이학석사 학위논문

A STUDY ON STACKED AUTOENCODERS AND ITS FINE-TUNING

적층 자가인코더의 지도학습적 활용에 대한 연구

2015년 2월

서울대학교 대학원

통계학과

신 재 혁

A STUDY ON STACKED AUTOENCODERS AND ITS FINE-TUNING

지도교수 박 병 욱

이 논문을 이학석사 학위논문으로 제출함

2014년 10월

서울대학교 대학원

통계학과

신 재 혁

신재혁의 이학석사 학위논문을 인준함

2014년 12월

위 원 장	오 희 석	(인)
부위원장	박 병 욱	(인)
위 원	원 중 호	(인)

A STUDY ON STACKED AUTOENCODERS AND ITS FINE-TUNING

by

Shin Jaehyeok

A Dissertation
submitted in fulfillment of the requirement
for the degree of
Master of Science
in
Statistics

The Department of Statistics
College of Natural Sciences
Seoul National University
February, 2015

Abstract

Shin Jaehyeok
The Department of Statistics
The Graduate School
Seoul National University

A stacked autoencoder is a kind of unsupervised deep learning algorithm which looks like the automatically learning features of input data, such as edges and objects in images. Stacked autoencoders have been used as building blocks to build and initialize multi-layer neural networks. Neural networks based on them have shown outstanding performance in natural images and speeches classification tasks. In this paper, we especially focus on the image analysis. We first introduce neural networks and autoencoders, and provide an explanation of what autoencoders actually learn. Then, we explain how to stack up autoencoders and how to use the fine-tuning method for the purpose of making a high-performance image classifier. Finally, we carry out a numerical study with MNIST handwritten digit database.

Keywords : Stacked autoencoder; Denoising autoencoder; Neural networks model; Deep learning.

Student Number : 2013-20219

Contents

Chapter 1

Introduction

Since McCulloch and Pitts (1943) initially devised a mathematical model to describe nervous activity, there have been many attempts to construct statistical models to imitate human brain's outstanding performance for the learning tasks such as pattern recognition and prediction. Rosenblatt (1958) created the perceptron, the simplest neural networks model consisting of simple linear calculations of input data. Before the 1980s, neural networks model was not used actively because of limited computing powers and the lack of efficient estimating algorithm. However, after the backpropagation algorithm had been introduced by Werbos (1974), various kinds of multi-layer neural networks was developed and applied to supervised learning areas. In the 1990s, much simpler methods such as penalized regressions and support vector machines overtook neural networks in popularity because of their simplicity and high-performance. Only after Hinton made a significant breakthrough in estimating multi-layer neural networks by introducing a fast and greedy learning algorithm (Hinton et al., 2006), multi-layer neural networks have received great attentions both from academic and industrial areas.

In a narrow sense, deep learning architectures often refer to multi-layer neural networks with greedy layer-wise estimating methods. The "greedy layer-wise" means that deep learning algorithms try to estimating sequentially each layer's parameters while fixing other layer's parameters, unlike conventional neural networks estimate whole parameters simultaneously. This greedy layer-wise estimating step is called **pre-training** in the machine learning literature. After finishing pre-training, we can stack up estimated layers to construct a multi-layer neural network. At this point, pre-training plays a role as an initializer in estimating a multi-layer neural network. Finally, we estimate parameters of whole network simultaneously using conventional backpropagation algorithm. This step is called **fine-tuning**. Since 2006, deep learning have been applied with success in lots of machine learning tasks such as classification(Bengio et al., 2007; Vincent et al., 2008), regression(Hinton and Salakhutdinov, 2008), dimensionality reduction(Hinton and Salakhutdinov, 2006) and natural language processing tasks(Collobert and Weston, 2008).

In this paper, we only consider image classification tasks, and focus on multi-layer neural networks based on **stacked autoencoders**. Autoencoders are kinds of (single hidden layer) neural networks which reuse input values as target values. In other words, they aim to reconstruct input values. Unlike other building blocks of deep learning architecture such as restricted Boltzmann machines (RBM), autoencoders have direct connection with conventional neural networks and, thus, they are easy to understand in the statistical point of view. A stacked autoencoder is a stacked version of autoencoders, usually consisting of 2-4 autoencoders. Rigorously, stacked autoencoders are unsupervised learning algorithm whose aim is also to reconstruct input values. However, in many studies, multi-layer neural networks based on the stacked autoencoder is just called a stacked autoencoder. From now on, we also use

this notation.

The remainder of this paper is organized as follows. In Chapter 2, we briefly review the feedforward neural networks model and backpropagation algorithm. In Chapter 3, we introduce autoencoders and three kinds of regularized versions of them - autoencoder with bottleneck constraint, sparse autoencoder and denoising autoencoder. Then, we provide a heuristic explanation of what autoencoders actually learn from input images. In Chapter 4, we explain how to stack up autoencoders and use fine-tuning method for the purpose of making a high-performance image classifier. In Chapter 5, we carry out numerical study with MNIST handwritten digit database.

Chapter 2

Feedforward neural networks

In this chapter, we review the feedforward neural networks model. We also see how to estimate this complex model using backpropagation algorithm.

2.1 Single neuron model

Assume we have a p-dimensional input random vector $\mathbf{X}$ and a target random variable Y. Let $\left\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \cdots, (\mathbf{x}^{(m)}, y^{(m)})\right\}$ be our training sample set. The single neuron model has the form

$$h_{\mathbf{W},b}(x) = f(\mathbf{W}^t\mathbf{x} + b) = f(\sum_{i=1}^{p} W_i x_i + b)$$

where $f(z) = \frac{1}{1+e^{-z}}$ is the sigmoid activation function, and $(\mathbf{W}, b) \in \mathbb{R}^{p+1}$

Note that if Y is binary and the loss function has logistic form, then this model is equivalent to the logistic regression model.

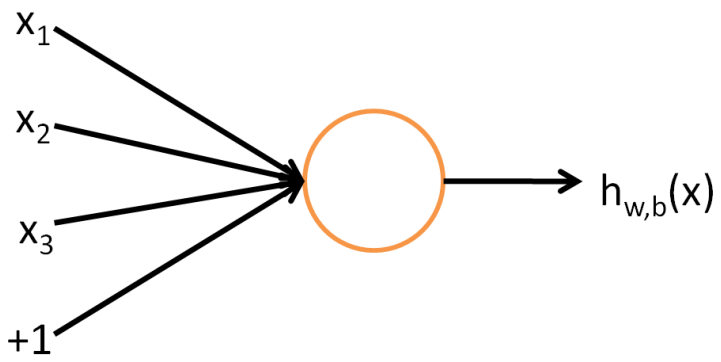


Figure 2.1: Single neuron model when $p = 3$ (Ng, 2014)

2.2 Feedforward neural networks

Feedforward neural networks consist of many neurons whose connection forms a hierarchical directed acyclic graph. For instance, the computation that the neural network in Figure 2.2 represents is given by

$$\begin{aligned} a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\ a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\ a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\ h_{\mathbf{W},b}(x) = a_1^{(3)} &= f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}) \end{aligned}$$

where

a_i^l : activation (output) of unit i in layer l

W_{ij}^l : weight associated with unit j in layer l and unit i in layer $l+1$

b_i^l : bias associated with unit i in layer $l+1$

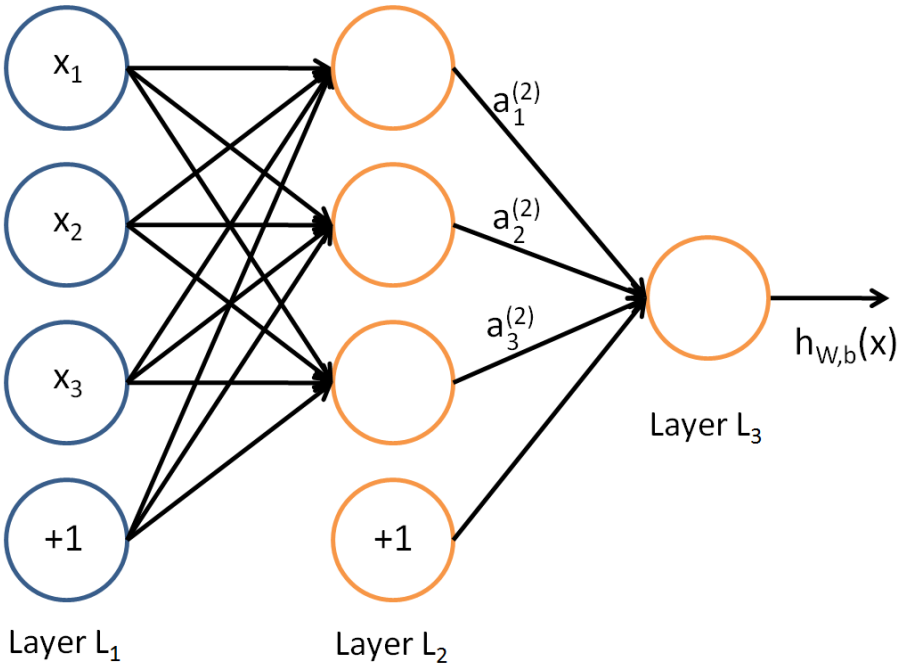


Figure 2.2: Feedforward neural network model with 3 layers (Ng, 2014)
Layer L_1 is *input layer*; Layer L_2 is *hidden layer*, and Layer L_3 is *output layer*. (+1)-circles indicate bias terms. Note that there are no arrows whose head is bias terms.

If we use vector notation ($f([z_1, z_2, z_3]) \equiv [f(z_1), f(z_2), f(z_3)]$), we can write the equation above more compactly as

$$\begin{aligned} z^{(2)} &= \mathbf{W}^{(1)} x + b^{(1)} \\ a^{(2)} &= f(z^{(2)}) \\ z^{(3)} &= \mathbf{W}^{(2)} a^{(2)} + b^{(2)} \\ h_{\mathbf{W},b}(x) &= a^{(3)} = f(z^{(3)}) \end{aligned}$$

2.3 Backpropagation Algorithm

If we use a fully connected neural network with n_l layers (layer l containing s_l units), we should estimate

$$\begin{aligned} \mathbf{W} &= \{W_{ij}^{(l)} | l = 1, \ldots, n_l, i = 1, \ldots, s_{l+1}, j = 1, \ldots, s_l\} \\ \mathbf{b} &= \{b_i^{(l)} | l = 1, \ldots, n_l, i = 1, \ldots, s_{l+1}\} \end{aligned}$$

We can estimate neural network by using empirical risk minimization principle. For instance, we can use squared-error loss function in the regression setting.

$$(\hat{\mathbf{W}}, \hat{\mathbf{b}}) = \arg\min_{\mathbf{W},\mathbf{b}} \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{\mathbf{W},\mathbf{b}}(\mathbf{x}^{(i)}) - y^{(i)} \right\|^2 \right) \right]$$

However, this model is too complex. To avoid overfitting issue, it might be better to add a ridge type penalty term to our risk function.

$$J(\mathbf{W}, \mathbf{b}) = \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{\mathbf{W},b}(\mathbf{x}^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2$$

Our estimators of parameters in feedforward neural networks are minimizers of target function $J(\mathbf{W}, \mathbf{b})$. To minimize $J(\mathbf{W}, \mathbf{b})$, we can use gradient descent algorithm.

Gradient descent algorithm

Step.1 initialize $\mathbf{W}, \mathbf{b}$

Step.2 update

$$W_{ij}^{(l)} \longleftarrow W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(\mathbf{W}, \mathbf{b})$$
$$b_i^{(l)} \longleftarrow b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(\mathbf{W}, \mathbf{b})$$

where α is the learning rate.

It is well-known that if a target function f is L-Lipschitz and the learning rate α satisfies $\alpha < \frac{2}{L}$, then the gradient descent algorithm will converge to a stationary point of f (Boyd and Vandenberghe, 2009). Since our target function $J(\mathbf{W}, \mathbf{b})$ is non-convex, the gradient descent algorithm with too small fixed learning rate might find a local minimum, not a global one. Hence, in many application, the learning rate α is taken as $\frac{\lambda_1}{\lambda_2 + t}$ to make α decreasing from a large value to 0 as algorithm step t increasing.

In general, the gradient descent algorithm involves complicated calculation of derivatives of target functions. However, due to specially form of neural networks' target function, we can use simple back-propagation algorithm to compute its derivatives

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(\mathbf{W}, \mathbf{b}) = \frac{\partial}{\partial W_{ij}^{(l)}} \frac{1}{2} \left\| h_{\mathbf{W},b}(\mathbf{x}) - y \right\|^2 + \lambda W_{ij}^{(l)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(\mathbf{W}, \mathbf{b}) = \frac{\partial}{\partial b_i^{(l)}} \frac{1}{2} \left\| h_{\mathbf{W},b}(\mathbf{x}) - y \right\|^2$$

Backpropagation algorithm (Werbos, 1974; Ng, 2014)

Step.1 Perform a feed-forward pass (computing the activations (outputs) for layers $L_2,\ L_3$ and so on up to the output layer L_{n_l})

Step.2 For each output unit i, set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{\mathbf{W},b}(\mathbf{x}) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

Step.3 For $l = n_l - 1, n_l - 2, \ldots, 2$

For each node i in layer l

set $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$

Step.4 Compute

$$\frac{\partial}{\partial W_{ij}^{(l)}} \frac{1}{2} \left\| h_{\mathbf{W},b}(\mathbf{x}) - y \right\|^2 = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} \frac{1}{2} \left\| h_{\mathbf{W},b}(\mathbf{x}) - y \right\|^2 = \delta_i^{(l+1)}$$

Combining the backpropagation and the gradient descent algorithm, we can estimate the parameters of our model.

$$(\hat{\mathbf{W}}, \hat{\mathbf{b}}) = \arg\min_{\mathbf{W},\mathbf{b}} J(\mathbf{W}, \mathbf{b}) = \arg\min_{\mathbf{W},\mathbf{b}} \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{\mathbf{W},\mathbf{b}}(\mathbf{x}^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2$$

Chapter 3

Autoencoders

In this chapter, we introduce autoencoders and three kinds of regularized versions of them - autoencoder with bottleneck constraint, sparse autoencoder and denoising autoencoder. Then, we provide a heuristic explanation of what autoencoders actually learn from input images.

3.1 Autoencoder with tied weights

Autoencoders are kinds of feedforward neural networks with a single hidden layer which reuse input values as target values. In other words, they aim to reconstruct input values. The first half of the autoencoder starting from input layer to hidden layer is called **encoding part**, and the other half of the autoencoder is called **decoding part**. Since autoencoders have lots of parameters, we often use so called "tied weights". It means that the parameters in the decoding part is always equal to the counterpart in encoding part except the bias terms, that is, the weights $\mathbf{W}$ in encoder and decoder are tied to each other. Autoencoders with tied weights are similar to RBM and have better

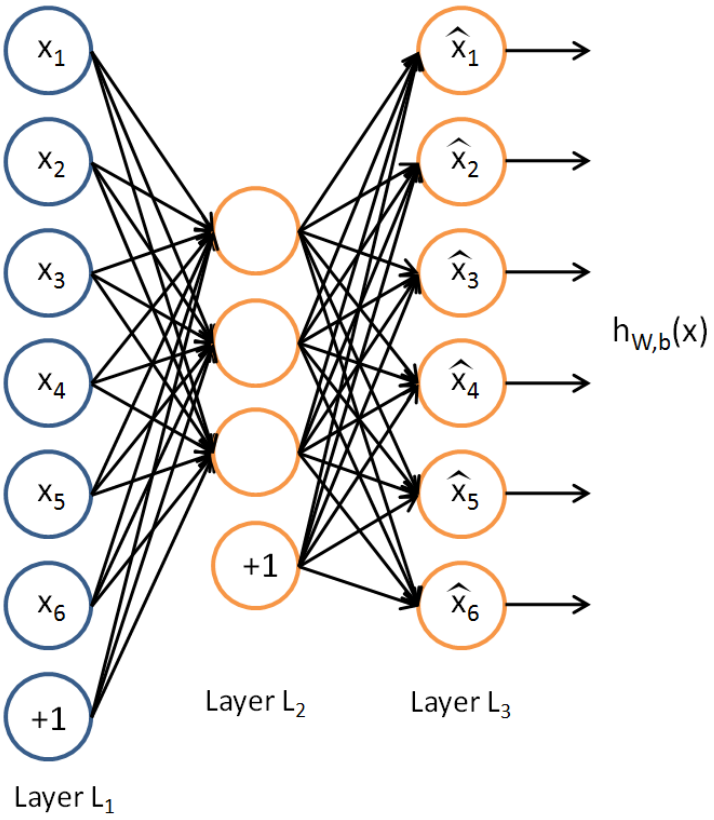


Figure 3.1: Autoencoder with 6 input units and 3 hidden units (Ng, 2014)

probabilistic interpretations than ones with untied weights (Vincent, 2011; Alain and Bengio, 2012). In this paper, we use L_2 loss function to measure the similarity between input images and output images of an autoencoder.

3.2 Regularization

At first glance, it sounds meaningless to reconstruct input images - we can just use the identity function to make perfect reconstruction function. Surprisingly, however, with some regularizations to prevent the autoencoder simply imi-

tates the identity function, the autoencoder looks like automatically learning features of input data, such as edges and contours in images (Vincent et al., 2008). In addition, regularized autoencoders can be used as building blocks of multi-layer feedforward neural networks, which will be discussed in the next chapter.

3.2.1 Autoencoder with "bottleneck" constraint

The simplest regularization is "bottleneck" constraint which impose on autoencoder to have less number of hidden units than the number of input units. It is called **undercomplete representation** in machine learning literature. Baldi and Hornik (1989) showed that autoencoder with linear activation function is equivalent to PCA, that is, we can consider autoencoder with sigmoid activation function as a kind of non-linear version of PCA. Hence, we can expect autoencoder with "bottleneck" constraint to be able to capture some interesting features of input images like PCA.

3.2.2 Overcomplete representation

Although the famous universal approximation theorem guarantees that the neural network model with sufficiently large number of hidden units can approximate any continuous function(Cybenko, 1989), undercomplete representation autoencoders could fail to capture complex high-dimensional features of input images. To overcome it, several kinds of overcomplete autoencoders have been devised since the 2000s. In this section, we introduce two kinds of them.

Sparse autoencoder

The first kind of overcomplete autoencoders is the **sparse autoencoder** (Ng, 2011) obtained by imposing sparsity constraints on the hidden units. It attempts to make the average of activations (outputs) of hidden units to be close to zero. More precisely, let $a_j^{(2)}(\mathbf{x})$ denotes the activation of j-th hidden unit corresponding to an input value x. Define the average activation of hidden unit j as follows

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[a_j^{(2)}(\mathbf{x}^{(i)}) \right]$$

We want to enforce $\hat{\rho}_j$ to be close to the sparsity parameter $\rho \approx 0$. To achieve this constraint, we add an additional penalty term to $J(\mathbf{W}, b)$

$$J_{\text{sparse}}(\mathbf{W}, \mathbf{b}) = J(\mathbf{W}, \mathbf{b}) + \beta \sum_{j=1}^{s_2} \text{KL}(\rho || \hat{\rho}_j),$$

where $\text{KL}(\rho||\hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$

Note that $\text{KL}(\rho||\hat{\rho}_j)$is the Kullback-Leiber divergence between $\text{Ber}(\rho)$ and $\text{Ber}(\hat{\rho}_\text{j})$. To estimate the spare autoencoder, we can use the modified version of the back-propagation algorithm to compute derivatives of $J_{\text{sparse}}(\mathbf{W}, \mathbf{b})$.

Backpropagation algorithm for sparse autoencoder(Ng, 2011)

Step.1 Perform a feed-forward pass (computing the activations for the hidden layer L_2 up to the output layer L_3)

Step.2 For each output unit i, set

$$\delta_i^{(3)} = \frac{\partial}{\partial z_i^{(3)}} \frac{1}{2} \left\| \mathbf{x} - h_{\mathbf{W},b}(\mathbf{x}) \right\|^2 = -(x_i - a_i^{(3)}) \cdot f'(z_i^{(3)})$$

Step.3 For each node i in the hidden layer, set

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)})$$

Step.4 Compute

$$\frac{\partial}{\partial W_{ij}^{(2)}} \frac{1}{2} \left\| h_{\mathbf{W},b}(\mathbf{x}) - \mathbf{x} \right\|^2 = a_j^{(2)} \delta_i^{(3)}$$
$$\frac{\partial}{\partial b_i^{(2)}} \frac{1}{2} \left\| h_{\mathbf{W},b}(\mathbf{x}) - \mathbf{x} \right\|^2 = \delta_i^{(3)}$$

Combining the above backpropagation and the gradient descent algorithm, we can estimate the parameters of the sparse autoencoder.

$$(\hat{\mathbf{W}}, \hat{\mathbf{b}}) = \arg\min_{\mathbf{W},\mathbf{b}} J_{\text{sparse}}(\mathbf{W}, \mathbf{b}) = J(\mathbf{W}, b) + \beta \sum_{j=1}^{s_2} \text{KL}(\rho || \hat{\rho}_j)$$

Denoising autoencoder

The second kind of overcomplete autoencoders is the **denoising autoencoder** (Vincent et al., 2008). The denoising autoencoder is required to reconstruct the original clean images from the randomly corrupted version of input images. It attempts to make the neural network be robust to meaningless perturbation of input images. More precisely, the denoising autoencoder tries to minimize the following denoising criterion

$$J_{\text{denoising}}(\mathbf{W}, \mathbf{b}) = \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{W,b}(\tilde{\mathbf{x}}^{(i)}) - \mathbf{x}^{(i)} \right\|^2 \right) \right]$$

where $\tilde{\mathbf{x}}^{(i)}$ is a corrupted version of $\mathbf{x}$ which can be made by a particular stochastic corruption process or simply injecting random noise to $\mathbf{x}^{(i)}$.

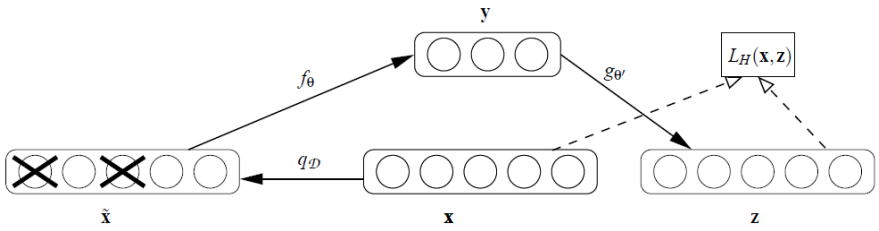


Figure 3.2: Denoising autoencoder (Vincent et al., 2010)
Using a stochastic corruption process ($q_\mathcal{D}$), the denoising autoencoder makes a $\tilde{\mathbf{x}}$, a corrupted version of $\mathbf{x}$. Then, the denoising autoencoder maps it to $\mathbf{y}$ and attempts to reconstruct $\mathbf{x}$, producing reconstruction $\mathbf{z}$. Reconstruction error is measured by loss $L_H(\mathbf{x}, \mathbf{z})$.

In this paper, we adapt the stochastic corruption process of Vincent et al. (2008), in which the process randomly sets some portion of values in each input image to zero. However, one can also simply inject the Gaussian random noise $\left\{\epsilon^{(i)}\right\}_{i=1}^{n} \sim i.i.d.\ \mathrm{N}(0, \sigma^2 I)$ to original images, which is convenient for mathematical analysis. Corrupted input images prevent the denoising autoencoder from simply imitating identity function. Roughly speaking, This method forces the denoising autoencoder to be insensitive to meaningless random perturbation of $\mathbf{x}$ and, at the same time, it makes the denoising autoencoder be sensitive to meaningful transformation of $\mathbf{x}$ in its support to reconstruct original images successfully. This explanation is based on the following theorem.

Theorem 1 (Alain and Bengio (2012)) *Let p be the probability density function of the data. Consider a population version of denoising autoencoder using the expected L_2 loss and corrupted $\tilde{\mathbf{x}} = \mathbf{x} + \epsilon$, with $\epsilon \sim \mathrm{N}(0, \sigma^2 I)$. If we assume*

that the non-parametric solutions $h_\sigma(\mathbf{x})$ *satisfies*

$$h_\sigma(\mathbf{x}) = \mathbf{x} + o(1) \quad as \quad \sigma \to 0$$

then, we can rewrite the loss as

$$J_{denoising}(h_\sigma) = \mathrm{E}\left[\|h_\sigma(\mathbf{x}) - \mathbf{x}\|_2^2 + \sigma^2 \|\frac{\partial h_\sigma(\mathbf{x})}{\partial \mathbf{x}}\|_F^2\right] + o(\sigma^2) \quad as \quad \sigma \to 0$$

The above theorem says that as we impose severer noise on input $\mathbf{x}$ (corresponding to larger σ^2), the denoising autoencoder have more contractive derivatives at input values. Since the denoising autoencoder is asked to reconstruct the original images, it should have larger derivatives at the points being far from input image space. This explanation is illustrated in Figure 3.3. Note that this regularization condition makes the denoising autoencoder more non-linear.

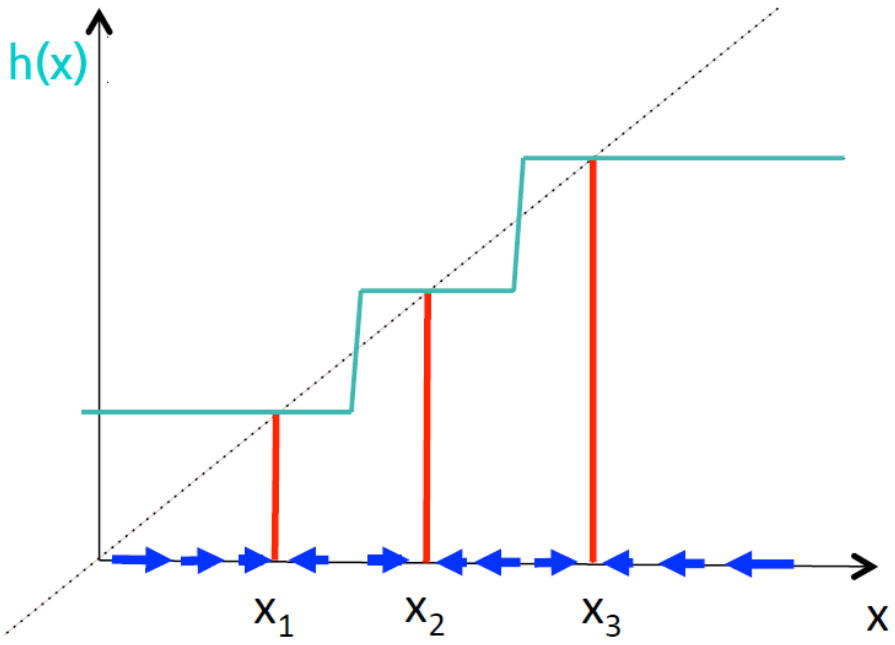


Figure 3.3: Plot of denoising autoencoder(Alain and Bengio, 2012). The input space is concentrated in three points and dashed line indicates identity function. The arrows shows the vector field of $h(\mathbf{x}) - \mathbf{x}$ pointing towards high density peaks.

Chapter 4

Stacked Autoencoders

In this chapter, we explain how to stack up autoencoders and use fine-tuning method for the purpose of making a high-performance image classifier.

4.1 Deep networks

When the perceptron - a neural network with no hidden layer - was devised, it was criticized that it cannot solve even the "XOR" problem, one of the simplest non-linear tasks. It is because that the perceptron is just a linear classifier, so it can only be successfully applied to almost linearly separable dataset. When a hidden layer is added to the perceptron, however, the neural network has a capacity to describe such linearly inseparable situation. Motivated by this experience, lots of studies have been conducted on neural networks with multiple hidden layers. However, these studies had not been successful for neural networks with more than two hidden layers until the mid-2000s. There are at least two issues in training neural networks.

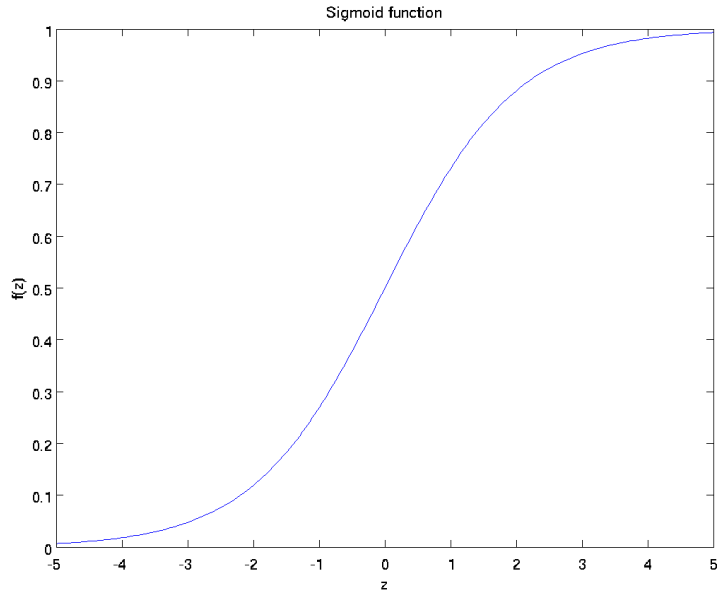


Figure 4.1: Plot of Sigmoid function(Ng, 2014).

- **Starting Values**

  Since neural networks are very complex and have many local minima in its risk function, they are sensitive to choice of starting values. In training feedforward neural networks, starting values for weights are usually chosen to be random values near zero. Since the sigmoid function (Figure 4.1) is roughly linear near zero, the neural network is nearly linear model at the beginning. As the weights increase, it becomes nonlinear model. Hence, too small starting values make the model be not able to learn the non-linearity of the given task sufficiently. Also, too large starting values often lead to poor solutions because of local minima in its risk function.

- **Diffusion of gradients**(Ng, 2014)

  Since we use backpropagation to compute the derivatives in the gradients

descent algorithm, the gradients that are propagated from the output layer to hidden and input layers of the network rapidly diminish as they are passing the layers. Hence, the derivative of the overall cost with respect to the weights in the earlier layers is very small. It makes the weights of the earlier layers change slowly, and the earlier layers fail to learn much. A closely related problem to the diffusion of gradients is the "saturation" problem. When the last hidden layer (closest to the output layer) has a enough number of units, the information from the derivative of the overall cost saturate only the last hidden layer. As a result, the neural network with multiple hidden layers does not often show higher performance than the neural network with a single hidden layer.

For the above reason, most studies on neural networks were concentrated on neural networks with a single hidden layer. In the mid-2000s, however, Hinton et al. (2006) made a breakthrough in these issues. He and his co-authors first successfully adapt the greedy layer-wise estimating method for estimating deep networks. The "greedy layer-wise" means that it tries to estimating sequentially each layer's parameters while fixing other layer's parameters, unlike conventional neural networks estimate whole parameters simultaneously. Although Hinton et al. (2006) used restricted Boltzmann machines - a undirected graphical model having similar architecture to autoencoder - as building blocks of deep networks, many deeplearning researches have been conducted based on various kinds of building blocks. In this paper, we follow Vincent et al. (2010) in which the denoising autoencoder is used as a building block. The denoising autoencoder has direct connection with conventional neural networks and, thus, it is easy to understand the connection between building blocks and the whole network.

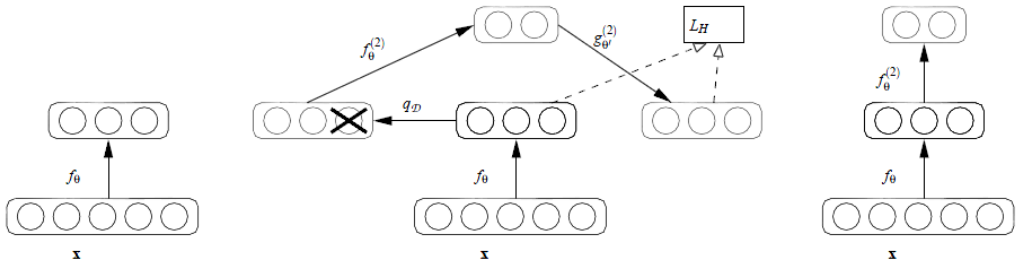


Figure 4.2: Stacking denoising autoencoders(Vincent et al., 2010).

4.2 How to stack up autoencoders

Stacking method for denoising autoencoders is similar to other stacking methods for different building blocks. First, estimate a denoising autoencoder using input images. In this step, we use corrupted input images. Then, calculate the values of hidden units in the previously estimated autoencoder. In this calculation we should use the uncorrupted input images to get representations of original images. After getting representations of input images, estimate another denoising autoencoder using the representations as new input values. From there, the procedure can be repeated until we get enough number of estimated autoencoders. After then, stack up them and add a (multi-class) logistic layer on the top to construct a multi-layer feedforward neural network. Finally, estimate the constructed multi-layer neural network via the gradient descent and the backpropagation algorithm using previously estimated weights of each layers as an initial value for the weights in whole network. This finally step is called **fine-tuning** and the previous steps are called **pre-training**. The entire process is illustrated in Figure 4.2. Note that the only difference from the multi-layer feedforward neural network in Chapter 2 is the method of initializing its weights.

Chapter 5

Application

In this chapter, we apply stacked denoising autoencoder to MNIST database of handwritten digits(LeCun and Cortes, 1998). It consists of a training set of 60,000 examples, and a test set of 10,000 examples. Each digit is size-normalized and centered to fit in a 28x28 pixel box.

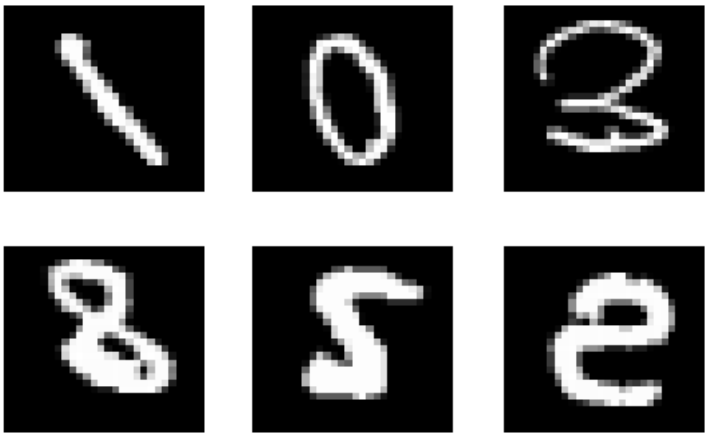
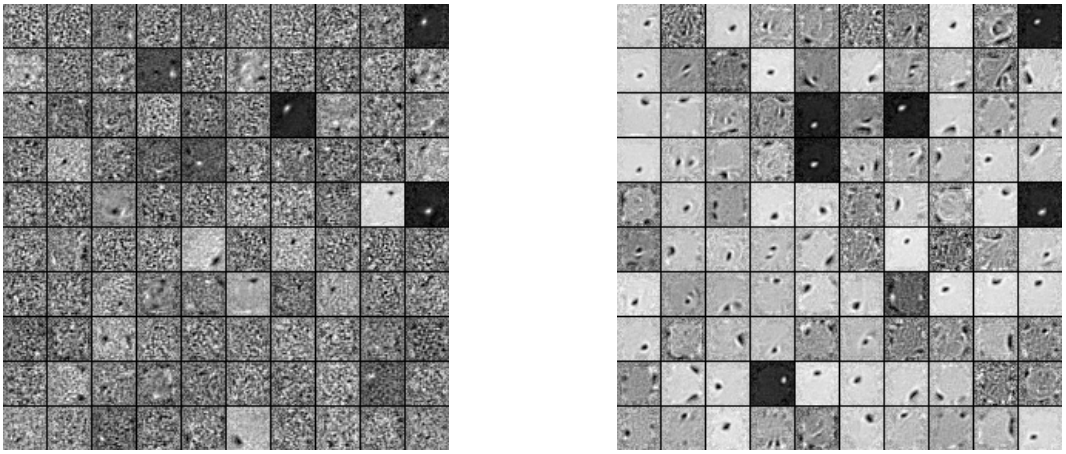


Figure 5.1: Sample images of MNIST database of handwritten digits



(a) Corruption level: 0%

(b) Corruption level: 20%

Figure 5.2: Visualization of weights in denoising autoencoders

Before we apply stacked denoising autoencoder, we first investigate what denoising autoencoder learn about the MNIST dataset. To follow Vincent et al. (2008), we randomly set 20 percent of values in each input image to zero. Using this stochastic corruption process, we estimate a denoising autoencoder with 500 hidden units. Visualization is one of the most efficient way to get the feel of what the autoencoder learned. Let the output of i-th hidden unit is

$$a_i^{(2)} = f\left(\sum_{j=1}^{784} W_{ij}^{(1)} x_j + b_i^{(1)}\right)$$

Under the norm constraint $||\mathbf{x}||^2 = \sum_{i=1}^{784} x_i^2 \leq 1$

$$\underset{\mathbf{x}:\ ||\mathbf{x}||^2 \leq 1}{\arg\max}\, a_i^{(2)} = \left(\frac{W_{i1}^{(1)}}{\sqrt{\sum_{j=1}^{784}(W_{ij}^{(1)})^2}}, \ldots, \frac{W_{i784}^{(1)}}{\sqrt{\sum_{j=1}^{784}(W_{ij}^{(1)})^2}}\right)$$

Hence, if we plot each hidden unit's normalized weights matrix, we can see what kinds of input images each hidden unit reacts most actively (yielding the largest output value). The resulting weights matrix plot is illustrated in Figure

5.2. Note that the weight matrix plot of autoencoder using corrupted input shows cleaner images. This is the reason why the autoencoder using corrupted input is called **denoising** autoencoder. Also note that each hidden units in denoising autoencoder can capture geometrical features of input images such as edges and contours while hidden units of simple autoencoder is not good at capturing features.

Now, we apply stacked denoising autoencoder to MNIST database of handwritten digits. Stacked denoising autoencoder includes various tuning parameters such as the number of hidden layers and units, the corruption level in the corruption process. In general, cross-validation technique is used to choose tuning parameters. For computational feasibility, however, we use tuning parameters setting in Vincent et al. (2010) and Theano-Development-Team (2013). To compare its performance with other methods, we also apply (multi-class) logistic model, feedforward neural networks with single hidden layer to MNIST dataset. For the logistic model, we set the learning rate to 0.13. To estimate feedforward neural network, we set the number of hidden units to 500, the tuning parameter for the L_2 penalty term to 0.0001, and the learning rate to 0.01. To estimate stacked denoising autoencoder, we set the number of hidden layers to 3. Each hidden layer consists of 1000 hidden units. We set the corruption rate for the first layer to 0.1, for the second to 0.2 and 0.3 for the last layer. While conducting pre-training process, we fixed the learning rate to 0.001. And for the fine-tuning part, we set the learning rate to 0.1. The testing errors for the three models are summarized in Table 5.1. In this application, we used Theano package and the Python codes in *Deep Learning Tutorials*(Theano-Development-Team, 2013). We can see that stacked denoising autoencoder shows the highest prediction accuracy among three models.

Table 5.1: Test errors of three models

Model	**Test error**	**Tuning parameter**
Logistic	**7.501%**	Learning rate : 0.13
Feedfoward neural network	**1.650%**	Learning rate : 0.01 Number of hidden units : 500 L_2 penalty parameter : 0.0001
Stacked denoising autoencoder	**1.322%**	Pre-training learning rate : 0.001 Fine-tuning learning rate : 0.1 Number of hidden layers : 3 Number of hidden units in each layer : 1000 corruption rate for the first layer : 0.1 for the second : 0.2; for the last layer : 0.3

Chapter 6

Conclusion

In this paper, we have studied feedforward neural networks, autoencoders and stacked denoising autoencoders. We noticed that stacked denoising autoencoder showed higher classification performance than multi-layer feedforward neural networks did. Since the only difference between the stacked denoising autoencoder and multi-layer feedforward neural network is the method of initializing its weights, it is surprising that these two methods provided significantly different results. This phenomena has been reported by many researches in various fields such as classification, regression, dimensionality reduction and natural language processing tasks. This gives us a lesson about the importance of initialization in non-convex estimating problems.

When investigating what kinds of data the deeplearning methods works well, we can notice that deeplearning has its strength in analyzing natural images or sounds dataset. They share a common feature that randomly generated matrix or sequences will probably be not seemed or sounded like natural things. It means that although these kinds of data are high-dimensional, they might be embedded in low-dimensional manifold-like structures. Also, since

slight transformation of images or sounds yield huge difference in their mathematical representations, these manifold-like structures might be highly curved. In this perspective, we can think that building blocks of deep networks like denoising autoencoders, which implicitly learn about manifold-like structures of input space (Alain and Bengio, 2012), help to find alternative representations of input values. Since these alternative representations could be more sensitive to changes on input space and robust to meaningless perturbations, they will be helpful not only to reconstruct input values, but also to propagate information from derivative of cost function to whole model units. There are few mathematical explanation of this phenomena. As a further topic, It will be interesting to study this phenomena rigorously in the statistical point of view.

References

Alain, G. and Bengio, Y. (2012). What regularized auto-encoders learn from the data generating distribution. *arXiv preprint arXiv:1211.4246*.

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

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

Boyd, S. and Vandenberghe, L. (2009). *Convex optimization*. Cambridge university press.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*. ACM, 160–167.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, **2** 303–314.

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

Hinton, G. E. and Salakhutdinov, R. (2008). Using deep belief nets to learn covariance kernels for gaussian processes. In *Advances in neural information processing systems*. 1249–1256.

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

LeCun, Y. and Cortes, C. (1998). The mnist database of handwritten digits.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, **5** 115–133.

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

Ng, A. (2014). Ufldl tutorial. `http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial`. Accessed: 2014-10-15.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, **65** 386.

Theano-Development-Team (2013). Deep learning tutorials. `http://deeplearning.net/tutorial/index.html`. Accessed: 2015-01-10.

Vincent, P. (2011). A connection between score matching and denoising autoencoders. *Neural computation*, **23** 1661–1674.

Vincent, P., Larochelle, H., Bengio, Y. and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In

Proceedings of the 25th international conference on Machine learning. ACM, 1096–1103.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y. and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, **11** 3371–3408.

Werbos, P. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences. *PhD thesis*.

국문초록

적층 자가인코더(Stacked autoencoder)는 비지도 깊은 학습 알고리즘의 일종으로 이미지의 선이나 물체와 같은 입력 자료의 특징들을 “자동적으로 학습”하는 것 같은 모습을 보인다. 다양한 종류의 적층 자가인코더들이 다층 신경망 모형을 구축하고 초기화하기 위해 사용되어 왔다. 적층 자가인코더들에 기반을 둔 신경망 모형들은 자연 이미지 분류문제나 음성 인식 문제에 있어 독보적인 성능을 보인다. 이 논문은 그 중 이미지 분석 문제에 초점을 두자 한다. 이 논문은 먼저 신경망 모형과 자가인코더를 소개하고 특히 자가인코더가 어떤 학습을 하는지에 대한 개략적인 설명을 제공한다. 이후엔 이러한 자가인코더들을 어떤 방법으로 적층할 수 있는지와 높은 성능의 이미지 분류기를 만들기 위한 지도학습적 활용 방법에 대해 소개한다. 끝으로 이 논문은 MNIST 손 글씨 숫자 데이터에 이 방식들을 적용한 결과를 제시하였다.

주요어 : 적층 자가인코더, 디노이징 자가인코더, 신경망 모형, 깊은 학습

학　번 : 2013-20219