Large-Scale Distributed Deep Learning using Apache Spark

A DISSERTATION

submitted in partial fulfillment of the requirement for the degree of

Master of Technology

in

Computer Technology

by

Disha Shrivastava

(2014EET2444)

Under the guidance of

Prof. Santanu Chaudhury & Prof. Jayadeva

Department of Electrical Engineering

Indian Institute of Technology Delhi

July 2016

CERTIFICATE

This is to certify that the work contained in this thesis titled **"Large-Scale Distributed Deep Learning using Apache Spark"** by **Disha Shrivastava** (2014EET2444) in partial fulfillment of the course work requirement of *Master of Technology* in *Computer Technology* program from *Department of Electrical Engineering*, Indian Institute of Technology, Delhi is a bonafide work carried out by her under my guidance and supervision. The matter submitted in this dissertation has not been admitted for an award of any other degree anywhere unless explicitly referenced.

Prof. Santanu Chaudhury
Department of Electrical Engineering
Indian Institute of Technology Delhi

Prof. Jayadeva
Department of Electrical Engineering
Indian Institute of Technology Delhi

ACKNOWLEDGEMENTS

I would like to express my deepest and sincere gratitude to my supervisors, Prof. Santanu Chaudhury and Prof. Jayadeva, Department of Electrical Engineering, IIT Delhi for their valuable support, guidance and encouragement throughout the course of this project work. I would also like to thank them for giving me the opportunity to work on a challenging project which required me to invest the best of my abilities and helped me widen my knowledge horizon greatly. I thoroughly enjoyed working on this project. I would also like to thank the other members of the evaluation committee for their valuable feedback during each presentation.I would take this opportunity to acknowledge the contribution of Mr. Manoj Sharma, PhD student at IIT Delhi for his useful collaboration in the segment of this thesis involving the work on noise resilient image super-resolution.

Last but not the least, I would like to thank my parents and God for being the pillar of my support during difficult times and motivating me to move forward with a positive attitude.

Disha Shrivastava

02 July 2016

ABSTRACT

Training deep networks is both expensive and time-consuming with the training period increasing in proportion to the size of data and growth in the number of model parameters.In this thesis, we provide a framework for distributed training of deep networks over a cluster of CPUs in Apache Spark.The framework implements both data parallelism and model parallelism making it suitable to use for deep networks which require huge amount of data for training and model parameters which are too big to fit into the memory of a single machine.It can be scaled easily over a cluster of cheap commodity hardware to attain significant speedups and obtain better results making it quite economical as compared to farm of GPU's and supercomputers.We have proposed a new algorithm for training of deep networks for the case when the network is partitioned across the machines(Model Parallelism) along with detailed cost analysis and proof of convergence of the same.We have developed implementations for Fully-Connected feedforward networks, Convolutional Neural Networks, Sparse Autoencoders, Recurrent Neural Networks and Long-Short Term Memory architectures.We have carried out extensive simulations to analyse the performance of our framework with variations in data and model sizes for different neural networks and with different number of worker nodes/partitions.In the end, we have shown the application of our developed framework to propose a new architecture for Noise Resilient Image Super-Resolution where we have not only outperformed state-of-art methods both in terms of PSNR and SSIM but also achieved significant reduction in training time.

Abbreviations Used

RDD	Resilient Distributed Datasets
SGD	Stochastic Gradient Descent
RNN	Recurrent Neural Networks
LSTM	Long Short Term Memory
CNN	Convolutional Neural Networks
SSDA	Sparse Stacked Denoising Autoencoder
HR	High Resolution
LR	Low Resolution
PSNR	Peak Signal-to-Noise Ratio
SSIM	Structural Similarity Index
BPTT	Backpropagation Through Time
tBPTT	Truncated Backpropagation Through Time

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In recent years, deep learning is applied in almost all the fields ranging from speech recognition to image and text recognition, and it is giving quite impressive results. Deep Learning seems to be the answer to "every" problem.Deep Networks had been applied successfully in both supervised and unsupervised settings to display tremendous improvement in performance.Many of the recent advances in deep networks though involve fitting large models (containing billions of parameters) to massive datasets (hundreds of gigabytes) to obtain better results.Thus it becomes quite time consuming (training period ranging from days to weeks) to train these networks.As we know, in deep learning hyper-parameter tuning plays a very important role in obtaining better results.If training the network takes so much time, there is very little flexibility left to experiment with different values of hyperparameters.People have used supercomputers and farms of GPUs in the past to produce required results but their use is restricted because they are expensive.Also, with large model parameters the training speedup with GPUs is small as the model doesn't fit into the GPU memory and CPU-GPU transfers prove to be a significant bottleneck.To break this gap, we thought of using a cluster of commodity hardware and perform distributed implementation of deep

learning algorithms which will help us not only to handle Big Data efficiently but also attain significant speedups with improved performance.**The distributed framework for the training of deep networks is implemented in Apache Spark and utilizes a cluster of cheap commodity hardware (CPUs) making it quite economical and flexible both in terms of usage and application.**

1.2 Objectives

The main objectives of this work can be stated as below:

1. **To develop distributed framework for using deep networks with Big Data and Large Model sizes on commodity hardware**

2. **To achieve scalability and optimal performance for a given configuration on Apache Spark**

1.3 Related Work

Much work has been done to build distributed frameworks for training deep networks. Google researchers build DistBelief [DCM+12] and Microsoft came up with Project Adams [CSAK14], both being distributed frameworks to train large scale models for deep networks over thousands of machines and utilizing both data and model parallelism.A central parameter server model has been used in [LAP+14] [HCC+13] in which the worker nodes asynchronously fetch and update the gradients on a master node which holds the updated model parameters, though their systems are better suited for sparse gradient updates.Recently, Google came up with TensorFlow [AAB+16], an improved version of DistBelief system which can be used for distributed training of deep networks by specifiying them as computational flow graphs and benefiting from its auto-differentiation and suite of first rate optimizers.FireCaffe [IAMK15] uses a cluster of GPUs to scale deep networks over Infiniband/Cray interconnects.They minimize

communication overhead by using reduction trees for aggregating gradient as compared to parameter server.

The systems described above show high performance in terms of speedup and accuracy and a fine grained control over scheduling. However, due to their demanding communication requirements, they are unlikely to exhibit the same scaling on a cluster of CPUs. Moreover, these systems are highly customized in terms of either requirement of particular hardware or software tools and hence they are difficult to integrate with general-purpose batch computational frameworks such as Spark and MapReduce. A recent work [MNSJ15] integrates Spark with Caffe.Their model however implements just data parallelism and no model parallelism and may not be suitable for deep networks with extremely large parameters which are too big to fit into the memory of a single machine.

1.4 About Apache Spark

AMPLab in University of California, Berkeley came up with Apache Spark which is a cluster computing framework [ZCF+10].This open source software later became part of Apache Software Foundation. Its first release came out on 30th May, 2014.Spark has been proved to be 100 times faster than Hadoop for iterative tasks (which are fairly common in machine learning) due to its in-memory computations and efficient, fault tolerant abstractions called Resilient Distributed Datasets (RDD) which can be rebuilt if a partition is lost [ZCD+12].

The reason why Spark was chosen is because it provides an open-source, generic batch computational framework which makes it easy to implement distributed optimization algorithms over a cluster of commodity hardware. A further advantage of integrating model training with Spark is that it provides efficient data-pipeline interfaces such as SQL, Hive, YARN, HDFS for obtaining, cleaning and processing data. If a user wishes to train a deep network on the output of a SQL query or on the output of a graph computation and to feed the resulting predictions into a distributed visualization tool, this can be done conveniently with Spark with the data kept in memory

all the time and without needing to write data to the disk intermediately.

1.5 Contributions of this Thesis

We produce a distributed and scalable framework for training of deep networks implementing both data and model parallelism (Downpour SGD) on a cluster of CPUs using only Apache Spark.This makes our framework complete and independent and the user does not require to install any additional software/hardware dependency.The unique contributions of this thesis can be written as below:

- We have proposed **a new algorithm for training of deep networks for the case when the network is partitioned across the machines(Model Parallelism)**; along with **detailed cost analysis** and **mathematical and experimental proof of convergence** of the algorithm(Chapter 3)

- We have implemented a **generic framework which can be used for training Fully-Connected feedforward networks, Convolutional Neural Networks, Sparse Autoencoders, Recurrent Neural Networks and Long-Short Term Memory architectures.This work has been done totally from scratch**.Till the time this thesis is written, Spark has officially released implementations only for Fully Connected nets; that too just implementing data parallelism which makes our implementation more unique and important(Chapters 2,4)

- We have **proposed a new architecture for Noise Resilient Image Super-Resolution where we have not only outperformed state-of-art methods both in terms of PSNR and SSIM but also achieved significant reduction in training time** (9X speedup compared to Caffe) using our distributed framework(Chapter 5)

- We have done a **thorough analysis of our framework with variations in data and model sizes and number of worker cores/nodes/partitions**.

Analysis done is useful in **understanding the scalability and distributed performance** of our framework as well as how Apache Spark handles these variations(Chapter 4)

1.6 Organization of the Thesis

This thesis is divided into six chapters.The first chapter briefs about the motivation for choosing the problem along with related work and unique contributions of our work.In Chapter 2, we provide a general overview of different neural network architectures(along with details of their training) which were used in our work.Chapter **??** develops the basic theory used to develop our framework along details of the proposed algorithm.Implementation details and results are discussed in Chapter 4.We finally show the application of our framework for noise resilient image super-resolution in Chapter 5 which is followed by Conclusions and Future Scope in Chapter 6. Appendix A provides details of the mathematical proof of convergence of our proposed algorithm of distributed backpropagation in case of model parallelism.The complete details of architecture and network parameters used for Noise Resilient Image Super-Resolution along with definition of evaluation metrices can be found in Appendix B.

Chapter 2

Neural Networks

In this chapter, we provide a basic overview of the concepts of different neural network architectures which we have implemented as part of this thesis.We also provide details of the significance of each model and how to train and test them.

2.1 Fully Connected Feedforward Networks

In a fully connected neural network, each neuron at one layer is connected to every other neuron in the other layer with distinct values of weights and biases.The network may have one or more hidden layers with non-linear activation functions (Fig 2.1).

2.1.1 Gradient Descent

The training of neural network involves finding the values of weights which minimize the error function, E between the output labels(t) and the prediction of the network(y).For a network with L layers with a squared loss function and assuming M number of training examples, this can be written as below:

$$E = \frac{1}{2}\sum_{i}^{M}(t_i - y_i^L)^2 \tag{2.1}$$

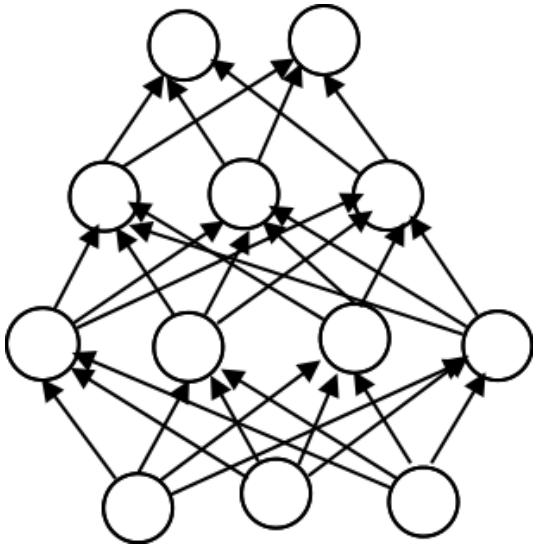


Figure 2.1: A fully-connected feedforward network

Given a differentiable function f, gradient descent provides one of the simplest algorithms for finding the minimum of the function in an iterative manner.If we start at any point on the function curve, movement in the direction of negative gradient will lead us to reach the local minima at some point.The main steps in gradient descent algorithm are stated below:

1. Initialization: We pick up a random point to start with, say x_0
2. Gradient: We compute the gradient at our current guess, $\Delta f(x_i)$
3. Update: Move in the direction of gradient by a step size of $\alpha; x_{i+1} = x_i - \alpha \Delta f(x_i)$
4. Iterate: We repeat steps 1-3 until the function, f converges or until $f(x_i) < \epsilon$ for some small tolerance ϵ.

There are different variants of gradient descent like stochastic gradient descent (one training example at a time), mini-batch gradient descent (batches of training examples) and full batch gradient descent or gradient descent (updation is done after all training examples).There are certain advanced methods of optimization as well like Conjugate gradient methods, Newton methods and Hessian based methods.Each of these methods can be used depending on the application at hand.

2.1.2 Backpropagation

In order to calculate gradients efficiently, we make use of an algorithm called backpropagation [HN89] which is based on repeated application of chain rule to calculate the gradients.The algo can be divided into two phases:

- **Forward Pass**

  1. We calculate the output of neuron j at layer l ,y_j^l as:

$$a_j^l = \sum_i w_{ij}^{l-1} y_i^{l-1} \tag{2.2}$$

$$y_j^l = \sigma(a_j^l) \tag{2.3}$$

     where $\sigma = 1/(1 + exp(-x))$ is the sigmoid activation function, w_{ij}^{l-1} is the weight between neuron i at layer l-1 and neuron j at layer l and y_i^{l-1} is the output at layer l-1.

  2. We repeat step 1 until the output layer is reached to get the final output of the network, y^L

- **Backward Pass** The derivative of the error function, E with respect to weight w_{ij}^{l-1} is given by:

$$\frac{\partial E}{\partial w_{ij}^{l-1}} = \frac{\partial E}{\partial y_j^l} * \frac{\partial y_j^l}{\partial a_j^l} * \frac{\partial a_j^l}{\partial w_{ij}^{l-1}} = \delta_j^l * y_i^{l-1} \tag{2.4}$$

$$\delta_j^l = \frac{\partial E}{\partial y_j^l} * \frac{\partial y_j^l}{\partial a_j^l} = \begin{cases} (y_j^L - tj) * y_j^L * (1 - y_j^L) & \text{if j is an output neuron} \\ (\sum_{k \in l+1} \delta_k^{l+1} * w_{jk}^l) * y_j^l * (1 - y_j^l) & \text{if j is an inner neuron} \end{cases} \tag{2.5}$$

We start from output layer till we reach the input layer calculating the gradients for each of the weights in the network which is then updated based on the update rule:

$$w_{ij} := wij - \alpha \frac{\partial E}{\partial w_{ij}} \tag{2.6}$$

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are variants of the multilayer perceptron models discussed before that are inspired by the structure of the human brain.In CNN, the neuron at each layer is connected to only a subset of spatially-located neurons in the previous layer (local connectivity) making the network connections sparse as compared to a feedforward neural networks.A feature map is obtained by repeated convolution of the segments of input image by a linear filter followed by applying a non-linearity.In addition, in CNNs, each filter units share the same parameterization (weight vector and bias) thereby reducing the number of free parameters in the network significantly leading to better generalization.A convolutional layer is usually followed by a subsampling or pooling layer which reduces the size of each feature map by taking the average or max of the pixels in a given neighbourhood and ensures translation invariance.The last layer is usually a fully connected layer and training is done exactly as described above by propagating the error back to the network.The features discussed above make CNNs pretty suitable for applications involving object detection and recognition in images where there is usually a spatial correlation among the surrounding pixels.

2.3 Sparse Denoising Auto Encoders

A denoising autoencoder is an unsupervised model for neural networks.Its main purpose is to obtain the original input from its noisy version.It first encodes the input to learn hidden representations, $y = \sigma(Wx+b)$ and then decodes the input back in the decoding step to produce $\widetilde{x} = \sigma(W'x + b')$ where $w' = W^T$ and $b' = b^T$. We then minimize the error between x and $\widetilde{x}$ to learn the weights of the network. We can have more number of neurons in the hidden layer as compared to the number of input neurons and enforce a sparsity constraint thereby ensuring that only a few neurons in the hidden layer get activated each time.A number of hidden layers can be stacked together by doing a layer-wise pretraining, discarding the output layer and using the hidden representations of preceding layer as inputs for pretraining the subsequent layer.A fine-tuning step is

then performed by propagating the error from the output layer back to the input layer and modifying the learned weights slightly.In this way, by using a deep Sparse Stacked Denoising Autoencoder(SSDA) we can obtain very powerful and better representations of the input data.

2.4 Recurrent Neural Networks

Recurrent Neural Networks(RNN) produce output which is not only dependent on the current input but also past values of the input.They are widely used in tasks which require processing of sequential information such as language modelling, machine translation and speech recognition.The figure below (Fig. 2.2) shows a simple recurrent neural net with a feedback connection at the hidden layer which is unrolled in time.s_t is the hidden state at time t which is given by $s_t = f(Ux_t + Ws_{t-1})$ where f is a non-linear function and output at time t, $o_t = f(Vs_t)$.

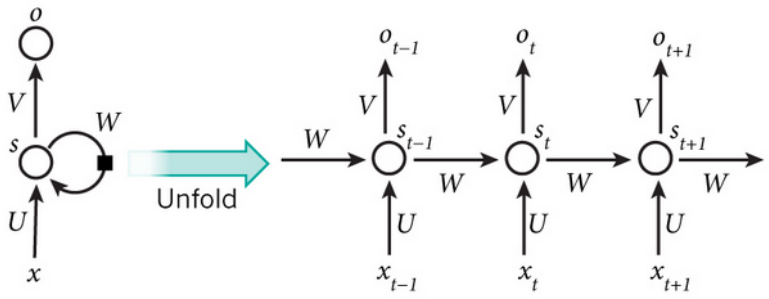


Figure 2.2: An recurrent neural network unfolded in time

2.4.1 Backpropagation Through Time

To train a RNN, we use a slightly modified version of the backpropagation algorithm which is called Backpropagation Through Time (BPTT) where we sum up the gradients for W for each time step. What is use in practise though is truncated backpropagation through time (tBPTT) where the error is propagated back upto a fixed number of time

steps (say T).The equations for BPTT with an error function E_t at each time step t can be written as:

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W} \tag{2.7}$$

$$\frac{\partial E_t}{\partial W} = \sum_{t=1}^{k} \frac{\partial E_t}{\partial o_t} * \frac{\partial o_t}{\partial s_t} * \frac{\partial s_t}{\partial s_k} * \frac{\partial s_k}{\partial W} \tag{2.8}$$

$$\frac{\partial s_t}{\partial s_k} = \prod_{i=k+1}^{t} \frac{\partial s_i}{\partial s_{i-1}} = \prod_{i=k+1}^{t} W^T diag[f'(s_{i-1})] \tag{2.9}$$

As can be seen from Eq. (2.9), repeated multiplication over long time sequences may lead to vanishing/exploding gradient problems while training RNN. To resolve this problem we use another architecture called LSTM which is discussed next.

2.5 Long Short Term Memory

Long Short Term Memory (LSTM) memory block provide an alternate means of calculating the hidden state.LSTM in its simplest form consist of three gates called input, forget and output gates.These gates act as digital gates which control how much of the input, previous hidden state and outputs will flow through them to the external network..The equations corresponding to an LSTM block can be written as below:

$$i_t = \sigma(W_i x_t + R_i s_{t-1} + b_i) \tag{2.10}$$

$$f_t = \sigma(W_f x_t + R_f s_{t-1} + b_f) \tag{2.11}$$

$$o_t = \sigma(W_o x_t + R_o s_{t-1} + b_o) \tag{2.12}$$

$$c'_t = tanh(W_c x_t + R_c s_{t-1} + b_c) \tag{2.13}$$

$$c_t = f_t \cdot c'_{t-1} + i_t \cdot c'_t \tag{2.14}$$

$$s_t = o_t \cdot tanh(c_t) \tag{2.15}$$

In the above equations, W's, R's and b's correspond to the input weights, recurrent weights and biases respectively.i_t, f_t, o_t, c_t, s_t denote the output of the input gate, forget gate, output gate, internal memory of the cell and hidden state at time t, respectively.The LSTM cells thus have an advantage obtained by multiplicative gates that gives it the flexibility to store and make use of information over longer periods of time.This helps it in learning long-term dependencies in an efficient manner as compared to RNNs.

In the next chapter, we provide details of the concepts involved in building the framework for distributed training of the various neural network models discussed above.We also discuss our proposed algorithm for distributed backpropagation in case of model parallelism along with detailed cost analysis and mathematical proof of convergence.

Chapter 3

Distributed Training Algorithms

In this chapter, we develop the basic theory required for the development of our framework for distributed training of deep networks.We start by explaining the concepts of data and model parallelism in Section 3.1 which is followed by introducing our new algorithm for model parallelism in Section 3.2; along with detailed cost analysis and mathematical proof of convergence for the same in subsequent sections.

3.1 Downpour SGD

We have implemented Downpour SGD proposed by [DCM+12] which provides asynchronous and distributed implementation of stochastic gradient descent. The training instances are divided across different machines **(Data Parallelism)**(Fig. 3.1). Each machine has its own copy of the whole network. If the network is big enough, it can be divided across different machines**(Model Parallelism)**(Fig. 3.2). Each machine operates on its own set of data and model replica to compute changes in parameters (weights and biases). These values are periodically sent to a central parameter server where the parameter updation is done according to the learning rule. The updated parameters are fetched from the parameter server asynchronously and with some delay.

The gradient updates may be out-of-order and the weights may be out of date.However,

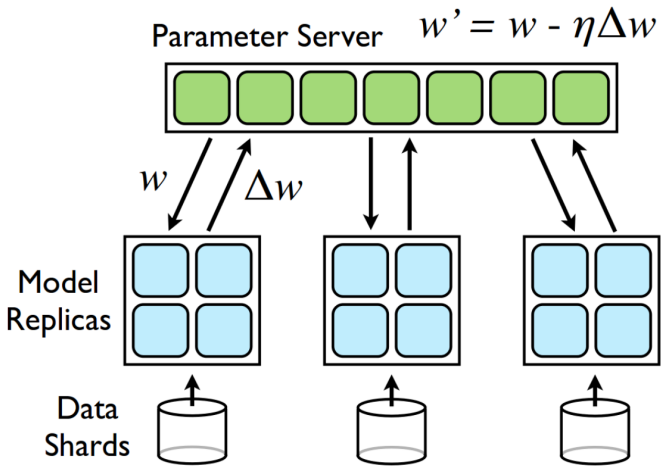


Figure 3.1: Figure explaining the concept of Downpour SGD. Image Source: [DCM+12]

it has been shown that the delay may be harmful initially but is harmless as the number of iterations and the size of training data increases [ZWLS10, ZLS09].The use case for data parallelism is when we have massive amount of training examples and the use case for model parallelism is when the neural network is too big to fit into the memory of a single machine. Hence, this model allows us to work with deep networks with large data and model sizes efficiently and achieve significant speedups.

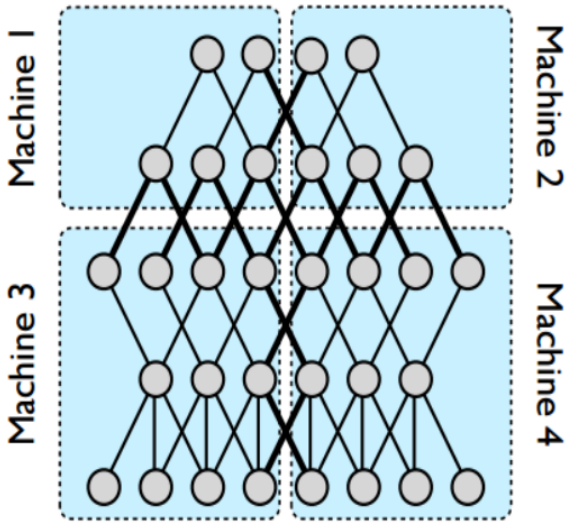
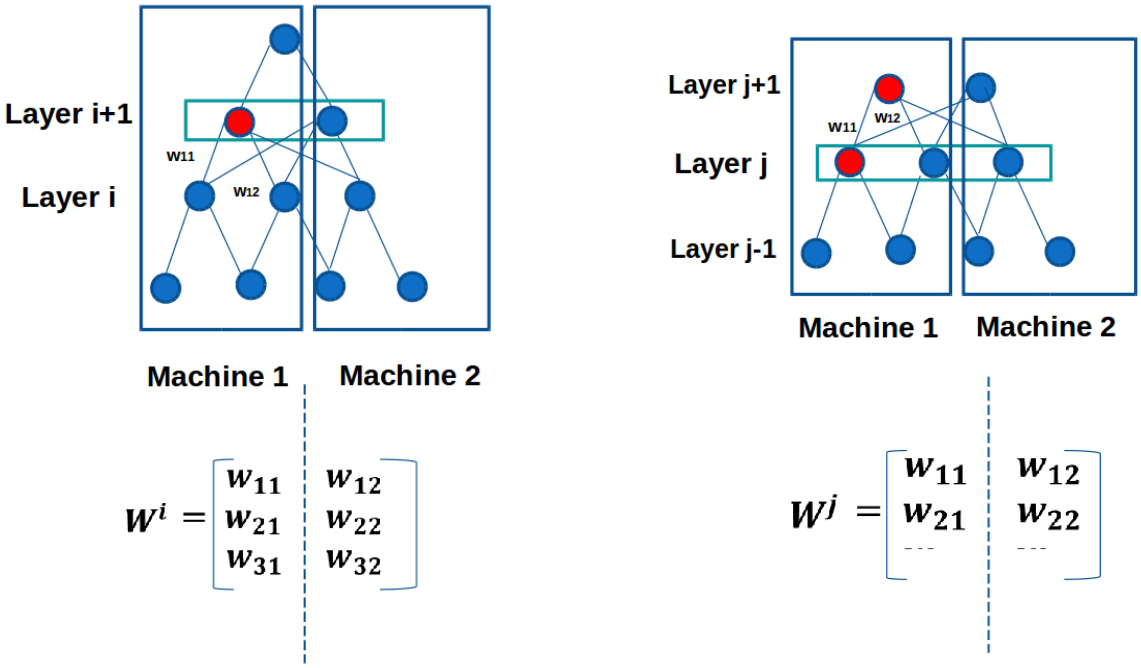


Figure 3.2: Figure explaining model parallelism. Image Source: [DCM+12]

3.2 My Proposed Algorithm for Model Parallelism

We have proposed and implemented a simple algorithm for **distributed backpropagation** for the case when the deep network is too big to fit into the memory of a single node and hence the network is divided across different nodes.The processes described below represent computational units(threads/processes) and are distributed across the nodes,i.e.,a single node may contain more than one processes.One of the processes is designated as master and the rest as slaves.The two phases of the distributed backpropagation can be described as below:



(a) Forward Pass

(b) Backward Pass

Figure 3.3: Illustration of distributed backpropagation in case of network partitioning across the machines

Forward Pass: The weight matrix is partitioned vertically in a uniform way(no.of columns/no. of processes) and distributed across the machines as shown in Fig.3.3a. The main steps in the algorithm at layer i+1 can be described as below:

1. The master process broadcasts layer i output vector, $a^i = [a_1 a_2 a_3]$ to all other processes

2. Each process computes its subset of layer i+1 output vectors using the weights it has e.g, On Machine 1 in Fig.3.3a for the neuron highlighted in red, the output is calculated as

$$a_1^{i+1} = \sigma([a_1 a_2 a_3] \odot [w_{11} w_{21} w_{31}]) \tag{3.1}$$

3. The master gathers from all the processes layer i+1âĂŹs output vectors a^{i+1}

4. We repeat from Step 1 till the output layer is reached.

Backward Pass: The algorithm at layer j can be written in the below steps:

1. Master process broadcasts the error vector of the previous layer, δ^{j+1} to all processes.

2. Each process computes the weight changes for its subset using the output vectors obtained during the forward pass e.g. in Fig.3.3b the change in weight w_{11} is calculated as

$$\partial E / \partial w_{11} = \delta_1^{j+1} * a_1^j \tag{3.2}$$

3. Each process computes its contribution to the error vector at layer j (Here, $\sigma'(in^j)$ is the gradient of the sigmoid activation function at layer j)

$$\delta^j = \sum_{j+1} \delta^{j+1} * W^j * \sigma'(in^j) \tag{3.3}$$

4. Master sums up the individual contribution of δ^j from each process

5. We repeat from Step 1 till the input layer is reached.

3.3 Cost Analysis of the Algorithm

To do a theoretical cost analysis of our model, we make use of the method given in [PLWH03] which is reproduced here for convenience of understanding.There are two

factors which influence the execution time of our model: the time taken for computation (T_{comp}) and the time taken for communication (T_{comm}).Ignoring the computation time, the general equation for the cost analysis can be written as:

$$T_{comm} = K[T_{lat} + N * T_{data}] \tag{3.4}$$

where, T_{lat} is the latency associated with sending a message, T_{data} is the time required to transmit a unit of data which is inversely proportional to the network speed, K is the total number of messages sent per epoch and N is the number of units of data sent. Let us assume the number of processes to be F and the number of training examples to be M. For each of the training examples, initially the master process sends the entire input data and the processes's portion of the output labels to rest of the slave processes generating (F - 1) messages (Eq. 3.6).In these equations, b_i= number of neurons at layer i, δ^i = error vector at layer i, n = total number of layers in the network and S(x) = size of the data structure x. During the forward pass,for each of the hidden layers, each process sends the output of its neurons to all other processes.Assuming this communication to be an all-to-all broadcast in a hypercube structure, each single process F requires $\log_2 F$ messages to be sent generating a total of $F \log_2 F$ messages(Eq. 3.7).The backward pass results in generation of two types of messages per hidden layer.The master process sends the error vector from previous layer δ^{j+1} to all other processes which is followed by each process sending its portion of the error vector of the current layer δ^j to the master resulting in 2(F - 1) messages(Eq. 3.8). Therefore, the value of K is calculated to be:

$$K = M[(F-1) + \sum_{i=1}^{n-2}(F \log_2 F + 2(F-1)) \tag{3.5}$$

- Number of units of data sent during initialization:

$$N_1 = (F-1)[S(b_0) + S(\frac{b_{n-1}}{F}) \tag{3.6}$$

- Number of units of data sent during Forward Pass:

$$N_2 = \sum_{i=1}^{n-2} F \log_2 F * S(\frac{b_i}{F}) \tag{3.7}$$

- Number of units of data sent during Backward Pass:

$$N_3 = \sum_{i=1}^{n-2}(F-1) * S(\delta^{i+1}) + (F-1) * S(\frac{\delta^i}{F}) \tag{3.8}$$

- Total Number of units of data:

$$N = \frac{N_1 + N_2 + N_3}{K} \tag{3.9}$$

Putting the values from Eq.(3.5),(3.6),(3.7),(3.8) and (3.9) in Eq. (3.4), we get the theoretical cost of our model.We can see that with increase in the number of nodes (or F) , there is a trade off between the speedup which can be obtained by increasing the degree of parallelization and the communication overhead.

3.4 Proof of Convergence of the Algorithm

We derive an upper bound on the expected regret and hence prove that our model of delayed stochastic gradient descent for the case when both data and network is distributed across different machines converges.The final result is of the form given below:

$$\begin{aligned} E[R[X]] \quad \leq \frac{10}{9}[\lambda\tau F^2 + [\tfrac{1}{2} + \tau]\frac{L^2}{\lambda}[1 + \tau + \log(3\tau + (H\tau/\lambda))] \\ + \frac{L^2}{2\lambda}[1 + \log T] + \frac{\pi^2\tau^2 H L^2}{6\lambda^2} \end{aligned}$$

Thus, we see that we have a dependency of the form $O(\tau \log \tau + \log T)$. We get two separate terms dependent on τ (delay) and T(number of iterations). This implies that when T is large, delay τ essentially gets averaged out in different iterations and our algorithm converges with the upper bound as given above.The proof is based along the lines discussed in [ZLS09] and the details of it can be found in the Appendix A.

In the next chapter, we provide complete details of our implementation of the distributed and scalable framework for training of deep networks using Apache Spark.We also discuss the extensive experiments carried out along with their results and through analysis of the same.

Chapter 4

Implementation Details

4.1 Cluster Setup

In order to carry out our experiments, we have setup a cluster of five nodes in Multimedia Lab, IIT Delhi.Out of these five machines, four machines had Intel Xeon E5-2600 processor/64GB RAM/20 cores and one with same processor but 32GB RAM/8 cores.One of them is designated the master node(central parameter server) and the rest as slaves/workers(Fig.4.1).We have formed a multi-node hadoop cluster for the Distributed File System storage part(HDFS) and Apache Spark in distributed mode has been setup on top of that.

4.2 Implementation Flow Diagram

The flow diagram of our basic implementation is shown in Fig.4.2.We distributed the data RDD's to different machines and gradients were calculated using suitable gradient descent techniques (mentioned later).The model if big enough was distributed across different machines by partitioning the weight matrix column-wise and training by the proposed algorithm for distributed backpropagation.The gradients were then collected from different machines at the master node and updation in parameters was

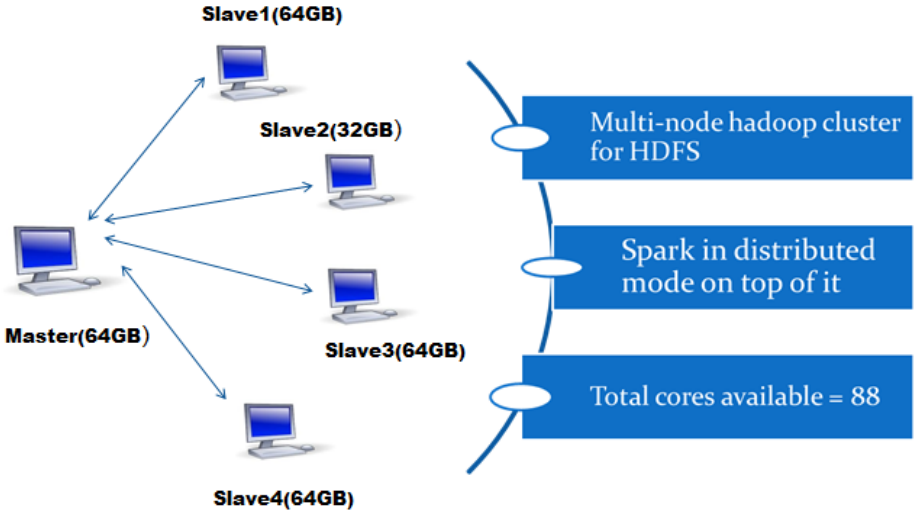


Figure 4.1: Diagram explaining the Cluster Setup

done according to the learning rule.This process was carried on for repeated number of iterations till the model converges.We used softmax at the output layer for multi-class classification.The prediction was made on a separate test set.For different types of neural networks changes are made just in the way forward and backward passes are performed in individual machines and some hyperparameters are changed accordingly. We use threads in individual machines to attain more speedup.Communication between processes in the same machine in case of network partitioning is via threads and via Akka framework for processes in different machines.The results obtained were averaged over multiple runs due to variation in CPU loads and varying network latency factors.All the results were obtained using only CPUs in all the machines and no GPUs were used.

4.3 Experimental Specifications

We carried out extensive simulations to analyse the time and accuracy performance of our distributed and scalable framework with increase in the number of worker nodes/partitions with varying sizes of data samples and model parameters; and for

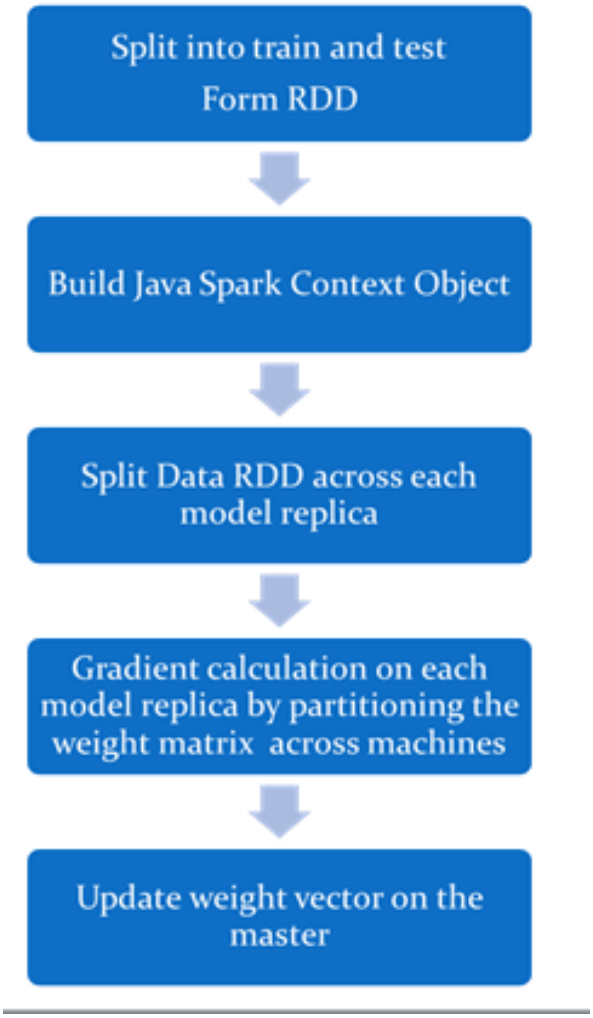


Figure 4.2: Flow Diagram of implementation

different neural network models like FC neural nets, CNN, SSDA, RNN and LSTM.We used both Apache Spark 1.4.0 with Hadoop 2.4.1 [Spa14b] with code written in Java for FC nets, SSDA and LSTM, Scala for CNN and Python for RNN.We used Eclipse IDE with Maven support for developing and debugging the code.We made use of JBlas [Bra13] and Mallet [mal] libraries for linear algebra and optimization, respectively.We experimented with a variety of gradient descent algorithms like SGD, Mini-batch gradient descent, conjugate gradient descent and LBFGS; each of which was found suitable for different networks.

4.3.1 Datasets Used

- Extended version of MNIST dataset [mni] which contains about 8.1 million samples with 28X28 binary images of handwritten digits from 0 to 9(10 output classes).This was used as a benchmark dataset for carrying out experiments on FC nets, CNN and SSDA.

- The Project Gutenberg [Har71] dataset which contains the complete works of Shakespeare.The dataset was used for the task of character level prediction using RNN and LSTM.

4.3.2 Network specific Training and Architecture Details

We experimented with different number of hidden layers, number of neurons in each layer and kernel sizes for each of the networks.We state below the specific details which were used to obtain the results mentioned in the next section.

1. **Fully Connected Neural Networks**: We used a three layer FC net with 784 (28 X 28) neurons at the input layer, 480 and 160 neurons in the first and second hidden layers respectively, followed by 10 neurons at the output layer corresponding to the 10 output classes.For the model parallelism part, we varied the number of neurons and number of layers to generate suitable number of model parameters.Sigmoid activation function was used throughout the network except the output layer where softmax was used for multi-class classification.Conjugate Gradient Descent was found to be the fastest and showed the best results among all the other gradient descent techniques discussed above.The number of model replicas was kept as 5 and the model was trained till convergence using Downpour SGD with distributed backpropagation as discussed in Section 4.2.The training and test data was divided in the ratio 4:1.

2. **CNN**: We used three convolution layers, two mean pooling layers and a FC layer at the output.The first convolution layer contained kernel of size 5 X 5 with 6 feature maps followed by a mean pooling layer of size 2 X 2.This was followed by

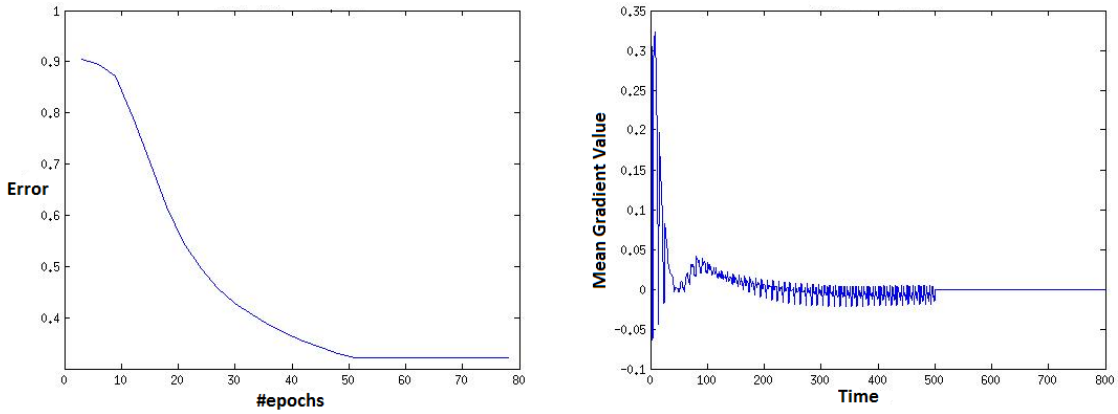
a second convolution layer containing kernel of size 5 X 5 with 12 feature maps followed by a mean pooling layer of size 2 X 2.The third convolution layer had 12 feature maps with kernel of size 4 X 4.The last layer was a FC layer with softmax.Mini-batch gradient descent(with momentum) with batch size of 16 was used.

3. **SSDA**: The model implemented for SSDA was used for the application in noise-resilient image super-resolution discussed in Chapter 5 and the complete details of the architecture, training and network parameters can be found in Section 5.2 and Appendix B.

4. **RNN and LSTM**: We used a randomly sampled sequence length of 1000 characters as input for our network which consist of number of neurons at the input and output layer equal to the size of the vocabulary (number of valid characters).The number of LSTM/ RNN units was kept as 200.The unrolled network (in time) was trained using tBPTT algorithm with 100 time steps.The output was produced at the end of each epoch till 300 characters were sampled out from the network.The only difference between the implementations of RNN and LSTM was that the RNN cells were replaced by the LSTM blocks at the hidden layer.The LSTM block implemented was the one discussed in [Gra12] consisting of four gates and no peephole connections.Mini-batch gradient descent with batch size of 32 along with RMSprop [TH12] was used for optimization.Our developed implementation was a multilayer RNN/LSTM which is capable of handing variable length input and output sequences by padding and masking.

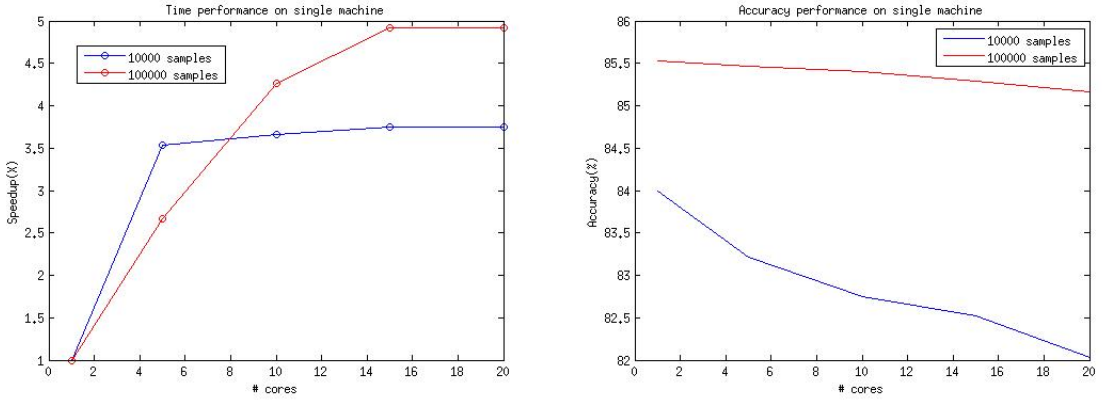
4.4 Results and Analysis

We started by obtaining plots 4.3a and 4.3b which experimentally prove the convergence of our proposed algorithm.In Fig.4.3a, we see that the net error converges with the number of epochs while Fig.4.3b shows that the mean gradient values converge to zero with time as is expected.



(a) Error Convergence Graph (b) Gradient value convergence

Figure 4.3: Plots demonstrating the correctness of the algorithm



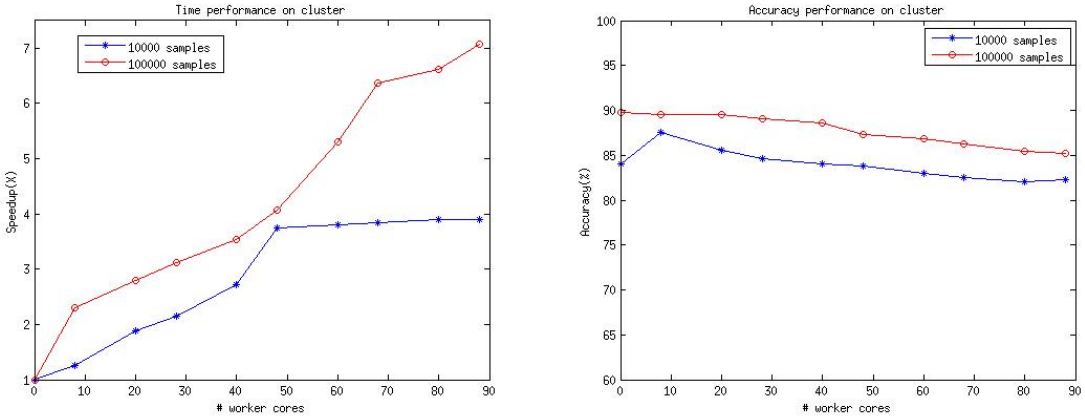
(a) Time performance (b) Accuracy performance

Figure 4.4: Performance on single machine for FC net

4.4.1 FC nets and CNN

In Fig.4.4a and Fig.4.4b, we show the performance of our framework on a single machine for the FC net with increase in the number of cores.We see that as the number of cores increases, we get more speedup due to increased degree of thread parallelization.The accuracy however reduces with the increase in the number of cores, though the decrease is less with increase in the number of data samples.

The results for cluster performance using FC nets are plotted in Fig.4.5a and Fig.4.5b; and using CNN are plotted in Fig.4.6a and Fig.4.6b for different sizes of the data samples.



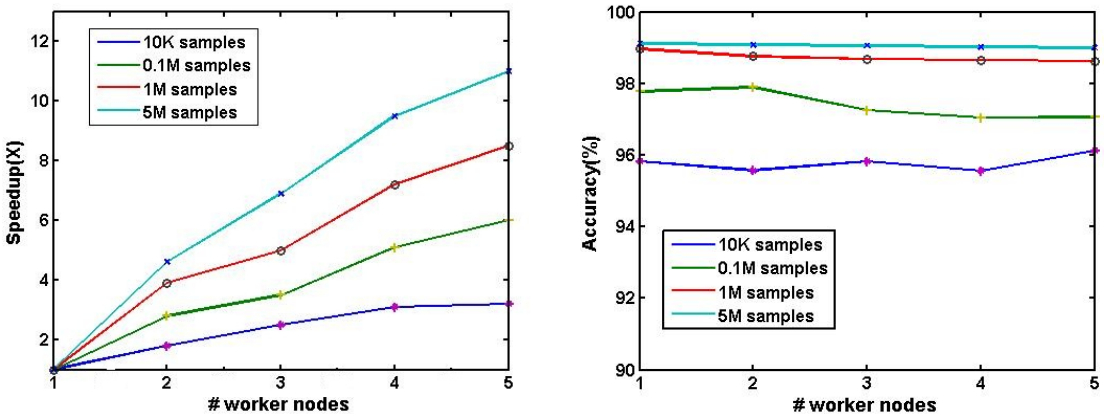
(a) Time Performance

(b) Accuracy Performance

Figure 4.5: Performance of the cluster for FC net

It can be seen from the plots that as the number of worker nodes/cores increases, the speedup increases gradually both in case of FC nets and CNN.There is some decrease in accuracy initially for small sizes of data samples but as the size of the data increases, the accuracy remains almost constant.Also, with large number of data samples, we get better values of accuracy as is expected.The accuracy values obtained for FC nets is rather low.The reason for this is that FC nets are not the best models for digit classification.With CNNs, we get reasonably good values of accuracies.

We have plotted the results for the variation of number of model parameters with number of partitions in Fig.4.7.A screenshot of the program execution has been shown in Fig.4.8 which captures the weights getting distributed to five different machines (corresponding to five different IP addresses).The speedup will be more for CNN a compared to FC net because of reduced communication overhead between the machines due to the local connectivity structure in CNN. We can see from the plot that as the size of model increases, we get more speedup as is expected.Also, it can be seen that for a given model size, there is not much improvement in performance after a certain



(a) Time Performance (b) Accuracy Performance

Figure 4.6: Performance of the cluster for CNN

number of partitions.This is as expected because the we can't obtain speedup beyond a certain point in distributed systems.However, the saturation point increases as we increase the model sizes.

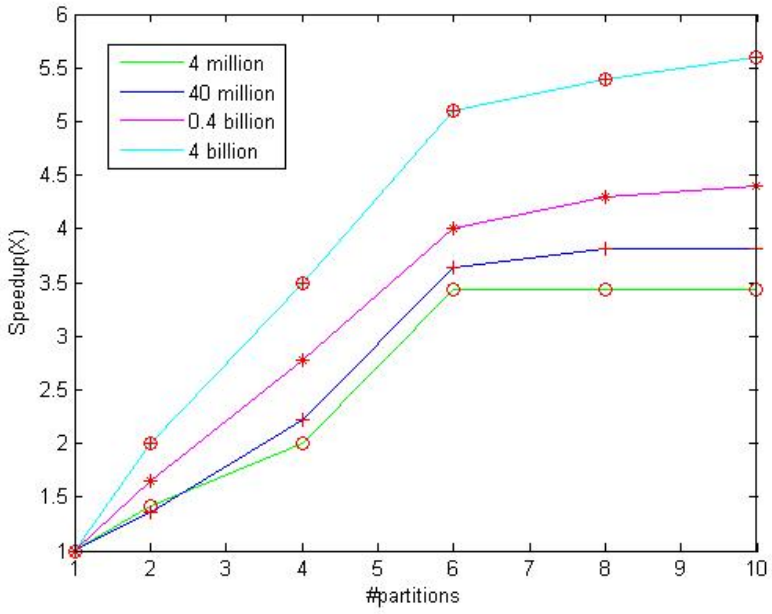


Figure 4.7: Performance for different sizes of model parameters

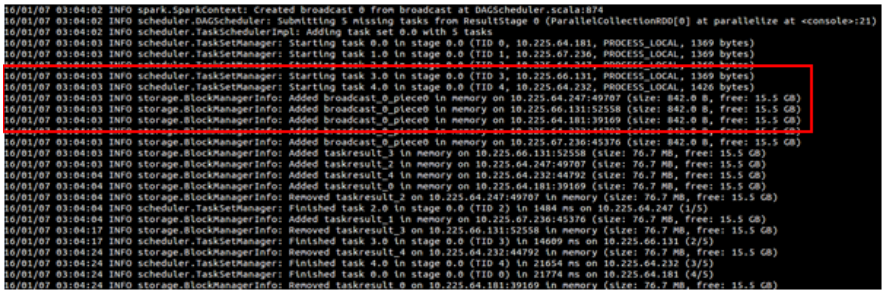
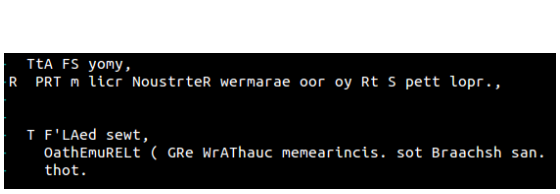


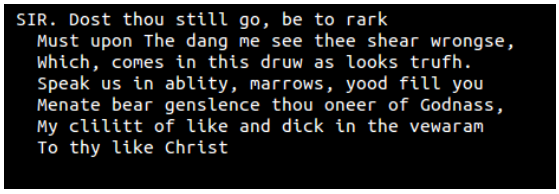
Figure 4.8: Screenshot showing the distribution of weights across the machines

4.4.2 RNN and LSTM

We also implemented a character-level predictor using RNN and LSTM.The training was done using tBPTT algorithm in both cases.Fig. 4.9a and Fig.4.9b shows the sample output obtained from RNN and LSTM after 5000 iterations of training on the text corpus containing the complete works of Shakespeare [Har71].As can be seen clearly the LSTM is able to learn the Shakespearean style much faster and in a better way as compared to RNN after training for same number of iterations. We have plotted the speedup obtained with increase in the number of worker machines in Fig.4.10.From the figure, it can be seen that the speedup increases uniformly with increase in the number of worker nodes thereby showing that our implementation of distributed LSTM is also scalable.



(a) RNN



(b) LSTM

Figure 4.9: Sample output comparison for RNN and LSTM after 5000 iterations

From the above results, it can be seen clearly that our framework performs better with more number of samples and larger size of model parameters and is quite scalable. Also, it is quite generic with implementations for majority of deep learning

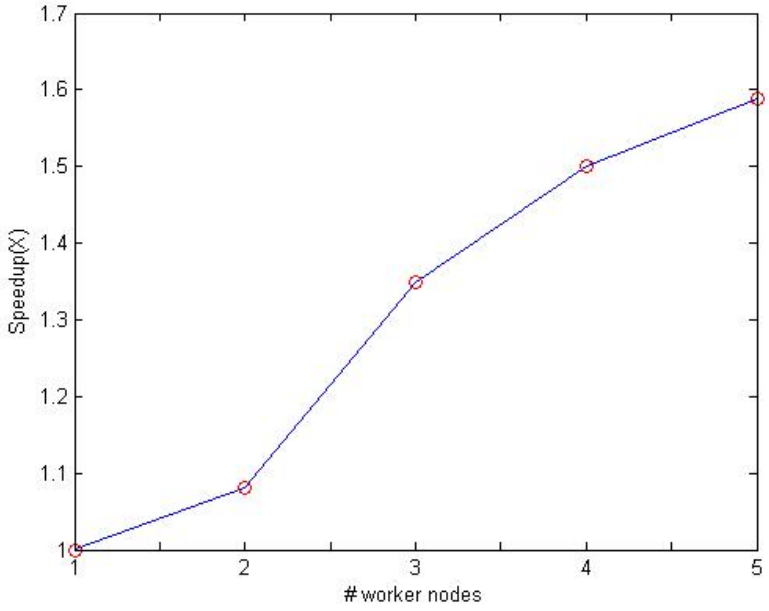


Figure 4.10: Scalability Performance for LSTM

architectures.A special point worth mentioning here is that all the implementations have been done from scratch.Till the time this thesis was written, there was no existing implementation of CNN, SSDA, RNN and LSTM in Apche Spark's MLLib/ML libraries [Spa14b].They just had the implementation of Multilayer perceptron that too implementing just data parallelism. This makes our work more novel and unique.In the coming chapter, we discuss the details of applying our developed framework for obtaining noise-free High Resolution image from noisy, Low Resolution image.

Chapter 5

Noise Resilient Image Super-Resolution

Having seen the performance of our framework on benchmark datasets, we tried to exploit the distributed and scalable nature of our developed framework in a real-world application which involves large amount of training data and model parameters.One such application is noise resilient image super-resolution which is widely used in a variety of computer vision tasks and bio-medical applications.Noise resilient image super-resolution is the task of getting High Resolution(HR) noise-free image from a noisy, Low Resolution(LR) image.This problem is challenging as texture and edge information is lost while using conventional approaches which mostly include denoising of image followed by super-resolution.

5.1 Related Work

Lots of work has been done in image denoising and super-resolution separately.Image denoising by wavelet filtering [LBU07] and wiener filtering [AP08] are the simplest techniques. Bayes least square with a Gaussian scale (BLS-GSM) [PSWS03] separates

noise from image in a transformed domain. In dictionary learning approaches [EA06], noisy patches and correct patches can be simultaneously represented as sparse linear combination of coupled dictionary atoms. EPLL [ZW11], BM3D [DFKE07] and deep-learning based technique [XXC12] are the best algorithms for image denoising. In image super-resolution, interpolation based approaches [DHX+07] were simpler but produced ringing and jagging artifacts.Manifold learning [CYX04] and sparse representation-based approach [YWHM10] were best until [NTA13, DLHT14] proposed deep learning techniques to learn the mapping between LR and HR patches. We found very less work where image denoising and super-resolution has been performed simultaneously. An algorithm which takes convex combination of frequency and orientation selective band of the denoised and noisy HR image is proposed in [SPA14a].A different framework by using median filter transform on parallelogram has been proposed by [LR16] to get noise free, super-resolution.

In this chapter, we propose a novel deep-learning based framework for noise free, image super resolution.We train the sparse stacked denoising auto-encoder (SSDA) for LR image denoising and deep convolutional neural network (deep CNN) for image super-resolution in a supervised way separately which is followed by cascading both the networks with learned weights and fine-tuning the combination on the dataset containing LR noisy image as input and corresponding HR (Ground Truth) images as target while preserving edge and textual information.Training our proposed framework on existing deep learning platforms was proving to be very time consuming. Due to the computational challenges in training our architecture,we made use of the distributed and scalable framework discussed before to achieve significant training speedup and obtain better performance by offering more flexibility.

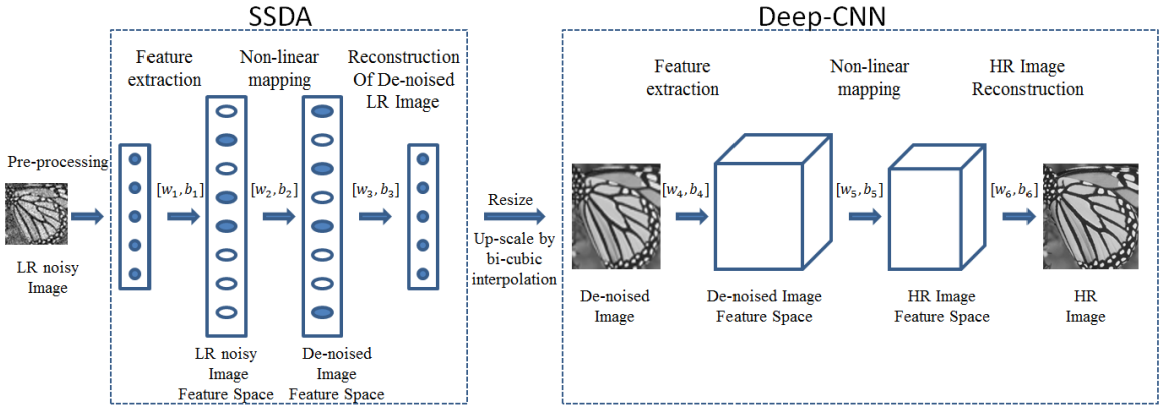


Figure 5.1: Block Diagram of the proposed architecture for noise-resilient image super-resolution

5.2 Architecture Used

5.2.1 Training of SSDA

Let x_i be the ith noisy LR image patch and y_i be the corresponding noise-free LR image patch $\forall i = 1, 2..N$ where, N is the total number of patches.

For the first layer of SSDA, the output is obtained as:

$$f_i = (w_1.x_i + b_1) \tag{5.1}$$

Here, w_1 and b_1 are the weight matrix and bias vector of the first layer respectively. Now LR image is reconstructed by

$$\hat{y_i} = (w_1^T.f_i + b_1^T) \tag{5.2}$$

To avoid over-fitting we enforce sparsity with SSDA. Reconstruction loss for pre-training of first layer of SSDA is then given as:

$$loss = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2} \parallel y_i - \hat{y_i} \parallel_2^2 + \beta KL(\hat{\rho} \parallel \rho) + \frac{\lambda}{2}\omega \tag{5.3}$$

Here, $\omega = \parallel w_1 \parallel_F^2 + \parallel w_1' \parallel_F^2$ is the weight regularization term, $\beta, \lambda, \hat{\rho} = 1/N \sum_{i=1}^{N} f_i$ are the sparsity, weight regularization coefficient and mean activation of hidden layer

respectively. $KL(\hat{\rho} \parallel \rho) = \rho.\log\frac{\rho}{\hat{\rho}} + (1-\rho).\log\frac{(1-\rho)}{(1-\hat{\rho})}$ is the KL divergence. After pre-training first layer of Stacked Denoising Autoencoder, we train the next layer by using the outputs of the preceding layer as inputs to the next layer.In a similar way we can stack desired number of layers.After pre-training each layer of SSDA, we fine-tune the learned weights to minimize the loss function given by (k is the number of layers):

$$F_{loss} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2} \parallel y_i - f_i^{finallayer} \parallel_2^2 + \frac{\lambda}{2}\sum_{j=1}^{k} \parallel w_j \parallel_F^2 \tag{5.4}$$

5.2.2 Training of deep CNN

The input to the deep CNN are denoised LR image patches obtained by inference of learned SSDA on noisy LR patches followed by enlarging them using bi-cubic interpolation up-to desired size (y_i^d) and target are HR image patches (z_i). We use the same architecture of deep CNN as described in [DLHT14].We learn the mapping F between HR image patches and enlarged LR image patches by minimizing the objective function given by:

$$T_{loss} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2} \parallel z_i - F(y_i^d) \parallel_2^2 \tag{5.5}$$

5.2.3 Training of Combined Network

After learning the SSDA and deep CNN, both the learned networks are cascaded as shown in Fig.5.1.We then fine-tune the whole network with the learned weights serving as initial weights on dataset having noisy LR patches as input and HR patches as target. Proposed framework learns image denoising and super-resolution simultaneously.This deep learning framework learns a mapping F between noisy LR patches and HR patches as $z_i = F(x_i)$.The mapping function F is a six layer operation which can be described as below:

- Feature extraction of noisy LR patches, $F1(x_i) = \sigma(x_i.w_1 + b_1)$
- Non-linear mapping of feature space of noisy LR patches into feature space of LR patches, $F2(x_i) = \sigma(F1(x_i).w_2 + b_2)$

- Reconstruction of denoised LR patches from their feature spaces, $F3(x_i) = F2(x_i).w_3 + b_3$.Then,resize the $F3(x_i)$ as a square matrix and upscale by bi-cubic interpolation up-to desired size.$F3(x_i)^* = \uparrow (resize(F3(x_i)))$

- Feature extraction of up-scaled denoised LR patches, $F4(x_i) = max(0, F3(x_i)^* w_4 + b_4)$

- Nonlinear mapping of feature space of up-scaled de-noised LR patches into HR patches feature space, $F5(x_i) = max(0, F4(x_i)^* w_5 + b_5)$

- Reconstruction of HR patches, $F(x_i) = F5(x_i)w_6 + b_6$

Here, w's are weight matrix of each layer and b's are biases vector corresponding to each layer. After fine-tuning the weights of proposed model, we can obtain a noise-free, super-resolved image from a noisy LR test image.

This framework contains millions of parameters(See Appendix B for details) and required training on large number(more than 1 million) of image patches to improve the generalization performance. Doing these on a single machine using existing deep-learning platforms was proving to be very time-consuming and computationally intensive with the training period ranging in weeks for training SSDA and deep CNN separately.The huge computational complexity of the framework and large memory requirements made it difficult to experiment with different values of hyperparameters as the model had to be trained again.To overcome this limitation, we made use of the distributed and scalable framework discussed before for training our model which resulted in significant speedup and allowed us more flexibility and improved performance.

5.3 Experiments

We implemented our proposed architecture for noise resilient image super-resolution in Apache Spark on a five node cluster as discussed earlier.For more details of the theory of developed distributed framework , refer to Chapter 3 and for details of experimental setup and implementation, refer to Section 4.2 and 4.3 of Chapter 4.We are

using ImageNet [RDS+15] dataset for training our framework.The exact details of the architecture and parameter values can be found in the Appendix B.To generate noisy input we add different types of noises to the downsampled ground truth image patches using inbuilt functions in Matlab.

Images	Conventional		State of Art		Our	
	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
Baby($\sigma = 20$)	30.69	0.8308	30.87	0.8324	32.54	0.8699
Lama($\sigma = 20$)	25.28	0.6106	25.31	0.6206	27.28	0.6676
Dog($\sigma = 20$)	28.31	0.7289	28.33	0.7337	29.98	0.8201
Lena($\sigma = 20$)	31.05	0.8423	31.13	0.8458	32.85	0.8738
Horse($\sigma = 25$)	24.09	0.5621	24.17	0.5659	27.27	0.6007
Porcupine($\sigma = 30$)	26.84	0.7902	27.12	0.7909	29.34	0.8300

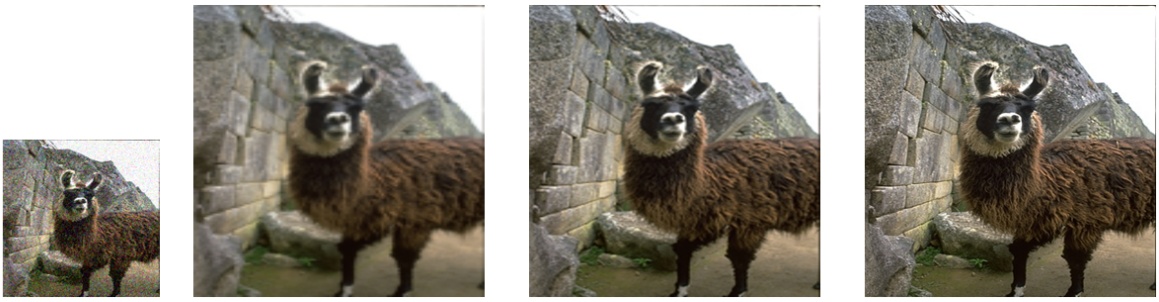
Table 5.1: Comparison of different methods with Gaussian noise

5.4 Results and Analysis

Results as shown in Table 5.1 and Fig.5.2a and Fig.5.2b demonstrate that our technique out-performs both the state of art method [SPA14a] and conventional method. Here, conventional method refers to best algorithm for image denoising followed by best algorithm for image super-resolution (i.e, SSDA [XXC12] + CNN [DLHT14]). For Gaussian noise, the improvement in terms of PSNR (Peak Signal to noise Ratio) and SSIM (Structural Similarity Index)[See Appendix B for definition] was 2.05 dB and 0.045 respectively when compared to state of art method; and 2.16 dB and 0.049 respectively when compared to conventional method. To test the robustness of our architecture, we applied other noises as well. The improvement in PSNR when compared to conventional method was 2.2 dB for blurring and 2.9 dB for salt and pepper

noise.The test images for Fig.5.2 were obtained from [MFTM01].To train the discussed noise-resilient image super-resolution architecture, our distributed framework took an average of 7s per epoch as compared to 60s per epoch in Caffe (without GPU in both) thereby achieving around 9X speedup.

(a) Dog



(b) Lama

Figure 5.2: Performance on test images: (i) Noisy LR (ii) HR by Conventional method (iii) HR by Our method (iv) Ground Truth HR

The last chapter concludes the discussion of our thesis with details of future scope and challenges faced.The mathematical proof of convergence of our proposed algorithm has been discussed in Appendix A ; and Appendix B describes the complete details of architecture and network parameters used for Noise Resilient Image Super-Resolution.

Chapter 6

Conclusions and Future Scope

6.1 Conclusions

In this thesis, we have developed a generic distributed and scalable framework for training and testing of deep networks in Apache Spark implementing both data parallelism and model parallelism.We have proposed a new algorithm for distributed backpropagation for the case when the neural network is partitioned across different machines along with detailed cost analysis and mathematical proof of convergence.We have successfully applied our framework to different neural network architectures like FC nets, CNN, SSDA, RNN and LSTM.Apache Spark has officially released its implementation of FC net only till the time this thesis was written.Moreover, their implementation just implements data parallelism and no model parallelism.The implementation of other deep learning frameworks like CNN, RNN , SSDA and LSTM is missing, which makes our developed framework novel and unique.After performing wide range of simulations, we have concluded that our framework is pretty scalable,i.e, the performance improves as we add more nodes to the cluster.We have achieved about 11X speedup with 5M samples for CNN just by just using a cluster of five nodes (CPU in all without GPU).

We have also applied our proposed framework in a novel architecture for noise free, image super resolution, which simultaneously performs image denoising and super-

resolution as well as preserves missing high frequency details.Results are more visually pleasing and have better PSNR and SSIM than conventional and state of art techniques and robust to different types of noises present in the training dataset. Moreover,the training time is reduced significantly(9X speedup) thereby allowing us to run our simulations with massive dataset and obtain better results.

We can conclude that our framework **performs best with increase in the size of data samples and large number of model parameters** and can be used conveniently for deep learning applications which require Big Data and Huge Model Sizes(which is now the case with almost all tasks) without requirement of expensive hardware.Hence,framework is distributed, scalable, economic and offers huge flexibility in terms of usage due to its integration with Spark which is a generic batch computational framework and its ability to rely on just a cluster of cheap commodity hardware thereby widening the range of users further.

6.2 Challenges Faced

- Due to the variation in CPU load over the nodes and network latency, simulations had to be averaged across multiple runs to obtain appropriate results.
- Since Apache Spark is relatively new as compared to other deep learning frameworks, development of code and debugging became quite tedious sometimes.It took significant time to get response from the developers of Apache Spark for resolving queries which were very specific and internal to Spark.

6.3 Future Scope

- The developed framework can be expanded to include more machines thereby increasing its computational power and obtaining improved performances for a variety of deep learning tasks.

- Models for Deep Belief Nets and Restricted Boltzmann machine(RBM) can be implemented making it a complete framework for deep networks.
- Integration of the developed framework with GPUs to achieve further speedup on individual machines utilizing the advantages of both cluster computing for scalability and distribution and the power of GPUs for carrying local computations.

Appendix A

Appendix A: Proof of Convergence

We proceed along the lines given in [LSZ09] and show that after making suitable modifications, our algorithm for distributed backpropagation reduces to the problem of delayed stochastic gradient descent and hence the generalized proof discussed in this paper can be applied to our problem as well; helping us arrive at the upper bound."Our goal is to minimize some parameter vector x such that the sum over convex functions $f_i : X- > R$ takes on the smallest value possible." This can be written as Eq.(A.1) or (A.2) as given below:

$$"f^*(x) := 1/|F| \sum_i f_i(x) \tag{A.1}$$

$$f^*(x) := E_{f\ p(f)}[f(x)]" \tag{A.2}$$

and correspondingly

$$x^* := argmin_{x \in \chi} f^*(x) \tag{A.3}$$

We know that sum of convex functions in the same domain is also convex. In our model, these correspond to the below quantities in our distributed back propagation algorithm:

Forward Pass => f_i: sum of a_is from different machines at the master process with an average delay τ_1

Backward Pass => f_i: sum of δ_js from different machines at the master process with an average delay τ_2.

Since the calculation of gradient in back propagation consists of the forward and backward stages, we can write the net delay, τ as the weighted sum of delays during the two processes (α_1 and α_2 are scaling factor)

$$\tau = \frac{\alpha_1\tau_1 + \alpha_2\tau_2}{\alpha_1 + \alpha_2} \tag{A.4}$$

In general the update rule in Stochastic Gradient Descent can be written as below:

$$w(t+1) = w(t) - \eta_t \Delta f_{t-\tau}(x) \tag{A.5}$$

where, w(t) = weights at time t, η_t= Learning Rate,$\Delta f_t(x)$= Gradient of f(x) at time t.Bregman divergence is defined as:

$$D(x||x^*) = 1/2 \parallel x - x^* \parallel^2 \tag{A.6}$$

We directly take up three theorems defined in [LSZ09] to help us arrive at the final result.The detailed proof of these theorems can be found in [LSZ09].

"Theorem 1: If we assume the cost function f to be Lipschitz continuous with a constant L and $max_{x,x'\in\chi}$D($x^{'}$)$\leq$ F^2 given $\eta_t = \frac{\sigma}{\sqrt{t-\tau}}$ for some constant $\sigma > 0$ and T to be the total number of iterations, the regret of the delayed update algorithm is bounded by

$$R[X] \leq \sigma L^2\sqrt{T} + F^2\frac{\sqrt{T}}{\sigma} + L^2\frac{\sigma\tau^2}{2} + 2L^2\sigma\tau\sqrt{T} \tag{A.7}$$

Theorem 2: Suppose that the functions f_i are strongly convex with parameter $\lambda > 0$. If we choose the learning rate as $\eta_t = \frac{\sigma}{\sqrt{t-\tau}}$ for t $> \tau$ and $\eta_t = 0$ for t $< \tau$, then under the assumptions of Theorem 1 we have the following bound:

$$R[X] \leq \lambda\tau F^2 + [\frac{1}{2} + \tau]\frac{L^2}{\lambda}(1 + \tau + \log T)" \tag{A.8}$$

A small delay should not significantly impact the update.This condition is equivalent to saying that small change in the value of x should not lead to major changes in values of gradients.For this, we assume that the Lipschitz-continuity of the gradient of f.This can be stated as given below:

$$" \parallel \Delta f_t(x) - \Delta f_t(x^{'}) \parallel \leq H \parallel x - x^{'} \parallel " \tag{A.9}$$

"Theorem 3: Under the assumptions of Theorem 2, in particular, assuming that all functions f_i are i.i.d. and strongly convex with constant λ and corresponding learning rate $\eta_t = \frac{\sigma}{\sqrt{t-\tau}}$ and Eq.(9) holds, we have the following bound on the expected regret:

$$\begin{aligned} E[R[X]] \leq \frac{10}{9}[\lambda\tau F^2 + [\frac{1}{2} + \tau]\frac{L^2}{\lambda}[1 + \tau + \log(3\tau + (H\tau/\lambda))] \\ + \frac{L^2}{2\lambda}[1 + \log T] + \frac{\pi^2\tau^2 H L^2}{6\lambda^2}" \end{aligned} \tag{A.10}$$

Thus, we see that we have a dependency of the form $O(\tau \log \tau + \log T)$. We get two separate terms dependent on τ and T. This implies that when T is large, delay τ essentially gets averaged out in different iterations and our algorithm converges with the upper bound as given above.So, our algorithm will perform the best with more number of iterations(more data samples) and larger model sizes which was shown to be true from experimental results as well.

Appendix B

Appendix B: Architecture and Parameter Details for Noise Resilient Image Super-Resolution

We trained a three layer SSDA framework with input as noisy LR patches of size 16x16. We set $\lambda = 0.0001$, $\beta = 0.01$, $\rho = 0.05$ and number of neurons = 1280 for first two layers of SSDA and $\lambda = 0.0001$, β =0, $\rho = 1$ and number of neurons = 256 for the third layer.Our deep-CNN has three convolutional layers, first two of these followed by ReLU activation function.It is trained on up-scaled denoised LR patches of size 32x32 and corresponding HR patches of same size.We used kernel sizes of 9X9 with 64 output maps, 5X5 with 32 output maps and 5X5 with 1 output map for the three layers of the CNN.

Number of parameters to deal with while pre-training of SSDA = 2 X 256 X 1280 + 2 X 1280 X 1280 + 2 X 1280 X 256 = 4587520 (approx. 4.6 million)

Number of parameters to deal with while fine-tuning of SSDA = 256 X 1280 + 1280

X 1280 + 1280 X 256 = 2293760 (approx. 2.3 million)

Number of parameters for training deep-CNN = 9 X 9 X 64 + 5 X 5 X 32 X 64 + 5 X 5 X 32 = 57184 (approx. 57K)

Number of parameters to deal with while fin-tuning the cascaded architecture = 256 X 1280 + 1280 X 1280 + 1280 X 256 + 9 X 9 X 64 + 5 X 5 X 32 + 5 X 5 = 2299769 (approx. 2.3 million)

According to [psn], "**Peak Signal-to-Noise Ratio (PSNR)** is the ratio between the maximum possible power of a signal and the power of noise. Given a noise-free mXn monochrome image I and its noisy approximation K, Mean Squared Error(MSE) is defined as:

$$MSE = \frac{1}{m\,n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2 \tag{B.1}$$

The PSNR (in dB) is defined as:

$$\begin{aligned} PSNR &= 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \\ &= 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \\ &= 20 \cdot \log_{10}\left(MAX_I\right) - 10 \cdot \log_{10}\left(MSE\right) \end{aligned} \tag{B.2}$$

Here, MAX_I is the maximum possible pixel value of the image. When the pixels are represented using 8 bits per sample, this is 255. More generally, when samples are represented using linear PCM with B bits per sample, MAX_I is $2B - 1$."

Structural Similarity Index (SSIM) is used for measuring the visual similarity between two images. According to the definition given in [ssi], "The SSIM index is calculated on various windows of an image. The measure between two windows x and y of common size NXN is:"

$$\text{"SSIM}(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{B.3}$$

with

- μ_x = average of x

- μ_y = average of y
- σ_x^2 = variance of x
- σ_y^2 = variance of y
- σ_{xy} = covariance of x and y
- $c_1 = (k_1 L)^2, c_2 = (k_2 L)^2$ two variables to stabilize the division with weak denominator
- L = dynamic range of pixel values($2^{\#bitsperpixel} - 1$)
- $k_1 = 0.01 and k_2 = 0.03$ by default"

"In order to evaluate the image quality this formula is usually applied only on luma, although it may also be applied on color (e.g., RGB) values or chromatic (e.g. YCbCr) values. The resultant SSIM index is a decimal value between -1 and 1, and value 1 is only reachable in the case of two identical sets of data."

Bibliography

[AAB+16] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[AP08] Raman Arora and Harish Parthasarathy. Spherical wiener filter. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*. IEEE, 2008.

[Bra13] L Braun. Mikio. jblas-linear algebra for java, 2013.

[CSAK14] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[CYX04] Hong Chang, Dit-Yan Yeung, and Yimin Xiong. Super-resolution through neighbor embedding. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1. IEEE, 2004.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, 2012.

[DFKE07] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. Image denoising by sparse 3-d transform-domain collaborative filtering. *Image Processing, IEEE Transactions on*, 16(8), 2007.

[DHX+07] Shengyang Dai, Mei Han, Wei Xu, Ying Wu, and Yihong Gong. Soft edge smoothness prior for alpha channel super resolution. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007.

[DLHT14] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *Computer Vision–ECCV 2014*. Springer, 2014.

[EA06] Michael Elad and Michal Aharon. Image denoising via sparse and redundant representations over learned dictionaries. *Image Processing, IEEE Transactions on*, 15(12), 2006.

[Gra12] Alex Graves. Neural networks. In *Supervised Sequence Labelling with Recurrent Neural Networks*, pages 15–35. Springer, 2012.

[Har71] Michael Hart. *Project gutenberg*. Project Gutenberg, 1971.

[HCC+13] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[HN89] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 593–605. IEEE, 1989.

[IAMK15] Forrest N Iandola, Khalid Ashraf, Mattthew W Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *arXiv preprint arXiv:1511.00175*, 2015.

[LAP+14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[LBU07] Florian Luisier, Thierry Blu, and Michael Unser. A new sure approach to image denoising: Interscale orthonormal wavelet thresholding. *Image Processing, IEEE Transactions on*, 16(3), 2007.

[LR16] Ezequiel López-Rubio. Superresolution from a single noisy image by the median filter transform. *SIAM Journal on Imaging Sciences*, 9(1), 2016.

[LSZ09] John Langford, Alexander Smola, and Martin Zinkevich. Slow learners are fast. *arXiv preprint arXiv:0911.0491*, 2009.

[mal] Mallet. http://mallet.cs.umass.edu/topics.php.

[MFTM01] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision*, 2001.

[mni] Extended mnist dataset. http://leon.bottou.org/papers/loosli-canu-bottou-2006.

[MNSJ15] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[NTA13] Toru Nakashika, Tetsuya Takiguchi, and Yasuo Ariki. High-frequency restoration using deep belief nets for super-resolution. In *Signal-Image Technology & Internet-Based Systems (SITIS), 2013 International Conference on*. IEEE, 2013.

[PLWH03] Mark Pethick, Michael Liddle, Paul Werstein, and Zhiyi Huang. Parallelization of a backpropagation neural network on a cluster computer. In *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[psn] Psnr definition. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio.

[PSWS03] Javier Portilla, Vasily Strela, Martin J Wainwright, and Eero P Simoncelli. Image denoising using scale mixtures of gaussians in the wavelet domain. *Image Processing, IEEE Transactions on*, 12(11), 2003.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3), 2015.

[SPA14a] Abhishek Singh, Fatih Porikli, and Narendra Ahuja. Super-resolving noisy images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[Spa14b] Apache Spark. Apache sparkâĎć-lightning-fast cluster computing, 2014.

[ssi] Ssim definition. https://en.wikipedia.org/wiki/Structural_similarity.

[TH12] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

[XXC12] Junyuan Xie, Linli Xu, and Enhong Chen. Image denoising and inpainting with deep neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[YWHM10] Jianchao Yang, John Wright, Thomas S Huang, and Yi Ma. Image super-resolution via sparse representation. *Image Processing, IEEE Transactions on*, 19(11), 2010.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[ZLS09] Martin Zinkevich, John Langford, and Alex J Smola. Slow learners are fast. In *Advances in Neural Information Processing Systems*, 2009.

[ZW11] Daniel Zoran and Yair Weiss. From learning models of natural image patches to whole image restoration. In *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011.

[ZWLS10] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, 2010.