Summary and discussion of: "Why Does Unsupervised Pre-training Help Deep Learning?"

Statistics Journal Club, 36-825

Avinava Dubey and Mrinmaya Sachan and Jerzy Wieczorek

December 3, 2014

1 Summary

1.1 Deep Learning and All That

Before getting into how unsupervised pre-training improves the performance of deep architecture, let's first look into some basics. Let's start with logistic regression, which is one of the first models for classification that is taught in machine learning. Logistic classification deals with the supervised learning problem of learning a mapping $\mathcal{F} : \mathcal{X} \rightarrow \mathcal{Y}$ given a set of training points $\mathbf{X} = \{\mathbf{x_1} \dots \mathbf{x_n}\}$ and a set of class labels $\mathbf{Y} = \{y_1, \dots, y_n\}$ where $\mathbf{x_i}$ is assigned a class label y_i. The mapping is defined by the function $p(Y = 1|X) = \frac{1}{1+\exp(-(W^T X+b))}$. There is another way of looking at the logistic classifier. One can think of the X as input to a node in a graphical model and the node does two things: it sums up the inputs multiplied by weights of the edges and then applies a sigmoid on the result. A diagram representing such a function is shown in Figure 1. The node that performs the summation and the non-linear transformation is called a neuron. The summation function is called input activation ($a(x) = W^T X + b$) and the non-linear transform ($h(x) = g(a(x))$ is called output activation of the neuron.

Let's take a look at an example in Figure 2. An AND function can clearly be modelled using a neuron but a XOR function cannot be directly modelled using a single neuron. If

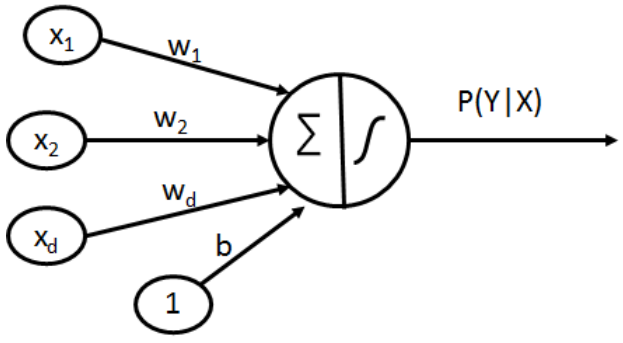

Figure 1: Diagram representation of logistic regression as a simple neural network

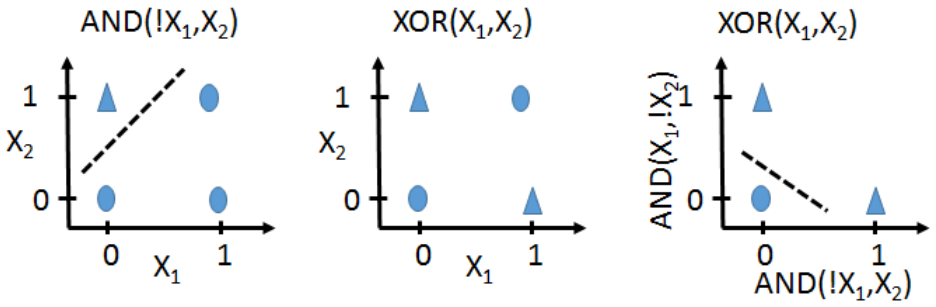

Figure 2: AND function on the left can be easily modelled by a single neuron, whereas a XOR function cannot be modelled using a single neuron, but after feature transform (on the right) it can be modelled

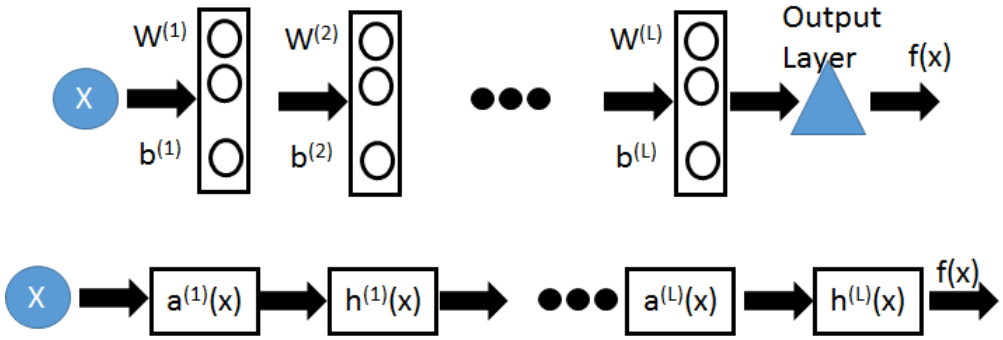

Figure 3: Top row represents a deep neural network, whereas the bottom row explains how the final function $f(x)$ is calculated

we can somehow transform the feature representation into a different representation shown in the right then another neuron can be used to learn the XOR function. This is the basic intuition behind deep neural networks. The hope of a multi-layered neural network is that the first layer of the neural network would learn a basic transformation of the input features. The second layer would learn a transformation of the output of the first layer and so on. This intuition will also be very helpful in pre-training.

Figure 3 shows a deep neural network of L layers. An interesting aspect of how the output function $f(x)$ is calculated is shown in the second row. The flow helps in identifying where the parameters of the model interact with the final output. This helps in learning the model. The parameters of the model are learned by empirical risk minimization and usually stochastic gradient descent is used to get to the solution. It can be easily seen that this is a non-convex optimization problem with multiple minima. To get the partial derivative of the model with respect to a particular parameter we find the place where it interacts with the final objective and using the chain rule we calculate backwards the derivative. This method is called back propagation.

1.2 Unsupervised Pretraining

Let's get back to one of the prior motivations of going deep with neural networks. One of the primary reason for doing so was to capture transformations of input features and then further transformations of the output of the first layer and so on. Learning a representation was not part of the learning method described above. One trick to learn the representation is to reproduce the input. Suppose we want to learn a L depth neural network. We start by first training the first layer so that the input can be produced as output. The is shown in the first row of Figure 4. By trying to reproduce the output the hope is that the neurons will learn a representation of the input. This is called an auto-encoder. (If noise is added to the input, but not to the output which is meant to be learned, then it is a denoising auto-encoder.) Next, weights of the first layer are kept fixed and the second layer's weights are learned so as to reproduce the output of the first layer. This hopes to capture a bit more complex of a representation than the first layer. This method of greedily training each layer one at a time is called pre-training with stacked auto-encoders. Finally all the weights are initialised to the learnt weights and the backpropagation algorithm is run on supervised data to get the final weights. This step is called finetuning. This procedure of learning has been shown to outperform just learning a DNN all at once from randomly initialised weights. There were two prominent hypotheses that were proposed in the paper to justify why this way of training works better than no pre-training.

The Optimisation Hypothesis: The first hypothesis is that since this is a non-convex optimisation problem the pre-training helps in providing a better initialisation point leading to better optimisation of the empirical risk.

The Regularisation Hypothesis: The second hypothesis that this paper checks for is whether such a method acts as a better regularisation and hence leads to a better generalization error.

1.3 Experiments in the Paper

Experimental setup: The paper presents a host of large-scale simulations that make useful connections to the two hypotheses. The main experiments are carried out on the MNIST dataset (or the Infinite MNIST dataset). The MNIST dataset [3] is comprised of 60,000 training and 10,000 testing examples of 28x28 handwritten digits in gray-scale and the task is to classify the digits into one of 10 classes (0-9). The InfiniteMNIST dataset [4] is an extension of the MNIST dataset and potentially contains an infinite number of examples obtained by performing random elastic deformations of the original MNIST digits. The experiments use Deep Belief Networks (DBNs) containing either Bernoulli RBM layers, Stacked Denoising Auto-Encoders (SDAE) with Bernoulli input units, or standard feed-forward multi-layer deep neural networks (DNNs) with 1-5 hidden layers.

Each hidden layer contains the same number of hidden units. The number of hidden units, learning rate, the L2 penalty / weight decay, and the fraction of stochastically corrupted inputs (for SDAE) are hyper parameters and are tuned by a grid search among a small set of values on the validation set: number of hidden units $\in \{400, 800, 1200\}$, learning rate $\in \{0.1, 0.05, 0.02, 0.01, 0.005\}$, L2 penalty coefficient $\in \{10^4, 10^5, 10^6, 0\}$, pre-training learning rate $\in \{0.01, 0.005, 0.002, 0.001, 0.0005\}$, corruption probability $\in \{0.0, 0.1, 0.25, 0.4\}$, and tied weights $\in \{yes, no\}$. For MNIST, the number of supervised and unsupervised

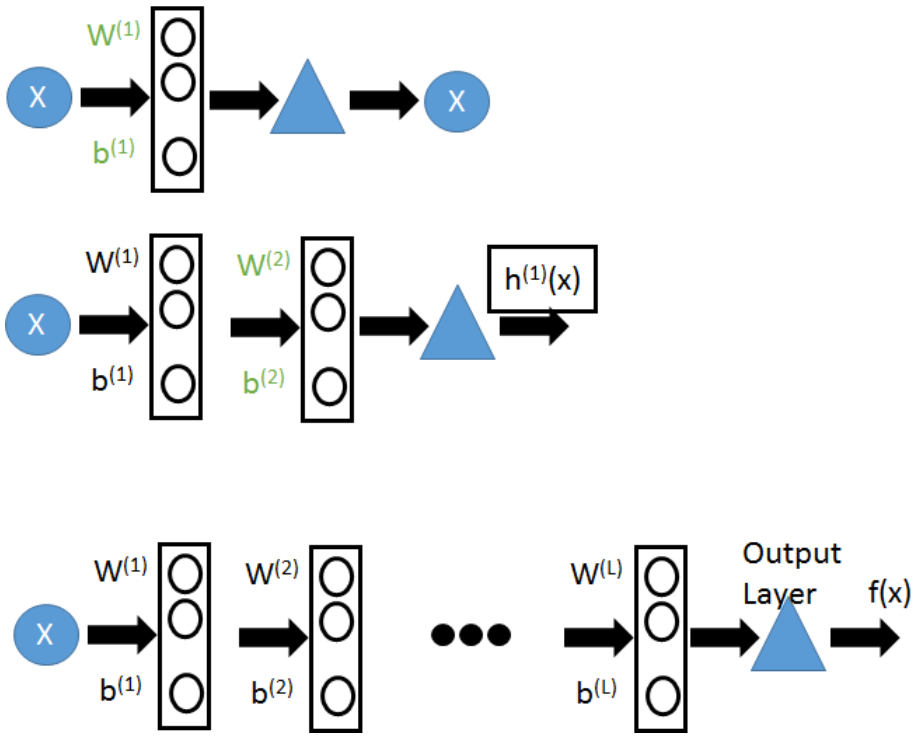

Figure 4: Pre-training a DNN. The first row represents the first layer being trained to reproduce X, the second row is where the first layer's weights are fixed and the second layer is trained to reproduce the output of the first layer, and so on. Finally we make use of the supervised data to learn all the weights after they have been initialised to the weights learnt in the previous step

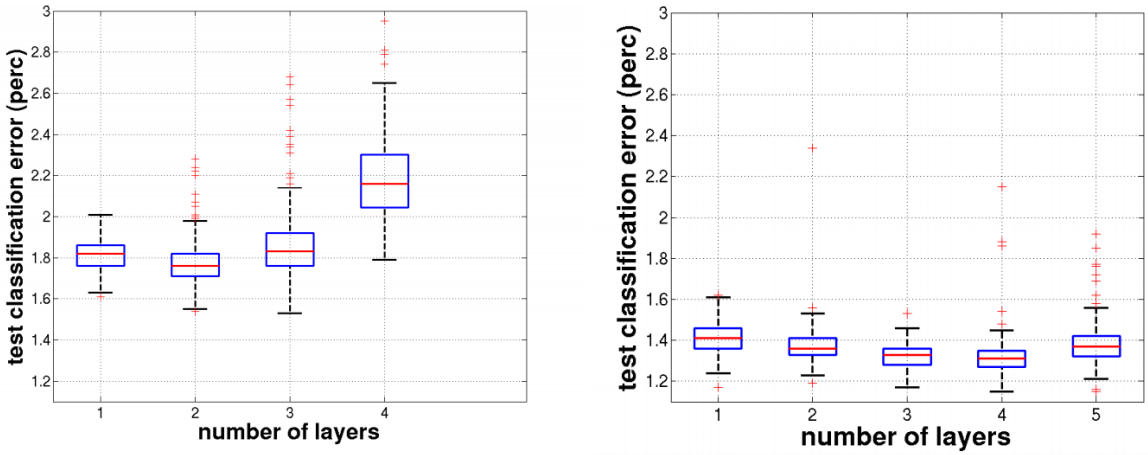

Figure 5: Plots showing the effect of depth on performance for a model trained on the MNIST dataset. Plot on the left shows the case when unsupervised pretraining is not used and the plot on the right shows the case when unsupervised pre-training is used. The depth of the network is varied from 1 to 5 hidden layers (networks with 5 layers failed to converge to a solution, without the use of unsupervised pretraining). Box plots show the distribution of errors associated with 400 different initialization seeds (top and bottom quartiles in box, plus outliers beyond top and bottom quartiles). Images taken from [1]

passes through the data (epochs) is 50 and 50 per layer, respectively. With InfiniteMNIST, we perform 2.5 million unsupervised updates followed by 7.5 million supervised updates. The standard feed-forward networks are trained using 10 million supervised updates. For MNIST, model selection is done by choosing the hyperparameters that optimize the supervised (classification) error on the validation set. For InfiniteMNIST, average online error over the last million examples was used for hyperparameter selection. In all cases, purely stochastic gradient updates were applied. For a given layer, weights were initialized using random samples from Uniform$\left[\frac{-1}{\sqrt{k}}, \frac{1}{\sqrt{k}}\right]$, where k is the number of connections that a unit receives from the previous layer.

Observation 1 (Better Generalization): Figure 5 plots the distribution of test classification error, obtained with and without pre-training, as the depth of the network varies from 1-5. As evident in the plot, the test classification error goes down as we move from 1 to 4 hidden layers, whereas without pre-training the error goes up after 2 hidden layers. The authors report that they were unable to effectively train 5-layer models without use of unsupervised pre-training. It is also evident that the error obtained on average with unsupervised pre-training systematically lower than without the pre-training. More importantly, it can be observed that the pre-trained model appears also more robust to the random initialization as compared to the the model without pre-training. This can be interpreted from observing the width of the quartile boxes in the two plots - the number of bad outliers grow more slowly when unsupevised pre-training is used. Also, the gain obtained with unsupervised pretraining is more pronounced as we increase the number of layers, as is the gain in robustness to random initialization.

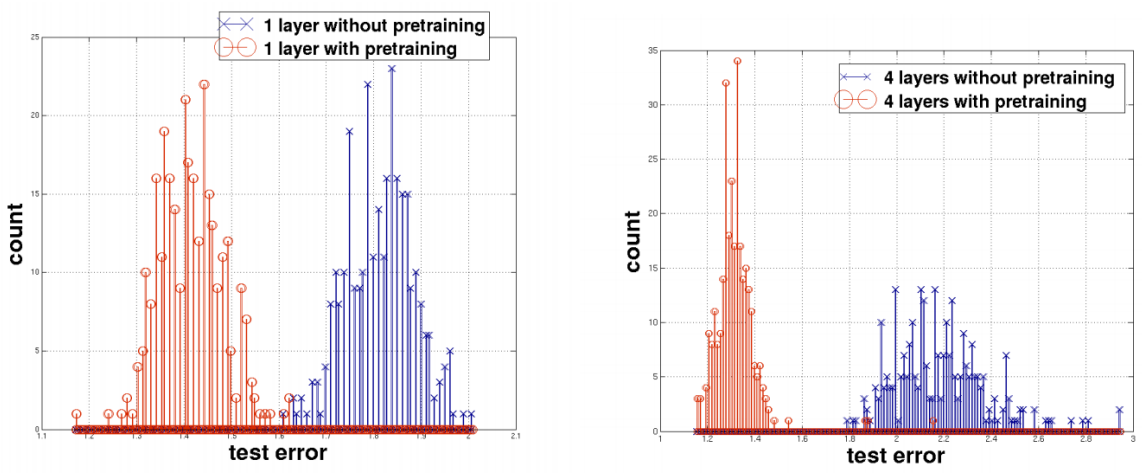

Figure 6: Plots showing histograms presenting the test errors obtained on the MNIST dataset using models trained with or without pre-training (400 different initializations each). The plot on the left is when the network employs 1 hidden layer, whereas the plot on the right is when the network employs 4 hidden layers. Images taken from [1]

The last observation is further clarified in Figure 6 which plots the histograms of the test errors using models trained with or without pre-training. The increase in error variance and mean for deeper architectures without pre-training as compared to the pre-trained model seen in the plot along with the findings in Figure 5 supports the authors arguments that **increasing depth increases the probability of finding poor apparent local minima when starting from random initialization** and **unsupervised pre-training is robust with respect to the random initialization seed**.

Observation 2 (Better Features): Figure 7 shows the weights (filters) of the first layer of a DBN before and after supervised fine-tuning when pre-training is used and also when pre-training is not used. For visualizing what units do on the 2nd and 3rd layer, the activation maximization technique [2] was used. First, we can roughly observe increasing granularity of features that are observed in various layers in these deep nets. The first layer learns some kinds of stroke-like local features, the second layer learns some kinds of edge features and the third layer learns features which more closely resemble the digits. When pre-training is not used, while the first layer filters do seem to correspond to localized features, 2nd and 3rd layers are not as interpretable anymore. The authors attribute this to the problem of high non-convex nature of the objective - **unsupervised pre-training "locks" the training in a region of the parameter space that is essentially inaccessible for models that are trained in a purely supervised way**.

Observation 3 (Different model trajectories): This experiment supports the last argument that the pre-trained and un-pre-trained models cover very different regions in parameter space - possibly alluding to the fact that the un-pre-trained parameter trajectories many indeed be getting stuck in many different apparently not-so-good local minimas. Figure 8 compares the function (the ordered set of output values associated with the inputs) represented by each network when pre-training is used or not. For a given model, all its

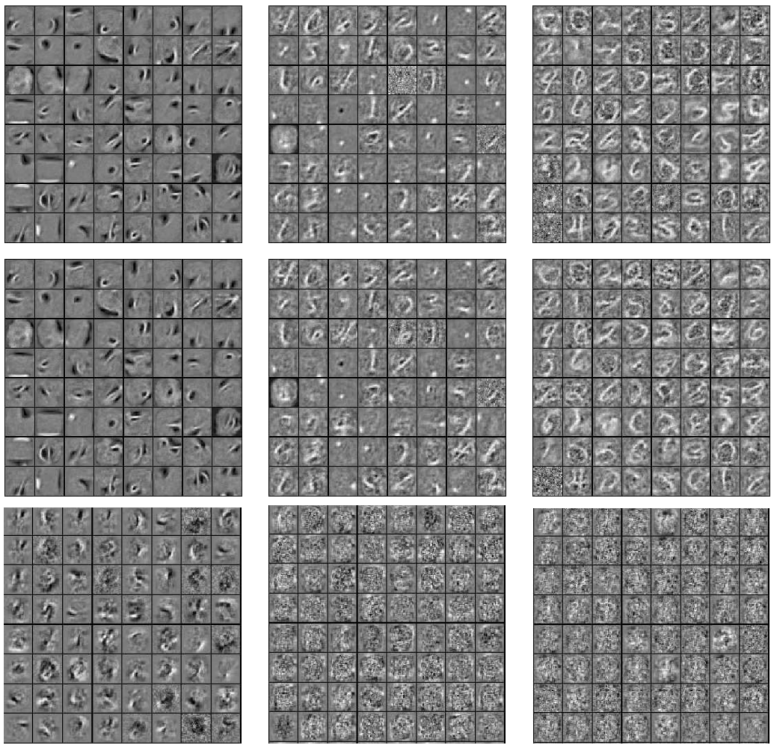

Figure 7: Visualization of image filters learned by a 3 layer DBN trained on InfiniteMNIST dataset. The top figures contain a visualization of filters after pre-training, the middle ones picture the same units after supervised fine-tuning, and the bottom ones picture the filters learned by a network without pre-training. From left to right: units from the 1st, 2nd and 3rd layers, respectively. Images taken from [1]

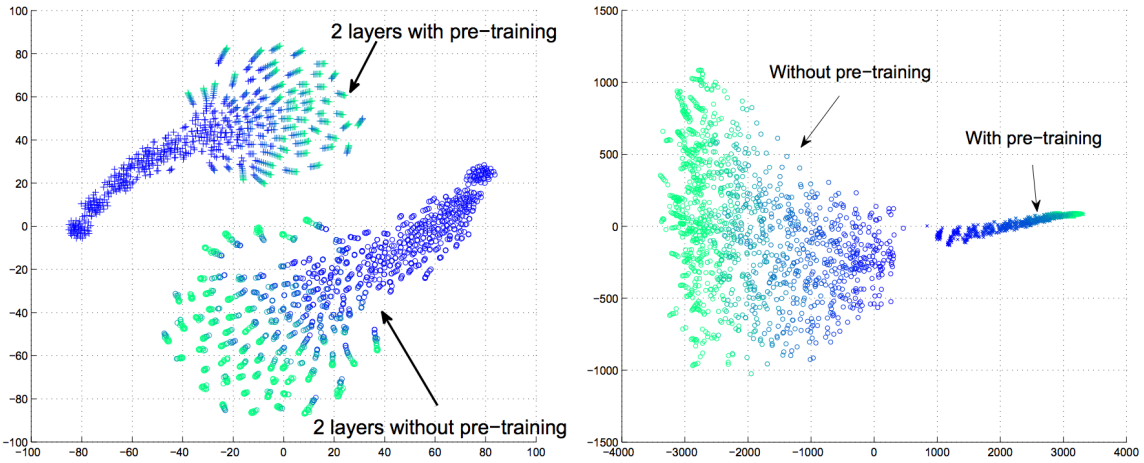

Figure 8: 2D visualizations with tSNE (left) and ISOMAP (right) of the functions represented by 50 networks with and 50 networks without pre-training, as supervised training proceeds over the MNIST dataset. Color from dark blue to cyan indicates a progression in training iterations. Images taken from [1]

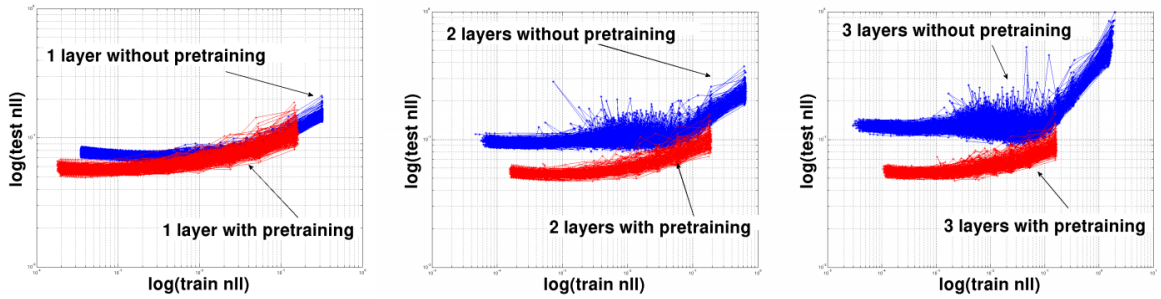

Figure 9: Evolution without pre-training (blue) and with pre-training (red) on the MNIST dataset of the log of the test NLL plotted against the log of the train NLL as training proceeds. Each of the 2×400 curves represents a different initialization. The errors are measured after each pass over the data. The rightmost points were measured after the first pass of gradient updates. Since training error tends to decrease during training, the trajectories run from right (high training error) to left (low training error). Images taken from [1]

outputs on the test set examples were computed and concatenated as one long vector summarizing where it stands in "function space" - leading to one such vector for each partially trained model. Many learning trajectories were plotted, one for each initialization seed, with or without pre-training. Using two well-known dimensionality reduction algorithms, these vectors were mapped to a two-dimensional space for visualization. The points are colored according to training iteration numbers, to help follow the trajectory movement.

The first plot (plot on the left) shows that: (a) **Pre-trained and not pre-trained models start and stay in different regions of function space**, (b) We see that all trajectories of a given type (with pre-training or without) initially move together. However, at some point, the **trajectories corresponding to the un-pre-trained models diverge and never get back close to each other**. However, the pre-trained trajectories, seem to be converging to a smaller space after an intermittent divergent behavior.

The second plot (plot on the right) show that the pre-trained models live in a disjoint and much smaller region of space than the un-pre-trained models. Indeed the pre-trained solutions appear to become all the same, and their self-similarity increases with time. This is another confirmation of the fact that while un-pre-trained parameter trajectories many indeed be getting stuck in many different local minimas, the local minimas obtained by pre-trained model are much smaller in number and lead to better solutions.

Observation 4 (Effect of Pre-training on Training Error): All of the above experiments are ambivalent with respect to the two hypothesis proposed in the paper. Evidence such as the control in the generalization error as the model complexity (number of layers) goes up (Figures 5 and 6) when pretraining is used points to the pretraining having some regularization effect. At the same time, Figure 8 point out that the pre-trained and un-pre-trained models indeed have different model trajectories hinting at some optimization effect as well. The next experiment wishes to discriminate between the two hypothesis.

To do so, it recalls a well known property of regularizers. The optimization and regularization hypotheses diverge on their prediction on how unsupervised pre-training should

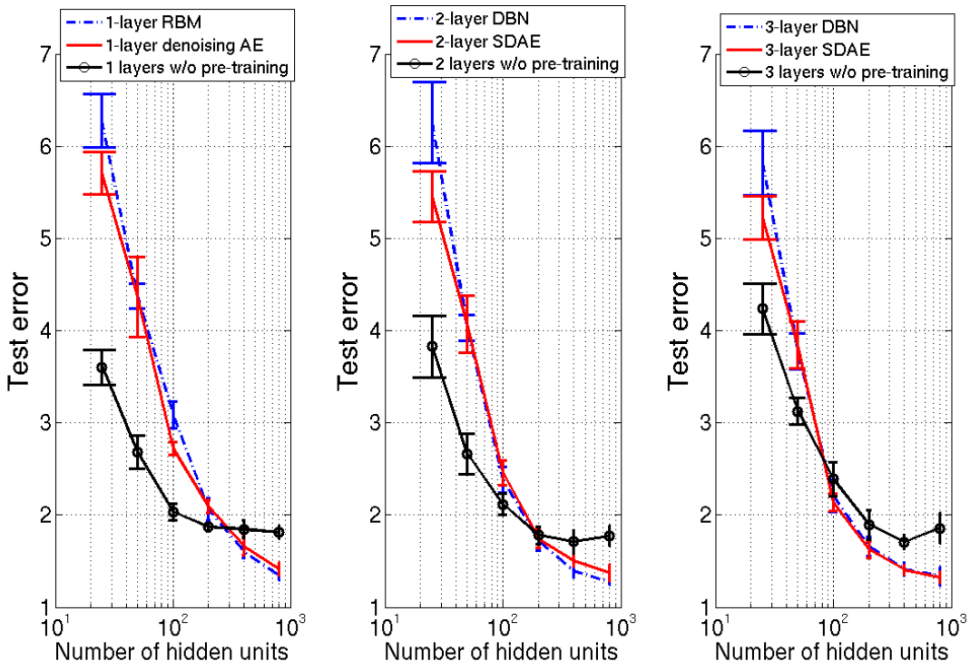

Figure 10: Effect of layer size on the changes brought by unsupervised pre-training, for networks with 1, 2 or 3 hidden layers on the MNIST dataset. Error bars have a height of two standard deviations (over initialization seed). Images taken from [1]

affect the training error: the former predicts that unsupervised pre-training should result in a lower training error, while the latter predicts the opposite. To ascertain the influence of these two possible explanatory factors, this experiment looks at the test cost (Negative Log Likelihood on test data) obtained as a function of the training cost, along the trajectory followed in parameter space by the optimization procedure. Figure 9 shows 400 of these curves started from a point in parameter space obtained from random initialization, that is, without pre-training (blue), and 400 started from pre-trained parameters (red) when the number of hidden layers were varied from 1-3. As the training progresses, the training error (and hopefully the test error) drops. So the curve proceeds from top-right end in the figures to the bottom-left.

It can be observed that: (a) Unsupervised pre-training reaches lower training cost than no pre-training for 1 layer DNNs, where it hints at better optimization; (b) At a same training cost level, the pre-trained models systematically yield a lower test cost than the randomly initialized ones. This hints at a better generalization and the regularization hypothesis.

Observation 5 (Influence of the Layer Size): Recall another well known property of regularizers - effectiveness of regularization increases as complexity of the model (number of hidden units, number of layers) increases. If the regularization hypothesis was to be true then we should see a trend of increasing effectiveness of unsupervised pre-training as the number of units per layer are increased. This is indeed the case in Figure 10. Figure 10 plots the generalisation error against the number of hidden units (varied from 10 to 1000)

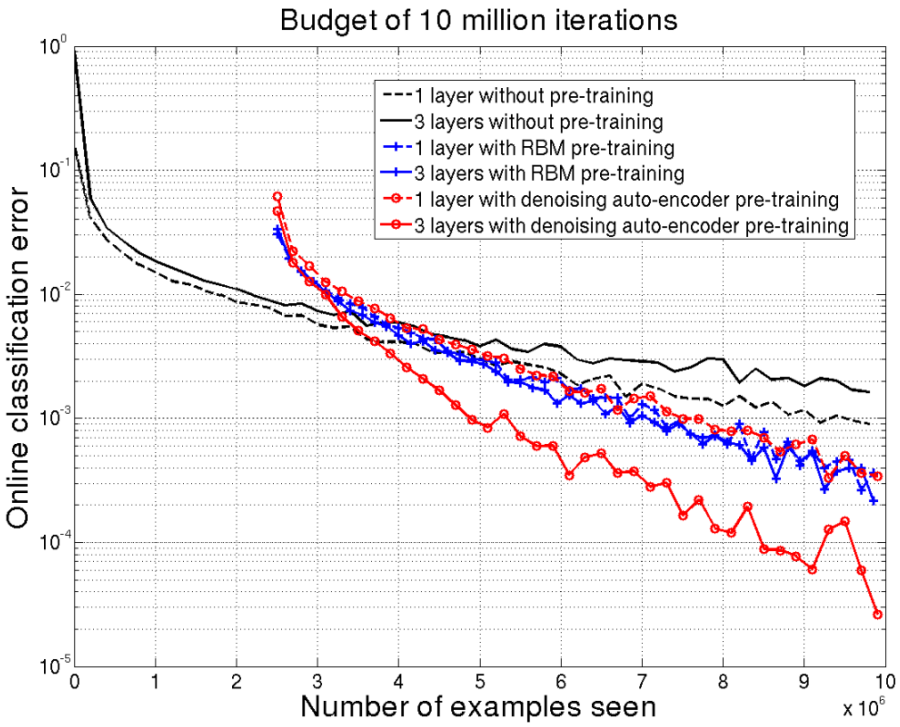

Figure 11: Comparison between 1 and 3-layer networks trained on the InfiniteMNIST dataset showing online classification error, computed as an average over a block of last 100,000 errors. Image taken from [1]

for 1-3 layer neural networks when pre-training (RBM pre-training and denoising auto-encoder pre-training) is used or not. A systematic effect is observed for all three cases: while unsupervised pre-training helps for larger layers and deeper networks, it also appears to hurt for small networks, hence, supporting the regularization hypothesis.

Observation 6 (Effect of Pre-training with Very Large Data Sets): To motivate these set of experiments, we recall another well known property of regularizers - the effectiveness of a canonical regularizer decreases as the data set grows. However, this stands in direct conflict with the optimization hypothesis. According to the optimization hypothesis, the early examples determine the basin of attraction for the remainder of training, and hence, the early examples have a disproportionate influence on the configuration of parameters of the trained models.

To further probe this, the paper plots the online classification error (on the next block of examples, as a moving average) for 6 architectures that are trained on InfiniteMNIST: 1 and 3-layer DBNs, 1 and 3-layer SDAE, as well as 1 and 3-layer networks without pre-training in Figure 11. Here, we can observe that (a) 3-layer networks without pre-training are worse at generalization, compared to the 1-layer equivalents and (b) more importantly, **the pre-training advantage does not vanish as the number of training examples increases**.

This is further supported in Figure 12 which plots the online classification error as the

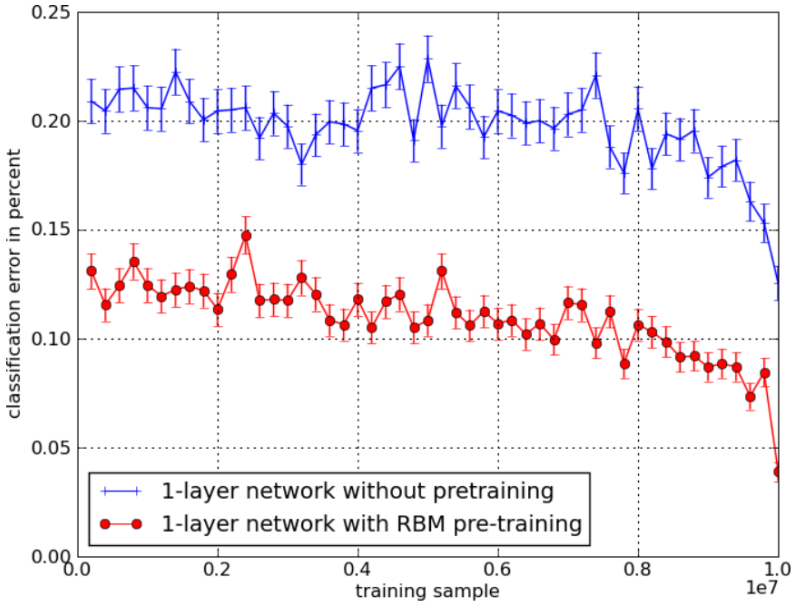

Figure 12: Error of 1-layer network with RBM pre-training and without, on the 10 million examples of the InfiniteMNIST dataset used for training it. Image taken from [1]

training sample size rises in relation to a fixed test set. It can be seen that the **pre-trained model is better across the board on the training set**. Both these results support an optimization effect and seem to disagree with the resularization hypothesis. In the face of this dichotomy, the authors claim that **while pre-training does have a regularization effect, unlike canonical regularizers (such as L1/L2), the effectiveness of unsupervised pre-training as a regularizer is maintained as the data set grows**.

Observation 7 (Effect of Example Ordering): In the previous experiment, we argued about the dependence of the basin of attraction on early data. The next experiment tests to what extent the outcome of the model is influenced by the examples seen at different points during training, and whether the early examples indeed have a stronger influence. The experiment quantifies the variance of the outcome with respect to training samples at different points during training, and compare these variances for models with and without pre-training. Figure 13 plots the variance of the output of the networks on a fixed test set. The **samples at the beginning seem to influence the output of the networks more than the ones at the end**. Moreover, this **variance is lower for the networks that have been pre-trained**. A caveat is that both networks also seem more influenced by the last examples used for optimization. This is simply a property of SGD with a constant learning rate - which gives more weight to recent datapoints.

Observation 8 (Pre-training only k layers): The next experiment explores the case when only the bottom k layers are pre-trained (top $n-k$ layers are randomly initialized) as opposed to all the layers. Figure 14 plots the outcome of this experiment for both MNIST and InfiniteMNIST datasets. The first plot (plot on the left) plots the trajectories (log(train NLL) vs. log(test NLL)). Again, the trajectories go roughly from the right to left and from top to bottom. Here, we see that **the final training error (after the same number of**

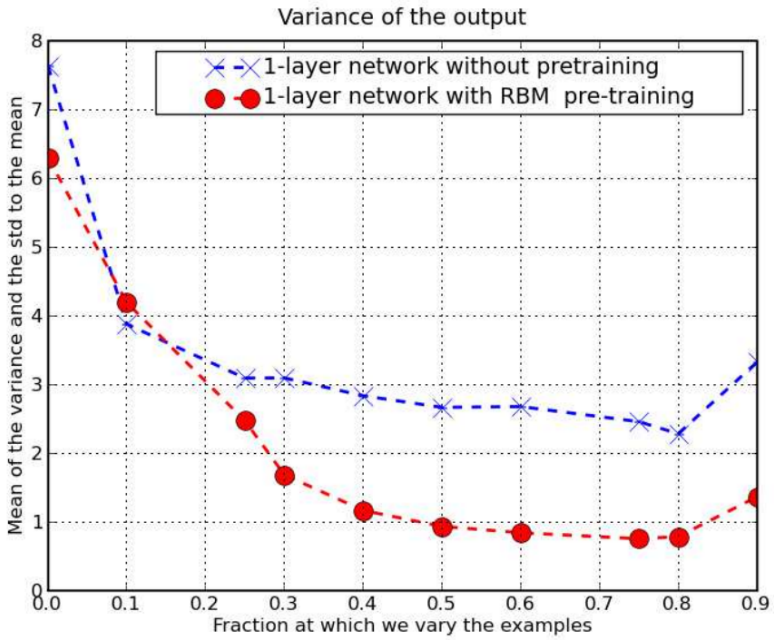

Figure 13: Variance of the output of a trained network with 1 layer. The variance is computed as a function of the point at which we vary the training samples. Image taken from [1]

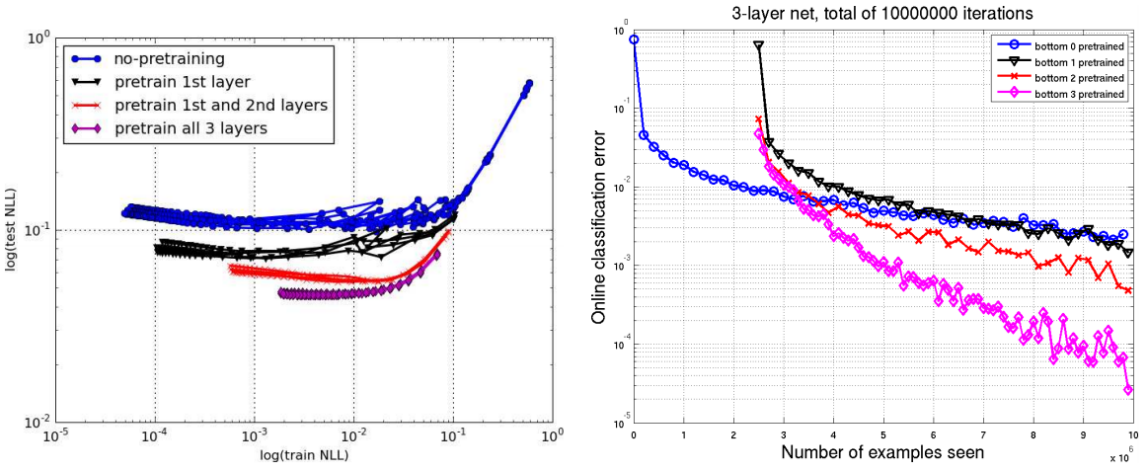

Figure 14: On the left: for MNIST, a plot of the log(train NLL) vs. log(test NLL) at each epoch of training. We pre-train the first layer, the first two layers and all three layers using RBMs and randomly initialize the other layers; we also compare with the network whose layers are all randomly initialized. On the right: InfiniteMNIST, the online classification error. We pre-train the first layer, the first two layers or all three layers using denoising auto-encoders and leave the rest of the network randomly initialized. Image taken from [1]

epochs) becomes worse with pre-training of more layers. For the second plot (plot on the right), online error is instead plotted on the InfiniteMNIST dataset. In both plots we see that **as we pre-train more layers, the models become better at generalization**. This too lends support to the regularization hypothesis.

2 Our Simulations

The simulations described in [1] are highly computationally expensive, requiring the fine-tuning of 6 hyperparameters and, according to the authors, "months of CPU time." We chose to perform a few smaller, more focused tests in our simulations.

In subsection 2.1, we confirm the original paper's claimed effects of layer size on training and test error, when running models to convergence, using 3-layer DNNs with and without pre-training via stacked denoising auto-encoders (SDAE). We also explore a comparison with dropout training. Subsection 2.2 explores the effect of the number of layers and computational budget when working with small layer sizes. Finally, subsection 2.3 returns to the same questions but with the goal of training auto-encoders themselves, rather than the predictive DNNs we have been training so far.

The simulations for subsection 2.1 use the Python library Pylearn2, which is built on top of another Python library Theano. The simulations in subsections 2.2 and 2.3 are based on the MATLAB code in the Stanford Deep Learning Tutorial [5].

2.1 How Do Layer Size and Dropout Affect 3-layer DNNs Run To Convergence?

Figure 15 shows convergence paths of the training and test negative log-likelihoods in our Pylearn2/Theano simulations, with and without pre-training. We trained DNNs with 3 hidden layers on the MNIST dataset. Pre-training used SDAEs. Each path (from several different random weight initializations) shows the negative log-likelihoods improving, from the upper right corner towards the lower left, both with and without pre-training.

The top plot shows DNNs with small layers of 64 hidden units each. As we expected from [1], pre-training is not effective with small hidden layers, so that both the final training error and the final test error tend to be higher with pre-training than without. Perhaps with such small hidden layers, the backpropagation algorithm may already be optimizing "well enough," while the layers may be too small for pre-training to encode a useful representation of the features.

The bottom plot shows larger DNNs with 500 units per hidden layer. Now we are in the range where pre-training can help, and we see the pattern from [1] suggesting that the pre-trained DNNs tend to have lower test error but higher training error. This is evidence for the regularization hypothesis (since pre-training generalizes better) and against the optimization hypothesis (since pre-training does not reduce training error).

Figure 16 shows convergence paths from the same experiment, with even larger DNNs of 1200 units per hidden layer, as well as a partial convergence path from the same DNN but with dropout (and no pre-training). Here we see that pre-training converges to lower training and test errors than without pre-training, which favors both the optimization and regularization hypotheses. The single incomplete run of dropout training, which did not

finish converging in the time allotted, suggests that dropout could eventually converge to a lower test error than the other two approaches.

2.2 How Does Number of Layers Affect Small-Layer DNNs With Fixed Convergence Budgets?

Figure 17 shows boxplots of the training and test error rates in our MATLAB simulations, with and without pre-training. Starting from 10 different random weight initializations, we trained DNNs with 1 to 4 small hidden layers, of 64 units each, on the MNIST dataset. Pre-training used SDAEs. Here, denoising means that when training each greedily-trained hidden layer, a random 10% of the inputs (data pixels or hidden units) to the layer were set to 0, while the unedited input was used as the output target.

In order to investigate the effects of computational budget and early stopping, we ran each initialization of each model for a specified convergence budget, in terms of iterations of batch gradient descent:

- Models marked "WithPretrain" ran for 50 iterations per greedily-trained hidden layer in the auto-encoder, plus another 50 iterations of backpropagation to finetune the final model, for a total of $50 \times (L + 1)$ iterations, where L is the number of hidden layers.
- For fair comparison with the total number of training iterations, the models marked "WithoutPretrainFull" ran backpropagation on the whole model (from random initial weights) for the same $50 \times (L + 1)$ total number of iterations.
- For fair comparison with the number of fine-tuning iterations (i.e. after the weights are either pre-trained or randomly initialized), the models marked "WithoutPretrain-EarlyStop" ran backpropagation on the whole model (from random initial weights) for only 50 total iterations.

The top plot of Figure 17 shows the training errors. In almost every case, "Without-PretrainFull" had enough iterations in its budget to converge fully to 0 training error. The model with pre-training had non-negligible training error that grew slowly with the number of layers, suggesting that it takes longer to converge with pre-training than without, even for the small layer sizes here. Finally, the models "WithoutPretrainEarlyStop" had training error that grew far more quickly (with number of layers) than for the models with pre-training. In fact, the models with pre-training had lower training error than "Without-PretrainEarlyStop" except for the very smallest (single-hidden-layer) setup. This suggests that pre-training does indeed help initialize the weights to better-than-random values for the training set, which supports the optimization hypothesis.

The bottom plot of Figure 17 shows the corresponding test errors. The patterns and conclusions are very similar to the training errors, except that the "WithoutPretrainFull" models no longer have negligible error.

This difference is made clearer in Figure 18, a scatterplot of the training and test errors with a line overlaid at $y = x$. Both early stopping and pre-training give values very close to the line, showing that their training and test errors are very close. This supports the regularization hypothesis: the performance on the training set is not overfit, and instead it

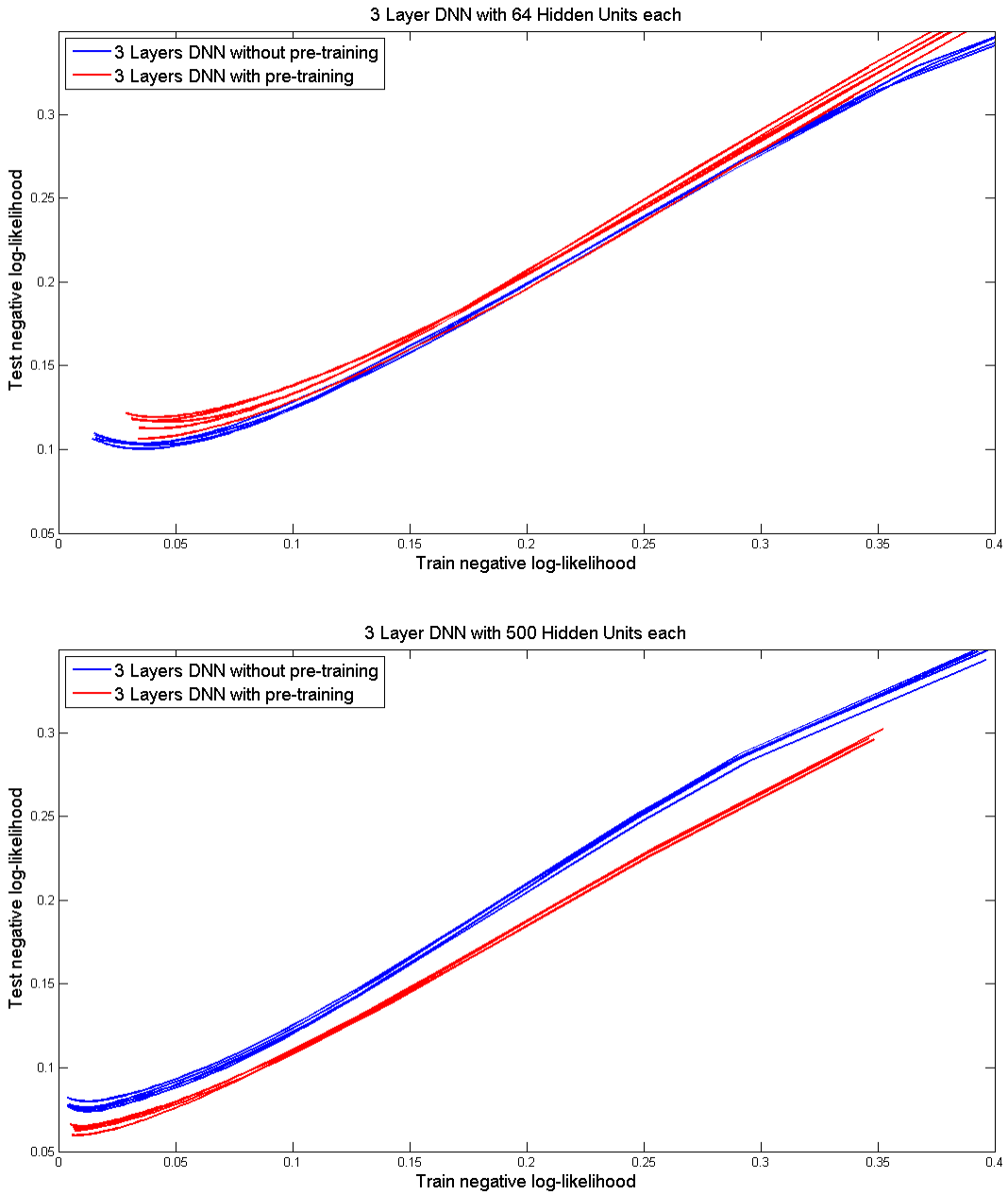

Figure 15: Convergence paths (starting at upper right and ending at lower left) of the training and test negative log-likelihoods from Pylearn2/Theano simulations on the MNIST dataset. 3-hidden-layer Deep Neural Networks, with either 64 (top) or 500 (bottom) units per hidden layer, were run until convergence. Pre-training used SDAEs.

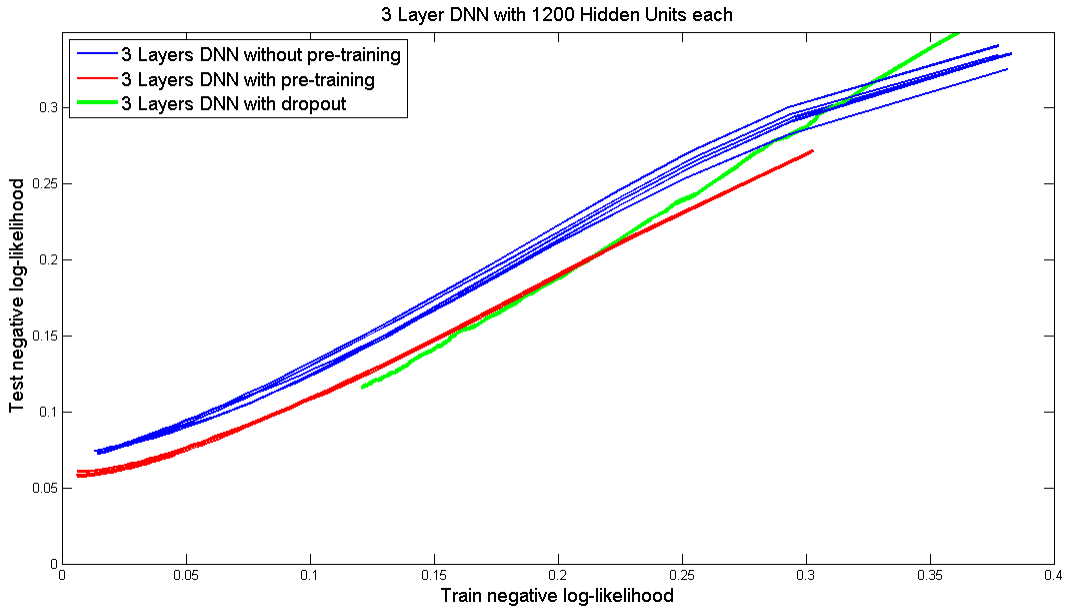

Figure 16: Convergence paths (starting at upper right and ending at lower left) of the training and test negative log-likelihoods from Pylearn2/Theano simulations on the MNIST dataset. 3-hidden-layer Deep Neural Networks with 1200 units per hidden layer were run to convergence, except for dropout, which did not converge in the time available. Pre-training used SDAEs.

is a good measure of the performance on future data. However, the "WithoutPretrainFull" model shows signs of overfitting: the training error is near 0, but the test error is far from the line.

2.3 How Does Number of Layers Affect Small-Layer Auto-Encoders With Fixed Convergence Budgets?

Our final test compares auto-encoders with and without pre-training. Here our pre-training is done just as before, except that we do not fine-tune after pre-training to predict the label (the digit which that MNIST image represents); instead, we fine-tune to predict the image itself again. Without pre-training, we fit the full model at once with backpropagation to predict the MNIST image from its own input.

(Note: neither the auto-encoders with nor without pre-training used denoising in this experiment.)

As in the previous simulation, the pre-training runs for 50 iterations per greedily-trained hidden layer, followed by another 50 iterations for finetuning of the whole model. Without pre-training, the auto-encoder has the "full budget" of $50 \times (L+1)$ total iterations starting from random initial weights. With no classification error to report, instead we report the root mean square error (RMSE) of the predictions: compute the squared error at each pixel, average over pixels, and take the square root.

In the top plot of Figure 19 we see the training RMSEs. The models with pre-training have a training error that rises with the number of hidden layers, but more and more slowly as the number of layers grows. This suggests a kind of regularization again: the pre-trained auto-encoder does not get dramatically worse as the number of parameters in the model grows. On the other hand, the model without pre-training begins with a lower RMSE for the single-hidden-layer network, but catches up to and even surpasses the pre-trained model's RMSE for the 4-hidden-layer network. This suggests that an auto-encoder with backpropagation alone is not enough: greedy layer-wise pre-training provides some additional benefit, at least for larger models and for this kind of convergence budget.

The test RMSEs, in the bottom plot of Figure 19, look almost identical and provide the same conclusions.

However, it may be unfair to compare this auto-encoder without pre-training to a general DNN without pre-training (as in previous sections), since the auto-encoder's model is so much larger than the other DNN's. In the latter, the final set of weights only takes each hidden unit to 10 output nodes (one per MNIST digit). In the auto-encoder, each hidden unit connects to $28^2 = 784$ nodes for the 28-by-28 pixelated images, leading to far more parameters in the final weight vector.

Finally, it may seem surprising that the single-hidden-layer auto-encoder without pre-training had much lower RMSEs than with pre-training. Most likely, this happens because of the two-step nature of the version with pre-training: First, we train the single-hidden-layer auto-encoder for 50 iterations. Second, we discard the top set of weights (from the hidden layer to the output layer), replace them with random initial weights, and finetune for a final 50 iterations. Those top-level weights must be discarded for pre-training other DNNs, but actually would have been useful here, so this two-step process puts pre-training at a disadvantage when the goal is to train a 1-hidden-layer auto-encoder.

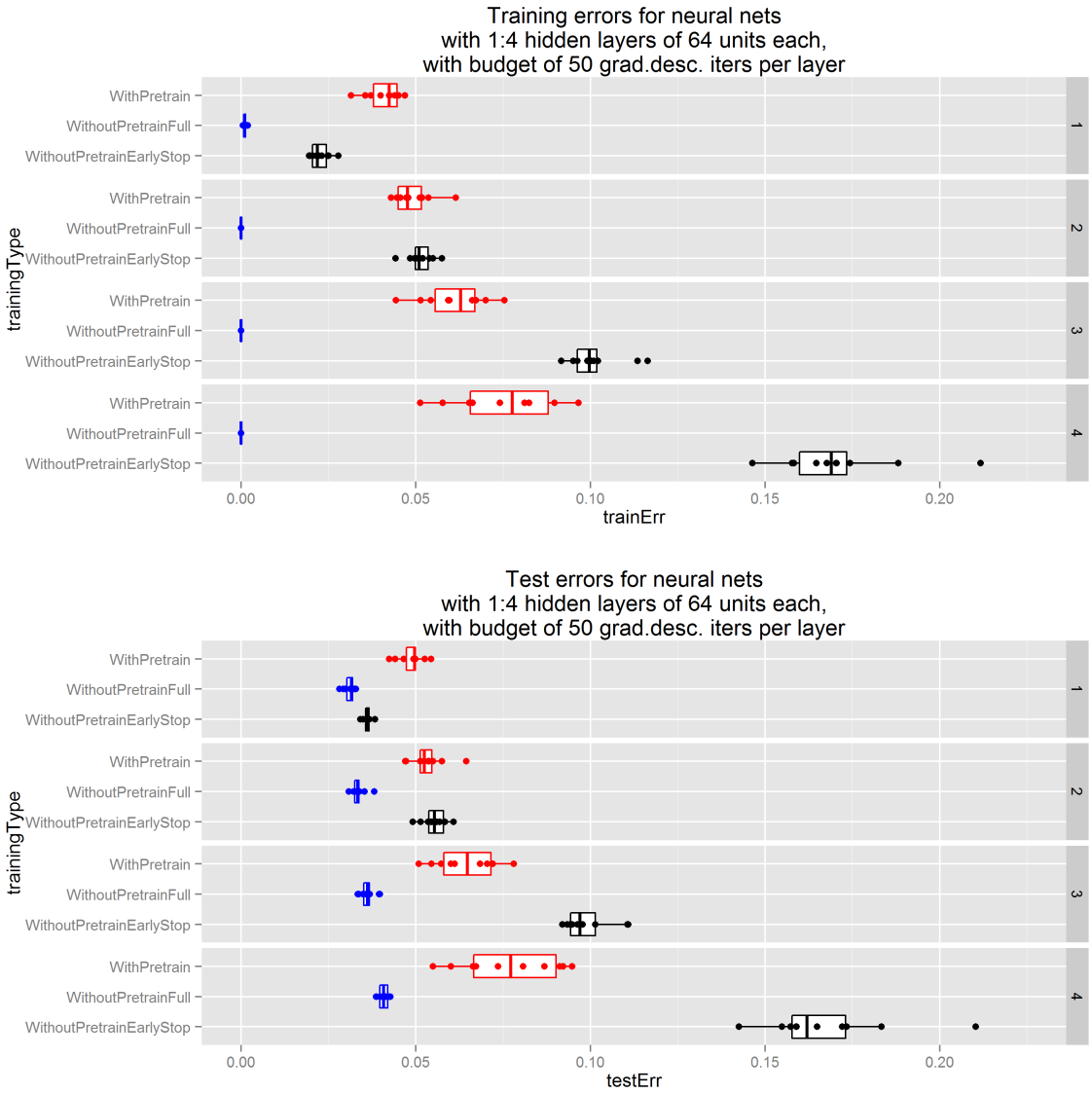

Figure 17: Boxplots of training and test errors from MATLAB simulations on the MNIST dataset. 1- to 4-hidden-layer Deep Neural Networks, with 64 units per hidden layer, were trained for a specified number of batch gradient descent iterations. Pre-training used SDAEs.

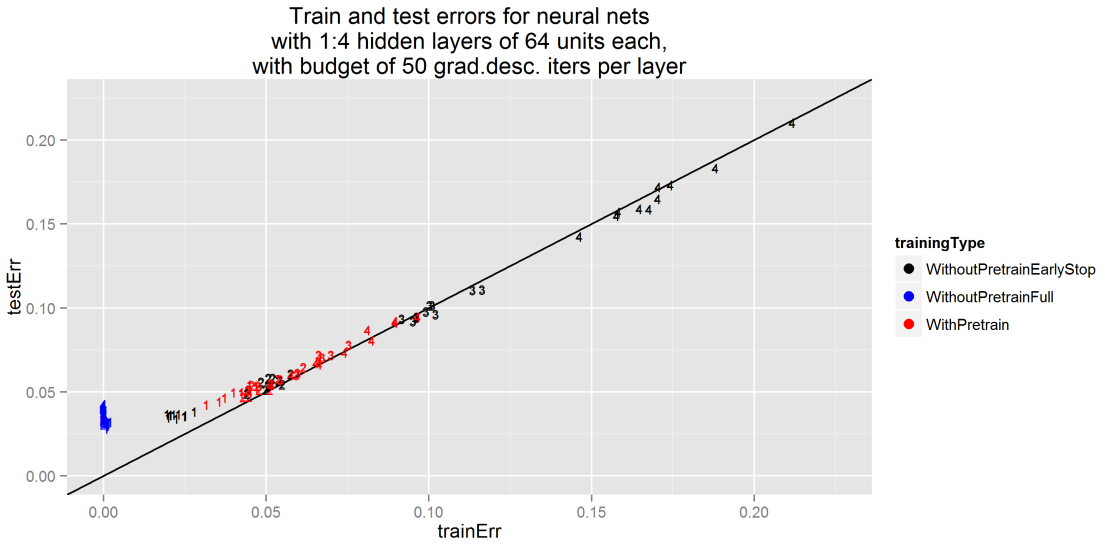

Figure 18: Scatterplot of training vs. test errors from MATLAB simulations on the MNIST dataset. 1- to 4-hidden-layer Deep Neural Networks, with 64 units per hidden layer, were trained for a specified number of batch gradient descent iterations. Pre-training used SDAEs. Each point is a number, indicating the number of hidden layers used for that run.

3 Discussion

Class discussion, on Deep Learning in general:

- **Is "Deep Learning" the same thing as "a Deep Neural Network (DNN) with unsupervised pre-training"?** There is an overlap, since the specified model is a common type of DL model. But the term DL also covers other deep architectures, such as Deep Belief Nets (DBNs), which are made of stacked Restricted Boltzmann Machines (RMBs) instead.
- **If your auto-encoder reduces dimensionality by having few hidden units, do you still benefit from denoising too?** Yes: denoising may help to **smooth** the functions that you are learning, while the bottleneck (having few hidden units) adds **sparsity**. Think of doing PCA with an L_2 penalty—each part gives you a different benefit.
- **Even the worst non-pre-trained network's performance on the MNIST data was really good, with accuracy in the high 90%'s, so there's not much room for pre-training to show dramatic improvement. How does Deep Learning do on harder problems?** Good question, and worth exploring given more time!
- **Why do people seem to be more focused on Deep Learning than on using kernels for new feature representations?** It may be easier to specify aspects of the model such as sparsity with an explicit deep network (with sparse connections at

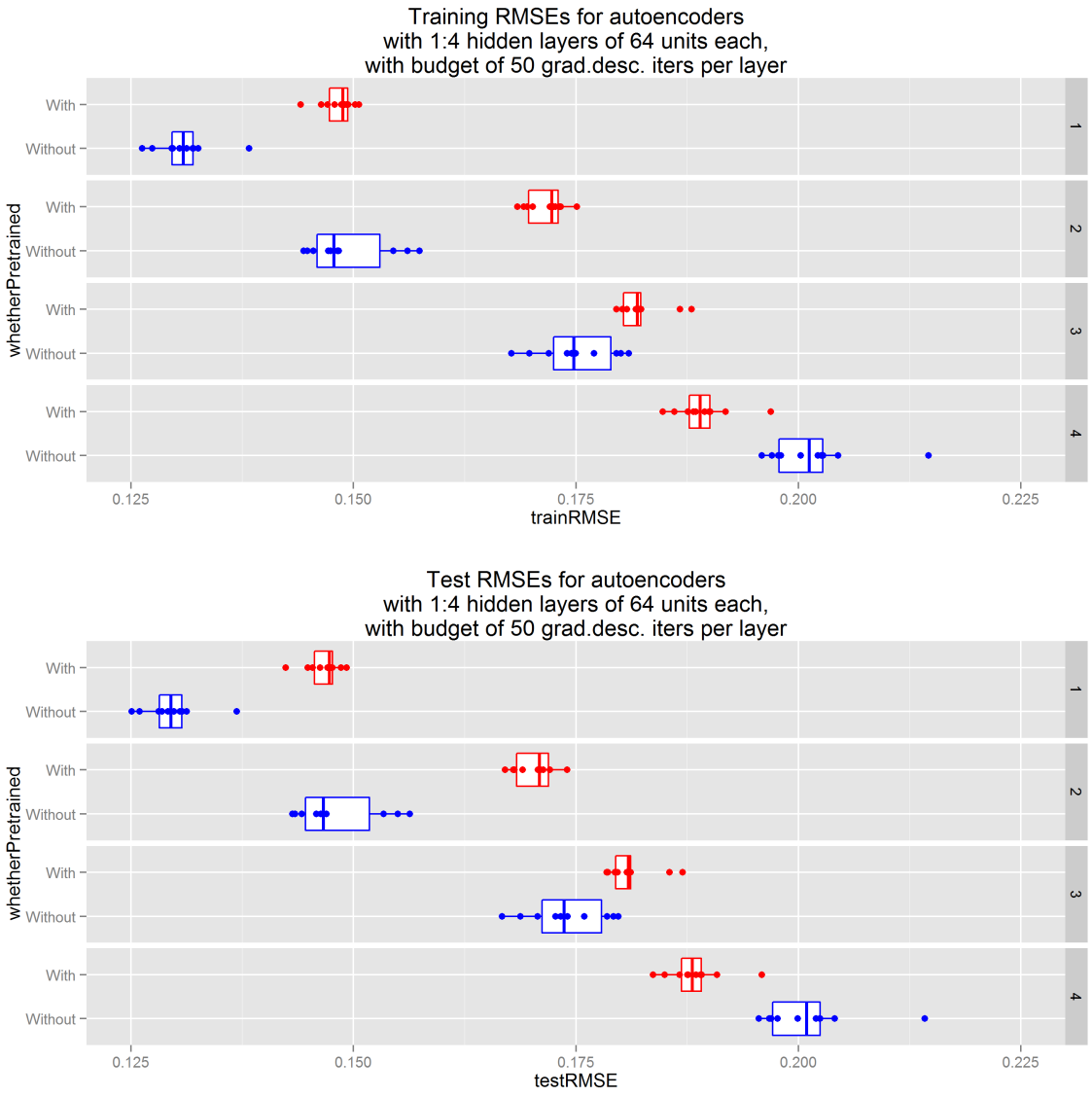

Figure 19: Boxplots of training and test RMSEs from MATLAB simulations on the MNIST dataset. 1- to 4-hidden-layer DNN auto-encoders, with 64 units per hidden layer, were trained for a specified number of batch gradient descent iterations. Pre-training used stacked auto-encoders (but no denoising).

each layer) than with a kernel representation. Kernels may also be harder to scale up. However, there is current research into "deep kernels" too.

- **Why does it seem Deep Learning is used mostly for image data? Why not for non-image data, like genetics?** There's no reason you couldn't use it for genetics too. But the auto-encoders are easier to check and visualize for image data: you can make those nice plots of the "image filters" that the weights correspond to, and inspect them visually, and print them in your NIPS paper. You can do the same with audio data, finding the sounds or pitches or rhythms that your hidden units correspond to. But it probably wouldn't look like anything meaningful with genetics data. Convolutional neural nets (which we didn't discuss today) are also nice with image data, since your network could have hidden layers that are e.g. edge detectors, which then get convolved around the image to build up higher-level representations... and again, there may not be an obvious equivalent interpretation with genetics data.

Class discussion, on the loss function and regularization vs. optimization hypotheses:

- **If pretraining is a regularizer, doesn't it actually change the shape of the loss function somehow?** Yes, in a way, but it's not an explicit penalty that you can simply add to the cost function and then optimize the sum. It may help to think about priors over the unmodified cost function instead.

- **We talk about pre-training as a "prior" on the basins of attraction. But what if you used a more aggressive line search that could jump from one basin of attraction to other?** Then the "prior" is clearly dependent on the optimization algorithm you use. Indeed, Deep Learning seems to be inherently intertwined with decisions about how to perform the optimization, when to stop convergence, etc.

- **Does pre-training somehow make the basins of attraction steeper, to make it harder to leave a basin?** Perhaps—that may be why the pre-trained results seems to have far less variability (over many random initializations) than without pre-training.

Class discussion, on the simulations:

- **Where does randomness come in to the simulations? Isn't backpropagation etc. deterministic?** It's a deterministic optimization, but we start with randomly initialized weights. Also, the denoising step in the pre-training involves adding random noise to the inputs.

- **How and when did the authors of [1] stop the convergence? Did they use the same number of runs for all numbers of layers, or more for larger networks, etc.?** This was unclear in the paper. But they seem to have optimized hyperparameters for each setting (by cross-validation), so that at a given layer size, number of layers, etc., the best pre-trained network is compared with the best non-pre-trained network.

- **How did we stop the convergence in our own simulations? How did we avoid overfitting?** We did not have time to cross-validate and find optimal hyper-parameters. For the Pylearn2/Theano simulations, we ran each model until it hit 0 training error, or until the change (over iterations) in negative train log-likelihood fell below a threshold. For the MATLAB simulations, we specified a fixed budget in terms of the number of batch gradient descent iterations.

- **Why did dropout converge more slowly than the neural nets?** Good question—we are not sure.

- **How much time (in minutes, not iterations) did our simulations take to train?** Without pre-training, the MATLAB simulations ran a whole network in around 2 minutes per hidden layer. With pre-training, the first auto-encoder layer took maybe 5 minutes (since the input and output are both huge, with 768 units each), but the hidden layers' auto-encoders were much faster (with just 64 units in and out), around half a minute, and then the fine-tuning took another 2 minutes.

Disclaimer: Some portions of the content (figures and summary) has been taken/paraphrased from the original paper [1]. We do not claim originality of the scribe.

References

[1] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.

[2] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. Technical Report 1341, University of Montreal, jun 2009. Also presented at the ICML 2009 Workshop on Learning Feature Hierarchies, Montréal, Canada.

[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Gaëlle Loosli, Stéphane Canu, and Léon Bottou. Training invariant support vector machines using selective sampling. *Large Scale Kernel Machines*, pages 2278–2324, 2007.

[5] Andrew Ng. Unsupervised feature learning and deep learning tutorial. `http://deeplearning.stanford.edu/tutorial/`. Accessed: 2014-12-03.