A thesis for the Degree of Master of Science

Matlab Based Machine Learning Software for Education

Waseem Abbas

Department of Ocean System Engineering

GRADUATE SCHOOL

JEJU NATIONAL UNIVERSITY

2016. 06.

Matlab Based Machine Learning Software for Education

Waseem Abbas

(Supervised by Professor Chong Hyun Lee)

A thesis submitted in partial fulfillment of the requirement for the degree of Master of Science

2016. 06

The thesis has been examined and approved.

Thesis director, Jinho Bae, Professor, Department of Ocean System Engineering

Chong Hyun Lee, Professor, Department of Ocean System Engineering

Juho Kim, PhD, Department of Ocean System Engineering

Date

Department of Ocean System Engineering

GRADUATE SCHOOL

JEJU NATIONAL UNIVERSITY

REPUBLIC OF KOREA

To

My Parents and Family

Acknowledgements

I would like to present my humble gratitude to Almighty ALLAH Who bestowed me courage and perservance to complete my studies with quite satisfaction and contentment.

I learnt a lot during my stay in OSE Lab to start my career in research and technology. This is all about the schooling of Prof. Chong Hyun Lee who polished my skills and provided me with guidance whenever I needed it. I would like to pay my heartiest thanks with best regards to Prof. Prof. Chong Hyun Lee, for putting faith in me. During my studies, Prof. Chong Hyun Lee extended his support in every way possible, be it be a financial situation for me or moral guidance I needed. I would also like to thank Prof. Jinho Bae for the advice and counselling. I am also grateful to Prof. Youngchol Choi for mentoring.

I am especially grateful to Mr Shawkat Ali for introducing me to this lab and help me take first steps. I am indebted for his support and advice.

I would like to thanks my colleagues in OSE Lab and friends in the Jeju University as well. Few of the names are coming in my mind including Arshad Hassan, Gul Hassan, Fasihullah Khan, Dr. Kamran Ali, Dr. Safdar Ali, Dr. Rasheed, Irshad Ali, Fazli Wahid, Dr. Jael Lee, Dr. Juho Kim, Lee KiBae, Kim Jhunhwa, Yuhan Choi, Hansoo Kim, Zubair and Afaq. I would like to extend my special gratitude to Dr Juho Kim and Arshad Ali for his kind support and advice in writing this thesis. If it weren't for Dr. Kim, I would have not be able to complete my thesis.

I would like to present my tribute to my humble Father, my late Mother, Sisters, Brothers, and fiancée. I would love to mention my fiancée Arfa Ali for the emotional support and undying love which she provided me throughout this journey. All of my family has always been a source of appreciation and admiration throughout my life either in my professional career or student life.

Last but not the least, I would like to dedicate this dissertation to my late Mother who still shines in my heart and will always be.

In the end, I would like to promise myself to always stay positive, see the best in people, stay true to myself and my duties and keep moving forward through hard work and persistence.

Waseem Abbas

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Machine learning, machine intelligence and artificial intelligence has attracted tremendous attention from industry and academia alike. The rise of machine learning is powered by none other than the zealous researchers who want to create intelligence machines. The topic has grabbed so much attention that machine learning has becoming a must-to-learn and must-to-acquire knowledge domain of today academia. Matlab being one of the most sought after platform for machine learning has attracted considerable attention from researchers. There are multiple matlab based toolboxes available aimed at teaching machine learning. Many of these toolboxes lack GUI support and coherent implementation. This dissertation is an effort to help young researchers grab some basic concepts of machine learning, especially classification through an easy-to-use matlab GUI platform. The classifiers discussed in this document are divided into two categories; shallow classifiers and deep classifiers. Similar pattern is followed throughout the development of the classifiers. In addition to standalone implementation of the classifiers, a GUI interface is developed for their comparison as well. The toolbox is tested on real world datasets to benchmarked against open source code snippets of the same classifiers. As the toolbox is developed with the sole purpose of convenience in mind, its target audience are researchers and instructors. The instructors can use this toolbox to teach a diverse range of classifiers. The researchers can use it to learn how classifiers work by employing them on their data. It can also be used to produce publishable results.

Key words: Matlab, Machine Learning, Algorithms, Classification, Accuracy, Prediction, Features, GUI.

ABBREVIATIONS AND NOTATIONS

ML	Machine learning
AI	Artificial intelligence
SNR	Signal to Noise Ratio
GUI	Graphical User Interface
OAO	One –Against -One
OAA	One-Against-All
LDA	Linear Discriminant Analysis
LSC	Least Squares Classifier
NB	Naïve Bayesian
PCA	Principal Component Analysis
SVM	Support Vector Machines
SMO	Sequential Minimal Optimization
QP	Quadratic Programming
LS	Least Squares
ELM	Extreme Learning Machines
DBN	Deep Belief Networks
RBM	Restricted Boltzmann Machine
HELM	Hierarchical Extreme Learning Machine
MLP	Multi-Layer Perceptron
CNN	Convolutional Neural Network

Chapter 1

INTRODUCTION

1.1. Machine Learning

Learning, like intelligence, has no precise definition. A dictionary definition goes like "to gain knowledge, or understanding of, or skill in, by study, instruction, or experience." Although often linked with living beings when mentioned, the same concept of learning is applicable to machines as well. If a machine is designed to learn performing a specific task from experience or data, it is referred to as *machine learning (ML)*. There isn't any concrete definition of machine learning as well. Most referred to definition is the one given by Tom Mitchell in 1997 as: "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E" [1].

ML should not be confused with pre-coded rules based decision models. A computer program which computes (or gives) an output based on states of inputs isn't necessary a learned program. For example, we know exactly how a logic gate works, we have two or more inputs and we some predefined rules (equations) dictate output states to these inputs. This is possible because we know all the possible combinations of inputs, but if all the possible combinations of inputs cannot be mapped exactly by one general equation, ML comes into play. ML solves problem that cannot be solved by numerical means alone.

As ML programs are supposed to learn from experience, they can be broadly classifier into two categories based on experience, supervised learning and unsupervised learning. In supervised learning, the program is *'trained'* on a pre-defined set of *'training examples'*, which then facilitate its ability to reach an accurate conclusion when given new data. On the other hand, in unsupervised machine learning: The program is given a bunch of data and must find patterns and relationships presents inside the data by optimizing some cost function.

In supervised learning applications, the goal is to fine tune a predictor function $h(x)$ where x is a vector specifying properties of observations on which the predictor function is applied. In practice, x is almost always multi-dimensional. Let's express the predictor by Eq. (1.1).

$$h(x) = \theta_0 + \theta_1 x \tag{1.1}$$

where $\theta_0 \; and \; \theta_1$ are constants. The goal of machine learning is to find the perfect values of $\theta_0 \; and \; \theta_1$ to make the predictor work as accurate as possible.

Optimizing the predictor $h(x)$ is done using *training examples*. For each training example, we have an input value x_{train}, for which we know the corresponding output, y in advance. Using the already known value, we find the difference between the correct value y, and predicted value $h(x_{train})$.With enough training examples, the differences help in measuring the *'wrongness'* of $h(x)$. We can then make $h(x)$ less wrong by tweaking the values of $\theta_0 \; and \; \theta_1$. This process is repeated over and over until the system has converged on the best values for $\theta_0 \; and \; \theta_1$. In this way, the predictor is said to be trained.

Based on task at hand, ML algorithms are broadly classified into five categories. These categories are classification, regression, clustering, density estimation and dimensionality reduction. This document deals with ML algorithms used for classification.

1.2. Classification

Classification is identifying to which of a set of groups (sub-populations) a new observation belongs on the basis of knowledge gained from observations (or instances) whose category membership is known. It falls under the umbrella of supervised learning, i.e. learning where a training set of correctly identified observations is available.

The observations are defined by some properties. These properties may be categorical (e.g. *'A'*, *'B'*, *'AB' or 'O'*, for blood type), ordinal (e.g. *'large'*, *'medium'* or *'small'*), integer-valued (e.g. the number of occurrences of a part word in an email) or continuous (e.g. a measurement of blood pressure). Terminology for these properties varies. In statistics, the properties of observations are termed explanatory

variables (or independent variables), and the classes to be predicted are termed as outcomes. In machine learning, the observations are often known as *instances*, the properties are termed *features* (grouped into a feature vector), and the possible categories to be predicted are *classes*.

An algorithm that implements classification is known as a *classifier*. The term *'classifier'* sometimes also refers to the mathematical function, implemented by a classification algorithm that maps input data to a category.

Classification can be treated as set of two separate problems – binary classification and multiclass classification. In binary classification, only two classes are involved. In multiclass classification, a new instance has to be labelled one of many classes. Since most of classification methods have been developed specifically for binary classification, multiclass classification is done by using multiple binary classifiers. This is done in one of the two ways, one-against-one approach and one-against-all approach.

In *one-against-all* (OAA), a single classifier is trained per class. Samples of that class are treated as positive samples while non-class samples are treated as negative samples. In this strategy, the base classifiers are trained to produce a real-valued confidence score for its decision, rather than just a class label. Reason behind this approach is that discrete class labels alone can lead to ambiguities as multiple classes can be predicted for one sample.

In the *one-against.-one* (OAO) strategy, $K\,(K-1)\,/\,2$ binary classifiers are trained for a K-multiclass problem. Each classifier receives the samples of a pair of classes from the original training set. It is supposed to distinguish these two classes. At prediction time, a voting scheme is applied. First, all $K\,(K-1)\,/\,2$ classifiers are evaluated on unseen sample, then the class that got the highest number of *'+1'* predictions is assigned as output class.

OAO strategy also has some weak points. It fails to correctly classify if two individual classes receive same amount of votes. It also fails to classify if by some random chance, all the classes receive same amount of votes.

1.3. Related work

Machine learning has attracted significant attention from Matlab community. Due to its rising popularity, ML was introduced into Matlab although Matlab itself is optimized for matrix operations [4]. Key features of Matlab ML toolbox are listed as follows.

- Regression techniques, including linear, generalized linear, nonlinear, robust, regularized, ANOVA, and mixed-effects models
- Classification algorithms for supervised ML, including support vector machines (SVMs), decision trees, k-nearest neighbor, Naïve Bayes, and discriminant analysis and neural networks
- Unsupervised machine learning algorithms, including k-means, k-medoids, hierarchical clustering, Gaussian mixtures, and hidden Markov models

For classification purpose, the built in apps and packages include following apps.

- Neural Networks
- Support Vector Machines
- Classification Trees
- Nearest Neighbors Classifiers
- Ensemble Classifiers
- Naïve Bayesian Classifiers

TeraSoft Inc has also developed its own Matlab based ML toolbox [5, 6]. This toolbox (MLT, or Machine Learning Toolbox) provides a number of essential for data clustering and pattern recognition. The Machine Learning Toolbox includes tools that can perform following tasks.

- Train various methods of pattern recognition or data classification.
- Perform input selection and feature extraction

This toolbox however is not accessible from graphical user interfaces (GUIs).

Another toolbox developed by Jason Weston, Andre Elisseeff, Gokhan Bakir and Fabian Sinz treats machine learning algorithms and datasets as objects [7]. Aside from easy use of base learning algorithms, algorithms can be plugged together and can be compared with. The modular nature of this toolbox makes it easier to plug different objects into a modular structure. The classification algorithms offered by this toolbox are mentioned below.

- Support Vector Machine (SVM)
- C4.5 for binary or multi-class
- k-nearest neighbors
- Multi-Kernel LP-SVM
- Bagging Classifier
- Sparse, online Perceptron

Although it's a very comprehensive toolbox, yet it does not support GUIs.

Carl Edward Rasmussen and Hannes Nickisch implemented main algorithms from Rasmussen and Williams: Gaussian Processes for Machine Learning [8, 9]. Although there are many learning algorithms available in this toolbox, it only offers two algorithms for classification. Moreover, the mentioned toolbox is scripts based, there is no GUI support available.

Antonio S. Cofiño, Rafael Cano, Carmen Sordo, Cristina Primo, and José Manuel Gutiérrez developed their own version of ML toolbox called MeteoLab [10]. It has following features.

- Principal Component Analysis (EOF and PCs visualization).
- Clustering (k-means, SOM).
- Canonical Correlation Analysis.
- Regression and time series models.
- Statistical downscaling (local forecast with analogs and k-NN).

- Probabilistic validation (skill scores, ROC curves, economic value curves, etc.).
- Applications of neural networks
- Applications of Bayesian networks

This toolbox also lacks a GUI implementation.

There are plenty of code snippets, source codes, and demos available on internet but all of them take care of one or another algorithm of machine learning. The comprehensive toolboxes available cover ML in general, there is no particular focus on assimilating ML algorithms solely for classification. Moreover, GUI implementation is very rare. Only Matlab built in ML toolboxes offer some support for GUIs but that also for individual algorithms.

1.4. Motivation

Machine learning has gained momentum in academia and industry alike. As advancement in computing technology and the boom in smart devices has resulted in humongous amounts of raw data, traditional data analysis and decision making algorithms are struggling to keep up. As they say, data is the new gold, but this new gold is worthless if not properly used. Consequently, machine learning algorithms are now being sought after the same way software's were sought after once. In Fig. 1.1, we have shown an infographics about the demand of machine learning in both academia as well as industry.

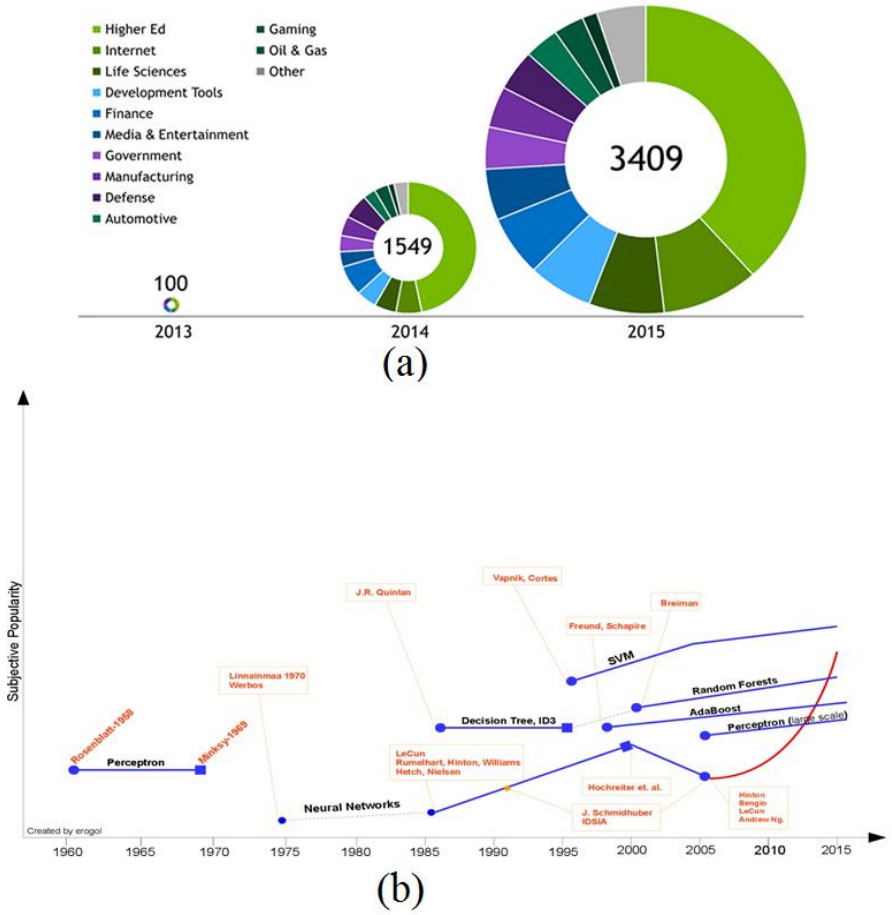

Figure 1.1 (a) Pie chart for number of entities (educational institutes, industrial corporations, government agencies etc.) which expressed interest for collaboration with NVidia Corp in ML. (b) Subjective popularity of ML algorithms [11].

As shown in Fig. 1.1 (a), number of entities who wanted to collaborate with NVidia in machine learning has increased from just 100 in 2013 to 3409 in 2015, a whopping 3400 %. Out of these 3409, more than 40% of these entities are academic institutions. Moreover, there is a steady rise in the subjective popularity of linear ML algorithms while an exponential rise in subjective popularity of deep learning. In Fig. 1.2, we have shown the most popular platforms for Kaggle competitions.

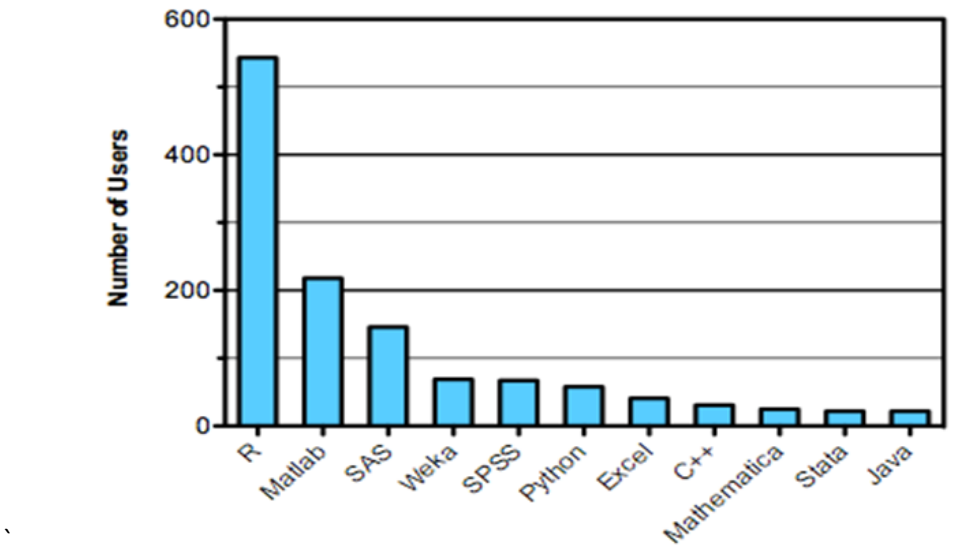

Figure 1.2 Number of Kaggle users per platform, entering a data science competition on Kaggle.com [12].

As shown in Fig. 1.2, Matlab is the second most popular platform for competing in ML open challenges which are routinely posted on Kaggle.com. We have established in literature review that existing Matlab toolboxes do not offer a coherent and GUI based approach for ML research on academia. To offer a GUI based ML platform accessible in Matlab, we have developed a GUI based simulator which supports some of the most widely used classification algorithms. The simulator is developed with sole purposes of accessibility, familiarity and ease of use in mind. The classification algorithms implemented are quite diverse in nature, yet their GUIs are purposely kept similar. The implemented algorithms can be compared to each other as well by following the same information flow as followed in individual classifiers

1.5. Thesis Layout

Thesis is organized in 6 chapters. A brief introduction to shallow classifiers is presented in chapter 2. This chapter discusses the flow chart and basic working principles of three linear classifiers, probabilistic classifiers and neuron based classifiers. Three linear classifiers namely Fisher classifier, least squares classifier and support vector machines (SVM) are discussed along with Naïve Bayesian classifier as probabilistic classifier and extreme learning machines as neuron based classifier.

Deep classifiers are discussed in chapter 3. Their working principle is first discussed followed by discussion of four deep classifiers namely Multi-Layer Perceptron, Convolutional Neural Network, Hierarchical Extreme Learning Machines (HELM) and Deep Belief Nets (DBN).

In chapter 4, the implementation of mentioned classifiers is discussed in details along with their working Graphical User Interfaces (GUI).

Chapter 5 presents the results about GUI performance on two cases of classification; one case is binary classification and the second case is multi-class classification.

In last chapter, the study is concluded along with some recommendations for future work.

Chapter 2

SHALLOW CLASSIFIERS

Since the discovery of statistical methods to the formulation of perceptron sand single layer feed forward networks, focus of machine learning algorithms was on designing classifiers which have one layer of processing elements. As only one layer of abstraction is involved in such classifiers, they are termed shallow classifiers. They are further subdivided into three categories based on the nature of processing elements. The categories are explained in detail in following sections.

2.1. Linear Classifiers

Classification by a *linear classifier* is achieved based on the value of a linear combination of the input features. Let's say input feature matrix x is a real-valued matrix, then output y of the classifier is calculated according to Eq. (2.1).

$$y = f(w^T x) = f\left(\sum_j w_j x_j\right) \tag{2.1}$$

where w is a real vector of weights (w^T represents transpose of w), f is a function that maps input feature matrix onto outputs, x_j is the $j-th$ feature while w_j is weight of feature x_j. The weight vector w is learned from a set of labeled training samples. For a two-class problem, operation of the classifier can be visualized by splitting a high-dimensional input space with a hyperplane; all points on one side of the hyperplane are classified as *'class'*, while the others are classified as *'non-class'*. The hyperplane is found by minimizing a cost function. Operation of linear classifier is represented by flow chart given in Fig. 2.1.

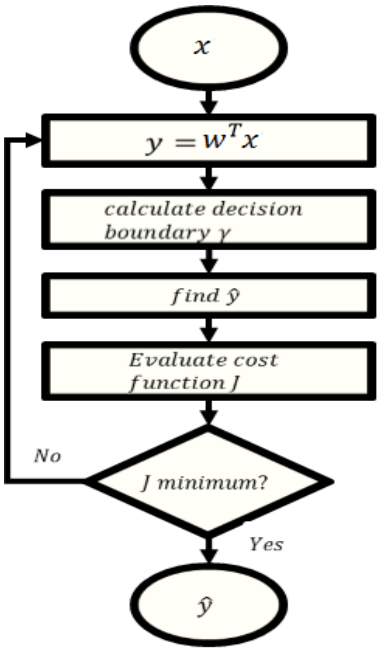

Figure 2.1 Flow chart for linear classifiers, Inputs are first projected on a line, then a threshold is used for classification, J represents the cost function which needs to be minimized and $\hat{y}$ is the output class.

Three linear classifiers are discussed in this chapter namely Fisher linear discriminant analysis (Fisher LDA) classifier, Least squares classifier (LSC) and support vector machines (SVM).

2.1.1. Fisher linear discriminant analysis

Fisher discriminant analysis or linear discriminant analysis represents basically same concept, but the terms are often used interchangeably [13]. It is most often used for dimensionality reduction in the pre-processing step. It projects a dataset onto a lower-dimensional space with good inter class-separability in order to avoid overfitting and reduce computational costs. Fisher's linear discriminant is a classification method which operates in exact manner. It projects high-dimensional data onto a line which can be divided into classes by a threshold. Projection is done in such a way that distance between class-means is maximized while class variance is minimized.

Suppose we have a set of observations described by a feature matrix x and has two classes, class 1 and class 0. Its class means are (μ_1, μ_0) and covariance matrices are $(\sum_1, \sum_0)$. The linear combination of features $w^T x$ (w^T represents transpose of w) will have mean $w^T \mu_i$ and variances $w^T \sum_i w$ where i is

either 0 or 1 where w specifies vector of weights for linear combination. Fisher defined the separation S between these two distributions to be the ratio of the inter-class variance to the intra-class variance [13] as shown in Eq. (2.2).

$$S = \frac{\sigma_{inter}^2}{\sigma_{intra}^2} = \frac{(w^T\mu_1 - w^T\mu_0)^2}{w^T\sum_1 w + w^T\sum_0 w} = \frac{(w^T(\mu_1 - \mu_0))^2}{w^T(\sum_1 + \sum_0)w} \tag{2.2}$$

where σ_{inter}^2 is inter-class variance while σ_{intra}^2 is intra-class variance. He showed that S is maximum if following relationship holds true.

$$w \propto \left(\sum\nolimits_1 + \sum\nolimits_0\right)^{-1}((\mu_1 - \mu_0)) \tag{2.3}$$

Eq. (2.2) is also written as a cost function dependent upon class means and variances. This is called Fisher criterion. For two class problem, the cost function or Fisher criterion expressed in Eq. (2.4).

$$J(w) = \frac{|\mu_1 - \mu_0|^2}{\sigma_1^2 + \sigma_0^2} \tag{2.4}$$

where σ_1^2 variance of class 1 while σ_0^2 is variance of class 0. The cost function is maximized for optimal class separation (classification). This method has strong parallels to linear perceptrons. For two class classification, the separating threshold is learned by optimizing cost function J on the training set. The data projection onto a one dimensional line and resulting separation threshold from Fisher criterion is shown in Fig. 2.2.

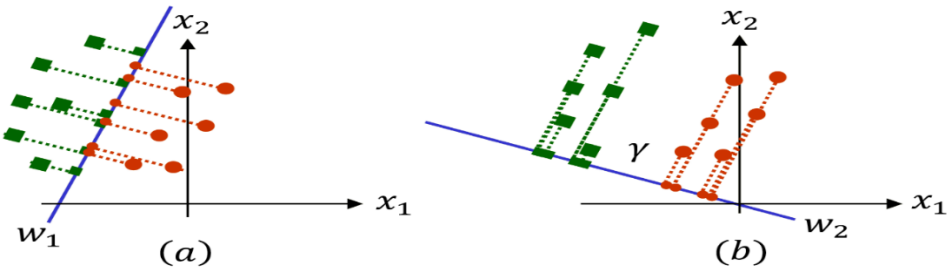

Figure 2.2 Projection of two dimensional data onto one dimensional line. Data is originally represented by two features x_1 *and* x_2. These are projected on a one dimensional line w for better class separation. (a) shows bad projection w_1 which cannot separate classes. (b) shows a good projection w_2 which gives much better class separation. The separating threshold γ is determined by fisher criterion over a training set

2.1.2. Least square classifier

Least squares is another example of linear classifiers in which the base assumption is that output is a linear combination of inputs feature space. The linear combination is often referred to as projected one dimensional space. In mathematical notion, if $\hat{y}$ is the predicted value and x_i is the $i-th$ input feature of p-dimensional input feature space, then $\hat{y}$ is found by Eq. (2.5).

$$\hat{y}(w, x) = w_0 + w_1 x_1 + \cdots + w_p x_p \tag{2.5}$$

where w the weight (coefficient) matrix and x is the input matrix. Fig. 2.2 is a good illustration of projection represented by Eq. (2.5). For two class classification problem, we have to find a good separation of the projected data which divides it in two classes. In least square classifier, the coefficients are calculated to minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation [14]. Mathematically it solves a problem expressed in (2.6).

$$\min_{w}(\|\hat{y} - y\|_2{}^2) \tag{2.6}$$

where $\hat{y}$ is the predicted outputs matrix while y is the actual labels matrix.

2.1.3. Principal component analysis (PCA)

PCA is a dimensionality reduction technique which transforms correlated features into smaller number of uncorrelated features [15]. A PCA model targets the variations in a set of variables by searching for uncorrelated linear combination of correlated variables while preserving the information as much as possible. These linear combinations are called principal components. Two primary objectives of PCA are listed as follows.

- To discover or to reduce the dimensionality of the data set.
- To identify new meaningful underlying variables.

PCA is based on Eigen analysis in which eigenvalues and eigenvectors of a square symmetric matrix are solved with sums of squares and cross product [16, 17]. The eigenvector associated with the largest eigenvalue has the same direction as the first principal component. The eigenvector associated with the second largest eigenvalue determines the direction of the second principal component and so on. The sum of the eigenvalues equals the trace of the square matrix and the maximum number of eigenvectors equals the number of rows (or columns) of this matrix. For an input feature space with *p* variables, *p* principal components are formed. Each principal component is calculated by taking a linear combination of an eigenvector of the correlation matrix (or covariance matrix or sum of squares and cross products matrix) with the variables. The eigenvalues represent the variance of each component.

2.1.4. Support vector machines (SVM)

SVM [18, 19, 20] are based on the concept of hyperplanes which provide a basis for decision boundaries. These hyperplanes separate a set of objects into different subsets having different class memberships. For example, in Fig. 2.3 (a), there are two classes, GREEN and BLUE. The objects belong either to class GREEN or BLUE. Black line in this case represents the decision boundary. On the right side of this decision boundary, all objects are GREEN and to the left, all objects are BLUE. Any new object falling to the right is labeled as GREEN (or labelled as BLUE should it fall to the left of the decision boundary).

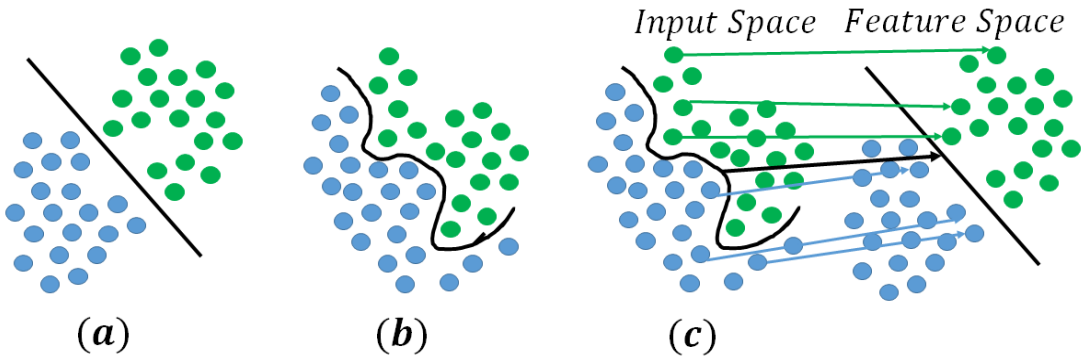

Figure 2.3 (a) Linear separation boundary (b) Complex non-linear separation boundary (c) Projection of nonlinear dimensions to linear hyper-plane.

The above given example represents a linear classifier with a linear decision boundary. Most classification tasks, however, are not linear. Therefore more complex structures are needed for good separation. This situation is depicted in Fig. 2.3 (b). For classification in such cases, nonlinear decision boundaries are used, and they are referred to as separating hyperplanes. Classifiers which use them as decision boundaries are called hyperplane classifiers. SVMs are one of such classifiers.

Fig. 2.3 (c) illustrates the basic idea behind SVMs. The original objects are mapped according to a set of mathematical functions, known as kernels. This process is called mapping (transformation). After kernel operation, the mapped objects are linearly separable and, therefore we only need linear decision boundaries to separate the GREEN and the RED objects.

Goal of SVM algorithms is to find the optimal separating hyperplane that gives the largest minimum distance on training examples. Twice of this minimum distance is referred to as *margin* within SVM's theory. The optimal hyperplane is found as follows.

Let's the optimal hyperplane is a function of input feature space x , weight vector w and bias b as shown in Eq. (2.7).

$$f(x, b, w) = b + w^T x \tag{2.7}$$

The optimal hyperplane can be represented in an infinite number of ways by scaling of w and b . As a matter of convention, the mostly used form is shown in Eq. (2.8).

$$|b + w^T x| = 1 \tag{2.8}$$

The training examples that are closest to the hyperplane are generally called *support vectors*. From geometry, the distance between a point x and a canonical hyperplane (b, w) is found by Eq. (2.9).

$$distance = \frac{|b + w^T x|}{\|w\|} = \frac{1}{\|w\|} = M \tag{2.9}$$

Maximization of M can be treated as an optimization problem of function $J(w)$ ($J(w)$ being the cost function) subject to constraint that the hyperplane should classify all the training examples x_{train} correctly. The constrained optimization of $J(w)$ is expressed in Eq. (2.10).

$$\min_{w,b} J(w) = \frac{1}{2}\|w\|^2 \, , subject\ to\ y_i(b + w^T x_{train}) \geq 1\ \forall i \tag{2.10}$$

where y_i represents each label of the training examples. This problem can be solved using Lagrange multipliers to obtain the weight vector w and the bias b which define the optimal hyperplane.

2.2. Probabilistic classifiers

A classifier which predicts a probability distribution of a given sample input over a set of classes instead of actual class is termed as probabilistic classifier [21, 22]. Probabilistic classifiers provide classification with a degree of certainty. Naïve Bayesian classifier is one of such classifiers.

The Bayesian classification represents a statistical supervised learning method. It assumes an underlying probability distribution which allows it to capture uncertainty by determining probabilities of the outcomes [23]. It is named after Thomas Bayes (1702-1761), who proposed the famous Bayes Theorem. Bayesian classification works on the *'naive'* assumption of independence between every pair of features in the features space. Given a class variable y and a feature vector x_1 through x_p, Bayes' theorem states the relationship in Eq. (2.11) to be true.

$$P(y|x_1, \dots, x_p) = \frac{P(y)P(x_1, \dots, x_p|y)}{P(x_1, \dots, x_p)} \tag{2.11}$$

where $P(y|x_1, \dots, x_p)$ is the conditional probability of output class y given x, $P(y)$ is prior probability of output class y, $P(x_1, \dots, x_p|y)$ is the conditional probabilities of feature space x given output class y while

$P(x_1, \dots, x_p)$ is the prior probability of feature space x. If naïve independence assumption is true, then Eq. (2.12) holds true.

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_p) = P(x_i|y) \tag{2.12}$$

This relationship can be simplified for all i as shown in Eq. (2.13)

$$P(y|x_1, \dots, x_p) = \frac{P(y)\prod_{i=1}^{p} P(x_i|y)}{P(x_1, \dots, x_p)} \tag{2.13}$$

Since $P(x_1, \dots, x_p)$ is constant given the input, classification is done according to Eq. (2.14 a) and (2.14 b).

$$P(y|x_1, \dots, x_p) \propto P(y)\prod_{i=1}^{p} P(x_i|y) \tag{2.14 a}$$

$$\hat{y} = argmax\left(P(y)\prod_{i=1}^{p} P(x_i|y)\right) \tag{2.14 b}$$

Maximum-A-Posteriori (MAP) estimation can be used to estimate $P(y)$ and $P(x_i|y)$; where $P(y)$ is the relative frequency of class y in the training set. Taking different distributions of $P(x_i|y)$ can result in different Bayesian classifiers. Flowchart of this process is shown in Fig. 2.4.

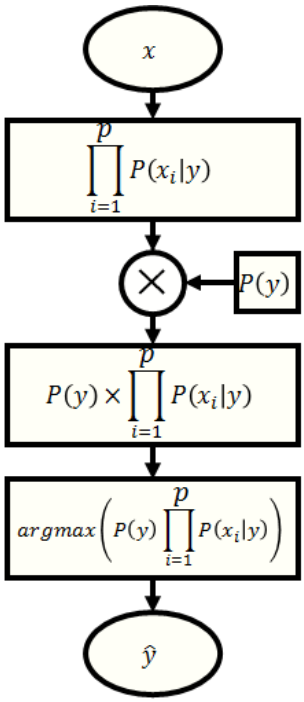

Figure 2.4 Flow Chart of Naïve Bayesian Classifier, Posterior probabilities are updated by taking into account the prior class probabilities and underlying distributions of the features.

In spite of their over-simplified assumptions, naive Bayes classifiers have proven to be powerful classification tools in many real-world application. They require a small amount of training data to estimate the necessary parameters. They can be extremely fast as well when compared to more sophisticated methods

2.3. Neurons based classifiers

Neurons based classifiers use a nonlinear mapping function to decide output class. The mapping functions are normally referred to as *neurons*. Unlike linear and probabilistic classifiers, the projected hyperplane is passed through an activation function before deciding the final output. Inspiration of non-linear activation function comes from biological neurons which are modelled as nonlinear mapping functions of biological stimuli. Extreme learning machines (ELM) represents one of such classifiers.

ELM [25, 26, 27] is an emerging paradigm of learning algorithms which deals with generalized single hidden layer feedforward neural networks (SLFN). Unlike traditional single layer networks, hidden node parameters are randomly generated and the output weights are analytically computed. Guang Bin

Huang introduced the concept of ELM in 2006. They proved that inputs weights and biases can be generated randomly without affecting the output.

In typical SLFN networks, there is an input layer, a hidden layer and an output layer. A standard SLFN is shown in Figure 2.5.

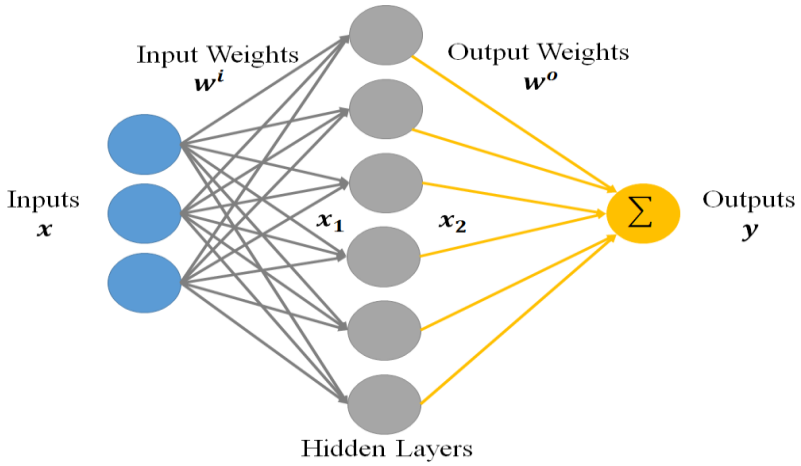

Figure 2.5 Standard single layer feedforward neural network. In case of an ELM, the inputs weights w^i and biases are generated randomly while output wights w^o are calculated analytically.

Let's say an input feature matrix x is introduced to the input layer. Input weights are denoted by w^i, inputs bias by b^i and output weights by w^o, the signal flows as follows in the network.

$$x_1 = \left(w^i\right)^T x \tag{2.15 a}$$

$$x_2 = f\left(x_1 + b^i\right) \tag{2.15 b}$$

$$y = (w^o)^T x_2 \tag{2.15 c}$$

$$w^o = y\, x_2^* \tag{2.15 d}$$

where $\left(w^i\right)^T$ represents transpose of w^i, y is the ELM output, x_2 is the output of nonlinear activation function applied on linear projections of inputs space and x_2^* is the Moore-Penrose pseudo inverse of x_2. It is to be noted that x_2 is not always necessarily a square matrix, so traditional methods of finding its inverse are not applicable. Moore-Penrose pseudo-inverse is a least square solution to this problem as shown in Eq. (2.16)

$$x_2 x_2^* x_2 = x_2 \quad (2.16\ a)$$

$$x_2\, x_2^* = I \quad (2.16\ b)$$

$$\|x_2 x_2^*\|^2 < \varepsilon \quad (2.16\ c)$$

Eq. (2.16 a) and (2.16 b) hold true if x_2 is a square matrix while Eq. (2.16 a) and (2.16c) hold true if x_2 is not a square matrix. Flowchart of an ELM is shown in Fig. 2.6.

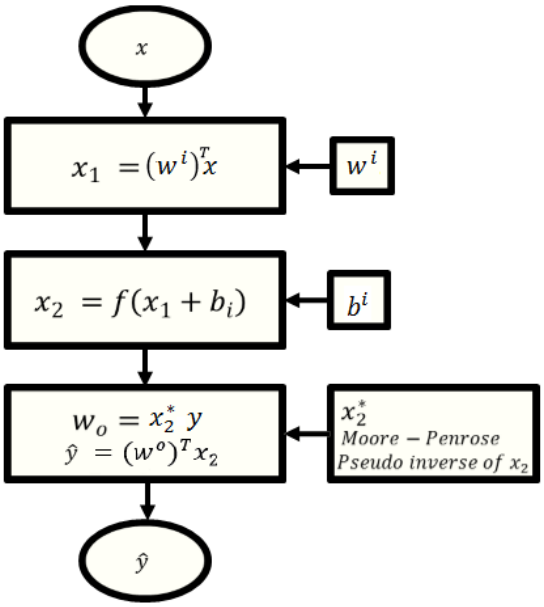

Figure 2.6 Flowchart of extreme learning machines

2.4. Chapter summary

Linear classifiers are the simplest models of learning. They classify a dataset into underlying classes by projecting input feature matrix on one dimensional line. In case of least squares classifier, input features are projected in such a way that least squares error between the targets and outputs is minimum. If the dataset is biased towards a specific class with both the classes have different underlying distributions, least squares projections are not optimal. This problem is taken care of in fisher classifier which ties the projection matrix to class means and class variances. Its performance is degraded when class means are closer to each other or within-class variance is too high. This problem is taken care of in support vector machines which still works on linear projections.

In terms of performance, fisher classifier and least squares classifier are comparable to each other although fisher classifier has s small advantage when class variances are not similar. SVM on the other hand is superior due to its ability to draw nonlinear decision boundaries. However all the three classifier are binary, so they have to be trained in one-against-one or one–against-all manner for multiclass classification. Their performance starts degrading nonetheless in multiclass classification.

Probabilistic classifiers works on product distributions over the original input feature space. Despite vastly simplified probabilistic assumptions, this approach is still proven to be successful on real world applications. Although it is not exactly known why this works so well. These classifiers struggle in multi-dimensional problems or multi-class problems.

Extreme learning machines is rather new concept but it has proven its worth already. The notion that input weights and biases can be generated randomly alone has given good generalization capability to ELM classifiers. Placing a small amount of randomness in classifiers give them some human touch as we humans tend to work on fuzzy logic instead of discrete Boolean logic. Although there is a considerable community in machine learning circles which is still not convinced about the usefulness of ELMs, yet some researchers are proactively presenting ELM as a new paradigm shift in machine learning.

Chapter 3

DEEP LEARNING CLASSIFIERS

3.1. Introduction

Deep learning (also known as hierarchical learning) is a branch of machine learning based on a set of algorithms with multiple processing layers. These processing layers attempt to model high dimensional data and extract meaningful abstractions by using complex nonlinear activations. The layers are composed of multiple nodes of nonlinear activations functions, normally referred to as *'neurons'*. As there are multiple layers of neurons involved representing complex inter-layers interactions, these classifiers are termed as deep classifiers [28, 29, 30].

This chapter deals with deep. Although environment and parameters of the mentioned classifiers are different, they all follow same training flowchart shown in fig. 3.1.

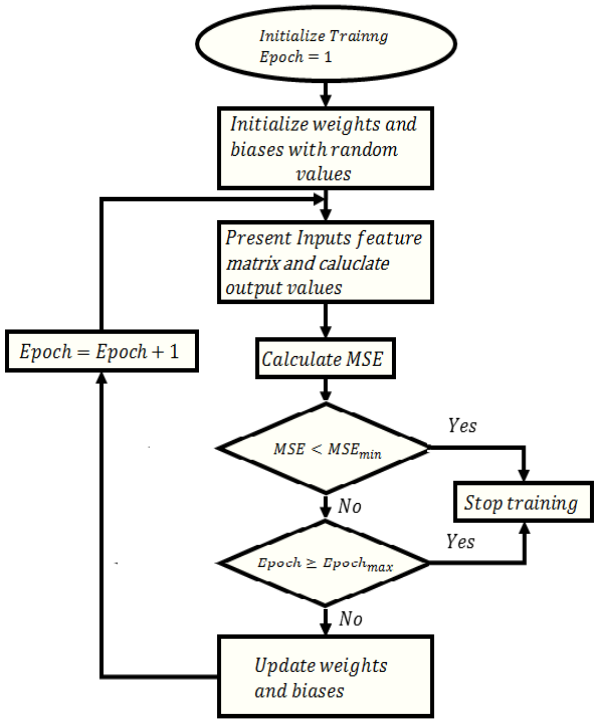

Figure 3.1 General flowchart for deep learning classifiers [31].

3.2. Multi-layer perceptron (MLP)

3.2.1. Architecture

An MLP is a comprised of simple units called '*perceptrons'*. The concept was introduced by Rosenblatt in 1957 [32, 33, 34, 35]. The perceptron calculates a single *output* from multiple real-valued *inputs* by evaluating linear combinations of the inputs according to its input *weights.* It also use one or another form of activation function for the output. This process is expressed mathematically in Eq. (3.1).

$$y = \varphi\left(\sum_{i=1}^{p} w_i x_i + b^i\right) = \varphi\left(w^T x + b^i\right) \tag{3.1}$$

where x_i represents $i-th$ input feature, w_i denotes the weight of input feature x_i, b^i is the input bias vector, x is the matrix form of input feature space, w is weight matrix (w^T represents transpose of w), T represents transpose and φ is the nonlinear activation function. The mapping process of inputs x onto y is shown in Fig. 3.2 (a).

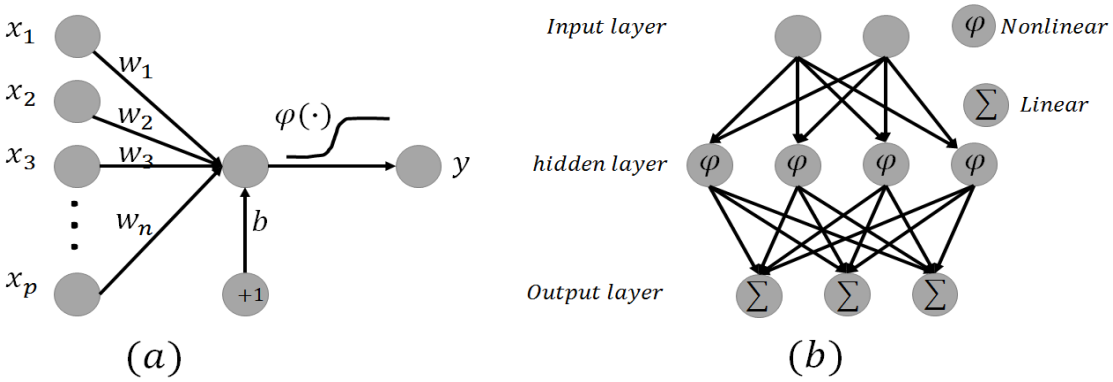

Figure 3.2 (a) Mapping of inputs onto outputs (b) Signal flow of an MLP.

Rosenblatt's originally proposed a *'Heaviside'* step function as the activation function φ for the perceptron. This is not the case anymore. In multilayer networks, the activation function is often chosen to be the logistic sigmoid $\frac{1}{1+e^{-x}}$ or the hyperbolic tangent $\tanh(x)$. They are related by Eq. (3.2).

$$\frac{\tanh(x)+1}{2} = \frac{1}{1+e^{-2x}} \tag{3.2}$$

These functions are preferred because they are close to linear near origin while saturate quickly when getting away from the origin. This allows MLP networks to model nonlinear mappings as well.

A single perceptron is not very useful because of its limited mapping ability. However a network of perceptron is a totally different story [19]. A typical *multilayer* perceptron (MLP) network consists of a source *input layer*, one or more *hidden layers* of computation nodes, and an *output layer*. The input signal is fed layer-by-layer. The signal-flow of an MLP with one hidden layer is shown in Fig. 3.1 (b)

3.2.2. Complexity of MLP

MLP is a fully connectionist model. It is a primitive form of deep learning, which trains by optimizing connection weights among all the nodes. Unlike traditional deep learning in which nodes of a layer are only visible to its adjacent layers, nodes of MLP in any layer are visible to all other nodes. Therefore it sometimes consumes huge amount of memory even though network size is not that big. To avoid such problems, keep the network depth and width as minimum as possible. There are no minimum values for depth (number of layers) and width (number of nodes in each layer) as the training is completely data dependent.

3.3. Convolutional Neural Network (CNN)

3.3.1. Architecture

A CNN [36, 37] is comprised of one or more convolutional layers assisted by a sub-sampling layer. The convolutional and sub-sampling layers are followed by one or more fully connected layers like the standard neural network. The architecture of a CNN is customized to exploit the 2D structure of an input image. They also have smaller number of parameters as compared to fully connectionist models of similar complexity, therefore they are easier to train.

A CNN consists of a number of convolutional and sub-sampling layers optionally followed by fully connected layers. The input to a convolutional layer is an $M \times M$ image where M is the height and width

of the image. The convolutional layer will learn $n \times n$ kernels where n is kernel size. The size of the filters gives rise to the locally connected structures which are convolved with the image to produce k feature maps of size $M - n + 1$. Each map is then subsampled with mean or max pooling over $p \times p$ regions where p ranges between 2 and 5. The Figure 3.3 illustrates a full layer in a CNN consisting of convolutional and subsampling sublayers.

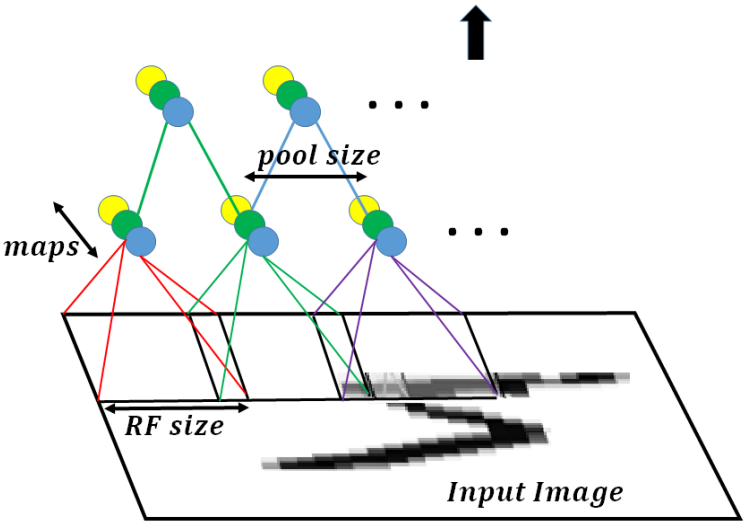

Figure 3.3 First layer of a convolutional neural network with pooling. Units of the same color have tied weights and units of different color represent different filter maps [38].

3.3.2. Back Propagation

Let $\delta^{(l+1)}$ be the error term for the $(l+1)-th$ layer in the network with cost function $J(w, b; x, y)$ where (w, b) represent the weights and biases while (x, y) represent the training data and label pairs. Provided $l - th$ layer is densely connected to the $(l+1)\ -th$ layer, the error for the $l - th$ layer is expressed in Eq. (3.3).

$$\delta^{(l)} = \left(\left(w^{(l)}\right)^T \delta^{(l+1)} \cdot f'\left(z^{(l)}\right)\right) \tag{3.3}$$

and the weight gradients $\nabla_{w^{(l)}}$ and bias gradients $\nabla_{b^{(l)}}$ for $l - th$ are expressed in Eq. (3.4 a) and (3.4 b).

$$\nabla_{w^{(l)}} J(w, b; x, y) = \delta^{(l+1)}\left(\alpha^{(l)}\right)^T \tag{3.4 a}$$

$$\nabla_{b^{(l)}} J(w, b; x, y) = \delta^{(l+1)} \tag{3.4 b}$$

If the $l-th$ layer is a convolutional and subsampling layer then the error is propagated backwards according to Eq. (3.5).

$$\delta_k^{(l)} = upsample(\left(w_k^{(l)}\right)^T \delta_k^{(l+)}) \cdot f'\left(z_k^{(l)}\right) \tag{3.5}$$

where k is the filter number and $f'(z_k^{(l)})$ is the $k-th$ partial derivative of the activation function. The error is propagated backward through pooling layer by calculating error term of each incoming unit of the pooling layer. In max pooling, the unit which was chosen as the max receives all the error.

Finally, to calculate the gradient with respect to the filter maps, the error matrices are flipped the same way as in convolutional layer. This process is express mathematically in Eq. (3.6 a) and (3.6 b).

$$\nabla_{w_k^{(l)}} J(w, b; x, y) = \sum_{i=1}^{m} \left(\alpha_i^{(l)}\right) * rot90(\delta_k^{(l+1)}, 2) \tag{3.6 a}$$

$$\nabla_{b_k^{(l)}} J(w, b; x, y) = \sum_{a,b} \left(\delta_k^{(l+1)}\right)_{a,b} \tag{3.6 b}$$

where $\alpha^{(l)}$ is the input to the $l-th$ layer, and $\alpha^{(1)}$ is the input image. The operation $\left(\alpha_i^{(l)}\right) * \delta_k^{(l+1)}$ represents convolution between $i-th$ input in the $l-th$ layer and the error with respect to the $k-th$ filter.

3.4. ELM based deep network

Extreme learning machine (ELM) is a new paradigm in learning algorithm for single hidden layer feedforward neural networks. The basic idea behind ELMs is that input weights and biases can be generated randomly while output weights can be estimated analytically. However, due to its shallow architecture, feature learning using ELM is not always effective for natural signals (e.g. images/videos).

To address this issue, deep architecture based on ELMs is proposed by many researchers [39]. They follow the concepts of traditional deep architectures; there is an input layer followed by multiple hidden layers and an output layer. The hidden layers perform unsupervised feature learning while final layer performs supervised fine tuning of output weights. In ELM based deep architecture, the hidden layers are based on ELMs instead of restricted Boltzmann machines (RBMs). The final supervised layer can either be an ELM or any other SLFN. A typical ELM based deep network is shown in Fig. 3.4.

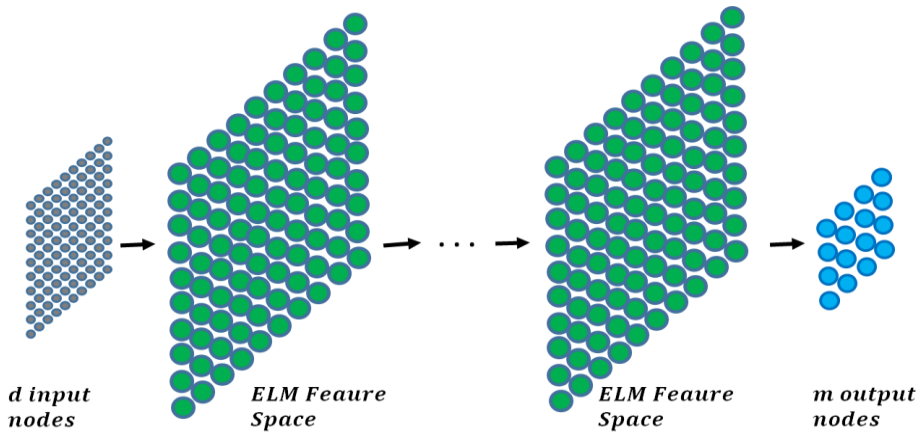

Figure 3.4 Architecture of ELM based deep network (The hierarchical ELM) [33].

As shown in the Fig. 4.4, input is first passed though ELM based auto-encoders which projects input features space to a different ELM feature space at each level. At the final level, the learned ELM feature space is mapped onto output via traditional supervised learning.

ELM auto-encoders operate differently than RBMs. ELM auto-encoders need an output to learn features, although it is supposed to learn the features in an unsupervised manner. This is achieved by presenting the input as output to the ELM. Let's say x is the input to the auto encoder, w^i represents the input weights, b^i represents input bias matrix and w^o represents output weight matrix. Features are stored in output weight matrix which is found by Eq. (3.7 a) and (3.7 b).

$$x_2 = f((w^i)^T x + b^i) \qquad (3.7\ a)$$

$$w^o = x\ x_2^* \qquad (3.7\ b)$$

where x_2^* is the Moore-Penrose pseudo inverse of x_2 . It is calculated using Eq. (2.16 a) ~ (2.16 c).

It is to be noted that the only difference between a standard ELM and an ELM auto-encoder is that for ELM auto-encoder, the input is used as the output as well to learn the output weights. It is similar to ELM in its operation and structure.

3.4. Deep belief networks (DBN)

DBNs are probabilistic models composed of multiple layers of neurons. The top two layers have undirected, symmetric connections between them while the lower layers receive top-down, directed connections from the layer above [40, 41, 42].

The two most significant properties of deep belief nets are following.

- Weights are learned layer-by-layer.
- The values of the nodes in every layer can be inferred by a single, backward pass that starts with an observed data vector in the final layer and uses the weights in the reverse direction.

Basic building block of DBN as a restricted Boltzmann Machine (RBM). Introduced by Hinton et al. RBM is a two layered structure where the subsequent layer learns patterns in inputs presented to the previous layer by minimizing a cost function.

Hinton showed that RBMs can be stacked and trained in a greedy manner to form DBNs. DBNs models the joint distribution between observed vector x and the $l-th$ hidden layer h^l according to Eq. (3.8).

$$P(x, h^1, \dots, h^l) = (\prod_{k=0}^{l-2} P(h^k|h^{k+1}))\, P(h^{l-1}, h^l) \tag{3.8}$$

where $x = h^0$, $P(h^k|h^{k+1})$ is a conditional pdf for the visible units dependent on the hidden units of the RBM at level k , and $P(h^{l-1}, h^l)$ is the visible-hidden joint distribution in the top-level RBM. This is illustrated in Figure 3.5.

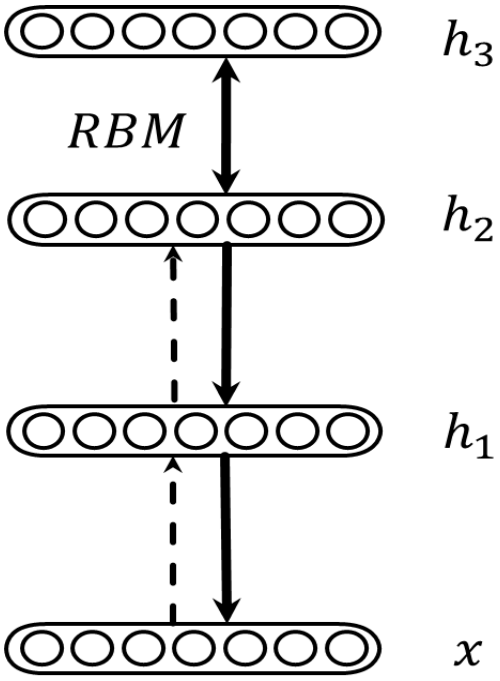

Figure 3.5 Signal flow and dependencies of RBMs [40].

The principle of greedy layer-wise unsupervised training can be summarized as follows.

1) Train the first layer as an RBM such that it models the raw input $x = h^0$ as its visible layer.

2) Data transformed by first layer is used as input to second layer for which two common solutions exist. This representation can be chosen as being the mean activations $P(h^{(1)} = 1|h^{(0)})$ or samples of $P(h^{(1)}|h^{(0)})$.

3) Train the second layer as an RBM, taking the transformed data (samples or mean activations) as training examples (for the visible layer of that RBM).

4) Iterate 2 and 3 for the desired all layers, each time propagating upward either samples or mean values.

5) Fine-tune all the parameters of this deep architecture with respect to a supervised training criterion

3.5. Chapter summary

As discussed above, every architecture of deep learning is different from its counterpart in one way or the other. Although they follow similar training block diagram, yet their parameters space and methods of operations are vastly different. For example, DBNs are composed of individual RBMs. The RBMs are trained in unsupervised manner, so they learn a model of the input distribution from which one can generate samples. Once the RBMs learns feature space, a final supervised layer is added for classification purpose.

MLP on the other hand follows a fully connectionist model where every layer and every node is trained in supervised manner which makes is computationally very expensive.

ELM based deep nets are similar to DBNs in training but the difference is that ELM based deep nets are comprised of ELM-auto-encoders instead of RBMs. Also, the final supervised layer is an ELM too. As discussed in previous sections, an ELM is exclusively a supervised learner, so for ELM auto-encoders, we do not have the luxury of unsupervised feature learning. Instead, the input data is presented to an ELM auto-encoder as the output as well. This way, the ELM auto-encoder learns to project input data onto a different dimension.

CNNs are inspired from human visual cortex. They are used exclusively for image data (or speech data if represented in time-frequency domain). In CNNs, there are normally two layers, one is the convolution layer and second is the sub-sampling or pooing layer. The convolutional layer is used to learn feature space while pooling layer is used to reduce feature dimensions in a way similar to visual cortex.

There are a bunch of other unsupervised representation learning algorithms and there has been only few quantitative comparisons between them. It is very hard to tell in advance which deep learner will perform better on a given dataset. Although deep classifiers give different performance on different datasets, they outperform shallow learners nonetheless.

Chapter 4

IMPLEMENTATION

4.1. Performance measure of classifiers

Before going into the details of implementation, we introduce some performance measures of the classifiers. These measures are used to evaluate how well a classifier is performing on a given dataset.

4.1.1. Precision

Precision is a performance measure of a classifier in terms of how well it can retrieve relevant class labels when it is evaluated on unseen observations. It signifies the fraction of retrieved class labels that are relevant. This performance measure is more insightful in a situation where we are concerned with the ratio of correctly retrieved class labels. For example we have employed a classifier to classify if we should go through a treatment procedure for a patient which has severe side effects, so correct diagnosis is of utmost importance. In this case, we are more interested in the precision of this classifier because we don't care that much about a patient being denied the treatment but we are concerned about the patient being given the treatment because if the diagnosis is wrong, we could exacerbate the patient medical situation.

4.1.2. Recall

Recall is also a performance measure of the classifier. It signifies performance of a classifier in picking as many correct labels of a class as possible. It becomes more insightful if we are concerned with retrieving correct labels of a class labels as much as possible. It range is 0 to 100.

4.1.3. Accuracy

Accuracy is a more general performance indicator of a classifier. It signifies the fraction of correctly classified labels. Its range is 0 to 100.

4.1.4. ROC Curve

In terms of classification, an ROC curve for a class is a graphical representation of relationship between true positives and false positives. In other words, we can say that an ROC curve is graphical representation of precision.

4.1.5. Confusion matrix

A confusion matrix, also known as an error matrix, is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning. Each column of the matrix represents the instances in a predicted class while each row represents the instances in an actual class (or vice-versa). The name stems from the fact that it makes it easy to see if the system is confusing two classes (i.e. commonly mislabeling one as another).

In a confusion matrix, true class labels are represented by rows (or columns) and predicted class labels are represented by columns (or rows if true class labels are represented by columns). Elements on diagonal shows percentage of correctly classifier labels for each class. Other elements correspond to false positives or false negatives depending on the dimension representing true class labels. A typical confusion matrix for 4 class problem is in Fig. 4.1.

	Class1	Class2	Class3	Class4
Class1	98.5369	0.0563	0.7878	0.6190
Class2	0.0489	98.2379	0.8321	0.8811
Class3	0.7852	0.1122	97.5883	1.5143
Class4	0.3840	0.5485	2.7976	96.2699

Figure 4.1 A standard confusion matrix for a dataset containing four classes.

4.2. Overview of implementation

All the classifiers mentioned in chapter 2-4 are implemented in Matlab to develop a compact and easy to use toolbox. There are total 9 classifiers and one dimensionality reduction technique which are implemented. The classifiers are listed as following.

- Linear: Fisher classifier, least squares classifier and support vector machines.
- Probabilistic: Naïve Bayesian classifier.
- Shallow: Extreme Learning Machine.
- Deep classifiers: deep belief networks, convolutional neural networks, hierarchical extreme learning machines and multi-layer perceptron.

Every classifier has its own Graphical User Interface (GUI). As the linear classifiers along with Naïve Bayesian classifier do not have many parameters, they are implemented in one GUI. Other classifiers have their individual GUIs following the same flowchart shown in Fig. 4.2.

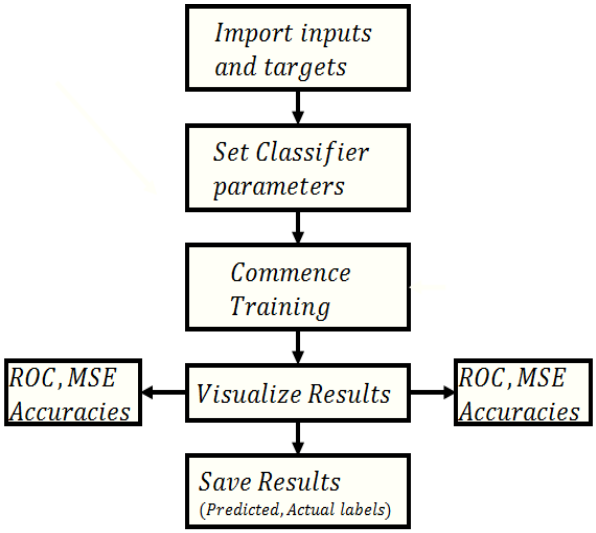

Figure 4.2 Master flow chart of GUIs.

4.3. GUI development environment

All the GUIs are developed using Matlab built in GUI development tool; the *Graphical User Interface Development Environment* (GUIDE). Although Matlab GUIDE tool takes care of all the necessary steps, some general rules are stated below.

- GUIDE tool can handle only one GUI instance, so only one GUI can be developed at a time.
- There is no support for parent-child GUI relationships in GUIDE.
- When GUIDE tool is launched, it automatically creates an opening function to the GUI which runs every time the GUI is launched.
- When an element (e.g. a push button) is added to the GUI, its callback functions is automatically added to the GUI. The call back function can be modified by the user to perform a specific task every time this element is accessed.
- There are several built-in elements available which can be inserted by *drag and drop* method. These elements include push buttons, list boxes, popup menus, editable text boxes, axes, button groups, sliders, editable tables and ActiveX elements.
- As the GUI is a figure in essence, every element of GUI is accessible by a handle called *tag*. These tags are identifiers to GUI elements.
- All the tags are stored in *'handles'* structure every time the GUI is saved. It means that if we want to access any GUI element inside the GUI, we can access it by its tag in *'handles'* structure.
- Manually created elements can also be stored in *'handles'* structure.

These rules are followed by the GUIDE tool of Matlab. We have followed a specific convention while writing callback functions and naming the GUI elements.

- A GUI element which belongs to a specific classifier has name in this format *'classifier name abbreviation – element name'*. For example we want to name push button for importing inputs into GUI space in Deep Belief Network classifier GUI, the name will be *'dbnImportInputs'*.

- All the variable names start with capital letters.
- All the structure names start with small letters.
- All the user defined function names are in small letters.

4.4. GUIs of classifiers

Although the callback functions behind individual GUIs are vastly different, yet their front end is developed similar to each other. Therefore if someone learns to operate one GUI, he/she can operate all the remaining GUIs. In this section, we will discuss the operation of GUI for MLP which is shown in Fig. 4.3

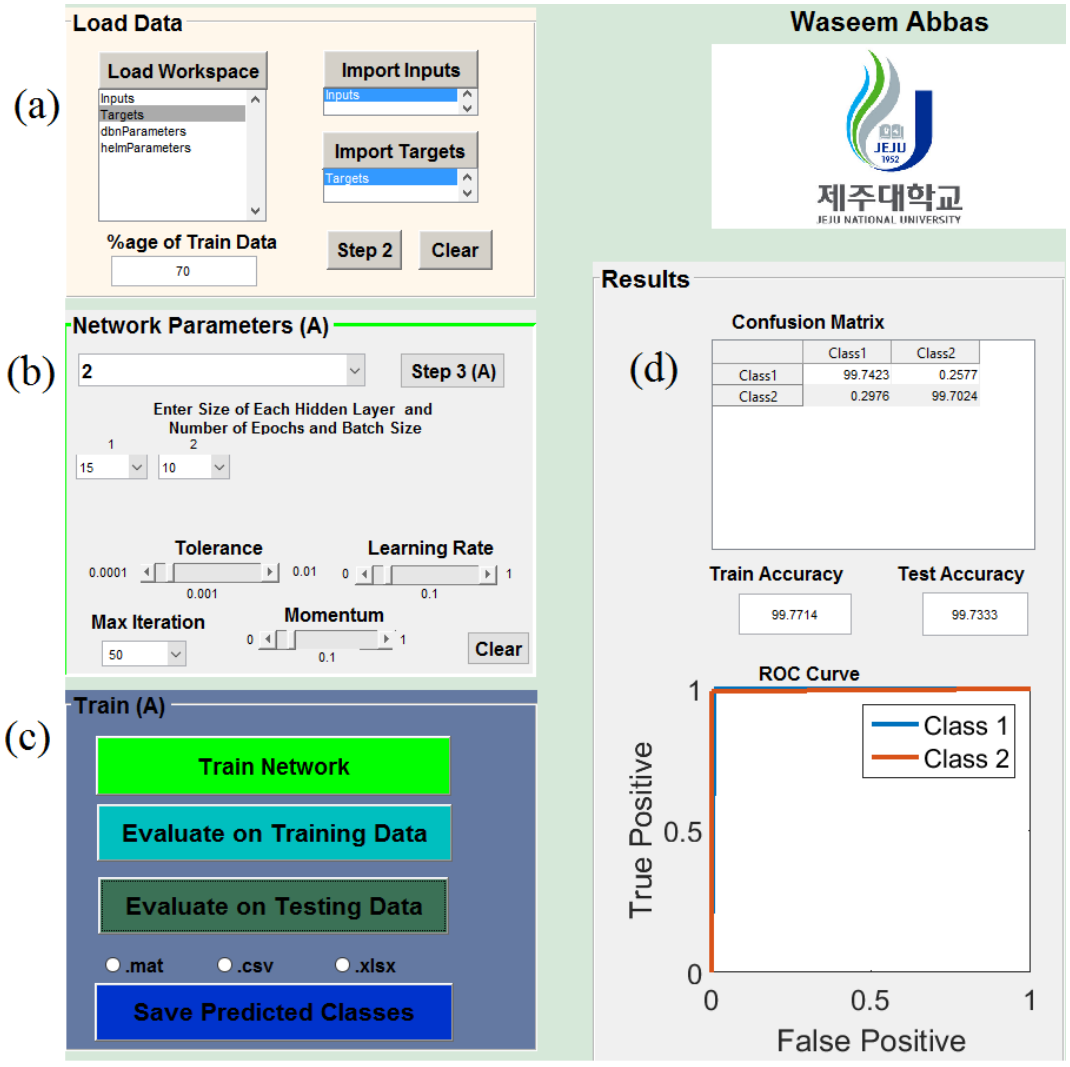

Figure 4.3 Typical GUI of a classifier (This specific GUI belongs to MLP Classifier).

There are four parts of the GUI, Load Data, Network Parameters, Train and Results.

4.4.1. Load Data

This panel is used to load data from workspace. Inputs and Targets should be loaded into Matlab workspace before importing them to GUI space. This panel has four push buttons, three list boxes and one editable text box. The push buttons are *'Load Workspace', 'Import Inputs', 'Import Targets'* and *'Step 2'*. Import and Targets can be imported in following steps.

1) Load data into Matlab workspace.
2) Click on *'Load Workspace'* button. All the variables (Inputs, targets) present in workspace will appear in the list box below the button.
3) Select input variable (also referred to as *feature matrix*) from the list box and then click on *'Import Inputs'*. Inputs variable (*feature matrix*) will be imported to and appear in the list box below the button.
4) Select target variable (also referred to as *labels* or *class matrix*) from the list box and then click on *'Import Targets'*. Target variable (*labels* or *class matrix*) will be imported to and appear in the list box below the button.
5) Once inputs and targets are imported, type the percentage of total observations to be used in training the classifier in corresponding edit box and then hit *'Step 2'* button. Next panel will be activated.

This panel is designed in such a way that it can be expanded to import inputs and target data from files as well. Currently, this panel only directs the GUIs to workspace variables storing inputs and targets. As every classifier must have inputs and targets for training and evaluation purposes, this panel is shared by GUIs of all classifiers. Moreover, there are internal checks placed in this panel which will stop user from progressing to next panels if the data is not in proper format.

4.4.2. Parameters

This panel is used to set parameters for the classifiers. As we know that parameters for classifiers are different than each other, so contents of this panel vary from classifier to classifier. Parameters of each classifier are discussed in the following section one by one.

4.4.2.1. Least squares classifier (LSC)

There is only one parameter accessible to user for least square classifier. This parameter is called *'error tolerance'*. It controls the minimum amount of least square error for convergence. Ideally, the smaller the error tolerance, the better the accuracy. However too small an error tolerance can result in classifier being not able to converge on a solution.

4.4.2.2. SVM

For SVM, 7 parameters need to be set by the user. These parameters are explained below.

Maximum iteration. This parameter controls the maximum number of iterations for SVM training. If SVM does not converges on a solution after searching for iterations more than this number, it will stop searching. It can be selected by setting value of a slider with same name.

Function tolerance: This parameters specifies minimum value of function tolerance. If value of optimization function does not change by an amount greater than this value after successive iterations, convergence is achieved. It can be selected by setting value of a slider with same name.

RBF sigma. It is a positive number specifying the scaling factor in the Gaussian radial basis function (RBF) kernel. This value is used if RBF is used as mapping kernel. It can be selected by setting value of a slider with same name.

Polynomial order. This positive number indicates order of polynomial if a polynomial is used as mapping kernel. It can be selected from a drop down menu (popup menu).

Kernel function. This parameter specifies the mapping kernel function of the SVM. It can be selected from a drop down (popup) menu. Options include linear kernel, quadratic kernel, polynomial kernel, radial basis function and least squares.

Method. A string specifying the method used to find the separating hyperplane. Choices are:

'SMO' - Sequential Minimal Optimization (SMO) method (default).

'QP' - Quadratic programming (requires an Optimization Toolbox license). It the L2 soft-margin SVM classifier. Method *'QP'* doesn't scale well for TRAINING with large number of observations.

'LS' - Least-squares method. It implements the L2 soft-margin SVM classifier.

It can be selected from a drop down (popup) menu with the same name.

Stopping criterion. This parameter tells the SVM to stop training if error falls below this threshold. It can be set by setting the value of a slider with same name.

Parameters panel for SVM GUI is shown in Fig. 4.4.

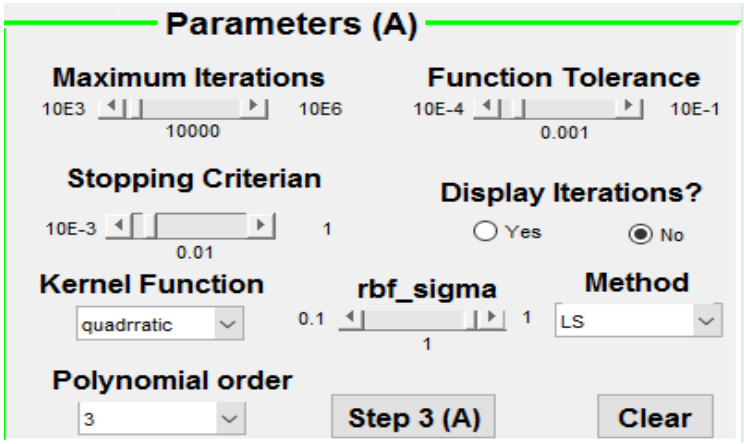

Figure 4.4 Parameters Panel for SVM classifier

4.4.2.3. ELM

There are only two parameters necessary for ELM. One is activation function and the second is number of nodes in hidden layer. These parameters are explained below.

Number of nodes. This parameter indicates the number of neurons in hidden layer. It can be set by typing the number of nodes in corresponding edit box. The higher the number, the more powerful ELM becomes. But if it is greater than the number of observations, there is a higher chance of overfitting.

Activation function. This parameter specifies activation function for nodes in hidden layer. It can be selected from a popup menu. The list includes *'sigmoid', 'sine', 'tribas', 'radbas'* and *'hardlim'*

The parameters can be set using panel shown in Fig. 4.5.

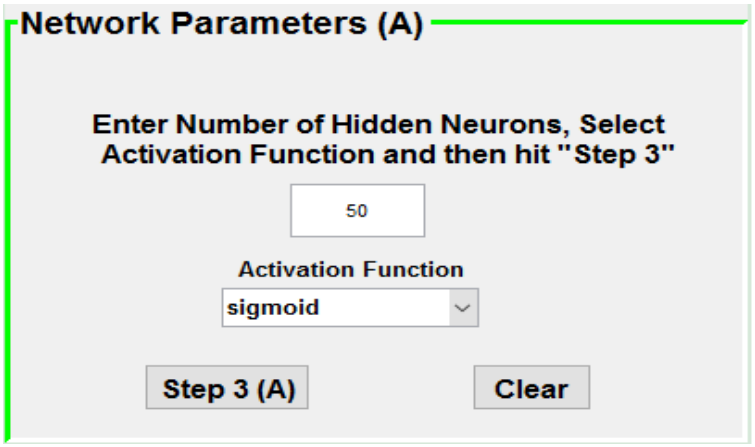

Figure 4.5 Parameter Panel for ELM

4.4.2.4. Deep belief networks (DBN)

For DBN, there are four parameters accessible to the user.

Number of hidden layers. This parameter specifies the number of hidden layers which should be trained as RBMs. It is a positive whole number and can be selected from popup menu of the same name.

Number of nodes in each layer. Once number of hidden layers is selected, same number of edit boxes will be created which can be used to enter number of nodes in each layer. As RBMs are used to construct the hidden layers, so number of nodes in subsequent layers should be less than number of nodes in current layer.

Number of epochs. This parameters specifies the number of training epochs. It can be set by typing the value in edit box of same name.

Batch size. This parameter specifies batch size of training matrix. It can be set by typing the value in edit box of same name.

Panel for these parameters is shown in Fig. 4.6.

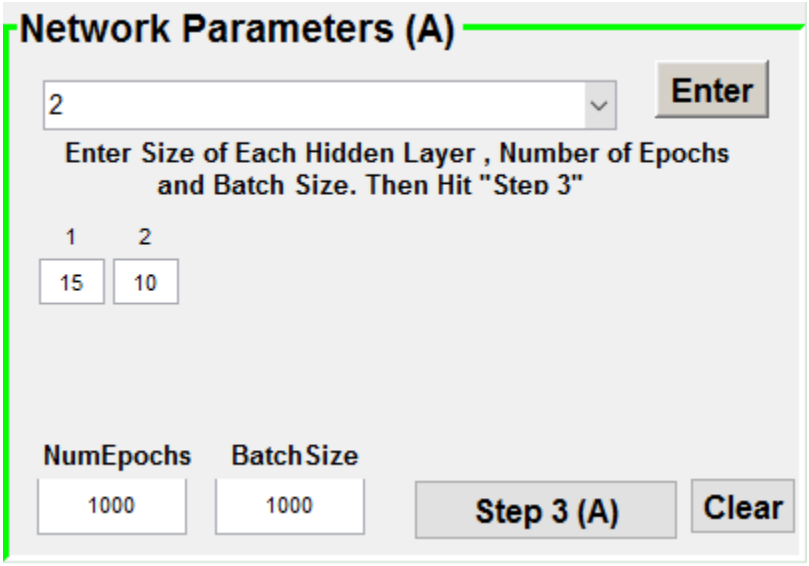

Figure 4.6 Parameters panel for DBN

4.4.2.5. HELM

For HELM, there are four parameters accessible to the user.

Number of hidden layers. This parameter specifies the number of hidden layers which should be trained as ELM auto encoders. It is a positive whole number and can be selected from popup menu of the same name.

Number of nodes in each layer. Once number of hidden layers is selected, same number of edit boxes will be created which can be used to enter number of nodes in each layer. It is to be noted that ELM sparse auto-encoder is used to construct hidden layers, so number of nodes in subsequent layers should be greater than the number of nodes in current layer. General practice is to select twice the number of nodes in next layer than the current layer.

Activation function. This parameter specifies activating function of each node (in each layer). Function specified by this parameter is used to project the weighted sum of inputs to another nonlinear dimension. It can be set by selecting appropriate activation function from a popup menu with the same name. The list includes *'sigmoid', 'sine', 'tribas', 'radbas'* and *'hardlim'*

The parameters can be set using panel shown in Fig. 4.7.

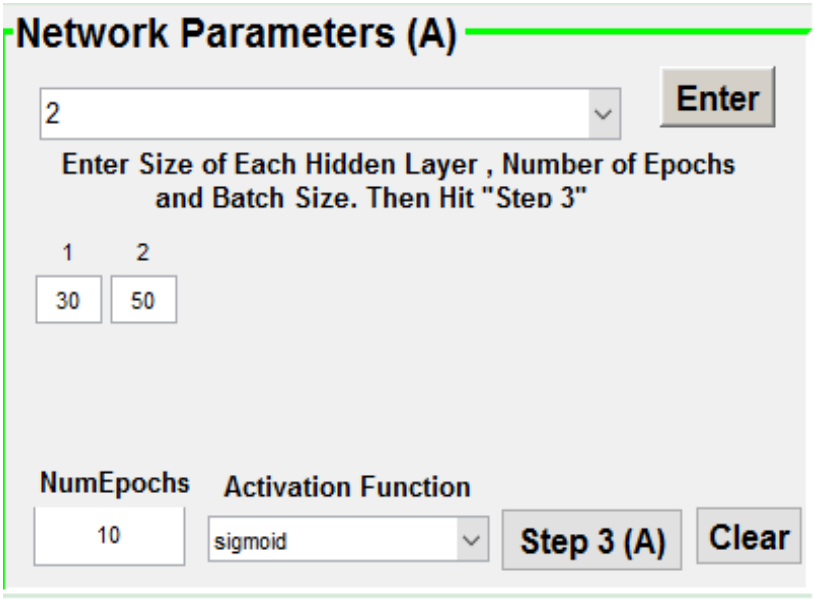

Figure 4.7 Parameters Panel for ELM auto-encoder based deep network

4.4.2.6. CNN

For CNN, there are six parameters accessible to the user. These parameters include number of layers, sub-sampling (pooling) scale, convolution kernel size, number of filters learned in each convolutional layer, number of epochs and batch size. The parameters are explained as following

Number of layers (convolution and pooling). This parameter specifies the number of convolution and pooling layers. It is a positive whole number and can be selected from popup menu of the same name. It is to be noted that exactly same number of pooling layers are created as convolution layers. So this parameter controls both of them. When selected, the system creates the same number of convolution and pooling layers and place them on top of each other.

Number of filters learned in each layer. This parameter controls the number of filters (features) leanred in each convolutional layer. The higher the number, the better the learning, Once number convolutional layers, same number of popup menus will be created which can be used to select number of convolutional filters to be learned.

Convolutional kernel size. This parameter controls the size of convolutional kernel.

Subsampling scale. This parameter controls scale of pooling. This scale is applied to all the pooling layers. The input image to pooling layer is down sampled by this scale either by max-pooling or mean pooling. In max pooling, maximum value of pixels being down sampled is selected while in mean-pooling, the mean value of pixels being down sampled is selected.

Number of epochs. This parameters specifies the number of training epochs. It can be set by typing the value in edit box of same name.

Batch size. This parameter specifies batch size of training matrix. It can be set by typing the value in edit box of same name.

Panel for these parameters is shown in Fig. 4.8.

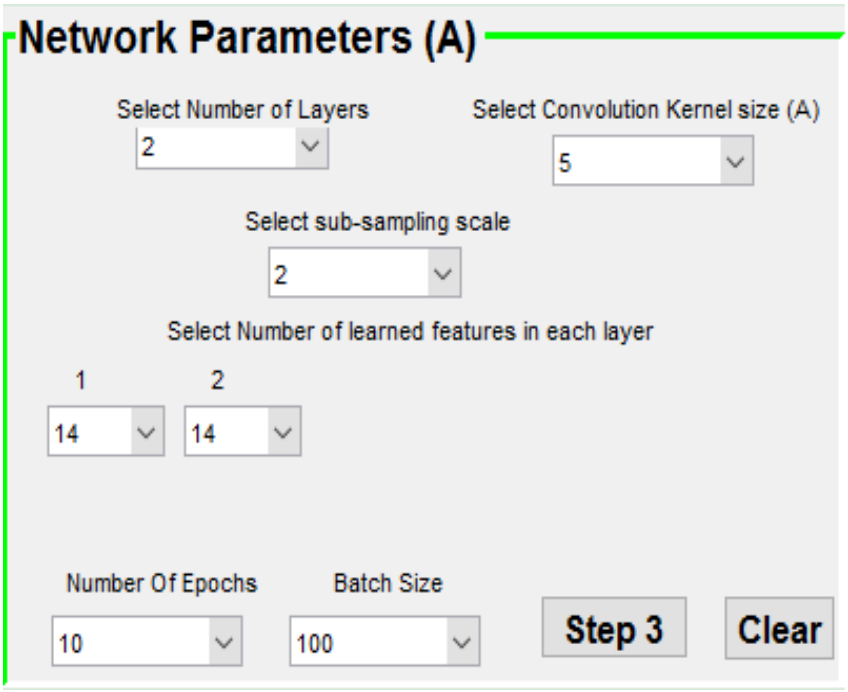

Figure 4.8 Parameters Panel for CNN

4.4.2.7. MLP

Parameters for MLP include Number of hidden layers, Number of nodes in each layer, Number of maximum iterations, learning rate, momentum and Error tolerance.

Number of hidden layers. This parameter specifies the number of hidden layers which should be trained as perceptrons. It is a positive whole number and can be selected from popup menu of the same name.

Number of nodes in each layer. Once number of hidden layers is selected, same number of edit boxes will be created which can be used to enter number of nodes in each layer

Maximum iterations. This parameter controls the maximum number of training runs allowed to MLP. It tells the system when to stop learning if error tolerance is not reached. It is advisable to use higher value for this parameter. In most of the cases, higher number of maximum iterations results in higher probability of convergence, but for some datasets, even higher number of iterations does not ensure convergence. It can be selected from a drop down menu of the same name.

Learning rate. This parameter controls the learning rate of MLP. Learning rate dictates how quickly the optimization algorithm will adapt to the training dataset and converge on solution in backpropagation. Higher learning rate means quicker adaptation but bad convergence and vice versa. It is selected by setting value of a slider bar with the same name.

Momentum. To help backpropagation algorithm converge on a global minimum instead of local minima, momentum is used. The momentum of MLP ties weights of current training steps to weights found in previous steps. It is selected by setting value of a slider bar with the same name.

Tolerance. This parameter controls the value of final training error. MLP is declared as trained when backpropagation error falls below this value. Along with maximum number of iterations, this is another performance measure of MLP training which helps us decide when to stop training. It is selected by setting value of a slider bar with the same name.

Parameter panel for MLP is shown in Fig. 4.9.

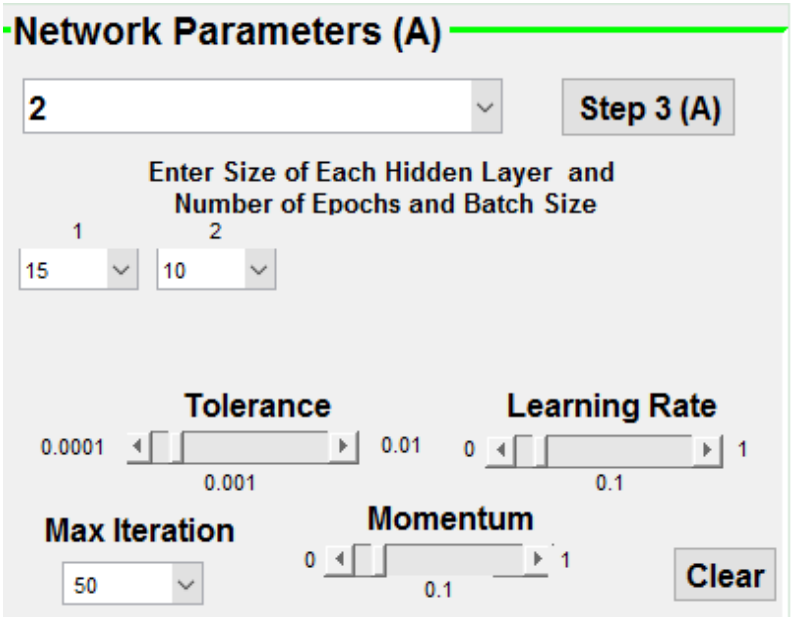

Figure 4.9 Parameters panel for MLP

The steps for proper parameter settings are listed as follows.

1) Select number of hidden layers from drop down menu
2) Let's say you have selected 4 hidden layers, you will see 4 popup menus created below. Each correspond to a hidden layer. Select number of neurons for each hidden layer
3) Select maximum iterations from corresponding drop down menu
4) Using corresponding sliders, specify the values of learning rate, momentum and tolerance
5) Once you have completed step a-d, click on *'step 3'*. Network parameters will be saved and next panel will be activated

4.4.3. Train

This panel is used to train and evaluate the classifier. All the core mathematical functions and algorithms discussed in previous sections run behind this panel. First, hit *'Train Network'* button. It will pick inputs and targets which were imported in *'Load Data'* panel. Then it will retrieve the parameters saved by *'Parameters'* panel. These inputs, targets and parameters are used to train the classifier.

Once the training is complete, an object with initials of classifier (for example, for MLP, the object name will be *mlp*) is created which stored trained classifier. This object is saved on memory. When we hit the *'Evaluate on Train data'* button, the training inputs and targets along with the trained MLP object are passed to testing function which returns the predicted labels and accuracy.

Same thing happens when hit *'Evaluate on Test Data'* buttons, with the only exception is that this time, unseen inputs (test inputs) are passed instead of seen inputs.

Actual targets and predicted targets for unseen data can be saved in three formats, comma separated values (.csv), excel (.xlsx) and Matlab files (.mat).

A typical training and evaluation panel is shown in Fig. 4.10.

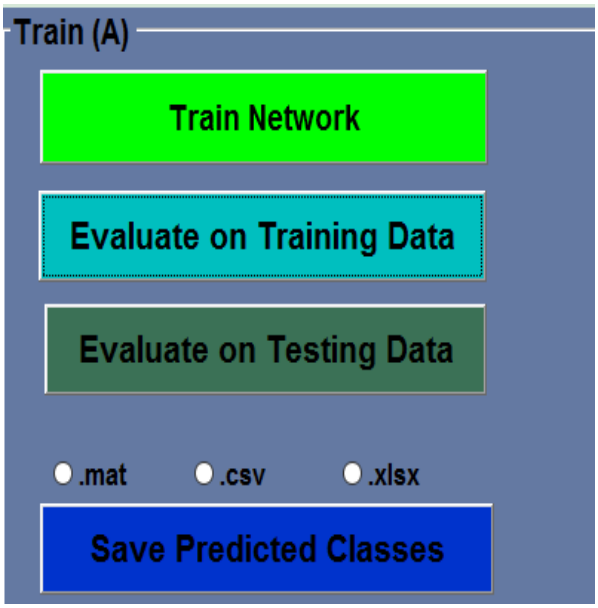

Figure 4.10 A typical training and evaluation panel, once the network is trained, predicted and actual labels can be saved to file as well

4.4.4. Results

In this panel, confusion matrix and ROC curve can be observed. If the network is being evaluated on training data, training accuracy will be shown along with confusion matrix for training targets. ROC curves are also plotted for training targets and predicted targets. Similarly, these performance visualizations are done for test data when the network is being tested on unseen inputs. Moreover, for deep networks, Mean Squared Error (MSE) for training can also be viewed. A typical results section is showed in Fig. 4.11.

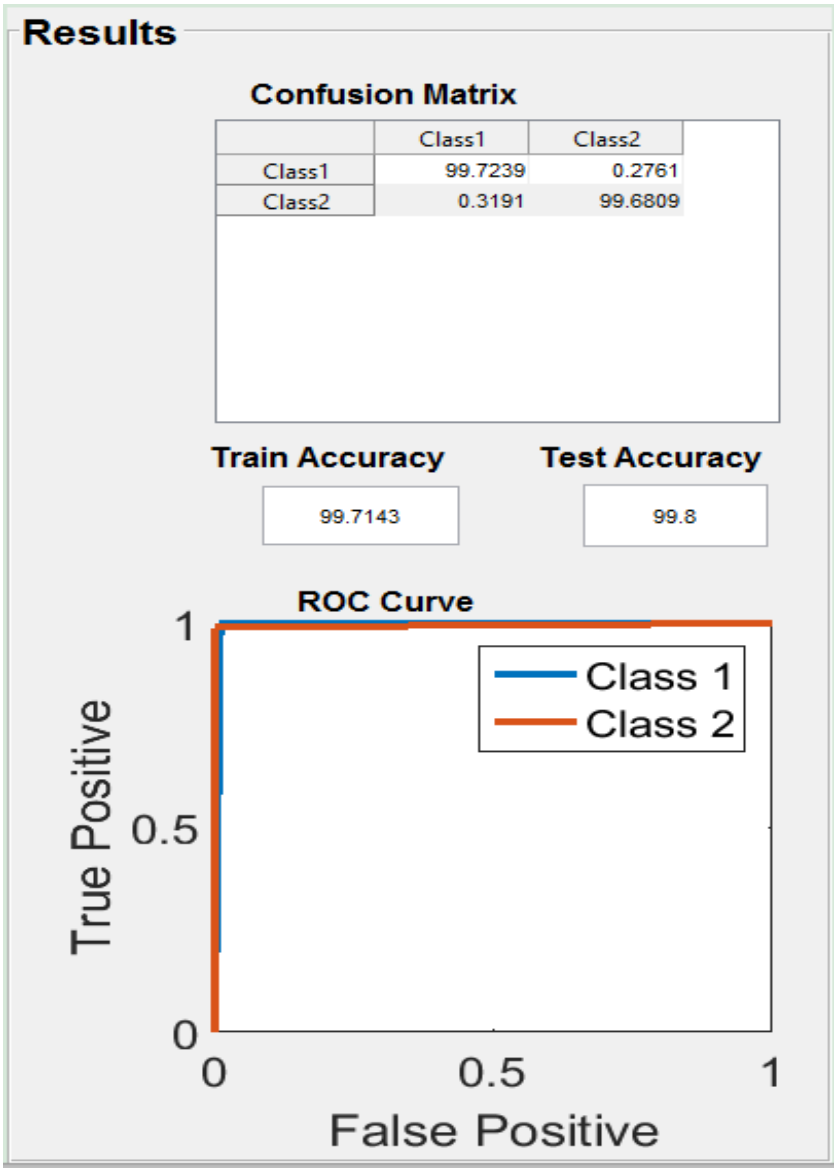

Figure 4.11 Results panel, confusion matrix shows number of correctly classified and number of wrongly classifier observations whole ROC curve indicates ratio of true positives to false positives graphically

4.5. Master GUI

One master GUI is also developed which can be used to train any classifier and compare its performance to other trained classifier. The master GUI follows same flowchart shown in Fig. 5.1. The data is loaded the same way as any individual GUI, however, the parameter selection is different for each

classifier. Once the classifier is selected, a child GUI is launched which gives access to parameters of only the selected classifier. The master GUI is shown in Fig. 4.12.

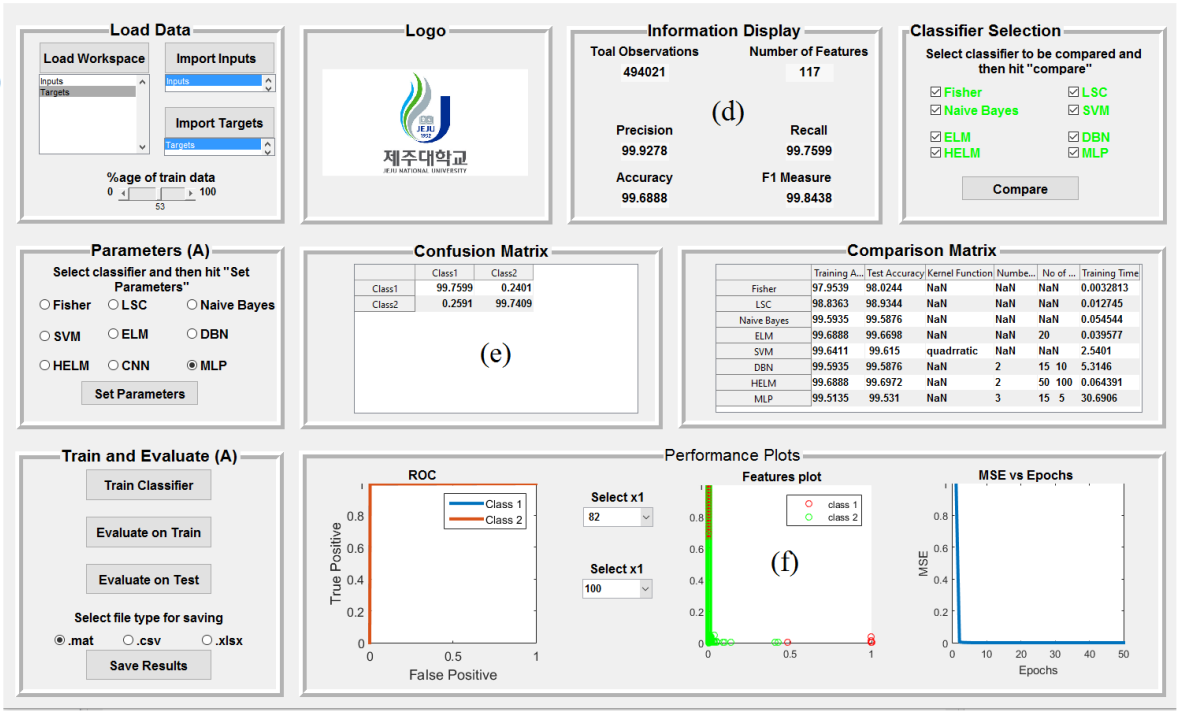

Figure 4.12 Outlook of master GUI, highlighted and numbered sections represent different panels

As shown in the Fig. 5.11, the master GUI has three main parts; part one is comprised of panels (a) to (c) (highlighted in green color), part two is comprised of panel (d), (e) and (f) and part three is comprised of panel (g) and (h). Out of these three main parts, part one is used to train and evaluate individual classifiers, part two is used to visualize performance of the classifier and part three is used to compare the performance of trained classifiers. The constituent panels are described in the following section.

4.5.1. Load Data

Panel (a) is used to load data from workspace. Inputs and Targets should be loaded in Matlab workspace first. *'Load workspace'* button will import all the variables that are loaded into workspace into GUI space and will appear in the list box below the button. Out of those variables, Inputs can be imported by selecting corresponding variable from the list box and then hitting *'Import Inputs'*. Similarly, targets can

be imported by selecting corresponding variable from the list box and then hitting *'Import Targets'*. Once the inputs and targets are imported, the slider can be used to set percentage of inputs which will be used for training the classifier.

4.5.2. Parameters

Panel (b) is used to select the classifier which we want to train and then set its parameters. Select the classifier which is to be trained and then hit *'Set Parameters'* button. It will launch a child GUI to set the parameters of corresponding classifier. GUIs for setting parameters of classifiers are similar to the parameter panels if their respective individual GUIs.

4.5.3. Train and Evaluate

Panel (c) works exactly the same as the corresponding panel shown in Fig. 4.2. The only difference is that this panel trains any classifier selected in panel two instead of any specific classifier. All the core mathematical functions and algorithms discussed in previous chapters run behind this panel. This panel is operated in exactly the same way as the panel shown in Fig. 4.9.

4.5.4. Results

The contents of panels (d), (e) and (f) are not accessible to users. They are only used for observing the results and performance of trained classifiers, so they are accessible to GUI callback functions only. In panel (d), information about the inputs (number of observations and number of features) is displayed. Once the classifier is trained, its performance measures (precision, recall, accuracy, F1 measure) are also displayed here. Result panels are shown in Fig. 4.13.

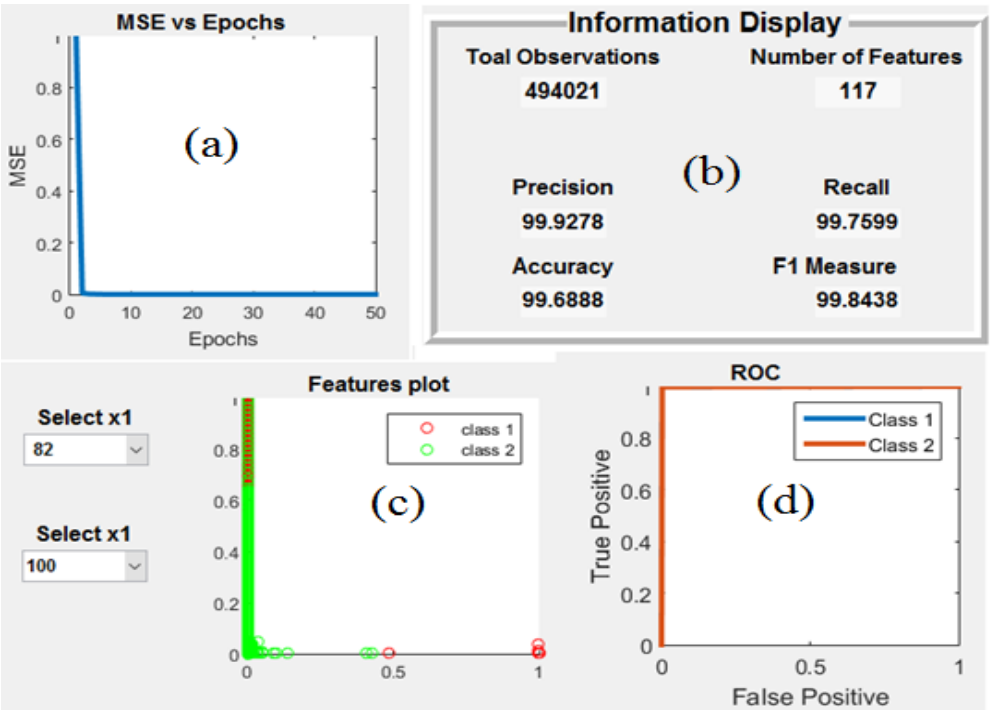

Figure 4.13 Results panels for master GUI. (a) MSE vs number of epochs, (b) Primary information display panel (c) Feature scatter plot panel, features can be visualized in 2D (d) ROC curve

4.5.5. Classifier Selection (For comparison)

Panel (g) is used to select the trained classifiers for comparison. There are 8 check boxes on this panel, one for each classifier. When the GUI is launched, it searches for the trained versions of these classifiers. If there are any trained versions of the classifiers present in the memory (saved from earlier sessions), the font color of corresponding classifier name will change to green. Font color of classifiers names not found in the memory will become red automatically.

Once the classifiers are detected, user can select the ones he/she wants to compare. It is to be noted that only those classifiers can be selected for comparison which are already trained. If a classifier is not trained, it should be trained first by accessing panels (a) ~ (c). Once the classifier is trained, font color of its corresponding check box in panel (g) will automatically turn green.

4.5.6. Comparison Results

Comparison results can be visualized in panel (h). Although comparison between the stated classifiers cannot be accurate because these classifiers work on different principles and are designed for different problem domains, we can still get an idea which classifier to choose given the input data. Performance comparison can be done in three measure; training accuracy, test accuracy and training time. Training and test accuracy indicates the classification performance while training time indicates speed of the classifier.

4.6. Summary

We have implemented a GUI version of the classifiers mentioned in chapters 2-4. Every classifier has their standalone GUI along with one master GUI. The standalone GUIs are used to train and evaluate individual classifiers. The master GUI is used to compare the classifiers as well. These GUIs are developed following the same flowchart shown in Fig. 5.1, regardless of the nature of the classifier. This is to ensure that all the GUIs are easy to understand and easy to use.

Chapter 5

RESULTS OF THE TOOLBOX

The classifiers discussed in previous chapters are tested on two types of data, one is binary class data and the second is multi-class data. These datasets were used to test the standalone GUIs as well as the master GUI. Two case are presented here for evaluation purpose.

5.1. Binary class classification

In this case, the classifiers are trained on computer network data from KDD cup. This data represents captured instances from network traffic. Input features represent different properties of a network connection. Number of features is 20. There are 4 continuous features and 16 discrete features. Continuous features are normalized, so their range is 0 to 1. There are two output classes, class 1 represents all the instances when incoming network connection request is a virus while class 2 represents normal (non-virus) connections. Scatter plot for two input features is shown in Fig. 5.1.

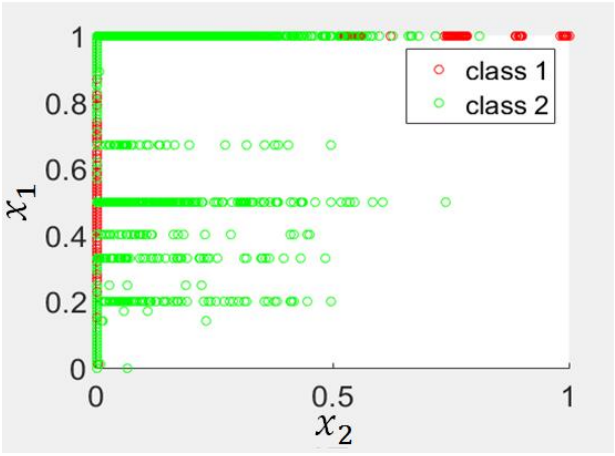

Figure 5.1 Scatter plot for two input features

All the listed classifiers achieve accuracy greater than 99% on this dataset. The ROC curve for all the classifier is also close to perfect. As all the classifiers produce similar ROC curves, only one ROC curve is presented here as shown in Fig. 5.2.

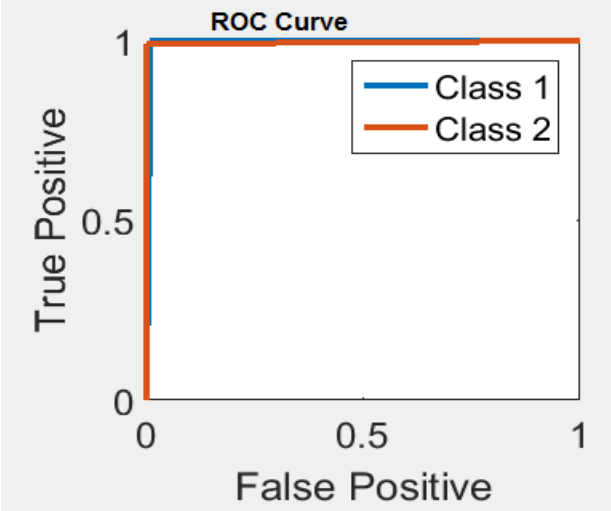

Figure 5.2 ROC curve for binary class classification results

Comparison results obtained after running main GUI on this data or shown in Fig. 5.3.

	Training Accuracy	Test Accuracy	Kernel Function	Number of layers	No of Nodes	Training Time
Fisher	97.6059	98.1834	NaN	NaN	NaN	0.023808
LSC	99.3825	99.6581	NaN	NaN	NaN	0.025305
Naive Bayes	99.6352	99.8282	NaN	NaN	NaN	0.067637
ELM	99.7714	99.7	NaN	NaN	20	0.091087
SVM	99.7286	99.8333	quadrratic	NaN	NaN	4.9455
DBN	99.6714	99.7	NaN	2	15 5	0.010026
HELM	99.7714	99.7333	NaN	2	30 50	0.069752
MLP	99.5438	99.4694	NaN	3	30 10	37.0884

Figure 5.3: Performance matrix for 2 class classification problem

As seen in Fig. 5.3, performance of all the classifiers is comparable to each other. This is because the data is well defined and has small number of features. Based on this case, we cannot draw concrete conclusion. Since every classifier achieves considerable accuracy on this dataset, so it is recommended to use linear classifiers for such datasets due to their simplicity and speed.

5.2. Multiclass classification

In this case, the classifiers are trained on hand written characters data. This data represents images of hand written characters. Each image is 28 pixels wide and 28 pixel high ($28 \times 28 = 784\ pixels$). Pixel data is normalized, so input range is 0 to 1. There are four output classes, digits 0 to 3. As the features represent individual pixels, therefore scatter plots of features against each other is not always meaningful since it cannot visualize the inherent relationships between individual pixels, but for the sake of visualization, scatter plot for two central pixels and a sample input image are shown in Fig. 5.4.

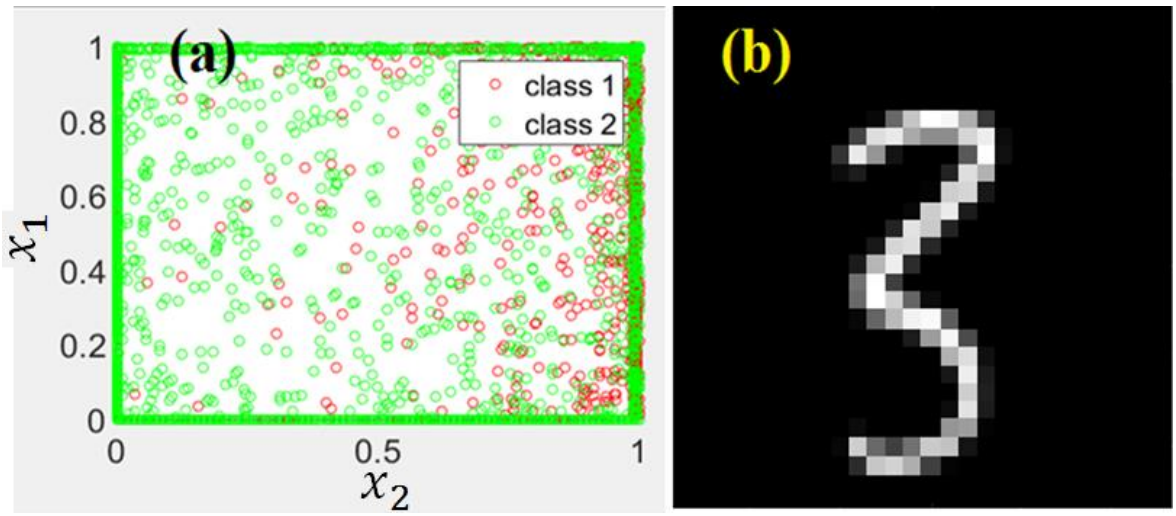

Figure 5.4 (a) Scatter plot for two central pixels (b): Sample input image

As evident from the scatter plot, individual features (pixels) are not quite descriptive, so linear and shallow classifiers are not expected to perform well. The performance table is shown in Fig. 5.5.

	Training Accuracy	Test Accuracy	Kernel Function	Number of layers	No of Nodes	Training Time
Fisher	NaN	NaN	NaN	NaN	NaN	0.95288
LSC	NaN	NaN	NaN	NaN	NaN	377.9813
Naive Bayes	NaN	NaN	NaN	NaN	NaN	0.91794
ELM	94.06163	94.12874	NaN	NaN	784	0.32557
SVM	NaN	NaN	quadrratic	NaN	NaN	4.9455
DBN	96.45083	96.63345	NaN	3	200 100 50	20.9644
HELM	98.24561	98.27633	NaN	3	800 1600 3...	13.2776
MLP	98.7688	97.71258	NaN	3	10 50 20	167.5095

Figure 5.5: Performance matrix for four class classification problem

As expected, performance of linear classifier is not at par with deep classifiers. Although ELM achieves considerable accuracy instead of its shallow architecture, its performance is still not as high as deep classifiers. This is evident form ROC curves as well which are well defined for deep networks but poorly defined for linear classifiers as shown in Fig. 5.6.

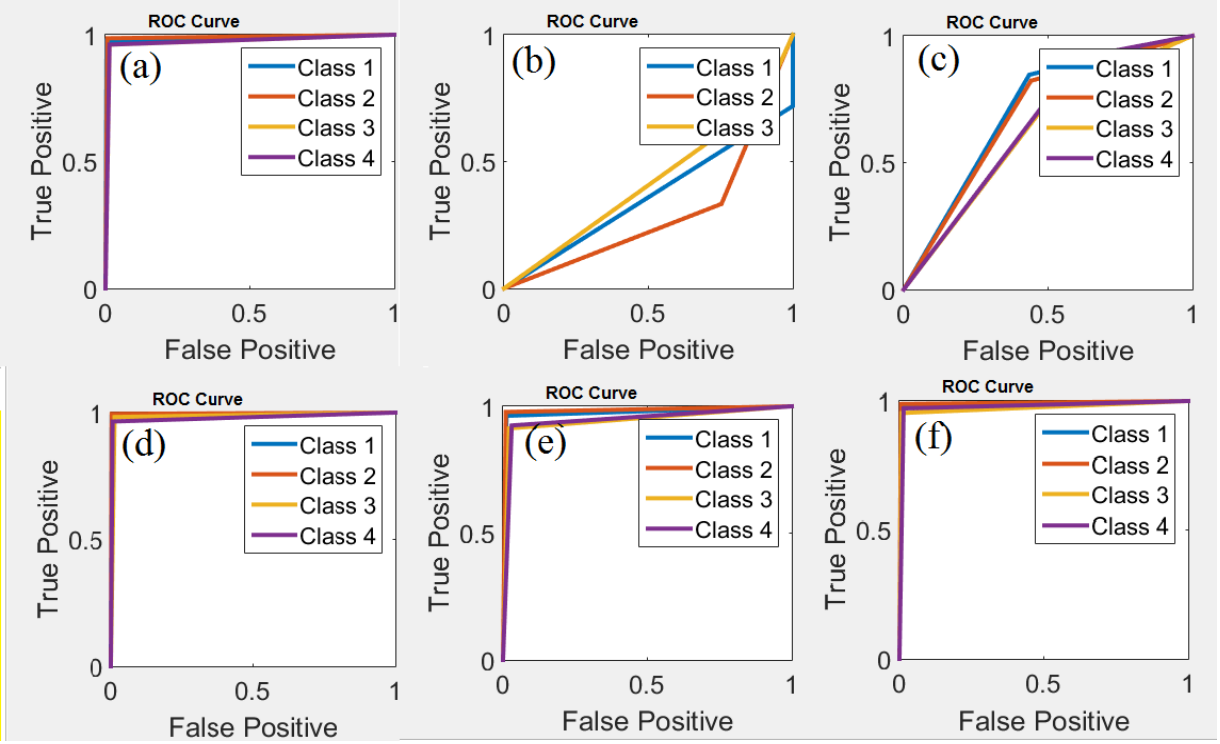

Figure 5. 6 ROC curves of classifiers for multiclass classification (a): ROC Curve for DBN classifier, (b): ROC Curve for Fisher classifier, (c): ROC Curve for Naïve Bayesian classifier, (d): ROC Curve for MLP classifier, (e): ROC Curve for ELM based deep net classifier, **(f):** ROC Curve for CNN classifier

As in case 2, the number of input features is 784 and it is also multi-class problems, deep networks show superior performance as they are specifically designed for this purpose. The only exception is MLP which is a fully connectionist model, so it's training time is far more than comparable deep networks. Moreover, as the number of classes increases, the power of deep networks become more evident.

Chapter 6

DISCUSSION AND CONCLUSION

6.1. Discussion

In this section, a general comparison of different machine learning toolboxes is given. Although there are many open source codes and toolboxes available on the internet, yet we have selected those toolboxes for comparison which are solely developed for machine learning. There are many data analysis toolboxes also available, but they are never solely intended for ML alone. The mentioned toolboxes are targeted for machine learning community. Moreover, all of the toolboxes being compared are developed in Matlab. We have excluded toolboxes which are not developed in Matlab from this comparison. Toolboxes which are compared with our toolbox are listed as follows.

- Matlab machine learning toolbox
- Toolbox developed by TeraSoft
- Toolbox developed by Jason Weston et al
- Toolbox developed by Carl Edward
- MeteoLab
- Open source code snippets

The comparison is done based on following criteria.

6.1.1. GUI Support

In this comparison criteria, we evaluate if the toolbox in question can be operated from a GUI with ease and convenience. Full GUI support means that they GUIs implemented are self-explanatory and mastering one GUI means the user can operate all GUIs with ease. Moreover, it also means that all the GUIs follow single flowchart, regardless of the nature of ML algorithm.

6.1.2. Coherent Implementation

In this criteria, we evaluate if the Matlab functions implemented are following one master template, regardless of the nature of classifier.

6.1.3. Modules Support

In this criterion, we evaluate that if the implemented classifier are modular or not. If a classifier is modular, than it can be used within other algorithms seamlessly as sub modules.

6.1.4. Diversity of algorithms

In this criterion, the toolboxes are compared based on the diversity of algorithms implemented.

6.1.5. Visualization

In this criterion, we compare the visualization power of the toolboxes.

Table 6.1 Comparison Table for different toolboxes against our toolbox

	GUI Support	Implementation	Modular Support	Number of classification algorithms	Diversity of algorithms	Visualization
Matlab ML Toolbox	High	No master template	Non modular	>10	Moderately diverse	Medium
Toolbox by TeraSoft	None	No master template	low	<6	Highly diverse	High
Toolbox by Jason et al	None	No master template	Full modular	<10	Moderately diverse	Medium
Toolbox by Carl Edward	None	No master template	Non modular	2	Limited diversity	High
MeteoLab	None	No master template	Slightly modular	Only forecasting	Moderately diverse	High
Open source code snippets	Medium	No master template	Non modular	>>10	Very diverse	High
Our Toolbox	Full GUI support	One master template	Full modular	9	Very diverse	Low

As given in table 6.1, most of the toolboxes available on the internet do not support extensive GUI layout. Matlab own toolbox does offer a modest GU interface, but the interface is not streamlined. Every classifier has its own layout which can become tricky if the user has no extensive knowledge of the classifier being deployed. Moreover, most of the toolboxes are primarily concerned with a specific band of classifiers. For instance, apart from toolbox developed by TeraSoft Inc, other mentioned toolboxes either target probabilistic classifiers, or linear classifiers or neural networks. Matlab own toolbox is getting more diverse with new versions. The latest version of Matlab, R2016a has now a deep learning module added to the toolbox too, but this module does run on previous versions of Matlab.

To ensure smooth running of the call back functions of the toolbox, we have benchmarked our toolbox against verified open source code snippets available. We have tested our toolbox on two datasets. One dataset is taken from open source MNIST repository. This dataset is comprised of images of hand written characters. The characters first ten numerals, i.e. 0 to 9. Each image is 28 pixels wide and 28 pixels high. Neurons based and deep classifiers are tested on this dataset. The second dataset is composed of observations from seismic data of a person either walking or running. Goal of this dataset is to determine the possibility of classification between a person walking or running.

We have picked similar open source codes and then used exactly same parameters for both the cases; our toolbox and open source codes and then compared the performance based on three parameters, training time, training accuracy and test accuracy. Open source codes were picked from established internet forums [43, 44, 45, 46, 47, 48, 49, 50. 51]. Parameters used for comparison are given in Table 6.2 while Table 6.3 summarizes the comparison results.

Table 6.2 Classifiers parameters for benchmarking

Least Squares Classifier	Error tolerance : 0.001
Fisher Classifier	None
Naïve Bayesian Classifier	None

Support Vector Machines	Kernel: Linear, Optimization method: *'SMO'*
Extreme Learning Machines (ELM)	Number of Neurons : 500, Activation Function: *'sigmoid'*
Deep Belief Nets	Number of layers : 2, Size matrix : [100, 100]
Hierarchical ELM	Number of layers : 3, Size matrix : [500, 1000, 2000], Activation Function: *'sigmoid'*
Convolutional Neural Network	Number of layers : 2, Filters learned in each layer: [12, 12], Convolutional Kernel size: 5, Sub-sampling scale: 2
Multilayer Perceptron	Number of layers : 2, Size matrix : [200, 100]

Table 6.3 Performance comparison of our codes to benchmark codes

	Training time (sec)		Training Accuracy (%)		Test Accuracy (%)	
	Ours	Benchmark	Ours	Benchmark	Ours	Benchmark
Least Squares Classifier	.04	.0076	72.26	70.20	71.65	71.8
Fisher Classifier	.0049	0.0038	70.43	71.85	70.23	69.17
Naïve Bayesian Classifier	.1682	.1103	57.7	58.3	57.3	57.3
Support Vector Machines	4.9	4.27	75	74.81	73.6	74.20
Extreme Learning Machines	4.85	9.7	91.06	91.14	90.6	90.6
Deep Belief Nets	56.2	35.4	95.7	94.55	95.95	94.17
Hierarchical ELM	9.2	5.56	98.14	99.00	97.5	97.7
Convolutional Neural Nets	769	765.4	95.5	95.49	95.3	95.28
Multilayer Perceptron	787.52	780.3	98.5	98.2	96.8	96.2

6.2. Conclusion and Future Work

A GUI based simulator is implemented for machine learning algorithms. We have targeted only multi-class classification part of machine learning which comes under the broad category of supervised learning. The GUIs are intended to be follow on general format so that a user who learns to run one GUI can run all of them without effort. They follow one flow chart for information flow through them. As we have implemented a diverse range of classifiers, their parameters and environments are completely different. However they are implemented in such a way that their corresponding functions looks similar to each other. Therefore, the callback functions of the GUIs also follow one format regardless of the nature of the classifier.

All the GUIs have a data import panel, a parameter setup panel, a training panel and a results panel. The data is imported from workspace and should be in a specific format. As every classifier works on different set of parameters, contents of the parameters panel might look different for different classifiers. For basic users who do not have much understandings of the parameters, they are set to their default values once the GUI is launched.

In the end, we have presented two case studies on which we have tested our GUI. In one case study, the input and target data is taken from KDD network connections data. In second case study, the input and target data is taken from MNIST hand written characters database. All the GUIs are tested on these dataset. Final GUI is used for classifiers comparison.

As mentioned earlier, the classifier discussed in previous chapters are designed for different kinds of problems, so a detailed comparison cannot be drawn between them. However, we can certainly draw some general conclusions about the effectiveness and computational cost of the classifiers. For instance, the linear classifiers are the simplest one, offering superior speed as compared to nonlinear and deep classifiers if the data is well defined and described by fewer dimensions. This is not true for real time data or data from natural signals (audio, video, images). As natural data is described by exceptionally large

number of features, and can have potentially many target classes, the performance of linear and shallow classifiers is not enough for practical purposes. Deep classifiers perform superbly in such cases.

This toolbox is developed with the sole purpose of familiarity and ease of use for experts and novice users alike. While doing so, the parameters settings of certain algorithms is kept as simple as possible so that beginners can also understand them well. This toolbox can be expanded so that more control is given to expert users who might want to access more parameters or control the classifiers at base level.

Only nine classifiers are implemented in this version of the toolbox. It can be expanded to include more classifiers. Also, regression, clustering, forecasting, prediction and optimization can be included in this toolbox. The toolbox can also be expanded for dimensionality reduction in a way that it offers multiple algorithm for dimensionality reduction. These algorithms can be implemented using the same flowchart such that they are easy to use as well.

In concluding remarks, the toolbox can be expanded into a full-fledged machine learning package for researchers and students who want to use Matlab as the development and research environment for machine learning.

APPENDIX A. TYPOGRAPHIC CONVENTIONS AND DATA FORMAT

This section deals with typographic conventions while naming variables and functions and data formats used throughout the toolbox.

A.1. Typographic Conventions

- Functions (script) names starting with Capital letters correspond to *'GUI functions'*
- Functions (script) names starting with small letters correspond to *'core mathematical functions'* used to develop the network
- All variable names start with capital letters
- All structure names start with small letters.
- Variables are named in such a way that they are self-explanatory. If a variable is to be named such that its name has multiple meaningful sections, all the sections are included in the name. For example we want to name a variable for training accuracy, the name should be *'TrainingAccuray'*
- Although structures names start with small letters, if a field of structure corresponds to a variable, it should start with capital letter. But if a field of structure corresponds to another structure, name of the field should start with small letter. For example, we have a structure named *'handles'* which has two fields, one corresponds to '*accuracy*' and second corresponds to a structure named *'dbn'*, it should be named as
    - handles.Accuracy = Training Accuracy
    - handles.dbn = dbn

A.2. Data Formats

All the classifiers mentioned in this document are examples of supervised learning. Therefore, like every supervised learning case, the classifier needs two arguments for training, input features matrix and output class matrix. The format of these matrices is explained below.

Let's say we have total N =1000 observations. These observations are described by p = 10 features and each observation corresponds to one of K classes. Input feature matrix dimension is $N \times p$ $(1000 \times 10\ in\ this\ case)$ while target class dimension is either $N \times K$ $(1000 \times 5\ in\ this\ case)$ or $N \times 1$ $(1000 \times 1\ in\ this\ case)$ depending upon the coding scheme. Observations corresponding to different classes are placed randomly in input matrix as follows.

$$entry\ 1: f_1 \quad f_2 \quad f_3 \dots \dots \dots f_{10} - - - - - - - - - class\ 4$$

$$entry\ 2: f_1 \quad f_2 \quad f_3 \dots \dots \dots f_{10} - - - - - - - - - class\ 1$$

$$entry\ 3: f_1 \quad f_2 \quad f_3 \dots \dots \dots f_{10} - - - - - - - - - class\ 2$$

$$entry\ 4: f_1 \quad f_2 \quad f_3 \dots \dots \dots f_{10} - - - - - - - - - class\ 3$$

$$entry\ 5: f_1 \quad f_2 \quad f_3 \dots \dots \dots f_{10} - - - - - - - - - class\ 5$$

$$entry\ 6: f_1 \quad f_2 \quad f_3 \dots \dots \dots f_{10} - - - - - - - - - class\ 1$$

.

.

.

$$entry\ 1000: f_1 \quad f_2 \quad f_3 \dots \dots \dots f_{10} - - - - - - - - - class\ i$$

For target data, there should be K (2 in this case) columns, for binary coding schemes as

$class\ 4$	0	0	0	1	0
$class\ 1$	1	0	0	0	0
$class\ 2$	0	1	0	0	0
$class\ 3$	0	0	1	0	0
$class\ 5$	0	0	0	0	1
$class\ 1$	1	0	0	0	0
$class\ 4$	0	0	0	1	0

Target data can be labelled directory as class names as well. For example, class one to five in this represent student 1 to 5 respectively, the target data can be written as follows.

$Student4$	4
$Student1$	1
$Student2$	2
$Student3$	3
$Student5$	5
$Student1$	1

$Student4$ 4

APPENDIX B. MATLAB FUNCTIONS USED

B.1. Fisher Classifier

B.1.1. fisher_training

This function is used to train a fisher object given training data. It projects the multi-dimensional data onto one dimensional plane. This is done by finding optimal weights for linear combination of inputs features. It then uses the training labels to estimate optimal separation threshold for the classes using fisher criterion. The fisher criterion is found by minimizing cost function expressed in Eq. (1). This multi-class fisher classifier is based on binary fisher classifier developed by Quan Wang [41]. Multi-class training is done in *one-against-all* manner.

Syntax

```
fisher=fisher_training(Train_X,Train_Y)
```

Input Arguments:

Train_X: Training inputs (Dimensions: Number of entries × Number of features)

Train_Y: Training targets (Dimensions: Number of entries × Number of classes if binary class coding is used, Dimensions: Number of entries × 1 if actual class label is used instead of binary class coding). For example, there are two output classes, class 1 and class 2. In binary coding scheme, there are two columns of Train_Y, once corresponds to class 1 and the second corresponds to class 2. All entries of column 1 where there is class 1 equals to 1 while rest are 0. Same goes for column 2 and class 2.

Output Arguments:

fisher: Trained fisher object. A typical fisher object contains following fields

Weights. Linear weights of each feature for linear combination while calculating one dimensional plane

Thresh. Optimal class separation threshold

A typical fisher object for training data with 20 features looks like

fisher = W: [20x1 double]

Thresh: -0.0044

B.1.2. fisher_testing

This function is used to test fisher classifier on data. It takes features weights of the fisher object, uses them to find optimal one-dimensional plane. Once the plane is calculated, the optimal threshold is used to separate the data into two classes.

Syntax

```
[Predicted, Precision, Recall, Accuracy, F1, Actual] =
fisher_testing(fisher,X,Actual,display)
```

Input Arguments:

X: Test Inputs (Dimensions: Number of entries × Number of features)

Actual: Test Targets (Dimensions: Number of rows equals to number of rows in X while number of columns equals to number of columns in Train_Y)

fisher: Trained fisher object (with fields w and threshold)

display: Optional parameter to display classification results in command window. Its value should be 1 for displaying and 0 for not displaying. This parameter does not change training or testing conditions, it only helps us see the results in command window.

Output Arguments:

Predicted: Predicted labels for Test Inputs

Precision: Fraction of class labels retrieved that are relevant. Range [0 100]

Recall: Fraction of relevant class labels that are retrieved. Range [0 100]

Accuracy: Fraction of correctly classified labels. Range [0 100]

F1: harmonic mean of Precision and Recall. Range [0 100]

Actual: Actual labels of Test Inputs

B.2. Least Squares Classifier

B.2.1. lsc_train

This function is used to train a least square classifier on training data. It performs least squares regression between the independent variables, Train_X and dependent variable Train_Y. After least squares regression, an arg max function is used to determine output class. The classifier is based on method developed by Geladi P. et al [14].

Syntax

```
[lsc] = lsc_train(Train_X,Train_Y,tol)
```

Input Arguments:

Train_X: As explained in section B.1.1.

Train_Y: As explained in section B.1.1.

tol: This parameter denotes the least square error tolerance which we are ready to allow for Eq. (2.6). As regression is done in the least square sense, this parameter dictates the value of least square error which we are ready to tolerate. We can specify any positive value for this parameter. Higher value generally means quicker convergence but bad classifier and vice versa.

Range [0 Inf]

Output Arguments:

lsc: Trained fisher object. A typical fisher object contains following fields

T score matrix of Inputs

P loading matrix of Inputs

U score matrix of Targets

Q loading matrix of Targets

B matrix of regression coefficients

W weight matrix of Inputs

B.2.2. lsc_test

This function is used to test least squares classifier on data. It takes an *'lsc'* object, inputs and targets as input arguments and apply Eq. (2.5) and (2.6) on them. First it projects the inputs matrix onto another dimension by using Input loading matrix P and input weight matrix W, then projects it on target space by multiplying it with loading matrix of targets.

Syntax

```
[Predicted, Precision, Recall, Accuracy] = lsc_test(lsc,X,Y)
```

Input Arguments:

X: Test Inputs (Dimensions: Number of entries × Number of features)

Y: Test Targets (Dimensions: Number of rows equals to number of rows in X while number of columns equals to number of columns in Train_Y)

lsc: Trained lsc object (with fields W,P and Q)

display: Optional parameter to display classification results in command window. Its value should be 1 for displaying and 0 for not displaying. This parameter does not change training or testing conditions, it only helps us see the results in command window.

Output Arguments:

Predicted: Predicted labels for Test Inputs

Precision: Fraction of class labels retrieved that are relevant. Range [0 100]

Recall: Fraction of relevant class labels that are retrieved. Range [0 100]

Accuracy: Fraction of correctly classified labels. Range [0 100]

B.3. Principal Components Analysis (PCA)

B.3.1. prin_comp_analysis

The function runs Principal Component Analysis on a set of data points. It works on Eigen value decomposition. Eigen analysis is performed on input matrix first and largest Eigen values along with their corresponding Eigen vectors are found. These Eigen values and Eigen vectors are then used to project the input matrix along its principal components. If user has specified smaller dimension as input parameter instead of original dimension, project matrix is returned with reduced dimensions along with projection map. This function is based on method proposed by Laurens van der Maaten, Maastricht University, 2007

Syntax

```
[MappedX, XMap] = prin_comp_analysis(X, NDims)
```

Input arguments

X: Features matrix on which we want to run PCA for dimensionality reduction

NDims: Target dimensions for the reduced dimensionality matrix. Let's say original dimensions of the input feature matrix X are $N \times M$. For dimensionality reduction, $NDims < M$

Output arguments

MappedX: Output feature vector with reduced dimensions

XMap: A structure which contains information about mapping of original feature matrix onto mapped features matrix. It has following fields

Mean. Means of retained principal components

M. Mapping matrix of principal components. It can be used to reduce dimensions of input feature vector. Let's input feature vector dimensions are $N \times M_1$ and our target dimension is $N \times M_2$ such that$M_2 < M_1$. Dimension of M in this case is $M_1 \times M_2$. We can reduce dimensions of input matrix A as follows

$$B = A \times M \tag{2.4}$$

Lambda. Eigen values of retained principal components

B.4. Support Vector Machines (SVM)

B.4.1. multisvmtrain

Used for training an SVM structure for multiclass classification. Training is done in one-against-all manner. For every class, we train an SVM models which treats data as it is comprised of two groups, one group is the class for which SVM is trained and the other group is the 'non class'

Syntax

```
[models] = multisvmtrain(Train_X,Train_Y,options)
```

Input Arguments:

Train_X: As explained in section B.1.1.

Train_Y: As explained in section B.1.1.

options: This structure contains parameter values for SVM training. Some of these values are set by calling optimset command. These values are

'MaxIter' Maximum number of iterations for SVM training

'Display' Status of training we want to see in the workspace. In the GUI, this is set for iterations which means after every 500 iterations, status of training is displayed in command window.

'TolFun' Function tolerance value for training

Following 'options fields' must be set by the user

Stopping criteria for function

'rbf_sigma' A positive number specifying the scaling factor in the Gaussian radial basis function kernel. Default is 1.

'Polynomial Order' Order of polynomial if kernel function is polynomial

Method. A string specifying the method used to find the separating hyperplane. Choices are:

'SMO' - Sequential Minimal Optimization (SMO) method (default). 'QP' - Quadratic programming (requires an Optimization Toolbox license). It the L2 soft-margin SVM classifier. Method 'QP' doesn't scale well for TRAINING with large number of observations.

'LS' - Least-squares method. It implements the L2 soft-margin SVM classifier.

An example is given below

```
options=optimset('MaxIter',MaxIter,'Display','iter','TolFun',TolFun);
options.StopCrit=0.01;
options.RBF_Sigma= 0.9;
options.PolynomialOrder=6;
options.Method='SMO';
options.KernelFunction='polynomial';
```

Output Arguments:

models: Trained SVM

B.4.2. multisvmtest

Syntax

```
[Predicted,Actual,Accuracy]=multisvmtest(X,Y,models)
```

Input Arguments

X: Inputs (Dimensions: Number of entries × Number of features)

Y: Targets (Dimensions: Number of entries × Number of classes)

models: Trained SVM

Output Arguments:

Predicted: Labels predicted by ELM

Actual: Actual labels

Accuracy: Prediction Accuracy

Help:

Type ‘help multisvmtrain’ to access help file of this function

B.5. Naïve Bayesian Classifier

B.5.1. naïve_bayes_train

This function is used to train a naïve Bayesian classifier on training data. It trains a probabilistic classifier. This classifier is based on binary naïve Bayesian classifier developed for Gaussian distribution of input data. Multi-class implementation is done in one-againt-all manner. Let’s there are N classes in the training data, the naïve Bayesian classifier object is trained for identify all the classes, one by one. For instance, the classifier is being trained for class ‘a’, all the input values corresponding to class ‘a’ will be

treated as class 1 and all the 'non-a' values will be treated as class 0. This way, we have N Likelihood-matrices and N-Evidence matrices in the final trained naïve Bayesian object.

Syntax

```
[naivebayes] = naive_bayes_train(Train_X, Train_Y)
```

Input Arguments:

Train_X: As explained in section B.1.1.

Train_Y: As explained in section B.1.1.

Output Arguments:

naivebayes: Trained Naïve Bayesian classifier object. A typical naïve Bayesian classifier object contains following fields

Likelihood_Matrix	A likelihood matrix for each input feature against each class
Priors	Prior probabilities of target classes
Evidence	Relative importance of each input feature in from classification point of view

B.5.2. naïve_bayes_test

This function is used to test naïve Bayesian squares classifier on data. It takes naïve Bayesian object, inputs and targets as input arguments and apply Eq. (6, 7, 8) on them. First it finds posterior probabilities of classes using feature evidence and likelihood matrix in association with prior probabilities. Then it uses argmax function to decide output class.

Syntax

```
[Predicted, Precision, Recall, Accuracy] =
naive_bayes_test(naivebayes,X,Y)
```

Input Arguments:

X: Test Inputs (Dimensions: Number of entries × Number of features)

Y: Test Targets (Dimensions: Number of rows equals to number of rows in X while number of columns equals to number of columns in Train_Y)

naivebayes: Trained naïve Bayesian classifier

Output Arguments:

Predicted: Predicted labels for Test Inputs

Precision: Fraction of class labels retrieved that are relevant. Range [0 100]

Recall: Fraction of relevant class labels that are retrieved. Range [0 100]

Accuracy: Fraction of correctly classified labels. Range [0 100]

B.5.3. naïve_bayes_classify

This function is used to apply a naïve Bayesian classifier for two class target data

Syntax

```
[Predicted_Classes, Posteriors] = naive_bayes_classify(naivebayes,X)
```

Input Arguments:

X: Test Inputs (Dimensions: Number of entries × Number of features)

naivebayes: Trained naïve Bayesian classifier

Output Arguments:

Predicted_Classes: Predicted labels for inputs X (there will be only two predicted labels)

Posteriors: Posterior probabilities for target classes

B.6. Extreme Learning Machines (ELM)

B.6.1. elm_train

Used for training the ELM.

Syntax

```
[ELM,Actual,Expected,TrainingAccuracy] = elm_train(Train_x,Train_y,
NumberofHiddenNeurons, ActivationFunction)
```

Input Arguments:

Train_X: Training inputs (Dimensions: Number of entries × Number of features)

Train_Y: Training targets (Dimensions: Number of entries × Number of classes)

NumberofHiddenNeurons: Number of neurons in hidden layer

ActivationFunction: Activation function of neurons. User can select from *'sigmoid'*, *'sine'*, *'hardlim'*, *'tribas'* and 'radbas'

Output Arguments:

ELM: Trained ELM (weights)

Actual: Training labels predicted by ELM

Expected: Training labels

TrainingAccuracy: Prediction Accuracy on training data

B.6.2. elm_predict.m

Syntax

```
[Actual,Expected, TestingAccuracy] = elm_predict(ELM,Test_X,Test_Y)
```

Input Arguments

Test_X: Testing inputs (Dimensions: Number of entries × Number of features)

Test_Y: Testing targets (Dimensions: Number of entries × Number of classes)

ELM: Trained ELM (weights)

Output Arguments:

Actual: Testing labels predicted by ELM

Expected: Testing labels

TestingAccuracy: Prediction Accuracy on testing data

B.7. Multi-Layer Perceptrons

B.7.1. train_mlp.m

This function is used to train the MLP network.

Syntax

```
[mlp MSE] = train_mlp(Train_X, Train_Y, Hidden, MaxIterations,
LearningRate, Momentum,Tolerance)
```

Input arguments:

Train_X: As explained in Fisher.

Train_Y: As explained in Fisher.

Hidden: Number of layers and number of neurons in each hidden layer. If the memory of the computational system allows, the more hidden layers the better. Hidden actually is a vector in which number of elements represents number of layers while value of each element represents number of hidden nodes (perceptrons)

in that layer. For instance, if we want to train an MLP with 4 hidden layers in which number of nodes are 100, 50, 20 and 10 respectively, then

Hidden = [100 50 20 10]

MaxIterations: Maximum number of training runs. This number basically tells the system when to stop learning if error tolerance is not reached. It is advisable to use higher value for this parameter. In most of the cases, higher number of maximum iterations results in higher probability of convergence, but for some datasets, even higher number of iterations does not ensure convergence.

LearningRate: Learning rate of MLP. This is the converging rate of the optimization algorithm in backpropagation. Higher learning rate means quicker adaptation but bad convergence and vice versa

Momentum: Learning momentum of MLP

Tolerance: Error tolerance of trained MLP. It is the stopping criteria for learning. The learning stops once training error falls below this value

Output arguments:

mlp: Trained network (weights of each layer).

MSE: Training error (for each training run)

B.7.2. test_mlp.m

This function is used for evaluating the trained MLP on test data. Weights obtained in training phase are used in a feed forward pass to arrive at the output which is referred to as *'predicted'*. Predicted output is compared to actual target labels to evaluate the performance of MLP.

Syntax

```
[Output CC] = test_mlp(mlp, Inputs, Targets)
```

Input arguments:

Inputs: Inputs (Dimensions: Number of entries × Number of features)

Targets: Targets (Dimensions: Number of entries × Number of classes)

mlp: Trained MLP network (weights of each layer)

Output arguments:

Output: Outputs of network

CC: Correlation between network outputs and original targets

B.7.3. update_mlp.m

This function is used for updating MLP connection weights. It is called inside *'train_mlp'* function. It updates connection weights by using error backpropagation. First inputs to the function are passed through the network layers to arrive at the output in a feed forward pass. The feedforward outputs are compared to actual target labels to compute error. This error is propagated backwards to compute gradients and adjust weights. This function is not directly accessible to end user. It is only accessible in *'train_mlp'* function.

Syntax

```
[mlp] = update_mlp(mlp, Inputs, Targets)
```

Input arguments:

Inputs: Inputs (Dimensions: Number of entries × Number of features)

Targets: Targets (Dimensions: Number of entries × Number of classes)

mlp: Initialized MLP after feed forward pass (weights of each layer)

Output arguments:

mlp MLP structure with updated weights

B.8. Convolutional Neural Network (CNN)

These functions are based on algorithms and code developed by [39].

B.8.1. cnnsetup.m

This function is used for initializing the structure of a CNN. It involved specifying number of convolutional layers, number of sub-sampling layers, convolution kernel size, sub-sampling scale and the architecture of final supervised mapping layers. This function accepts training input image set, target labels set and a crude structure containing information about the layers. It returns a CNN structure with specified architecture.

Syntax

```
cnn = cnnsetup(cnn, Train_X, Train_Y)
```

Input Arguments:

Train_x: Training inputs (Dimensions: Number of Images × Image Height × Image Width)

Train_Y: As explained in Fisher.

cnn: A structure which contains information about number of Convolution layers and number of sub-sampling layers. Fields of structure contain information about size of convolution kernel, sub sampling (pooling) scale and number of filters to be learned. For example, to initialize a *'cnn'* structure for one input layer, two convolution layers and two pooling layers, following syntax should be followed

```
cnn.layers = {
   struct('type', 'i') %input layer
   struct('type', 'c', 'outputmaps', 20, 'kernelsize', 5) %convolution layer
```

```
    struct('type', 's', 'scale', 2) %sub sampling layer
    struct('type', 'c', 'outputmaps', 20, 'kernelsize', 5) %convolution layer
    struct('type', 's', 'scale', 2) %subsampling layer
};
```

Output Arguments:

cnn: A CNN network initialized with specified parameters

B.8.2. cnntrain.m

This function is used to train an initialized CNN structure. It accepts training image set and training labels set along with initialized CNN structure. First a training image is passed through a convolutional layer followed by sub-sampling layer. This duo of convolutional layer and sub-sampling layer achieves two things; one thing is that it learns pattern in the image and the second is that it resized the image to smaller dimension, therefore reducing its dimensionality. This process is repeated for specified number of convolutional and sub-sampling layers. Once we arrive at supervised layer, error back propagation is used to learn weights of supervised layers.

Syntax

```
cnn = cnntrain(cnn, Train_X, Train_Y, opts)
```

Input Arguments

Train_X: Testing inputs (Dimensions: Number of Images × Image Height × Image Width)

Train_Y: Testing targets (Dimensions: Number of Images × Number of classes)

cnn: Initialized CNN (returned by cnnsetup)

opts: option structure which contains following fields

Learning Parameter (alpha). This parameter controls speed of gradient adaptation. There is a tradeoff between convergence accuracy and convergence speed. When learning parameter is small, convergence is almost always achieved by finding global minimum but it takes too much time. On the other hand, large alpha means quick convergence with the risk of being stuck in local minimum.

Batch size. Instead of training CNN over whole image set, it is trained over batches of the image set.

Number of Training epochs

Output Arguments:

cnn: Trained CNN

B.8.3. cnntest.m

This function is used to test the performance of CNN on an input-label pair. First the inputs are passed through all the layers of CNN (convolutional, sub-sampling and supervised mapping), to arrive at the output node. Values of the output node is compared to actual labels of input images.

Syntax

```
[Er, Bad,Predicted,Actual] = cnntest(cnn, X,Y)
```

Input Arguments

X: Inputs (Dimensions: Number of Images × Image Height × Image Width)

Y: Targets (Dimensions: Number of Images × Number of classes)

cnn: Trained CNN

Output Arguments:

Actual: Actual Labels

Expected: Labels predicted by network

Er: Error

Bad: Indices of misclassified images

B.8.4. Cnnff.m

This function implements the feedforward pass of the CNN. The images are fed into the input layer, passed through convolutional layers and subsequent pooling layers to arrive at the output which is compared to actual labels for error back propagation. Architecture of Figure 3.3. is implemented in this function. It is an internal function, so it is not accessible to the end user. This function is called inside *'cnntrain'*

Sytntax

```
[cnn,Out] = cnnff(cnn, Batch_X)
```

Input arguments

Batch_X: A batch of images

cnn: CNN Structure

Output

cnn: CNN with weights adjusted in feed forward pass

Out: Output of a feedforward pass

B.8.5. Cnnbp.m

This function implements the back propagation pass for fine tuning the weights. First the error is calculated by subtracting output of feedforward pass from actual labels. Then these errors are back propagated to find the gradients of cost function. The gradients are then used to adjust the connection weights. This function represents the implementation of eq. 3.4 (a), (b) and 3.6 (a), (b). This is also an

internal function. It is called after *'cnnff'* (which implements the feedforward pass) inside the *'cnntrain'*. It helps fine tune the supervised layer weights by error back propagation.

Sytntax

```
cnn = cnnbp(cnn, Batch_Y,Out)
```

Input arguments

Batch_Y: A batch of Image labels which will be used to fine tune learned weights

cnn: CNN Structure

Out: Output of a feedforward pass which will be used to calculate errors by comparing it to Batch_Y

Output

cnn: CNN with weights adjusted by back propagation of errors

Data format

Let's say we have total N =1000 images and 10 classes. Let's say image dimensions are

$$Image\ Dimensions = H \times W \quad where\ H\ is\ image\ height\ and\ W\ is\ image\ width$$

Images should be stored in 3D array with dimensions

$$\boldsymbol{N} \times H \times W$$

Target data dimensions are

$$N \times 10$$

B.9. ELM based deep net

B.9.1. helm_train

This function is used for training the ELM based deep network. It takes cares of all the steps involved in the training i.e. input weights and biases generation, initialization of ELM auto-encoders and training. Once ELM auto-encoders are trained, it also trains the final layer in a supervised manner.

Syntax

```
[stack,TrainingAccuracy,Predicted,Actual] =
helm_train(Train_X,Train_Y,SizeMatrix,NumEpochs,ActivationFunction)
```

Input Arguments:

Train_X: As explained in Fisher.

Train_Y: As explained in Fisher.

SizeMatrix: Number of layers and number of neurons in each hidden layer. Let's say we want to create a network of 4 hidden layers, 1st layer having 100 nodes, 2nd having 200 nodes, 3rd having 400 nodes and 4th having 500 nodes. Then SizeMatrix = [100 200 400 500]

ActivationFunction: Activation function of neurons. There are 5 kinds of activation function available; 'sigmoid', 'sine', 'hardlim', 'tribas' and 'radbas'. This activation function is the implementation of Eq. 2

NumEpochs: Number of training epochs

Output Arguments:

stack: Trained network (connection weights of each layer)

Actual: Actual training labels

Predicted: Training labels predicted by the network

TrainingAccuracy: Prediction Accuracy on training data

B.9.2. helm_test.m

This function is used to evaluate ELM based deep network on input data.

Syntax

```
[Predicted, Accuracy, Actual] = helm_test(Test_X, Test_Y,
stack,ActivationFunction)
```

Input Arguments

Test_X: Testing inputs (Dimensions: Number of entries × Number of features)

Test_Y: Testing targets (Dimensions: Number of entries × Number of classes)

stack: Trained network (weights of each layer)

ActivationFunction: Activation function of neurons. It should be same as used in while training the network.

Output Arguments:

Predicted: Testing labels predicted by network

Actual: Testing labels

Accuracy: Prediction Accuracy

B.9.3. Sparse_elm_autoencoder

This function is based on ELM sparse auto-encoder proposed by Wentao Zhu et al. ELM auto-encoder estimates the weights by assuming that output of the auto-encoder is actually a transformed version of the input. Let's say 'A' is the input matrix and 'b' is the output matrix. By definition, 'b' is a transformed version of 'A' and we solve following equation by Moore-Penrose method to estimate linear weights 'x'

$$Ax = b$$

$$x = pinv(A)b$$

Syntax

```
W = sparse_elm_autoencoder(A,B,Lam,Itrs)
```

Input arguments

A: Input matrix transformed by auto-encoder. By cross referencing this parameter to the structure presented in Fig. 2, $A = x_2$

B: output of encoder. By cross referencing this parameter to the structure presented in Fig. 4-6, $B = x$

Lam: Error tolerance of Moore-Penrose pseudo inverse. By cross referencing to Eq. (4.7 f), $Lam = \varepsilon$

itrs: number of epochs

Output

W: Learned weights of auto-encoders. By cross referencing to Eq. (4.7 c), $W = w_o$

B.10. Deep Belief Networks

Implementation is done in three layers. In terms of functions hierarchy, there are three kinds of functions; tier 1, tier 2 and tier 3. Tier 1 function are directly accessible to end users. They can be accessed from command window directly. User can alter the input arguments as they please. These functions are responsible for initializing and training the network as a black box. Tier 2 functions are only accessible to tier 1 function. Their input arguments depend upon the behavior of tier 1 function. Users cannot change their input arguments at will because it will cause serious logical errors. These functions deals with individual layers. Tier 3 functions are the work horses of the DBN. They are only accessible in tier 2 functions. These functions can access the individual nodes of the network.

B.10.1. createAndTrainDBN: tier 1 function

This function is used to create a deep belief network object and then train it on the given input-target pairs. It accepts training inputs, training targets, number of training epochs and information about depth and width (architecture) of the network and return a trained DBN object along with some optional outputs. The DBN object is trained in following steps.

1) Using information about the architecture (number of layers and size of each layer), a network of restricted Boltzmann machines (RBM) is initialized along with input layer and final supervised layer.
2) Using given inputs, each layer is trained in an unsupervised manner as an RBM.
3) The RBMs are used to compute an output.
4) The computed output is compared to actual target and error is calculated.
5) This error is propagated backwards through all layers to arrive at initial input layer.
6) Difference between computed output and actual output is used to adjust connection weights.

Syntax

```
[dbn,Actual,Predicted, Accuracy] = createAndTrainDBN (Train_X,
Train_Y, SizeMatrix,BatchSize, NumEpochs)
```

Input Parameters

Train_X: As explained in Fisher.

Train_Y: As explained in Fisher.

SizeMatrix: Number of layers and number of neurons in each hidden layer

Let's say we want to create a network of 4 hidden layers, 1st layer having 100 nodes, 2nd having 200 nodes, 3rd having 400 nodes and 4th having 500 nodes. Size matrix is a row or column matrix having 4 elements such that SizeMatrix = [100 200 400 500]

NumEpochs: Number of training epochs

BatchSize: Number of input entries in each batch

Output Parameters

dbn: Trained network (connection weights of each layer)

Actual: Actual training labels

Predicted: Training labels predicted by the network

Accuracy: Prediction Accuracy on training data

B.10.2. testDBN: tier 1 function

This function is used to evaluate deep network on input data.

Syntax

```
[Accuracy,Predicted,Actual]=testDBN(dbn,X,Y)
```

Input Arguments

X: Testing inputs (Dimensions: Number of entries × Number of features)

Y: Testing targets (Dimensions: Number of entries × Number of classes)

dbn: Trained network (weights of each layer)

Output Arguments:

Predicted: Testing labels predicted by network

Actual: Testing labels

Accuracy: Prediction Accuracy

B.10.3. rbmtrain: tier 2 function

This function is used to train individual layers of DBN as RBMs. It is an internal function called inside *'createAndTrainDBN'*. It takes a batch of inputs feature matrix along with initialized RMB layer and trains it in an unsupervised manner by optimizing a cost function. It is a child function of *'createAndTrainDBN'*. In terms of hierarchy, this is a tier 2 function which means that it is only accessible to tier 1 function and not the end users.

Syntax

```
rbm = rbmtrain(rbm, X, opts)
```

Input Arguments

X: Input feature matrix (Dimensions: BatchSize × Number of features)

rbm: Initialized RBM structure

Output Arguments:

rbm: RBM layer with updated weights

B.10.4. nntrain: tier 2 function

This is also a tier 2 function, accessible inside *'createAndTrainDBN'*. It implements the supervised learning stage of final layer of DBN. As the final stage supervised, so it is trained as a regular feed-forward neural network. Training is done in three steps; 1st step is a feed forward pass, second is the error back propagation pass and the third is adjustment of connection weights and gradients. In feed-forward pass, input feature matrix is passed through trained RBM layers. Output of the feed-forward pass is compared to actual targets to calculate errors. The errors are propagated backward to calculate error gradients. Finally, the gradients are applied to connection weights to fine tune the network.

Syntax

```
[nn]  = nntrain(nn, Train_X, Train_Y, opts)
```

Input Arguments

Train_X: Training inputs (Dimensions: Number of entries × Number of features)

Train_Y: Training targets (Dimensions: Number of entries × Number of classes if binary class coding is used, Dimensions: Number of entries × 1 if actual class label is used instead of binary class coding). For example, there are two output classes, class 1 and class 2. In binary coding scheme, there are two columns of Train_Y, once corresponds to class 1 and the second corresponds to class 2. All entries of column 1 where there is class 1 equals to 1 while rest are 0. Same goes for column 2 and class 2.

nn: This structure represents the unfolded form of DBN which is trained in unsupervised manner. When a DBN is initialized in *'createAndTrainDBN'* and its layers are trained in unsupervised manner in *'rbmtrain'*, it is unfolded into a standard neural network. The unfolding is done in such a way that the RBMs becomes hidden layers of neural network. This unfolded structure is then passed to *'nntrain'*

opts: This structure specifies different parameters used in training the neural network (supervised stage of DBN). The parameters include number of training epochs, batch size and activation function.

Output Arguments:

nn: DBN network with fine-tuned weights.

B.10.5. nnff: tier 3 function

This is also a tier 3 function. It implements the feed-forward pass for final supervised learning.

Syntax

```
nn = nnff(nn, Batch_X, Batch_Y)
```

Input Arguments

Batch_X: A batch of training inputs (Dimensions: Number of entries × Number of features)

Train_Y: A batch of training targets

nn: This structure represents the unfolded form of DBN which is trained in unsupervised manner. When a DBN is initialized in *'createAndTrainDBN'* and its layers are trained in unsupervised manner in *'rbmtrain'*, it is unfolded into a standard neural network. The unfolding is done in such a way that the RBMs becomes hidden layers of neural network.

Output Arguments:

nn: DBN network with updated loss values and errors

nnbp: tier 3 function

This is also a tier 3 function. It implements the error back-propagation pass.

Syntax

```
nn = nnff(nn)
```

Input Arguments

nn: NN structure with updated loss values and errors. In essence, output of *'nnff'* becomes input of *'nnbp'*

Output Arguments:

nn: DBN network with backward propagated errors. The backward propagated errors are used to calculate gradients for weights adjustment

B.10.6. nnapplygrads: tier 3 function

This is also a tier 3 function. It implements the error back-propagation pass.

Syntax

```
nn = nnff(nn)
```

Input Arguments

nn: NN structure with backward propagated errors. The backward propagated errors are used to calculate gradients for weights adjustment

Output Arguments:

nn: DBN network with fine-tuned weights

Bibliography

[1] Tom Mitchell, “Machine Learning”, McGraw Hill, (1997).

[2] https://www.google.co.kr/webhp?sourceid=chrome-nstant&rlz=1C1ASUC_ enKR607KR607&ion =1&espv= 2&ie = UTF-8#q=machine%20learning

[3] https://support.google.com/websearch/answer/106230?hl=en

[4] http://www.mathworks.com/help/stats/index.html

[5] http://www.terasoft.com.tw/

[6] http://mirlab.org/jang/matlab/toolbox/machineLearning/

[7] http://people.kyb.tuebingen.mpg.de/spider/main.html

[8] http://gaussianprocess.org/gpml/

[9] http://www.gaussianprocess.org/gpml/code/matlab/doc/

[10] http://grupos.unican.es/ai/meteo/MeteoLab.html

[11] “Accelerating AI with GPUs: A new computing model”, NVidia blogs on deep learning, NVidia, https://blogs.nvidia.com/blog/2016/01/12/accelerating-ai-artificial-intelligence-gpus/

[12] Kaggle platform for data science competitions, https://www.kaggle.com/wiki/Software

[13] McLachlan, G. J. “Discriminant Analysis and Statistical Pattern Recognition”. Wiley Inter-science. ISBN 0-471-69115-1. MR 1190469, (2004).

[14] Geladi, P and Kowalski, B.R., "Partial Least-Squares Regression: A Tutorial", Analytica Chimica Acta, 185, 1-7, (1986).

[15] Jolliffe I.T. “Principal Component Analysis”, Series in Statistics, 2nd ed., Springer, NY, 487 p. 28, (2002).

[16] Aitchison, J. “Principal component analysis of compositional data”, Biometrika 70:57-65, (1983).

[17] Aitchison, J. “Reducing the dimensionality of compositional data sets”, J. Math. Geol. 16:617-35, (1984).

[18] Suykens, J.A.K., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J., “Least Squares Support Vector Machines”, World Scientific, Singapore, (2002).

[19] Cristianini, Nello; and Shawe-Taylor, John; “An Introduction to Support Vector Machines and other kernel-based learning methods”, Cambridge University Press, ISBN 0-521-78019-5, (2000).

[20] Kecman, V., “Learning and Soft Computing”, MIT Press, Cambridge, MA, (2001).

[21] Ashutosh Garg and Dan Roth, “Understanding Probabilistic classifiers”, Department of Computer Science and the Beckman Institute University of Illinois, Urbana, IL. 61801, USA, to appear in ECML’01

[22] Ben Taskar, Eran Segal, Daphne Koller, “Probabilistic Classification and Clustering in Relational Data”, Computer Science Dept. Stanford University Stanford, CA 94305

[23] Rish, Irina, “An empirical study of the naive Bayes classifier”. IJCAI Workshop on Empirical Methods in AI, (2001).

[24] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, “Extreme Learning Machine for Regression and Multiclass Classification”, IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics, vol. 42, no. 2, pp. 513-529, (2012).

[25] Z. Bai, G.-B. Huang, D. Wang, H. Wang and M. B. Westover, "Sparse Extreme Learning Machine for Classification," IEEE Transactions on Cybernetics, vol. 44, no. 10, pp. 1858-1870, (2014).

[26] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, “Extreme Learning Machine: A New Learning Scheme of Feedforward Neural Networks,” 2004 International Joint Conference on Neural Networks (IJCNN'2004), (Budapest, Hungary), (2004).

[27] G.-B. Huang, Q.-Y. Zhu and C.-K. Siew, “Extreme Learning Machine: Theory and Applications”, Neurocomputing, vol. 70, pp. 489-501, (2006).

[28] Deng, L.; Yu, D. “Deep Learning: Methods and Applications”, Foundations and Trends in Signal Processing **7** (3-4): 1–199, (2014).

[29] Bengio, Yoshua , “Learning Deep Architectures for AI”, Foundations and Trends in Machine Learning **2** (1): 1–127, (2009).

[30] Bengio, Y.; Courville, A.; Vincent, P., "Representation Learning: A Review and New Perspectives", IEEE Transactions on Pattern Analysis and Machine Intelligence **35** (8): 1798–1828. arXiv: 1206.5538, Doi: 10.1109/tpami.2013.50, (2013).

[31] R. Collobert and S. Bengio , "Links between Perceptrons, MLPs and SVMs", Proc. Int'l Conf. on Machine Learning (ICML), (2004).

[32] Rudolf Kruse, Frank Klawonn, Christian Moewes, Matthias Steinbrecher, Pascal Held, "Multi-Layer Perceptrons", Chapter: Computational Intelligence, Part of the series Texts in Computer Science pp 47-81

[33] Rosenblatt, Frank, "The Perceptron--a perceiving and recognizing automaton". Report 85-460-1, Cornell Aeronautical Laboratory, (1957).

[34] Auer, Peter; Harald Burgsteiner; Wolfgang Maass, "A learning rule for very simple universal approximators consisting of a single layer of perceptrons", Neural Networks **21** (5): 786-95. doi:10.1016/j.neunet, (2007).

[35] Rumelhart, David E., Geoffrey E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation". David E. Rumelhart, James L. McClelland, and the PDP research group. Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundations. MIT Press, (1986).

[36] "Convolutional Neural Networks– DeepLearning 0.1 documentation", DeepLearning 0.1. LISA Lab, (2013).

[37] Wentao Zhu, Laiyun Qing, Guang-Bin Huang, "Hierarchical Extreme Learning Machine for Unsupervised Representation Learning", International Joint Conference on Neural Networks (IJCNN), p1-8, (2015)

[38] "Supervised learning and optimization", Tutorial from Stanford online repository on Deep Learning, http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

[39] Hinton, G., "Deep belief networks". Scholarpedia **4** (5): 5947. doi:10.4249/scholarpedia.5947, (2009).

[40] Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", (2012).

[41] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J., "Learning representations by back-propagating errors". Nature 323 (6088): 533–536. doi:10.1038/323533a0, (1986).

[42] Quan Wang, Signal Analysis and Machine Perception Laboratory Department of Electrical, Computer, and Systems Engineering Rensselaer Polytechnic Institute, Troy, NY 12180, USA

[43] http://read.pudn.com/downloads103/sourcecode/math/421402/drtoolbox/techniques/train_rbm.m__.htm

[44] https://sites.google.com/site/mihailsirotenko/projects/cuda-cnn

[45] http://www.codeforge.com/read/222934/train_mlp.m__html

[46] http:// mathworks.com/ matlabcentral/ fileexchange/ 37737-naive-bayes -classifier/ content/ NaiveBayes Classifier.m

[47] http://www.ntu.edu.sg/home/egbhuang/elm_codes.html

[48] http://kr.mathworks.com/matlabcentral/fileexchange/38310-deep-learning-toolbox

[49] https://github.com/elan2wang/AI_Algorithm/tree/master/lda

[50] http://www.kernel-methods.net/matlab/algorithms/pls.m

[51] http://www.mathworks.com/help/stats/svmtrain.html