# Reducing Dimensionality of Data Using Neural Networks

Ayushman Singh Sisodiya

Indian Institute of Technology, Kanpur

**Abstract.** In this report I have described ways to reduce the dimensionality of data using neural networks and also have how to overcome the problems in training such a network. The main result of my project revolves around the idea that there is a huge gap between the two pre-training methods I have used.

**Key words:** Autoencoders, Pre-training, RBM's, Stacked Autoencoders, Cross-entropy

## 1 Introduction and Motivation[1]

As we all know data dimensionality reduction is a very well known and necessary problem to solve. Many data can be represented in much lower dimension than it is already in. As data is increasing at a very rapid rate storing and managing is becoming more and more difficult. Other than this reducing dimensionality of data has found its ways in many research related areas as a pre-processing step or maximize discrimination. So many research has already been gone to achieve this task effectively and efficiently.

Some of the most popular and common ways to do so are principal component analysis (PCA), independent component analysis (ICA), locally linear embeddings, Isomap etc. have been proposed. In most cases we need the orignal data back from the reduced data. Many of the efficient algorithms have been devised that reduce the dimensioality of data very effectively but are unable to recover the original data. But the method we are going to describe is known to be effectively reconstruct the original data.

### 1.1 Autoencoders

Auto encoders or auto-associative neural networks produce output which is same as or say similar to its input with some constraints like bottle neck and tied weights. The bottleneck constraint is in which one or more than one hidden layer of the network has less nodes than the input layer. This constraint allows the network to learn a reduced version of the input, which is what we want to do. It has been shown that with linear activation of the nodes and with

[1] Disclaimer : This section is heavily inspired by Ankit Bhutani's thesis[4]

mean-squared error as the loss function for the network, it learns to extract the principal components of the data.

The benefits of the deep neural networks over the shallower ones are very significant but the implementation of deep neural networks is very costly in terms of computation and memory usage until 2006. As deep neural networks were very prone to problems like vanishing gradient descent and the stuck of optimization in a local minima, So not much of progress was done before 2006. In 2006 [6] Hinton introduced a pre-training method that initializes the weights close to a good solution. It was proved that any gradient based method used to train the network will only works well only if the initial weights are close to a good solution[7]. Hinton described that Restricted Boltzmann Machines introduced by smolensky [9] can be trained in a greedy manner to find good set of weights to initialize the neural networks. A fine-tuning method will follow which gave results which were very astonishing at that time.

Another way to pre-train the network to good initial weights is using shallow autoendcoders introduced by Bengio[3]. But the results obtained from this method were not as good as were obtained by using RBM [1]. Although initializing weights of deep autoencoder using shallow autoencoder has not been explored in detailed as mentioned in [4].

## 2 Datasets

I have used two datasets for this project. Both of them are described below.

### 2.1 MNIST

This is one of the most popular datasets. It has $70,000$ binary images of 10 different classes. The size of each image is 28 x 28. The whole dataset of $70,000$ images is split into 3 sets. The training set has $50,000$ images. The validation set has $10,000$ images and the test data has $10,000$ images. Some images from the dataset are shown below. These images are taken from here

**Fig. 1.** MNIST dataset sample images

### 2.2 2D-RobotArm

The dataset contains $23,968$ binary images of 2 robot arms as shown in the figure below. The image is 100 x 100. The length of the arm is 40 pixels and

width is 5 pixels. The first arm is allowed to be in any position whereas the 2nd robot arm is restricted to angles between $-105°$and $105°$. Sample images from the dataset are shown below. These images are taken from [4]

**Fig. 2.** Sample images from 2D-robot-arm dataset

## 3 Some Basic Knowledge[2]

As I have already given brief description of the models we are using, but a detailed explanation is required to really understant what is happening and why. I will begin by explaning the basic structure of shallow autoencoder.

### 3.1 Shallow Autoencoder

As I have already mentioned, a neural network whose output is same or similar to that of its input is called autoencoder. A shallow autoencoder is that in which there is only one hidden layer or say total 3 layers, one input,one hidden and one output. Any autoencoder with more than three layers is called Deep Autoencoder. The activation function I have used is sigmoid function which is

$$\frac{1}{1+exp^{-t}} \tag{1}$$

and the cosr function used is mean-squared error function. Another well known and good cost function is reverse cross-entropy which is defied as follows

$$L(X,Z;\theta) = -\sum_{i=1}^{n}(x_i log z_i + (1-x_i)(log(1-z_i))) \tag{2}$$

This function works very well when the values are between 0 and 1, which is it in our case. But we have used the mean squared error function. Baldi and Hornik [2] showed that using linear activation function and at global minima the network has extracted the first $n$ principle components where $n$ is the size of the hidden layer with lowest number of nodes. But it is not the case with other activation functions. For other activation functions the network is somehow forced to learn

[2] Disclaimer : This section is heavily inspired by Ankit Bhutani's thesis[4]

a representation of input in low dimensional space.

I have already explained the bottle-neck constraint but didn't really talked about the tied weight constraint. In tied weight constraint we force the weights of the decoder to be the transpose of the tied weights of the encoder. The encoder and decoder part is shown in the figure below. This image is taken from here

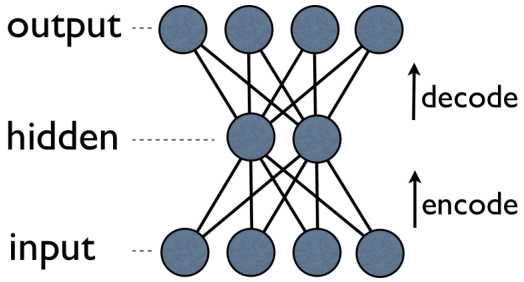


**Fig. 3.** Autoencoder

### 3.2 Denoising Autoencoder

In denoising autoencoder we basically add some noise to the input of the autoencoder but the output is same as the orignal input. The main reason behind this heuristic is that we are forcing out network to learn the main underlying structure in our input. This method was used by Vincent [1] to find good initial weights.

### 3.3 Restricted Boltzmann Machine(RBM)

Restricted Boltzmann machine are basically a variant of Boltzmann Machine with the restriction that the nodes in the same layer cannot have a connection i.e. it forms a bipartite graph. RBM's are said to learn probability distribution over the set of inputs. The energy of the RBM is calculated by the formulae(taken from wikipedia)

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{i,j} h_j \tag{3}$$

where $w_{i,j}$ is the weight matrix, $h_j$ is the hidden unit, $v_i$ is the visible unit,$a_i$ is the visible unit offset and $b_j$ is the hidden layer offset. and the probability distribution is given by

$$P(v,h) = \frac{1}{Z} e^{-E(v,h)} \tag{4}$$

The structure of RBM is shown Fig.4 This image is taken from here The most commonly used process used for training RBM is contrastive divergence algorithm [5]

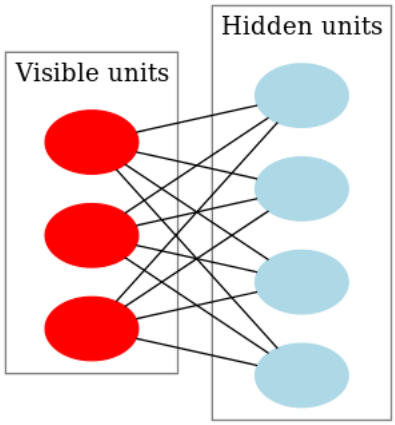


**Fig. 4.** RBM Strucuture

## 4 Methodology

Here we will describe how to pre-train suing the models described above.

### 4.1 Pre-training using RBM

Let us suppose our architecture of neural network is $n_1 - n_2 - n_3 - n_2 - n_1$. Here we will use the methodology described in Hinton 2006 [7]. So we well start by making a RBM with $n_1$ visible units and $n_2$ hidden units on all the training data using the method described in [5]. Now we have cconverted all out input in binary vector of size $n_2$. Now we will take this as the input to another RBM of with $n_2$ as the visible units and $n_3$ as the hidden units and will do the same as mentioned above. Now we have got $W_1$ and $W_2$. Due to tied weights constraints $W_3 = W_2'$ and $W_4 = W_1'$. After doing this we do what is called the fine-tuning step. In that we do simple backpropogation to fine tune the weights. The image clearly describes the method. After unfolding our RBM the network will look

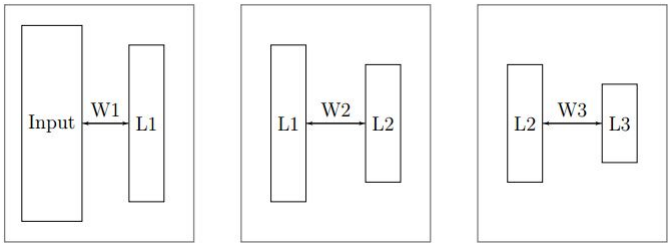


**Fig. 5.** RBM Pre-train

like this The above figures are taken from [4]

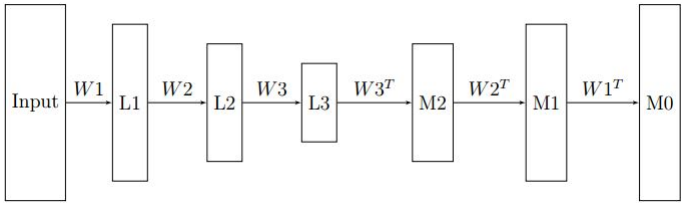


**Fig. 6.** Autoencoder

### 4.2 Pre-training using shallow autoencoders

These are also used in a greedy layer wise fashion as described above in RBM's. Lets me explain it by using an example. Suppose the architecture is of the form $n_1 - n_2 - n_3 - n_2 - n_1$. Than first we take $n_1$ and $n_2$. Than make a shallow autoencoder as $n_1 - n_2 - n_1$. Train this on the training set. Than we take $n_2$ and $n_3$ and make a shallow autoencoder of the form $n_2 - n_3 - n_2$ and train it on the activations abtained from the first network and unfold the same way as dont in RBM. For denoising one we just add some noise to the input and do the same as described above. Followed by fine-tuning using backpropogation. The figure below describes the method and is taken from here.

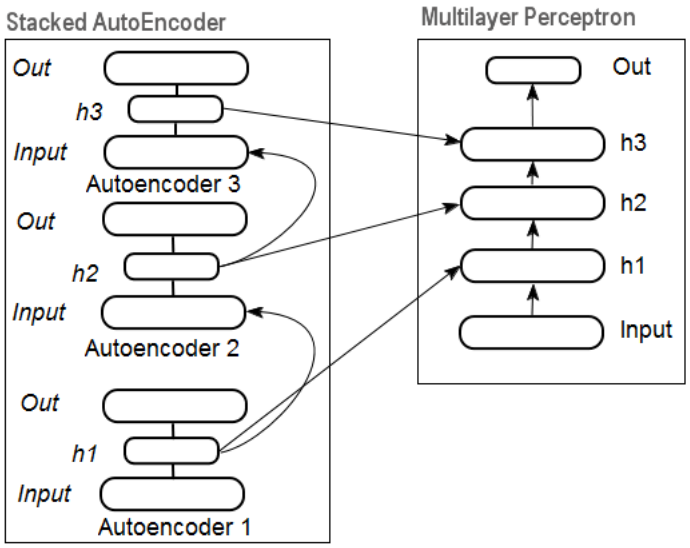


**Fig. 7.** Stacked Autoencoder

## 5 Results

I have used the matlab library by R.B. Palm [8]

### 5.1 MNIST

The architecture we used for this dataset are -
a –> $784 - 500 - 250 - 30 - 250 - 500 - 784$
b –> $784 - 1000 - 500 - 30 - 500 - 1000 - 784$
and we have used 3 methods to pre-train them. They are -
1 - using RBM
2 - using Stacked way
3 - using Denoising Stacked

From now on we will refer to the architecture and its method from the numbers above. So if the architecture is $784 - 500 - 250 - 30 - 250 - 500 - 784$ and the pre-training method is RBM, then we will refer to it as a-1. This is just for our convenience. First I will show the reconstruction by all 6 ways and than will try to justify my results.

**Fig. 8.** Orignal Image

This image is constructed by me from the dataset available here
Now the reconstruction by all 6 methods are below. Note that I have trained and pre-trained all 6 methods for 10 epochs

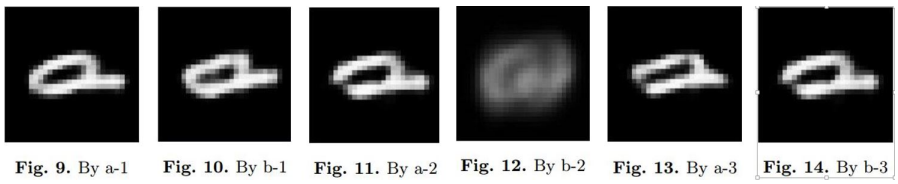

**Fig. 9.** By a-1 **Fig. 10.** By b-1 **Fig. 11.** By a-2 **Fig. 12.** By b-2 **Fig. 13.** By a-3 **Fig. 14.** By b-3

**Fig. 9.** Reconstruction of the image by all 6 models

You can see the images formed are pretty good except for the case with b-2. I have an explanation for this which I will give later.

First we will try to visualize the weight of the pre-trained network for all 6 models. The visualization of the weights for different models for different epochs are shown below

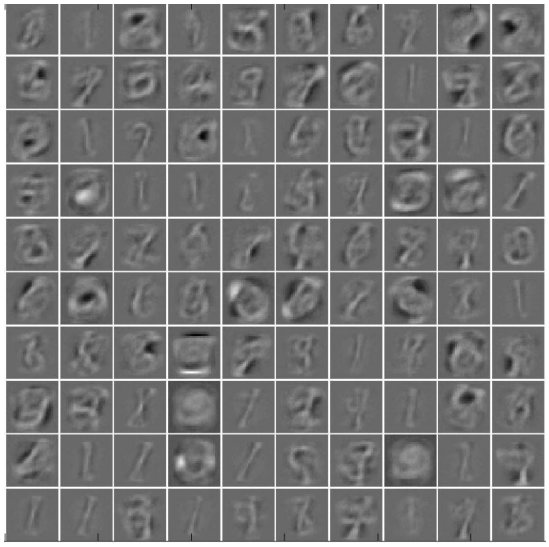

**Fig. 10.** RBM with architecture 784-100 and epoch-1

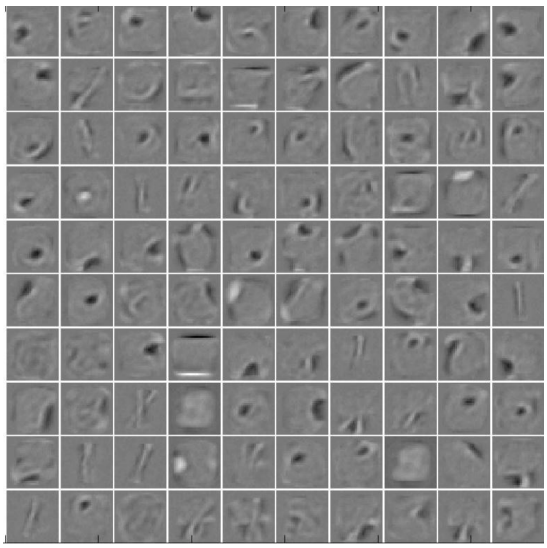

**Fig. 11.** RBM with architecture 784-100 and epoch-10

You can see that the features learned after 10 epoch are more significant than when learned after 1 epoch. You can see some strokes of the shapes of the figure. This was for pre-training with RBM.

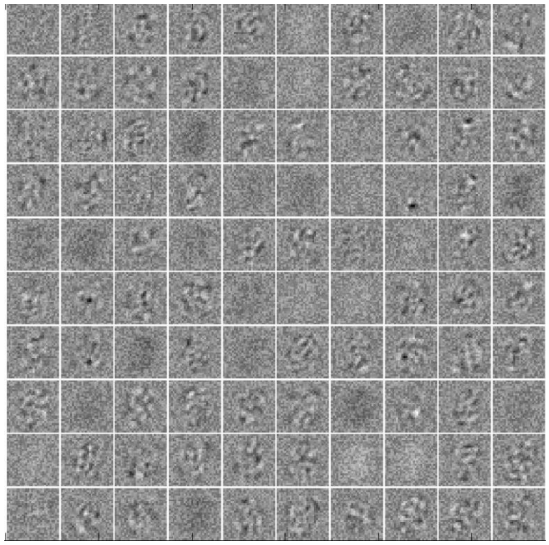

**Fig. 12.** stacked 784-100 epoch-1 and no noise

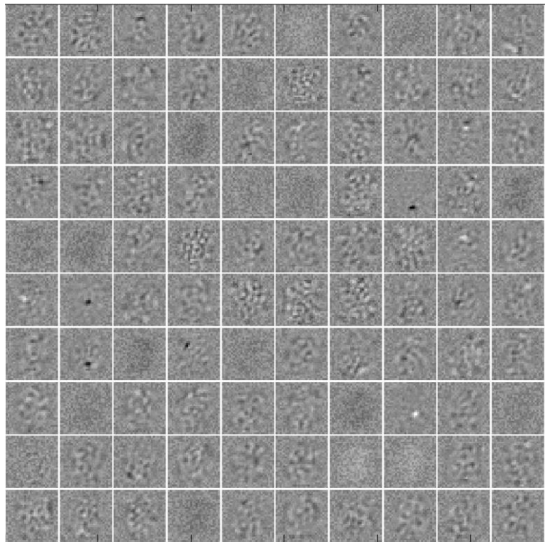

**Fig. 13.** stacked 784-100 epoch-10 and no noise

The same argument can go for Fig. 12 and Fig. 13 also. Weight learnt gets more and more significant as epochs increases.

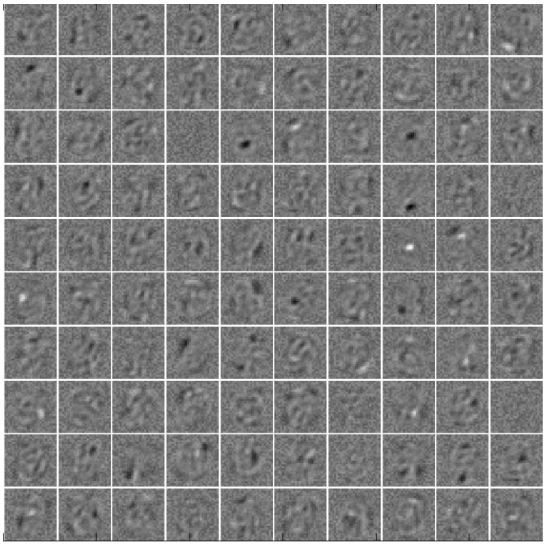

**Fig. 14.** stacked 784-100 epoch-1 and noise

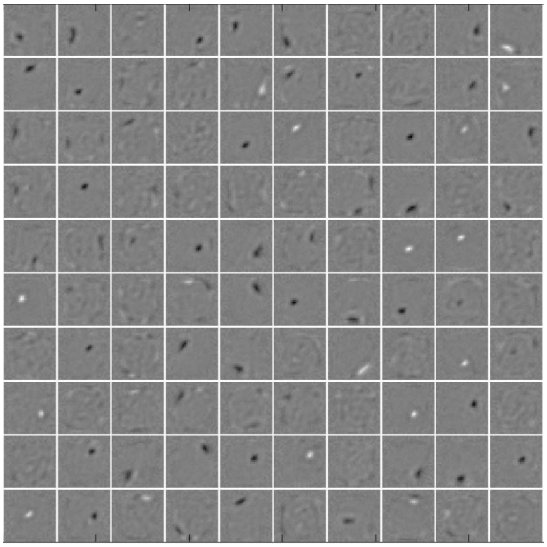

**Fig. 15.** stacked 784-100 epoch-10 and noise

You can say the same for Fig. 14 and for Fig. 15 also.

**Some Stats** Reconstruction mean squared over test data for MNIST for various architecture and pre-training methods are
RBM-784-500-250-30 for 10 epochs 5.0478
RBM-784-1000-500-30 for 10 epochs - 5.5902
Stacked-784-500-250-30 with 0.5 noise for 10 epochs - 3.3391
Stacked-784-500-250-30 with 0 noise for 10 epochs - 5.0929
Stacked-784-1000-500-30 with 0.5 noise for 10 epochs - 3.7217
Stacked-784-1000-500-30 with 0 noise for 10 epochs - 26.0131
Now I will try to explain these results as to why the stacked with 1000 nodes in the 2nd layer performs so poorly. First of all it is well established that the nodes in the second layer should be greater than that of the first layer, so as to reduce in information loss. So by that means the model of stacked with 1000 in 2nd layer should perform better than or atleast equal to the model of stacked with 500 units in the 2nd layer. The reason for that can be explained from the graphs below. These graphs are the RBM error(in case of RBM) or mean squared error in case of stacked.

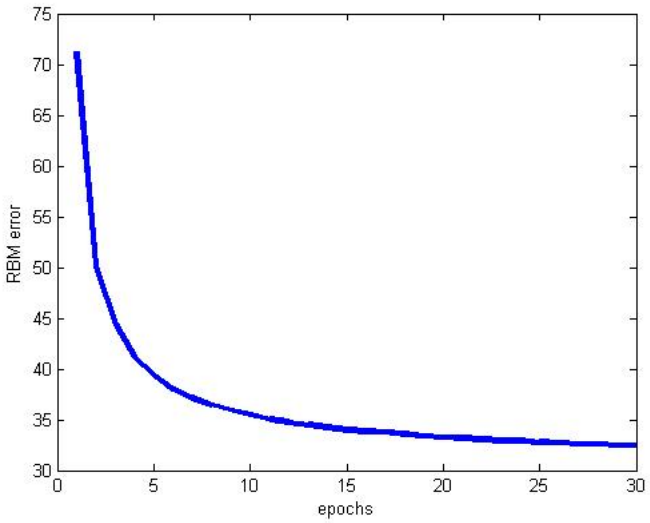


**Fig. 16.** Error Vs Epoch for RBM 784-1000

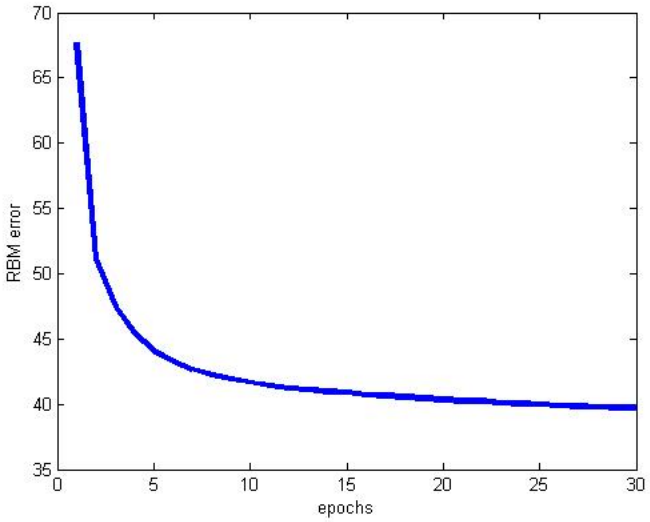


**Fig. 17.** Error Vs Epoch for RBM 784-100

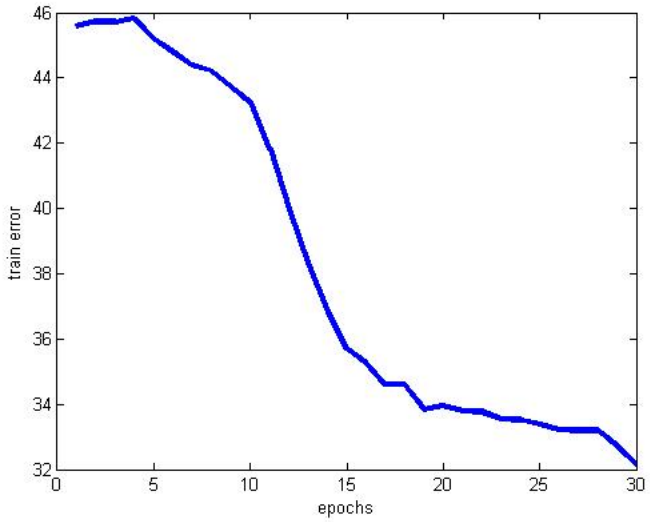


**Fig. 18.** Error Vs Epoch for stacked 784-1000 no noise

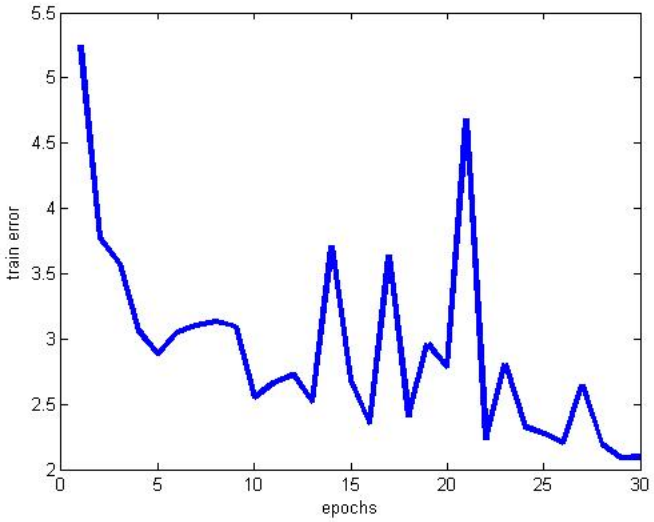


**Fig. 19.** Error Vs Epoch for stacked 784-100 no noise

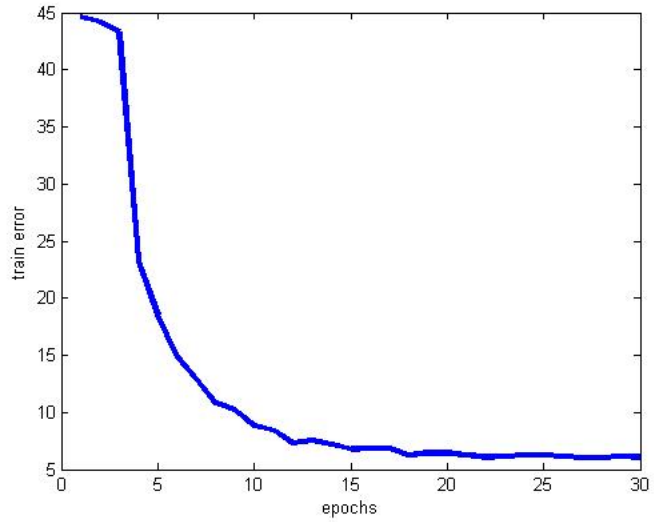


**Fig. 20.** Error Vs Epoch for stacked 784-1000 with noise

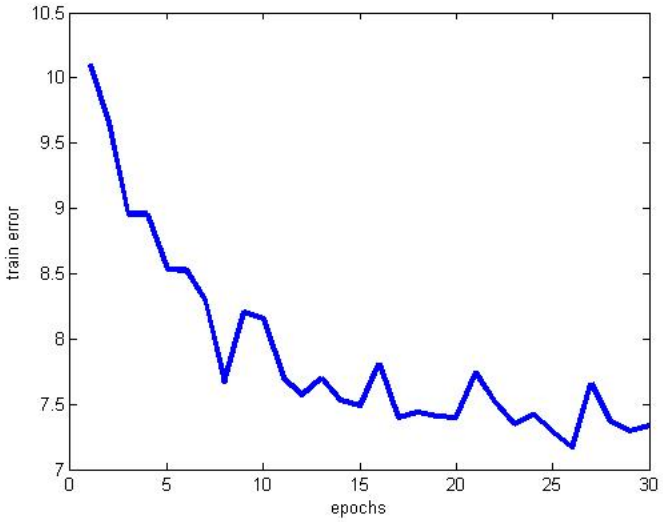


**Fig. 21.** Error Vs Epoch for stacked 784-100 with noise

As we can see in case of RBM the curve nearly flattens out after 10 epochs. For stacked noise it is the same, but for stacked with no noise for the model 784-100 the graph underfits after 10 epochs, so if we would have trained it further the results we got will be poorer. Now for the Stacked with 1000 units, you can see that the error at 10 epoch is very high compared to other. So that is why it performs so poorely. If I would have trained it more than it would have came close to RBM.

### 5.2 2D-Robot Arm

As the image was 100 x 100. So the input vector is of size $10,000$ which is very large to be trained on my machine. So initially the architecture I was using is $10000 - 1000 - 500 - 30 - 500 - 1000 - 10000$. The image is reconstructed by me from the dataset available here

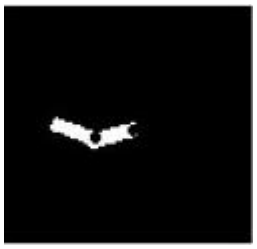

**Fig. 22.** Orignal Robot Image

The reconstructed image for the architecture defined above for rbm and stacked are shown below

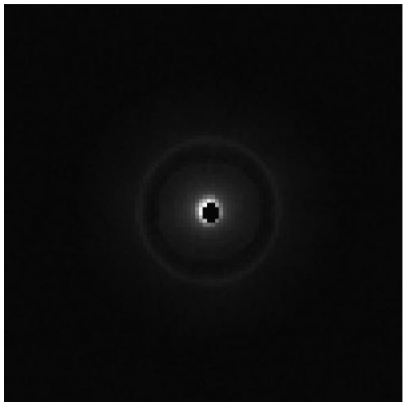

**Fig. 23.** rbm

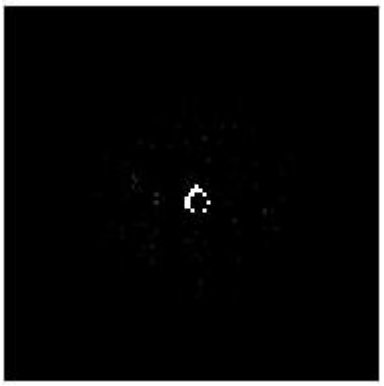

**Fig. 24.** stacked

The problem here is that our network is learning the average of all the inputs. I tried to find out why and came to the conclusion that as the information loss from 10000 to 1000 units is huge. So I tried to pre-train the network using RBM for architecture $10000-10000$ and $10000-15000$ and than tried to visualize the weights but the results remain the same. It is shown in Fig.25

which was still the average of all the inputs. So I wrote a script which selected only the images whose first arm is between $0°$ and $180°$ and the result is shown in Fig.26.

So I had no intution why is this happening and so was not able to figure out.

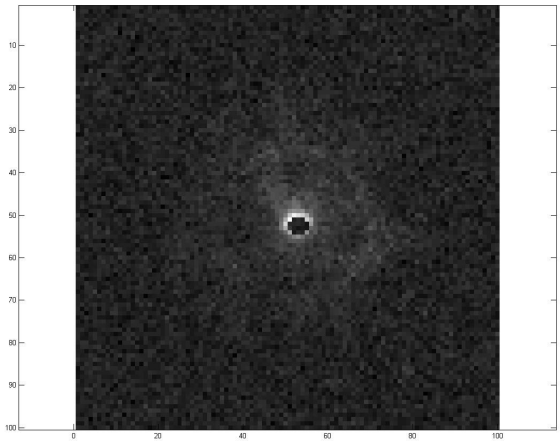


**Fig. 25.** rbm for 10000-10000

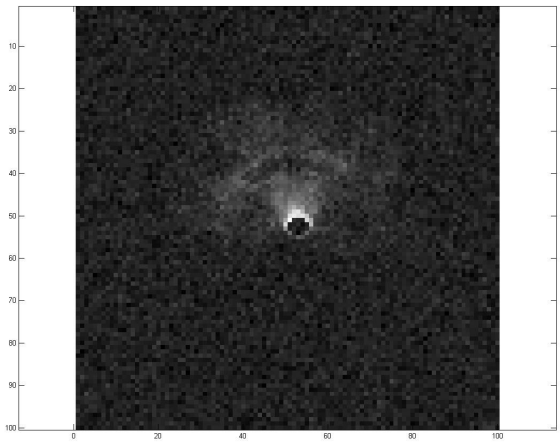


**Fig. 26.** rbm for 10000-10000

## 6 Conclusion

There is huge gap between Stacked autoencoder and RBM. This need to be filled. The fact that stacked requires more pre-training than RBM make stacked much of less importance. Although their final results are comparable but stacked need more of a computation. Many work has been done to bridge this gap which includes alternate layer sparsity and intermediate fine-tuning and has achieved some good results which are mentioned in [4].

## References

1. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion.
2. Pierre Baldi and Kurt Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58, 1989.
3. Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.
4. Ankit Bhutani. *Alternate Layer Sparsity and Intermediate Fine-tuning for Deep Autoencoders*. PhD thesis, INDIAN INSTITUTE OF TECHNOLOGY, KANPUR, 2014.
5. Asja Fischer and Christian Igel. An introduction to restricted boltzmann machines. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 14–36. Springer, 2012.
6. Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.
7. Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
8. R. B. Palm. Prediction as a candidate for learning deep hierarchical models of data, 2012.
9. Paul Smolensky. Information processing in dynamical systems: Foundations of harmony theory. 1986.