Deep Neural Networks state of the art and testing

David Buchaca & Claudio Levinas

January 20, 2013

1 The breakthrough of training Deep Neural Networks

Until 2006, DNN were seen as a intractable and impractical type or Neural Networks. This has primarily two reasons, the poor training and generalization errors obtained by DNN and the everlasting problem of gradient-based training getting stuck in a local minima. In fact as the architecture gets deeper it is more probable that backpropagation could get stuck in a local minima so it becomes more difficult to obtain a good generalization. This happens even though $l+1$ layer networks can easily represent what a l-layer network can, whereas the converse is not true.

It was discovered in [1] that using Restricted Boltzmann Machines (RBM) much better results could be achieved by pre-training each layer, one after the other (starting from the first layer) with an unsupervised learning algorithm. After [1] was published several statistical comparisons [2,3] have "demonstrated" the advantages of pre-training versus random initialization.

The main idea behind pre-training consist in training first the lower layer with an unsupervised learning algorithm (such a RBM) giving rise to an initial set of parameter values for the first layer of a neural network. Then using the output of the first layer (as input for another layer), initializing the second layer with an unsupervised learning algorithm and so on. After having initialized several layers, the whole NN can be trained with a supervised learning algorithm as usual.

It is hypothesized that in a well trained DNN, hidden layers form a good representation of the data, which helps to make good predictions.

2 Deep Generative Architectures

Generative models are just graphical models in which nodes represent random variables and edges represent the dependency. One of the big problems about Neural Networks has been the initialization of the weights.

Neural networks have been using sigmoid functions in order to model the output of a neuron given its input. One of the main reasons to do so has been that the sigmoid function is differentiable so we can compute the derivatives of the output of a neuron given its input. This nice smooth property is the core idea in which backpropagation was founded. But apart from that there has not been any other big reason to use the sigmoid function instead of another non decreasing differentiable function from 0 to 1. In subsection 2.1 we will see that the sigmoid function arises naturally when we compute the probability of a hidden unit ($h_i = 1$) given the input, or when we compute the probability of a visible unit ($v_i = 1$) given the hidden layer.

2.1 Restricted Boltzmann Machine

Deep Neural networks are based on Restricted Boltzmann Machines which are a particular case of energy-based models. Energy-based models associate an energy to each configuration of the variables of interest. Learning consists on changing the energy function so that desirable configurations have low energy. This type of probabilistic models define a probability distribution through a energy function as follow:

$$P(x) = \frac{e^{-Energy(x)}}{Z}$$

Where Z is called "partition function" and it is defined as follows:

$$Z = \sum_{x \in Input} e^{-Energy(x)}$$

We are particularly interested in a concrete type of energy-based model, in RBMs. Restricted Boltzmann Machines are undirected graphical models with the following energy function:

$$E(v, h) = -b^t v - c^t h - v^t W h$$

where b and c are column vectors (transposed) and v and h are column vectors containing the visible and hidden units respectively.

RBMs must have the following structure

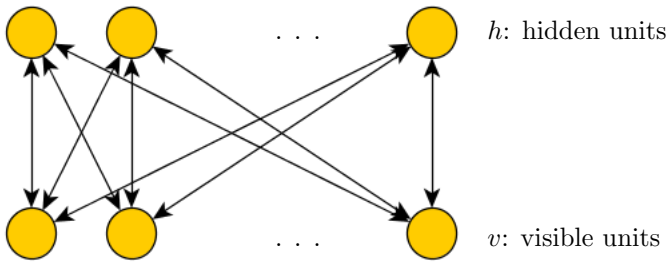


Figure 1: There are no links between units in the same layer. All links are between input (visible) units v_i and hidden units h_j

Some observations about RMB

Observation 2.1 Given v and h and g the sigmoid function, any RBM satisfies that the hidden units are independent given the visible units. That is:

$$P(h|v) = \prod_i P(h_i|x)$$

The prove is straight forward:

$$P(h|v) := \frac{e^{-Energy(v,h)}}{\sum_h e^{-E(v,h)}} = \frac{e^{b^t v + c^t h + v^t W h}}{\sum_h e^{b^t v + c^t h + v^t W h}} = \left(\frac{e^{b^t v}}{e^{b^t v}}\right) \frac{e^{c^t h + v^t W h}}{\sum_h e^{c^t h + v^t W h}} = \prod_i P(h_i|x)$$

Observation 2.2 Given v and h binary valued vectors, and g the sigmoid function, any RBM satisfies:

$$P(h_i = 1|v) = g(c_i + v^t W i)$$

$$P(v_i = 1|h) = g(b_i + v^t W i)$$

We will prove the first equality (the second is also true, v and h play symmetrical roles in the energy function).

Using the previous observation we know that:

$$P(h|v) = \frac{e^{c^t h + v^t W h}}{\sum_h e^{c^t h + v^t W h}} = \frac{\prod_i e^{c_i h_i + v^t W_i h_i}}{\prod_i \sum_{\widehat{h_i}} e^{c_i \widehat{h_i} + v^t W_i \widehat{h_i}}} = \prod_i \frac{e^{c_i h_i + v^t W_i h_i}}{\sum_{\widehat{h_i}} e^{c_i \widehat{h_i} + v^t W_i \widehat{h_i}}}$$

If we impose that $h_i = 1$ we have

$$P(h_i = 1|v) = \frac{e^{c^t + v^t W}}{1 + e^{c^t + v^t W}}$$

that is true because we are assuming that the hidden variables take binary values, so:

$$\sum_{\widehat{h_i}\in\{0,1\}} e^{c_i\widehat{h_i}+v^tW_i\widehat{h_i}} = e^{c_i 0+v^tW_i 0} + e^{c_i 1+v^tW_i 1} = 1 + e^{c_i+v^tW_i}$$

Observation 2.3 In a RBM the increment rule for the weights can be computed as a difference of two correlations:

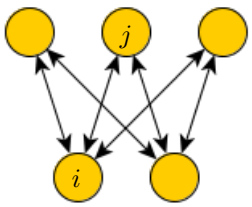
$$\frac{\partial \log P(v)}{\partial w_{ij}} =< s_i s_j >_v - < s_i s_j >_{model} \text{ [1]}$$

- $< s_i s_j >_v$ is the expected value of the product of states after runnning the Markov chain until it reaches its stationary distribution (thermal equilibrium) when v is fixed on the visible units.
- $< s_i s_j >_{model}$ is the expected value of the product of states at thermal equilibrium with no clamping (no fixed input).
- All this implies

$$\Delta w_{ij} = \epsilon\,(< s_i s_j >_{data} - < s_i s_j >_{model})$$

An efficient mini-batch learning procedure for RBM

Given a RBM we would like to know how to update the weights between a hidden unit j and a visible unit i.



h: hidden units

v: visible units

We have already seen in observation 2.3 that

$$\Delta w_{ij} = \epsilon\,(< s_i s_j >_{data} - < s_i s_j >_{model})$$

The problem is that, in order to measure $< s_i s_j >_{model}$ we should start with a training vector on the visible units then alternate between updating all the hidden units in parallel and updating all the visible units in parallel until the chain reaches an equilibrium. This might take a long time so it is not practical.

[1] A prove of this equality can be find in "The elements of Statistical learning" chapter 17.4.

Hinton realized empirically that learning still could work well by starting the Markov chain at the data, and only running for two steps as follows:

- Start with a training vector in the visible units
- Update all the hidden units in parallel.
- Update all the visible units in parallel to get a so called "reconstruction"
- Update the hidden units again.

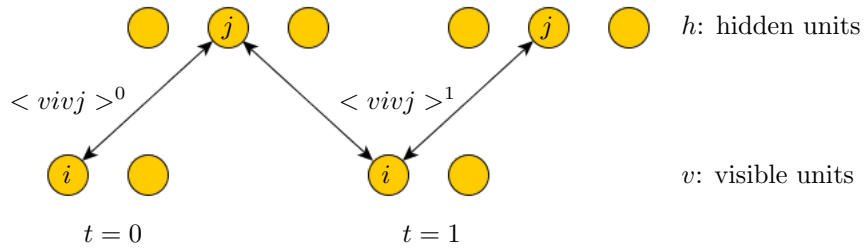
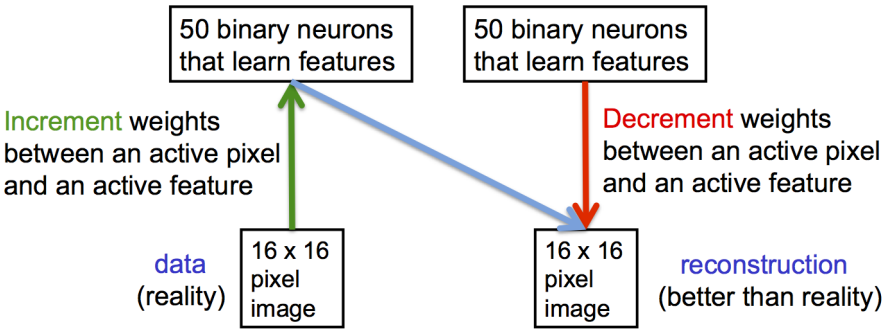


Figure 2: The learning rule is $\Delta w_{ij} = \epsilon(< vivj >^0 - < vivj >^1)$

Example: How to learn a set of features that are good for reconstructing images of a digit



2.2 Deep Belief network

A Belief Network is a directed acyclic graph composed of stochastic variables. A deep belief network with observed vector v and hidden layers h^1, h^2, h^3 has the following form:

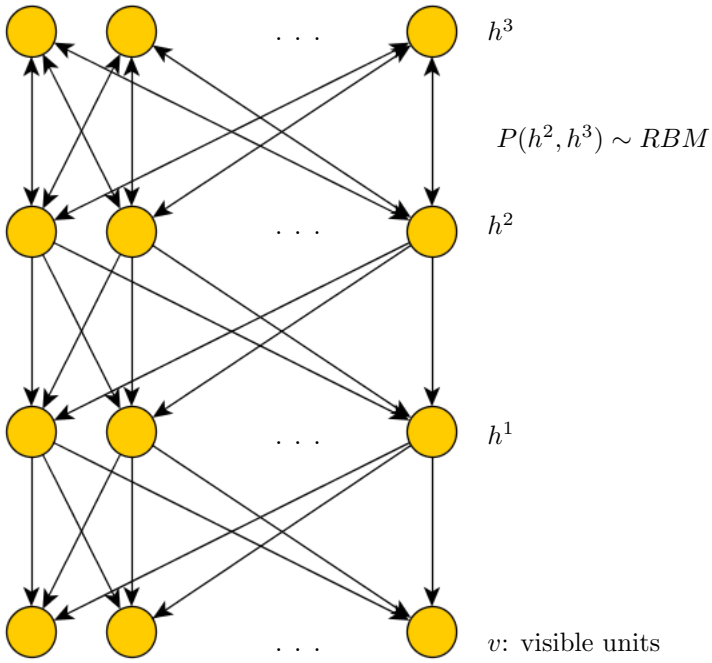


Figure 3: The visible vector is connected to h^1 which is connected to h^2 and between h^2 and h^3 there is a RBM.

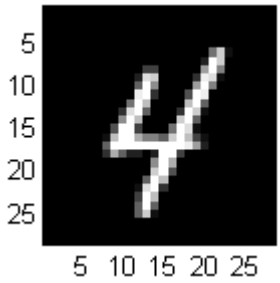
The RBM at the top minimizes the error between the targets and their model predictions, conditional to the input features v. It can extract information from the input features that can be useful for predicting the lables. Unlike supervised learning techniques a RBM can use some of its hidden units to model structure in the feature vectors that is not immediately relevant for predicting the labels. These features may turn out to be useful when combined with features derived from other hidden layers. This is one of the advantages of DNN, it is said that this kind of architecture achieves high levels of abstraction.

It is important to point out that the terms Deep Neural Network and DBN could be taken as equivalent because no one will train a NN with hundreds of layers in the classical way. In the literature we have seen both terms mixed without any distinction.

3 Deep Neural Networks vs Neural Networks: MNIST testing

3.1 MNIST Database

The MNIST database of handwritten digits [5] has become a standard tool to measure machine learning algorithms. The database consists scanned handwritten digits ranging from 0 to 9, arranged in two sets, one for training with 60,000 instances, and one for testing with 10,000 instances. The digits are grayscale images, and are normalized in size to fit a 28 by 28 window; that is, each scanned number (an instance) consists of 784 features. An example of one of the instances is depicted as follows:



The MNIST database has been used since 1998 to measure the performance of, amongst others, K-Nearest Neighbours, SVMs, Neural Networks, Convolutional Neural Networks, and most recently, Deep Neural Networks. Errors quoted in [5] range from 12% for a 1 layer (linear) Neural Network, to 0.23% for a Deep Neural Network. The ubiquitous nature of the database in machine learning makes it a excellent tool for comparative purposes.

3.2 Arrangement of the Experiment

In order to properly compare the Deep Neural Network with a conventional Neural Network, state of the art implementations of both strategies were secured. For the Deep Neural Network, the source code used for testing was that provided by Hinton and Salakhutdinov in [6]. For a conventional Neural Network, the Matlab Neural Network toolbox was used.

The implementation of the Deep Neural Network makes use of all training samples available (60,000 instances), and tests the performance with the test set (10,000 instances). The source code was written in such a way that after a maximum number of epochs is chosen, the algorithm will run until this number is reached (no early stopping). The Matlab Neural Network implementation, on the other hand, offers early stopping, but uses part of the training samples to do this.

It was decided that the best strategy to follow was to use early stopping in both cases, so 15% of the training samples (9,000 instances) were held out for

validation, leaving the rest (51,000 instances) for training. Matlab's implementation of early stopping performs validation of the model trained in each epoch, and keeps the best (lowest) error. If the error then increases during the next 6 consecutive epochs, early stopping is enforced.

A similar strategy was added to the Deep Neural Network implementation. However, heuristically it was found that the threshold used by Matlab (6 consecutive worse results) was inadequate for the Deep Neural Network. After many trial and error attempts it was decided that the optimal number of worse results should be 40. This is because the neural network fluctuates considerably, and using only 6 worse results yielded a much too early stopping point.

In sample and out of sample errors quoted later, refer to those found at the optimal epoch, not at the point where the algorithm was stopped. In other words, the algorithm goes on for an extra 40 epochs, in order to validate an optimal stop point, in the same way as Matlab does.

The maximum number of epochs was set to 500 for the case of the Neural Network, and to 100 for the case of the Deep Neural Network. The Deep Neural Network appears to converge much faster than the conventional Neural Network, and requires much less iterations of the algorithm; however, as it will be shown, each epoch takes considerably more time to back-propagate.

A warm-up execution of the Deep Neural Network, was tried, running with the default parameter values suggested by Hinton and Salakhutdinov, and found to take 40 hours to process. The architecture of the neural network defined by them was a three-layer network, with the first and second layers having equal number of hidden units, but the third layer having four times that number. The first and second layers were given 500 hidden units, while the last layer had 2000. Maximum epochs was set to 200, and the algorithm run without early stopping, using all the 60,000 training samples. Results were outstanding. The *error out* of this configuration was merely 0.97%.

The same architecture (3-layers, $\times 1, \times 1, \times 4$) was used to test both neural networks, however, in order to have a better understanding of their performance as a function of hidden units, their number ranged 25, 50, 75, 100, 200, 300, 400 and 500. As mentioned above, validation of training was used in both cases, for early stopping, with the same 9,000 instances in both neural networks.

3.3 Testing Source Code

Both neural networks were tested in Matlab via the implementations mentioned. The source code of Deep Neural Network was touched in two instances, to create the batches for validation and not re-parse the MNIST database on each run, and to early stop in order for the experiment to be feasible. For this purpose, the files *makebatches.m* and *backpropclassify.m* were renamed *<name>_orig.m*, and altered files took the original name.

Execution of the tests can be carried out by running *test_all_models.m* in the *deep learning* and the *neural network* directories. Alternatively, in order to run a single test (a test for a single value of the hidden units parameter), one

can run *test_deep_learning(numhid, maxepoch)* or *test_neural_network(numhid, maxepoch)*.

Results of the procedures are automatically saved in the working directory, and follow the convention <*#HU*>_<*descriptor*>.

3.4 Experimental Results

The results of running both neural networks with the architecture proposed by Hinton and Salakhutdinov, but with varying number of hidden units (#HU) are tabulated below:

#HU	#epochs	time	secs/epoch	train misses	E_{in}	test misses	E_{out}
25	214	04:46	1	1757	3.45	539	5.39
50	140	05:54	2	1094	2.15	390	3.90
75	140	08:42	4	1032	2.02	339	3.39
100	241	16:57	4	495	0.97	281	2.81
200	156	24:06	9	423	0.83	243	2.43
300	195	47:14	14	375	0.74	208	2.08
400	226	01:23:32	22	341	0.67	206	2.06
500	328	02:47:06	30	310	0.61	206	2.06

Table 1: Matlab Neural Network results

#HU	#epochs	time	secs/epoch	train misses	E_{in}	test misses	E_{out}
25	21	11:16	11	1858	3.64	567	5.67
50	48	16:57	12	115	0.23	334	3.34
75	22	16:46	16	71	0.14	245	2.45
100	13	22:34	26	106	0.21	203	2.03
200	43	02:07:15	92	0	0.00	147	1.47
300	58	05:23:01	198	0	0.00	125	1.25
400	36	06:25:18	304	0	0.00	118	1.18
500	19	06:06:26	373	0	0.00	115	1.15
500	200	1:16:02:13	721	0	0.00	97	0.97

Table 2: Deep Neural Network results

All results correspond to running the tests with a 3-layer architecture consisting of $\times 1$, $\times 1$, $\times 4$ number of hidden units (#HU), with 51,000 training instances, 9,000 validation instances, and 10,000 testing instances. The only exception is the last line of the Deep Neural Network table, which was trained with all 60,000 instances, and tested on 10,000 instances (no validation was done, and the algorithm run until the maximum number of epochs was reached at 200).

Matlab's Neural Network yielded the following E_{in} and E_{out} errors, as a function of the number of hidden units:

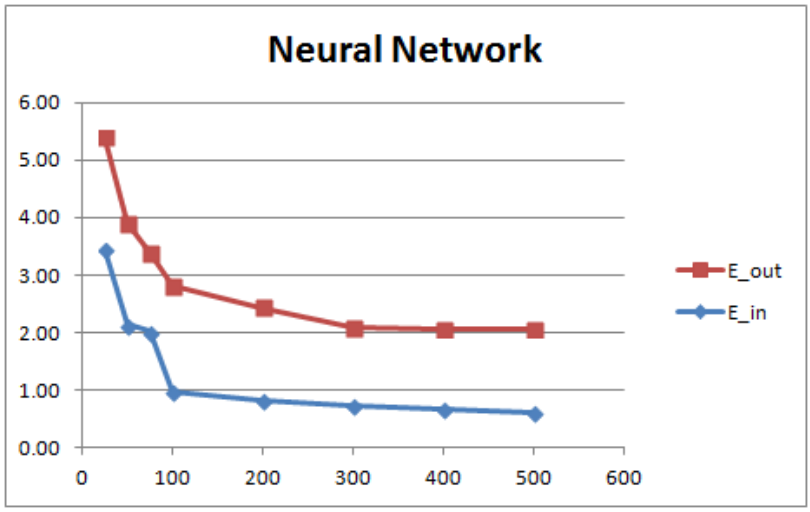


Figure 4: Neural Network E_{in} and E_{out} results

The Deep Neural Network yielded the following E_{in} and E_{out} errors, as a function of the number of hidden units:

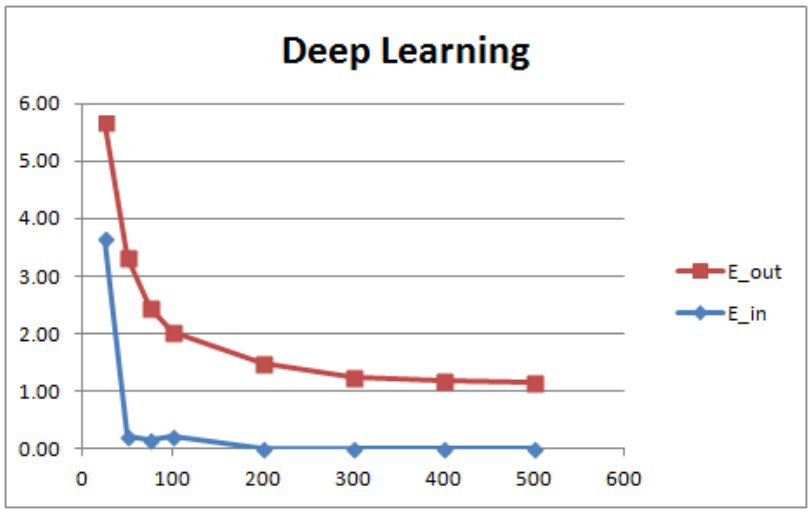


Figure 5: Deep Neural Network E_{in} and E_{out} results

A comparative between the performance of both networks is best seen in the following figure, which depicts the E_{out} as a function of the number of hidden units for both networks:

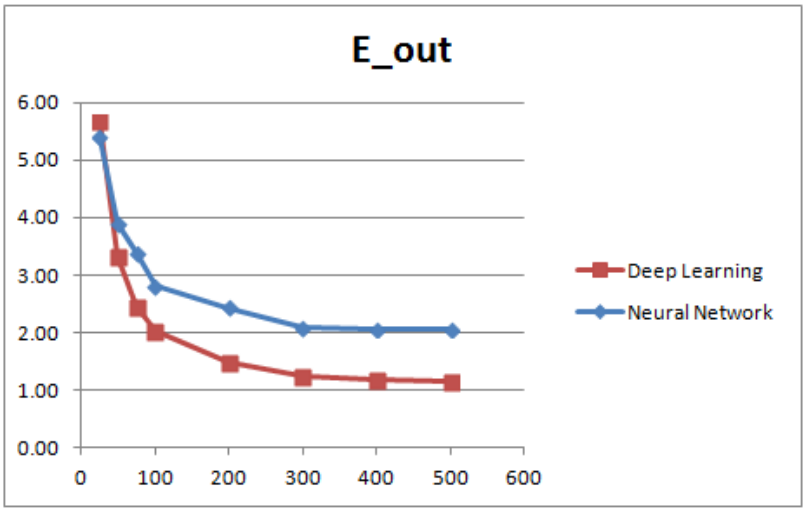


Figure 6: Comparative of E_{out} results

A second important comparative pertains the amount of time (in seconds) that took on average each network to complete one epoch.

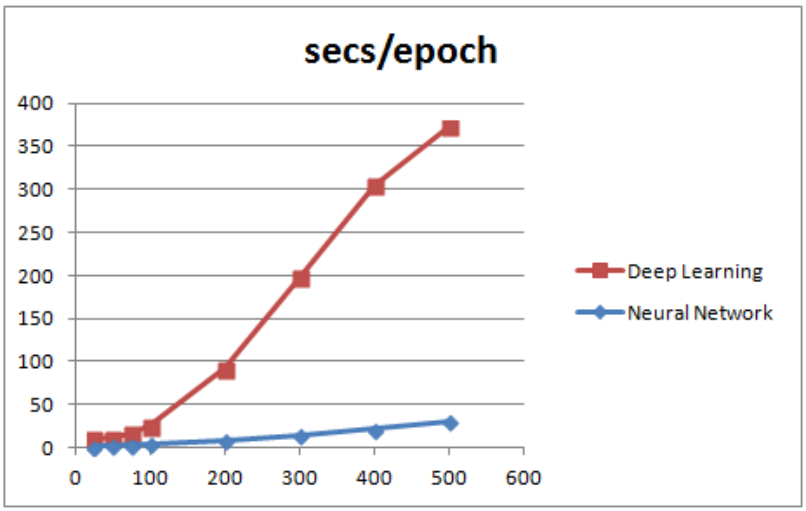


Figure 7: Comparative of epoch times

Lastly, the following figure shows the general trendline of epochs needed by each network to converge (early stop):

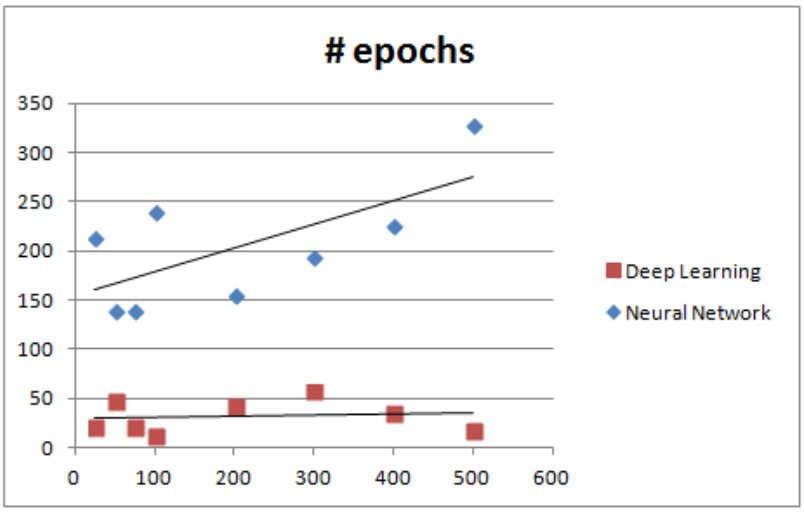


Figure 8: Comparative of early stopping

3.5 Discussion of the Results

Figures 4 and 5 show the relationship between E_{in} and E_{out} for both networks. As expected, E_{out} tracks E_{in}, and performance is worse out of sample than in sample.

Figure 6 shows the overall results obtained by both neural networks under the same identical testing conditions (same architecture and number of hidden layers). It is immediately clear that the Deep Neural Network outperformed the conventional Neural Network. In fact, with the 500 hidden units as suggested by Hinton and Salakhutdinov, the Deep Neural Network did so by 0.9%. This is a significant improvement, and might start to explain the enthusiasm of the community with the Deep Learning algorithm.

It might be the case that another network architecture could have benefited the conventional Neural Network more. This remains an interesting point for future exploration. It should be stated, however, that the results obtained for Matlab's Neural Network seem consistent with those reported by LeCun et al. in [5].

For the particular problem of MNIST classification, it is apparent that neural networks with high number of hidden units outperform those with fewer units. In both cases studied, tests for 300 to 500 hidden units yielded considerably better results than tests for fewer hidden units. Furthermore, both networks appear to be stabilizing around 400/500 hidden units, with little improvement past that number.

Figure 7 compares both networks from another important perspective, namely, time performance. In general, the Deep Neural Network required almost a factor of 10 to complete an epoch (backpropagate). This is significant, and somehow pales the previous performance improvement seen. In order to run all tests of the Neural Network, a combined time of 5h 58m 07s was needed, while the Deep Neural Network, under identical architecture and test range, required a combined time of 21h 9m 33s (a factor of 3.5).

While the Deep Neural Network does take considerably more time to complete an epoch, it does appear to converge in less iterations. This observation is shown in Figure 8 which describes the overall trend in the number of epochs run until early stopping. On average, the Neural Network required 205 iterations, while the Deep Neural Network required 32.5 (a factor of 6.3). Despite converging in less iterations, the excess processing time per epoch still renders it significantly slower.

Table 2 has an added row that reports the results of running the Deep Neural Network as suggested by Hinton and Salakhutdinov. Indeed, their setup yielded a better out of sample error, when compared to the tests conducted herein, however, it should be noticed that this particular architecture took 40 hours to process, which is nearly double the time it took to complete all tests for the Deep Neural Network with early stopping.

3.6 Conclusions and Future Work

The Deep Neural Network outperformed the conventional Neural Network in the tests conducted by almost an entire percent. This is a significant improvement for this particular classification problem. The better performance did, however, come with a hefty price; it took nearly 4 times more to process the same set of experiments, under identical setups.

It is conceivable that improvements in the implementation of the Deep Neural Network will bring these results closer together. Matlab's Neural Network toolbox has a highly optimized source code, while the version offered by Hinton and Salakhutdinov is perhaps better described as a *proof of concept*. Perhaps future implementations will render the time differences redundant.

This experiment focused only on comparing both neural networks under the architecture proposed by Hinton and Salakhutdinov. Many architectures can be devised, and perhaps found to outperform the one reported here. This particular issue remains open for future exploration.

References

[1] Geoffrey E.Hinton and Simon Osindero. *(2006) A fast learning algorithm for deep belief nets* (Neural Computation, 18 , 1527-1161).

[2] Y. Bengio, P. Lamblin and H. Larochelle, *"Greedy layer-wise training of deep networks"* (Advances in Neural Information Processing Systems, 19, pp- 153-169 MIT Press, 2007.)

[3] D. Erhan, P.A Manzagol, Y. Bengio, S. Bengio, and P. Vincent, *"The difficulty of training deep architectures and the effect of unsupervised pre-training"* (Proceedings of the Twelfth International Conference on Artificial intelligence and Statistics, pp. 153-160, 2009)

[4] The Elements of Statistical Learning: Data Mining, Inference and Prediction.

[5] Y. LeCun, C. Cortes, *The MNIST Database of handwritten digits*, `http://yann.lecun.com/exdb/mnist/`

[6] R. Salakhutdinov and G. Hinton, *Deep Learning Neural Network Matlab Implementation*
`http://www.cs.toronto.edu/~hinton/MatlabForSciencePaper.html`