Distributed Training and Inference of Deep Learning Models For Multi-Modal Land Cover Classification

Maria Aspri 1,2, Grigorios Tsagkatakis 2,* and Panagiotis Tsakalides 1,2

1 Institute of Computer Science, Foundation for Research and Technology-Hellas (FORTH), GR70013 Heraklion, Greece; aspri@ics.forth.gr (M.A.); greg@ics.forth.gr (G.T.); tsakalid@ics.forth.gr (P.T.)

2 Computer Science Department, University of Crete, Greece,GR70013 Heraklion, Greece

* Correspondence: greg@ics.forth.gr; Tel. +30-2811-392725


Abstract: Deep Neural Networks (DNNs) have established themselves as a fundamental tool in numerous computational modeling applications, having bypassed the critical step of feature extraction. Despite their capabilities in modeling, training large-scale DNN models is a very compute-intensive task that most single machines are often incapable to accomplish. In order to address this issue, different parallelization schemes have been proposed. Nevertheless, network overheads as well as optimal resource allocation pose as major challenges, since network communication is generally slower than intra-machine communication while some layers are more computationally expensive than others. In this work, we consider a novel multimodal DNN based on the Convolutional Neural Network architecture and explore several different ways to optimize its performance when training is executed on an Apache Spark Cluster. We evaluate the performance of different architectures via the metrics of network traffic and processing power, considering the case of land cover classification from remote sensing observations. Furthermore, we compare our architectures with an identical DNN architecture modeled after a data parallelization approach by using the metrics of classification accuracy and inference execution time. The experiments show that the way a model is parallelized has tremendous effect on resource allocation and hyperparameter tuning can reduce network overheads. Experimental results also demonstrate that proposed model parallelization schemes achieve more efficient resource utilization and more accurate predictions compared to data parallelization approaches.

Keywords: Distributed Deep Learning; Model Parallelization; Convolutional Neural Networks; Multi-modal observation classification; Land Cover Classification;

1. Introduction

Deep Learning models, and more specifically Deep Neural Networks (DNNs) with multiple layers between the input and output layers, are becoming a pervasive tool in modern data analysis applications due to their efficiency in learning appropriate data representations [1,2]. When trained with large amounts of data, DNNs provide extremely accurate solutions and as a result, they have successfully been considered in a plethora of fields, including remote-sensing [3,4], speech recognition [5] and computer vision [6] among others.

Graphical Processing Units (GPUs) are the ideal commodity hardware for training DNNs since they were developed to deal with parallel computations and have large memory bandwidth. Even though significant advances have been made in both GPU hardware and DNN architectures, the fact remains that, as datasets increase in size and network architectures increase in complexity, DNNs require even more resources in both computational power and memory demand. As a result, both training and inference on large-scale models to competitive accuracy, essentially require a cluster of
multiple machines connected by a network. Training under a distributed paradigm is also appropriate
when dealing with Big Data, such as the ones associated with Earth Observation [7]. In such cases, and
especially when considering scenarios involving multiple sources of observations, each stored at a
different location, transferring of all required data into a single location can be very challenging, even
prohibiting for very large volumes [8,9].

Subsequently, Distributed Deep Learning has received considerable attention, and many
approaches and frameworks have been proposed and implemented [10], making various distributed
and parallel computing schemes applicable to a variety of problems that have been previously
addressed by sequential approaches. The performance of those approaches is however highly
dependant not only on the connectivity structure and the computational needs of the model, but also on
the cluster’s characteristics. For example, DNN layers with local connectivity structures, like the one in
Convolutional Neural Networks (CNNs), have generally low communication requirements, and tend
to be more compliant to distribution, than fully connected architectures like Multi-Layer Perceptron.
Moreover, processing times between different machines tend to differ, causing bottlenecks when
transcending to different computation phases and leading to non-ideal speedups. Intercommunication
is also a major factor to take into consideration, since different network technologies provide different

Motivated by this, we explore the optimization of a distributed multi-modal CNN architecture
and evaluate its performance in multi-class land cover classification, inspired by the fact that CNNs are
proven to be highly effective on such scenarios [12,13]. We explore scenarios of model parallelization
as our distributed approach and consider different parallelization architectures in order to understand
which one is the most appropriate for model distribution. Moreover, we compare our architecture with
another popular distributed deep learning approach called data parallelization [14,15], and provide
experimental results on each approach’s ability to make fast and accurate predictions. As a result, our
study makes the following contributions:

- A detailed analysis on the trade-offs between different parallelization architectures, focusing on
  the case of remote sensing data analysis.
- Formulation of a novel distributed processing model for the case of multi-modal remote sensing
  observation classification.
- Comparison between model and data distributed DNN approaches for the problem of
  multi-modal data classification.
- Insights into model parallelization behavior, providing directions on better allocation of the
  resources of computer clusters.

2. State of the art

2.1. Distributed Deep Learning and Model Parallelization

Several methods have been proposed for parallelizing and distributing a DNN model[16,17] as it
is possible to partition both the forward evaluation and the backpropagation phases among parallel
processors and across different machines. In general, two main classes of distributed processing have
been considered, namely data parallelism, model parallelism and hybrid approached [18].

In model parallelization, each worker has a local copy of the entire network and is responsible for
a different part of the processing pipeline. In model parallelism, different machines are responsible for
the computations in different parts of the network. In this case, the weights are split equally among the
machines, and all of them train on the same mini batch. The general idea behind hybrid parallelism is
to combine different parallelization schemes in order to overcome the drawbacks of each other.

The fact that the DNN is not stored in a single device helps in memory conservation, however
it also incurs additional communication after every layer. Considering this, it suffices to say that
the architecture of a DNN plays an important role in the overall performance, since it creates layer
inter-dependencies, which, generate a significant amount of communication overhead. In order to
resolve this issue, the authors in [19] propose a method for partitioning a DNN in such a way that each machine will be responsible for twice the number of its neurons. The authors in [20] propose an alternative method, in which they use Cannon’s matrix multiplication algorithm in order to reduce communication between fully connected layers, but it only produces better efficiency and speedups over simple partitioning on small-scale fully connected networks.

In contrast of model parallelism, in data parallelization every worker receives a complete copy of the model and its parameters. The dataset is distributed among the machines through different partitions, which train the copies locally. Each copy 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. 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 [21].

Since data parallel approaches train a copy of the entire model on individual computer systems, they require some method of combining results and synchronizing the model parameters between machines. The prominent examples of parameter synchronization involve synchronous and asynchronous training, while coordination is facilitated by a machine 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, its 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].

Regarding hybrid parallelism, the general idea behind it 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 [2], in which they apply data parallelism to convolutional layers, and model parallelization 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 [31] distributed training is performed simultaneously on multiple model replicas, while each replica is trained on different data samples.

An earlier version of this work was presented in [32]. That work differs from this one in that we considered a different type of problem, namely spectroscopic redshift estimation from astrophysical observations, and we considered the scenario of data-parallelism for training a CNN model.

2.2. Deep Learning in Land Cover Classification

In the last few years the availability of Earth observation satellites has been significantly increased, providing an extremely large number of geographical images, covering almost every corner of the Earth’s surface. As a result, data volumes of remote sensing images get increased on a daily basis producing, among others, images with different spatial and spectral resolutions [33].

Regarding land cover classification, many approaches have been proposed, which mostly exploit handcrafted features generated by remote sensing images [34,35]. However, it is difficult for handcrafted features to consider the details of real data, thus rendering those approaches unable to achieve an optimal balance between discriminability and robustness. This challenge gets even worse when taking the data volumes of remote sensing images into consideration, since the same image can change multiple times in a short interval.
A DNN on the other hand, with its ability to automatically learn hierarchical intermediate representations [2,36], has been established as the next trend in the remote sensing classification [37]. As a result, a considerable amount of studies related to deep learning for remote sensing image analysis have been published in the past years, with many of them focusing on unsupervised generative graphical models, such as Deep Belief Networks (DBNs) or Sparse AutoEncoders (SAEs). The authors in [38] tested a DBN on urban land use, using polarimetric SAR (polSAR) data. Meanwhile, the authors in [39] applied a stacked SAE to PolSAR image classification and those in [40] applied adaptive boosting of restricted Boltzmann machines to PolSAR image classification. CNNs are also extremely popular in literature for land cover classification approaches. The authors in [41] applied CNNs to PolSAR image classification. In [42], the authors replace pooling layers in their CNN with wavelet-constrained pooling layers. The Neural Network is then used in conjunction with superpixels and a Markov random field to produce the final segmentation map.

One important thing to note is that remote sensing data are often provided in different modalities, spectrally, spatially and temporally inhomogeneous. As a result, data fusion is becoming of significant importance in remote sensing classification. Data fusion is based on the exploitation of complementary information of data originating from different sources. Current works of remote sensing dealing with DNNs and data fusion mostly revolve around two strategies: early fusion and late fusion. Early fusion uses multiple modalities simultaneously in a Neural Network, usually by concatenating extracted features of each data source into a single data cube. The authors in [43] extract different types of spatial filters and concatenate them in a single tensor, which is then used to learn a supervised stack of AEs. Furthermore, the authors in [44] train a two-stream CNN that process PolSAR and hyperspectral data in parallel while fusing the resulting information at a later convolutional layer.

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. Moreover, these approaches are able to fuse predictions from different models, producing more robust and accurate results. For example, the authors in [45] fuse predictions obtained by a CNN with ones obtained by a random forest classifier, trained using handcrafted features. However, the authors in [46] propose a different approach in late fusion, in which they fuse two decision maps obtained by pre-trained models by learning a fusion network based on residual learning.

Last, we note a very recent work which explores in great detail the benefits of distributed deep learning for remote sensing applications [7]. Our work is different in that we focus on a particular case, that of multi-modal classification, and explore the tradeoffs between model and data parallelism.

3. Proposed methodology

3.1. Overall CNN architecture

In this work, we consider a CNN architecture that efficiently combines features from multiple modalities by employing a two-stream CNN architecture for assigning each pixel, processed at a patch level, to one specific class. The proposed architecture consists of a sequence 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, fusing the features before the first fully connected layer.

The proposed multi-modal CNN consists of three modules, one for processing HSI data, one for processing VHR-RGB data, and one for feature fusion and classification. The first two modules apply 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 2D models fail to accurately encode.

Due to the fact that CNNs require a considerable amount of data in order to ensure that sufficient generalization is achieved during training, the following preprocessing steps are applied in our dataset
before going ahead with any learning or classification. Firstly, the four separate VHR-RGB tiles had to be merged, as well as resized along with the HSI image, to match the spatial size of the ground truth data. Then, 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. Furthermore, for each class, we randomly chose 400 distinct patches from each image with a selected pixel as a starting point, resulting in overall 8000 (8K) patches. The proposed processing pipeline is depicted in Figure 1 while the specific parameters of the network are shown in Figure 2.

Figure 1. Processing pipeline for the proposed multimodal CNN.

<table>
<thead>
<tr>
<th>VHR branch</th>
<th>HSI branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv3D: #filter =2 + ReLU</td>
<td>Conv3D: #filter =8 + ReLU</td>
</tr>
<tr>
<td>Conv3D: #filter =4 + ReLU</td>
<td>Conv3D: #filter =16+ ReLU</td>
</tr>
<tr>
<td>Max-Pooling</td>
<td>Max-Pooling</td>
</tr>
<tr>
<td>Batch Normalization</td>
<td>Batch Normalization</td>
</tr>
<tr>
<td>Conv3D: #filter =8 + ReLU</td>
<td>Conv3D: #filter =16+ ReLU</td>
</tr>
<tr>
<td>Conv3D: #filter =8 + ReLU</td>
<td>Conv3D: #filter =16+ ReLU</td>
</tr>
<tr>
<td>Max-Pooling</td>
<td>Max-Pooling</td>
</tr>
<tr>
<td>Batch Normalization</td>
<td>Batch Normalization</td>
</tr>
<tr>
<td>Conv3D: #filter =32 + ReLU</td>
<td>Conv3D: #filter =32+ ReLU</td>
</tr>
<tr>
<td>Conv3D: #filter =64 + ReLU</td>
<td>Conv3D: #filter =64+ ReLU</td>
</tr>
<tr>
<td>Max-Pooling</td>
<td>Max-Pooling</td>
</tr>
<tr>
<td>Batch Normalization</td>
<td>Batch Normalization</td>
</tr>
<tr>
<td>Conv3D: #filter =128 + ReLU</td>
<td>Conv3D: #filter =128+ ReLU</td>
</tr>
<tr>
<td>Conv3D: #filter =256 + ReLU</td>
<td>Conv3D: #filter =256+ ReLU</td>
</tr>
<tr>
<td>Conv3D: #filter =512 + ReLU</td>
<td>Conv3D: #filter =512+ ReLU</td>
</tr>
</tbody>
</table>

 Merge(VHR,HSI)  Flatten  Dropout(0.9)  
 Dense: \#filter =1000 + ReLU  
 Dense: \#filter =500 + ReLU  
 Dense: \#filter =20 +  
 Activation – SoftMax

Figure 2. Parameters of the multi-modal classification network.
3.2. Distributed CNN architectures

Figure 3 presents a high level block diagram of the two different schemes, showcasing the access to the same data by all workers for the case of data parallelism and different data for the proposed model parallelism scheme. To quantify the performance gains associated with each methods, we explore the differences in terms of both processing time and accuracy.

In order to fully exploit our cluster resources, we consider two different distributed deep learning architectures, based on the multimodal CNN described in subsection 3.1. As a result our CNNs are consisted of three branches, one for each modality and a final one for fusion and classification. For our initial architecture, we adopted a baseline approach, in which we assign each branch to a different worker, as illustrated in Figure 4.

For the second architecture, we took into consideration the traffic loads of our network. Thus we modified our previous model distribution into the one depicted in Figure 5. In this approach we
migrated the last convolutional-pooling block along with flatten and dense layers of both RGB and HSI branches. The rest of the layers remained on the same machines as before.

![Optimized architecture](image)

**Figure 5.** The optimized architecture. The architecture is similar to the "baseline" case, however, intermediate results from convolution layers are communicated among workers.

4. Distributed processing framework

4.1. System Overview

The model parallel architecture considered in this work is based on TensorFlowOnSpark\(^1\) [47], for managing TensorFlow-based distributed DNN workflows on Apache Spark clusters [14]. Apache Spark extends the MapReduce model [15] by the use of an elastic persistence model, which provides the flexibility to persist data records, either in memory, on disk, or both. Therefore, Spark favors the 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 an initial dataset.

TFoS exploits Spark’s native mechanisms for automatic cluster configuration while bypassing its communication philosophy, thereby allowing for direct tensor communication between TF processes. This framework also implements the concept of Parameter Server (PS) [48] for enabling the smooth integration of TensorFlow code over Spark clusters, with minimum changes on the original TensorFlow code. In a data parallelization approach, PS is responsible for providing a global average of network parameters, while the rest of the nodes are responsible for training, whereas in a model parallelization approach PS is responsible for distributing parts of a model to the nodes as well as acting as a training coordinator between the nodes.

4.2. 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\(^2\) 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 6 presents our cluster setup. It depicts Spark nodes connected via Ethernet. The Master configures the cluster through the driver, and Workers communicate through TFoS tensor communication. 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 traffic on every node [49].

5. Experimental Analysis

5.1. Dataset

The dataset used for our experiments was collected by NCALM at the University of Houston (UH) on February 16, 2017, covering the University of Houston campus and its surrounding areas. The dataset contains images of multi-source optical remote sensing data, containing (a) hyperspectral (HSI) data covering a 380-1050nm spectral range with 50 bands at a 1-m GSD, resulting in an image of size 601 x 2384 x 50; and (b) 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. A Ground truth image presented in Figure 7 was also provided as raster at a 0.5m GSD, and it is corresponding to 20 urban land use and land cover classes, listed in Table 1. Note that the dataset was developed in the context of the 2018 IEEE GRSS Data Fusion Contest "Advanced multi-sensor optical remote sensing for urban land use and land cover classification" [50].

Figure 7. Exemplary RGB (top) and land cover annotation (bottom).
Table 1. The original set of labels and their corresponding class.

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Corresponding Label</th>
<th>Class ID</th>
<th>Corresponding Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Healthy Grass</td>
<td>1</td>
<td>Stressed Grass</td>
</tr>
<tr>
<td>2</td>
<td>Artificial Turf</td>
<td>3</td>
<td>Evergreen Trees</td>
</tr>
<tr>
<td>4</td>
<td>Deciduous Trees</td>
<td>5</td>
<td>Bare Earth</td>
</tr>
<tr>
<td>6</td>
<td>Water</td>
<td>7</td>
<td>Residential Buildings</td>
</tr>
<tr>
<td>8</td>
<td>Non-Residential Buildings</td>
<td>9</td>
<td>Roads</td>
</tr>
<tr>
<td>10</td>
<td>Sidewalks</td>
<td>11</td>
<td>Crosswalks</td>
</tr>
<tr>
<td>12</td>
<td>Major thoroughfares</td>
<td>13</td>
<td>Highways</td>
</tr>
<tr>
<td>14</td>
<td>Railways</td>
<td>15</td>
<td>Paved Parking Lots</td>
</tr>
<tr>
<td>16</td>
<td>Unpaved Parking Lots</td>
<td>17</td>
<td>Cars</td>
</tr>
<tr>
<td>18</td>
<td>Trains</td>
<td>19</td>
<td>Stadium Seats</td>
</tr>
</tbody>
</table>

5.2. Result Evaluation

In this section, the results of our experiments are presented and discussed, in order to illustrate the performance of the model parallelization schemes introduced in this paper. More specifically, we first investigate the impact training under a distributed scenario in subsection 5.2.1 and subsequently investigate different parameters including the batch size in subsection 5.2.2. In subsection 5.2.3, we consider the two different schemes for model-parallelism training of the proposed multi-modal CNN and report the utilization of resources associated with each approach. Last, in subsection 5.3, we consider the case of model and data parallelism for the process of inference and compare these two approaches in terms of achieved processing time and resource utilization.

5.2.1. Effects of distributed environments on training time

In this section we compare the performance of the (baseline) distributed architecture with an identical multimodal CNN architecture deployed on a single machine setup. Figure 8 illustrates the performance of each architecture, when trained for 20 epochs each.

![Figure 8](image)

Figure 8. Training Accuracy with respect to elapsed training time.

The results demonstrate the advantage of the distributed architecture in terms of training accuracy, in contrast to the conventional single-processing-unit approach. More specifically, even though both implementations achieve comparable accuracy, the distributed training architecture attains higher accuracy significantly faster. We note that in both cases, the processing units are CPU which is not optimized for CNN training and therefore, significantly higher gains could potentially be achieved if GPUs were utilized.
5.2.2. Impact of Batch Size

The objective of this set of experiments was to examine if and how the hyperparameter of batch size affects model parallelization, motivated by the fact that since batch size defines the number of samples that will be propagated through the network, it will by extension affects the amount of data transfer over the network between different machines.

Figure 9 depicts both the incoming (left) and outgoing (right) traffic of the cluster for different batch sizes, during a time period of about 70 minutes, for the case of the "baseline" architecture. 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 the training process.

It should however be noted that by using a very large batch can lead to a significant degradation in terms of the model capabilities, as measured by its ability to generalize. This occurs due to the fact that large-batch methods tend to converge to local minima [51]. In conclusion, larger batch sizes can be beneficial for model parallelism approaches since they greatly reduce network overheads, as long as they do not negatively impact the model’s convergence.

![Figure 9. Incoming and Outgoing cluster traffic of the baseline architecture for different Batch Sizes.](image)

5.2.3. Impact of architectures on resource utilization

Even though batch size adjustments can significantly reduce communication overheads, further improvements are feasible by considering different allocations of the processing pipeline among workers, through different architectures. Figures 10 and 11, present the incoming and outgoing network traffic for the two architectural scenarios considered in the work, namely the "baseline", and the "optimized" respectively. In these plots, the horizontal axis corresponds to the individual worker, the diagonal axis to the sample batch and the vertical to the traffic measured in MBytes per second. From these plots, a number of observations can be made.

The first observation one can make is that for the case of the baseline architecture, bulk of the traffic is actually associated with two workers, worker 1 which acts as the PS, and worker 4 which is responsible for the merging and classification part of our multimodal CNN. The high incoming network traffic on the worker 4 is expected since it is responsible for the merging stage and thus need to collect data from both workers 2 and 3. It also expected that PS (worker 1) will 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 compared to other workers of the cluster, since, as stated before, is responsible for the part of the model with the fully connected layers, and therefore has more parameters to send. The PS also exhibits high outgoing traffic since it communicates the parameters received from the fully connected layers to both workers 2 and 3.
Comparing the two architectures, we observe that the ‘optimized’ architecture is characterized by significantly less incoming and outgoing traffic compared to the ‘baseline’ architecture. More specifically, the outgoing traffic of the merging worker can be as low as 2.5MB/sec compared to the baseline one, where it reaches about 10MB/sec. As a result, traffic on the PS is significantly reduced, reaching about 2.5MB/sec for the incoming and 1.5MB/sec for the outgoing traffic. We also observe that for the case of the outgoing traffic, worker 2 exhibits the highest amount of traffic compared to the rest. In fact, the amount of outgoing traffic for the worker is approximately the same as the case of the ‘balanced’ architecture, but becomes significantly more pronounced due to the drop in traffic of the other workers. Overall, for both incoming and outgoing traffic, we can clearly see the significant benefits offered by the ‘optimized’ architecture compared to the ‘baseline’.

In addition to network traffic, we also consider the utilization of available computational resources, measured in terms of CPU utilization. Figure 12 illustrates the performance of our cluster in terms of CPU usage, when executing the ‘baseline’ and ‘optimized’ approach respectively. From this figure, we can make the observation that both PS (worker 1) and the main classification machine (worker 4) do not utilize significant amounts of 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, requires significantly more computational resources, reaching around 70% of CPU of the ‘baseline’ and 50% for the ‘optimized’ on average.
5.3. Training using Model And Data Parallelization Comparison

The experimental results reported in the previous sections demonstrated the impact that a carefully designed architecture can have on the utilization of resources. In this section, we study and compare the performance characteristics of the ‘optimized’ model parallelism architecture with the more traditional data parallelism approaches, during both training and inference stages. We compared the data and model parallelism scheme with respect to two critical system variables, namely network traffic and associated CPU load.

Regarding the network traffic, Figures 13 and 14 demonstrate that for the data parallelization scheme, a significant amount of traffic is associated with PS since its is primarily responsible for synchronizing the different workers, while all workers exhibit similar network traffic characteristics. On the other hand, for the proposed model parallelization scheme, the traffic associated with PS is significantly lower compared to the data parallelization scheme, at the same level as the other workers. Furthermore, worker #3, the one associated with the fusion, and thus the majority of the fully connected layers, has higher network traffic compared to the other workers, nevertheless, it is still less compared to the traffic for the data parallelization scheme.

Regarding the CPU load for the data and model parallelism schemes shown in Figures 15 and 16, a first observation is that in both data and model parallelism scheme, the PS worker is associated with the lowest CPU load, although based on the previous discussion, it is associated with the bulk of the network traffic. As expected, for the case of data parallelism, all three workers exhibit the same CPU load, which is in the order to 90% percent. On the other hand, the majority of processing for the
Figure 14. Incoming traffic per worker for the model parallelism scheme.

Figure 15. CPU load per worker for the data parallelism scheme.

Figure 16. CPU load per worker for the model parallelism scheme.

The model parallelism scheme is associated with worker #3, the one responsible for the feature fusion and classification using fully connected layers.

5.4. Inference using Model And Data Parallelization Comparison

In order to derive convincing conclusion, we consider the inference process of the trained networks using 1000 (1K) and 14000 (14K) testing examples. For both parallelization approaches, we trained each model using an identical dataset of 8K patches and performed inference with the aforementioned
datasets. Table 2 presents the results of our experiments with regards execution time, while Table 3 illustrates inference accuracy for each parallelization scheme. We note that for the data parallelization scheme, the entire testing dataset was split among the workers and executed in parallel.

**Table 2.** Evaluation time comparison between the model parallelization and the ‘baseline’ and ‘optimized’ data parallelization schemes.

<table>
<thead>
<tr>
<th># examples</th>
<th>Data Parallelization</th>
<th>Model Parallelization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>baseline</td>
<td>optimized</td>
</tr>
<tr>
<td>1K</td>
<td>51.7s</td>
<td>26.0s</td>
</tr>
<tr>
<td>14K</td>
<td>787.2s</td>
<td>500.1s</td>
</tr>
</tbody>
</table>

**Table 3.** Accuracy comparison between data parallelization and each one of our proposed model parallelization architectures.

<table>
<thead>
<tr>
<th># examples</th>
<th>Data Parallelization</th>
<th>Model Parallelization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>baseline</td>
<td>optimized</td>
</tr>
<tr>
<td>1K</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>14K</td>
<td>0.83</td>
<td>0.84</td>
</tr>
</tbody>
</table>

The results in Tables 2 and 3 demonstrate the clear superiority of the model parallelism scheme compared to the data parallelism scheme in terms of both time required for inference. In terms of the processing speed, i.e., the execution time, we observe that both model parallelization schemes are able to perform the inference at significantly less time compared to the data-splitting scheme employed in data parallelization. Between the two model parallelization schemes, there is a clear benefit in using the optimized scheme, especially when dealing with larger amounts of data, in which can the optimized scheme is 27% faster.

Furthermore, Table 3 suggests that, even though inference accuracy between both parallelization approaches is similar for both datasets, model parallelization provides slightly better results. Moreover, we observe that the optimized scheme can achieve superior performance for the more 14K examples case. These results clearly indicate that the optimized model parallelization scheme is the preferable approach in the case of inference, since it can make faster and more reliable predictions.

6. Conclusions

In this paper we proposed a multimodal CNN for land cover classification and performed model parallelization architectures, in order to improve its performance on a distributed environment. Moreover, we compared our most efficient architectures with the popular data parallelization approach. We conducted experiments with patches extracted from RGB and HSI images, and our results provide the following contributions:

- 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 Parallelization 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 parallelization’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 reduce 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 non-allocated resources.

• For land cover classification, model parallelization schemes tend to perform better than data parallelization since they provide significantly faster and slightly more accurate predictions.

As last point, we should note that although distributed training is highly recommend for training a DNN, testing/inference may be more efficiently executed using more appropriate hardware platform. For instance, platforms like Field Programmable Gate Arrays (FPGAs) appear offer much more favorable characteristics for inference compared to more typical GPU platforms [52].

Author Contributions: G. T. developed the methodology; M. A. did the experimental analysis and provided the original draft; G.T. and P.T. did the review and editing of the final document.

Funding: his research work was partially funded by the Marie Skłodowska-Curie CALCHAS project (contract no. 842560) within the H2020 Framework Program of the European Commission and by the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under HFRI Faculty grant no. 1725 (V4-ICARUS).

Acknowledgments: The authors would like to thank the National Center for Airborne Laser Mapping and the Hyperspectral Image Analysis Laboratory at the University of Houston for acquiring and providing the data used in this study, and the IEEE GRSS Image Analysis and Data Fusion Technical Committee.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations
The following abbreviations are used in this manuscript:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DNN</td>
<td>Deep Neural Network</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>GSD</td>
<td>Ground Sampling Distance</td>
</tr>
<tr>
<td>HSI</td>
<td>Hyperspectral Image</td>
</tr>
<tr>
<td>PS</td>
<td>Parameter Server</td>
</tr>
<tr>
<td>RGB</td>
<td>Red-Green-Blue</td>
</tr>
<tr>
<td>VHR</td>
<td>Very High Resolution</td>
</tr>
</tbody>
</table>

References


© 2020 by the authors. Submitted to *Remote Sens.* for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).