Software Abstractions for Large-Scale Deep Learning Models in Big Data Analytics

Ayaz H. Khan[1], Ali Mustafa Qamar[2], Aneeq Yusuf[3], Rehanullah Khan[4]
College of Computing and Information Sciences[1,3]
Karachi Institute of Economics and Technology, Karachi, Pakistan
College of Computer, Qassim University[2,4]
Mulaidah, Saudi Arabia

***Abstract*—The goal of big data analytics is to analyze datasets with a higher amount of volume, velocity, and variety for large-scale business intelligence problems. These workloads are normally processed with the distribution on massively parallel analytical systems. Deep learning is part of a broader family of machine learning methods based on learning representations of data. Deep learning plays a significant role in the information analysis by adding value to the massive amount of unsupervised data. A core domain of research is related to the development of deep learning algorithms for auto-extraction of complex data formats at a higher level of abstraction using the massive volumes of data. In this paper, we present the latest research trends in the development of parallel algorithms, optimization techniques, tools and libraries related to big data analytics and deep learning on various parallel architectures. The basic building blocks for deep learning such as Restricted Boltzmann Machines (RBM) and Deep Belief Networks (DBN) are identified and analyzed for parallelization of deep learning models. We proposed a parallel software API based on PyTorch, Hadoop Distributed File System (HDFS), Apache Hadoop MapReduce and MapReduce Job (MRJob) for developing large-scale deep learning models. We obtained about 5-30% reduction in the execution time of the deep auto-encoder model even on a single node Hadoop cluster. Furthermore, the complexity of code development is significantly reduced to create multi-layer deep learning models.**

Keywords*—Big data; deep learning; deep auto-encoders; Restricted Boltzmann Machines (RBM)

I. Introduction

Big volumes of data have been started to accumulate based on the advancements in sensor technology, the Internet, social networks, wireless communication, and inexpensive memory in various formats such as numerical, textual, and image. Such a high volume of data can be analyzed using statistical and Computational Intelligence (CI) tools based on neuro-computing, fuzzy logic, clustering, Bayesian networks, Principal Component Analysis (PCA), etc. for an efficient data management by reducing its size and developing non-parametric models based on extracted information and its knowledge base [1]. These workloads are normally processed with a distribution on massively parallel analytical systems. GPUs (Graphics Processing Units), MICs (Many Integrated Cores) or FPGAs (Field Programmable Gate Arrays) etc. are available as co-processors to accelerate the required computations in various algorithms with the distribution of data among the processors and co-processors to support bigger workloads.

Deep learning can be considered as an extension to Machine learning methods for learning data representations [2]. In the area of image classification, face detection and recognition, the features of an image can be represented in various ways based on pixel intensity values, set of edges, specific regional shapes. Deep learning provides efficient algorithms for unsupervised or semi-supervised feature learning and extraction. Several deep learning architectures including convolutional deep neural networks, Recurrent Neural Networks (RNN), and Deep Belief Networks (DBN) are applicable to the various fields of computer science such as speech recognition, computer vision, natural language processing, and bioinformatics to produce state-of-the-art analytical results. Deep learning is an active research area both in industry and academia to solve various practical examples such as image and speech recognition, neural machine translation, traffic management, and cancer detection. Furthermore, it has been successfully applied in task classification, object detection, motion modeling, dimensionality reduction, and network flow prediction [3]. Various software solutions have been provided in the market for deep learning using different parallel computing architectures including programming language extensions, libraries, frameworks.

In order to ease the development of deep learning models for big data analytics, there is a dire need of a software API with high level abstractions to create multi-layer deep learning models with the capability of processing big training data that is in high volume, velocity and variety. We explored several parallel algorithms, optimization techniques, tools and libraries related to big data analytics and deep learning on various parallel architectures. Based on our exploration and analysis, we identified the basic building blocks for the parallelization of the deep learning models and proposed a parallel software API using PyTorch, Hadoop Distributed File System (HDFS), Apache Hadoop MapReduce (MR) and MapReduce Job (MRJob) for developing large-scale deep learning models. We obtained about 5-30% reduction in the execution time of the deep auto-encoder model even on a single node Hadoop cluster. Furthermore, the complexity of code development is significantly reduced to create multi-layer deep learning models.

The rest of the paper is organized as follows: Section II presents a summary of the latest research trends in using deep learning approaches for big data analytics, Section III reviews the current software tools and libraries in the domain of deep learning, Section IV provides an in-depth analysis of the basic building blocks for deep learning in big data analytics, Section V explains the proposed API for the development

of deep learning models, whereas Section VI presents the evaluation of the proposed API in terms of performance and its usage. Lastly, Section VII concludes the paper and highlights future research directions.

II. Literature Review

Deep learning is a branch of Machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using multiple processing layers, with complex structures or otherwise, composed of multiple non-linear transformations. Deep learning is part of a broader family of Machine learning methods based on learning representations of data. An observation (e.g. an image) can be represented in many ways, such as a vector of intensity values per pixel, or in a more abstract way as a set of edges, regions of a particular shape, etc. Some representations are better than others at simplifying the learning task (e.g., face recognition or facial expression recognition) from examples. In recent years, deep learning approaches have gained significant interest because while processing unstructured data, it doesn't require to label everything to discover patterns. It uses big data, and the computational power of the GPU to gain speed and accuracy [4]. One of the promises of deep learning is replacing handcrafted features with efficient algorithms for unsupervised or semi-supervised feature learning and hierarchical feature extraction. Deep learning can achieve outstanding results in various fields. However, it requires so significant computational power that massively parallel processors and/or numerous computers are often required for practical applications. Deep learning algorithms are based on distributed representations. The underlying assumption behind distributed representations is that the observed data is generated by the interactions of factors organized in layers. Deep learning adds the assumption that these layers of factors correspond to various levels of abstraction or composition. Varying numbers of layers and layer sizes can be used to provide different amounts of abstraction. Deep learning exploits this idea of hierarchical explanatory factors, where more abstract higher levels are learned from the lower level ones. Deep learning helps to disentangle these abstractions and pick out which features are useful for learning. For supervised learning tasks, deep learning methods obviate feature engineering, by translating the data into compact intermediate representations akin to principal components, and derive layered structures which remove redundancy in the representation. Many deep learning algorithms are applied to unsupervised learning tasks. This is an important benefit because unlabeled data is usually more abundant than labeled data. An example of a deep structure that can be trained in an unsupervised manner is a Deep Belief Network (DBN). A DBN is composed of a stack of Restricted Boltzmann Machines (RBMs). A core component of the DBN is a greedy, layer-by-layer learning algorithm, which optimizes the DBN weights at a time complexity linear to the size and depth of the networks. Separately and with some surprise, initializing the weights of an Multi-Layer Perceptron (MLP) with a correspondingly configured DBN often produces much better results than that with random weights. As such, MLPs with many hidden layers, or Deep Neural Networks (DNN), which are learned with unsupervised DBN pre-training followed by Back-Propagation fine-tuning is sometimes also called DBNs in the literature [4].

Several technologies and their correlations have been explored [5] to be useful in big data analytics for future volume prediction and deep knowledge of data. This helps in taking proactive and better strategic decisions in the business community focusing on unstructured data and open source technologies including Apache Flume, Apache Sqoop, Apache Pig, Apache Hive, Apache ZooKeeper, Mongo DB, Apache Cassendra, Apache Hadoop, MapReduce, Apache Splunk and Apache Spark. An in-depth analysis of different hardware platforms and related software frameworks suitable for big data analytics is presented in [6] based on various matrices, including fault tolerance, scalability, I/O bandwidth requirements, distributed and real-time processing. It has been found that the right choice of the platform should be based on proper investigation of the application/algorithm needs. The decision has to be made on the basis of the results' frequency requirements, the size of data to be processed, and the number of iterations to build a model. A case study of various implementations of K-means clustering algorithm has been presented taking consideration of various algorithmic and system level issues. The analytical results can be further strengthened by investigating other algorithms such as decision trees, nearest neighbors, page ranking, and etc. In order to develop highly scalable applications, a combination of multiple platforms can be utilized such as Hadoop as a horizontal scaling platform and GPUs as a vertical scaling platform to perform the analysis in real-time. Chen et al. [7] presented a review of technical challenges and the latest advances in the related technologies for the four phases of big data analytics that are data generation, data acquisition, data storage, and data analysis. Several open problems and future directions have been discussed in several representative applications such as enterprise management, Internet of Things (IoT), online social networks, medical applications, collective intelligence, and smart grid. An end-to-end big data benchmark, BigBench [8], has been proposed by addressing the variety, velocity, and volume aspects of big data systems in the domain of product retail businesses with physical and online stores. The benchmark contains the structured data adopted from the TPC-DS benchmark, semi-structured data captured from the user responses on the retailers' websites, and unstructured data captured from the online product reviews. The benchmark has been designed to generate the data upon a set of queries based on the source of data, types of query processing, and techniques used in analysis as three data dimensions. The response time feasibility of BigBench has been evaluated on Teradata Aster Database with 200 Gigabyte of big data set and executing queries developed using Teradata Aster SQL-MR. Further evaluation of the BigBench is planned to be done on one of the Hadoop eco-systems like HIVE.

Wang [9] proposed a method to process network traffic streaming data with unknown protocol using neural network and deep learning approaches. The proposed method can be applied on feature learning, protocol classification, anomalous protocol detection and unknown protocol identification. The method is beneficial in comparison to the traditional methods that have poor adaptation and are difficult to automate. Agneeswaran et al. [10] reviewed three generations of tools/paradigms for iterative machine learning algorithms in the context of big data analytics. The third generation tools/paradigms such as *Spark* and *GraphLab* were found to be the

most promising in the implementation of the large number of machine learning algorithms in terms of horizontal scalability. It has been identified that more sophisticated paradigms such as *Bulk Synchronous Parallel* (BSP) based paradigms and graph processing paradigms need to be considered in the implementation of a number of machine learning algorithms in addition to Hadoop's Map-Reduce paradigm for big data analytics. Bengio [11] examined the scalability issues of deep learning algorithms for larger models and datasets to develop more efficient and powerful inference and sampling procedures with reduced optimization difficulties. Enhancements in training deep learning algorithms have been achieved [12] using more sophisticated optimization methods including Limited memory BFGS (L-BFGS) and Conjugate Gradient (CG) with linear search instead of using Stochastic Gradient Descent Methods (SGDs) as the traditional approach. The experiments have been performed by considering both algorithmic extensions such as sparsity regularization and hardware extensions such as GPUs or Computer Clusters. The use of L-BFGS in convolutional network model obtained 0.69% set test error on the standard MNIST dataset which is a state-of-the-art result among other related algorithms. However, L-BFGS was found to be highly competitive to SGDs/CG for dimensional problems. Significant performance improvements of L-BFGS and CG over SGDs have been observed with the use of sparse auto-encoders on GPUs. The performance trend is almost linear to the number of machines in use of locally connected networks and convolutional networks. Furthermore, it has been found that Map-Reduce framework can also be utilized in the computation of gradients using L-BFGS for locally connected networks or other networks with a relatively small number of parameters.

In terms of statistical analysis, machine learning, pattern recognition, data fusion, data mining, and numerical analysis, big data infrastructure and analytics are directly related to the traditional data sciences [13]. However, deep analysis of big data requires the use of massively parallel computing concepts with large numbers of high-end servers. Such an analysis has been performed on US DoD (Department of Defense) big data for pattern recognition, anomaly detection and data fusion using various methods like Lexical Link Analysis (LLA), System-Self Awareness (SSA), and Collaborative Learning Agents (CLA) as an unsupervised learning or deep learning. In order to satisfy the needs of traffic flow prediction in real-world applications, deep architecture models have been applied [3] on big traffic data with inherently spatial and temporal correlations. In this method, feature extraction of the generic traffic flow has been done using stacked auto-encoder while training has been performed in a greedy layer-wise fashion. The performance evaluation has been performed on PeMS dataset to compare with the BP NN, SVM, and RBF NN models. The proposed method is found to be superior than the other competing methods. Further investigation can be performed using other deep learning algorithms for traffic flow prediction with the application on different public open datasets to examine their effectiveness. Deep learning on big data has been applied [14] for complex pattern extraction, data tagging, semantic indexing, simplifying discriminative tasks, and fast information retrieval on an un-labeled and un-categorized raw dataset. The study highlights the usefulness of deep learning in solving specific problems of big data analytics while suggesting improvements in deep learning to overcome the challenges in big data analytics. The work can be extended to focus on other aspects of big data analytics that are variety and velocity of data, large-scale models, and distributed computing.

```python
import tensorflow as tf
import matplotlib.pyplot as plt

W = tf.Variable([0.3], dtype=tf.float32) #1
b = tf.Variable([-0.3], dtype=tf.float32) #2
x = tf.placeholder(tf.float32) #3
linear_model = W*x + b #4
y = tf.placeholder(tf.float32)

plt.scatter([1, 2, 3, 4], [0, -1, -2, -3]) #5
s = tf.Session()

init = tf.global_variables_initializer() #8
s.run(init) #9

squared_deltas = tf.square(linear_model - y)#10
loss = tf.reduce_sum(squared_deltas)#11

result = s.run(linear_model,{x: [1,2,3,4]})#12
result_loss = s.run(loss,{x: [1,2,3,4], y: [0,-1,-2,-3]})#12
print (result,result_loss)

plt.plot([1, 2, 3, 4], result*[0,-1,-2,-3],'r')#13
plt.show()
```

Fig. 1. Tensor Flow Linear Regression Model

III. Existing Solutions and Tools

This section reviews mostly used software tools and libraries in the domain of deep learning.

A. TensorFlow

TensorFlow (TF) is the most popular deep learning package on github [15]. TensorFlow is an interface for expressing machine learning algorithms on heterogeneous distributed systems [16]. The intent behind its development was to create a framework that supports scalable machine learning and an easy to use programming paradigm. Before starting TensorFlow, Google had used DistBelief as their first-generation machine learning system. The old generation did not have support for a large portion of hardware. The second generation of machine leaning framework, which is TensorFlow, does solve this problem and added more features. The advance in hardware, especially in GPU supported deep learning.

Fig. 1 shows a basic example of a linear regression model that takes a sample of training data and evaluates the model using a loss function. The loss function calculates the distance between the provided data and the model. Furthermore, TensorFlow provides *tf.train* API, which is an optimizer called gradient descent to reduce the loss in the model. TensorFlow operations have both CPU and GPU implementations; if TF finds a GPU device, then it automatically executes GPU implementations of the used operation instead of the CPU one. Moreover, it also provides an API function *device* to set a specific device for a certain code block.

B. PyTorch

PyTorch is a deep learning library for Python. It has been developed by Facebook and is mainly used for natural language processing. It has two high-level features: tensor computation

that comes with GPU acceleration and deep Artificial Neural Network (ANN) built with taped-based auto-grad system. In PyTorch, one can use the old python packages such as Numpy, cython, and Scipy to maximize the use of PyTorch. It provides *tensors* that can execute commands using either GPU or CPU, and speed up compute by a huge amount.

1) Advantages of using PyTorch:

1) The debugging process is easy, making it easier to understand and follow the code.
2) It has the same as well as some more features and layers that happen to be in Torch like (Grus, CONV1, 2, 3D; LSTMCONV 1, 2, 3D; LSTM, Unpool).
3) Could be a Numpy extension To GPUs.
4) It is fast and some consider it the fastest among other libraries *define by run* for example dynet and chainer.
5) With PyTorch, one can build a strong network structured by its computation.
6) In PyTorch, the overheat in the framework is minimal.
7) Making neural network is easy and requires no extensions.

PyTorch figured out a new way of building neural networks, using tape recorder and replay it. Other frameworks like Caffe, Theano, TensorFlow and CNTK use a static view. When they build a neural network, they have to use the same neural network and cannot change it; although they can but they have to start from the scratch. However, in PyTorch, there is a new way called *Reverse-mode auto-differentiation*, that allows the user to change the neural network and to modify without overheating or lags.

C. Caffe2

Caffe2 is a deep learning framework which is simple and helps to use the algorithms of new models. Using the GPU power, we can bring the creation to scale with Caffe2 libraries that support cross-platform operations. The operators are a basic computation unit of Caffe2. It is a flexible layers' version of Caffe since it comes with more than 400 different operators.

D. Comparison of Deep Learning Frameworks

TensorFlow is a powerful deep learning framework, with a lot of documentation and is good in visualization. Furthermore, it has the ability to build a strong model for many platforms. Therefore, TensorFlow is good in building a model for production, used to build models for mobile platforms, and has a good community support. On the other hand, PyTorch is relatively a newer framework and is growing up fast. For now, PyTorch is good for research and building products with the non-functional requirements, good for testing and debugging. PyTorch was designed with additional capabilities like the ability to trace and debug errors, and building a dynamic neural network. While the other frameworks like Tensorflow, Theano and Caffe use static neural networks, lack the ability to trace and debug errors, and may require more time in finding the errors. In addition, PyTorch is a new framework that happens to grow fast, and could in the near future use the same advantages as found in other frameworks, like having its own visualization. In short, PyTorch is a newer framework, that is more flexible than its competitors.

Caffe2 supports a large-scale deployment. It brings the Torch and itself together to support the multi-GPUs as it provides the same level of support. It can work on both single-host and multi-host GPUs workstations.

E. Customize Code Optimizations of Deep Learning Algorithms

Olas et al. [17], [18] presented the implementations of Restricted Boltzmann Machine (RBM) and Deep Belief Net (DBN) using Intel Xeon Phi CoProcessor (Many Integrated Core). The algorithms are fully implemented in C++ language using the OpenMP standard for parallelizing computation. The transformation of computations was performed in such a way that efficient implementations of matrix and vector operations available in the *BLAS* library can be utilized. For example, the operation of summing the elements of a matrix is replaced with a matrix-vector multiplication, where the vector contains all ones. All the codes are compiled using Intel C++ Compiler available in Intel Parallel Studio XE 2015 environment. Additionally, the Intel Math Kernel Library (MKL) is used for the efficient implementation of *BLAS* routines. Furthermore, in order to generate pseudo-random numbers in particular, the *SIMD-oriented Fast Mersenne Twister pseudo-random number generator VSL_BRNG_SFMT19937* is utilized.

IV. Building Blocks for Deep Learning in Big Data Analytics

Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently generated heat) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture [19]. In addition, GPU development during the last few years has contributed to a growth in the concept of deep learning. Parallel computing in deep learning in its *natural form* would mean improvements in training time from months to weeks or days. Deep learning has many different algorithms such as auto-encoders, denoising auto-encoders, stacked denoising auto-encoders, Restricted Boltzmann Machines (RBM), Deep Belief Networks (DBN).

A. Restricted Boltzmann Machines (RBMs)

RBM was invented by Geoff Hinton. This algorithm can automatically find the patterns in data by reconstructing inputs. It is used in dimensional reduction, classification, regression, collaborative filtering, feature learning, and topic modeling. It is increasingly being used in supervised and unsupervised learning scenarios. The first layer of the RBM is called the visible or input layer, and the second is the hidden layer as shown in Fig. 2. Each circle in the graph represents a neuron-like unit called a node, and the calculations take place in the nodes. The nodes are connected to each other across layers, but no two nodes of the same layer are linked. As their name implies, RBMs are a variant of Boltzmann machines, with the restriction that their neurons must form a bipartite graph. By contrast, *unrestricted* Boltzmann machines may have connections between hidden units. Therefore, there is no intra-layer communication. This restriction allows for more efficient training algorithms than are available for the general

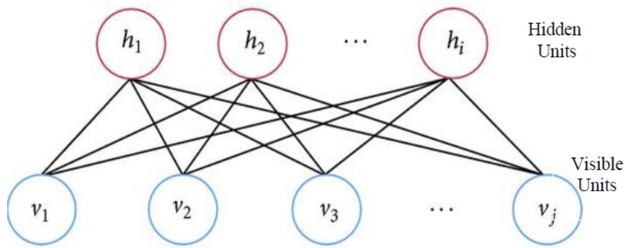

Fig. 2. Restricted Boltzmann Machine

class of Boltzmann machines, in particular the gradient-based contrastive divergence algorithm. Since the inputs from all visible nodes are being passed to all of the hidden nodes, RBM can be defined as a symmetrical bipartite graph [20].

1) RBM Implementation Structure: RBM consists of two steps. Each step has effect of different parameters:

1) *n_visible*: Number of visible units. It is used to train per one iteration train_X (column size)
2) *n_hidden*: Number of hidden units
3) *train_n*: Sets number or iteration number of train_X (row size)

a) Training step (Contrastive Divergence function): Contrastive divergence is used to calculate the gradient (the slope representing the relationship between a network's weights and its error), without which no learning can occur. The parameters here are:

1) k: The number of times the contrastive divergence is run
2) *train_x*: Sample data used for training purpose
3) *input[j]*: or visible units is a sample from the training distribution (it's one row of train_x) for the RBM (vector size is # of n_visible)
4) *learning_rate*: Like momentum, affects how much the neural net adjusts the coefficients on each iteration as it corrects for errors. This parameter helps to determine the size of the steps, the net takes down the gradient towards a local optimum. A large learning rate will make the net learn fast, and may overshoot the optimum. A small learning rate will slow down the learning, which can be inefficient.
5) N: sample row size (# of train_N). Samples will be processed row by row. Each iteration will process the complete number of column (size is # of n_visible)

b) Testing step (Reconstruction): The dependent parameters here are:

1) *test_x*: Sample data that is used for testing purpose
2) *reconstructed_x*: variable which is used for sigmoid and the trained data

2) Principal Factor Analysis on RBM: We have applied Principal Factor Analysis (PFA), a well-known statistical method for finding the parameters that are affecting the performance of any system. For RBM implementation, the following factors (see Table I for execution times) were analyzed:

1) k: contrastive divergence steps
2) v: the number of visible neurons
3) h: the number of hidden neurons
4) N: training set dimension

Table II shows the ANOVA table as a result of Principal Factor Analysis of RBM factors. It shows that the factor, which strongly affects the performance of RBM is k, that represents the contrastive divergence steps. These steps are strongly sequential in execution. Therefore, parallelization will not benefit from this factor because of strong flow dependency of loop iterations. At the second level, the factor N (training set dimensions) is showing significant variations in execution time of RBM. Since the input samples (N) are processed independently in RBM storing the resulting weights on different indices, therefore, it is a good candidate for parallelization so as to gain significant performance improvement of deep learning models.

B. Deep Belief Nets (DBNs)

A Deep Belief Network (DBN) is a type of deep neural network, composed of multiple layers of latent variables (hidden units), with connections between the layers but not between units within each layer. As what has been introduced before, the most important use of RBM is as learning modules that are composed to form deep belief nets. RBMs are shallow, two-layer neural networks that constitute the building blocks of deep belief networks (see Fig. 3). It can be formed by *stacking* RBMs and optionally fine-tuning the resulting deep network with gradient descent and back-propagation. Therefore, DBN can be defined as a stack of Restricted Boltzmann machines (RBM). Each RBM layer communicates with both the previous and subsequent layers. The nodes of any single layer don't communicate with each other laterally. This stack of RBMs might end with a *Softmax*[1] layer to create a classifier, or it may simply help cluster unlabeled data in an unsupervised learning scenario. When trained on a set of examples in an unsupervised way, a DBN can learn to probabilistically reconstruct its inputs. The layers then act as feature detectors on inputs. After this learning step, a DBN can be further trained in a supervised way to perform classification. With the exception of the first and final layers, each layer in a deep-belief network has a double role: it serves as the *hidden layer* to the nodes that come before it, and as the input (or *visible*) layer to the nodes that come after. The reason of using DBN is to recognize, cluster and generate images, video sequences and motion-capture data. A continuous deep-belief network is simply an extension of a deep-belief network that accepts a continuum of decimals, rather than binary data [21].

MNIST is a good place to begin exploring image recognition and DBNs. The first step is to take an image from the dataset and to convert its pixels from continuous gray scale to binary. Typically, every gray-scale pixel with a value higher than 35 becomes a 1, while the rest are set to 0. The MNIST dataset iterator class performs this operation [21], [22].

[1]*Softmax* is a function used as the output layer of a neural network that classifies input. It converts vectors into class probabilities. It normalizes the vector of scores by first exponentiating and then dividing by a constant.

TABLE I. RBM EXECUTION TIME (MSEC) WITH DIFFERENT PARAMETERS

v →		6			12			24		
		h			h			h		
k	N	3	6	12	3	6	12	3	6	12
1	1000	1.58	2.48	4.41	2.47	3.88	6.73	4.36	6.70	11.53
	2000	3.13	5.01	8.72	4.96	7.82	13.48	8.87	13.52	22.96
	3000	4.67	7.51	13.01	7.49	11.68	20.34	13.19	20.08	34.18
5	1000	4.76	7.11	11.94	8.01	11.51	18.46	10.85	14.73	22.59
	2000	9.45	14.20	23.88	15.87	22.98	29.17	21.77	31.07	44.91
	3000	14.24	21.33	35.72	23.91	34.69	40.10	32.80	46.56	67.24
10	1000	6.72	9.92	15.93	11.33	15.77	24.22	20.51	27.34	40.34
	2000	13.57	19.69	31.95	22.52	31.30	48.29	40.84	54.77	80.88
	3000	20.14	29.49	47.81	33.86	47.15	72.48	61.63	81.63	121.31

TABLE II. RBM PRINCIPAL FACTOR ANALYSIS - ANOVA TABLE

Main Effects	% Variations
k	41.65
v	17.60
N	25.83
h	15.07

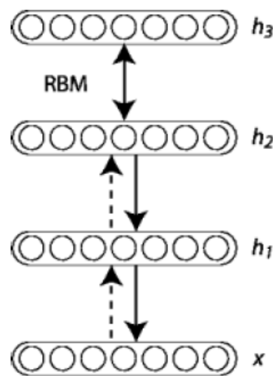

Fig. 3. Deep Belief Network

V. PROPOSED SOFTWARE ABSTRACTIONS FOR DEEP LEARNING MODELS

Based on our analysis of deep learning algorithms like RBM and DBN in Section IV-A2, it has been found that the performance of the algorithms is highly affected by the iterations on visible and hidden nodes which are strongly dependent on each other and are not suitable for parallel implementations. The only effective parallelization in these algorithms is to distribute the input samples among several workers (processes or threads) to obtain a significant speedup of deep learning model execution. Furthermore, there are several frameworks and available libraries that provide efficient implementation of these algorithms. In order to utilize these frameworks and libraries for developing large-scale deep learning models for big data analytics, we need to extend these tools to execute on multiple computing nodes where each node has a portion of input samples and runs the model in parallel. At the end, the final output of the learning process needs to be accumulated at the single (master) node. This requires an in-depth knowledge of writing parallel programs and concepts of data distribution. In order to ease the development of deep learning models for big data analytics, we propose a parallel software API as an extension of PyTorch with HDFS and MapReduce frameworks. The following sub-sections explain the tools used in the API, it's process flow, usage and functions.

A. Used Tools

a) PyTorch: is a scientific computing library, which is developed as a GPU-enabled alternative for *Numpy*. It is a deep-learning platform that provides both speed and flexibility. Much like other deep-learning libraries, PyTorch makes use of *Tensors* for storing data and training of neural networks [23]. In addition to this, PyTorch makes it fairly simple to create *computational graphs*, an important aspect of neural networks. Furthermore, the PyTorch library comes with an autograd function, so as to automate the process of calculating the gradient descent, whilst training a network. It is important to note here that loading data into PyTorch is a difficult process. The data needs to be converted into a form that can be read by the library's DataLoader class. Moreover, the CUDA library also needs to be installed and enabled in PyTorch for GPU computations.

b) Hadoop: is an open-source and Java-based framework, which allows distributed processing of datasets, across a cluster of connected computers [24]. The core components of the framework include Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop YARN and Hadoop MapReduce. In the following process, HDFS and MapReduce would be of key importance. One of the main reasons for Hadoop's popularity is its adherence to the principle that hardware failures of individual machines, within a cluster, should be handled automatically by the framework.

c) MRJob: is a module developed by *Yelp*, to create Hadoop MapReduce jobs in Python, instead of the conventional Java code [25]. As compared to other MapReduce libraries for Python, MRJob allows the users to keep the mappers and reducers, both, in a single class. Moreover, it allows users to define a multi-step mapper and switch input and output formats, with a single line of code.

B. API Process Flow

In order to develop a distributed neural network API, we constructed a simulation of a Restricted Boltzmann Machine, the most basic kind of neural network. Although the entire process was developed and tested using the MNIST dataset, the configuration of the network has been kept dynamic, so that the users of the API can alter the configurations as per their needs. It should be noted here that the API makes use of PyTorch and MRJob at the backend.

The API makes use of three modules, namely the *Neural Network Configuration* (NNC) module, *MapReduce Configuration* (MRC) module and the *Network Definition* (ND) module. The NNC module comprises of three classes, the

```
newnet=rbm(0.015, 0.8, 64)
MRDL.newnet=newnet
net=newnet.create_net(layer_count=4, node_seq=[784,500, 100, 10])
MRDL.net=net
MRDL.run()
iter1=newnet.combine_weights(MRDL.weights)
net=newnet.create_net(layer_count=4, node_seq=[784,500,100,10])
newnet.set_w(iter1, net)
```

Fig. 4. Code Example for Model Configuration of 784 → 500 → 100 → 10

net class, the *data_load* class and a *ToTensor* class, whereas the MRC module comprises of the *MR_dist* class and the ND module comprises of *RBM* class. It should be noted here that the net class inherits from PyTorch's *torch.nn.module* and overrides the feed-forward function.

Since everything has been well-defined and properly implemented throughout the API, the users just need to interact with the ND and MRC modules. The users need to create an object of the *rbm* class, to set the parameters for the network to follow. This includes the learning rate, the momentum and the batch size, respectively. Each of the three parameters need to be passed to the constructor of the class object. This object should be then passed to the *newnet* variable of the *MR_dist* class.

This would then be followed by defining the layers of the network, which can be done by calling the *create_net* function of the *rbm* class object, the number of layers and the list containing the number of nodes for each layer. This all should be passed as arguments to the function. The output of this function should be passed to the *net* variable of the *MR_dist* class.

It should be noted here that *create_net* function creates a new list, which contains the nodes of the network in linear configuration. Similarly, *Xavier* weights are specified for each layer, with the *relu function* being set as the default *gain function*. Furthermore, the feed forward function makes use of the logarithmic softmax function, as the default feed forward function.

The users then need to call the *run* function of the *MR_dist* class, which executes the distributed training of the RBM. Following the training, the users would need to combine the output from multiple reducers, for which they can call the *combine_weights* function of the *rbm* class object. The variable *weights* from the *MR_dist* class should be passed to the function as an argument. The user can then call the *set_w* function of the *rbm* class object, passing the output of the last function and the weights to be applied as arguments. This will create a network that has the trained weights and is ready to be used.

Note: The aforementioned process has been defined for a network of just 4 layers: 1 output, 2 hidden and 1 input layer. Subsequent hidden layers, if needed, can be added easily. For example, Fig. 4 shows the code for a configuration of 784 → 500 → 100 → 10.

In order to test the accuracy, the users would need to first call the *data_load* function of the *rbm* object and pass the path of the test data file as an argument. The output of this function should be saved in a variable. Finally, the users would just need to call the *test* function of the *rbm* object, passing the loaded data variable and the network as arguments to the function. The results of the test would then be printed as output on the user screen.

The addition of a *data_load* class and a *ToTensor* class, to the NNC module was necessary, so as to ensure that all data passed into the network is in a consistent and specified format. As such, the data being passed to the API needs to be in CSV format and should contain a label column and adjoining value columns. The reasons to choose this format, as the default data format for the API include:

- The fact that most of the data available for training models can easily be found in CSV formats.
- It is easy to convert data from different sources to a CSV format.
- Hadoop reads a file line by line and does not split the data from the middle of a line.

The *data_loader* object can be created by passing the absolute path of the file, as an argument to the class object. Additional data can also be passed to the object, as well as any transformations that need to be applied to the data. It should be noted here that the *data_loader* class inherits from PyTorch's *torch.utils.data.Dataset*. Moreover, to make individual instances of the data fetchable, as label and corresponding value, the *__getitem__* function of the parent class is over-ridden. In addition to this, the *ToTensor* class is used as a transformation mechanism to convert all of the data which has been passed to a tensor for GPU computations.

Similarly, the *MR_Dist* class inherits from the MRJob class, from the MRJob module and over-rides the reducer and mapper functions. It is important to note here that since training and running the network requires multiple steps, a multi-step mapper was defined in the class. There are basically three steps involved as part of the mapper: the data splitting step, the data collection, loading and transformation step. Each of these three steps are defined as a separate function in *MR_Dist*.

This is followed by a reducer function, which basically trains networks on the split data and then collects the weights of the trained networks. As such, the *reducer* and the *steps* functions of the parent class are over-ridden, in the *MR_Dist* class. The user just needs to initiate an object of this class in the main program and MRJob would take care of the rest.

C. API Usage

In order to use the API, the user would have to follow the following steps (see Fig. 5):

1) Make sure that all of the data is in a CSV format, with a label's section and a values' section.
2) Create an object of the *rbm* class, from the ND module, passing the *learn_rate* (type float), momentum (type float) and *batch size* (type integer) as arguments.
3) Set the *newnet* variable in the *MR_dist* class equal to the newly created *rbm* object.
4) Create a neural network by calling the *create_net* function of the newly created *rbm* object and passing the number of nodes (*nodes*, of type list) and layers (*num_layers*, of type integer) in the arguments.
5) Set the *net* variable in the *MR_dist* class equal to the output of the previous function.

```
if __name__ == '__main__':
    newnet=rbm(0.015, 0.8, 64)
    MRDL.newnet=newnet
    net=newnet.create_net(layer_count=4, node_seq=[784,500,100,10])
    MRDL.net=net
    MRDL.run()
    iter1=newnet.combine_weights(MRDL.weights)
    net=newnet.create_net(layer_count=4, node_seq=[784,500,100,10])
    newnet.set_w(iter1, net)
    testdata=newnet.load_data(data_path='test.csv')
    newnet.test(loaded_data=testdata, nNet=net)
```

Fig. 5. Proposed API Usage Example.

6) Call the *run* function from the *MR_dist* class to begin training.
7) Call the *fetch_weights* function of the trained *rbm* object, to fetch the newly defined weights and store them in a variable.
8) Call the *combine_weights* function from the *rbm* object, passing the weights variable from the *MR_dist* class, as an argument.
9) Keep the data in your HDFS directory.
10) Run your Python script in Hadoop streaming, by typing 'python3 *path_to_python_script* *path_to_data_file* -r hadoop > *path_to_output_file*' in the terminal.

Once the script has finished running, the final network would be saved in the specified output directory and can be later viewed and used for future work.

D. API Functions

a) Net.__init__(self, num_layers, nodes): This is basically the network initialization function and is used to create the neural network that would be later trained on some data. The argument *num_layers* identifies how many layers the RBM would have and what is the size of the *nodes* list. The *nodes* list in turn inputs how many nodes each of the layers is supposed to have. Once an object of this class is created, this function is called and a neural network, having the specified configuration, is created. The network is also assigned *Xavier Uniform* weights and the gain function for the entire network is set to the *Relu* gain function.

b) Net.forward(self, x): This function overrides the default feed-forward function of the *torch.nn.module* class and returns the logarithmic softmax of the gain values, for the entire network. The function works by passing the training values, iteratively through the individual layers of the network and adjusts the weights, as needed.

c) Data_load.__init__(self, file, direct, transform): This is the object initialization function for the *data_load* class. It takes as arguments the path of the CSV file to be read for the data, the path of any additional directories to use and the list of transformations to apply. The function then reads the CSV file and splits it into two variables: X and Y. The variables contain the training data and the corresponding label of the data respectively. It should be noted here that the first index of the CSV file is considered to be the label, while the remaining columns are classified as training data.

d) Data_load.__len__(self): This function returns the number of data points present in the specified dataset.

TABLE III. HADOOP CLUSTER CONFIGURATIONS

Component/Configuration	Description/Value
Processor	Intel(R) Core(TM) i5-7300HQ @ 2.5 and 2.5 GHz
Memory	8 GB
Hard Disk	1 TB
Yarn CPU Cores	4
Yarn Memory	10240 MB
Scheduler Max Memory	8192 MB
Scheduler Min Memory	512 MB
Yarn Virtual Memory Check	Disabled
MapReduce CPU Cores	4
MapReduce Memory	4096 MB
Mapper Memory	2048 MB
Reducer Memory	2048

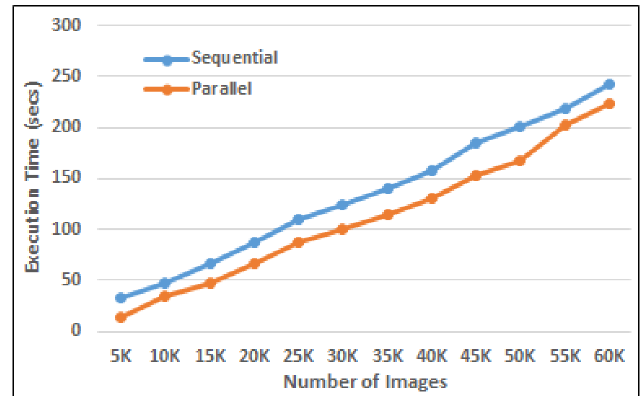

Fig. 6. Performance comparison of Deep Autoencoder using Proposed API

e) Data_load.__getitem__(self, index): The function works to fetch a data point from an index, specified by the user, from the supplied data. The data is first selected from the variables X and Y and the list of specified transformations are applied on X. The updated label and data is then returned as a single tuple, representing the data and label, in respective order.

f) ToTensor.__call__(self, sample): This function is initialized as soon as the class is called. The function takes a list of data as an argument and transforms it to a *tensor*, which is then passed through the neural network.

VI. API EVALUATIONS IN TERMS OF PERFORMANCE AND USAGE

In order to evaluate the performance of our proposed software API, we configured a single-node Hadoop cluster in a workstation with the configurations as shown in Table III.

Fig. 6 shows the execution time in seconds for both sequential (in PyTorch) and parallel implementations of deep autoencoder on MNIST dataset using the proposed software API with different number of input images for model training. The obtained results show significant improvement in performance (about 58%) for an input size of 5000 images while the improvement percentages are decreasing as the input size increases such that the performance improvement is only about 8% for an input size of 60000 images. The reason for this behavior is heavy read operations from permanent storage and extensive memory usage to store the input dataset into memory for processing. Therefore, if we extend the cluster configurations to have multiple data nodes then the input dataset will be distributed among several data nodes and the

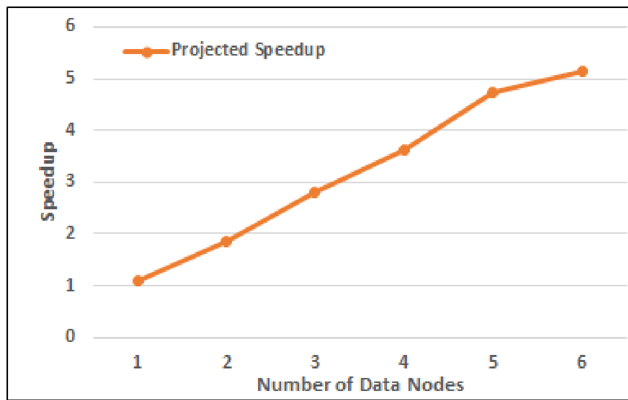

Fig. 7. Projected Speedup based on Parallel Time Estimation

training will be performed on each partition of the data in parallel. This would give more speedups as we increase the number of data nodes in the cluster. Fig. 7 shows the projected speedup estimating the reduction of execution time of parallel implementation based on the size of the data partition that each node will contain for processing.

Furthermore, the complexity of code development is significantly reduced to create multi-layer deep learning models. This is achieved by using a list of visible and hidden neurons provided by the user and *Net.__init__* function of the proposed API will generate the required PyTorch code to add layers into the model.

VII. Conclusion and Future Work

In this paper, we proposed a software API for fast development of large-scale deep learning models for big data analytics. The idea was to analyze the datasets with a higher amount of volume, velocity, and variety. The API proposal is on the basis of our exploration of the latest trends in the development of parallel algorithms, optimization techniques, tools and libraries related to big data analytics and deep learning on various parallel architectures. Initially, we assumed the need of parallelizing the deep learning algorithms as basic building blocks for the parallel software API. However, with the statistical analysis of a deep learning algorithm (RBM), we found that the factors affecting the most on the performance of these algorithms are highly sequential in nature and parallelizing using these factors will not be beneficial because of strong flow dependencies in the code. Furthermore, there are several frameworks and available libraries that provide efficient implementations of these algorithms. Therefore, there is a need to extend these frameworks to do the model execution on multiple computing nodes, where each node has a portion of input samples and runs the model in parallel. Hence, in order to ease the development of deep learning models for big data analytics, we propose a parallel software API as an extension of PyTorch with HDFS and MapReduce frameworks. We obtained significant improvements in the deep learning models using the proposed API in reduction of the execution time and code complexity. We have evaluated the API by implementing a deep auto-encoder using MNIST dataset on a single node Hadoop cluster. In future, we plan to setup a multinode Hadoop cluster and run the implementation with various number of data nodes.

Acknowledgment

This work is supported by the Deanship of Scientific Research at Qassim University under the project no. 1374-coc-2016-1-12-S.

References

[1] B. K. Tannahill and M. Jamshidi, “Big data analytic paradigms-from PCA to deep learning,” in *AAAI Spring Symposium - Technical Report*, vol. SS-14-04, 2014, pp. 84–90.

[2] X.-W. Chen and X. Lin, “Big Data Deep Learning: Challenges and Perspectives,” *IEEE Access*, vol. 2, pp. 514–525, 2014.

[3] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-y. Wang, “Traffic Flow Prediction With Big Data : A Deep Learning Approach,” *Intelligent Transportation Systems, IEEE Transactions on*, vol. 16, no. 2, pp. 865–873, 2015.

[4] L. Deng and D. Yu, *Deep Learning: Methods and Applications*. NOW Publishers, May 2014.

[5] J. Zakir, T. Seymour, and K. Berg, “Big Data Analytics,” *Issues in Information Systems*, vol. 16, no. 2, pp. 81–90, 2015.

[6] D. Singh and C. K. Reddy, “A survey on platforms for big data analytics,” *Journal of Big Data*, vol. 2, no. 1, p. 8, 2014.

[7] M. Chen, S. Mao, and Y. Liu, “Big data: A survey,” in *Mobile Networks and Applications*, vol. 19, no. 2, 2014, pp. 171–209.

[8] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, “BigBench,” in *Proceedings of the 2013 international conference on Management of data - SIGMOD ’13*. New York, New York, USA: ACM Press, 2013, pp. 1197–1208.

[9] Z. Wang, “The Applications of Deep Learning on Traffic Identification,” *Black Hat USA*, 2015.

[10] V. S. Agneeswaran, P. Tonpay, and J. Tiwary, “Paradigms for Realizing Machine Learning Algorithms,” *Big Data*, vol. 1, no. 4, pp. 207–214, 2013.

[11] Y. Bengio, “Deep learning of representations: Looking forward,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7978 LNAI, 2013, pp. 1–37.

[12] Q. V. Le, A. Coates, B. Prochnow, and A. Y. Ng, “On Optimization Methods for Deep Learning,” *Proceedings of The 28th International Conference on Machine Learning (ICML)*, pp. 265–272, 2011.

[13] Y. Zhao, D. J. MacKinnon, and S. P. Gallup, “Big Data and Deep Learning for Understanding DoD Data,” *CrossTalk*, vol. 28, no. 4, pp. 4–11, 2015.

[14] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, “Deep learning applications and challenges in big data analytics,” *Journal of Big Data*, vol. 2, no. 1, p. 1, 2015.

[15] E. B. Yoav Shoham, Raymond Perrault and J. Clark, “The ai index 2017 annual report,” Stanford, Review Report, 2017.

[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[17] T. Olas, W. K. Mleczko, R. K. Nowicki, and R. Wyrzykowski, *Adaptation of Deep Belief Networks to Modern Multicore Architectures*. Cham: Springer International Publishing, 2016, pp. 459–472.

[18] T. Olas, W. K. Mleczko, R. K. Nowicki, R. Wyrzykowski, and A. Krzyzak, *Adaptation of RBM Learning for Intel MIC Architecture*. Cham: Springer International Publishing, 2015, pp. 90–101.

[19] B. Barney, “Introduction to parallel computing,” https://computing.llnl.gov/tutorials/parallel_comp/#Whatis, (Accessed on 01/31/2019).

[20] “A beginner’s tutorial for restricted boltzmann machines - deeplearning4j: Open-source, distributed deep learning for the jvm,” https://jrmerwin.github.io/deeplearning4j-docs/restrictedboltzmannmachine, (Accessed on 12/31/2018).

[21] "Deep-belief networks in java - deeplearning4j: Open-source, distributed deep learning for the jvm," https://mgubaidullin.github.io/deeplearning4j-docs/, (Accessed on 12/31/2018).

[22] "Deep learning tutorials — deeplearning 0.1 documentation," http://www.deeplearning.net/tutorial/, (Accessed on 12/31/2018).

[23] "What is Pytorch?" https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py, (Accessed on 03/30/2019).

[24] S. P. Bappalige, "An Introduction to Apache Hadoop for big data," https://opensource.com/life/14/8/intro-apache-hadoop-big-data, (Accessed on 03/30/2019).

[25] Yelp and Contributors, "Why mrjob?" https://mrjob.readthedocs.io/en/latest/guides/why-mrjob.html#overview, (Accessed on 03/30/2019).