USE OF DROPOUTS AND SPARSITY FOR REGULARIZATION OF AUTOENCODERS IN DEEP NEURAL NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

By
Muhaddisa Barat Ali
January, 2015

USE OF DROPOUTS AND SPARSITY FOR REGULARIZATION OF AUTOENCODERS IN DEEP NEURAL NETWORKS

By Muhaddisa Barat Ali

January, 2015

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ömer Morgül (Advisor)

Prof. Dr. Enis Çetin

Assoc. Prof. Dr. Selim Aksoy

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

USE OF DROPOUTS AND SPARSITY FOR REGULARIZATION OF AUTOENCODERS IN DEEP NEURAL NETWORKS

Muhaddisa Barat Ali
M.S. in Electrical and Electronics Engineering
Advisor: Prof. Dr. Ömer Morgül
January, 2015

Deep learning has emerged as an effective pre-training technique for neural networks with many hidden layers. To overcome the over-fitting issue, usually large capacity models are used. In this thesis, two methodologies which are frequently utilized in deep neural network literature have been considered. Firstly, for pre-training the performance of sparse autoencoder has been improved by adding p-norm of the sparse penalty term to an over-complete case. This efficiently induces sparsity to the hidden layers of a deep network to overcome over-fitting issues. At the end of the training, features constructed for each layer end up with a variety of useful information to initialize a deep network. The accuracy obtained is comparable to the conventional sparse autoencoder technique.

Secondly, the large capacity networks suffer from complex co-adaptations between the hidden layers by combining the predictions of each unit in the previous layer to generate the features of the next layer. This results to certain redundant features. So, the idea we propose is to induce a threshold level on the hidden activations to allow only the highest active units to participate in the reconstruction of the features and suppressing the effect of less active units in the optimization. This is implemented by dropping out k-lowest hidden units while retaining the rest. Our simulations confirm the hypothesis that the k-lowest dropouts help the optimization in both the pre-training and fine-tuning phases giving rise to the internal distributed representations for better generalization. Moreover, this model gives quick convergence than the conventional dropout method.

In classification task on MNIST dataset, the proposed idea gives the comparable results with the previous regularization techniques such as denoising autoencoders, use of rectifier linear units combined with standard regularizations. The deep networks constructed from the combination of our models achieve favorably the similar state of the art results obtained by dropout idea with less time complexity making them well suited to large problem sizes.

Keywords: Deep learning, regularization, sparse autoenoder, dropout.

ÖZET

DERİN SİNİR AĞLARINDA OTO-KODLAYICININ DÜZENLENMESİ İÇİN TERKİNİM VE SEYREKLİK KULLANIMI

Muhaddisa Barat Ali
Elektrik ve Elektronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: Prof. Dr. Ömer Morgül
Ocak, 2015

Derin öğrenme, birçok saklı katmandan oluşan sinir ağları için etkili bir öncül eğitme yöntemi olarak ortaya çıkmıştır. Aşırı beslemenin problemlerini yenebilmek için genellikle kapasite modelleri kullanılır. Bu tezde, derin sinir ağları konusunda sıklıkla uygulanan iki metot ele alınmaktadır. İlk olarak, öncül eğitimde seyrek oto-kodlayıcının performansı aşırı tamamlanmış durumlara seyrek ceza teriminin p-normu eklenerek geliştirilmiştir. Bu durum, aşırı besleme sorunlarının üstesinden gelebilmek için derin sinir ağındaki saklı katmanlarda etkili şekilde seyrekliğe sebep olur. Eğitme sonunda, her bir katman için derin ağı başlatacak geniş çeşitlilikte özellikler elde edilir. Elde edilen hatasızlık bilinen seyrek oto-kodlayıcı yöntemleri ile karşılaştırılabilir durumdadır.

İkinci olarak, bir sonraki katmanın özelliklerini belirlemek için önceki katmanlardaki her bir birime ait tahminlerin birleştirilmesi sonucu geniş kapasite ağları, saklı katmanlar arasındaki karmaşık yardımcı-uyarlamalara katlanmak zorunda kalırlar. Bu durum ihtiyaç fazlası özelliklerin doğması ile sonuçlanır. Bu nedenle, önerilen fikir sadece en yüksek aktif birimleri özelliklerin belirlenmesinde kullanıp, eniyileme sırasında daha az aktif birimlerin etkisini bastırabilmek için saklı aktivasyonlar üzerinde bir eşik değeri belirlenmesidir. Bu işlem k-en düşük saklı birimi ihmal edip diğer birimleri tutarak uygulanabilir. Yapılan simülasyonlar k-en düşük terkinimin, daha iyi genellemeler için içsel dağılım gerçeklemelerine sebep olan öncül eğitme ve ince ayar eniyilemelerine yardımcı olduğu tezini doğrulamaktadır. Üstelik bu model bilinen terkinim metoduna kıyasla daha hızlı yakınsamaktadır.

MNIST bilgi setinin sınıflandırılmasında önerilen teknik, oto-kodlayıcıları

gürültüden arındırma, lineer doğrultucu birimlerin normal düzenleyiciler ile beraber kullanımı gibi önceki düzenleme teknikleri ile karşılaştırılabilir sonuçlar vermektedir. Önerilen modellerin bileşimi kullanılarak oluşturulan derin ağlar, terkinim fikrinin temel sonuçlarına oldukça benzer ama özellikle geniş boyutlu problemler için çok daha az vakit alan çıktılar vermektedir.

Anahtar sözcükler: Derin öğrenme, düzenleme, terkinim, seyrek oto-kodlayıcı.

Acknowledgement

I would express my sincere gratitude to my supervisor Prof. Dr.Ömer Morgül for being a great mentor to introduce me into the wonderful field of academic research and to encourage me to explore new fields and research directions. His guidance, insightful suggestions and tireless assistance throughout my work is remarkable. I feel lucky to be one of his students.

I would like to thank Prof. Dr. Enis Çetin and Dr. Selim Aksoy for being members of my thesis committee.

I kindly appreciate the struggle of my all instructors during my graduate study. Specially, Prof. Dr.Ömer Morgül and Dr. Çigdem Gunduz Demir are two of my admirable professors triggering my enthusiasm towards Artificial Neural Networks

I am thankful to Ismail Uyanık for his timely co-operation and arranging computer for me to carry out my experiments.

I would like to express my sincere gratitude to my best friend Huma Shehwana who has been a great companion to support, to entertain and to help me get through this agonizing period in the most positive way. I would also acknowledge the proper guidance provided by my lab fellow Caner Odabaş. Without their moral support, I would not have maintained my motivation till the end.

I am also grateful to Bilkent University, Electrical and Electronic Engineering Department and Higher Education Commission of Pakistan for funding my graduate study.

Most importantly I would like to thank my family and my husband Mr. Zakir Hussain. Without their support and encouragement I would have not been able to complete my Master of Science study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Image Recognition and Learning Models

Pattern recognition and computer vision is becoming popular for thousands of applications like search engines, autonomous robots and medical applications such as cancer detection. The recognition task requires the interpretation of images. The study of these problems are much popular in machine learning where image interpretations do not base on raw pixel values but some intermediate higher level representations. This task resembles the way humans solve complicated problem by decomposing it into subproblems. The common dilemma is to extract features for training the classifier that are in some way robust to rotation, distortion, lighting etc. Success of the recognition process depends highly on the quality of these features. Difficult artificial intelligence problems with high factors of variations can be solved by promising tools such as representation learning algorithms [3]. Let $X \in \mathbb{R}^m$ denote a random input vector and $Y \in \mathbb{R}^n$ a random output vector. A learning/predictive model is to approximate a function $F : X \in \mathbb{R}^m \rightarrow \mathbb{R}^m$ such that $Y = f(X)$, where Y is the prediction when X is given at the input. The learning capability of a model is estimated using a loss function which penalizes errors in prediction. We will consider a learning model which solves recognition tasks by using same strategy of decomposition.

1.2 Deep Neural Networks and Deep Learning

Deep Neural Networks try to mimic the complex human learning systems, extracting features at different levels of abstraction from the input without depending completely on human-crafted features. Neural network consisting of input layer, a hidden layer and an output layer is called as shallow or vanilla network where features are computed using only one hidden layer of computation without any regularization, variable selection or committee techniques [4]. When a network has multiple hidden layers, theoretically it is thought to allow computation of more complex features of the input. Each hidden layer computes complex transformation of the previous layer and hence generates a deep model[5]. But in practice, networks with 3 or more hidden layers are found to stuck at poor solutions. In such deep architectures, logistic sigmoid activations drives the upper layers into saturation [6]. So, as the gradients propagates from layer to layer, they shrink exponentially with the number of layers, " called as vanishing gradient problem" [7]. That is why deep vanilla neural networks can't learn a model that makes higher levels of hierarchy from the composition of lower layers. Recently, Hinton's break through [8] proposed a solution to the training problem of deep vanilla neural networks. A greedy layer wise unsupervised training algorithm was proposed which will be discussed in chapter 2. Soon after, related algorithms based on auto-encoders were introduced [9]. In this thesis, we discuss three steps for training deep neural networks in different ways and compare the role of each method with experimental evidence:

1. Pre-training each layer in a greedy way.
2. Using unsupervised learning for each layer in order to preserve useful information from input at each layer.
3. Fine-tuning the whole network with some criteria of interest.

The results obtained from the simulation prove this strategy to be better than traditional random initialization of weights in supervised multilayer perceptron networks (MLP).

1.2.1 Related Work

The early development of deep learning in 2006 focused on MNIST hand written digit classification problem [8] breaking the supremacy of SVMs (1.4 % error) on this dataset. The latest accuracy rates are still won by deep networks [10]. The two most commonly used methods are restricted Boltzmann machine (RBMs)[11, 12] and autoencoders [9, 13]. These models have been used with various modifications to learn better features. The learning process includes starting from a randomly initialized model (often a neural network), optimizing its hyperparameters according to an unsupervised training criterion, stacking output of one model as the input of the other to construct a deep network and then further improving it with a supervised fine-tuning step. The final model is capable to learn higher level abstractions since the data goes through several non-linear transformations. Deep learning research focuses to find ways of guiding the learning process to learn a good generalization of the data and avoid overfitting.

One of the well known unsupervised learning systems is the encoder-decoder paradigm also know as Autoencoder. An encoder transforms the inputs into a feature vector known as the code vector which is decoded by the decoder to reconstruct the input from the coded representation. The autoencoder architecture is useful for two reasons [9]:

1. After training, computing the code/feature vector is obtained by running the input through the encoder.

2. Running the code vector through decoder reconstructs the input that provides to check if the code has captured the relevant information in the data. [14].

Some algorithms do not have a decoder and in order to provide reconstruction, they have to proceed through expensive Markov Chain Monte Carlo (MCMC) sampling methods [15]. To find the code vector, other learning algorithms without encoders need to go through an expensive optimization algorithm [16]. In this

work, we will focus only on autoencoder as a building block of a deep neural architecture.

One practical approach to improve the learning process can be to change the regularization in the objective function to get a good feature representation. For example by adding a sparsity constraint, a good sparse feature representation has been obtained [11, 14, 17]. Another example is the contractive Auto-encoder that learns by penalizing changes in the feature representation which becomes robust to the small changes in the input [18]. A similar effect has been observed in denoising auto-encoder where features are obtained by learning to reconstruct inputs from the noisy inputs [1, 19]. Recently another technique called "dropout" has been proposed for avoiding overfitting in which the complex co-adaptation between the layers are prevented by randomly dropping 50% of the hidden units for each training example [2]. This method allows each hidden unit learn meaningful representations independently which the other units have learned and hence the mistakes one unit might bring is not dependent on being fixed by others.

1.2.2 Motivation

Over the last decade, researchers across different communities have proposed several models which generalize well on wide variety of recognition tasks. These models have proved state-of-the-art results in many challenging learning problems, where data exhibits high degree of variations and have been successfully implemented in many applications. So these works give us an idea of how powerful deep neural networks are as a machine learning tool. Previously, Hinton obtained an error rate of 1.25% after six weeks of training on MNIST data set [8]. Moreover, studies on brain energy consumption suggest that neurons encode messages in a sparse and distrubuted way [20]. To take the advantage of sparse representation and to shrink weeks of training to few hours motivate us to propose our model. Our work is mainly inspired by the ideas proposed by [17, 21, 22]. The aim of this thesis is to obtain similar accuracy results that is already obtained in the literature [2, 21]. We observed the performance of our

model using benchmark MNIST data set.

1.2.3 Contribution

Model combination nearly always improves the performance of machine learning methods. Combining several models always prove to be helpful specially when they are different individually. For deep network model, our contribution consists of training the model in different following possible ways.

1. To enforce sparsity on an autoenocder, we propose a model with a change in sparsity constraint of a sparse autoencoder. Previously, the candidate used for the penalty function was KL (Kullback-Leibler) divergence [17] that was used to penalize the objective function. We introduce the p-norm of the penalty function instead and find the performance better than the former function. This method can be applied for the pre-training of each autoencoder to initialize each hidden layer of a deep network.

2. To promote the dense network towards sparser structure, hidden neurons are dropped out randomly [2, 21]. This technique takes longer training time due to random sampling of neurons and causes noise generation. We have suggested a variant of this method, where instead of random dropouts neurons with lowest activations are dropped out. In this way, neurons with high enough activations take part in the reconstruction of the next layer features during each training epoch.

3. Model performance is analyzed by pre-training the autoencoder through p-norm sparse autoencoder. The pre-trained weight vectors thus obtained are used for the initialization of a deep network. This deep network is further fine-tuned by stochastic gradient descent (SGD) along with inducing regularization by k-lowest dropouts on each hidden layer. The final model gives us similar accuracy as obtained by the dropout technique [2, 21] but it took less convergence time. k-lowest dropout technique is also applied on training of autoencoders for pre-training phase. The learned weights

are further used to initialize a deep neural network which is fine-tuned through simple SGD. This also results about the same misclassification rate as obtained before.

1.2.4 Organization of Thesis

The work implemented by this thesis is presented in five Chapters and one Appendix. The first chapter presents the main objectives of the thesis and a brief description of the problem and related work. Chapter 2 presents the theoretical background of the concepts and different models used . Chapter 3 presents our proposed model and the hype-parameters used. Chapter 4 covers full description of the problem and simulation results obtained from different models and our proposed model. The comparison is evaluated as well. Chapter 5 ends with general conclusions of the thesis and future works, what has been done and what can be done in the future development of the model. In Appendix section, one may find description of the algorithms used in this thesis.

Throughout the thesis, notations used are given in Table 1.1 and abbreviations used are given in Table 1.2.

Table 1.1: Notation used in the thesis

Notation	Definition
W^l	weight matrix of the layer l
b^l	the bias vector of the layer l
s_l	the size of the layer l
l	the number of layers of a network
$z_{(l)}$	The weighted sum of inputs to units in layer l
x	the input training set
y	the label/target output set
a^l	activation of layer l
$\sigma(.)$	sigmoid activation function
$g(.)$	softmax activation function

$J(.)$	error function of the model
λ	the weight decay parameter
ρ	the sparsity parameter
$\hat{\rho}$	the empiric sparsity of the network
α	the weight for sparseness
β	the momentum
ϵ	the learning rate
δ_l	the gradient of the layer l
$\hat{y}$	predicted output label
$\mathcal{L}$	the loss function
C	size of the output layer
$v^{(i)}$	input vector of an autoencoder
$\hat{v}^{(i)}$	reconstructed vector of an autoencoder
$h^{(i)}$	hidden vector of an autoencoder
W'	weight matrix of decoder layer
b'	bias vector of decoder layer
ω	p-norm of sparse penalty
γ	learning parameter of bias of sparse autoencoder
q_D	mask corruption with probability D
Γ	set of k-lowest dropouts
Γ^c	retained set of hidden units
k	number of lowest dropout in hidden layer l
$\bar{k}$	set of highest retained hidden units
ℓ_1	L_1 regularization
ℓ_2	L_2 regularization
p	value of norm for sparse penalty
C	class label
$\mathbb{D}$	dataset
K	fold value for cross validation
h	combination function of a neuron
B	block of mini-batch
O	computational complexity

Table 1.2: Abbreviations used in the thesis

Abbreviation	Description
AE	Autoencoder
MCMC	Markov Chain Monte Carlo
SGD	Stochastic Gradient Descent
RBM	Restricted Boltzmann Machine
SAE	Stacked Autoencoder
DA	Denoising Autoencoder
SDA	Stacked Denoising Autoencder
ReLUs	Rectifier Linear Units
SpAE	Sparse Autoencoder
FT	Fine-tuning
MLP	Multilayer Perceptron
KL	Kullback Leibler
GPU	Graphics Processing Units
CUDA	GPU Accelerated Libraries
L-BFGS	Low-memory Broyden Fletcher Goldfarb Shanno

Chapter 2

Theoretical Background

Deep multi-layer neural networks have many layers of non-linearities representing highly varying functions. Before a decade, it was not clear how to train such deep architectures, because gradient based optimization that used to start from random weight initialization appeared to yield poor solutions.

The best results obtained on supervised learning tasks often involve an unsupervised learning phase. The unsupervised pre-training appears to play a regularization role (though not in the usual sense) and an aid to optimization in a supervised learning. The break through for deep architectures like Deep belief Networks [12] and stacked auto-encoder [13] came in 2006 are based on greedy layer-wise unsupervised pre-training followed by supervised fine-tuning. Each layer is pre-trained for nonlinear transformation of its inputs that learns the main variations in its input. An alternative meaning is that pre-training helps for initializing the network in a region of the parameter space where a better local optimum is found. It sets the stage for a final training phase where the deep network is fine tuned using supervised training criterion with a gradient based optimization. These findings put deep networks among the realm of semi-supervised learning methods at a unique place [23] and a model to bring better generalization. It reduces mean test error and its variance for sufficiently large models.

2.1 The Perceptron Neuron Model

An artificial neuron is a mathematical model of the behavior of a biological neuron. Perceptron is one of the basic neuron models which is widely used in literature [4, 24]. It receives information in the form of a set of numerical input signals which is integrated with a set of free parameters to produce an output signal. Figure 2.1 presents a graphical view of the perceptron where $x_1, x_2, ..., x_n$ are the input signals and y as the output. A set of free parameters consists of a bias b and a vector of weight values $w_1, w_2, ..., w_n$. A combination function s sums all the weighted input signals to produce a net signal z. An activation function $f(.)$ takes z as an argument and produce y. The set of free parameters $\theta = \{b, w\}$ allows a neuron model to learn to perform a task [4]. In other words, θ can be assumed as parameterized function space mapping input $X \in \mathbb{R}^n$ to an output $Y \in \mathbb{R}$.

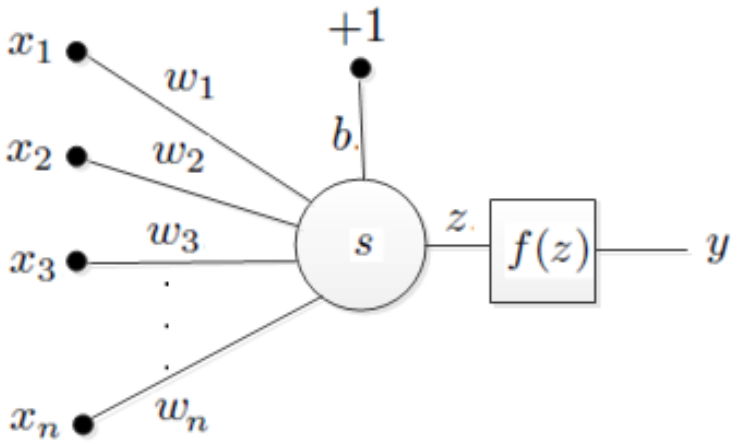

Figure 2.1: Perceptron neuron model

Mathematically, a perceptron neuron model can be represented by the expression given as:

$$y = f(b + \sum_{i=1}^{n} w_i x_i), \tag{2.1}$$

where $f(.)$ can be any activation function out of many choices discussed in

detail in Section 2.2.

2.2 Activation Function

The activation function $f(.)$ takes the net input of a neuron z and outputs signal y. In practice, we can take varieties of functions, few of the most commonly used activation functions are defined here [4, 25].

2.2.1 The Sigmoid

Logistic unit as a feature detector

The output $f(z)$ of a neuron with logistic function is bounded by 0 and 1 nonlinearly, which is fully differentiable:

$$f(z) = \frac{1}{1 + \mathrm{e}^{-z}} \quad where \quad z = b + \sum_{i=1}^{n} x_i w_i, \tag{2.2}$$

Logistic unit as a feature detector accepts x as an amount of evidence for a certain feature. It gives $f(z)$ as the probability that the feature is present. If there is evidence in the favor of the feature, z is positive and it becomes negative if favor is against. Every extra amount of evidence for the feature increases the probability $f(z)$ of its presence. The slope of this function is highest at $f(z) = 0.5$ as shown in Figure 2.2. It shows confusion for the presence of feature and any evidence in favor or against will take it away in either direction. This is how it impacts back-propagation of the gradient during training of a neural network (detail is given in Section 2.3).

Tangent hyperbolic function

A rescaled version of the logistic function is tangent hyperbolic given as:

$$f(z) = tanh(z) = \frac{\mathrm{e}^{z} - \mathrm{e}^{-z}}{\mathrm{e}^{z} + \mathrm{e}^{-z}}, \tag{2.3}$$

Its output range is [-1,1]. One of the possible problems with this activation function is that the error surface can be very flat near the origin. It can

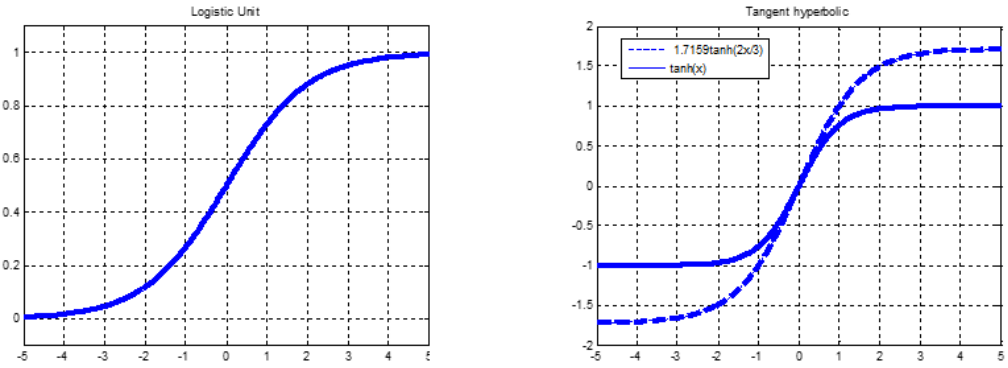

Figure 2.2: **Left:** Logistic function **Right:** Tangent hyperbolic function

be overcome by extending the vertical stretch of the function response to widen the linear region across zero origin as shown in Figure 2.2 [25]. A typical extension is given below:

$$f(z) = 1.7159\ tanh\Big(\frac{2}{3}z\Big). \tag{2.4}$$

2.2.2 Softmax Function

Softmax function is always used for output layer units in many probabilistic multiclass classification methods and multinomial logistic regression. If L is the output layer of a network and C is the output layer size, then it is a normalized exponential form which is a generalization of the logistic function that squashes a C-dimensional input vector z from the $(L-1)$ layer of any real value distribution to a C-dimensional vector $f(z)$ in the range $(0, 1)$. The function is given as:

$$\hat{y}_j = f(z) = \frac{e^{z_j(L)}}{\sum_{c \in C} e^{z_c(L)}}, \tag{2.5}$$

which is also the predicted probability for the j'th class given an input vector z. In softmax function each output depends on all the inputs, so gradient becomes a whole Jacobian matrix. An error function from each output propagates back to each input, so the whole Jacobian is to be computed.

2.2.3 Rectifier Linear Units(ReLUs)

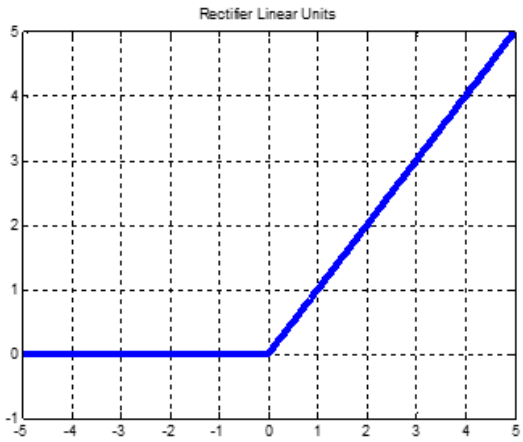

Figure 2.3: Rectifier linear unit function saturates at 0 for input values less than zero and is linear for inputs greater than 0.

ReLU is piece-wise linear activation that is neither differentiable nor bounded as shown in Figure 2.3. Its derivative can take only two values 0 and 1 due to which it easily obtains a sparse representation given as:

$$y'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

ReLUs make the training proceeds better when neurons are either off or operating mostly in a linear regime [20]. After uniform initialization of the weights, about 50% of hidden units become zero and this fraction can be increased with other sparsity inducing regularization. It speeds up the training time [26]. In some literature rectifying neurons have been defined behaving more like biological neurons than logistic units [20]. For each input only a subset of corresponding neurons are activated which is the only non-linearity in the network. The activated outputs remain linear function of the inputs. The final model can be assumed as an exponential number of linear models. Moreover, ReLU units saturate at 0 which is useful when using hidden activations as input features for a classifier. The input of a network selects a subset of active hidden neurons by ReLUs and computation becomes linear in this subset [20].

ReLU is not a helpful choice for output units because when a unit is saturated i.e.$z < 0$, no gradient propagates inside the network. But inside if some units are saturated, the gradient manages to go through a subset of other hidden units [27].

2.3 Multilayer Perceptron Network Architecture (MLP)

Like biological nervous system, an artificial neural network is composed of interconnected layers of neurons. This network is called as multilayer perceptron and can be represented with an acyclic graph that is known as feed forward architecture. Thus a multilayer perceptron network consists of an input layer, one or more hidden layer neurons and a set of output neurons as shown in Figure 2.4, where the network with neurons in feed-forward architecture are grouped into a

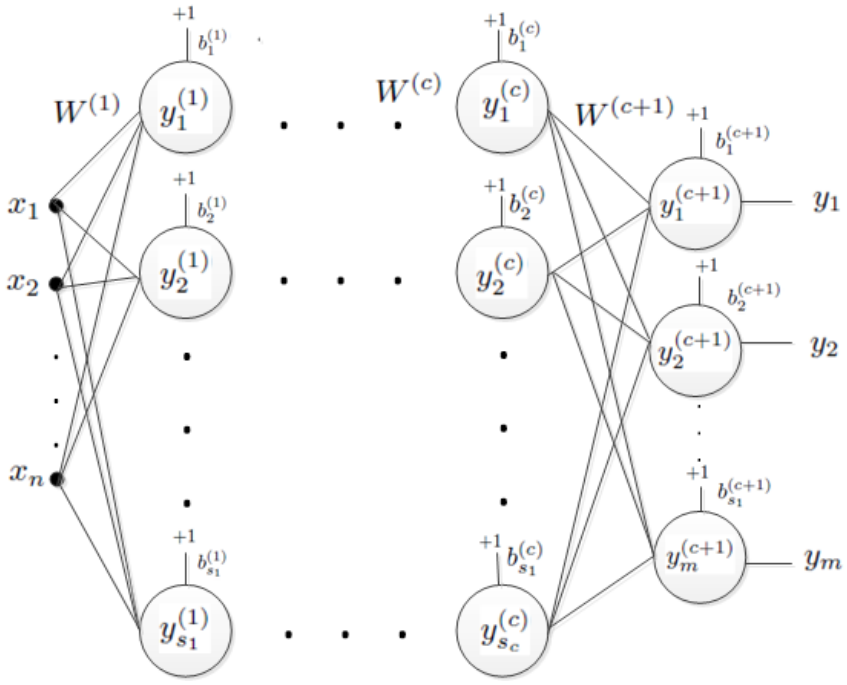

Figure 2.4: A multilayer perceptron architecture

sequence of number of layers $L = l^{(1)}, ..., l^{(c+1)}$ so that neurons in any layer is connected to every neuron in the next layer. Layer $l^{(0)}$ is the input layer consisting of n number of fan-ins and $l^{(c+1)}$ is the output layer consisting of m number of fan-outs. The hidden layers $l^{(1)}, ..., l^{(c)}$ consist of $s_1, s_2, ..., s_c$ hidden neurons. Signal flows layer by layer starting from the input layer, passes through hidden layers up to the output layer. Figure 2.4 shows a MLP network with n inputs, c hidden layers each with s_i neurons and m neurons in the output layer. The output of each neuron is a function of the inputs, so the activations of all neurons in the output layer can be found in a deterministic pass [4].

In multilayer networks, nonlinear transformations are learned by backpropagating the errors which endows these networks with universal approximation capabilities to model extremely complex tasks. These networks are able to learn their own internal representation from data $X \in \mathbb{R}^n$.

2.3.1 Feedforward Phase

A multilayer neural network learns a feedforward nonlinear mapping between input and output. The model consists of many interconnected neurons shown in Figure 2.4. Let s_l denote the size of each layer. As defined previously $n^{(c+1)}$ denotes the number of layers and $\theta = \{W, b\}$ are the free parameters, where $W^{(l)} \in \mathbb{R}^{(s_{l+1} \times s_l)}$. Let $y_j^{(l)}$ denote the representation of unit j in layer l. The function $f : \mathbb{R} \mapsto \mathbb{R}$ can be any of the activations from Section 2.2. The weighted sum of inputs $z_j^{(l)}$ to unit j in layer l, including the bias input when external input is given generally becomes:

$$z_j^{(1)} = \sum_{i=1}^{n} W_{ji}^{(1)} x_i + b_j^{(1)}, \tag{2.6}$$

$$y_j^{(1)} = f(z_j^{(1)}). \tag{2.7}$$

Similarly for next layers feed forward activations are computed accordingly as:

$$z_k^{(l+1)} = \sum_{j=1}^{s_l} W_{kj}^{(l+1)} y_j^l + b_k^{(l+1)}, \tag{2.8}$$

$$y_k^{(l+1)} = f(z_k^{(l+1)}), \tag{2.9}$$

for $k = 1, ..., s_l$ and $j = 1, ..., s_{(l-1)}$. Moreover, matrix-vector operations can benefit us to perform calculations quickly. Theoretically, different activations can be used for each neuron. However, in practice only one type of activation is used for each layer.

2.3.2 Backpropagation Algorithm

Traditional Stochastic Gradient Descent (SGD) passes one example per iteration. This successive process makes SGD challenging for distributed inference, so it is impractical for large datasets. A common practice is to employ mini-batch training, which aggregates a group of examples at each epoch. This technique speeds up the stochastic convex optimization problem and parallelizes the process. However, an increase in minibatch size decreases the rate of convergence [28]. So, care should be taken in the selection of the batch size. The gradient computed by batch SGD is less noisy (averaged over a large number of samples) and makes use of matrix operation to parallelize the process. To train the network, we would need training examples $X \in \mathbb{R}^n$ and corresponding target outputs $Y \in \mathbb{R}$. When the input x_i is presented to the network, it produces an output $\hat{y}_i$ different in general from the target y_i. We want to make $\hat{y}_i$ equal to y_i. For this purpose, parameters $\theta = \{W, b\}$ are initialized to a small random value near zero $\mathcal{N}(0, \sigma^2)$ for variance $\sigma^2 = 0.01$. By using this optimization process our aim is to minimize the function error:

$$\min_{\theta} J(\theta), \tag{2.10}$$

where $J(\theta)$ is a cost/error function to be minimized. A natural selection, which is widely used in neural network though is to update the weight in the negative gradient direction. This approach is the application of standard gradient descent method to the supervised learning of neural network. One possible way to evaluate this gradient vector is back-propagation algorithm. This objective function defines the task the neural network is supposed to perform and gives a measure of how far the actual performance is from the desired one. This is obtained by a learning procedure such as Stochastic Gradient Descent (SGD) or back-propagation in which free parameters θ are adjusted based on the learning

rule [4].

According to the back propagation algorithm to learn the input samples, the parameters $\theta = \{W, b\}$ are update after each mini batch of inputs:

$$\theta^t \leftarrow \theta^{t-1} - \epsilon_t \frac{1}{B} \sum_{t=s_t}^{s_t+B} \frac{\partial J(\theta_t, x_t)}{\partial \theta}, \tag{2.11}$$

where B determines the size of batch at iteration t and ϵ_t is the learning rate at time t. The important step is to compute the partial derivatives given in Equation 2.11 by back-propagation algorithm. The intuition is that after following feed forward pass to compute the activations throughout the network including the output values, for each node j in layer l, a gradient $\delta_j^{(c+1)}$ is computed which is responsible component for the error in the outputs. This term for the output nodes can be directly measured using any of the two criterion; mean square error (MSE) or cross-entropy (CE) which will be discussed in Section 2.3.2.1. The error criterion is often decided based on certain output activation functions that expresses the gradient $\delta_i^{(L)}$ as the difference between the actual output and the desired reference output. In probabilistic terms, for any classification problem the main purpose of a neural network is to maximize the likelihood $\mathcal{L}$ of observing the training data. It is equivalent to minimizing the negative log likelihood of the loss function as:

$$J(\theta) = -\ln \mathcal{L}. \tag{2.12}$$

The main idea is to determine the most likely class that an input pattern belongs to and to model the posterior probabilities of a class given the input data. True posterior probability is a global minimum for both the cross-entropy and square error criteria. So neural network can be trained by minimizing either function.

2.3.2.1 Mean square error (MSE)

The error function is commonly given as the sum of squares of the differences between desired outputs y_i and neural network's response $\hat{y}_i$ given as:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\|y_i - \hat{y}_i\|^2, \tag{2.13}$$

where the sum of square error is divided by m for the normalization to find mean square error. When modeling a distribution, square error is found to be more bounded comparatively and the optimization is therefore more robust to outliers. But using average square error cost over all input batch with logistic units has some drawbacks. If the desired output is close to 0 or 1, there is almost no gradient for a logistic unit to fix up the error. The derivatives tend to plateau-out. Hence, square error criterion is suitable when problem is regression (real valued input) rather than binary. Selection of activation function should be matched to the error function to naturally produce the simple form of the derivative of the error with respect to the output. There is a natural pairing of the error function and the choice of the output activation function. In the regression problem MSE criteria is used paired with linear activation function. From Figure 2.4 if n is the number of input nodes, $l^{(1)}$ to $l^{(c+1)}$ are the corresponding layers and $\delta^{(c+1)}$ is the output layer gradient then the gradient of MSE cost $\nabla_{W^l} J(\theta)$ in matrix vectorial notation is followed by the given steps:

Until the stopping criteria meets, for $i = 1$ to n

1. Perform feed forward pass, computing the activations from $l^{(1)}$ to $l^{(c+1)}$.

2. Starting from output layer, gradients are computed given below using " $\bullet$ " element-wise product operator or Hadamard product.

$$\delta^{(c+1)} = \frac{1}{2}\frac{\partial\|y - \hat{y}\|^2}{\partial z^{(c+1)}} = -(y - \hat{y}) \bullet f'(z^{(c+1)}). \tag{2.14}$$

3. For $l = c, c-1, c-2, ..., 1$ and including $b^{(l)}$ in vector $W^{(l)}$ with all ones as its input vector computes:

$$\delta^{(l)} = ((W^{(l)})^T \delta^{(l+1)}) \bullet f'(z^{(l)}). \tag{2.15}$$

4. The desired partial derivatives are computed as:

$$\nabla_{W^l} J(\theta) = \delta^{(l+1)} (y^{(l)})^T. \tag{2.16}$$

5. Then the update rule becomes:

$$\Delta W^{(l)} = \beta \Delta W^{(l)} + \nabla_{W^{(l)}} J(\theta), \tag{2.17}$$

$$W^{(l)} := W^{(l)} - \frac{\epsilon}{m} \Delta W^{(l)}. \tag{2.18}$$

Given above if $f(z)$ is the logistic function then $f'(z_i^{(l)}) = y_i^{(l)}(1 - y_i^{(l)})$ and β denotes the momentum constant, details are given in Section 3.1.4. Usually, $\Delta W^{(l)}$ is initialized from zero or uniformly randomly distributed in a neighborhood of 0.

2.3.2.2 Cross Entropy (CE)

In practice cross entropy comparatively leads to faster training speed and better generalization performance [29]. It is specifically developed for binary targets. Unlike square-error that relies upon the absolute error, cross entropy error depends upon the relative errors of the network outputs. Hence it may perform well on both large and small target values as they tend to result in similar relative errors. For classification problem, the targets represent labels defining probabilities of class membership. If $y_j^{(i)}$ is the target output and $\hat{y}_j^{(i)}$ is the prediction of an output neuron for m number of inputs from the previous layer then for C separate binary classifications, output layer with C logistic sigmoid output units is an appropriate choice as given in Equation 2.19. The binomial Kullback Leibler (KL) divergence or cross-entropy error function becomes:

$$J(\theta) = -\sum_{i=1}^{m} \sum_{j=1}^{C} y_j^{(i)} \ln(\frac{\hat{y}_j^{(i)}}{y_j^{(i)}}), \tag{2.19}$$

$$J(\theta) = -\sum_{i=1}^{m}\sum_{j=1}^{C}\{y_j^{(i)} \ln \hat{y}_j^{(i)} + (1 - y_j^{(i)}) \ln(1 - \hat{y}_j^{(i)})\}. \tag{2.20}$$

This cost has a very steep derivative when the output is wrong. For a standard multiple-class classification problem, where each input is assigned to one of C mutually exclusive classes then C output units can use softmax activations. The error function then used is called multiple-class cross-entropy [24].

For a two class calssification problem, the cross entropy from Equation 2.20 becomes:

$$J(\theta) = -\sum_{i=1}^{m}\{y^{(i)} \ln \hat{y}^{(i)} + (1 - y^{(i)}) \ln(1 - \hat{y}^{(i)})\}. \tag{2.21}$$

For example, in case of $C = 2$ a network with sigmoid unit will use a single output. Alternatively, in case of softmax activation two output units will be used. For binary classifications we use logistic units with the corresponding cross-entropy criteria. For multi-class classification softmax outputs paired with its defined criteria is used. Update rule is same as given in Section 2.3.2.1 except because of the cross entropy, the cost function in step 2 is replaced by:

$$\delta^{(c+1)} = -(y - \hat{y}). \tag{2.22}$$

2.4 Deep Autoencoder

An autoencoder [5] consists of an input layer, one or more hidden layers and an output layer. The model has an encoder and a decoder. Its goal is to reconstruct the input object at the output layer through a hidden layer. Generally, if the dimension of the internal layer is less than that of the input layer, autoencoder performs dimensionality reduction task. On the contrary, if hidden layer size is greater then we enter the realm of feature detection. These properties make it competent enough to be the building blocks for deep architectures. For such networks each hidden layer is first trained individually in a setting as shown in Figure 2.5, followed by a fine-tuning step as shown in Figure 2.6. The hidden layer activation in the encoder from the visible layer to the hidden units in layer

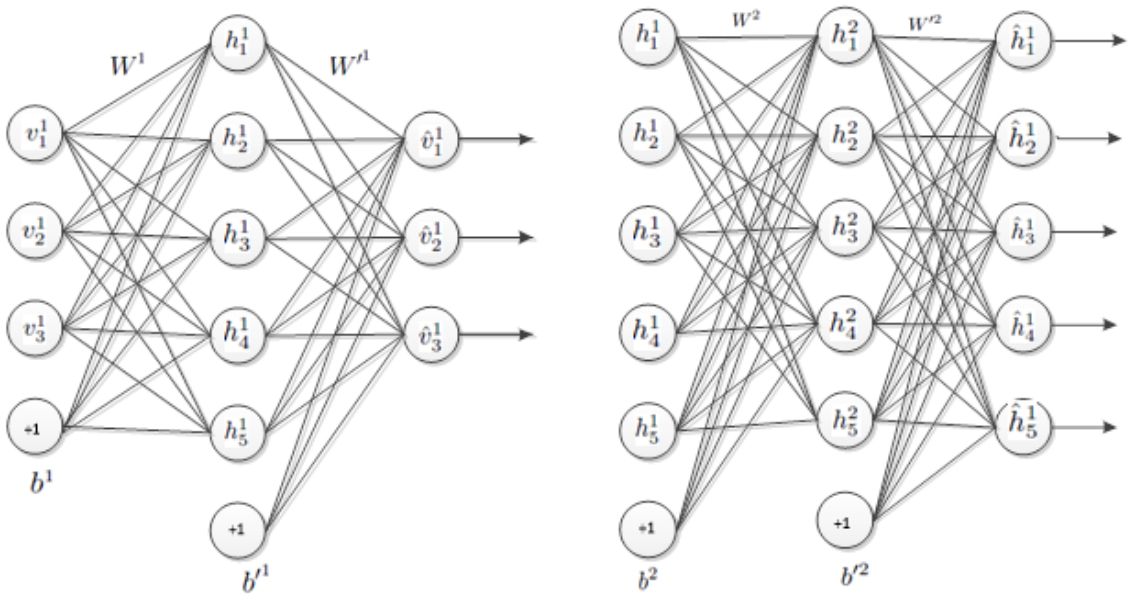

Figure 2.5: **Left:** Pretraining of an over-complete autoencoder for layer $l = 1$ of a multilayer network **Right:** Pretraining of second autoencoder for layer $l = 2$

i is given as:

$$h_\theta^{(i)} = f(W^{(i)} v^{(i)} + b^{(i)}). \tag{2.23}$$

Its parameter set is $\theta^{(i)} = \{W^{(i)}, b^{(i)}\}$ where $f(.)$ is a logistic nonlinearity, usually sigmoid function $f(x) = \frac{1}{1+e^{-x}}$. If input object is not between 0 and 1, linear function $f(x) = x$ can be used as an activation function for the reconstruction layer. Using this code/feature vector, the hidden representation is mapped back to s_l dimensional vector $\hat{v}_{\theta'}^{(i)}$:

$$\hat{v}_{\theta'}^{(i)} = f(W'^{(i)} v^{(i)} + b'^{(i)}), \tag{2.24}$$

with appropriate parameters $\theta' = \{W', b'\}$. In general $\hat{v}^{(i)}$ should not be interpreted as an exact reconstruction of $v^{(i)}$ but to learn parameters that can distinguish the features contained in input samples. This yields a reconstruction error to be optimized for real valued inputs $v \in \mathbb{R}^{s_l}$ given as:

$$\mathcal{L}(v, \hat{v}) = \|v - \hat{v}\|^2. \tag{2.25}$$

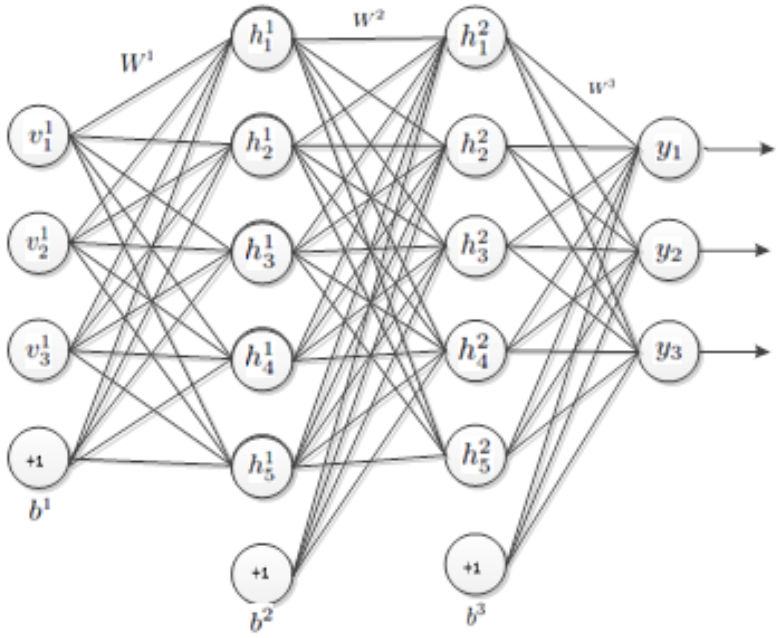

Figure 2.6: The feed forward network is fine-tuned with $\theta^1 = \{W^1, b^1\}$ obtained from first autoencoder and $\theta^2 = \{W^2, b^2\}$ obtained from second autoencoder. Output layer is replaced by a softmax classifier

This is the square error objective to be optimized. It is achieved by batch stochastic gradient descent (SGD) of an autoencoder and viewing it as the minimization of batch loss measure.

$$W^* = \min_{W,W'} \mathcal{L}(v, \hat{v}) \tag{2.26}$$

where W^* is the set of parameters $\{\theta, \theta'\} = \{W, b, , W', b'\}$ that obtains minimum reconstruction error or loss function $\mathcal{L}(v, \hat{v})$. The mean square error (MSE) becomes:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \|\hat{v}^{(i)} - v^{(i)}\|^2. \tag{2.27}$$

Reconstruction error is computed and then is back-propagated to update the encoder and decoder weights for as long as the error is reduced. Similarly, the next layers can be trained by assuming them each as separate autoencoders. Since the input layer is equal to the output, its goal is to learn an approximation of the identity function. However instead, putting constraints on the network

can reveal meaningful complex representation of the data. A network with small encoder weights W and large decoder weights W' may encode an uninteresting identity function [30]. One possible solution is to use tied weights, i.e., $W = W'^T$. It gives the benefit of reduced parameters for updating [31]. If linear activation function is used in the hidden layer, such autoencoder functions as PCA (Principle Component Analysis) in its under-complete case. However, a nonlinear autoencoder is capable of extracting more complex structures from the data. The code vector obtained with square reconstruction error and linear decoder are in the span of the principle eigenvector of the input covariance matrix. For logistic decoder, where we want to reconstruct value range [0,1] cross-entropy error function is more appropriate.

The decoder part of each autoencoder doesn't contribute in stacking of a deep architecture shown in Figure 2.6, as that is only used for training of the encoder layer. The first hidden code vector $h_\theta^{(1)}$ learned is used as the input to the following autoencoder and is trained as a separate one. In this way s_l number of feature/code vectors $h_\theta^{(s_l)}$ can be stacked on the top of each other. This way of training is called "greedy layer-wise unsupervised pre-training" as the stacked architecture has no access to the label/output vector [30].

2.5 Autoencoder and Sparsity

In deep learning algorithms one of the claimed objective is to separately represent the variations of the data [5]. A dense representation is highly mixed and dependent because any change in the input changes most of the features in the representation vector. Sparse representation enables the network to roughly conserve small changes of the input by its non-zero hidden units and thus enables more robust structures. Amount of information may vary from one input sample to another. Sparse representation enables the network to use variable size active hidden neurons for the prediction of the input. However, forcing too much sparsity may hurt the predictive performance as it reduces the model capacity [20].

A simple autoencoder encodes an input $v^{(i)}$ and decodes it back to reconstruct $\hat{v^{(i)}}$. The identity function, copying inputs to outputs, is not a useful way of learning. Additional constraints should be added on either the network or the error function to make it extract useful features even if it is over-complete. If a sparsity constraint is imposed on the hidden units, then the autoencoder can learn interesting features in the data even if the hidden layer size is large. The effect of this constraint keeps the neurons to be inactive most of the time [17]. If $h_j^{(2)}$ represents the j^{th} hidden unit's activation which is a function of input vector v in an autoencoder, the average activation of hidden unit i (averaged over a batch of input data set) is given as:

$$\hat{\rho}_i = \frac{1}{m} \sum_{j=1}^{s_2} [h_j^{(2)}(v^{(i)})]. \tag{2.28}$$

The hidden representations are made sparse by giving the units a very small expected activations. The value ρ is a sparsity parameter that denotes the desired frequency of activation of the hidden units. Let $\hat{\rho}_i$ indicate the average thresholded activations or the activation probability of hidden layer j over the whole training data set. The constraint is enforced to approximate:

$$\hat{\rho}_i = \rho. \tag{2.29}$$

Typically a small value of ρ close to 0 works well. To satisfy Equation 2.29, the hidden unit's activations must be inactive most of the time during the whole training set, that's why it is called as sparse autoencoder. If $\hat{\rho}_i$ is deviating too much from ρ, it is penalized by addition of KL penalty term which measures how different two distributions are which is given as [17]:

$$KL(\rho \| \hat{\rho}_i) = \rho \log \frac{\rho}{\hat{\rho}_i} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_i}. \tag{2.30}$$

This causes very rear number of neurons to be active in the hidden layer by minimizing the difference between ρ and $\hat{\rho}_i$.

When sparsity is enforced in the hidden layer by addition of KL penalty term in the objective function given in Section 2.27, the overall cost takes the form as

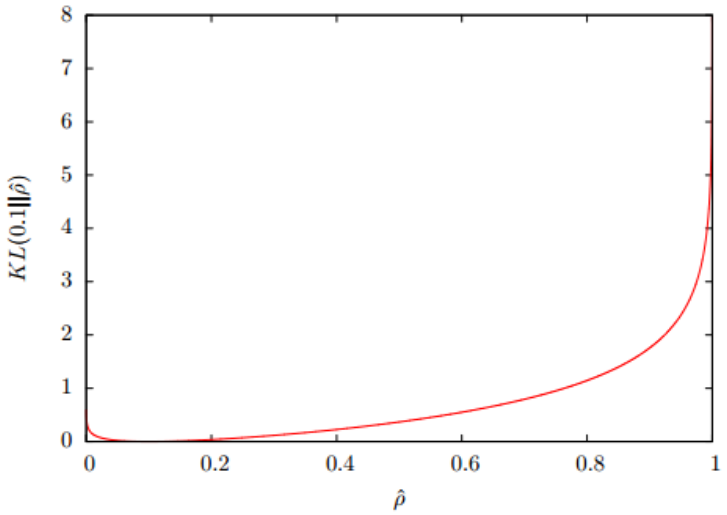

Figure 2.7: KL Sparse penalty function for $\rho = 0.1$. When $\hat{\rho} = \rho$ $KL(\rho\|\hat{\rho}) = 0$ and it increases as $\hat{\rho}$ diverges from ρ.

below:

$$J_{sparse}(\theta) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|\hat{v}^{(i)} - v^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\|W\|^2 + \alpha\sum_{j=1}^{s_2} KL(\rho\|\hat{\rho}_i), \qquad (2.31)$$

where $\hat{v}$ is the output obtained by passing input v through the logistic units in the hidden and output layers [5] and s_2 is the hidden layer size. The ℓ_2 weight decay has been added in the cost which is discussed in Section 2.6.2. After successful learning, autoencoder should decompose the inputs into a combination of useful features in the form of hidden layer activations. The number of features learned is equal to the hidden layer size s_2. The hyperparameters λ and α determine the relative importance of the weight decay regularization term and the sparseness term in the cost function. The term $\hat{\rho}_i$ depends on $\theta = \{W, b\}$ because it is the average activation of hidden unit i and the activations ultimately depend on θ. To incorporate Equation 2.30 into error cost for sparse autoencoder, backpropagation is modified to compute the hidden layer gradient given in Equation 2.32 while the output layer gradient is computed in the conventional manner.

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)}\delta_j^{(3)}\right) + \alpha\left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}\right)\right) f'(z^{(2)}). \qquad (2.32)$$

2.6 Regularization Techniques on Deep Neural Networks

Though the training inputs have information about the regularities in the mapping from input to output in a neural network but it also contains sampling error. When the model is trained, it cannot distinguish between real regularities and the ones caused by sampling error. Hence, it fits both types. We need a neural network model, that may have enough capacity to fit the right regularities from the image. This can be done by using different regularization methods.

2.6.1 Pretraining

Unsupervised pretraining performs as a network pre-conditioner, moving the parameter towards an appropriate range for further supervised training. It has been described as a way of initializing the model to a point in parameter space that makes the optimization process more effective, in the sense of reaching a lower minimum of the empirical cost function [9]. It is also defined as a form of regularization that minimizes variance and introduces bias towards configurations of the parameter space [23].

2.6.2 ℓ_p-norm Based Regularization

Let $W \in \mathcal{R}^q$ be a vector where $q = |w|$ is the size of W, then ℓ_p-norm of W is defined as:

$$\|W\|_p = \Big(\sum_{j=0}^{|w|} |W_j|^p \Big)^{\frac{1}{p}}, \tag{2.33}$$

where commonly used values for p are 1 and 2 hence called ℓ_1/ℓ_2 respectively. ℓ_2 is a type of regularization that involves adding an extra term to the cost function that penalizes the squared weights [32]. It keeps the weights small unless weights obtain big error derivatives. It prevents the network from using useless

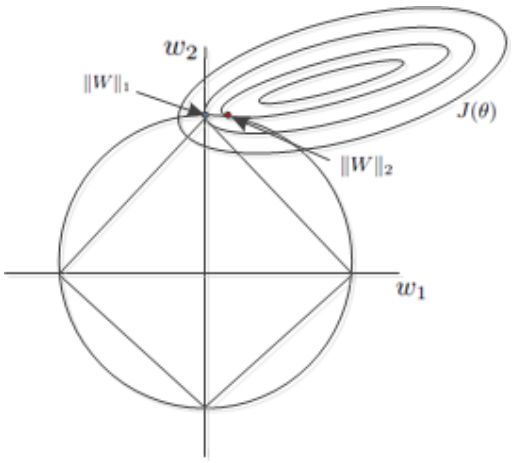

Figure 2.8: When ℓ_2-ball meets quadratic cost, it has no corner and it is very unlikely that the meet point is on any of the axes.While when ℓ_1 ball meets quadratic cost, it is very likely that the meet point is at one of the corners.

weights and gives smooth regularized loss function $J(\theta)$. As a result it improves generalization by not allowing the network to fit the sampling error and enabling each feature to contribute a bit for output predication. Large weights can hurt generalization in two ways. Excessive large weights leading to output layer units can cause excessive variance of the output far beyond the range of the data. One disadvantage of ℓ_2 penalty is that it tends to shrink the weights but sets none of them exactly to zero. This problem is called as Ridge Regression [33]. The weight decay parameter λ determines how you trade off the original cost $J(\theta)$ with the large weights penalization and controls the regularization effect in both ℓ_2 and ℓ_1 regularized cost. When ℓ_2 penalty is used for autoencoder, the mathematical expression obtained for its cost function from Equation 2.27 becomes:

$$J_{regularized(L_2)}(\theta) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|\hat{y}^{(i)} - y^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\|W\|_2^2. \tag{2.34}$$

The cost function obtained from Equation 2.34 is smooth and optimum is the stationary or 0-derivative point. This point can become very small when λ is increased but can never be zero.

On the other hand, ℓ_1-norm can lead weights towards zero [34]. All gradients take value either 1 or -1 and point towards the axis of the smallest entry. Changing

the parameters in this direction shrink all parameters by the same amount and finally leads to zero sparse parameter vectors.

$$J_{regularized(L_1)}(\theta) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|\hat{y}^{(i)} - y^{(i)}\|^2\right)\right] + \lambda\|W\|_1^1. \tag{2.35}$$

The cost function obtained from Equation 2.35 is non-smooth. The optimum of the cost is either the point with 0-derivative or one of the irregularities (corners, kinks, etc). So, the optimum point may be 0 even if 0 isn't the stationary point of the cost function J.

2.6.3 Max-norm Constraint

An other way of regularizing the network is to put a constraint on the maximum squared length of the incoming weight vector of each hidden unit [2, 21]. If any update doesn't satisfy this constraint, the vector of incoming weights is scaled down to the allowed length. This technique neither lets hidden units trapped near zero nor allows weights to explode. If w represents the vector of incoming weights to any hidden unit and if c is the upper bound on this magnitude then the network is to be optimized under the constraint $\|w\|_2 \leq c$. It projects w onto a surface of a ball of radius c which implies that norm of any weight can take maximum value of c.

2.6.4 Denoising Autoencoder

This is a stochastic variant of the simple auto-encoder which even with a high capacity model cannot learn the identity mapping. Ordinary autoencoders are found slightly worse than RBMs in a comparative study [35]. But there are many ways to obtain variants of autoencoders to compete the performance of RBMs. One of which is the denoising auto-encoder. It is trained to denoise a noisy version of its input and performs significantly better than simple autoencoders. A simple autoencoder, where $\hat{v}$ is of the same dimensionality as v can achieve perfect reconstruction simply by learning an identity function where further constraints

need to be applied to obtain useful features. Using over-complete but sparse representation has received much attention recently [1].

Denoising autoencoder is trained to get a clean output from the corrupted version of the input. It is done by first corrupting the original input v into $\tilde{v}$ by introducing zeros by means of a stochastic mapping $\tilde{v} \sim q(\tilde{v}|v, D)$ with probability D, which is sampled as:

$$q(\tilde{v}_i|v_i, D) = \begin{cases} 0 & \textit{with probability } D, \\ v_i & \textit{otherwise.} \end{cases}$$

where for each input v, a fixed number of components are chosen randomly with probability D and their value is forced to 0, while the others are left untouched. The corrupted input $\tilde{v}$ is then mapped to a hidden code vector $h_\theta^{(i)} = \sigma(W^{(i)}\tilde{v}^{(i)} + b^{(i)})$ from which we get $\hat{v}^{(i)}$ as close as possible to the uncorrupted input $v^{(i)}$. As compared to a simple autoencoder, it now obtains a deterministic mapping to a corrupted input. Hence, it forces the learning of more important features than an identity function. Usually masking noise is used for the corruption, where a fraction m of the elements v is forced to 0. Moreover, stacking denosing autoencoder to initialize a deep network is much the same way as stacking RBMs or AEs [1].

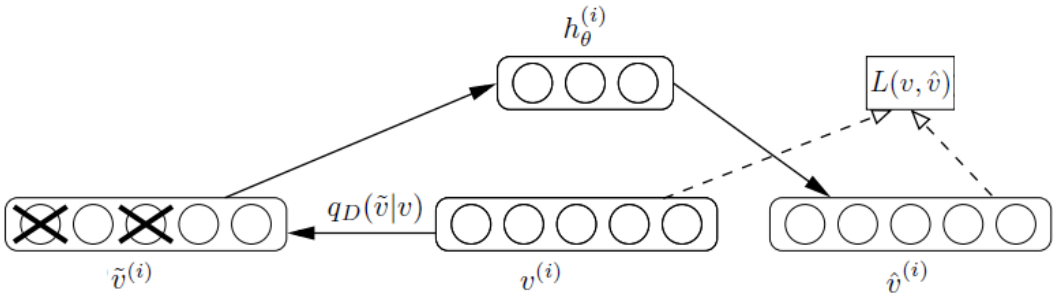

Figure 2.9: Input v is corrupted to $\tilde{v}$ with masking noise. Denoising Autoencoder (DAE) maps it to hidden layer h and attempts to reconstruct a clean input v [1].

2.6.5 Drop-outs

Another way of avoiding overfitting in a feedforward network is dropout proposed by [2, 21]. For each hidden unit, dropout prevents co-adaptation by making the

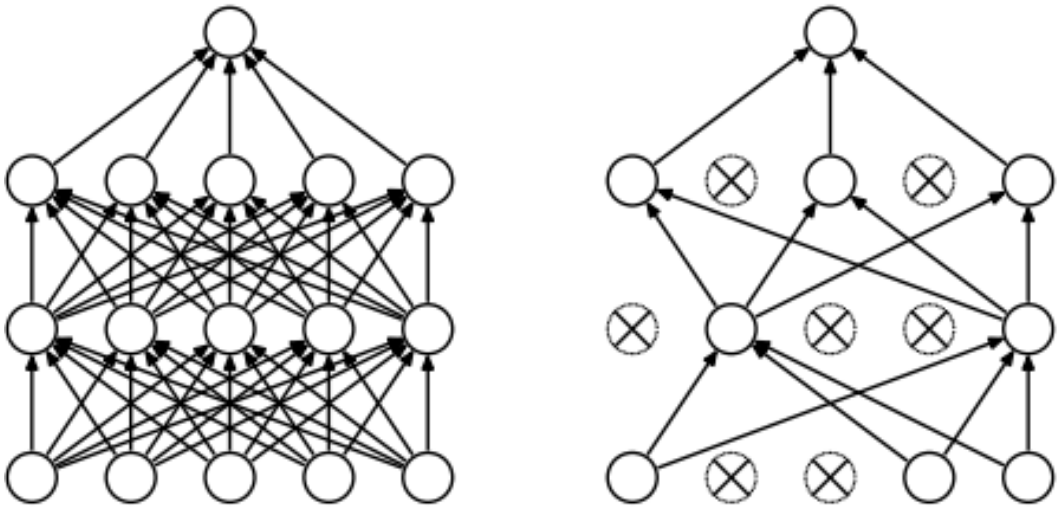

Figure 2.10: **Left:** A standard neural network with 2 hidden layers. **Right:** One possible thinned network after dropout regularization. The crossed sign ($\times$) on nodes represent the dropped out units with all its incoming and outgoing connections in the input and hidden layers [2].

unit learn independently to correct its mistake without relying on other units. This technique consists of training different thinned neural nets on the data and provides a way of approximately combining exponentially many different architectures efficiently. It is done by dropping a unit out along with all its incoming and outgoing connections. The choice of which unit to drop is random with a probability of p, which is selected using a validation set. By dropping out p units from n hidden units, for each pass we can sample 2^n possible combinations. So, we actually train a collection of 2^n thinned networks. Each unit is retained with a probability of $(1-p)$ during training. At test time, the outgoing weight of each hidden unit is downscaled by multiplying it with its retained probability value $(1-p)$. In this way, the predictions of 2^n networks are estimated at test time. This method improves generalization and prevents over-fitting issue. Dropouts

can be applied to train graphical models such as RBMs as well.

Let $l \in \{1, ..., L\}$ be the layers of the hidden layers with L layers. let $z^{(l)}$ denote the sum of output units and $a^{(l)}$ denote the corresponding activation output units of layer l. With dropout, the feed-forward operation from Equation 2.6 is given as:

$$\begin{aligned} a^{(l)} &= x^{(l)}, && (2.36) \\ r_j^{(l)} &\sim Bernoulli(1-p), && (2.37) \\ \tilde{a}^{(l)} &= \mathbf{r}^{(l)} \bullet a^{(l)}, && (2.38) \\ z^{(l+1)} &= W^{(l+1)}\tilde{x}^{(l)} + b^{(l+1)}, && (2.39) \\ a^{(l+1)} &= f(z^{(l+1)}), && (2.40) \end{aligned}$$

where $r_j^{(l)}$ is jth neuron's Bernoulli random number in layer l and $\mathbf{r}^{(l)}$ is a vector of Bernoulli random variables each of which is with a probability p of being 1 and $\bullet$ represents the element wise vector multiplication. For each layer $r^{(l)}$ is sampled and is multiplied element wise with its corresponding output layer $a^{(l)}$ to generate the thinned outputs $\tilde{a}^{(l)}$ which are then used as inputs for the next layer. Accordingly, the gradients are then backpropagated though the same thinned paths.

At test time, it is computationally complex to average the predictions of all networks. However, it can be made feasible by a simple approximate averaging method. The weights of all possible thinned networks are approximately averaged as follows:

$$W_i^{(l)} = (1-p)W_i^{(l)}. \quad (2.41)$$

Chapter 3

Methodology

3.1 Selection of the Hyper-parameters

Choosing hyperparameter values is equivalent to the task of model selection. There are no magic values for the hyperparameters as the optimal values vary with the type of data used. A set of such parameters is related to deep neural networks as well which need to be tuned to get a precise model.

3.1.1 Weight Initialization

Initialization gives a small bias to the weights because when weights are set to zero, the outputs will also be zero which cause the gradients to be zero. The weights should be small enough, to enable the network remain activated in the linear part of its nonlinear function because the gradients are largest and fastest in this region [36]. Biases can generally be initialized from zero but care should be taken to initialize the weights to break the symmetry. One way that has been proposed depends on the fan-in (Input of the nodes) and fan-out (output of the nodes)[6]. A uniform $(-r, r)$ is sampled with the given possible ways as:

$$r = \sqrt{\frac{6}{fan-in + fan-out}}, \tag{3.1}$$

$$r = 4\sqrt{\frac{6}{fan-in + fan-out}}, \tag{3.2}$$

where Equations 3.1 is used for sigmoid and Equation 3.2 is used for hyperbolic tangent sigmoid activation functions. A zero-mean Gaussian $\mathcal{N}(0, \sigma^2)$ with a small value of $\sigma^2 =$ as 0.1 or 0.01 also works well [37].

3.1.2 Mini-Batch size

Instead of single input example, we use mini-batch updates on the average of the gradients inside each block B. With large B we can parallelize multiply-add operations per second. But the number of updates per computation decreases, which slows down the convergence because less updates can be done in the same computing time [27]. Smaller values of B may help for exploring more in the parameter space. For the efficiency of the estimator, it is very important to sample the mini batches independently. Faster convergence has been observed if the inputs for a mini batch are chosen randomly for each epoch. During backpropagation, the gradients are averaged over the training cases in each mini-batch.

3.1.3 Learning Rate ϵ

A default value of 0.01 typically works well for standard multilayer networks. With respect to time t this parameter can be exponentially decayed as $\epsilon^t = \epsilon_0 f^t$. It starts at ϵ_0 (applied to the average gradient in each minibatch) and is multiplied by f^t after each epoch where f is the multiplying factor. As the learning rate decays, the algorithm keeps on taking smaller steps and finds the right step size at which it makes learning progress [21].

3.1.4 Momentum β

Momentum is used to speed up learning and controls how fast the old examples become down weighted in the moving average of the past gradients. It adds a fraction β of the previous weight update to the current one. Usually we use constant momentum of value [0.3,0.5]. The weight update rule discussed in Section 2.11 is modified by the addition of momentum as:

$$u^t = \beta^t u^{t-1} + \epsilon_t \frac{1}{B} \sum_{t=s_t}^{s_t+B} \frac{\partial J(\theta_t, x_t)}{\partial \theta}, \tag{3.3}$$

$$\theta^t = \theta^{t-1} - u^t. \tag{3.4}$$

For dropout model, the weight update of the error back propagation algorithm takes β in the form given as:

$$\beta^t = \frac{t}{T}\beta_f + (1 - \frac{t}{T})\beta_i \quad t < T, \tag{3.5}$$

where t is the current epoch, T is the total number of epochs, β_i is the initial momentum and β_f is the final momentum. The Equation 3.5 gives the increasing values of momentum till the number of epochs reaches at its maximum. Dropouts introduce noise in the gradients, hence a lot of gradients cancel each other. To overcome this issue dropout networks are made stable using a high final momentum and decreasing learning rate ϵ that distributes gradient information over large number of updates [21].

3.1.5 Number of Hidden Units and Layers

The size of each layer in a multi-layer model controls the capacity. If some regularization technique is used then it is important to choose the number of neurons s_l large enough in each hidden layer not to hurt generalization. However, it then requires more computation. Usually, using the same size for all the layers generalize well [9, 27] but selection of s_l size may be data dependent. An overcomplete hidden layer works better than an under-complete one to significantly capture the relevant information along with irrelevant during unsupervised pre-training.

In conventional neural network, error goes up after many hidden layers normally. However, pretraining allows error to decrease as we move from 1 to 4 hidden layers but after 5 layers of depth even this technique can't help [23]. Pretraining is especially useful in optimizing the parameters of the lower level layers.

3.2 p-norm Sparse Autoencoder - Unsupervised Training

The encoder f maps input $v \in \mathbb{R}^{s_1}$ to a higher dimensional latent space $h \in \mathbb{R}^{s_2}$. The decoder does the opposite. In this setting, the proposed sparsity regularizer restricts the range h of the learned encoder f. Imposing Sparsity constraint on the hidden units can make autoencoder discover interesting features from the input data. In sparse autoencoder, we would like to constrain the neurons to be inactive most of the time [17]. We need to estimate the expected activation value for each hidden unit. The average hidden activation is regularized to a predefined sparsity hyperparameter ρ. Let $\omega(\theta)$ denote the proposed sparsity regularizer. The proposed method of imposing penalty to satisfy this constraint is defined as:

$$\omega(\theta) = \alpha \left\| \frac{1}{m} \sum_{j=1}^{s_2} f(v_j^{(i)}) - \rho \right\|_p^p, \tag{3.6}$$

where the mean hidden activations of the s_2 layer is given by:

$$\hat{\rho_j} = \frac{1}{m} \sum_{j=1}^{s_2} f(v_j^{(i)}). \tag{3.7}$$

The expression shows that $\hat{\rho}$ ultimately depends on θ. If ρ is set near to 0, a sparse latent representation is produced by the encoder f. If $\hat{\rho_j}$ is either too smaller or too larger than ρ, it can be suspected that the sample v is either not of the same type as training example or might be corrupted with high level of noise. The p-norm is a real number where $p \geq 1$. For $0 < p < 1$ the resulting function does not define a norm due to violation of the triangle inequality. However, for $p \geq 1$, the p-norm of the difference of mean activations $\hat{\rho_j}$ and ρ gives a

differentiable function. So, p-norm of the difference pushes $\hat{\rho}_j$ towards ρ and all hidden units get activated equal number of times. Our new optimization objective combined with ℓ_2 regularized regression from Equation 2.31 becomes:

$$J_{sparse}(\theta) = \left[\frac{1}{m}\sum_{j=1}^{m}\left(\frac{1}{2}\|\hat{v}_j^{(i)} - v_j^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\|W\|^2 + \omega(\theta), \tag{3.8}$$

$$J_{sparse}(\theta) = \left[\frac{1}{m}\sum_{j=1}^{m}\left(\frac{1}{2}\|\hat{v}_j^{(i)} - v_j^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\|W\|^2 + \alpha\,\|\hat{\rho}_j - \rho\|_p^p. \tag{3.9}$$

The objective function for the decoder part of an autoencoder is optimized by Equation 2.27 in a conventional SGD way without the ω term. As sparsity is imposed on the hidden layer only, so Equation 3.8 is optimized only for the corresponding layer given as:

$$\frac{\partial J_{sparse}(\theta)}{\partial\theta} = \frac{\partial J(\theta)}{\partial\theta} + \frac{\lambda}{2}\frac{\partial\|W\|^2}{\partial\theta} + \alpha\frac{\partial\|\hat{\rho}_j - \rho\|_p^p}{\partial\theta}. \tag{3.10}$$

The activations of all three layers are denoted by:

$$\begin{aligned} a^{(1)} &= v, \\ a^{(2)} &= f(W_1 a^1 + b_1), \\ a^{(3)} &= f(W_2 a^2 + b_2), . \end{aligned}$$

If $\delta^{(3)}$ and $\delta^{(2)}$ represent the gradients for the output layer and the hidden layer of an autoencoder respectively then the parameters $\theta = \{W, b\}$ are computed as:

$$\delta^{(3)} = -(a^{(1)} - a^{(3)}) \bullet a'^{(3)}, \tag{3.11}$$

$$\delta^{(2)} = \left(W_1^T\delta^{(3)} + \alpha\ p\ |\hat{\rho}_j - \rho|^{p-1}\ sgn(\hat{\rho}_j - \rho)\right) \bullet a^{(2)}(1 - a^{(2)}), \tag{3.12}$$

$$\Delta W_2 = \frac{1}{m}\delta^{(3)}(a^{(2)})^T + \lambda W_2, \tag{3.13}$$

$$\Delta W_1 = \frac{1}{m}\delta^{(2)}(a^{(1)})^T + \lambda W_1, \tag{3.14}$$

$$\Delta b_2 = \frac{1}{m}\sum\delta^{(3)}. \tag{3.15}$$

$$\tag{3.16}$$

The parameters are updated as follows:

$$W_2 := W_2 - \epsilon \Delta W_2, \tag{3.17}$$

$$W_1 := W_1 - \epsilon \Delta W_1, \tag{3.18}$$

$$b_2 := b_2 - \epsilon \Delta b_2, \tag{3.19}$$

$$b_1 := b_1 - \epsilon \gamma (\hat{\rho}_j - \rho). \tag{3.20}$$

To satisfy the expectation constrain, if $\hat{\rho}_j > \rho$, then we would like hidden unit j to become less active by decreasing b_1^i and if $\hat{\rho}_j < \rho$, then we will try to activate unit j to become more active which is done by increasing b_1^i.

If training set is small enough to fit easily in computer memory, we can compute forward passes on all our input sets and keep the resulting activations in memory and compute the $\hat{\rho}_j$s. But if data is too large to fit in memory, for each mini batch we can keep running estimate $\hat{\rho}_j$ of $\frac{1}{m}\sum_{j=1}^{s_2} a_j^{(2)}$ by updating:

$$\hat{\rho}_j = 0.99\hat{\rho}_j + 0.01 a_j^{(2)}. \tag{3.21}$$

This update is applied on each iteration of gradient descent to make $\hat{\rho}$ close to ρ which gives exponential decayed weighted average for the last 100 observed values of $a_j^{(2)}$ in a mini batch. For each jth neuron , $\hat{\rho}_j$ is initialized from zero. The parameter γ is an additional learning constant for bias b_1 update of the hidden layer units and λ controls the weight decay. Other parameters like α controls sparseness and ϵ controls how fast the weights are upated. The value of ρ takes the hidden layer towards more sparsity as becomes smaller.

3.3 k-Lowest Dropout Regularization

This idea is motivated from the biological role of a neuron that activates if input impulse is high enough than a certain threshold. In multilayer networks, as each unit of a previous layer is connected to every other units of the next layer. This makes each unit dependent on every other units and hence produces complex co-adaptation and over-fitting. If the idea of neuron's activation after crossing

a certain threshold level is implemented on such type of densely interconnected networks, the units can be allowed to learn more independently during training, as not all the units remain connected to all other units and the neurons learn relying on their own. This way the issue of overfitting can be avoided and co-adaptation between the units of the network layers can be reduced.

The idea of threshold on the hidden layer activations can be applied by a variable k. During training, each time introducing new input sample to the network causes different hidden neurons give different response based on the features detected in the current input sample. On each input sample, selection of k-lowest activations of each hidden layer is equivalent to setting a threshold in a biologial neuron. This technique reduces co-adaptation between hidden layers and allows only important information to be learned. To prevent overfitting and complex co-adaptation between hidden layers the idea of dropout was previously proposed [2]. Our proposed method "k-lowest dropout" is a variant of this idea. As we think of a neuron being active if its output obtains value close to more positive value and inactive if value becomes close to 0 or more negative. For fine tuning when each layer is already pre-trained , each hidden layer has learned some complex features of the input samples. When any input sample is presented, hidden units specific for that input sample's structure give high activations for its prediction. Along with this set of hidden units, other units may also contribute a little in prediction which causes sampling noise and results in overfitting. The proposed idea is to dropout those unimportant hidden units to reduce the sampling noise for the prediction. Those unimportant number of hidden units is decided by the value k. If k number of units are dropped out then $(1 - k)$ highest number of activated units remain responsible to pass the signal to next layer units. The value of k is determined by validation data. Moreover, this technique has been effective enough for the pre-training of autoencoders as well. Simulations show that parameters learned from such autoencoders enable the initialization of a deep neural network in a good region of parameter space, where after some fine-tuning step the network is able to give good performance for the classification task.

Mathematically, the derived expressions for the proposed method is discussed

below. We know that a network's layer output is computed as:

$$a^{(l+1)} = f(W^{(l+1)}a^{(l)} + b^{(l+1)}). \tag{3.22}$$

After the above feed forward pass, the outputs are computed for each layer. If k is the fraction of the number of hidden units in a layer that is to be dropped out then the set of those dropped out activations at layer l are denoted as $a^{(l)}_{\Gamma}$ and the set of retained hidden units's activations are represented by $a^{(l)}_{\Gamma^c}$.

$$r^l_j = \begin{cases} 1 & \mathrm{if} r^l_j \in a^{(l)}_{\Gamma^c} \\ 0 & \mathrm{if} r^l_j \in a^{(l)}_{\Gamma} \end{cases}$$

where r_j is the masked value for the jth neuron. Its value as 1 shows that it is one of the features selected in $(1-k)$ highest activations, while it is dropped out if the value it obtains is 0. The masked vector $\mathbf{r}^{(l)}_{\Gamma}$ represents the whole hidden vactor after k-lowest dropout effect. The activations $a^{(l)}$ of the hidden layer l from Equation 3.22 are computed as:

$$\tilde{a}^{(l)}_{\Gamma} = \mathbf{r}^{(l)}_{\Gamma} \bullet a^{(l)}, \tag{3.23}$$

$$z^{(l+1)} = W^{(l+1)}\tilde{a}^{(l)}_{\Gamma} + b^{(l+1)}, \tag{3.24}$$

$$a^{(l+1)} = f(z^{(l+1)}). \tag{3.25}$$

The notation (Γ) stands for the set of k-lowest dropouts, each unit of which is set to 0 while the set of (Γ^c) activations $a_{(\Gamma)^c}$ or in other words $(1-k)$ number of each hidden layers are retained. On each forward pass depending on the structure of input sample, activations of hidden layer l are sorted and $a^l_{(\Gamma)^c}$ are selected while dropping $a^l_{(\Gamma)}$. When the error is back-propagated, $a^l_{(\Gamma)}$ set contributes zero gradient for the corresponding input.

However, this technique may add an extra computational complexity in sorting the activations of each hidden layer. Let n be the number of hidden units then

by selection sorting method selection of the lowest element require scanning all n units (this takes $n-1$ comparisons) and then swapping it into the first position. Finding the second lowest requires scanning the remaining $n-1$ units and similarly it goes on as $(n-1)+(n-2)+...+2+1 = \frac{n(n-1)}{2} \in O(n^2)$. The worst case of which may reach $O(n^2)$. In dropping out the units randomly, we dont come across this step and computation time is saved. However, random dropout needs more number of epochs to converge to an optimal solution but the convergence point in k-lowest dropouts is obtained within few epochs.

3.4 Visualizing Features

To obtain a better understanding of the features of autoencoder/feed forward network [17], we can visualize the distribution of orientation, positions and scales of the learned filters by finding what features would cause the neurons to be maximally activated. Such representations can be qualitatively analyzed when hidden unit i activation is computed as:

$$a_i^{(l+1)} = f\Big(\sum_{j=1}^{s_2} W_{ij}^{(l)} v_j + b_i^{(l)}\Big). \tag{3.26}$$

To visualize the edges learned by a neuron, we find an input v that maximizes the function a. Note that $f(.)$ puts a bound on a with values in range $[0, 1]$. So, we constrain the input a by its norm $\|v\| \leq 1$. The input v that maximizes the activation of hidden unit i is given by setting v_j to:

$$v_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{N}(W_{ij}^{(1)})^2}} \quad \forall j = 1, ..., N, \tag{3.27}$$

where N is the number of hidden neurons.

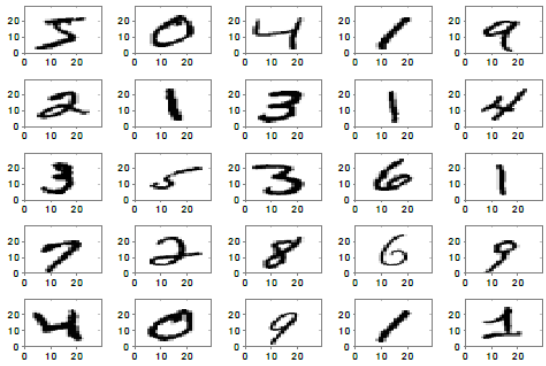

Figure 3.1: Samples from MNIST data set. Axes number shows the pixel coordinate.

3.5 Problem Formulation

MNIST hand written digit data set is a very popular benchmark for machine learning methods [38]. It has 60,000 training set and 10,000 test set. Each input image consists of 28×28 pixels giving an input dimensionality of 784. The number of classes is 10 (digits from 0 to 9) and the inputs are used by scaling it between 0 and 1 as a pre-processing step. Our focus is not to emphasize the model's usefulness on a wide range of task, but to demonstrate its properties empirically and compare it with other models. Such experimental work requires several weeks of CPU computation time, which has bound us to use more data sets. Our whole model took few hours for pre-training and fine-tuning steps in Matlab on a 2.66 GHz Xeon processor. Our model confirms the good performance of deep networks and helps to suggest moving on to more complicated problems. A sample of images in each class of the dataset is given in Figure 3.1.

C	training	test
0	5923	980
1	6742	1135
2	5958	1032
3	6131	1010
4	5842	982
5	5421	892
6	5918	958
7	6265	1028
8	5851	974
9	5949	1009

Table 3.1: The number of images belonging to class C in MNIST dataset

Chapter 4

SIMULATIONS AND EVALUATION

In this chapter the evolution of the work done is discussed and steps that lead to parameterize the final model are described. We denote the dataset as $\mathbb{D}$ and indicate train, validation and test set as: $\mathbb{D}_{train}$, $\mathbb{D}_{valid}$ and $\mathbb{D}_{test}$. The validation set $\mathbb{D}_{valid}$ (discussed in Section 4.1) is used as part of training and is not the same as the test set $\mathbb{D}_{test}$. The test set is used to evaluate the performance of the learning algorithm. It is cheating to use the test set $\mathbb{D}_{test}$ as part of learning. It is used to evaluate the final generalization error.

4.1 Data Preprocessing

We have performed data normalization as a preprocessing step. Data normalization is performed by simple rescaling in each dimension so that the final data lies in the range [0,1]. In MNIST data set, pixel values are in the range [0,255]. So, it is normalized by dividing the values by 255.

The validation set $\mathbb{D}_{valid}$ is used to tune hyper-parameters for the model which

is the part of the learning procedure. While training the network, our objective is not only to learn the training set $\mathbb{D}_{train}$ but also to improve prediction for future test examples $\mathbb{D}_{test}$. To achieve this, the network should not come across over-fitting. Selection of optimal values of hyperparameters can help in this regard. The ideal procedure for the selection is grid search, however this method is intractable when we have a large number of hyper-parameters. One possible way is to find the values through K-fold cross validation. The training data is divided into K folds. On the first training run $\frac{K-1}{K}$ of the training data is used for training as $\mathbb{D}_{train}$ and network's accuracy is determined by $\frac{1}{K}$ of the remaining data called validation set $\mathbb{D}_{valid}$. It is repeated unless $\frac{1}{K}$ subset is used exactly once as $\mathbb{D}_{valid}$ then K accuracy results are averaged. Each possible values of parameters are K-fold cross validated which gives an estimate of optimal values of the network. Due to time constraint we validated the values for K=3.

Learning is fastest when network use the most unexpected sample. Therefore it is more convenient to choose a sample that is most unfamiliar to the system. Passing a batch of inputs randomly from shuffled samples can perform this task. In all experiments during training of both the phases, we used a mini-batch size of 100.

4.2 Multilayer Perceptron Model

In this experiment, neural network with different layers have been trained on MNIST dataset using simple back-propagation algorithm without any regularization and pre-training. As stated previously, the SGD based optimization from random weight initialization of a deep network is found to end up with poor solutions. Deep neural networks with 3 or more hidden layers are found to converge at poor solutions and the error is found to increase after 3 layers as shown in Figure 4.1, where test error results are obtained on shallow (one hidden layer) and many hidden layers network based on error back-propagation algorithm. For each network depth, layer weights are initialized randomly from Equation 3.1. The best hyper-parameters found on validation set are ϵ=0.1 for learning rate

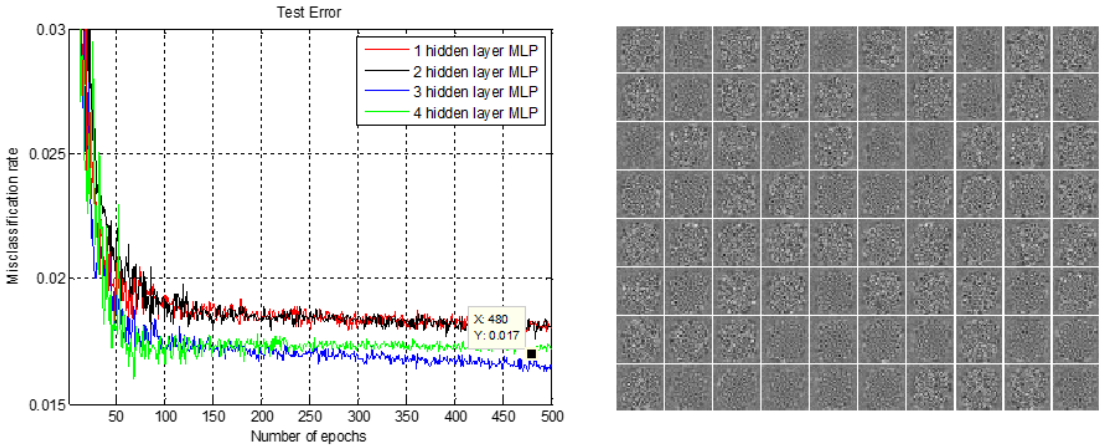

Figure 4.1: **Right:** Test error curves for different depths of MLP model with no pretraining. **Left:** The weights learned present no any meaningful edge or curves and is more like a random distribution.

MLP architecture	Train error %	Test error %
Shallow(1 hidden layer) network	0.071%	1.80 %
2 hidden layers network	0.029 %	1.76%
3 hidden layers network	0.013%	1.63%
4 hidden layers network	0.067%	1.70%

Table 4.1: Misclassification error rate of MLP networks on different layer depths with no pre-training

and β=0.5 for momentum. For all models layer size s_l is selected as 1000 for each layer. Logistic function and softmax activation function are used for hidden layers and output layer respectively. The test errors obtained on different layer architectures are given in Table 4.1 which shows that we are unable to effectively train MLP models for more than 3 layers without pre-training. Each image patch in Figure 4.1(Left) shows the visualization of the weight vector of all of the inputs and a specific hidden unit from Equation 3.27. Grey color shows weight value of 0, black represent weight value of negative and white color stands for a weight value that is positive. The network has not learned a meaningful pattern such as strokes or edges from the hand written digits that may characterize the different characters.

Without pre-training, the lower layers are initialized at poor regions but still the top layers learn the input set almost perfectly. The top layers being closer to the output layer form a shallow fat network [9].

4.3 Model Selection

A neural network model consists of many hyper-parameters. Selection of these values for the final model are decided on $\mathbb{D}_{valid}$ through some experiments to analyze the individual effect of each parameter on the model's performance and to obtain its optimal value. Early stopping criteria is used on the validation set $\mathbb{D}_{valid}$, which means that networks are not allowed to reach local minima due to the time and computation cost. Through out the experiments, we pretrain the network using an autoencoder (AE) or its variants for single hidden layer network and stack autoencoder (SAE) or its variants for deep neural network, whose training includes two steps: pretraining and finetuning. This training procedure is the way through which we train our deep neural models.

4.3.1 Effect of Layer Size

The experiment is performed on simple autoencoder without any regularization for few epochs to analyze the behavior of the test error curve on the increased size of the hidden layer of an autoencoder. For 1 hidden layer neural network, one autoencoder is pretrained with $\epsilon = 0.1$, $\beta = 0.5$, logistic function at the corresponding hidden and output layer for 50 epochs. The first layer weight of the neural network is replaced with the encoder of the autoencoder and is fine-tuned with a softmax output layer with the same ϵ and β values. The procedure is repeated for the same network setting with different hidden layer size e.g. 25, 50, 100, 200, 400, 800, 1000. Similarly, a neural network with 2 hidden layers can be pretrained with SAE (stack autoencoder). After SAE-1 (first AE in the stack) is pre-trained, the hidden code vector is put as an input for the

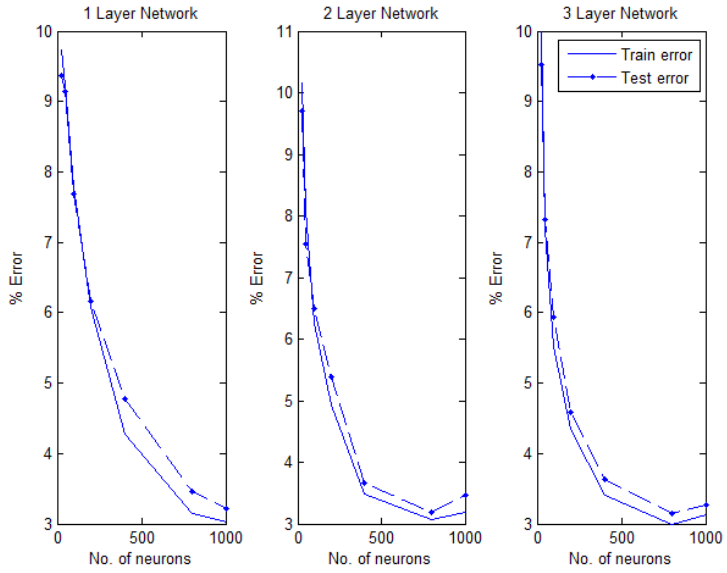

Figure 4.2: Effect of layer size on SAE(stack autoencoder), 1-3 hidden layer network on MNIST dataset

SAE-2 (second autoencoder) and is pretrained accordingly. The network is fine-tuned when encoder of SAE-1 is put at the first hidden layer weight and encoder of SAE-2 is put at the second layer weight of the final network followed by a softmax output layer. Again, this procedure is repeated for all the given number of hidden units. The experiment is repeated for three hidden layer deep network with 3 autoencoders stacked on the top of each other. From Figure 4.2 it is very obvious that pretraining hurts performance with smaller layers due to the fact that it acts as an additional regularizer. Small networks have a limited capacity and further addition of any bias (pre-training) can restrict its performance. As the layer size increases, the effect of pre-training shows its usefulness towards good generalization. Usually for SAE, 800 number of neurons is optimal but if other regularizers are added in addition to improve the performance then larger number of neurons is preferred. Through out our experiments we have set 1024 as an optimal layer size.

4.3.2 Effect of Activation Functions

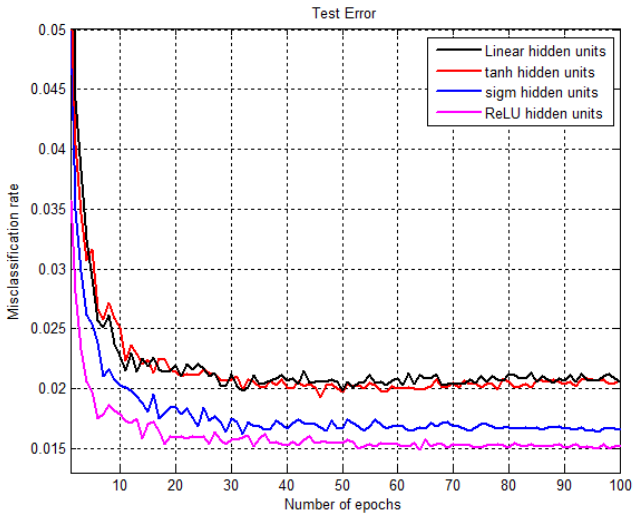

Figure 4.3: Effect of different activation functions on autoencoder with mean square error criteria with logistic units at reconstruction/output layer of the decoder

In this experiment we are using network architecture (784-1000-10), where input layer is the image pixels consisting of 784 units with 1000 hidden layer units and 10 output softmax units to analyze the effect of different activation functions (Section 2.2). The first layer pre-trained weight vector of the network is obtained from the encoder layer of AE with $\epsilon = 0.01$, $\beta = 0.3$ parameters and is then fine-tuned by placing a softmax output layer. Fine-tuning step includes parameter setting as decaying learning rate with $\epsilon = 1$ with a multiplying factor of 0.998 on each epoch from Equation 3.1.3. If neural network uses ReLUs as hidden units, then hidden layer of corresponding autoencoder also consists of ReLUs for pre-training step. However, output layer of each AE uses logistic unit to squash the outputs between 0 and 1 for reconstruction of inputs. Same steps are performed with different activations to get the test error curves as shown in Figure 4.3. Due to sparse inducing behaviour of ReLUs, it outperforms the accuracy of networks with other activation function and gives more generalized error.

4.3.3 Effect of Optimized Cost

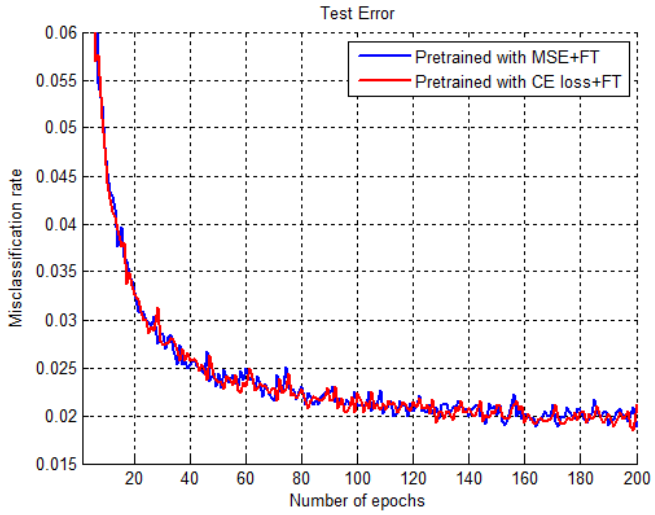

Figure 4.4: Test error curves obtained on cost optimization using cross entropy (CE) and mean square error (MSE) during pre-training phase

As previously stated in Section 2.3.2, we can use any of the two error criterion mean square error (MSE) or cross entropy (CE) for autoencoder training. As in an autoencoder input is reconstructed at the output layer, logistic output units should be used to squash the reconstructed input pixels values between 0 and 1. In Figure 4.4 both types of error criterion is used for the training of autoencoder using $\epsilon = 0.01$ and $\beta = 0.3$. Using pre-trained weight layer from autoencoder, the network is fined tuned using decaying learning rate of $\epsilon = 1$ with multiplying factor $f = 0.998$ from Section 3.1.3 and momentum $\beta = 0.5$. For MNIST data set, no considerable difference of the two training criterion is observed on the network response after fine tuning. So for nearly all the models on the upcoming experiments, we have used MSE with logistic units at the reconstruction layer.

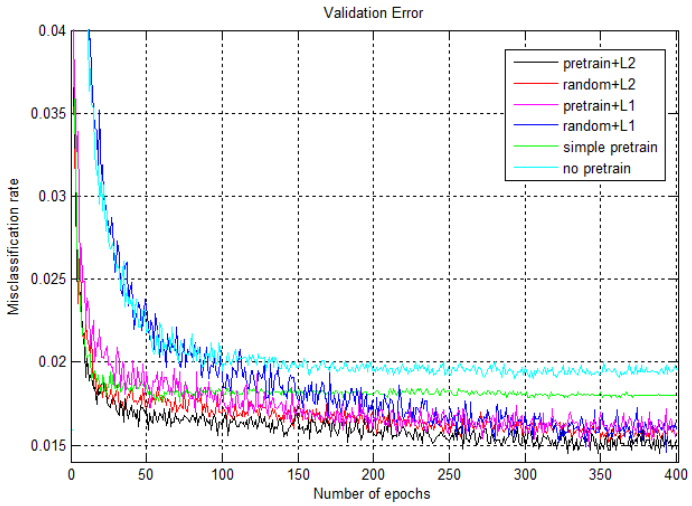

Figure 4.5: Test error curves of standard regularizations on MNIST

4.3.4 Effect of Standard Regularizations

For a shallow (1 hidden layer) network with architecture (784-1024-10), effect of different standard regularizers such as pre-training, ℓ_1 penalty, ℓ_2 penalty and their combine effect has been reported in Table 4.2.

We stated previously that pre-training used for the initialization of a weight layer of a network can be assumed as restricting the optimization procedure to a smaller volume of parameter space and acts as a type of regularizer. Through out our experiments, we use autoencoder structure and its variants for pre-training of a neural network architecture. In this experiment for training of an autoencoder, hyperparameter values are set as $\epsilon = 0.1$, $\beta = 0.3$ on validation set for 200 epochs. After initializing the network from pre-trained weights, the structure is fine-tuned (FT) by few passes of SGD or back propagation algorithm. The hyperparameters used for this step are decaying $\epsilon = 1$ with $f = 0.998$ and $\beta = 0.3$. The improvement obtained on test error by pre-training is shown in Figure 4.5, where the difference in performance is very obvious with a value of 0.13% compare to the error rate obtained from randomly initialized weights of the network.

Method	Test classification error %
Randomly initialized weights	1.93%
Pretrain + FT	1.80%
ℓ_1 and random weights	1.59%
Pretrain and ℓ_1 + FT	1.53%
ℓ_2 and random weights	1.54%
Pretrain and ℓ_2 + FT	1.45%

Table 4.2: Comparison of standard regularizers

The effect of regularizer on the cost function such as ℓ_1 penalty term has also been analyzed. As stated in Section 2.6.2, ℓ_1 penalty avoids weights to explode during training. It shrinks the weights by the same amount and finally leads to zero sparse parameter vectors. When the regularized cost function is optimized by back-propagation algorithm on randomly initialized neural network, a minimum of 1.59% test error is obtained with hyper-parameters $\epsilon = 0.1$, $\beta = 0.3$, and $\lambda = 0.00001$ (weight decay constant). Attempted values of $\lambda = 0.00001$ were $\lambda \in \{0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001\}$. Combine effect of pre-training and ℓ_1 regularization is also experimented and test error obtained is improved to be 1.53%. In this experiment, an autoencoder is first trained with $\epsilon = 0.1$, $\beta = 0.3$, and $\lambda = 0.00001$ for 200 epochs. Then with the encoder weight vector, network is fine tuned using decaying $\epsilon = 1$ with $f = 0.998$, $\beta = 0.3$ and $\lambda = 0.00001$ for 400 epochs. Hence, this network is regularized with both pre-training and ℓ_1 penalty term.

Similarly, the above experiment is repeated with ℓ_2 penalty term. As stated in Section 2.6.2, ℓ_2 shrinks higher weights more than smaller weights but it never leads them towards zero. Firstly, ℓ_2 penalty is added in the error cost and network is trained by back propagation algorithm on randomly initialized weights with the previously defined hyper-parameters. The minimum error obtained is 1.54%. Secondly, when the ℓ_2 weight penalty with pre-training effect (using autoncoder) with the same hyper-parameters is implemented, the test error with a minimum of 1.45% is obtained as given in Table 4.2. Though ℓ_1 penalty induces sparsity in the weights, use of ℓ_2 penalty proved to be better for the defined model setting.

4.3.5 Stack Denoising Autoencoder (SDA)

To understand the qualitative effect of noise level on an autoencoder, denoising autoencoder has been experimented with zero masking noise as before discussed in Section 2.6.4. The corruption level for zero masking of the input is set as 25%, which is proved to be an optimal value [19]. In denoising autoencoder (DAE), corrupted input is given at the input and clean version of the input is reconstructed at the output layer. Similar to an autoencoder, DAE training includes two phases: pre-training and fine-tuning. The performance of stack denoising autoencoder (SDA) from SDA-1 to SDA-3 (1 to 3 number of DAE stacked on the top of one another) has been analyzed using logistic hidden units and rectifier linear units (ReLUs).

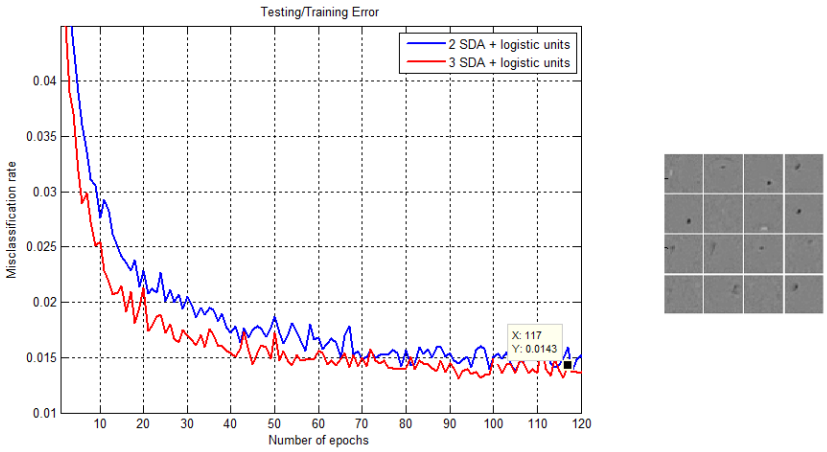

Figure 4.6: **Left**:Test error curves on MNIST dataset with different depths of SDA, each hidden layer with 1024 logistic units and input masking noise of 25%. **Right**: Visualization of learned filters which are more like a local blob detectors.

For pretraining of SDA using logistic hidden units, we tried $\epsilon \in \{0.1, 0.01, 0.001\}$ on validation set and value of 0.01 gave the lowest validation error with $\beta = 0.3$. An additional regularizer like ℓ_1 penalty on the cost function is added to promote sparsity on weight space and to avoid its values from blowing up where λ is set to 0.00001. For fine-tuning phase, the same hyperparameters are used with 25% noisy inputs. Using this setting when deep neural networks

with 2 and 3 hidden layers were trained using SDA technique, test errors of 1.43% and 1.30% are obtained respectively as shown in Figure 4.6.

Similarly, experiment is repeated using ReLUs as the hidden units of denoising autoencoder with the same hyperparameters [20]. To put constraint on the unbounded behavior of ReLUs, ℓ_1 penalty is also used with $\lambda = 0.00001$. For 1 hidden layer, 1 DAE is pretrained. Similarly for 2 and 3 hidden layers, corresponding 2 and 3 stacks of denoising autoencoders are pre-trained. Same hyper-parameters are used for the fine-tuning of each network.

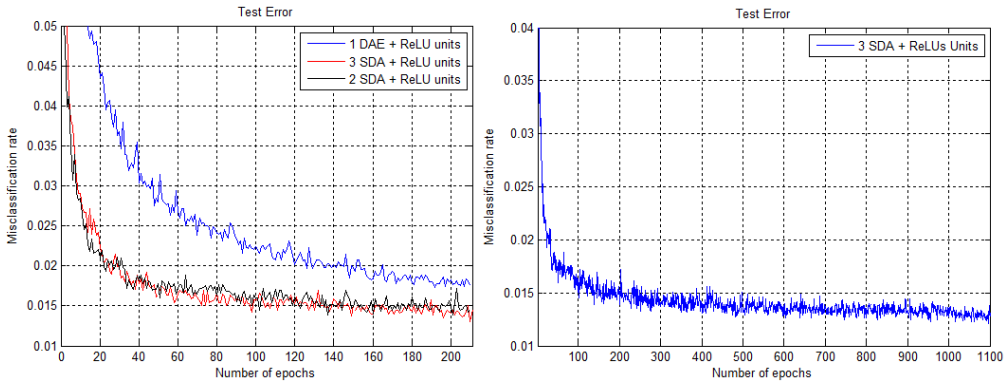

Figure 4.7: **Left:** Test error curves on MNIST dataset with different depths of denoise autoencoders using ReLUs with 1024 number of units per layer. **Right:** Test curve of the network with depth 3 that approaches a minimum of 1.21%

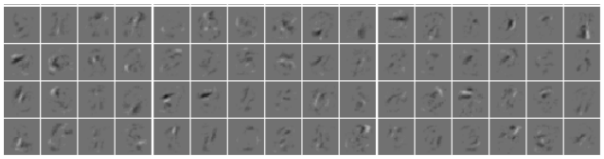

Figure 4.8: Feature visualization of the filter learned by denoising autoencoder using ReLUs which consists of more like parts of characters.

From Figure 4.7 it can be seen that as the depth increases, accuracy increases and reaches as low as 1.30% with 3 hidden layers network for 210 number of epochs. When this model is further continued to fine-tune for 1100 epochs, the accuracy converges to the lowest possible value of 1.21% error. Visualization of

selection of filters learned by this model is given in Figure 4.8.

4.3.6 Importance of Pre-training on Different Layers

This experiment shows the effect of pre-training on different layers of a deep network. We have taken the example of stack denoising autoencoder for the pre-training phase of a network with 2 hidden layers. The hyper-parameters used are input zero masked level of 20%, $\epsilon = 0.01$, $\beta = 0.3$ and logistic hidden units. For fine-tuning phase decaying $\epsilon = 1$ with $f = 0.998$ is used. Some layers of the network are replaced with pre-trained weights and some are initialized randomly. Network with first layer pre-trained performs nearly as good as when both the hidden layers are pretrained. The network with only pretrained second layer performs as badly as all layers initialized with random weights. The test error curves obtained show the importance of pre-training the lower layers, as it makes the input for the second layer. Moreover, fixing the pretrained parameters of first and second layer and training a softmax classifier on the top of it further improves the accuracy when all the layers are fine-tuned jointly. This result is presented by the label "All pre-trained" in Figure 4.9.

4.3.7 Regularization by Sparse Autoencoder

To avoid overfitting, one of the possible ways to initialize a deep neural network by a pre-trained parameters is the weights obtained from two forms of an overcomplete sparse autoencoder as discussed in Sections 2.5 and 3.2 , where a constraint is put on the objective function to allow the neurons to be inactive most of the time. For that purpose, the average hidden activation is regularized to a predefined sparsity hyper-parameter ρ. The estimated average activation of the hidden activations during the training is $\hat{\rho}$. The aim is to minimize the difference of ρ and $\hat{\rho}$.

Previously, Kullback Leibler (KL) measure of the penalty term was used in

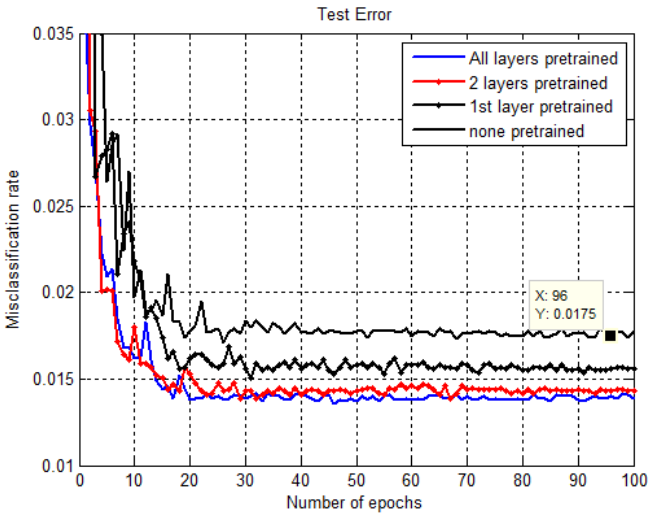

Figure 4.9: Effect of pre-training on different layers of 2 hidden layers deep network pretrained by stack denoising autoencoder.

the sparse autoencoder design for which cost function used is given in Equation 2.30. Using KL penalty in sparse autoencoder gives useful learned pre-trained parameters that can be used to initialize a deep neural network in a good regime. For pre-training phase the parameters such as sparsity parameter ρ, weight decay parameter λ are chosen on validation set $\mathbb{D}_{valid}$. We tried $\rho \in \{0.001, 0.01, 0.05.0.07, 0.1.0.5\}$ and 0.1 is found optimal. Similarly, 0.003 is selected out of tried values $\lambda \in \{0.0001, 0.001, 0.003, 0.01\}$ along with $\epsilon = 0.01$, $\beta = 0.3$ and $\alpha = 2$. For the fine-tuning phase hyper-parameters used are decaying $\epsilon = 1$ with a factor $f = 0.998$ and $\beta = 0.5$. The test error curve obtained is given in Figure 4.10 and with this setting the minimum test error for MNIST dataset is given in Table 4.3.

We propose to take the p-norm of the difference of ρ and $\hat{\rho}$ instead of its KL divergence as discussed in Section 3.2. The problem to be optimized is given in Equation 3.8 for the pre-training of sparse autoencoder with the defined penalty term. Using p-norm of the penalty gives improved performance compared to the KL penalty term. The hyper-parameters used for the pre-training and the fine-tuning phases are the same as used before except some

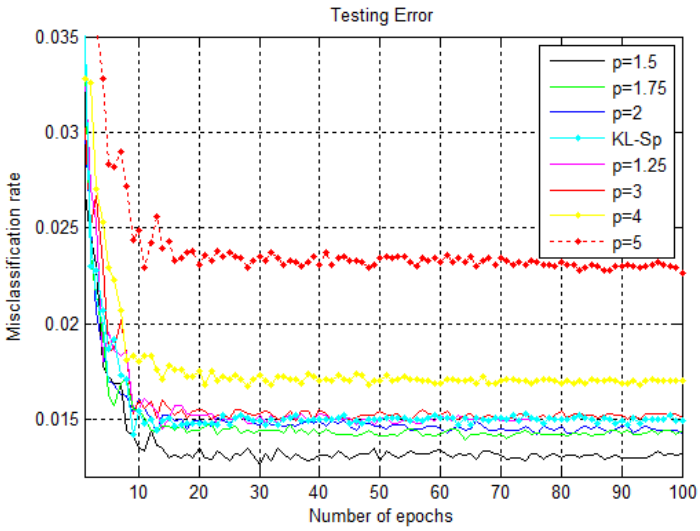

Figure 4.10: Performance comparison of KL-sparsity with different values of p-norm sparsity for a sparse autoencoder

additional hyper-parameters which are $\gamma = 0.1$ (weight constant for bias) and $p \in \{1.5, 1.75, 2, 3, 4, 5\}$. For this setting, the lowest test error is found for $p = 1.5$ given in Table 4.3 which outperforms the KL sparity method. The filter obtained by the p-norm sparse AE is given in Figure 4.11. In filter visualization if the length of the strokes are long, they may not factor the inputs into parts. It may overfit against the validation set. The filters learned here seems to be more local and sharper and can combine to reconstruct any digit.

Sparse penalty function	p	Test Classification error %
p-norm	1.5	**1.28**
p-norm	1.75	1.40
p-norm	2	1.42
p-norm	3	1.49
p-norm	4	1.68
p-norm	5	2.26
KL	-	1.47

Table 4.3: Test error comparison on KL and p-norm sparsity for different values of p on MNIST dataset

Using same hyper-parameters as discussed above, the performance of tied

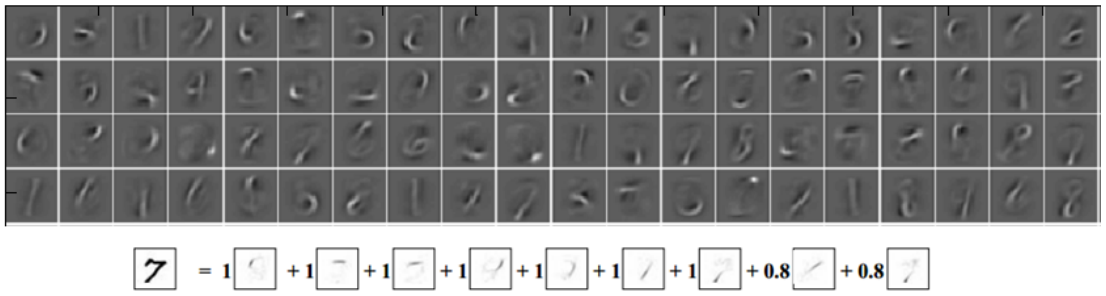

Figure 4.11: **Top**: A subset of encoder filters learned by p-norm sparse AE when trained on the MNIST dataset. **Bottom**: An example of reconstruction of a digit randomly selected from the test data set. The digit is reconstructed by adding "parts" with few weights of the decoder as positive coefficients.

sparse autoencoder and untied sparse autoencoder are compared in Figure 4.12. No considerable difference is realized between the two types. However, p-norm sparse autoencoder with $p = 1.5$ apparently performs better in untied case.

4.3.8 Regularization by Dropouts

Dropping units has been a successful way of avoiding over-fitting in deep neural networks [2, 21]. We have implemented the idea using different layers of deep neural network. The idea is to drop randomly the input and hidden units with a probability p as discussed in Section 2.6.5. The advantage of this method is that no pre-training is required and the process of dropping units is performed during simple back-propagation, which results to give a new thinned network for each pass beacuse of random dropouts. However, it needs many number of epochs to converge to a good solution.

For input layer $p = 0.2$ and for hidden layers $p = 0.5$ is used for each network. The hyper-parameters includes exponentially decaying learning rate of 7 with 0.998 multiplying factor [21]. Max-norm regularizer is implemented for each hidden node and $c = 15$ is selected by cross-validation from Section 2.6.3. Momentum is initiated with 0.5 and is increased linearly over all the epochs to a

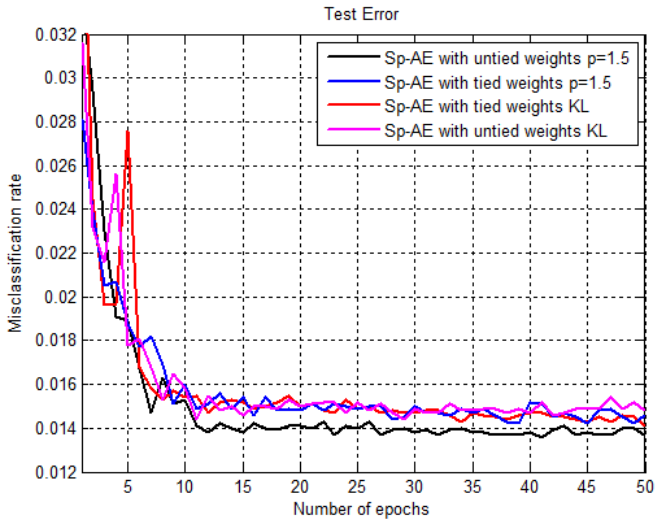

Figure 4.12: **Left:** Test error curves of tied sparse autoencoder with untied sparse autoencoder.

final value of 0.9 from Equation 3.1.4. As the learning rate decays, the algorithm takes smaller steps and finds the right step size for learning progress. High final momentum distributes gradient information over large number of updates making learning stable where each gradient is for a thin network [21]. Features learned by dropouts are simpler and look like strokes of the digits which give better generalization as shown in Figure 4.13.

As for each epoch we get a new thinned network, at test time average of the predictions of all thinned network is obtained. This is achieved by multiplying the outgoing weight of each hidden unit by $(1 - p)$ from Equation 2.41. For $p = 0.5$ each hidden unit is retained with a probablity of 0.5 and at test time the outgoing weights of each hidden unit is averaged by multiplying it with 0.5. We have experimented with up to 3 number of hidden layer deep networks. The minimum test error obtained with 3 hidden layer is 0.93%. With even more large networks, result can be further improved. The combine effect of dropout along with other basic regularization techniques (ℓ_2 weight decay, ReLUs) have been implemented. In each case, dropout gives huge improvement on the network results reported as given in Table 4.4.

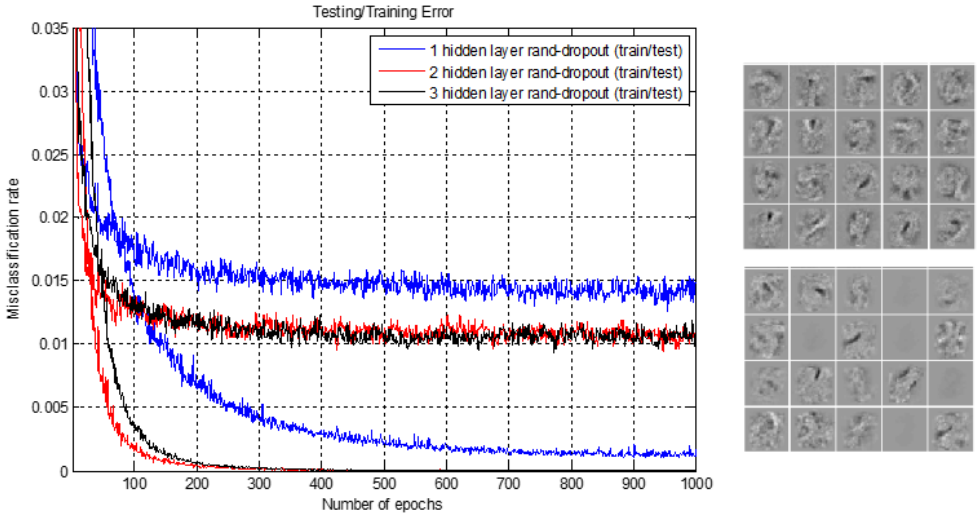

Figure 4.13: **Right:** Test error for different layer architectures with drop out for 1024 units per layer.**Left:** Visualization of features learned (Top) 2 hidden layer network (Bottom) 1 hidden layer network.

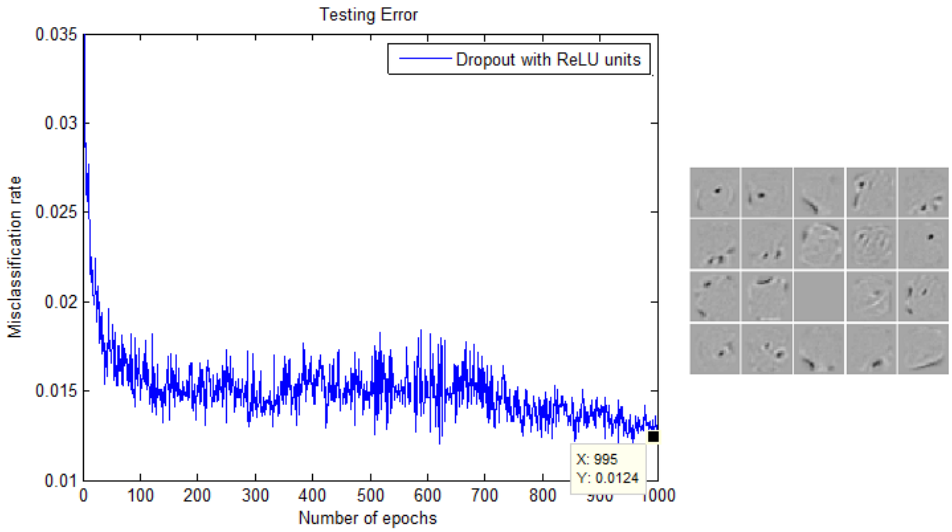

Figure 4.14: **Right:** Test error of dropout method on 2 layer network with ReLUs. **Left:** Features learned on MNIST with one hidden layer autoencoder with ReLUs for $p = 0.5$. The units have learned to detect edges, strokes and spots in different parts of the image.

Dropout out technique can be used as a pre-training step of an autoencoder with the same hyper-parameters, after which it is fine-tuned using simple back-propagation algorithm. The result obtained from this network is same as obtained from the network using dropouts with ReLU i.e. 1.24%.

One of the drawbacks of dropout method is the training time it takes. The parameter updates are very noisy. In each epoch, training is performed on a different random thinned network and it takes many hours to converge to the lowest error. So, this method presents a trade off between accuracy and training time.

Method	Unit type	architecture	Test error (min)%
Dropout NN and ℓ_2	Logistic	2 hidden layers	1.25
Dropout NN and max-norm	Logistic	1 hidden layer	1.30
Dropout NN and max-norm	Logistic	2 hidden layers	1.02
Dropout NN and max-norm	Logistic	3 hidden layers	**0.93**
Dropout NN	ReLU	2 hidden layers	1.24
Dropout AE + FT	Logistic	2 hidden layers	1.24

Table 4.4: Comparison of different models based on random dropout

4.4 k-Lowest Dropout Method

The idea we have proposed is a variant of random dropout method. In this technique dropouts are applied on the input layer in the same random manner but for the hidden layers k-lowest dropouts are applied as discussed in Section 3.3. Each new input hidden activations are sorted in an ascending order and first k-lowest activations are dropped out retaining $\bar{k} = 1 - k$ units to take part during back propagation algorithm. During each epoch based on the input sample, selection of k-lowest activation is equivalent to setting a threshold in a biological neuron. This technique reduces co-adaptation between hidden layers and allows only important information to be learned. Unlike dropout method, it leads the network to converge within few epochs with rectifier linear units and gives classification result nearly the same as that of dropout networks. This technique can be used

as the fine-tuning phase of a pre-trained deep neural network and also as a pre-training phase of an autoencoder or its stack, which are discussed in the following section.

4.4.1 k-Lowest Dropout Finetuning on p-norm Sparse Autoencoder

To evaluate the performance of k-lowest dropout idea as a fine-tuning step, we have implemented it on a pre-trained parameters obtained from p-norm sparse autoencoder.

Pre-training of a p-norm sparse autoencoder is done in the same manner as is discussed in the experiment for $p = 1.5$ in Section 4.3.7. The encoder layers obtained are put at hidden layer weights of a deep neural network. Once the network is initialized at a good regime of parameter space, network is then fine-tuned. The hyperparameters used for this phase are decayed $\epsilon = 1$ with multiplying factor 0.998, $\beta = 0.3$, and 25% of input zero masking noise. The hidden layer activations used are ReLUs and output layer units use softmax functions. The hyper-parameter k is selected on validation set $\mathbb{D}_{valid}$ for $k \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$. The lowest error obtained is for $k = 0.4$ and $\bar{k} = 1 - k = 0.6$ where $\bar{k}$ is the retained number of units during each pass as shown in Figure 4.15. During fine-tuning, we first fix the parameters of the first and second layers and train a softmax classifier on top of the second layer. Then, jointly all layers are fine-tuned. This setting gives minimum test error of 0.99% within 120 epochs shown in Figure 4.17. With $k = 0.5$ the test error obtained is 1.07% within 180 epochs from Figure 4.16. Using this k-lowest arrangement, model works well upto two layers of hidden units. One issue may arise during first few epochs is that the algorithm may greedily assigns each hidden units to a group of input samples. In the following epochs, these hidden units will be picked up again in $\bar{k}$ and the remaining units would not be selected to be given chance to adjust themselves. It means too much sparsity may prevent the gradients of these dead hidden units to update themselves. This problem can be

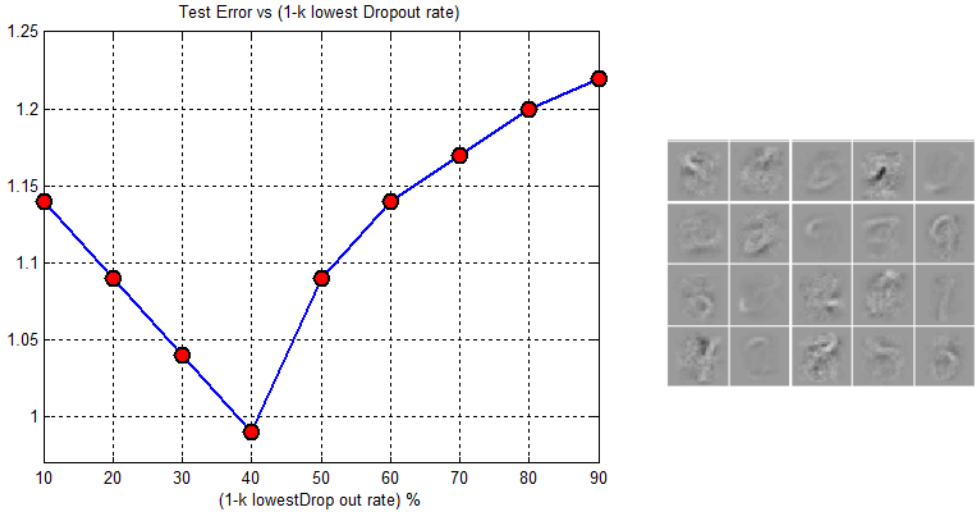

Figure 4.15: **Right:** Classification accuracy versus $\bar{k} = 1 - k$ dropout rate. **Left:** Visualization of features learned by k-lowest dropout. The featuress show local oriented stroke detectors and digit parts such as loops.

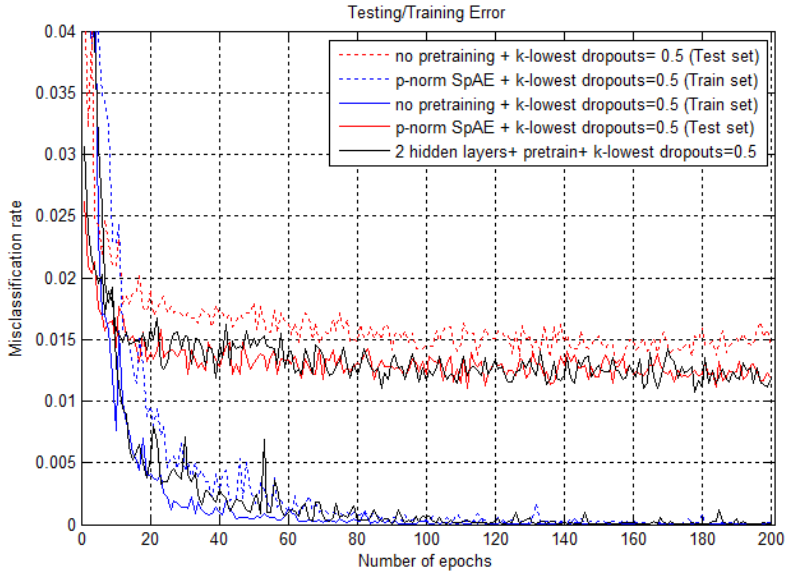

Figure 4.16: Training and testing performances for k=0.5 using k-lowest dropout on different layer networks

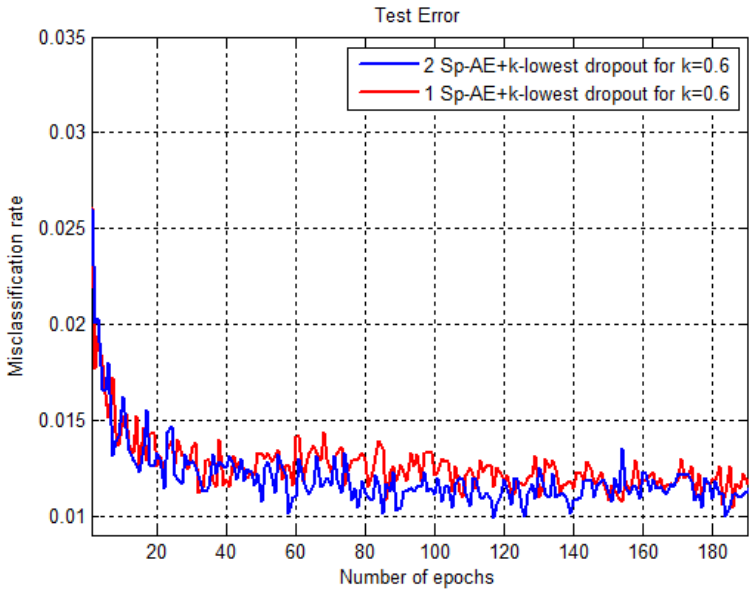

Figure 4.17: Training and testing performances for k=0.6 using k-lowest dropout on different layer networks

addressed by following scheduling of training during fine tuning [22]. Training can be started by assuming $k = 0\%$ (all hidden units active) for first epoch, so that it starts by no any dropout. Then we may linearly increase the dropout level from $k = 0$ to $k = 0.6$ over the first half of the epochs. This trains all the hidden units in such a way so that all of the hidden units have a significant chance to be chosen. For the second half of the epochs, network is maintained to train on $k = 0.6$. With this scheduling of training, we get filters shown in Figure 4.15.

To check the effectiveness of k-lowest dropout idea, experiment is also performed on randomly initialized network without any pre-training phase. Within 140 number of epochs, it reached to a minimum of 1.35% test error as shown in Figure 4.16.

4.4.2 k-Lowest Dropout Autoencoder and Fine-tuning

In this experiment, we evaluate the performance of k-lowest dropout autoencoder as a pre-training phase. The autoencoder is designed with ReLUs at hidden layer and logistic units on reconstruction layer. For training autoencoders hyper-parameters are set as $\epsilon = 0.1$, $\beta = 0.5, \lambda = 0.0001$. ℓ_2 weight decay is also used with input zero masking noise of 25%. The training schedule discussed before for the fine-tuning via k-lowest dropout is also used for pre-training phase. The encoder layer obtained from each autoencoder is put in deep neural network's hidden layers and is fine-tuned with same hyperparameters. This gives good accuracy rate upto hidden layers of three as given in Table 4.5. Deep network with 3 hidden layers which are pre-trained with 3 k-lowest dropout autoencoder stacked on top of the other gives test error of 0.97% within 70 number of epochs. Similarly, improved error of 0.94% is obtained with 2 hidden layer final network in 35 number of fine-tuned epochs.

Unlike random dropouts, both models of k-lowest consume less CPU time for about the same accuracy on MNIST dataset as given in Table 4.7.

Method	Architecture	Test Error(min)%
1.5-norm SpAE + k-lowest dropout FT	1 hidden layers	1.04
1.5-norm SpAE + k-lowest dropout FT	2 hidden layers	**0.99**
k-lowest dropout + FT	1 hidden layers	1.23
k-lowest dropout + FT	2 hidden layers	**0.94**
k-lowest dropout + FT	3 hidden layers	**0.97**

Table 4.5: Comparison of different k-lowest dropout models

4.5 Summary

In order to compare the performance of our algorithm, we trained a number of architectures on MNIST dataset including multilayer perceptron network, denoising autoencoder with logistic units and ReLUs, dropout network with different standard regularizers, sparse autoencoder on KL penalty and p-norm penalty, k-lowest

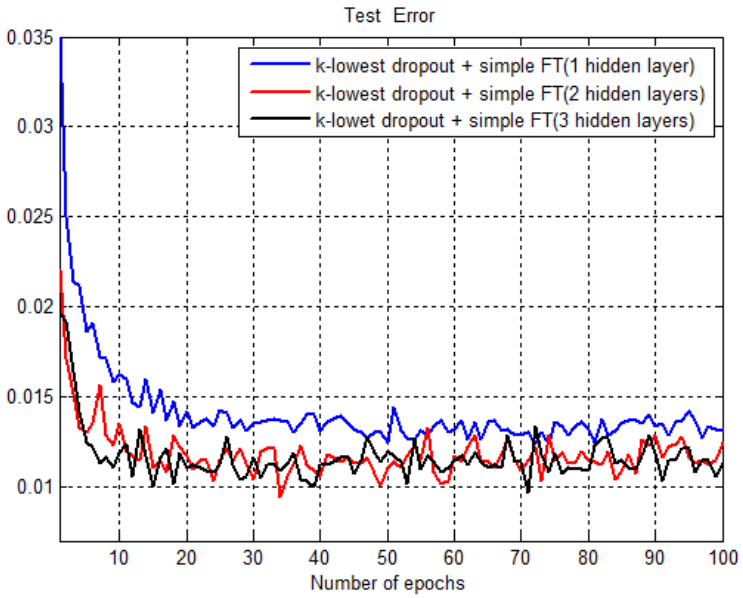

Figure 4.18: **Right:** Test error curve of k-sparse dropout on p-norm sparse AE. **Left:** Test error curve of k-lowest dropout autoenocder with simple fine-tuning.

dropout networks and few hybrid models based on pre-training and fine-tuning phases shown in Table 4.6. The usefulness of each method is then evaluated by examining the error rate of the classifiers.

The deep dropout network obtained lowest misclassification test error on MNIST dataset but from Table 4.7 the convergence time it takes is more than time taken by our proposed model. So, there is a trade off between time complexity and accuracy for deep dropout networks. So, our proposed idea can be considered as a practical choice in deep neural network algorithms.

Method	Architecture	Test Error(min)%
MLP	3-hidden layers	1.63
KL-SpAE + FT	1 hidden layer	1.47
1.5-nomr SpAE + FT	1 hidden layer	1.28
DAE (logistic unit) + FT	3 hidden layers	1.30
DAE (ReLUs) + FT	3 hidden layers	1.21
Dropout NN + max-norm constraint	3 hidden layers	**0.93**
1.5-norm SpAE + k-lowest dropout FT	3 hidden layers	**0.99**
k-lowest dropout + FT	2 hidden layers	**0.94**

Table 4.6: Summary of Comparison on different regularization techniques of deep neural networks

Method	CPU Time(hour)
Random dropout network	12.068 H
p-norm SpAE + k-lowest dropout FT	7.43 H
k-lowest dropout + FT	8.46 H

Table 4.7: Comparison of the CPU time for the minimum error obtained by 2 hidden layers network of different dropout methods on a 2.66 GHz Xeon processor

True/Predicted	0	1	2	3	4	5	6	7	8	9
0	974	1	0	0	0	1	2	1	1	0
1	0	1131	1	1	0	0	2	0	0	0
2	3	1	1022	0	1	0	0	4	1	0
3	0	0	2	1000	0	3	0	1	2	2
4	0	0	2	0	966	0	2	1	0	11
5	2	0	0	4	0	883	2	1	0	0
6	2	2	0	0	1	0	953	0	0	0
7	0	1	7	0	0	0	0	1017	1	2
8	4	0	2	0	0	1	1	2	961	3
9	1	1	1	3	7	3	0	0	0	993

Table 4.8: Confusion matrix of a result obtained from k-lowest dropout network with 0.99% test error

Figure 4.19: The 99 misclassified test cases for the given confusion matrix in Table 4.8 where subscript represents the expected label and superscript represents the network's guess

Chapter 5

CONCLUSION AND FUTURE WORK

This work is motivated by the need to improve regularization techniques for deep neural networks. Regularization is an additional information that is introduced in a model to solve an ill-posed problem to prevent over-fitting. To obtain good generalization, effect of different regularizers were studied first individually and then were experimented with their combine effect. On the basis of experiments following conclusions are made:

1. We have implemented various experiments to evaluate the performance of basic regularizers such as pre-training, ℓ_1, ℓ_2, max-norm constraint, rectifier linear units (ReLUs) on the neural network and obtained the idea of network's behavior with each individual regularization. Pre-training of a deep network is carried out using autoencoder structures. The parameters thus obtained are used for initializing the fine-tuning of a deep network in a good regime. Moreover it has been observed that in a deep architecture, pre-training of lower layers is more important than that of upper layers. Standard regularizers such as ℓ_1 induces sparsity in the parameter space and thus helps to perform feature selection and does not allow the weight values to blow up. But as it is non-differentiable, it works well combined

with other techniques. ℓ_2 weight decay does not allow values to explode. Unlike ℓ_1 weight penalty, it puts more weight decay on large weight values and puts less weight decay on small weight values. Similar weight constraint performs well efficiently by max-norm constraint in networks where high values of hyperparameters are used. Combinations of these regularizers make a model that controls over-fitting. Sparsity can be induced by using a type of activation function called rectifier linear units ReLUs. It has the ability to produce sparseness when used in the hidden layer and itself acts as a regularizer.

2. Other regularization method includes imposing some variations on the autoencoder structures such as Denoising Autoencoder (DA). It learns the representations of the input that are robust to small irrelevant changes in input. Such autoencoders can be stacked to initialize a deep neural network with logistic or rectifier linear units at the hidden layers combined with ℓ_1 penalty term. A deep network trained with this setting leads us to a test error of 1.30% with sigmoid hidden units and 1.21% with ReLUs.

3. Sparse autoencoder is experimented using KL penalty term. We propose to replace this term with the pth-norm and found better performance for 1.5-norm sparse autoencoder than KL-sparse autoencoder. The training ended up with a test error of 1.28%.

4. We implemented the previously proposed idea of random "dropout" and obtained the same results (0.93%) on MNIST dataset as given in the literature. The training time this technique takes is recorded on a 2.66 GHz Xeon processor.

5. We propose k-lowest dropout idea on multilayer network which obtains the same accuracy result as dropout technique gives but with less computational cost comparatively. Instead of dropping out randomly, we sort the hidden units and k-lowest units are dropped out while retaining the activated ones in response to the corresponding input sample. This way computation time is saved by not sampling the hidden units randomly. The rate of convergence need only few fine-tuning epochs.

6. In summary, the principle contributions of this thesis are related to two regularized models. First model includes pre-training through the p-norm sparse autoencoder and fine-tuning with the k-lowest dropout idea. This approach gives an error of 0.99% on MNIST dataset. Second model includes pre-training the autoencoders through k-lowest dropout idea which are stacked on top of each other for the final fine-tuning step through the same k-lowest dropout idea. This approach gives us an error of 0.94%.

For our future work, we plan to increase the network capacity, extend its depth and apply the idea on graphical models such as Restricted Boltzmann Machines (RBMs) and will check its performance on other image dataset such as NORB, CIFAR-10, CIFAR-100, ImageNet, SVHN etc using CUDA libraries on Graphics Processing Units (GPU). Other optimization techniques such as Conjugate Gradient and L-BFGS (Low-memory Broyden Fletcher Goldfarb Shanno) a quasi Newton minimizer could also be incorporated into proposed ideas in this work.

Bibliography

[1] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, pp. 1096–1103, ACM, 2008.

[2] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[3] I. Arel, D. C. Rose, and T. P. Karnowski, "Deep machine learning-a new frontier in artificial intelligence research [research frontier]," *Computational Intelligence Magazine, IEEE*, vol. 5, no. 4, pp. 13–18, 2010.

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[5] Y. Bengio, "Learning deep architectures for ai," *Foundations and trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[6] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[7] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

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

[9] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," *Advances in neural information processing systems*, vol. 19, p. 153, 2007.

[10] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1058–1066, 2013.

[11] H. Lee, C. Ekanadham, and A. Y. Ng, "Sparse deep belief net model for visual area v2," in *Advances in neural information processing systems*, pp. 873–880, 2008.

[12] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[13] C. Poultney, S. Chopra, Y. L. Cun, *et al.*, "Efficient learning of sparse representations with an energy-based model," in *Advances in neural information processing systems*, pp. 1137–1144, 2006.

[14] Y.-l. Boureau, Y. L. Cun, *et al.*, "Sparse feature learning for deep belief networks," in *Advances in neural information processing systems*, pp. 1185–1192, 2008.

[15] Y. W. Teh, M. Welling, S. Osindero, and G. E. Hinton, "Energy-based models for sparse overcomplete representations," *The Journal of Machine Learning Research*, vol. 4, pp. 1235–1260, 2003.

[16] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788–791, 1999.

[17] A. Ng, "Sparse autoencoder," in *CS294A Projects in Artificial Intelligence Lecture notes*, (Standford University), p. 72, 2011.

[18] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 833–840, 2011.

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

[20] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, vol. 15, pp. 315–323, 2011.

[21] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[22] A. Makhzani and B. Frey, "k-sparse autoencoders," *CoRR*, vol. abs/1312.5663, 2013.

[23] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?," *The Journal of Machine Learning Research*, vol. 11, pp. 625–660, 2010.

[24] C. M. Bishop *et al.*, "Neural networks for pattern recognition," 1995.

[25] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, (London, UK, UK), pp. 9–50, Springer-Verlag, 1998.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[27] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*, pp. 437–478, Springer, 2012.

[28] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu, "Sample size selection in optimization methods for machine learning," *Mathematical programming*, vol. 134, no. 1, pp. 127–155, 2012.

[29] P. Golik, P. Doetsch, and H. Ney, "Cross-entropy vs. squared error training: a theoretical and experimental comparison.," in *INTERSPEECH*, pp. 1756–1760, 2013.

[30] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring strategies for training deep neural networks," *The Journal of Machine Learning Research*, vol. 10, pp. 1–40, 2009.

[31] Y. Bengio, A. C. Courville, and P. Vincent, "Unsupervised feature learning and deep learning: A review and new perspectives," *CoRR, abs/1206.5538*, vol. 1, 2012.

[32] A. N. Tikhonov, "On the stability of inverse problems," in *Dokl. Akad. Nauk SSSR*, vol. 39, pp. 195–198, 1943.

[33] A. Y. Ng, "Feature selection, l1 vs. l2 regularization, and rotational invariance," in *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, (New York, NY, USA), pp. 78–, ACM, 2004.

[34] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[35] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proceedings of the 24th international conference on Machine learning*, pp. 473–480, ACM, 2007.

[36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[37] G. Hinton, "A practical guide to training restricted boltzmann machines," *Momentum*, vol. 9, no. 1, p. 926, 2010.

[38] C. C. Yann Lecun, "The mnist database of handwritten digits," http://yann.lecun.com/exdb/mnist/, June, 2014.

Appendix A

Algorithms

Algorithm 1: Stochastic Gradient Descent with Mini-Batching (SGD)

Data: Step size ϵ, mini-batch *size*
Input: Sample $x_1, \ldots, x_m$, $w_1 = \mathcal{N}(0, \sigma^2)$
for $i = 1$ *to* $n = m/b$ **do**
  $l_i(w_i) = \frac{1}{b} \sum_{t=b(i-1)+1}^{b_i} l(w_i, x_t)$;
  $w'_{(i+1)} := w_i - \epsilon \nabla(l_i)(w_i)$
end
return $\bar{w} = \frac{1}{n} \sum_{i=1}^{n} w_i$;

Algorithm 2: Greedy-wise pretraining of autoencoders in deep network (L,J,n,W,b,x). Initialize all layers in a multilayer neural network in an unsupervised way except the last layer, with the greedy layer-wise procedure in which each added layer is trained as an autoencoder that tries t0 reconstruct its input.

Data: step size ϵ, momentum β, input x, reconstruction error criterion $J(\omega, \beta)$, number of layers L, W^i and b^i Weight matrix for level i for i from 1 to L

Input: define $v^0 = x$, sample w^l, w'^l, from uniform$(-r, r)$, $b^l = 0$, $b'^l = 0$

for $l = 1$ *to* L **do**

  define $v^l = f(b^l + W^l v^{l-1})$;

  define $\hat{v}^l = f(c^l + W'^l v^l)$;

  while *not stopping criteria* **do**

    for $i = 1$ *to* $l - 1$ **do**

      compute v^i from v^{i-1};

    end

    compute v^l from v^{l-1};

    compute $\hat{v}^l$ from v^l;

    compute the error criteria J in reconstructing v^{l-1} from $\hat{v}^l$;

    compute $\frac{\partial J}{\partial \omega}$, for $\omega = (W^l, b^l, W'^l, b'^l)$;

    update: $\omega \leftarrow \omega - \epsilon \frac{\partial J}{\partial \omega}$;

  end

end

return W^l, b^l;

Algorithm 3: Stack denoising autoencoder (L,J,W,b,x). The training set is passed through a corruption process and the autoencoder is trained to reconstruct the original input.

Data: step size ϵ, momentum β, input x, reconstruction error criterion $J(\omega, \beta)$, number of layers L, a corruption process to sample noisy input $\tilde{v} \sim q(\tilde{v}|v)$.

Input: define $v^0 = x$, sample w^l, w'^l, from uniform$(-r, r)$, $b^l = 0$, $b'^l = 0$

for $l = 1$ *to* L **do**

  define $v^l = f(b^l + W^l v^{l-1})$;

  define $\hat{v}^l = f(c^l + W'^l v^l)$;

  while *not stopping criteria* **do**

    for $i = 1$ *to* $l - 1$ **do**

      compute v^i from v^{i-1};

    end

    compute $\tilde{v}^{l-1} \sim q(\tilde{v^l}|v^l)$;

    compute v^l from $\tilde{v}^{l-1}$;

    compute $\hat{v}^l$ from v^l;

    compute the error criteria J in reconstructing v^{l-1} from $\hat{v}^l$;

    compute $\frac{\partial J}{\partial \omega}$, for $\omega = (W^l, b^l, W'^l, b'^l)$;

    update: $\omega \leftarrow \omega - \epsilon \frac{\partial J}{\partial \omega}$

  end

end

return W^l, b^l;

Algorithm 4: p-norm sparse autoencoders in deep network (L,J,n,W,b,x). Sparsity parameter ρ is used to induce sparseness in hidden layer of each autoencoder. The cost is optimized according to $J_{sparse}(\omega, \Omega)$.

Data: step size ϵ, momentum β, input x, reconstruction error criterion $J_{sparse}(\omega, \Omega, \beta)$, number of layers L, W^i and b^i weight matrix for level i for i from 1 to L, sparsity parameter ρ, sparsness constant α, batch size n.

Input: define $v^0 = x$, sample w^l, w'^l, from uniform$(-r, r)$, $b^l = 0$, $b'^l = 0$

for $l = 1$ *to* L **do**

  define $v^l = f(b^l + W^l v^{l-1})$;

  define $\hat{v}^l = f(c^l + W'^l v^l)$;

  while *not stopping criteria* **do**

    for $i = 1$ *to* $l - 1$ **do**

      compute v^i from v^{i-1};

    end

    compute v^l from v^{l-1};

    compute $\hat{\rho} = \frac{1}{n} \sum_{i=1}^{n} v_i^l$;

    compute $\Omega = \alpha \|\hat{\rho} - \rho\|_p^p$;

    compute $\hat{v}^l$ from v^l;

    compute the error criteria $J_{sparse}(\omega, \Omega, \beta)$ in reconstructing v^{l-1} from $\hat{v}^l$;

    % J_{sparse} from Equation 3.8;

    compute $\frac{\partial J_{sparse}(\omega, \Omega, \beta)}{\partial \omega}$, for $\omega = (W^l, b^l, W'^l, b'^l)$;

    update: $\omega \leftarrow \omega - \epsilon \frac{\partial J_{sparse}(\omega, \Omega, \beta)}{\partial \omega}$;

  end

end

return W^l, b^l;

Algorithm 5: Fine-tuning of pretrained deep neural network (L,J,W,b,x). Construct L number of mulltilayer neural network and Initialize its l^{th} layer from l^{th} layer of pre-trained autoencoder for supervised training.

Data: step size ϵ, momentum β, input x, output y, reconstruction error criterion $J(\omega, \beta)$, number of layers L.

Input: define $v^0 = x$, W^l and b^l weight matrix obtained from autoencoder for layer l from 1 to L

while *not stopping criteria* **do**

  %Forward propagation ;

  for $l = 1$ *to* $L - 1$ **do**

    $v^l = f(b^l + W^l v^{l-1})$;

  end

  $\hat{y} = \text{softmax } (b^{L-1} + W^{L-1} v^{L-1})$;

  %Back propagation and parameter update;

  compute the error criteria J for prediction $\hat{y}$ and y;

  compute $\frac{\partial J}{\partial \omega}$, for $\omega = (W^l, b^l)$;

  update: $\omega \leftarrow \omega - \epsilon \frac{\partial J}{\partial \omega}$

end

return $\hat{y} \approx y$;

Algorithm 6: Random dropout technique in deep neural network (L,J,W,b,x). Construct L number of MLP network and train it with back propagation algorithm with dropouts at the input and hidden layers with probability q and p respectively.

Data: step size ϵ, input x, momentum β, output y, reconstruction error criterion $J(\omega, \beta)$, number of layers L, W^l and b^l weight matrix for layer l from 1 to L, input layer dropout rate q, hidden layer dropout rate p.

Input: define $v^0 = x$, sample w^l from uniform$(-r, r)$, $b^l = 0$

while *not stopping criteria* **do**

%Forward propagation ;

$r_j^l \sim$ Bernoulli(q);

$\tilde{v}^0 = r^l \bullet v^0$;

for $l = 1$ *to* $L - 1$ **do**

$m_j \sim$ Bernoulli(p);

$\tilde{v}^{l-1} = m^l \bullet v^{l-1}$;

$v^l = f(b^l + W^l \tilde{v}^{l-1})$;

end

$\hat{y}$ =softmax $(b^{L-1} + W^{L-1} v^{L-1})$;

%Back propagation and parameter update;

compute the error criteria J for prediction $\hat{y}$ and y. compute $\frac{\partial J}{\partial \omega}$, for $\omega = (W^l, b^l)$;

update: $\omega \leftarrow \omega - \epsilon \frac{\partial J}{\partial \omega}$

end

return $\hat{y} \approx y$;

Algorithm 7: k-lowest autoencoder training algorithm in deep network (L,J,n,W,b,x). Sort each hidden layer activation of an autoencoder in the stack into v_{Γ^c} and v_{Γ} category. Each input is being zero masked with probability q.

Data: Step size ϵ, momentum β,Input x, Reconstruction error criterion $J(\omega, \beta)$, Number of layers L, A corruption process to sample noisy input $\tilde{v} \sim q(tildev|v)$, W^i and b^i Weight matrix for level i for i from 1 to L

Input: define $v^0 = x$, sample w^l, w'^l, from uniform$(-r, r)$, $b^l = 0$, $b'^l = 0$

for $l = 1$ *to* L **do**

  define $v^l = f(b^l + W^l v^{l-1})$;

  define $\hat{v}^l = f(c^l + W'^l v^l)$;

  while *not stopping criteria* **do**

    for $i = 1$ *to* $l - 1$ **do**

      compute v^i from $\tilde{v}^{i-1}$

    end

    compute $\tilde{v}^{l-1} \sim q(\tilde{v^l}|vl)$;

    compute v^l from $\tilde{v}^{l-1}$;

    $$m_j^l = \begin{cases} 1 & \text{if} m_j^l \in v_{\Gamma^c}^{(l)} \\ 0 & \text{if} m_j^l \in v_{\Gamma}^{(l)} \end{cases}$$

    $\tilde{v}^l = m^l \bullet v^{l-1}$;

    $v^l = f(b^l + W^l \tilde{v}^{l-1})$;

    compute $\hat{v}^l$ from v^l;

    compute the error criteria J in reconstructing v^{l-1} from $\hat{v}^l$;

    compute $\frac{\partial J}{\partial \omega}$, for $\omega = (W^l, b^l, W'^l, b'^l)$;

    update: $\omega \leftarrow \omega - \epsilon \frac{\partial J}{\partial \omega}$;

  end

end

return W^l, b^l;

Algorithm 8: k-lowest dropout technique for fine-tuning in deep neural network (L,J,W,b,x). Find L number of MLP network with its pre-trained weights. Sort each hidden layer activation into v_{Γ^c} and v_{Γ} category. Each input is being zero masked with probability q and train it with supervised back propagation algorithm.

Data: step size ϵ, momentum β,input x, output y, reconstruction error criterion $J(\omega, \beta)$, number of layers L, W^l and b^l weight matrix for layer l from 1 to L, input layer dropout rate q.

Input: define $v^0 = x$, pretrained w^l and b^l, k lowest dropout value

while *not stopping criteria* **do**

%Forward propagation ;

$r_j^l \sim$ Bernoulli (q);

$\tilde{v}^0 = r^l \bullet v^0$;

for $l = 1$ *to* $L - 1$ **do**

$v^l = f(b^l + W^l \tilde{v}^{l-1})$;

$$m_j^l = \begin{cases} 1 & \text{if} m_j^l \in v_{\Gamma^c}^{(l)} \\ 0 & \text{if} m_j^l \in v_{\Gamma}^{(l)} \end{cases}$$

$\tilde{v}^l = m^l \bullet v^{l-1}$;

$v^l = f(b^l + W^l \tilde{v}^{l-1})$;

end

$\hat{y}$ =softmax $(b^{L-1} + W^{L-1} v^{L-1})$;

%Back propagation and parameter update;

compute the error criteria J for prediction $\hat{y}$ and y;

compute $\frac{\partial J}{\partial \omega}$, for $\omega = (W^l, b^l)$;

update: $\omega \leftarrow \omega - \epsilon \frac{\partial J}{\partial \omega}$;

end

return $\hat{y} \approx y$;