Master Thesis

Implementation of Face Recognition system over Spark, with TensorFlow

by
Chatzistyi Liia

Athens, October 2017
Implementation of Face Recognition system over Spark, with TensorFlow

By

Chatzistyli Ilia

This master thesis has been developed in cooperation with DBLabs

Supervisors:

Titsias Michael, Assistant Professor at the Department of Informatics in Athens University of Economics and Business

Korkofigkas Antonis, in charge of DBLabs
Abstract

This MSc thesis deals with the subject of Face Recognition over a cluster; a currently developing technology with multiple real-life applications. The goal is to develop a complete cloud-based cluster for Face Recognition by using high developed software and big-data frameworks.

The face recognition is accomplished by a Deep Learning method based on state-of-the-art Convolutional neural networks (CNN) which are used to extract relevant facial features and compare faces between them in an efficient way. This topology is trained and evaluated using the Labelled Faces in the Wild (LFW) dataset.

The platform was formed by using a range of big-data frameworks. In this context, the main infrastructure is an Amazon cluster over which Apache’s Hadoop and Spark were implemented. Upon this cluster Tensorflow was used, which is Yahoo’s Big Data Machine Learning (ML) platform, named TensorflowOnSpark. This was used for conducting machine learning and deep neural networks for the face recognition project. Also, a streaming process was created to approach a real-time implementation, where the snapshots enter the created system with Spark Streaming and after appropriate pre-treatment (which involves alignment of the faces with multi task CNN processes) are trained or classified.

In the course of the experimental study, we focused on the performance of Tensorflow over Spark compared to a standalone execution. We conclude, that the distributed solution gives real-time results.

Keywords

Face recognition, Convolutional Neural Networks, Machine Learning, Neural Networks, Computer Vision, AWS, Apache Hadoop, Apache Spark, Spark Streaming, Python, LFW, Tensorflow, TensorflowOnSpark, Facenet
Table of Contents

1 Introduction .................................................................................................................. 1
  1.1 Face Recognition Problem...................................................................................... 1
  1.2 Problem Formulation ............................................................................................ 1
  1.3 Related Work........................................................................................................... 2
  1.4 Thesis Outline ......................................................................................................... 2

2 Theoretical Basis .......................................................................................................... 5
  2.1 Multi-Layer Neural Networks ................................................................................. 5
    2.1.1 Architecture ..................................................................................................... 5
    2.1.2 Loss Function .................................................................................................. 7
  2.2 Convolutional Neural Networks .............................................................................. 8
    2.2.1 Layers Types .................................................................................................... 9
  2.3 Deep Convolutional Networks ............................................................................... 10
  2.4 Multi-task Cascaded Convolutional Networks [15] ............................................... 12
    2.4.1 Approach ....................................................................................................... 12

3 Big Data Frameworks .................................................................................................. 15
  3.1 Hadoop .................................................................................................................. 15
    3.1.1 Hadoop Distributed File System ..................................................................... 15
    3.1.2 MapReduce 1.0 ............................................................................................... 16
    3.1.3 YARN ............................................................................................................. 17
  3.2 Spark ....................................................................................................................... 19
    3.2.1 Spark Components ......................................................................................... 19
    3.2.2 RDD ............................................................................................................... 20
    3.2.3 Spark Streaming ............................................................................................. 21
  3.3 Apache Spark and Hadoop: Working Together ................................................... 23

4 Tensorflow .................................................................................................................... 25
  4.1 TensorflowOnSpark ............................................................................................... 28
    4.1.1 Command Line Interface ................................................................................. 30
    4.1.2 Application Programming Interface (API) ................................................... 30

5 Technical Specifications ............................................................................................... 31
  5.1 Infrastructure ......................................................................................................... 31
5.2 Cluster Architecture .................................................................................. 32
  5.2.1 Terraform ............................................................................................ 32
  5.2.2 Amazon Web Credentials & key-pair Setup ....................................... 33
  5.2.3 Instances Set up .................................................................................... 33

6 Methodology .................................................................................................. 37
  6.1 Dataset ...................................................................................................... 37
  6.2 Methodology ............................................................................................. 37
    6.2.1 Conversion Guide from TensorFlow to TensorFlowOnSpark .......... 37
    6.2.2 Code Pre-processing .......................................................................... 38
  6.3 Streaming process .................................................................................... 39
  6.4 Train a Classifier on LFW Dataset ............................................................ 39

7 Experiment Results ....................................................................................... 41
  7.1 Face Recognition Process ......................................................................... 41
  7.2 Streaming process .................................................................................... 42
  7.3 Non-distributed Classification .................................................................... 42

8 Conclusion and Future Work ....................................................................... 43
  8.1 Conclusion ................................................................................................ 43
  8.2 Future Work ............................................................................................. 43

9 Bibliography .................................................................................................. 45

10 Appendices ................................................................................................... 49
  10.1 Terraform Scripts ................................................................................... 49
  10.2 Set root access to instances ..................................................................... 61
  10.3 Instance's Start – Stop scripts .................................................................. 62
  10.4 Multiple Face Alignment script ............................................................... 62
  10.5 Face Recognition scripts ......................................................................... 66
    10.5.1 Alignment script ............................................................................... 66
    10.5.2 Training script .................................................................................. 67
    10.5.3 Classification script .......................................................................... 67
  10.6 Standalone execution scripts .................................................................... 68
  10.7 Spark streaming script ............................................................................. 69
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Detection-Alignment-Recognition pipeline</td>
<td>2</td>
</tr>
<tr>
<td>2-1</td>
<td>Analogy of biological neuron (left) and of the model of neural network (right)</td>
<td>5</td>
</tr>
<tr>
<td>2-2</td>
<td>Neural network with three layers</td>
<td>6</td>
</tr>
<tr>
<td>2-3</td>
<td>Illustration of training with triplet loss</td>
<td>8</td>
</tr>
<tr>
<td>2-4</td>
<td>Example of a case justifying padding</td>
<td>9</td>
</tr>
<tr>
<td>2-5</td>
<td>Inception module, naïve version (left)</td>
<td>Inception module with dimension reduction (right)</td>
</tr>
<tr>
<td>2-6</td>
<td>The Facenet network architecture</td>
<td>11</td>
</tr>
<tr>
<td>2-7</td>
<td>Pipeline of cascaded framework which includes three-stage multi-task deep CNN</td>
<td>12</td>
</tr>
<tr>
<td>3-1</td>
<td>The architecture of a Hadoop File System</td>
<td>16</td>
</tr>
<tr>
<td>3-2</td>
<td>MapReduce 1.0</td>
<td>17</td>
</tr>
<tr>
<td>3-3</td>
<td>YARN. The cluster is composed by one master and 4 slave nodes</td>
<td>18</td>
</tr>
<tr>
<td>3-4</td>
<td>Differences between the Components of Hadoop 1 and 2.</td>
<td>19</td>
</tr>
<tr>
<td>3-5</td>
<td>Spark Components</td>
<td>20</td>
</tr>
<tr>
<td>3-6</td>
<td>Spark Streaming (1)</td>
<td>22</td>
</tr>
<tr>
<td>3-7</td>
<td>Spark Streaming (2)</td>
<td>22</td>
</tr>
<tr>
<td>3-8</td>
<td>Spark over Hadoop deployments</td>
<td>23</td>
</tr>
<tr>
<td>4-1</td>
<td>Binary counter neural network with matrices</td>
<td>26</td>
</tr>
<tr>
<td>4-2</td>
<td>Tensor Graph from Tensorboard</td>
<td>27</td>
</tr>
<tr>
<td>4-3</td>
<td>ML Pipeline with multiple programs on separated clusters</td>
<td>28</td>
</tr>
<tr>
<td>4-4</td>
<td>TensorFlowOnSpark for deep learning on Spark Cluster</td>
<td>28</td>
</tr>
<tr>
<td>4-5</td>
<td>TensorFlowOnSpark system Architecture</td>
<td>29</td>
</tr>
<tr>
<td>5-1</td>
<td>Visual view of instance in AWS site</td>
<td>35</td>
</tr>
<tr>
<td>6-1</td>
<td>Processing Schema</td>
<td>38</td>
</tr>
<tr>
<td>7-1</td>
<td>Example of Aligned photo</td>
<td>41</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Face Recognition Problem

Inside a computer, images are represented by matrices with numbers corresponding to colour. The process of categorizing images into one or more classes is called image classification and is a difficult problem for a machine. There are several solutions to this, which span from the simple nearest neighbour classification to more complex neural network approaches. Face recognition is an image classification problem and can be solved with neural networks.

It is trivial for most people to recognize someone they have seen before, but how the brain processes signals from the eyes is still unknown. This makes Face Recognition an interesting problem. How can we make a computer interpret images like a human? What are the important features and how to process them?

Face Recognition is one of the areas from Computer Vision (CV) that has drawn more interest for long. The practical applications for it are many, ranging from biometrical security, surveillance to identification in login systems and personalized technology. Because of the possibilities, it has attracted interest in various areas. Thus, it is a really difficult problem, and in the past years’ quality results have been obtained. In fact, this problem is usually split into different sub-problems to make it easier to work with, mainly face detection in an image, followed by the face recognition.

Through the years, many algorithms and techniques have been used, such as eigenfaces or Active Shape models. However, the one that is currently used, and provides the best results, consists in Deep Learning (DL), especially the Convolutional neural networks (CNN) are the state-of-the-art method for face recognition. Finally, these methods are currently obtained high quality results.

1.2 Problem Formulation

The best face recognition methods are based on CNN and require a long time to train. The thesis tackles this issue by aiming to apply face recognition and compare based on a state-of-the-art system based CNN. The Facial recognition can be thought of as part of a Detection – Alignment – Recognition pipeline [1]. Each phase in the pipeline generates the input for the next phase as show in Figure 1-1.
In detection phase, the positions of faces in an image are located. Usually, the largest face found is considered to belong to the person identified by that image. The rectangular bounding box of the largest face found is then cropped from the image. In the alignment phase, the cropped image is transformed to a canonical pose, which generally means rotating and translating the image so that the eyes would be level and the nose approximately in the centre of the image. Finally, the recognition phase involves matching the aligned image to either a known identity or comparing two images to guess whether they belong to the same person.

1.3 Related Work

This thesis explores a CNN method, mainly based on FaceNet [2]. FaceNet can directly map the face image to Euclidean space, the distance of space represents the similarity of face images. As long as the mapping space is generated, face recognition, authentication and clustering tasks can easily complete. The method is based on deep Convolutional neural network with high performance at the Labelled Faces in the Wild (LFW) benchmark.

As far as, the implementation the main idea was the use of Tensorflow [3], which was originally developed by researchers and engineers working on the Google Brain team within Google’s Machine Intelligence Research organization, for the purposes of conducting machine learning and deep neural networks research. The system is general enough to be applicable in a wide variety of other domains, as well.

Based on the idea of Tensorflow [3], developed by Google, in this Thesis was used TensorflowOnSpark [4]. TensorflowOnSpark bring scalable deep learning to Apache Hadoop [5] and Apache Spark [6] clusters by combining salient features from deep learning framework Tensorflow and big-data frameworks Apache Spark and Hadoop. TensorflowOnSpark enables distributed learning on a cluster of GPU and CPU servers.

1.4 Thesis Outline

Finally, a brief description of the contents of the diploma thesis is presented.
On the **second chapter**, there is a brief description of the theoretical basic of the face recognition methods used. There is an introduction on Convolutional Neural Networks and the implemented Deep Learning methods.

Since this thesis is a practical implementation of the aforementioned methods, in the **third chapter** there is a detailed description of the tools used to implement the Face Recognition system. In particular, is described how Apache’s tools, Hadoop and Spark, operate separately but also in combination, for the creation of the Spark cluster.

Since, the cluster created upon Amazon’s machines, in **chapter four** are described the steps followed to create this. There is a thorough description of the software tools used, such as Terraform script and further configurations.

The next two chapters include the **Methodology** and the **Experimental results**. First, is explained the guide line followed, the used dataset and the experimental procedure. The main concept is about configuring a real-time face recognition process, this includes the creation of a classifier based on the faces of the dataset (training) and then an examination of this, upon new coming faces. Also, it was implemented a streaming process which automates the face recognition, so that every time a new picture reaches Hadoop File System a classification procedure is initialized. For the experimental phase, the performance of this Spark system was compared with this of a standalone instance.

Finally, the last chapter includes the **conclusion** and a brief representation of further improvements or possible work upon this face recognition system.
2 Theoretical Basis

This chapter introduces the fundamentals of Depp Neural Networks (DNN) and describes the theoretical basis on Convolutional Neural Networks.

2.1 Multi-Layer Neural Networks

The simplest form of deep Artificial Neural Network (ANN) is feed forward Multi-Layer Neural Network (MLNN). Similarly, as biological neurons in our brain, artificial neuron is the elementary building block in an ANN (Figure 2-1). Its function is to receive, process and transmit signal information. Artificial neuron receives one or more input units corresponding to dendrites in the brain.

$i_{th}$ input unit will be $\equiv$the inputs are weighted by real numbers expressing the importance of the respective inputs to the output (denoted by $w_i$).

![Figure 2-1 Analogy of biological neuron (left) and of the model of neural network (right)]

Another important term is bias (denoted by b), which adds constant values to the input. In biological terms, a bias can be considered to be a measure of how easy is to get a neuron to fire.

2.1.1 Architecture

Neural networks consist of layers with nodes. Figure 2-2 illustrates a basic neural network with three layers and Figure 2-1(right) illustrates a single node. Each layer is connected to the next with weight w, which are multiplied to the output of the nodes in the previous layer, x. Each node additionally has a bias, b. The input layer has three nodes that are fully connected to the four nodes in the hidden layer which in turn are fully connected to two output nodes. Layers
between input and output layers are called hidden layers and these transform the input to something that the output layer can use.

![Diagram of a neural network with three layers](image)

*Figure 2.2 Neural network with three layers*

Each connection gives the node a signal. These signals are added to each other to produce a sum \( z \), as in equation (2.1) where \( n \) is the number of connections to the previous layer. The sum is then passed through an activation function such as ReLU or Logistic, which produces the node output, \( y \). Equation (2.2) and (2.3) demonstrate how the node output is calculated with ReLU and Logistic activation functions respectively.

\[
z = \sum_{i=1}^{n} w_i \cdot x_i + b \quad (2.1)
\]

\[
y = \max (0, z) \quad (2.2)
\]

\[
y = \frac{1}{1 + e^{-z}} \quad (2.3)
\]

The network architecture is important and needs to be designed to fit its purpose. The number of input nodes need to be same as the dimension of the data that should be classified. Furthermore, the number of hidden layers and the width (number of node) is important. More and wider hidden layers allow the network to represent more complicated functions. This does, however, increase training and evaluation time, since there are more parameters to train. Also, this means that the network improves the fitting of the data. This can lead to overfitting which means that the network can classify the training samples efficiently (including outliers) while not generalize to unseen data. Several methods such as dropout techniques have been studied to overcome this issue.
The output layer design is important and depends on the used loss function. The loss function is used during training to update network variables. One way to design the output layer is to have the same number of output nodes as the number of classes representing the input data. The expected output would be a one-hot vector.

### 2.1.2 Loss Function

Neural networks need to be trained using enough training samples to enable it to reach high accuracy and generalize to unseen data. The training set need to contain enough variance to be a good representation of the general case. When using faces, for example, two different individuals are not a good enough representation of the general population. Images of 200 individuals would, however, contain a larger variety of people, achieving a more general representation of the population.

During training, the data should be split into a training set used to train the network parameters, and test set, used to test the network accuracy. This is done to obtain an estimate of how it would perform in a practical application when the data is unknown. If the accuracy on the training set is much higher than on the test set it is a sign of an over fitted network. There are two major ways to split the data and they depend on the quantity of images per identity belonging to the data set and what type of loss is used.

Cross entropy [7] loss uses vector with one element for each class, where the largest element determines the classification. This means that the network need to be trained using images of all identities which it should be able to recognize. Also, it need to be retrained if an identity is added. At least one image per person is needed in both training and test set, however, to achieve good results it requires more images. The first way to split the data is to use a percentage of the images for each person for training and the rest for testing.

Triplet loss allows training of a generic network that can classify any person without the need to retrain the network. It makes the network learn a mapping from input to a generic feature space. When using triplet loss one can split the data set to contain images of all individuals in both training and test set, however, some data sets, such as LFW [8], which is the dataset used in the current thesis, sometimes has just one image per person. In this case, we need to use triplet loss and another way to split the data.

The triplet loss idea was introduced by Google in “FaceNet: A Unified Embedding for Face Recognition and Clustering” [2] and can be used in networks that map data to a point in a d-dimensional space. It is based on measuring the distance between these points and obtains its name from using triplets of points, one anchor point \( x^a \), one positive point \( x^p \), and one negative point \( x^n \). The anchor and positive point belong to the same class while the negative point should belong to a different class. The total loss tries to minimize
the distance between the anchor and the positive point while maximizing the distance between the anchor and the negative point. It is defined by equation (2.4) where $\beta$ is a margin that is enforced between positive and negative pairs. Figure 2-3 illustrates training with triplet loss. There are three points an anchor, a positive and a negative. Triplet loss aims to minimize the L2-distance between the anchor and the positive point while maximizing the distance between the anchor and the negative point.

$$L = \sum_{i=1}^{N} \max \{0, \|x_i^a - x_i^p\|_2^2 - \|x_i^a - x_i^n\|_2^2 + \beta\} \quad (2.4)$$

![Figure 2-3 Illustration of training with triplet loss](image)

The choice of triplets is very important for the convergence of the model. As shown is

### 2.2 Convolutional Neural Networks

Among the Deep Neural Networks, the ones that are most widely used in Computer Vision problems are the Convolutional Neural Networks, based in the Multi-Layer Perceptron architecture. Regular neural networks use vectors as input and have fully connected layers, which means that each element is connected to all nodes in the next layer. Images, then, need to be reshaped to vectors and the number of weights, between the input layer and one node in the hidden layer, is the same as the number of pixels. Larger images would, therefore, result in an unreasonable amount of weights. Convolutional neural networks (CNN’s) use images as input and convolution kernels, made up out of weights, as connections to the next layer. This means that weights are shared on different spatial positions and the needed amount of weights are lowered. These weights are arranged in 3D volumes and transforms the input image into an output of node activations.

The stacked neuron layers are organized, so their outputs form 3D volumes. The input of the first layer, that is the image itself, can also be considered such a 3D volume: width x height x channels. Each stack of layers use a different kernel
configuration, and all the layers in it are connected to all layers in previous stack. The main parameters to take under consideration in convolutional technology are:

**Kernel size.** which is the width and height of the receptive field for the neurons in that stack. Usually, both sides are of equal size, and then it’s called square kernel.

**Stride,** indicates the distance between the centres of each region’s input space. As such, a stride of 1 means that each neuron processes the same region as their neighbour except for one column. The larger the stride the smaller the output width and height of that layer.

Furthermore, in some cases, the neurons in the borders of the layer cannot process a whole receptive field. This may happen due to the stride. For example, in Figure 2-4 we have an example of a 4x4 image being processed by a 3x3 kernel with a stride of 2. As the difference between receptive fields is 2 pixels, the last column of the receptive field of the second neuron “falls” outside the image, and is not processed. In order to address this issue, one possibility is to add a border around of image of 0’s. This way, we guarantee that all neurons process a receptive field. The padding usually is 1 or 2.

![Figure 2-4 Example of a case justifying padding](image)

Finally, using convolutional layers, the networks are able to learn to recognize local features, such as edges or corners, by restricting the receptive field of hidden units to local connectivity and to add shift invariance by enforcing spatially shared weights. Furthermore, spatial or temporal subsampling in the form of pooling layers reduces sensitivity to shifts and distortion. Convolutional layers are going to be described thoroughly in the next subchapter.

### 2.2.1 Layers Types

In this section are introduced the types of used layer, convolutional, pooling and fully connected layer.

**Convolutional layers,** have weights arranged in kernels that are convolved with input. Each pixel in the 3D convolution output is mapped through an activation
function (such as ReLU) to produce a 3D output volume of node activations. The use of convolutional layers keep the size which is accomplished by the use of zero padding. It adds zeros around the image before the convolution, and then keeps only the central part of the result with the same size as the input.

**Max-pooling or average-pooling layers**, are includes between successive stacks of convolutional ones, in order to progressively reduce the size of the image representation. It works by taking each channel of its receptive field, and resizes it by keeping only the maximum of its values. It is usually used with 2x2 kernels, and a stride of 2, which has each side size. This reduces the overall size in 75% by picking the largest of 2x2 patches. This kind of layers does not have weights that need training, and it only uses the stride and kernel size parameters. Its utility consists in reducing the amount of weights to learn, which, also, reduces computational time as well as probability of overfitting.

**Fully connected Layers**, are basically neural layers connected to all neuron from previous layers. In this case, they do not use any of the introduced parameters, using instead the number of neurons. The output the produce could be understood as a compact feature vector representing the input image. Such layers are usually placed at the end of a CNN.

### 2.3 Deep Convolutional Networks

In Facenet network is trained the CNN using Stochastic Gradient Descent (SGD) with standard backpropagation and AdaGrad [9]. In most experiments start with a learning rate 0.05 which decreasing to finalize the model. The modes are initialized from random, similar to [10], and trained on a CPU cluster for 1,000 to 2,000 hours. The detection in the loss (and increase in accuracy) slows down drastically after 500h training, but additional training can still significantly improve performance.

The category of model used in based on GoogLeNet style Inception models [10]. These models have parameters sized from 6.6M to 7.5M and FLOPS between 500M and 1.6B. These models are dramatically reducing in size (both depth and number of filters), so that they can be run on a mobile phone

**Inception Module**

Inception modules were introduced by Google in “Going Deeper with Convolutions” [11] and are used to increase the network size without uncontrolled increase in computational complexity. They are combinations of convolutional and max pooling layers and are based on the idea that images should be processed at several different scales. The module sends the input from the previous layer on four different paths and the combination of the paths outputs is the module output. The first is a 1x1 convolutional layer, the second is a 1x1 followed by a 3x3 convolution layer, the third is a 1x1 followed by a
5x5 convolution layer and the last path is a 3x3 max pooling layer followed by a 1x1 convolution layer.

![Inception module, naive version (left) / Inception module with dimension reduction (right)](image)

The created inception model of Facenet is shown in Figure 2-6.

<table>
<thead>
<tr>
<th>type</th>
<th>output size</th>
<th>depth</th>
<th>#1x1</th>
<th>#3x3 reduce</th>
<th>#3x3</th>
<th>#5x5 reduce</th>
<th>#5x5</th>
<th>pool proj (p)</th>
<th>params</th>
<th>FLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1 (7x7x3, 2)</td>
<td>112x112x64</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9K</td>
<td>119M</td>
</tr>
<tr>
<td>max pool + norm</td>
<td>56x56x64</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m 3x3, 2</td>
<td></td>
</tr>
<tr>
<td>inception (2)</td>
<td>56x56x192</td>
<td>2</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td>16</td>
<td>32</td>
<td>m 32p</td>
<td>115K</td>
<td>360M</td>
</tr>
<tr>
<td>norm + max pool</td>
<td>28x28x192</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m 3x3, 2</td>
<td></td>
</tr>
<tr>
<td>inception (3a)</td>
<td>28x28x256</td>
<td>2</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td>16</td>
<td>32</td>
<td>L2, 64p</td>
<td>228K</td>
<td>179M</td>
</tr>
<tr>
<td>inception (3b)</td>
<td>28x28x320</td>
<td>2</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td>32</td>
<td>64</td>
<td>L2, 128p</td>
<td>545K</td>
<td>107M</td>
</tr>
<tr>
<td>inception (3c)</td>
<td>14x14x640</td>
<td>2</td>
<td>0</td>
<td>128</td>
<td>256</td>
<td>32</td>
<td>64</td>
<td>L2, 128p</td>
<td>595K</td>
<td>117M</td>
</tr>
<tr>
<td>inception (4a)</td>
<td>14x14x640</td>
<td>2</td>
<td>256</td>
<td>96</td>
<td>128</td>
<td>32</td>
<td>64</td>
<td>L2, 128p</td>
<td>654K</td>
<td>128M</td>
</tr>
<tr>
<td>inception (4b)</td>
<td>14x14x640</td>
<td>2</td>
<td>224</td>
<td>112</td>
<td>224</td>
<td>32</td>
<td>64</td>
<td>L2, 128p</td>
<td>722K</td>
<td>142M</td>
</tr>
<tr>
<td>inception (4c)</td>
<td>14x14x640</td>
<td>2</td>
<td>192</td>
<td>128</td>
<td>256</td>
<td>32</td>
<td>64</td>
<td>L2, 128p</td>
<td>717K</td>
<td>56M</td>
</tr>
<tr>
<td>inception (4d)</td>
<td>14x14x640</td>
<td>2</td>
<td>160</td>
<td>144</td>
<td>288</td>
<td>32</td>
<td>64</td>
<td>L2, 128p</td>
<td>1.6M</td>
<td>78M</td>
</tr>
<tr>
<td>inception (4e)</td>
<td>7x7x1024</td>
<td>2</td>
<td>0</td>
<td>160</td>
<td>256</td>
<td>64</td>
<td>128</td>
<td>m 3x3, 2</td>
<td>717K</td>
<td>56M</td>
</tr>
<tr>
<td>inception (5a)</td>
<td>7x7x1024</td>
<td>2</td>
<td>384</td>
<td>192</td>
<td>384</td>
<td>48</td>
<td>128</td>
<td>L2, 128p</td>
<td>1.6M</td>
<td>78M</td>
</tr>
<tr>
<td>inception (5b)</td>
<td>7x7x1024</td>
<td>2</td>
<td>384</td>
<td>192</td>
<td>384</td>
<td>48</td>
<td>128</td>
<td>m 128p</td>
<td>1.6M</td>
<td>78M</td>
</tr>
<tr>
<td>avg pool</td>
<td>1x1x1024</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fully conn</td>
<td>1x1x128</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>131K</td>
<td>0.1M</td>
</tr>
<tr>
<td>L2 normalization</td>
<td>1x1x128</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7.5M</td>
<td>1.6B</td>
</tr>
</tbody>
</table>

*Figure 2-6 The Facenet network architecture*

This model in the case of our implementation is used as it trained from Facenet researchers and trained on a combination of FaceScrub and CASIA-WebFace Datasets [12]. The CASIA-WebFace dataset contains 494,414 images [13]. This training set consists of total of 453 453 images over 10 575 identities after face detection. Some performance improvement has been seen if the dataset has been filtered before training. Some more information about how this was done
will come later. The best performing model has been trained on a subset of the MS-Celeb-1M [14] dataset. This dataset is significantly larger but also contains significantly more label noise, and therefore it is crucial to apply dataset filtering on this dataset.

### 2.4 Multi-task Cascaded Convolutional Networks [15]

Multi-task Cascaded Convolutional Networks used for the face detection and alignment. These tasks are essential to face applications, such as face recognition. However, the large visual variations of faces, such as occlusions, large pose variations and extreme lightings, impose great challenges for these tasks in real world applications.

#### 2.4.1 Approach

In this chapter is described the approach towards joint face detection and alignment.

The pipeline of the approach is shown in Figure 2-7. Given an image, is initialised a resize to it to different scales to build an image pyramid, which is

![Image of pipeline](image_url)

*Figure 2-7 Pipeline of cascaded framework which includes three-stage multi-task deep CNN*
the input of the following three stage cascaded framework:

**Stage 1:** We exploit a fully convolutional network, called Proposal Network (P-Net), to obtain the candidate facial windows and their bounding box regression vectors. Then candidates are calibrated based on the estimated bounding box regression vectors. After that, we employ non-maximum suppression (NMS) to merge highly overlapped candidates.

**Stage 2:** All candidates are fed to another CNN, called Refine Network (R-Net), which further rejects a large number of false candidates, performs calibration with bounding box regression, and conducts NMS.

**Stage 3:** This stage is similar to the second stage, but in this stage, we aim to identify face regions with more supervision. In particular, the network will output five facial landmarks’ positions.

In Figure 2-7, for example, first the candidate windows are produced through a fast Proposal Network (P-Net). After that, we refine these candidates in the next stage through a Refinement Network (R-Net). In the third stage, The Output Network (O-Net) produces final bounding box and facial landmarks position.
| Theoretical Basis | Multi-task Cascaded Convolutional Networks [15] |
3 Big Data Frameworks

3.1 Hadoop

Apache Hadoop is an open source software project. It is designed as a big data solution. In this chapter is introduced the Hadoop Distributed File System (HDFS) and YARN which are two main technologies of Hadoop, based on its official documents [5].

3.1.1 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is a scalable file system for large distributed data-intensive applications. Unlike other distributed file systems, HDFS is designed to be built from low-cost commodity component which required it to be highly fault-tolerant.

A HDFS cluster consists of a master server called NameNode and several slave servers called DataNodes. The architecture of HDFS is illustrated in Figure 3-1. The NameNode manages the metadata including file names, locations, replications and the client’s access to files. The DataNode manages the storage. Data in Hadoop cluster is split into smaller pieces called blocks and are stored in DataNodes through the cluster. The block size can be set by users for optimization. These blocks are automatically replicated for fault tolerance. The replication factor is configurable and the default values is set to 3. The first two replications are put on the same reach but different DataNodes, and the third replication is put on different rack. The placement of replication is key to HDFS reliability and performance.
When a client wants to read from HDFS or write to it, the first connection is with NameNode to get the locations of files and the access permit. Then it contacts directly to these DataNodes. It is necessary to mention that data never flow directly through NameNodes, only the information stored there. The NameNode is also responsible for detecting the condition of DataNodes. Each DataNode sends messages to the NameNode periodically (also called heartbeats). NameNode marks a DataNode as dead if it cannot detect heartbeat from that node and will stop sending I/O requests to that node. Data stored on dead nodes in not available to access anymore and will be re-replicated by the NameNode.

3.1.2 MapReduce 1.0

MapReduce subchapter is an intermediate step in order to improve the understanding of YARN. MapReduce was first developed by Google. It is a massively scalable, parallel processing programming model and software framework for generating large data sets. MapReduce is the processing part of Hadoop. Based on the technology of MapReduce and HDFS, processing can be executed at the location of data which reduces the cost of data transferring. The term MapReduce stands for two functions: Map and Reduce. Both of Map tasks and Reduce tasks work on key-value pairs. The main idea is to map the input data into key-value pairs and group together values with the same keys then the reduce function merges together these values with the same keys.
Figure 3.2 MapReduce 1.0

Figure 3-2 describes the basic architecture of MapReduce. It was drawn based on the paper “MapReduce: Simplified Data Processing on Large Cluster” [16]. Just like the HDFS system, MapReduce cluster consists of a master server called JobTracker and a number of slave servers called TaskTrackers. The JobTracker is responsible for assigning tasks and the TaskTrackers is responsible for computing. When a job comes “into” MapReduce, it is split into small pieces of tasks, and then, JobTracker assigns these tasks to each TaskTracker.

There are three steps, map, shuffle and reduce. The first step, map, is where data is computing. The JobTracker will always try to pick nodes with local data for a Map task to reduce data transmission. If nodes with local data already have enough tasks running, the JobTracker will assign the task to a node in the same rack. The second step, shuffle, is when the result of Map process are sorted in this step and assign to TaskTrackers which are running the reduce task. The intermediate data is combined here and distilled to have the final output.

3.1.3 YARN

YARN (Yet Another Resource Negotiator) or MapReduce 2.0 (MRv2) is the cluster resource manager. It is a key feature in Hadoop 2 version that significantly improves the scalability of the cluster and supports MapReduce and non-MapReduce applications, such as Spark, MPI, Giraph and Hamr, running on the same cluster. The core idea of YARN is to split the two functions of JobTracker into separate processing engine – the ResourceManager and the ApplicationMaster.

In Figure 3-3 is illustrated the architecture of YARN. There is a global ResourceManager for entire cluster deployed on the master node which assigns CPU, memory and storage applications running on the cluster. It, also, tracks heartbeats from NodeManagers and ApplicationMasters. The NodeManager is
deployed on each slave node which monitors the resource usage including CPU, memory, disk, network and communicates with the ResourceManager. The ApplicationMaster launches every time a job submitted to the cluster. It negotiates resources from ResourceManager and works with the NodeManager to execute the task. The container is created by ResourceManager upon request. A certain amount of resources is allocated on slave nodes for executing applications and will be de-allocated when the application is completed.

![Figure 3-3 YARN. The cluster is composed by one master and 4 slave nodes](image)

The process of running job illustrated in Figure 3-3 is a MapReduce job. Application B is submitted by the green client and has run on the cluster. The blue client just submitted the Application A to the ResourceManager. The ResourceManager launched an ApplicationMaster for the job on the second NodeManager. Then, the ApplicationMaster requested resources from the ResourceManager. Two containers were launched by the ResourceManager on two different slave nodes. The container IDs would be sent back to the ApplicationMaster. Finally, the Application A began to execute. Map tasks and reduce tasks were run in containers. The progress was updated to the ApplicationMaster.
Figure 3-4 Differences between the Components of Hadoop 1 and 2.

Figure 3-4 is comparing the architecture of Hadoop 1 and Hadoop 2. Since most functions of JobTracker are moved to ApplicationMasters running on the slave nodes, YARN significantly reduces the responsibility of master node and improves the scalability of the cluster. According to the paper “Apache Hadoop YARN: Yet Another Resource Negotiator” [17], the limitation of Hadoop cluster is increased from 4000 nodes to 7000.

3.2 Spark

Spark is an open source cluster computing framework for solving big data problems. It was first developed in the AMPLab at UC Berkeley in 2009. Now it has become one of the most widely used programs to solve big data problems. Spark is designed to support applications which rescue a working set of data across multiple parallel operations while also providing the same scalability and fault tolerance properties as MapReduce [18]. These applications are mainly classified into two types: iterative jobs and interactive analysis.

According to its website, Spark runs programs up to 100x faster than Hadoop MapReduce if it is used in-memory computing only. It can run 10x faster if it is used with combination of memory and disks. RDD, or resilient distributed dataset, is a distributed shared memory system and support reuse of data and intermediate results in memory. Spark is also designed to be used easily. It provides APIs in Java, Scala, Python and R shells. It can run on YARN and accessing data from HDFS. In this chapter, is introduced Spark according its official document [6].

3.2.1 Spark Components

According to Spark documents, Figure 3-5 gives a short overview about how SparkContext and coordinates the applications. The SparkContext need to
connect to a cluster manager to get memory and processing resource to execute the programs. Spark can run on three cluster managers, standalone mode, Mesos mode and YARN mode.

![Spark Components Diagram]

**Figure 3-5 Spark Components**

Standalone mode, is provided by Spark distribution. The cluster can be launched manually or by launch scripts. It can be simply used on a single node for testing. Mesos mode is a cluster manager provided by Apache.

YARN mode which is used in this thesis implementation. There are two deploy modes that can be used to launch Spark application on YARN. In cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application. In client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

The cluster manager allocates resources on executors. Different applications run on different executor processes which means they cannot share data in memory. The cluster manager, also, sends the executor information back to driver program, that connection is set up between driver and workers. Tasks are sent to executors by SparkContext.

### 3.2.2 RDD

A resilient distributed dataset is a read-only and fault-tolerant collection of objects that support parallel processing [6]. To achieve the fault-tolerant goal Spark use the lineage. That means if a partition is lost, Spark can rebuild that partition by the lineage information instead of going back to the checkpoint. That, also, helps the system to save the cluster network resource by avoiding replicating data for checkpoint. As discussed in paper “Spark: Cluster Computing with Working Sets” [18], RDDs can only be created in four ways:
- From a file in HDFS or other shared file systems.
- By parallelizing a Scala collection in the driver program.
- By transforming an existing RDD form one type to another type using the flatMap operation.
- By changing the persistence of an existing RDD.

When a job is submitted to Spark, workers read data from HDFS or other distributed file systems and cache it in memory. Data can be reused for each iteration. RDD helps to reduce reading from and writing to disk which contributes to speed up the processing. Finally, RDD is read-only which mean that any changes to a RDD partition will cause a creation of new partition [19].

### 3.2.3 Spark Streaming

Streaming programming has been widely used in High Performance Computing community. Spark support both batch and streaming data processing. Overview, Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Apache Kafka [20], Flume [21], Kinesis [22], text files or TCP sockets and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to file systems, databases, and live dashboards. In fact, you can apply Spark’s machine learning and graph processing algorithms on data streams.

Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

Discretized Stream or DStream is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream. Internally, a DStream is represented by a continuous series of RDDs, which is Spark’s abstraction of an immutable, distributed dataset (see Spark Programming Guide for more details). Each RDD in a DStream contains data from a certain interval, as shown in Figure 3-6.
Any operation applied on a DStream translates to operations on the underlying RDDs. For example, in the earlier example of converting a stream of lines to words, the flatMap operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream. This is shown in Figure 3-7.

These underlying RDD transformations are computed by the Spark engine. The DStream operations hide most of these details and provide the developer with a higher-level API for convenience.

**Text File Stream**

In the current thesis, we will use the text file streaming for reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.), a DStream can be created by the command `streamingContext.textFileStream(dataDirectory)`. Spark Streaming will monitor the directory `dataDirectory` and process any files created in that directory (files written in nested directories not supported). Note that

- The files must have the same data format.
- The files must be created in the `dataDirectory` by atomically *moving* or *renaming* them into the data directory.
- Once moved, the files must not be changed. So, if the files are being continuously appended, the new data will not be read.
3.3 Apache Spark and Hadoop: Working Together

Spark is intended to enhance, not replace, the Hadoop stack. From day one, Spark was designed to read and write data from and to HDFS, as well as other storage systems, such as HBase and Amazon’s S3. As such, Hadoop users can enrich their processing capabilities by combining Spark with Hadoop MapReduce, HBase, and other big data frameworks.

Also, Spark was constantly focused on making it as easy as possible for every Hadoop user to take advantage of Spark’s capabilities. No matter whether you run Hadoop 1.x or Hadoop 2.0 (YARN), and no matter whether you have administrative privileges to configure the Hadoop cluster or not, there is a way to run Spark. In particular, there are three ways to deploy Spark in a Hadoop cluster: standalone, YARN, and SIMR, as show on Figure 3-8.

![Spark over Hadoop deployments](image)

**Figure 3-8 Spark over Hadoop deployments**

**Standalone deployment:** With the standalone deployment one can statically allocate resources on all or a subset of machines in a Hadoop cluster and run Spark side by side with Hadoop MR. The user can then run arbitrary Spark jobs on her HDFS data. Its simplicity makes this the deployment of choice for many Hadoop 1.x users.

**Hadoop Yarn deployment** [23]: Hadoop users who have already deployed or are planning to deploy Hadoop Yarn can simply run Spark on YARN without any pre-installation or administrative access required. This allows users to easily integrate Spark in their Hadoop stack and take advantage of the full power of Spark, as well as of other components running on top of Spark.

**Spark in MapReduce (SIMR)** [24]: For the Hadoop users that are not running YARN yet, another option, in addition to the standalone deployment, is to use SIMR to launch Spark jobs inside MapReduce. With SIMR, users can start
experimenting with Spark and use its shell within a couple of minutes after downloading it! This tremendously lowers the barrier of deployment, and lets virtually everyone play with Spark.
4 Tensorflow

Tensorflow is an open-source software library for machine learning across a range of tasks. It is a system for building and training neural networks to detect and decipher patterns and correlations, analogous to, but not the same as, human learning and reasoning. It is used for both research and production at Google. It was developed by the Google Brain [25] team for internal Google use, and was released under the Apache 2.0 open source licence.

Google’s TensorFlow has been a hot topic in deep learning recently. The open source software, designed to allow efficient computation of data flow graphs, is especially suited to deep learning tasks. It is designed to be executed on single or multiple CPUs and GPUs, making it a good option for complex deep learning tasks. In its most recent incarnation – version 1.0 – it can even be run on certain mobile operating systems. This introductory tutorial to TensorFlow will give an overview of some of the basic concepts of TensorFlow in Python. These will be a good stepping stone to building more complex deep learning networks, such as Convolutional Neural Networks, natural language models and Recurrent Neural Networks in the package.

Tensorflow converts the neural network matrices to Tensors. For example, in Figure 4-1 captures a neural network with 2 layers. One step in “running” the neural network is to multiply the value of each weight by the value of its input unit, and then to store the result in the associated hidden unit.

We can redraw the units and weights as arrays, or what are called lists in Python. From a math standpoint, they’re matrices. We’ve redrawn only a portion of them in the diagram. Multiplying the input matrix with the weight matrix involves simple matrix multiplication resulting in the five-element hidden matrix/list/array.
In TensorFlow those lists are called tensors. And the matrix multiplication step is called an operation, or op in programmer-speak, a term you’ll have to get used to if you plan on reading the TensorFlow documentation. Taking it further, the whole neural network is a collection of tensors and the ops that operate on them. Altogether they make up a graph like this in Figure 4-2.
Figure 4-2 is a snapshot taken from Tensorboard, a tool for visualizing the graph [26] as well as examining tensor values during and after training. The tensors are the lines, and written on the lines are the tensor’s dimensions. Connecting the tensors are all the ops, though some of the things you see can be double-clicked on in order to expand for more detail, as we’ve done for layer1 in the second snapshot.

At the very bottom is x, the name we’ve given for a placeholder op that allows us to provide values for the input tensor. The line going up and to the left from it is the input tensor. Continue following that line up and you’ll find the MatMul op, which does the matrix multiplication with that input tensor and the tensor which is the other line leading into the MatMul op. That tensor represents the weights.
4.1 TensorFlowOnSpark

TensorflowOnSpark is an open source framework for distributed deep learning on big-data clusters developed by Yahoo. It’s a fact that in order to gain insight from massive amount of data, is needed to deploy distributed deep learning (DL). Existing DL frameworks often require us to set up separate clusters for deep learning, forcing us to create multiple programs for a machine learning pipeline (see Figure 4-3). Having separate clusters requires us to transfer large datasets between them, introducing unwanted system complexity and end-to-end learning latency.

![Figure 4-3 ML Pipeline with multiple programs on separated clusters](image)

After TensorFlow’s initial publication, Google released an enhanced TensorFlow with distributed deep learning capabilities in April 2016. In October 2016, TensorFlow introduced HDFS support. Outside of the Google cloud, however, users still needed a dedicated cluster for TensorFlow applications. TensorFlow programs could not be deployed on existing big-data clusters, thus increasing the cost and latency for those who wanted to take advantage of this technology at scale.

![Figure 4-4 TensorFlowOnSpark for deep learning on Spark Cluster](image)
TensorFlowOnSpark (TFoS) enables distributed TensorFlow execution on Spark and Hadoop clusters. As illustrated in Figure 4-4 above, TensorFlowOnSpark is designed to work along with SparkSQL, MLlib, and other Spark libraries in a single pipeline or program (e.g. Python notebook).

TensorFlowOnSpark supports all types of TensorFlow programs, enabling both asynchronous and synchronous training and inferencing. It supports model parallelism and data parallelism, as well as TensorFlow tools such as TensorBoard on Spark clusters.

Any TensorFlow program can be easily modified to work with TensorFlowOnSpark. Typically, changing fewer than 10 lines of Python code are needed. Many developers at Yahoo who use TensorFlow have easily migrated TensorFlow programs for execution with TensorFlowOnSpark.

TensorFlowOnSpark supports direct tensor communication among TensorFlow processes (workers and parameter servers). Process-to-process direct communication enables TensorFlowOnSpark programs to scale easily by adding machines. As illustrated in Figure 4-5, TensorFlowOnSpark doesn’t involve Spark drivers in tensor communication, and thus achieves similar scalability as stand-alone TensorFlow clusters.

![Figure 4-5 TensorFlowOnSpark system Architecture](image)

TensorFlowOnSpark provides two different modes to ingest data for training and inference:

- **TensorFlow QueueRunners**: TensorFlowOnSpark leverages TensorFlow’s file_readers and QueueRunners to read data directly from HDFS files. Spark is not involved in accessing data.
• **Spark Feeding**: Spark RDD data is fed to each Spark executor, which subsequently feeds the data into the TensorFlow graph via feed_dict.

### 4.1.1 Command Line Interface

TFoS programs are launched by the standard Apache Spark command, `spark-submit`. As illustrated below, users can specify the number of Spark executors, the number of GPUs per executor, and the number of parameter servers in the Command Line Interface. A user can also state whether they want to use TensorFlow (`` -- tensorboard``) and/or RDMA (`` -- rdma``).

TFoS programs are launched by the standard Apache Spark command, `spark-submit`. As illustrated below, users can specify the number of Spark executors, the number of GPUs per executor, and the number of parameter servers in the CLI. A user can also state whether they want to use TensorFlow (``` -- tensorboard```) and/or RDMA (``` -- rdma``).

```
spark-submit --master $[MASTER] \\
$[TFoS_HOME]/examples/slim/train_image_classifier.py \\
  --model_name inception_v3 \\
  --train_dir hdfs://default/slim_train \\
  --dataset_dir hdfs://default/data/imagenet \\
  --dataset_name imagenet \\
  --dataset_split_name train \\
  --cluster_size $[NUM_EXEC] \\
  --num_gpus $[NUM_GPU] \\
  --num_ps_tasks $[NUM_PS] \\
  --sync_replicas \\
  --replicas_to_aggregate $[NUM_WORKERS] \\
  --tensorboard \\
  --rdma
```

### 4.1.2 Application Programming Interface (API)

TFoS, finally, provides a high-level Python API (illustrated in sample Python notebook [27])

- `TFCluster.reserve()`, construct a TensorFlow cluster from Spark executors
- `TFCluster.start()`, launch TensorFlow program on the executors
- `TFCluster.train()` or `TFCluster.inference()`, feed RDD data to TensorFlow processes
- `TFCluster.shutdown()`, shutdown TensorFlow execution on executors
5 Technical Specifications

This chapter is a technical report of the cluster implementation and architecture.

5.1 Infrastructure

In order to implement the face recognition platform, was used the Amazon Elastic Computer Cloud infrastructure (Amazon EC2) [28]. It is a web service that provides secure resizable computer capacity in the cloud. It is designed to make web-scale cloud computing easier for developers. Amazon EC2’s simple web service interface allows obtaining and configuring capacity with minimal friction. It provides complete control of computing resources and leaves developers to run on Amazon’s proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. Finally, it provides developers the tools to build failure resilient applications and isolate them from common failure scenarios.

Amazon provides a wide selection of instance types optimized to fit different use cases. Instance types comprise varying combinations of CPU, memory, storage and networking capacity and gives the flexibility to choose the appropriate mix of resources for any application [29]. In the case of this thesis M4 instances were used, they are the latest generation of General Purpose Instances. This family provides a balance of compute, memory and network resources and it is a good choice for many application. Some of the features of these instances are:

- 2.3 GHz Intel Xeon® E5-2686 v4 (Broadwell) processors or 2.4 GHz Intel Xeon® E5-2676 v3 (Haswell) processors
- EBS-optimized by default at no additional cost
- Support for Enhanced Networking
- Balance of compute, memory, and network resources

The model used is m4.xlarge which has 4 vCPU and 16 GiB memory.

This chapter contains a detailed description of the steps followed for the cluster creation and the architecture of it.
5.2 Cluster Architecture

5.2.1 Terraform

For the creation of the cluster Terraform [30] was used. Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. It can manage existing and popular service providers as well as custom in-house solutions.

Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

In the case of this thesis Terraform was used upon AWS for deploying a 3-instances cluster.

Terraform commands (CLI) [31]

Terraform is controlled via a very easy to use command-line interface (CLI). It is only a single command-line application: terraform. This application then takes a subcommand such as "apply" or "plan". The complete list of subcommands is in the navigation to the left.

The terraform CLI is a well-behaved command line application. In erroneous cases, a non-zero exit status will be returned. It also responds to -h and --help as you'd most likely expect.

The basic Terraform commands which we will mainly use in this thesis are plan, apply and destroy.

- Terraform plan is used to create an execution plan. Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.
- Terraform apply is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a terraform plan execution plan.
- Terraform destroy is used to destroy the Terraform-managed infrastructure.

Terraform Installation

The first step was to download the latest Terraform version (0.10.5) from its official site [32] locally. It doesn’t require any installation we just need to set it
to the PATH variable so that it is accessible from our system in any path with the following “export” command.

After executing the above steps, we can confirm Terraform's installation in our system by running terraform command. We should see the following output.

5.2.2 Amazon Web Credentials & key-pair Setup

After creating our amazon account, we create an access key ID and a secret key. The secure way to store these credentials as recommended by Amazon [33] is storing them in a hidden folder under a directory called “credentials”. This file can be accessed by Terraform to retrieve them.

Therefore, we add the Access key and the secret key to the credentials file by replacing ACCESS_KEY and SECRET_KEY with the originals given from Amazon.

Finally, is necessary to restrict access to this file only to the current user, by typing the following command:

The next step is to create a key pair so that terraform can access the newly created VMS. Notice that this is different than the above credentials. The Amazon credentials are for accessing and allowing the AWS service to create the resources required, while this key pair will be used for accessing the new instances.

The first step is to log into AWS console and select “Create Key Pair”, add a name and click “Create”. After this, AWS will create a .pem downloadable file. Move this file to the .aws directory and restrict the permissions as shown below.

Now this key pair is ready to be used either via a direct SSH to our instances, or for terraform to use this to connect to the instances and run some scripts.

5.2.3 Instances Set up

The basic cluster instances were decided to be 3, one master and two slaves. Also, was created a fourth instance from which we will connect with VPN over SSH to ensure the security of the system. The auxiliary instance used to safe login is named “Desktop” and the cluster instances are “cm1” the master,”cs1” and “cs2” the slaves. This subchapter is about the Terraform scripts created to generate these above instances.

Desktop Instance

The terraform script for the creation of Desktop instance is the following. The script sets

- As input directory of the credentials the .aws folder created locally
- the IP address where the Desktop is registered (10.168.0.25)
- the subnet mask of the instance
- The type of the EC2 instance, since, Desktop is just a pipeline to the cluster there is no need for high performance.

After the execution, we can see that a new instance named “Desktop” has already created in Amazon site. To this instance, we assign an Elastic IP, in our case this IP is 34.212.52.21. After using pem key to connect to the new instance we should take our public key and add it to the .ssh/authorized_keys file of Desktop instance. After this configuration, it is possible to SSH the instance without pem key.

Before creating the cluster instances, we should use the created Desktop instance to connect via VPN over SSH. To achieve this, we used the SSHuttle tool [34].

Cluster Instances

In the following Terraform script are set the information for the creation of the cluster instances cluster master 1 (cm1) and cluster slaves 1 and 2 (cs1, cs2). In this code are set

- the IP address of each instance (cm1:10.168.1.11, cs1: 10.168.1.12, cs2: 10.168.1.13)
- the type of EC2 instances
- the networking settings ingress and egress IP addresses which allow to ssh the machines (port 22) and to access all Desktop instance addresses.
- Finally, in this terraform script is called the install-hadoop.sh script which is executed in instances after their creation.

This script is focused on configuring the installation of Hadoop and Spark on instances, the main processes executed in this script are:

- Installation of Python 2.7 [35]
- Configuration of IP addresses of the cluster instances in /etc/hosts file
- Installation of JDK for building applications using the Java programming language [36]
- Set up of Hadoop 2.7.3 and its configuration files
- Set up of Spark 2.2.0 and its configuration files

After saving these two scripts in the same file and by ensuring that Terraform is active as show in 5.2.1 we can run command “terraform plan” to create an execution plan and then “terraform apply” to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a terraform plan execution plan. The final view of the Amazon account looks like Figure 5-1.
Set root access to instances

Amazon allows to first connect only to Ubuntu user, by which we created the 4 instances. Although, since we want to have full access to the instances so we will moderate the ssh configuration file to allow root login and then, as root user we will set the exported files to the .bashrc file of root user so it will be able to use Java, Hadoop and Spark.

Connection to Instances
For the remote login to instances is used the Secure Shell (SSH) [37].
6 Methodology

6.1 Dataset

Created by Huang et al. in 2007, Labelled Faces in the Wild (LFW) is a database of face images commonly used for reporting the performance of face recognition algorithms. It contains 13233 RGB images of 5749 different individuals (an average of ~2.3 images per person). The image dimensions are 250x250. The creators of the database also defined pairs of images and divided them into sets meant for training face recognition algorithms and reporting their performance [38]. While all of the networks compared in this paper were trained on different, larger databases, the LFW was still used for performance reporting; its wide use makes it well suited for comparing different techniques.

6.2 Methodology

For the Facenet implementation in Tensorflow was used GitHub code from [39], as described earlier on the report. This Tensorflow code was converted to TensorFlowOnSpark, as a first step of the analysis. Thereinafter, I changed the Facenet implementation for multiple face detection, this code is advertised in the analogue point of appendices. Therefore, a main change made on the existing code is the use of Hadoop File System while trying to keep the already in use functions.

The methods that briefly described above are described thoroughly bellow.

6.2.1 Conversion Guide from TensorFlow to TensorFlowOnSpark

The steps followed for the conversion are described here.

- Add PySpark and TensorFlowOnSpark import, the files containing a main() function and a call to tf.app.run().
- Replace the main() function with main_fun(argv, ctx), the argv parameter will contain a full copy of the arguments supplied at the PySpark command line, while ctx parameter will contain node metadata, like job_name and task_id. Also, it is important that the “import tensorflow as tf” occurs within this function, since this will be executed /imported on the executors. And, if there are any functions used by the main function, ensure that they are defined or imported inside the main_fun block, as show bellow.
- Replace the tf.app.run() method to launch TensorFlowOnSpark cluster. tf.app.run() executes the TensorFlow main function. Replace it with the
following code to set up PySpark and launch TensorFlow on the executors. Note that we're using argpashere mostly because the tf.app.FLAGS mechanism is currently not an officially supported TensorFlow API.

- Replace the tf.train.Server() with TFNode.start_cluster_server(). In distributed TensorFlow application, there is typically code that, extracts the addresses for the ps and worker nodes from the command line arguments, creates a cluster spec and starts the TensorFlow server.
- Add TensorFlowOnSpark-specific arguments. Since most TensorFlow examples use the tf.app.FLAGS mechanism, we leverage it here to parse our TensorFlowOnSpark-specific arguments (on the executor-side) for consistency. If your application uses another parsing mechanism, just add two arguments, one for the number of used GPUs and whether to use or not RDMA between GPUs. Towards this thesis, these arguments didn’t add because we used only CPU mode.

The changes described above are shown in Appendices changed FaceNet code.

6.2.2 Code Pre-processing

The existing Facenet code uses SciPy’s function imread [40]. This function uses the Python Imaging Library (PIL) to read an image. Because, imread couldn’t be used in Hadoop File System was created a process for local pre-processing with imread function. This processing is described with the following Figure 6-1. The main idea is that every time that we want to either align the images, train or classify the images each instance undertakes to process part of the images. Since, an instance undertakes the images these are removed from the main images directory to a new on process directory so that the other instance which processes the same data directory does not use the same data. After the local processing(s) all the processed data are returned to the HDFS.

![Figure 6-1 Processing Schema](image-url)
6.3 Streaming process

For the streaming process was used a text File Spark Streaming which is “listening” to a specific HDFS directory and every time new picture is saved in this directory the face recognition, alignment and identification process is initialized. The streaming procedure is either triggering a training process or a classification one.

6.4 Train a Classifier on LFW Dataset

For the experiment, we trained a classifier using a subset of the LFW images. The LFW dataset is split into a training and a test set. Then a pre-trained model is loaded, and this model is then used to generate features for the selected images. The pre-trained model is typically trained on a much larger dataset in order to give decent performance (in this case a subset of the MS-Celeb-1M dataset).

- Split the dataset into train and test sets
- Load a pre-trained model for feature extraction
- Calculate embeddings for images in the dataset
- **mode=TRAIN:**
  - Train the classifier using embeddings from the train part of a dataset
  - Save the trained classification model as a python pickle
- **mode=CLASSIFY:**
  - Load a classification model
  - Test the classifier using embeddings from the test part of a dataset
7 Experiment Results

7.1 Face Recognition Process

The experiments consist of three phases, the alignment of the dataset, the training and the classification of each individual. First, we align all the images of the dataset. The second phase, is the training which takes as input all the aligned dataset and is training with 30 pictures of each person and as final step the rest of pictures, of each person, are used for testing the created classifier.

Alignment

The alignment made for every picture of LFW Dataset. they are trimmed in new size 160 x 160 with margin 32. The total number of images are 13,233 and the final successful aligned images are 13,211. Finally, the execution time was 10,31 minutes.

An example of alignment is show in Figure 7-1.

![Alignment Example](image)

Figure 7-1 Example of Aligned photo

Training

The training phase splits the whole dataset into classes that consists of 40 pictures minimum of which the 30 used to train the classifier. The total found classes are 19 with 665 pictures. This means that we will use 19 individuals who have at least 40 images which will be used for the training procedure. The execution time of the training process was 2,46 minutes.

Classification
The classification phase, finally, used 19 classes consisted of 1200 images. These images are the rest of the trained images. The execution time was 3,40 minutes and the accuracy of the classifier is 97,3%.

### 7.2 Streaming process

For the streaming process the followed methodology was, by using Spark Streaming to listen to a specific HDFS directory. Every time a new snapshot is copied in this directory the classification procedure starts. This experiment aimed to simulation a real-time face recognition application. The implemented steps are the following:

- Copy to HDFS a picture of a person that was not used to train the classifier
- Spark streaming which already is listening to this HDFS directory identifies the entrance of the new image and initializes the face recognition procedure by executing a script which, first aligns the face in the image and then identifies the illustrated person

The mean total execution time for the alignment and the classification of one picture is 40 seconds. If we used a batch of 10 snapshots for the classification the mean executed time was 40,7 seconds which means that the most of the time is consumed to initialize the procedure and the read the embeddings model.

### 7.3 Non-distributed Classification

For the comparison of TensorFlow versus TensorFlowOnSpark the followed methodology was to implement Python experiments. After setting the PytonPath to the source code of FaceNet as taken from GitHub page [39] we execute a classification only to one of the machine, while trying to implement a non-distributed process.

First, the classifier created with the same specifications as these in Error! Reference source not found.. This created classifier was used for the testing phase where new images of the trained individuals should be recognized. The accuracy was again 97.3% -same with the distributed execution- but the execution time was 4 minutes, almost 20 seconds more than the more than the spark execution. So, even though, it was expected that the spark execution due to the local transfers, described in chapter 6.2.2 would be slower than the non-distributed execution, this didn’t happen because of the time spent for the load of the embeddings model.
8 Conclusion and Future Work

8.1 Conclusion

This MSc thesis aimed to highlight how a TensorFlow implementation can feet over a spark system and eventually come to a distributed new solution for machine learning problems. The machine learning problem which was approached was face recognition and the used code came from the Facenet which is implemented in TensorFlow. For the implementation over Spark used TensorFlowOnSpark. After face recognition was implemented upon Spark cluster there were also made experiments of this algorithm upon a standalone machine to eventually compare the two performances. As it was expected the two execution times were different with the distributed execution was shorter than the standalone as expected. Finally, by trying to approach a real-time face recognition a streaming process was created with Spark Streaming and indeed came out with a real-time solution.

Facenet algorithm used a function which was not compatible with Hadoop File System, so the code was transformed to work upon the local File System by transferring the data between the two systems. Even though it was expected that the transfer of data between Hadoop File System and local file system would cost more time to the distributed time, this didn’t happen. Finally, the created system was able for real-time Face Recognition with 99% accuracy on LFW dataset and the alignment and classification of every new incoming snapshot took 40 seconds. The important observation here was that if we use a batch of images for recognition then the execution time was 0.3 seconds more, this indicates that the majority of the time is the load of the model which extracts the face embedding and the transfer of the data between the two file systems.

Consequently, the implemented algorithms seem to work well in indoor environment relatively reliable with proper speed. Although the implementation is based on Python code, the system can run on real-time.

8.2 Future Work

The algorithms developed in this thesis can be used for any face classification problem, one only has to change the data set and retrain the networks. It would be interesting to see how this distributed approach work on other data sets which is a general classification problem such as face recognition from video frames.

According to the FaceNet paper [2] training on larger data sets improves network accuracy. They compare accuracies after training on 2.6-265 million images. The accuracy increases when using up to 26 million images and then levels out.
This can be put in perspective to the data sets used in this thesis containing only 3040 and 65084 images. So, finding/creating a larger data set is an important part of the future work.

Also, in the streaming process, instead of Spark Streaming this real-time implementation could be implemented with Kafka Streaming. Spark Streaming was finally selected, because the combination of the YARN execution with Kafka streaming came out with a python problem this could be resolved with the conversion of the system to Scala. So, in a new Scala implementation over Spark we could use Kafka streaming.

Furthermore, for the classification, we could use a threshold for “recognizing” unknown faces, e.g. when the softmax results to a percentage lower than 35% it could be “recognized” as a non-know individual. There is, also, a new perspective over this issue which argues over creating a new class of “unknown” individuals where are images of many different persons [41].

Finally, this algorithm result in general face recognition models. It can be used in practical applications such as login systems or surveillance. Future work could very well include the development of such a system.
9 Bibliography


# 10 Appendices

The Linux shell scripts and Python scripts are included in this chapter.

## 10.1 Terraform Scripts

The results of running “terraform” command:

```bash
terraform
```

The available commands for execution are listed below. The most common, useful commands are shown first, followed by less common or more advanced commands. If you’re just getting started with Terraform, stick with the common commands. For the other commands, please read the help and docs before usage.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>Builds or changes infrastructure</td>
</tr>
<tr>
<td>console</td>
<td>Interactive console for Terraform interpolations</td>
</tr>
<tr>
<td>destroy</td>
<td>Destroy Terraform-managed infrastructure</td>
</tr>
<tr>
<td>env</td>
<td>Environment management</td>
</tr>
<tr>
<td>fmt</td>
<td>Rewrites config files to canonical format</td>
</tr>
<tr>
<td>get</td>
<td>Download and install modules for the configuration</td>
</tr>
<tr>
<td>graph</td>
<td>Create a visual graph of Terraform resources</td>
</tr>
<tr>
<td>import</td>
<td>Import existing infrastructure into Terraform</td>
</tr>
<tr>
<td>init</td>
<td>Initialize a new or existing Terraform configuration</td>
</tr>
<tr>
<td>output</td>
<td>Read an output from a state file</td>
</tr>
<tr>
<td>plan</td>
<td>Generate and show an execution plan</td>
</tr>
<tr>
<td>push</td>
<td>Upload this Terraform module to Atlas to run</td>
</tr>
<tr>
<td>refresh</td>
<td>Update local state file against real resources</td>
</tr>
<tr>
<td>show</td>
<td>Inspect Terraform state or plan</td>
</tr>
<tr>
<td>taint</td>
<td>Manually mark a resource for recreation</td>
</tr>
<tr>
<td>untaint</td>
<td>Manually unmark a resource as tainted</td>
</tr>
<tr>
<td>validate</td>
<td>Validates the Terraform files</td>
</tr>
<tr>
<td>version</td>
<td>Prints the Terraform version</td>
</tr>
</tbody>
</table>

All other commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>Debug output management (experimental)</td>
</tr>
</tbody>
</table>
The “main.tf script for the Desktop instance creation:

```hcl
provider "aws" {
  region = "us-west-2"
  shared_credentials_file = "/Users/iliach/.aws/credentials"
}

resource "aws_security_group" "sg" {
  name = "Security group"
  description="SEcurity groupTEst"
  vpc_id = "vpc-abbe7cc"d"
  ingress {"from_port = 22
          to_port = 22
          protocol = "tcp"
          cidr_blocks = ["0.0.0.0/0"]
  }
  # outbound internet access
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "Desktop" {
  count = 1
  ami = "ami-cd372ab4"
  instance_type = "t2.micro"
  key_name = "key_pair1"
  subnet_id = "subnet-52889435"
  tags {
    Name = "Desktop"
  }
  private_ip = "10.168.0.25"
}
```
vpc_security_group_ids = ["${aws_security_group.sg.id}"
}

The “main.tf” script for the cluster creation:

variable "ips" {
  default = {
    "0" = "10.168.1.12"
    "1" = "10.168.1.13"
  }
}

variable "hostnames" {
  default = {
    "0" = "cs1"
    "1" = "cs2"
  }
}

provider "aws" {
  region = "us-west-2"
  shared_credentials_file = "/Users/iliach/.aws/credentials"
}

resource "aws_security_group" "sg" {
  name = "Security group11"
  description="security group test"
  vpc_id = "vpc-abbe7ccd"
  ingress {
    from_port = 22
    to_port = 22
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/16"]
  }
  ingress {
    from_port = 0
    to_port = 0
  }
```hcl
region "aws_instance" "Datanode" {
    count = 2
    ami = "ami-2568745c"
    instance_type = "m4.xlarge"
    key_name = "key_pair1"

    subnet_id = "subnet-dd1d2b94"
    tags {
        Name = "${lookup(var.hostnames,count.index)}"
    }
    private_ip = "${lookup(var.ips,count.index)}"

    vpc_security_group_ids = ["${aws_security_group.sg.id}"

    provisioner "file" {
        source = "install-hadoop.sh"
        destination = "/tmp/install-hadoop.sh"

        connection {
            type = "ssh"
            user = "ubuntu"
            private_key = "${file("/Users/iliach/.aws/key_pair1.pem")}"
        }
    }

    provisioner "local-exec" {
```
command = "cat /Users/iliach/.ssh/id_rsa.pub | ssh -o StrictHostKeyChecking=no -i ~/.aws/key_pair1.pem ubuntu@$lookup(var.ips,count.index) 'cat >> .ssh/authorized_keys'"

provisioner "local-exec" {
    command = "cat /Users/iliach/.ssh/id_rsa.pub | ssh -o StrictHostKeyChecking=no -i ~/.aws/key_pair1.pem ubuntu@$lookup(var.ips,count.index) 'cat >> .ssh/id_rsa.pub'"
}

provisioner "local-exec" {
    command = "cat /Users/iliach/.ssh/id_rsa | ssh -o StrictHostKeyChecking=no -i ~/.aws/key_pair1.pem ubuntu@$lookup(var.ips,count.index) 'cat >> .ssh/id_rsa'"
}

provisioner "remote-exec" {
    inline = [
        "chmod +x /tmp/install-hadoop.sh",
        "/tmp/install-hadoop.sh",
        "/opt/hadoop-2.7.3/bin/hadoop namenode -format",
    ]
    connection {
        type = "ssh"
        user = "ubuntu"
        private_key = "${file("/Users/iliach/.aws/key_pair1.pem")}"  
    }
}

resource "aws_instance" "Namenode" {
    count = 1
    ami = "ami-2568745c"
    instance_type = "m4.xlarge"
    key_name = "key_pair1"
    subnet_id = "subnet-dd1d2b94"
    tags {
        Name = "cm1"
    }
    private_ip = "10.168.1.11"
    vpc_security_group_ids = ["${aws_security_group.sg.id}"

    provisioner "file" {
        source = "install-hadoop.sh"
        destination = "/tmp/install-hadoop.sh"
connection {
  type     = "ssh"
  user     = "ubuntu"
  private_key = "${file("/Users/iliach/.aws/key_pair1.pem")}"
}
}
provisioner "file" {
  source      = "install-zeppelin.sh"
  destination = "/tmp/install-zeppelin.sh"
  connection {
    type     = "ssh"
    user     = "ubuntu"
    private_key = "${file("/Users/iliach/.aws/key_pair1.pem")}"
  }
}
provisioner "local-exec" {
  command = "cat /Users/iliach/.ssh/id_rsa.pub | ssh -o StrictHostKeyChecking=no -i ~/.aws/key_pair1.pem ubuntu@10.168.1.11 'cat >> .ssh/authorized_keys'"
}
provisioner "local-exec" {
  command = "cat /Users/iliach/.ssh/id_rsa.pub | ssh -o StrictHostKeyChecking=no -i ~/.aws/key_pair1.pem ubuntu@10.168.1.11 'cat >> .ssh/id_rsa.pub'"
}
provisioner "local-exec" {
  command = "cat /Users/iliach/.ssh/id_rsa | ssh -o StrictHostKeyChecking=no -i ~/.aws/key_pair1.pem ubuntu@10.168.1.11 'cat >> .ssh/id_rsa'"
}
provisioner "remote-exec" {
  inline = [
    "chmod +x /tmp/install-hadoop.sh",
    "/tmp/install-hadoop.sh",
    "chmod +x /tmp/install-zeppelin.sh",
    "/tmp/install-zeppelin.sh",
  ]
  connection {
    type     = "ssh"
    user     = "ubuntu"
    private_key = "${file("/Users/iliach/.aws/key_pair1.pem")}"
The “install-hadoop.sh” script:

```
#!/bin/bash
sudo apt-get -y install git
sudo apt-get -y install vim
sudo apt-get -y install python2.7
echo '
10.168.1.11 cm1
10.168.1.12 cs1
10.168.1.13 cs2' | sudo tee --append /etc/hosts > /dev/null
sudo cp -rp /home/ubuntu/.ssh /root/
sudo chmod 700 /home/ubuntu/.ssh
sudo chmod 600 /home/ubuntu/.ssh/id_rsa
cd /opt/
sudo tar xzf jdk-8u131-linux-x64.tar.gz
cd /opt/jdk1.8.0_131/
sudo update-alternatives --install /usr/bin/jar jar /opt/jdk1.8.0_131/bin/jar 2
sudo update-alternatives --install /usr/bin/javac javac /opt/jdk1.8.0_131/bin/javac 2
sudo update-alternatives --set jar /opt/jdk1.8.0_131/bin/jar
sudo update-alternatives --set javac /opt/jdk1.8.0_131/bin/javac
echo '
export JAVA_HOME=/opt/jdk1.8.0_131
export JRE_HOME=/opt/jdk1.8.0_131/jre
export PATH=$PATH:/opt/jdk1.8.0_131/bin:/opt/jdk1.8.0_131/jre/bin' | sudo tee --append /home/ubuntu/.bashrc > /dev/null
cd /opt/
sudo wget http://apache.mirrors.tds.net/hadoop/common/hadoop-2.7.3/hadoop-2.7.3.tar.gz
sudo tar xzvf hadoop-2.7.3.tar.gz
```
echo '  
echo 'export HADOOP_HOME=/opt/hadoop-2.7.3
export PATH=$PATH:$HADOOP_HOME/bin
export HADOOP_CONF_DIR=/opt/hadoop-2.7.3/etc/hadoop' | sudo tee --append /home/ubuntu/.bashrc > /dev/null

<!--
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
   http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License. See accompanying LICENSE file.
-->
<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
   <name>yarn.nodemanager.aux-services</name>
   <value>mapreduce_shuffle</value>
</property>

<property>
   <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
   <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>

<property>
   <name>yarn.resourcemanager.hostname</name>
   <value>cm1</value>
</property>

</configuration>

' | sudo tee /opt/hadoop-2.7.3/etc/hadoop/yarn-site.xml > /dev/null

sudo cp /opt/hadoop-2.7.3/etc/hadoop/mapred-site.xml.template /opt/hadoop-2.7.3/etc/hadoop/mapred-site.xml

echo '<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
   http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
   <name>mapreduce.jobtracker.address</name>
   <value>cm1:54311</value>
</property>

</configuration>
```xml
<configuration>' | sudo tee /opt/hadoop-2.7.3/etc/hadoop/mapred-site.xml > /dev/null

echo '-'xml version="1.0" encoding="UTF-8"?'>
 <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
 <!--
 Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
 http://www.apache.org/licenses/LICENSE-2.0
 Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License. See accompanying LICENSE file.
 -->
 <!-- Put site-specific property overrides in this file. -->
 <configuration>
 <property>
 <name>dfs.replication</name>
 <value>2</value>
 </property>
 <property>
 <name>dfs.namenode.name.dir</name>
 <value>file:///opt/hadoop-2.7.3/hadoop_data/hdfs/namenode</value>
 </property>
 <property>
 <name>dfs.datanode.data.dir</name>
 <value>file:///opt/hadoop-2.7.3/hadoop_data/hdfs/datanode</value>
 </property>
</configuration>' | sudo tee /opt/hadoop-2.7.3/etc/hadoop/hdfs-site.xml > /dev/null

echo 'cm1' | sudo --append /opt/hadoop-2.7.3/etc/hadoop/masters > /dev/null
```
echo 'cs1
cs2' | sudo tee --append /opt/hadoop-2.7.3/etc/hadoop/slaves > /dev/null

sudo sed -i -e 's/export\ JAVA_HOME=${JAVA_HOME}/opt\/jdk1.8.0_131/g' /opt/hadoop-2.7.3/etc/hadoop/hadoop-env.sh
sudo mkdir -p /opt/hadoop-2.7.3/hadoop_data/hdfs/namenode
sudo mkdir -p /opt/hadoop-2.7.3/hadoop_data/hdfs/datanode
sudo chown -R ubuntu /opt/hadoop-2.7.3

cd /opt/
sudo wget http://apache.mirrors.tds.net/spark/spark-2.2.0/spark-2.2.0-bin-hadoop2.7.tgz
sudo tar -xvzf spark-2.2.0-bin-hadoop2.7.tgz
echo 'export SPARK_HOME=/opt/spark-2.2.0-bin-hadoop2.7'
export PATH=$PATH:$SPARK_HOME/bin' | sudo tee --append /home/ubuntu/.bashrc > /dev/null

sudo chown -R ubuntu /opt/spark-2.2.0-bin-hadoop2.7

cd spark-2.2.0-bin-hadoop2.7

cp conf/spark-env.sh.template conf/spark-env.sh

echo 'export JAVA_HOME=/opt/jdk1.8.0_131
export SPARK_MASTER_HOST=cm1
export HADOOP_CONF_DIR=/opt/hadoop-2.7.3/etc/hadoop
export HADOOP_HOME=/opt/hadoop-2.7.3
export SPARK_WORKER_CORES=1 ' | sudo tee --append conf/spark-env.sh > /dev/null

echo 'cs1
cs2' | sudo tee --append conf/slaves > /dev/null

cp conf/spark-defaults.conf.template conf/spark-defaults.conf

cd /opt
sudo wget http://www-eu.apache.org/dist/zookeeper/zookeeper-3.4.9/zookeeper-3.4.9.tar.gz
sudo tar -zxf zookeeper-3.4.9.tar.gz
cd zookeeper-3.4.9

sudo mkdir data

echo ' tickTime=2000
dataDir=/home/ubuntu/opt/zookeeper-3.4.9/data
clientPort=2080
initLimit=5
syncLimit=2' | sudo tee --append conf/zoo.cfg > /dev/null

cd /opt
sudo wget http://www-eu.apache.org/dist/kafka/0.10.1.0/kafka_2.11-0.10.1.0.tgz
sudo tar -zxf kafka_2.11-0.10.1.0.tgz
cd kafka_2.11-0.10.1.0

echo ' clientPort=2080
server.1=cm1:2888:3888
server.2=cs1:2888:3888
server.3=cs2:2888:3888
initLimit=5
syncLimit=2

' | sudo tee --append config/zookeeper.properties > /dev/null

echo ' broker.id=1
port=9092
host.name=cm1
zookeeper.connect=cm1:2080,cs1:2080,cs2:2080
log.segment.bytes=60000000

' | sudo tee --append config/server.properties > /dev/null

The script executed to start Hadoop and spark in master instance:
hdfs namenode -format #executed only the first time to initialise hdfs
$HADOOP_HOME/sbin/start-dfs.sh
$HADOOP_HOME/sbin/start-yarn.sh
$HADOOP_HOME/sbin/mr-jobhistory-daemon.sh start historyserver
$SPARK_HOME/sbin/start-all.sh

10.2 Set root access to instances

For the root access, we first change the configurations of /etc/ssh/sshd_config, with the following changes we will allow root login and password authentication by commenting or uncommenting:

```bash
...
PermitRootLogin yes
# Only allow root to run commands over ssh, no shell
#PermitRootLogin forced-commands-only
...
# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes
#PermitEmptyPasswords no
# EC2 uses keys for remote access
#PasswordAuthentication no
...
```

Then we should reload the sshd setting by typing “sudo service sshd reload”

Finally we also copy the following paths from Ubuntu user’s .bashrc file to root’s .bashrc

```bash
export JAVA_HOME=/opt/jdk1.8.0_131
export JRE_HOME=/opt/jdk1.8.0_131/jre
export PATH=$PATH:/opt/jdk1.8.0_131/bin:/opt/jdk1.8.0_131/jre/bin
export HADOOP_HOME=/opt/hadoop-2.7.3
export PATH=$PATH:$HADOOP_HOME/bin
export HADOOP_CONF_DIR=/opt/hadoop-2.7.3/etc/hadoop
export SPARK_HOME=/opt/spark-2.2.0-bin-hadoop2.7
export PATH=$PATH:$SPARK_HOME/bin
```
after restarting the instances, you can have root access and run *Hadoop* and *spark* from root user.

### 10.3 Instance’s Start – Stop scripts

The start and stop script are: start-script.sh and stop-script.sh and the initializw or terminate the Hadoop and spark processes.

<table>
<thead>
<tr>
<th>Start-script.sh</th>
</tr>
</thead>
<tbody>
<tr>
<td>#!/bin/bash</td>
</tr>
<tr>
<td>sudo service postgresql start</td>
</tr>
<tr>
<td>$HADOOP_HOME/sbin/start-dfs.sh</td>
</tr>
<tr>
<td>$HADOOP_HOME/sbin/start-yarn.sh</td>
</tr>
<tr>
<td>$HADOOP_HOME/sbin/mr-jobhistory-daemon.sh start historyserver</td>
</tr>
<tr>
<td>$SPARK_HOME/sbin/start-all.sh</td>
</tr>
<tr>
<td>$SPARK_HOME/sbin/start-thriftserver.sh</td>
</tr>
<tr>
<td>sudo service postgresql start</td>
</tr>
<tr>
<td>service cassandra start</td>
</tr>
<tr>
<td>./start-notebook.sh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stop-script.sh</th>
</tr>
</thead>
<tbody>
<tr>
<td>!#/bin/bash</td>
</tr>
<tr>
<td>$HADOOP_HOME/sbin/stop-dfs.sh</td>
</tr>
<tr>
<td>$HADOOP_HOME/sbin/stop-yarn.sh</td>
</tr>
<tr>
<td>$HADOOP_HOME/sbin/mr-jobhistory-daemon.sh stop historyserver</td>
</tr>
<tr>
<td>$SPARK_HOME/sbin/stop-all.sh</td>
</tr>
<tr>
<td>$SPARK_HOME/sbin/stop-thriftserver.sh</td>
</tr>
<tr>
<td>sudo service postgresql stop</td>
</tr>
<tr>
<td>service cassandra stop</td>
</tr>
</tbody>
</table>

### 10.4 Multiple Face Alignment script

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from scipy import misc
import sys
import os
import argparse
```
import tensorflow as tf
import numpy as np
import facenet
import align.detect_face
import random
from time import sleep

def main(args):
    sleep(random.random())
    output_dir = os.path.expanduser(args.output_dir)
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
    # Store some git revision info in a text file in the log directory
    src_path, _ = os.path.split(os.path.realpath(__file__))
    facenet.store_revision_info(src_path, output_dir, ' '.join(sys.argv))
dataset = facenet.get_dataset(args.input_dir)

    print('Creating networks and loading parameters')

    with tf.Graph().as_default():
        gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=args.gpu_memory_fraction)
        sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options, log_device_placement=False))
        with sess.as_default():
            pnet, rnet, onet = align.detect_face.create_mtcnn(sess, None)

            minsize = 20 # minimum size of face
            threshold = [0.6, 0.7, 0.7] # three steps's threshold
            factor = 0.709 # scale factor

            # Add a random key to the filename to allow alignment using multiple processes
            random_key = np.random.randint(0, high=99999)
bounding_boxes_filename = os.path.join(output_dir, 'bounding_boxes_%05d.txt' % random_key)

            with open(bounding_boxes_filename, "w") as text_file:
                nrof_images_total = 0
                nrof_successfully_aligned = 0
                if args.random_order:
random.shuffle(dataset)
for cls in dataset:
    output_class_dir = os.path.join(output_dir, cls.name)
    if not os.path.exists(output_class_dir):
        os.makedirs(output_class_dir)
    if args.random_order:
        random.shuffle(cls.image_paths)
    for image_path in cls.image_paths:
        nrof_images_total += 1
        filename=os.path.splitext(os.path.split(image_path)[1])[0]
        output_filename=os.path.join(output_class_dir, filename+'.png')
        print(image_path)
        if not os.path.exists(output_filename):
            try:
                img = misc.imread(image_path)
            except (IOError, ValueError, IndexError) as e:
                errorMessage = '{}: {}'.format(image_path, e)
                print(errorMessage)
            else:
                if img.ndim<2:
                    print('Unable to align "%s" % image_path)
                    text_file.write('%s\n' % (output_filename))
                    continue
                if img.ndim == 2:
                    img = facenet.to_rgb(img)
                    img = img[:,:,0:3]
                    bounding_boxes, _ = align.detect_face.detect_face(img,
                        minsize, pnet, rnet, onet, threshold, factor)
                    nrof_faces = bounding_boxes.shape[0]
                    if nrof_faces>0:
                        det = bounding_boxes[:,:4]
                        det_arr = []
                        img_size = np.asarray(img.shape)[0:2]
                        if nrof_faces>1:
                            print("nrfo>1")
                            if args.detect_multiple_faces:
                                for i in range(nrof_faces):
```python
det_arr.append(np.squeeze(det[i]))
else:
    bounding_box_size = (det[:,2]-
det[:,0])*(det[:,3]-det[:,1])
    img_center = img_size / 2
    offsets = np.vstack([ (det[:,0]+det[:,2])/2-
    img_center[1], (det[:,1]+det[:,3])/2-img_center[0] ])
    offset_dist_squared = np.sum(np.power(offsets,2.0),0)
    index = np.argmax(bounding_box_size-
orient_dist_squared*2.0) # some extra weight on the centering
    det_arr.append(det[index,:])
else:
    det_arr.append(np.squeeze(det))

for i, det in enumerate(det_arr):
    det = np.squeeze(det)
    bb = np.zeros(4, dtype=np.int32)
    bb[0]=np.maximum(det[0]-args.margin/2,0)
    bb[1]=np.maximum(det[1]-args.margin/2,0)
    img_size[1])
    img_size[0])
    cropped = img[bb[1]:bb[3],bb[0]:bb[2],:]
    scaled = misc.imresize(cropped,
    (args.image_size, args.image_size), interp='bilinear')
    nrof_successfully_aligned += 1
    fn = "{}{}.{}".format(output_filename.split('.')[0], i,nn,
    output_filename.split('.')[-1])

    output_filename_n=os.path.join(output_class_dir, fn+'.png')
    misc.imsave(output_filename_n, scaled)
    text_file.write('%s %d %d %d %d
' %
(output_filename_n, bb[0], bb[1], bb[2], bb[3]))
else:
    print('Unable to align "" % image_path)
    text_file.write("%s\n" % (output_filename))

print('Total number of images: %d' % nrof_images_total)
print('Number of successfully aligned images: %d' %
nrof_successfully_aligned)
```

65
def parse_arguments(argv):
    parser = argparse.ArgumentParser()

    parser.add_argument('input_dir', type=str, help='Directory with unaligned images.')
    parser.add_argument('output_dir', type=str, help='Directory with aligned face thumbnails.')
    parser.add_argument('--image_size', type=int, help='Image size (height, width) in pixels.', default=182)
    parser.add_argument('--margin', type=int, help='Margin for the crop around the bounding box (height, width) in pixels.', default=44)
    parser.add_argument('--random_order', help='Shuffles the order of images to enable alignment using multiple processes.', action='store_true')
    parser.add_argument('--gpu_memory_fraction', type=float, help='Upper bound on the amount of GPU memory that will be used by the process.', default=1.0)
    parser.add_argument('--detect_multiple_faces', type=bool, help='Detect and align multiple faces per image.', default=True)

    return parser.parse_args(argv)

if __name__ == '__main__':
    main(parse_arguments(sys.argv[1:]))

10.5 Face Recognition scripts
10.5.1 Alignment script

spark-submit \
--master yarn \
--deploy-mode cluster \
--queue ${QUEUE} \
--num-executors 2 \
--executor-memory 7G \
--total-executor-cores 3 \
--driver-memory 5g \
--conf spark.dynamicAllocation.enabled=false \
--conf spark.yarn.maxAppAttempts=1 \

10.5.2 Training script

```bash
spark-submit
--master yarn
--deploy-mode cluster
--queue ${QUEUE}
--num-executors 2
--executor-memory 13G
--total-executor-cores 3
--driver-memory 13G
--conf spark.dynamicAllocation.enabled=false
--conf spark.yarn.maxAppAttempts=1
--archives hdfs:///user/root/Python.zip#Python
--archives hdfs:///user/root/Python.zip#Python
/root/facenet_/facenet/src/classifier.py TRAIN
hdfs://cm1:9000/user/root/lfw_dataset /root/lfw_mtcnnpy_160/ \
--image_size 160 \ 
--margin 32 \ 
--random_order
```

10.5.3 Classification script

```bash
spark-submit
--master yarn
--deploy-mode cluster
--queue ${QUEUE}
--num-executors 2
--executor-memory 12G
--total-executor-cores 3
--driver-memory 12G
--conf spark.dynamicAllocation.enabled=false
```
10.6 Standalone execution scripts

For the training phase the executed commands are the following, after defining the PYTHONPATH,

```
export PYTHONPATH = /WhereTheOriginalFaceNetCodeIs/facenet/src/
```

After the export the Training command is:

```
python /root/facenet_standalone/facenet/src/classifier.py TRAIN 
/root/lfw2_mtcnnpy_160 
/root/facenet_/facenet/models/facenet/20170511-185253/ 
/root/facenet_standalone/facenet/models/lfw_classifier.pkl 
--batch_size 1000 
--min_nrof_images_per_class 40 
--nrof_train_images_per_class 35 
--use_split_dataset
```

And the classification command:

```
python /root/facenet_standalone/facenet/src/classifier.py CLASSIFY\ 
/root/lfw2_mtcnnpy_160 /root/facenet_/facenet/models/facenet/20170511-185253/ 
/root/facenet_standalone/facenet/models/lfw_classifier.pkl 
--batch_size 1000 
--min_nrof_images_per_class 40 
--nrof_train_images_per_class 35 
--use_split_dataset
```
### 10.7 Spark streaming script

```python
import base64
import json
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
import re
import os

def fileName(data):
    cmd1 = '/opt/hadoop-2.7.3/bin/hdfs dfs -mkdir /user/root/stream_data/
    os.system(cmd1)
    cmd2 = '/opt/hadoop-2.7.3/bin/hdfs dfs -mkdir /user/root/stream_data/camera_photos/
    os.system(cmd2)
    debug = data.toDebugString()
    pattern = re.compile("hdfs:/.*/.png")
    files = pattern.findall(debug)
    for file in files:
        print(file)
        print("n")
    os.system("sh /root/classify.sh")
    print(files)

from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext

ssc = StreamingContext(sc, 10)
lines = ssc.textFileStream("hdfs:///user/root/stream_data/camera_photos")
files = lines.foreachRDD(fileName)

ssc.start()  # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```

And the classify.sh script which is called in the streaming process is:

```bash
export PYTHON_ROOT=./Python
export LD_LIBRARY_PATH=${PATH}
export PYSPARK_PYTHON=${PYTHON_ROOT}/bin/python
export SPARK_YARN_USER_ENV="PYSPARK_PYTHON=Python/bin/python"
```
export PATH=${PYTHON_ROOT}/bin:$PATH
export QUEUE=default
export PYTHONPATH=/root/facenet_/facenet/src/

/opt/spark-2.1.1-bin-hadoop2.7/bin/spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --queue ${QUEUE} \
  --num-executors 2 \
  --executor-memory 12G \
  --total-executor-cores 3 \
  --driver-memory 12g \
  --conf spark.dynamicAllocation.enabled=false \
  --conf spark.yarn.maxAppAttempts=1 \
  --archives hdfs://user/root/Python.zip#Python \
  --conf spark.executorEnv.LD_LIBRARY_PATH="$JAVA_HOME/jre/lib/amd64/server" \
  /root/facenet_/facenet/src/classifier.py CLASSIFY hdfs://cm1:9000/user/root/stream_data/ \
  /root/facenet_/facenet/models/facenet/20170511-185253/ \
  /root/facenet_/facenet/models/lfw_classifier.pkl

Finally, the results for the 10 streamed pictures of Ariel Sharon are:

```
Loaded classifier model from file "/root/facenet_/facenet/models/lfw_classifier.pkl"

0  Ariel Sharon: 0.708
1  Ariel Sharon: 0.756
2  Ariel Sharon: 0.750
3  Ariel Sharon: 0.685
4  Ariel Sharon: 0.677
5  Ariel Sharon: 0.765
6  Ariel Sharon: 0.711
7  Ariel Sharon: 0.702
8  Ariel Sharon: 0.732
9  Ariel Sharon: 0.752

Accuracy : 1.00
```