Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Semantic segmentation for Remote sensing imagery with distributed deep learning pipeline

Author: Tung Nguyen (12012963, 2611152)

1st supervisor: Dr. Adam Belloum
daily supervisor: Dr. Berend Weel

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 10, 2019

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Deep Learning has become one of the fastest-growing trends in Remote Sensing field since it enables Remote-sensing experts as well as non-experts to exploit feature representations learned exclusively from satellite imagery instead of handcrafted features extracted primarily on domain-specific knowledge. Unfortunately, training deep learning model requires enormous amount of computational effort, usually provided by GPU. Scaling up computation from one CPU or GPU to many can enable much faster training and research progress and make it feasible to cope with very big datasets. Existing approaches for facilitating multi-GPU/CPU training under current platforms bring about considerable communication overhead and require users to heavily adapt their local-built training model to distributed one, prompting many researchers to avoid the nuisance and stick with slower single-GPU/CPU training. Furthermore, to the best of my knowledge, there has been no complete pipeline to acquire, preprocess and analyse datasets with deep learning models at large scale. This research proposes a comprehensive pipeline for distributed deep learning with the combination of Apache Spark, the distributed deep learning framework - Horovod, Tensorflow and shows a proof of concept for the pipeline by implementing a distributed U-net deep learning model to solve semantic segmentation problem. We validate our solution on Inria aerial image dataset. As a result, our pipeline does speed up deep learning training especially when increasing from 1 to 2 machines. In addition, the trained model U-net shows great performance in terms of accuracy and Intersection over Union.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the development of remote sensing technologies, especially the improvement of spatial, time and spectrum resolution, Remote sensing (RS) data are characterized by its extreme volume, wide variety of data types and the explosive velocity at which the data must be processed. Meanwhile, the remote sensing textures of the same ground object present different features in various temporal and spatial scales. Therefore, it is difficult for not only professionals but also non-experts to the field of remote sensing to manually describe overall features of remote sensing big data with different temporal and spatial resolution.

Deep learning with its unique advantages makes comprehensive features automatic and ridiculously easy. Different from traditional machine learning techniques in which most of the applied features need to be identified by an domain expert in order to reduce the complexity of the data and make patterns more visible to learning algorithms to work, deep learning algorithms attempt to learn high-level features from data in an incremental manner. This eliminates the need of domain expertise and hardcore feature engineering.

Unfortunately, it is still beyond the ability of a single machine to handle large-scale data sets as deep learning techniques demands extremely high computing power to train in reasonable time. Distributing data and learning process to multiple workstations by leveraging available big data frameworks is a perceptible method to scale up machine learning tasks. Implementation of distributed deep learning models generally requires heavy modifications of local ones in conjunction with taking machine communication overhead into consideration. This results in a considerable need to build a pipeline that simplifies model creation process; ideally build one model and train efficiently in both local machine and cluster.

To drive our research, we define three main questions and answer them in next sections:

1. **How to scale up deep learning model with less modifications of current model-building code and utilize power of high-performance cluster**

Motivation: Regarding Tensorflow as the champion among most popular deep learning frameworks [2] used by data scientists, researchers and big data

developers, an evaluation was conducted by Uber engineers to see how deep learning models need to be adjusted while switching from local mode to cluster mode [3] using Tensorflow API. After following the documentation and code examples, it was not always clear which code modifications needed to be made to distribute model training code. The standard distributed TensorFlow package [4] introduces many new concepts: `workers, parameter servers, tf.Server(), tf.ClusterSpec()`, `tf.train.replicas()` and to name a few. While this API might suit several certain cases, in many scenarios it would yield subtle, hard-to-diagnose bugs especially when users need to manually handle connections and different jobs across nodes/machines. Locating and fixing these bugs have prompted users to climb a steep learning curve of concepts they almost never care about. They might only want to keep existing model and scale it up.

2. **How to build a complete pipeline for distributed storage, preprocessing and training large amount of data?**

Motivation: The combination of deep learning and big data frameworks seems theoretically promising solution to cope with large datasets. However, to the best of my knowledge, existing approaches, excluding Tensorflow, merely focus on distributed training phase in which replicated data stored in head node will be distributed across nodes at run-time. When it comes to massive data, storing intermediate products after preprocessing at a single machine/node might be impossible due to the limited volume capacity of a single node. Even it is feasible, it takes a long time to preprocess repeatedly while starting training phase. Hence, construction of an all-inclusive pipeline to streamline processing, storing intermediate product and analyzing data across different machines/nodes are virtually inevitable.

3. **Can training process of deep learning neural network be accelerated by leveraging the pipeline?**

Motivation: While distributed Deep Learning can enables to reduce training time, it is prone to some inherent trade-offs from distributed systems, comprising communication overhead, scalability. Naturally, performance of distributed deep learning system needs to be evaluated by comparing with baseline in terms of running time.

In this research, we will propose an end-to-end solution to acquire, store, preprocess and train deep learning models on big datasets using a set of big data frameworks: Hdfs, Spark, Petastorm, Tensorflow and Horovod. In particular, we augment 30GB of images of Inria aerial dataset, transform it to parquet format with Spark, use petastorm as an adaptor to feed processed data to Tensorflow dataset and finally train U-net model to do semantic segmentation at large scale with Horovod.

In the next chapter, related work will be discussed followed by the theoretical background for this research. The following chapter explains the rationale behind our proposed pipeline for distributed deep learning. The next two chapters will cover implementation of the pipeline and our research result. The future work and conclusions of this research will be laid out in the final chapters.

Chapter 2

Related work

In this chapter, we chronologically introduce recent distributed deep learning frameworks that support to build a full pipeline for big-data storage, processing, visualization as well as hard-core feature engineering reduction (in other words, less expertise in specific domain is demanded). We define four main criteria entirely driven by our first and second research questions to filter available frameworks: provision of deep learning algorithms and GPU usage, support to fetch distributed datasets as training input, less modifications of source code when switching from local to cluster model, well-organized documents and support by large developer community (Github repository's stars) and big companies.

Apache Hadoop MapReduce and Apache Spark [5] are well-known frameworks that facilitates the distribution of massive data collections across multiple nodes within a cluster of commodity servers and keeps track of that data, enabling big-data processing and analysis far more effectively than was possible previously. As only Apache Spark supports traditional machine learning algorithms with MLLib, both lacks the implementation for Deep Learning.

This shortage has motivated technical students, developers and researchers to develop distributed deep learning frameworks. Before digging deep into frameworks, we should know that there is a catch in distributed deep learning. We can not simply split the data into N partitions and train N separate models, then combine them at the end nor break the model into small components and train on each component separately with all the data. The neural network has a single set of parameters used to make its predictions. At training time, we use those parameters to make a set of predictions for a single batch of training examples, measure our error in the form of a gradient, then backpropagate our gradients up through the network to adjust the parameters. For the next batch, our predictions will in theory have improved and we continue this process until we reach convergence (no more change in parameters) or we complete some number of epochs.

The Google engineers obviously knew about this problem, and came up with a system known as **DistBelief** [6]. To enhance data parallelism, DistBelief assigns a training process to a subset of the data on each node, but in order to maintain state consistency, they all send and receive training parameter updates through a centralized parameter server. The underlying al-

gorithm at work here is known as Downpour SGD, a variant of asynchronous stochastic gradient descent [7], where individual nodes are able to send updates at different frequencies without waiting on all the other nodes (Figure 2.1). To separate tasks accordingly to heterogeneous machines, an coordinator is employed into what called Sandblaster L-BFGS batch optimization so that it assigns each of the N model replicas a small portion of work, much smaller than 1/Nth of the total size of a batch, and assigns replicas new portions whenever they are free (Figure 2.2). With this approach, faster model replicas do more work than slower replicas. To further manage slow model replicas at the end of a batch, the coordinator schedules multiple copies of the outstanding portions and uses the result from whichever model replica finishes first.

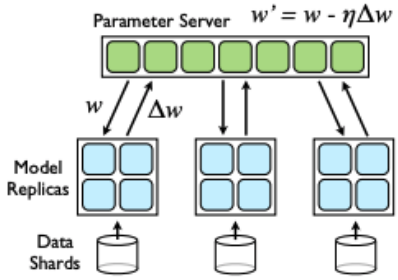


Figure 2.1: Downpour SGD. Model replicas asynchronously fetch parameters w and push gradients Δw to the parameter server [6]

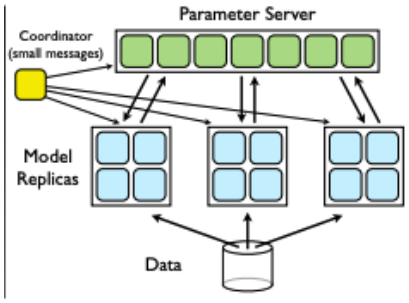


Figure 2.2: Sandblaster L-BFGS. A single 'coordinator' sends small messages to replicas and the parameter server to orchestrate batch optimization [6]

By incorporating this single parameter server, DistBelief imposes a single bottleneck and a single point of failure. The more nodes you have, the more requests are going to be sent to that server. In order to tackle the problem, Baidu researchers proposed a technique from the high-performance computing community to improve the parameter averaging and communication efficiency, called ring-allreduce. The algorithm was based on ring-allreduce algorithm introduced in the 2009 paper by Patarasuk and Yuan [8]. In the ring-allreduce algorithm, shown on Figure 2.3, each of N nodes communicates with two of its peers for $2 * (N - 1)$ times. During this communication, a node sends and receives chunks of the data buffer. In the first $N - 1$ iterations, received values are added to the values in the node's buffer. Overall, the algorithm proceeds in two steps: first, a scatter-reduce, and then, an all-gather. In the scatter-reduce step, the nodes will exchange data such that every node ends up with a chunk of the final result. In the all-gather step, the GPUs will exchange those chunks such that all GPUs end up with the complete final result. The realization that a ring-allreduce approach can improve both usability and performance motivated Uber engineers to built upon it and come up with **Horovod** framework.

SystemML [9] is a declarative large-scale machine learning (ML) framework based on Apache Spark, which is characterized by automatic algorithm customization. ML algorithms in SystemML are specified in a high-level,

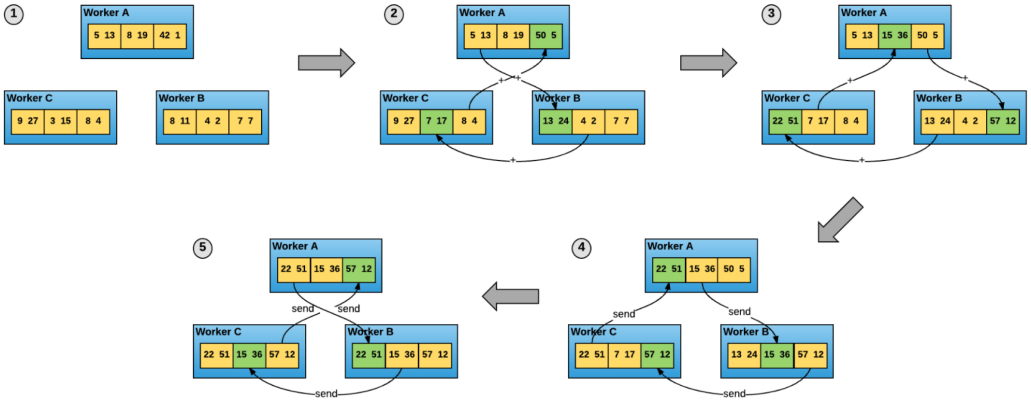


Figure 2.3: The ring-allreduce algorithm allows worker nodes to average gradients and disperse them to all nodes without the need for a parameter server [3]

declarative machine learning (DML) language. DML scripts are compiled into mixed driver and distributed jobs and thus enables data scientists to run deep learning applications without the presence of the middle jobs of system programmer. DML's syntax closely follows R, thereby minimizing the learning curve to use SystemML. Algorithms specified in DML are automatically optimized based on data and cluster characteristics using rule-based and cost-based optimization techniques. The optimizer automatically generates hybrid run time execution plans ranging from in-memory, single-node execution, to distributed computations on Spark or Hadoop. This ensures both efficiency and scalability. Automatic optimization reduces or eliminates the need to hand-tune distributed run time execution plans and system configurations.

Deeplearning4j [1] is an open source distributed learning framework running on top of Apache Spark. It was launched by Adam Gibson and Skymind team in 2014 with the ambition to provide wide support for deep learning algorithms including distributed and centralized version. Deeplearning4j has two implementations of distributed training: Parameter averaging, a synchronous stochastic gradient descent implementation with a single parameter server implemented entirely in Spark as well as Gradient sharing, applying an asynchronous stochastic gradient descent based on Strom neural network training paper by Nikko Stromwith [10]. Users are directed towards the later implementation which superseded the former implementation. The key feature of Gradient sharing approach is that opposed to relaying all parameters across the network, only updates that are above a user specified threshold are communicated. In other words, we start out with an update vector (1 entry per parameter) that needs to be communicated. Instead of passing the vector as-is, only the large elements are transferred in a quantized way (which is a sparse binary vector). Note that updates below the threshold are not discarded but accumulated in a "residual" vector to be applied later. A centralized parameter server is replaced by peer to peer communication as indicated in Figure 2.4. Overall, DL4J seems to be competitive in terms of speed and ease of use. However, it is a Java-based, industry-focused, commercially supported, distributed deep-learning framework, which in turn is

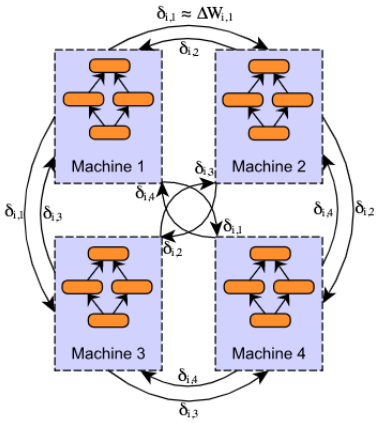


Figure 2.4: Nodes communication in DeepLearning4j framework [1]

of less interest to researchers.

In 2015, Microsoft introduced **CNTK** [11], a unified deep-learning framework that describes neural networks as a series of computational steps via a directed graph. In this directed graph, leaf nodes represent input values or network parameters, while other nodes represent matrix operations upon their inputs (Figure 2.5). Also, CNTK supports symbolic loops over sequential data and automatically unroll the loops. That is to say, CNTK can describe Recurrent Neural Network model through its programming model more naturally than other frameworks (Figure 2.6). Another unique feature is that different mini batches of data containing sequences of different lengths are automatically packed and padded, which are mostly used in speech process as batches of speech data have similar structure. Regarding data-parallel training, CNTK adopts two strategies to improve the performance namely communicating less each time and communicating less often. The idea of the first strategy is to apply 1-bit SGD algorithms [12] to quantize gradients to 1 bit per value and carry over quantization error to next minibatch so that exchanged package size is significantly reduced. At the same time, the mini batches incrementally increase to speed up the training process, which is called Automatic MB sizing algorithm [13].

Apache MXN [14] is designed for both efficiency and flexibility. It allows you to mix symbolic and imperative programming to maximize efficiency and productivity. It also contains a dynamic dependency scheduler that automatically parallelizes both symbolic and imperative operations on the fly. A graph optimization layer on top of that makes symbolic execution fast and memory efficient. MXNet supports an efficient deployment of a trained model to various devices ranging from low-end devices such as mobile devices, Internet of things devices, serverless computing and containers to higher-end GPU based cluster. Despite running on heterogeneous devices, MXNet can still achieve almost linear scale with multiple GPUs or CPUs.

FireCaffe [15] has a similar idea to prior frameworks for speed and scalability improvement by reducing overhead of communicating between servers without accuracy degradation. The framework has three key pillars. First, reduction trees are selected among a number of communication algorithms because it is more efficient and scalable than the traditional parameter server

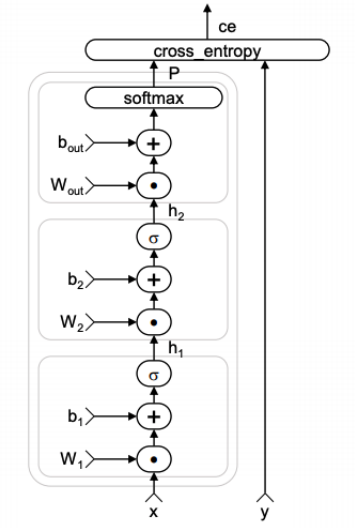
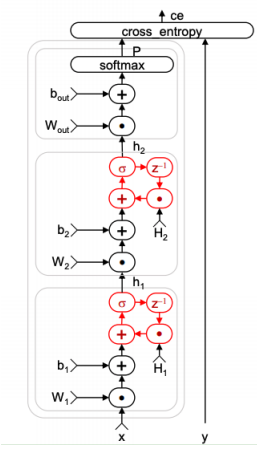


Figure 2.5: CNTK directed graph [11]




```
h1 = sigmoid(x  @ W1 + past_value(h1) @ H1 + b1)
h2 = sigmoid(h1 @ W2 + past_value(h2) @ H2 + b2)
P  = softmax(h2 @ Wout + bout)
ce = cross_entropy(P, y)
```

Figure 2.6: CNTK graph of Reconcurrent Neuron Network [11]

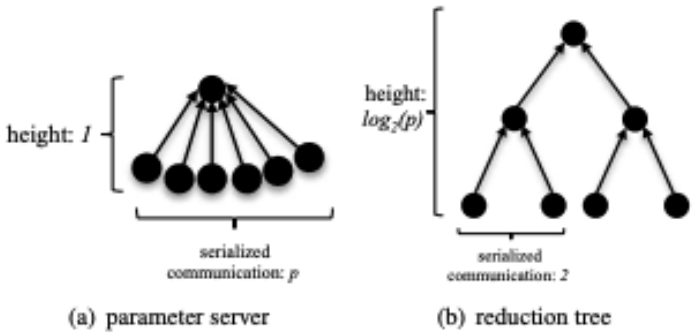


Figure 2.7: How parameter servers and reduction trees communicate weight gradients [15]

approach. While the parameter server communication overhead scales linearly with p (branching factor of a tree), reduction tree communication is much more efficient because it scales logarithmically as $O(log(p))$ (Figure 2.7). Second, authors optionally increase the batch size to cut down the total quantity of communication during training and determine hyperparameters that allow to reproduce the small-batch accuracy while training with large batch sizes. Finally, Infiniband or Cray interconnects are opted to achieve high bandwidth between GPU servers. When training GoogLeNet and Network-in-Network on ImageNet, FireCaffe achieves a 47x and 39x speedup over a single GPU, respectively, when training on a cluster of 128 GPUs.

Chapter 3

Theoretical Background

3.1 Remote Sensing

3.1.1 Definition

> *Remote sensing is the practice of deriving information about the Earth's land and water surfaces using images acquired from an overhead perspective, using electromagnetic radiation in one or more regions of the electromagnetic spectrum, reflected or emitted from the Earth's surface [16]*

The process of remote sensing comprises an interaction between incident radiation and the targets of interest. This is exemplified by the use of imaging systems where the following seven elements are involved (Figure 3.1).

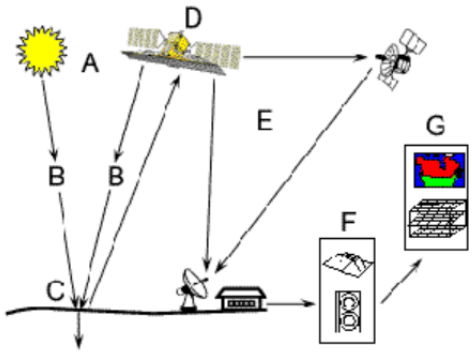


Figure 3.1: Remote sensing process [17]

- **Energy Source or Illumination (A):** Remote sensing requires an energy source which illuminates or provides electromagnetic energy to the target of interest.
- **Radiation and the Atmosphere (B):** the energy will interact with the atmosphere it passes through while traveling from source to target. This interaction might repeat as the energy reflects from the target to the sensor.

- **Interaction with the Target (C):** once the energy reaches to the target via the atmosphere, it might absorb, go through or reflect depending on the properties of both the target and the radiation.
- **Recording of Energy by the Sensor (D):** after the energy has been emitted from the target, sensors which are located remotely from the target will collect and record the electromagnetic radiation.
- **Transmission, Reception, and Processing (E):** the energy recorded by the sensors has to be transmitted, often in electronic form, to a receiving and processing station where the data are digitalised into images.
- **Interpretation and Analysis (F):** the processed image is visualized and interpreted to extract information about the target.
- **Application (G):** the final element of the remote sensing process is achieved when we apply the information we have been able to extract from the imagery about the target in order to better understand it, reveal some new information, or assist in solving a particular problem

3.1.2 Image resolution

There are various types of sensors, ranging from on board of airplanes to on board of satellites, measures the electromagnetic radiation at specific ranges (usually called bands). Those measures are quantized and converted into a digital image. The resulting images have different characteristics (resolutions) depending on the sensor:

- **Spatial resolution:** refers to the number of pixels utilized in construction of the image. Images having higher spatial resolution are composed with a greater number of pixels than those of lower spatial resolution.
- **Spectral resolution:** the ability to resolve spectral features and bands into their separate components.
- **Radiometric resolution:** usually measured in bits (binary digits), is the range of available brightness values. For example an image with 8 bit resolution has 256 levels of brightness.

For satellites sensors, there is also the temporal resolution, which is the time required for revisiting the same area of the Earth (NASA, 2013).

3.2 Deep Learning

Deep learning is a subdomain of machine learning that inspired by simulation of the structure of the human brain, called Artificial Neural Networks (ANN) and designed to recognise pattern. In 1957, Rosenblatt invented the first perceptron neural network , for image recognition. However, until the 2010s

since vast amount of data and high computing power enables neural networks to improve the state-of-the-art in many applications, it has become more and more popular.

Unlike more traditional methods of machine learning techniques, deep learning classifiers are trained through feature learning rather than task-specific algorithms. In other words, the machine will learn patterns in input data without the need for human operation. This in turn helps to overcome feature engineer difficulties in machine learning process which is expensive in terms of time and expertise [18].

Another major distinguishing aspect of deep learning compared to more traditional methods is the ability to enhance the performance of the classifiers with increases in amount of data. Generally, if we construct larger neural networks and train them with more and more data, the performance of algorithm keeps increasing. In contrast, the performance of other machine learning techniques may reach a plateau even when more data are constantly fed into the algorithms (Figure 3.2).

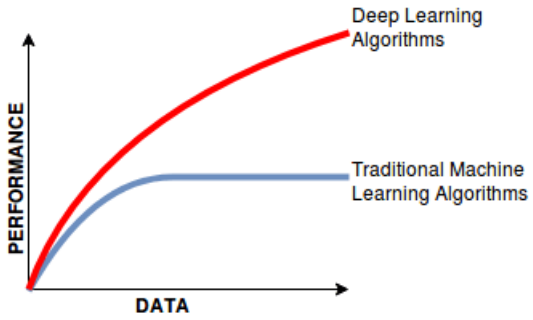


Figure 3.2: Increasing data consistently yields better performance

Deep learning algorithms attempts to find associations between a set of inputs and outputs. The basic structure of a deep learning algorithm is represented as below:

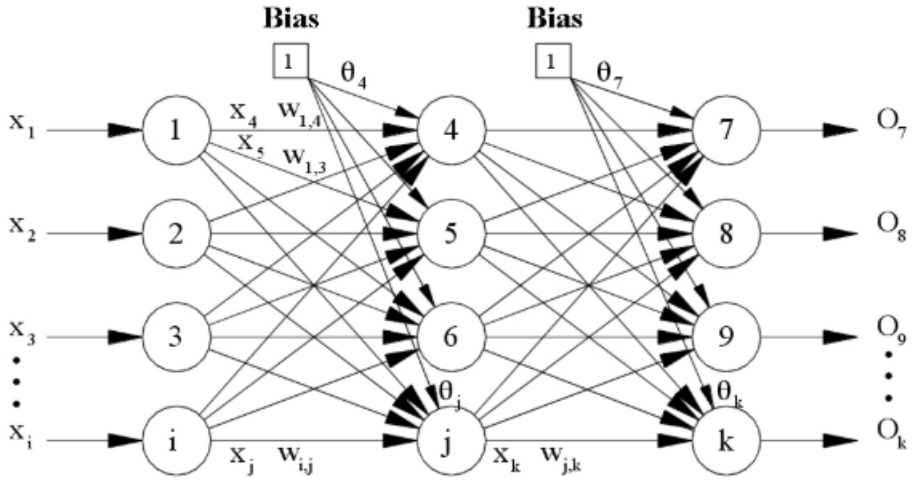


Figure 3.3: Deep Learning neural network structure [19]

A deep neural network is composed of input, hidden, and output layers, each may contain multiple nodes (also called neurons). Input layers take in a numerical representation of data (e.g. images with pixel specs), output layers generate predictions, while hidden layers correspond to most of the computation. A mathematical representation of such a neuron is displayed in figure 3.4.

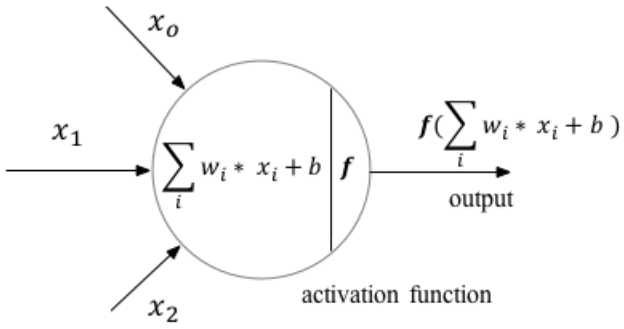


Figure 3.4: Deep Learning neural network mathematical model

Each neuron has a set of inputs x, and each of these inputs is assigned with a specific weight w and then the neuron first calculates a weighted sum of these inputs:

$$y = x_1w_1 + x_2w_2 + x_3w_3 + ... + x_nw_n + b \tag{3.1}$$

After that, this y output will be passed through an activation function *f*. This activation function is a non-linear function, introducing non-linearity into the network and enabling the network to model non-linear dependencies between the target variable and the input variable(s).

After the neural network passes its inputs all the way to its outputs, the network evaluates how good its prediction was through a loss function which might varies in different applications. For example, one of the most popular loss functions - the "Mean Squared Error" loss function, is shown below:

$$loss = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{3.2}$$

The goal of deep learning network is ultimately to minimize this loss by adjusting the weights and biases of the network. This is achieved by a process called "back propagation". Through gradient descent, the network backtracks through all its layers to update the weights and biases of every node in the opposite direction of the loss function. Simply put, every iteration of back propagation should result in a smaller loss function than previous. The continuous updates of the weights and biases of the network basically turns it into a precise function approximator that models the relationship between inputs and expected outputs.

3.3 Convolutional Neuronal Networks

3.3.1 Problem space

Convolutional Neuronal Network (CNNs) is subcategory of deep learning neural networks, which was invented to resolve several drawbacks of Multiple Layer Perceptrons (MLPs), especially when it comes to image processing. MLPs assigns one perceptron (neuron) to each input (e.g. pixel in an image). The amount of weights rapidly grows unmanageable for large images. For a small-size image 128 x 128 pixel image with 3 color channels there are 49152 weights that must be generated. As a result, high computing power is demanded for training and overfitting [20] can take place.

Another common problem is that MLPs recognizes differently to an input (images) and its shifted version. For example, if a picture of a cat appears in the top left of the image in one picture and the bottom right of another picture, the MLPs will try to correct itself and assume that a cat will always appear in the bottom right of the image. We therefore need a solution to find the spatial correlation of the image features (pixels) in such a way that we can see exactly the cat in our picture no matter of its positions.

3.3.2 Convolutional Neuronal Networks architecture

Convolutional Neuronal Networks (CNNs) takes an image as input, it will see an array of pixel values. Let's say we have a color image in PNG form (3 color channels) and its size is 128 x 128. The representative multiple-dimensional array will be 128 x 128 x 3. Each of value in the array is given from 0 to 255 which illustrates the pixel intensity at that point. The idea is that you give CNNs this array of numbers and it will pass it through a series of convolutional, pooling (usually max pooling), transposed and fully connected layers, and an output will be yielded as numbers that describe image classes or the probability of the image being a certain class.

In the scope of this research, we only provide some brief introduction about key layers that we adopted in our later neural network. In-depth architecture of Convolutional Neuron Networks is available under the context of the course "CS231n: Convolutional Neural Networks for Visual Recognition" [21] from Stanford

3.3.2.1 Convolutional Layer

The primary purpose of Convolutional Layer in case of a Convolutional Neuronal Networks is to extract features from the input image. Convolutional Layer learns image features by scanning through the entire image with small squares of input data.

We will take only 1 colour channel of image to explain about convolutional layer but just keep in mind that generally image includes 3 channels of RGB. Every image can be considered as a matrix of pixel values. Consider a 5 x 5 image whose pixel values are only 0 and 1 and another 3 x 3 matrix as filter or kernel in Figure 3.5. We slide the orange matrix over our original image

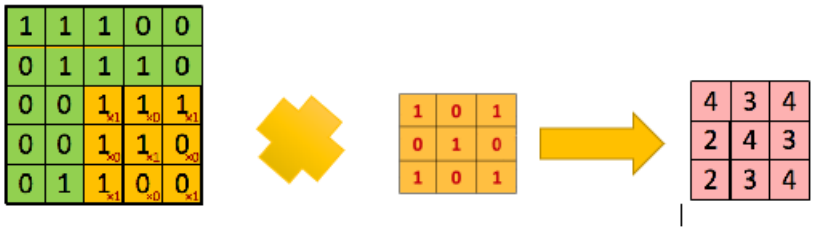


Figure 3.5: Convolution Operation [21]

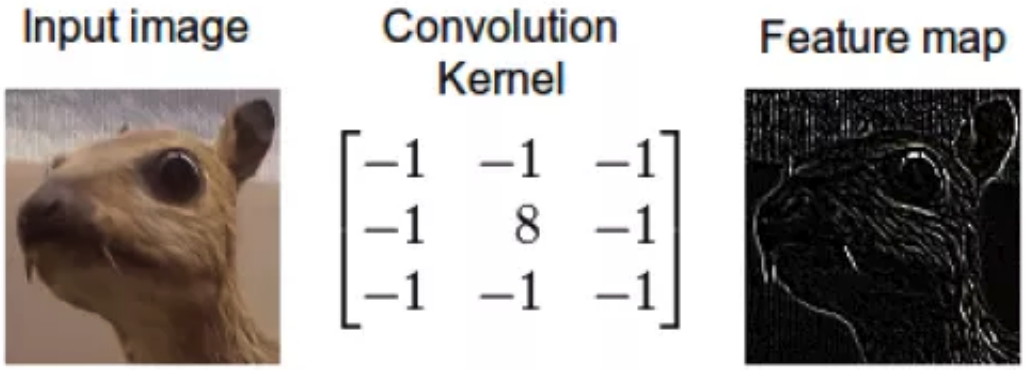


Figure 3.6: Effect of convolution filter/kernel. The image was captured from the website [1]

(green) by 1 pixel, called stride and for every position, we compute element-wise multiplication between the two matrices and add the multiplication outputs to get the final value which forms a single element of the output matrix - a feature map.

Different values of the filter matrix will produce different feature maps for the same input image. In the figure 3.6 , we can see the effect of convolution on a mice image with a convolution filter/kernel before the convolution operation. This leads to the detection of different features from an image (e.g: lines, edges, curves and etc).

3.3.2.2 Pooling Layer

Pooling layer, also called subsampling or downsampling, reduces the dimensionality of each feature map but retains the most important information. Pooling layer can be of various types: Max, Average, Sum. In practice, Max Pooling [22] has been shown to have better accuracy. In case of Max Pooling, we define a spatial neighborhood (for example, a 2×2 window) and take the maximum value within that window to represent the window value (Figure 3.7).

[1] https://timdettmers.com/2015/03/26/convolution-deep-learning/

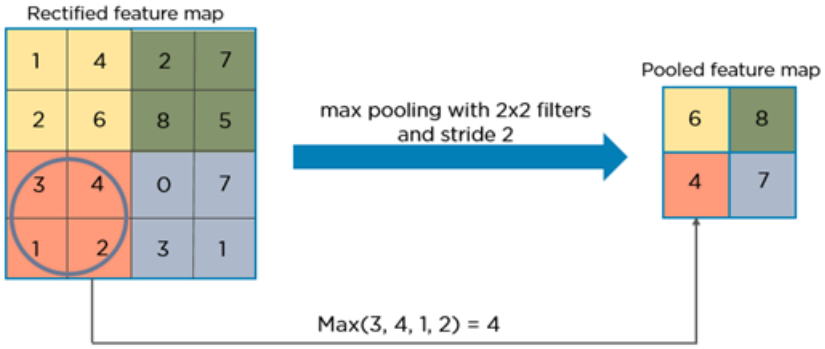


Figure 3.7: Max Pooling Operation. The image was captured from the website [2]

3.3.2.3 Fully Connected Layer

The output from the previous convolutional and pooling layers represents high-level features of the input image. The purpose of the Fully Connected layer is to aggregate these features for classifying the input image into different classes or corresponding probability of classes (in case softmax activation function is used after fully connected layer) based on a training dataset. The term "Fully Connected" indicates that every neuron in the previous layer is connected to every neuron on the next layer (Figure 3.8).

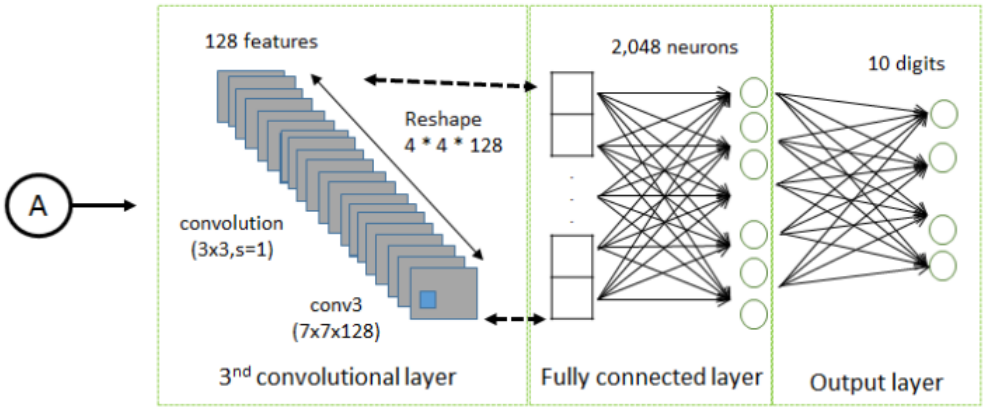


Figure 3.8: Fully Connected Layer. The image was captured from website [3]

3.3.2.4 Transposed Convolution Layer

Transposed convolution is a technique for upscaling an image with learnable parameters. On a high level, transposed convolution is exactly the opposite process of a normal convolution (Figure 3.9). As for Transposed convolution layer, the input volume is a low resolution image and the output volume is a

[2]https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks
[3]https://exploreai.org/p/tensorflow-cnn

high resolution image. In other words, normal convolution can be expressed as a matrix multiplication of input image and filter to produce the output image. By just taking the transpose of the filter matrix, we can reverse the convolution process.

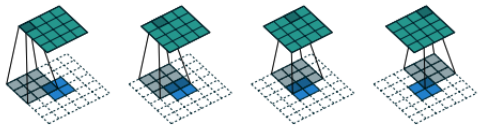


Figure 3.9: The transpose of convolving a 3×3 kernel over a 4×4 input using unit strides. It is equivalent to convolving a 3×3 kernel over a 2×2 input padded with a 2×2 border of zeros using unit strides [23].

3.4 Semantic segmentation with U-net neural network

3.4.1 Semantic segmentation

Semantic image segmentation [24] describes the process of associating each pixel of an image with a class label such as buildings, cats, dogs and etc. This task is commonly referred to as dense prediction as we are predicting for every pixel in the image (Figure 3.10).

Figure 3.10: An original image of a building on the left side and the output of semantic segmentation process on the other side

The expected output in semantic segmentation are not just labels for an image or bounding boxes covering objects in the image. The output itself is a high resolution image (typically of the same size as input image) in which each pixel is classified to a particular class. Thus semantic segmentation is categorized as a pixel level image classification.

3.4.2 U-NET Architecture and Training

The U-NET is a convolutional network architecture for fast and precise segmentation of Bio Medical images, which is developed by Olaf Ronneberger

et al [25]. The architecture contains two paths with 23 convolutional layers in total as illustrated in Figure 3.11. The first path is the contraction path, also called as the encoder which captures "WHAT" is there in the image. The encoder merely consists of the repeated application of two 3x3 convolution layers (unpadded convolutions), each followed by a rectified linear unit (ReLU [26]) activation function and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step, the number of feature channels will be doubled. The second path is a symmetric expansive path (decoder) which enables accurate localization using transposed convolutions. Every step in the expanding path involves an upsampling of the feature map, followed by a 2x2 convolution that halves the number of feature channels and then a integration with an accordingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. This concatenation builds up the precision of U-net by reducing errors and rules out blurry borders of combined data in every convolutional layer. The final layer, a 1x1 convolution, is used to map each feature vector to the desired number of classes.

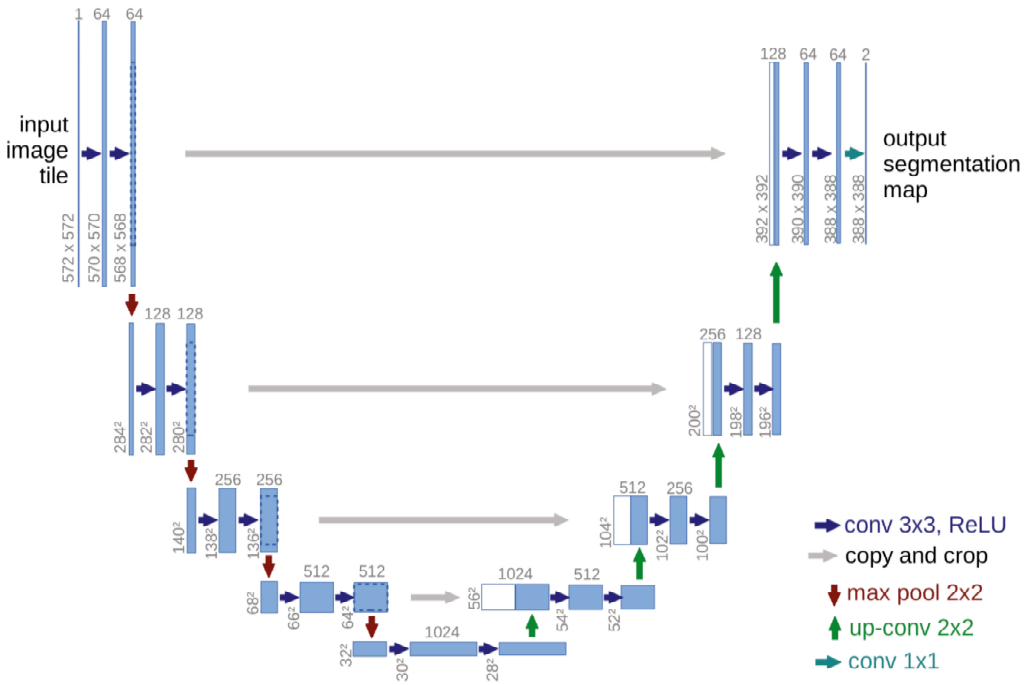


Figure 3.11: U-net architecture. Each blue box corresponds to a multi-channel feature map. The size x-y of feature map is described at the bottom left of the box. White boxes represent corresponding copies of feature maps. The arrows denote the different operations [25]

Chapter 4

Proposed Distributed Deep Learning pipeline

In this chapter, we will propose an end-to-end pipeline for distributed deep learning and elaborate on the reason why we choose different components in the pipeline. Generally, our pipeline contains 4 main phases (Figure 4.1):

- **Data storage:** At this phase, the input data will be stored and available across different nodes/machines since our main objective is to cope with big data solution for Deep Learning.
- **Preprocessing**: An integral phase in deep learning as the quality of data and the useful information that can be derived from it directly affect the ability of our model to learn and therefore, it is extremely important that we preprocess our data before feeding it into our model. For a large dataset, it is logically necessary to devolve preprocessing tasks on different machines.
- **Training**: The training phase will be taken in distributed manner so that we can utilize the power of high performance cluster. Since the final trained model only takes up a couple of megabytes, our pipeline is designed to save it to only one node/machine.
- **Predicting**: Predicting is the final phase of our pipeline which uses trained model (usually in hdf5 format) from previous step to yield predictions.

The detailed implementation of the pipeline will be described in the next chapter.

4.1 Input data storage

While the actual input data is highly dependent on the application using the data, we can determine how this data is stored. Hadoop Distributed File System [27] (HDFS) provides reliability, security, possibly an performance increase due to locality as well as support for a wide variety of file formats. It would therefore be logically to use this system for the input data. We

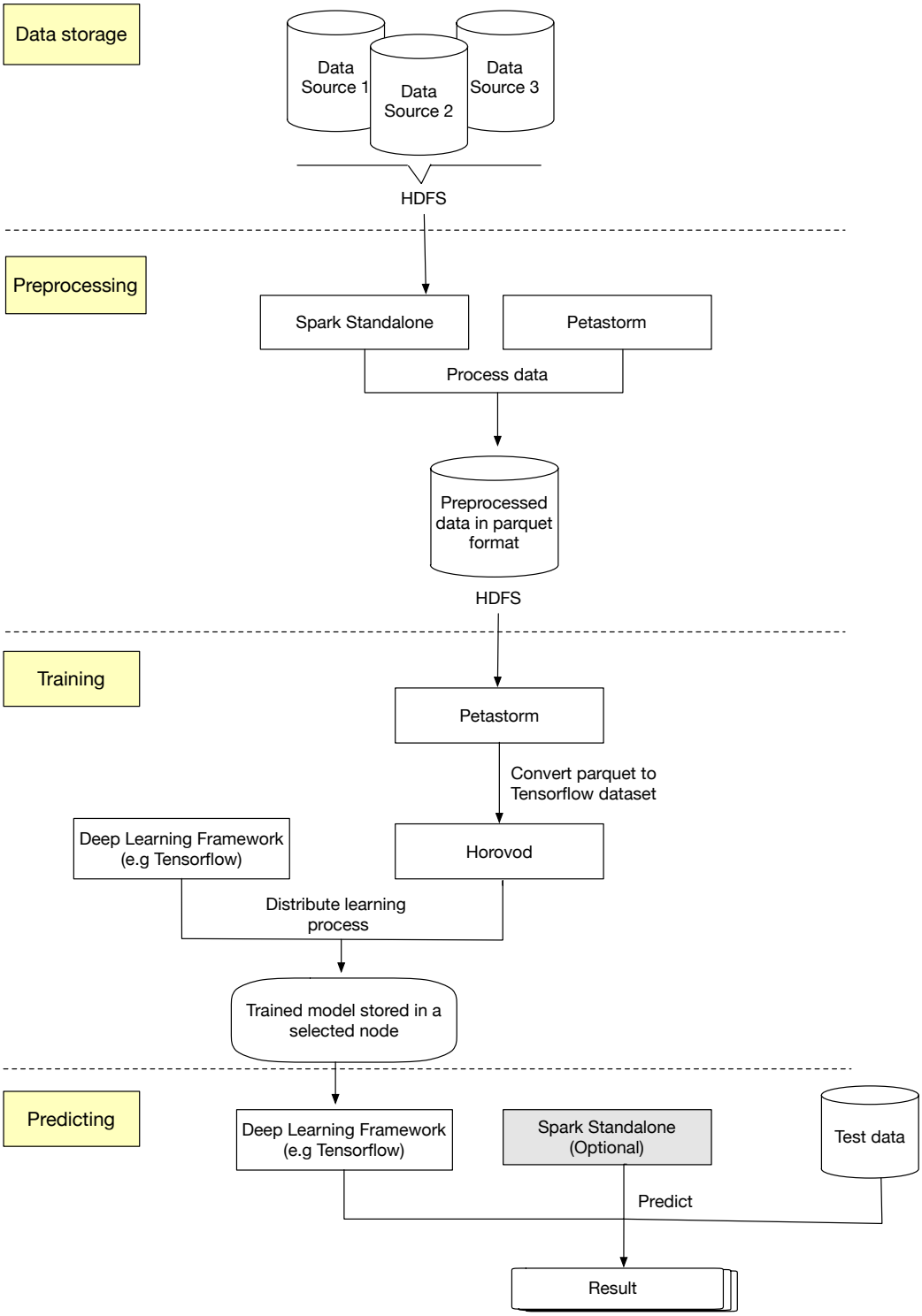


Figure 4.1: Our proposed pipeline for distributed deep learning

assumed that one machine/node is not able to store entirely a very large input dataset due to disk volume capability. Therefore, HDFS was deployed so that the input data was distributed to multiple nodes/machines. We have provided an easy-to-use scripts [1] to deploy HDFS to cluster environment as well as uploading input data to HDFS. The script will allocate a certain number of nodes on DAS-4 run HDFS services on each node, fetch input data by Http protocol and automatically upload those data to HDFS.

4.2 Preprocessing

This pipeline focuses on applications which are batch-based. While the input data is constantly generated, new data that is generated will be delivered in big batches. The applications generate results from the fixed amount of data that is available at the time of execution.

The fundamental ideas of big data batch processing derive from the map reduce model [28]. The map reduce model describes a model to analyze large amounts of data by splitting it up in to key value pairs. The actual processing in the map reduce model is done in two simple steps: map and reduce. In the map stage the key/value pair is transformed into intermediate key/value pairs. In the reduce step, these intermediate pairs are aggregated to create the output. The strength of this model lies in the fact that the map stage can be done simultaneously over many machines, which is the same with reduce stage, thus creating a very scalable system.

Among the various implementations, one of the most widely used is Apache Spark. Spark [29] is an open source project claiming to be the next generation of Hadoop Map Reduce as it could outperform predecessor by leveraging memory computation. The project rapidly reaches maturity, and is selected by the industry. It could therefore be a very good candidate for our pipeline.

We associate Spark Standalone [30] with Petastorm [31], a library developed by Uber to encode and compress input data then save the final product to Apache Parquet format [32]. The reason that we pick Apache Parquet as an intermediate product is that it supports fast access to individual data columns. This facilitates training step as training data are generally split into feature and label sets which can be matched accordingly to feature and label columns in parquet files.

4.3 Training

After contemplating the advantages and disadvantages of different distributed deep learning frameworks in chapter Related work, we ultimately choose Horovod as the most suitable distributed deep learning framework for four key reasons. The first reason is that it has the capability to scale up (both GPU and CPU usage) almost linearly. Figure 4.2[2] shows the number of

[1] https://github.com/ThrowMeForALoop/RemoteSensing

[2] Image was captured from https://github.com/horovod/horovod

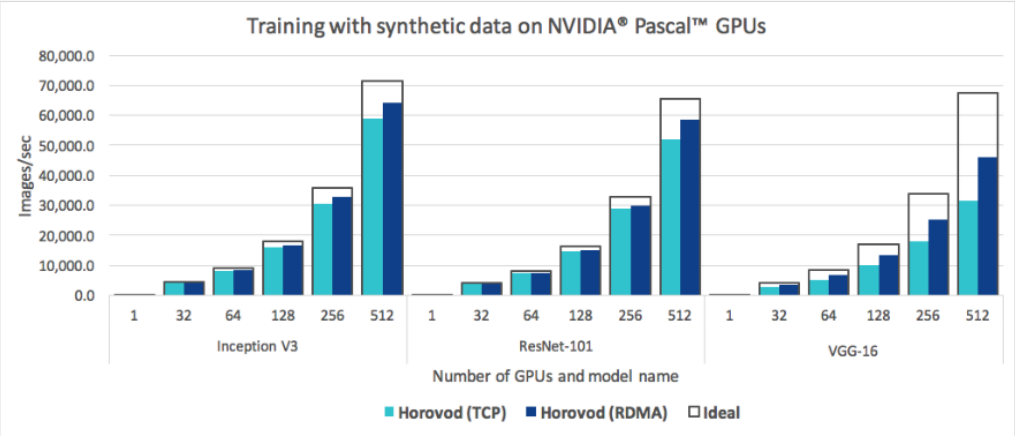


Figure 4.2: Horovod benchmark

images processed per second by different deep learning networks when increasing number of total GPUS across 128 servers. Horovod achieves 90% scaling efficiency for both Inception V3 [33] and ResNet-101 [34] model. Secondly, a few lines of codes needs to be added to local program in order to make it distributed [3]. Horovod also provides horovod.spark package, a convenient wrapper around Open MPI that makes running Horovod jobs in Spark clusters easy. Finally, it receives strong support from active developers as well as Uber engineers.

In this step, we first use Petastorm to load and decode data from parquet files once it was encoded and compressed in the previous step. This is followed by another data transformation step depending on the selected back-end deep learning framework running on top of Horovod. Specifically, if Tensorflow runs on top of Horovod, we need to convert Parquet files to Tensorflow Dataset which is later fed to Horovod in order to train deep learning model. The final trained model will be stored only in a preferable worker/node to prevents other workers from corrupting them.

To switch from CPU and GPU usage, we only need to install NVIDIA Collective Communications Library (NCCL) and specify the library path to LD_LIBRARY_PATH environment variable. Horovod will automatically recognize available GPUs and use them up.

4.4 Predicting

This phase is the same as ordinary predicting step in Machine Learning process when we load the model generated from training step and forecast the result using test dataset. Generally, test dataset is much smaller in size than training dataset so we can opt to predict the outcome in a node/machine. In case that test dataset is big, Spark Standalone can be used to distribute predicting tasks.

[3]https://github.com/horovod/horovod#usage

Chapter 5

Experiments

Our proposed pipeline is reasonably constructed based on our evaluation of different frameworks and tools in chapter 2 and chapter 4. We want to assess whether our pipeline works as expected and locate the bottlenecks of the pipeline that can be improved later. Furthermore, we are deeply interested in the ease of implementing an application which complies with the pipeline. We therefore have to implement proof of concept applications.

5.1 Remote Sensing applications

The goal of our applications is to do semantic segmentation of aerial imagery. In particular, given our application an image covering dissimilar urban settlements, it will classify all pixels as building class or not-building class.

Note that as there are limited computing resources on DAS-4 cluster [35] , we can only obtain 16 nodes/machines with CPUs and 1 node with 1 GPU. Unfortunately, application with deep learning model training requires enormous amounts of computation which can be only fulfilled by GPUs usage. To overcome this obstacle, we have to implement two experiments with the same source code but different environments: the first one runs on small dataset with distributed CPUs and the second runs on the whole bigger dataset with 1 GPU. In that way, we can firstly validate the validity of our pipeline with CPUs experiment. Secondly, the final result of our pipeline if using GPUs can be implied based on second experiment when the adopted distributed deep learning framework, Horovod, demonstrated that training deep learning model in local or distributed manner will exactly be the same as long as global mini-batch size is consistent and the model converges [1]

Two applications are written in Python and split up into four main parts: storage, preprocessing, training and predicting. Input data storage as well as big data frameworks deployment scripts are basically a modified version of "Deployment scripts and configuration for launching a variety of Big Data frameworks on the DAS-4/5" by Amusaafir [2]

[1] https://github.com/horovod/horovod/issues/390

[2] https://github.com/amusaafir/das-bigdata-deployment

5.2 Hardware

Applications have been run on the DAS-4 cluster, which includes 6 clusters with dual quad-core compute nodes. The hardware specifications of machines are listed in Table 5.1

Table 5.1: Hardware Specification per node

Component	Name
CPU	2 x Intel(R) Xeon(R) CPU,E5620,@ 2.40GHz
Cores	8 (16 threads)
Memory	24 GB
Network	IB and GbE

Several machines in DAS-4 come equipped with GPUS which can only allocated through SLURM [36]. In the GPU experiment, we make use of one Titan Black GPU with 6144 MB memory. Taking GPU memory into consideration, we can wisely choose number of data records in later training batches.

5.3 Software

As regards big data deployment, the whole Hadoop platform, including HDFS and MapReduce comes in one package. The version installed on the machines is 2.6.0. For spark, the version installed is 2.4.0. The number of worker cores and amount of memory allocated for each worker are adjusted in Spark configuration so that it matches to hardware specification of each node (Table 5.1).

All dependencies and code environment for our applications are listed below:

```
- python=3.6.8
- tensorflow=1.12 or tensorflow-gpu=1.12 in case of GPU use
- horovod=0.16.3
- petastorm=0.7.2
- openmpi=4.0.1
- h5py=2.9.0
- gcc-5=5.2.0
- libgcc=7.2.0
- pillow=6.0.0
- packaging=16.8
- future=0.17.1
- opencv-python=4.1.0.25
```

Since the `tensorflow.keras` version 1.12 is unable to automatically determine the shape of image data at transformation step (from Parquet to Tensorflow Dataset) and throws the error, a hot-fix solution [3] was applied

[3]https://github.com/tensorflow/tensorflow/issues/24520

to bypass the issue. Optionally, updating to newer version of Tensorflow can entirely fix the issue. However, we will stick to Tensorflow 1.12 when DAS-4 only support it as the latest version. It is also worth noting that DAS-4 CPU with Intel® AVX instruction support is not compatible with Horovod versions lower than 0.16.2 [4].

5.4 Input dataset

In this research, we choose Inria Aerial images [37] as our input dataset among high-quality datasets available under repositories [5]. There are two main reasons for the decision. All images of the dataset have high spatial resolution (at a 30 cm resolution) which certainly helps to enhance the accuracy of deep learning model. Also, the size of the dataset suits our computation resource fairly well. Specifically, our program needs roughly half an hour to accomplish a training round (epoch) with 1 Titan Black GPU. In other words, we can sufficiently train our model on approximately 100 epochs for two days on DAS-4 (Note that all GPU jobs can be only run on DAS-4 for maximum of two days).

Inria Aerial dataset includes 180 image tiles (also called feature data) of size 5000×5000 and 3-color channels (Figure 5.1 and Figure 5.2), covering a surface of 1500 m × 1500 m each (at a 30 cm resolution). There are 36 tiles for each of the following regions: Austin, Chicago, Kitsap County, Western Tyrol and Vienna. Each image is in GeoTIFF format (TIFF with georeferencing, but the images can be used as any other TIFF). Images are named by a prefix associated with the region (e.g. austin- or chicago-) followed by the tile number (1-36). The reference data (also called label data) is in a different folder and named correspondingly to those of the color images. In the case of the reference data, the tiles are single-channel images with values 255 for the building class and 0 for the not building class. **The input dataset will be divided into three smaller dataset: training (60%), cross validation (20%), test (20%) that will be fed to according steps. This division is faithfully followed the rule-of-thumb by renowned data scientist Andrew Ng [38].**

[4] https://github.com/horovod/horovod/issues/1071

[5] https://github.com/chrieke/awesome-satellite-imagery-datasets


Figure 5.1: Austin 3-color channel image (left) and Austin reference image (right)

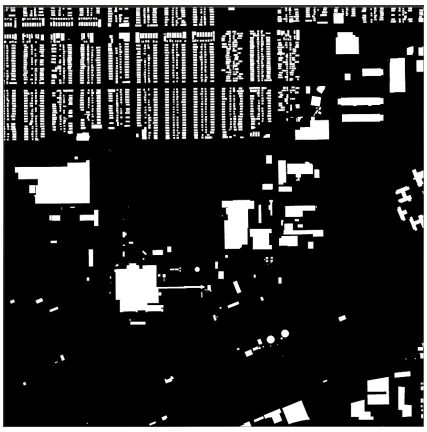



Figure 5.2: Chicago 3-color channel image (left) and Chicago reference image (right)

5.5 Implementation

In previous chapter, we have addressed the challenges for building a complete pipeline to efficiently acquire, store, preprocess, process and analyze big dataset. We have also elaborated on the rationale behind our decision on picking up different components in the pipeline. In this chapter, we will present detailed implementation of two experiments we have applied our proposed solution: one consumes multiple CPUs across machines/nodes (ranging from 2 to 16 nodes) to train distributed deep learning U-net model, the other trains the model locally with 1 GPU. We deliberately outline the four phases of two experiments side-by-side in Figure 5.3. In this way, the common steps between two experiments will be placed in the middle of the figure. Each experiment has it own working steps due to the differences between local and distributed environment, which is marked by postfixes (e.g 2.1 and 2.2 are the second steps in distributed CPUs and local GPU experiments respectively). We mainly focus on the illustration of steps 2, 3, 4.1, 5.1, 6 (6.1 and 6.2) and 7 since it enables us to answer our research questions.

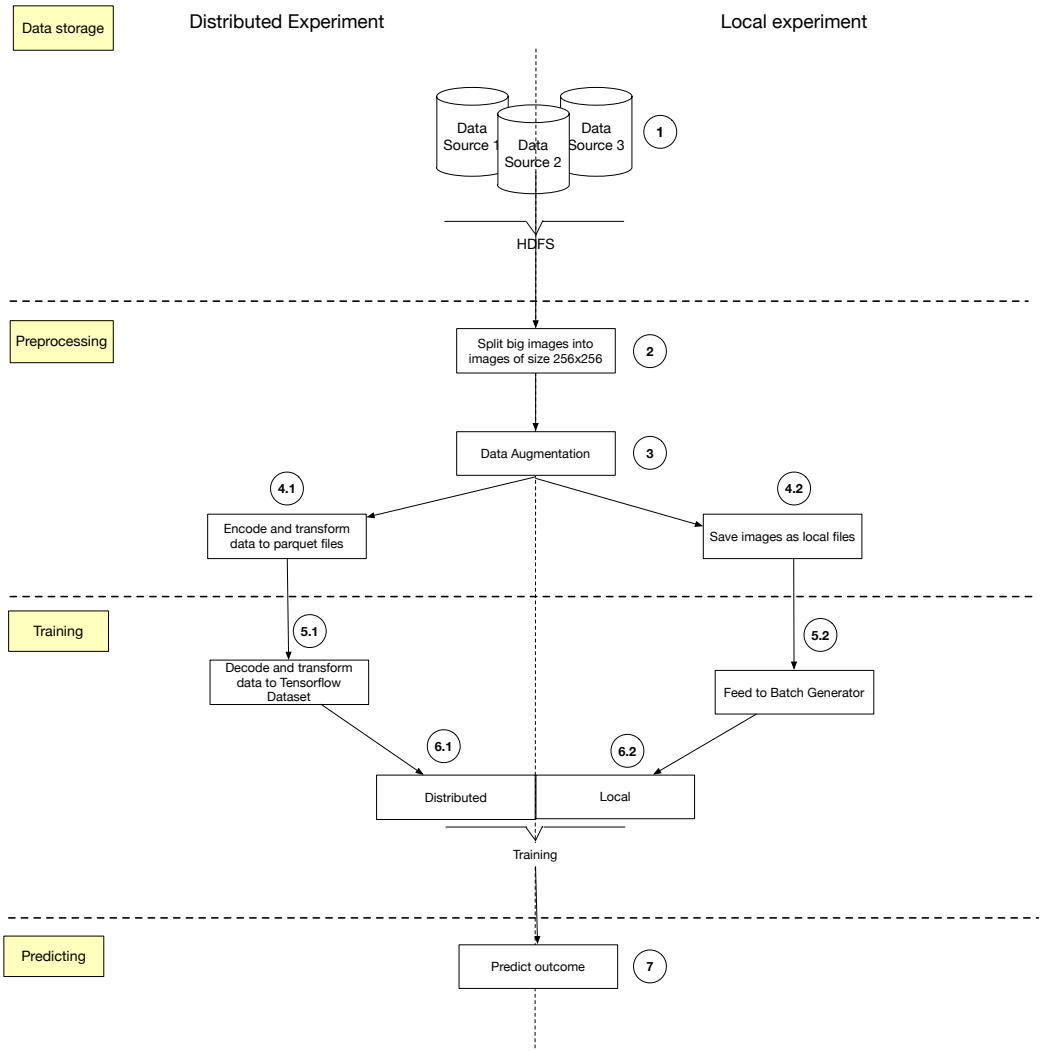


Figure 5.3: Distributed and local experiments with common steps in the middle line and different steps on the different sides

5.5.1 Preprocessing - Step 2, 3, 4.1

This phase is certainly driven by the next phase - processing when data have to be transformed and saved as intermediate products that can facilitate the processing stage. As mentioned before, the data set used in this research includes images of size 5000 x 5000 with 3 channels whose volume is relatively large for a sophisticated deep learning model. We therefore use slide window of size 128 x 128 (Figure 5.4) to split each big image into smaller pieces in step 2. There will be overlaps between pieces at edges of each images which is acceptable because we want to avoid cutting an piece with area out of image space (Figure 5.5). The pseudo code of splitting algorithm is described in the algorithm box 1. We apply this algorithm to both feature and reference images.

Algorithm 1: Split image

Input: Original image
Output: Small pieces of image with size 128 x 128

```
1  i ← 0
2  j ← 0
3  while i is still less than height of original image do
4      while j is still less than width of original image do
5          top_left_x ← j
6          top_left_y ← i
7          if top_left_x + 128 is greater than image width then
8              top_left_x ← imagewidth − 128
9          if top_left_y + 128 is greater than image height then
10             top_left_y ← imageheight − 128
11         snip an image within the box (top_left_x, top_left_y,
             top_left_x+128, top_left_y+128)
12         save new image to HDFS
           /* Slide window to the right                          */
13         top_left_x ← top_left_x + 128
       /* Slide window to the bottom                             */
14     top_left_y ← top_left_y + 128
```

At last of this phase (step 3 and 4.1), all feature and labels images needs to be augmented and transformed into parquet files that are in appropriate format for training phase. The detailed steps will be shown below:

1. Read all feature and reference images from HDFS as binary files into resilient distributed dataset. [39] (RDD)

2. Convert the binary form of images into numpy array, followed by loss-less data compression (NdarrayCodec class from petastorm package [6]). Optionally, data augmentation of numpy array of each image can be done at this step.

3. Associate feature images with corresponding reference images by their names.

[6] https://petastorm.readthedocs.io/en/latest/api.html

Figure 5.4: Continuous slide windows

Figure 5.5: Overlapped windows at edges of an image

4. Convert RDD to Dataframe [7] and save as parquet files. The schema of parquet files is shown in the table 5.2.

Table 5.2: Preprocessed Parquet File Schema

Field Name	Field Type	Encode algorithm
features (feature data)	np.uint8	NdarrayCodec
masks (reference data)	np.uint8	NdarrayCodec

5.5.2 Training - Steps 5.1, 6.1 and 6.2

A slightly modified U-Net deep learning architecture will be applied to do semantic segmentation of feature images in preprocessed input dataset of images of size 128 x 128 (Figure 5.6). Two drop out layers (rate = 50%) d4 and d5 will be added to original architecture (Section 3.4.2) right after convolutional layer 4 and 5 respectively to prevent overfitting when the number of parameters are fairly large. Also, convolutional layer 10 (size 128x128 with 1 filter/kernel) needs to be added at the end of the architecture since the outcomes of the model are one-color-channel images with size 128 x 128.

The left hand side is the contraction path (Encoder) where we apply regular convolutional and max pooling layers. In the Encoder, the size of the image gradually reduces while the depth gradually increases, starting from 128 x 128 x 3 to 8 x 8 x 1024. This basically means the network detects the "WHAT" information in each image. The right hand side is the expansion path (Decoder) where we apply upsampling along with regular convolutions. In the decoder, the size of the image gradually increases and the depth gradually decreases, starting from 8 x 8 x 1024 to 128 x 128 x 1. Intuitively, the Decoder recovers the "WHERE" information (precise localization) by gradually applying up-sampling. Below is the detailed explanation of the architecture:

- **c1, c2, . . . , c9** are the output tensors of Convolutional Layers of kernel size 3x3 using padding to keep the size of input.
- **p1, p2, p3 and p4** are the output tensors of Max Pooling Layers of kernel size 2x2.
- **merge 6, merge 7, merge 8 and merge 8** are the output tensors of Merge Layers of tensors with same size.
- **u6, u7, u8 and u9** are the output tensors of up-sampling layers followed by convolutional layers of kernel sizes which corresponding with left merged tensors.
- **c10** is the output of tensor of Convolutional Layer of kernel size 1x1 so that the output of the architect can classify pixels into two categories.
- **d4, d5** are the output tensors of Dropout layers (rate = 50%).

[7] https://spark.apache.org/releases/spark-release-1-3-0.html

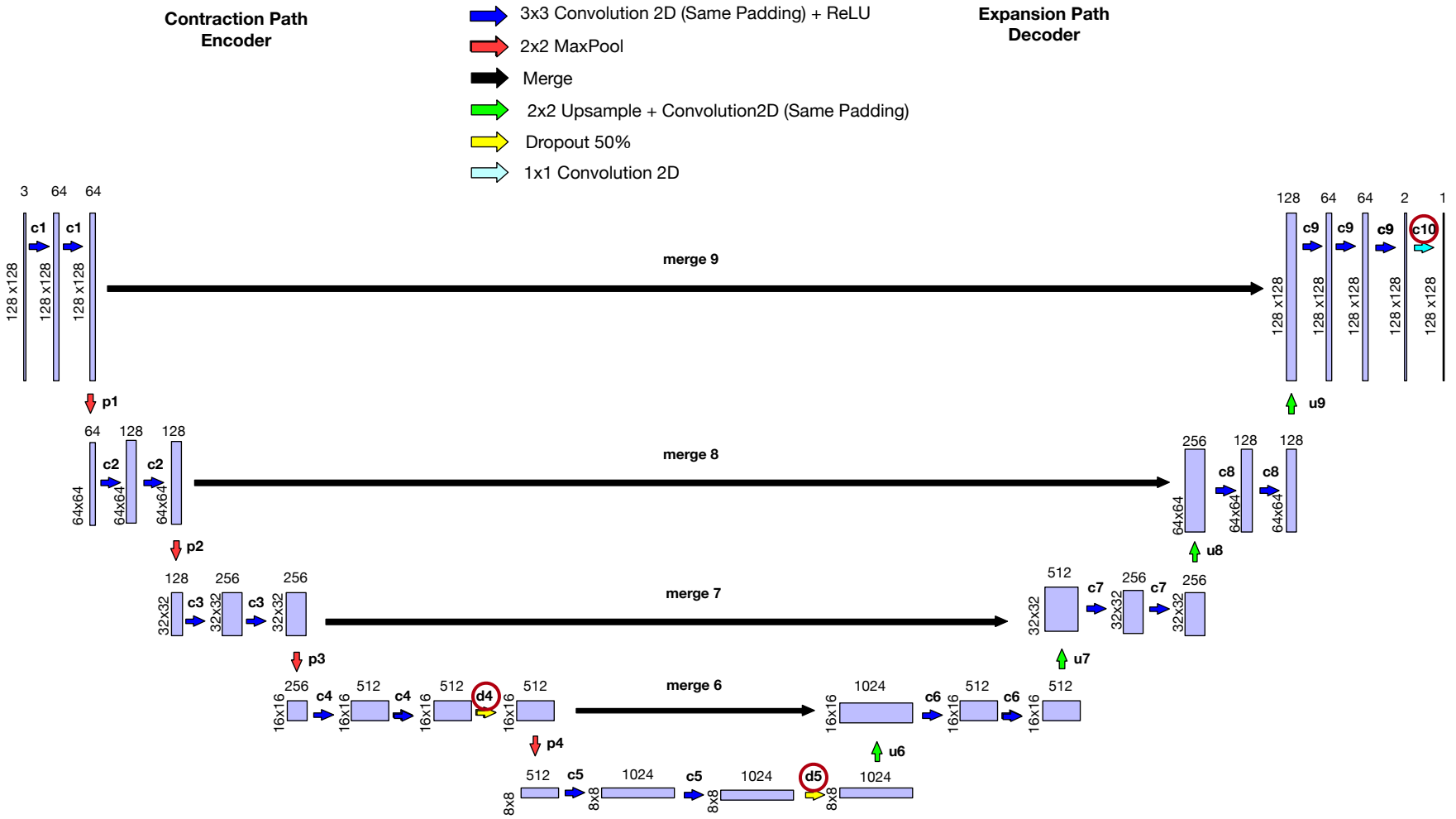


Figure 5.6: Modified U-Net deep learning model architecture. Additional layers to the original model are marked as red circles

The setup of local and distributed U-net model will be described at listing 5.1 and listing 5.2 accordingly. Listing 5.3 and Listing 5.4 present the differences between running training model with local and distributed manner. The additional steps to make local model distributed are highlighted in red; black paragraphs are steps that are the same in local and distributed versions; the blue number indicators show the similar steps. Since we focus on how we can easily distribute deep learning model rather than itself, we will not dig deep into the construction of deep learning model which is exactly the same in both approaches. The source code for our model is available under our repository[8].

The training will stop when 30 epochs have accomplished or an early stop with patience equals 5 occurs. In other words, if the accuracy of model is not improved by 5 continuous epochs, the training phase will come to a halt. 16 are reasonably chosen as batch size as the total volume of images fits our available GPU memory. The outcome of training phase of both local and distributed versions will be saved as `unet_building.hdf5` file by message passing interface (MPI) process with rank = 0 [40]

Listing 5.1: Step 6.2 - Local U-net model setup

```
1. Define image size
2. Define number of image channels

# Local model is available at
# https://github.com/ThrowMeForALoop/RemoteSensing/blob/
    master/train_model.py

3. Init local deep learning model.

# Use Adam optimizer for deep learning model
4. Init Adam optimizer.

5. Compile local deep learning model with binary cross
   entropy loss and accuracy metric.
```

[8]https://github.com/ThrowMeForALoop/RemoteSensing/blob/master/train_model.py

Listing 5.2: Step 6.1 - Adapt local U-net model to distributed version by just modifying local optimizer

```
1. Define image size
2. Define number of image channels

# Local model is available at
# https://github.com/ThrowMeForALoop/RemoteSensing/blob/
    master/train_model.py

3. Init local deep learning model.

# Use Adam optimizer for deep learning model
4. Init Adam optimizer.

# Modify local to distributed optimizer
opt = horovod.DistributedOptimizer(opt)

5. Compile local deep learning model with binary cross
   entropy loss and accuracy metric.
```

Listing 5.3: Step 6.2 - Run local training steps

```
1. Configure tensorflow with GPUs.

(8) Create local training data generator to feed training
   images to U-net model by batch

(9) Create local validation data generator to feed images
   to the model in validation steps.

(10) Start training model with training and validation
   datasets
```

Listing 5.4: Step 6.1 - Run training model in multiple executors to scale it up

```
# The steps occurs in each executor:

1. Configure tensorflow with GPUs

2. Horovod: initialize Horovod inside each trainer (
   executor).

3. Horovod: pin GPU to be used to process local rank (one
   GPU per process), if GPUs are available

4. Horovod: restore from checkpoint, use hvd.load_model
   under the hood

5. Horovod: adjust learning rate based on number of
   processes
```

```
6. Add Model Checkpoint callback which stores temporary
   model

7. Horovod: save checkpoints only on the first worker to
   prevent other workers from corrupting them

(8) Decode each row of training parquet table and convert
   it to Tensorflow records

(9) Decode each row of validation parquet table and convert
    it to Tensorflow records

(10) Train the model with local data in each executor

#The step takes place in global manner, which is started by
    driver application.

1. Serialize local model to bytes

2. Create Spark session to distribute training tasks

3. Horovod: run training processes with \texttt{train\_fun}
    function in spark executors.
```

5.5.3 Predicting - Step 7

There are two ways to predict and classify images using the result from previous phase (`unet_building.hdfs` file). The first way is to distribute the predicting function with Spark as we did with training function. The other is to locally predict outcome of each test image. In this research, we are in favour of the later approach because it is not only sufficient for small test dataset (only 20% of big dataset) but also simplify visualization and data analysis. The output of predicting phase is a numpy array of size 128 x 128 x 1, whose elements indicate the probability of pixels of the input image over two classes: building or not-building. As general, we select 0.5 as threshold to distinguish between building or non-building pixel.

Chapter 6

Results

In this chapter, we will evaluate our two experiments' result in terms of scaling up training model as well as how accurate distributed training model is, based on inference of local training. In the first experiment, we will scale up training by increasing number of machines/nodes from 2 to 8 nodes by a step of 2 (each node uses 8 cores). The scarcity of GPUs forces us to use CPUs as an alternative and reduce the number of training and validation images to 128 and 64 post-split images of size 128 x 128 respectively (Images are collected after splitting process) since it is infeasible to train a complex deep learning with a big dataset with CPUs. Fortunately, as mentioned before, the result of the second experiment with 1 GPU on entire big dataset can provide a supplement to the result of first experiment on the same dataset because Horovod has already proved that as long as the global-mini batch remains, there is no difference between the performance of distributed and local deep learning models.

To scale up a complicated U-net deep learning model, total 30 lines of codes are modified and added, which accounts for roughly 18% of original source code. The outcome of model training will be introduced in the next two sections.

6.1 Scaling up

In figure 6.1 we can see the measurements of the average training time per epoch using different number of nodes/machines. The raw data can be seen in table 6.1. From first sight, we can see that distributed deep learning does help to decline training time, especially when increasing number of nodes from 1 to 2 nodes. An interesting observation can be made for the measurements of 4 nodes and 8 nodes experiments. There are only slight improvement in terms of training times even if we double the number of nodes.

To further investigate why training time has not been reduced as expected when more nodes were added to our tests, we implement a script to collect average CPUs usage in each node in case of 8-node test. Surprisingly as seen from table 6.2, our training process using Horovod can not utilize the whole CPU power. Only two cores out of 8 cores in each machine were stressed out to work at almost full capacity while the others only use around

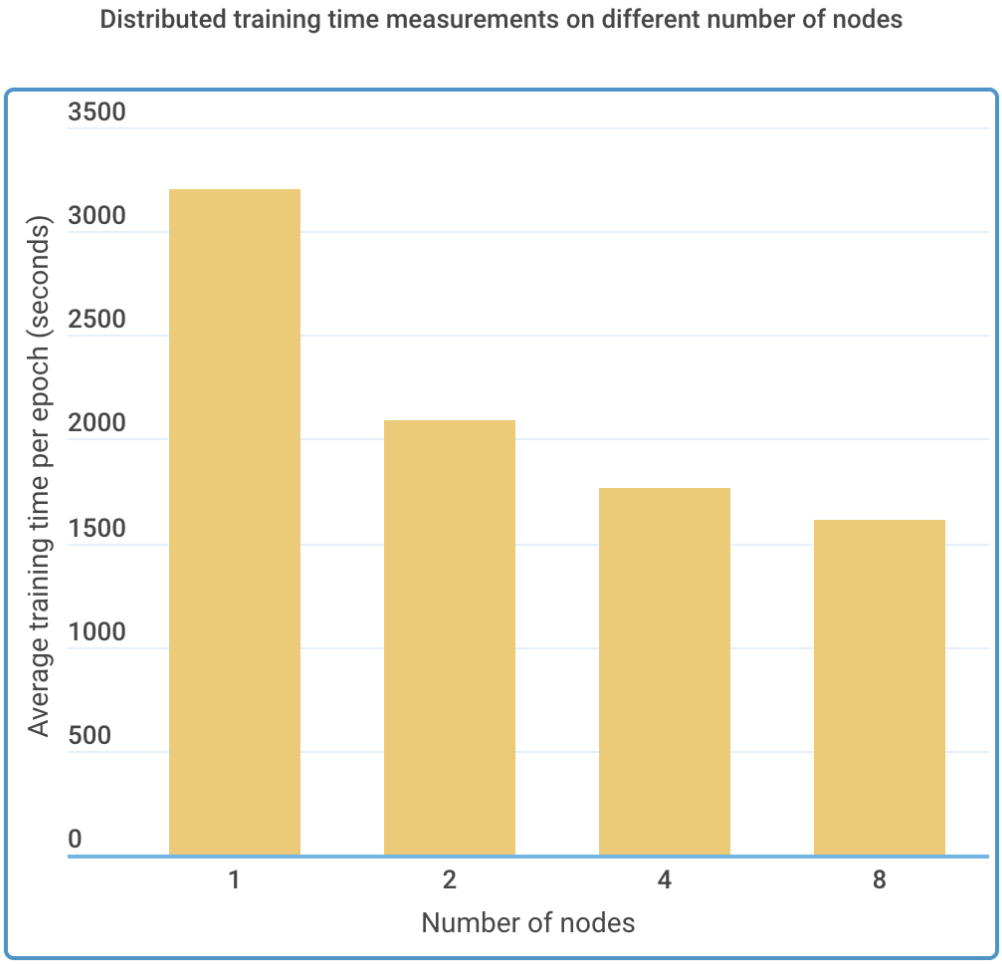


Figure 6.1: Measurements of training time per epoch

30% of their designed capacity. From our point of view, Horovod has not been optimized for DAS-4 machine's CPUs with AVX architecture although Horovod developers have recently had an hot-fix version for this processor architecture at https://github.com/horovod/horovod/pull/1083

6.2 Performance of trained model

When evaluating a standard deep learning model, we usually classify our predictions into four categories: true positives, false positives, true negatives, and false negatives [41]. For the dense prediction task of image segmentation, it's not immediately clear what counts as a "true positive" since classification of each pixel alone can not entirely reflect how well the model detects separate segments of an image in a specific context. To reinforce the evaluation of a deep learning model, Intersection over Union (IoU) [42] is usually used along with accuracy as two evaluation metrics for semantic segmentation. The formula of Intersection over Union can be seen in the figure 6.2:

Table 6.1: Scaling up training model

Number of nodes	Batch Size	Average training time for 1 epoch (seconds)
1	16	3204
2	8	2092
4	4	1860
8	2	1709

Table 6.2: Avarage core usage during training with Horovod

Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8
87.0%	36.5%	38.0%	32.9%	32.1%	85.4%	20.7%	20.3%

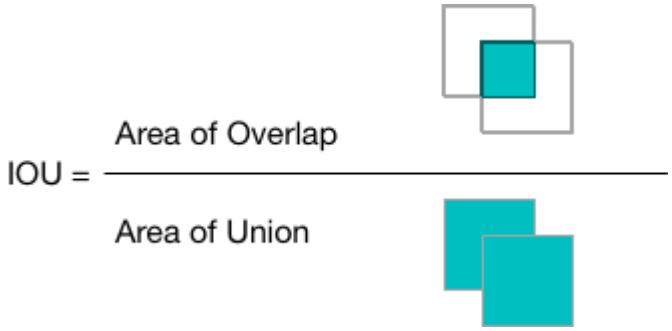


Figure 6.2: Computing the Intersection of Union is a division of the area of overlap between the bounding boxes by the area of union

As seen from table 6.3, we achieve very high accuracy and IoU. In practice, an IOU score [43] > 0.5 is normally considered a good prediction. Also, we examine the ground truth data and predicted data from our model to validate our result. Figure 6.3 and figure 6.4 show the close similarities in the positions of white pixels that represent segments of buildings. To put it another way, our model is able to capture precisely building's pixels except those around the edge. This weakness will be improved in the near future.

Table 6.3: Semantic Segmentation accuracy and IOU of our trained model

Mean accuracy	Mean IOU
0.95	0.66

(a) Ground truth data of a Austin's building

(b) Outcome of our trained model for an Austin's building

Figure 6.3: Comparison between ground truth and our outcome of our trained model for an Austin's building

(a) Ground truth data of a Chicago's building

(b) Outcome of our trained model for an Chicago's building

Figure 6.4: Comparison between ground truth and our outcome of our trained model for an Chicago's building

Chapter 7

Conclusion

With this thesis, we initially showed the needs for applying deep learning in Remote Sensing field as well as obstacles that has slowed down the advancements in distributed deep learning researches and industrial applications. Unfortunately, as far as we know, there has been no end-to-end solution to cope with these issues at large scale.

With the interest of overcoming those drawbacks, we proposed a complete pipeline that helps data scientists, researchers and big data developers to easily scale up their deep learning model by adding several lines of code without worry about overheads generated by communication between machines/nodes. For example, to distribute the training of a sophisticated local U-net deep learning model, we only need to add to and modify 30 lines of code in total concerning 131 lines of code in local model. In other words, little effort need to be made in order to scale up local deep learning model and this in turn answered our first question. For a simpler deep learning model to classify handwritten digit MNIST [44], this number is much smaller.

By evaluating different big data platforms in terms of performance, scalability, distributed learning support, we have chosen Horovod with Tensorflow as back-end, Apache Spark and Petastorm as core components for our proposed pipeline. Put the matter another way, we have sufficient factors to build and end-to-end distributed learning pipeline which fulfilled the second research question. Two applications for Remote Sensing Semantic Segmentation were implemented to prove the validity as well as efficiency of our pipeline. The first application followed strictly to our pipeline in order to store, preprocess and distribute deep learning tasks across multiple computers. Due to the shortage of GPU resources, we can only run this application on multiple CPUs to do semantic segmentation on a small dataset of 128 images using modified U-Net architecture. To generalize our result for the bigger dataset, another application was implemented and run locally on 1 GPU Titan Black.

Testing those applications by using DAS-4 environment showed interesting results. As for distributed training, our proposed pipeline did reduce the training time especially when increasing number of nodes from 1 to 2. However, Horovod, a component in our pipeline struggles when adding more machines/nodes. This is caused by the lack of CPUs optimization in Horovod implementation for AVX processor architecture. For the performance of

trained model, the local application indirectly showed an exceptional result of distributed version. Accuracy and IoU are around 0.95 and 0.66 respectively despite of minor issues of misclassifying pixels around the edge of buildings.

The research in this thesis has raised a few new questions which can be pursued in future research.

Firstly our proposed pipeline is composed of different software like HDFS, Spark, Horovod, Petastorm and Tensorflow. These pieces of software were personally chosen in the architecture based on some important criteria. Maybe some other software that may be less widely known, or does not fit the aspects precisely may end up being better. For example, HDFS was selected due to its popularity as well as the versatility in storing various file formats though it has not been an absolutely ideal file system for separate small files like images. [45]

Since we are not able to run our experiments with multiple GPUs, we wonder whether our pipeline can achieve as much scaling efficiency as Horovod's benchmark on other deep learning models (Inception V3 and ResNet-101). If the performance of Horovod is not as good as expected, which platforms can be used as alternatives ?

In the chapter 4, we mentioned the transformation step which convert parquet files into Tensorflow dataset. Although it is necessary to make preprocessed data applicable to training phase, we believe that there will be better approaches. For example, we can implement a library to convert images directly to Tensorflow dataset or other types which match to deep learning framework running on top of Horovod.

The last question is how to improve the performance of our U-net deep learning model even though our research has not concentrated on deep learning performance tuning. One of trendy approaches is "Transfer learning" where pre-trained models are used as the starting point on computer vision tasks given the vast compute and time resources required to develop neural network models on the problem.

Bibliography

[1] Skymind. DL4J Distributed Training: Technical Explanation. `https://deeplearning4j.org/docs/latest/deeplearning4j-scaleout-technicalref#parameteravg`, 2018. [Online; accessed 13/03/2019].

[2] Jeff Hale. Deep learning framework power scores 2018. `https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a`, 2018. [Online; accessed 18/06/2019].

[3] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[5] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[7] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[8] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[9] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and

Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *2011 IEEE 27th International Conference on Data Engineering*, pages 231–242. IEEE, 2011.

[10] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[11] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

[12] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[13] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. On parallelizability of stochastic gradient descent for speech dnns. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 235–239. IEEE, 2014.

[14] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[15] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.

[16] James B Campbell and Randolph H Wynne. *Introduction to remote sensing*. Guilford Press, 2011.

[17] Natural Resources Canada. Fundamentals of Remote Sensing - Introduction. `https://www.nrcan.gc.ca/earth-sciences/geomatics/satellite-imagery-air-photos/satellite-imagery-products/educational-resources/9363`, 2015. [Online; accessed 13/03/2019].

[18] Martin Längkvist, Lars Karlsson, and Amy Loutfi. A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, 42:11–24, 2014.

[19] The University of New South Wales. Machine Learning Tutorials. `https://www.cse.unsw.edu.au/~cs9417ml/MLP2/BackPropagation.html`. [Online; accessed 13/03/2019].

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[21] Justin Johnson and Andrej Karpathy Stanford. Cs231n: Convolutional neural networks for visual recognition.

[22] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[23] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[24] Shervin Ardeshir, Kofi Malcolm Collins-Sibley, and Mubarak Shah. Geo-semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2792–2799, 2015.

[25] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[26] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[27] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[28] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[29] Lei Gu and Huan Li. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 721–727. IEEE, 2013.

[30] Spark standalone mode. `https://spark.apache.org/docs/latest/spark-standalone.html`. [Online; accessed 18/06/2019].

[31] Cheng O. Gruener, R. and Y. Litvin. Introducing petastorm: Uber atg's data access library for deep learning. `https://eng.uber.com/petastorm/`, 2018. [Online; accessed 18/06/2019].

[32] Apache Parquet. `https://parquet.apache.org/`. [Online; accessed 18/06/2019].

[33] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[35] DAS-4. `https://www.cs.vu.nl/das4/`. [Online; accessed 18/06/2019].

[36] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[37] Emmanuel Maggiori, Yuliya Tarabalka, Guillaume Charpiat, and Pierre Alliez. Can semantic labeling methods generalize to any city? the inria aerial image labeling benchmark. In *2017 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 3226–3229. IEEE, 2017.

[38] Andrew NG. Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization. `https://www.coursera.org/learn/deep-neural-network`. [Online; accessed 18/06/2019].

[39] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[40] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced features of the message passing interface*. MIT press, 1999.

[41] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[42] Md Atiqur Rahman and Yang Wang. Optimizing intersection-over-union in deep neural networks for image segmentation. In *International symposium on visual computing*, pages 234–244. Springer, 2016.

[43] Pang-Ning Tan. *Introduction to data mining*. Pearson Education India, 2018.

[44] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

[45] Liu Jiang, Bing Li, and Meina Song. The optimization of hdfs based on small files. In *2010 3Rd IEEE international conference on broadband network and multimedia technology (IC-BNMT)*, pages 912–915. IEEE, 2010.