ucc

Coláiste na hOllscoile Corcaigh, Éire
University College Cork, Ireland

Accelerating Deep Learning on an Apache Spark Environment using GPUs

Final Year Project 2018
CS4501

By: Dlo Bagari
Student Number:114702261
Supervisor
Prof. John P. Morrison

Second Reader: John Herbert

Contents

ABSTRACT

Deep Learning models, capable of high-accuracy prediction, require training with large volumes of data resulting in a huge number of linear algebra computations.

This report presents the results of a novel study employing state of the art Graphics Processing Units (GPUs) to speed up the execution of a sophisticated algorithm, the Convolutional Neural Network (CNN), in an Apache Spark environment, to identify specific objects in a large set of arbitrary images.

The results presented here indicate that the approach take can reduce the execution time compared to using CPU alone by a factor of 28.

1. INTRODUCTION

Object recognition comes naturally to humans. Our brains have evolved to recognise patterns and images with easy. In contrast, by construction a digital computer, based on processing numbers is not good at this task.

Why is it that our brains are good at this task? Can we construct a model of the brain within our computer to perform image recognition? If we could, there are a huge number of applications that could exploit this ability.

1.1 Object recognition

Before we can hope to be able to build a machine capable of recognising objects within images, we need to understand how we, as humans, solve this problem. Looking to the following image:

We use our eyes and our brain to understand what this image is about. Quickly we can recognize the object in this image and analyze it with our brain's power. And if you have been asked what type of dog is in the above image? Your eyes will focus, and different parts of this image and your brain will analyze what your eyes are focusing on, finally, your mind will make a decision based on its experience even if the object is not clear or you do not have enough time to think.

This task is relatively easy for a human. However, not all object recognition task are straightforward.

Consider the following for example. The images in Figure 1.1 representing different variants of the same object. Trying to distinguish between them is not simple.

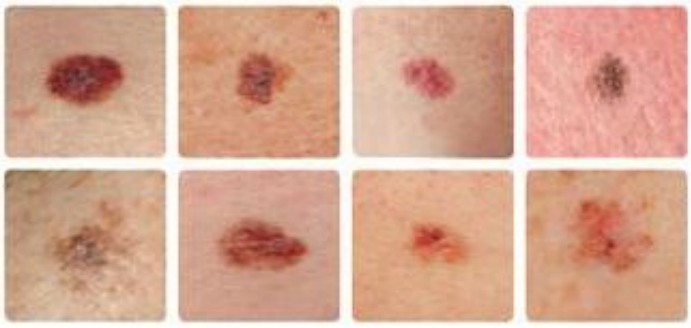


Figure 1.1: types of skin cancer

We can immediately decide that all the images are similar, and they represent almost the same thing, but the colors and shapes are different. However, since we typically have no experience in understanding how these objects could be categorised to reflect interesting properties, it is hard to say that two or more of these images reflect these properties.

Images in Figure 1.1 are images of types of skin cancer representing the most commonly diagnosed cancer in the United States. On average over five million cases are diagnosed each year, more than 100,000 of these cases are the deadliest form of skin cancer, they cause over 9,000 deaths a year, costing the United States healthcare system more than 8 billion dollars per year. Doctors can not simply distinguish between these type of deadly skin cancers, a sample of skin tissue needs to be sent to the laboratory and it this process can sometimes take weeks for a skin tissue to be examined. Even surgeons struggle to remove only the affected area on the skin when intervening to treat the affected area.

Can this be done better? Computers are useful for solving complex tasks, but can they recognize the type of skin cancer better than the human? To answer this question let's see how a computer sees one of the above images as illustrated in figure 1.2:

255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	245	255	245	255	255	255
255	255	255	255	255	255	255	255	255	225	1	7	0	0
255	255	255	255	255	255	255	255	255	255	5	126	137	138
255	255	255	255	255	255	255	255	251	255	0	124	120	115
255	255	255	255	255	255	255	255	248	255	4	128	138	134
255	255	255	255	255	255	255	255	254	253	4	115	123	134
255	255	255	255	255	255	255	255	253	255	9	145	125	124
255	255	255	255	255	255	255	255	255	255	0	146	120	126
255	255	255	255	255	255	255	255	255	255	1	128	129	128
255	255	255	255	255	255	255	255	255	255	1	128	129	128
255	255	255	255	255	255	255	255	255	255	1	128	129	128
255	255	255	255	255	255	255	255	255	255	1	128	129	128
255	255	255	255	255	255	255	255	255	255	1	128	129	128
255	255	255	255	255	255	255	255	255	255	1	128	129	128
255	255	255	255	255	255	255	255	255	255	1	128	129	128

Figure 1.2: how a computer sees an image

From Figure 1.2, it is evident that a computer sees images as a sequence of numbers, and it can display these number as an image on a screen, but these numbers in themselves have no semantic meaning - unless it is provided by the programmer. In general, this cannot be done for all interesting variants of a potentially cancerous object. The question then becomes: can a computer that is equipped with a rudimentary understanding of the properties of cancerous spots evolve that understanding by examining and classifying millions of such images? To answer this question, we will first explain what a digital image is.

1.2 Digital Image

A digital image is a collection of pixels, and each pixel contains a color, each color can be represented by three numbers ranging from 0 to 255, these three numbers represent the colors Red, Green and Blue. For an example (255, 0, 0) represents the color Red. Using different values for these numbers, we can create whatever color we wish.

Consider a colored image consisting of 2x2 pixels. This image can be represented as a 3-Dimensional Array, with every pixel having a value between 0 and 255, and based on this information computers can work with images, Figure 1.3, below shows the three colors in three channels.

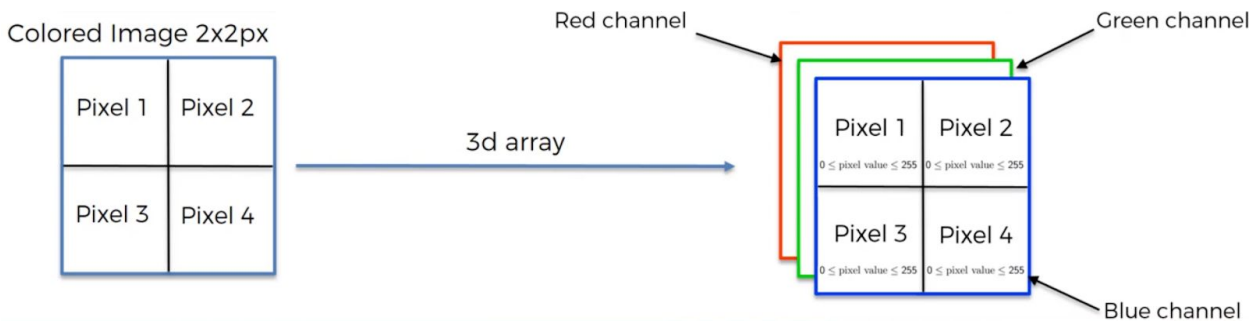
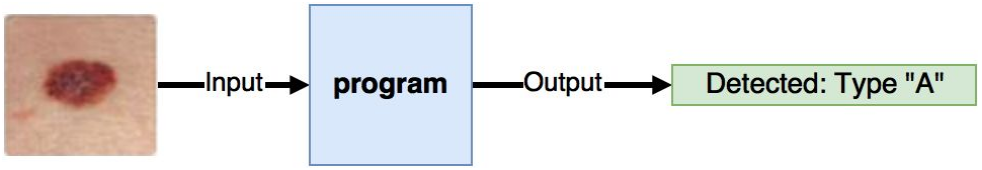
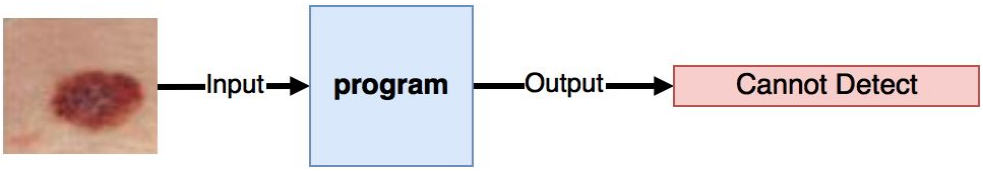


Figure 1.3: three channels in a colored image

So if we have a colored image for one type of skin cancer with size 200 x 200 pixels, then that image is represented by 200 x 200 = 40000 pixels, and each pixel represented by three numbers. Each number is an integer that ranges from 0 (black) to 255 (white). So in total, we have 40000x3 =120000 numbers to represent that image. Now we ask our question again, *how can we associate a meaning (in term of the application domain) with each of these* 120000 numbers*?* Assume we have written a program with a set of rules to follow, and our program takes an image as input and outputs the type of Skin Cancer:



Now if the position of the spot in the image is moved, we will still have the same amount of numbers, but not the same exact sequence of numbers and there is no guarantee that our program will detect the type of the skin cancer because the sequence of numbers is changed.



To ensure that our program recognizes the same object in an image that is presented to our program in different ways, we have to change or add more rules to our program. It is tough to write a program to figure out what these numbers will

represent because we cannot define reliable and straightforward rules in our program to map any given skin cancer image to its type. Even if we have a good idea about how to write a program to solve this problem, the program is going to be extremely complicated because we will use a combination of a massive number of weak rules, and even if we can find reliable rules, these rules may need to be changed, necessitating a change in the program as well.

The alternative solution for this problem would be to let the computer find its own rules to solve this task. And this will lead us to another question, how a computer can write down its own rules or algorithm to solve a task? We need to find a way for computers to learn and obtain experience; building their own instructions for solving problems. Fortunately, Developers and researchers have made incredible advancements in artificial intelligence over the past few years, they have managed to create a learning mechanism for computers which is an abstraction of ***our brain***, the most powerful learning mechanism on the earth. And with this learning mechanism, we can improve the computer vision, and achieve high success rates for Object recognition and image classification, by training computers and letting them learn from existing data to build their own algorithms for solving a problem related to the given data set. And here we are facing the most critical question, how long will take to train a computer? This will depend on the learning mechanism configurations and the data which will be used to teach a computer. When a computer learns from the given data, it has to do heavy computations, and these computations will slow down the learning process. We don't want to wait for long time to test and figure out if a computer has learned or not. We need a quick way to train a computer for two reasons:

- If a computer has learned, then we might reconfigure the learning mechanism and retrain that computer to obtain better performance.
- If a computer has not learned, then we can change the learning mechanism configurations and retrain that computer.

In this report, we will build a learning mechanism for computers, and train a computer in a distributed environment to categorize millions of images, then we will accelerate the training process to reduce the amount of time spent on training process from weeks to hours.
The following images are some examples from our training data:

If we succeed in training a computer which will organize the object in a given image, then our trained computer can be used to solve multiple problems, such as building an image search engine App for mobile phones. For example, consider the process of repairing a machine, and replacing an item in that machine, or you have an item, and you need to buy another one, but you are not sure what that item is called, you may spend hours guessing a name for that item and searching for that item in e-commerce websites until you find what are you looking for. Or you can just take a picture of that item with your mobile phone and let the image search engine App to take care of finding that item on e-commerce websites or in shopping centers.

The report will be divided into the following section:

- Background: here we will introduce existing learning mechanisms and tools for accelerating the training process.
- Analysis: here we will analyze our problem, and find out how the existing learning mechanisms and tools for accelerating the training process will help us to solve our problem.
- Design: here we will explain how we will build a system to solve our problem, and what libraries we will use.
- Implementation: here we will transform our design to executable code.

- Training and evaluation: here we will run our application, measure the performance of our application, and maximize our application performance.
- Conclusion: here we measure the project results, define how much we speed up training deep learning application in a Spark Environment using GPUs, and how the required time for training a deep learning application can be more reduced.

2. BACKGROUND

Today we have a significant amount of stored data, and the amount of data we store is rapidly growing. Discovering a way to make a machine capable of learning from stored data, will address the primary challenge in big data, which is how to transform data into actionable knowledge to improve the way we look at the world around us and to minimize errors when making complex decisions.

This section introduces existing machine learning approaches, machine learning libraries, and the Spark environment for training a machine on Graphics processing units (GPUs) in parallel.

2.1 Data and Machine learning

Arthur Samuel proposed the following definition for Machine Learning:

"Machine Learning relates to the study, design, and development of the algorithms that give computers the capability to learn without being explicitly programmed."

--(Arthur Samuel)

A dataset refers typically to content available in the structured or unstructured format. In structured data, each element of the dataset conforms to a predefined structure, whereas in an unstructured dataset, elements do not conform any predefined structure.

The primary ingredient for any machine learning algorithm is knowledge about the data and its type. For a machine to learn to solve a problem, a collection of suitable datasets need to be available in an appropriate format and structure.

The amount, format, availability and structure of the data, partitions machine learning types into three main classes of learning:

- Supervised learning: learns from labeled data. A record associated with each datum in a dataset contains a pair ([inputs], output). Supervised learning aims to produce a general mapping function F, that transforms

inputs to outputs, such that F([inputs])→output. An example of supervised learning is image classification, where each record in the training dataset is a 'pair(image, label).' The learning algorithm analyzes the input image and learns how to map it to the specified label.

- Unsupervised learning: Understanding the dataset, exploring it, and discovering hidden patterns in the dataset by focusing on the structure of the dataset for building machine learning models. Records in the dataset are not labeled and contain only inputs. An example of unsupervised learning is the process of grouping similar inputs in a recommendation system.
- Reinforcement learning: the learning process in Reinforcement learning is entirely different to the previous learning types. It assumes that an agent, which can be a robot or a computer program, interacts with a dynamic environment to achieve a specific goal. The environment can be a set of statuses, and the agent takes different actions to move from one state to another until it reaches the goal state. When an agent reaches the goal state, it receives a reward. When moving, an agent learns what action it should take to move from one state to another state until it reaches the goal state. An example of Reinforcement learning is a program for driving a vehicle.

To map a problem to one or more machine learning types, first, we have to understand the problem in general, the structure of the training dataset, and the end goals.
Once the learning type is selected, we have to look more closely at the raw data, and transform the raw data so that it can be understood, managed, and analyzed. Data typically originates from many different sources, and consequently datasets may differ widely in their structure or have little or no structure at all. For these reasons a raw dataset may not be immediately suitable for training purposes. It must be **preprocessed** to remove noise, to deal with missing, duplicated or incorrect values, to change the distance between data points, removing outliers, adding more features to the data, or reducing the number of dimensions.
After data preprocessing, the data is ready to be **fit** into a machine learning algorithm to train a model. A wide variety of machine learning algorithms are available, such as k-nearest neighbors, naïve Bayes, decision trees, support vector machines, logistic regression, k-means, and so on.

The next step after training a model is the **validation** step; this step is related to all the previous steps. In the validation step, we find out how well the model has learned from the training data, and how the model will perform on new unseen data. If the model is too specific, overfitting to the training data has taken place,

and if the model is too generic, underfitting to the training data is at fault. For example, if we were to ask a machine, about the weather in Cork and the answer is always 'raining' (although this is indeed correct most of the time!) underfitting is the likely cause and this machine will not be really useful for making valid predictions. Figure 2.1, explains the required steps of machine learning:

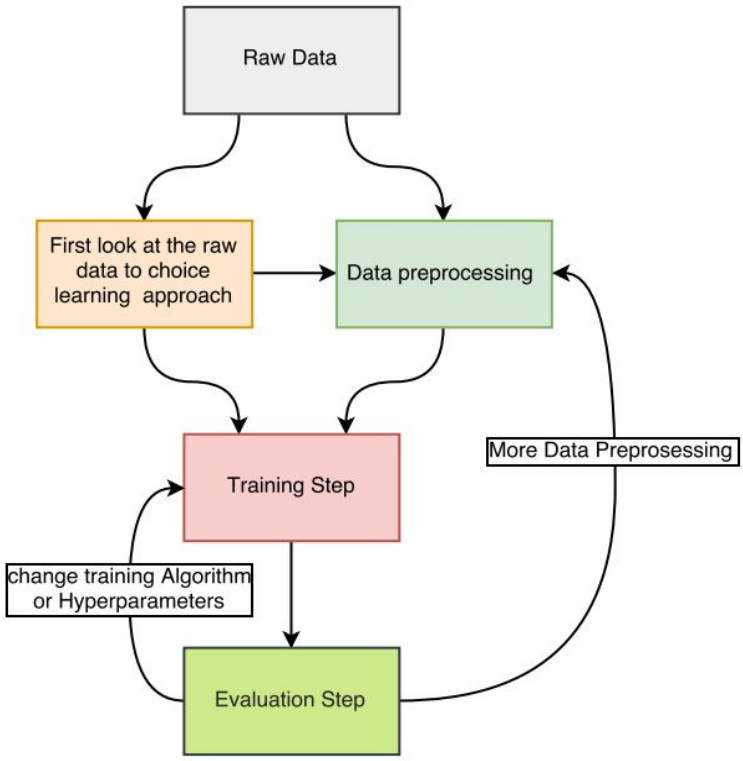


Figure 2.1: Machine learning steps

2.2 Machine Learning Applications

We try to keep our devices as smart as possible by applying machine learning technology to applications running on our devices. The following table contains some machine learning applications.

Industry	Applications
Health-care	Evidence-based medicine, epidemiological surveillance, drug events prediction, and claim fraud

	detection.
Financial	Credit risk scoring, fraud detection, and anti-money laundering.
Web	Online campaigns, health monitoring, and ad targeting.
Networks	Cyber security, smart roads, and sensor health monitoring.
Environment	Weather forecasting, pollution modeling, and water quality measurement.
Retail	Inventory, customer management and recommendations, layout, and forecasting.

2.3 Machine Learning Libraries

There is increasing interest in machine learning research within industry and academia and many influential open source libraries have been built which implement machine learning applications more efficiently.
The following table compares some of the most popular libraries for machine learning.

Software	Open Source	Platform	Written in	interface	Cuda support	Convolutional nets	Parallel execution	OpenMP support
Apache MXNet	Yes	Linux, macOS, Windows, AWS, Android, iOS, JavaScript	Small C++ core library	C++,Python , Julia,Matlab , JavaScript, Go, R, Scala, Perl	Yes	Yes	Yes	Yes

Deeplearning4j	Yes	Linux, macOS, Windows, Android (Cross-platform)	C++, Java	Java, Scala, Clojure, Python(Keras),Kotlin	Yes	Yes	Yes	Yes
TensorFlow	Yes	Linux, macOS, Windows	C++, Python	Python(Keras), C/C++, Java, Go, R	Yes	Yes	Yes	Yes
Theano	Yes	Cross-platform	Python	Python(Keras)	Yes	Yes	Yes	Yes
Caffe2	Yes	Linux, macOS, Windows	C++, Python	Python, MATLAB	Yes	Yes	Yes	Yes

2.4 Deep Learning

Deep learning employs algorithms that learns directly from samples in an pre-existing dataset. Deep Learning frameworks are used to specify the learning approach and appropriate training datasets. A machine uses these frameworks to develop algorithms to recognize patterns in the training dataset and subsequently in the production data. The Deep Learning process continually rewrites the learning algorithm in contrast to the Machine Learning approach in which the learning algorithm is fixed. Thus, in Machine Learning feedback from the machine reinforces certain pathways through the learning algorithm whereas in Deep Learning, the number of pathways through the learning algorithm may change depending on the machine feedback. Deep learning has become an important approach to all industries. For example, deep learning helps the healthcare industry to address complicated tasks such as cancer detection and drug discovery.

2.4.1 Neural Network

A Neural Network is a highly structured network. The fundamental unit of the neural network is a node (artificial neuron), which is loosely based on the biological neuron in the mammalian brain. Biological neurons are responsible for receiving and sending messages in the form of electrical currents called nerve impulses; they have input channels called Dendrites and an output channel called Axon, the input channel receives an electrical signal passes that signal to the

central core in neuron where the signal is processed and analyzed, then send the signal to the next neuron via the output channel. Figure 2.2, is a picture of a biological neuron and its components:

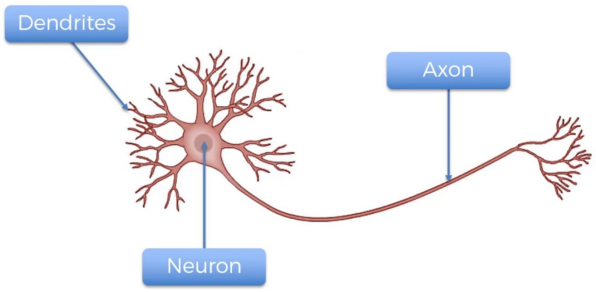


Figure 2.2: A biological neuron components

The connection between nodes (neurons) in neural network carries a weight, and develop during the learning process. Figure 2.3 illustrates a neural network with one node.

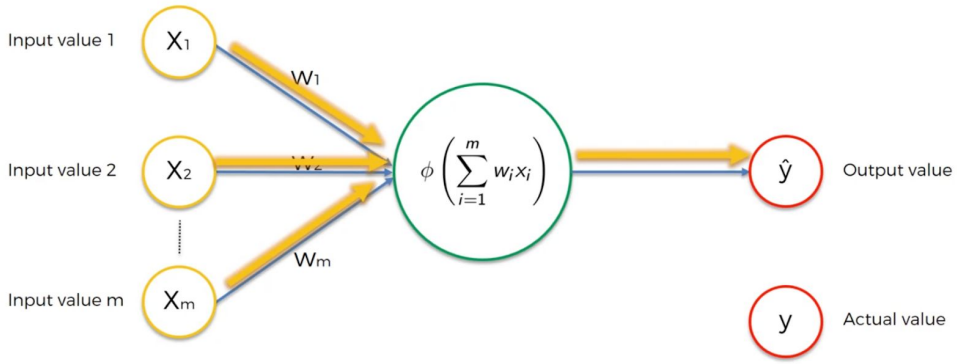


Figure 2.3: components of neural network with one node

in Figure 2.3, input values $\{X_i, \ldots, X_m\}$ are the features of each example in a training dataset. The connection between two nodes carry a weight which represents the learned information extracted from the raw training data. All input values go through a function that sums up all input values multiplied by corresponding weights, and add a bias to the calculated result; the result is fed into an activation function to produce the estimated value **ŷ**. A neuron in neural network attempts to find a function that describes the relationship between $\{x_1, x_2, \ldots, x_n\}$ and the actual value **y** in a linear equation such as:

$$Y = \text{bias} + W_1 * X_1 + W_2 * X_2 + \ldots\ldots + W_n * X_n$$

The estimated value $\hat{\mathbf{y}}$ is used to calculate the error, by comparing the estimated value $\hat{\mathbf{y}}$ with actual value $\mathbf{y}$, this error is then fed back through the network by various techniques to update the weight on each connection between two nodes. Each layer in the neural network is a collection of nodes, the first layer is known as the input layer, the final layer is known as the output layer, and all layers between the input layer and the output layer are known as hidden layers. Figure 2.4, shows how the neurons in each layer are fully connected to all neurons in next layer:

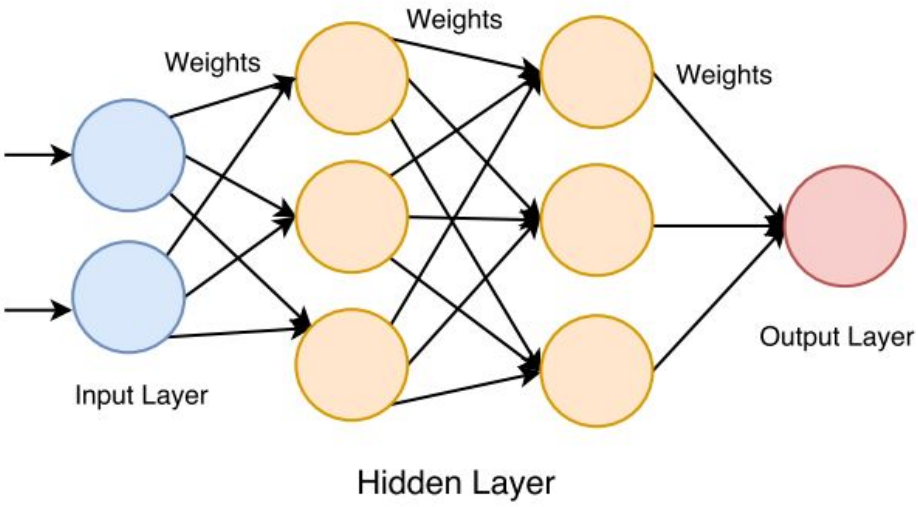


Figure 2.4: neural network layers

Each node in the first hidden layer receives a set of inputs from the input layer and processes (activates) these inputs, then its score is passed on as an input to the next hidden layer for further activation (this why we call it Deep Learning). These activations are then passed to the next hidden layer and so on until they reach the output layer, where the final result is produced. The process of sending activations to the next layer until they reach the output layer is known as forward-propagation (Forward Prop).

All the nodes in a layer (except the input layer) have the same classifier (Activation Function). Each classifier produces a different value, and this value is not random: we always get the same value with the same input. In Figure 2.5 a

node in hidden layer receives a set of inputs from the input layer and outputs a value for next layer:

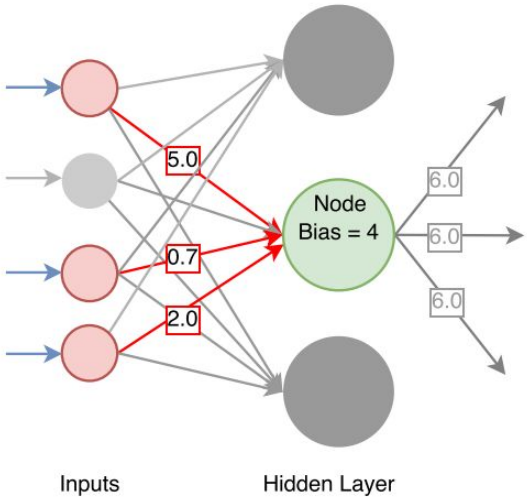


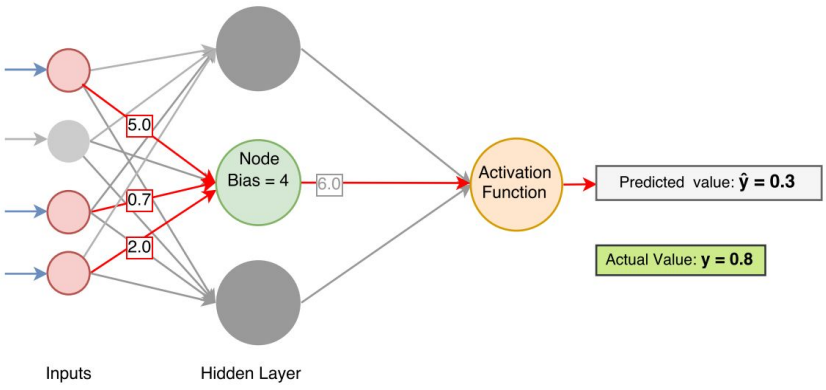
Figure 2.5: inputs, outputs of a Node in hidden layer

After each forward-propagation, the output from forward-propagation is compared with the actual value to calculate the cost (difference between predicted value and the actual value). During the network training process, we want to minimize the cost. To achieve this goal, the network updates the weights and biases after N number of training examples, the process of updating the weights and biases is known as ***backpropagation*** **(back prop).**

2.4.1.1 Forward-propagation and backpropagation

forward-propagation:

Each node in each hidden layer receives set of inputs from the previous layer, and produces a value to nodes in the next layer, when the output layer receives inputs, it predict a value **ŷ:**



Backpropagation:

- A cost function (loss function) is used to calculate the difference between the estimated value **ŷ** and the actual value **y**. If **ŷ** equal to **y,** we don't do anything.
- If **ŷ is** not equal to **y,** a gradient (derivative or slope) is calculated**,** which measures the rate at which the cost will change with respect to a change in a weight or bias. Here we are trying to minimize the error between **ŷ** and **y,**
- the result of the gradient is fed into network, which updates each weight involved in the forward-propagation process. Figure 2.6 shows the back propagation process for one node in the hidden layer.

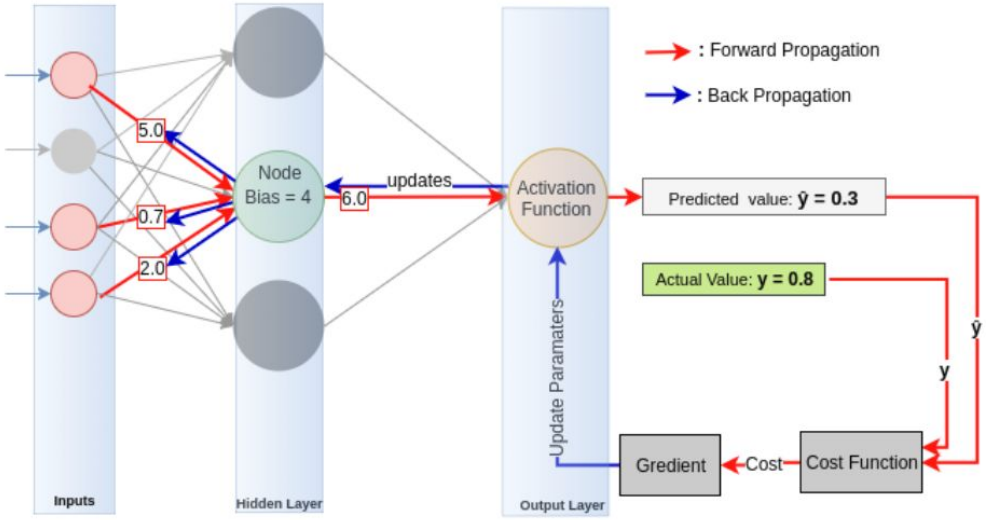


Figure 2.6: Backpropagation to update a node's parameters

2.4.1.2 Activation functions

In neural networks, the activation function of a node controls the output of that node to the next layer. It simply maps a particular output to a particular input or set of inputs, and it also used to impart nonlinearity into the network's modeling capabilities.

In forward-propagation steps in a neural network, we have some steps where we use an activation function, to calculate (or control) the passed over values. Another way to simply explain what the activation function does is to assume that each node in the neural network is an integrated circuit which has N inputs and M outputs as illustrated in Figure 2.7. Each input voltage is between 0 – 230 and all outputs have the same voltage between 3 and 5, and the essential thing in this

circuit is to break linearity, where the relationship between input voltage and the output voltage is nonlinear.

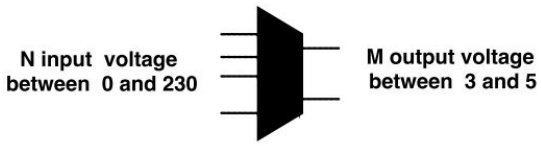
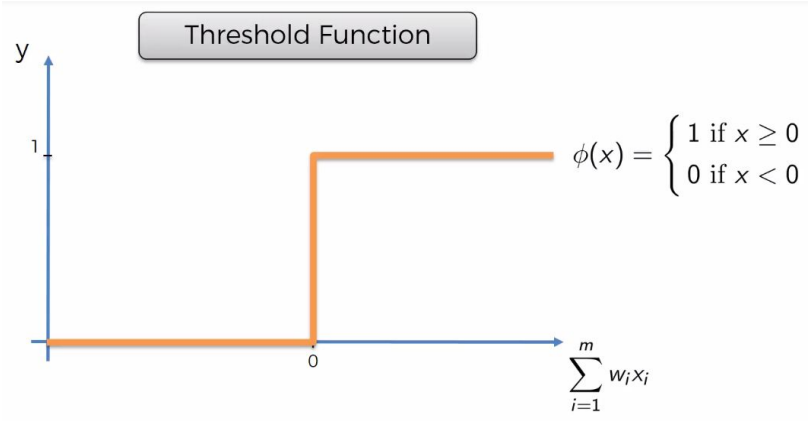


Figure 2.7: integrated circuit has N inputs and M outputs

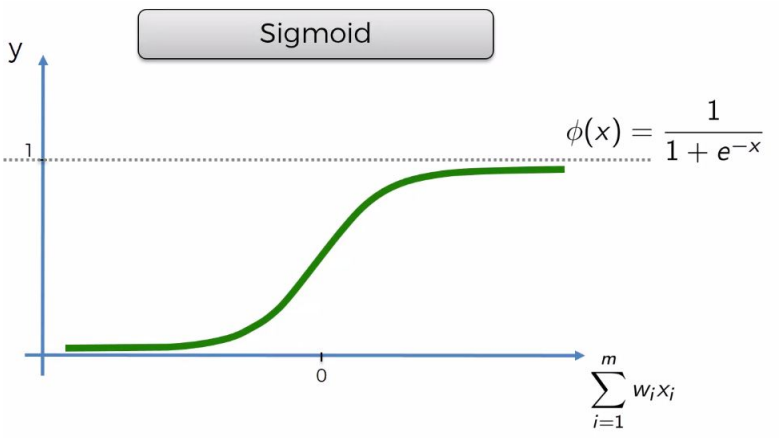
There is a range of Activation function can be used in a neural network the follows are four types of them:

- *Threshold Function (Step Function)*
  threshold function is very simple function, it can be used for binary classification (where we have only two classes), the output of this function can be zero or one:



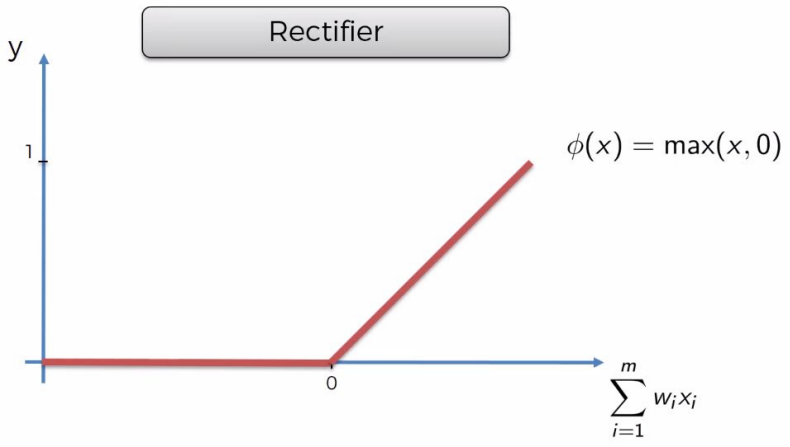
Threshold Function, *Source: Super Data Science*

- *Sigmoid Function*
  The sigmoid function is used in Logistic Regression part of Machine Learning. The output of this function is smooth, unlike the threshold function. Weighted sum values below zero drop off and weighted sum values above zero approximates towards one. This function is beneficial in the output layer. Especially for probability prediction. The sigmoid function produces the following curve:



Sigmoid function, Source:Super Data Science

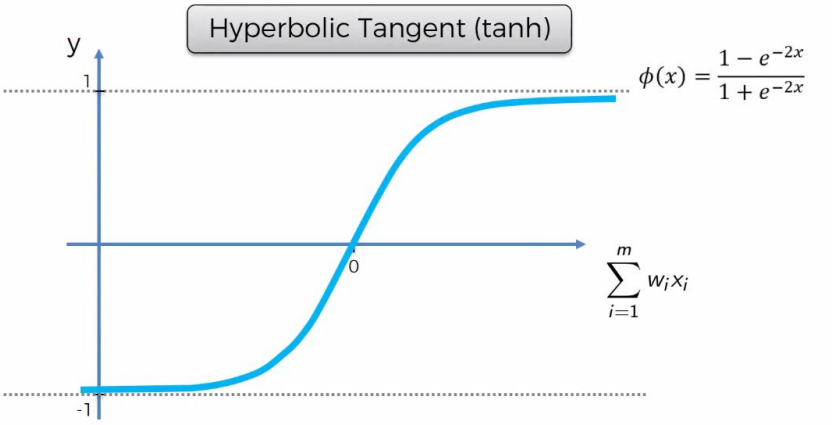
- *Rectifier Function*
  The rectifier function is the most widely used activation function for hidden layers in the Neural Networks. It looks as follows:



Rectifier function, Source:Super Data Science

Weighted sum values below zero will pass a zero to the output signal. Weighted sum above zero progress gradually towards one.

- *Hyperbolic Tangent (tanh) function*
  The output from the Hyperbolic Tangent function looks like:



Hyperbolic Tangent (tanh) function, Source:Super Data Science

The hyperbolic Tangent function is similar to the sigmoid function, however the difference is that the output values for weighted sums below zero go negative one.

2.4.1.3 Benefit of Neural Networks

When we only need to find simple patterns in a given dataset, a basic classification algorithm and tools can be used, like logistic regression or SVM, which they are typically enough to find simple patterns. However, when a task is to discover complex patterns, the task becomes extremely hard for a machine to do. Neural networks allow a machine to analyze complex patterns by breaking the complex patterns down into series of simpler patterns. For example when a neural network tries to decide whether an image contains a human face, it first use edges to detect different parts of the face, the eyes, nose, ears, lips and so on, then combine the result to form the whole face, and this is based on the way our brain would detect a human face in an image. When the number of possible patterns increases in data, we typically increase the number of nodes in hidden layers, however, this requires more time to train a neural network. The good news is we have the processing power and the speed of GPUs which can reduce the amount of time required to train a neural network.

2.5 Concurrency and Parallelism

Machine learning requires a significant amount of data to train a model. Typically the training process can be slow due to the number of required computations and

data transformations. To speed up the training process, we need better techniques and algorithms for execution of the training process. A conventional approach for speeding the execution of a program is concurrency and parallelism. With concurrency, we can split the training dataset into small blocks and assign each block to a task, and with parallelism, each task can execute independently and at the same time.

2.5.1 Processes and Threads

A process is an executable program requires some resource, it requires at least one processor to execute the code, memory to hold the instructions, and I/O devices. Every process has an address space comprised of the memory locations accessible to the process. If a process, during its execution, runs out of the memory and tries to use the memory location of another process, the operating system terminates that process and raise an exception.
A thread is a sequence of executable instructions within a process. We can break the sequence of instruction execution in a process by breaking the sequence of instructions to parts and assign each part to a thread (task). For example, consider we have following instructions to be executed in one process:

```
instruction 1
instruction 2
instruction 3
....
....
a := 0;
while a not equal 10 do
      //ask user to enter a number
      a:= user_input();
instruction N-1
instruction N
```

Here the process has one thread (main thread) which contains all instructions. These instructions execute sequentially from instruction 1 to instruction N. when the while loop starts the next instruction after the end of the while loop will not execute until the while loop execution is finish. While the process is waiting for the user input in the while loop, the process cannot execute any further instructions, to continue the execution of the instructions we can put the while loop instructions in a thread as follows:

```
instruction 1
instruction 2
instruction 3
....
....
....
new Thread(
      a := 0;
      while a not equal 10 do
            //ask user to enter a number
            a:= user_input();
    ).start();
instruction N-1
instruction N
```

Here the process has multiple threads: one main thread and one child thread (which contains a while loop instruction). We can run these threads concurrently in the same process as they share the same address space and have access to the same memory location.

A race condition arises when two or more threads concurrently access the same memory location, and at least one of the threads tries to update the location. For example, if we have two or more threads running in parallel and each thread might be executing the following instruction :

```
//'number'  is a shared memory location among the threads
//generateRandomNumber() returns a random number
number := generateRandomNumber();
```

The value of variable 'number' is unpredictable, the last thread executes the instruction to assign a value to variable 'number ' is a winner. To avoid this scenario, we have to set locks (synchronization) on shared memory where only one thread can update (write to) any location in shared memory at any time.

2.6 Apache Hadoop

Data is everywhere. We get data from social media, sensors, stored data and a variety of others. Traditionally we store data in a relational database in our desktop computer; however, when that data grows very quickly from few

gigabytes to hundreds or thousands of gigabytes, we reach the limits of our desktop computer. What should we do? Hadoop is an open source framework of the Apache foundation. It is designed to handle parallel processing on a massive volume of data which could be structured, unstructured or semi-structured. This massively parallel processing is done with an excellent performance which is suitable for many business environments for processing batches of datasets and producing quick results. Hadoop splits and distributes dataset across multiple machines, and instead of moving the data to computation, Hadoop sends the computation to the dataset which decreases I/O operations since the dataset size is much larger than the computation code. Each machine processes a split of the dataset and stores the processing result on its local file system, then Hadoop gathers all results and reduces them to one final result , and this process is called MapReduce. Figure 2.8 shows the steps involved in processing a dataset in Hadoop (MapReduce):

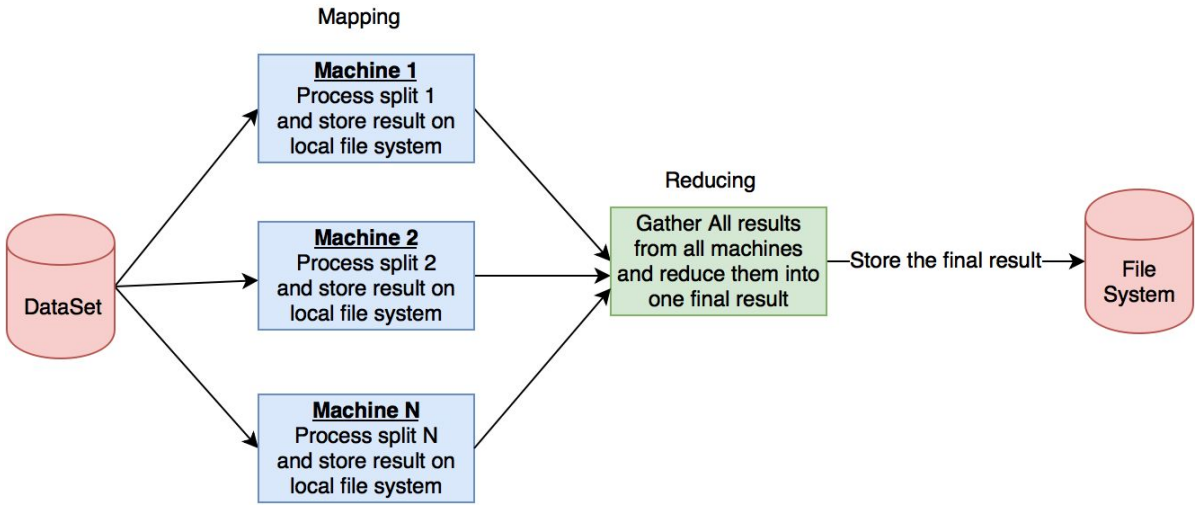


Figure 2.8 split dataset and process each split individually using MapReduce.

Resource-Manager is one of Hadoop components which allocate resources for jobs and tracks all jobs execution. When a job stops responding to N resource-manager requests, the resource-manager considers that job has failed, and it allocates another machine to recompute the failed job from the beginning which increases the I/Os operations and slow down the performance.

In Figure 2.8 each machine writes its result to its local file-system, and if we need to do more computation on MapReduce result, Hadoop has to read the stored result again from disk to memory which causes more disk I/O. Due to this disadvantage, the amount of the time takes to run an iterative MapReduce algorithm is no longer acceptable in most situations, which means MapReduce

jobs are only appropriate in a specific set of use cases. We need a better environment where each task can write its result to a distributed or shared memory rather than writing their result to disk, and it can handle fault tolerance more efficient.

2.7 Apache Spark

Apache Spark was designed as a computing platform to be fast, general-purpose, and easy to use. It extends the MapReduce model and takes it to a whole other level; it keeps everything in the memory as long as possible which decrease disk I/O operations and increase the performance of iterative algorithms, such as machine learning algorithms.
Apache Spark handles fault tolerance using Resilient Distributed Datasets (RDD) by recompute the data from the previous stage rather than recompute all jobs. Spark is even faster than MapReduce for complex application on disk.

2.7.1 Components of a Spark job

- *Application:* the Spark job JAR which contains our Spark application and its Dependencies. It can be a single job, multiple jobs chained together, or an Interactive Spark session.
- *Spark Driver:* Each Spark application can have only one driver which handles the Spark execution context and converts the application into a directed graph of tasks.
- *Spark application master:* Each Spark application can have only one Spark application master, which manage the resource allocation. When we run Spark on Hadoop via YARN, the Spark application negotiates with YARN resource manager for resources allocation on the cluster.
- *Spark Executor:* here where the Spark application tasks get executed, an executor can run multiple tasks on a single JVM instance, and multiple executors can run in a single node in a cluster, Spark executors get their instructions form Spark drivers about what tasks they need to run.
- *Spark task:* For each part of the Resilient Distributed dataset (RDD), the Spark executor launches a task to perform the required operations on that part.

Figure 2.9 shows the Spark components involved in the execution of a Spark application.

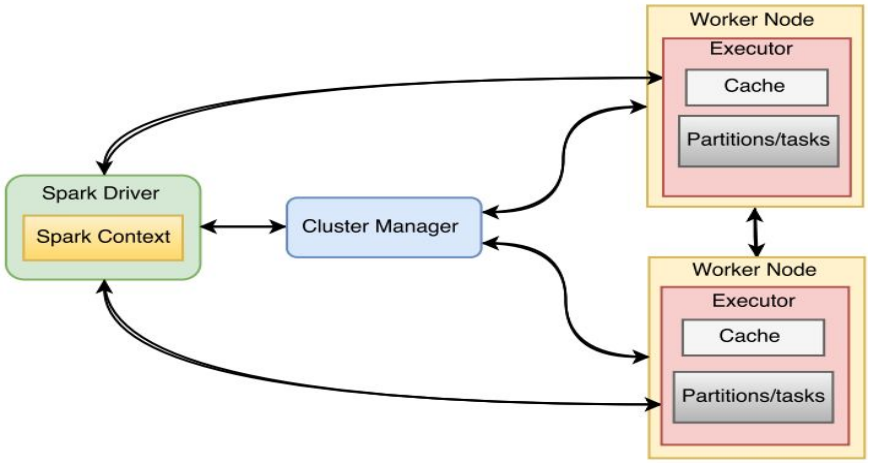


Figure 2.9 Apache Spark components

2.8 Deep Learning with GPU vs CPU

Wikipedia defines GPU and CPU as follows:
“A GPU is a chip traditionally designed and specialized for rendering images, animations and videos for the computer’s screen.GPUs have many cores, sometimes up to 1000 cores, so they can handle many computations, and also they are good at fetching large amounts of memory”.
“A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.“
Deep Learning networks, during their training process, perform a massive amount of matrix and vector multiplications. To speed up these multiplications we need a device which can perform these multiplications in parallel. GPU was designed to handle this kind of matrix operations in parallel, and it performs better than CPUs in term of matrix operations. Consider the case in which we have following two matrices, and we need to multiply them:

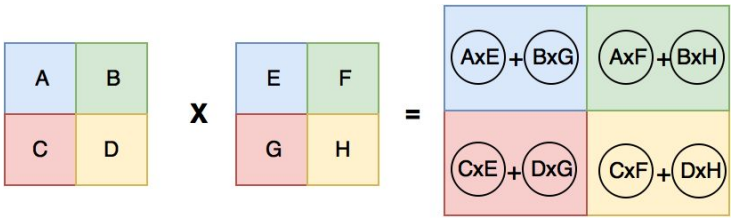


Figure 2.10: two matrix multiplication

In Figure 2.10, we have two matrices of size 2x2 and therefore to multiply them, we must perform eight multiplication operations and four addition operations.
A single core CPU takes matrix operations in serial, one element at a time, if we assume each operation takes one CPU clock cycle, then a CPU needs 12 clock cycles to perform this matrices multiplication. However a single GPU could have hundreds or thousands of cores, and it performs matrices multiplication in parallel. With GPUs, we can perform all mathematical operation in Figure 2.10 within two clock cycles. So since Deep Learning networks during the training process do this mathematical operation many times, we can send these operations to the GPU and after the GPU performs all operations, it will save the results in main memory where a CPU can read them.

3 ANALYSIS

Training deep learning networks is an iterative process, performs a massive amount of computation (forward-propagation and backpropagation).
This section illustrates why these massive computations occur in deep learning networks and how we can configure an environment to handle and accelerate these computations.

We will start by analyzing our training data in order to understand its structure, and explore how we can convert raw data to machine learning formats.

3.1 Analyze Training Data

The performance of a neural network is typically related to the training dataset. We will start analyzing our problem by analyzing our data. the given data is 78 GB file in BSON format.
Wikipedia defines BSON data format as follows:
"BSON is a computer data interchange format used mainly as a data storage and network transfer format in the MongoDB database. It is a binary form for representing simple data structures, associative arrays (called objects or documents in MongoDB), and various data types of specific interest to MongoDB. The name 'BSON' is based on the term JSON (JavaScript Object Notation) and stands for Binary JSON. a JSON is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types". A document such as {'hello':'world'} is stored in BSON format as follows:

```
Bson:
  \x16\x00\x00\x00                 // total document size
  \x02                             // 0x02 = type String
  hello\x00                        // field name
  \x06\x00\x00\x00world\x00        // field value (size of value, value,
                                   //null terminator)
  \x00                             // 0x00 = type EOO ('end of object')
```

Each record in BSON file is structured as follows:

Record
- |__id: an integer that represents the image id
- |__imgs
- | |__picture
- | | |__binary: a binary sequence which represents the picture
- | | |__type: an integer that represents the type of the picture
- |__category_id: an integer that represents the category id

The content of an instance of BSON file record looks as follows:

```
{_id=87,
imgs=[
```

```
        { "picture" :
          { "$binary" : "/9j//DRD3CiiitRBRRRQAUUUUAFFFF
                         FABRRRQAUUUUAc7q/HiGyPon/s1bit
                         /APZjWwr/ACiuFytUkaWukTkhhimOn
                         OthiM/7xqaoofufialrpp/CiXuFFFF
                         eOS4nhBMkDbio6lT1/ofwrJ//2Q==",
            "$type" : 0
          }
        }
     ],
category_id=1000010461}
```

Our targets in each BSON file record are:

- "_id": the image's id
- "$binary": a binary sequence represents an image
- "Category_id": id of the image's category

For each record we do the follows operations:

- Extracts all targets
- Create a directory and name it as 'Category_id'. All images which have the same 'Category_id' saved in the same directory
- Create an empty image file and name it as '_id' + '.jpg' in the corresponding directory
- Load the '$binary' into that image file.

After reading the BSON file, process each record, count number of categories, count number of images in each category, and group images by category name we obtain N numbers of categories, where each one contains M numbers of images. We simply group all images by their categories name into one directory. Figure 3.1 illustrates how the structure of created directories looks.

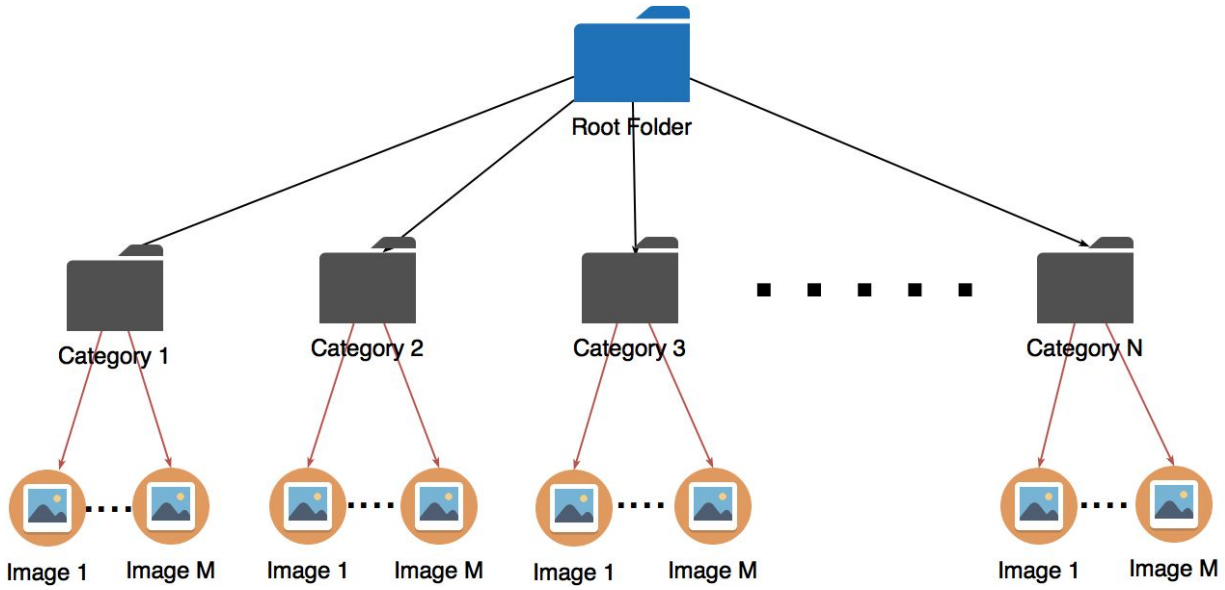


Figure 3.1: Grouping images by category name

Next, we will analyze the information about the training dataset using a set of non-standard Python libraries. Python is a great programming language to analyze a dataset and do some investigations on it, Follows are the libraries we will use to analyze the training dataset:

- *Jupyter*: jupyter.org defines 'jupyter library' as follows:

"The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more."

- *Pandas*: pypi.python.org defines 'Pandas Library' as follows:

"pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with 'relational' or 'labeled' data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language."

- *Numpy*: numpy.org defines 'numpy Library' as follows:

"is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays."

- *Matplotlib*: matplotlib.org defines 'matplotlib library' as follows:

"Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hard copy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook and web application servers"

Importing libraries into jupyter notebook.

```
Jupyter console 5.2.0

Python 3.5.2 (default, Nov 23 2017, 16:37:01)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import pandas as pd
        import matplotlib.pyplot as plt
        import numpy as np
```

Read the "counts.txt" file which contains information about the training data, and create a DataFrame. Pandas.pydata.org defines 'DataFrame as follows:

"Pandas DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects."

```
In [2]: dataFrame = pd.read_csv("counts.txt", names=["category_id",
"number_of_images"])

In [3]: # Display first 5 rows
          dataFrame.head(5)
Out[3]:
   category_id  number_of_images
0   1000012700                13
1   1000012694                14
2   1000004490               253
3   1000004488               328
4   1000004482               566
```

Each row in the DataFrame contains two fields

- "Category_id": An integer represent the id of the category

- “Number_of_images”: An integer represent the number of images in that category

Count the number of categories and the total number of images in the data

```
In [4]: # Number of categories
        dataFrame["category_id"].count()
Out[4]: 5270

In [5]: # Total number of images
        dataFrame["number_of_images"].sum()
Out[5]: 7069896
```

Get a description about number of images in the categories.

```
In [6]: dataFrame["number_of_images"].describe()
Out[6]:
count     5270.000000
mean      1341.536243
std       4941.011223
min         12.000000
25%         69.000000
50%        200.000000
75%        718.500000
max      79640.000000
Name: number_of_images, dtype: float64
```

From the result of categories description, we see there is no balance between the number of images in each category. The smallest category contains 12 images, the biggest one contains 79640, and the average size of categories is 1341 images. We can visualize the unbalancing between categories to better understand the differences between these numbers and see a particular issues in the dataset.
Figure 3.2 describes the numbers of images in categories as a BarChart.

```
In [10]: #create a Barchart
         plt.figure(figsize=(20,10))
         plt.bar(categories, array_num_of_images, align='center')
```

```
        plt.ylabel('Number of images',  fontsize=20)
        plt.xlabel('categories',  fontsize=20)
        plt.title('number of images in each category',  fontsize=20)
        plt.show()
```

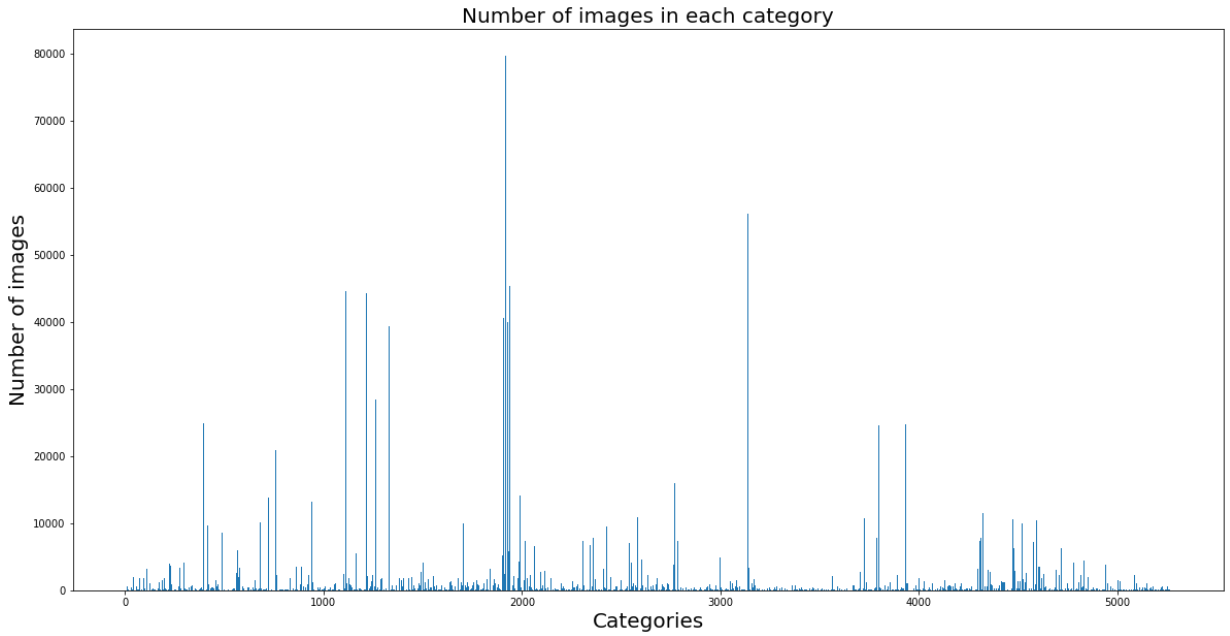


Figure 3.2 Each line represents the number of images in one category

Sort categories by number of images and visualize them.

```
In [11]: array_num_of_images  =sorted(dataFrame_np[:,1])
In [12]: plt.figure(figsize=(20,10))
         plt.bar(categories, array_num_of_images, align='center')
         plt.ylabel('Number of images', fontsize=20)
         plt.xlabel('categories', fontsize=20)
         plt.title('number of images in each category', fontsize=20)
         plt.show()
```

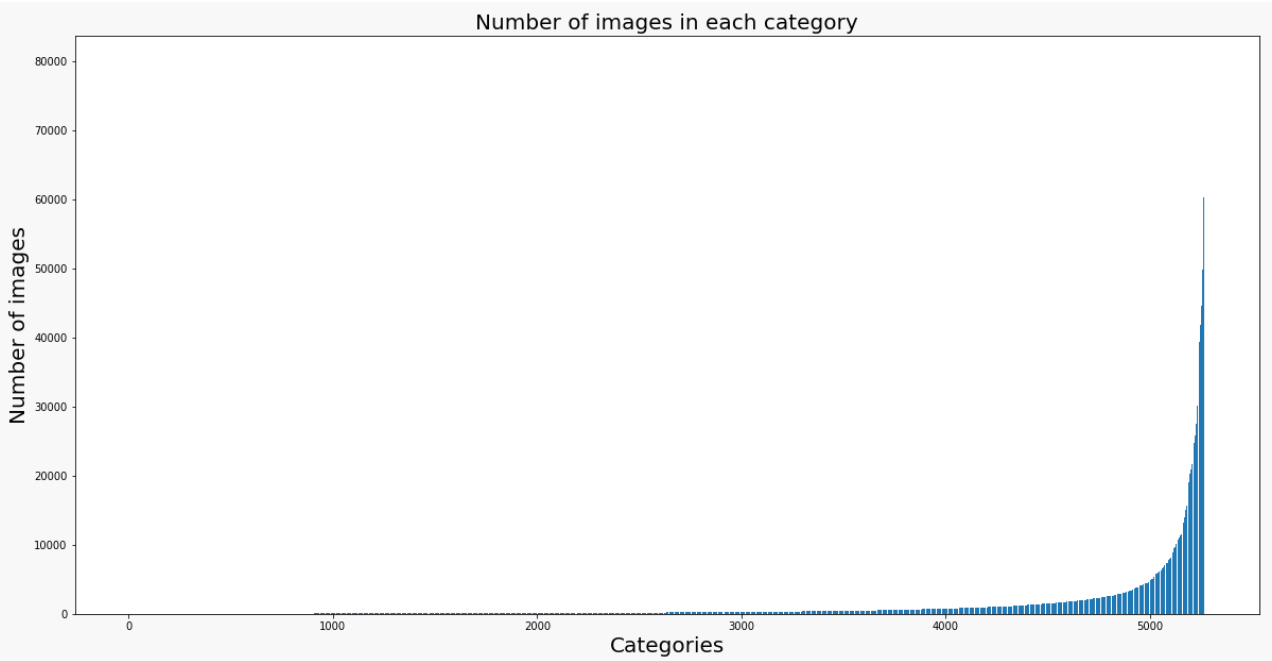


Figure 3.2 number of images categories

In Figure 3.2, we easily see the difference between the size of the categories. Most of them contain a small number of images, and a small portion of them contain a big number of images. Such unbalance between the size of categories can have a significant effect on a deep learning network performance trained with this dataset. The deep learning network will learn more about categories which have a big quantity of images, and less about the other categories.

Partitioning the dataset into parts according to the size of categories can give us a better view to make some decisions about setting some level of balancing across all categories. The following code creates a pie chart, which represents the percentage of each partition.

```
In [17]: from collections import defaultdict
In [18]: group_categories = defaultdict(int)
In [20]: # Group categories, the difference between each group is 20000
images
          for i in array_num_of_images:
              group_categories[(i // 20000) * 20000] +=1
In [21]:  category_in_groups =np.array([ group_categories [i] for i in
groups ])
In [22]:  plt.figure(figsize=(10,5))
```

```
        labels = ["0 - 20k", "20k - 40k", "40k - 60k", "60k -80k"]
        explode = (0, 1.2, 3, 4.9)
        plt.pie(category_in_groups, explode=explode, labels=labels,
                autopct='%1.1f%%',  startangle=0)
    plt.axis('equal')
    plt.show()
```

Figure 3.3 Pie chart representing categories in partitions

Figure 3.3, represents the total number of categories as follows:

- 98.6% of categories have the number of images between 0 and 20,000
- 0.9% of categories have the number of images between 20,000 and 40,000
- 0.3% of categories have the number of images between 40,000 and 60,000
- 0.2% of categories have the number of images between 60,000 and 80,000

The first partition will have a negative effect when we train a deep learning network with this dataset. To find a solution to reduce that effect, first, we will do more analysis on that partition.

Number of categories in the first partitions is 5195

```
In [23]: group_20000 = dataFrame[(dataFrame["number_of_images"] < 20000)]
In [25]: group_20000.count()
Out[25]:
category_id         5195
```

```
number_of_images    5195
dtype: int64
```

Number of categories which have number of images less the 1000 is 4234

```
In [26]: group_20000[(group_20000["number_of_images"] <1000)].count()
Out[26]:
category_id         4234
number_of_images    4234
dtype: int64
```

The total number of categories is 5270, and the number of categories which have a number of images less than 1000 is 4234. As we can see, almost 80% of the categories contains a tiny amount of images (less than 1000). We need to find a way to deal with these categories. We have the following options:

- Discard categories which have a number of images less than 1000.
- Generate more images from the existing images to increase the number of images in these categories.

3.2 Convolutional Neural Network

The size of each image in our training dataset is 180x180 pixels. Can we rely on a simple hidden layer to process these relatively large images?
Consider that we have a simple hidden layer with 3 neurons, and the input data to this layer is a colored image with size 180x180 pixels. If we convert the input image to a vector, we get a vector of size 180x180x3 = 97200. The total parameters in that hidden layer would be 97200x3 = 291600 ('size of vector' x 'number of neurons'), and this number is far too large to estimate their values and for this reason, we cannot rely only on a simple hidden layer in image processing. We need a way to reduce the number of parameters by reducing the size of the image without losing features in that image. This is where Convolutional Neural Networks (CNN) become very useful.

Wikipedia defines the Convolutional Neural Network(CNN) as follows:
"CNN is one of the commonly used deep learning algorithms. CNN achieves the object recognition by partitioning a single big image into a number of smaller

images (tiles), each tile can be processed independently, Based on the processing results of all tiles, the algorithm can predict the object from the original image"

What can a convolutional neural network do? For a computer to recognize an object in an image, the computer first needs a mechanism to learn higher-order features in the image. With a CNN, a computer transforms an image into different forms, extracts features from these forms, and learns about these features. This makes CNNs an excellent learning mechanism for computer vision and some other machine learning problems.

3.2.1 Convolution:

Convolution is a combined integration of two functions, and shows how one function modify the shape of another function.

The convolution function in math: $$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

Convolution can be used to represent many types of image operations, a digital filter in terms of signal processing, or to create new features.

A **convolution kernel (feature detector)** is a filter to detect features in the input image. We use a convolution kernel to filter the input image looking for a specific feature and reduce the size of the input image without losing important features. Put simply, convolution kernels allow deep learning networks to focus only on the features that are important in the input image.

Richard M. Reese in his book 'Machine learning for java developers' defines some traditional Feature detector as follows:

- "Edge detection: finds boundaries of objects within an image Corner detection identifies intersections of two edges or other interesting points, such as line endings, curvature maxima/minima, and so on.
- Blob detection: identifies regions that differ in a property, such as brightness or color, compared to its surrounding regions.
- Ridge detection: identifies additional interesting points at the image using smooth functions.
- Hough transform: identifies particular patterns in the image".

To explain how the convolution function uses feature detection to extract features from an image fed into a CNN, consider we have an input image with size 7x7

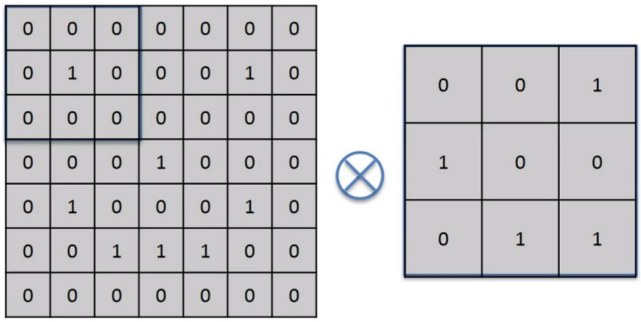
pixels (each pixel in this image becomes a feature after being multiplied by the color channel), and we have a feature detector with size 3x3. The left matrix in Figure 3.4 represents the input image, and the right matrix represents the feature detector.

0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

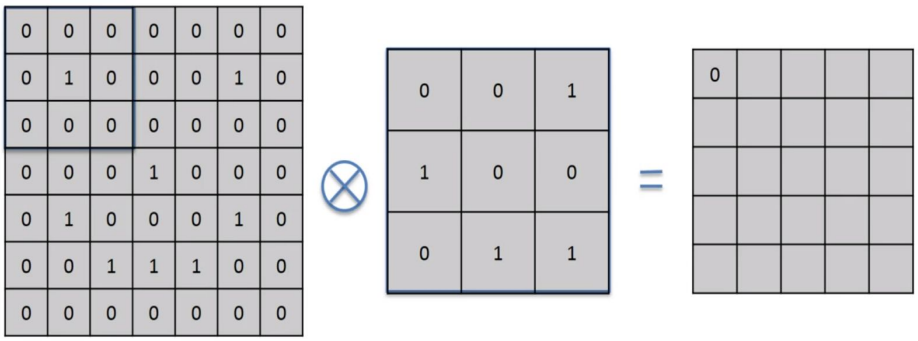
0	0	1
1	0	0
0	1	1

Figure 3.4 : Input Image 7x7 px *Convolution kernel (feature detector) 3x3 px*

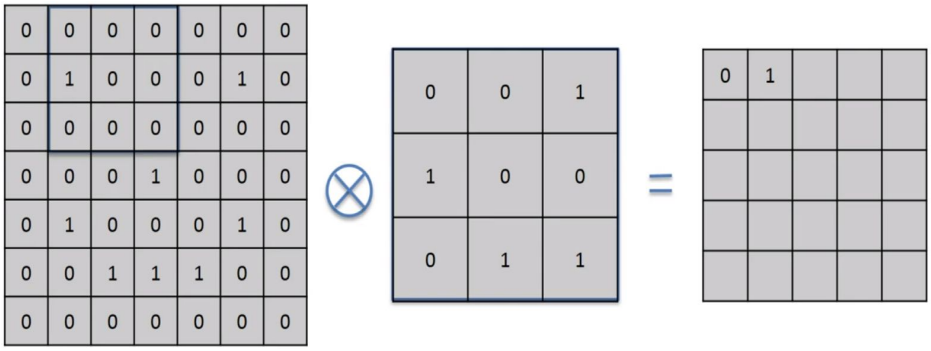
We apply the kernel to the input image by overlapping the kernel on top of the image and performing multiplication across the numbers at the same location, and sum up the result of these multiplications to get a single number. For example, we overlap the kernel with the top left region of the input image



The convolution result at that location is :
0 x 0 + 0 x 0 + 0 x 1 + 0 x 1 + 1 x 0 + 0 x 0 + 0 x 0 + 0 x 1 + 0 X 1 = 0



Then we move the kernel to the right by one pixel, and we calculate the next convolution result:



0 x 0 + 0 x 0 + 0 x 1 + 1 x 1 + 0 x 0 + 0 x 0 + 0 x 0 + 0 x 1 + 0 X 1 = 1

We keep moving the kernel to the right by one pixel and calculate the convolution for that location until the kernel reaches the top right corner of the image. Then we return the kernel to the top left of the input image, and we move the kernel down by one pixel. We repeat the convolution for every possible pixel location until we reach the bottom right corner of the input image as illustrated in Figure 3.5. The result of applying a convolution kernel on the input image is called **Feature Map**.

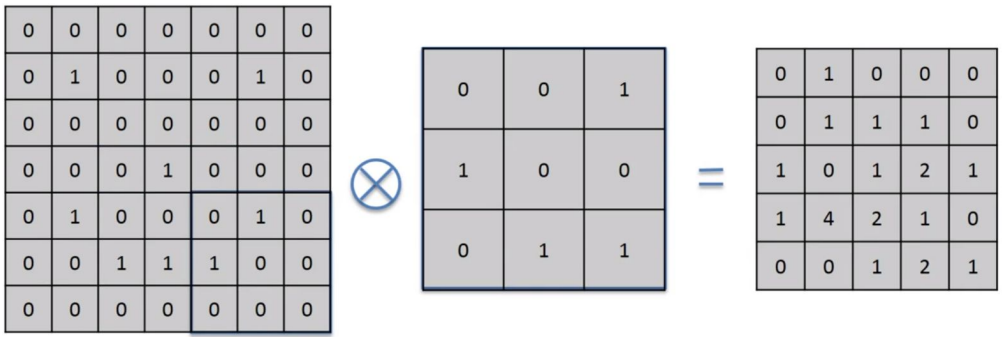


Figure 3.5 Input Image *Kernel (feature detector)* *Feature Map*

Figure 3.6 illustrates an example of applying an edge detector to an image.

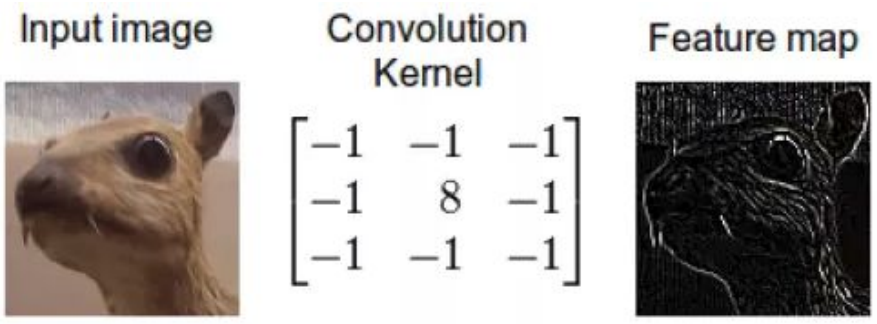


Figure 3.6: apply edge detector to an image

3.2.2 Convolutional Layer

The convolutional layer is where most of the massive computations happen, which cause a feature detector to determine if a given pattern occurs in the input image. The weight and bias of this layer effect how the feature detector searches for a specific pattern.

Neuron connection in a convolutional layer is different from neuron connection in a fully-connected hidden layer. To explain neuron connection in a convolutional layer, we have to use some imagination, due to the complexity of these connections. Imagine we have a wall which represents the image in Figure 3.7:

Figure 3.7: image of University College Cork

Also, imagine that we have a series of flashlights shining on the wall, which will create a group of overlapping circles as Figure 3.8 illustrates.

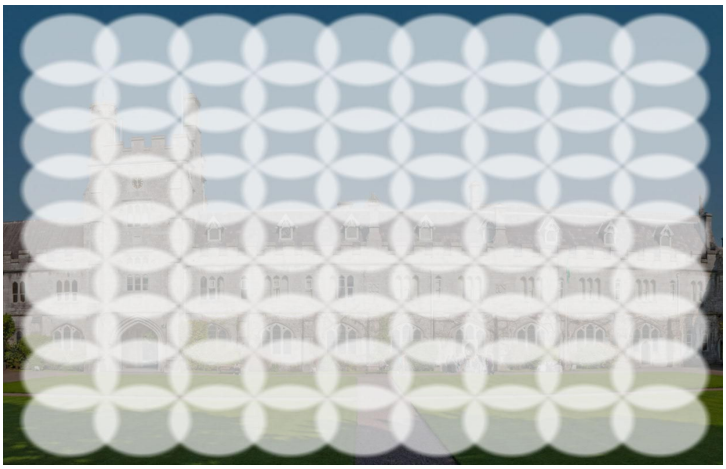


Figure 3.8: series of flashlights shining on an image of University College Cork

The goal of each flashlight is to search for the same specific pattern but each flashlight is searching for this pattern in a different part of the image. Combining their result is a result of a feature detector (filter) applied to that image, which determines whether a specific pattern occurs in the image and in which part of the image. In this example, we have a 9x9 grid of flashlights, which is all considered to be one feature detector of size 9x9 px. Looking to that image from the top, we can see how that series of the flashlights look like, as Figure 3.9 illustrates.

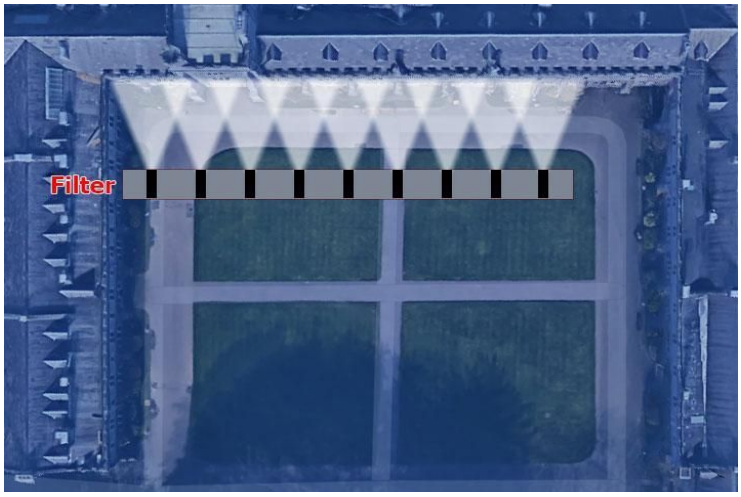


Figure 3.9: series of one flashlights

Now we can add more series of flashlights to detect more patterns. In the Figure 3.10, we have three series of flashlights, where the first flashlight in all series shines on the same location, the second flashlight in all series shines on the same location and so on.

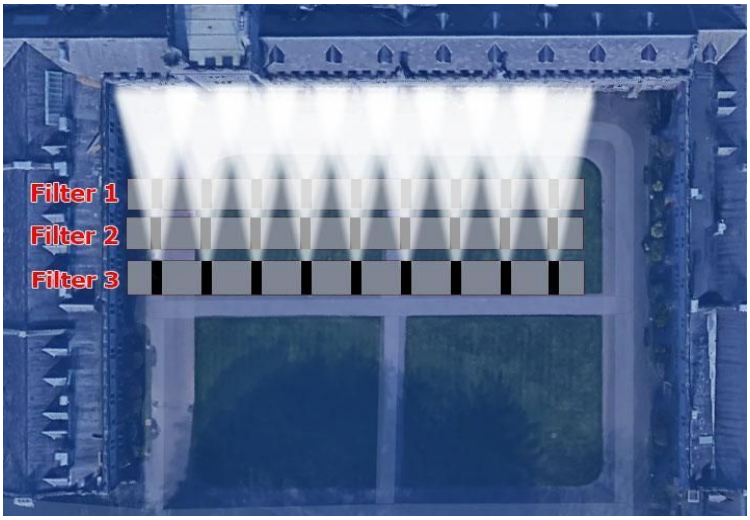


Figure 3.10: series of three flashlights

In Figure 3.10, we have three series of flashlights which represent three feature detectors all shining on that wall, and all are looking for different patterns in parallel. If we consider each flashlight as a neuron and all feature detectors (filters) are in one layer, then we have a convolutional layer of 9x9x3 (a 3-dimensional grid of neurons), where all neurons on each row map to the same part of an input image, as illustrated in Figure 3.11.

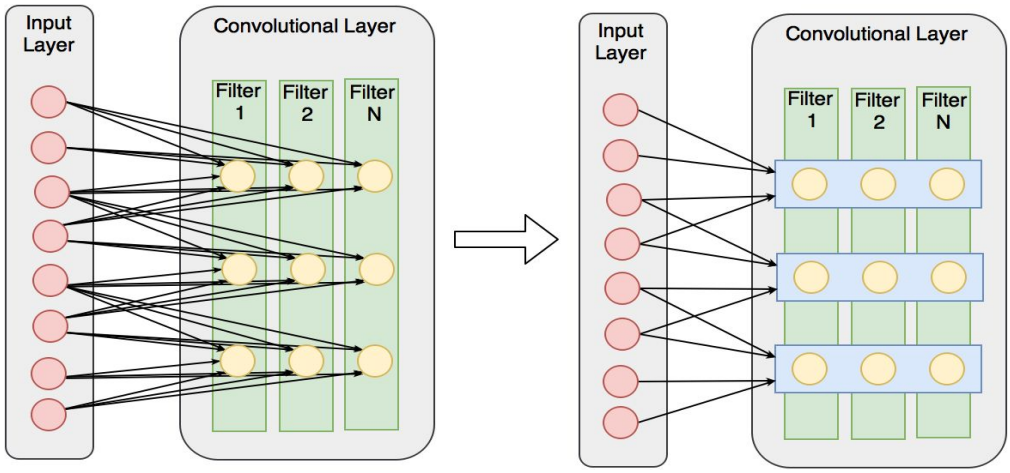


Figure 3.11 an Convolutional Layer with 3-dimensional grid of neuron

In other words, each neuron is only connected to the input neurons it shines on. Figure 3.12 shows how one row of neurons in a convolutional layer connects to a particular section of the input image.

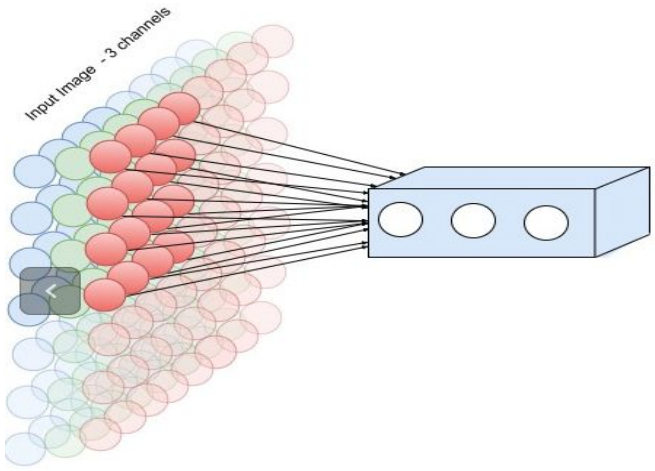


Figure 3.12: one row of neurons in convolutional layer mapped to particular section of the input image.

Since neurons in the same feature detector connect to the same number of input neurons, and they search for the same pattern, they must have the same weight and bias parameters, which allow a feature detector to scan the input image in parallel by looking for a specific pattern in different sections of that image. Figure 3.13 shows how a feature detector searches for a particular pattern in parallel.

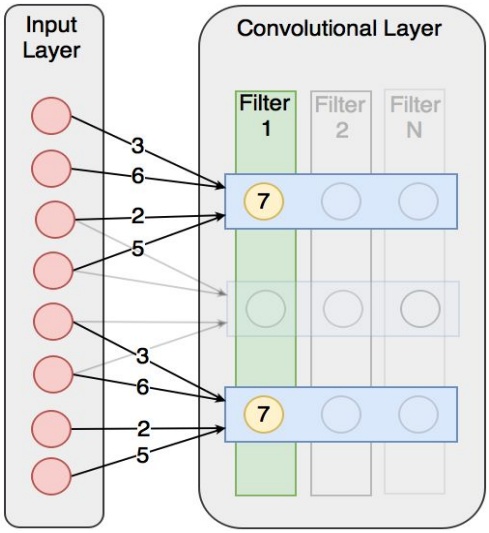


Figure 3.13: two neurons in a feature detector searching for the same pattern in parallel

The outputs of a convolutional layer has to be controlled by an activation function to ensure that the output value is in a specific range. Next we will explain why we need an activation function in CNNs.

3.2.3 Nonlinear Activation

A formula of a linear activation function can be as following:
F(x) = x, this function simply outputs whatever was input.
If we use that linear activation function in convolutional layers, the network will output a linear function of the input.

Normally, the transmission between pixels in an image is highly nonlinear, specially when there are different objects in the image. However, when we apply convolution kernels to an image this might actually cause linearity in transmission between pixels. For example, the transmission between pixels after applying convolution kernels could be from White to Grey to Black, a linear transmission.
The convolutional layer needs a nonlinear activation function To ensure there is no linearity in each neuron's output. Simply put, an activation function helps to build up the simple patterns discovered by the convolutional layer. For each node in a convolutional layer, we need an activation function, for this, each node in a convolutional layer connected to one node containing an activation function. The Figure 3.14 shows the connection between nodes in a convolutional layer and activation function nodes.

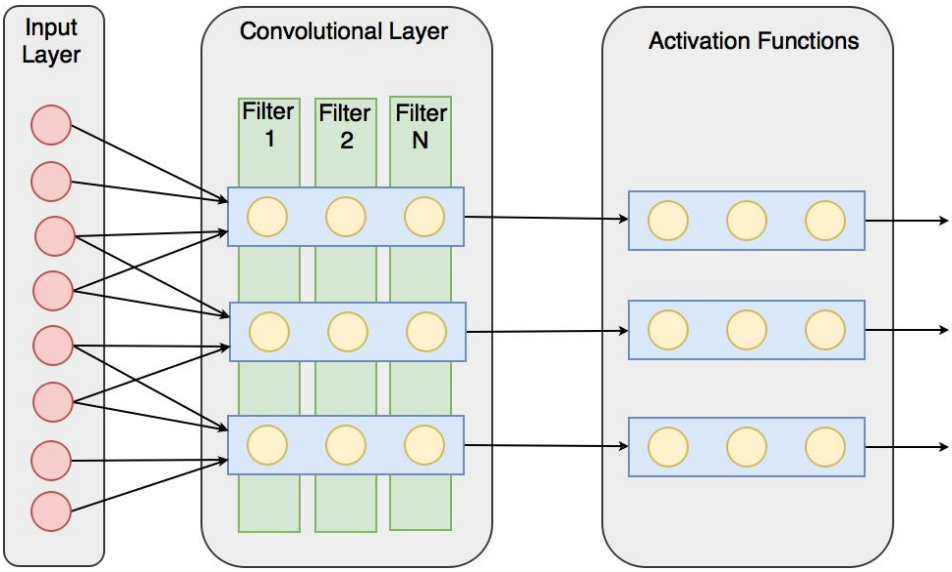


Figure 3.14: connection between a Convolutional layer and activation functions

There is a range of functions which provide nonlinearity, such as Sigmoid Function, Rectifier Function and the Hyperbolic Tangent function as illustrated in section 2.4.1.2 “Activation functions.”

Which Activation function can add better performance to the convolutional neural network? The sigmoid function goes between zero and one which means the average of this function is 0.5. Deep learning networks prefer the average of any neuron's activations to be equal to zero with a standard deviation of 1 which makes learning for the next layer easier. The Hyperbolic Tangent (tanh) function is a shifted version of Sigmoid Function, and it goes between -1 and +1, which means the average the of tanh function's output is equal to zero and is making the tanh function more suitable for neuron's in deep learning networks. However, if the input of the tanh function is very large or very small, then the gradient of this function becomes very close to zero which can slow down gradient descent and prevent the values of weights from changing. This issue is known as vanishing gradient.

Rectifier Function is the most popular activation function used in deep learning networks. The formula of this function is : ReLU(x) = max(x, 0).

- If x is positive, then the gradient is equal to 1
- If x is negative, then the gradient is equal to 0

With choosing Rectifier Function as an activation function in a deep learning network, the gradient mostly will be far from zero which increase the performance of the learning process.
The Figure 3.15 illustrates where a Rectifier Function takes its place in CNN’s architecture.

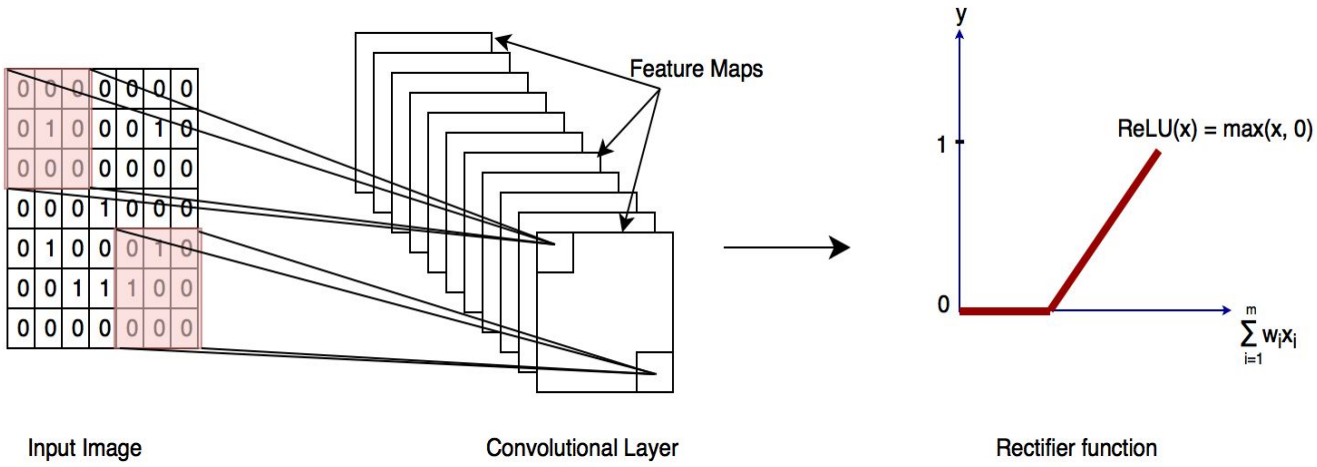


Figure 3.15 convolutional layer passes its output to Rectifier activation function

We have analyzed how a convolutional layer finds features in specific locations in the input image. However, a CNN model should only learn about the most relevant features discovered by a convolutional layer regardless to their location. For example the objects in the Figure 3.16 contains images of same object where the feature's location, color, brightness in each image is different.

Figure 3.16: in each image the feature of the same object are in different location.

A dimension reduction is applied to discovered features to obtain the most relevant features. This operation is called a pooling layer in a CNN's architecture. Another advantage of pooling layer is to reduce the number of parameters and the computation in CNN and control overfitting.

3.2.4 Pooling Layer

In CNN architecture, multiple instances of a convolutional layer and activation layers tile together in a sequence to build more complex patterns. The problem with this is that the number of possible patterns becomes exceedingly large. To ensure the network only focuses on the most relevant patterns discovered so far a pooling layer is used for dimensionality reduction by reducing the resolution of the feature maps without losing the information related to the most relevant features. Each pooled feature map corresponds to one feature map of the previous layer. The pooling layer operates on every slice (typically the size of the slice is 2x2 pixels) of feature map in parallel to resize that slice. There are several pooling operations, such as mean, sum and max operation. For example consider we have a 4x4 pixel feature map as represented as in the Figure 3.17, where the numbers in circles represent relevant features:

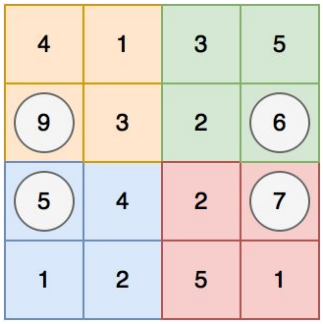


Figure 3.17: A feature map contains four relevant features

Each color in Figure 3.17 represents a slice 2x2 of pixels of that feature map. A max operation is applied to each slice which keeps only the maximum value in that slice (a relevant feature). Figure 3.18 illustrates the steps involved in applying a max operation to the feature map in Figure 3.17:

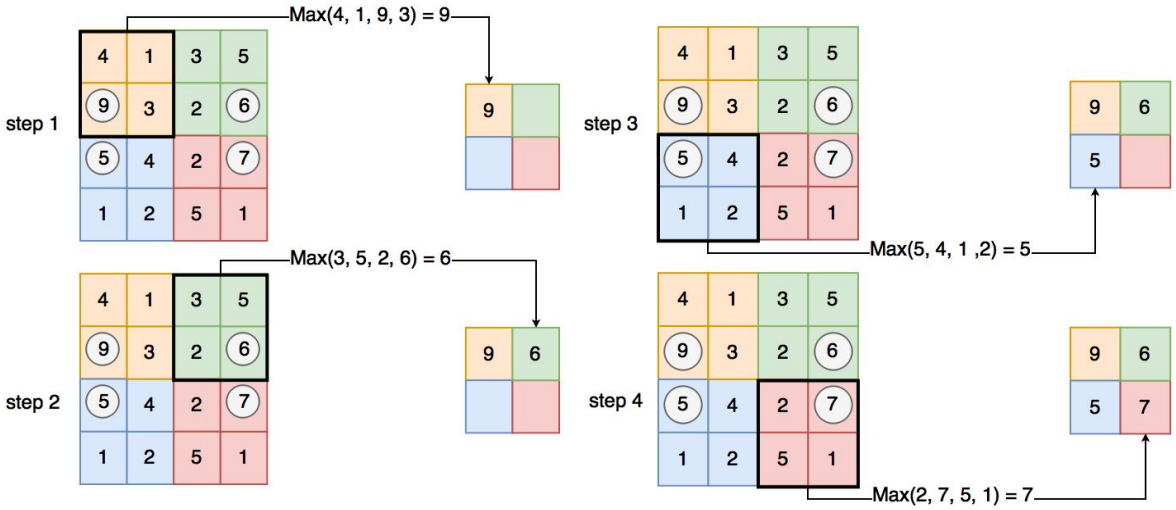


Figure 3.18 steps involved in applying a max operation on a feature map

With convolutional layers, activation layers and pooling layers, CNNs can discover a range of complex patterns in the input data. However, CNNs have no understanding of what these patterns mean. To make a CNN capable of understanding what these patterns mean, a series of a fully connected layers is added to CNN after pooling layer.

3.2.5 Fully Connected Layer

A fully connected layer is like a regular hidden layer, but neurons in a fully connected layer have full connections to all activations in the previous layer. This layer makes CNN capable of understanding the outputs of the pooling layers and

classifying these outputs. This layers only accept inputs as a vector of numbers, for this reason, the output of the pooling layer is converted to one vector of numbers. The Figure 3.19 shows how a flattening operation takes its place to convert the output of a pooling layer to a vector and pass it to a fully connected layer.

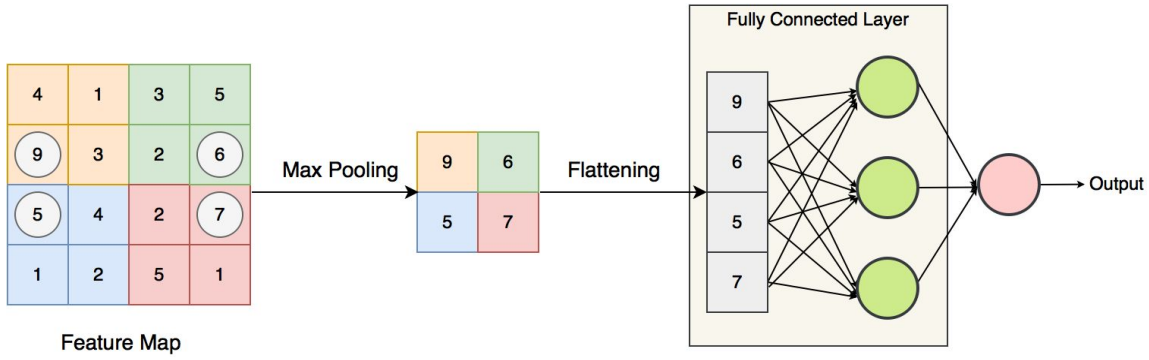


Figure 3.19 : flattening the output of a pooling layer

3.2.6 Learning Process

Section 2.4.1.1 explains how forward-propagation works to predict a value ŷ, and how backpropagation allows a deep learning network to move the cost information from the end of the network to all parameters inside the network.

When a deep learning network is initialized all parameters are initialized with a random number close to zero. During the learning process a cost function such as 'mean squared error' is used to calculate the cost (error) i.e. the difference between the predicted value and the actual value. To minimize this cost, all parameters involved in the forward-propagation process must be adjusted. What approaches can we use to change these parameters?. One approaches to do that is a brute force approach. This approach tries all possible values for parameters to minimize the cost. Figure 3.20 shows the best value for a parameter to minimize the cost.

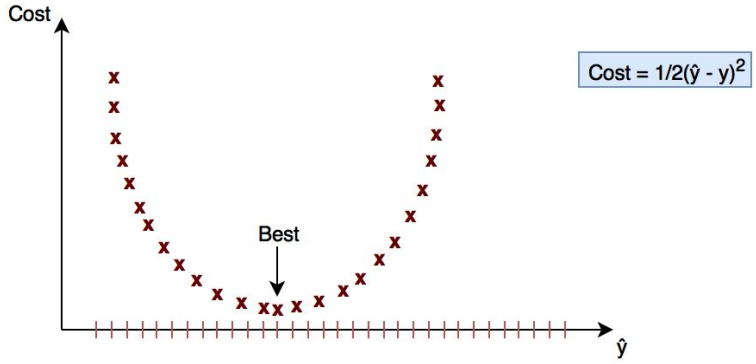


Figure 3.20: the best value for a parameter to minimize the cost

This approach looks good if a deep learning network has one or two parameters which need to be updated. However, a CNN model could have thousands of parameters and therefore if this simple approach was applied to a CNN model, it would face a Curse of Dimensionality. Wikipedia defines the Curse of Dimensionality as follows:

"The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience".

To explain why a deep learning network would face the curse of dimensionality if we use brute force approaches to change the parameters in that network, imagine we have a deep learning network as illustrated in the Figure 3.21.

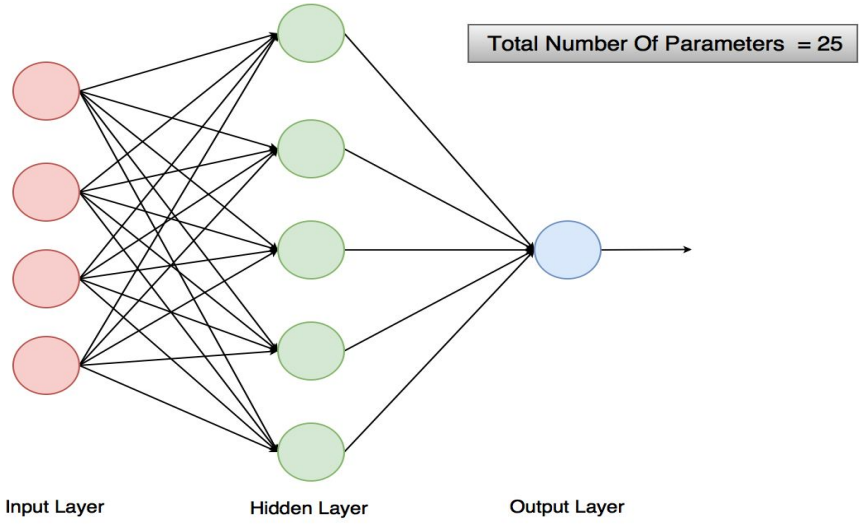


Figure 3.21, a neural network with 25 parameters

In the Figure 3.21, the neural network has 25 parameters. If we consider the number of possibility for each parameter is 1000, then we have in total $1000^{25} = 10^{75}$ Combinations. This number is so large that even with supercomputers it would take a million of years to find the best combination which will minimize the cost.

Another approach to minimize the cost is Gradient Descent which is an iterative optimization algorithm to find the minimum of a function. The gradient descent process starts with initializing all parameters with random values close to zero, and repeatedly do the following operations after every iteration over the training data until it finds the minimum cost:

- Calculates the cost function, then finds the gradient (slope) of the cost

- If the gradient is negative, then decrease the parameters involved in forward-propagation process
- If the gradient is positive, then increase the parameters in forward-propagation

Figure 3.22 shows how gradient descent would find the minimum value for a cost function:

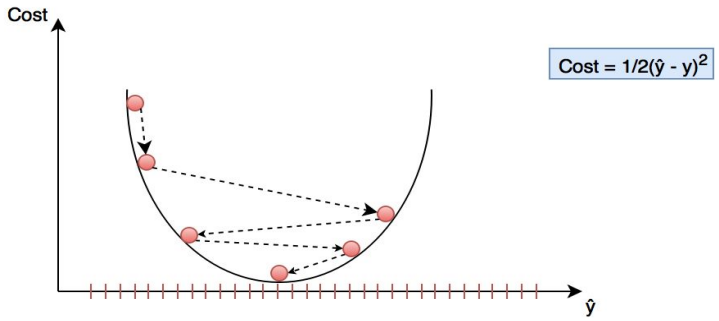


Figure 3.22: gradient descent to find the minimum of a function

The Gradient Descent approach might work very well to minimize a convex cost function with only one minimum point. However, in multidimensional space, a convex cost function can turn into a non-convex function. For example, the function in Figure 3.23 represents a non-convex cost function which has two local minimum points and one global minimum point.

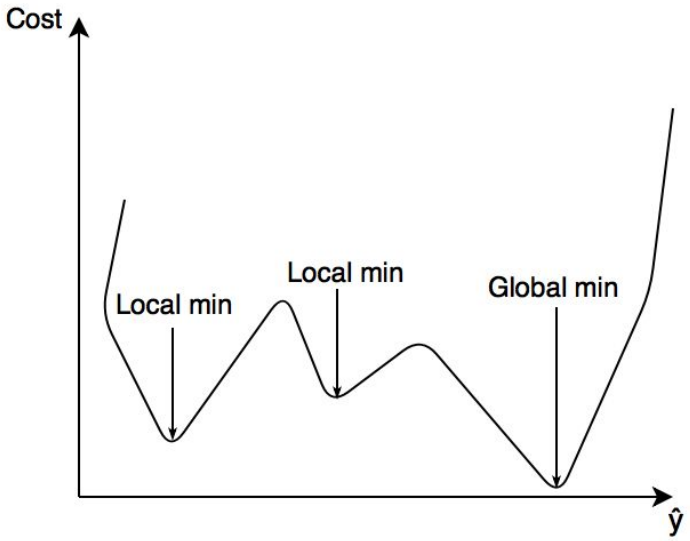


Figure 3.23: a non convex cost function

Using Gradient Descent to find the minimum cost in the Figure 3.23 may only find a local minimum cost and the algorithm could become stuck there. We want gradient descent to reach the global minimum cost and not get stuck in a local minimum cost. A better approach is Stochastic Gradient Descent (SGD). In SGD, we find the gradient and update the parameters after each example in the training

data. This can be a slow process. However, it guarantees the minimum cost will be pushed as close as possible to the global minimum cost. There are many other approaches we can use to minimize the cost function, such as AdaGard, Adam, AdaDelta, and BFGS.

3.3 Benefits of Apache Spark

Apache Spark is a general parallel-processing engine that can execute on its cluster, on a Hadoop cluster via Hadoop YARN (Yet Another Resource Negotiator), or on Apache Mesos cluster. Spark reads data from disk only once and uses some techniques around caching the frequently used data in memory with Resilient Distributed Datasets (RDD). RDD is an immutable fault tolerant collection of elements. Since the frequently used data is cached in memory as RDD, Spark can execute parallel iterative algorithms on that data with high performance. Spark stores the output of every operation on an RDD in a distributed memory where other analytics can be executed on stored output. Figure 3.24 illustrates the logical representation of executing an iterative job on Spark.

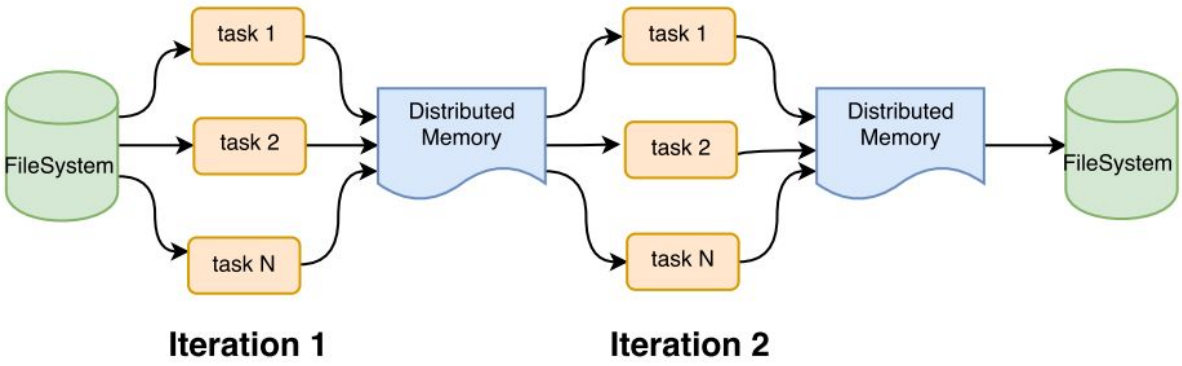


Figure 3.24: The execution of an iterative job on Spark, after each iteration results are saved in distributed memory.

Before Spark reads data from disk to creates an RDD, it first creates a directed acyclic graph (DAG) of all transformation operations we perform on that RDD, such as filtering operation, mapping each element in RDD to a particular function, or grouping the elements by a specific property.
After Spark creates a DAG, it waits for an action operation to be performed on that RDD. When an action operation is performed on RDD, the execution of DAG starts and it creates the RDD in memory. For this reason, RDD is considered as lazy evaluation. The following are the benefits of Lazy Evaluation:

- Fault tolerance: Spark start creating the DAG steps until it finds a step that asks it to perform an action on the RDD, each step has a pointer points to the previous step, and this makes RDDs fault tolerant since Spark knows the steps to recreate any step in RDD. If any part of the RDD fails, instead of rereading all data from disk and repeat the whole process, Spark will use the DAG to go to the previous step and recreate only the failed part.
- Optimizing resource usage: as Spark knows all the steps to be performed to create an RDD, Spark can merge some steps to a single step which reduce the resource usage.

There are three methods to create an RDD:

- Create an RDD from the data already resides within Spark such as an array of elements.
- Create an RDD from a dataset which can come from any storage source supported by Hadoop such as Hadoop Distributed File System (HDFS), HBase, Cassandra or Amazon S3.
- Create an RDD by performing a transformation operation on an existing RDD.

An RDD is parallelized; each process (one CPU core) in distributed system will receive one partition.

3.4 Spark Resource Allocation

Spark can run locally on a single machine with a single JVM (Java Virtual Machine). However, more often Spark is used to process data stored in a distributed storage system such as HDFS, HBase, Cassandra or Amazon S3. Figure 3.25, illustrates Spark's location in an data processing ecosystem.

A cluster manager manages the execution of a Spark application across the cluster. When a Spark application is submitted to a cluster, the cluster manager will allocate required resources (memory, CPU, GPU) to each Spark component involved in the execution of that application. Spark currently supports three kinds of cluster managers: Standalone cluster manager, Apache Hadoop YARN and Apache Mesos.

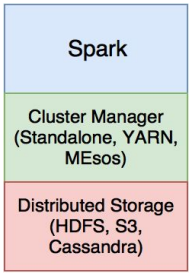


Figure 3.25, Data processing ecosystem including Spark

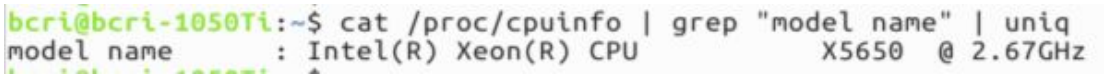
A Spark application consists of the Spark driver which is where the high-level Spark logic is written. The Spark program runs in the Spark driver and sends instructions (tasks) to spark executors. Each Spark executor has its own Java Virtual machine (JVM) to execute all received tasks. A Spark application can contain a single or multiple Spark jobs defined by one Spark Context in the Spark driver. A Spark Context contains Spark configurations which specify how the spark environment should be configured to execute a given job.

Analyzing a Spark job to estimate the required resources is the first step which must be performed before submitting that job. How much memory and how many cores can we allocate to a Spark job to obtain the best execution performance?
To push the speed of a Spark job execution to the maximum, a set of Spark configurations must be defined. These configurations depend on the size, steps, and kind of the job. A Spark job that cache a lot of data and perform iterative computation has a different configuration than those that contain a few very large shuffles. Typically, configuring a Spark job is as much an art as a science.

Gathering information about a cluster is essential to understand the GPU and CPU properties and the amount of available memory in the cluster which will host our Spark environment.
Depending on available resources in the cluster we can find out the limit on each Spark resource request. In our cluster we have eight machines as follows:

- One master node: This node is responsible for allocating resource to executors and send instructions and data to them and receives set of information from executors regard to the execution process. The amount of communication between this node and workers nodes can be very high. We need to ensure there will be enough cores on this node to respond to any communications and execute tasks related to BLAS libraries and Spark driver.

CPU Model Name in the master node:



Number of processors in the master node:

```
bcri@bcri-1050Ti:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU           X5650  @ 2.67GHz
Stepping:              2
CPU MHz:               2659.918
BogoMIPS:              5319.83
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
```

The master node has 24 cores, 2 thread per core

Amount of Memory in the master node:

```
bcri@bcri-1050Ti:~$ cat /proc/meminfo | grep MemTotal
MemTotal:       24665828 kB
```

In total we have 24 cores and 24 GB of memory for the master node.

- Seven workers (slaves):these nodes will host executors, the resource in these node are the same.

CPU Model Name in workers node:

```
bcri@bcri-1050ti-bm-1:~$ cat /proc/cpuinfo | grep "model name" |uniq
model name      : Intel(R) Core(TM)2 Duo CPU     E8400  @ 3.00GHz
```

Number of processors in each worker node:

```
bcri@bcri-1050ti-bm-1:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 23
Model name:            Intel(R) Core(TM)2 Duo CPU     E8400  @ 3.00GHz
Stepping:              10
CPU MHz:               1998.000
CPU max MHz:           3000.0000
CPU min MHz:           1998.0000
BogoMIPS:              5984.77
Virtualisation:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              6144K
NUMA node0 CPU(s):     0,1
```

Each worker node has two cores, one thread per core

Amount of Memory in each worker node:

```
bcri@bcri-1050ti-bm-1:~$ cat /proc/meminfo | grep MemTotal
MemTotal:        8092764 kB
```

GPU device in each worker node :

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.111                 Driver Version: 384.111                  |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 105...  Off  | 00000000:01:00.0 Off |                  N/A |
| 45%   27C    P8   ERR! /  75W |     10MiB /  4037MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

For each worker node we have 2 cores, 8GB memory and one GPU with 4GB memory.
In total we have the following available resources for executors:

- 14 cores
- 56GB memory (only 47GB available for Spark environment)
- 7 GPUs with total Memory of 28GB. Each GPU can run round 700 threads in parallel which mean total number of numerical computation threads can be run on GPUs is around 4900 threads.

3.5 Distributed Deep Learning

Training a CNN model in a distributed system is a hard task to achieve. An efficient mechanism must be found to train that model with multiple computers. How can multiple computers train one CNN model in parallel? Consider the following scenario:
We have one CNN model M, training data D, and a cluster with one master node and seven worker nodes.
In order to use all available resources in the cluster and perform distributed deep learning in an efficient way, a copy of the model M must be sent to each worker node in the cluster. Then each worker node trains its local model M_{copy} on a part of the data D. After every iteration over the data D, each computer sends its parameters to a master computer which hosts the model M. The master computer calculates the average of all received parameters and updates the parameters in the main model M, then it sends these updates to each worker node in the cluster and each worker node updates all parameters in its local model M_{copy}. Figure 3.26 illustrates how each worker node shares the parameters from its local model M_{copy} with other nodes in the cluster.

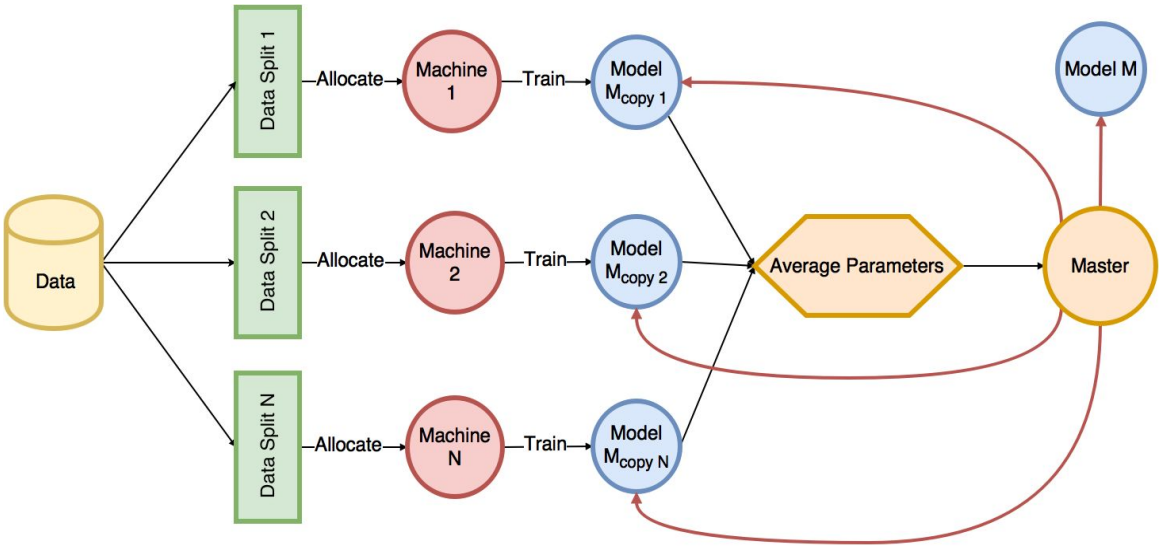


Figure 3.26 a mechanism to train a deep learning model in a distributed System

In Figure 3.26, each machine sends its local model parameters to the master machine after training its local model with its allocated data splits. However, if the size of an input data is huge and the number of computers in the cluster is small, then each split of the data will be large.
Our training data size is 78GB, and we have seven workers in our Spark cluster. Each worker will launch a task to process 11.1GB of the training data and send its model's parameters to the master after training its models with 11.1 GB of data. This process will reduce the I/O operations. However the model's accuracy will suffer since each worker will share its model's parameters after processing a big chunk of the data. An efficient solution here is to let each worker share its model's parameter after processing N examples of the training data.

To reduce the communication between worker nodes and master node without affecting the CNN model's training performance, the distributed deep learning mechanism should follow the following process:

- Initialize all parameters randomly based on the model's configurations.
- Each worker node receives a copy of the model form master node.
- Split the training data into M partitions where M is the number of CPU cores in the cluster.
- Distribute partitions across the cluster, where each core in the cluster is assigned to handle one partition.
- In each worker node, split each partition into K splits (mini-batches), where each split will contain N examples.
- In the worker nodes, for each mini-batch, launch a task (thread).
- In all worker nodes, after processing each task, send parameters to the master node.
- When the master receives parameters from all other nodes in the cluster, find the average parameters and update all models in the cluster.

Figure 3.27 illustrates how a mini-batch approach is used in a distributed deep learning mechanism to perform efficient distributed deep learning.

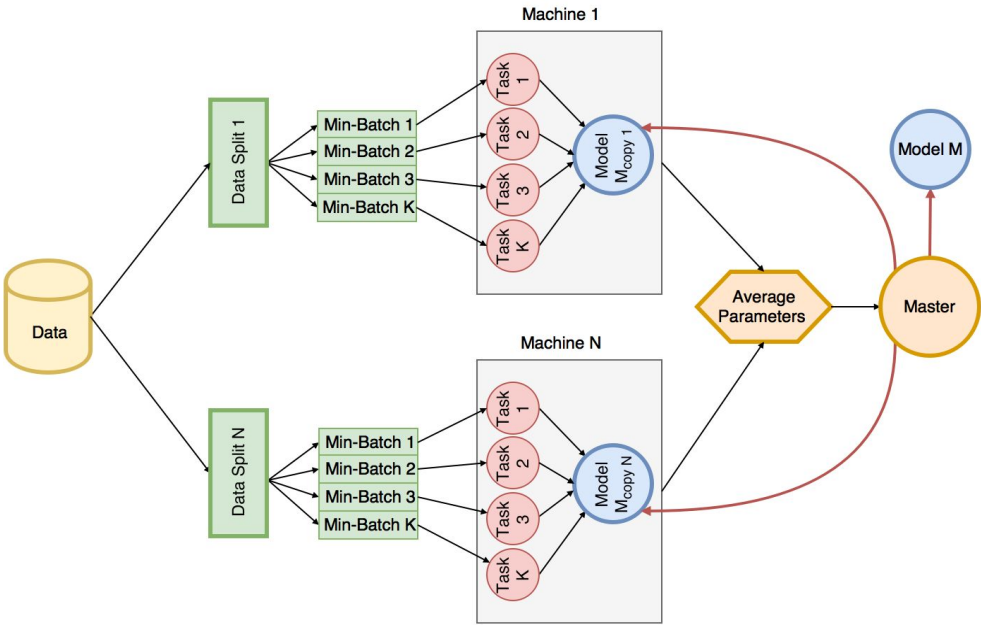


Figure 3.27: using mini-batch approach to improve a distributed deep learning performance

In Figure 3.27, the Average Parameters component receives parameters from worker nodes after each mini-batch is processed which increase the I/O operations between worker nodes and the Average Parameters component. To reduce I/O operations, worker nodes can be configured to send their parameters to the master node after processing 5 to 10 mini-batches. Figure 3.28 shows how the average parameters are calculated in a system with three worker nodes, after processing five min-batches.

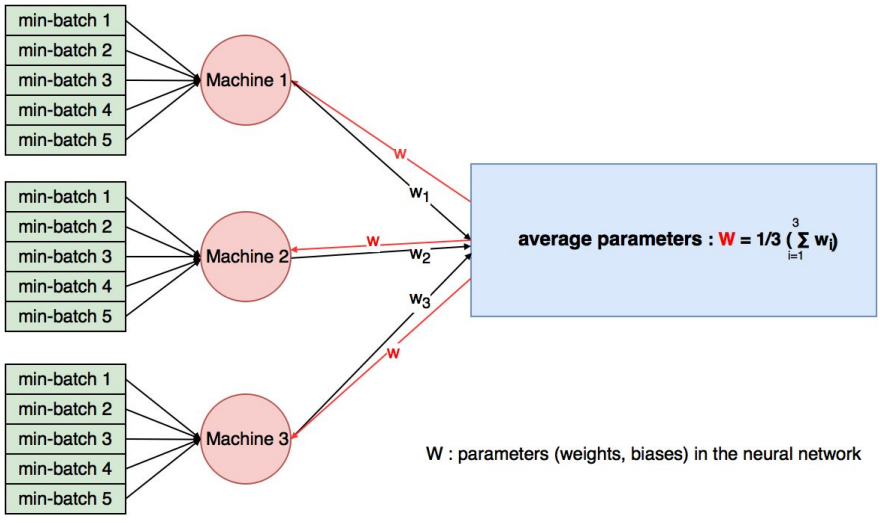


Figure 3.28: sending parameters to a parameter server after each machine process 5 mini-batches

For each Average Parameters process, all machines will wait to receive new parameters back from the master node, and this waiting stage will affect the execution performance.

3.6 Distributed training data

In a distributed deep learning the training data must be distributed efficiently to ensure there is always sufficient data on each machine to be processed. In a Spark environment, the master node is responsible for sending the data to the executors. Executors cannot execute any task until all required data has arrived and been stored in their memory.

Training a CNN model with 78GB of data in a Spark environment using GPUs can become a prolonged process if GPUs do not get enough data to process.

A direct approach is one of the approaches for moving the data to executors. In the direct approach, each time a Spark application runs, it creates a training RDD and a testing RDD, and for every iteration over the training data it repartitions the training RDD to N partitions (N is the number of cores in the cluster) and distributes the partitions across the cluster. This approach might work well to distribute a small training data across a cluster. However, if the training data is enormous, then this approach has some disadvantages which are outlined below.

- Each time a spark application runs it creates fresh RDDs from the same training data. This process could be an expensive process especially if a developer needs to run a Spark application multiple times to estimate some hyperparameters for the running spark application. For example, running a Spark deep learning application to process 10 GB of images requires 5 hours to create one RDD from 10GB of images on a computer with 24 cores. If a developer needs to run that Spark application 10 times to estimate hyperparameter values for the running application, then 10x5 = 50 hours is required solely to create RDDs.
- A master node distributes RDD partitions across a cluster. Each worker node in the cluster stays idle until all required data has arrived. Here the worker nodes' performance depends on the master node's performance when distributing the RDD partitions. For example, consider we have seven

RDD partitions, each of which is 1000MB, seven worker nodes, which can each process 100MB of data in 10 seconds, and a master node which can transform 100MB of data per second. The master node sends each RDD partition to each worker node. The time required for the master node to transform one RDD partition to one worker node is 1000/100 = 10 seconds. Which means that when the master node distributes RDD partitions, the first worker node receives the first RDD partition in 10 seconds, the second worker node receives the first RDD partition in 20 seconds and so on. The first worker node processes the received first RDD partition in 10 seconds and then stay idle for 80 seconds until it received the second RDD partition from the master node.

Given that each node's idle time in this example is roughly 90% using CPUs, we can see that the idle time will actually increase using GPUs as they will be able to process the same data in 2 to 5 seconds but must still wait the same amount of time to receive the next batch of data.

Another approach for distributing a training data is the 'export approach'. In this approach, when a Spark application runs for the first time, it creates the required RDDs, partitions them and stores them in a distributed file system in batched and serialized form. When the Spark application runs again it can skip the creation of RDD partitions and instead each executor can load the required RDD partitions directly from the distributed file system. The advantages of this approach are as follows:

- The master node use less memory and avoids RDDs creation and repartition overhead.
- Executors can load RDD partitions directly from a distributed file system to keep the GPUs busy.
- The time required to train a distributed deep learning model is reduced by doing less communication and more computation.

Figure 3.29 shows how the export approach can be used within a distributed deep learning application.

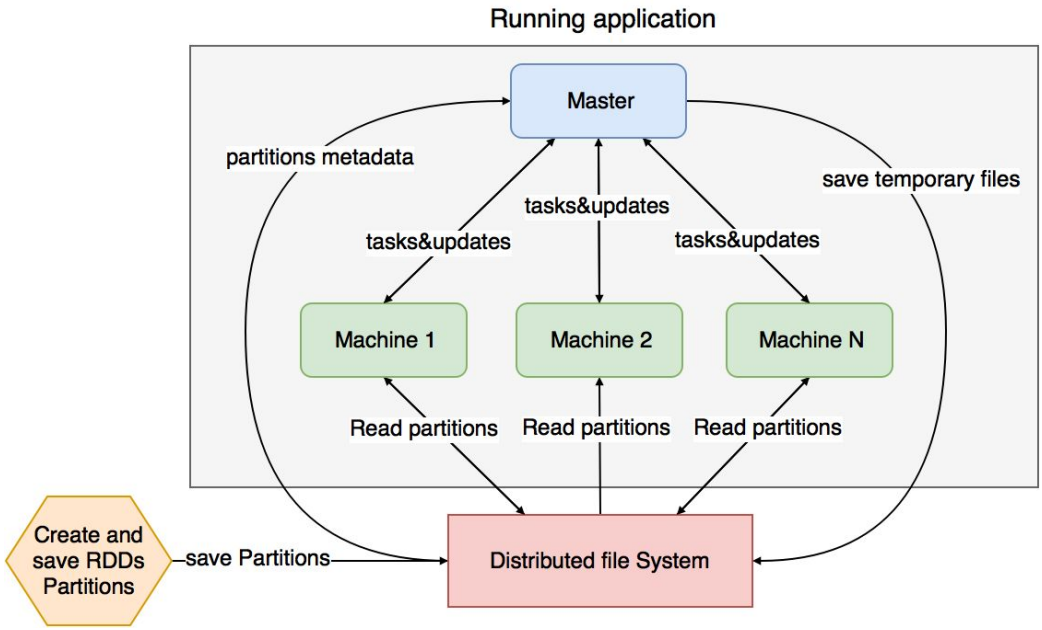


Figure 3.29, running a distributed deep learning application with distributed training data using the export approach

3.7 Benefits of GPUs in Deep Learning

The execution performance of a deep learning algorithm on a Spark environment is limited by the available cores and memory in that environment. Spark launches and executes all tasks in parallel and most of these tasks require massive linear algebra calculations. Moreover, these calculations slow down the whole execution process. Some advance linear algebra libraries such as OpenBLAS, Atlas, and Intel Math Kernel Library (MKL) are used to speed up these calculations. However, these libraries' threads need their own allocation of cores and memory to perform the required calculations. For example, if we have a computer with four cores, and we have four deep learning tasks and each task generates ten linear algebra threads, than the total required linear algebra threads would be 4x10 = 40.

Moreover, the total of all threads needed to be executed concurrently on 4 cores is 44 threads (4 tasks + 40 linear algebra threads). If these linear algebra threads could be executed somewhere outside of these 4 cores, then the execution of the whole process becomes much faster, and this is what general-purpose computing on graphics processing units (GPGPU) offers.

GPGPU is the use of GPUs to perform the computation in an application traditionally handled by CPUs. In GPGPU, the sequential parts of the application are run on CPUs and GPUs accelerate the computationally-intensive parts. Figure 3.30 illustrated how GPUs and CPUs work together to perform a task.

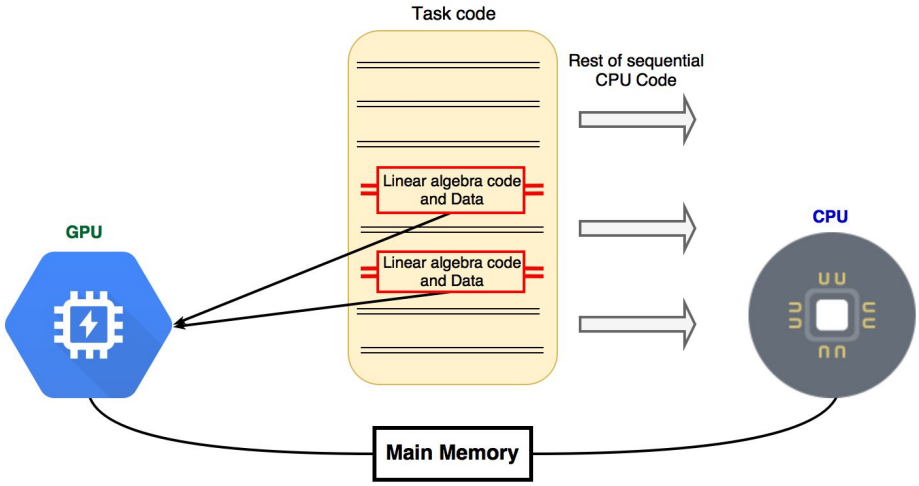


Figure 3.30: executing a task using GPUs and CPUs

Figure 3.31 shows the steps to perform a numerical operation on GPUs using GPGPU.

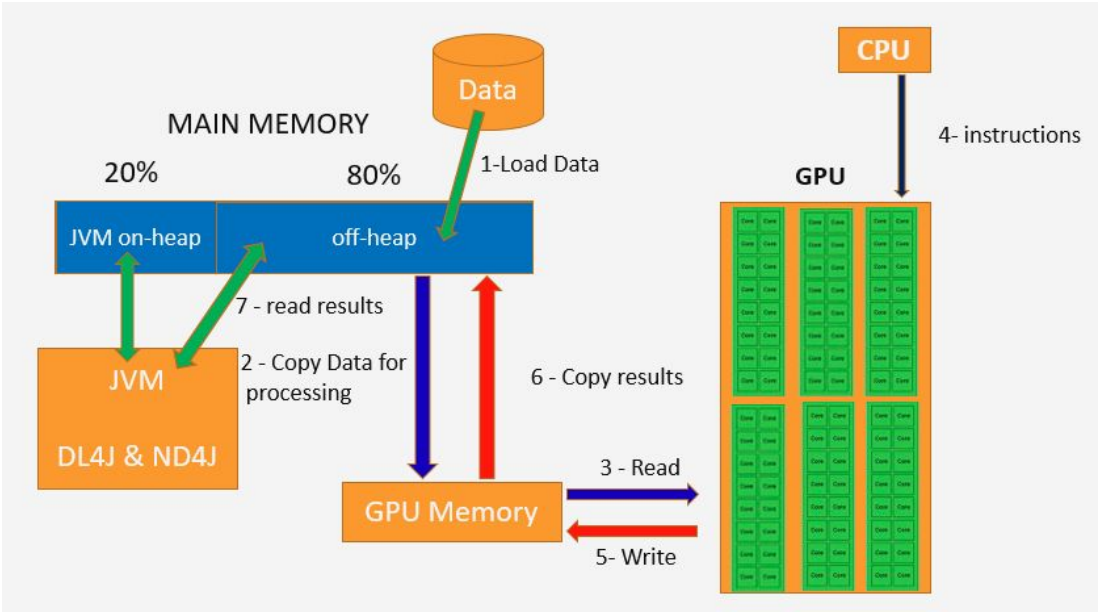


Figure 3.31: Steps involved in a GPGPU operation

Training a deep learning networks is an iterative process and a CNN has many parameters which should get updated within each iteration; it involve heavy computations that are mostly linear algebra operations (matrix multiplication, computing the inverse of a matrix, Gradient calculation). A matrix multiplication requires a large amount of computing power, especially when the numbers in the matrix are floating points. For example, assume we have a convolutional neural network with 16 layers with around 140 million weights and biases. To pass an image through this network would require around 140 million multiplications. So for each image in our dataset, each iteration it will require around 140 million multiplications, we can imagine now how much computation we will need to do, especially when we have big dataset and many iterations. With the power of GPUs we can fetch input images as matrices from main memory as a big chunk, and do matrix multiplication in parallel and at the same time by running the same code on different sections of the same array(row or column), where CPUs fails to do that. Why CPU cannot?. CPUs are designed for executing instructions in sequential way, CPUs are good to fetch small amount data and instructions from main memory and execute these instructions on that data, and they run tasks sequentially, even if there are multiple cores in them. For this reason we can consider CPUs are not fast enough for operations on big chunks of data they are not suitable for high parallelism. We can use a GPU together with a CPU to accelerate the execution of a training task, by sending matrices multiplications threads from that task to GPU, while the remainder of the code still runs on the CPU. after GPU execute all required operation on received data, it copies the result of these operation back to host main memory The following diagram shows how GPU and CPUs can work together:

3.8 Moving data to GPU

Data transfer between GPU and CPU can affect the performance of a GPU program. GPUs are fast in processing data and to keep them busy, they always should have enough tasks and data in their memory to be processed.

By default the CPU data allocations are pageable, and GPU cannot access data directly from pageable memory. To move the data from main memory to GPU, the Cuda driver first allocate a temporary array called 'Pinned' in main memory, copy the data from pageable memory to the Pinned, and then transfers the data from Pinned to GPU memory via PCIe bus.
Nathan Edwards defines PCIe as follows:
“ A PCIe connection consists of one or more data-transmission lanes, connected serially. Each lane consists of two pairs of wires, one for receiving and one for

transmitting. You can have one, four, eight, or sixteen lanes in a single consumer PCIe slot--denoted as x1, x4, x8, or x16. Each lane is an independent connection between the PCI controller and the expansion card, and bandwidth scales linearly, so an eight-lane connection will have twice the bandwidth of a four-lane connection. This helps avoid bottlenecks between the CPU and the graphics card." The PCIe bandwidth limits the speed of transforming the data from Pinned to GPU.

Our spark cluster has seven identical worker nodes each hosting one GeForce GTX 1050 Ti GPU device. To find how much data could transform between GPU and CPU in each worker node, 'NVIDIA BandWidthTest' tool can be used to find the size of the transfer size, Pageable transfers, and Pinned Transfers between GPU and CPU as illustrated in Figure 3.32:

```
bcri@bcri-1050ti-bm-7:~/code-samples-master/series/cuda-cpp/optimize-data-transfers$ nvprof ./bandwidthtest
==22228== NVPROF is profiling process 22228, command: ./bandwidthtest

Device: GeForce GTX 1050 Ti
Transfer size (MB): 16

Pageable transfers
  Host to Device bandwidth (GB/s): 2.077614
  Device to Host bandwidth (GB/s): 1.506272

Pinned transfers
  Host to Device bandwidth (GB/s): 2.646863
  Device to Host bandwidth (GB/s): 1.664977
```

Figure 3.32: Using NVIDIA BandWidthTest tool to find how much data could transform between GPU and CPU in each worker node.

In Figure 3.31, the output of NVIDIA BandWidthTest tool shows the limitations in each worker node for moving data between GPU and CPU are as follows:

- The transfer size is 16MB per second.
- Pinned Transfers: the bandwidth for pinned transfers from CPU to GPU is 2GB per second, from GPU to GPU is 1.6GB per second.

In our cluster each GPU device has 4GB memory, This means each worker node can fill up its GPU's memory with a required data in 2 second if and only if the required data is ready to be transferred. How quick can the required data be ready for transforming from CPU to GPU? Since RDD partitions are managed to be created and stored in a distributed file system each worker node which host a GPU device can directly load the required data from the distributed file system to its main memory which ensure all worker would always have enough data in their memory ready to be transferred to GPU.

3.9 Hardware accelerator

A hardware accelerator is computer hardware used to perform some functions more efficiently than is possible in software running on a CPU.
There are many hardware accelerators can be used for accelerating deep learning application such as NVIDIA, AMD and TensorFlow Processing Units (TPUs). The most popular one is Nvidia which is one of the primary vendors of GPU. NVIDIA has high-level language, known as CUDA which provide a robust API to implement programs for NVIDIA's GPUs. NVIDIA's GPUs have different features and architects and their performance highly dependent on their memory size. For deep learning applications, the GPU memory size is significantly essential to fetch high dimensional data.
Each worker node in our Spark environment hosts a 'Nvidia GeForce GTX 1050 Ti' with a memory capacity of 4GB to handle to handle the required computation for training a CNN model.

3.10 Memory issues

When a job is submitted to a Spark cluster via Spark submit, a set of specific configurations required to allocate resource to spark executors, and spark driver such as the amount of memory and number of cores. These configurations are significant to push the job's execution performance to the maximum. Spark is all about the memory. Efficiently managing the memory makes Spark capable to successfully executes a job with the excellent performance and zero out of memory errors.

Deep learning network needs memory for matrices or vectors operations. How much memory these operations require?. It depends on the size of matrix or vector.For example a colored image with size 32x32 pixels has 3072 floating point number and each floating point number is represented by 4 bytes, so in total this image requires ~12 KB memory plus memory required for labels. If the size of the matrix is significant, then operating on that matrix inside a JVM may use all the memory allocated to that JVM. Increasing the memory size for the JVM may address this issue. However, to accelerate operation on a matrix with some c++ code from an advance BLAS library such as OpenBLAS, MKL or cuBLAS that

matrix should be moved to outside JVM because a c++ code cannot execute inside JVM.

4. DESIGN

This section illustrates how to design a cluster that meets the needs and expectations for accelerating neural networks using GPUs. The newly designed cluster is planned to reduce the execution time compared to using CPU alone by a factor of 2 hundred.

The cluster hosts Apache Spark computing environment for accelerating convolutional neural network application built with the Deeplearning4j library based on Graphical Processing Units (GPUs) .

4.1 Data Preparation Pipeline(DPP)

A training dataset required to train a CNN model that can automatically classify products from a set of images.
The given training data is a real-world dataset, ~75GB. Real-world data is often inconsistent, incomplete, or lacking in specific behaviors and is expected to contain many errors. Data preprocessing is a method of prepare raw data for further processing.
The given training dataset go through multiple data preprocessing steps which prepare and transform the training dataset into an understandable format for CNN's algorithms running in Spark environment using GPUs. These steps are as follows:

- Read the training dataset from disk, extract images, apply required specification on each image and save them into a disk.
- Split data into two parts, one part for training and the other part for testing.
- Create RDDs for training the network and testing the network.
- Partition RDDs and distribute them across the cluster.

Figure 4.1 shows the components of the Data Preparation Pipeline (DPP) and how the data moves through this pipeline:

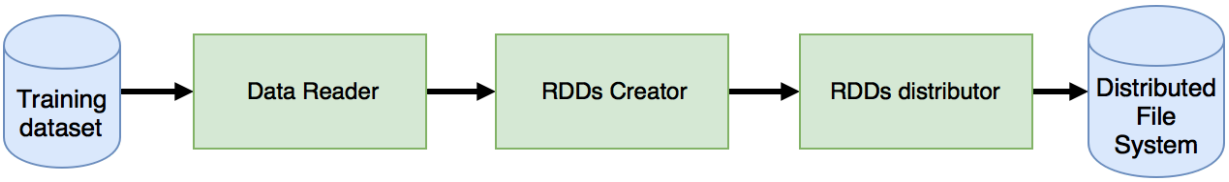


Figure 4.1: components of Data Preparation Pipeline(DPP).

4.1.1 Data Reader

The given training dataset is ~75GB file in BSON format which contains around 7 million of records, each record has the following attributes:

- The record Id
- The image in this record
- name of the category

An instance of Java InputBufferedStream is used to move chunks of this data into memory, 'BSON' library to decode and extract required attributes from each record, and Java FileOutputStream instance to store extracted images to a disk. The Figure 4.2 illustrates how the first component of DPP reads and handles each record in the training dataset.

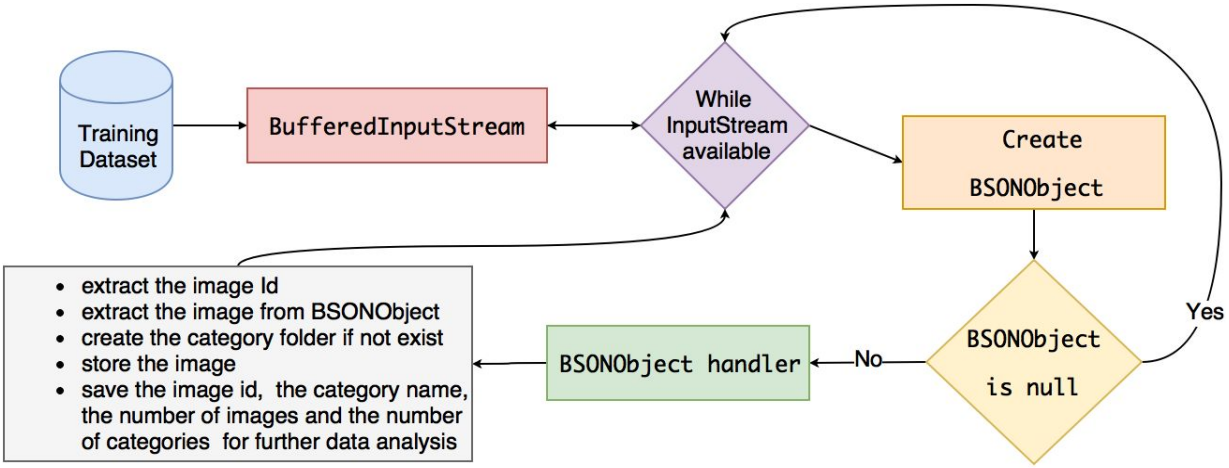


Figure 4.2: Data Reader component to Reading and handling each record in training dataset

Data Reader component creates a root directory which contains a folder for each category containing all images related to it as illustrated in figure 4.3.

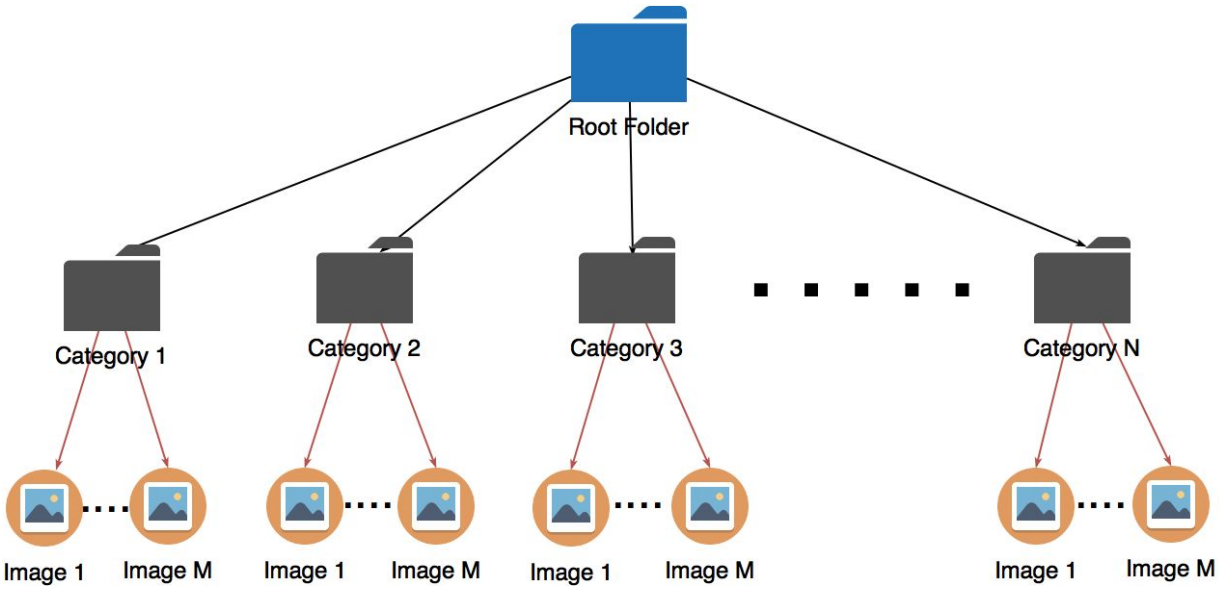
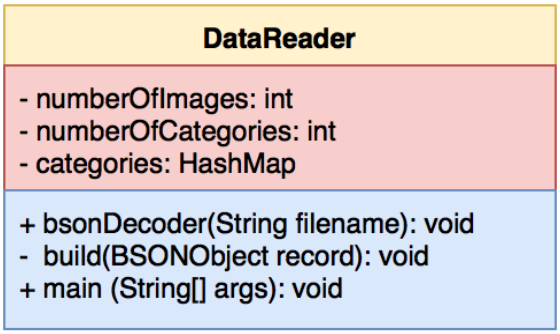


Figure 4.3: Structure of the root file created by Data Reader component.

The UML for the Data Reader class is :



4.1.2 RDDs Creator

Estimating Hyperparameters for a CNN model requires running that model multiple with a different combination of Hyperparameters. Traditionally each time the model runs in Spark environment it would go through the following steps to create required RDDs:

- Read the categories from a distributed file system
- Split data into two parts: the training and the testing datasets

- Convert datasets into a format acceptable by the model
- Create RDDs from the datasets

Minimizing RDDs creation overhead would reduce the time required to train a CNN model in spark environment. RDDs Creator component creates required RDDs from training and testing datasets only once and stores them in Hadoop Distributed File System (HDFS) in a serialized format. When the CNN model runs in Spark environment, each Spark worker node loads serialized RDDs objects directly from HDFS.

The neural network algorithms require the input data to be in a specific form (matrix or vector). For CNNs the input data must have the following specification:

- The format of all image must be acceptable by DL4J.
- Each image must be vectorized and represented as 3- dimensional array for colored images, or 2-dimensional array for grayscale images.
- Input data must be splittable to split the input data to training split and testing split.

DataVec library is used as a first step to preprocess the data into a format that DL4j can easily read. *Alex Black* in his book 'Deep Learning' defines DataVec library as follows:
"DataVec library is a library for handling machine learning data. DataVec handlesthe Extract, Transform, and Load (ETL) or vectorization component of a machine learning pipeline. The goal of DataVec is to simplify the preparation and loading of raw data into a format ready for use for machine learning. DataVec includes functionality for loading tabular (comma-separated values [CSV] files, etc.), image, and time-series datasets, both for single machine and Distributed (Apache Spark) applications."

Dl4j together with DataVec handle training data in an efficient way, the data is loaded only when required instead of loading all our data into memory at once. First we will split the data into two parts, one part for training and the other one for testing, then we will vectorize all images in each split, normalize them, and save them into two lists of DataSet (batch), each DataSet will contain 128 images. Next we will use *apache Spark library* to create two JavaRDDs from two lists of DataSets, serialize these JavaRDDs and save them in disk.

The following Figure shows the process of creating RDDS using DataVec library:

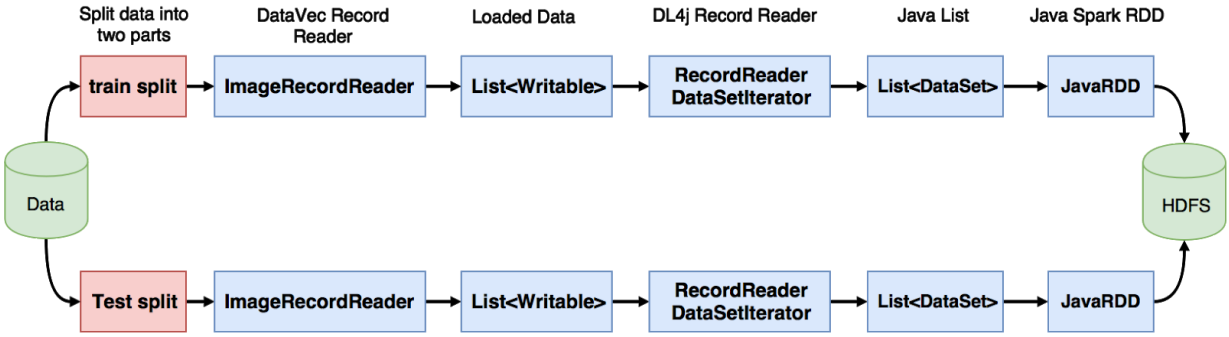


Figure 4.4: RDDs Creator component process to create RDDS using DataVec library

Java classes involved in RDDs Creator process are as follows:

- PreprocessData class: to preprocess the data.
- JavaRDDsCreator class: to create required RDDs.

Figure 4.5 shows the components for each these classes and the relation between them.

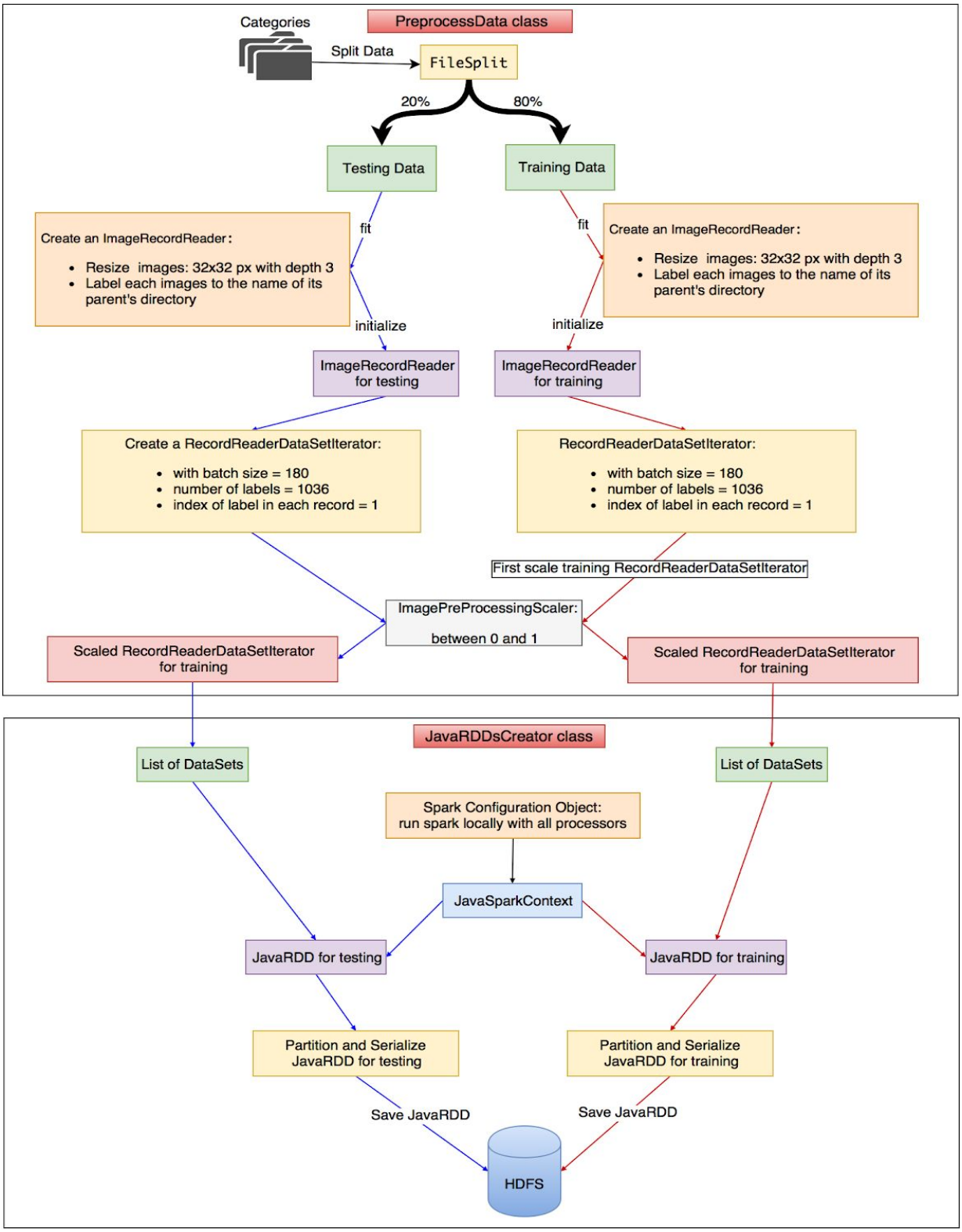
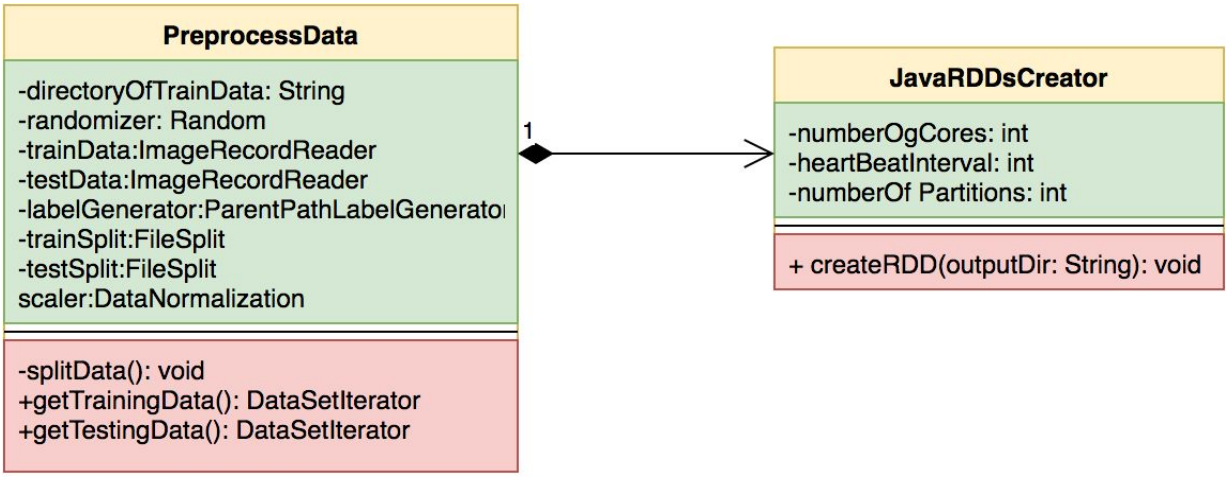


Figure 4.5: components of PreprocessData class and JavaRDDsCreator class, and the relationship between them

UML class diagrams for PreprocessData class and JavaRDDsCreator class and their relationship:



Hadoop Distributed File System(HDFS) is used to store RDD partitions in a distributed way across the Spark environment. Each worker node in Spark environment become a slave node for HDFS. Spark worker nodes which hosting GPUs configured to host HDFS datanodes and Spark master node configured to host HDFS namenode. Figure 4.6 illustrates the architecture of the distributed file system in Spark environment.

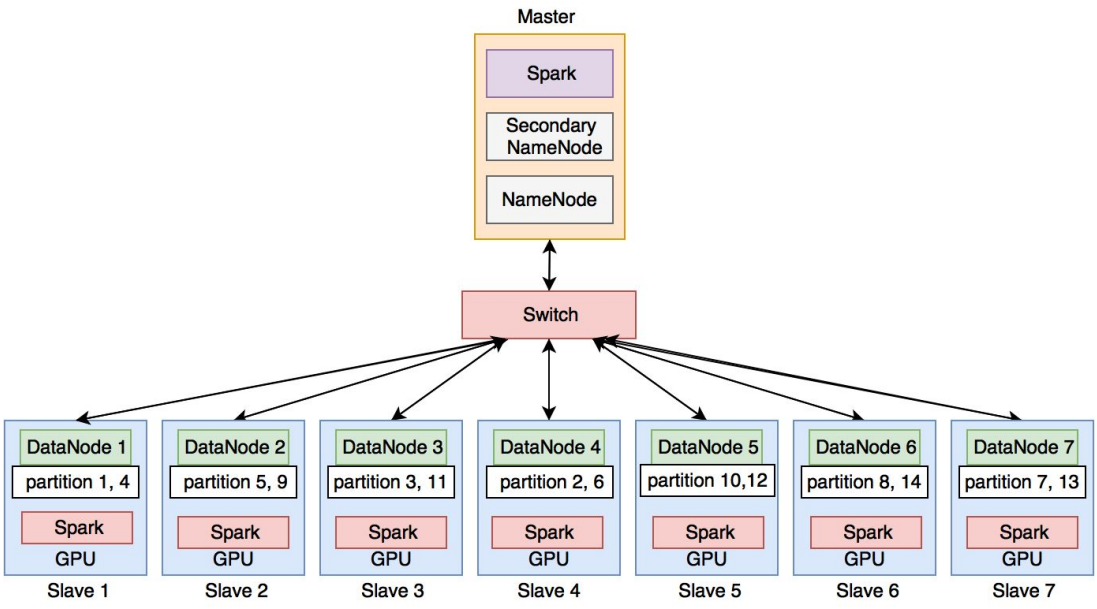


Figure 4.6: the architecture of the distributed file system in Spark environment

4.2 Convolutional Neural Network Model

Designing a convolutional neural network model is a piece of art depends on the training data and the environment which would host the execution of the model. Push GPUs to their maximum computation capacity depends on the GPUs memory, and executors memory. Increasing the number of images in each mini-batch would increase the required computation on GPUs. Processing a mini-batch with 180 images (each images with size 32x32 pixels) on a CNN model with 1.2 million parameters would keep a GPU with 4GB busy almost all the time.

DeepLearning4J and ND4J libraries provide many different types of highly optimized and tested functions and algorithms which provide the quickest way to configure a CNN model capable of using GPUs for numerical computations. DeepLearning4J and ND4J communicate and operate on GPUs via CUDA Toolkit, NVIDIA cuBLAS and NVIDIA CUDA Deep Neural Network library (cuDNN). NVIDIA defines these libraries as follows:

- CUDA Toolkit: "provides a development environment for creating high performance GPU-accelerated applications. With the CUDA Toolkit, you can develop, optimize and deploy your applications on GPU-accelerated embedded systems, desktop workstations, enterprise data centers, cloud-based platforms and HPC supercomputers. GPU-accelerated CUDA libraries enable drop-in acceleration across multiple domains such as linear algebra, image and video processing, deep learning and graph analytics."
- NVIDIA cuBLAS : "The NVIDIA cuBLAS library is a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS). Using cuBLAS APIs, you can speed up your applications by deploying compute-intensive operations to a single GPU or scale up and distribute work across multi-GPU configurations efficiently."
- NVIDIA CUDA Deep Neural Network library (cuDNN): "is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers."

Figure 4.7 shows a CNN model architecture and properties for each layer.

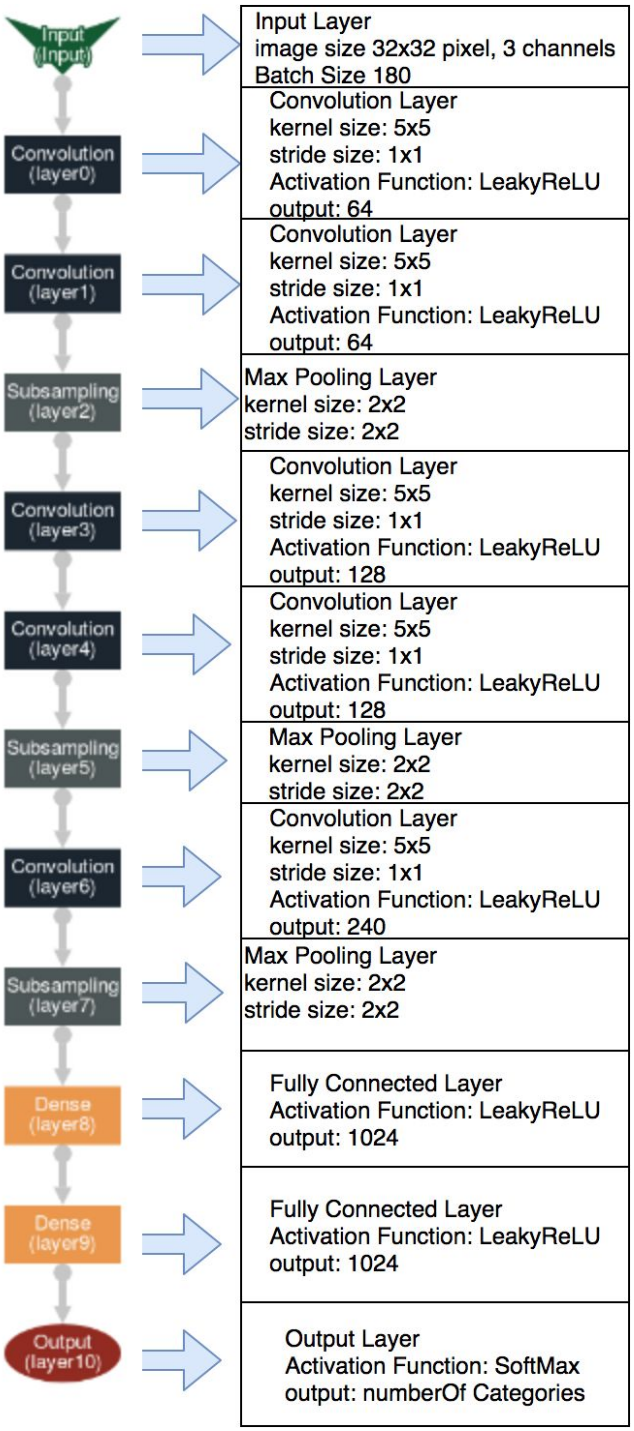
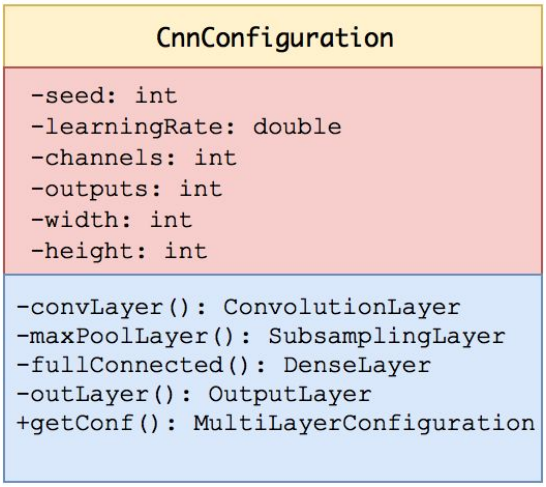


*Figure 4.7: CNN model architecture and properties for each layer*s

The UML class diagram for configure the Model in figure 4.7:



4.3 Training CNN Model on GPUs and Spark

A GPU acceleration code should be developed to send the data and commands to GPUs to perform a numerical computation. Fortunately DL4J, and ND4J has already developed all the code required for sending data and commands to GPUs. A developer needs to install and configure the required libraries in an environment which hosts GPUs and use these libraries APIs to make a deep learning application capable of using both GPUs and CPUs resources.

Deeplearning4j (DL4J) works with GPUs provided by NVIDIA it uses 'N-dimensional array for java' (ND4J) library to perform all numerical computation tasks in parallel on GPUs in a Standalone or distributed Spark environment. Each worker node in our Spark environment host a GPU device. The master node does not host any GPU device, due to this the Spark environment must be configured to distribute training tasks automatically between available CPUs and GPUs in the cluster as follows:
If a GPU is available on a node, then a high priority should be given to GPU to do linear algebra operations on that node.
If a GPU is not available on a node, then a high priority should be given to CPU to do linear algebra operations on that node.

A CNN model training process on Spark environment: the Spark master node launches the required number of executors in Spark worker nodes, and it sends a copy of the model to each executor in the cluster. The Spark master assigns training tasks to each executor. Every executor distributes the execution of each training task between CPUs and GPUs allocated to it. An Executor distributes a training task as follows: sending linear algebra computations to GPUs via ND4J and the rest of computations execute on CPUs.
Figure 4.8 illustrates how the Spark environment looks like, after submitting a job to the environment.

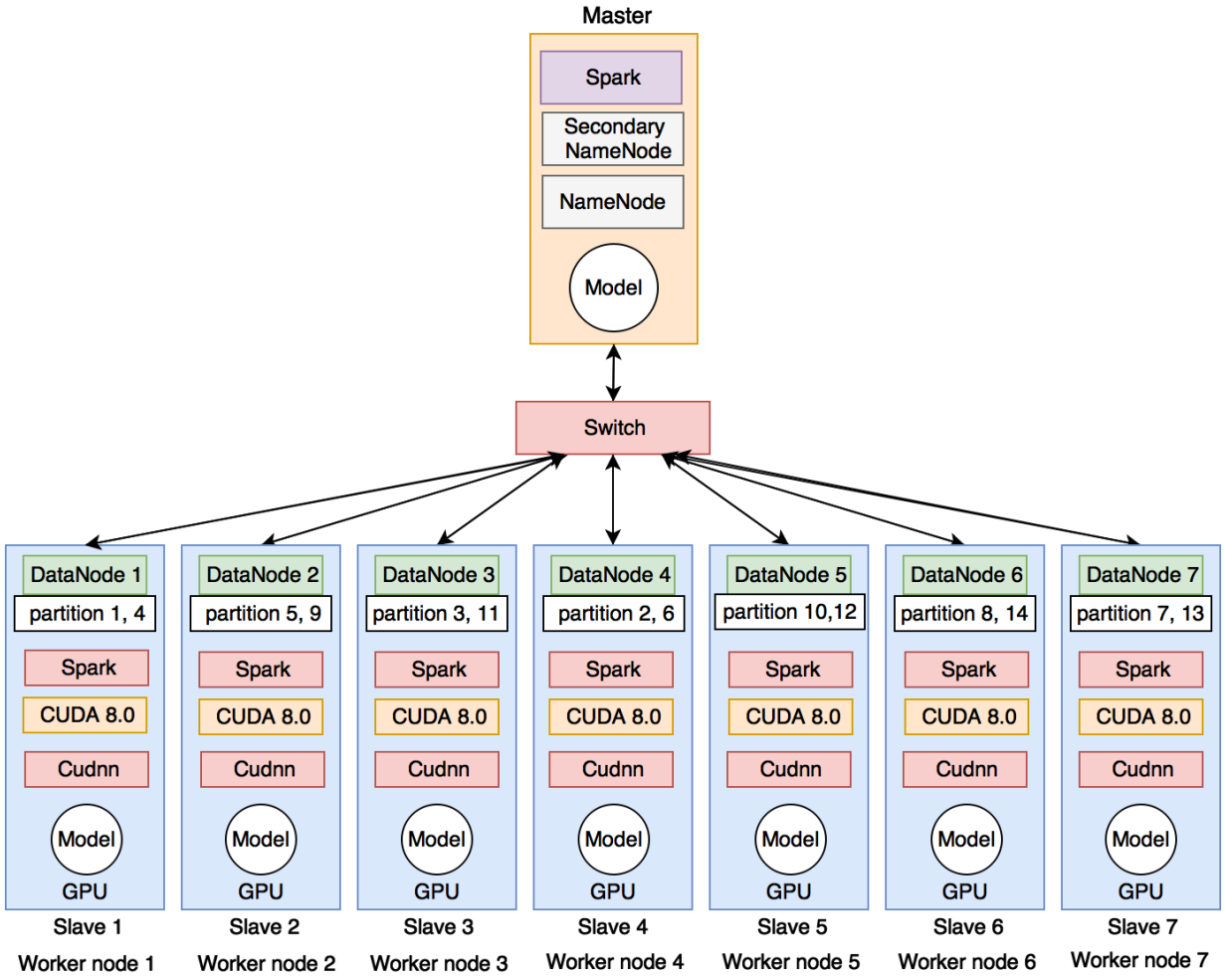


Figure 4.8: distributing a model and training tasks in a Spark Environment

In each iteration over the training data every worker node trains its local model with RDD partitions on its local disk, each RDD partition contains N mini-baches, after the model trains with K mini-batch(K in range 4 to 7) it sends its parameter to the average parameter service on the master node. The Average Parameter

Service receives parameters from each worker node, calculate the average for these parameters, update the parameters in master's model, then the master node broadcasts its model's parameters to all models in the cluster which means after training a model with K mini-batches, all the models would have the same parameters.
Figure 4.9 shows how the Average Parameter Service works for every K mini-batches.

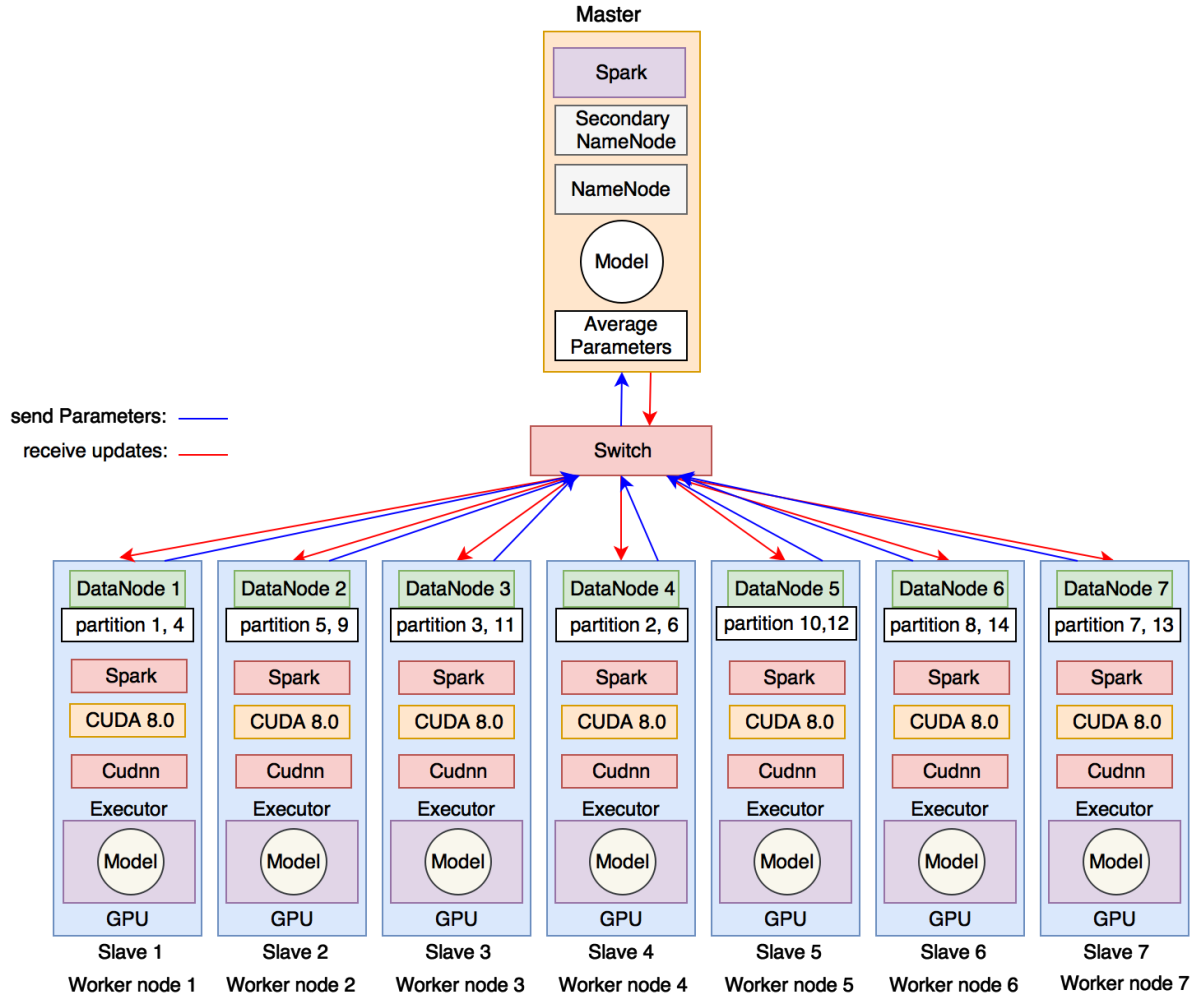


Figure 4.9: Average Parameters Service after every K min-batches

Each executor sends its local model's parameters to the Average Parameters Service after every K min-batches. The executors should always have enough mini-batches in their allocated memory, to ensure that each executor prefetches M min-batches (M in range 5 to 10), where m >= K. this would also reduce pausing the training process to load mini-batches from HDFS.

Figure 4.10 illustrates how an executor sends its model's parameters to the Average Parameters Service after K min-batches.

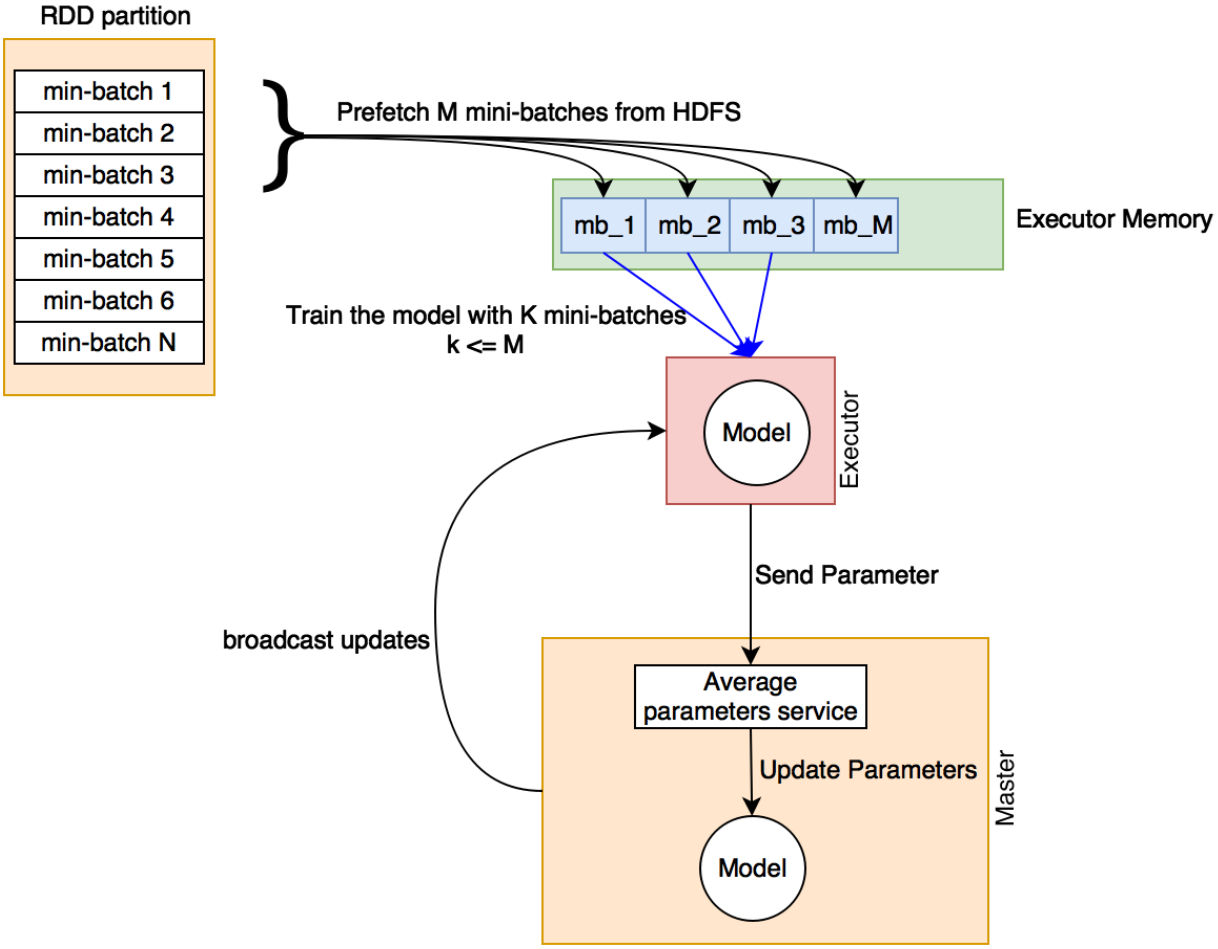


Figure 4.10: loading M mini-batches from HDFS to an executor memory and send parameters to Average Parameter Service after processing every K mini-batches

4.4 Memory Management

ND4J allocate memory for matrices or vectors outside of the JVM heap in a separate block of memory called Off-heap memory. The benefit of using off-heap memory for ND4J operations is to efficiently use c++ code from BLAS libraries to do linear algebra operations. Moreover, off-heap memory is also necessary for GPUs operations with CUDA since GPUs cannot operate with CUDA inside JVM heap. ND4J applies all required operations on matrices in off-heap memory and passes a pointer to the JVM which points to the results.
Figure 4.11 illustrates an ND4J operation on a matrix inside off-heap memory with CPU backend and passing a result pointer to JVM.

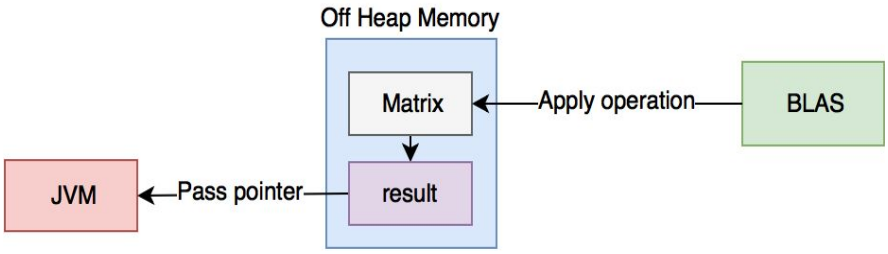


Figure 4.11: an ND4J operation on a matrix inside off-heap memory

Figure 4.12 illustrates an ND4J operation on a matrix inside off-heap memory with GPU backend and passing a result pointer to JVM.
Libnd4j is a C++ ND4J library work together with a BLAS library to perform numerical operations on matrices or vectors. By default, libnd4j uses cuBLAS for GPU-backend, and OpenBLAS for CPU-backend, but there are much faster BLAS libraries for CPU-backend such as Intel Math Kernel Library(MKL). Improving the performance of ND4J library requires recompiling libnd4j library to use all GPUs compute capability for GPU-backend and to link DN4J with MKL for CPU-backend. After recompiling Libnd4j library, ND4J links with two BLAS libraries cuBLAS and MKL. Two backend environment variables BACKEND_PRIORITY_GPU (for cuBLAS), and BACKEND_PRIORITY_CPU (for MKL) must be set to define which BLAS library ND4J has to use.

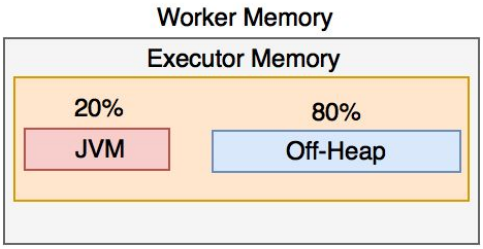
ND4J uses a backend with higher priority as BLAS. The values for BACKEND_PRIORITY_GPU and BACKEND_PRIORITY_CPU in the Spark environment are defined as follows:

- For every spark node hosts a GPU (all worker nodes):
  BACKEND_PRIORITY_GPU = 100
  BACKEND_PRIORITY_CPU = 0
- For every Spark node does not host any GPU (only master node):
  BACKEND_PRIORITY_GPU = 0
  BACKEND_PRIORITY_CPU = 100

Spark jobs run in Spark executors, and each executor has fixed amount of memory. If a task inside a Spark executor exceeds the amount of memory allocated to that executor, then Spark kills that executor and throws “outOfMemory” exception. To avoid these exceptions we need to increase the off-heap memory size by increasing the amount of memory allocated to the executor. In our Spark cluster, each worker node has 6.7GB available memory, and it would host only one Spark executor. Since no RDDs would be cached inside JVM memory each Spark executors partitions its memory as follows:

- 20% of executor memory for JVM heap

- 80% of memory for off-heap.



Spark.executor.memory property defines the amount of memory for each executor in a spark cluster. 10% of an executor memory is reserved for Spark memory Overhead. *org.bytedeco.javacpp.maxbytes* property defines the amount of off-heap memory for ND4J operation. After off-heap memory is reserved, the remain memory of executor will be used for JVM and other libraries. Finally, *Spark.driver.memory* property specify the Spark driver memory.

When a Job is submitted the Spark environment, the arguments for Spark-submit would be specified as follows:

```
--class “ClassName” --num-executors 7 --executor-cores 2 --executor-memory 5G
--driver-memory 16G --conf
"Spark.executor.extraJavaOptions=-Dorg.bytedeco.javacpp.maxbytes=4368709120" --conf
"Spark.driver.extraJavaOptions=-Dorg.bytedeco.javacpp.maxbytes=4368709120" “jar file
path”
```

Dorg.bytedeco.javacpp.maxbytes is specified in bytes.
One more argument we need to all to the spark submit command is *--conf spark.locality.wait=0,* this property will let spark to transfer data to executor without waiting for executors to be free which increase the performance of a spark job especially since the RDD partitions are stored in HDFS.

When spark will load RDD partition to memory, it can not estimate the size of objects in the RDD partitions, if the size of objects too large: this can cause the out of memory exceptions or increase the Garbage Collection frequency (removing no longer used objects from memory)
, to avoid these exceptions we can configure spark to store objects in memory in serialized form, using the serialized Storage-level such as MEMORY_ONLY_SER.

4.5 Serialization

In any distributed application the serialization plays an important roles. In our spark cluster and During training stage spark has to shuffle data, move java objects between disk and memory, before spark move any java object it will serialize that object, and after that object is been moved spark has to deserialize that object. By default spark use java standard serialization library to serialize any java object implements java.io.Serializable. The result of serializing a java object can be in big format which will require more memory, and also serializing a java object with java serialization is quite slow. We need a better library for serialization java object in our spark cluster. Kryo is a serialization library, using Kryo with spark can reduce the amount of time taken to serialize objects, which will increase the performance in our spark cluster. We can not directly use kyro to serialize objects in off-heap memory, we need to download the source code of the kyro library, make some changes in source code to implement a registrator for ND4J objects, then we configure the spark to use the Nd4J kryo registrator. If registrator is not configured correctly spark will throw NullPointerException when trying to serialize java objects, or we will get NaN in some numerical operations.

5. IMPLEMENTATION

DL4J has a collection of tools that provide a full platform for deep learning. Multiple dependencies are needed to wire DL4J with spark and GPU. these dependencies needs the following system configuration requirements:

- Ubuntu 16.0.4
- Spark 2.*
- java version "1.8.0_*"
- Scala 2.11.*
- Python 3.*
- Maven 3.2.5 or above: to control how these dependencies are wired together, and to build uber jar files,which will contains all java classes and all their dependencies in one single jar file. the advantage of building a uber jar file is to distribute a jar file and not care at all whether or not dependencies are installed at the destination.

- Git
- Cuda 8 for GPUs
- NVIDIA CUDA Deep Neural Network library (cuDNN)
- Intel Math Kernel Library (MKL)
- Libnd4j
- Cmake: to compile Libnd4j
- IDLE: we will use Intellij

And also we need to define the following environment variables:

- JAVA_HOME: define the location of java in the environment
- SPARK_HOME: define the location of spark in the environment
- MKL_NUM_THREADS: set the number of threads to be used by MKL
- OMP_NUM_THREADS: set the number of OpenMP threads will be used for BLAS calls and other native calls.
- LIBND4J_HOME: define the location of LIBND4J (Native operations for nd4j) in the environment
- LD_LIBRARY_MATH: define the location of math libraries(MKL, CUDA and CuDNN) in the environment
- BACKEND_PRIORITY_CPU: priority to use CPU as backend. a small value as “0” will ignore CPU backend, this variable is needed to switch between CPU and GPU backends
- BACKEND_PRIORITY_GPU:priority to use GPU as backend. Value as “100” will force the environment to use GPU as backend for ND4j operations.

5.1 Install Dependencies

Maven is an automated build tool for Java projects, and it ensures that a java project would always have the latest version of all its dependencies. It works well with Integrated Development Environments (IDEs) such as IntelliJ.

A maven’s Project Object Model (pom.xml) is created using IntelliJ to define the required dependencies and their version to wiring Dl4j with Spark and GPUs.
pom.xml

- variables for versions in pom.xml are as follows:

```
<java.version>1.8</java.version>
<nd4j.version>0.9.1</nd4j.version>
<dl4j.version>0.9.1</dl4j.version>
<datavec.version>0.9.1</datavec.version>
<dl4j.Spark.version>0.9.1_Spark_2</dl4j.Spark.version>
<datavec.Spark.version>0.9.1_Spark_2</datavec.Spark.version>
<jcommon.version>1.0.23</jcommon.version>
<maven-compiler-plugin.version>3.6.1</maven-compiler-plugin.version>
<maven-shade-plugin.version>2.4.3</maven-shade-plugin.version>
<exec-maven-plugin.version>1.4.0</exec-maven-plugin.version>
<maven.minimum.version>3.3.1</maven.minimum.version>
<jackson.version>2.6.6</jackson.version>
<scala.plugin.version>3.2.2</scala.plugin.version>
```

- Maven Enforcer: ensures we have up to date version of Maven

```
<build>
   <plugins>
       <plugin>
           <artifactId>maven-enforcer-plugin</artifactId>
           <version>1.0.1</version>
           <executions>
               <execution>
                   <id>enforce-default</id>
                   <goals>
                       <goal>enforce</goal>
                   </goals>
                   <configuration>
                       <rules>
                           <requireMavenVersion>

<version>[${maven.minimum.version},)</version>
                               <message>********** Minimum Maven Version is
${maven.minimum.version}. Please upgrade Maven before continuing (run "mvn
--version" to check). **********</message>
                           </requireMavenVersion>
                       </rules>
                   </configuration>
               </execution>
            </executions>
```

```
        </plugin>

        <!-- Automated Code Formatting -->
        <plugin>
            <groupId>com.lewisd</groupId>
            <artifactId>lint-maven-plugin</artifactId>
            <version>0.0.11</version>
            <configuration>
                <failOnViolation>true</failOnViolation>
                <onlyRunRules>
                    <rule>DuplicateDep</rule>
                    <rule>RedundantPluginVersion</rule>
                    <rule>VersionProp</rule>
                    <rule>DotVersionProperty</rule>
                </onlyRunRules>
            </configuration>
            <executions>
                <execution>
                    <id>pom-lint</id>
                    <phase>validate</phase>
                    <goals>
                        <goal>check</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.eclipse.m2e</groupId>
                <artifactId>lifecycle-mapping</artifactId>
                <version>1.0.0</version>
                <configuration>
                    <lifecycleMappingMetadata>
                        <pluginExecutions>
                            <pluginExecution>
                                <pluginExecutionFilter>
                                    <groupId>com.lewisd</groupId>

<artifactId>lint-maven-plugin</artifactId>
                                    <versionRange>[0.0.11,)</versionRange>
```

```
                                    <goals>
                                        <goal>check</goal>
                                    </goals>
                                </pluginExecutionFilter>
                                <action>
                                    <ignore/>
                                </action>
                            </pluginExecution>
                        </pluginExecutions>
                    </lifecycleMappingMetadata>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

- Set required dependencies:

```
<dependencies>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.module</groupId>
        <artifactId>jackson-module-scala_2.11</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>org.nd4j</groupId>
        <artifactId>nd4j-native-platform</artifactId>
        <version>${nd4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.nd4j</groupId>
         <artifactId>nd4j-cuda-8.0-platform</artifactId>
```

```
        <version>${nd4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.deeplearning4j</groupId>
        <artifactId>dl4j-Spark_2.11</artifactId>
        <version>0.9.1_Spark_2</version>
    </dependency>



    <dependency>
        <groupId>org.deeplearning4j</groupId>

<artifactId>dl4j-Spark-parameterserver_${scala.binary.version}</artifactId>
        <version>${dl4j.Spark.version}</version>
    </dependency>
    <dependency>
        <groupId>com.beust</groupId>
        <artifactId>jcommander</artifactId>
        <version>${jcommander.version}</version>
    </dependency>
    <dependency>
        <groupId>org.deeplearning4j</groupId>
        <artifactId>deeplearning4j-core</artifactId>
        <version>${dl4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.deeplearning4j</groupId>
        <artifactId>deeplearning4j-ui-model</artifactId>
        <version>${dl4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.deeplearning4j</groupId>
        <artifactId>deeplearning4j-zoo</artifactId>
        <version>${dl4j.version}</version>
    </dependency>
    <dependency>
         <groupId>org.nd4j</groupId>
        <artifactId>nd4j-base64</artifactId>
        <version>${nd4j.version}</version>
    </dependency>
    <dependency>
```

```
        <groupId>org.nd4j</groupId>
        <artifactId>nd4j-cuda-8.0</artifactId>
        <version>${nd4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.datavec</groupId>
        <artifactId>datavec-api</artifactId>
        <version>${nd4j.version}</version>
    </dependency>
    <dependency>
        <groupId>commons-io</groupId>
        <artifactId>commons-io</artifactId>
        <version>2.6</version>
    </dependency>
</dependencies>
```

Once IntelliJ downloads all dependencies defined in pom.xml file, all dependencies API will be available in IntelliJ to implement the following java classes:

- Hyperparameter interface: contains shared Hyperparameter between all classes
- DataReader: Read BSOn file and decode images
- CnnConfiguration: configures CNN model.
- PreprocessData class : load data, split data and preprocessing data
- JavaRDDsCreator: creates required RDDs
- CnnOnSpark2: contains Spark driver logic, training and evaluation process .

5.2 Shared Hyperparameter

Hyperparameters are the variables which determine how the network is built and trained such as learning rate, number of neurons in a layer, size of images and so on. Some of these values are used in multiple classes. To manage these shared variables a java interface “Hyperparameters” is implemented as follows:

```
interface Hyperparameter {
  int width = 32;
  int height = 32;
  int depth = 3;
```

```
int seed = 12345;
int outputs = 143;
int minBatchSize = 180;
int learningRate = 0.01;
int labelIndex = 1;
double trainSize = 0.8;
String dataPath = "/home/bcri/train_data";
int epochs = 30;
int averagingFrequency = 5;
int prefetchNumBatches = 7;
```

5.3 Read BSON File

The implementation of DataReader class.

Create an instance of BSON Decoder:

```
BSONDecoder decoder = new BasicBSONDecoder()
```

The data could be stored on multiple types of storage, such as local file system, distributed file system, or cloud storage. We need to open a pipeline to move the data to a machine which hosts the BSON Decoder. The size of the dataset is big; it cannot fit into that machine's main memory so we must load the data in chunks. An efficient way to do that is by using Java BufferedInputStream class, which opens an input stream to read the data from specified sources as chunks.

Create an instance of Java InputStream:

```
InputStream inputStream = new BufferedInputStream(new FileInputStream(file));
```

Next, we link the decoder with input stream, and read each record as a BSONObject, which contains all information about that record.

```
BSONObject obj = decoder.readObject(inputStream)
```

Extract the image from each record and save it in corresponding file.

```
Map<String, Object> map = record.toMap();
List<Object> list = (List<Object>) map.get("imgs");
```

```
Map<String, byte[]> images = (Map<String, byte[]>) list.get(0);
FileOutputStream writer = new FileOutputStream(new
File("/home/bcri/extract/" +category_Id+"/"+ image_id));
byte[] img =images.get("picture");
writer.write(img);
```

5.4 Generating more training data

The more data we have, the better training we can achieve. Since the given training data contains images, some image transforms can be applied to existing images to obtain more data. Applying an image transform on an image creates a new where the object in the image will not change, only the position and color of the object will change. This process helps to set a balance between the number of images in each category and avoid overfitting as well. Figure 5.1 shows the difference between the number of images in each category before generating new images from existing images.

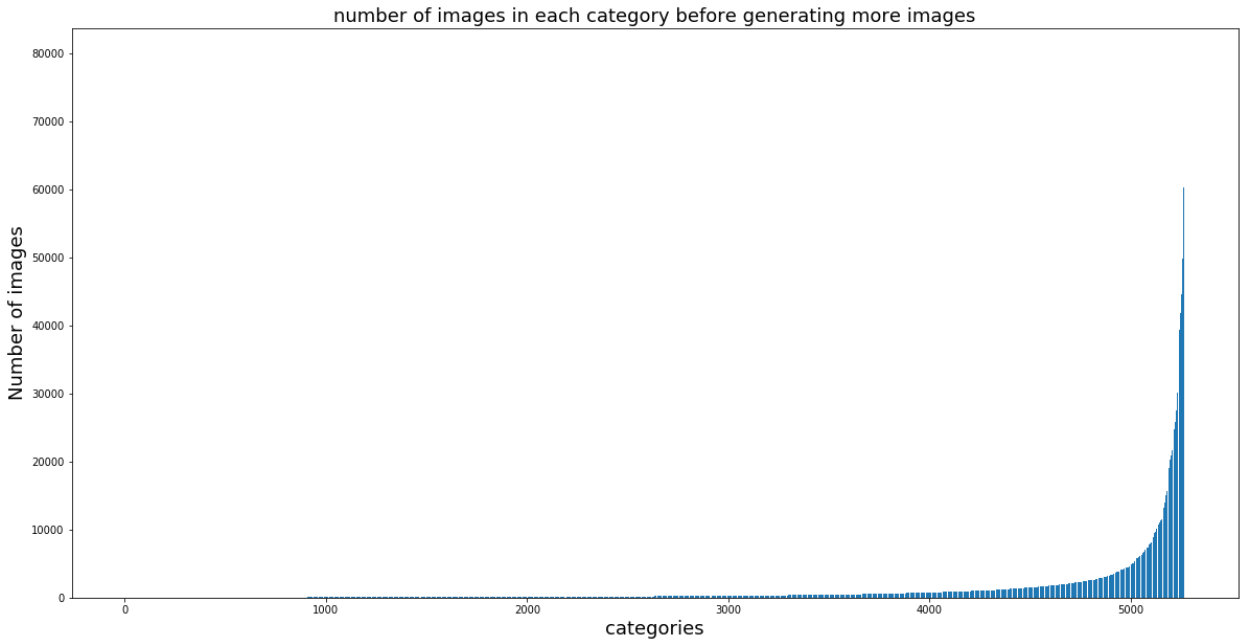


Figure 5.1: number of images in each category before generating new images from existing images

The following python script visits all categories folders and generate more images in each folder until the number of images in that folder becomes 8000 :

```
#!/usr/local/env python3
import Augmentor
import numpy
import sys

Max_number_of_images = 80000
#read file which contains information about each category
file = open ("counts.txt", "r")
#create a dictionary where each key represents folder name, and value for
each #key represent the number of images will be generated.
categories = {}
for line in file:
    parts = line.strip().split(',')
   categories[parts[0]] = 80000 - int(parts[1])
for i in categories:
    p = Augmentor.Pipeline("/home/bcri/data/train/"+i,
                           output_directory="/home/bcri/data/train/"+i)
    #register transforms
    p.flip_random(probability=1)
    p.rotate_random_90(probability=0.9)
    p.rotate180(probability=0.9)
    p.rotate270(probability=0.9)
    p.rotate(probability=0.5, max_left_rotation=5, max_right_rotation=10)
    p.rotate(probability=0.5, max_left_rotation=10, max_right_rotation=10)
    p.shear(probability=0.4, max_shear_left=5, max_shear_right=5 )
    p.skew(probability=0.6)
    p.zoom(probability=0.4, min_factor=1, max_factor=1.6)
    #generate images
    p.sample(categories[i])
sys.exit(0)
```

Figure 5.2 shows the number of images in each categories after generating new images.

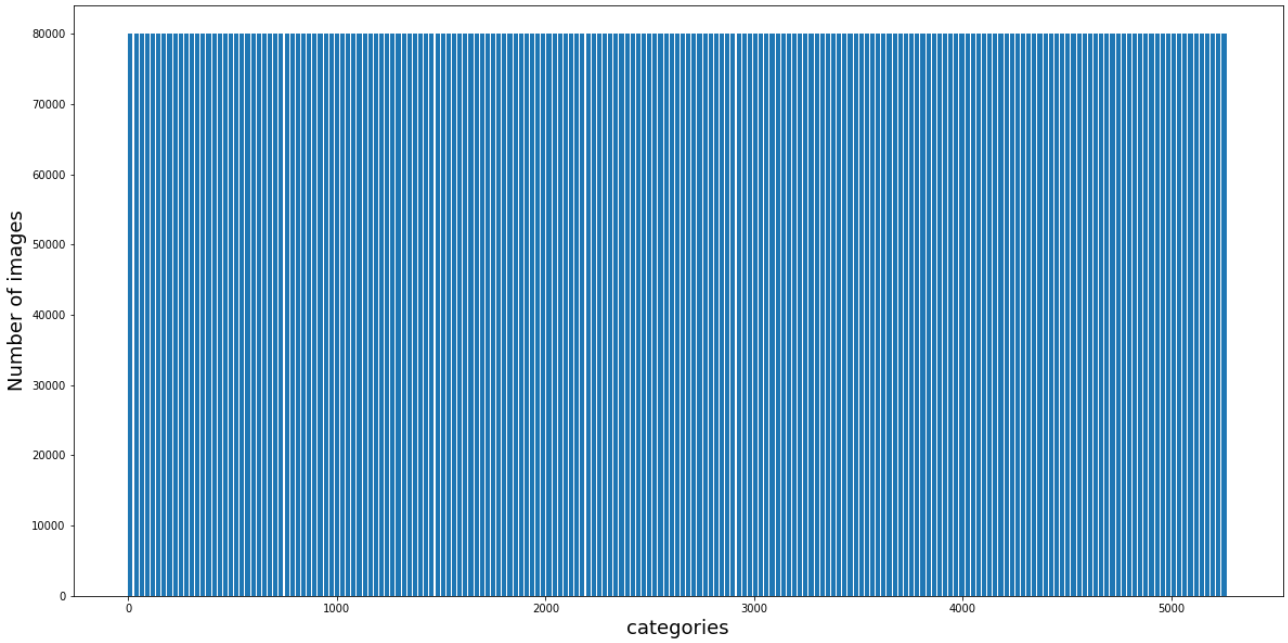


Figure 5.1: number of images in each category after generating new images from existing images

5.5 Loading Training Data

PreprocesData.java class reads the data from disk, splits the data into two parts (training dataset, testing dataset), changes the format of the dataset into a format acceptable by Dl4j image processing algorithms.

Specify the root directory for category folders and the allowable image formats in DataVec, and a Random Instance to randomize the order of images.

```
//root directory for categories
File DataDirectory = new File(Hyperparameter.dataPath);
Random rnd = new Random(Hyperparameter.seed);
String[] imageExtensions = NativeImageLoader.ALLOWED_FORMATS;
```

PreprocesData.java labels each image in the data to its parent file's id to let CNN model to know how to classify that image. An instance of ParentPathLabelGenerator is used to associate each image with a label.

```
ParentPathLabelGenerator labelGenerator = new
ParentPathLabelGenerator();
```

Split up the root directory to two splits, train data (80%) and test data (20%).

```
FileSplit inputSplit = new FileSplit(DataDirectory, imageExtensions, rnd);
InputSplit[] splitsFiles = inputSplit.sample(balancedPathFilter,
                        Hyperparameters.trainSize, 1 - Hyperparameters.trainSize);
InputSplit trainSplit = splitsFiles[0];
InputSplit testSplit = splitsFiles[1];
```

An instance of DataVec's ImageRecorderReader is used to resize and convert images to a format acceptable by Dl4j and create an image record reader.

```
ImageRecordReader train = new ImageRecordReader(Hyperparameters.height,
                    Hyperparameters.width, Hyperparameters.depth, labelGenerator);
train.initialize(trainSplit);

ImageRecordReader test = new ImageRecordReader(Hyperparameters.height,
                    Hyperparameters.width, Hyperparameters.depth, labelGenerator);
train.initialize(testSplit);
```

A RecordReaderDataSetIterator instance is used to iterate over elements in an image record reader.

```
DataSetIterator trainDataSetIterator = new  RecordReaderDataSetIterator(train,
                                             Hyperparameters.minBatchSize,
                                             Hyperparameters.labelIndex,
                                             Hyperparameters.outputs);
DataSetIterator testDataSetIterator = new  RecordReaderDataSetIterator(test,
                                            Hyperparameters.minBatchSize,
                                            Hyperparameters.labelIndex,
                                            Hyperparameters.outputs);
```

An instance of DataNormalization is used to normalize the value of each pixel to be between 0 and 1.0.

```
DataNormalization scalier = new ImagePreProcessingScaler(0, 1);
scalier.fit(trainDataSetIterator);
trainDataSetIterator.setPreProcessor(scalier);
```

```
testDataSetIterator.setPreProcessor(scalier);
```

5.6 Creating Java RDDs

JavaRDDsCeator class receives preprocessed datasets from PreprocesData class and creates required RDDs.

Create a SparkConf instance and set the required configurations to run Spark in local mode with all available cores.

```
//create an instance of Spark configuration
SparkConf SparkConf = new SparkConf();
//set master to local and use all cores for execution
SparkConf.setMaster("local[*]");
SparkConf.setAppName("create RDDs");
```

Prepare the data for JavaRDD creation process by storing the preprocessed data to Java list.

```
PreprocesData pipeline  = new PreprocesData();
//training data
DataSetIterator trainDataSetIterator = pipeline.getTrainDataSetIterator();
List<DataSet> trainDataSets = new ArrayList<>();
while (trainDataSetIterator.hasNext())
    trainDataSets.add(trainDataSetIterator.next())
//testing data
DataSetIterator testDataSetIterator = pipeline.getTestDataSetIterator();
List<DataSet> testDataSets = new ArrayList<>();
while (testDataSetIterator.hasNext())
testDataSets.add(testDataSetIterator.next());
```

Create an SparkContext instance to create the required RDDs and partition them to 14 partition since there are 14 cores in our Spark worker nodes. Each partition will be processed by one core.

```
JavaSparkContext SparkContext = new JavaSparkContext(SparkConf);
// create RDD for training data
JavaRDD<DataSet> trainData = SparkContext.parallelize(trainDataSets,14);
```

```
JavaRDD<DataSet> testData = SparkContext.parallelize(testDataSets, 14);
```

Store the RDD partitions in a disk

```
//save training RDDs to specified directories
trainData.saveAsObjectFile("/home/bcri/RDDS/train");
testData.saveAsObjectFile("/home/bcri/RDDS/test");
```

After training RDD and testing RDD created and stored on a disk we move them to HDFS which is configured to replicate each RDD partition in each Spark worker node.

```
$ hdfs dfs -put  ~/RDDS/train/* /user/RDDs/train
$ hdfs dfs -put  ~/RDDS/test/* /user/RDDs/test
```

5.7 CNN Configuration

The class CnnConfiguration.java contains techniques that are related to the CNN architecture to create and configuration CNN model return it.
The fields for this class are as follows:

```
private final int depth;       //number of channels
private final int width;       //width of the input image
private final int height;      //height of the input image
private final int numOutput;   // number of labels
private final int seed;        //seed for randomization
```

A neural network is defined through a NeuralNetConfiguration.Builder object as follows:

```
MultiLayerConfiguration cnnConf = new NeuralNetConfiguration.Builder()
```

The parameters for configuration the CNN model are as follows:

- Seed: an integer to be used as a seed for any randomization during network configuration.
- Iterations: an integer define after how many iterations the model will do a gradient search

- WeightInit: determines which algorithm will be used to initialize weights, biases and add randomization to the weight initialization progress. WeightInit.RelU is applied for weight initialization to obtain the best performance with leaky-ReLU activation function.
- optimizationAlgo: defines the algorithm for gradient search. Stochastic Gradient Descent is used to guarantee the model would find the global minimum cost function.
- Updater: defines the algorithm for update weights and biases. Updater.Adam is used to aiming the best performance.

```
.seed(seed)
.iterations(1)
.weightInit(WeightInit.RELU)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.ADAM)
```

- Regularization: to avoid overfitting a regularization algorithm needs to be applied to the model. The option we have are L1, L2, Max-Norm, and dropout. L2 doesn't drive small weights to zero providing better overfitting control.

```
.regularization(true).l2(0.0005)
```

- learningRate: this Hyperparameter is the most important Hyperparameter, it has a strong effect on the model performance.
  If learning rate is too small, then model can take longer time to learn. And if learningRate is too big, then the model training process will likely be Unstable. learningRate schedules is used to start training process with large learningRate, than reduce learningRate during the training process.

```
Map<Integer, Double> lrSchedule = new HashMap<>();
lrSchedule.put(0, 0.03);
//drop rate after N iterations
lrSchedule.put(1000, 0.025);
lrSchedule.put(3000, 0.005);
lrSchedule.put(4000, 0.0001);
```

```
.learningRate(0.03)
.biasLearningRate(0.02)
.learningRateDecayPolicy(LearningRatePolicy.Schedule)
.learningRateSchedule(lrSchedule)
```

- Layers configurations: setting the number of layer and neurons per layer determine how many total parameters the model has. More parameters, means more complex model, more freedom for the model to learn. The layers used in the CNN model are as follows:

❖ Convolution Layer:multiple Convolution Layer are used to apply feature detectors on the input images and extract features from them.

```
.layer(0, new ConvolutionLayer.Builder(5,5)
.cudnnAlgoMode(ConvolutionLayer.AlgoMode.PREFER_FASTEST)
         .nIn(depth)
         .stride(1, 1)
         .nOut(20)
         .activation(Activation.LEAKYRELU)
         .build())
```

This layer has the following properties:
Filter size: the size of the filter to extract features from the input image
CunDDN algorithm: which algorithm to use according to CuDNN, by default DL4J will use the fastest available algorithm "ConvolutionLayer.AlgoMode.PREFER_FASTEST", but this can increase the amount of memory usage, and cause memory Errors, "ConvolutionLayer.AlgoMode.NO_WORKSPACE" can be used to reduce memory usage.
Input data channels: the number of channels, since the input Images are colored the value for this property would be 3
Stride: define the steps to take when filtering across the input image. (1, 1) means to move one step to go from left to right and one step to move from top to down.
Activation function: an activation function which controls the output

of this layer
Number of outputs: define the number of outputs from this layer to the next layers. This number will be the number of inputs for the next layer.

- Subsampling Layer: after a sequence of Convolutional Layer a Subsampling Layer is added to reduce the size of feature maps without losing the high level features in the image. The following is how we will define this layer with Max-pooling operation:

```
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
       .kernelSize(2, 2)
       .stride(2, 2)
       .build())
```

This layer will have the following properties:
Pooling type: the type of the pooling operation, max pooling.
Kernel size: the size of the filter, (2, 2).
Pooling stride: define the number of steps when the filter moves across the image. (2, 2) means the filter moves two steps horizontally and two steps down to move to the next row.

- Dense Layer: each Dense layer is a hidden layer, a sequence of this layer are used to obtain better learning. Dl4j automatically sets The number of inputs for this layer which is the number of outputs from the pervious layer.

```
.layer(2 , new DenseLayer.Builder()
       .activation(Activation.LEAKYRELU)
       .nOut(128)
       .build())
```

- Output Layer: each Deep learning network has only one output layer. The number of neurons in this layer is equal to the number of categories in training data. The softmax is used as activation function to ensure that

predicted values are sum up to 1. The cost function NEGATIVELOGLIKELIHOOD is used a cost function for this layer

```
.layer( 3, new
OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
       .nOut(numOutput)
```

- SetInputType: calling this method adds preprocessors to handle the transition between the Convolutional layers, pooling layers and fully connected layers, and it does some additional configuration where necessary, like setting the number of inputs for each layer based on the size of the previous layer.

```
.setInputType(InputType.convolutional(height, width, depth))
```

- Backprop and pre-train: to active the backpropagation process and defines this network is not pre-trained.

```
.backprop(true)
.pretrain(false)
```

And finally, we call the build method to build an instance of `MultiLayerConfiguration`

```
.build();
```

5.8 Spark Driver

DL4j supports training deep networks on a Spark cluster to accelerate network training. Dl4j defines a class called SparkDl4jMultiLayer for training deep learning network on Spark.

CnnOnSpark2 java class contains the logical part of the spark job. In this class, we define how a distributed deep learning will execute in parallel.

Define some fields for this class and set some declaration on them. The declaration will be used to assign values to these field via command line. Jcommander library is used for parsing command line parameters.

```
@Parameter(names = "-epochs", description = "Number of epochs for training")
private int epochs = Hyperparameters.epochs;
@Parameter(names = "-averagingFrequency", description = "averaging Frequency
for Training Master to update parameters")
private int averagingFrequency = averagingFrequency;
@Parameter(names = "-prefetchNumBatches", description = "prefetch number of
Batches per worker")
private int prefetchNumBatches = Hyperparameters.prefetchNumBatches;
```

Define a logger for this class.

```
private static final org.slf4j.Logger log = LoggerFactory.getLogger(CnnOnSpark2.class);
```

Set the data type to half-precision to obtain the following benefits:

- “2x less GPU ram used.”
- “up to 200% performance gains on memory-intensive operations, though the actual performance boost depends on the task and hardware used.”

```
DataTypeUtil.setDTypeForContext(DataBuffer.Type.HALF);
```

Define the environment for GPU backend, allow the environment to use multiple GPUs and set some configurations to cache up to 4GB of GPU RAM.

```
CudaEnvironment.getInstance().getConfiguration().allowMultiGPU(true)
        .setMaximumDeviceCacheableLength(1024 * 1024 * 1024L)
        .setMaximumDeviceCache(4L * 1024 * 1024 * 1024L)
        .setMaximumHostCacheableLength(1024 * 1024 * 1024L)
        .setMaximumHostCache(4L * 1024 * 1024 * 1024L)
        .setMaximumGridSize(512)
        .setMaximumBlockSize(512);
```

Pausing any task for garbage collection effects on the task execution performance. The environment for GPU backend will pause GPUs every 10 seconds to perform garbage collection.

```
Dd4j.getMemoryManager().setAutoGcWindow(10000);
```

When the application starts first thing has to do is to parse the command line argument.

```
JCommander jCommander = new JCommander(this);
try {
   jCommander.parse(args);
} catch (ParameterException e2) {
   jCommander.usage();
   try {
       Thread.sleep(1000);
   } catch (InterruptedException e) {
       e.printStackTrace();
   }
}
```

Create a Spark configuration instance and set some configurations.

```
SparkConf SparkConf = new SparkConf();
```

Spark can be configured in three ways:

- Set configuration when we submit a Spark jobs by using command line
- Set configuration in the Spark-defaults-conf file.
- Set configuration with SparkConf object.

the following properties are set with SparkConf object to execute a Spark job on Spark environment.

- Spark Master: ip address of Spark master
- File System: set HDFS as default file system
- Application name: a name for the job
- Number of executors: the number of executor for the job
- Number of cores for each executor: all executors in Spark environment will have the same number of cores, and this number represent the number of tasks to be executed at the same time in each executor.
- Executor memory: heap size for each executor, also Spark executors have overhead memory which is equal to (executor memory * 0.10)

- Spark shuffle and storage memory fraction: is to further control the memory in each executor and specify the mount of memory for transforming and persisting data.
  Spark.shuffle.memoryFraction: to specify the amount of memory to be used for aggregation during the shuffle process in Spark.
  Spark.storage.memoryFraction: mount of memory for caching RDDs, since we will not catch any RDDs, we will set this property to 0.
- Spark serializer: Kryo serialization and deserialization library.
- Compression codec: compression library LZF to improve shuffle performance.

Before adding above properties to `SparkConf`, a KryoRegistrator needs to be implemented to register ND4j with Kryo library which makes kyro to be capable for serializing ND4J objects in off-heap memory.

```
import com.esotericsoftware.kryo.Kryo;
import de.javakaffee.kryoserializers.SynchronizedCollectionsSerializer;
import de.javakaffee.kryoserializers.UnmodifiableCollectionsSerializer;
import org.apache.Spark.serializer.KryoRegistrator;
import org.nd4j.linalg.factory.Nd4j;
import org.nd4j.linalg.primitives.AtomicDouble;

public class Nd4jRegistrator implements KryoRegistrator {
    @Override
    public void registerClasses(Kryo kryo) {
        kryo.register(Nd4j.getBackend().getNDArrayClass(),
                      new Nd4jSerializer());
        kryo.register(Nd4j.getBackend().getComplexNDArrayClass(),
                      new Nd4jSerializer());
        kryo.register(AtomicDouble.class, new AtomicDoubleSerializer());
        UnmodifiableCollectionsSerializer.registerSerializers(kryo);
        SynchronizedCollectionsSerializer.registerSerializers(kryo);
    }
}
```

```
Nd4jRegistrator needs to be compiled and stored in the following directory:
~/.m2/repository/org/nd4j/nd4j-kryo_2.11/0.9.1/nd4j-kryo_2.11-0.9.1.jar/org/nd4j
```

Set the master node's address , port and a name for our application

```
sparkConf.setMaster("Spark://bcri-1050ti-bm-master:7077");
sparkConf.setAppName("Running CNN on Spark 2");
```

Set HDFS as a default file system

```
sparkConf.set("spark.hadoop.fs.defaultFS",
                "hdfs://bcri-1050ti-bm-master:9000");
```

Set memory properties

```
sparkConf.set("Spark.driver.memory", "16g");
sparkConf.set(" Spark.shuffle.memoryFraction", "0.75");
sparkConf.set("Spark.storage.memoryFraction", "0");
sparkConf.set("Spark.executor.memory","6g")
```

Set serializer and register ND4J with Kyro, and set the size of buffer for Kyro serializer.

```
sparkConf.set("Spark.serializer","org.apache.Spark.serializer.KryoSerializer");
sparkConf.set("Spark.kryo.registrator", "org.nd4j.Nd4jRegistrator");
sparkConf.set("Spark.kryo.insafe", "true");
SparkConf.set("Spark.kryoserializer.buffer.max", "512m");
```

Set type of compression.

```
SparkConf.set("Spark.io.compression.codec", "lzf");
```

Selecting a garbage collector for the JVM and Spark and Cleaning up cached RDDs after they are no longer needed.

```
SparkConf.set("Spark.executor.extraJavaOptions", "-XX:+UseG1GC");
```

Create an instance of JavaSparkContext, which is a Spark driver for our application.

```
SparkContext = new JavaSparkContext(SparkConf);
```

Create a configuration instance for our deep learning network.

```
MultiLayerConfiguration cnnConf  = new CnnConfiguration().getConf();
```

Create a TrainingMaster instance, which specifies how the distributed deep learning approach works. ParameterAveragingTrainingMaster is used as a distributed deep learning approach with following configurations:

- batchSizePerWorker: to control the mini-batch size for each executor.
- averageFrequency: to control how frequently the parameters are averaged and redistributed.
- workerPrefetchNumBatches: to avoid Spark executors to wait for data to be loaded we can prefetch some number of mini-batches into executor's memory.
- saveUpdater: Each Spark executor has an updater, which is an algorithm to update the parameters. This updater has a state. If saveUpdater is set to true, then the updater state is averaged and returned to the master along with the parameters. When the master node receives these updates, it updates its model parameters and updater and redistributes these updates to all workers.
- StorageLevel: defines the type of the storage level, StorageLevel.MEMORY_ONLY_SER to stores ND4J objects in serialized form in memory.
- rddTrainingApproach: RddTrainingApproach.Export allows each executor to load datasets object directly from HDFS.

```
TrainingMaster trainingMaster = new
ParameterAveragingTrainingMaster.Builder(Hyperparameter
.batchSize)
```

Create a SparkDl4jMultiLayer instance using the SparkContext, the network configuration, and TrainingMaster objects

```
SparkDl4jMultiLayer SparkDl4jMultiLayer = new SparkDl4jMultiLayer(SparkContext,
                                              cnnConf,trainingMaster);
```

Read training and testing RDDs from HDFS.

```
JavaRDD<DataSet> trainData = SparkContext.objectFile("/user/RDDS/train");
```

```
JavaRDD<DataSet> testData = SparkContext.objectFile("/user/RDDS/test");
```

DL4J UI configurations

```
UIServer uiServer = UIServer.getInstance();
uiServer.enableRemoteListener();
StatsStorageRouter remoteUIRouter = new
RemoteUIStatsStorageRouter("http://bcri-1050ti-bm-master:9001");
SparkNet.setListeners(remoteUIRouter,
         Collections.singletonList(new StatsListener(null)));
```

The training RDD fits into the `SparkDl4jMultiLayer` where the network learns from examples in training RDD. this process is repeated N times where N is the number of epochs

```
SparkDl4jMultiLayer.fit(trainData)
```

During the training process, many temporary files are created. At the end of the training process, these files are removed and spark Context session is closed

```
trainingMaster.deleteTempFiles(sparkContext);
sparkContext.close();
```

Finally, the performance of the trained model is evaluated. If the performance of the model is low, then we need to retune the model and rerun the application.

```
Evaluation evaluation = sparkDl4jMultiLayer.doEvaluation(testData,
Haperparameter.minBatchSize, new
Evaluation(Hyperparameter.outputs))[0];
log.info.evaluation.stats();
```

5.9 Monitoring GPUs Usage

Nvidia provide a tool called nvidia-smi to track memory usage, GPU utilization and temperature of GPU. to measure the amount of time where GPUs are not idle during an application execution, each Spark worker nodes sends its GPUs parameters to the Spark master node every 0.01 second. Spark master displays GPUs usage in a 5 second window.

The following command line runs on each Spark worker node to read GPU used memory, GPU utilization and store them on a file every 0.01 seconds:

```
while true; do nvidia-smi --query-gpu=utilization.gpu,memory.used
--format=csv >> gpu_utilization1.log; sleep 0.01; done
```

Each Spark worker node sends the GPUs records to the Spark master node every 2 second with the following command line:

```
while true; do scp ./gpu_utilization1.log bcri@172.17.230.254:/home/bcri/;
sleep 2; done
```

Spark master node reads GPUs records and display plot them by executing ‘gpuMonitor.py’ script every 2 second with the following command:

```
while true; do ./gpuMonitor.py; sleep 2; done
```

‘gpuMonitor.py’ script:
Reads GPU usage file and generates a list for GPU used memory and GPU utilization for every 5 seconds and plot both lists.

6. TRAINING AND EVALUATION

This section provides the steps to run the spark environment, run an application in Spark environment, evaluate the performance of the Spark environment and the performance of the running application. All screenshots in this section are captured during the application running time.

6.1 Running Spark environment evaluation

Apache spark 2, Apache Hadoop 2.75, CUDA toolkit and all other required deep learning frameworks are installed and configured on each node in the spark environment.

In Spark master node a set of environment available required as follows:

```
export JAVA_HOME=/usr/lib/jvm/java-8-oracle
```

```
export SPARK_HOME=/home/bcri/spark
export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
export BACKEND_PRIORITY_GPU=0
export BACKEND_PRIORITY_CPU=100
export LIBND4J_HOME=/home/bcri/libnd4j
export
LD_LIBRARY_PATH=/opt/intel/lib/intel64:/opt/intel/mkl/lib/intel64:/usr/local/cuda/l
ib64:$HADOOP_HOME/lib/native:$LD_LIBRARY_PATH
export MKL_NUM_THREADS=24
export OMP_NUM_THREADS=12
Export MKL_BLAS=24
export MKL_DYNAMIC=FALSE
export OMP_DYNAMIC=FALSE
export HADOOP_HOME=/home/bcri/hadoop-2.7.5
export HADOOP_CONF_DIR=/home/bcri/hadoop-2.7.5/etc/hadoop
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

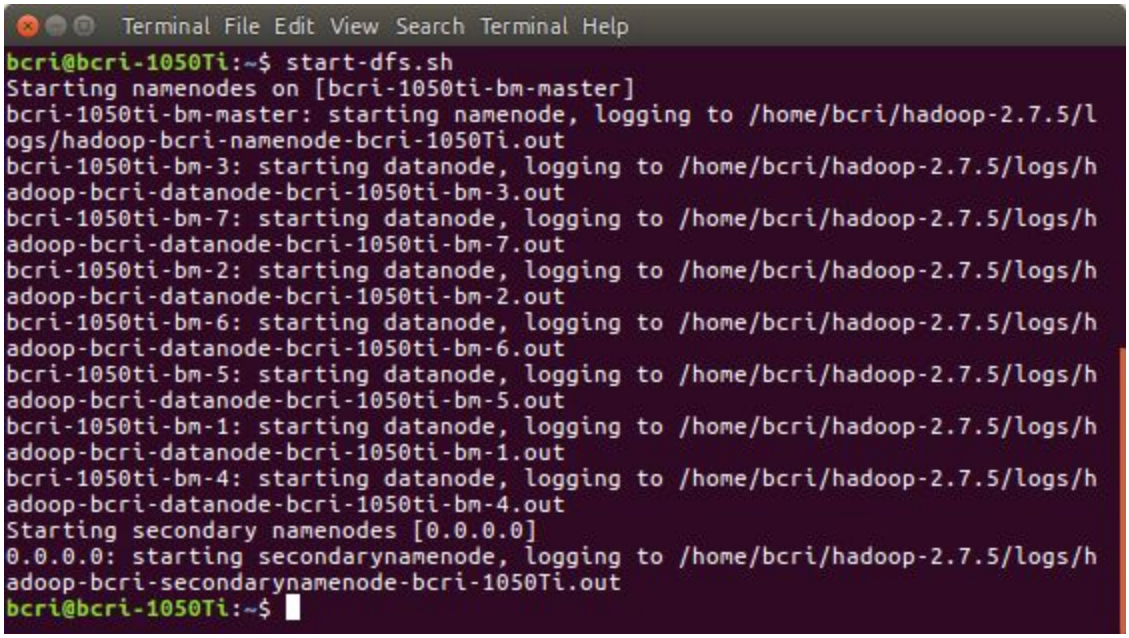
In each Spark worker node a set of environment variables required as follows:

```
export LIBND4J_HOME=/home/bcri/libnd4j
export
LD_LIBRARY_PATH=/opt/intel2/lib/intel64:/opt/intel2/mkl/lib/intel64:/usr/local/cuda
-8.0/lib64
export HADOOP_HOME=/home/bcri/hadoop-2.7.5
export HADOOP_CONF_DIR=/home/bcri/hadoop-2.7.5/etc/hadoop
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
export PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

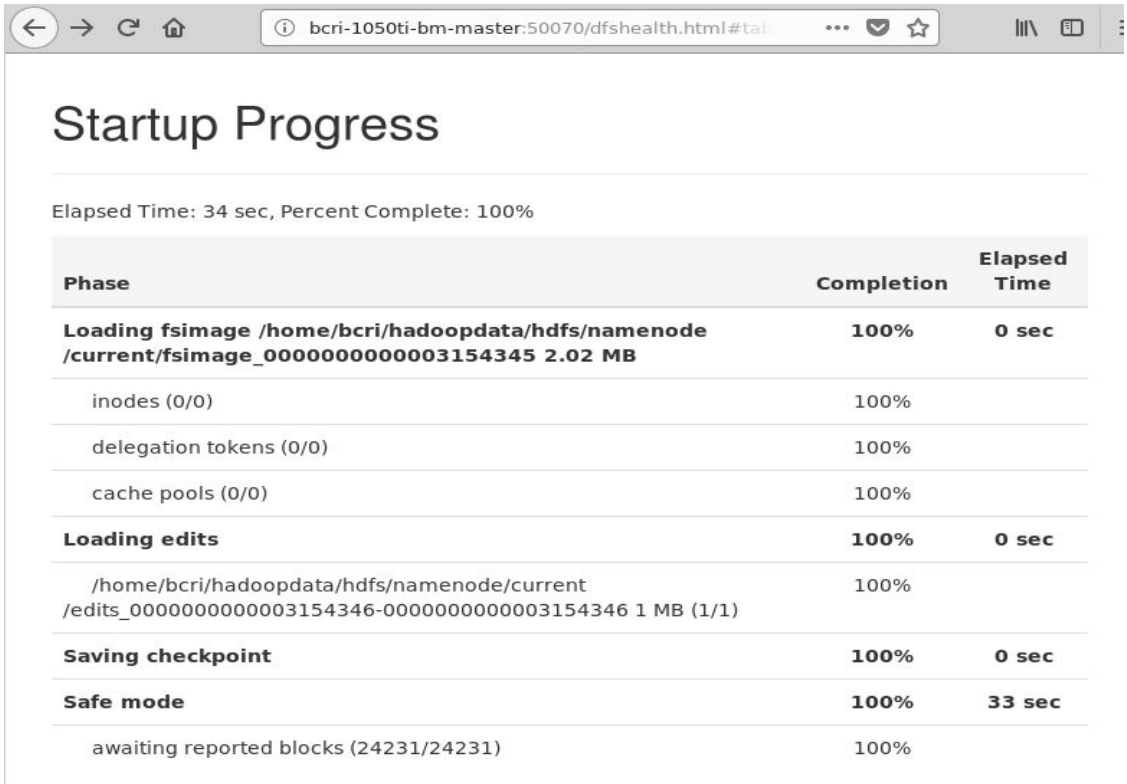
Extra configurations are required for each Spark worker to link Spark with HDFS. These configurations are added to the spark-env.sh file as follows:

```
SPARK_MASTER_HOST=bcri-1050ti-bm-master
<HADOOP_CONF_DIR=/home/bcri/hadoop-2.7.5/etc/hadoop
```

Running HDFS components: Running the following command in Spark Master node starts up HDFS service:



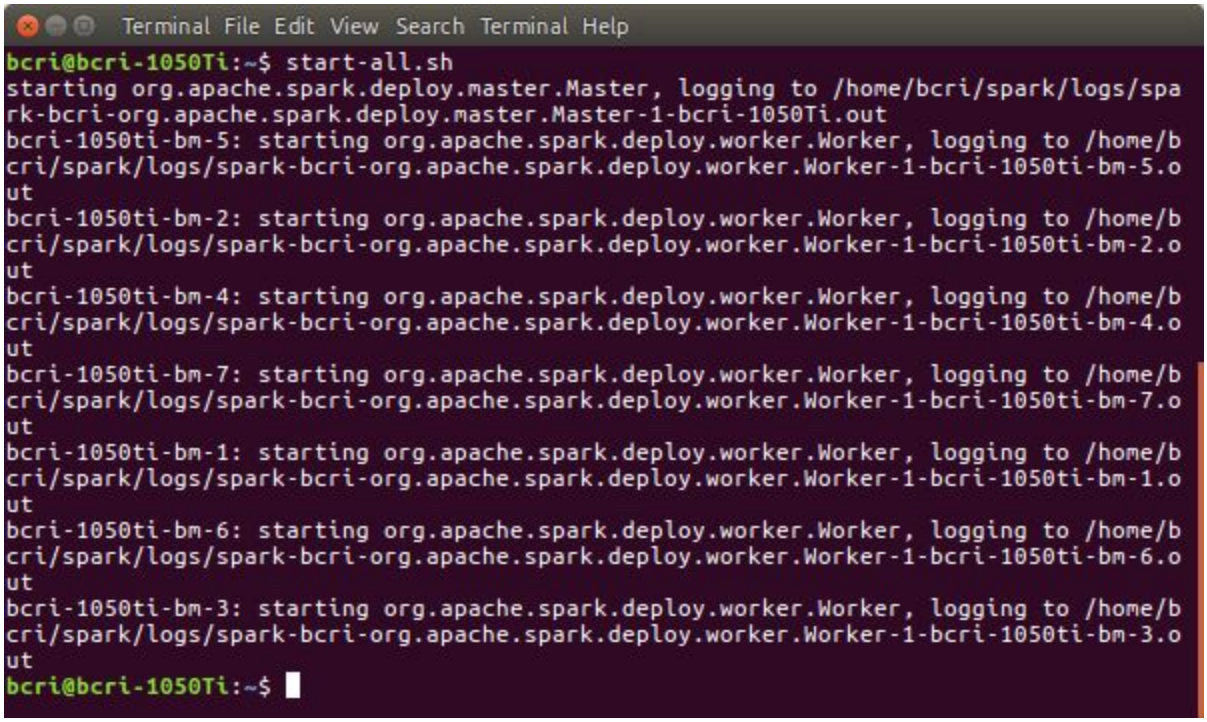
Hadoop provides Web User Interface to HDFS which is useful for essential status monitoring and file browsing operations. Accessing http://bcri-1050ti-bm-master:50070/dfshealth.html#tab-startup-process with a web browser confirms that all HDFS slaves are started with Elapsed Time: 34 sec, Percent Complete:100%:



In total there are 7 live datanodes with 3.08TB capacity.

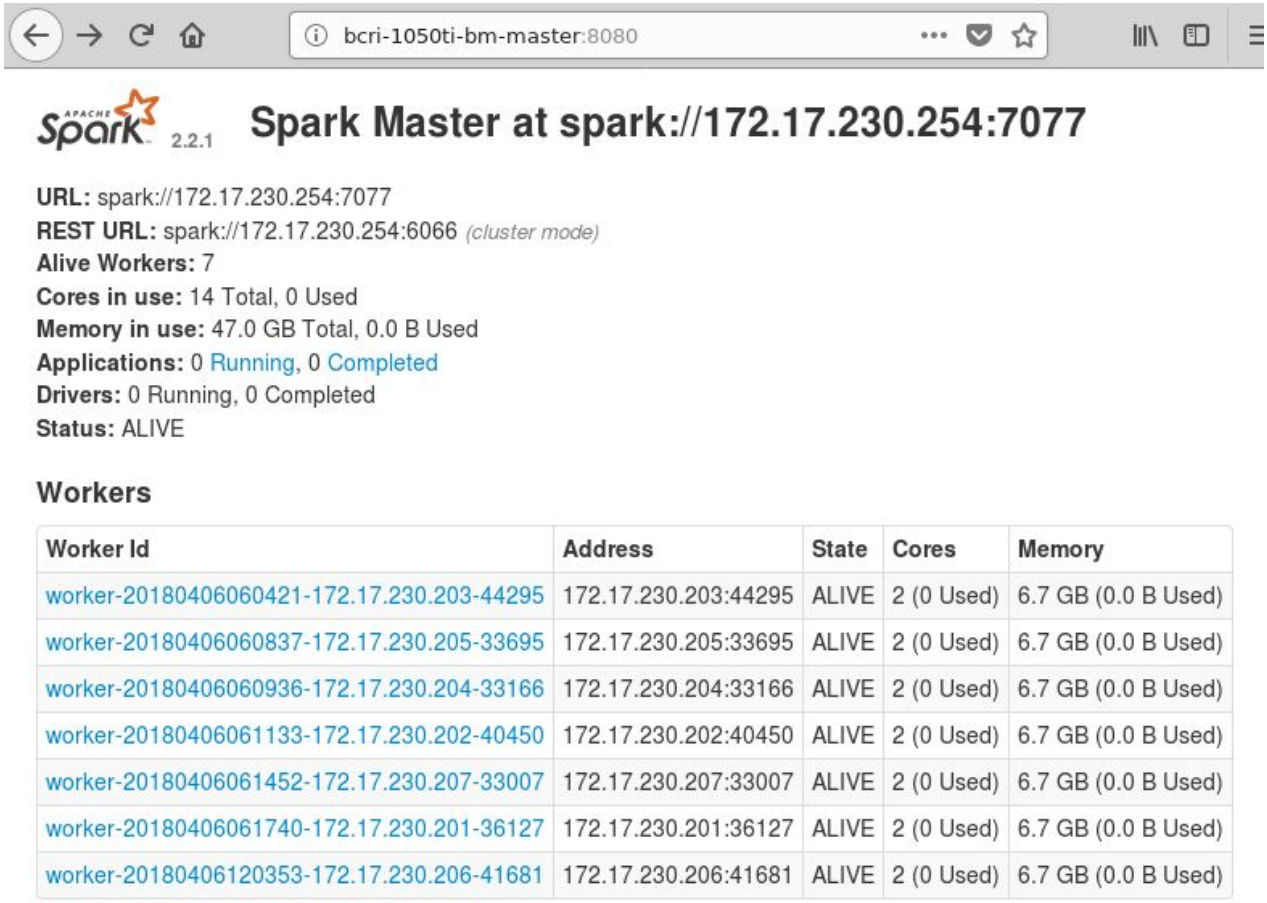
Configured Capacity:	3.08 TB
DFS Used:	103.97 GB (3.3%)
Non DFS Used:	742.02 GB
DFS Remaining:	2.09 TB (68.06%)
Block Pool Used:	103.97 GB (3.3%)
DataNodes usages% (Min/Median/Max/stdDev):	3.15% / 3.27% / 3.60% / 0.15%
Live Nodes	7 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)

Running Spark master node and worker nodes: running the following command line in Spark master node runs all required spark components to execute a Spark jobs.



Spark UI is the web interface of a running Spark application to inspect and monitor Spark jobs execution in a web browser. Accessing

http://bcri-1050ti-bm-master:8080 confirms that all spark workers are registered with the Spark master and gives a summary of the Spark cluster.In total, the Spark cluster has 7 worker nodes each worker node has 2 cores and 6.7 available memory



6.2 Running Application

To run an application on Spark, the application and all its dependencies are packaged in a jar file by invoking the following command in the root directory of the application which contains the pom.xml file.

```
mvn package
```

creates a jar file in the ./target subdirectory.

To execute the application on Spark, we submit the jar file to Spark with following properties:

- Number of executors 2
- Memory for each executor 6GB
- Number of cores for each executor 2 cores
- Memory for spark driver 16GB
- Off-heap memory size for each executor 5GB
- Use port 9001 for deeplearning4j UI
- Directory for jar file is ./job.jar

Submit the job to Spark environment:

```
spark-submit --class CnnOnSpark2 --num-executors 2 --executor-cores 7
--executor-memory 6G --driver-memory 16G --conf
"spark.executor.extraJavaOptions=-Dorg.bytedeco.javacpp.maxbytes=5368709120
-Dorg.deeplearning4j.ui.port=9001" --conf
"spark.driver.extraJavaOptions=-Dorg.bytedeco.javacpp.maxbytes=5368709120
-Dorg.deeplearning4j.ui.port=9001" --driver-java-options
"-Dorg.deeplearning4j.ui.port=9001" ./job.jar
```

Once the Job is submitted to the Spark environment, Spark launches required components (Spark master, Spark executors) each Spark component shows its launching process by outputting some logs to the console. Figure 6.1 shows the output of an executor during the launching process which specifies the type of ND4J backend, amount of available memory, number of cores, the BLAS for Nd4J operations and the type of GPU device.

```
INFO executioner.DefaultOpExecutioner: Backend used: [CUDA]; OS: [Linux]
INFO executioner.DefaultOpExecutioner: Cores: [2]; Memory: [5.8GB];
INFO executioner.DefaultOpExecutioner: Blas vendor: [CUBLAS]
INFO executioner.CudaExecutioner: Device name: [GeForce GTX 1050 Ti]; CC: [6.1]; Total/free memory: [4233625600]
INFO memory.MemoryStore: Block broadcast_1 stored as values in memory (estimated size 72.9 KB, free 3.9 GB)
INFO rdd.HadoopRDD: Input split: hdfs://172.17.230.254:9000/user/trainData/RDD1/part-00001:268435456+134217728
INFO broadcast.TorrentBroadcast: Started reading broadcast variable 0
```

Figure 6.1: stderr log page for an executor during launching

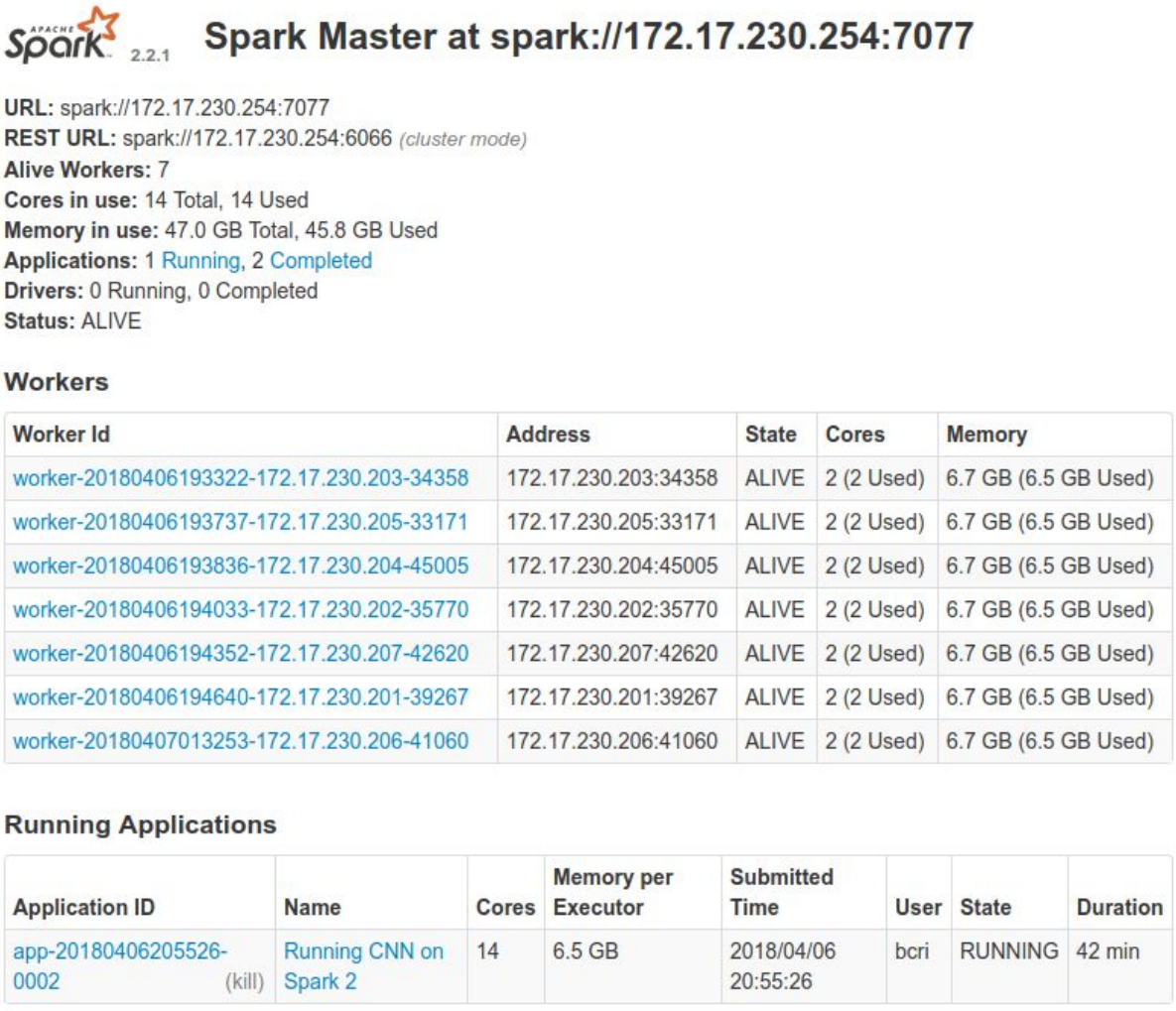
The output of Spark master during launching process is different from outputs of executors. There is no GPU device on Spark master, and due to this, the spark

master gives CPU backend a higher probability for ND4J operations. Figure 6.2 shows Spark master logs during the launching process.

```
INFO factory.Nd4jBackend: Loaded [CpuBackend] backend
INFO reflections.Reflections: Reflections took 623 ms to scan 222 urls, producing 150119 keys and 168141 values
INFO nativeblas.NativeOpsHolder: Number of threads used for NativeOps: 12
INFO reflections.Reflections: Reflections took 470 ms to scan 1 urls, producing 31 keys and 227 values
INFO nativeblas.Nd4jBlas: Number of threads used for BLAS: 12
INFO executioner.DefaultOpExecutioner: Backend used: [CPU]; OS: [Linux]
INFO executioner.DefaultOpExecutioner: Cores: [24]; Memory: [14.2GB];
INFO executioner.DefaultOpExecutioner: Blas vendor: [MKL]
```

Figure 6.2: stderr log page for Spark master during launching

Once all Spark executors are launched spark master calculates the available number of cores and based on that assigns a task to each core in each executor. Figure 6.3 shows how Spark UI looks like after all workers have been used by Spark master to execute the submitted job



Spark 2.2.1 **Spark Master at spark://172.17.230.254:7077**

URL: spark://172.17.230.254:7077
REST URL: spark://172.17.230.254:6066 *(cluster mode)*
Alive Workers: 7
Cores in use: 14 Total, 14 Used
Memory in use: 47.0 GB Total, 45.8 GB Used
Applications: 1 Running, 2 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20180406193322-172.17.230.203-34358	172.17.230.203:34358	ALIVE	2 (2 Used)	6.7 GB (6.5 GB Used)
worker-20180406193737-172.17.230.205-33171	172.17.230.205:33171	ALIVE	2 (2 Used)	6.7 GB (6.5 GB Used)
worker-20180406193836-172.17.230.204-45005	172.17.230.204:45005	ALIVE	2 (2 Used)	6.7 GB (6.5 GB Used)
worker-20180406194033-172.17.230.202-35770	172.17.230.202:35770	ALIVE	2 (2 Used)	6.7 GB (6.5 GB Used)
worker-20180406194352-172.17.230.207-42620	172.17.230.207:42620	ALIVE	2 (2 Used)	6.7 GB (6.5 GB Used)
worker-20180406194640-172.17.230.201-39267	172.17.230.201:39267	ALIVE	2 (2 Used)	6.7 GB (6.5 GB Used)
worker-20180407013253-172.17.230.206-41060	172.17.230.206:41060	ALIVE	2 (2 Used)	6.7 GB (6.5 GB Used)

Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20180406205526-0002 (kill)	Running CNN on Spark 2	14	6.5 GB	2018/04/06 20:55:26	bcri	RUNNING	42 min

Figure 6.3: Spark UI display necessary information about the Spark environment after a job is submitted to the Spark cluster.

TaskSetManager is a Spark master component for scheduling tasks tracking each task and handling tasks if they fail. TaskSetManager schedules the tasks where each executor is assigned to handle one task for one RDD partition. Figure 6.4 shows hows tasks are scheduled for first 14 RDD partitions.

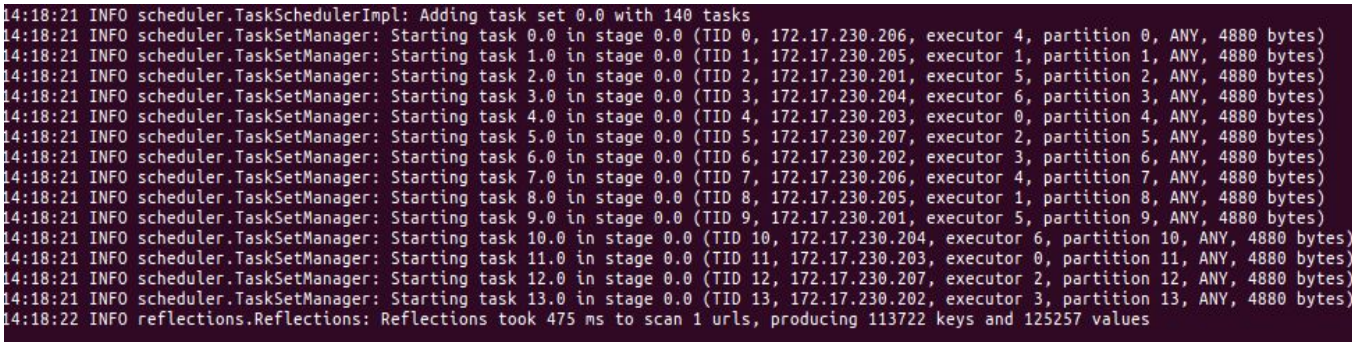


Figure 6.4: Starting 14 tasks to handle first 14 RDD partitions

Each executor can only execute two tasks at the same time as illustrated in Figure 6.4, which means the total number of task could be executed at the same time is 7x2 = 14 (number of executors * number of cores for an executor). When an executor receives a task, it loads the required RDD partitions directly from HDFS and distributes the task execution between CPU and GPU.

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(8)	960	1.7 GB / 39.5 GB	0.0 B	14	14	0	969339	969353	212.3 h (86.3 h)	53.4 GB	466.8 GB	649.5 GB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(8)	960	1.7 GB / 39.5 GB	0.0 B	14	14	0	969339	969353	212.3 h (86.3 h)	53.4 GB	466.8 GB	649.5 GB	0

Executors

Show 20 entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	10.10.230.254:44669	Active	96	207.8 MB / 10.5 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
0	172.17.230.207:33495	Active	105	207.9 MB / 4.2 GB	0.0 B	2	2	0	105594	105596	27.6 h (11.6 h)	6.7 GB	67.3 GB	91.2 GB	stdout stderr	Thread Dump
1	172.17.230.205:46446	Active	102	207.8 MB / 4.2 GB	0.0 B	2	2	0	147180	147182	37.8 h (14.7 h)	8.1 GB	66.7 GB	93.4 GB	stdout stderr	Thread Dump
2	172.17.230.204:34762	Active	130	208.1 MB / 4.2 GB	0.0 B	2	2	0	138183	138185	40.7 h (15.6 h)	7.9 GB	67.4 GB	92.5 GB	stdout stderr	Thread Dump
3	172.17.230.202:36624	Active	126	208 MB / 4.2 GB	0.0 B	2	2	0	143901	143903	20.8 h (9.2 h)	8.1 GB	66 GB	92.9 GB	stdout stderr	Thread Dump
4	172.17.230.206:39660	Active	128	208 MB / 4.2 GB	0.0 B	2	2	0	137798	137800	28.7 h (11.8 h)	7.4 GB	66.5 GB	93.5 GB	stdout stderr	Thread Dump
5	172.17.230.201:42941	Active	123	208.1 MB / 4.2 GB	0.0 B	2	2	0	134250	134252	31.6 h (12.9 h)	7.1 GB	66.4 GB	92.4 GB	stdout stderr	Thread Dump
6	172.17.230.203:39400	Active	150	208.2 MB / 4.2 GB	0.0 B	2	2	0	162433	162435	25.1 h (10.5 h)	8 GB	66.4 GB	93.6 GB	stdout stderr	Thread Dump

Figure 6.4: spark executors executing 14 tasks at time same time

6.3 GPUs Performance

Spark executors distribute the execution of each task between CPU and GPU. all linear algebra code are sent to GPU via ND4J, and the sequential code is executed on CPU sequential.

Each executor is configured to load the required RDD partition directly from HDFS instead of waiting for the data to be sent by the master node. This configuration reduces the communication between executors and Master node and forces executors to do more computations which decrease the amount of the time where GPUs are idle. Figure 6.5 shows GPU utilization and used memory in one Spark worker node.

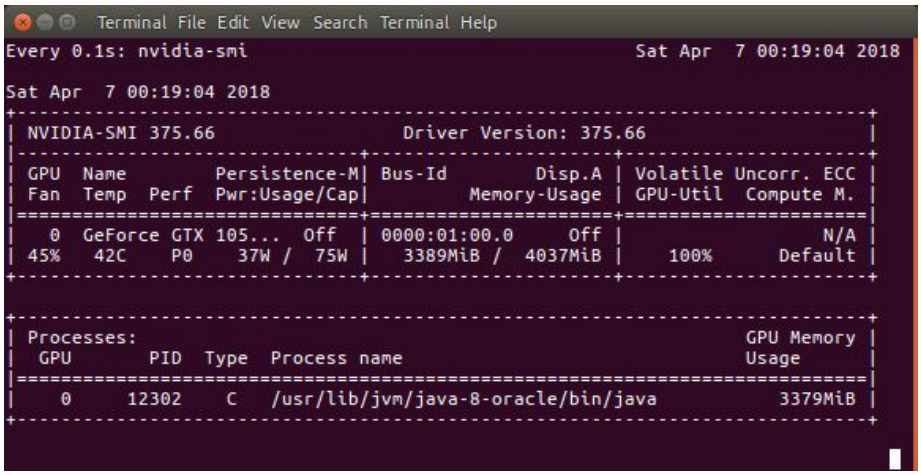


Figure 6.5: GPU utilization and used memory in one Spark node

Spark executors train their local CNN model with three mini-batches and send its model's parameters to the master node where all parameters are averaged and redistributed across the cluster. This process is repeated for every three mini-baches, and it is an expansive process, it pauses all the executors until all parameters are averaged, and new parameters are broadcasted. While executors wait for new parameters to arrive from master node all CPUs stays idle. Figure 6.6 shows the performance of one GPU in 5 seconds window. In average 92% memory is used, and 54% GPU is busy over time.

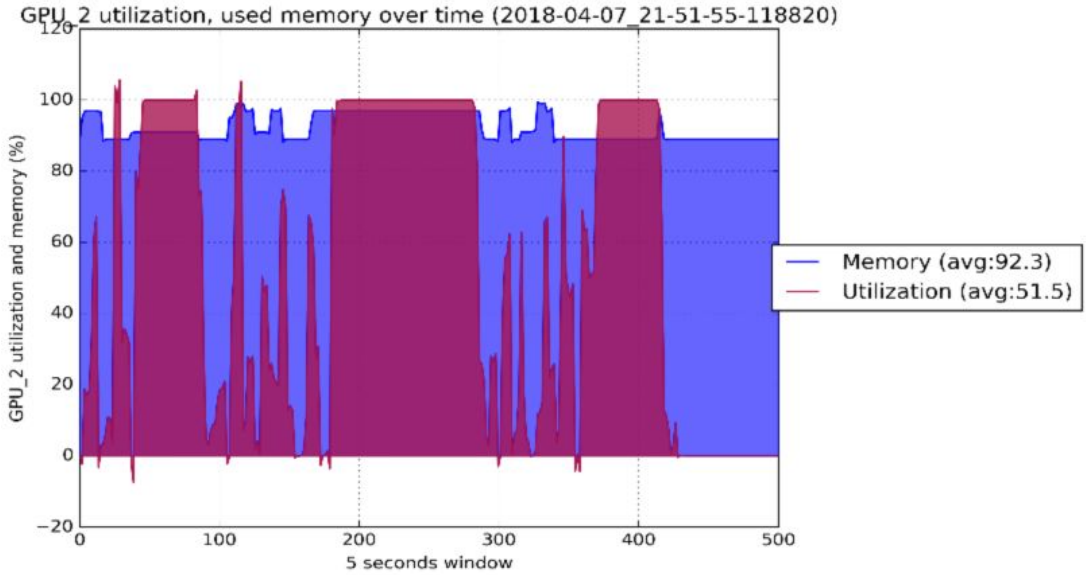


Figure 6.6: Performance of one GPU in 5 second, 92% memory used, 51% busy over time

6.4 Training CNN Model

Training a CNN model is an iterative process, and various decisions need to be made regarding the hyperparameters used such as learning rate, number of layers, number of neurons, number of epochs and the size of mini-batches. To obtain good performance, we train a CNN model with many combinations of hyperparameters and each time the model is trained it needs to be validated to measure its performance. Once a high performance is obtained, we rely on last used hyperparameters.

Dl4j provide a user interface to visualize the current network status and the process of training in real time. The user interface is typically used for selecting hyperparameters to retune neural networks to obtain good performance for a network and monitoring the performance of the model something unusual happens.

Dl4j UI support Spark, each executor sends its model's current status and the process of learning to the master node which visualizes this information on a web browser. UI is accessed via a browser with is URL:
http://bcri-1050ti-bm-master:9001/train
Information is collected from each executor and routed to UI when the fit method on a network is called to start the training process.

To obtain a good CNN model for object recognition the model needs to be trained with a big data containing a massive amount of images. The original size of the given training data is ~ 78GB, after generating more images from existing images (Data augmentation) and create RDDs from them, the size of the training data become ~114GB (11.5 million images).
RDDs for training and testing process are created, serialized and distributed across the Spark cluster where each Spark node have a full copy of each RDD. Figure 6.7 shows the size of RDDs in spark worker nodes.

Node	Last contact	Admin State	Capacity	Used	Non DFS Used	Remaining	Blocks	Block pool used	Failed Volumes	Version
bcri-1050ti-bm-1:50010 (172.17.230.201:50010)	1	In Service	450.06 GB	114.69 GB	201.14 GB	111.34 GB	20943	114.69 GB (25.48%)	0	2.7.5
bcri-1050ti-bm-2:50010 (172.17.230.202:50010)	2	In Service	450.06 GB	114.69 GB	149.49 GB	162.99 GB	20943	114.69 GB (25.48%)	0	2.7.5
bcri-1050ti-bm-4:50010 (172.17.230.204:50010)	2	In Service	450.06 GB	114.69 GB	138.47 GB	174.02 GB	20943	114.69 GB (25.48%)	0	2.7.5
bcri-1050ti-bm-6:50010 (172.17.230.206:50010)	1	In Service	450.06 GB	114.69 GB	136.07 GB	176.41 GB	20943	114.69 GB (25.48%)	0	2.7.5
bcri-1050ti-bm-3:50010 (172.17.230.203:50010)	0	In Service	450.06 GB	114.69 GB	150.05 GB	162.43 GB	20943	114.69 GB (25.48%)	0	2.7.5
bcri-1050ti-bm-7:50010 (172.17.230.207:50010)	2	In Service	450.06 GB	114.69 GB	134.73 GB	177.75 GB	20943	114.69 GB (25.48%)	0	2.7.5
bcri-1050ti-bm-5:50010 (172.17.230.205:50010)	0	In Service	450.06 GB	114.69 GB	141.94 GB	170.54 GB	20943	114.69 GB (25.48%)	0	2.7.5

Figure 6.7: "Block pool used" column represent the size of RDDs in Spark worker nodes

Dl4j UI shows model's scores in each iteration (mini-batch), these scores represent the errors (cost or loss) which model makes during its learning process. The model's errors decrease over the time if the model is learning from the training data. Figure 6.8 shows the model's score at each iteration.

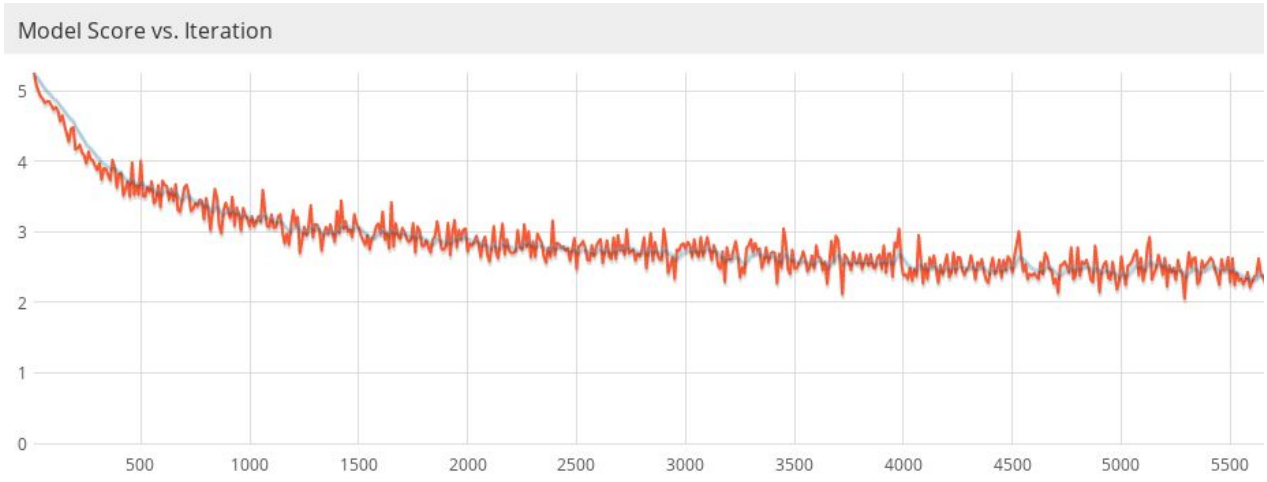


Figure 6.8: model's score in each iteration

Based on figure 6.8, the training process is stable and model makes less errors over time. If there is an increase in the score, then the most likely problem is too large learning rate.

The other charts in Dl4j UI allow monitoring the training process. The parameter ratios chart in figure 6.9 shows the change in the parameters between each iteration, if that change is below -6.0, then the model is not learning anything.

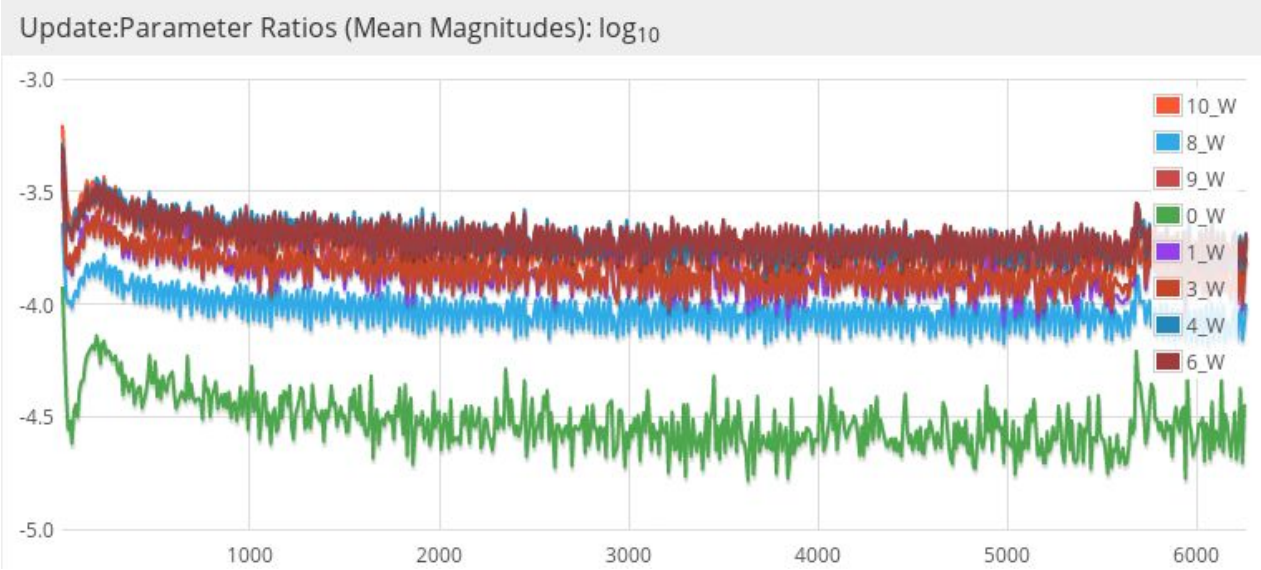
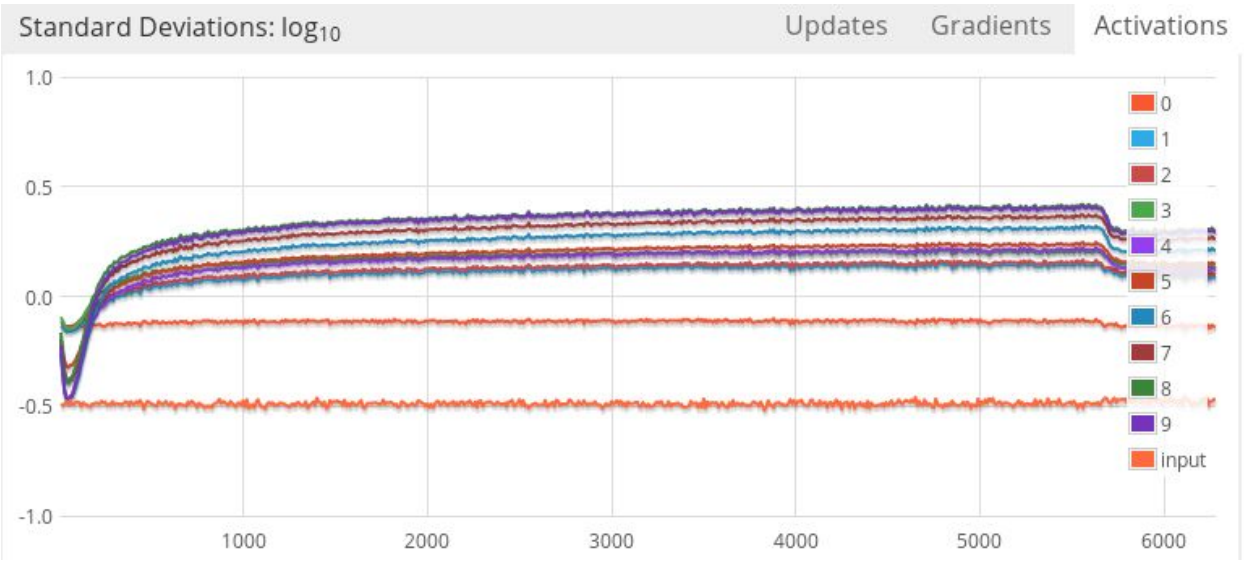


Figure 6.9: learning process in each iteration

To detect if a change in activation function input does not change its output (vanishing and exploding activations) the activation score in the dl4j UI Standard Deviation chart should not keep growing until it reaches infinity, as figure 6.10 shows.



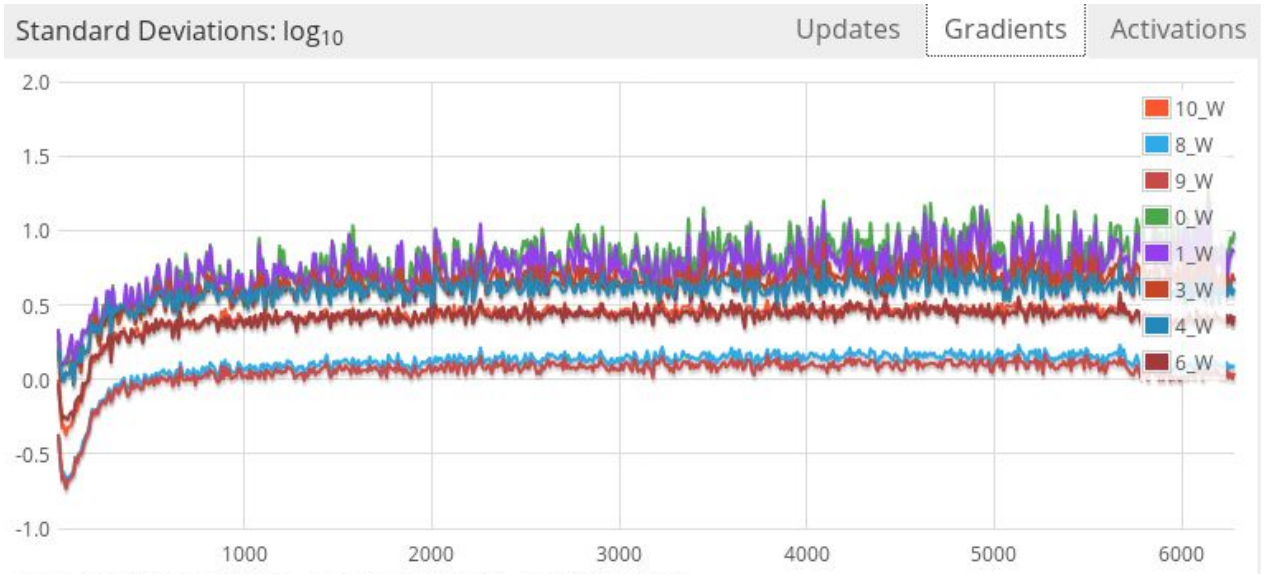


Figure 6.10: changing in activations and gradients

6.5 Evaluating CNN performance

Evaluation is the last step in training a deep learning model where the model is tested, and its performance is measured. This step is related to all the preceding steps to some degree. If a model is too specific, then it overfits to the data used for training, and this makes the model to not perform well on a new data. The model can be too generic, meaning that it underfits the data used for training.

After a model is built and trained, it has to pass the evaluation step, and if it passes, then the model can be trusted. DL4j offers an Evaluation class for finding how much a trained model gets a certain type of prediction correct more often than the other. The evaluation class measures a model's performance by building a Confusion Matrix (a table of confusion) which is a table of rows and columns that represent the predictions and the actual values for a model. Confusion Matrix is used to better understanding how well a model is performing based on giving the correct answers at the appropriate time. Figure 6.11 shows the structure of the Confusion Matrix.

	Positive predicted	Negative predicted
Positive Actual	True Positive (TP)	False Negative (FN)
Negative Actual	False Positive (FP)	True Negative (TN)

Figure 6.11: structure of the Confusion Matrix

The terms in Confusion Matrix are:

- True positives: the number of correct predictions that an instance is negative.
- True negatives: the number of incorrect predictions that an instance is positive
- False positives: the number of incorrect of predictions that an instance negative
- False negatives: the number of correct predictions that an instance is positive

Based on Confusion Matrix terms the percentage of a model's correct guesses can be calculated as follows:

- Accuracy: Overall, how often is a model making correct predictions?
  Accuracy = (TP + TN) / (TP + TN + FP + FN)
- precision: when a model predict yes, how often this prediction is correct?
  precision = TP / (TP + FP)
- recall: When the actual value is yes, how often does a model predict yes
  recall = TP / (FP + FN)
- F1 score: is a is a weighted average of the precision and recall score
  F1 score = 2TP / (2TP + FP + FN)

Figure 6.12 is the result of the CNN model trained with all training data and with following hyperparameters:

Number of layers	11
Total Parameters	1922023
Learning rate	0.01
Number of epochs	20

```
==========================Scores========================================
 # of classes:    143
 Accuracy:        0.8396
 Precision:       0.8443
 Recall:          0.8396
 F1 Score:        0.8485
========================================================================
18/04/03 23:11:25 INFO CnnOnSparkGPU:

Dl4j===> ************************************
18/04/03 23:11:25 INFO CnnOnSparkGPU: Dl4j===> ****finished, total time = 22536.777250000000354 minutes***
18/04/03 23:11:25 INFO CnnOnSparkGPU:
```

Figure 6.12: the performance of the CNN model trained with all training dataset.

6.6 CPU vs GPU performance

The difference between GPUs and CPUs performance can be measured by training the same model on CPUs and GPUs with same training data.
Figure 6.13 shows the required time to train a CNN model with 70MB (15,500 images) training data and number of epochs 1.

CPU time

```
==========================Scores========================================
 # of classes:    2
 Accuracy:        0.5656
 Precision:       0.5718
 Recall:          0.5656
 F1 Score:        0.5738
========================================================================
18/04/15 14:45:55 INFO CnnOnSparkCPU:

Dl4j===> ************************************
18/04/15 14:45:55 INFO CnnOnSparkCPU: Dl4j===> ****finished, total time = 31.303350000000624 minutes***
18/04/15 14:45:55 INFO CnnOnSparkCPU: Dl4j===> Epoch_0 time = 30.27996666666727 minutes
18/04/15 14:45:55 INFO CnnOnSparkCPU:
```

GPU time

```
==========================Scores========================================
 # of classes:    2
 Accuracy:        0.4541
 Precision:       0.4144
 Recall:          0.4540
 F1 Score:        0.5927
========================================================================
18/04/15 14:07:53 INFO CnnOnSparkGPU:

Dl4j===> ************************************
18/04/15 14:07:53 INFO CnnOnSparkGPU: Dl4j===> ****finished, total time = 2.4998166666667165 minutes***
18/04/15 14:07:53 INFO CnnOnSparkGPU: Dl4j===> Epoch_0 time = 2.380383333333381 minutes
18/04/15 14:07:53 INFO CnnOnSparkGPU:
```

Figure 6.13: CPU time and GPU time for training a model with 15,500 images.

In Figure 6.13, comparing GPUs training time with CPUs training time, GPUs are 12.5X time faster.

to obtain the real difference between GPUs and CPUs performance in a distributed deep learning environment, the CNN model is trained again with same training data size for 5 epochs:
Training the model with: training data = 70MB, epochs = 5.

CPU time

```
==========================Scores========================================
 # of classes:    2
 Accuracy:        0.6544
 Precision:       0.7417
 Recall:          0.6543
 F1 Score:        0.7343
========================================================================
18/04/15 19:18:17 INFO CnnOnSparkCPU:

Dl4j===> ************************************
18/04/15 19:18:17 INFO CnnOnSparkCPU: Dl4j===> ****finished, total time = 229.32676666667123 minutes***
18/04/15 19:18:17 INFO CnnOnSparkCPU: Dl4j===> Epoch_0 time = 24.012166666667145 minutes
18/04/15 19:18:17 INFO CnnOnSparkCPU: Dl4j===> Epoch_1 time = 61.88400000000124 minutes
18/04/15 19:18:17 INFO CnnOnSparkCPU: Dl4j===> Epoch_2 time = 47.389350000000945 minutes
18/04/15 19:18:17 INFO CnnOnSparkCPU: Dl4j===> Epoch_3 time = 43.391650000000865 minutes
18/04/15 19:18:17 INFO CnnOnSparkCPU: Dl4j===> Epoch_4 time = 50.60230000000101 minutes
18/04/15 19:18:17 INFO CnnOnSparkCPU:
```

GPU time

```
==========================Scores========================================
 # of classes:    2
 Accuracy:        0.8124
 Precision:       0.8146
 Recall:          0.8123
 F1 Score:        0.8200
========================================================================
18/04/15 19:28:58 INFO CnnOnSparkGPU:

Dl4j===> ************************************
18/04/15 19:28:58 INFO CnnOnSparkGPU: Dl4j===> ****finished, total time = 8.13151666666683 minutes***
18/04/15 19:28:58 INFO CnnOnSparkGPU: Dl4j===> Epoch_0 time = 4.158583333333416 minutes
18/04/15 19:28:58 INFO CnnOnSparkGPU: Dl4j===> Epoch_1 time = 0.9890166666666864 minutes
18/04/15 19:28:58 INFO CnnOnSparkGPU: Dl4j===> Epoch_2 time = 0.8967500000000179 minutes
18/04/15 19:28:58 INFO CnnOnSparkGPU: Dl4j===> Epoch_3 time = 0.8966500000000179 minutes
18/04/15 19:28:58 INFO CnnOnSparkGPU: Dl4j===> Epoch_4 time = 1.00635000000002 minutes
18/04/15 19:28:58 INFO CnnOnSparkGPU:
```

Following table represent CPU and GPU performance with a specific training data size and number of epochs:

Training data size	Number of epochs	Training Time CPU	Training Time GPU	GPU Speed Up
70MB (15,500 images)	1	31.30 minutes	2.49 minutes	12.5X

70MB (15,500 images)	5	229.3 minutes	8.13 minutes	28.2X

In CPU backend: in each epoch Spark creates RDDs repartition them and distributed them which increase training time.
In GPU backend: avoiding RDDs creation and repartition overheads in each epoch which decrease the training time.

Since training a deep learning model is an iterative process, the model should be train multiple time with the same training dataset the results presented in above table indicate that the approach take can reduce the execution time compared to using CPU alone by a factor of 28.

7. CONCLUSION

Accelerating Deep Learning on an Apache Spark Environment using GPUs is an art as a science requires a set of libraries for numerical computation, moving data to GPUs, performing computation on GPUs and distributing a task execution between CPUs and GPUs. These libraries need to be configured well in the Spark environment to obtain high performance.

The result presented in this report showed how a Spark environment could handle the execution of a distributed deep learning job and reduce the execution time compared to using CPU alone by a factor of 28.

The execution time can be reduced more by upgrading the hardware limitation such as increasing the memory in each worker node which decreases garbage collection service (~40% of the execution time is spent on garbage collection). Using faster accelerators such as Nvidia Tesla allows the Spark environment to handle much more complex deep learning application.

Finally, the Spark environment is configured to reduce the amount of communication between nodes and increase the amount of computation, in the average Spark environment keeps GPUs busy 50% over time. To increase GPUs usage Spark worker nodes has to do more less communication and this can be done by increasing the number of cores in each Spark worker node.

8. REFERENCES

- https://spark.apache.org/docs/latest/configuration.html
- Bill Chambers, spark, Spark - The Definitive Guide by
- Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional, 2010.
- Souza, Alan MF, and Fabio M. Soares. Neural network programming with Java. Packt Publishing Ltd, 2016.
- https://www.kaggle.com/c/cdiscount-image-classification-challenge/data
- https://deeplearning4j.org/spark
- J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.
- http://spark.apache.org/research.html
- http://hadoop.apache.org
- https://deeplearning4j.org/gpu
- https://nd4j.org/
- http://cs231n.stanford.edu/syllabus.html
- Reese R.M., Reese J.L., Grigorev A. - Java Data Science Made Easy - 2017