it between 0 and 1, although as it saturates at either tail of 0 or 1, the gradient at these regions is almost zero. In those cases, the backpropagation algorithm fails at parameter modifying and preceding those parameters to the next layers. Sigmoid has the following mathematical form:
  $$y = \sigma(x) = \frac{1}{1 + e^{(-x)}}$$
- **Hyperbolic Tangent**
  This activation function takes a real value and presses it between -1 and 1, but has the same drawbacks as the Sigmoid activation function. It's mathematical form is the following:
  $$y = 2\sigma(2x) - 1$$
- **Rectified Linear Unit** (ReLU)
  ReLU has become extremely popular in the few past years, since it involves cheap computational operations, compared to the expensive exponentials of other functions such as Sigmoid. Moreover, due to its linearity, it does not suffer from the vanishing gradient of the aforementioned functions. However, ReLU does not taking into account negative information. It's mathematical form is the following:
  $$y = max(0, x)$$

3.1.1.3 Pooling Layers

A Pooling Layer's objectives are firstly, to provide invariance to slightly different inputs and secondly, to reduce the dimensionality of feature maps. These layers are usually introduced between subsequent convolutional and activation layers. Pooling can be described mathematically as:

$$p_R = P_{i \in R(Z_i)}$$

where P is a pooling function over the region of pixels R. Max Pooling is most of the time, the preferable method of choice, as it avoids cancellation of negative elements, and prevents blurring of activations and gradients throughout the neural network. A Pooling Layer is defined by its aggregation function, the size of the area where it is applied(e.g. height, width), and the properties of the convolution itself (e.g. padding).

3.1.2 Classification

3.1.2.1 Fully Connected Layers

A Fully Connected layer is responsible for deducing valid predictions, while taking into account the input observations. It is a function which applies linear transformations on vector inputs of dimension C and produces vector outputs of dimension

D. These layers also have a bias parameter, b. Mathematically, Fully Connected layers can be expressed as:

$$y = Ax + b$$

$$y_i = \sum_{j=1}^{C} (A_{i,j}, x_j) + b_i$$

Multiple Fully Connected layers can be stacked together, to compose even deeper architectures. In those cases, the final classification layer is composed with as many output units as the number of classes of the addressed problem. A probabilistic activation function should also be employed, usually in the form of Softmax Regression. After applying this activation function, each component will be in the interval (0, 1), and the components will add up to 1, so that they can be interpreted as probabilities. This property, makes softmax an exceptional choice for the problem of multi-class classifcation.

3.1.3 Regularization

3.1.3.1 Batch Normalization Layers

Batch Normalization layers can be accounted more as normalizers, however they have been shown to work very effectively as regularizers [38]. These layers add a normalization step to the network, which make the inputs of each trainable layers comparable across features. By doing this, they ensure a high learning rate while keeping the network learning. The main intuition behind batch normalization lies in the fact that, the more neural network deepens, the higher the probability that the neuronal activations of intermediate layers might diverge significantly from desirable values. Batch Normalization is becoming extremely popular in Deep Learning, since in many cases helps the network converge faster and lead to an overall higher accuracy.

3.1.4 Dropout Layers

One of the most popular techniques in CNNs, Dropout Layers can help narrow down the effects of overfitting, since they can be used to temporarily decrease the total parameters of the network at each training iteration. All the neurons in the network are associated with a probability p and each neuron can be temporarily dropped from the network, along with its connections, according to that probability. For each training iteration a random portion of the original network is dropped, leading to smaller variations of its initial structure, as the value of p gets higher. In the testing phase, dropout is not applied at all.

3.2 Data Fusion and Deep Neural Networks

A lot of machine learning models have been implemented with a focus on a single type of data, e.g. image or text. However, real world data usually comes with different modalities. A modality refers to the way something happens or is experienced, and a research problem is characterized as multimodal when it includes multiple modalities, characterized by different statistical properties. At first sight, fusing different modalities for improving the performance of a learning approach seems appealing, since signals from different modalities often carry complimentary information about the same event. In reality though, it is a hard task due to practical challenges such as varying levels of noise and conflicts between modalities.

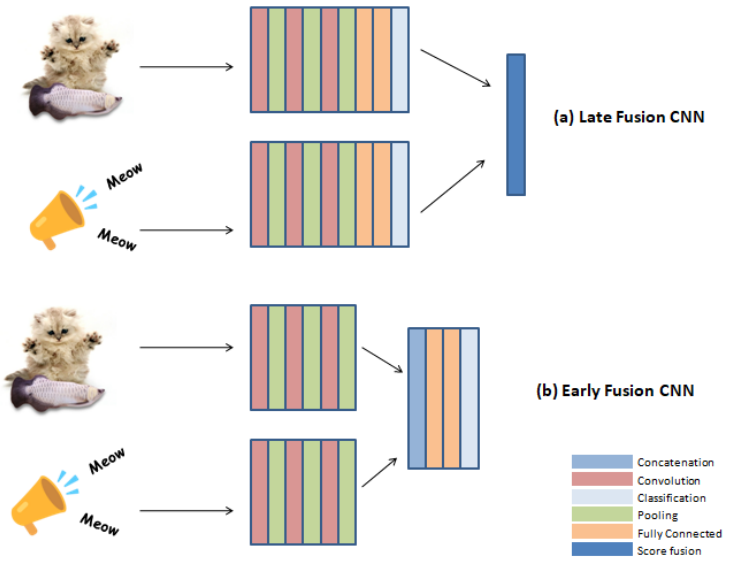


Figure 3.2: Visual representation of early and late fusion.

The general idea of multimodal fusion in DNNs is to perform the fusion in a joint hidden layer of a neural network. One of the most accepted categorization of multimodal fusion is to split it into two distinct categories: early and late fusion [39]. On a machine learning model, early fusion is performed after the feature extraction step and the creation of feature vectors. Feature vectors from different modalities exhibit different characteristics of a same pattern, and combining those features keeps effective discriminant information while eliminating redundant information [40]. As a result, early fusion can exploit the correlation and interactions between low level features of each modality. Another advantage of early fusion is than it requires the training of a single model, making the training pipeline easier compared to late fusion. On the other hand, late fusion approaches train a separate classifier for each modality present in a dataset. Afterwards, the individual classification scores are fused into a final classification score [41, 42]. Despite the

obvious increased computational burdens, these approaches make it easier to make predictions when some of the modalities are missing, and also allow for more flexibility, as different predictors can model each individual modality better.
Deep Neural Networks have been used extensively for the task of multimodal fusion, mainly to fuse information for visual and media question answering [53], gesture recognition [52], and video description generation [54]. A big advantage of DNNs in data fusion is their capacity to learn from large amount of data. As a result, they show exceptional performance, being able to learn complex decision boundaries that other classifiers struggle with [43]. However, DNNs are difficult to decipher, since it is hard to tell which modalities or features play an important role on the final predictions. Furthermore, on scenarios that don't have large training datasets available their performance leave a lot to be desired.

3.3 Distributed Frameworks

3.3.1 Apache Spark

Apache Spark is the mainstream technology for in-memory analytics [44] over commodity hardware. Spark extends the MapReduce model [25] by the use of an elastic persistence model, which provides the flexibility to persist these data records, either in memory, on disk, or both. Therefore, Spark favors iterative processes met in machine learning and optimization algorithms. Briefly, Spark organizes the underlying infrastructure into a hierarchical cluster of computing elements, comprised of a master and a set of N workers. The master is responsible for the configuration of the cluster, while the workers perform the learning tasks submitted to them through a driver program, along with a part of an initial dataset. The partitioning of the dataset relies on the concept of the Resilient Distributed Dataset (RDD), which is defined as a read-only collection of data records. Through the driver, an application program has control over the initial data parallelization into RDDs, and can also apply transformations on existing RDDs. These transformations are lazy, meaning that RDDs are only computed after an *action* is performed. Actions are operations that return a value to the application program or export data to a storage system. As a result, a set of pipelined transformations of an RDD will not be executed until an action is commanded.

The execution of a Spark application program is performed in three distinct phases: (a) the configuration of the operational parameters, (b) the parallelization of the dataset into RDDs, and (c) the assignment and execution of the learning tasks. During phase (b), data blocks needed for computations are deployed on the cluster through the driver throughout the lifetime of the application program. Phase (c) fires with the request of performing a learning task in the form of an action on an RDD. Afterwards, the Task Manager service assigns the execution of the learning task to the workers, in the form N stages. The result of a stage returns back to the driver program, and the Task Manager assigns another stage of the same learning task, until all stages have been completed. This procedure

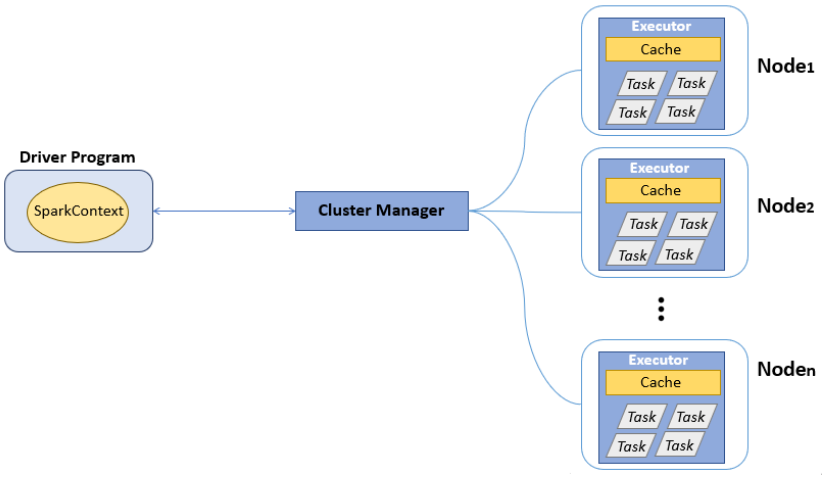


Figure 3.3: Apache Spark Standalone Deployment.

is repeated for the remaining learning tasks, until all stages have been completed. As stated before, results can either stay on the driver or be exported to a storage system.

3.3.2 TensorFlowonSpark

The distributed DNN architecture considered in this work is based on TensorFlowOnSpak [1] [45], for managing TensorFlow-based distributed DNN workflows on Apache Spark clusters. Opposed to its counterparts (e.g., Distributed TensorFlow [46]) TensorFlowOnSpark (TFoS) exploits Spark native mechanisms for the automated configuration of the cluster, while the direct tensor communication favors scalability, simply by adding machines into the cluster. This aspect is also highlighted in in Chapter 5, indicating that the use of TFoS can yield a scalable solution on the problem of galaxy velocity parameter estimation. Furthermore, TFoS enables smooth integration of TensorFlow code over Spark clusters, with minimum changes on the original TensorFlow code.

In a typical TFoS cluster, each node is coordinated by Spark and acts as a container that locally executes the operations of TensorFlow graphs. A node is randomly selected to act as the PS and is responsible for providing a global average of network parameters, while the rest of the nodes are responsible for training and operating on the RDDs defined by the underlying Spark architecture. TFoS bypasses the communication philosophy of Spark, thereby allowing direct tensor communication between TF processes.
A typical TFoS system is illustrated in figure 3.4. Each Spark executor is coordinated by the Spark driver and receives a copy of the TensorFlow graph. One of them is selected to be the parameter server at random, while the rest are responsible

[1] https://github.com/yahoo/TensorFlowOnSpark

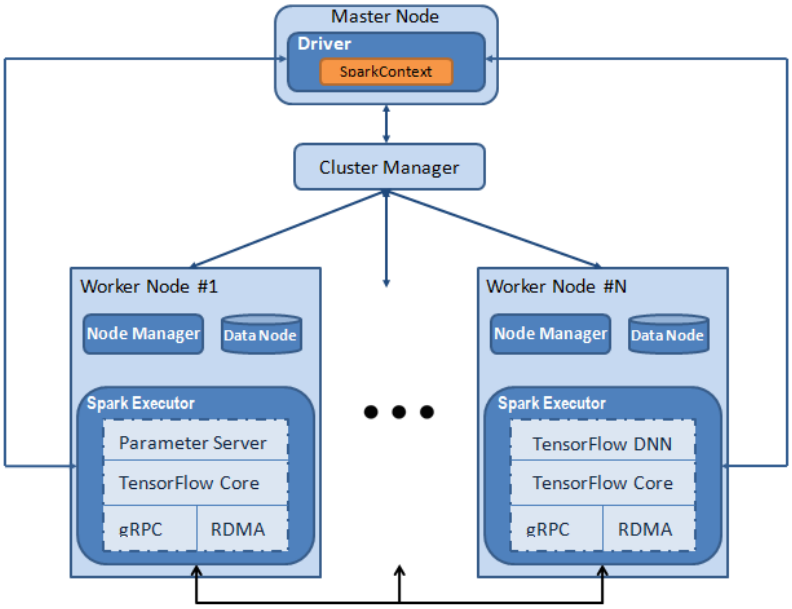
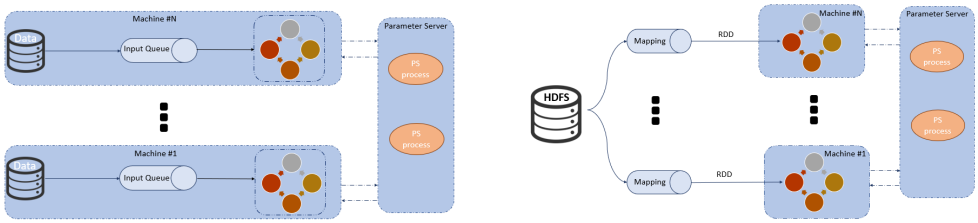


Figure 3.4: Example TFoS System Architecture.

for training the replicas. In addition, TFoS supports direct Tensor communication among TensorFlow processes, enabling it to scale easily by adding machines. So far, two distinct modes exist in order to start a TFoS cluster: (a) TensorFlow and (b) Spark mode.



(a) TensorFlow Input Mode (b) Spark Input Mode

Figure 3.5: TensorFlowOnSpark Data Ingestion Modes

Spark mode, uses RDDs and to feed the data to the Workers. This is useful for implementing Spark pipelines, but since Python uses only one thread to serialize RDDs into the replicas, it can create significant performance bottlenecks. *TensorFlow mode* on the other hand, takes advantage of TensorFlow's QueueRunner and file reader multi-thread functions or Python libraries functions for data ingestion.

Chapter 4

Cluster Setup

To benchmark our proposed distributed DNN architectures, the experiments were conducted on an Apache Spark Standalone cluster of 5 PCs, featuring TensorFlow Core version 1.2.1, Apache Spark version 2.1.1, Apache Hadoop version 2.6, and TensorFlowOnSpark version 1.0.0. The DNN architectures were implemented using Keras [1] version 2.0.4. , a high level neural network API written in Python. The Master of the cluster is configured with an Intel Core i7-6700 3.40GHz CPU, and has allocated 8GB memory and 450 GB disk space. Meanwhile, the workers are configured with Intel Core i5-3470 3.20GHz CPUs, have allocated 2GB memory and 450 GB disk space.

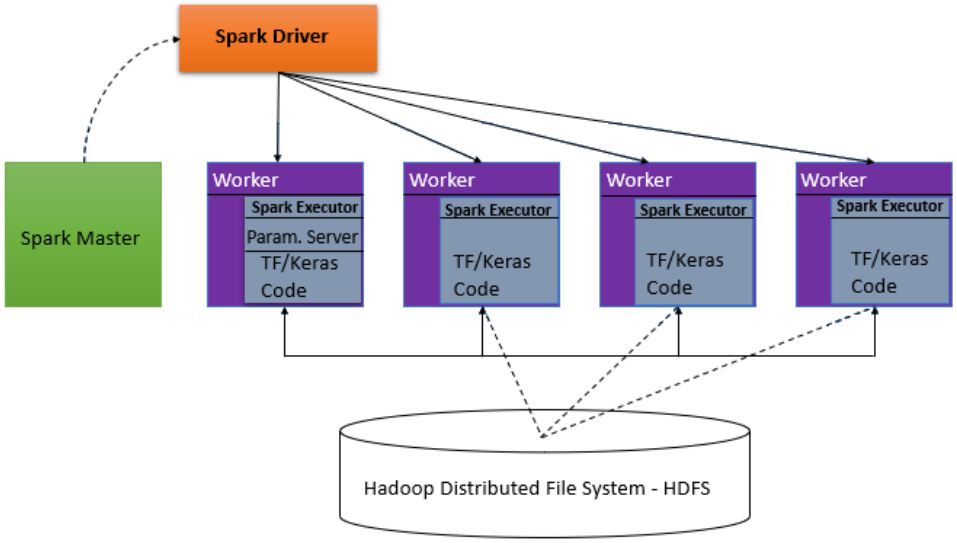


Figure 4.1: TFoS Cluster Setup.

Figure 4.1 presents our cluster setup. It depicts Spark nodes connected via

[1] F. Chollet, Keras .https://github.com/fchollet/keras, 2015.

Ethernet. The Master configures the cluster through the driver, and workers communicate through TFoS tensor communication. Hadoop's Distributed File System (HDFS) [47] is configured as the primary storage system, in order to provide high-performance access and rapid transfers of data between the nodes responsible for training our DNN architectures. All nodes run Ubuntu Linux 16.04. For the purpose of identifying our system's bottlenecks, Ganglia is used to monitor CPU, memory, and network utilization on every node [48].

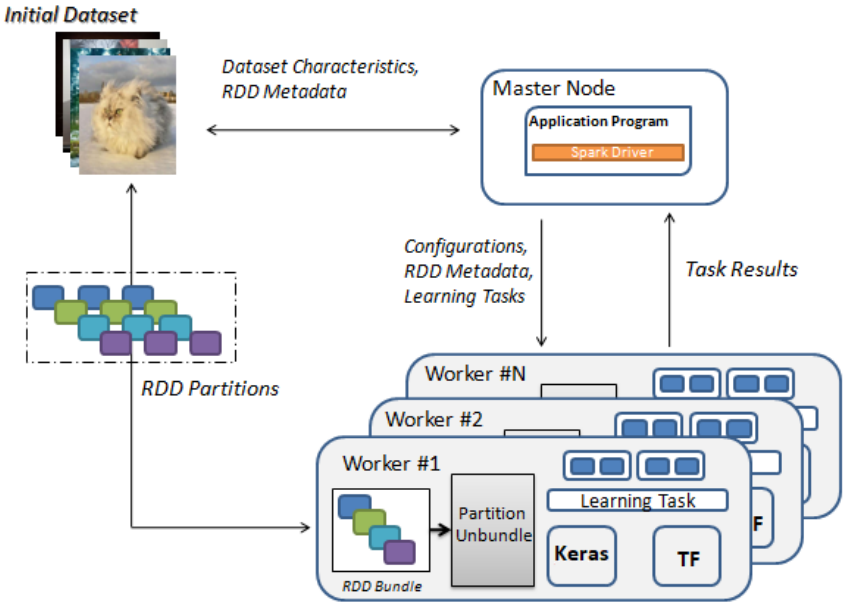


Figure 4.2: TFoS Learning Scheme.

Figure 4.2 illustrates our learning scheme for Spark Input approaches. The input dataset along with the respective labels has to be jointly processed for solving our learning tasks, therefore multiple such bundles have to be created. Apache Spark is responsible to model each individual dataset into RDDs, and then create RDD bundles through multiple transformations. The resulting bundled partitions and the learning tasks are then parceled into the workers, which are responsible for separating the RDD bundles, which are then provided as inputs to our architectures. As stated before, the libraries of Keras and TensorFlow are incorporated into each worker for implementing our learning tasks. Through the driver (application program), the Master can control several aspects of the distributed learning process, and send configuration parameters and RDD metadata to the workers. Once finished, the result of each task returns back to the driver.
For TensorFlow input approaches, the procedure remains relatively the same, except for the RDD bundle part. Since we don't use Spark to feed data to our cluster in these approaches, we store our data locally on each worker, and provide them directly to our learning tasks through TensorFlow and Python libraries.

Chapter 5

Distributed Deep Learning in Astronomy

In this chapter, we explore the problem of accurate redshift estimation from spectroscopic observations through a distributed framework. In the first section, we present our adopted dataset and elaborate on its specifications. Next, we introduce our proposed one dimensional CNN and explain its migration in a distributed environment through our data paralellism scheme. Finally, we demonstrate our experimental evaluation and elaborate on our findings.

5.1 Dataset

Redshift (z) is parameter encoding the shift of the spectrum of astronomic objects like galaxies towards longer (redder) wavelengths due to the accelerated expansion of the universe. As a result, redshift estimation plays a fundamental role in observational cosmology, as it can used to accurately estimate galaxies' radial distances and their position.
For our experiments, we produce simulated spectroscopic one dimensional data. In order to generate realistic observations, our dataset follows specific redshift, color, magnitude and spectral type distributions, modeled after data received from a space telescope that measures the characteristic distance scale imprinted by primordial plasma oscillations in the galaxy distribution. Moreover, each signal encodes spectroscopic content in the range of 1.1 to 2.0 μm which is mapped to 1800 distinct spectral bands. In our setup, we treat the problem of redshift estimation as a multi-class classification problem by dividing the redshift range z = [1 - 1.8) to 800 classes.

5.2 Proposed Framework

5.2.1 One dimensional CNN for Redshift estimation

Since the structural form of the spectra used for redshift estimation is one dimensional, our CNN architecture should also utilize one dimensional convolutional operations. As a result, we applied an one dimensional CNN, consisting of two layers of convolutions with non-linear activation functions followed by a dropout layer, a flatten layer and one fully connected layer, which produces the final classification output. The convolutions employ kernels of length 8 while the Rectified Linear Unit activation function is employed. Pooling layers have been excluded from our model, due to their property to render the network oblivious to changes applied on the initial input. Given that these changes are vital to render our network capable to differentiate between individual redshifted states, by using pooling and consequently suppress them, we actually distort our model's ability to identify different redshifts. Our architecture is presented in figure 5.1.

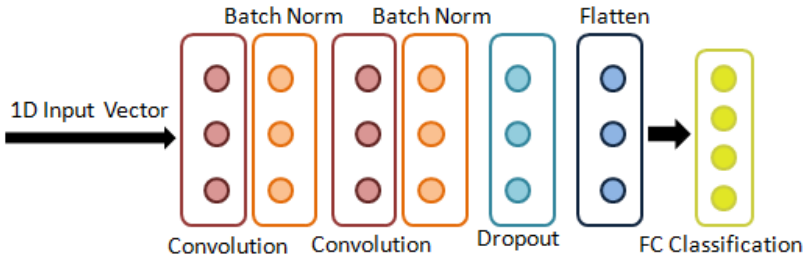


Figure 5.1: One Dimensional CNN for Spectroscopic redshift estimation.

5.2.2 Asynchronous Data Parallelism Scheme

In this work, we migrated the one dimensional CNN architecture described in the previous subsection to the distributed framework introduced on chapter 4, following a data parallelism approach. The simulated dataset, along with their respective labels is given as an input in the form of an RDD bundle, compressed by using the $zipWithIndex$ transformation. The RDD is then split across different machines. Then, the training process is performed locally, by using a replica of the whole CNN. At the end of each epoch, the PS collects the resulting global variables and updates them.A more detailed depiction of our RDD generation scheme can be seen in Figure 5.2.

We opted to perform asynchronous training for updating the PS, therefore, an issue known as the stale gradient problem emerged [49]. In order to overcome this issue, instead of updating global parameters immediately, the PS was forced to

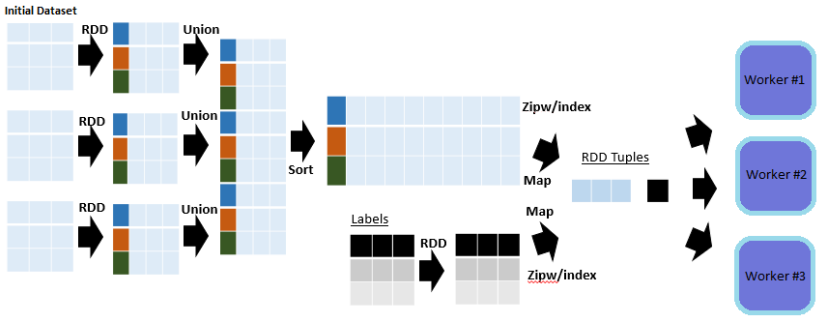


Figure 5.2: RDD Bundle Generation for the Astrophysics Data.

wait to collect some number s of updates ΔW_j from any of the M training workers, such as $1 \leq s \leq M$. The parameters are then updated according to the following equation:

$$W_{i+1} = W_i - \frac{1}{s} \sum_{j=1}^{s} \lambda(\Delta W_j) \Delta W_j \tag{5.1}$$

where $\lambda(\Delta W_j)$ is a scalar staleness-dependent scaling factor[**?**]. Once the variables are updated they are redistributed across the machines, so they can begin the next epoch. Although this update routine mitigates the issue of the scale gradients, it adds overhead to the network since it slows down the overall training process.

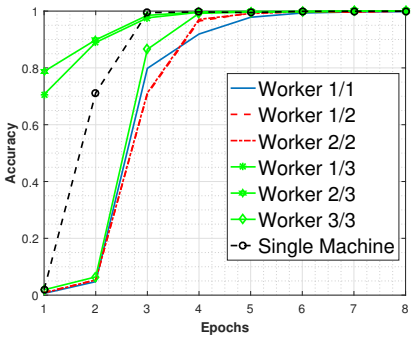
5.3 Experimental Results

A series of experiments were conducted using two sets simulated noisy data from subsection 5.1, consisted of 5K and 15K training samples respectively. Our goal is to study the impact of the following parameters: (i) the number of distributed computing nodes, and (ii) the distribution of data among the nodes. Regarding the data distribution, we explore two cases, where in the first case the same data is presented to all working nodes, while in the second case, disjoint sets of examples are utilized by each node. The experiments presented below, were also conducted on the conventional CNN described in section 5.2.1, which serves as our baseline.

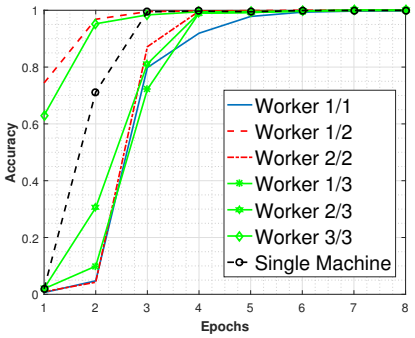
5.3.1 Evaluation

Figure 5.3 presents the performance of the CNN as a function of training accuracy. From top to bottom, Figures 5.3(a),(b) present the results of the 5K samples set for the cases of same and different data, respectively, while Figures 5.3(c),(d) provide the results for the 15K samples set.

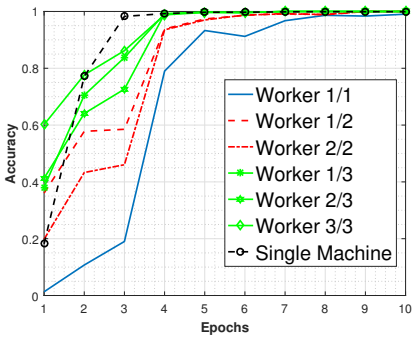
We notice that that in all cases, using multiple working nodes has a positive impact in terms of accuracy, for a given number of training epochs. For the case of two workers, we observe that when same data are used, the performance between 1 and 2 working nodes is similar, while when different sets of examples are considered,



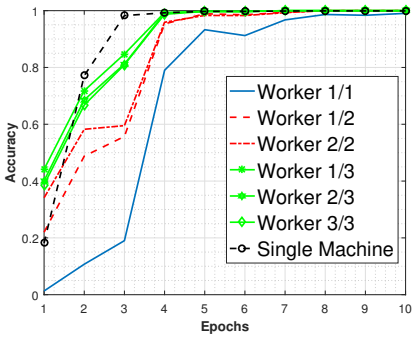
(a) 5K samples set, same data on each worker.



(b) 5K samples set, different data on each worker.



(c) 15K samples set, same data on each worker.



(d) 15K samples set, different data on each worker.

Figure 5.3: Training epochs for various cluster sizes.

there is a significant difference in terms of accuracy by each of the distributed nodes. For the case of three computing nodes, there is a clear benefit in both cases of using the same or different training examples.

Specifically for the 15K samples set, we observe that the larger amount of training samples has a positive effect on the system's learning capacity and the performance variation across different machines is less prominent. The sequential CNN attains over 75% accuracy faster for this set, at three epochs, but the cluster gains similar behavior as we add more machines. Specifically, in the one-node cluster, the CNN needs about 8 epochs to reach the same accuracy as the sequential case, while for the 3-node cluster the amount of epochs required is reduced to 4.

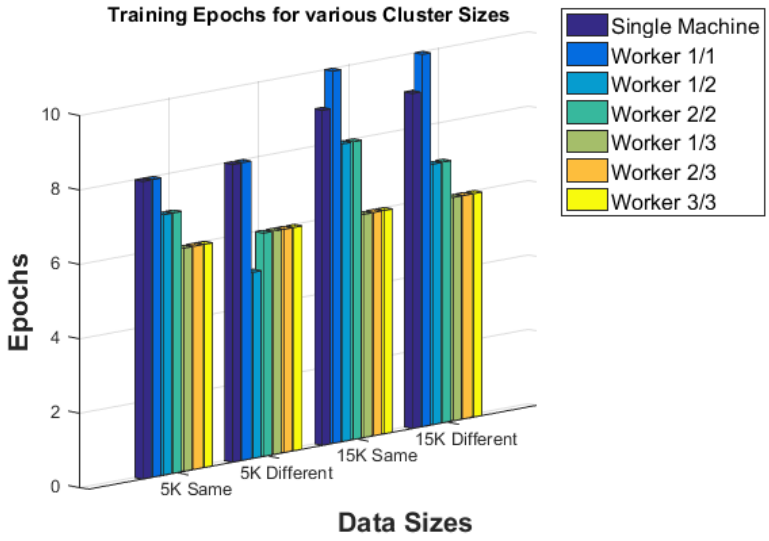


Figure 5.4: Number of epochs required to reach a plateau in training accuracy performance.

Figure 5.4 presents the number of epochs required to reach a stable performance, which is introduced through an early stopping criterion in terms of differences in accuracy between successive training epochs. These results clearly support the argument that distribution processing platforms can have a significant advantage in terms of processing time, leading to a 25% reduction in the time need to achieve stable performance, for both training datasets. Furthermore, we also observe that for larger number of computing nodes, the impact of data distribution becomes less important.

A natural question to ask is how training loss behaves in our distributed framework. Figure 5.5 depicts the number of epochs required to reach a stable performance, using identical criteria as before but in terms of loss differences between epochs instead of accuracy, while Figure 5.6 presents the values of the achieved by the loss function, both on the 5K dataset.

The results of Figure 5.5 once again demonstrate the positive effect of our

Criteria 1 (Patience=1, minimum delta=0.001)			
		5K-Same Data	5K-Different Data
Single Machine		8 Epochs	8 Epochs
1 Worker		8 Epochs	8 Epochs
2 Workers	Worker #1	7 Epochs	6 Epochs
	Worker #2	7 Epochs	5 Epochs
3 Workers	Worker #1	6 Epochs	6 Epochs
	Worker #2	6 Epochs	6 Epochs
	Worker #3	6 Epochs	6 Epochs

Table 5.1: Epochs needed to achieve stable performance for strict early stopping parameters.

platform in terms of distributed processing time, reducing it by 50% when different sets of examples is considered. However, when each worker sees the same data the impact, although visible, is negligible. Meanwhile, Figure 5.6 suggests that, when the same data is used, the performance between most of the nodes is similar, while when different sets of examples is considered, there is a significant difference in terms of loss in the cases of two- and three-nodes cluster. We also observe that loss behaves similar to accuracy, in that it drops faster on the sequential CNN, however similar performance is attained one epoch later, regardless of its size.

Criteria 2 (Patience=3, minimum delta=0.0001)			
		5K-Same Data	5K-Different Data
Single Machine		14 Epochs	14 Epochs
1 Worker		16 Epochs	16 Epochs
2 Workers	Worker #1	13 Epochs	15 Epochs
	Worker #2	13 Epochs	15 Epochs
3 Workers	Worker #1	13 Epochs	10 Epochs
	Worker #2	13 Epochs	10 Epochs
	Worker #3	13 Epochs	10 Epochs

Table 5.2: Epochs needed to achieve stable performance for lenient early stopping parameters.

To further evaluate the distributed framework, we applied another set of early stopping parameters, tested on the 5K samples dataset, and present the results in Table 5.2, while the initial, more lenient parameters are presented in Table 5.1. We experimented on various cluster sizes, considering both cases of data distribution. Training accuracy was selected as the quantity to be monitored. The set of parameters we decided to experiment with are: (i) patience, which represents the number of epochs before stopping, once accuracy stops improving and (ii) minimum delta, a threshold to quantify whether accuracy has improved or not at the end of each epoch. As a result, our new criteria (right column) are less strict than the old ones (left column), as indicated by the higher patience and the smaller minimum delta.

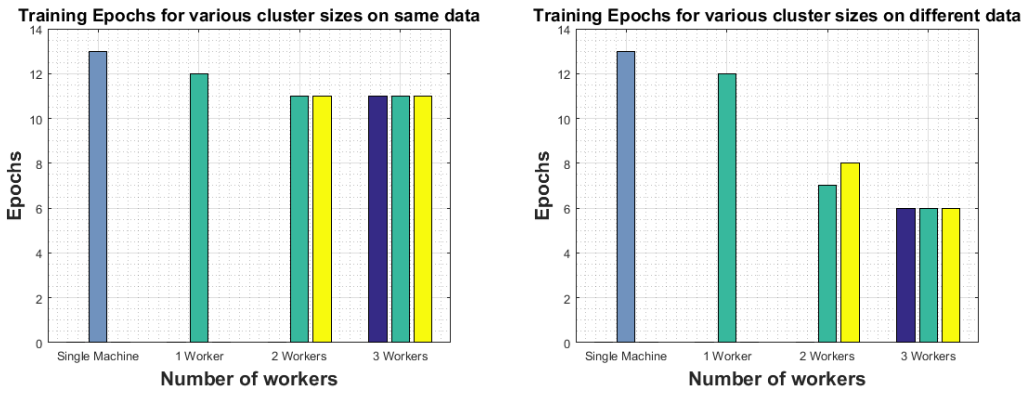


Figure 5.5: Epochs required to obtain the minimum loss.

These changes lead to a higher number of epochs required for training. Similar behavior can be observed in both cases of early stopping, with the amount of epochs decreasing as more nodes are added into the cluster. However, for our new set of parameters, we observe that after a certain number of nodes the amount of epochs required to achieve a stable performance remains the same, when the same set of examples is taken into consideration. On the other hand, this is not true for the case of different sets of data, as in the 3 nodes case the results show a 30% reduction in processing time. However, TFoS does not perform well compared to non-parallelized CNN when same data are considered, since it needs 13 epochs to reach a a stable performance for the 3-node cluster, while the sequential CNN needs 14. When different datasets are considered, the performance of TFoS caps at 10 epochs for the 3-node cluster, much better than the sequential CNN.

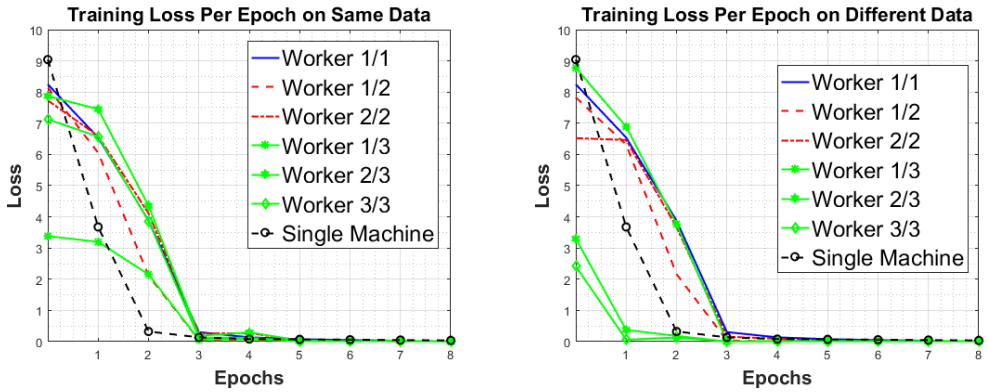


Figure 5.6: Loss performance for different cluster sizes.

Chapter 6

Distributed Deep Learning in Remote Sensing

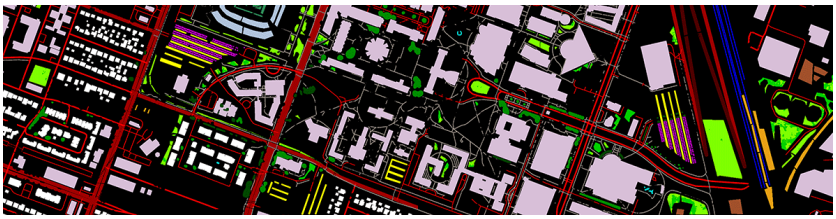
In this chapter, we concentrate on the problem of urban land use and land cover classification through the use of a multimodal distributed Deep Neural Network. In the first section we introduce the dataset used to conduct our experiments. In the following sections, we describe our patch generation technique, in order to address the limited size of our adopted dataset, and present our conventional CNN architecture to confront the problem of multimodal learning. Furthermore, we propose several model parallelism approaches for our Convolutional Neural Network and extensively study their trade-offs, which are then elevated as general trade-offs between model parallelism techniques. Finally, in the last section, we demonstrate our experimental evaluation and comment on the deduced results.

6.1 Dataset

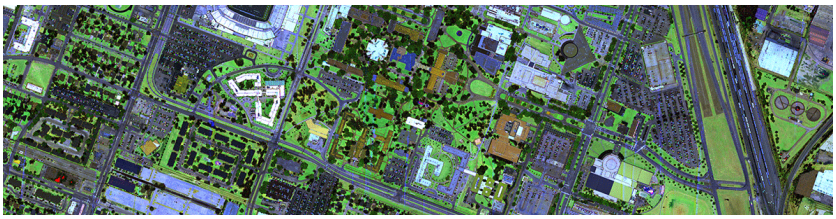
The dataset used for this part of our work was available as a training set for 2018 IEEE GRSS Data Fusion Contest[1], organized by the Image Analysis and Data Fusion Technical Committee, and its main focus was fusion methodologies for multi-source remote sensing data. It was collected by NCALM at the University of Houston (UH) on February 16, 2017, covering the University of Houston campus and its surrounding areas, and the task to be performed for the competition was urban land use and land cover classification.

Multiple sensors were used in order to acquire this dataset, including an Optech Titam MW (14SEN/CON340) with integrated camera (a LIDAR sensor operating at three different laser wavelengths), a DiMAC ULTRALIGHT+ (a very high resolution color imager) with a 70 mm focal length, and an ITRES CASI 1500 (a hyperspectral imager). The produced multi-source optical remote sensing data were the following:

[1] https://www.grss-ieee.org/community/technical-committees/data-fusion/2018-ieee-grss-data-fusion-contest/



(a) Ground truth of the dataset.



(b) Color composite of MS-LiDAR intensity data.

Figure 6.1: Example parts of the dataset.

- Multispectral-LiDAR point cloud data at 1550 nm, 1064 nm, and 532 nm; Intensity rasters from first return per channel and DSMs at a 50-cm GSD. This part of the dataset was not included in our experiments.
- Hyperspectral data covering a 380-1050 nm spectral range with 50 bands at a 1-m GSD, resulting in an image of size 601 x 2384 x 50.
- Very high resolution RGB imagery(VHR-RGB) at a 5-cm GSD. The image is organized into four separate tiles, resulting in an image of size 12020 x 47680 x 3, when concatenated.

Ground truth was also provided as raster at a 0.5-m GSD, superimposable to airborne images. It is corresponding to 21 urban land use and land cover classes, listed in Table 6.1. The ground truth as well as the LiDAR parts of the dataset are presented in Figure 6.1.

6.2 Proposed Framework

6.2.1 Input Preprossesing

By reading the above section, one can easily deduce that our dataset is small, since it consists of only one image per modality. In addition, we decided to ignore the MS-LiDAR intensity data due to our cluster size, so we end up with two modalities instead of three and with an even smaller dataset. This can be quite problematic, since it may cause our Neural Network to underperform and render it prone to overfitting. In order to mitigate this, we opted to use the oversampling technique of image patch generation.

Number of Class	Corresponding Label
0	Unclassified
1	Healthy Grass
2	Stressed Grass
3	Artificial Turf
4	Evergreen Trees
5	Deciduous Trees
6	Bare Earth
7	Water
8	Residential Buildings
9	Non-Residential Buildings
10	Roads
11	Sidewalks
12	Crosswalks
13	Major thoroughfares
14	Highways
15	Railways
16	Paved Parking Lots
17	Unpaved Parking Lots
18	Cars
19	Trains
20	Stadium Seats

Table 6.1: The original set of labels and their corresponding class.

Image patches can be extracted from a training set in numerous ways [50, 51]. In our work, we generated a training set using the following steps: (a) Firstly, the four separate VHR-RGB tiles had to be merged into one, and then resized along with the HSI image to match the spatial size of the ground truth data; (b) Image patches of size 25 x 25 pixels were cropped out by sliding through each training image. The generated patches contain the same pixels from both images, in order to get complementary information. And (c) for each class, we randomly chose 400 patches from each image with a selected pixel as a starting point, resulting in overall 8000 patches. This is not the case for Class "0" which was removed from our experiments. A high level pipeline of our proposed multimodal CNN can be seen in Figure 6.2.

6.2.2 Multimodal CNN for Remote Sensing Classification

Usually in multisensor setups where more than three bands are available, the general approach is to employ multiple neural network branches/streams and then fuse the produced features to obtain a classification map. Therefore, we tried to implement a network that efficiently combines features from multiple modalities. More specifically, the main goal was to design a patch-level CNN that takes in

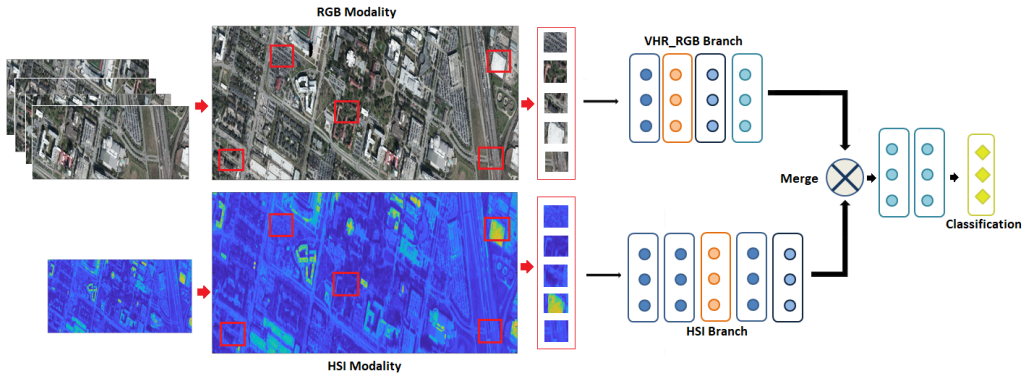


Figure 6.2: The overall pipeline of the proposed multimodal CNN. The inputs are 25 x 25 image patches of RGB and HSI images, and the output is the land prediction label.

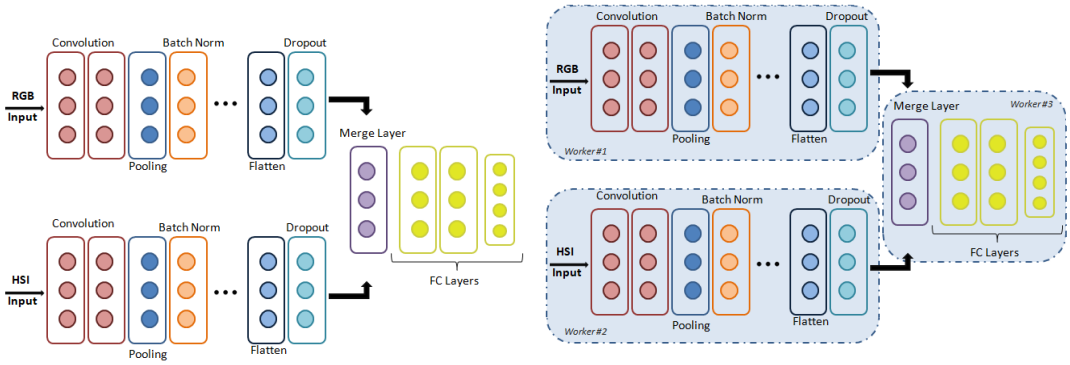
multiple patches with multiple bands each, and provides good quantitative results. The proposed fusion architecture is depicted in Figure 6.3(a). It is consisted of convolutional and pooling layers in order to extract features, and dropout and batch normalization layers to avoid overfitting. These are followed by two fully connected layers and a final softmax scoring layer. In order to construct a more memory efficient DNN, we adopted an early fusion approach and fused the features before the first fully connected layer.

The proposed fusion Neural Network consists of three parts, one for HSI data, one for VHR-RGB data, and one for classification. The first two branches contain 3D convolutional layers that operate on their respective input patches to generate features. We decided on three dimensional convolution, since our hyperspectral data also contain spatial information that 2 dimensional models fail to accurately extract features from. The input size of the RGB branch is 25 x 25 x 3, where 3 is the number of bands. For HSI input on the other hand, we chose eleven out of the available fifty bands, resulting with a size of 25 x 25 x 11. The reason behind this decision is that our cluster lacks the computational resources to process HSI patches with a higher number of bands.

6.2.3 Model Parallelism Schemes

We adopted three sets of approaches to parallelize our model through the machines of the cluster. In the first approach, which also serves as our baseline, we assign each branch to a different worker, while the PS is responsible for supervising the whole training procedure. This approach is a bit naive, since not only it does not fully utilizes the resources of our cluster, but also produces unnecessary network overheads. In our second set of approaches we tried to do smarter branch assignments in order to reduce network traffic, and on the third set, our goal is to achieve

maximum resource utilization while keeping traffic as low as feasibly possible. Hyperparameter tuning is also applied in order to study the magnitude of the impact hyperparameters have on the behavior of our cluster.
Since we decided on model parallelism schemes for this case study, we removed Spark from the data feeding procedure and opted for TensorFlow Input mode to ingest data. Therefore, our datasets are stored locally in each worker instead of HDFS, and we are using Python functions to directly feed the data to our CNN.



(a) Single Machine Architecture (b) Default Distributed Architecture

Figure 6.3: The network structure of proposed multimodal CNN, along with its distributed equivalent. Both have three main parts: The VHS-RGB part at the top branch, the HSI part at the bottom branch and merging part that leads to classification.

6.3 Experimental Results

6.3.1 Default Approach

Our initial CNN along with its default distributed approach are depicted on Figure 6.3. As stated before, in this approach we split the network into three branches, one for each modality and a final one for fusion and classification, and assign each branch to a different worker. Figure 6.4 presents the performance between the distributed and the non-distributed CNN in terms of accuracy and loss, when given as input the 8k patches described in section 6.2.1. We notice that the performance between the two is similar, with the distributed approach achieving slightly better results. However, the distributed approach introduces network overheads due to inter-machine communications and does not fully utilize the resources of our cluster, both of which prolong the overall training time.

In the following sections we describe the procedure we used to reduce those overheads, as well as smarter ways to slice our model in order to efficiently exploit the resources of our cluster.

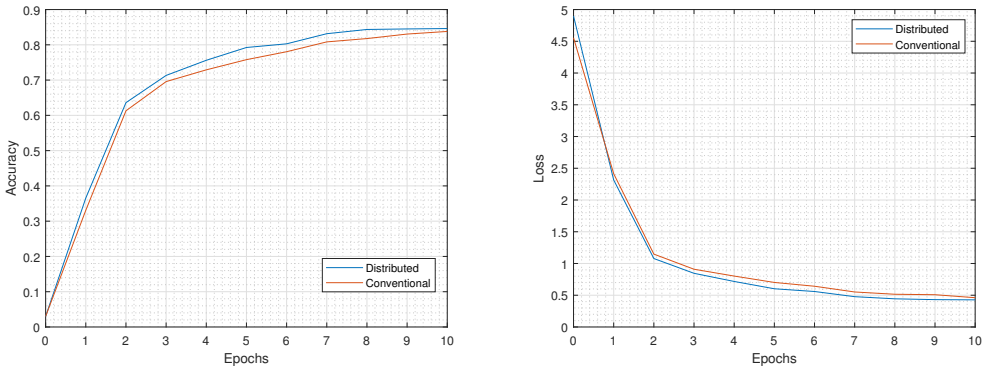


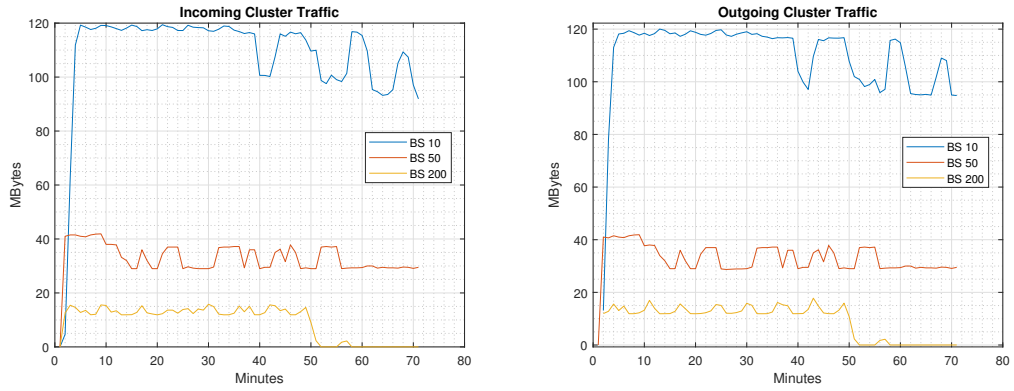
Figure 6.4: Training loss and accuracy for distributed and single machine approaches after 10 epochs.

6.3.2 Hyperparameter Tuning

The objective of this set of experiments was to examine if and how hyperparameters affect the overall performance of a distributed CNN model. Our findings establish that the hyperparameter of batch size has a tremendous impact on communication overheads. Since it actually defines the number of samples that will be propagated through the network, it will also, by extension, define the number of data transfers between machines during the training and testing phases of our experiments. Figure 6.5 depicts both the incoming (a) and outgoing (b) traffic of our cluster for different batch sizes, in a time interval of about 70 minutes. These plots indicate that for a batch size of 10, network traffic can reach values as high as 120 MB/sec compared to the batch sizes of 50 and 200, which get values more than 5 times lower. As a result, high batch sizes can greatly accelerate our experiments, and more specifically their, extremely time consuming, training phase. However, it should be noted that when using a very large batch there is a significant degradation in the quality of the model, as measured by its ability to generalize. This happens due to the fact that that large-batch methods tend to converge to local minimizers of the training function. To conclude, larger batch sizes can be beneficial for model parallelism approaches since they greatly reduce network overheads, as long as they don't cause a negative impact on a model's convergence. Another important think to discuss is the peculiar behavior of traffic when the value of batch size is 10 or 50. In these cases, we observe a drop of both traffics after some amount of time. However, the batch size of 200 still provides better results.

6.3.3 Network Overhead

Even though we managed to significantly reduce communication overheads through batch size adjustments , one can consider if further improvements are feasible. Figure 6.6 illustrates the incoming and outgoing traffic of each worker of our cluster



(a) (b)

Figure 6.5: Cluster Traffic for different Batch Sizes.

for our default architecture with a batch size of 200. The network was trained for 10 epochs.

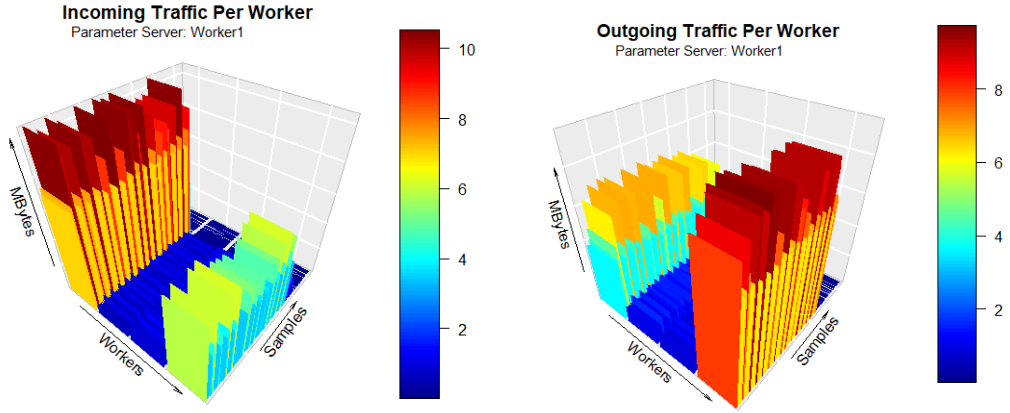


Figure 6.6: Communication traffic per worker of our default architecture.

The first thing we observe is that the bulk of traffic is actually gathered between two workers, worker 1 which acts as the PS, and worker 4 which is responsible for the merged part of our multimodal CNN. It is natural to expect high incoming network traffic on the worker responsible for the merging, since it collects data from both worker 2 and worker 3. It also natural for the PS to have the highest incoming traffic, as it receives data form a branch consisted mostly of fully connected layers, which have much more parameters than the convolutional ones. The cluster's behavior also makes sense for outgoing traffic. In this case, worker 4 has the highest traffic of every other machine on the cluster, since, as stated before, is responsible

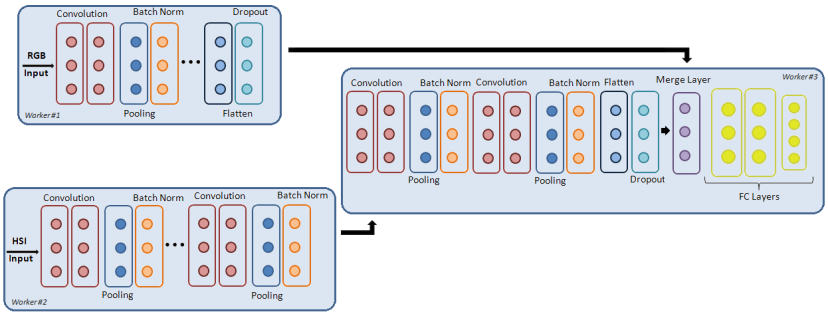


Figure 6.7: Second Model Parallelism Scheme.

for the part of the model with the fully connected layers, and therefore has more parameters to send. As for the PS, it has to send a part of the parameters received from the fully connected layers to both worker 2 and worker 3, so the traffic also tends to be high.

The aforementioned results were used as basis to construct another distributed

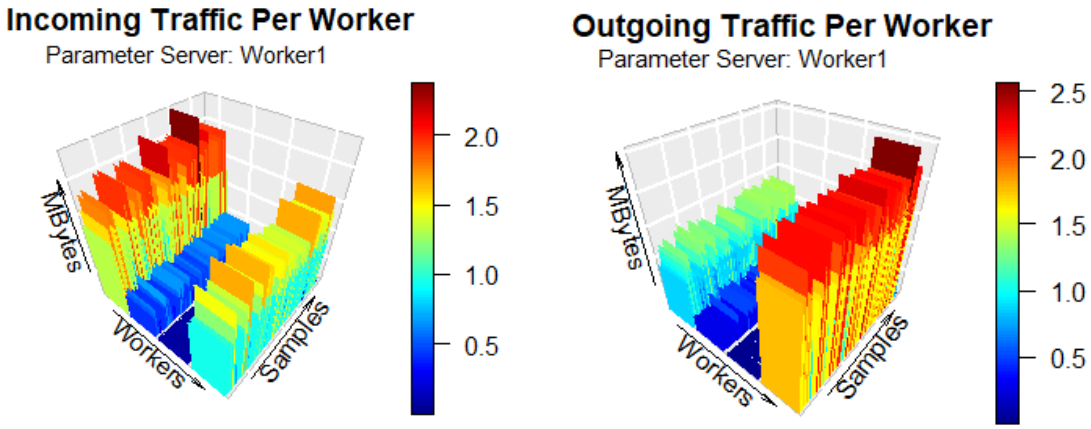
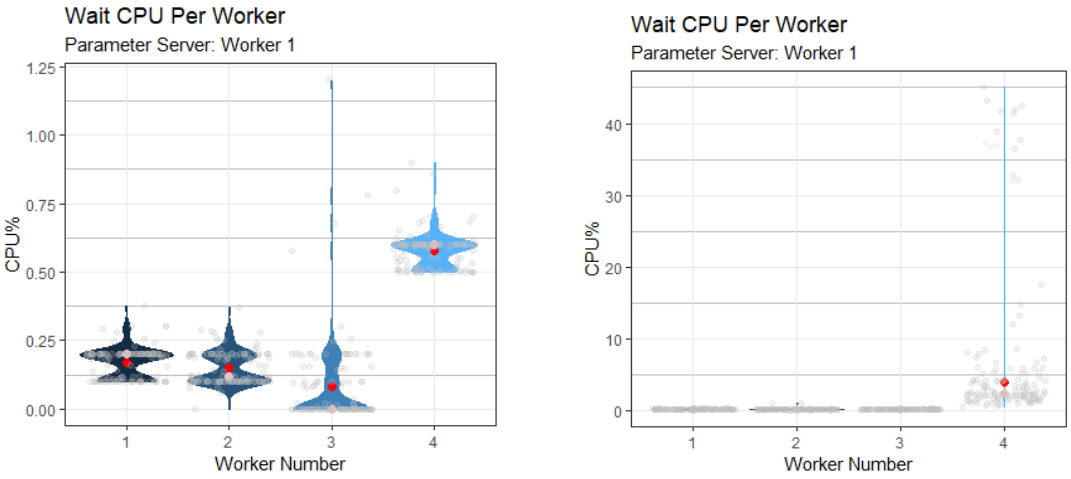


Figure 6.8: Communication traffic per worker of the architecture of Figure 6.7.

model for the CNN, which is depicted in Figure 6.7. In this approach, we migrated the flatten and dropout layers of our HSI branch along with its two last blocks of convolution pooling and batch normalization layers to the merging worker. Our goal was to reduce the incoming traffic in the merging worker, owing to the fact that the less convolutional layers a branch has, the less data it needs to send during communications. The RGB branch remained the same. To evaluate the new distributed architecture, we present its network traffic on Figure 6.8. Predictably, traffic follows similar behavior to our initial approach, mostly being gathered between the PS and the node responsible for the merging.

However, the amount of both incoming and outgoing traffic has been significantly reduced, in comparison with our default model parallelism scheme. Specifically, the outgoing traffic of the merging worker can be as low as 2.5MB/sec compared to the baseline's, where it reaches about 10MB/sec. As a result, traffic on the PS has also become pretty low, reaching about 2.5MB/sec incoming traffic, as well as 1.5MB/sec outgoing traffic. While these results look promising, they are actually a side effect, since this alternative scheme greatly increases the CPU wait metric of the merging worker, as it has quite a few instances that reach values as high as 45%, as illustrated in Figure 6.9. This means that, on these instances, about 45% of the time this worker does nothing but waits for resources to be freed, which actually implies that either there are I/O tasks that are not yet scheduled or, more importantly, there are learning tasks waiting to be executed. This has a negative effect of the performance of our model, since it can exponentially increase execution times. In comparison, with a wait CPU as high as 1.25%, our default model is much faster and more productive, despite the higher communication overheads.

Since this approach was proven to be counterproductive, we decided to study



(a) Baseline Architecture

(b) Architecture of Figure 6.7

Figure 6.9: Wait CPU Per Worker for our two distributed Architectures.The shape of each box shows CPU distribution for each worker. Gray dots define sample points. Red dots illustrate the mean value, while pink ones the median of our data.

another one, not as computationally expensive. Therefore, we concluded to split our model as described in Figure 6.10. This time we migrated only one convolutional, pooling block of the HSI branch instead of two, and we did the same for the VHR-RGB branch. The rest architecture remained the same. The impact of this architecture on the traffic of our cluster can be seen on Figure 6.11. Once again, we observe that traffic is gathered mostly between two workers.It is also apparent

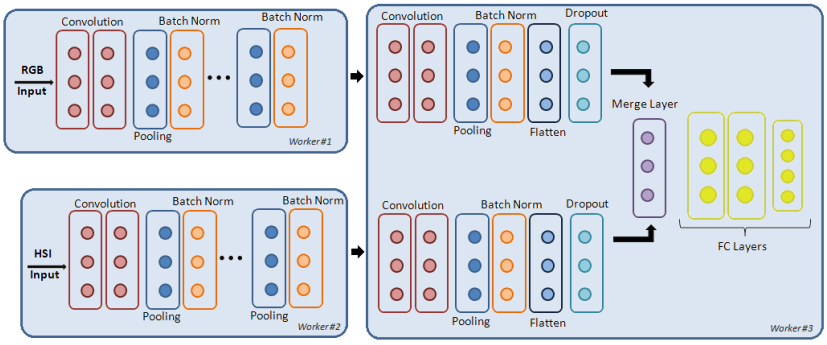


Figure 6.10: Third Model Parallelism Scheme.

that this time, worker 2 is responsible for branch merging instead of worker 4. This occurs because TFoS task manager randomly assigns jobs to workers each time we fire a new application on the cluster. The values of both incoming and outgoing traffic are marginally higher than the ones presented in Figure 6.8, but they are also significantly lower than the values of our baseline. Merging worker's Wait

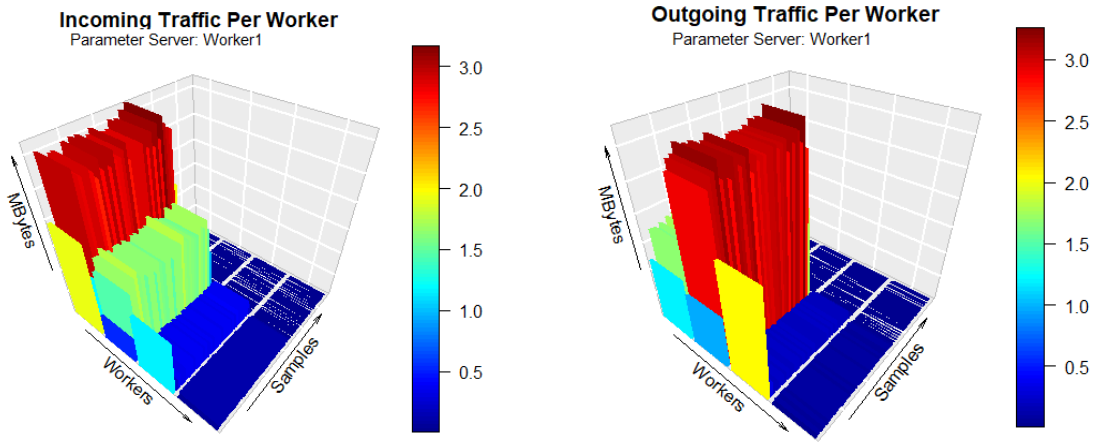


Figure 6.11: Communication traffic per worker of the architecture of Figure 6.10

CPU can get values as high as 16%, as Figure 6.12 indicates. However, this is true only for one time instance, since there is only one grey dot for these values, while the rest range from 0% to 1%. Wait CPU values from the rest of the workers are generally small. Therefore we can safely deduce that this architecture is ideal for reducing communication overheads, due to presenting the most optimal trade-off between traffic and Wait CPU out of the three presented in this work.

Removing layers has also been considered as an alternative approach to our problem. Even though this approach managed to significantly reduce the traffic

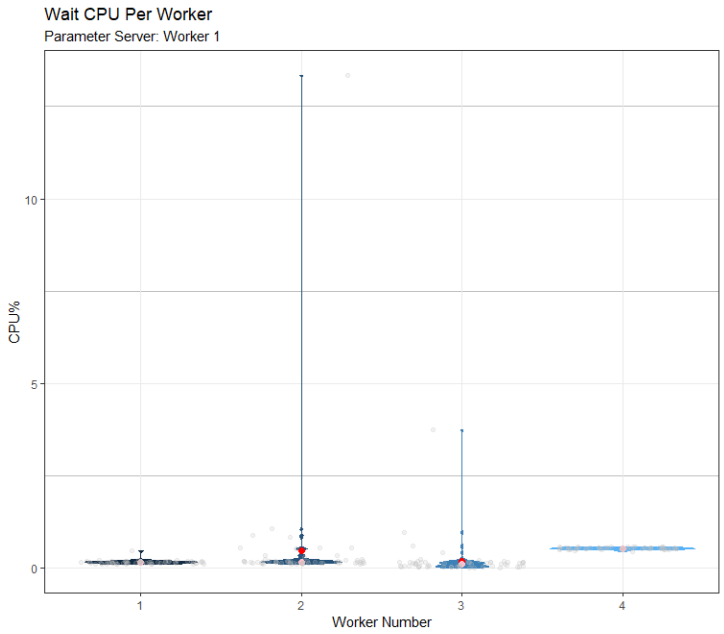


Figure 6.12: CPU Wait of Figure 6.10 's Architecture

of our network, it also hindered our model's ability to generalize, producing poor evaluation results. Therefore, it was considered suboptimal and was abandoned.

6.3.4 Cluster Resources Exploitation

Figure 6.13 illustrates the performance of our cluster in terms of CPU and Memory usage, when executing our baseline approach.
By observing the left plot, we deduce that the PS and the merging worker 4 don't deploy much CPU, since the former just redistributes parameters to the rest of the cluster, and the latter is in charge of only a small number of layers. On the other hand, worker 3, which is responsible for the HSI branch, is the most heavy duty machine in the cluster. This indicates that this branch is the most computationally expensive part of our model and it should be split accordingly in order to both release some CPU from worker 3, but also to exploit CPU resources that remain idle on the rest of the worker.

The results of the right plot however, illustrate that this time the PS is the most heavy duty worker out of the four, while the memory usage on the machines containing the branches of our CNN tends to be less than 2GB. One approach to deal with the issue of high memory consumption on the PS is to simply provide it with more memory. An alternative solution is to deploy a second PS to share the load. However, with this option we take the risk to increase network traffic, as it essentially adds an extra machine to our cluster. Nevertheless, with our current setup none of the aforementioned solutions are viable.

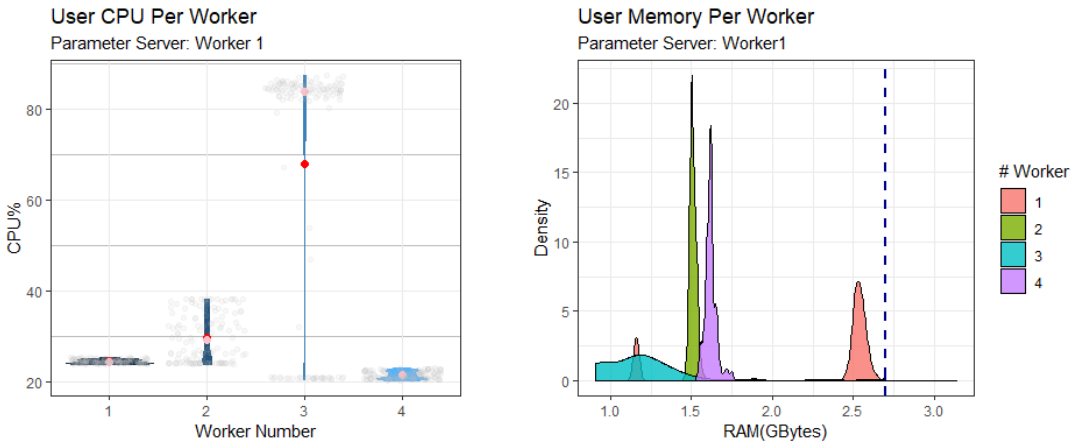
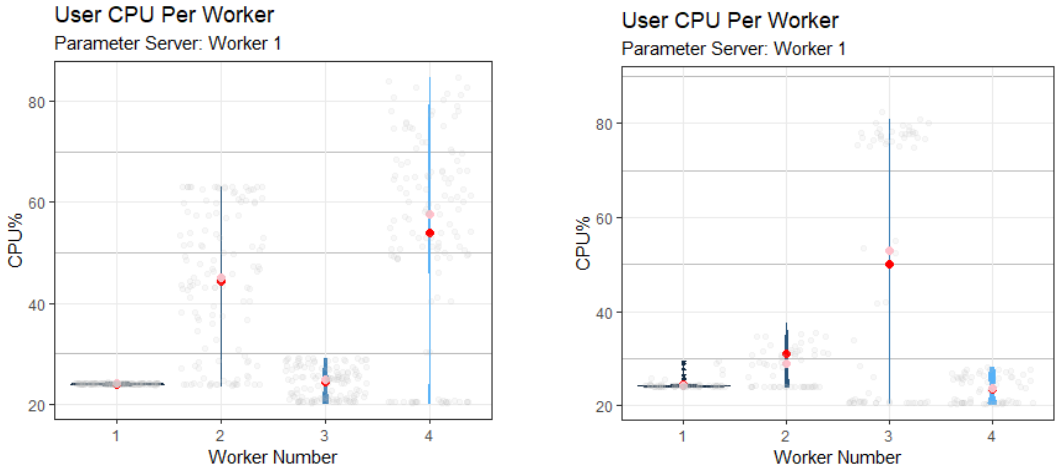


Figure 6.13: Resource Metrics for our baseline architecture.



(a) CPU of second Scheme

(b) CPU of third Scheme

Figure 6.14: CPU Usage of Parallelism Schemes presented in section 6.3.3

The performance of each worker in terms of CPU usage for the parallelism schemes discussed in the previous section is illustrated in Figure 6.14. We observe that the third scheme, which was proven to be the best in reducing communication overhead, does not allocate resources very well, since it puts most of the load on worker 3, which is responsible for the HSI branch, while workers 2 and 4 don't utilize even 50% of their available CPU, similar to our baseline case. Surprisingly, the second method performs slightly better, as in this case Worker 2 and Worker 4 utilize 65% and 85% of their CPU respectively, with only worker 3 underperforming. However, this scheme suffers from high Wait CPU and there are many instances where the cluster waits for resources to be freed, in order to continue training our

network. Hence, both of these schemes do not exploit the resources of our cluster in an optimal way. For the sake of improving our resource utilization we present another model parallelism scheme, illustrated in figure 6.15.

In this distributed architecture, we split the HSI branch in three parts. The

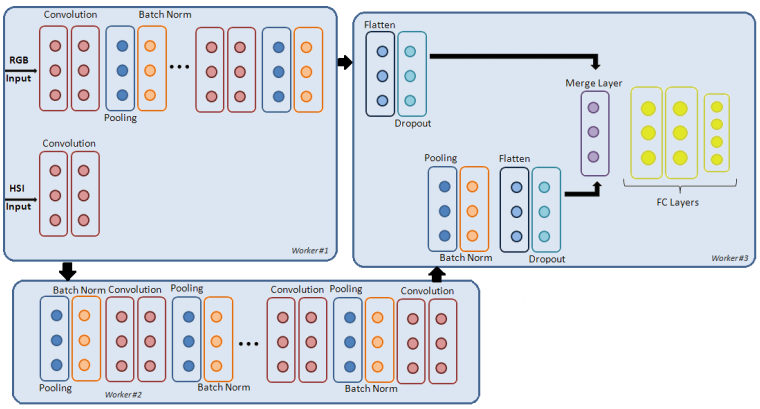


Figure 6.15: Third Model Parallelism Scheme.

first part consists of the first two convolution layers and is migrated on the worker responsible for the VHR-RGB branch of our CNN. The second part contains the rest of the layers except the last and is located on its own worker. Finally, the third part which has the final pooling and batch normalization layers, as well as the flatten and dropout layers, is put on the merging worker, along with the dropout and flatten layers of the VHR-RGB branch. The way this scheme affect our cluster's traffic is depicted in Figure 6.16.

The plots in Figure 6.18 show that this time our network behavior is different when compared with our previous scheme. Traffic is not anymore clustered between the PS and the merging worker, even though they still send and receive the biggest amount of data, but is scattered among all the machines of the cluster. This is also reflected in more peaks in both incoming and outgoing traffic , as well as the higher values of traffic in general.

In the next plot one easily deduce that this architecture provides better resource exploitation. The workload is more balanced between the machines of the cluster and there is more than 50% healthy CPU utilization in every worker. The results also present that the HSI worker, in our case worker 3, shared some of its load, since its CPU usage caps at around 70% instead of around 80%, which was the case previous architectures. Wait CPU plot reports mostly low values. Wait CPU on worker 4 has some instances that goes over 10% but most of them are gathered around 5%, which is still manageable for our cluster.

In an attempt to attain even better resource utilization, we present a variation of the previous scheme, illustrated in Figure 6.18. In this variation, we once again split the HSI branch in three parts. The first part now consists of the first convolution-pooling-batch normalization block and is again migrated on the worker responsible

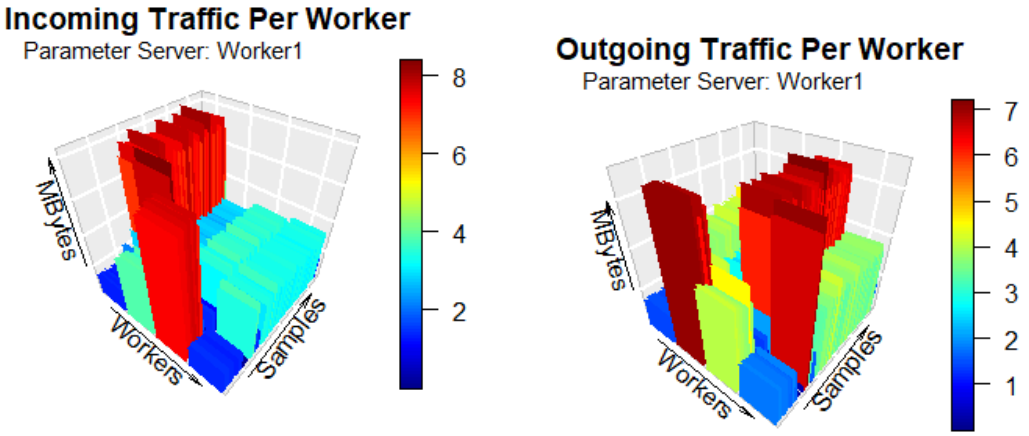


Figure 6.16: Communication traffic per worker of the architecture of Figure 6.15

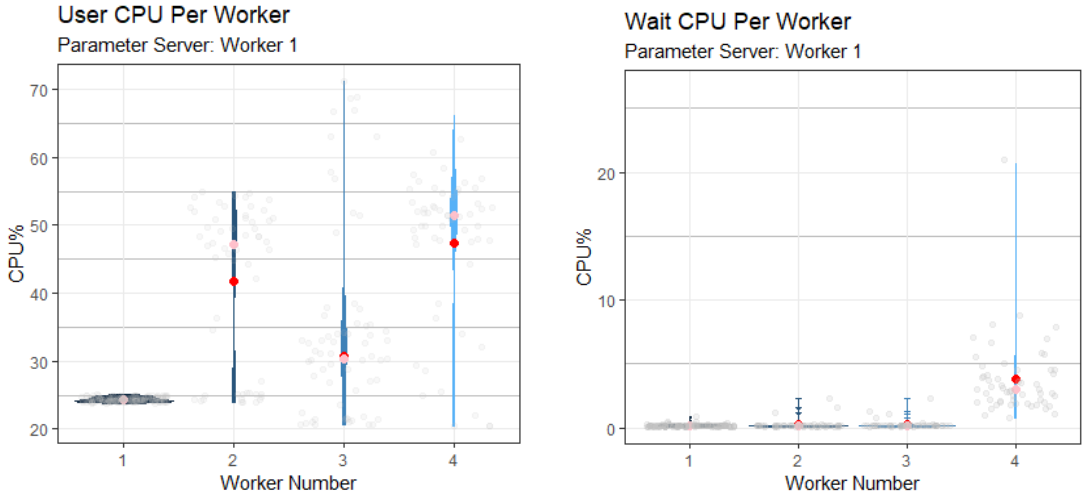


Figure 6.17: CPU Usage per worker of the architecture of Figure 6.15

for the VHR-RGB branch of our CNN. The second part contains the rest of the layers except the last convolution-pooling-batch normalization block and is again, located on a sole worker. This block, as well as the flatten and dropout layers, is now located on the merging worker, along with the dropout and flatten layers of the VHR-RGB branch. The way this scheme affect our cluster's traffic is depicted in Figure 6.18.

The results of this experiment show that outgoing traffic behaves similarly

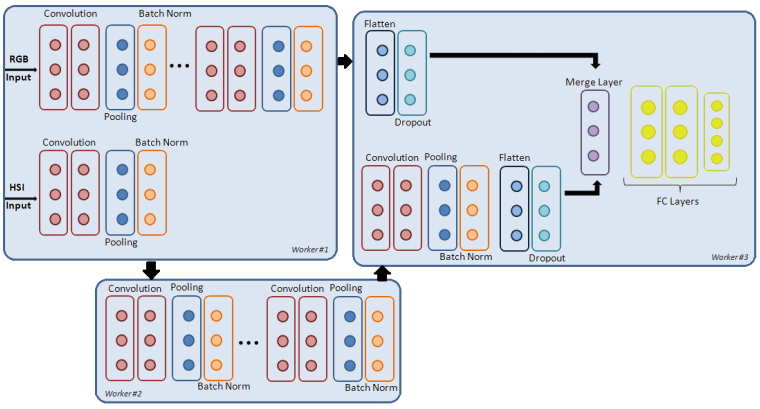


Figure 6.18: Fourth Model Parallelism Scheme.

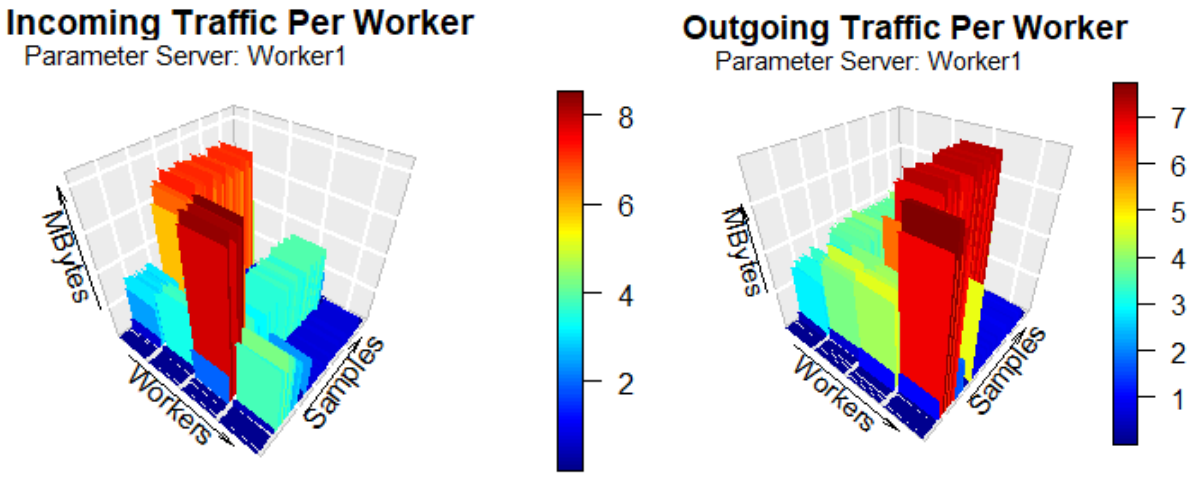


Figure 6.19: Communication traffic per worker of the architecture of Figure 6.18

to our previous scheme. It is once again scattered among the machines of our cluster, since the way our model is split split requires bigger volumes of data to be transferred between workers. Incoming traffic follows a similar trend, however traffic dispersion is not as prominent in this case. The most of incoming data are delivered on the machines acting as the PS and the merging worker and the rest of the workers receive bigger volumes of data but on fewer time instances. The lower number of peaks on workers 2 and 4 depicted in Figure 6.19 (left) further confirm this claim.

The results of Figure 6.20 demonstrate that this scheme achieved a noticeable improvement in CPU usage. As expected, the load is almost equally shared between workers supervising parts of the CNN, and each machine achieves more than 55%

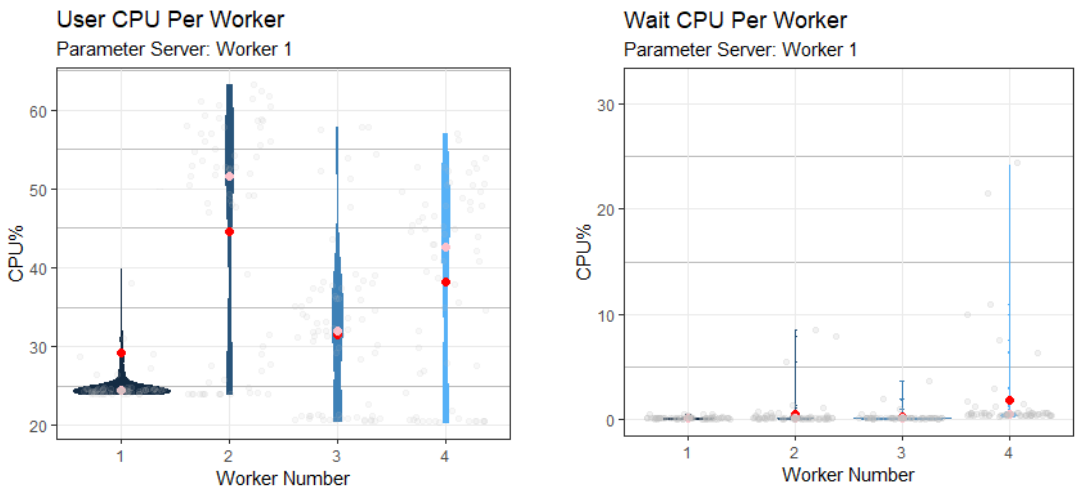


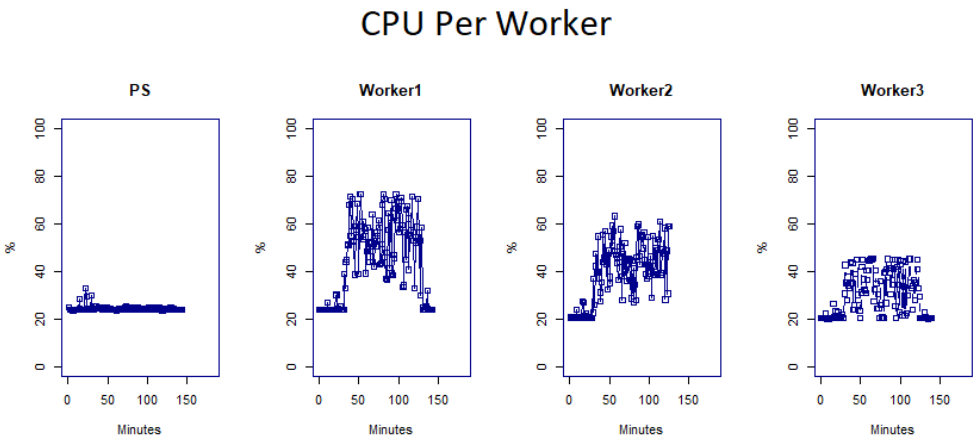
Figure 6.20: CPU Usage per worker of the architecture of Figure 6.18

CPU utilization. This time the worker responsible for the VHS-RGB branch has the instances that attain the highest Wait CPU values, pointing to the fact that trying to migrate different parts of each branch to the same device can increase waiting times, predominantly on the initialization phase of parallelization. To conclude, even though this model parallelism scheme causes bigger network traffic than the ones we studied in section 6.3.3, it still causes less bottlenecks than our initial proposed architecture. In addition it provides the most optimal cluster utilization out of every model parallelism scheme presented in this work.

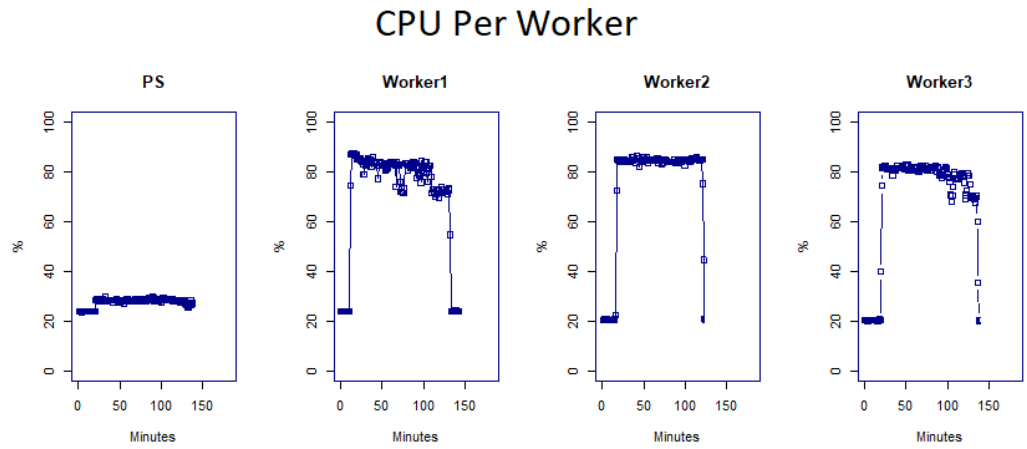
6.3.5 Model And Data Parallelism Comparison

In this final section, we will briefly compare our method with a data parallelism approach. Specifically, we will comment on the differences between resource utilization, considering metrics of CPU usage, memory usage and network traffic. Two set of experiments were conducted in order to come to a satisfying conclusion. In the first experiments we run a data parallelism scheme using the same 8K patches we used for the experiments described on the previous sections. In the second set, we run both parallelization schemes with a dataset consisted on 2.5K new image patches from the dataset described on section 6.1. Since we didn't perform any form of optimization in our data parallelism approach, we opted to use the model parallelism approach discussed in section 6.3.1 for this section.

Figures 6.21 and 6.22 present the performance of both Model and Data Parallelism on our cluster as a function of CPU usage. Figure 6.21 presents the results of the 2.5K image patches set, while Figure 6.22 provides the results for the 8K



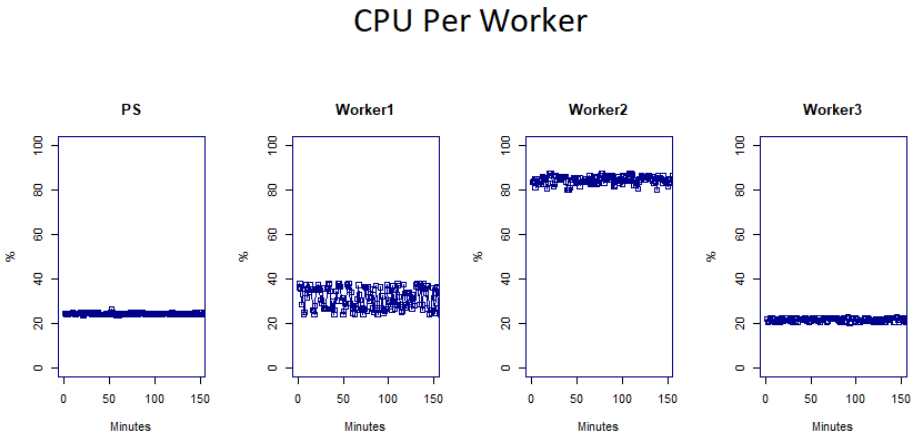
(a) 2.5K samples set, Model Parallelism.



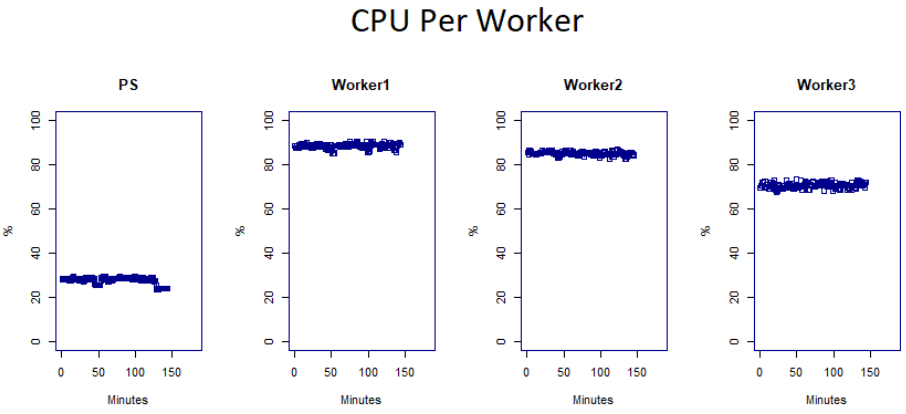
(b) 2.5K samples set, Data Parallelism.

Figure 6.21: CPU usage of both Parallelization Approaches.

samples set. We notice that for both datasets, Model Parallelism is the less computationally expensive approach, which is expected, given the fact that we isolate the branch of the network with the highest complexity to a single machine. In data parallelism on the other hand, this branch resides in every worker that is not the PS, thus causing every worker to allocate more CPU. In all cases, the Parameter Servers of both approaches have similar CPU consumption.



(a) 8K samples set, Model Parallelism.

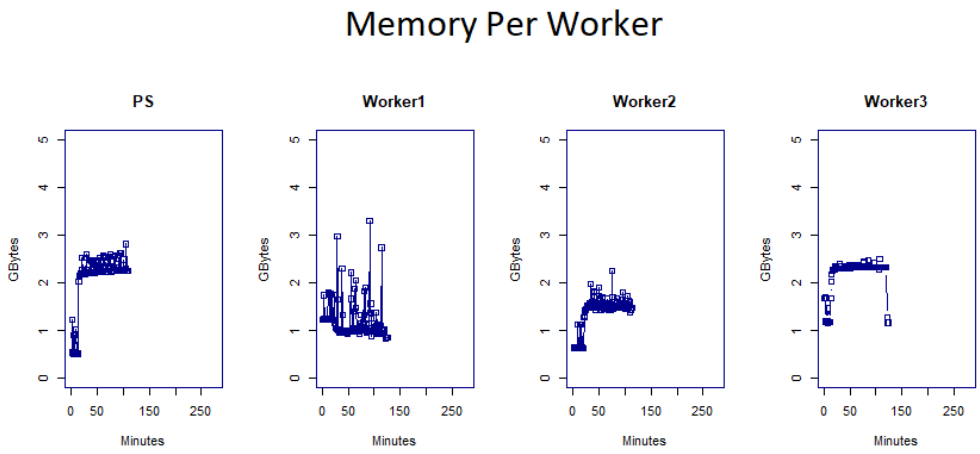


(b) 8K samples set, Data Parallelism.

Figure 6.22: CPU usage of both Parallelization Approaches.

Figures 6.23 and 6.24 present the performance of both Parallelization approaches on our cluster as a function of Memory usage. Figure 6.23 illustrates the results of the 2.5K image patches set, while Figure 6.24 presents the results for the 8K samples set. For both approaches, it is obvious that the more image patches we include in our training data the more memory our worker will need to allocate for our experiments. However, once again Model Parallelism is the approach that

requires the least amount of resources, since each device has to operate on parts of our CNN, instead of the whole model, as it happens on the Data Parallelism Approach. Once again, the Parameter Server of both Model and Data Parallelism behave in a similar manner.

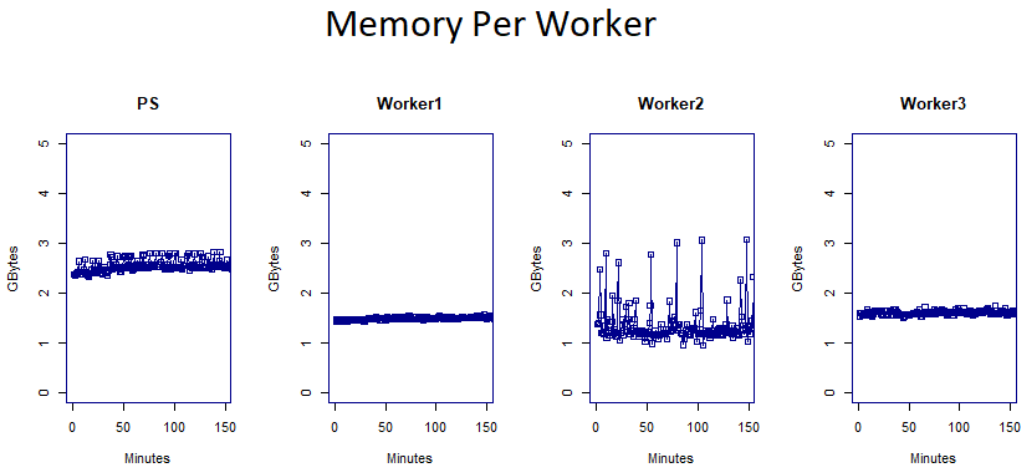


(a) 2.5K samples set, Model Parallelism.

(b) 2.5K samples set, Data Parallelism.

Figure 6.23: Memory usage of both Parallelization Approaches.

Figures 6.25 and 6.26 demonstrate the impact of both Parallelization approaches on the Incoming Traffic of our cluster. Figure 6.25 presents the results of the 2.5K image patches set, while Figure 6.26 illustrates the results for the 8K samples set. This time, the size of the training set does not affect at all the amount of data each worker receives, and this is true for both approaches. As expected for Data Parallelism, the Parameter Server receives a big amount of data, since in these

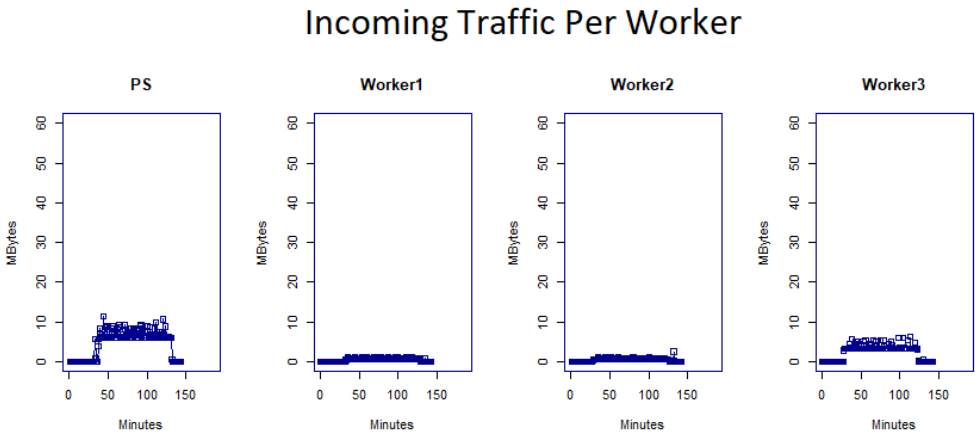


(a) 8K samples set, Model Parallelism.

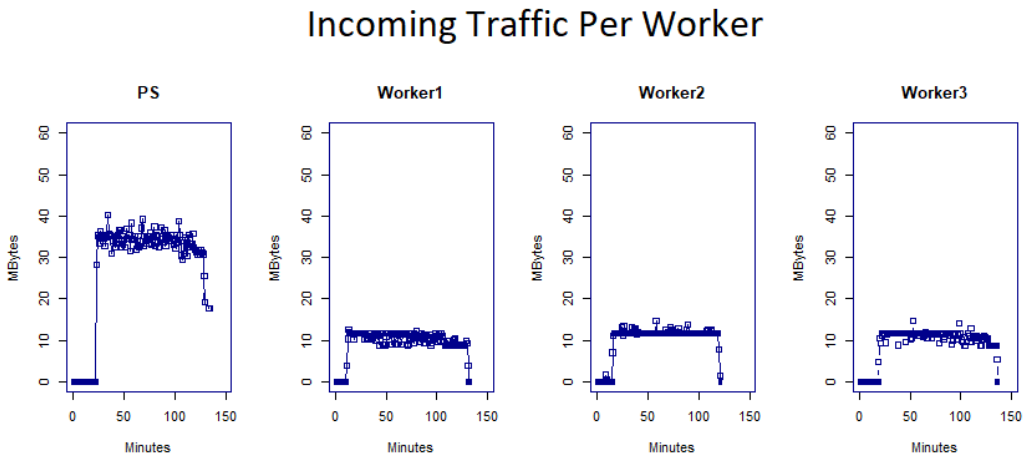
(b) 8K samples set, Data Parallelism.

Figure 6.24: Memory usage of both Parallelization Approaches.

approaches each worker has to send their gradients on the PS, so it can average and update them. The rest of the workers have a similar amount of incoming traffic, since the Parameter Server has to send the updated parameters to every one of them. By observing these results, one can assume that Model Parallelism is yet again the more favorable approach of the two, which is true for this use case analysis. However, it should be taken into consideration that in Data Parallelism approaches most traffic occurs at the end of each epoch, when the workers want to send their weights on the PS. Model Parallelism approaches though, create layer interdependencies, which means that machines must communicate more frequently, and while in our case it is not a matter of concern, it can cause serious bottlenecks for other DNN architectures.



(a) 2.5K samples set, Model Parallelism.



(b) 2.5K samples set, Data Parallelism.

Figure 6.25: Incoming Traffic of both Parallelization Approaches.

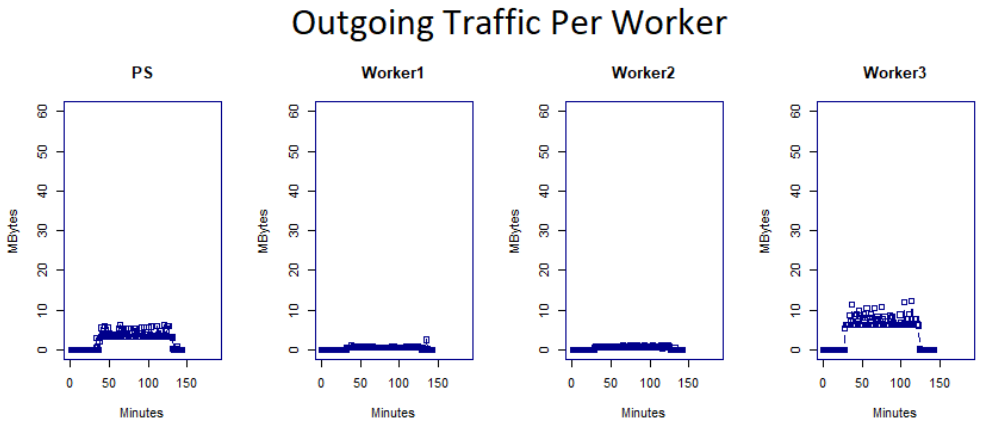
(a) 8K samples set, Model Parallelism.

(b) 8K samples set, Data Parallelism.

Figure 6.26: Incoming Traffic of both Parallelization Approaches.

Figures 6.27 and 6.28 demonstrate the impact of both Parallelization approaches on the Incoming Traffic of our cluster. Figure 6.27 presents the results of the 2.5K image patches set, while Figure 6.28 illustrates the results for the 8K samples set. Once again, the size of the training set does not affect at all the amount of data each worker receives, and this is true for both approaches. For Data Parallelism, Outgoing traffic's behavior is similar to Incoming Traffic's. Once again, the Parameter Server sends the largest amount of data, since it has to dispatch the updated gradients to the workers, while the workers in turn send their new results back to the PS after the end of each epoch. For Model Parallelism, most Outgoing traffic is gathered between the PS and the merging Worker. The remarks for Incoming

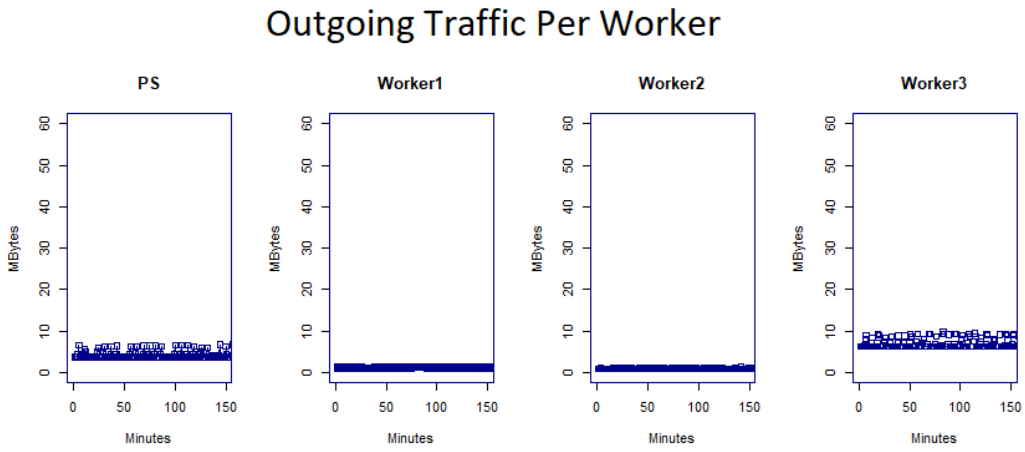
Traffic can also be applied here. Even though Model Parallelism is the better approach for our study, it can be a less appealing choice in other scenarios due to their layer interdependencies.



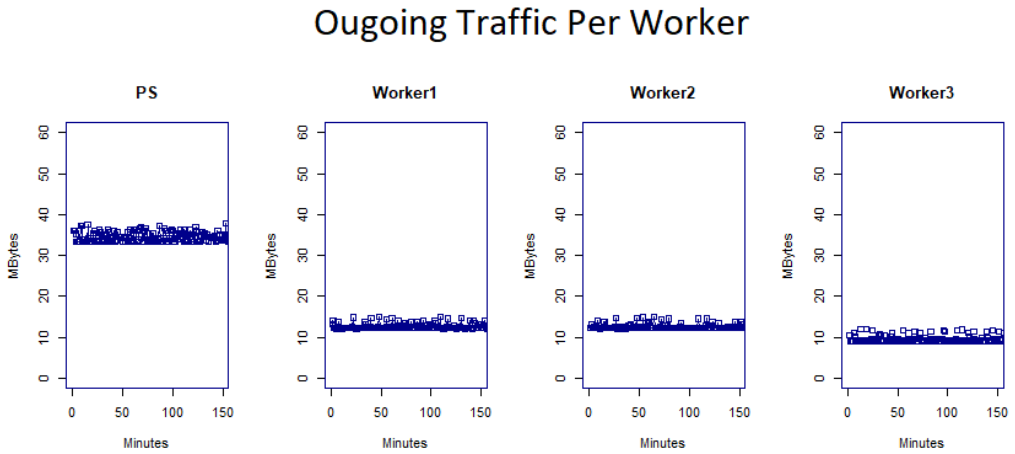
(a) 2.5K samples set, Model Parallelism.

(b) 2.5K samples set, Data Parallelism.

Figure 6.27: Outgoing Traffic of both Parallelization Approaches.



(a) 8K samples set, Model Parallelism.



(b) 8K samples set, Data Parallelism.

Figure 6.28: Outgoing Traffic of both Parallelization Approaches.

Chapter 7

Conclusions

Regarding the first part of our work, our proposed setup implements data parallelism, in tandem with asynchronous distributed training for the problem of spectroscopic redshift estimation in astronomy. We conducted experiments with noisy simulated data, and our results provide the following contributions:

- Distributed CNNs have a similar behavior to traditional setups, in the sense that larger amount of training samples has a positive effect on the system's learning capacity. As a result, it is safe to assume that transporting a DNN to a distributed environment does minimal damage on the network's overall performance.

- Multiple workers have a positive effect on our distributed CNN, since they help our network to converge much faster than the sequential case. However, they also add considerable overhead to the network, due to the higher number of transfers and bottlenecks caused by the PS.

- In terms of accuracy, the impact of data distribution becomes less important the higher the number of training node a cluster has. For fewer number of nodes though, our experiments showed better performance when each node has access to the same amount of training data.

- However, when we conduct experiments in terms of loss, data distribution has high impact regardless of cluster size. In this case, for different data per worker the cluster converges faster the more nodes we add, while for same data the differences are minuscule no matter the cluster size.

- Furthermore, we concluded that for distributed training to be worthwhile, the computation benefit of multiple machines, has to outweigh the introduced overheads. This usually happens when training time on a single machine becomes extremely large, either due to network complexity or large amounts of data.

In the second case study, we proposed a multimodal CNN for land cover classification and performed model parallelism architectures, in order to improve its performance on a distributed environment. We conducted experiments with patches extracted from RGB and HSI images, and our results provide the following contributions:

- As with the data parallelism case, migrating branches of multimodal CNNs to different machines does not affect the model's learning capacity, and a distributed deep learning system can show similar behavior to traditional setups.
- The number of samples that will be propagated through the network greatly affects communication overheads between workers of a cluster. Model Parallelism approaches benefit from larger batch sizes, since they significantly reduce network traffic. However, a very big number of samples causes significant degradation in the quality of the model.
- Through this study, we gave insight on model parallelism's behavior, and as a result, provide directions for better resource allocation on commodity clusters. For instance, since the PS is the most memory consuming module of model parallelism, we can assign it to the machine with the most memory available. Or because the merging part of a multimodal CNN accumulates a lot of traffic, we can assign it to the machine with the most expensive networking cables.
- The way we parallelize our model has a tremendous impact on a cluster's resource allocation. Smart model split can release the load of heavy duty workers and put to use otherwise idle resources, while a naive split can cause severe bottlenecks through heavy network traffic and waiting learning tasks due to a lack of unallocated resources.

Bibliography

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. Nature 521, 7553 (2015), 436-444.

[2] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In NIPS, pp. 1106-1114, 2012.

[3] Graves, Alex, Mohamed, Abdel-rahman, and Hinton, Geoffrey. Speech recognition with deep recurrent neural networks. In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pp. 6645-6649. IEEE, 2013.

[4] Esteva, A., Kuprel, B., Novoa, R., Ko, J., Swetter, S., Blau, H., and Thrun, S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. Nature, 542(7639):115-118.

[5] Chen, C., Seff, A., Kornhauser, A.L., Xiao, J.: DeepDriving: Learning affordance for direct perception in autonomous driving. In: ICCV (2015)

[6] J. B. Heaton, N. G. Polson, J. H. Witte, Deep learning in finance, arXiv preprint arXiv:1602.06561 (2016).

[7] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. Neural Comput. 1, 4 (Dec. 1989), 541-551.

[8] K. Simonyan, A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.

[9] V. Hegde, S.Usmani. Parallel and Distributed Deep Learning

[10] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, A. Strehl, and V. Vishwanathan. Hash kernels. In AISTATS, 2009.

[11] R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In NAACL, 2010.

[12] Y. Bengio. 2013. Deep Learning of Representations: Looking Forward. In Statistical Language and Speech Processing, SLSP. Proceedings.

[13] S. Lee, J.K. Kim, X. Zheng, Q.Ho, G.A Gibson, and E. P Xing, "On model parallelization and scheduling strategies for distributed machine learning," in Advances in neural information processing systems, 2014.

[14] M. Zinkevich, M. Weimer, L. Li, and A.J Smola, "Parallelized stochastic gradient descent," in Advances in neural information processing systems, 2010

[15] U. A. Muller and A. Gunzinger. 1994. Neural net simulation on parallel computers. In Neural Networks, IEEE International Conference on, Vol. 6.

[16] L. Ericson and R. Mbuvha. 2017. On the Performance of Network Parallel Training in Artificial Neural Networks.

[17] S. Lee, S. Purushwalkam, M. Cogswell, D. J. Crandall, and D. Batra. 2015. Why M Heads are Better than One: Training a Diverse Ensemble of Deep Networks.(2015).

[18] R. Raina, A. Madhavan, and A. Y. Ng. 2009. Large-scale Deep Unsupervised Learning Using Graphics Processors. InProc. 26th Annual International Conferenceon Machine Learning.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Scorrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016

[20] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Cor-rado, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. InICML, 2012

[21] Mu Li.Scaling Distributed Machine Learning with System and Algorithm Co-design. PhD thesis, Carnegie Mellon University, 2017.

[22] M. Li, D. G Andersen, J.W. Park, A. J Smola, A. Ahmed, V. Josifovski, J. Long, E. J Shekita, and B. Su, "Scaling distributed machine learning with the parameter server," in OSDI, 2014.

[23] D. Das, S. Avancha, D. Mudigere, K. Vaidyanathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," CoRR, 2016.

[24] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li. 2010. Parallelized Stochastic Gradient Descent. InProc. 23rd International Conference on Neural Information Processing Systems.

[25] J. Dean and S. Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters.Commun. ACM51, 1.

[26] J. Jiang, B. Cui, C. Zhang, and L. Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. InProc. 2017 ACM International Conference on Managementof Data.

[27] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng. 2011. On Optimization Methods for Deep Learning. InProc. 28th International Conferenceon Machine Learning.

[28] A. Gaunt, M. Johnson, M. Riechert, D. Tarlow, R. Tomioka, D. Vytiniotis, and S. Webster. 2017. AMPNet: Asynchronous Model-Parallel Training for DynamicNeural Networks.

[29] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. 2013. Asynchronous stochastic gradient descent for DNN training. In IEEE International Conference on Acoustics, Speech and Signal Processing.

[30] A. Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks.

[31] A. Krizhevsky, I. Sutskever, and G. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems 25.

[32] J. Dean et al. 2012. Large Scale Distributed Deep Networks. InProc. 25th International Conference on Neural Information Processing Systems - Volume 1.

[33] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and timeseries. The handbook of brain theory and neural networks, 3361(10):1995, 1995.

[34] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[35] A. van den Oord, S. Dieleman, and B. Schrauwen. Deep content-based music recommendation. In Proceedings of Neural Information Processing Systems (NIPS). 2013.

[36] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: deep neural networks with multitask learning. In Proceedings of the 25th international conference on Machine learning, pages 160-167. ACM, 2008.

[37] Nadav Cohen, Or Sharir, and Amnon Shashua. 2016b. On the Expressive Power of Deep Learning: A Tensor Analysis. In 29th Annual Conference on Learning Theory (Proceedings of Machine Learning Research), Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir (Eds.), Vol. 49. PMLR, Columbia University, New York, New York, USA, 698-728.

[38] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International Conference on Machine Learning.

[39] Snoek, C., Worring, M., and Smeulders, A. (2005). Early versus late fusion in semantic video analysis. In Proceedings of the ACM international conference on multimedia.

[40] J. Ngiam, A. Khosla, Mi. Kim, J. Nam, H. Lee, and Andrew Y. Ng. Multimodal deep learning. InInternational Conference on Machine Learning (ICML), Belle-vue, USA, June 2011.

[41] E. Morvant, A. Habrard, and S. Ayache. Majority vote of diverse classifiers for late fusion. In Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition and Structural and Syntactic Pattern Recognition (SSPR), 2014.

[42] E. Shutova, D. Kiela, and J. Maillard. Black holes and white rabbits: Metaphor identification with visual features. In Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.

[43] A. Eitel, J. T. Springenberg, L. Spinello, M. Riedmiller, andW. Burgard. Multimodal deep learning for robust rgb-dobject recognition. InIEEE/RSJ International Conference onIntelligent Robots and Systems (IROS), Hamburg, Germany,2015

[44] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, et al. Apache Spark: A Unified Engine for Big Data Processing. In: Commun. ACM 59.11 (2016).

[45] X. Lu, H. Shi, R. Biswas, M. Haseeb Javed, and D. K Panda, "Dlobd: A comprehensive study of deep learning over big data stacks on hpc clusters," IEEE Transactions on Multi-Scale Computing Systems, 2018.

[46] C. Jia, J. Liu, X. Jin, H. Lin, H. An, W. Han, Z. Wu, and M. Chi, "Improving the performance of distributed tensorflow with RDMA," International Journal of Parallel Programming, 2018.

[47] M. Saouabi and A. Ezzati, "A comparative between hadoop mapreduce and apache spark on HDFS," in Proceedings of the 1st International Conference on Internet of Things and Machine Learning, IML 2017.

[48] Matthew L. Massie, Brent N. Chun, and David E.Culler. The ganglia distributed monitoring system:Design, implementation, and experience. Parallel Computing, 2004.

[49] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. InProc. International Conference on Learning Representations (ICLR).

[50] Yang, J., Wang, Z., Lin, Z., Cohen, S., Huang, T.: Coupled dictionary training forimage super-resolution. TIP 21(8), (2012)

[51] E. Loupias, N. Sebe, S. Bres, and J. Jolion, "Wavelet-basedSalient Points for Image Retrieval," ICIP 2000, vol. 2, Vancouver, Canada, Sept. 2000.

[52] D. Wu, L. Pigou, P.-J. Kindermans, N. D.-H. Le, L. Shao, J. Dambre, and J.-M. Odobez, "Deep dynamic neural networks for multimodal gesture segmentation and recognition," IEEE Transactions on Pattern Analysisand Machine Intelligence, 2016.

[53] M. Ren, R. Kiros, and R. Zemel. Exploring models and data forimage question answering. InNIPS, 2015

[54] Vinyals, O., Toshev, A., Bengio, S. nad Erhan, D. Show and tell: a neural image caption generator. In Proc. International Conference on Machine Learning (2014).