NOVEL GREEDY DEEP LEARNING ALGORITHMS

By

Ke Wu

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:

Malik Magdon-Ismail, Thesis Adviser

Elliot Anshelevich, Member

Mohammed Zaki, Member

Rensselaer Polytechnic Institute
Troy, New York

November 2015
(For Graduation December 2015)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

This thesis becomes a reality with the support and help of many individuals. I would like to extend my sincere thanks to them.

I would like to express my special gratitude to my adviser, Dr. Magdon-Ismail for imparting his knowledge and expertise in this study. I really enjoyed discussing the problems with him.

I also thank Dr. Anshelevich and Dr. Zaki for being my committee member and providing valuable suggestions.

I am highly indebted to Dr. Breneman for encouraging and supporting me to pursue this degree, which has opened a new world for my future.

I also thank my girlfriend Wei Zou and my friend Zheqin Yang for supporting and accompanying me through the days in RPI.

ABSTRACT

Multilayer neural networks have seen a resurgence under the umbrella of deep learning. Current deep learning algorithms train the layers of the network sequentially, improving the algorithmic performance as well as providing some regularization. While the current algorithms have shown great prediction power in many problems, disadvantages in model interpretability and training time still exist and may not be easily overcome under the current framework. In order to solve the problems, we developed two new training algorithms for deep networks which train *each node in the network* sequentially.

The new algorithms are designed to simulate human's learning process, where features for different objects are identified, understood and memorized iteratively. The main difference between the two new algorithms is the partition function used to split the input data into subsets. A certain feature will be learned from one subset, instead of from the whole data set. The Greedy-By-Node (GN) algorithm is based on an additive-feature assumption which to some extent resembles the boosting algorithms, where the input data is sorted and partitioned based on their distance to the most common feature learned by the first inner node. The subsets closer to the common feature will be learned earlier, while harder problems are intrinsically covered by more inner nodes and learned at later stage. The Greedy-By-Class-By-Node (GCN) algorithm directly utilizes the data labels and assumes that data in each class share common features. A special cache mechanism and a parameter called "amnesia factor" are also introduced in order to keep the speed while provide control over the "orthogonality" between learned features. Our algorithms are orders of magnitude faster in training, create more interpretable internal representations at the node level, while not sacrificing on the ultimate out-of-sample performance.

1. INTRODUCTION

1.1 Multilayer Neural Networks

Multilayer neural networks have gone through ups and downs since their arrival in [1–3]. The current resurgence in the "deep" multilayer network owes itself largely to the efficient greedy layer by layer algorithms for training, as well as the ability to create hierarchical representations of the data within each layer. In the era of "big data" applications in diverse application areas, these two concerns are real: the deep network should be trained quickly; the internal layers should contain meaningful representations of the data to provide some insight into what complex features the nonlinear neurons are capturing. We explore these two dimensions of training a deep network. Assume a standard machine learning from data setup [4], with N datapoints $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ representing the task to be learned; $\mathbf{x}_n \in \mathbb{R}^d$ and $y_n \in \{0, 1, \ldots, c-1\}$ (multi-class setting).

We refer to [4, e-Chapter 7] for the basics of multilayer neural networks, including notation which we very quickly summarize here. In figure 1.1 we show an example feedforward neural network architecture. Such a network is considered "deep" because it has many ($\gg 2$) layers. Throughout this thesis, we assume that a neural network architecture as shown in Figure 1.1 has been fixed. The neural network implements a function whereby in each layer (ℓ), the output of the previous layer ($\ell - 1$) is transformed into the output of the layer ℓ until one reaches the final layer on top, which is the output of the network. The function implemented by layer ℓ is

$$\mathbf{x}^{(\ell)} = \tanh(\mathrm{W}^{(\ell)}\mathbf{x}^{(\ell-1)}),$$

where $\mathbf{x}^{(\ell)}$ is the output of layer ℓ, and the matrix of weights $\mathrm{W}^{(\ell)}$ (having the appropriate dimensions to map a vector from layer $\ell - 1$ to a vector in ℓ) is a set of parameters that have to be learned from the data. Using the data to identify all the parameters $\{\mathrm{W}^{(1)}, \mathrm{W}^{(2)}, \ldots, \mathrm{W}^{(L)}\}$ is the training phase of the neural network.

The backpropagation approach to training a multilayer network [5], which

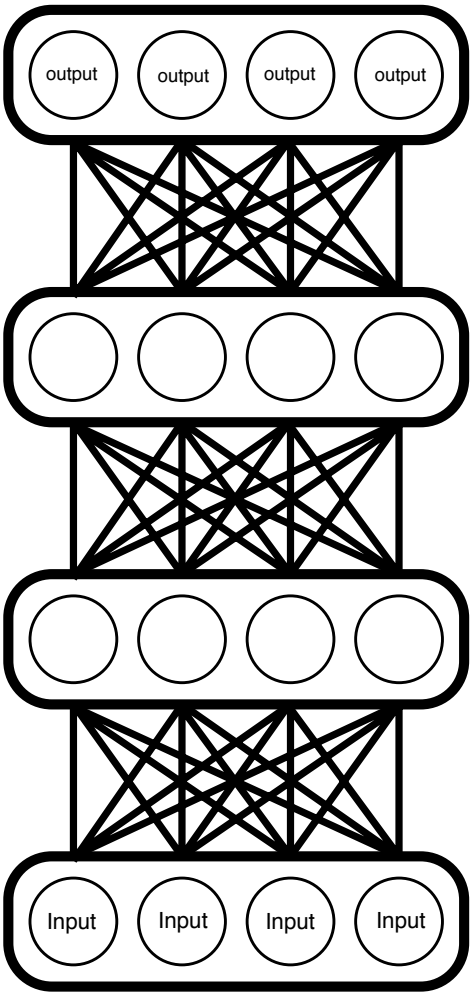

Figure 1.1: Multilayer Neural Network.

was the popular approach to training a neural network, trains all the weights in the network simultaneously, allowing the network maximum flexibility. The current approach to training deep networks is layer-by-layer: train the first layer weights $W^{(1)}$; now train the second layer weights $W^{(2)}$, *while keeping the first layer weights fixed*; and so on until all the weights have been determined. In practice, once all the weights have been learned in the "greedy" layer by layer manner (often referred to as pre-training), the best results are obtained by fine tuning all the weights using a few iterations of backpropagation. This process is illustrated in Figure 1.2.

A question arises as to how one should train each internal layer. There are two dominant approaches with very similar ideas. The first is as an unsupervised nonlinear auto-encoder of the data [6]; this approach is appealing to build meaningful

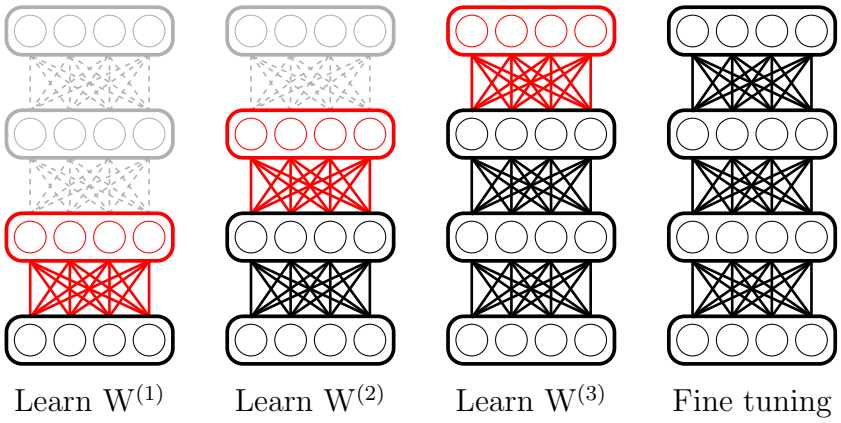

Figure 1.2: Layer-by-layer greedy deep learning algorithm.

hierarchical representations of the data in the internal layers. The second is as a supervised encoder; this approach primarily targets prediction performance. Deep learning has enjoyed great success in several areas and hence considerable effort has been expended in optimizing the pre-training. The two main considerations are:

1. Pre-training time and training/test performance of the final solution. There is no doubt that the greedy layer-by-layer pre-training is significantly more efficient than full backpropagation, and appears to be better at avoiding bad local minima [7]. Our algorithms will show an order of magnitude speed gain over greedy layer-by-layer pre-training.
2. Interpretability of the representations learned in the internal layers. From now on, we will use the USPS digits data (10 classes) as a strawman benchmark to illustrate our approach. For this strawman dataset, we may look at the internal representation learned by the first layer of the deep network from the lower layer weights going into a certain node, since the values of the weights to the corresponding pixel locations can be seen as the high-level features built from the lower level. Figure 1.3 shows the result for the unsupervised auto-encoder as well as the supervised encoder. It is not clear what features this internal representation is capturing.

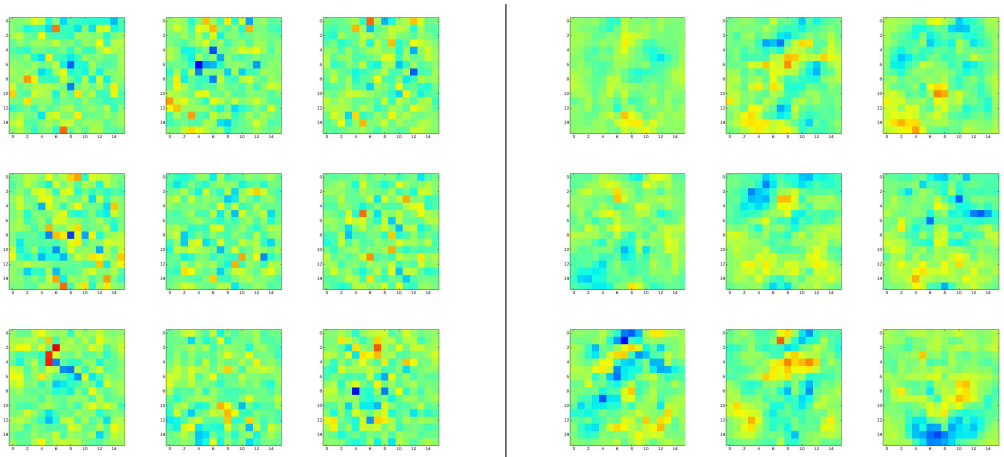

Figure 1.3: Features of nine nodes in the first internal layer after greedy layer-by-layer pre-training. There is no clear resemblence to digits, though the supervised features look slightly better than the unsupervised.

1.2 Our Contribution: Greedy Node-by-Node Pre-Training

The thrust of our approach is to learn the weights into each *node* of the network in a sequential greedy manner: *greedy-by-node* (GN) for the unsupervised version and *greedy-by-class-by-node* (GCN) for the supervised version. Figure 1.4 illustrates the first 5 steps for the network above. The motivation for this approach is to mimic

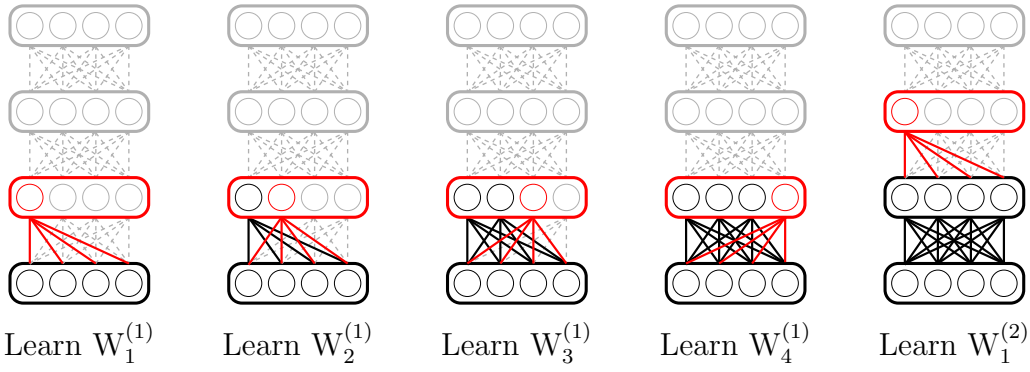

Figure 1.4: Node-by-node greedy deep learning algorithm.

a human learner who hardly builds all features (outputs of the internal nodes) at once from all objects. Instead, features are built one by one while processing the data in a sequential manner. Our algorithm learns the features in each layer one-by-one, using a part of the data to learn each feature. Our contributions are

1. The specific algorithm to train each internal node.
2. How to select the data with which to train each internal node.

The goal of this study is *not* to improve the accuracy of deep learning, rather, to improve efficiency and interpretability of the features, while *maintaining* the accuracy of the deep learning. The standard deep learning algorithm uses every data point to process every weight in the network. In our algorithm, only a subset of the data processes a particular weight, which defines a higher-level feature. Further, by training node-by-node using "relevent" data, our algorithm produces features that are more interpretable. Figure 1.5 shows the features of the first nine nodes of the first internal layer on the digit classification task. Compared with the features in Figure 1.3, our algorithm picks up visually more intuitive features. Our algorithms can be run using the unsupervised auto-encoder setting or the supervised setting.

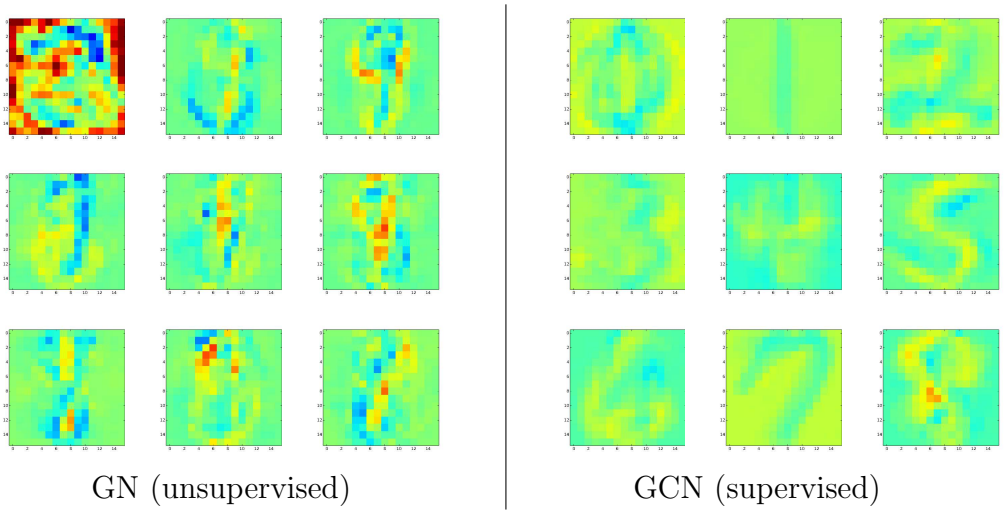

Figure 1.5: Features of nine nodes in the first internal layer after greedy node-by-node pre-training. The resemblence to digits is clear. The supervised features look slightly better than the unsupervised.

1.3 Related Work

To help motivate our approach, it helps to start back at the very begining of neural networks, with Rosenblatt [1] and Widrow et al. [2]. They introduced the adaline, the adaptive linear (hard threshold element), and the combination of

multiple elements came with Hoff [3], the madaline (the precursor to the multi-layer perceptron). Things cooled off a little because fitting data with multiple hard threshold elements was a combinatorial nightmare. There is no doubt softening the hard threshold to a sigmoid and the arrival of a new efficient training algorithm, backpropagation [5], was a huge part of the resurgence of neural networks in the 1980s/1990s. But, again, neural networks receeded, taking a back seat modern techniques like the support vector machine [8]. In part, this was due to the facts that multilayer feedforward networks were still hard to train iteratively due to convergence issues, suffered extensively from local minima [9, 10], and are extremely powerful [11] and hence easy to overfit to data. For these reasons, and despite the complexity theoretic advantages of deep networks (see for example the short discussion in [12]), application of neural networks was limited mostly to shallow two layer networks. Multi-layer (deep) neural networks are back in the guise of deep learning/deep networks, and again because of a leap in the methods used to train the network [13]. In a nutshell, rather that address the full problem of learning the weights in the network all at once, train each layer of the newtork sequentially. In so doing, training becomes manageable [12, 13], the local minima problem when training a single layer is significantly diminished as compared to the whole network and the restriction to layer by layer learning reigns in the power of the network, helping with regularizing it [7]. A side benefit has also emerged, which is that each layer succesively has the potential to learn hierarchical representations [7, 14]. As a result of these algorithmic advances, deep networks have found a host of modern applications, ranging from sentiment classification [15], to audio [16], to signal and information processing [17], to speech [18], and even to the unsupervised and transfer settings [19]. Optimization of such deep networks is also an active area, for example [20, 21].

Studies have been focused on generating better representations of the input data. Besides the original deep belief network [13] and autoencoder [12], the stacked denoising autoencoder [22, 23] has been widely used as a variant to the classic autoencoder. The idea is to use a corrupted input data for pre-training to reconstruct the original one. The resulted autoencoder can be seen as a way to define a man-

ifold [22]. Sparse coding [24, 25] has been developed to learn features effectively and prevent system from activating same subset of nodes constantly [25]. It has been shown that when applying on small image patches, this method is able to learn local and meaningful features. However the training requires extra steps and may be slower than classical ones. In learning meaningful features in images, convolutional neural network [26] and the deep convolutional neural network [27] are widely used, however the computation cost and complexity are often much higher than autoencoder and the feature learning is based on the local filter defined by the modeler. Since we are trying to explore more general methods, these techniques are not focused. In addition, we also find that our methods are conceptually similar to clustering-autoencoder systems, because under some settings, the whole algorithm can be roughly seen as a clustering method followed by and even embedded in a deep neural network. A few related studies have been reported. Ref [28] is focused on utilizing the deep network as a support for unsupervised clustering. Ref [29] proposes a method that an unsupervised learning algorithm is embedded in a classic deep network, targeted for semi-supervised learning. Ref [30] uses K-means clustering to preprocess the data and then use the samples from each cluster as the input to train a deep network. This method trades off the information from unselected training set for lower computation time, which may not be widely applicable since labeled data are often scarce and valuable. Besides, while K-means clustering may have advantages in large-scale system, it has been pointed out that it may not be an effective way to learn good representations [31].

In this thesis, we are proposing a new algorithmic enhancement to the deep network which is to consider each node separately. It not only explicitly achieves the sparsity of node activation but also requires much shorter computation time. We have found no such approaches in the literature.

2. METHODS

2.1 Greedy Node-by-Node Deep Learning Algorithms

The basic step of the two new algorithms for pre-training the network is to train a two layer network as illustrated in the Figure 2.1. The red (middle) layer is being trained (assume it has dimension d_2); the inputs to this layer come from the outputs of the previous layer, having dimension d_1. The dimension of the output layer is also d_1, following the settings of autoencoder. We use linear output-nodes (the algorithm is easy to adapt to a sigmoid output-nodes) and stochastic gradient descent (SGD) for the optimization of the pre-training.

As mentioned in previous section, the standard layer-by-layer algorithm trains all the weights at the same time, using the whole data set. The idea of our algorithms is to only use a fraction of data to learn one feature at a time from the inputs. In order to do that, the training of each layer is done within multiple stages. At each stage, we only update the weights corresponding to one inner node using a similar process as autoencoder–the weights are trained in order to reproduce the input data and the loss are measured by Euclidean distance between the input and the output. After all the features are obtained and all the weights learned, a full forward propagation will be applied to the whole data set just as the standard pre-training does. Then the same procedure will be done for the next layer. To make

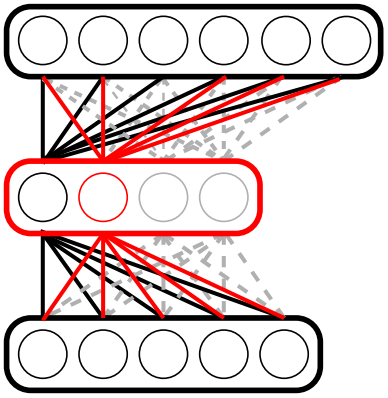

Figure 2.1: Node-by-node training of a basic network.

this greedy learning algorithm work, three questions need to be answered: 1. how to learn features iteratively from fractions of training data? 2. how to partition the training data? 3. how to control the orthogonality of the learned features?

To answer the first question, suppose we have already got multiple fractions of data of size S_{sub}, naive ideas can be using each fraction for each inner node independently. This idea is close to using the medoids as the representations for each fraction, because when using a single inner node for autoencoder, medoids are the global minimum in terms of the Euclidean distance(in practice, it may instead learn the first principal component, see next chapter). This idea totally ignores the information provided by other data and it is very likely the medoids for each fraction are numerically similar (for example, a large portion of background), so all the nodes may end up with learning similar features, which will deteriorates the discriminative power of the model.

To solve the issue mentioned above, one solution seems to be using the first fraction to train the first node and then using the second fraction on the first two nodes (for forward propagation) but only updating corresponding weights for the second node. In this way, the new feature learned relies on the previous knowledge, thus it can learn the generative representations similar to the standard pre-training algorithms. However, since for stage i the forward propagation is applied on all the current i ($\leq d_2$) nodes, the total number of numerical operations in forward propagations turns out to be $\sum_{i=1}^{d_2} i = O(d_2^2)$, which makes the computation much slower, since the standard algorithms only have $O(d_2)$ operations.

Our method utilizes the linearity of the transformation from the inner layer to the output layer. Suppose $\boldsymbol{W}$ is the weight matrix between the inner layer to the output layer with dimension $d_1 d_2$ (ignoring the bias node for the inner layer), and the outcome of the d_2 inner nodes is a vector v of length d_2, the output after forward propagation is then $\boldsymbol{W}v$. From basic linear algebra one can find that $\boldsymbol{W}v = \sum_{j=1}^{d_2} W_{\cdot j} v_j$, where $W_{\cdot j}$ are the weights coming out of the inner node j and v_j is the outcome value of this node (v_j is a scalar value). This equation shows that the contributions from the inner nodes to the output layer are independent. Assuming the output layer uses a linear activation function (identity function), the

output of the network can be seen as the summation of all the d_1 dimension vectors generated from each inner node and in backpropagation they are constrained by the summation, since the loss function is the Euclidean distance between the output and the target. Consequently, it is not necessary to forward-propagate through all the first i nodes at stage i, as the solution mentioned early. One can store the output layer from stage $i-1$ and then add it to the new output only through the node i to get the same result. This process is equivalent to propagate through i nodes but only update the weights for the newest one. In this way, for each stage, we only need to propagate through one inner node, which results in an overall number of operations of $O(d_2)$.

For the second question, we here propose two methods corresponding to the two new algorithms GN and GCN. For GN, we use the first node to learn the overall feature (ideally should be close to the first principal component of all data), then sort all the data (with a total number of N) based on Euclidean distances to the overall feature. For rest the d_2-1 nodes, each node will be used to learn the features of $N/(d_2-1)$ data in an increasing order (data with smaller distance will be used first). In this way, difficult cases are intrinsically covered by more inner nodes. For GCN, we directly use the class labels to partition the data. Suppose there are n_k data with the label k, for the corresponding nodes, each will use cn_k/d_2 samples, where c is the number of classes. In this method, each class has d_2/c partitions. The learning process first iterates all the classes for the first partition, then starts over for the rest of the partitions.

For the third question, in order to incorporate the information learned by previous nodes, the output (reconstruction from the output layer) from previous $i-1$ nodes is stored and added to the result from the forward propagation through the current node. Since back propagation is applied based on the distance between the input and the summed output, the stored information can be seen as constraints to the training for node i, which forces the newly learned feature to be orthogonal to the previous ones. This process is different from the standard pre-training because at the beginning of stage i, the $i-1$ nodes were already trained to encode the input information, but for the standard pre-training, only the combination of all d_2 nodes

can be used to reconstruct the input. The premature of the learned weights from the new algorithms may lead to the over-saturation of the output layer and eventually result in abnormal features and numerical instability. To mitigate this issue, instead of directly adding the stored output to the new output for back propagation, we introduce a new parameter called amnesia factor (AF). The usage of this factor is closely related to the first question on iterative learning. The stored output is first multiplied by an amnesia factor and then added to the current output. The AF actually provides control on the "orthogonality" between new feature and old features. A high amnesia factor means a strong influence from previous learned features. A value of zero means no memorization from previous learning at all, which is equivalent to the idea of using medoids or first principal component in the first question.

2.2 Algorithm Analysis

The basic step for classic pre-training algorithm is to train a two layer network as illustrated in Figure 1.1. The network is trained to reproduce the input (unsupervised) or the final target (supervised). The two algorithms are very similar in structure. For the conciseness of discussion, we here use the unsupervised pre-training (auto-encoder) as an example for analysis.

In a classic auto-coder with one hidden layer and trained using stochastic gradient descent (SGD), the number of operations (fundamental arithmetic operations) p for one data entry required by training process, with activation function implemented in inner layer, can be computed as follows:

$$\begin{aligned} p &= ((d_1+1)*d_2 + d2 + (d_2+1)*d_1)(\text{Forward Propagation}) \\ &\quad + (d_1 + (d_2+1)*d_1 + (d_1d_2 + 2d_2) + (d_1+1)*d_2)(\text{Back Propagation}) \\ &= 5d_1d_2 + 3d1 + 5d2, \end{aligned} \tag{2.1}$$

where d_1 is the dimension of the inputs ($d_1 + 1$ takes account of the bias term), d_2 is the dimension of the second layer. For a training data set with size N, one epoch requires Np operations. To facilitate the discussion, in this paper, unless otherwise

mentioned, we set the number of epochs needed for training to be a fixed value E, leading to the total number of operations for SGD (p_{SGD}) of NpE. And in practice, the overall loss is evaluated several times during training to determine whether the convergence is reached. If a total of T times of loss computation are made, the number of operations for loss computation is

$$\begin{aligned} p_{loss} &= TN((d_1+1)*d_2+d_2+(d_2+1)*d_1+2d_1) \\ &1 \leq T \leq E \end{aligned} \tag{2.2}$$

As a result, the total number of operation for a classic auto-encoder training can be determined as:

$$\begin{aligned} p_{AE} &= p_{SGD} + p_{loss} \\ &= NE(5d_1d_2+3d_1+5d_2) + TN(2d_1d_2+3d_1+2d_2) \end{aligned} \tag{2.3}$$

The GN algorithm is directly derived from the classic autoencoder. For each node i, after E epochs of training, one extra forward propagation is performed with all data and the results will be stored after added by the previous stored results. When training with node i, the output will be computed as the sum of the "memorized value" times the amnesia factor ($[0,1]$) and the output from forward propagation through node i. As a requirement of such algorithm, the inner layer should not have the bias node, which should not be a problem if data are centralized beforehand. The number of operations for the first node is

$$\begin{aligned} p_{node(1)} &= NE((d_1+1+1+d_1)(\text{Forward Propagation}) \\ &\quad + (d_1+d_1+(d_1+2)+d_1+1))(\text{Back Propagation})) \\ &= NE(6d_1+5) \end{aligned} \tag{2.4}$$

For ith node ($2 \leq i \leq d_2$), the number of operations is

$$\begin{aligned} p_{node(i)} = \frac{NE}{d_2 - 1}((d_1 + 1 + 1 + d_1)&(\text{Forward Propagation}) \\ &+ d_1(\text{Add the stored output from pervious nodes}) \\ &+ (4d_1 + 3))(\text{Back Propagation}) \\ = \frac{NE}{d_2 - 1}&(7d_1 + 5) \end{aligned} \tag{2.5}$$

After the first node, the Euclidean distance (loss) of each sample will be sorted, it will take an extra $O(N\log(N))$ operations. We here assume that for each node, T times of loss computation are made to check the convergence.

$$\begin{aligned} p_{loss} &= TN(2d_1 + 2 + 2d_1) + \sum_{i=2}^{d2} \frac{TN}{d_2 - 1}((d_1 + 1) * 1 + 1 + 1 * d_1 + 2d_1) \\ &= TN(8d_1 + 4) \end{aligned} \tag{2.6}$$

And for each node, one extra forward propagation with the whole data set will be performed, the results will be stored and used in equation 2.5. The number of operation here is:

$$\begin{aligned} p_{extra} &= N((d_1 + 1) * 1 + 1 + 1 * d_1) * d_2 \\ &= N(2d_1 + 2)d_2 \end{aligned} \tag{2.7}$$

The total operation for this design is:

$$\begin{aligned} p_{GN} &= p_{node(1)} + \sum_{i=2}^{d2} p_{node(i)} + p_{loss} + p_{extra} + O(N\log(N)) \\ &= NE(13d_1 + 10) + TN(8d_1 + 4) + N(2d_1d_2 + 2d_2) + O(N\log(N)) \end{aligned} \tag{2.8}$$

The conceptual basis for this algorithm is that the features for a certain data set are additive: the most basic feature is learned in the first node, and the newer features can add to the older ones. Similarly, for each inner node, a new feature can be learned from the new fraction of training data since the old knowledge is used to constrain the training and force the learning to be performed in a different

direction.

The GCN algorithm extends from the GN algorithm but in an supervised flavor. Instead of using the results from the first node and a partition function, we directly make each node learn from a proportion of data in a single class. The number of operations for this algorithm can be easily computed based on equation 2.5, 2.6 and 2.7:

$$p_{GCN} = NE(7d_1 + 5) + TN(4d_1 + 2) + N(2d_1d_2 + 2d_2) \tag{2.9}$$

To facilitate the comparison, we here also show the total number of operations for classic unsupervised auto-encoder with the same setting ($d_1 + 1$ input nodes, d_2 hidden nodes and d_1 output nodes):

$$p_{AE} = NE(5d_1d_2 + d_1 + 5d_2) + TN(2d_1d_2 + 2d_1 + 2d_2) \tag{2.10}$$

Comparing equations 2.8, 2.9 and 2.10, it can be observed that the speed-up for the two new algorithms increases with a larger d_2, which indicates that they may show larger boost in terms of speed for high-dimensional problems. However a large size (N) of data set will slow down the GN algorithm due to the extra sorting routine.

Figure 2.2, Figure 2.3, Figure 2.4 shows the simulated results for the running time with other parameter set in typical value range($N = 10000$, $E = 1000$, $T = 100$). Based on the simulation, one can find that the speed-up remains constant with the change of d_1 but increases with d_2. This result implies that the new algorithms can achieve significant speed-up for high dimension problems, in which the dimension of the inner layer d_2 should be proportional to d_1 so as to be able to handle the complexity. Figure 2.4 shows the comparison when the ratio between d_1 and d_2 are fixed to be 0.75. In this case, we can observe that the speed-up of the two algorithm eventually converges to a certain value.

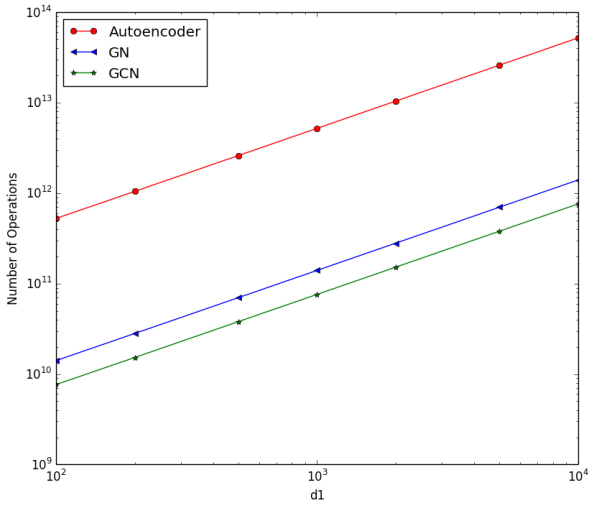

Figure 2.2: The number of operations change with the input dimension d_1, autoencoder, GN and GCN, with d_2 set to be 100.

2.3 Pseudocode

The pseudocode for the GN algorithm to train one inner layer is as follows:

The pseudocode for GCN algorithm is similar. The only difference is that the partition function is not used and for the whole dataset S, each class will be splitted into d_2/c partitions.

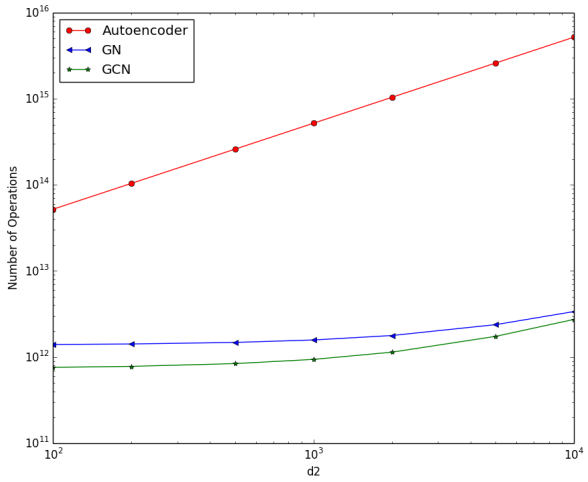

Figure 2.3: The number of operations change with the input dimension d_2, autoencoder, GN and GCN, with d_1 set to be 10000.

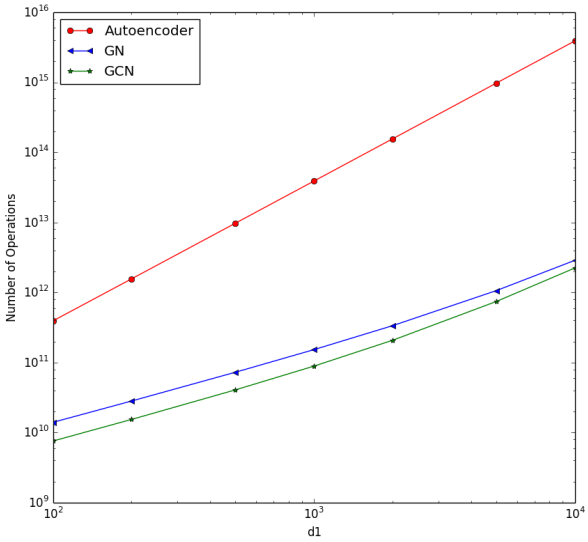

Figure 2.4: The number of operations change with the input dimension d_2, autoencoder, GN and GCN, with the d_2/d_1 set to be 0.75.

Algorithm 1: GN algorithm

```
Function GN(S,weights_0_intial)
    Data: dataset S, stored_values, weights_0, weights_1
    Result: weights_0, weights_1
    initialize weights_1, stored_values, amnesia_factor;
    foreach node_index in 0:d_2 do
        Epoch = 0;
        if node_index == 0 then
            ℰ = indexes for whole dataset 𝒮;
        else
            ℰ = indexes of data selected by partition rule;
        end
        while True do
            foreach i in ℰ do
                single_node_output computed from Forward Propagation;
                output = single_node_output + stored_values[i] *
                amnesia_factor;
                Back Propagation based on output;
                Update the corresponding parts of weights_0, weights_1;
            end
            if Epoch > Epoch_limit then
                break;
            end
        end
        Forward Propagation for whole dataset;
        Store the result from Forward Propagation to stored_values;
        if node_index == 0 then
            Apply partition rule to data;
            /* in this paper, the partition rule is to rank the
               losses based on the Forward Propagation        */
        end
    end
```

3. COMPARISON TO PRINCIPAL COMPONENT ANALYSIS

3.1 Principal Component Analysis

As one of the most classic dimension reduction method, Principal component analysis (PCA) [32] has already been invented for more 100 years. However the resemblance of PCA to the unsurpervised autoencoder (AE) is still important for us to gain a deep understanding on autoencoder and how the dimension reduction should be done in order to capture the most relevant information. In this section we will briefly review the fundamental concept of PCA and show that at least in the conceptual level, PCA and AE are very similar. The goal of PCA analysis is to find an orthonormal matrix W, so that columns in W transform the original data X into a new coordinate system. The new coordinate system is chosen in the following process: direction for the largest variance of X will be the first column, also called first principal component (first PC), and of all directions that orthogonal to the first PC, the one with largest variance is chosen as the second PC. The process continues until all d (the dimension of X) PC's are found. Each column in W is called a PCA loading. As shown in the equation below:

$$T = XW \tag{3.1}$$

Since the columns in W can be ranked by the percentage of variance that explained in each direction (the corresponding eigenvalue of the covariance matrix) , it is possible to "encode" the information in T by less dimensions. This idea is the basis of PCA. Suppose only the first L components are used, the matrix used to project X is not W but W_L, resulting in T_L which only has dimension of L instead of d, it is very similar to the concept of AE when using L inner nodes for reconstruction.

$$T_L = XW_L \tag{3.2}$$

For reconstruction, the T_L can be easily reconstructed to get the approximation of X by multiplying a W_L^T

$$X_{recon} = XW_L W_L^T \tag{3.3}$$

From the equation above, one can see that W_L can be regarded as the lower weights in the AE network and if the activation functions in the inner nodes are linear, W_L^T is actually the upper weights used to "decode" the information and reconstruct inputs. Baldi [33] has discussed the autoencoder with linear activation function with more rigorous analysis. However, in practical use the linear activation function can not handle any nonlinearity and sigmoidal function is widely used. How much the AE resembles PCA and how the new algorithms related to them are important points to study.

3.2 Comparison between PCA, AE and GN

As people have noticed and mentioned, especially if the activation function is linear, AE and PCA are very similar [33]. But very few discussions have been made when the activation function is nonlinear, such as sigmoid function, which is actually widely used in practice. Furthermore, since the new algorithms implement quite different fundamental ideas, it is also interesting to investigate whether the new algorithms can provide a different dimension reduction from the two classical algorithms. In this study, only GN (the unsupervised version) is considered since the experiments are in general unsupervised and no labels are considered. We here consider two data sets: iris and swiss roll. The iris data sets are simple and can be considered as in linear manifold and the swiss roll data set is a typical nonlinear manifold example. For iris data set, we first consider a network with only one inner nodes, as shown in Figure 3.1. After training the network as the classic autoencoder on the original data set X, we obtain the reconstructed data X_{recon}, then we plot 2D PCA on both X and X_L. As shown in Figure 3.2, the X_{recon} contain almost no other information but the first principal component. Then instead of using only one inner nodes, we use two and directly plot the outcomes of the inner nodes for each data points (after the nonlinear activation transformation). Figure 3.3 shows that the actual outcomes of the two inner nodes are very similar to those from 2D PCA.

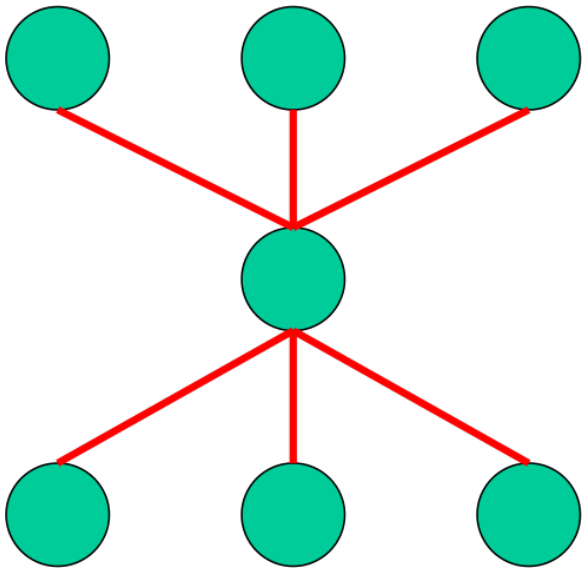

Figure 3.1: Network with one inner node

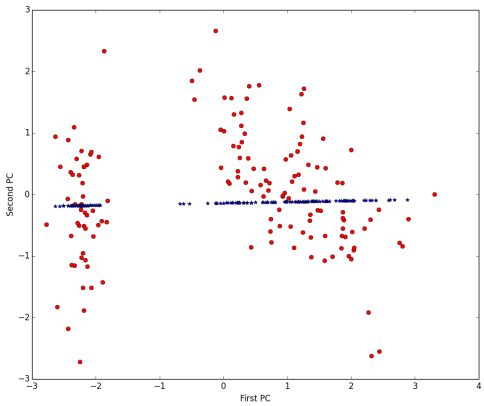

Figure 3.2: 2D PCA of original data (red) and reconstructed data (blue) using one inner node

It is clear that for the iris data set, the AE and PCA are almost the same and the d_2 inner nodes corresponds to the first d_2 principal components. A similar result of the two inner nodes' outcomes can also be seen for the swiss role data set. Figure 3.4 shows the original data set and it can be observed from Figure 3.5 that the outcomes from the inner nodes are still similar to the 2D PCA but not

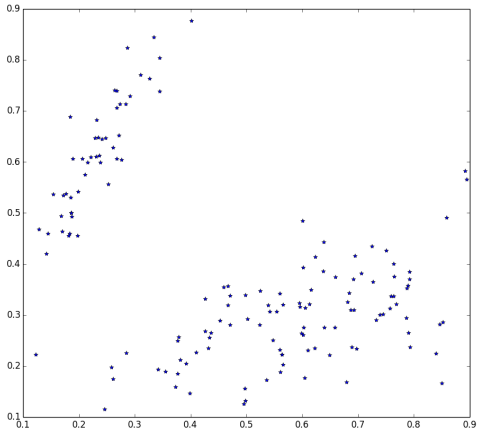

Figure 3.3: Encoding iris data set using two inner nodes.

necessarily capturing the nonlinear manifold structure. Additionally, it should be noted that the bias terms in the inner layer does not change such behavior and the plots are very similar (results not shown).

GN algorithm builds up the encoding using only part of the data. Figure 3.6 shows the outcomes of inner nodes in a two-inner-node network for both data sets. It can be observed that under this setting, the results are less similar to that of AE and PCA. Since all data are used for the first nodes and the system is just like in Figure 3.1, this result indicates that the partial data (half of the data set) used for the second node can still almost capture the information for the second PC.

A special feature about the GN algorithm is that the size of inner nodes can be set to be larger than the input layer without any risk of learning trivial solutions. It is due to the fact that each node will only work on certain subset of the data, so that it is impossible for the inner layer to learn an trivial encoding (in classic AE, it is possible that the system learns an identity function when the size of the inner layer is larger or equal to the input layer). Figure 3.7 shows the outcomes of the first two nodes when the size of inner layer is 10 (the input data only has three dimensions). What surprises us is that the system learns the correct nonlinear manifold, which has never been reported in similar researches for AE. It should be noted that the since the first node encode identical information as normal AE

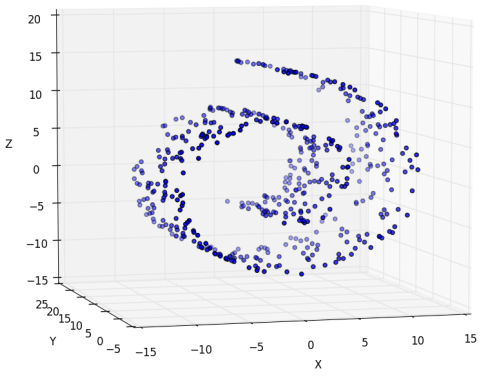

Figure 3.4: Swiss roll data set.

with one inner nodes, this result indicates that a smaller part of the data actually provide a better information for the encoding. The reason seems to be that the smaller amount of data provide a local information which reveals a feature that may not gain consensus within larger amount of data. Further analysis for this behavior is definitely needed.

From the comparison between PCA, AE and GN, we observe that AE and PCA are very similar to each other, even with nonlinear activation function, in contrast, GN algorithm seems to encode and learn feature from the data sets in a different way, and may be able to reveal more localized features.

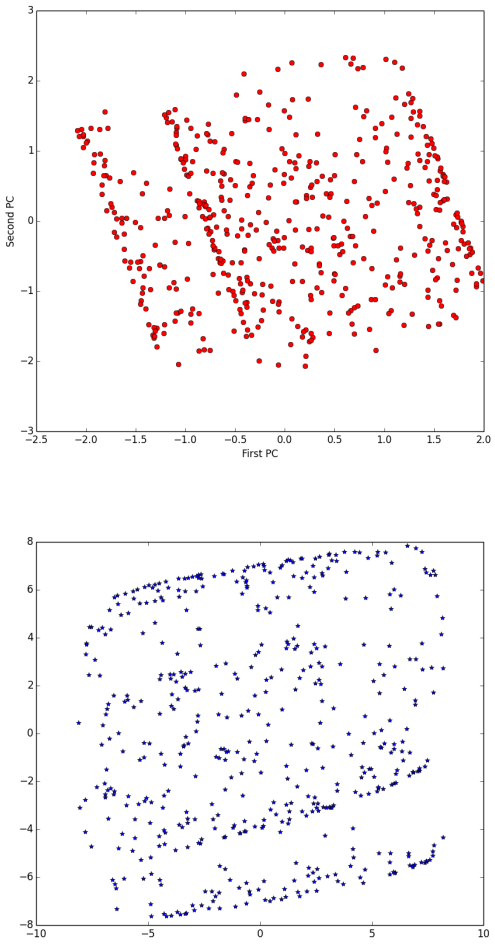

Figure 3.5: 2D PCA(top) and AE(bottom) for swiss roll data set.

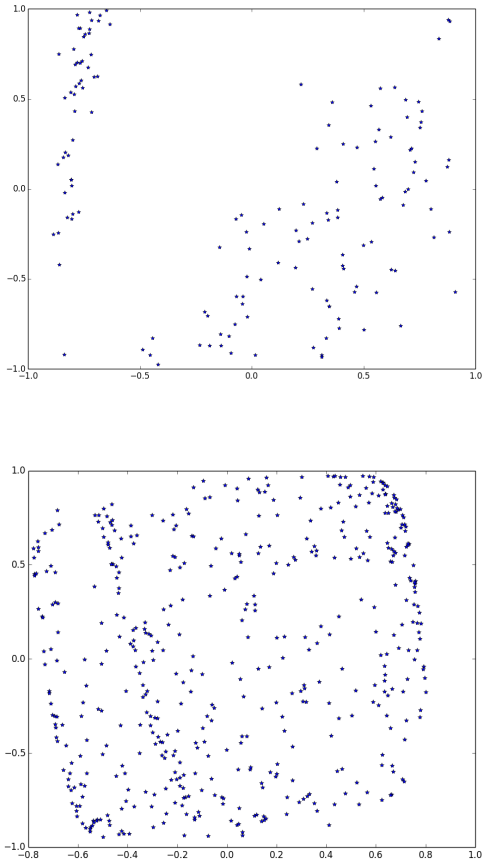

Figure 3.6: Outcomes of two inner nodes using GN algorithm with a two-inner-node network. Top: iris. Bottom: swiss roll.

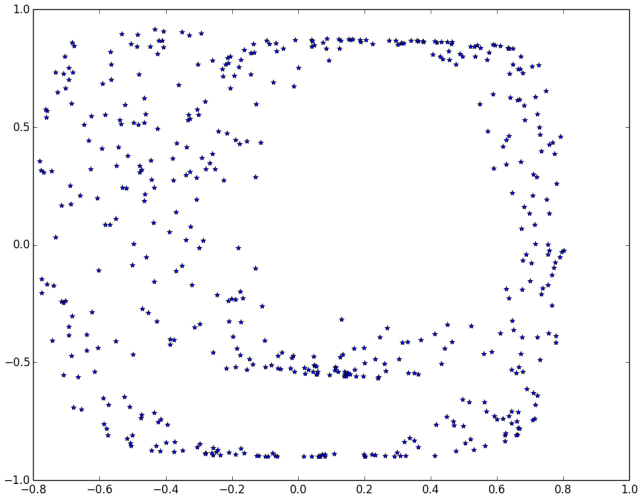

Figure 3.7: Outcomes of first two inner nodes using GN algorithm with a ten-inner-node network on the swiss roll data set.

4. RESULTS AND DISCUSSION

4.1 Speed-up and Preliminary Performance Results

From the chapter 2 we proved in theory that the two new algorithms will provide a speed-up for a relatively large network. The driving force behind is the partition of the training task on inner nodes. We here compare the running time and performance of supervised pre-training, unsupervised pre-training, GN and GCN algorithms for the USPS digit data set. To facilitate comparison and obtain reasonable results, we used a fixed iteration of 300 for pre-training with an initial learning rate 0.001, 500 iterations of output layer with logistic regression, with an initial learning rate 0.002 and 20 iterations of fine-tuning with fixed learning rate 0.001. It should be noted that for supervised pre-training, the extra logistic regression is skipped since the pre-training for the final layer is exactly a logistic regression. The shrinkage of the learning rate is set to be rather small - decrease to half after 1000 epochs after the first 100 epochs:

$$r = \frac{r_0}{1 + epoch/1000.0} \tag{4.1}$$

A $L2$ regularization term is applied on all the learning process with the coefficient 1.0. It should be noted that this "weight decaying" regularization term is not always necessary in deep learning, but we found it may help in the training with single inner node. In this paper, we are not trying to optimize all hyperparameters to get the highest possible test performance for certain data sets, instead we try to compare the four algorithm with one predefined set of parameters, as long as the test performance is reasonable and training is numerically stable. Table 4.1 shows the results for running time. Note that the network column only shows the inner nodes and for each entry the input layer has 257 nodes and the output layer has 10 nodes. RT and PT are for the whole running time and pre-training time (running time minus the time for fine-tuning and data pre-processing), respectively. 7291 data are in training and 2007 data in test. All the experiment was done using a

Table 4.1: Running-time and preliminary performance for supervised pre-training, unsupervised pre-training, GN and GCN

Type	Network	RT (s)	PT (s)	Training score	Test score
Supervised	200, 150	1363	1293	1.000	0.939
Unsupervised	200, 150	4112	4042	0.999	0.933
GN	200, 150	674	604	0.999	0.931
GCN	200, 150	409	339	0.998	0.923

single Intel i-7 3370 3.4GHz CPU.

According to the experiment, the GN algorithm shows a comparable prediction performance comparing to unsupervised pre-training algorithm. It gains a speed-up of 6.7 against unsupervised algorithm and 2.1 against supervised algorithm. The GCN algorithm shows the worst performance but gains a speed-up of 12 against unsupervised algorithm and 3.8 against the supervised algorithm. The algorithms are all implemented in python, which may unavoidably include some overhead comparing to compiling language like C++. However, the four algorithms are implemented using very similar code structure to ensure the running time is comparable. In the experiment above, the amnesia factor has been set to be 1.0, which may not be the optimal value. In the next section, we show that a better choice of the amnesia factor can significantly improve the prediction performance.

4.2 Amnesia Factor

In the algorithms developed in this study, the amnesia factor plays a very important role. It is also a new parameter that does not exist in the classic algorithms. Understanding the effect and behavior of the amnesia factor will help us to fully investigate and utilize the new algorithms.

The amnesia factor (AF) is used to control the influence of the results from the previous forward propagation when training the system with the node i. During the training, the data propagate through node i and then added by the result from a scaled previous output. In other words, the output when training node i is not directly compared to the input for backpropagation as classic autoencoder does.

Instead, it is compared to the input data minus the previous result with a scaling factor. The previous output can be analogized to the "memory" of previous learned knowledge/feature. In this way, the node i should learn the new feature with the previous information "in mind". The scaling factor used here is the amnesia factor. The amnesia factor can also be seen as a measure of the preassumption of the contribution of each node to the final model. In this sense, the traditional autoencoder can be seen as having an amnesia factor of $1/d_2$, where d_2 is size of the hidden layer. According to our study, this factor is very important for both of the new algorithms. We here do a preliminary study of its effect on GCN algorithm.

Figure 4.1 shows the training loss of each node in the first hidden layer for a GCN algorithm in a 256-200-150-10 network. As mentioned previously, this loss is computed as the Euclidean distance between the input and the summed output (the summation of the scaled previous output and the output from the current inner node). It was found that that loss with amnesia factor 1.0 (which means no loss of memory) drifted to high value as more nodes are used, and resulting in less meaningful features and worse prediction accuracy. A large error means that with the previous knowledge, there is no new feature can be obtained using the current node and partition of the input data. This may be due to the fact that, as the number of nodes getting larger, it becomes increasingly harder to train a sub-network since there are a lot of constraints existed. Consequently it seems like a better learning setting (a better choice of learning rate) will solve this problem. However the learned features are likely to have extreme values-in order to compensate the some peaks and holes caused by a strong influence of previous information, which may eventually lead to the numeric instability. A smaller amnesia factor can be used to mitigate this issue. Every time instead of adding the whole "memorized information" onto the new output, we shrink the "memorized value" by an amnesia factor in order to give some "space" for the new features. This treatment allows some overlap between learned features, and the extent of the overlap is actually controlled by the AF. As shown in figure 4.1, a smaller AF can effectively control the drifting.

Table 4.2 shows the result of a 256-200-150-10 network with learning rate 0.001 and amnesia factor 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4 and 0.0. All the other settings are

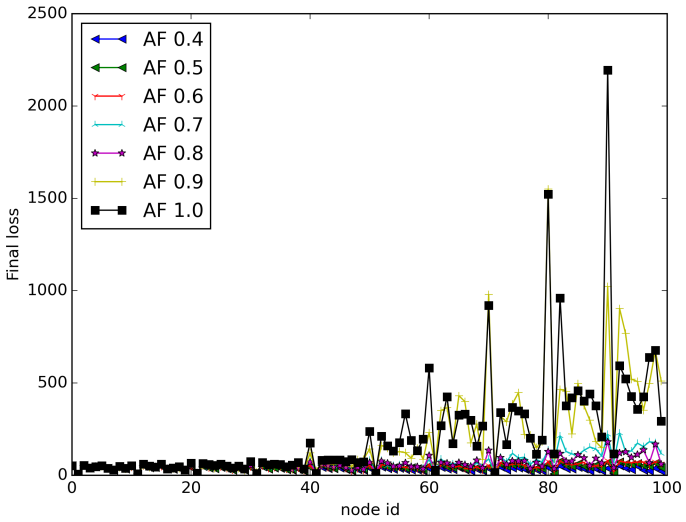

Figure 4.1: Final loss after 300 epochs for each inner node in the first layer using GCN algorithm with 256-200-150-10 network. AF denotes amnesia factor

Table 4.2: Effect of the amnesia factor

Learning rate	**Amnesia Factor**	**Training score**	**Test score**
0.001	1.0	0.998	0.923
0.001	0.9	0.999	0.922
0.001	0.8	0.999	0.926
0.001	0.7	0.999	0.932
0.001	0.6	0.999	0.927
0.001	0.5	0.999	0.931
0.001	0.4	0.999	0.934
0.001	0.0	0.974	0.915

the same as in section 4.1. It is clear that some optimal value exists between 1.0 and 0.0.

The result agrees with our analysis that a too large amnesia factor could lead to worse performance since it will be hard to optimize given strong constraints. And we also found that amnesia factor of zero was not ensured to give a good performance, it can be explained by the intrinsic duplicated features. In addition to that, if the amnesia factor is zero, each node essentially represents the medoid or

the first principal component of corresponding training data. This scenario can be compared to using K-means, where the local representation of data, instead of the generative one, will be used. In that case, the fine structure of the input data may not be learned and it is also against the whole idea behind deep learning. Using a non-zero amnesia factor can be seen as applying a balance between the two types of representations. A higher AF will lead to generative model of inputs but induce numerical issue for training. A lower AF will lead to local model of inputs but result in loss of information. However the optimal amnesia factor may be highly depending on size of certain layer. Further study will be needed in order to understand the intrinsic meaning of this parameter and its relation to different partition scheme.

4.3 Other Datasets

Three data sets obtained from UCI machine learning repository [34] were also used to compare the algorithms. The MUSK (version 2) data set describes molecules that are judged by experts as musks and non-musks [34] with 166 attributes. 4618 data was used for training and 1980 data was used for test. ISOLET data set contains the audio information for 26 letters spoken by human speakers [34] with 617 attributes. 4366 data was used for training and 1872 data was used for test. For ISOLET data set, we used a smaller learning rate of 0.0001. CNAE-9 contains 1080 documents of text business descriptions of Brazilian companies categorized into 9 categories [34]. 864 data were used for training while 216 data were used for test. Different from the other two data sets, the data matrix for CNAE-9 is highly sparse. We found that a learning rate up to 0.1 can produce a reasonable performance for all the new algorithms while the classical algorithms use a learning rate of 0.01, even though better performance can be easily obtained if we use different learning rates for each algorithm (for example, GCN with learning rate of 0.15 has the test score of 0.935). For the three data sets, the GCN algorithm uses an amnesia factor of 0.4. Note that since GCN requires the number of nodes in each hidden layer to be divisible by the number of class, the network for all the algorithms were set accordingly (that is why the first layer for CNAE-9 is set to be 594, not 600). Table 4.3 shows the comparison between supervised, unsupervised, GN and GCN

Table 4.3: Comparison between algorithms in extra data sets

Data Set	**Method**	**Network**	**Training score**	**Test score**
MUSK	Supervised	120-80	1.000	0.991
MUSK	Unsupervised	120-80	1.000	0.993
MUSK	GN	120-80	1.000	0.984
MUSK	GCN	120-80	1.000	0.989
ISOLET	Supervised	260-78	0.997	0.946
ISOLET	Unsupervised	260-78	0.999	0.953
ISOLET	GN	260-78	1.000	0.944
ISOLET	GCN	260-78	1.000	0.935
CNAE-9	Supervised	594-396	1.000	0.954
CNAE-9	Unsupervised	594-396	0.996	0.944
CNAE-9	GN	594-396	0.995	0.940
CNAE-9	GCN	594-396	0.994	0.921

algorithm. The results are in general consistent with the USPS handwriting data set we used previously. It shows that GCN algorithm has 1 2% decrease in accuracy for ISOLET and CNAE-9 , which has also been observed for the USPS data, but it performs very well for MUSK dataset. Whether the performance is dependent on the intrinsic data structure or the parameter tuning or both requires further analysis.

5. CONCLUSION AND FUTURE DIRECTIONS

In this thesis, we developed two novel deep learning algorithms that origin from the idea of simulating human's learning process. A significant difference between classical deep learning algorithms and human's learning process is that deep learning always tries to obtain the hierarchical features of all the objects at the same time (train all inner nodes at the same time), while a human learner learns one or few objects at one time and capable of learning new features with memorization. Based on this understanding, we develop two methods, corresponding to supervised learning (GCN) and unsupervised learning (GN), that learn one feature of one certain group of training data at a time. Such design helps to construct more human-recognizable features with a significant speed-up. In addition to that, we introduce the new concept of "amnesia factor" in combination with an algorithm design of caching. "Amnesia factor" is used to control the memorization effect and the "orthogonality" between the learned features. With the utilization of caching, the running time of the algorithm is also well controlled and has a speed-up up to ten times. Meanwhile, as tested on several benchmark datasets, the prediction accuracy of the new algorithms only decreased slightly.

In the future, we would like to investigate three subproblems. First, we will look into the training parameter tuning and try to find a systematic way to choose hyper-parameters such as the learning rate and amnesia factors. Several questions need to be answered here: Is there a better way to partition the data? How the choose the optimal amnesia factor? Should the learning rate be adjusted differently for each inner node and each layer? What is the factor behind the difference on the prediction performance? Second, we will investigate the higher-level features and see if they can provide extra informations on the problems. Third, as the most important feature of the new algorithms, learning in an iterative way actually makes the scaling of the methods to large datasets with parallel computing difficult. In classical methods, the matrix-vector and matrix-matrix multiplication can be easily implemented for multicore or GPU computing. In the new methods, since

one feature depends on the feature developed previously, it is in nature an iterative process. One way to handle such difficulty could be to learn several features at the same time in different CPU but exchange information (as a memorization constraint) every few epochs (to simulate a group of human learners). The adaption of the algorithm to parallel computing will surely requires certain amount of changes in the algorithm design. And how to "keep the spirits" while run the algorithm in parallel will be an interesting yet difficult problem to solve in the next stage.

LITERATURE CITED

[1] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychol. Rev.*, vol. 65, no. 6, p. 386, Nov. 1958.

[2] B. Widrow, "Adaptive switching circuits," in *IRE WESCON Conv. Rec.*, Los Angeles, CA, 1960, pp. 96–104.

[3] M. Hoff Jr, "Learning phenomena in networks of adaptive switching circuits," Ph.D. dissertation, Depart. of Elect. Eng., Stanford Univ., Stanford, CA, May 1962.

[4] Y. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning From Data: A Short Course*. n.p.: AMLBook, 2012.

[5] D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, D. Rumelhart and J. McClelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 318–362.

[6] G. W. Cottrell and P. Munro, "Principal components analysis of images via back propagation," in *Proc. SPIE 1001, Visual Commun. and Image Process.*, Cambridge, MA, 1988, pp. 1070–1077.

[7] D. Erhan *et al.*, "Why does unsupervised pre-training help deep learning?" *J. Mach. Learn. Res.*, vol. 11, pp. 625–660, Feb. 2010.

[8] V. Vapnik, *The Nature of Statistical Learning Theory*. New York, NY: Springer Science, 2000.

[9] M. Gori and A. Tesi, "On the problem of local minima in backpropagation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, no. 1, pp. 76–86, Jan. 1992.

[10] K. Fukumizu and S.-i. Amari, "Local minima and plateaus in hierarchical structures of multilayer perceptrons," *Neural Networks*, vol. 13, no. 3, pp. 317–327, Apr. 2000.

[11] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[12] Y. Bengio *et al.*, "Greedy layer-wise training of deep networks," in *Adv. Neur. Inform. Process. Syst.*, vol. 19, Vancouver, Canada, 2007, pp. 153–160.

[13] G. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[14] H. Lee *et al.*, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proc. 26th Int. Conf. on Mach. Learn. (ICML'09)*, Montreal, Canada, 2009, pp. 609–616.

[15] X. Glorot, A. Bordes, and Y. Bengio, "Domain adaptation for large-scale sentiment classification: A deep learning approach," in *Proc. 28th. Int. Conf. Mach. Learn. (ICML'11)*, Bellevue, WA, 2011, pp. 513–520.

[16] H. Lee *et al.*, "Unsupervised feature learning for audio classification using convolutional deep belief networks," in *Adv. Neur. Inform. Process. Syst.*, Vancouver, Canada, 2009, pp. 1096–1104.

[17] D. Yu and L. Deng, "Deep learning and its applications to signal and information processing," *IEEE Signal Process. Mag.*, vol. 28, no. 1, pp. 145–154, Jan. 2011.

[18] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," *IEEE Trans. Acoust., Speech, Signal Process.*, pp. 8599–8603, May 2013.

[19] Y. Bengio, "Deep learning of representations for unsupervised and transfer learning," in *JMLR W&CP: Proc. Unsupervised and Transfer Learn.*, Bellevue, WA, 2011, pp. 1–20.

[20] J. Ngiam *et al.*, "On optimization methods for deep learning," in *Proc. 28th. Int. Conf. Mach. Learn. (ICML'11)*, Bellevue, WA, 2011, pp. 265–272.

[21] J. Martens, "Deep learning via hessian-free optimization," in *Proc. 27th. Int. Conf. Mach. Learn. (ICML'10)*, Haifa, Israel, 2010, pp. 735–742.

[22] P. Vincent *et al.*, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, pp. 3371–3408, Mar. 2010.

[23] P. Vincent *et al.*, "Extracting and composing robust features with denoising autoencoders," in *Proc. 25th Int. Conf. on Mach. Learn. (ICML '08)*, Helsinki, Finland, 2008, pp. 1096–1103.

[24] M. Ranzato, Y.-L. Boureau, and Y. LeCun, "Sparse feature learning for deep belief networks," in *Adv. Neur. Inform. Process. Syst.*, Vancouver, Canada, 2008, pp. 1185–1192.

[25] C. Poultney *et al.*, "Efficient learning of sparse representations with an energy-based model," in *Adv. Neur. Inform. Process. Syst.*, Vancouver, Canada, 2006, pp. 1137–1144.

[26] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," in *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA: MIT Press, 1995, pp. 255–258.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Adv. Neur. Inform. Process. Syst.*, Lake Tahoe, CA, 2012, pp. 1097–1105.

[28] G. Chen, "Deep learning with nonparametric clustering," *ArXiv Preprint ArXiv:1501.03084*, Jan. 2015.

[29] J. Weston *et al.*, "Deep learning via semi-supervised embedding," in *Neural Networks: Tricks of the Trade*, G. Orr and K. Müller, Eds. New York, NY: Springer, 2012, pp. 639–655.

[30] K. Faraoun and A. Boukelif, "Neural networks learning improvement using the k-means clustering algorithm to detect network intrusions," *INFOCOMP J. Comput. Sci.*, vol. 5, no. 3, pp. 28–36, Apr. 2006.

[31] A. Coates and A. Y. Ng, "Learning feature representations with k-means," in *Neural Networks: Tricks of the Trade*, G. Orr and K. Müller, Eds. New York, NY: Springer, 2012, pp. 561–580.

[32] K. Pearson, "On lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosph. Mag. and J. of Sci.*, vol. 2, no. 11, pp. 559–572, 1901.

[33] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," in *JMLR W&CP: Proc. Unsupervised and Transfer Learn.*, vol. 7, Bellevue, WA, 2012, pp. 37–50.

[34] M. Lichman, "UCI machine learning repository," 2013, date last accessed: Oct-30-2015. [Online]. Available: http://archive.ics.uci.edu/ml