Alternate Layer Sparsity and Intermediate Fine-tuning for Deep Autoencoders

A thesis submitted in partial fulfilment of the requirements

for the degree of Master of Technology

by

Ankit Bhutani

Supervised by

Prof. Amitabha Mukerjee

Prof. K S Venkatesh

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

July 2014

Certificate

It is certified that the work contained in this thesis entitled ' Alternate Layer Sparsity and Intermediate Fine-tuning for Deep Autoencoders' by Ankit Bhutani (Roll No. Y9227094) has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Prof. Amitabha Mukerjee,

Department of Computer Science and Engineering,

Indian Institute of Technology, Kanpur

Kanpur - 208016

Prof. K S Venkatesh,

Department of Electrical Engineering,

Indian Institute of Technology, Kanpur

Kanpur - 208016

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

Abstract

Department of Electrical Engineering

Master of Technology

Alternate Layer Sparsity and Intermediate Fine-tuning for Deep Autoencoders

by Ankit Bhutani

In this work, we present two new strategies to improve dimensionality reduction using deep autoencoders. It is shown that by enforcing sparsity and bottleneck constraints on alternate layers of a deep autoencoder, the performance of deterministic auto-associative pre-training models like shallow autoencoders can be made comparable to, and in some cases even better than, stochastic models like Restricted Boltzmann Machines (RBMs). We also introduce a new technique called intermediate fine-tuning which improves the performance of all deep autoencoder models which employ stackable modules like RBMs for pre-training. The improvement in performance is demonstrated for the benchmark MNIST dataset as well as some synthetic datasets for which the intrinsic dimensionality is explicitly known.

Acknowledgements

I would like to take this opportunity to thank the people who have helped me in preparation of this thesis. Firstly, I would like to thank my parents without whose unconditional love, support, guidance, motivation, sacrifices and encouragement, this thesis would never have been possible. I shall remain forever indebted to them. I am also thankful to my brother and sister-in-law for supporting and encouraging me.

I would like to express my respect and regards for both my supervisors who guided me for the past one and a half years. I am grateful to them for providing me ample freedom to choose a topic of my interest and deciding how to pursue it. They were always available for providing their valuable advice and have been a source of inspiration for me.

I would also like to thank my friends and wing-mates who have made my stay at IIT Kanpur in general and the past one and a half years of thesis work in particular, most enjoyable. There have been times during the course of this thesis when I have been disheartened by failures. It was these friends who have always managed to bring a smile on my face even in the face of disappointments and kept me going.

Last but not the least, I can't thank IIT Kanpur enough for the numerous opportunities, its wonderful research facilities and excellent academic training which enabled me to complete this thesis.

Contents

List of Figures

List of Tables

Dedicated to

my parents

Chapter 1

Introduction

In many situations, data can be represented in much fewer dimensions than the number of dimensions in which it is originally available. Reducing the dimensionality of data can be extremely beneficial as it can cause magnitudes of reduction in data saving costs, transmission costs and computation time for various tasks. Besides this, dimensionality reduction finds use in many research areas as a preprocessing step to reduce the redundancy or maximize discrimination. One of the numerous possible examples is in HMM based phonetic recognition ([Hu and Zahorian, 2010]).

It is for such reasons that dimensionality reduction has been one of the key areas of research for a long time. Based on the type of data in question, many methods like principal component analysis (PCA), independent component analysis (ICA), locally linear embeddings, ([Roweis and Saul, 2000]), Isomap ([Tenenbaum et al.,

2000]) etc. have been proposed. For a detailed review on the recent advances in dimensionality reduction, see [van der Maaten et al., 2009].

In certain situations like robot motion planning, it is desirable to be able to reconstruct the original data from its reduced form or for producing data in original space using new points in reduced space (also called out-of-sample reconstruction[1]). Many popular algorithms used commonly are able to reduce the dimensionality of data but do not have explicit methods for reconstructing the original data. One of the methods that is capable of such reconstructions is the autoencoder which forms the focus of our present work.

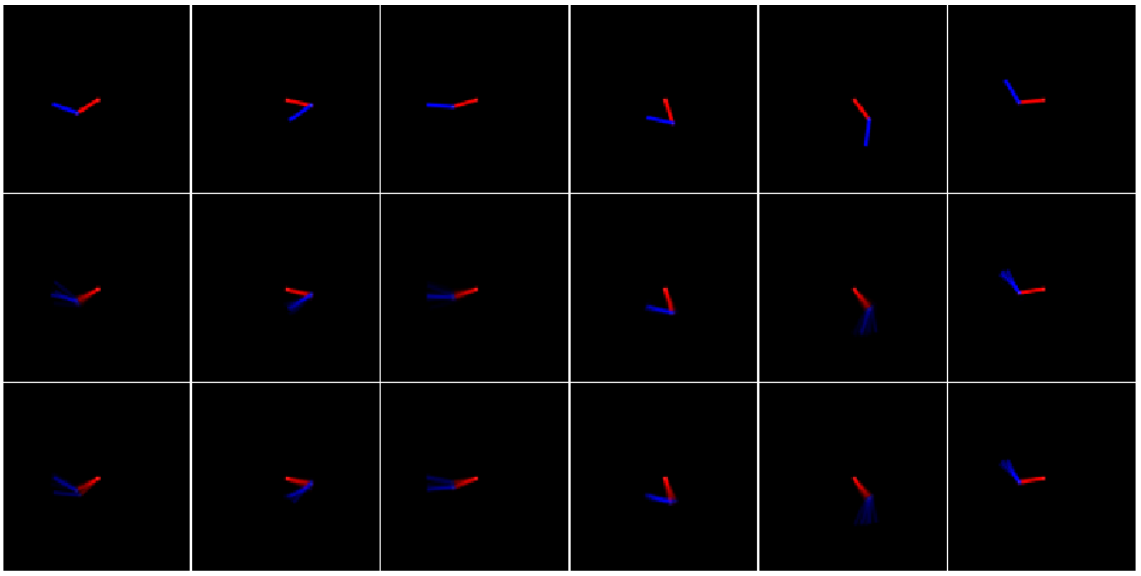


FIGURE 1.1: Out of sample reconstruction using Local Quadrature Reconstruction (LQR) and Linear-Reconstruction: Top row - Original Data, Middle Row - Reconstruction using LQR, bottom row - Reconstruction using Linear-Reconstruction. ([Dhingra, 2013])

In the next few sections, we formally introduce the problem being tackled and review the relevant literature.

[1]For details, please refer [Dhingra, 2013] or [Bengio et al., 2004]

1.1 Autoencoders and Deep Learning

Auto-associative neural networks or autoencoders were first introduced by [Rumelhart et al., 1986] for learning more about the structure of data without using any labels i.e. to enable unsupervised learning. The defining criterion of such networks is that they seek to produce an output which is very similar to the input while satisfying some constraints within the network. One common constraint is the bottleneck constraint in which the a hidden layer of the neural network has less number of dimensions are compared to the input and output layer. Such a constraint forces the network to learn a compressed representation of the data thereby enabling dimensionality reduction.

[Baldi and Hornik, 1989] showed that an autoencoder with linear activations and mean-squared error as loss function (for terminology see section 2.1) learns to extract the principal components of the data i.e. the autoencoder could be used to perform PCA. [Kramer, 1991] extended this idea to form non-linear PCA using a 5-layered network with alternate linear and sigmoid activations.

Autoencoders have since been found useful for a variety of tasks like dimensionality reduction ([Zemel and Hinton, 1995]), novelty detection ([Thompson et al., 2002], [Japkowicz et al., 1995]), data retrieval ([Thompson et al., 2003]) etc. Though it was well-agreed that there are several advantages[2] of deep neural networks that are extremely hard or costly (in terms of memory usage or computation time) to obtain through shallower ones; still most of the work before 2006 remained restricted to relatively shallow autoencoders (only 2-3 hidden layers) as deep autoencoders

[2]For a recent opinion on advantages of deep architectures, see chapter 2 of [Bengio, 2009]

with non-linear activation functions were highly prone to problems like vanishing of error gradients in lower layers and the optimization being stuck in shallow local minima. These problems were not just specific to autoencoders but to neural networks in general and not much progress could be made on deep neural networks until 2006[3].

It is theorized that due to the non-convex nature of the involved optimization and several other factors, any gradient based method used to train a deep network "works well only if the initial weights are close to a good solution" [Hinton and Salakhutdinov, 2006]. [Hinton et al., 2006] showed that Restricted Boltzmann Machines [Smolensky, 1986] could be trained generatively in a greedy unsupervised manner to find a good set of weights for a deep belief network. Using these weights to initialize the network and then fine-tuning it based on the supervised objective provided results which were comparable to, and in many cases better than, the state of the art at that time. This unsupervised layer-wise training of RBMs before supervised training is known as pre-training while the supervised part is called fine-tuning. Using a similar procedure, [Hinton and Salakhutdinov, 2006] showed that RBMs could also be used to learn good initial weights for a deep autoencoder (this is covered in detail in section 2.3).

On similar lines as above, [Ranzato et al., 2007] and [Bengio et al., 2007] showed that deterministic shallow autoencoders (covered in detail in section 2.1) could also be used to find good initial weights for supervised tasks. However, these initial results using shallow autoencoders were not as good as the ones obtained

[3]One notable exception to this is Convolutional Neural Networks (see [LeCun and Bengio, 1995])

using RBMs. [Vincent et al., 2010] using denoising autoencoders (section 2.1.2) and [Rifai et al., 2011] using contractive autoencoders were able to achieve results comparable to the ones achieved by RBMs.

Though both RBMs and shallow autoencoders have been shown to achieve good results in finding initial weights for both supervised tasks as well as deep autoencoders and RBMs have been used to find good initial weights for deep autoencoders also but the use of autoencoders for finding good initial weights for deep autoencoders has not been explored in detail.

In this work, we first try to bridge this gap and then look for strategies to improve the performance of both deterministic models like shallow autoencoders and stochastic models like Restricted Boltzmann machines for the task of dimensionality reduction using deep autoencoders.

1.2 Main Contributions of the work

Following are the main contributions of this thesis:

1. **Establishing performance benchmarks for training deep autoencoders using deterministic pre-training models:** As mentioned earlier, the use of shallow autoencoders for pre-training of deep autoencoders has not been studied in detail. In chapter 3, we provide results from several

experiments showing the performance by using shallow autoencoders for pre-training. The effect of varying autoencoder types and layer-sizes has been studied in detail.

2. **Bridging the performance between stochastic and deterministic pre-training models:** Our preliminary experiments in chapter 3 reveal that pre-training with Restricted Boltzmann Machines (RBMs) shows better results than the pre-training with shallow autoencoders commonly used in literature[4]. However, we show that by introducing some modifications, the performance of shallow autoencoders can be made comparable to, and in some cases even better than, RBMs.

3. **Introduction of alternate sparsity:** Sparsity has been exploited in neural networks mainly for maximizing discrimination and making the learnt features easily interpretable. In this work, we show that by using sparsity and bottleneck constraints on alternate layers of a deep autoencoder, better reconstruction performance can be achieved in dimensionality reduction tasks.

4. **Introduction of inter-mediate fine-tuning:** In chapter 4, we introduce a new strategy for improving the reconstruction performance of deep autoencoders. We show that for very deep autoencoders, inter-mediate fine-tuning significantly improves the performance of all kinds of models considered in this work.

[4]Note that the use of shallow autoencoders in literature is mainly for pre-training of deep classifier networks and not for the kind of tasks for which we are using them here

In the remaining chapters, we provide the details of the above mentioned work. Chapter 2 lays down the basic framework for our experiments by explaining the models, datasets and resources used in this work. Chapters 3 and 4 discuss the modifications introduced by us and and their benefits. Chapter 5 concludes our discussion and makes some suggestions for possible related future work.

Chapter 2

Preliminaries

In this chapter, we explain the existing theoretical models necessary for understanding the modifications introduced by us in the upcoming chapters. We begin our discussion by reviewing the shallow autoencoder model and its variants in section 2.1. Restricted Boltzmann Machines are covered in section 2.2. Section 2.3 deals with Deep Autoencoder and its training for dimensionality reduction which is the main focus of this work. Remaining sections give a brief description of the resources used by us and the datasets on which the experiments are conducted.

2.1 Shallow Autoencoder

Auto-associative networks or autoencoders are neural networks which are usually trained for producing an output which is quite similar to the input fed to them. For the sake of compactness, we shall refer to auto-associative networks with only 1 hidden layer as shallow autoencoders or simply autoencoders. Autoencoders

with more than one hidden layer shall be referred to as deep autoencoders. In this section, we first explain the basic autoencoder and then present some of its popular variants.

2.1.1 Basic Autoencoder

Given an input data in the form of a vector $X = [x_1, x_2, x_3, \ldots, x_{n_v}]$ with n_v dimensions, a basic autoencoder with n_h number of hidden units is a neural network which can be characterized by 4 parameters, 2 equations and the objective of optimally reconstructing the input data. The parameters are:

- W_1: An $n_v \times n_h$ real-valued matrix whose elements $w_{ij}^{(1)}$ represent the weight on the connection between i^{th} component of input vector and j^{th} hidden unit
- W_2: An $n_h \times n_v$ real-valued matrix whose elements $w_{ij}^{(2)}$ represent the weight on the connection between i^{th} hidden unit and j^{th} component of output vector
- B_1: An n_h dimensional real-valued vector whose components $b_i^{(1)}$ represent the bias on i^{th} hidden unit
- B_2: An n_v dimensional real-valued vector whose components $b_i^{(2)}$ represent the bias on i^{th} component of output vector

The governing equations are:

$$y_j = f_1\left(\sum_{i=1}^{n_v} x_i w_{ij}^{(1)} + b_j^{(1)}\right) \tag{2.1}$$

$$z_j = f_2\left(\sum_{i=1}^{n_h} y_i w_{ij}^{(2)} + b_j^{(2)}\right) \tag{2.2}$$

where $Y = [y_1, y_2, \ldots, y_{n_h}]$ represent the activations of hidden units and $Z = [z_1, z_2, \ldots, z_{n_v}]$ represent output of the network. f_1 and f_2 are known as activation

functions. Commonly used activation function are sigmoid, tanh, rectified linear units or the most basic linear activation. In this work, we have made use of only linear and sigmoid activations. Linear Activation is the identity function ($f(t) = t$) whereas the sigmoid function is given below:

$$sigmoid(t) = \frac{1}{1 + exp(-t)} \tag{2.3}$$

The objective of training the network is to adjust the parameters W_1, W_2, B_1 and B_2 in such a way that the output vector Z becomes as similar as possible to the input vector X. Mathematically, the difference between the output and input is measured using a loss function. The most commonly used loss functions are Mean-squared error for real-valued data and reverse cross-entropy for probabilistic as well as binary data. As the experiments considered in this work involve only binary and probabilistic data, we have used reverse cross-entropy as the loss-function. Mathematically, reverse cross-entropy is defined as follows:

$$L_{cross-entropy}(X, Z; \theta) = -\sum_{i=1}^{n_v} (x_i \log z_i + (1 - x_i) \log(1 - z_i)) \tag{2.4}$$

where θ corresponds to the set of parameters W_1, W_2, B_1, B_2. The adjustment of parameters or training is carried out using the standard backpropagation algorithm ([Rumelhart et al., 1986]), which essentially performs a gradient descent on the loss-function, or one of its many variants.

It can be easily seen that in the trivial case of an autoencoder with linear activation functions and $n_h > n_v$, it is possible to produce an identity function from the

network. In the more interesting case of $n_h < n_v$, [Baldi and Hornik, 1989] showed that at global minimum of mean-squared error between the output and input, the network learns to extract the first n_h principal components of the data as the hidden activations and that this error surface is convex. However, this is not true for the case of sigmoid or other non-linear activation functions.

In the more general scenario of non-linear activation functions, an autoencoder network can be thought of as attempting to learn a representation of the input in an n_h dimensional space in such a way that using the set of weights W_2, the original input may be reconstructed appropriately. In case of $n_h < n_v$, this network can be seen as trying to find a low-dimensional representation of the data.

In general, the weight matrix W_2 is restricted to be equal to the transpose of matrix W_1 throughout the training to avoid learning of uninteresting or trivial representations. Training paradigms following this type of restriction are said to be using 'tied weights'. Unless otherwise stated, weights are assumed to be tied for all the experiments in this work where shallow autoencoders are being trained.

2.1.2 Denoising Autoencoder

The denoising autoencoder is a basic autoencoder only with one major difference. In denoising autoencoder, the input is first partially corrupted and then fed to the network. The network is trained to optimally reconstruct the original input from this partially corrupted one. The main rationale behind using this criterion is that it forces the autoencoder to learn the main underlying structure in the data which

may be sufficient to appropriately reconstruct the original input vector. Denoising autoencoders have been used by [Vincent et al., 2010] to find good initial weights (pre-training) for training deep multilayer neural networks used for classification.

2.1.3 Contractive Autoencoder

Contractive Autoencoders attempt to learn robust features for classification by reducing sensitivity of the hidden layer activations to the input. The loss function which contractive autoencoders seek to minimize comprises of 2 terms out of which one is the usual reverse cross-entropy. The second term is the "Frobenius norm of the Jacobian matrix of the encoder activations with respect to the input" [Rifai et al., 2011]. Using the same notation as used earlier, the modified loss function is:

$$L_{contractive}(X, Z; \theta) = -\sum_{i=1}^{n_v} (x_i \log z_i + (1 - x_i) \log(1 - z_i)) + \lambda \|J_{f_1}(X)\|_F^2 \quad (2.5)$$

where λ is the parameter which can be used to control the weight being given to the penalty term which is calculated as follows:

$$\|J_{f_1}(X)\|_F^2 = \sum_{i,j} \left(\frac{\partial y_j}{\partial x_i} \right)^2 \quad (2.6)$$

where Y=$[y_1, y_2, \ldots, y_{n_h}]$ represent the hidden layer activations. Except for the modification in loss function and hence in the gradient, the rest of the training procedure remains the same.

2.1.4 Sparse Autoencoder

Though there are a lot of methods for using sparse representations in neural network training, but for the present work, we use the formulation as given in [Ng, 2011]. Similar to the contractive autoencoders, sparse autoencoders also work by introducing an additional term to the loss function. This additional term penalizes any deviations in average activation of a hidden unit from a specific (usually very low) value. To understand this, consider a training set of m examples $X_1, X_2, \ldots$ till X_m. The average activation of a hidden unit over this training set can be computed as:

$$\bar{y_j} = \sum_{i=1}^{m} y_j(X_i) \tag{2.7}$$

In order to have a sparse representation, it is desirable to have a low average activation for all the hidden units. Let y_d represent a desirable value of average activation. Then, the penalty term essentially represents the KL-divergence between a distribution represented by the average activation of hidden units and a distribution where the average activation of every unit is y_d. Mathematically,

$$E(\bar{Y}; y_d) = \sum_{i=1}^{n_h} \left(y_d \log \frac{y_d}{\bar{y_i}} + (1 - y_d) \log \frac{1 - y_d}{1 - \bar{y_i}} \right) \tag{2.8}$$

Thus, the final loss function summed over all training examples is:

$$L = \sum_{i=1}^{m} L_{cross-entropy}(X_i, Z_i; \theta) + \lambda E(\bar{Y}; y_d) \tag{2.9}$$

where λ is a parameter that can be changed to modify the weight assigned to the penalty term.

2.2 Restricted Boltzmann Machine (RBM)

Restricted Boltzmann machines are energy-based bidirectional bipartite graphical models that can be used for modeling probability distribution over a set of inputs. Given a set of inputs with n_h dimensions, it tries to find a mapping from the input space to an n_h dimensional space such that the likelihood of the input examples being generated from their corresponding n_h dimensional representations is maximized. This maximization problem can be shown to be equivalent to minimizing the Gibbs free energy of the graphical model. The exact solutions to problem are in general extremely time-consuming but the same can be approximated by making use of Gibbs sampling which is a Markov Chain Monte-Carlo (MCMC) method. Just executing 1 step of the corresponding Markov chain is sufficient to obtain the samples required for approximating the required objective. The process most commonly used for training RBMs is Contrastive Divergence. More details on RBMs and contrastive divergence can be found in [Fischer and Igel, 2012], [Bengio and Delalleau, 2009] and [Bengio, 2009]

2.3 Deep Autoencoder

Deep Autoencoders are auto-associative networks with more than one hidden layer. Though, the defining feature of autoencoders (both deep and shallow) is the property to have the output of the network as similar as possible to the input but for the present work, we consider only fully-connected symmetric autoencoders. To understand what we mean by symmetric full-connected deep autoencoders, consider the deep autoencoder shown in fig. 2.1. Symmetry here indicates that the number of hidden units in layer L1, or equivalently its size, must be equal to that of layer M1. Similarly, size of L2 and M2 must be equal. Quite trivially, the size of output layer must be equal to the input. Fully connected means that every unit in a hidden layer is connected to every unit in the succeeding and preceding layers. In shallow autoencoders considered earlier, the hidden layer was fully connected.

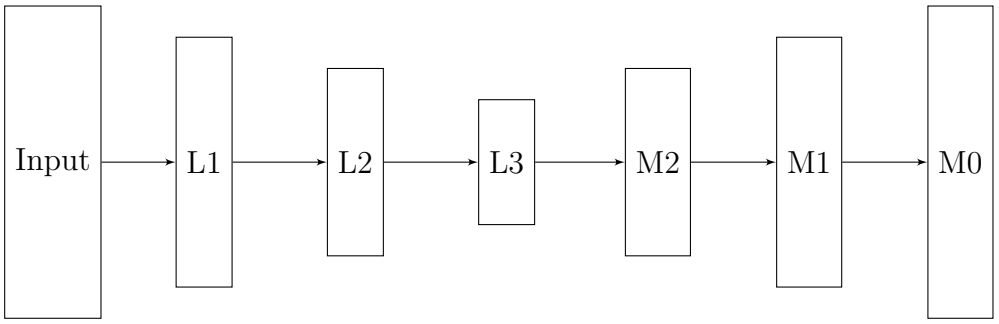


FIGURE 2.1: Deep Autoencoder
For deep autoencoders considered in this work, size of layer L1 is same as M1 while size of layer L2 is same as that of M2. Similarly, size of layer M0 is same as the dimensionality of the input.

As the deep autoencoders we consider are symmetric, in order to describe the architecture of the network we only describe the number of units in each layer of the first half. For example, if the input dimensionality is 2000 and we we want the layers of the autoencoder to have 1000, 500, 250, 500 and 1000 units respectively

in the 5 hidden layers, then we give the architecture as -1000-500-250- . The number of units in the output layer can be inferred directly as it is equal to the dimensionality of the input.

2.3.1 Training Deep Autoencoders using RBMs

The methodology followed by us for training deep autoencoders is similar to that introduced by [Hinton and Salakhutdinov, 2006]. We first explain their method which uses RBMs for pre-training and then describe our framework which uses autoencoders in place of RBMs. To understand this, let's assume that the dimensionality of the input is n_v and the required network architecture is $-n_1-n_2-n_3-$. The training is carried out in 2 phases namely pre-training and fine-tuning as described below:

Pre-training: In the beginning, an RBM with n_1 hidden units and n_v visible units is trained to model the input distribution. Once this RBM has been trained, the activations of hidden layer are obtained for the complete data. Thus, for every example in the dataset, we now have a binary vector with n_1 units. This new dataset of n_1 dimensions is now used to train an RBM with n_2 hidden units and n_1 visible units. Similarly, after training this second RBM, a new dataset of n_2 units is obtained through the its hidden activations which is further used to train a third RBM with n_3 hidden units. This completes the pre-training phase.

Fine-tuning: After the completion of pre-training phase, we have 3 sets of weights, one for each RBM trained in the pre-training phase. These weights are

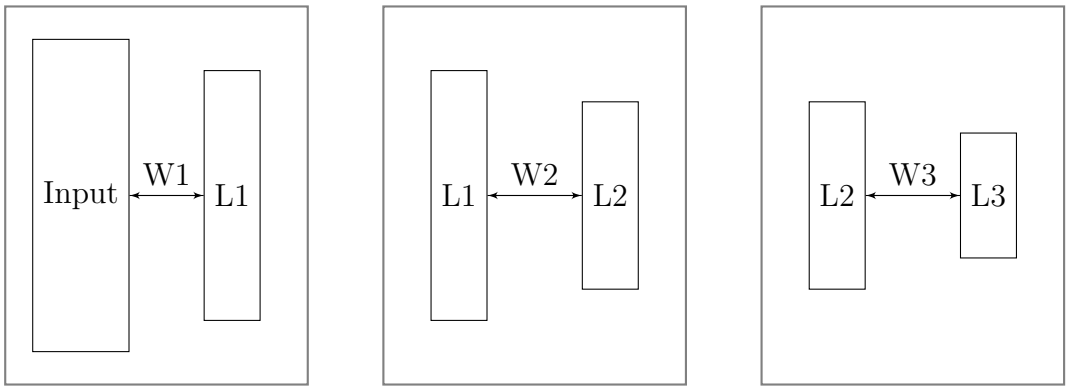


FIGURE 2.2: Pre-training of the autoencoder

used to initialize the first half of a deep autoencoder with the desired architecture. The 2nd half of the autoencoder is initialized with the transposed weights of the corresponding layer from the first half. That is weights of the connections between L3 and M2 are initialized by using the transposed weight matrix of the weights between L2 and L3. Similarly, weights between M2 and M1 are initialized by transposing the weights between L1 and L2. Further, the weights between M1 and the output layer M0 are initialized by transposing the weights between input layer and L1. After this initialization is complete, now the network is trained by the usual backpropagation algorithm ([Rumelhart et al., 1986]) or any of its variants.

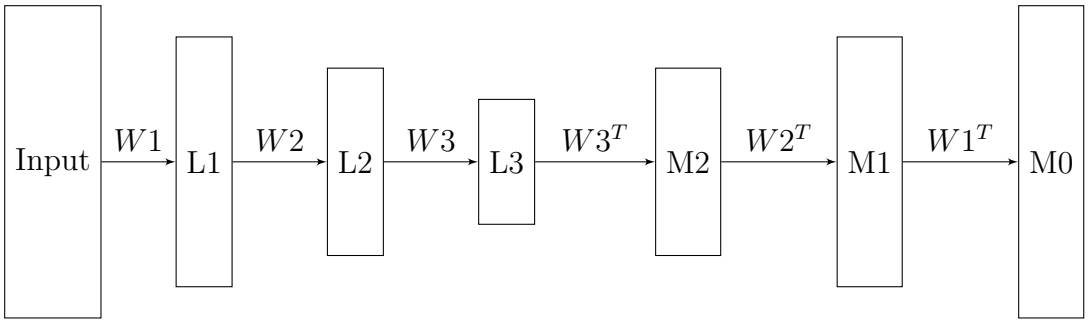


FIGURE 2.3: Initializing the deep autoencoder

2.3.2 Training Deep Autoencoders using shallow autoencoders

Similar to RBMs, shallow autoencoders can also be used for the pre-training phase. Shallow autoencoders are also pre-trained in a layer-wise fashion as done above. First an autoencoder is trained on the original input. After its training is complete, the hidden activations of the autoencoder are used to obtain a new dataset which can be used to train successive shallow autoencoders. The fine-tuning proceeds exactly as with the RBMs.

Though the training procedures for RBMs as well as shallow autoencoders are similar, but as we show in the beginning of the next chapter, the results obtained using basic autoencoder or its variants are usually not as good as the results obtained by using RBMs. Chapter 3 attempts to bridge this gap between the performance of RBMs and autoencoders by incorporating certain modifications.

2.4 Resources

In this section, we explain the various resources used by us for conducting the experiments.

2.4.1 Theano

As the neural network algorithms are such that they can be parallelized very efficiently, the codes were run on a machine with a GPU card. The library used

for this was theano [Bergstra et al., 2010][1]. Theano is a python library, tightly integrated with numpy, which makes use of symbolic computing and symbolic differentiation. Theano dynamically generates Cuda code in order to run the program on GPU.

2.4.2 Machine Specifications

All the codes involving neural networks were run on a 64 bit machine having AMD Athlon 1.96 GHz dual core processor with Tesla C1060 33 MHz card having 240 cores. The RAM of the system was 2 GB and the memory available on GPU was 256 MB. Using GPU with the help of Theano provided average speed gains of 8x-10x. For example, training a deep autoencoder for MNIST data took approximately 36 hours on the same machine without using GPU and 3 hrs 49 mins on with GPU. In both cases, basic autoencoders were used for pre-training and the architecture used was -1000-500-250-30- .

2.5 Datasets

We consider 5 datasets for our experiments out of which one is the standard MNIST dataset of hand-written digits and 4 are synthetic datasets for which the intrinsic dimensionality is explicitly known though not used much. The datasets are described below:

[1]more information about theano is available at http://deeplearning.net/software/theano/

2.5.1 MNIST

This is one of the most popular datasets in image processing and hand-written digit classification tasks. It consists of 70,000 binary images of size 28×28 having 10 different classes each corresponding to a digit from 0-9. The complete data is split into 3 sets - training set having 50,000 images, validation set having 10,000 images and test set having 10,000 images. The training set is used for training the networks while the validation set is used to evaluate the performance of the trained network and to decide the stopping criterion. The performance is reported by applying the best validation model on the test set. Henceforth, we mention this split in the form of a tuple like (50,10,10). Sample image from the MNIST database are given in fig. 2.4

FIGURE 2.4: Sample images from MNIST Dataset

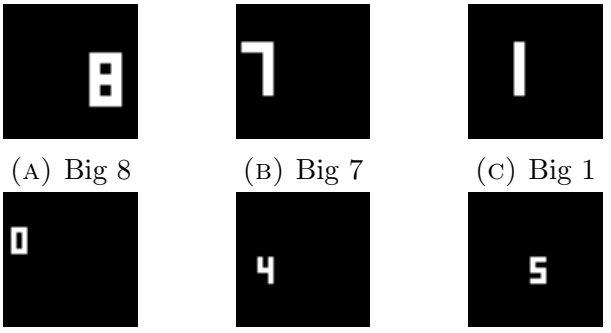
2.5.2 Square and room

This dataset consists of nearly 100,000 binary images of size 40×40. Some examples from the dataset are shown in fig. 2.5. All images contain a 6×6 size square and a room like hollow box with total width of 22 pixels and height of 25 pixels. The thickness of walls of the room is 5 pixels. Different images show the square and the box in random non-overlapping positions obtained by the translation of the room and square in horizontal and vertical directions. The training-validation-test split for the data was (80,10,10).

FIGURE 2.5: Square and room Dataset

2.5.3 Big and small digits

This dataset contains 50×50 binary images of digits from 0-9 in 2 different sizes referred as big and small. Each image contains only one digit either in small or big size. The digits are in fixed shapes as they would appear on a digital watch. For each small digit, there are 1596 images formed by translating the digit in horizontal and vertical directions. Similarly for each big digit, there are 1008 images. The training-validation-test split used for this dataset is (80,10,10). Sample images for this dataset are presented in fig. 2.6.



(A) Big 8 (B) Big 7 (C) Big 1

(D) Small 0 (E) Small 4 (F) Small 5

FIGURE 2.6: Big and Small Digits Dataset

2.5.4 3d Robot Arm

This dataset contains 10,000 binary images with size 140×140. The images have three robotic arms connected as shown in fig. 2.7. Different images in this dataset consist of the arms placed at different angles. The length of each arm is 40 pixels

and width is 5 pixels. The three angles deciding the positions of the arms represent the 3 degrees of freedom. The training-validation-test split used for this dataset is (80,10,10).

FIGURE 2.7: Robot Arm 3d Dataset

2.5.5 2d Robot Arm

The images in this dataset are similar to the ones in above mentioned 3d robot arm dataset except that instead of 3 robot arms, it has just 2 robot arms and hence only 2 degrees of freedom. The dataset contains 23,968 binary images of size 100×100 pixels. In this dataset, while the first arm is allowed to be in any position, the 2nd arm was restricted to have an angle between -105^0 and 105^0 with 0 being taken in the direction of first arm. This was done to avoid any significant overlap between the 2 arms. Sample images from this dataset are shown in fig. 2.8. The training-validation-test split for this dataset was also kept at (80,10,10).

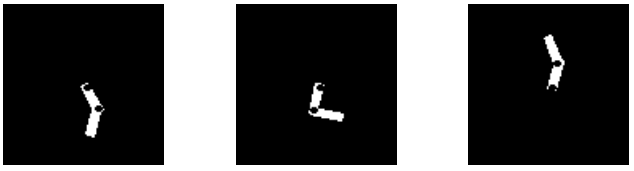


FIGURE 2.8: Robot Arm 2d Dataset

Chapter 3

Bridging the gap between RBMs and Shallow Autoencoders

Till now, we have described how stacked RBMs have been used for pre-training deep networks used for classification purposes as well as for deep autoencoders used in dimensionality reduction. For pre-training deep classifiers, the use of stacked shallow autoencoders has also been studied in considerable depth. However, the use of shallow autoencoders for pre-training deep autoencoders has not been explored in detail. In this chapter, we seek to bridge this gap.

3.1 Using Stacked Shallow Autoencoders to initialize Deep Autoencoders

We first try to evaluate the performance of standard shallow autoencoder models (described in section 2.1) commonly used for other purposes in the literature. Experiments were conducted on the MNIST dataset and square and room dataset

using RBMs and these shallow autoencoder models. The network architectures used for respective datasets are given below:

1. **MNIST Dataset:** -1000-500-250-30-
2. **Square and room Dataset:** -2000-400-100-24-4-

The minimum reverse cross-entropy achieved in each case is given in table 3.1.

Model \Dataset	MNIST	square and room
Basic AE	70.45	46.29
Sparse AE	61.43	39.99
Contractive AE	57.87	32.08
Denoising AE	57.40	31.76
RBM	56.61	24.31

TABLE 3.1: Reconstruction error with RBM and standard autoencoder models

From the values in table 3.1, it can be seen that though denoising autoencoder and contractive autoencoder come close to the performance achieved by RBM, there is still some difference. In terms of time taken for training, the basic autoencoder trains fastest and contractive autoencoder takes much more time than all other models. The total time taken for pre-training and fine-tuning different models is given in table 3.2.

Model \Dataset	MNIST	square and room
Basic AE	3 hrs 49 mins	4 hrs 25 mins
Sparse AE	4 hrs 20 mins	5 hrs 9 mins
Contractive AE	9 hrs 31 mins	14 hrs 59 mins
Denoising AE	4 hrs 50 mins	5 hrs 47 mins
RBM	5 hrs 05 mins	6 hrs 20 mins

TABLE 3.2: Training time for RBM and simple stacked autoencoder models on a 33 MHz tesla C1060 GPU card with 240 cores. For full specifications, kindly refer section 2.4.

Table 3.2 shows clearly that the time taken for training contractive autoencoders is greater than twice of that used for almost all other models. This is because of

the huge computation time incurred in calculating the jacobian matrix[1]. Given that the performance of contractive autoencoders for the task of dimensionality reduction is quite comparable with the denoising autoencoders and the computation time is considerably high, we have not used contractive autoencoders for larger datasets and other experiments described in rest of the thesis.

Among the remaining stacked autoencoder models, we see that sparse autoencoders with sparsity being enforced on every layer does not work well. However, as we show in the next sections, by enforcing sparsity on only alternate layers and putting some restrictions on network architecture, reconstruction performance comparable to RBMs can be achieved.

3.2 Using sparsity for dimensionality reduction

There is evidence from the work done in signal processing and several related fields ([Taubock and Hlawatsch, 2008], [Reid, 1982]) that sparsely represented data is relatively easier to compress. However, most of these works assume a lot of prior knowledge about the original space as well as higher dimensional space in which sparse representations of the data are being considered. Taking a relatively simplistic view of it, we extend this idea for our setting in which we assume very little or no prior knowledge about the vector spaces in which the data lies and hypothesize that even in the non-linear dimensionality reduction scenario encountered in the hidden layers of our network, dimensionality reduction may be

[1]For more details on the exact algorithm see section 2.1.3 or the original paper by [Rifai et al., 2011]

aided if the representation of data is sparse. In order to test the hypothesis, we conducted a few experiments the results of which indeed support this hypothesis.

We compared the results of 3 strategies for reducing the dimensionality of data.

1. **Strategy A:** Under this strategy, we used a basic autoencoder (as described in section 2.1.1) with one hidden layer having $N_h < N_v$ hidden units where N_v is the number of input dimensions.

2. **Strategy B:** Under this strategy, we first transform the data into a sparse representation of the same dimensions using a sparse autoencoder (section 2.1.4), and then feed the representations learnt by the hidden units of the sparse autoencoder to a similar autoencoder as used in strategy A.

3. **Strategy C:** Under this strategy, we transform the data to obtain projections in an N_v dimensional space similar to the one obtained in first step of strategy B except that we do not enforce sparsity. This step in strategy C is only to bring the strategies B and C on a similar footing. Similar to strategies, we now feed these N_v-dimensional representations to the basic autoencoder with N_h hidden units.

The reconstruction error by employing all the 3 strategies is presented in table 3.3. The data used for this experiment is obtained using hidden activations for 2d Robot Arm dataset at inter-mediate stages.

Strategy \Network Architecture	$N_v = 1000$ $N_h = 200$	$N_v = 5000$ $N_h = 1000$	$N_v = 100$ $N_h = 50$
Strategy A	241.72	714.91	43.63
Strategy B	198.23	603.51	40.01
Strategy C	235.97	687.23	43.98

TABLE 3.3: Comparison of the strategy employing sparsity against strategies using no regularization

3.2.1 Enforcing Sparsity: When and Where ?

Results illustrated in table 3.3 indicate that it may be beneficial from a dimensionality reduction perspective to obtain sparse representations. So, we have two objectives at hand. Most importantly, we wish to reduce the dimensionality of data. But in order to reduce the dimensionality, it would be better to first have the data in a sparse form. The obvious choice in such a scenario is to enforce sparsity at every successive layer. However, results from our earlier experiments (table 3.1) indicate that enforcing sparsity on every successive layer does not yield good results. In particular, during pre-training, it was observed that when the number of units in the hidden layer is less than the number of units in the input layer, enforcing sparsity causes suboptimal results. In order to solve this problem and still make maximal use of sparsity, we break this task into 2 layers. Given an N-dimensional input, we do not directly go to a lower dimensional representation. We first add one layer of sparse autoencoder having the same or slightly more number of hidden units as compared to the input layer and then use a layer of basic autoencoder having less number of hidden units. The same is represented pictorially in fig. 3.1. The intuition here is that while attempting to obtain sparse representations, it is helpful to project data in a slightly higher dimensional space.

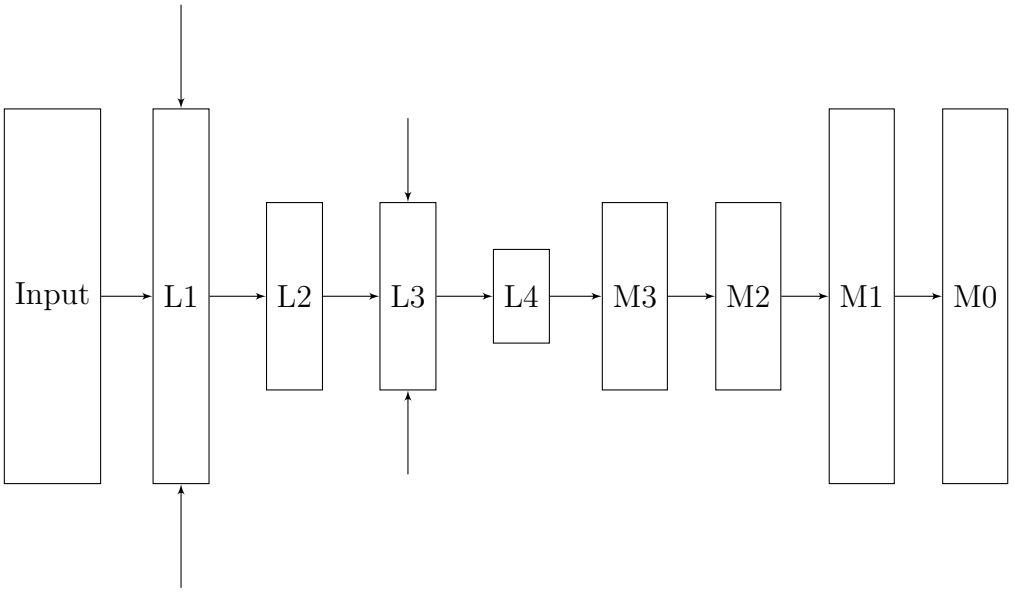


FIGURE 3.1: Implementing alternate sparsity. The extra arrows on L1 and L3 are to indicate that sparsity was enforced on these layers. Please note that size of Input and L1 have been kept equal. Also, the size of L2 and L3 has been kept equal. This is to show the constraint required to enforce alternate sparsity

3.2.2 Momentum and Weight Decay for autoencoders

Momentum and Weight Decay are two additional terms that can be used in training[2]. They are used to prevent unnecessary oscillations, over-fitting and to speed up the training. They have been widely used in training RBMs but there use in training shallow autoencoders has not been seen much.

Mathematically, the momentum term for updating a particular parameter is the previous update for the same term multiplied by a scalar between 0 and 1. It is used to prevent oscillations during training by reducing those components of the present update which are in opposite direction to the previous one and at the same time reinforcing the updates in the same direction thus increasing the speed

[2]A useful reference for understanding the importance of momentum and the intricacies of its implementation is [Sutskever et al., 2013]

of training. Weight decay term simply keeps reducing the magnitude of weights by a very small fraction. This mainly helps in avoiding over-fitting.

For adjusting the weights given to both these, we follow the strategies advised in [Hinton, 2012].

3.2.3 Putting it all together

In order to study the effect of alternate sparsity, momentum and weight decay, we tested it with all the 5 datasets described in section 2.5. Results for the same are presented in table 3.4. The network architecture used for each dataset is given below. The layers for which sparsity was enforced are shown in bold:

1. MNIST: **1000**-500-**500**-250-30-
2. Square and room: 400-**400**-100-**100**-24-**24**-8-4-
3. Big and small digits: 1000-**1000**-200-**200**-50-**50**-25-10
4. Robot Arm 2d: 2000-**2000**-400-**400**-100-**100**-25-**25**-8-2-
5. Robot Arm 3d: 5000-**5000**-1000-**1000**-200-**200**-40-**40**-10-3-

Dataset \Paradigm	DAE	DAE + momentum	SAE	SAE + momentum	RBM
MNIST	57.80	57.85	57.25	56.95	56.91
Square and room	21.76	20.99	21.58	14.56	15.89
Big and small digits	45.78	45.23	39.85	26.45	29.82
Robot Arm 2d	151.43	140.66	141.88	76.84	82.04
Robot Arm 3d	326.58	320.70	317.84	280.11	290.11

TABLE 3.4: Reconstruction error using the strategy of alternately sparse layers against other models

Table 3.4 shows that alternately sparse layers show improvement over denoising autoencoders. Stabilizing the training with momentum and weight decay further

improves the performance of alternately sparse model and makes it comparable with the RBMs.

3.3 Role of network architecture

One of the constraints for implementing alternate layer sparsity is that it places a restriction on the network architecture as every alternate layer has to be of the same or more size as compared to the previous one. This is a requirement for successfully implementing our suggested model but is not required by other paradigms such as denoising autoencoders or RBMs. It is important to verify if the performance of these models is influenced in any way by the the constraint on network architecture.

For this, we compare 3 different types of network architectures on 3 datasets as mentioned below:

1. **Type 1:** This is the network architecture compatible with alternate sparsity and is the same as used for experiments reported in table 3.4

2. **Type 2:** This is the set of network architectures formed by uniformly reducing the successive layer sizes. Details are as follows:

    - Square and room: 400-200-100-50-24-16-8-4-
    - Big and small digits: 1000-500-200-100-50-25-10-
    - Robot Arm 2d: 2000-1000-400-200-100-50-25-15-8-2-

3. **Type 3:** This is the set of network architectures formed by just eliminating the sparsity layer. Details are as follows:

    - Square and room: 400-100-24-8-4-
    - Big and small digits: 1000-200-50-25-10-
    - Robot Arm 2d: 2000-400-100-25-8-2-

With the above mentioned network architectures, results of experiments with denoising autoencoders are reported in table 3.5 and corresponding results for RBM are in table 3.6.

Dataset \Architecture Type	Type 1	Type 2	Type 3	SAE
Square and room	20.99	18.76	35.68	14.56
big and small digits	45.23	35.98	51.64	26.45
Robot Arm 2d	151.43	123.85	242.37	76.84

TABLE 3.5: Effect of Network architecture on the performance of Denoising Autoencoders

Dataset \Architecture Type	Type 1	Type 2	Type 3	SAE
Square and room	15.89	14.34	24.68	14.56
big and small digits	29.82	27.53	31.64	26.45
Robot Arm 2d	82.04	78.39	89.16	76.84

TABLE 3.6: Effect of Network architecture on the performance of RBMs

The results shown above reveal that though uniformly reducing the size of layers improves the performance of both denoising autoencoders as well as RBMs, it does not improve their performance beyond that achieved by alternately sparse layers except marginally in one case. Also, the results show that the change in network architecture affects denoising autoencoder more strongly as compared to

the RBMs. Reducing the number of layers is seen to affect the performance of both the models negatively.

3.4 Concluding Remarks

The experiments described in this chapter show that by putting some reasonable restrictions on the network architecture and by incorporating alternate layer sparsity, momentum and weight decay in the training procedure for stacked autoencoders, their performance can be made comparable with (and in some cases better than) that of stacked Restricted Boltzmann Machines for the task of finding good initial weights of deep autoencoders.

Also, it has been shown that denoising autoencoders and RBMs perform slightly better if the architecture restrictions necessary for implementing alternate layer sparsity are relaxed. However, this improvement does not yield results better than the ones achieved by using the alternate sparsity model. Thus, in totality, it seems beneficial to use the alternate sparsity model over other models considered in this work.

In the next chapter, we present a relatively more generic technique for improving the performance of all such models which use stacked modules for finding initial weights for training deep autoencoders.

Chapter 4

Intermediate Fine-tuning (IFT)

As mentioned in chapter 2, one of the popular reasons cited for the initial failures in training deep networks is that due to the random initializations and non-convex nature of the objective function, gradient based iterative optimization methods used for training neural networks work "well only if the initial weights are close to a good solution" [Hinton and Salakhutdinov, 2006]. Thus, a lot of work after 2006 has concentrated on finding good initializations for the networks through stacked RBMs or stacked autoencoders. For pre-training successive layers, these methods make use of the outputs of previous layers which have already been pre-trained. It is worth noting that though the previous layers have been pre-trained for a local objective (reconstruction of the output of layer immediately before it), however they have not been fine-tuned for the global objective and hence, it is possible that the output given by the previous layers is not very close to the output they are likely to produce during or after the global fine-tuning modifies their weights. Though this effect is relatively insignificant and can be easily neglected

for autoencoders having only 2-3 encoding layers, this effect becomes more relevant while training very deep autoencoders.

In order to avoid this, we propose that instead of applying a global fine-tuning only after pre-training all the layers, it should be applied at intermediate steps also after pre-training every 2-3 layers. We explain different steps of the procedure used by us in detail in section 4.1. In section 4.2, we present the results of several experiments conducted by us to understand the benefits of intermediate fine-tuning. More discussion on the these results and their implications is presented in section 4.3.

4.1 Methodology

We explain the method and different steps involved in the it by taking the case of 2d Robot Arm dataset. We refer to the objective of optimal reconstruction of the input as the global objective. The network architecture is -2000-500-250-120-60-30-15-8-2-

Phase 1: The first three layers are pre-trained by training one layer at a time using the output of the previous hidden layer as input to the next layer. After pre-training of the first three layers is complete, we make an inverted copy of the layers in reverse order to form a full autoencoder having six layers i.e. the network is now a deep autoencoder with the architecture -2000-1000-500-. The weights in decoding layer are initialized by transposing the weight matrix learnt in the encoding layers. This deep autoencoder having six layers is now trained with

the objective of optimally reconstructing the input. Lets denote this 6 layer deep autoencoder as outer autoencoder.

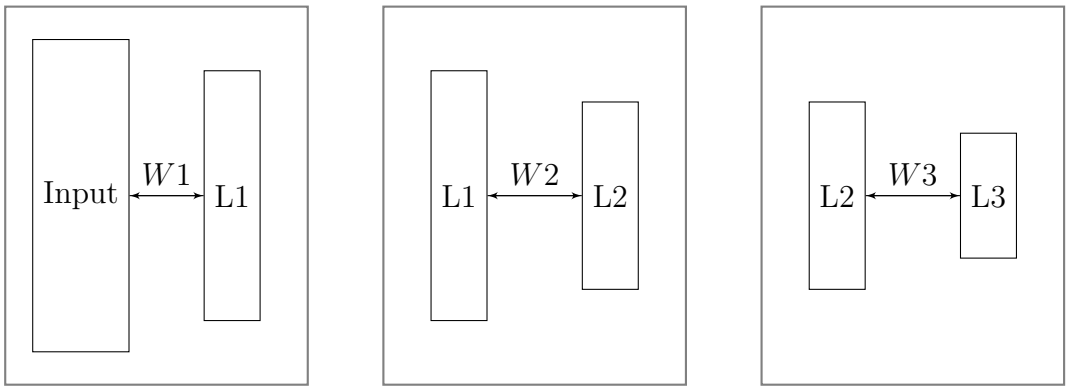


FIGURE 4.1: Pre-training of the autoencoder in Phase 1

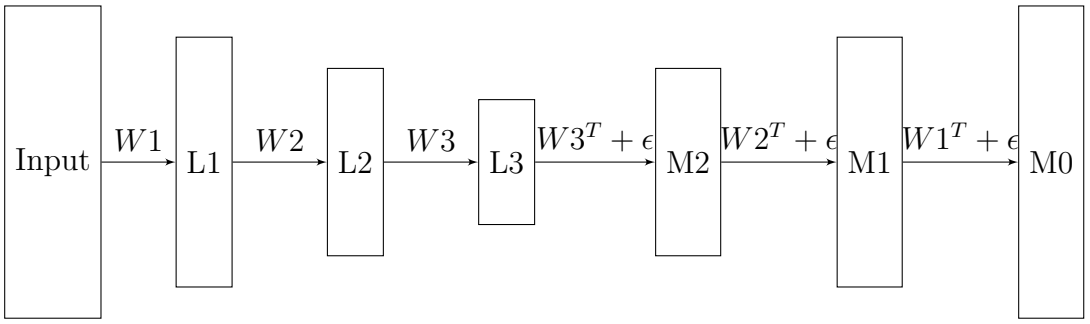


FIGURE 4.2: Outer Autoencoder in Phase 1

Phase 2: After the fine-tuning for outer autoencoder is complete, the encoding of the dataset in 500 units per input sample is obtained by a forward pass of the data through the network only till the third layer having 500 units. For some time, let's assume that the task at hand is only to find the weights which can reconstruct this 500 unit input optimally with the successive hidden layer sizes being fixed at 250 and 120 respectively. We call this our local objective. We proceed in exactly the same manner as we did for the global objective and first pre-train the two layers in a greedy layer-wise manner, then invert them and fine-tune the whole network having four layers for the local objective. We term this smaller four layer deep autoencoder having the layer sizes as -250-120- as inner autoencoder.

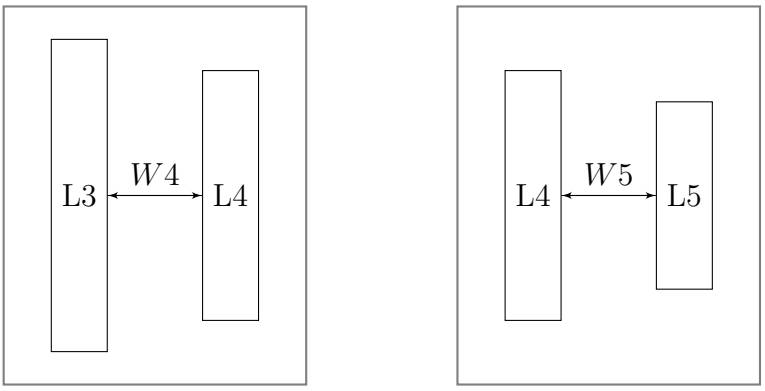


FIGURE 4.3: Pretraining of the autoencoder in Phase 2

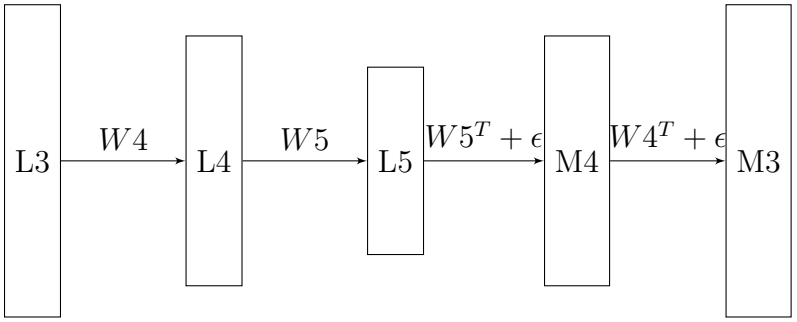


FIGURE 4.4: Inner Autoencoder in Phase 2

Phase 3: In the next step, we initialize a 10 layer deep autoencoder having the layer sizes as -2000-1000-500-250-120-. The weights in these layers are initialized at the values taken from outer and inner autoencoders which have been trained in phase 1 and phase 2. This joining of the two networks can be understood as splitting the outer autoencoder into half and then inserting the inner autoencoder at the middle from where the outer autoencoder has been split. The layers in the final deep autoencoder belonging to the inner and outer autoencoders have been shown by different bounding boxes in fig 4.5. Now, this complete 10 layer autoencoder is trained on the global objective of optimally reconstructing the original input.

For the next steps, this 10 layer deep autoencoder takes the role of outer autoencoder using which a 120 dimensional dataset is formed and an inner autoencoder is

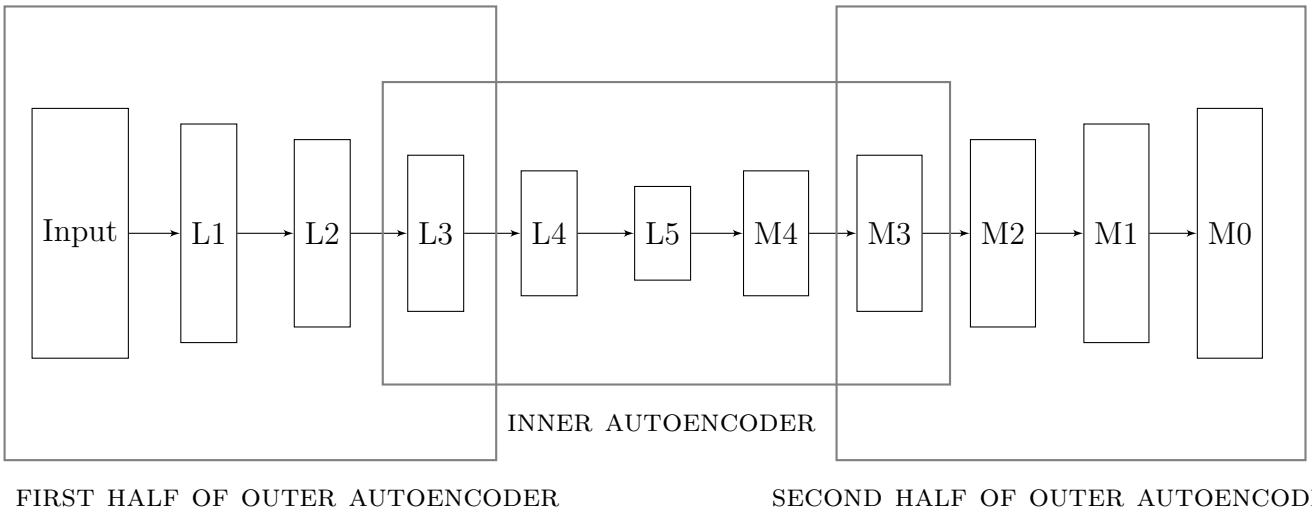
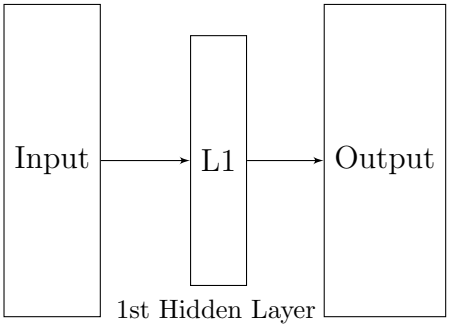


FIGURE 4.5: Deep Autoencoder formed by joining Outer Autoencoder and Inner Autoencoder in Phase 3

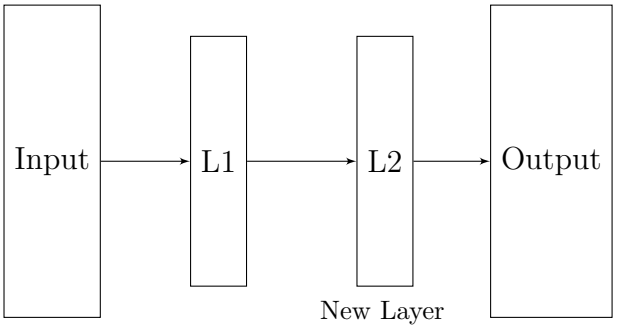
again trained for the objective of optimally reconstructing this 120 dimension input. This process continues till the desired network architecture has been achieved.

At this point, we wish to explain a seemingly similar approach used in [Jain and Seung, 2008] who also use a global reconstruction criterion while incrementing the layers but as we shall explain, their network architecture and training approach is still very different owing to the different requirements of their task. The network used by them has 5-6 convolutional layers, all of the same size, besides a fully connected final decoding layer which gives the output. The network is designed to recover the original image from a partially corrupted image. They start with just one hidden layer and train the weights to optimally reconstruct the original input in a similar fashion as denoising autoencoder (2.1.2) though the architecture of their encoding layer and the final objective is slightly different. After training with one hidden layer is complete, another hidden layer is added between the first hidden layer and the decoding layer(see fig. 4.6). Also, the weights in the decoding layer learnt earlier are discarded. Similarly, the process continues by adding one

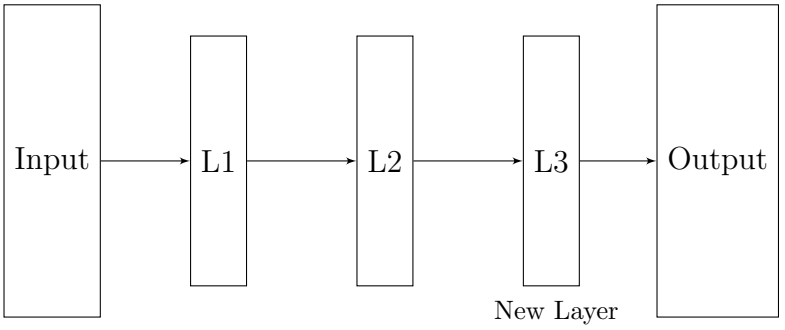
layer at a time till the final required architecture is achieved. In this process, at the addition of every new layer, the two new sets of weights which are being tuned are always adjacent to layer from which backpropagation of error starts and thus, the problem of vanishing gradients does not surface.



STEP 1: TRAINING WITH 1 HIDDEN LAYER



STEP 2: TRAINING AFTER ADDING 2ND HIDDEN LAYER



STEP 3: TRAINING AFTER ADDING 3RD HIDDEN LAYER

FIGURE 4.6: Training of the autoencoder for image denoising with procedure used in [Jain and Seung, 2008]

As opposed to this inherent proximity of the new weights with output of the image denoising network, our method introduces two hidden layers at a time and both of these layers come up in the middle of the network which in our case can be very far from the final layer (network output) from which the error has to back-propagate. So, we first pre-train the newly introduced layers on the local criterion of reconstructing the output of first half of the network learnt till now and then fine-tune the whole network for the global reconstruction task.

In the next section, we present some experiments, the results of which show that intermediate fine-tuning indeed helps in improving reconstruction performance of the networks.

4.2 Experiments and Results

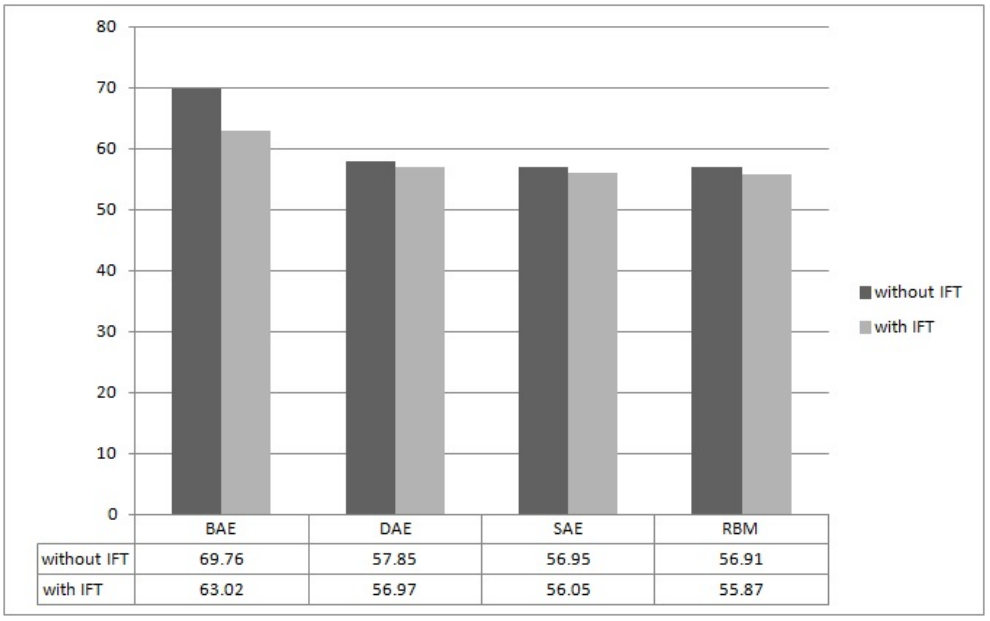
We conducted experiments on all 5 datasets described in section 2.5 and report the minimum reverse cross-entropy achieved in each case with and without using intermediate fine-tuning.

The different pre-training models used and the notation used for them are given below:

1. BAE: Basic Autoencoders
2. DAE: Denoising Autoencoders
3. SAE: Autoencoders with sparsity enforced on only alternate layers in accordance with the strategy advocated in chapter 3
4. RBM: Restricted Boltzmann Machines

The architecture used for different datasets is given along with the results for each one of them. In all the below mentioned deep autoencoders, activation function for all layers is sigmoid except for the middle layer which has linear activations. For example, for the case of MNIST, the layer with size 30 has linear activation.

MNIST: The architecture used was -1000-500-500-250-30- . The results are displayed in fig. 4.7



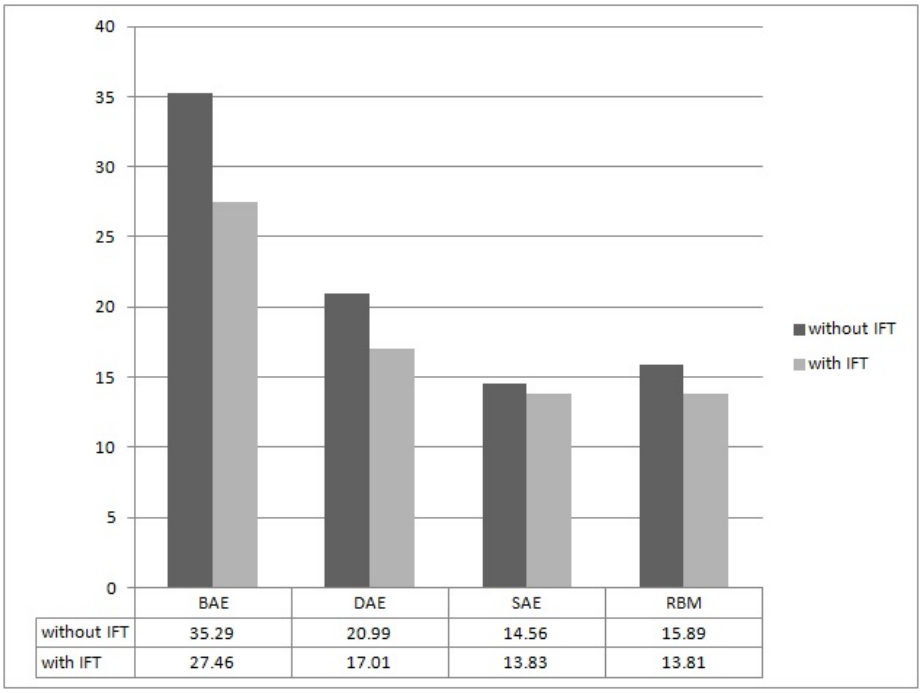
	BAE	DAE	SAE	RBM
without IFT	69.76	57.85	56.95	56.91
with IFT	63.02	56.97	56.05	55.87

FIGURE 4.7: Reconstruction performance with and without IFT for MNIST dataset

Square and room: The architecture used was -400-400-100-100-24-24-4- . The results are displayed in fig. 4.8

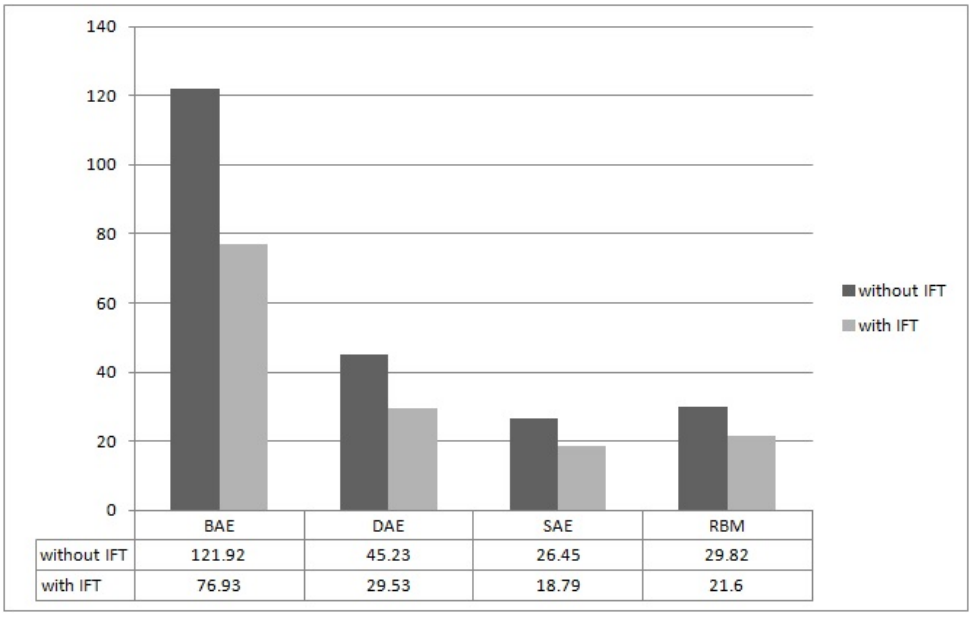
Big and small digits: The architecture used was -1000-1000-200-200-50-50-25-10- . The results are displayed in fig. 4.9

2d robot arm: The architecture used was -2000-2000-400-400-100-100-25-25-8-2- . The results are displayed in fig. 4.10



	BAE	DAE	SAE	RBM
without IFT	35.29	20.99	14.56	15.89
with IFT	27.46	17.01	13.83	13.81

FIGURE 4.8: Reconstruction performance with and without IFT for square and room dataset

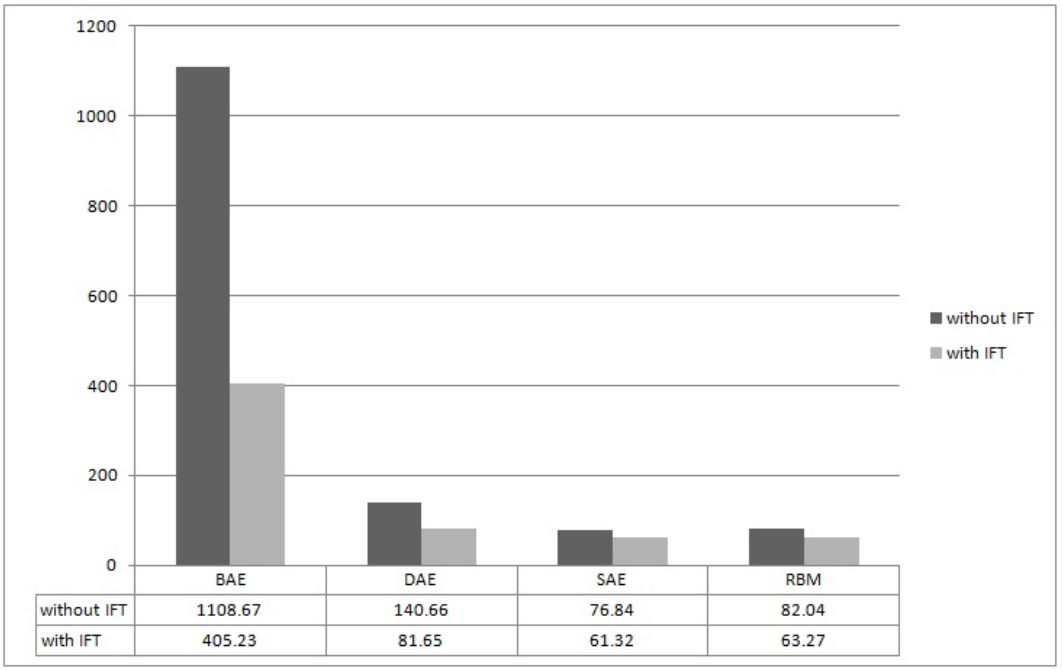


	BAE	DAE	SAE	RBM
without IFT	121.92	45.23	26.45	29.82
with IFT	76.93	29.53	18.79	21.6

FIGURE 4.9: Reconstruction performance with and without IFT for big and small digits dataset

3d robot arm: The architecture used was -5000-5000-1000-1000-200-200-40-40-10-3- . The results are displayed in fig. 4.11

The above mentioned results clearly show that intermediate fine-tuning improves



	BAE	DAE	SAE	RBM
without IFT	1108.67	140.66	76.84	82.04
with IFT	405.23	81.65	61.32	63.27

FIGURE 4.10: Reconstruction performance with and without IFT for 2d robot arm dataset

the reconstruction performance of all the models under consideration. Through this, we achieve better results as compared to the results achieved using RBMs without intermediate fine-tuning which is the method described in [Hinton and Salakhutdinov, 2006]. On MNIST dataset, the network using RBMs for pre-training perform better while for three out of the remaining four datasets, networks with alternately sparse layers perform better. On square and room dataset, though the performance with stacked RBMs is better than the one with alternately sparse layers but the difference is negligible.

It can be seen that the benefits derived from intermediate fine-tuning become more significant in autoencoders having more layers as compared to the ones having lesser layers. In order to study this effect in detail, we conducted two sets of experiments on the Robot arm datasets with 2 degrees of freedom. In one set, we pre-train the model by incrementally stacking layers without any intermediate

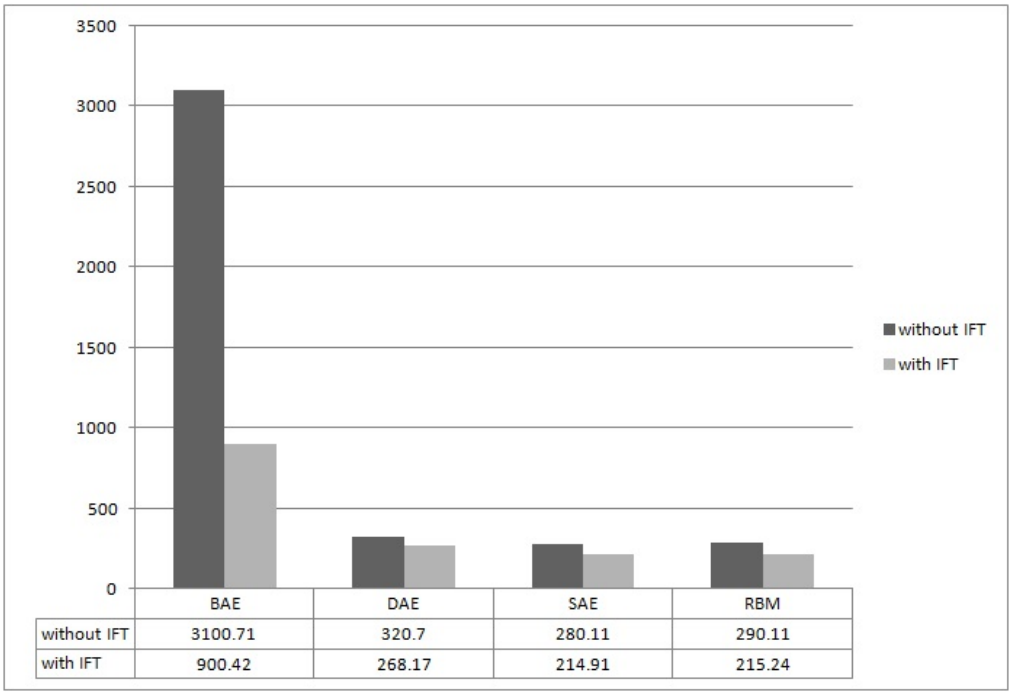


FIGURE 4.11: Reconstruction performance with and without IFT for 3d robot arm dataset

fine-tuning till a particular layer (referred as checkpoints below) and then fine-tune the network by inverting the layers as per the procedure used in [Hinton and Salakhutdinov, 2006]. In the second set, we perform the same steps as above except that we also fine-tune the network at all intermediate checkpoints prior to the checkpoint under consideration. In fig. 4.12, we report the difference in minimum reconstruction error achieved for both sets of experiments. Individual reconstruction errors are reported in table 4.1. More information relevant to the experiments is presented below:

- **Strategy used:** Alternately Sparse Layers as suggested in chapter 3
- **Network Architecture for 2d dataset:** 2000-2000-400-400-100-100-25-25-8-2 and reverse
- **Checkpoints for 2d dataset:** layer numbers 3, 5, 7, 9, 10

Checkpoints \Sets	Set 1	Set 2
layer 3 (400 units)	20.95	20.95
layer 5 (100 units)	14.56	12.71
layer 7 (25 units)	21.01	17.36
layer 9 (8 units)	29.73	24.68
layer 10 (2 units)	76.84	61.32

TABLE 4.1: Reconstruction error at different stages of training the autoencoder for 2d Robot Arm set with and without using IFT

It can be seen in table 4.1 that there is a notable increase in the reconstruction error as we go deeper but in this case the increase is not due to the increase in depth. It is attributed mainly to the reduction in the number of units being used to encode the data.

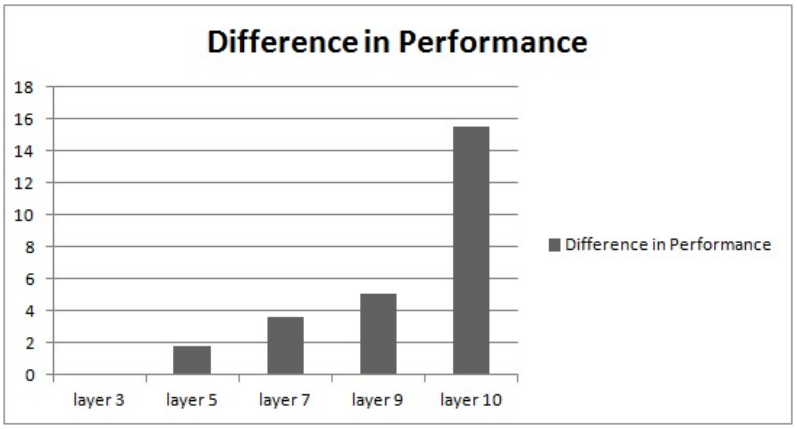


FIGURE 4.12: Reconstruction error at different stages of training the autoencoder for 2d Robot Arm set with and without using IFT

As the above results show, the difference in the results obtained with and without using inter-mediate fine-tuning becomes more and more significant as the depth increases.

The above mentioned results are all in terms of reverse cross-entropy. In order to see if this network is able to serve the purpose for out-of-sample reconstruction and other similar tasks, we reconstructed some images from the test set of 2d robot arm. The reconstructed images using alternately sparse layers with inter-mediate

fine-tuning are shown in fig. 4.13. It can be seen clearly that the reconstructed images are very similar to the original images and it is almost impossible to tell the difference by just looking at them.

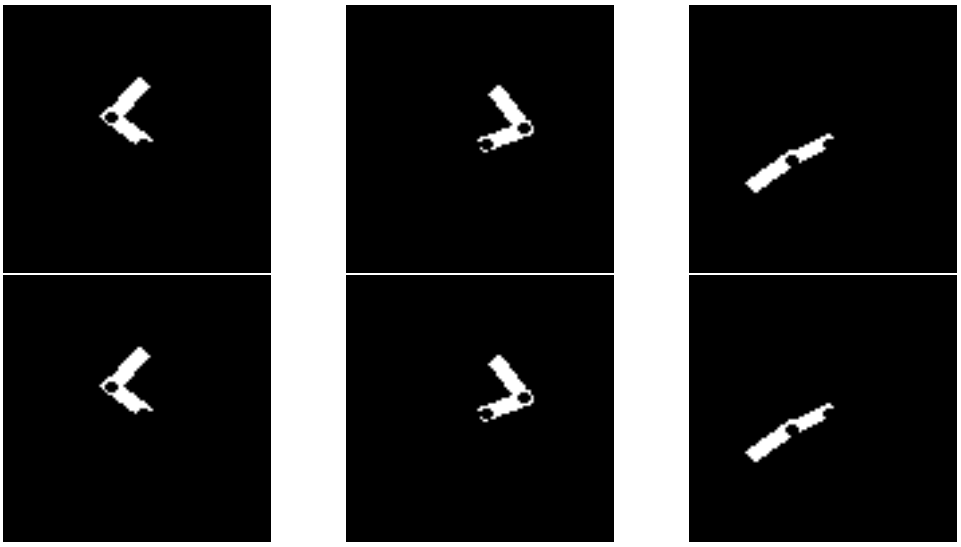


FIGURE 4.13: Reconstructed test images: Top row shows original images and bottom row shows reconstructions obtained from the trained network

More discussion on various aspects of intermediate fine-tuning, specially those related to training time, is presented in the next section.

4.3 Discussion

Taking a look at the results presented in the above section makes it clear that the performance of the models in terms of re-generation of the original input from the reduced space is significantly improved by using inter-mediate fine-tuning.

In terms of the time taken for pre-training and fine-tuning the models, the time taken in inter-mediate fine-tuning steps is compensated as the number of iterations required to fine-tune the model in final stage is much less as compared to the

autoencoders fine-tuned only at the end. For example, for training the autoencoder on 2d robot arm dataset using alternately sparse layers, it takes nearly 3 hours to pre-train without inter-mediate fine-tuning and approximately 9 hours to fine-tune. With intermediate fine-tuning, it takes a little more than 8.5 hours to complete the pre-training but less than 2 hours to fine-tune the final model. The exact times are reported in table 4.2. Machine specifications used for the experiment are given section 2.4.

Method \Time Taken	without IFT	with IFT
BAE	7 hrs 37 mins	9 hrs 21 min
DAE	12 hrs 51 mins	12 hrs 32 mins
SAE	11 hrs 54 mins	11 hrs 23 mins
RBM	13 hrs 47 mins	13 hrs 27 mins

TABLE 4.2: Time Taken in training with and without IFT on a machine using a 33 MHz Tesla C1060 GPU card with 240 cores

For the experiments reported above, we perform inter-mediate fine-tuning after every 2 layers. Another possibility is to perform this inter-mediate fine-tune after every layer. We have not done this because preliminary experiments with such a strategy showed negligible or no improvements in terms of performance whereas training time increased significantly. One more possibility is to fine-tune after every three layers instead of two layers. Though this reduces the training time, but we did not pursue this further as the initial experiments revealed that the reconstruction performance was being compromised.

There are several other possibilities in terms of network architectures and other parameters which have not been pursued in this work and may be taken up as future work based on this thesis. Several such possibilities are briefed about in the next chapter which also presents some concluding remarks on this work.

Chapter 5

Conclusion and Future Work

In this work, we presented two new techniques for improving dimensionality reduction by deep autoencoders. We demonstrated that for pre-training deep autoencoders, the performance of stacked Restricted Boltzmann Machines was better than most deterministic models considered in this work. However, with alternate sparsity, as explained in chapter 3, pre-training with stacked shallow autoencoders provided better reconstruction performance as compared to the ones using stacked RBMs.

Inter-mediate fine-tuning was introduced as a generic technique for improving the performance of both deterministic models and RBMs. Given any type of stackable module for pre-training and irrespective of the architecture used, inter-mediate fine-tuning improved the results. The improvement in performance was more significant for autoencoders having more layers as compared to the ones having lesser layers. For less than three layers, the benefits were negligible.

Despite the above mentioned advances made in using sparsity for deep autoencoders for dimensionality reduction, a lot remains to be explored. An in-depth analysis and mathematical formulation of how sparsity is aiding dimensionality reduction can be an interesting theme. Also, it needs to be seen if it is possible to remove the architectural constraints required for enforcing alternate layer sparsity and still obtain similar benefits.

Different regularization strategies which can benefit from such an alternate formulation can be explored. There have been some attempts to combine different regularization strategies for autoencoders but the results reported are not very encouraging. One such example is the work of [Chen et al., 2013] where an attempt has been made to combine denoising criterion with regularization as used in contractive autoencoders. However, most such works use both strategies on all layers. It may be interesting to find out which strategies can complement each other the way sparsity and bottleneck constraints have done in our work.

As deep autoencoders are able to find an explicit mapping between the original higher dimensional space and the reduced space, applications requiring out-of-sample extensions and out-of-sample reconstructions can benefit a lot from the improved performance. This can be particularly helpful in robot motion planning. One such example is with the Robot Arm datasets where several configurations may be unachievable due to obstacles.

We have explored some variants of shallow autoencoders in this work. Similarly,

many variants of RBMs have also been proposed in literature. It may be worthwhile to explore if alternate layer strategies can be beneficial for some RBM variants too. The applicability of alternate sparsity for deep classifier networks can also be explored in future works based on this thesis.

Bibliography

Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58.

Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127.

Bengio, Y. and Delalleau, O. (2009). Justifying and generalizing contrastive divergence. *Neural Computation*, 21(6):1601–1621.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153.

Bengio, Y., Paiement, J.-F., Vincent, P., Delalleau, O., Le Roux, N., and Ouimet, M. (2004). Out-of-sample extensions for lle, isomap, mds, eigenmaps, and spectral clustering. *Advances in neural information processing systems*, 16:177–184.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Chen, F.-q., Wu, Y., Zhao, G.-d., Zhang, J.-m., Zhu, M., and Bai, J. (2013). Contractive De-noising Auto-encoder. *ArXiv e-prints*.

Dhingra, B. (2013). *Local Quadrature Reconstruction Using Smooth Manifolds*. PhD thesis, Indian Institute of Technology, Kanpur, Kanpur, India.

Fischer, A. and Igel, C. (2012). An introduction to restricted boltzmann machines. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 14–36. Springer.

Hinton, G. E. (2012). A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade*, pages 599–619. Springer.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

Hu, H. and Zahorian, S. A. (2010). Dimensionality reduction methods for hmm phonetic recognition. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 4854–4857. IEEE.

Jain, V. and Seung, H. S. (2008). Natural image denoising with convolutional networks. In *Advances in Neural Information Processing Systems (NIPS 2008)*, volume 21, pages 769–776.

Japkowicz, N., Myers, C., Gluck, M., et al. (1995). A novelty detection approach to classification. In *IJCAI*, pages 518–523.

Kramer, M. A. (1991). Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243.

LeCun, Y. and Bengio, Y. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361.

Ng, A. (2011). Sparse autoencoder. *CS294A Lecture notes*, page 72.

Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2007). Efficient learning of sparse representations with an energy-based model. In *Proceedings of NIPS*.

Reid, J. K. (1982). A sparsity-exploiting variant of the bartels—golub decomposition for linear programming bases. *Mathematical Programming*, 24(1):55–69.

Rifai, S., Vincent, P., Muller, X., Glorot, X., and Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 833–840.

Roweis, S. T. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326.

Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning internal representations by error propagation. de rumelhart and jl mcclelland (eds.), parallel distributed processing. *Foundations, MIT Press. Cambridge, MA*.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147.

Taubock, G. and Hlawatsch, F. (2008). A compressed sensing technique for ofdm channel estimation in mobile environments: Exploiting channel sparsity for reducing pilots. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 2885–2888. IEEE.

Tenenbaum, J. B., Silva, V. d., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323.

Thompson, B., Marks, R., Choi, J., El-Sharkawi, M., Huang, M.-Y., and Bunje, C. (2002). Implicit learning in autoencoder novelty assessment. In *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, volume 3, pages 2878–2883.

Thompson, B., Marks, R., and El-Sharkawi, M. (2003). On the contractive nature of autoencoders: application to missing sensor restoration. In *Neural Networks,*

2003. Proceedings of the International Joint Conference on, volume 4, pages 3011–3016 vol.4.

van der Maaten, L. J., Postma, E. O., and van den Herik, H. J. (2009). Dimensionality reduction: A comparative review. *Journal of Machine Learning Research*, 10(1-41):66–71.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 9999:3371–3408.

Zemel, R. S. and Hinton, G. E. (1995). Learning population codes by minimizing description length. *Neural computation*, 7(3):549–564.