Titre: Title:	Learning Activation Functions in Deep Neural Networks
Auteur: Author:	Farnoush Farhadi
Date:	2017
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Farhadi, F. (2017). Learning Activation Functions in Deep Neural Networks [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2945/

UNIVERSITÉ DE MONTRÉAL

LEARNING ACTIVATION FUNCTIONS IN DEEP NEURAL NETWORKS

FARNOUSH FARHADI
DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INDUSTRIEL)
DÉCEMBRE 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

LEARNING ACTIVATION FUNCTIONS IN DEEP NEURAL NETWORKS

présenté par : FARHADI Farnoush
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. ADJENGUE Luc-Désiré, Ph. D., président
M. LODI Andrea, Ph. D., membre et directeur de recherche
M. PARTOVI NIA Vahid, Doctorat, membre et codirecteur de recherche
M. CHARLIN Laurent, Ph. D., membre

DEDICATION

This thesis is dedicated to my beloved parents, Ahmadreza and Sholeh, who are my first teachers and always love me unconditionally.

This work is also dedicated to my love, Arash, who has been a great source of motivation and encouragement during the challenges of graduate studies and life.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Prof. Andrea Lodi for his permission to be my supervisor at Ecole Polytechnique de Montreal and more importantly for his enthusiastic encouragements and invaluable continuous support during my research and education. I am very appreciated to him for introducing me to a MITACS internship in which I have developed myself both academically and professionally.

I would express my deepest thanks to Dr. Vahid Partovi Nia, my co-supervisor at Ecole Polytechnique de Montreal, for his supportive and careful guidance and taking part in important decisions which were extremely precious for my research both theoretically and practically.

I would also like to thank the examining committee for taking time out from their busy schedule to read and review my thesis and provide me with their insightful comments.

Further acknowledgements go to the members of the Excellent Research Chair in Data Science for Real-time Decision-making who were eager to help me whenever needed.

I would like to express my deepest gratitude and special thanks to Lane Cochrane, the Chief Analytics Officer at iPerceptions, for giving me the opportunity to do an internship within the company. For me it was a unique experience to be at iPerceptions and to do the research on machine learning. I would like to thank Sylvain Laporte, the Chief Technology Officer at iPerceptions, who in spite of being extraordinarily busy with his duties, took time out to give me necessary advice. I am especially thankful to all my collegues at R&D team at iPerceptions for their careful and valuable guidance to make life easier.

And on a personal note, a special word of thanks goes to my parents who always stand behind me and support me spiritually throughout my life. My special thanks to Arshin, Mehrnoush and Siavash, my sisters and brother-in-law whose encourage and guides always help me in my personal and professional life. I am grateful for the love, encouragement, and tolerance of Arash, the man who has made all the difference in my life. Without his patience and love, I could not have completed this thesis.

I am especially grateful to Atefeh, my best friend, who has been there whenever I needed her. She always has the time to hear about my successes and failures and has always been a faithful listener.

I would like to thank my dear friend David Berger for helping me in translating the abstract into French and Chris Nael for his assistance in general editing.

RÉSUMÉ

Récemment, la prédiction de l'intention des utilisateurs basée sur les données comportementales est devenue plus attrayante dans les domaines de marketing numérique et publicité en ligne. La prédiction est souvent effectuée par analyse de l'information du comportement des visiteurs qui comprend des détails sur la façon dont chaque utilisateur a visité le site Web commercial. Dans ce travail, nous explorons les méthodes statistiques d'apprentissage automatique pour classifier les visiteurs d'un site Web de commerce électronique d'après des URL qu'ils ont parcourues. Plus précisément, on vise à étudier comment les réseaux de neurones profonds de pointe pourraient efficacement prédire le but des visiteurs dans une session de visite, en utilisant uniquement des séquences d'URL. Dans un réseau contenant des couches limitées, le choix de la fonction d'activation a un effet sur l'apprentissage de la représentation et les performances du réseau.

Typiquement, les modèles de réseaux de neurones sont appris en utilisant des fonctions d'activation spécifiques comme la fonction sigmoïde symétrique standard ou relu linéaire. Dans cette mémoire, on vise à proposer une fonction sigmoïde asymetrique paramétrée dont la paramètre pourrait être controlé et ajusté dans chacun des neurons cachées avec d'autres paramètres de réseau profond en formation. De plus, nous visons à proposer une variante non-linéaire de la fonction relu pour les réseaux profonds. Comme le sigmoïde adaptatif, notre objectif est de régler les paramètres du relu adaptatif pour chaque unité individuellement, dans l'étape de formation.

Des méthodes et des algorithmes pour développer ces fonctions d'activation adaptatives sont discutés. En outre, une petite variante de MLP (Multi Layer Perceptron) et un modèle CNN (Convolutional Neural Network) applicant nos fonctions d'activation proposées sont utilisés pour prédire l'intention des utilisateurs selon les données d'URL. Quatre jeux de données différents ont été choisis, appelé les données simulées, les données MNIST, les données de revue de film, et les données d'URL pour démontrer l'effet de sélectionner différentes fonctions d'activation sur les modèles MLP et CNN proposés.

ABSTRACT

Recently, user intention prediction based on behavioral data has become more attractive in digital marketing and online advertisement. The prediction is often performed by analyzing the logged information which includes details on how each user visited the commercial website. In this work, we explore the machine learning methods for classification of visitors to an e-commerce website based on URLs they visited. We aim at studying on how effectively the state-of-the-art deep neural networks could predict the purpose of visitors in a session using only URL sequences. Typical deep neural networks employ a fixed nonlinear activation function for each hidden neuron. In this thesis, two adaptive activation functions for individual hidden units are proposed such that the deep network learns these activation functions in training. Methods and algorithms for developing these adaptive activation functions are discussed. Furthermore, performance of the proposed activation functions in deep networks compared to state-of-the-art models are also evaluated on a real-world URL dataset and two well-known benchmarks, MNIST and Movie Review benchmarks as well.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND NOTATIONS

1NN	one hidden-layer neural network
2NN	two hidden-layer neural network
adarelu	adaptive relu activation function
adasigmoid	adaptive sigmoid activation function
ANN	Artificial Neural Networks
BOW	Bag of Words
CM	Company data
CNN	Convolutional Neural Networks
GB	Gradient Boosting
GLMs	Generlazied Linear Models
IRLS	Iteratively Reweighted Least Squares
LDA	Latent Dirichlet Allocation
LR	Logistic Regression
MLE	Maximum Likelihood Estimates
MLP	Multi Layer Perceptrons
MNIST	Modified National Institute of Standards and Technology dataset includes hand-written digit images
MR	Movie Review data
NB	Naive Bayes
NLP	Natural Language Processing
Pr	Precision
Re	Recall
relu	rectifier linear unit
RF	Random Forest
SM	Simulated data
SVM	Support Vector Machine
Tf-idf	Term frequency-inverse document frequency
URL	Universal Resource Locator

Notations

n	data size
m	number of data features
$\mathbf{x}$	input vector $\mathbf{x} = [x_0, ..., x_m]^\top$ augmented by 1
$\mathbf{X}$	input matrix $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n]$ where $\mathbf{x}_i = [x_0, ..., x_m]^\top$
$\mathbf{y}$	output vector $\mathbf{y} = [y_1, ..., y_n]$
$\mathbf{Y}$	output matrix $\mathbf{Y} = [\mathbf{y}_1, ..., \mathbf{y}_n]$
$\mathbf{w}$	weight vector $\mathbf{w} = [w_0, ..., w_m]^\top$ including bias
$\mathbf{W}$	weight matrix $\mathbf{W}$ of size $(m+1) \times K$
w_0	bias
K	number of classes in multi-classification problem
$\mathcal{L}$	loss function
L	number of hidden layers in MLPs
l	index of layers in MLPs
N^l	number of neorons in each layer l
act	activation function in MLP
$\mathbf{f}$	output of softmax layer in MLP with K entries l
$\mathbf{W}^l$	weight matrix in layer l in MLP
$\mathbf{h}^l$	result of applying activation function in a layer in MLP
$\boldsymbol{\eta}$	weighted sum $\mathbf{Wx}$ in MLP
K'	number of mini-batches in mini-batch stochastic gradient descent
γ	learning rate
α	parameter of adaptive activation function
$\mathbf{t}$	response vector
$\mathbf{T}$	response matrix for n examples
ϕ	basis function
$\mathbf{\Phi}$	matrix of basis functions
ℓ	log likelihood
E	Error function
$\mathcal{N}$	Normal distribution
σ	dispersion parameter
μ	mean
ε	noise
Ω	Diagonal matrix in weighted least squares
$\mathbf{z}$	working dependent vector
η	linear predictor

$\mathbb{V}$	mathematical operator of variance
ε_0	margin in perceptron
$t_{\max}$	maximum number of iterations in perceptron learning rule
$\beta_{\max}$	upper bound in perceptron learning rule

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Recently, online e-commerce application has become one of the most popular approaches for people to browse, compare and purchase products. On the other hand, product marketing and online advertisers realize its importance for providing cost-effective opportunities to improve decision-making in digital marketing strategies. Most of the revenue brought to the industry are featured by fitting a probabilistic model to historical event data, in order to predict how a current visitor will behave in the near future, based on their behavior profile. The concept of user behavioral profiling includes collecting and analyzing multiple events, in which each event could be attributed by a single origin entity according to obtain more information about the origin entity. This process consists of two phases :

1. Data collection including recording, storing and tracking.
2. Looking for recognizing a desired pattern based on state-of-the-art machine learning techniques on collected data.

Suppose that a customer intends to shop online for a pair of shoes. He surfs the net and searches for shoes anonymously, but makes no purchase. Subsequently, he visits a web site of the local market that displays advertisements offering new collection of a specific brand. While no personal identification information might have been collected, his interest in shoes has been noted. Online publishers and advertisers aim at increasing the performance of advertising through user behavior information. Particularly, behavioral targeting gets benefit from the information collected from an individual's web-browsing behavior to display a better advertisement match to that user in real-time. The collected data from a user behavioral tracking includes the pages he visits, the duration time he visits each page, the links he clicks on, the search engine or web-browsers he uses, the referral medium to this web site and the objects that he interacts with. All of this information allow websites to create a profile that links to that visitor's web browser. In light of this situation, our motivation is to develop a methodology for predicting the user intentions based on the web page URLs they navigated, and to leverage this information to enhance marketing strategies. The prediction is typically performed based on the web page content along with the link structure of the web graph (Chakrabarti *et al.*, 1998). There are several advantages to predict the visitors intentions using only the URLs they navigated and this is the problem we studied in this thesis.

One benefit of such an approach is time efficiency. The URL could reflect a small fraction of the corresponded page content which mostly enables a much faster feature building and accelerates the classification, consequently.

Using the URLs for classification has several applications in different domains. (Chakrabarti *et al.*, 1999) introduces a fast web crawler to discover the relevant web pages to a pre-defined topic set using a starting set of URLs. Accordingly, such systems could speed-up the web page classification by predicting the topic of URLs before downloading their contents and avoiding consuming the excess bandwidth. Using URLs has another application in personalized web browsers wherein the visited URLs could be annonated based on the visitors' topic interests (Baykan *et al.*, 2008, 2009; Qi et Davison, 2009; Kan, 2004; Kustanowitz *et al.*, 2006).

Another line of our work focuses on using machine learning approaches to predict the user intentions. Generally most prior work includes Logistic Regression and Decision Trees (Richardson *et al.*, 2007; Yan *et al.*, 2014) due to their simplicity and effectiveness. Although, these are widely-used approaches in this domain, these methods could not tackle the sparsity and high-dimensionality challenges seen in text data (McMahan *et al.*, 2013). Furthermore, the standard machine learning methods could not explain the latent features of data or reveal the complex relationship among features (Liu *et al.*, 2015).

Identifying the user intent behind the URL sequence is a crucial step toward reaching this goal. User intention prediction using only URLs is a challenging task because URLs are usually short, and identifying the exact intent of the visitors requires more context beyond the extracted tokens from URLs.

In recent decade, the field of deep neural networks (deep learning) achieved considerable success in pattern recognition and text classification. There are a lot of studies and practical applications of deep learning on images, video or text classification, the application of these concepts on user intention prediction are just about to make their first steps in research (Liu *et al.*, 2015; Vieira, 2015; Korpusik *et al.*, 2016; Lo *et al.*, 2016; Hashemi *et al.*, 2016). Our approach in this work is to use deep learning methods to find URLs sequence representations and apply them as features to predict the user intention. Deep learning approaches are widely-used approaches in text classification tasks through leveraging word vector representations like word2vec (Kim, 2014). In this thesis, we learn URL sequence representations using Convolutional Neural Networks (CNNs) which is trained on top of word vector representations. However, CNNs were originally applied in computer vision, they have shown a great success in several natural language processing tasks, such as text classifications, sentiment analysis (Kim, 2014), semantic parsing (Yih *et al.*, 2014) and sentence modeling (Kalchbrenner *et al.*, 2014; Kim, 2014). Our work differs from text classification in the sense that URLs are often short and unstructured.

Typically, the network parameters are learned to fit the data, while the activation functions are pre-specified to be a sigmoid, tanh or relu function. A deep neural network applying any

of these common nonlinear functions can approximate arbitrarily functions (Hornik *et al.*, 1989; Cho et Saul, 2010). In a network with limited layers, the choice of activation function has an effect on the representation learning and the network performance (Agostinelli *et al.*, 2014). Although the rectifier linear unit (relu) function does not saturate like the sigmoid nonlinear function where corresponding units suffer from the "relu dying" problem during training. Hence, developing an adaptive activation function that is capable of fast training of deep neural networks more accurately has attracted attention in recent years (Agostinelli *et al.*, 2014; Jarrett *et al.*, 2009; Glorot *et al.*, 2011; Goodfellow *et al.*, 2013; Springenberg et Riedmiller, 2013; Dushkoff et Ptucha, 2016).

In this context, this thesis develops a variant of deep neural networks using adaptive forms of standard activation functions in order to predict the user intentions by focusing on URL sequences they visited.

An excerpt of the most relevant work in the context of deep neural networks is reviewed in following section. An introduction of applying adaptive activation functions is presented in sequel.

1.1 Deep Neural Networks

According to (Schmidhuber, 2015), the idea of deep networks referred to the work of (Ivakhnenko et Lapa, 1966). The idea of early neural network models dates back to the work of (McCulloch et Pitts, 1943) but their models did not learn. The simple networks that worked properly based on supervised learning were trained by (Rosenblatt, 1958; Widrow, 1962). The neural network-based systems trained with backpropagation and minimization of error by gradient descent were discussed in (Kelley, 1960; Bryson, 1961; Bryson et Denham, 1962). Furthermore, the work of using other optimization methods, namly steepest and momentum to accelerate the backpropagation were introduced in (Bryson, 1961) and (Rumelhart *et al.*, 1988), repectively. Some advanced algorithms to control the backpropagation by adjusting a a global learning rate were introduced by (Battiti, 1989) and (LeCun *et al.*, 1993) as well. An efficient backpropagation to minimize the loss function rather than error was introduced by (Dreyfus, 1973) and the first neural network using this efficient backpropagation was described in detail in (Werbos, 1982). The related papers regarding the developing of such neural network architectures were published in (LeCun *et al.*, 1998b). (Rojas, 1996) and (Witten *et al.*, 2016) also provided a formulation for backpropagation in a matrix-vector format. The idea of optimization through stochastic gradient descent was developed by (Robbins et Monro, 1951).

The first deep learning network of the multi-layer perceptrons variant was trained by (Ivakhnenko et Lapa, 1966). The proposed network, called GMDH, includes the hidden neurons using a type of polynomial activation function and the hidden layers were learned by regression. The hierachical representation of this network containing 8 hidden layers was described in details in (Ivakhnenko, 1971). (Bottou, 2012) and (Bengio, 2012) presented some theoretical and empirical tricks and recommendations for learning deep neural networks with stochastic gradient descent.

The origin of the convolutional neural networks (CNNs) as the deep variant of neural networks were first inspired from the work on the animal visual cortex by (Hubel et Wiesel, 1962) where they discovered that the cells in cat's visual cortex act as local filters that will be fired in response to a specific visual pattern inputs (e.g. edge orientations). In this context, the first modern CNN, "Neocognitiveon", was proposed by (Fukushima et Miyake, 1982). The "LeNet", a CNN-based model augmented by max-pooling and using backpropagation was proposed by (LeCun *et al.*, 1998a). The "LeNet" architecture has been an essential part of several competition-wining solutions and state-of-the-art proposed CNN-based models in computer vision (Krizhevsky *et al.*, 2012).

The crucial effect of parameter initialization in deep networks was discussed in (LeCun *et al.*, 1998b) and recently in (Krizhevsky *et al.*, 2012). Further, (Glorot et Bengio, 2010) discussed several weight initialization and how the different combination of number of input and output units in weight initialization formulation along with different type of activation functions could be employed in deep neural networks.

Applying dropout as a regularization method in deep neural networks was introduced by (Srivastava *et al.*, 2014). Furthermore, using batch normalization to speed up the deep network training by normalizing the layer inputs was introduced in (Ioffe et Szegedy, 2015). Their method treats as a regularizer and is less careful regarding the parameter initialization.

Recently, CNN models have achieved considerable success in computer vision and speech recognition domains (Krizhevsky *et al.*, 2012; Graves *et al.*, 2013). In the context of natural language processing most of the work was focused on word vector representation (Bengio *et al.*, 2003; Mikolov *et al.*, 2013). By the virtue of existing a strong spatially correlation in natural languages, CNN models could be used effectively for such tasks as well and achieve excellent results compared to the state-of-the-art approaches. (Shen *et al.*, 2014) introduced a CNN latent semantic model for search query retrieval. (Kalchbrenner *et al.*, 2014) presented a dynamic CNN model using dynamic k-max pooling in sentences modeling applications. (Kim, 2014) exhibited that a simple CNN model including one convolutional layer and public word2vec word vectors could outperform the state-of-the-art machine learning models for

sentiment analysis on multiple benchmarks.

1.2 Adaptive Activation Functions

A sufficiently large neural network using a sigmoid, tanh or rectifier linear unit (relu) function in nonlinearity components could approximate any arbitrary complex function (Hornik *et al.*, 1989) and (Cho et Saul, 2010). However, in deep networks with a specific number of hidden layers, the choice of the activation functions has an important impact on the network performance. For example, sigmoid suffers from the gradient vanishing and "dying relu" often occurs in relu neurons. One proposed approach to mitigate these difficulties in deep neural networks is applying maxout, which leverages the dropout model averaging technique (Goodfellow *et al.*, 2013).

Most the state-of-the-art deep neural networks choose the activation function as a hyper parameter in the training. In this context, prior work focused on genetic and evolutionary approaches such that activation function for each hidden unit was chosen from a predefined set (Yao, 1999). Further, (Turner et Miller, 2014) proposed a combination of these methods with a scaled parameter which was tuned in the training step. Selecting more than one activation function for each unit in a deep CNN model was studied in (Dushkoff et Ptucha, 2016).

In recent years, developing activation functions that speeds up the training step and produces the accurate results has attracted remarkable attention. (Agostinelli *et al.*, 2014) introduced an adaptive piece-wise linear function that could be learned for each hidden neuron, individually. A parametric form of relu to generalize the standard relu function was proposed by (He *et al.*, 2015). Their proposed activation function enables deep networks to train extremely deep models and achieved excellent results on the GoogleLeNet dataset. (Zhang et Woodland, 2015) introduced a parameterized symmetric form of the sigmoid by adding a scale parameter. Additionally, an adaptive relu with a parameter to scale the hinge-like shape was proposed in their work as well. (Hou *et al.*, 2016, 2017) presented a smooth piece-wise polynomial activation function to decrease the bias of the regression neural networks. The capability of their proposed activation function in approximating any continuous function was discussed in their paper. Note that relu is a sort of piece-wise linear function wherein the negative part is always assigned to zero. By contrary, the leaky relu has a small predefined negative slope for the negative inputs (Maas *et al.*, 2013). Compared to leaky relu, elu (Clevert *et al.*, 2015) attempted to exponentially decrease the slope from a predefined value to zero. Using elu mostly accelerates training by preventing a bias shift that often occurs

in relu networks. (Qian *et al.*, 2018) proposed a mixed function of leaky relu and elu as an adaptive function that could be learned in a data-driven way. A scaled linear and non-linear combination of leaky relu and elu was explored in their work. Furthermore, the performance of a hierarchical integration of these activation functions compared to standard relu in a deep CNN model was well investigated on several image benchmarks.

Inspired from (Agostinelli *et al.*, 2014; Zhang et Woodland, 2015) and (Qian *et al.*, 2018), our contribution in this thesis is to study the effect of asymmetry on a parametric sigmoid. The goal is to propose an adaptive sigmoid function with a parameter that could be controlled and fitted for each hidden neuron along with other deep network parameters in training. Furthermore, we aim at proposing a smooth variant of relu function for deep networks. Like adaptive sigmoid, our goal is to tune the parameters of the adaptive relu for each unit separately in training step. We also indicate that these parametric form of activation functions improve the performance of deep networks in multiple classification tasks on text and image data.

1.3 Research Objectives

The goal of this master thesis is to explore the problem of user intention prediction using URLs dataset. In particular, this thesis proposes two novel adaptive activation functions employed in deep neural networks, namely in Multi Layer Perceptrons (MLP) and in CNNs. On one hand, the standard variant of deep networks was applied for learning the visitors intentions. On the other hand, a slight variant of CNN architecture proposed in (Kim, 2014) along with our proposed activation functions was used to classify the user intentions based on URLs they navigated. Aside from this, the performance of the MLP and CNN models using our proposed activation functions is evaluated on several datasets, namely simulated data, MNIST and Movie Review. Thus, discovering the answers of the following questions are the topic of this thesis :

- How can an adaptive version of the standard activation functions be developed ?
- How can the parameters of such adaptive functions be trained in neural networks ?
- How accurate the deep networks based on these adaptive activation functions can predict the users intentions based on the URLs they visited ?
- How accurate our proposed models perform the prediction versus the state-of-the-art deep networks in different applications ?

1.4 Our Contributions

The main contribution of this thesis is developing a parametric activation function for deep neural networks and evaluation on user intention prediction with URLs data. The thesis contributions are listed in detail as follows :

1. Developing and proposing a parametric and adaptive form of two standard activation functions, an asymmetric sigmoid and a smooth version of relu, for deep neural networks such that corresponding parameters could be learned along with the network parameters in training.
2. Evaluating the performance of deep networks with the proposed adaptive activation functions on a real-world URL-based data.
3. A quantitative analysis that evaluates the performance of the deep networks with adaptive activation functions on different benchmarks, such as MNIST and Movie Review.

1.5 Thesis Structure

The rest of this thesis is organized as follows. Chapter 2 focuses on overview the primary concepts of the thesis, linear regression and generalized linear regression, which are used in describing our suggested activation functions. Chapter 3 covers the fundamental concepts that are required to understand the architecture of deep neural networks. It also provides a deeper look at perceptron and convolutional neural networks while explaining how these models are trained. Our proposed adaptive activation functions for deep networks and their properties are described in Chapter 4. Chapter 5 gives a brief overview of the different datasets used within this thesis. Four different datasets have been chosen, namely Simulated data, MNIST data, Movie Review data and URLs data. Many experimental results on these data are demonstrated and the effect of selecting different activation functions on the models performance is investigated, separately in detail. The results and analysis of the deep networks with proposed activation functions on chosen datasets are discussed at the end. Chapter 6 contains the conclusion of the thesis and possibilities of further work on the subject.

CHAPTER 2 OVERVIEW

2.1 Linear Regression

A linear regression model assumes that the regression function $\mathbb{E}(y|\mathbf{x})$ is linear regarding the input vector $\mathbf{x}$. In this section the linear methods for regression is described and linear method for classification applications is described in the next section.

In linear regression [1], we aim at explaining a dependent response variable y using linear predictor functions whose unknown model parameters are learned from the independent regressor variables $x_1, x_2, ..., x_m$ (which is different from independent random variables). The simplest form for a linear regression model consists of a linear combination of the input variables and takes the form :

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + ... + w_m x_m, \tag{2.1}$$

where $\mathbf{x} = [x_1, ..., x_m]^\top$. The important property of the linear regression model is that it is a linear function of the regression coefficients $w_1, ..., w_m$. The linear regression model remains linear even if one of the regressors is a non-linear function of the other regressor or of the all data predictors. In other words, the model is considered linear as long as it is linear in terms of its own parameter $\mathbf{w}$, where $\mathbf{w}$, is an m-dimensional parameter vector and w_0 is the intercept (a constant term). The elements in the $\mathbf{w}$ vector are called regression coefficients. It is often required to show the intercept separately in several statistical learning procedures for linear models. For prediction purposes, linear regression is typically applied to learn a predictive model based on an observed data $\mathbf{x}$ and y values. By developing such a learned model, we would then be able to make a prediction of the value of y for an unseen vector of $\mathbf{x}$. Linear regression model estimation and inference focuse on fitting $\mathbf{w}$ parameter (Bishop, 2006).

Linear regression definition can be extended by considering linear combinations of fixed nonlinear functions of the input variables for a univariate x as following :

$$y(x, \mathbf{w}) = w_0 + \sum_{j=1}^{m} w_j \phi_j(x), \tag{2.2}$$

1. This section is heavily based on notes and notations inherited from (Bishop, 2006).

where $\phi_j(x)$ are considered as basis functions and m is the total number of parameters in the model. The intercept parameter w_0 is known as any fixed bias in the data and is often referred to as the bias parameter. By defining the basis function $\phi_0(x) = 1$ for bias,

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{m} w_j \phi_j(x) = \boldsymbol{\phi}^\top \mathbf{w}, \tag{2.3}$$

where $\mathbf{w} = [w_0, ..., w_m]^\top$ is the vector of m regressors and $\boldsymbol{\phi}(x) = [\phi_0(x), ..., \phi_m(x)]$ is the vector of basis functions at input $\mathbf{x}$. By considering nonlinear basis functions, the function $y(\mathbf{x}, \mathbf{w})$ could be a nonlinear function of the input vector $\mathbf{x}$.

Functions of the form (2.3) are typically called linear models, since this function is linear with respect to its parameter $\mathbf{w}$. This linearity makes the analysis of the model extremely convenient.

There exist several possible choices for the basis functions, like :

$$\phi_j(x) = \exp\left\{-\frac{(x - \mu_j)^2}{2\sigma^2}\right\}, \tag{2.4}$$

where ϕ_j is usually called as a Gaussian basis function and the parameters μ_j and σ denote the location and spatial scale of the basis functions in input space, respectively. Another choice for the basis function is defined as

$$\phi_j(x) = \text{sigmoid}(x) = \frac{1}{1 + \exp(\frac{x - \mu_j}{\sigma})}. \tag{2.5}$$

Also tanh function could be applied as the basis function as it is a rescaling form of the logistic sigmoid and takes the form

$$\tanh(x) = 2\text{sigmoid}(2x) - 1. \tag{2.6}$$

Hence, equivalently for a linear combination of logistic sigmoid functions, we have a linear combination of tanh function with scaled regression coefficients as well. Most of the discussion in this section is independent of the specific form of basis function set. The remainder of the section will apply the assumption that the basis function vector $\phi(x)$ is simply the identity $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$ (Bishop, 2006).

2.1.1 Maximum Likelihood Estimation and Least Squares

A linear regression model assumes that the response variable t is explained by a linear combination of the $\mathbf{x}$'s as follows :

$$t = y(\mathbf{x}, \mathbf{w}) + \varepsilon, \tag{2.7}$$

where t is a single response variable, and

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + ... + w_m x_m$$

and ε is a zero mean Gaussian random variable with precision or inverse variance equals $\frac{1}{\sigma^2}$ and is often called the error term or noise. ε is considered as an unobserved random variable that adds noise to the linear regression model between the response variable and predictors. Thus, the model takes the form

$$p(t|\mathbf{x}, \mathbf{w}, \sigma^2) \equiv \mathcal{N}(t|\mathbf{x}, \mathbf{w}, \sigma^2)$$

where $\mathcal{N}$ is known as a Gaussian distribution for a single real-valued variable x and defined as

$$p(t|\mathbf{x}, \mathbf{w}, \sigma^2) = \frac{1}{\sqrt{(2\pi\sigma^2)}} \exp\left\{-\frac{1}{2\sigma^2}(t - \mathbf{x}^\top \mathbf{w})^2\right\}$$

If one assumes a linear regression, then typically a least squares method is used to estimate the model coefficients and then the best prediction for an unseen value of $\mathbf{x}$, will be obtained by the conditional mean of the response variable (Bishop, 2006). Least Squares method searches for the best parameters w_i in (2.7) by minimizing the sum of squares of total error. In the case of a Gaussian conditional distribution, the error term ε has mean zero and constant variance σ^2 such that $\mathbb{E}(\varepsilon) = 0$ and $\text{var}(\varepsilon) = \sigma^2$ which implies that the conditional distribution of t given $\mathbf{x}$ is Unimodal. Hence, for the conditional mean, we can simply write :

$$\mathbb{E}(t|\mathbf{x}) = \int t\text{p}(t|\mathbf{x})dt = y(\mathbf{x}, \mathbf{w}). \tag{2.8}$$

Note that this assumption may not be true for some applications (Bishop, 2006).

To model data $(\mathbf{x}_1, t_1), ..., (\mathbf{x}_n, t_n)$ where $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n]$, $\mathbf{x}_i = [x_1, ..., x_m]^\top$ and t_i is a real-valued target variable and under the assumption that the input data are drawn independently identically from the distribution (i.i.d.) (2.7), the following equation for the corresponding

probability density function is formulated as the likelihood as below :

$$\mathcal{L}(\mathbf{w}, \sigma) = p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma) = \prod_{i=1}^{n} \frac{1}{\sqrt{(2\pi\sigma^2)}} \exp\left\{-\frac{1}{2\sigma^2}(t_i - \mathbf{x}_i^\top \mathbf{w})^2\right\}, \tag{2.9}$$

where the likelihood is known as the function of the adaptable coefficients $\mathbf{w}$. It should be noted that in linear regression learning as a supervised learning problem, one aims at learning the model parameters $\mathbf{w}$ rather than the distribution of the predictors $\mathbf{X}$. Therefore, $\mathbf{X}$ will always be developed in the set of conditioning variables and could be dropped from expressions for simplifying the notations as $p(\mathbf{t}|\mathbf{w}, \sigma)$.

Since the logarithm of the likelihood is a monotonically increasing function, it reaches to the maximum value at the same value as the likelihood. It is more convenient to use the log-likelihood instead of the likelihood function because it is additive with increasing data. Hence, by taking the logarithm of the likelihood function and making use of the Gaussian distribution for a single real-valued variable $\mathbf{x}$, we have :

$$\log \mathcal{L}(\mathbf{w}, \sigma) = \log p(\mathbf{t}|\mathbf{w}, \sigma) = \sum_{i=1}^{n} \log p(t_i|\phi_1(x_i), ..., \phi_m(x_i), \mathbf{w}, \sigma^2)$$

$$= -\frac{n}{2} \log 2\pi\sigma^2 - \frac{1}{\sigma^2} E(\mathbf{w}), \tag{2.10}$$

where E is the sum-of-squares error function and defined by :

$$E = \frac{1}{2} \parallel \mathbf{t} - \mathbf{y} \parallel^2$$

$$= \frac{1}{2} \sum_{i=1}^{n} (t_i - y_i)^2$$

$$= \frac{1}{2} \sum_{i=1}^{n} \left\{t_i - \boldsymbol{\phi}_i^\top \mathbf{w}\right\}^2, \tag{2.11}$$

and $\boldsymbol{\phi}_i = [\phi_0(\mathbf{x}_i), ..., \phi_m(\mathbf{x}_i)]$ (Bishop, 2006).

As clearly understood from (2.10) and (2.11), for any fixed $\sigma > 0$, maximizing likelihood $\mathcal{L}$ is equivalent to minimizing a sum-of-squares error E.

MLE attempts to find the best parameter values so that the observed data has the biggest probability by differentiating the likelihood function with respect to its parameters. As seen from (2.11), this algorithm is often easier when the trick of maximizing the log-likelihood

is used in replace of the likelihood function itself to determine $\mathbf{w}$ and σ parameters. In this case, by taking the logarithm of the product of terms in likelihood, we would have a sum of individual logarithms, which is often easier to take the derivatives than the original likelihood. By taking the gradient of the log likelihood, we have :

$$\bigtriangledown \log p(\mathbf{t}|\mathbf{w}, \sigma) = \sum_{i=1}^{n} \{t_i - \boldsymbol{\phi}_i{}^{\top}\mathbf{w}\}\boldsymbol{\phi}_i. \tag{2.12}$$

Setting (2.12) to zero and maximizing the log-likelihood with respect to the weight $\mathbf{w}$ and noise precision σ^2 parameters, gives us :

$$\mathbf{w}_{\mathrm{ML}} = (\boldsymbol{\Phi}^{\top}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\top}\mathbf{t}$$

and

$$\sigma^2_{\mathrm{ML}} = \frac{1}{n}\sum_{i=1}^{n} \{t_i - \boldsymbol{\phi}_i{}^{\top}\mathbf{w}_{\mathrm{ML}}\}^2, \tag{2.13}$$

where $\boldsymbol{\Phi}$ is an $n \times (m+1)$ matrix, whose elements are calculated by $\boldsymbol{\Phi}_{ij} = \boldsymbol{\phi}_j(\mathbf{x}_i)$, as follows :

$$\boldsymbol{\Phi} = \begin{bmatrix} \phi_0(x_1) & \dots & \phi_m(x_1) \\ \phi_0(x_2) & \dots & \phi_m(x_2) \\ \vdots & \ddots & \vdots \\ \phi_0(x_n) & \dots & \phi_m(x_n) \end{bmatrix}, \tag{2.14}$$

where $\boldsymbol{\Phi} = [\boldsymbol{\phi}_0, ..., \boldsymbol{\phi}_n]^{\top}$ and $\boldsymbol{\phi}_i = [\phi_0(\mathbf{x}_i), ..., \phi_m(\mathbf{x}_i)]^{\top}$ (Bishop, 2006).

2.1.2 Multiple Outputs

Multiple output linear regression, which is known as a multiple response model in the literature, refers to the standard linear regression, except the output is a vector for one input example. The goal of multiple output linear regression is to simultaneously predict multiple target variables. In this case, the weights vector is replaced with a weight matrix with dimension $(m+1) \times K$ and K is the dimension of outputs and defined as $\mathbf{W} = [\mathbf{w}_1, ..., \mathbf{w}_K]$ where $\mathbf{w}_j = [1, w_1, ..., w_m]^{\top}$, $\forall j = 1, ..., K$. For $(\mathbf{x}_1, \mathbf{t}_1), ..., (\mathbf{x}_n, \mathbf{t}_n)$ where $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n]$, $\mathbf{x}_i = [x_1, ..., x_m]^{\top}$ and $\mathbf{t}_i$ is a vector with K dimension, this problem yields to multiple independent regression by using the same sets of basis functions for modeling all the elements of

target vectors as

$$\mathbf{Y}_{n\times K} = \mathbf{\Phi}_{n\times m+1}\mathbf{W}_{m+1\times K}, \tag{2.15}$$

where $\mathbf{\Phi}$ is an $n \times m+1$-dimensional matrix with elements $\mathbf{\Phi}_{ij} = \boldsymbol{\phi}_j(\mathbf{x}_i)$ and $\mathbf{Y} = [\mathbf{y}_1, ..., \mathbf{y}_n]$. $\mathbf{y}_i$ as the response is a vector of K outputs, and $\mathbf{W}$ is a $(m+1) \times K$ weight matrix. We suppose that we have Gaussian noise $\varepsilon_i \sim \mathcal{N}(0, \mathbf{I}_n)$ (Bishop, 2006).

By assuming that $\mathbf{\Phi}$ indicates the input matrix whose columns are $\mathbf{x}_1, ..., \mathbf{x}_n$ where $\mathbf{x}_i = [x_0, ..., x_m]^\top$, then in light of this situation, the set of observations $\mathbf{t}_1, ..., \mathbf{t}_n$ could be combined into a matrix $\mathbf{T}_{n\times K}$ where $\mathbf{t}_i = [t_1, ..., t_K]$.

Therefore, the conditional distribution of the response vector will be written in compact form as :

$$p(\mathbf{T}|\mathbf{\Phi}, \mathbf{W}, \sigma) \equiv \mathcal{N}(\mathbf{T}|\mathbf{\Phi}\mathbf{W}, \sigma^2\mathbf{I}_{n\times n}). \tag{2.16}$$

Therefore, log likelihood function could be written as

$$\mathcal{L}(\mathbf{W}, \sigma) = \log p(\mathbf{T}|\mathbf{\Phi}, \mathbf{W}, \sigma) = \frac{nK}{2}\log(\frac{\sigma^2}{2\pi}) - \frac{\sigma^2}{2}\sum_{i=1}^{n} \parallel \mathbf{t}_i - \mathbf{\Phi}_i\mathbf{W} \parallel^2 . \tag{2.17}$$

Maximizing the log likelihood function with respect to parameter matrix $\mathbf{W}$, gives

$$\mathbf{W}_{\mathrm{ML}} = (\mathbf{\Phi}^\top\mathbf{\Phi})^{-1}\mathbf{\Phi}^\top\mathbf{T}, \tag{2.18}$$

and considering the obtained result for each target $\mathbf{t}_k$ is defined as

$$\mathbf{w}_k = (\mathbf{\Phi}^\top\mathbf{\Phi})^{-1}\mathbf{\Phi}^\top\mathbf{t}_k, \tag{2.19}$$

where $\mathbf{t}_k = [t_{1k}, ..., t_{nk}]^\top$ (Bishop, 2006).

2.2 Generalized Linear Models (GLMs)

General linear regression models for a single variable output as described in previous section, are of the following form

$$\mathbb{E}(y_i) = \mu_i = \mathbf{x}_i^\top\mathbf{w},$$

where the expected value μ_i is a linear form of $m+1$ predictors taking the explanatory values $\mathbf{x}_i = [1, x_{i1}, \dots x_{im}]$ for the i^{th} row of input matrix $\mathbf{X}$. The parameter μ is the mean of

a Gaussian distribution with fixed variance σ and $\mathbf{w}$ represents the weight vector parameters. Furthermore

$$y_i \sim \mathcal{N}(\mu, \sigma^2), \tag{2.20}$$

where y_i is the independent response.

A linear regression model is made up of two main components : i) the linear structure and ii) the probability distribution. Clearly, in dealing with linear models, the Gaussian distribution plays a crucial role, where it assumes that the independent responses y_i come from a Gaussian distribution. There are numerous real-world scenarios where this condition is not met. A simple example is a binary response variable. Examples where the response variable indicates that the input picture either belongs to class cat or dog (i.e., 0 or 1). Moreover, there are several scenarios where the response variable is continuous, but it has distribution other than the Gaussian distribution, like reliability models.

Another situation where general linear regression models work inappropriately happens when the variance of response variable depends on the mean, which is a function of the mean. Generalized Linear Models (GLMs) extend the traditional linear model to address all of these situations.

GLMs allows the mean to depend on the explanatory variables using a link function, and the response variable to be from one of the exponential family of distributions (Myers *et al.*, 2012).

(Nelder et Baker, 1972) introduced an initial formulation of GLMs and showed that several most common linear regression models are members of the exponential family and could be treated in a general framework. They also discussed that the maximum likelihood estimates of these models could be calculated using the Iterated Reweighted Least Squares (IRLS) algorithm (Charnes *et al.*, 1976). The reason that GLMs are particularly restricted to members of exponential family of distributions is due to the fact that the numerical IRLS estimation algorithm only works within this family (Lindsey, 2000).

2.2.1 Exponential Family

Assume that there are a set of independent random response variables y_i and the probability (density) function of the form :

$$f(y_i; \eta_i) = r(y_i)s(\eta_i)\exp(y_i)u(\eta_i)] = \exp[t(y_i)u(\eta_i) + v(y_i) + w(\eta_i)], \tag{2.21}$$

where $\eta_i = \mathbf{x}_i^\top \mathbf{w}$ denotes the position where the distribution lies within the range of response values called the location parameter (Lindsey, 2000). Any distributions is said to belong to a (one-parameter) exponential family if the probability density function can be written in this way. One must keep in mind that the capital letter used for the random variables and a small letters indicate the observed values (Lindsey, 2000).

By letting $y_i = t(y_i)$ and $\theta = u(\eta_i)$, the canonical form for the random variable, the parameter, and the family could be calculated simply and the model could be written as :

$$f(y_i, \theta_i) = \exp\left\{y_i\theta_i - b(\theta_i) + c(y_i)\right\}, \tag{2.22}$$

where $b(.)$ is the natural parameter of the distribution, $c(.)$ is a specific known function and θ represents the location parameter. Other parameters in addition to the parameter of interest θ are considered as nuisance parameters and they would be supposed as parts of the b or c functions (Dobson et Barnett, 2008).

Therefore, the response y_i is a set of independent random variables with means, say μ_i, so that it could be written as :

$$y_i = \mu_i + \varepsilon_i. \tag{2.23}$$

2.2.2 Exponential Dispersion Family

An extension of the exponential family includes a fixed scale parameter for the distribution, σ. So, the equation in (2.21) will be become :

$$f(y_i; \theta_i, \sigma) = \exp\left\{\frac{y_i\theta_i - b(\theta_i)}{a_i(\sigma)} + c(y_i, \sigma)\right\}, \tag{2.24}$$

where θ_i is the canonical form of the location parameter and it is a function of the mean, μ_i and σ is called scale parameter. $a(.)$ is a particular function which is often appeared as $a(\sigma) = \frac{\sigma}{p_i}$, where p_i is a fixed constant, usually 1 (Myers *et al.*, 2012; Agresti et Kateri, 2011). One of the most eminent examples of the exponential family is Normal distribution. The probability density function for a random variable y with mean μ and variance σ^2 is expressed as :

$$f(y_i; \mu_i, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(y_i - \mu_i)^2}{2\sigma^2}\right\}$$

where μ_i denotes the location parameter and it is the parameter of interest and σ^2 is consi-

dered as a nuisance parameter. As a result we have :

$$= \exp\left\{\left(y_i\mu_i - \frac{\mu_i^2}{2}\right)\frac{1}{\sigma^2} - \frac{y_i^2}{2\sigma^2} - \frac{1}{2}\log(2\pi\sigma^2)\right\} \tag{2.25}$$

the above probability density function is a continuous distribution which is of the form of exponential family with $\theta = \mu$ as a location parameter and $a(\sigma) = \sigma$. The other terms are

$$b(\theta_i) = \frac{\theta_i^2}{2}$$

and

$$c(y_i, \sigma) = -\frac{1}{2}\left\{\frac{y_i^2}{\sigma} + \log(2\pi\sigma)\right\}.$$

The Normal distribution is one of the best known special case of the exponential distributions which is widely used to model continuous data that have a symmetric distribution (bell-shaped). The most important reason of its popularity is because it approximates many natural phenomena so well. The second reason is that if we have a data that are not Normally distributed (e.g., the distribution is skewed) and take sufficiently large random samples from the data with replacement, then the distribution of the sample mean will approximate a Normal distribution. This result is stated and proved in details in the Central Limit Theorem. Moreover, if continuous data are not Normally distributed it would be worthwhile trying to apply a log or square transformation, like $\log(y)$ or $\sqrt{y}$, so that the transformed data are approximately Normal. The form of exponential family also belong to the exponential dispersion family, with $a_i(\phi) = 1$ (Dobson et Barnett, 2008).

2.2.3 Mean and Variance

There is a special relationship between the mean and the variance for the distributions belonging to the exponential (dispersion) families. The variance of density depends on the scale parameter and on the mean that could be proven in the following way (Lindsey, 2000; Basu, 1955).

Lemma 2.2.1. *Suppose that the likelihood function for an observation is expressed as :*

$$\mathcal{L}(\theta_i, \phi; y_i) = f(y_i; \theta_i, \sigma), \tag{2.26}$$

Now, the first derivative of the logarithm of likelihood, called the score function, takes the

form of :

$$U_i = \frac{\partial \log \{\mathcal{L}(\theta_i, \sigma; y_i)\}}{\partial \theta_i}. \tag{2.27}$$

The maximum likelihood estimation as a solution of the above equation will be obtained by computing the score function for the whole set of observations and setting that to zero. It is easy to see that

$$\mathbb{E}[U_i] = 0 \tag{2.28}$$

and

$$\mathbb{V}[U_i] = \mathbb{E}[U_i^2]$$

$$= \mathbb{E}\left\{-\frac{\partial U_i}{\partial \theta_i}\right\}, \tag{2.29}$$

under mild regularity conditions for exponential family (Lindsey, 2000; Basu, 1955).

Proof. According to the equation (2.24), the log likelihood for exponential dispersion families could be written as

$$\ell(\theta_i, \phi; y_i)] = \log \{\mathcal{L}(\theta_i, \sigma; y_i)\} = \frac{y_i\theta_i - b(\theta_i)}{a_i(\sigma)} + c(y_i, \sigma). \tag{2.30}$$

Now, as a result of (2.26), we have

$$U_i = \frac{\partial \ell}{\partial \theta_i} = \frac{y_i - \frac{\partial b(\theta_i)}{\partial \theta_i}}{a_i(\sigma)}. \tag{2.31}$$

On the other hand from (2.27) we have

$$\mathbb{E}[y_i] - \frac{\partial b(\theta_i)}{\partial \theta} = \mu_i, \tag{2.32}$$

and from (2.31) it could be inferred that

$$U_i' = -\frac{\frac{\partial^2 b(\theta_i)}{\partial \theta^2}}{a_i(\sigma)}. \tag{2.33}$$

These all yield from (2.27), (2.30) and (2.31) to the following

$$\mathbb{V}(U_i) = \frac{\mathbb{V}[y_i]}{a_i^2(\sigma)}$$

$$= \frac{\frac{\partial^2 b(\theta_i)}{\partial \theta^2}}{a_i(\sigma)} \tag{2.34}$$

so that

$$\mathbb{V}(y_i) = \frac{\partial^2 b(\theta_i)}{\partial \theta^2} a_i(\sigma). \tag{2.35}$$

Suppose that $\frac{\partial^2 b(\theta_i)}{\partial \theta^2} = \sigma_i^2$ called the variance function which obviously depends on the mean μ_i (or θ_i). Therefore :

$$\mathbb{V}[y_i] = a_i(\sigma)\sigma_i^2$$

$$= \frac{\sigma \sigma_i^2}{p_i}, \tag{2.36}$$

that is only a product of the dispersion σ and a function of the μ. Notice that in above equations θ_i is the parameter of interest while σ is considered as a nuisance parameter. The variance function and $b(\theta_i)$ of a distribution member of the exponential families are uniquely identified (Lindsey, 2000). Variance function for some prominent members of the exponential distribution families are indicated in Table (2.1).

2.2.4 Linear Structure

The linear regression model could be written as a linear predictor. In this case, the canonical location parameter is expressed as a linear function of other parameters and takes the

Distribution	Variance Function
Poisson	$\mu = \mathrm{e}^{\theta}$
Binomial	$n\pi(1-\pi) = n\mathrm{e}^{\theta}/(1+\mathrm{e}^{\theta})^2$
Normal	1
Gamma	$\mu^2 = (-1/\theta)^2$
Inverse Gaussian	$\mu^3 = (-2/\theta)^{\frac{3}{2}}$

Table 2.1 Variance function for different distributions (Lindsey, 2000)

following form :

$$\theta_i(\mu_i) = \sum_{j=0}^{m} x_{ij} w_j = \mathbf{x}_i^\top \mathbf{w},$$

$$\boldsymbol{\theta} = \mathbf{X}^\top \mathbf{w}, \tag{2.37}$$

where $\mathbf{w}$ is a vector of $m+1$ parameters, the input matrix $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n]$ is a set of explanatory variables where $\mathbf{x}_i = [x_0, ..., x_m]^\top$, and $\boldsymbol{\theta} = \mathbf{X}^\top \mathbf{w}$ is the linear structure. The corresponding link function is called canonical if $\theta_i = \eta_i \ \ \forall i = 1, ..., n$. Note that vector $\mathbf{x}_i$ is augmented by 1 and w_0 is bias parameter. In (2.37), θ_i is indicated explicitly to be a function of the mean, something that was implicit in previous sections. This strictly linearity regarding the parameters, but not essentially the explanatory variables, could be further generalized by using a monotonic transformation on the mean, with $g(\boldsymbol{\mu})$. The transformed mean follows a linear model, so that

$$g(\boldsymbol{\mu}) = \boldsymbol{\eta} = \boldsymbol{\theta} = \mathbf{X}^\top \mathbf{w}, \tag{2.38}$$

and $\boldsymbol{\eta}$ is called the linear predictor. In GLMs, instead of modeling the mean, we will focus on a one-to-one continuous differentiable transformation of mean (Lindsey, 2000).

2.2.5 GLMs Components

Generalized linear models include three main components : 1) response distribution, 2) linear predictor and 3) link function. Let us look at them closely.

1) Response distribution (error structure) : specifies a probability distribution of the response variable y_i with mean μ_i which we assume that it is in the exponential family with a constant scale parameter and it is appeared in canonical form of (2.24). This is sometimes called a Noise Model or Error Structure.

2) Linear predictor : the linear predictor is a quantity that identifies the set of explanatory variables of the model and specifies more specifically the coefficients of the linear combination which is always done via

$$\boldsymbol{\eta} = \mathbf{X}^\top \mathbf{w}, \tag{2.39}$$

where $\mathbf{X}^\top \mathbf{w}$ is the linear structure (model) and says how the location of the response variables relates to explanatory variables and $\boldsymbol{\eta}$ is called the linear predictor (Rodriguez, 2012). In other words, the linear predictor characterizes the GLMs model and the model will be found via a link function.

3) Link function : provides the relation between random response variables and linear struc-

ture so that

$$\eta_i = g(\mu_i) \quad , \quad \forall i = 1, 2, ..., n.$$

It describes how the expected value of the response links to the linear predictor of explanatory variables :

$$\mu_i = \mathbb{E}(y_i) = g^{-1}(\eta_i) = g^{-1}(\mathbf{x}_i^\top \mathbf{w}).$$

So far, we have studied Normal distribution that could be described as an example of a GLMs model with Normal errors and identity link, where

$$\eta_i = \mu_i. \tag{2.40}$$

One may use a one-to-one continuous differentiable transformation of the mean called $g(\boldsymbol{\mu})$ for defining the link predictor. The transformation function, $g(\boldsymbol{\mu})$, is often called the link function and must be monotonic and differentiable (Myers *et al.*, 2012).

Since the link function is one-to-one, the inverse function of $g(.)$ is thus obtained by :

$$\mu_i = g^{-1}(\mathbf{x}_i^\top \mathbf{w}). \tag{2.41}$$

It should be noted that the expected value of y_i, μ_i is transformed in the above equation. Other examples of link functions are identity, log, logit and probit functions. When $\eta_i = \theta_i$ is employed then, it says that η_i is a canonical link, e.g. the identity function is considered as the canonical link for the Normal distribution (Rodriguez, 2012). Table (2.2) demonstrates the canonical links for the most prominent exponential distribution mostly used with GLMs (Myers *et al.*, 2012).

When the response distribution belongs to the exponential dispersion family with a constant

Table 2.2 Canonical Link function for different distributions from (Myers *et al.*, 2012)

Distribution	Link Function	Canonical Link Function
Poisson	log	$\eta_i = \log(\mu_i)$
Binomial	logit	$\eta_i = \log(\frac{\pi_i}{1-\pi_i}) = \log(\frac{\mu_i}{n_i-\mu_i})$
Normal	identity	$\eta_i = \mu_i$
Gamma	reciprocal	$\eta_i = \frac{1}{\mu_i}$
Inverse Gaussian	reciprocal^2	$\eta_i = \frac{1}{\mu_i^2}$

scale parameter, then by employing the canonical link function, all unknown parameters of the linear structure have sufficient statistics (Lindsey, 2000). In this situation, the link function simplifies the parameter estimation process while the model is working with a numerical estimation method such as the Iteratively Reweighted Least Squares (IRLS) algorithm (Lindsey, 2000). One must keep in mind that the link function η_i, does not transform the data, but the population on the mean (Myers *et al.*, 2012) and choosing an inappropriate link function might lead to considerable issues while fitting a GLMs model.

2.2.6 Maximum Likelihood Estimates

This section is about parameter estimation for generalized linear models using maximum likelihood. Although explicit mathematical equations can be found for estimators in some particular cases (e.g. closed-form functions), numerical methods based on a form of IRLS are often required. We first study briefly the least squares and weighted least squares methods and then describe the MLE for GLMs using an iterative variant of weighted least squares called Iteratively Reweighted Least Squares (IRLS).

Least squares : the simplest form of the method of least squares in linear regression consists of finding the estimator $\hat{\mathbf{w}}$ that maximizes the log-likelihood which is equivalent to minimizing the sum of squares of the differences between y_i's and their expected values as

$$\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{z}, \tag{2.42}$$

where z_i is called as the working dependent variable.

Weighted least squares : in the cases wherein the variability of the error is unequal across the predictions, a standard method is used to compensate this nonconstant error variance by inserting a diagonal matrix $\mathbf{\Omega}$ in estimating the parameter $\mathbf{w}$ such that the observed heteroscedasticity will be alleviated. The $\mathbf{\Omega}$ matrix could be calculated by taking the error variance of the i^{th} response at trail estimation as described in (2.34) and assigns its inverse to the i^{th} entry on diagonal $\mathbf{\Omega}$ matrix. The main motivation in this case is that large error variances will be mitigated by multiplication of the reciprocal (Gill, 2000).

To further describe the weighted regression, recall the standard form of the linear regression as

$$y_i = \mathbf{x}_i^\top \mathbf{w} + \varepsilon_i.$$

Premultiplying each term in the above equation by the square root of the diagonal values,

which is produced from a Cholesky factorization, yields

$$\boldsymbol{\Omega}^{\frac{1}{2}} y_i = \boldsymbol{\Omega}^{\frac{1}{2}} \mathbf{x}_i \mathbf{w} + \boldsymbol{\Omega}^{\frac{1}{2}} \varepsilon_i. \tag{2.43}$$

Thus, under assumption of having heteroscedasticity in error term as $\varepsilon \sim (\mathbf{0}, \sigma^2 \mathbf{V})$, it could be induced from (2.43) that $\varepsilon \sim (\mathbf{0}, \sigma^2 \boldsymbol{\Omega} \mathbf{V}) = (\mathbf{0}, \sigma^2)$ and therefore, heteroscedasticity is eliminated. As a result, the parameter estimates will be performed by minimizing the $(\mathbf{y} - \mathbf{X}^\top \mathbf{w})^\top \boldsymbol{\Omega}^{-1} (\mathbf{z} - \mathbf{X}^\top \mathbf{w})$ is rather than $(\mathbf{y} - \mathbf{X}^\top \mathbf{w})^\top (\mathbf{z} - \mathbf{X}^\top \mathbf{w})$ in ordinary least squares. The estimator for weighted least squares takes the form

$$\hat{\mathbf{w}} = (\mathbf{X}^\top \boldsymbol{\Omega} \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{\Omega} \mathbf{z}, \tag{2.44}$$

where $\mathbf{X}$ is the model matrix, $\boldsymbol{\Omega}$ is a diagonal weight matrix with entries ω_i and $\mathbf{z}$ is the vector of responses with entries z_i (Gill, 2000).

Iteratively Reweighted Least Square (IRLS) : when the individual variances in reciprocal diagonal matrix $\boldsymbol{\Omega}$ are a function of the mean, $v_i = f(\mu_i) = f\{\mathbb{E}(y_i)\}$, and when the $f(.)$ is known, the estimation algorithm could be still straightforward. A solution to solve the problem in this circumstance is to iteratively estimate the weight matrix $\boldsymbol{\Omega}$, by improving the estimates in each trial.

Given $\boldsymbol{\eta} = g^{-1}(\mathbf{X}^\top \mathbf{w})$, the estimator $\hat{\mathbf{w}}$ could provide an estimate for mean and vice versa. The IRLS algorithm could estimate these variables through improving the weight matrix in an iterative manner as follows (Gill, 2000) :

1. Initialize the diagonal weight matrix $\boldsymbol{\Omega}$, generally equals to one ($\omega_i = 1$).
2. Estimate $\mathbf{w}$ using weighted least squares in (2.44) with the current weight values.
3. Update the weight matrix by the new estimates for the mean.
4. Repeat steps 2 and 3 until the iterative process converges in such a way that the estimate parameter of $\hat{\mathbf{w}}$ changes by less than a prior known threshold.

Suppose that the method of maximum likelihood applied to GLMs with a canonical link of the form $\eta_i = g(\mu_i) = \mathbf{x_i}^\top \mathbf{w}$. Therefore, the log-likelihood function will take the form :

$$\ell(\mathbf{w}; \mathbf{y}) = \log \mathcal{L}(\mathbf{w}; \mathbf{y}) = \sum_{i=1}^{n} \left[\frac{\{y_i \theta_i - b(\theta_i)\}}{a(\sigma)} + c(y_i, \sigma) \right], \tag{2.45}$$

where $\boldsymbol{\eta} = g(\boldsymbol{\mu_i}) = \mathbf{X}^\top \mathbf{w}$ and $\mathbb{V}(\mathbf{y}_i)$ is calculated by (2.34). The derivative of the log-

likelihood easily seen to be

$$\frac{\partial \ell}{\partial \mathbf{w}} = \frac{\partial \ell}{\partial \theta_i}\frac{\partial \theta_i}{\partial \mathbf{w}}$$

$$= \sum_{i=1}^{n} \frac{1}{a(\sigma)} \left\{ y_i - \frac{\partial b(\theta_i)}{\partial \theta_i} \right\} x_i$$

$$\sum_{i=1}^{n} \frac{1}{a(\sigma)} (y_i - \mu_i) x_i \tag{2.46}$$

Using the canonical link, the maximum likelihood estimates for the parameters are obtained by solving the following score equations for $\mathbf{w}$.

$$\mathrm{U}(\mathbf{w}) = \sum_{i=1}^{n} \frac{1}{a(\sigma)} (y_i - \mu_i) x_i = 0, \tag{2.47}$$

where in most cases, $a(\sigma)$ is considered as known prior parameters. Hence, equating this to zero to obtain the solution gives us the equation below in a matrix form :

$$\mathrm{U}(\mathbf{w}) = \mathbf{X}^\top (\mathbf{y} - \boldsymbol{\mu}) = 0, \tag{2.48}$$

where $\boldsymbol{\mu} = [\mu_1, \mu_2, ..., \mu_n]$ and $\mu_i = \mathbf{x}_i^\top \mathbf{w}$. Note that μ_i is a function of $\eta_i = \mathbf{x}_i^\top \mathbf{w}$ and is not necessarily a linear function. Thus, there is not a closed-form solution for $\mathbf{w}$. This system of equations is called the maximum likelihood score equations which is operative for GLMs with a canonical link. A Taylor series approach could be applied to calculate the following approximation regarding the $\mathbf{w}^*$

$$\boldsymbol{\mu} \sim \boldsymbol{\mu}^* + \boldsymbol{\Omega}(\mathbf{X}^\top \mathbf{w} - \mathbf{X}^\top \mathbf{w}^*), \tag{2.49}$$

where $\boldsymbol{\mu}^* = g^{-1}(\mathbf{X}^\top \mathbf{w}^*)$ (Hastie *et al.*, 2009).

Therefore, the following linear approximation will be found for the score for $\mathbf{w}$:

$$\frac{\partial \ell}{\partial \mathbf{w}} = \mathbf{X}^\top \boldsymbol{\Omega}(\mathbf{z} - \mathbf{X}^\top \mathbf{w}), \tag{2.50}$$

where $\mathbf{z} = \mathbf{X}^\top \mathbf{w}^* + \boldsymbol{\Omega}^{-1}(\mathbf{y} - \boldsymbol{\mu}^*)$ is known as the working dependent variable. This approximation is calculated based on $\mathbf{w}^*$ or, equivalently, μ^* and are considered as constants in the

equation. Solving this approximation gives the maximum likelihood estimator $\hat{\mathbf{w}}$ as follows

$$\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{\Omega} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{\Omega} \mathbf{z},$$

where $\mathbf{\Omega}$ plays the same role as the diagonal weighted matrix in weighted least squares. One must keep in mind that for the canonical link, $\mathbf{\Omega}$ matrix is obtained by the mean-variance relationship, and it affects in the variability of $\mathbf{w}$ as well. The maximum likelihood solution for $\hat{\mathbf{w}}$ could be estimated through progressively improving weights with IRLS as described before (Myers *et al.*, 2012).

Suppose that we have a trial estimate of the parameters $\hat{\mathbf{w}}$. Then, the estimated linear predictor $\hat{\boldsymbol{\eta}}_i = \mathbf{x}_i^\top \hat{\mathbf{w}}$ could be obtained and used to evaluate the fitted values $\hat{\boldsymbol{\mu}}_i = g^{-1}(\hat{\boldsymbol{\eta}}_i)$, consequently. Having the $\hat{\boldsymbol{\eta}}_i$ and $\hat{\boldsymbol{\mu}}_i$ quantities, the term z_i as the working dependent variable could be identified as :

$$z_i = \hat{\eta}_i + (y_i - \hat{\mu}_i) \frac{\partial \eta_i}{\partial \mu_i}, \tag{2.51}$$

where the term $\frac{\partial \eta_i}{\partial \mu_i}$ denotes the derivative of the link function calculated at the current iteration (the trial estimate). Furthermore, the iterative weights turns out to be :

$$\omega_i = \frac{p_i}{b''(\theta_i)(\frac{\partial \eta_i}{\mu_i})^2}, \tag{2.52}$$

with the assumption that $a(\sigma) = \frac{\sigma}{p_i}$ and $b''(\theta_i)$ is the second derivative of $b''(\theta_i)$ with respect to the θ_i calculated at the trial estimate. Finally, the estimate of the parameter $\mathbf{w}$ through regressing the working dependent variable z_i on the predictors x_i through employing the weights w_i are acquired. The updated procedure will be repeated until the iterative process converges to a threshold. Hence the estimates for the parameter $\mathbf{w}$ can be simply obtained using an IRLS algorithm (Charnes *et al.*, 1976).

In (McCullagh, 1984), authors proved that this procedure is equivalent to Fisher scoring and yields to maximum likelihood estimates. Remember that for the data from Normal distribution with identity link $\eta_i = \mu_i$, the derivative equals

$$\frac{\partial \eta_i}{\partial \mu_i} = 1,$$

and therefore, the weights $\mathbf{\Omega}$ are constant and no iteration is needed.

2.2.7 Logistic Regression

In this section [2], we consider generalized linear models for the situation where the outcome variables take on only binary scale, 0 and 1. For example, the responses may be either success or failure, as generic terms of the two categories (Myers *et al.*, 2012).

Suppose that the model takes the form :

$$y_i = \mathbf{x}_i^\top \mathbf{w} + \varepsilon_i$$

where $\mathbf{x}_i = [1, x_{i1}, x_{i2}, ..., x_{im}]^\top$, $\mathbf{w} = [w_0, w_1, ..., w_m]^\top$ and the responsible variable y_i takes the value 1 if the outcome is a success or 0 if the outcome is a failure. Hence, it is supposed that y_i is a Bernoulli random variable with probabilities $p(y_i = 1) = \pi_i$ and $p(y_i = 0) = 1 - \pi_i$ where π_i is the parameter of interest. The distribution of y_i is written as :

$$p\{y_i\} = \pi_i^{y_i}(1 - \pi_i)^{(1-y_i)} \quad , \quad y_i \in \{0, 1\},$$

and it could be rewritten as :

$$\begin{aligned} p\{y_i\} &= \pi_i^{y_i}(1 - \pi_i)^{(1-y_i)} \quad , \quad y_i \in \{0, 1\} \\ &= \exp\left\{ y_i \log\left(\frac{\pi_i}{1 - \pi_i}\right) + \log(1 - \pi_i) \right\}, \end{aligned}$$

which is obviously a member of the exponential family, see (2.22).

If there are n such independent random variables $y_0, ..., y_n$ therefore, the joint probability can be written as

$$\prod_{i=0}^{n} \pi_i^{y_i}(1 - \pi_i)^{(1-y_i)} = \exp\left\{ \sum_{i=0}^{n} y_i \log\left(\frac{\pi_i}{1 - \pi_i}\right) + \sum_{i=0}^{n} \log(1 - \pi_i) \right\}. \tag{2.53}$$

The expected value of the response variable y_i, mean μ_i, will be

$$\mu_i = \mathbb{E}(y_i) = \pi_i, \tag{2.54}$$

2. This section is heavily based on notes and notations inherited from (Myers *et al.*, 2012).

Furthermore, the variance of the response is not constant :

$$\mathbb{V}(y_i) = \sigma^2_{y_i} = \mathbb{E}\{y_i - \mathbb{E}(y_i)\}^2 = (1-\pi_i)^2\pi_i + (0-\pi_i^2)(1-\pi_i) = \pi_i(1-\pi_i)$$

$$\mathbb{V}(y_i) = \sigma^2_{y_i} = \mathbb{E}(y_i)[1-\mathbb{E}(y_i)] = \pi_i(1-\pi_i). \tag{2.55}$$

The main crucial problem with above linear probability model is that although π_i is a probability restricted to the interval $[0,1]$, it would be possible that the fitted values ${\mathbf{x}_i}^\top\mathbf{w}$ might be less than zero or greater than one (Myers *et al.*, 2012).

To ensure that π_i is always bounded between 0 and 1, it would be modeled by a transformation function of the π_i, says $g(\pi_i)$, and model the transformation as a linear function of $\mathbf{x}_i^\top$s as :

$$\eta_i = g(\pi_i) = {\mathbf{x}_i}^\top\mathbf{w}, \tag{2.56}$$

where η_i is the linear predictor and $g(.)$ is the link function. Then, the inverse transformation is the cumulative distribution function (c.d.f.) of the random variables as :

$$\pi_i = g^{-1}(\eta_i), \tag{2.57}$$

which enables us to return from logits to probabilities. One commonly-used choice of $g(.)$ in this context is to consider the canonical link of the binomial distribution, θ_i, as the link function in such a way that :

$$\eta_i = g(\pi_i) = \theta_i = \log\left(\frac{\pi_i}{1-\pi_i}\right). \tag{2.58}$$

The ratio $\frac{\pi_i}{1-\pi_i}$ is often called the odds and the logarithm the odds is called the logit transformation (or log-odds). The logit transformation maps π_i which is restricted to the interval $[0,1]$, to the real line. Solving for π_i in the above equation gives the logistic distribution as follows :

$$\pi_i = g^{-1}(\eta_i) = \frac{\exp(\eta_i)}{1+\exp(\eta_i)} = \text{logistic}(\eta_i). \tag{2.59}$$

The right-hand side of equation (2.59), the logistic function, is a nonlinear function of the predictors. To express the effect on the probability when increasing a predictor by one unit, while the other variables is kept fixed, an approximation answer could be obtained by taking

the first derivative of probability with respect to predictors as follows

$$\frac{\partial \pi_i}{\partial x_{ij}} = w_j \pi_i (1 - \pi_i). \tag{2.60}$$

This expression is only applicable for continuous predictors. As understood from (2.60), the effect of the j^{th} predictor on the π_i depends on regressor w_j and the probability itself (Agresti et Kateri, 2011).

There are other popular alternatives to the logit function, namely probit and complimentary log-log, as link functions that result in the cumulative standard Normal and Gumbel distribution functions, respectively. The probit link is defined as

$$\eta_i = \Phi^{-1}(\pi_i), \tag{2.61}$$

where Φ denotes the cumulative distribution function and the complimentary log-log link is

$$\eta_i = \log\left\{\log(1 - \pi_i)\right\}, \tag{2.62}$$

where $\pi_i = g^{-1}(\eta_i) = \exp\{-\exp(\eta_i)\}$ is the Gumbel distribution. Logit, probit and complementary log-log are all monotonic and differentiable functions (Agresti et Kateri, 2011).

2.2.8 Maximum Likelihood Estimates in Logistic Regression

The logistic regression is considered under the framework of generalized linear models that have binomial error and logit link function. Thus, according to the general theory developed for generalized linear models, the parameter estimates could be simply obtained. In this section an underlying procedure to compute the estimates will be summarized briefly. Typically, logistic regression model is of the form

$$y_i \sim \text{Bernoulli}(\pi_i),$$

where y_i's are independent from each other and y_i takes on only two possible values, 0 and 1. Since the response variables are independent, the likelihood function for n independent bernoulli observations equals to a product of the densities. As a result, the likelihood function

has the form (Myers *et al.*, 2012)

$$\mathcal{L}(\mathbf{w}, y_1, ..., y_n) = \prod_{i=1}^{n} f_i(y_i) = \prod_{i=1}^{n} \pi_i^{y_i}(1-\pi_i)^{1-y_i}. \tag{2.63}$$

Since the logarithm is a monotonically increasing function of its argument, maximizing the log likelihood, ℓ, is equivalent to maximizing the likelihood itself. Thus the log-likelihood is

$$\ell(\mathbf{w}, y_1, ..., y_n) = \log \mathcal{L}(\mathbf{w}, y_1, ..., y_n) = \log \prod_{i=1}^{n} f_i(y_i)$$

$$= \sum_{i=1}^{n} \left[y_i \log\left(\frac{\pi_i}{1-\pi_i}\right) + \sum_{i=1}^{n} \log(1-\pi_i) \right]. \tag{2.64}$$

Since from (2.53) we have $1-\pi_i = [1+\exp\{\mathbf{x}_i{}^\top \mathbf{w}\}]^{-1}$ and $\eta_i = \log(\frac{\pi_i}{1-\pi_i}) = \mathbf{x}_i{}^\top \mathbf{w}$, the log-likelihood take the form :

$$\ell(\mathbf{w}) = \log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{n} y_i \mathbf{x}_i{}^\top \mathbf{w} - \log[1+\exp(\mathbf{x}_i{}^\top \mathbf{w})]$$

$$= \mathbf{w}^\top \mathbf{X}\mathbf{y} - \sum_{i=1}^{n} \log\left\{1+\exp(\mathbf{x}_i{}^\top \mathbf{w})\right\}. \tag{2.65}$$

Now, suppose that y_i is the number of 1's at the i^{th} observation . So, the kernel of the log-likelihood is of the form :

$$\ell(\mathbf{w}) = \mathbf{w}^\top \mathbf{X}\mathbf{y} - \sum_{i=1}^{n} \log\left\{1+\exp(\mathbf{x}_i{}^\top \mathbf{w})\right\}, \tag{2.66}$$

where $\mathbf{X}$ is the input matrix and $\mathbf{y}$ is the response vector. Taking the derivative of the log-likelihood with respect to $\mathbf{w}$, gives us :

$$\frac{\partial \ell}{\partial \mathbf{w}} = \mathbf{X}\mathbf{y} - \sum_{i=1}^{n} \left\{ \frac{1}{1+\exp(\mathbf{x}_i{}^\top \mathbf{w})} \right\} \exp(\mathbf{x}_i{}^\top \mathbf{w}) \mathbf{x}_i^\top. \tag{2.67}$$

Since

$$\frac{\exp(\mathbf{x}_i{}^\top \mathbf{w})}{1+\exp(\mathbf{x}_i{}^\top \mathbf{w})} = \frac{1}{1+\exp(-\mathbf{x}_i{}^\top \mathbf{w})} = \pi_i, \tag{2.68}$$

thus

$$\frac{\partial \ell}{\partial \mathbf{w}} = \mathbf{X}\mathbf{y} - \sum_{i=1}^{n} \pi_i \mathbf{x}_i^{\top}. \tag{2.69}$$

Moreover, since π_i denotes the mean of the Bernoulli random variable, the right-hand side of the equation could be expressed in the form of following matrix (Myers *et al.*, 2012)

$$\mathbf{s} = \mathbf{X}(\mathbf{y} - \boldsymbol{\mu}),$$

where

$$\boldsymbol{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \end{bmatrix}. \tag{2.70}$$

Consequently, the solutions are the maximum likelihood estimation (MLE) of the following equation called score equation :

$$\mathbf{X}(\mathbf{y} - \boldsymbol{\mu}) = \mathbf{0}. \tag{2.71}$$

Note that the $\mathbf{w}$ parameters in $\boldsymbol{\mu}$ and the entries in the $\boldsymbol{\mu}$ vector are nonlinear based on the model

$$\mu_i = \frac{1}{1 + \exp(-\mathbf{x}_i{}^{\top}\mathbf{w})} \quad , \quad \forall i = 1, ..., n.$$

As a result, an IRLS approach is applied to solve the equation (2.71) as discussed similarly for GLMs (Myers *et al.*, 2012).

Lemma 2.2.2. *An IRLS method could be used to solve the equation* $\mathbf{s} = \mathbf{X}(\mathbf{y} - \boldsymbol{\mu}) = \mathbf{0}$.

Proof. Similarly, we have

$$\min \mathbf{s} = \min_{\mathbf{w}} \sum_{i=1}^{n} \left[\frac{(y_i - \mu_i)^2}{\sigma_i^2} \right], \tag{2.72}$$

with fixed variance σ_i^2. Consequently, taking the derivative of the numerator of $\mathbf{s}$ gives us :

$$\frac{\partial \mathbf{s}}{\partial \mathbf{w}} = -2 \left\{ \frac{\sum_{i=1}^{n} (y_i - \mu_i)}{\sigma_i^2} \right\} (\frac{\partial \mu_i}{\partial \mathbf{w}}), \tag{2.73}$$

where $\frac{\partial \mu_i}{\partial \mathbf{w}} = \pi_i(1 - \pi_i)\mathbf{x}_i = \sigma_i^2 \mathbf{x}_i$. Hence, the solution obtained by the minimization of the

weighted residual sum of squares with fixed variance σ_i^2 takes the form

$$\frac{\partial \mathbf{s}}{\partial \mathbf{w}} = \sum_{i=1}^{n}(y_i - \mu_i)\mathbf{x}_i = 0$$

$$\mathbf{X}(\mathbf{y} - \boldsymbol{\mu}) = \mathbf{0} \tag{2.74}$$

which is identical to the score solution of MLE produced by (2.63) and produces the estimator values of the MLE (Myers *et al.*, 2012). Given a parameter of the estimate iteration, $\hat{\mathbf{w}}$, the linear predictor $\hat{\eta}_i = \mathbf{x}_i^\top \hat{\mathbf{w}}$ and the fitted values $\hat{\mu}_i = \text{logit}^{-1}(\eta_i)$ could be calculated. Having these quantities, the working dependent variable and the iterative weight could be found to estimate the parameters of logistic regression. Let the model is defined by

$$\eta_i = \text{logit}(\pi_i) = \log \frac{\pi_i}{1 - \pi_i}. \tag{2.75}$$

It is more convenient to show the link function as a function of μ_i as

$$\eta_i = \log \mu_i(1 - \mu_i) = \log(\mu_i) - \log(1 - \mu_i). \tag{2.76}$$

By taking the derivative of the η_i with respect to the μ_i, we have :

$$\frac{\partial \eta_i}{\partial \mu_i} = \frac{1}{\mu_i} + \frac{1}{1 - \mu_i} = \frac{1}{\mu_i(1 - \mu_i)} = \frac{1}{\pi_i(1 - \pi_i)}. \tag{2.77}$$

Thus, the working dependent variable will be as the general form below :

$$z_i = \eta_i + (y_i - \mu_i)\frac{\partial \eta_i}{\partial \mu_i}$$

$$= \eta_i + \frac{y_i - \pi_i}{\pi_i(1 - \pi_i)}, \tag{2.78}$$

and the iterative weight will take the form

$$w_i = \frac{1}{b''(\theta_i)(\frac{\partial \eta_i}{\partial \mu_i})^2}$$

$$= \frac{1}{\pi_i(1-\pi_i)}[\pi_i(1-\pi_i)]^2$$

$$= \pi_i(1-\pi_i). \tag{2.79}$$

Finally, the estimate of the parameter $\mathbf{w}$, through regressing the working dependent variables z_i, on the predictors x_i, through employing the weights ω_i are acquired as follows :

$$\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{\Omega} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{\Omega} \mathbf{z}, \tag{2.80}$$

where $\mathbf{X}$ is the model matrix, $\mathbf{\Omega}$ is a diagonal weight matrix with entries ω_i obtained by (2.52) and $\mathbf{z}$ is the vector of responses with entries z_i calculated via (2.51). The updated procedure will be repeated until convergence. Finally, the estimates for the parameter $\mathbf{w}$ of logistic regression is found using an IRLS algorithm.

Suppose that $\hat{\mathbf{w}}$ is the estimate of parameter $\mathbf{w}$ of the logistic regression model fitted by IRLS. If all the model assumptions are correct, then it could be shown asymptotically (Myers *et al.*, 2012)

$$\mathbb{E}(\hat{\mathbf{w}}) = \mathbf{w} \tag{2.81}$$

and

$$\mathbb{V}(\hat{\mathbf{w}}) = (\mathbf{X}^\top \hat{\mathbf{V}} \mathbf{X})^{-1}, \tag{2.82}$$

where $\hat{\mathbf{V}}$ is an $n \times n$ diagonal matrix including the estimated variance of each element on the main diagonal. Thus, the i^{th} entry on the diagonal of the $\hat{\mathbf{V}}$ would be :

$$v_{ii} = \hat{\pi}_i(1-\hat{\pi}_i). \tag{2.83}$$

As a result, the estimated value of the linear predictor has the form $\hat{\boldsymbol{\eta}}_i = \mathbf{x}_i^\top \hat{\mathbf{w}}$ and the fitted model for logistic regression is given by :

$$\hat{y}_i = \hat{\pi}_i = \frac{\exp(\hat{\eta}_i)}{1+\exp(\hat{\eta}_i)} = \frac{\exp(\mathbf{x}_i^\top \mathbf{w})}{1+\exp(\mathbf{x}_i^\top \mathbf{w})} = \frac{1}{1+\exp(-\mathbf{x}_i^\top \mathbf{w})}. \tag{2.84}$$

There is an interesting relationship between weighted least squares and score equation in (2.62) which is obtained by maximum likelihood (Myers *et al.*, 2012). Using the weighted

residual sum of squares obtained by

$$s = \sum_{i=1}^{n} \left\{ \frac{(y_i - \mu_i)^2}{\sigma_i^2} \right\} \tag{2.85}$$

where $\mu_i = \pi_i$ and σ_i^2 is the Bernoulli variance at the i^{th} observation with

$$\sigma_i^2 = \pi_i(1 - \pi_i) = \frac{\exp(-\mathbf{x}_i^\top \hat{\mathbf{w}})}{\left\{1 + \exp(-\mathbf{x}_i^\top \hat{\mathbf{w}})\right\}^2}. \tag{2.86}$$

CHAPTER 3 DEEP NEURAL NETWORKS

3.1 Perceptron

In the context of neural networks, the term perceptron refers to a network consisting one single neuron, initially presented by Rosenblatt in its most basic form (Rosenblatt, 1962; McCulloch et Pitts, 1943). The aim of the following section is to give a better understanding of perceptrons applied to binary classification. An introduction of the architecture of two neural networks related to this thesis, MLPs and CNNs, are described in sequel.

3.1.1 Perceptron Algorithm

The perceptron algorithm is a linear discriminant model for supervised learning of binary classification problems and belongs to the broad family of on-line learning algorithms (Cesa-Bianchi et Lugosi, 2006). Perceptrons play a key role in the evaluation of machine learning and neural network algorithms. The perceptron algorithm was first introduced by Frank Rosenblatt, an American psychologist, at the Cornell Aeronautical Laboratory in late 1950s (Rosenblatt, 1957). We also denominate the perceptron algorithm as the single-layer perceptron, the simplest feedforward neural network, to discern it from a so-called multilayer perceptron, which is used for more complicated and deep neural networks.

3.1.2 Model

Before we begin, one should note that the notation that is used in the perceptron and neural network sections are different in appearance with that used in the linear regression context. The perceptron model was one of the first artificial neural network models to build a generalized linear model in which it first maps a real-valued input vector $\mathbf{x} = [x_1, ..., x_m]^\top$ to a single binary value y of the form

$$y = g(\mathbf{w}^\top \mathbf{x}), \tag{3.1}$$

where the nonlinear function $g(.)$ is given by a step function of the form

$$g(x) = \begin{cases} -1 & x \geq b, \\ +1 & x \leq b, \end{cases} \tag{3.2}$$

where $\mathbf{w}$ is a real-valued weight vector $[w_1, ..., w_m]^\top$ and b is the threshold. Also, $\mathbf{w}^\top \mathbf{x}$ is the dot product of weight vector $\mathbf{w}$ and input $\mathbf{x}$ as

$$\mathbf{w}^\top \mathbf{x} = \sum_{j=1}^{m} w_j x_j, \tag{3.3}$$

where m is the number of input features. The perceptron outputs a non-zero value $+1$ only when the weighted sum exceeds the threshold. Otherwise, it assumes the deactivated value -1. Mostly, in the case of a binary classification problem, the value of $y \in (0, 1)$ is used to classify $\mathbf{x}$ as either a positive or a negative example. But, for the perceptron algorithm, however, it is more proper to use target values $y = +1$ for class positive and $y = -1$ for class negative, which makes the choice of the activation function harmonious (Bishop, 2006).

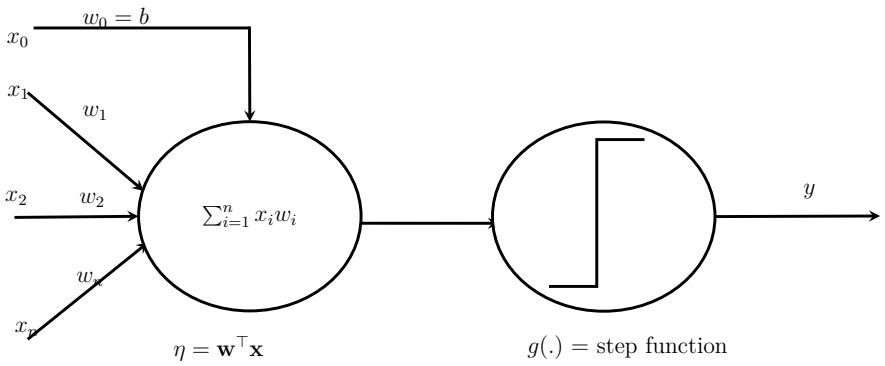


Figure 3.1 A simple model with step function

Models based on the $g(.)$, used by the perceptron in a linear model, is one of the most widely-used machine learning models. Albeit, in many cases they are learned in different ways how the original models were trained. The key idea of the function $g(.)$ appears simple and is elaborated in the name : the Heaviside step function which returns 1 if the input exceeds zero, and 0 for any negative input. We could consider a perceptron as a single artificial neuron which uses the Heaviside step function as the activation function. Suppose that X_0 and X_1 be two sets of points in an $m+1$-dimensional euclidean space. Then, X_0 and X_1 are linearly separable if there exists a real-valued weight vector $\mathbf{w} = [w_0, ..., w_m]^\top$, a real value b in such a way that each point $\mathbf{x} \in X_0$ satisfies $\sum_{j=0}^{m} w_j x_j \geq b$, where each point $\mathbf{x} \in X_1$ satisfies

$\sum_{j=0}^{m} w_j x_j < b$, and $\mathbf{x}$ is a real-valued vector $\mathbf{x} = [x_0, ..., x_m]^\top$.

Recall that $\sum_{j=0}^{m} w_j x_j \geq b$ and $\sum_{j=0}^{m} w_j x_j < b$ are two regions on the hyperplane which are separated by the line $\sum_{j=0}^{m} w_j x_j + b = 0$. Therefore, if we consider two vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ from two separate regions as two data points on a hyperplane, then the perceptron model could partition the input space into two separate halfspaces along a hyperplane and gives us the half-space on the plane that this points belongs to.

We call such regions so-called linearly separable regions as they are separated by a single line. If the value of parameter b is negative, then the weighted sum of inputs have to produce a positive value greater than $|b|$ to push the perceptron over the 0 threshold. Therefore, the bias parameter changes the spatial position but not the direction of the decision boundary. Figure 2.1. illustrates an example of a binary classification (Bishop, 2006).

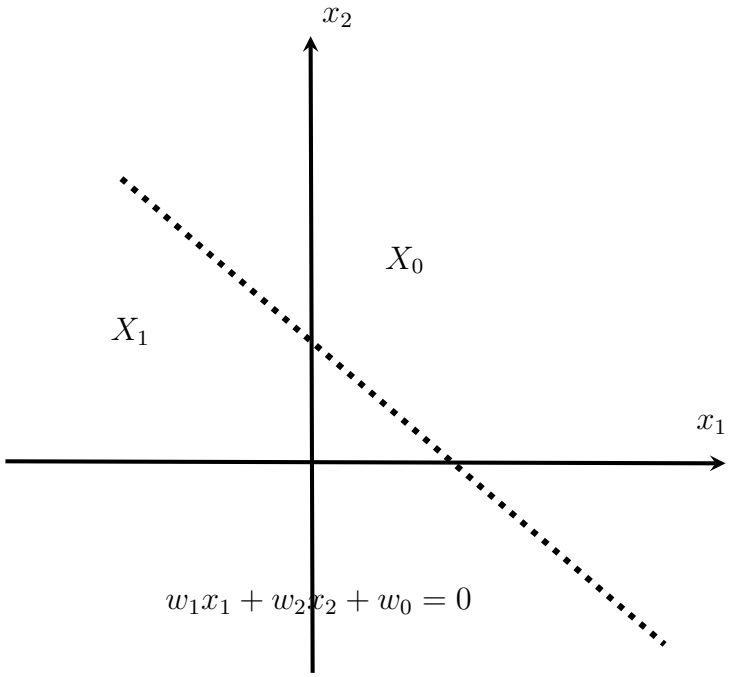


Figure 3.2 Demonstration of the hyperplane as decision boundary for a two-dimensional and binary classification problem

To simplify the representation, we aim at dealing with a perceptron model with threshold $b = 0$ due to the fact that linear separations are forced to go through the origin of the input space. Therefore, a widely-used method is to append an extra feature to input vector $\mathbf{x}$ and set it to 1 and the resulting $(m + 1)$ dimensional vector is called the extended input vector $\mathbf{x} = [1, x_1, x_2, ..., x_m]^\top$. Then, we aim at learning a $m + 1$ dimension weight vector $\mathbf{w}$ in this extended space and consider w_0 as the bias parameter. With the notation $\mathbf{w} =$

$([w_0, w_1, ..., w_m]^\top$, we have :

$$u = \mathbf{w}^\top \mathbf{x} = w_0 + \sum_{j=1}^{m} w_j x_j$$

$$= w_0 + \mathbf{w}^\top \mathbf{x}. \tag{3.4}$$

Hence, we consider the two perceptron models demonstrated in Figure 3.2 as equivalent models. The bias parameter (threshold) of the left perceptron model has been turned to the weight $-b$ of an additional input feature linked to the constant value 1, without loss of generality. We could interprete this extra weight linked to a constant as a bias parameter.

3.1.3 Basic Perceptron Learning Rule and Convergence

Suppose that we have the input matrix $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n]$ is a set of n input vectors where $\mathbf{x}_i = [x_0, ..., x_m]^\top$. Assume that we receive a $m+1$ dimensional input vector $\mathbf{x}$ at time-stamp t denoted by $\mathbf{x}(t) = [1, x_1(t), x_2(t), ..., x_m(t)]^\top$, and we define a $m+1$ dimensional weight vector $\mathbf{w}$, $\mathbf{w}(t) = [w_0(t), w_1(t), w_2(t), ..., w_m(t)]^\top$ where t represents the current time-step in the model training process and $\mathbf{x}_i$ is the observed example at step t. Correspondingly, the linear dot product $\mathbf{x}$ and $\mathbf{w}$ are written in the following form :

$$u(t) = \mathbf{w}^\top(t)\mathbf{x}(t) = w_0(t) + \sum_{j=1}^{m} w_j(t) x_j(t), \tag{3.5}$$

where, $w_0(t)$ interprets the bias parameter in time-step t. Considering a fixed t, the equation $\mathbf{w}^\top \mathbf{x} = 0$ defines a hyperplane as the decision boundary between two separate classes of inputs in an $m+1$ dimensional space. Remember the first assumption that our data originated from two sets of points in an $m+1$ dimensional euclidean space, where X_0 and X_1 are sufficiently linearly separable. It means that the input variables of the perceptron came from two linearly separable classes C_1 and C_2 where the union of C_1 and C_2 classes is the complete space. Hence, there is a weight vector $\mathbf{w}$ in such a way that we may say :

$$\mathbf{w}^\top(t)\mathbf{x}(t) \geq 0 \ \ , \ \forall \mathbf{x}(t) \in X_0 \, , \ (\mathbf{x}(t) \in C_1)$$

$$\mathbf{w}^\top(t)\mathbf{x}(t) < 0 \ \ , \ \forall \mathbf{x}(t) \in X_1 \, , \ (\mathbf{x}(t) \in C_2) \tag{3.6}$$

Therefore, given the subsets of training vectors from X_0 and X_1, the learning problem for

the perceptron model is then to find a weight vector $\mathbf{w}$ where the two above inequalities are satisfied.

The basic perceptron algorithm for training the weight vector $\mathbf{w}$ is now be formulated in the following form :

– If data $\mathbf{x}(t)$, with the training input vector at iteration t, is properly classified by the weight vector $\mathbf{w}(t)$ calculated at time-step t, then accordingly, there is no need to adapt the weight vector of the perceptron model :

$$\begin{cases} \mathbf{w}(t+1) = \mathbf{w}(t) & \text{if } \mathbf{w}^\top(t)\mathbf{x}(t) \geq 0 \text{ and } \mathbf{x}(t) \in C_1, \\ \mathbf{w}(t+1) = \mathbf{w}(t) & \text{if } \mathbf{w}^\top(t)\mathbf{x}(t) < 0 \text{ and } \mathbf{x}(t) \in C_2, \end{cases} \tag{3.7}$$

– Else, a correction is made to the weight vector $\mathbf{w}$ and it is modified according to the following rule :

$$\begin{cases} \mathbf{w}(t+1) = \mathbf{w}(t) - \gamma(t)\mathbf{x}(t) & \text{if } \mathbf{w}^\top(t)\mathbf{x}(t) \geq 0 \text{ and } \mathbf{x}(t) \in C_2, \\ \mathbf{w}(t+1) = \mathbf{w}(t) + \gamma(t)\mathbf{x}(t) & \text{if } \mathbf{w}^\top(t)\mathbf{x}(t) < 0 \text{ and } \mathbf{x}(t) \in C_1, \end{cases} \tag{3.8}$$

where $\gamma(t)$ is the learning rate parameter.

To prevent the error and to update the weights, we need to increase $\mathbf{w}$ if $\mathbf{x}$ is positive and belongs to class C_1. Otherwise, the update needs to be opposite. The algorithm does not update weights if the existing model correctly make predictions for the current training data. Learning parameter $\gamma(t)$ controls how fast or slow we will move towards the optimal weights at time-step t. Unlike the gradient-based algorithms, we can drop the learning rate parameter γ easily as it only affects on scaling the weight parameters by some fixed constant. Therefore, it does not have impact on the learning behavior of the perceptron algorithm (Ng, 2000). If $\gamma(t)$ is set to γ, a fixed positive value during the training time, then we have a fixed-increment updating rule for the perceptron model. Choosing a very large γ constant, may skip the global minimum and picking a small one may lead to an exhaustive search to reach to convergence. Therefore, selecting a proper value for learning rate is crucial. In the following, we present a proof of convergence for perceptron model from (Block, 1962) considering a fixed-positive learning parameter for which $0 \leq \gamma < 1$.

Lemma 3.1.1. *For a fixed-positive learning rate $\gamma(t) = 1$, iterations t and $\mathbf{w}(0) = 0$, the learning for weights of the perceptron model will be terminated after at most $t_{\max}$ iterations.*

Proof. To prove the perceptron convergence, we first present a proof for the initial condition

$\mathbf{w}(0) = 0$ from (Block, 1962). Suppose that $\mathbf{w}^\top(t)\mathbf{x}(t) < 0$, $\forall t = 1, 2, ...$ and the input vector $\mathbf{x}(t) \in X_0$. That is the case that we have a false classification for the vectors $\mathbf{x}(1), \mathbf{x}(2), ...$, as the second condition is not held. Then, given the fixed-value learning parameter $\gamma(t) = 1$, we have

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{x}(t) \ \ , \ \ \mathbf{x}(t) \in C_1. \tag{3.9}$$

Initializing $\mathbf{w}(0) = 0$, we could iteratively solve this equation for $\mathbf{w}(t+1)$, obtaining the result

$$\mathbf{w}(t+1) = \mathbf{x}(1) + \mathbf{x}(2) + ... + \mathbf{x}(t). \tag{3.10}$$

By virtue of assuming that the classes C_1 and C_2 are linearly separable, there is a solution $\hat{\mathbf{w}}$ for which $\mathbf{w}^\top(t)\mathbf{x}(t) > 0$ for the vectors $\mathbf{x}(1), ..., \mathbf{x}(t) \in X_0$. Now, for a fixed $\hat{\mathbf{w}}$, we could define a positive number ε_0 as margin as follows :

$$\varepsilon_0 = \min_{\{\mathbf{x}(t) \in X_0\}} \hat{\mathbf{w}}^\top \mathbf{x}(t) \tag{3.11}$$

Consequently, if we multiply both sides of equation (3.10) by the vector $\hat{\mathbf{w}}^\top$, the following equation is obtained :

$$\hat{\mathbf{w}}^\top \mathbf{w}(t+1) = \hat{\mathbf{w}}^\top \mathbf{x}(1) + \hat{\mathbf{w}}^\top \mathbf{x}(2) + ... + \hat{\mathbf{w}}^\top \mathbf{x}(t). \tag{3.12}$$

Therefore, considering the definition given in (3.11), we get

$$\hat{\mathbf{w}}^\top \mathbf{w}(t+1) \geq t\varepsilon_0. \tag{3.13}$$

Then, given two vectors $\hat{\mathbf{w}}$ and $\mathbf{w}(t+1)$, and using a Cauchy–Schwarz inequality

$$\| \hat{\mathbf{w}} \|^2 \| \mathbf{w}(t+1) \|^2 \geq \left\{ \hat{\mathbf{w}}^\top \mathbf{w}(t+1) \right\}^2, \tag{3.14}$$

where $\| \, . \, \|$ indicates the euclidean norm of the argument vector and the inner $\hat{\mathbf{w}}^\top \mathbf{w}(t+1)$ gives a scalar quantity. Recalling from equation (3.13) that $[\hat{\mathbf{w}}^\top \mathbf{w}(t+1)]^2 \geq t^2 \varepsilon_0^2$ and from equation (3.14) we deduce that $\| \hat{\mathbf{w}} \|^2 \| \mathbf{w}(t+1) \|^2 \geq [\hat{\mathbf{w}}^\top \mathbf{w}(t+1)]^2$. Subsequently,

$$\| \hat{\mathbf{w}} \|^2 \| \mathbf{w}(t+1) \|^2 \geq t^2 \varepsilon_0^2. \tag{3.15}$$

Analogously,

$$\| \mathbf{w}(t+1) \|^2 \geq \frac{t^2 \varepsilon_0^2}{\| \hat{\mathbf{w}} \|^2}. \tag{3.16}$$

Then, we continue the work in a different direction by rewriting (3.9), specifically in the form

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{x}(t) \ \ , \ \ t = 1, 2, ..., t_{\max} \ \ , \ \ \mathbf{x}(t) \in X_0 \tag{3.17}$$

where $t_{\max}$ is the maximum time iteration.

Calculating the squared euclidean norm of both sides of the above equation gives

$$\| \mathbf{w}(t+1) \|^2 = \| \mathbf{w}(t) \|^2 + \| \mathbf{x}(t) \|^2 + 2\mathbf{w}^\top(t)\mathbf{x}(t). \tag{3.18}$$

Since $\mathbf{w}^\top(t)\mathbf{x}(t) \leq 0$, we could conclude from (3.18) that

$$\| \mathbf{w}(t+1) \|^2 \leq \| \mathbf{w}(t) \|^2 + \| \mathbf{x}(t) \|^2, \tag{3.19}$$

or, equivalently,

$$\| \mathbf{w}(t+1) \|^2 - \| \mathbf{w}(t) \|^2 \leq \| \mathbf{x}(t) \|^2 \quad , \ t = 1, 2, ..., t_{\max}. \tag{3.20}$$

Referring to initial condition $\mathbf{w}(0) = 0$, we sum these inequalities for $t = 1, ..., t_{\max}$, and obtain the following inequality

$$\| \mathbf{w}(t+1) \|^2 \leq \sum_{t=1}^{t_{\max}} \| \mathbf{x}(t) \|^2$$

$$\leq n\beta_{\max}, \tag{3.21}$$

where $\beta_{\max}$ is an upper bound given by

$$\beta_{\max} = \max_{\mathbf{x}(t) \in X_0} \| \mathbf{x}(t) \|^2, \tag{3.22}$$

and n is data size (Block, 1962).

Equation (3.21) declares that the squared euclidean norm of the weight vector $\mathbf{w}(t+1)$ mostly increases linearly according to the number of time-steps t. Another consequence inferred from

equation (3.21) is quite clearly in conflict with the result obtained by (3.16) for sufficiently large values of t. As a result, we can set a limit on iteration t and state that $t \leq t_{\max}$. Then both equation (3.16) and (3.21) are satisfied with the equality sign meaning that $t_{\max}$ is obtained by the equation

$$\frac{t^2{}_{\max}\varepsilon_0^2}{\| \hat{\mathbf{w}} \|^2} = t_{\max}\beta_{\max}. \tag{3.23}$$

So far, it has been proven that for a fixed-positive learning rate, the perceptron learning will be terminated after at most $t_{\max}$ iterations. Note that $\hat{\mathbf{w}}$ or $t_{\max}$ are not the unique solutions, meaning that the perceptron is capable of finding more than one solution. One important point to remember is that if we have two data points with different labels but very close to each other, the convergence time might be too long as the margin ε_0 is too small in this case. Therefore, the convergence time depends on margin and not the number of training data or features. A fixed-increment covergence theorem for the perceptron learning algorithm is first presented by Rosenblatt in (Rosenblatt, 1962). "Suppose that the subsets of training vectors X_0 and X_1 are linearly separable in the sense that the inputs to the perceptron come from these two subsets. The perceptron learning algorithm converges after some t' iterations, that is

$$\mathbf{w}(t') = \mathbf{w}(t'+1) = \mathbf{w}(t'+2) = ... \tag{3.24}$$

$\mathbf{w}(t')$ is a solution vector for $t' \leq t_{\max}$."

In the next step, we aim at considering the updating rule for adapting the weights of the single-neuron perceptron, with a variable learning rate $\gamma(t)$. Particularly, on the assumption of $\gamma(t)$ would be the smallest integer for meeting the following condition,

$$\gamma(t)\mathbf{x}^\top(t)\mathbf{x}(t) > |\mathbf{w}^\top(t)\mathbf{x}(t)| \tag{3.25}$$

we realize that if the inner product $\mathbf{w}^\top(t)\mathbf{x}(t)$ at iteration t leads in an incorrect sign, then $\mathbf{w}^\top(t+1)\mathbf{x}(t)$ at time-step $t+1$ would reach to the right answer. It means that if $\mathbf{w}^\top(t)\mathbf{x}(t)$ has an incorrect sign, at iteration t, then we may improve the learning procedure at iteration $t+1$ by setting $\mathbf{x}(t+1) = \mathbf{x}(t)$. This procedure is iterated until the example is classified correctly by the perceptron model. Note that the perceptron updates the weights vector only

on those data examples on which it makes a mistake.

Algorithm 1: Perceptron Convergence Algorithm

Data: $\mathbf{x}(t) = [1, x_1, ..., x_m]^\top$, $\mathbf{w}(t) = [w_0, w_1, ..., w_m]^\top$, $y(t)$,$d(t)$, $0 < \gamma \leq 1$

1 Initialize $\mathbf{w}(0)$ with zero values;

2 t = 0;

3 **while** $t \leq t_{\max}$ **do**

4 Activate the perceptron by applying continuous-valued input vector $\mathbf{x}(t)$ and desired response $d(t)$;

5 Compute the actual response $y(t) = \text{sgn}[\mathbf{w}^\top(t)\mathbf{x}(t)]$ where sgn(.) is the step function;

6 Update the weight vector $\mathbf{w}$ of the perceptron model by following adapting rule :

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \gamma \{d(t) - y(t)\} \mathbf{x}(t),$$

where

$$d(t) = \begin{cases} +1 & \text{if } \mathbf{x}(t) \in C_1, \\ -1 & \text{if } \mathbf{x}(t) \in C_2. \end{cases}$$

7 t = t + 1

8 **end**

A summary of the perceptron convergence algorithm is demonstrated by Algorithm 1 given in (Lippmann, 1987). One important point to note in Algorithm 1 is the symbol sgn(.), in step 5 is used for calculating the prediction of the perceptron and behaves according to following rule :

$$\text{sgn}(x) = \begin{cases} +1 & x \geq 0, \\ -1 & x \leq 0. \end{cases} \tag{3.26}$$

We state the quantized response $y(t)$ of the perceptron in the form of

$$y(t) = \text{sgn}\left\{\mathbf{w}^\top(t)\mathbf{x}(t)\right\}. \tag{3.27}$$

Recall that the input vector $\mathbf{x}(t)$ is an $(m+1) \times 1$ vector whose first feature is fixed at $+1$ throughout the calculation. Accordingly, the weight vector $\mathbf{w}(t)$ is an $(m+1) \times 1$ vector whose first feature interpretes as the bias. Furthermore, we should remember that a quantized

desired response $d(t)$, in Algorithm 1 is defined by

$$d(t) = \begin{cases} +1 & \text{if } \mathbf{x}(t) \in C_1, \\ -1 & \text{if } \mathbf{x}(t) \in C_2. \end{cases} \tag{3.28}$$

Therefore, an update rule for adapting the weight vector $\mathbf{w}(t)$ is obtained in the form of the error-correction learning rule

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \gamma \{d(t) - y(t)\} \mathbf{x}(t). \tag{3.29}$$

Setting a learning rate has been recognized as problematic for the perceptron learning algorithm. Keep in mind two conflicting requirements : setting learning rate parameter γ too high can cause the algorithm to diverge and a small γ setting makes it slow to converge (Lippmann, 1987). An extension of the update rule makes the learning rate a decreasing function $\gamma(t)$ of the iteration number t, in such a way that large changes in the parameters are seen in the first iterations, while the later iterations cause only the parameter tuning (Darken et Moody, 1990).

3.1.4 Perceptron Learning Rule on Loss Function

In this Section, we discuss the perceptron learning rule with gradient descent on the perceptron loss function. Remembering the linear separability property of the perceptron which is explained in Section 3.1.3, the main idea in learning a perceptron model is to fit the input data in an online fashion to find a weight vector that minimizes the error metric of the classification which is equal to the number of the misclassified data. Suppose that we have a set of input examples from class 0, $X_0 \subset R^{m+1}$, called the set of positive data instances and another set of inputs from class 1, $X_1 \subset R^{m+1}$, called the set of negative data. Then, our goal is to generate a perceptron model that yields +1 for all examples from X_0 and -1 for data from X_1. Given X_0 and X_1 which must be separated by a perceptron model, a learning algorithm should automatically fit the weights necessary for the solving the problem. Two sets of X_0 and X_1 must be separated in $m+1$ dimensional space such that a perceptron model calculates the binary linear function f with $f(\mathbf{x}) = +1$ for $\mathbf{x} \in X_0$ and $f(\mathbf{x}) = -1$ for $\mathbf{x} \in X_1$. The separability of the linear model f relies on the adapted weights values. We calculate the loss of the classification by computing the number of misclassified data points

which is defined as :

$$E(\mathbf{w}) = \sum_{n}^{i=1} E_i(\mathbf{w}),$$

where

$$E_i(\mathbf{w}) = \max\{0, -y_i f(\mathbf{x}_i)\}, \tag{3.30}$$

and E_i is a modified form of the Hinge loss which is convex, but not differentiable. The perceptron loss function defined over all of weight space and attempts to learn a perceptron model with minimum cost. If $\mathbf{x}_i$ is correctly classified, then $E_i(\mathbf{w}) = 0$. The aim of the learning process is to reach the global minimum over all examples where $E = 0$ in an iterative manner (McCulloch et Pitts, 1943).

The data is linearly separable if there is a weight vector $\mathbf{w}$ so that for all data examples $\mathbf{x}_i$,

$$y_i \mathbf{w}^\top \mathbf{x}_i \geq 0. \tag{3.31}$$

A simple learning algorithm for training the perceptron computes the loss function E, and uses this to adapt the weight vectors through a form of gradient descent (McCulloch et Pitts, 1943). The key idea of perceptron learning is to iterate over the training input examples and update the weight vector $\mathbf{w}$ such that would make $\mathbf{x}_i$ more likely to be classified correctly.

Algorithm 2 illustrates how the perceptron learns the weight vector $\mathbf{w}$ in an online fashion

using loss function and gradient descent (a basic version of (McCulloch et Pitts, 1943)).

Algorithm 2: Perceptron Learning Algorithm with Gradient Descent

Data: Training data $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n]$, where $\mathbf{x}_i = [x_0, ..., x_m]^\top$, $\mathbf{y} = [y_1, ..., y_n]$, $0 < \gamma \leq 1$

, threshold $b < 1$ and maximum number of iterations $t_{\max}$. Initialize $\mathbf{w}(0)$ with zero values;

$t = 0$;

while *not converged or* $t \leq t_{\max}$ **do**

  $\Delta = 0$;

  for $\mathbf{x}_i \in \mathbf{X}$ *and target* $\mathbf{y}_i \in \mathbf{y}$ **do**

    Calculate the output of the perceptron as $f(\mathbf{x}_i) = \sum_{j=0}^{m} w_j x_{ij}.$;

    if $y_i f(\mathbf{x}_i) < 0$ **then**

      $\Delta = \Delta - \bigtriangledown E_i$;

    end

  end

  $\Delta = \frac{\Delta}{n}$;

  $\mathbf{w}(t+1) = \mathbf{w}(t) - \gamma\Delta$;

  if $|\Delta| < b$ **then**

    converged = False;

    Break ;

  end

  t = t + 1

end

The learning process will be done by initializing the weights with very small random values for weight vector $\mathbf{w}$ (typically zero) and assess the performance of the prediction by comparing the predicted labels, $f(\mathbf{x}_i)$, with the actual ones, y_i. Then, the algorithm will update the initial guess for the estimated weights by changing the weight vector values in the correct direction by implementing a form of gradient descent. The weight adaptation of $\mathbf{w}(t)$ to $\mathbf{w}(t+1)$ is done such that $\mathbf{w}(t+1)\mathbf{x}_i > \mathbf{w}(t)\mathbf{x}_i$ which is obtained by the following update rule :

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \gamma \bigtriangledown E_i,$$

where

$$\bigtriangledown E_i = \begin{cases} 0 & y_i \mathbf{w}\mathbf{x}_i > 0, \\ -y_i \mathbf{x}_i & \text{otherwise.} \end{cases} \tag{3.32}$$

Suppose that $\mathbf{x}_i$ is an example of false classification while it is positive.

$$f(\mathbf{x}_i) = \sum_{j=0}^{m} w_j x_{ij} < 0.$$

This process will be repeated in each iteration in an attempt to decrease the loss function E_i and converge to the situation where we have no misclassified data point. If the input vectors are not linearly separable, then the learning algorithm will never converge to a point where all input vectors are classified correctly. In (McCulloch et Pitts, 1943), it has been proven in details that the perceptron learning algorithm could not terminate if the training data is not linearly separable. With this convergence proof, a perceptron algorithm will eventually properly classify all the training data in a linearly separable space. However, this means that this model is trained specifically for the training set, and could not generalize well on validation or test datasets. If the data is not linearly separable, then it is required to set a limit on the maximum number of iterations to terminate the learning algorithm.

3.1.5 Perceptron with Sigmoid Function

Remember from the perceptron section, the perceptron learning algorithm looks for the minimum of the loss function in weight space using the method of gradient descent. The weight vector which minimizes the loss function is considered as the solution of the learning problem. Indeed this algorithm requires computation of the derivative of the loss function at each time step. Therefore, to guarantee the smoothness and differentiability of the loss function, we have to apply a continous form of the activation function instead of step function in perceptron model (Rojas, 2013). One of the most popular non-linear activation functions for this purpose is the logistic sigmoid, defined as :

$$s(x) = \text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}, \tag{3.33}$$

so that $s(.)$ is always monotonically increasing, continuous, differentiable and bounded in $0 \leq s(.) \leq 1$. The derivative of the sigmoid with respect to input x, required later on in this section, is easily calculated as follows :

$$s'(x) = s(x) \times \{1 - s(x)\} \tag{3.34}$$

which is usually employed for calculating the weight updates in training the perceptron model with gradient descent. Below is the graph of a sigmoid function and its derivative corresponding to the variable x.

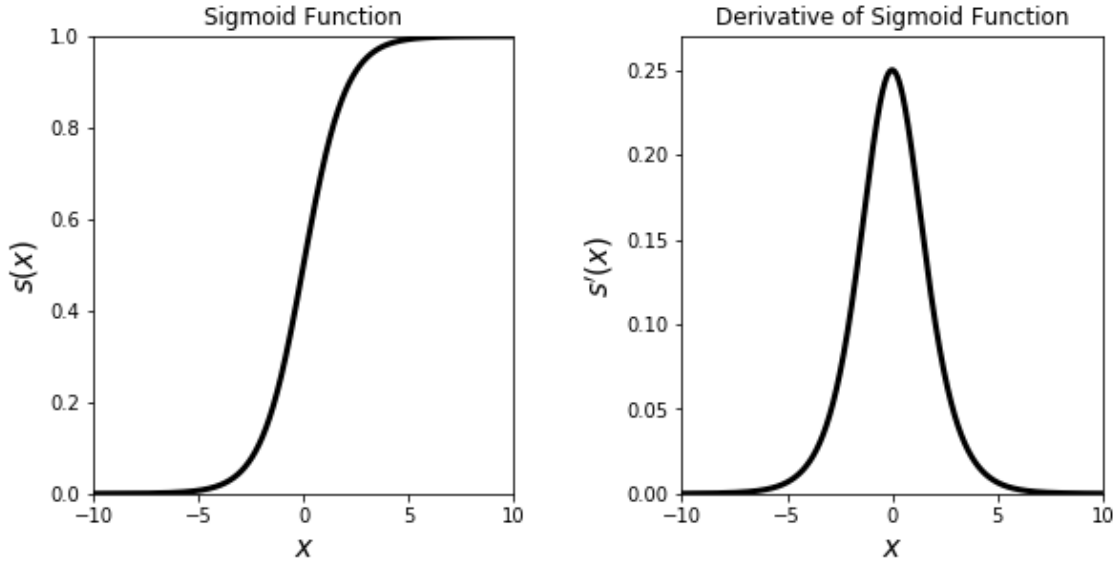


Figure 3.3 Sigmoid and its derivative function according to equations (3.33) and (3.34).

Therefore, we consider the case of learning a perceptron model whose activation function is a sigmoid instead of step function in which we have a single target variable y such that $y = 1$ indicates class C_1 and $y = 0$ referes to data belongs to class C_2. The application of the sigmoid activation function in a perceptron model is illustrated in Figure 3.4.

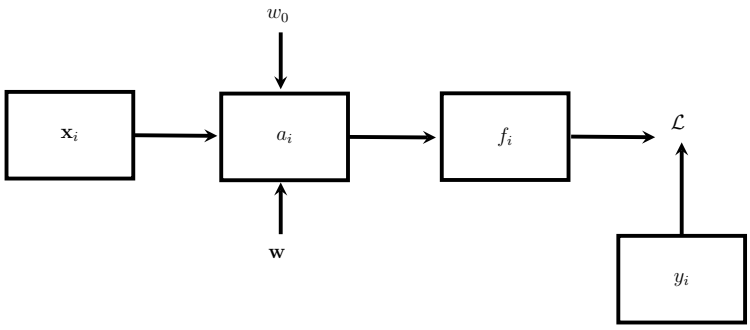


Figure 3.4 The application of the sigmoid activation function in a perceptron model.

Suppose that we have a perceptron model where $\mathbf{x}_i$ is the i^{th} pattern of the input $\mathbf{x}$ including $m+1$-dimentional data, $\mathbf{x}_i = [1, x_{i1}, x_{i2}, ..., x_{im}]^\top$, weight parameters $\mathbf{w} = [w_0, ..., w_m]^\top$ where w_0 is the bias parameter, a_i is the weighted sum of the inputs, and f_i is the calculated output

of the perceptron model which is obtained by the sigmoid function. Additionally, y_i is used as the corresponding target label for input $\mathbf{x}_i$.

In this case, the model first computes a linear combination of its input signals and then applies a sigmoid function to the result as follows :

$$f_i = \text{sigmoid}(\mathbf{w}^\top \mathbf{x}_i) = \frac{1}{1 + \exp(-\mathbf{w}^\top \mathbf{x}_i)}. \tag{3.35}$$

We can describe f_i as the conditional probability $p(C_1|\mathbf{x}_i)$, with $p(C_2|\mathbf{x}_i)$ given by $1 - f_i$. The output of the perceptron model with sigmoid function can be interpreted as probabilities since we have the conditional distribution of targets given inputs as a Bernoulli distribution of the following form :

$$p(y_i|\mathbf{x}_i, \mathbf{w}) = f_i(\mathbf{x}_i)^{y_i}\{1 - f_i(\mathbf{x}_i)\}^{1-y_i}, \tag{3.36}$$

where the predicted class is 1 if $f_i \geq 0.5$ and 0 if $f_i < 0.5$.

Assume that we have a training set including n data examples $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n]$ and corresponding multivariate target responses $\mathbf{Y} = [\mathbf{y}_1, ..., \mathbf{y}_n]$, then the general form of loss function for a multiclassification problem, which is given by the negative log likelihood, is a cross-entropy loss function and it takes the form :

$$\mathcal{L}(\mathbf{W}) = \sum_{i=1}^{n} \mathcal{L}_i(\mathbf{W}),$$

where

$$\mathcal{L}_i(\mathbf{W}) = -\sum_{k=1}^{K}\{y_{ik} \log(f_{ik}) + (1 - y_{ik}) \log(1 - f_{ik})\}, \tag{3.37}$$

where $\mathbf{W}$ is a weight matrix of the size $(m+1) \times K$, and K is the number of desired classes. Therefore, $\mathbf{a}_i = \mathbf{W}^\top \mathbf{x}_i$ and $\mathbf{f}_i = f(\mathbf{a}_i) = f(\mathbf{W}^\top \mathbf{x}_i)$ and

$$f_{ik} = f(a_{ik}) = f(\mathbf{w}_k^\top \mathbf{x}_i), \tag{3.38}$$

is the sigmoid output of $\mathbf{x}_i$ corresponding to class k. Note that $\mathbf{y}_i$ should be represented in one-hot vector format and $\mathbf{f}_i$ is calculated into one-hot vector format as well with k entries (Harris et Harris, 2010).

We now have a model capable of calculating the total loss for a given training data. The weights in the perceptron model are the only parameters that can be modified to make the

cross-entropy loss $\mathcal{L}$ as low as possible which is a continuous and differentiable function of the weights $\mathbf{W}$ in the model. Taking the derivative of the loss function respecting the activation for a particular output gives the following equation :

$$\frac{\partial \mathcal{L}_i}{\partial \mathbf{w}_k} = \frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_k}\frac{\partial \mathbf{f}_i}{\partial \mathbf{a}_i}\frac{\partial \mathcal{L}_i}{\partial \mathbf{f}_i}$$

$$= \frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_k}\frac{\partial \mathcal{L}_i}{\partial \mathbf{a}_i}. \tag{3.39}$$

To compute the partial derivative of the loss with respect to $\mathbf{a}_i$, first we do :

$$\frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_k} = \frac{\partial \mathbf{w}_k^\top \mathbf{x}_i}{\partial \mathbf{w}_k} = \begin{bmatrix} 0 & \mathbf{x}_{i0} & 0 \\ \vdots & \ddots & \vdots \\ 0 & \mathbf{x}_{im} & 0 \end{bmatrix}, \tag{3.40}$$

where vector $\mathbf{x}_i$ is kept in the column k of the above matrix. Hence, we have :

$$\begin{aligned} \frac{\partial \mathcal{L}_i}{\partial \mathbf{a}_i} &= \frac{\partial \Big[-\sum_{k=0}^{K} \{ y_{ik} \log f_{ik} + (1 - y_{ik}) \log(1 - f_{ik}) \} \Big]}{\partial \mathbf{a}_i} \\ &= \mathbf{y}_i - \frac{1}{1 + \exp(-\mathbf{a}_i)} \\ &= -\mathbf{y}_i - p(y_{ik}|\mathbf{x}_i) \\ &= \mathbf{y}_i - \mathbf{f}_i. \end{aligned} \tag{3.41}$$

Now, we could rewrite the partial derivative of loss with respect to $\mathbf{a}_i$ as :

$$\frac{\partial \mathcal{L}_i}{\partial \mathbf{a}_i} = (\mathbf{y}_i - \mathbf{f}_i), \tag{3.42}$$

where $\mathbf{f}_i$ is the logistic sigmoid function of ${\mathbf{w}_k}^\top \mathbf{x}_i$ and $\mathbf{y}_i$ is the target vector for pattern i. Now having $\frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_k}$ and $\frac{\partial \mathcal{L}_i}{\partial \mathbf{a}_i}$, we could formulate the gradient of loss function $\mathcal{L}$ corresponding to the weight matrix $\mathbf{w}_k$ in the following compact form :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_k} = \frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_k}\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i}$$

$$= -(\mathbf{y}_i - \mathbf{f}_i)\mathbf{x}_i, \tag{3.43}$$

where the gradient matrix computation for the i^{th} data example is formulated simply by using the error times input vector. Then, to minimize the cross-entropy loss $\mathcal{L}$ by using an iterative process of gradient descent which gives us a weight update rule with a gradient ascent rule, we have :

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) - \gamma \frac{\partial \mathcal{L}}{\partial \mathbf{w}_k}, \tag{3.44}$$

where γ denotes a learning rate, a parameter used to interpret the step length of each iteration in the negative gradient direction. If we compare this to the update rule using the gradient of least squares function, we figure out that it looks similar but with one important difference : the f function we used as the activation function is now a non-linear and differentiable function. With this extension of the basic perceptron model the algorithm learning problem now reduces to the question of computing the gradient of a cross-entropy loss function with respect to the weight vector by finding a minimum of the error function, where the gradient equals to zero. Once we have the derivative of the loss function, we can then adapt the model's weight parameters iteratively. Algorithm 3 demonstrates how the perceptron model

with the sigmoid function learns the weight vector $\mathbf{w}_k$ using cross-entropy loss function.

Algorithm 3: Perceptron Learning Algorithm with Sigmoid and Cross-entropy Loss.

Data: Training data $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n]^\top$, and $\mathbf{x}_i = [1, x_{i1}, ..., x_{im}]^\top$, $\mathbf{Y} = [\mathbf{y}_1, ..., \mathbf{y}_n]$, $\mathbf{y}_i \subset R^K$ and $\mathbf{y}_i$ is of the one-hot form, learning rate $0 < \gamma \leq 1$, threshold $b < 1$, maximum number of iterations $t_{\max}$, converged = True

Result: fitted matrix $\mathbf{W}$, $\mathbf{W} \subset R^{m+1} \times R^K$

Initialize $\mathbf{W}(0)$ with zero values;
Initialize $t = 0$;
while *not converged* ***or*** $t \leq t_{\max}$ **do**
    $\Delta = 0$;
    for $\mathbf{x}_i \in \mathbf{X}$ *and target* $\mathbf{y}_i \in \mathbf{Y}$ **do**
        Calculate the weighted sum of the input as $\mathbf{a}_i = \mathbf{w}_k^\top \mathbf{x}_i$;
        Calculate the output of the perceptron as $\mathbf{f}_i = \text{sigmoid}(\mathbf{a}_i) = \text{sigmoid}(\mathbf{w}_k^\top \mathbf{x}_i)$.;
        Calculate the cross-entopy loss function as

$$\mathcal{L}_i = -\sum_{k=1}^{K} \{y_{ik} \log(f_{ik}) + (1 - y_{ik}) \log(1 - f_{ik})\} .$$

        Update the weights vectors according to the gradient of the loss function $\mathcal{L}$ with respect to the all weight attributes as

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \gamma \frac{\partial \mathcal{L}_i}{\partial \mathbf{w}_k} \mathbf{x}_i.$$

        ;
        if $\frac{\partial \mathcal{L}_i}{\partial \mathbf{w}_k} < b$ **then**
            converged = False;
            Break ;
        end
    end
    t = t + 1
end

In (Simard *et al.*, 1993), the authors indicate that using the cross-entropy loss function instead of the sum-of-squares for a classification problem results in an acceleration in learning as well as generalizing performance.

3.1.6 Perceptron Issues

Intuitively, it is desired to find a hyperplane to be generalized more accurately on unseen data. Support Vector Machines (SVMs) are robust linear classifiers which find a separating hyperplane by maximizing margin. Additionally, the most classic example of the perceptron's inability to solve problems with a linearly inseparable space are XOR, AND and XNOR boolean functions. As seen in Table 3.1, a single-layer perceptron is not capable of computing all the boolean functions with two variables x_1 and x_2.

Table 3.1 The result table for some boolean functions over two variables x_1 and x_2, T represents value 1 and F denotes value 0.

x_1	x_2	AND	NAN	OR	XOR	NOR	XNOR
F	F	F	T	F	F	T	T
F	T	F	T	T	T	F	F
T	F	F	T	T	T	F	F
T	T	T	F	T	F	F	T

The perceptron only could perform its functionality for those boolean functions that the points with label 1 can be separated from the points with label 0 linearly. Obviously, two boolean functions in the above table, XOR and XNOR, cannot be computed in this way. A computational analysis is provided to show the linearly inseparable pattern of the XNOR function as follows :

$$f = \text{sgn}(w_1 x_1 + w_2 x_2 - b),$$

where b is threshold. Therefore,

1) $x_1 = 0, \;\; x_2 = 0 \rightarrow \;\; w_1 x_1 + w_2 x_2 - b > 0 \rightarrow w_0 < 0,$

2) $x_1 = 0, \;\; x_2 = 1 \rightarrow \;\; w_1 x_1 + w_2 x_2 - b < 0 \rightarrow w_2 < b,$

3) $x_1 = 1, \;\; x_2 = 0 \rightarrow \;\; w_1 x_1 + w_2 x_2 - b < 0 \rightarrow w_1 < b,$

4) $x_1 = 1, \;\; x_2 = 1 \rightarrow \;\; w_1 x_1 + w_2 x_2 - b > 0 \rightarrow w_1 + w_2 > b.$

As a result of the second and third inequalities, w_1 and w_2 are both negative while the last inequality implies that $w_1 + w_2 \geq b$ which cannot be true and makes a conflict. Figure (3.5) illustrates the XNOR that cannot be separated with a single line. The solution for the pattern of (x_1, x_2) consists of two separated lines l_1 and l_2.

(Minsky *et al.*, 1969) studied the possibility of learning linear decision boundaries for all boolean logic functions. They indicated that it is impossible for a single-unit perceptron to learn the linearly inseparable patterns like XOR and XNOR while a multi-layer perceptron

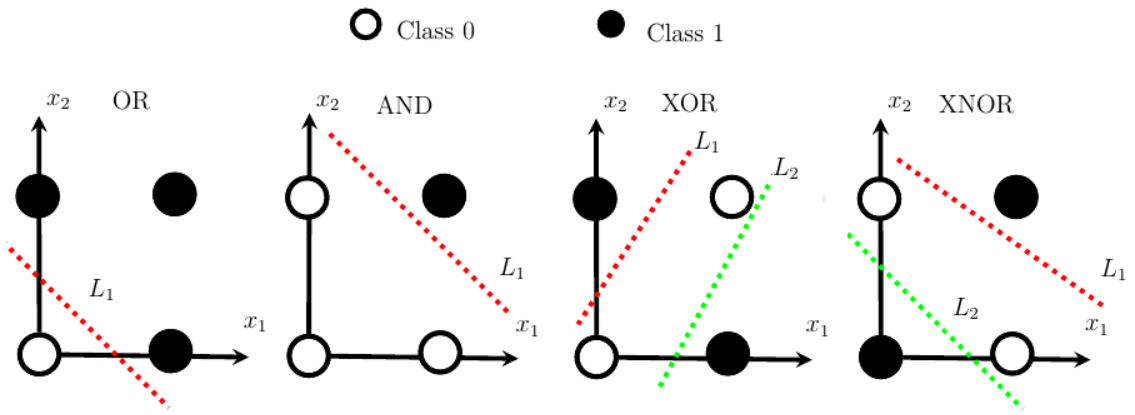


Figure 3.5 A demonstration of OR, AND, XOR and XNOR boolean functions.

is capable of learning all boolean functions. Then, "if the training data is linearly separable, then the perceptron learning algorithm is capable of finding one of the possible hyperplanes solutions and not the best one" (Minsky *et al.*, 1969). Note that the criteria for discrimination between the non-linear separable spaces is not the shape of activation function in the perceptron model, however the activation function provides the linear separability. Indeed, we have to use more than one hyperline or any other non-linear functions to discriminate the points of XOR and XNOR by implementing at least two separate perceptron models. Intuitively, we could use a combination of multiple-layer perceptrons to separate the intermediate results on the basis of sign which will be discussed in detail in sequel.

3.2 Multi Layer Perceptrons (MLPs)

Multi-layer Perceptrons (MLP) [1] are the feedforward Artificial Neural Networks (ANN) which are originally inspired by the capabilities of human cognition and brain system in processing information (Haykin *et al.*, 2009). The MLP models have been developed as a generalization of the mathematical paradigms of the biological nervous systems according to the following assumptions (Witten *et al.*, 2016; Haykin *et al.*, 2009; Fausett et Fausett, 1994) :

1. MLPs are typically organized in different fully-connected layers which are made up of a finite number of neurons.
2. Patterns are given to the MLP model through the input layer, which is fully connected to an intermediate hidden layer. This intermediate layer is linked to another hidden layer or directly to the output layer via the weighted connections to produce the output of the MLP model.

1. This section is heavily based on notes and notations inherited from (Witten *et al.*, 2016).

3. Neurons in successive layers are connected by means of directed weighted connections across layers and there is no connection within the same layer.
4. Information processing is performed in neurons by applying a non-linear activation function to the weighted sum of the neurons' input.
5. The output of a neuron in one layer broadcasts to all neurons in the next layer.

A simple architecture of a MLP network with m input neurons, l hidden layers each of them includes N^l neurons, and one output layer with k neurons is demonstrated in Figure 3.6. Note that in this graph, we indicate the hidden units by the square shapes on purpose to emphasize that those units are intermediate deterministic variables and are not formulated as random variables, rather than the output units which typically are produced probabilistically and therefore are illustrated by circles (Witten *et al.*, 2016).

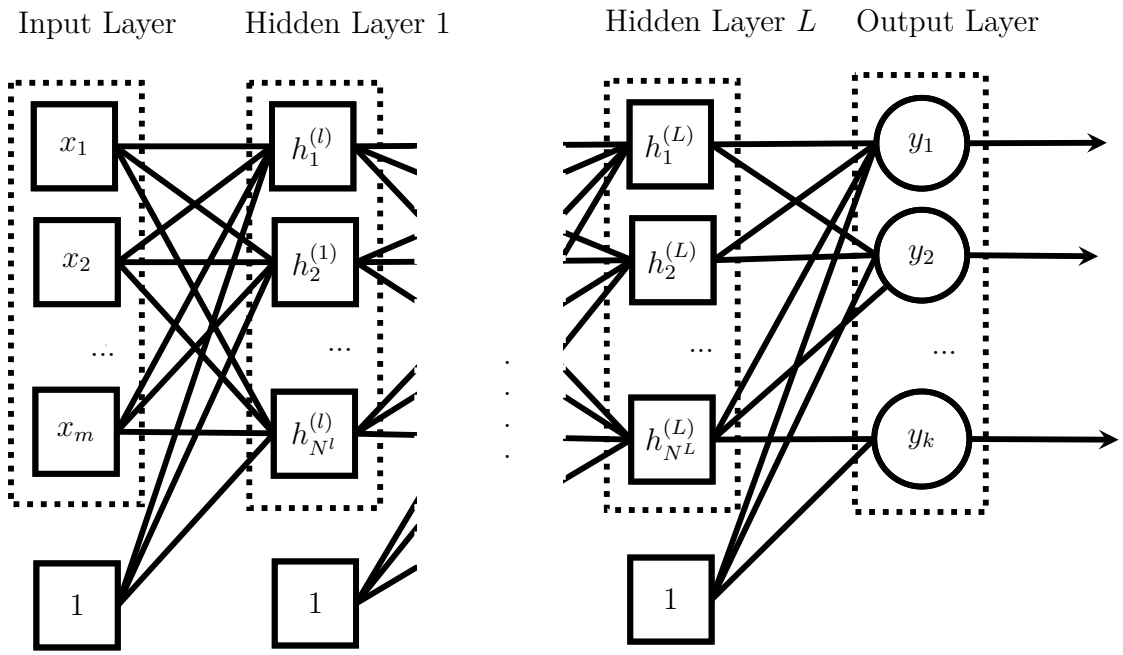


Figure 3.6 An architecture of a feed-forward neural network with L hidden layers (reproduced from (Witten *et al.*, 2016)).

There are several types of Artificial Neural Networks depending on
– the pattern of the connections among the neurons (architecture),
– the learning algorithm applied for adjusting the connection weights and
– the activation functions used in neurons.
In this thesis we are only focused on two of them :

1. Multi-layer Perceptrons (MLPs) and

2. Convolutional Neural Networks (CNNs).

In this chapter, we first begin by considering the functional conncepts of the MLP model, including the architechture, parameters of the basic model, activation functions and loss function, and we then discuss the problem of learning the weights parameters within a minimum negative of log likelihood framework, which involves the solution of a gradient descent-based optimization problem. This requires the evaluation of the gradient of the log likelihood against the weights parameters, and we shall see how an effective formulation can be obtained using the backpropagation learning algorithm afterwards. We also discuss two regularization methods and some techniques to accelerate the performance of the MLPs.

MLP models are typically organized in three different components : input layer, at least one hidden layer and one logistic output layer with non-linear and differentiable activation functions capable of learning nonlinear decision boundaries (oppose to a single perceptron with nonlinearity) (Bishop, 2006; Witten *et al.*, 2016; Haykin *et al.*, 2009). A classical architecture for a MLP model is made up of a fully-connected hidden layer consisting a finite number of neurons behave as activation functions. Input patterns are fed to the network through the input layer which is then connected to a hidden layer. Similarly the hidden layer is connected to the output layer and it is processed to produce the output prediction of the model.

We can comprise the MLP model with several hidden layers of perceptron neurons. Typically neurons are grouped in layers and each neuron in one layer receives input only from neurons in the preceding layer. One single layer network can be considered as a linear decision boundary. There are different types of single layer networks like Hebb (Hebb, 1949), ADALINE (Widrow et Hoff, 1960) and Perceptrons (Rosenblatt, 1957).

Each hidden layer l connects N^l input units to N^{l+1} output units in the next layer $l+1$ simply which means that a unit's output is one of the input for all units in the next hidden layer. This broadcasting will be performed via fully weighted connections. Note that the inputs and outputs for a hidden layer are easily distinguishable from the inputs and outputs of the entire model. Furthermore, we can easily set different activation functions in different hidden layers, or even different neurons within a hidden layer.

Each link represents the data flow and is associated with one weight. Each neuron in each layer behaves like a single-neuron perceptron model by applying a non-linear activation function to the weighted sum of its input signals which was described in detail in Chapter 2. The first layer known as the input layer consists of the pattern vector $\mathbf{x}$. No computation is performed in this layer. The first hidden layer receives the input vector from the input layer, processes

it by a non-linearity, and transmits its output as vector $\mathbf{h}^l$ to the next hidden layer.

$$\mathbf{h}^l = \text{sigmoid}({\mathbf{W}^l}^\top \mathbf{h}^{l-1}) = \frac{1}{\left\{1 + \exp({\mathbf{W}^l}^\top \mathbf{h}^{l-1})\right\}}. \tag{3.45}$$

This process repeats in all L hidden layers. The last hidden layer receives the inputs from the preceding hidden layer as $\mathbf{h}^{L-1}$ and sends this vector to the output layer. After applying the logistic activation functions by output neurons, the output layer then produces the output vector $\mathbf{f}$. The output of the MLP is typically made up of computations that can be obtained by the outputs of the consecutive intermediate layers denoted by $\mathbf{h}^l$. Therefore, such models generate the output could be formulated among the L layers (Witten *et al.*, 2016).

Typically, the input layer in a MLP is considered as layer 0, referring to an L layer network means that we have a MLP with $L+1$ weighted non-input layers, including L hidden layers and one output layer. In this chapter, we suppose that all neurons in a same non-input layer apply the same activation functions. We have noted before that the sigmoid function is an appropriate and differentiable non-linear activation function that applies a smooth thresholding for perceptron models. Hence, we simply perform the non-linearity on the hidden layers using logistic sigmoid activation functions and softmax for the output layer as well. Therefore, the output of a two layer network model $\mathbf{f}$ takes the following form :

$$\mathbf{f} = \text{out}(\boldsymbol{\eta}^{L+1}), \quad , \quad \boldsymbol{\eta}^{L+1} = \mathbf{W}^{L+1}\mathbf{h}^L$$

where

$$\mathbf{h}^L = \text{act}(\boldsymbol{\eta}^L), \quad , \quad \boldsymbol{\eta}^L = \mathbf{W}^L\mathbf{h}^{L-1} \vdots \mathbf{h}^l = \text{act}(\boldsymbol{\eta}^l), \quad , \quad \boldsymbol{\eta}^l = \mathbf{W}^l\mathbf{h}^{l-1} \vdots$$
$$\mathbf{h}^1 = \text{act}(\boldsymbol{\eta}^1), \quad , \quad \boldsymbol{\eta}^1 = \mathbf{W}^1\mathbf{h}^1$$
$$\mathbf{h}^1 = \mathbf{x}, \tag{3.46}$$

and $act(.)$ is a nonlinearity used as the activation function (Witten *et al.*, 2016).

A computation graph in Figure 3.7 demonstrates the forward propagation and architecture of the decomposed model to indicate how the MLP predictions and the loss function $\mathcal{L}$ are computed for a $L+1$ layer network.

$$\mathbf{f} = f\left\{\boldsymbol{\eta}^{L+1}(\mathbf{h}^L(\boldsymbol{\eta}^L(...(\mathbf{h}^l(\boldsymbol{\eta}^l(...(\mathbf{h}^1(\boldsymbol{\eta}^1(\mathbf{x})))...))...))))\right\}, \tag{3.47}$$

where $\boldsymbol{\eta}^l$ has the form of a linear weighted sum of $\mathbf{W}^l$ including bias and $\mathbf{h}^{l-1}$ in such a way that :

$$\boldsymbol{\eta}^l = \mathbf{W}^l\mathbf{h}^{l-1}. \tag{3.48}$$

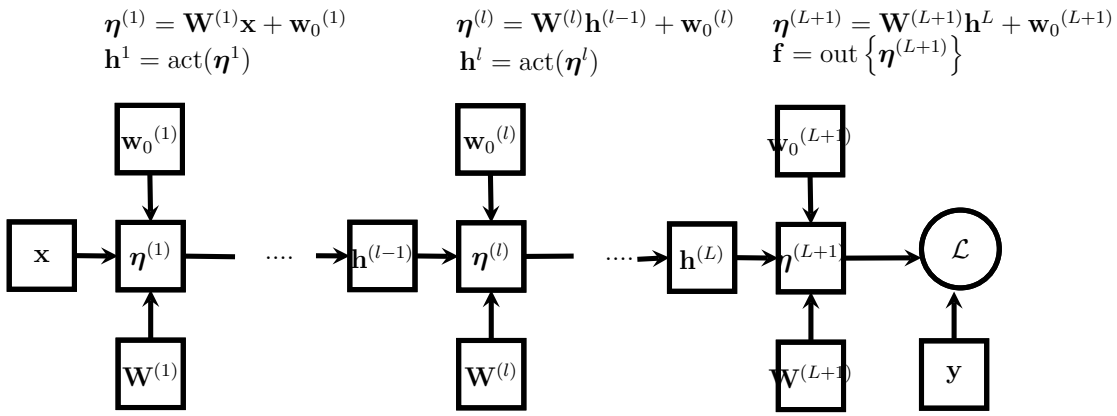


Figure 3.7 The forward propagation and structure of a multi-layer feed-forward neural network reproduced from (Witten *et al.*, 2016).

and for the first layer ($l = 1$), we can write :

$$\boldsymbol{\eta}^1 = \mathbf{W}^1\mathbf{x}, \tag{3.49}$$

where $\mathbf{x}$ is an example of the input data augmented by one in vector form and $\mathbf{W}$ is the weight vector with the bias appended at the end. Similarly, for the next hidden layers we have :

$$\boldsymbol{\eta}^l = \mathbf{W}^l\mathbf{h}^{l-1}, \tag{3.50}$$

where $\mathbf{h}^{l-1} = \text{act}(\boldsymbol{\eta}^{l-1})$ (Witten *et al.*, 2016).

3.2.1 Activation Functions

A non-smooth perceptron activation function means that the error rate of the model is a discontinuous function of the weight parameters which makes it difficult to adjust optimal weights by minimizing the loss function. To address this problem, we apply a continuous activation function. We saw in the previous chapter that the logistic sigmoid is an appropriate choice for this purpose which is capable of learning nonlinear decision boundaries for nonlinear separable spaces. Activation functions in MLP should be nonlinear, continuously differentiable and monotonically non-decreasing (Rosen-Zvi *et al.*, 1998). Additionally, it is desired to choose an activation function which its derivative would be computed easily.

With the assumption of using a nonlinear activation function e.g. sigmoid, it could be proven that an MLP is a universal function approximator. This means that a single-hidden layer MLP associated with sufficiently large and finite number of neurons can approximate

and learn any continuous function on a compact domain with an arbitrary accuracy (Cybenko, 1989). This property is required while we apply the backpropagation learning method using gradient-based approaches. To use the backpropagation learning method, the activation function should be differentiable so that the range of the activation function is finite. Typically, by choosing an activation function which is bounded in a certain range of limits, the gradient-based optimization methods regularly tend to be more stable, as pattern presentations considerably change only limited weight parameters. For activation functions with infinite range, learning is typically more efficient since pattern presentations remarkably tend to affect most of the weight parameters. Hence, we should necessarily select smaller learning rates for updating parameters (Snyman, 2005).

If the activation function used is monotonic, then the error surface associated with a single-layer network (no hidden layer) is guaranteed to be convex, like logistic regression (Wu, 2009). According to the proof provided in (Kingman, 1961), if function f is concave and g is convex and monotonically non-increasing over a univariate domain, then $h(x) = g\{f(x)\}$ would be convex, consequently. In their work, they proved that the output and global minima for a single perceptron could be found. But for multi-layer perceptrons, the problem becomes more complicated, yet is still possible. While moving to the solution using backpropagation through a gradient-based method, the monotonically decreasing loss function looks like a convex function with several local minima (Goodfellow *et al.*, 2014). Therefore, the main feature of the learning problem is to derive the update rules for the weights adjustments via a gradient-based algorithm. During the learning, backpropagation specifies the influence proportion on each neuron in the next layer. Using a nonmonotonic activation function may cause that increasing the neuron's weight affect less on the neurons in next layer and model might not converge to the solution.

Another desired property for choosing an activation function is to approximate near the origin. In this case, if the weight parameters are initialized with small random values, the MLP model tend to learn efficiently. Otherwise, we should care specifically while initializing the weights (Sussillo, 2014).

The most common choice of activation function used in MLP is so-called logistic sigmoid as it takes a real-valued input varied from $-\infty$ to $+\infty$ and saturates it to a bounded range between 0 and 1, which are the values used to represent the output class for a binary classification problem.

Despite the popularity of the sigmoid, a sigmoidal activation function in the form of a hyperbolic tangent is sometimes preferred empirically and theoretically for deep MLPs. Two forms of hyperbolic tangent activation functions are commonly used : $\tanh(x)$ whose range

is normalized to the range of -1 to 1, and $\frac{1}{1+\exp(-x)}$ which is vertically bounded to 0 and 1. The former function is a rescaling of the logistic sigmoid :

$$\tanh(x) = 2\text{sigmoid}(2x) - 1. \tag{3.51}$$

The hyperbolic tangent typically transform the data from domain $[-\infty, +\infty]$ to $[-1, +1]$ and it is symmetric around the origin while the sigmoid is not. The outputs of a logistic sigmoid will be always a positive value in $[0, 1]$. The nonsymmetry property of sigmoid around zero makes it more prone to saturation of the next layers during the training via gradient-based algorithms and making learning more difficult consequently.

In (LeCun *et al.*, 1998b), authors show that logistic sigmoid has been already demonstrated to slow down learning due to its non-zero mean which yields singular values in the Hessian during gradient-based learning. To address this problem, they provide evidence in great detail that normalizing the initial inputs to have mean 0 and variance 1 along the features typically makes a better and faster convergence during gradient-based learning. They also suggested that the activation function should be chosen as anti-symmetric and takes the form :

$$f(x) = a \tanh(bx), \tag{3.52}$$

where the $f(.)$ maps $[-\infty, +\infty]$ to $[-a, +a]$, $a = 1.7159$ and $b = \frac{2}{3}$.

Sigmoid and tanh function, however, both computes their derivatives very simply and efficiently which is a compelling reason for using them during the gradient optimization in a MLP.

While sigmoid and tanh have been commonly used activation functions, the more recent work of (Glorot *et al.*, 2011) shows evidence that rectified linear units (relu) provide faster and more effective learning of deep neural networks on complex and high-dimensional data. Relu computes the function $f(x) = \max(0, x)$ and simply thresholds the input matrix at zero. One of the most important advantages of relu, compared to sigmoid and its counterpart tanh, is that it does not require expensive computation consisting of only comparison and multiplication. Furthermore, there is no saturation in relu which means that we have an efficient backpropagation without vanishing or exploding gradient that makes it a proper choice particularly for networks with deep layers. These advantages combined with its ability to produce sparse activations are considered as benefits for optimizing deep MLPs and CNNs. On minus side, relu units suffer from an important potential problem during training : dying relu neurons. Dying problem occurs when no gradient flows backward through the relu unit

and therefore, that neuron subsequently will never fire from that point on. Typically, large number of units in a multi-layer network may pushed into dead states, while a high value selected for the learning rate. One attempt to alleviate this problem is to set a small learning rate during the weights update in gradient descent.

Another approach to mitigate this issue is applying Leaky relu instead of relu which allows a small negative slope when a unit is not active. A parametric rectified linear unit (prelu) activation function that generalizes the rectified unit by making the slopes into a parameters that are adapted along with the other network parameters were proposed by (He *et al.*, 2015).

There are several commonly-used activation functions you may encounter in practice. Figure 3.8 illustrates several common activation functions and their functionalities.

3.2.2 Loss Function

We have seen how an MLP network can produce a corresponding output $\mathbf{f}$, given an input vector $\mathbf{x}$, along with a weight matrix $\mathbf{W}$ which has been adjusted correctly. Note that the output layer is made up of multiple neurons in such a way that each neuron indicates one of the classes that we want the model to train. Hence, we can consider the output layer as a distribution of the probabilities of the multi classes that we expect the network must predict an example to. For example, if the MLP has two classes, given an input $\mathbf{x}$ the network might predict $\mathbf{f} = [0.1, 0.9]$, meaning that it believes the second class is most probable as the prediction of $\mathbf{x}$. Given a true label, for example $(0, 1)$, we can tell if the trained model made a correct prediction or not. Giving a probabilistic representation of the model outputs, provides a more general perspective of model learning (Witten *et al.*, 2016).

A simple approach to the problem of model parameters estimation is to minimize a loss function. Learning a MLP network is much like learning a linear classifier, one defines a loss

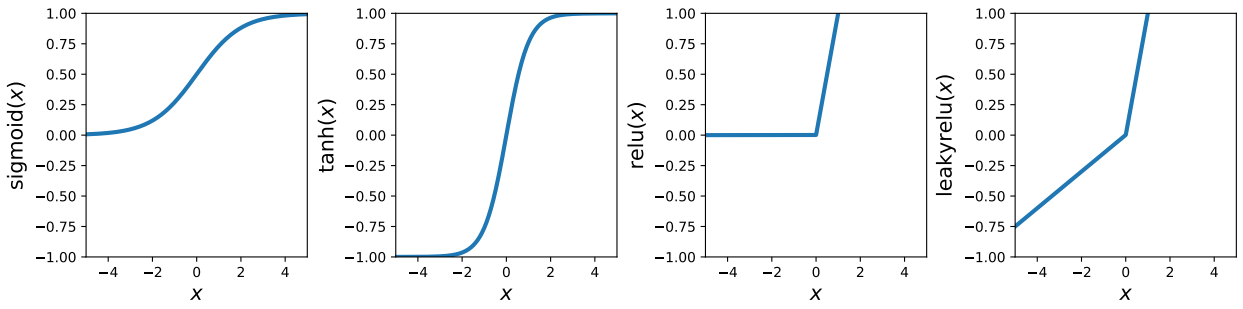


Figure 3.8 Common activation functions

function or cost function $\mathcal{L}(\mathbf{f}, \mathbf{y})$ as a function that represents some "loss" associated with the prediction $\mathbf{f}$ while the actual target is $\mathbf{y}$.

Hence, we need a formulation to understand how good a certain set of the adjusted weight parameters are given the true labels and prediction probabilities. Therefore, for an input vector $\mathbf{x}$, $\mathcal{L}(\mathbf{f}, \mathbf{y})$ is defined as the loss function, where $\mathbf{y}$ is the true target vector and $\mathbf{f}$ denotes the output vector produced at softmax layer in network. Each element in vector $\mathbf{f}$ represents the probability prediction for which the corresponding class belongs to (Witten *et al.*, 2016).

During the backpropagation in MLP, our goal is to minimize a loss function. So far, we have viewed MLP as a general class of parametric nonlinear models which map an input vector $\mathbf{x}$ to an output vector $\mathbf{y}$. One of the most common loss functions employed for classification problems is cross entropy and it is defined in equation (3.37).

3.2.3 Backpropagation Learning Algorithm

A feed forward neural network can essentially be thought of as logistic regression, but with a projection to some hidden layers to render the input more linearly separable. This method was initialized with random weights and was trained via the minimization of backpropagated error using a gradient descent-based method. The backpropagation algorithm is used to find a local optimum of the loss function in an iterative manner. First, the parameters of the model are initialized to some values. Next, the gradient of the loss function with respect to the model parameters is computed and used to update the initial parameters. The principle task is to compute the gradient recursively based on chain rule. Suppose that $\mathcal{L}\{\mathbf{f}(\mathbf{x}; \mathbf{W}), \mathbf{y}\}$ is the loss function of a single-hidden layer MLP with a softmax logistic regression for the multinomial output layer. Multinomial vector $\mathbf{y}$ is represented by a vector wherein the elements in dimension k is 1 for the corresponding class and the other elements indicated by 0. The output function is defined by $\mathbf{f} = [f_1(\boldsymbol{\eta}), ..., f_K(\boldsymbol{\eta})]$ where $\boldsymbol{\eta}(\mathbf{x}; \mathbf{W}) = [\eta_1(\mathbf{x}; \mathbf{w}_1), ..., \eta_K(\mathbf{x}; \mathbf{w}_k)]$ and $\eta_k(\mathbf{x}; \mathbf{w}_k) = \mathbf{w}_k^\top \mathbf{x}$, with the column vector $\mathbf{w}_k$ includes the k^{th} row of the parameter matrix $\mathbf{W}$. Here, the parameter of interest is $\mathbf{W}$ including weights and biases $\mathbf{w_0}$. Therefore, the loss function at softmax output layer will be defined as (Witten *et al.*, 2016)

$$\mathcal{L}(\mathbf{W}) = -\sum_{k=1}^{K} y_k \log f_k(\mathbf{x}),$$

where

$$f_k(\mathbf{x}) = \frac{\exp\{\eta_k(\mathbf{x})\}}{\sum_{c=1}^{K} \exp\{\eta_c(\mathbf{x})\}}. \tag{3.53}$$

Note that $\mathbf{x}$ is augmented by 1 and bias is the last entry in each $\mathbf{w}_k$. Taking the partial derivative of loss function $\mathcal{L}$ with respect to any parameter vector $\mathbf{w}_k$ using the chain rule produces the following equation in reversed form as (Witten *et al.*, 2016)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_k} = \frac{\partial \boldsymbol{\eta}}{\partial \mathbf{w}_k} \frac{\partial \mathbf{f}}{\partial \boldsymbol{\eta}} \frac{\partial \mathcal{L}}{\partial \mathbf{f}} = \frac{\partial \boldsymbol{\eta}}{\partial \mathbf{w}_k} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\eta}}. \tag{3.54}$$

To calculate each element of the gradient vector of loss with respect to vector $\boldsymbol{\eta}$, we have (Witten *et al.*, 2016)

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \eta_j} &= \frac{\partial \left\{ -\sum_{k=1}^{K} y_k \left[\eta_k - \log\{\sum_{c=1}^{K} \exp(\eta_c)\} \right] \right\}}{\partial \eta_j} \\
&= -\left\{ y_{k=j} - \frac{\exp(\eta_{k=j})}{\sum_{c=1}^{K} \exp(\eta_c)} \right\} \\
&= -\left\{ y_j - p(y_j|\mathbf{x}) \right\} \\
&= -\left\{ y_j - f_j(\mathbf{x}) \right\}.
\end{aligned} \tag{3.55}$$

Hence, the gradient of loss with respect to the vector $\boldsymbol{\eta}$ could be shown in the vector form as

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\eta}} = -\left\{ \mathbf{y} - \mathbf{f}(\mathbf{x}) \right\} = \boldsymbol{\Delta}, \tag{3.56}$$

where $\boldsymbol{\Delta}$ is usually referred to as the error (Witten *et al.*, 2016).

Next, for computing the gradient of the vector $\boldsymbol{\eta}$ with respect to the $\mathbf{w}_k$ in vector form, we have

$$\frac{\partial \eta_j}{\partial \mathbf{w}_k} = \left\{ \begin{array}{ll} \frac{\partial \mathbf{w}_k^\top \mathbf{x}}{\partial \mathbf{w}_k}, & ,j = k \\ 0, & , j \neq k \end{array} \right\},$$

which implies that

$$\frac{\partial \boldsymbol{\eta}}{\partial \mathbf{w}_k} = \mathbf{H}_k = \begin{bmatrix} 0 & x_1 & 0 \\ 0 & x_2 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix} \tag{3.57}$$

where the k^{th} column of the above matrix includes the vector $\mathbf{x}$.

To recap, using quantities from above equations the derivative of loss function with respect

to the given parameter $\mathbf{w}_k$ will be calculated by

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_k} = \frac{\partial \boldsymbol{\eta}}{\partial \mathbf{w}_k} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\eta}} = \begin{bmatrix} 0 & x_1 & 0 \\ 0 & x_2 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix} [\mathbf{y} - \mathbf{f}(\mathbf{x})], \tag{3.58}$$

which achieves the gradient of loss with respect to any given parameter vector $\mathbf{w}_k$ that is stored in the k^{th} row of the parameter matrix $\mathbf{W}$. Computing the gradient of the loss with respect to the entire matrix $\mathbf{W}$, will be formulated in a compact form as (Witten *et al.*, 2016)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = -[\mathbf{y} - \mathbf{f}(\mathbf{x})]\mathbf{x}^\top$$

$$= \boldsymbol{\Delta}\mathbf{x}^\top. \tag{3.59}$$

Now, consider a multi-layer network with L hidden layers using the same activation function at each hidden layer and a logistic regression layer with softmax output at the end. The gradient of the k^{th} parameter vector of the output layer will be computed by (Witten *et al.*, 2016)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_k^{(L+1)}} = \frac{\partial \boldsymbol{\eta}^{(L+1)}}{\partial \mathbf{w}_k^{(L+1)}} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\eta}^{(L+1)}}, \tag{3.60}$$

where $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\eta}^{(L+1)}} = -\boldsymbol{\Delta}^{(L+1)}$.

Hence, we have

$$= -\frac{\partial \boldsymbol{\eta}^{(L+1)}}{\partial \mathbf{w}_k^{(L+1)}} \boldsymbol{\Delta}^{(L+1)}$$

$$= -\mathbf{H}_k^{(L)} \boldsymbol{\Delta}^{(L+1)}, \tag{3.61}$$

where $\mathbf{H}_k^{(L)}$ is a matrix consists of the activation functions in layer L in column k and

$$\boldsymbol{\Delta}^{(L+1)} = -[\mathbf{y} - \mathbf{f}(\mathbf{x})], \tag{3.62}$$

represents the error term of the softmax output layer. With a little readjustment, the formulation for computing the gradient of L with respect to entire parameter matrix can be

written as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L+1)}} = -\boldsymbol{\Delta}^{(L+1)} \tilde{\mathbf{h}}_{(L)}^{\top}, \tag{3.63}$$

which includes the error term to be backpropagated across the network layer by layer (Witten *et al.*, 2016).

Therefore, the error backpropagation through the output layer L to layer $l < L$ will be done via a recursive computation as

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{w}_k^{(l)}} &= \frac{\partial \boldsymbol{\eta}^{(l)}}{\partial \mathbf{w}_k^{(l)}} \frac{\partial \mathbf{h}^{(l)}}{\partial \boldsymbol{\eta}^{(l)}} \frac{\partial \boldsymbol{\eta}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\eta}^{(l+1)}} \\
&= -\frac{\partial \boldsymbol{\eta}^{(l)}}{\partial \mathbf{w}_k^{(l)}} \frac{\partial \mathbf{h}^{(l)}}{\partial \boldsymbol{\eta}^{(l)}} \frac{\partial \boldsymbol{\eta}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \boldsymbol{\Delta}^{(l+1)} \\
&= -\frac{\boldsymbol{\eta}^{(l)}}{\mathbf{w}_k^{(l)}} \boldsymbol{\Delta}^{(l)},
\end{aligned} \tag{3.64}$$

where $\boldsymbol{\Delta}^{(l)}$ is determined in terms of $\boldsymbol{\Delta}^{(l+1)}$ as follows

$$\begin{aligned}
\boldsymbol{\Delta}^{(l)} &= \frac{\partial \mathbf{h}^{(l)}}{\partial \boldsymbol{\eta}^{(l)}} \frac{\partial \boldsymbol{\eta}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \boldsymbol{\Delta}^{(l+1)} \\
\boldsymbol{\Delta}^{(l)} &= \mathbf{D}^{(l)} \mathbf{W}^{\top(l+1)} \boldsymbol{\Delta}^{(l+1)}
\end{aligned} \tag{3.65}$$

so that $\mathbf{D}^{(l)} = \frac{\partial \mathbf{h}^{(l)}}{\partial \boldsymbol{\eta}^{(l)}}$ and $\mathbf{W}^{\top(l+1)} = \frac{\partial \boldsymbol{\eta}^{(l+1)}}{\partial \mathbf{h}^{(l)}}$ (Witten *et al.*, 2016).

Note that $\mathbf{D}^{(l)}$ preserves the partial derivative of the activation function at layer l with respect to the inputs of the layer beforehand and it is often a diagonal matrix since the activation functions typically works on an element-wise basis. Furthermore, according to Figure(3.9), $\mathbf{W}^{\top(l+1)}$ could be simply obtained from the definition of $\boldsymbol{\eta}$ at each layer l as $\boldsymbol{\eta}^{(l)}(\mathbf{h}^{(l)}) = \mathbf{W}^{\top(l+1)} \mathbf{h}^{(l)}$.

Consequently, the gradient for the k^{th} vector of parameters of the layer l in a multi-layer network could be written as the product of the following matrices, compactly (Witten *et al.*, 2016)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_k^{(l)}} = -\mathbf{H}_k^{(l-1)} \mathbf{D}^{(l)} \mathbf{W}^{\top(l+1)} \ \ldots \ \mathbf{D}^{(L)} \mathbf{W}^{\top(L+1)} \boldsymbol{\Delta}^{(L+1)}. \tag{3.66}$$

Having above quantities, softmax output function and corresponding loss at logistic regression layer with/without any regularization terms, the MLP model could be formulated and it

could be optimized and fitted using the backpropagation and (stochastic) gradient descent in an iterative manner. The backpropagation could be performed recursively by using the recursive definition of $\boldsymbol{\Delta}^{(l)}$. Figure (3.9) demostrates the error backpropagation computation and parameter updates required for a gradient-based training (Witten *et al.*, 2016).

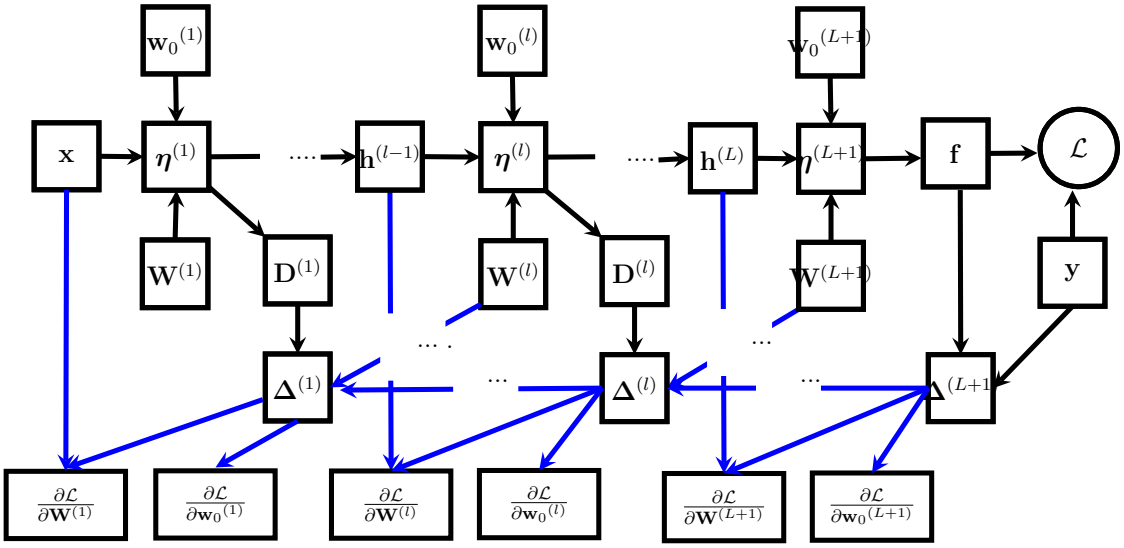


Figure 3.9 Backpropagation in a L layer fully-connected network reproduced from (Witten *et al.*, 2016).

3.2.4 Mini-batch Stochastic Gradient Descent (Mini-batch SGD)

MLPs and deep network models are mostly optimized through mini-batch stochastic gradient descent such that the model parameters will be updated based on the gradients calculated from a small subset of the data called a batch. Mini-batch gradient descent algorithm is a variation of the standard gradient descent algorithm splitting the training data into small mini-batches that are randomly chosen from disjoint partitions of the training data. Algo-

rithm 4 shows a pseudocode for mini-batch SGD in details (Witten *et al.*, 2016).

Algorithm 4: Mini-batch Stochastic Gradient Descent Algorithm (Witten *et al.*, 2016)

Data: Training data $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n]^\top$, $\mathbf{x}_i \subset R^{m+1}$, $\mathbf{Y} = [\mathbf{y}_1, ..., \mathbf{y}_n]$, $\mathbf{y} \subset R^K$, $0 < \gamma \leq 1$, threshold $b < 1$, , converged = True

Result: fitted $\mathbf{W}$

Initialize $\mathbf{W}$;
$t = 0$;
while *not converged* **do**
  Create K' mini-batches as $\mathbf{B} = [B_1, ..., B_{K'}] = \text{Shuffle}(\mathbf{X})$;
  for $k \in \{1, ..., K\}$ **do**
    $\mathbf{g} = \frac{1}{|B_k|} \sum_{i \in B_k} \left[\frac{\partial \mathcal{L}(f(\mathbf{x}_i; \mathbf{W}), \mathbf{y}_i)}{\partial \mathbf{W}} + \frac{|B_k|}{n} \lambda \frac{\partial R(\mathbf{W})}{\mathbf{W}} \right]$;
    $\Delta \mathbf{W} \leftarrow \mathbf{g} + \alpha \Delta \mathbf{W}$;
    $\mathbf{W} \leftarrow \mathbf{W} + \Delta \mathbf{W}$
  end
  t = t + 1
end

The algorithm will use mini-batches B_k include a set of data indexes to compute the error. Then, update model parameters based on the gradient average to reduce the variance of the gradient. The mini-batch SGD works similar to standard SGD ; the process starts by initializing the model parameters, entering a parameter updating loop and is ended by evaluating the validation set. Compared to standard SGD, the iterative process will continue over the mini-batches and the model parameters will be updated after processing each mini-batch. In each iteration, a set of mini-batches are processed that represent the whole training data in one epoch (Witten *et al.*, 2016).

Hence, after a mini-batch processing the model parameters will be updated using the gradient of loss function $\mathcal{L}$ and a regularization term as follows

$$\mathbf{W}(t+1) \leftarrow \mathbf{W}(t) - \gamma \left[\frac{1}{|\mathbf{B}_k|} \sum_{i \in B_k} \left\{ \frac{\partial \mathcal{L}(f(x_i; \mathbf{W}), \mathbf{y}_i)}{\partial \mathbf{W}} \right\} + \frac{|\mathbf{B}_k|}{n} \lambda \frac{\partial R(\mathbf{W})}{\partial \mathbf{W}} \right], \tag{3.67}$$

where γ is the learning rate that could be considered as a small fixed value ; n represents the number of training data ; the k^{th} batch includes $\mathbf{B}_k$ examples and indicated by set $B_k = B(t, k)$ of the indexes that maps them into original training set ; $\mathcal{L}(f(\mathbf{x}_i; \mathbf{W})$ denotes the loss function obtained by batch example $\mathbf{x}_i$, true label $\mathbf{y}_i$ and parameters $\mathbf{W}$; and term $R(\mathbf{W})$ is used for regularizer with weights λ (Witten *et al.*, 2016).

Mini-batch gradient descent algorithm explores a trade-off among the robustness of standard SGD and the efficiency of mini-batch SGD. Most of the common deep learning models use the mini-batch SGD implementation in terms of optimization method. The number of examples in a mini-batch to find one that works well for a given data set could be considered as a hyperparameter of the model and could be tuned through a cross-validation (Witten *et al.*, 2016).

3.2.5 Regularization

In multi-layer networks, weight decay regularization is added to the loss function while computing the backpropagation gradients. The weight decay method is standard for regularization and it is widely used in training other models like linear regression. This method tries to force the model weights toward zero unless there are big gradients to prevent it, which makes corresponding model more interpretable. Note that this regularization method is often applied on the weights parameters not to the biases.

There are two types of weight regularization methods ; L_1 and L_2 norms. L_1 norm is often used in machine learning context, e.g. in regression and it is of the form

$$R_{L_1}(\mathbf{W}) = \sum_{j=0}^{m} |\mathbf{w}_j|. \tag{3.68}$$

It tries to penalize the absolute value of the non-relevant weights by giving a coefficient of zero to these weights. Applying L_1 can lead to sparsity in the model.

By contrary, L_2 norms penalize the square value of the weight which yields to assign all the weights to smaller values close to zero.

$$R_{L_2}(\mathbf{W}) = \sum_{j=0}^{m} \mathbf{w}_j^2. \tag{3.69}$$

In neural networks a weighted combination of L_1 and L_2 norms is added to the loss function at the softmax layer to penalize the weight parameters as follows (Witten *et al.*, 2016)

$$\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f(\mathbf{x}_i, \mathbf{W}), \mathbf{y}_i) + \lambda_1 R_{L1}(\mathbf{W}) + \lambda_2 R_{L2}(\mathbf{W}). \tag{3.70}$$

3.2.6 Dropout

Large and deep networks are typically work so slowly. Dropout is a regularization method for addressing this problem by providing a way of approximately combining exponential number of different neural network models in a reasonable time. The term "dropout" refers to omitting units and temporarily removing them from the model architecture, along with all their incoming and outgoing connections during the training (Hinton *et al.*, 2012). This means that the connections of dropped units to the activation of downstream units is temporally ignored on the feed-forward pass, thus their corresponding parameters are excluded from the updates on the backpropagation and their neighbors cannot rely on theses dropped units in the parameter tuning as well.

Overfitting is another crucial important challenge in deep neural networks with a large number of parameters. The main idea behind dropout strategy is to prevent units from complex co-adapting and therefore counteract overfitting. The overfitting will be significantly decreased by randomly deleting some units on each training case (Srivastava *et al.*, 2014).

Consider a network with L hidden layers. Suppose that the $\mathbf{a}^{(l)}$ indicates the input vector into layer l and the $\mathbf{h}^{(l)}$ represents the corresponding output vector where $\mathbf{h}^{(1)} = \mathbf{x}$.

A feed-forward pass in the network can be shown as (Witten *et al.*, 2016)

$$\boldsymbol{\eta}^{(l+1)} = \mathbf{W}^{(l+1)}\mathbf{h}^{(l)}$$

$$\mathbf{h}^{(l+1)} = \text{act}\{\boldsymbol{\eta}^{(l+1)}\} \quad , \quad l \in \{1, ..., L\}, \tag{3.71}$$

where act(.) could be any activation function. Using the dropout method, the feed-forward pass will be described as

$$r_i^{(l)} \sim \text{Bernoulli}(p)$$

$$\tilde{\mathbf{h}}^{(l)} = \mathbf{r}^{(l)} * \mathbf{h}^{(l)}$$

$$\boldsymbol{\eta}^{(l+1)} = \mathbf{W}^{(l+1)}\tilde{\mathbf{h}}^{(l)}$$

$$\mathbf{h}^{(l+1)} = \text{act}\{\boldsymbol{\eta}^{(l+1)}\}, \tag{3.72}$$

where $\mathbf{r}^{(l)}$ is a Bernoulli random variables vector where each element has probability p of being 1 (Witten *et al.*, 2016). This vector is sampled at each hidden layer and multiplied with the outputs of that layer $\mathbf{h}^{(l)}$ to build a thinned outputs $\tilde{\mathbf{h}}^{(l)}$ and then passed to the next layer as

the input. During the model training, the gradient of the loss function are backpropagated using the thinned network. At test time, the weights are rescaled as $\mathbf{W}_{test}^{(l)} = p\mathbf{W}^{(l)}$. If a unit was kept with probability p at training time, corresponding weights are scaled-down or multiplied by p at test step and the resulting model will run without dropout (Srivastava, 2013). By applying dropout regularization on a network with n units, the model could work like an ensemble of 2^n smaller networks (Witten *et al.*, 2016).

3.2.7 Parameters Initialization

Weight initialization is one of the most important prerequisites in speeding up the training of deep neural networks. Typical gradient descent-based algorithm from random initialization work poorly in deep networks (Glorot et Bengio, 2010). Bias parameters are all typically initialized to zeros but a particular weight initialization value yields the learning to an acceptable approximation or to a false one. For example, initializing the weights to zero while using tanh activation functions at all layers yield a zero gradient with no updates in backpropagation (Witten *et al.*, 2016).

One common solution is to initialize weight matrix at each layer with the following widely-used heuristic :

$$w_{ij} = \mathrm{U}[-q, q], \tag{3.73}$$

where $\mathrm{U}[-q, q]$ denotes the uniform distribution in the interval $(-q, q)$ and q is the number of units in the preceding hidden layer. Several methods have been proposed for choosing the best number for q with the intention of selecting smaller weights for units with more inputs. For example, for a given hidden layer l, one commonly used choice is to consider $\mathrm{U}[-\frac{1}{\sqrt{q}}, \frac{1}{\sqrt{q}}]$, where q denotes the number of units in the preceding layer that equals to the dimensionality of the $\mathbf{h}^{(l-1)}$ (Glorot et Bengio, 2010).

3.3 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are special types of MLPs that are inspired from biology which have proven excellent performance in several machine learning applications and especially in image related tasks.

The input layer receives images which are all size normalized and centered. Each unit in a layer receives inputs from a relatively small spatial zone in the preceding layer (LeCun *et al.*, 1998a). The visual system includes a complex set of cells that are sensitive to small sub-set of the visual field, called a receptive field. Theses sub-sets are tiled across the image to cover

it entirely and behave as local filters in the image. Local receptive fields units can extract useful features such as oriented edges, endpoints and corners from the image. Then, these elementary features are combined by the succeeding layers with the intention of discovering higher-order features (LeCun *et al.*, 1998a). Each filter is mathematically described by a set of small weights that is multiplied by a small spacial area of the given image and then pass it to the feed-forward network. This multiplication is repeated over the image using the same weights. By using CNNs, one can learn a large number of such filters jointly in a data-driven fashion, while learning the other network parameters (Witten *et al.*, 2016).

By applying a filter over an image, the output would be a new modified image including the filter's response at each spatial region. Each filter will be replicated across the entire image while sharing the same parameters at each spatial location. The new image considered as a feature map containing particular features discovered the entire original image, e.g. edges, vertical/horizontal lines or corners (LeCun *et al.*, 1998a).

In a typical CNN, the model architecture built on a series of convolution and subsampling operations, while the last layer of the architecture consists of a fully-connected multi-layer network with a softmax output layer. The subsampling operation performs sampling over every n^{th} output and prevent unnecessary computations by measuring every n^{th} pooling computation.

3.3.1 Convolutional Layer

A deep network takes the original image as input that is often centered and normalized in terms of size. Then each unit in a layer receives inputs from a relatively small spatial region in the preceding layer, so-called receptive field. A generic CNN, so-called LeNet5 (LeCun *et al.*, 1998a), for recognizing digit characters is illustrated in following figure.

Let $\mathbf{x}$ be a simple vector.A filtering operation is performed by multiplying vector $\mathbf{x}$ by a matrix of weight parameters $\mathbf{W}$ composing a particular low-level representations as follows

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

$$= \begin{bmatrix} w_1 & w_2 & w_3 & & & \\ & w_1 & w_2 & w_3 & & \\ & & & \ddots & & \\ & & & w_1 & w_2 & w_3 \end{bmatrix}, \tag{3.74}$$

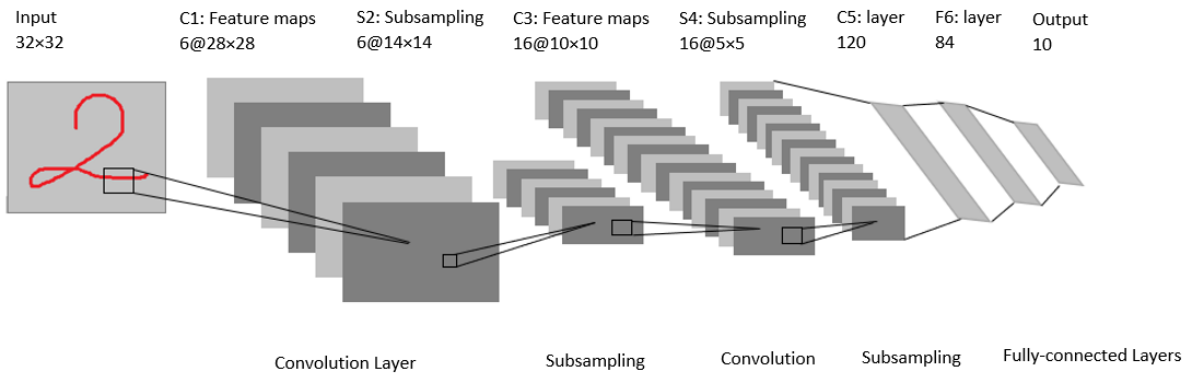


Figure 3.10 A slight architecture of leNet5 as a CNN model reproduced from (LeCun *et al.*, 1998a).

where the empty cells denote zero values (Witten *et al.*, 2016). This matrix shows an example of a filter with three nonzero parameters and stride 1 which means that the filter jumps across the image one pixel at a time. Setting stride to a higher value, e.g. 2, yields producing smaller output images spatially. It is more convenient to apply zero-padding dimensions of the input image by putting zeros around the border such that the spatial size of the filter response and the input image will be the same. By reorganizing the rows of a 2D image compactly in a one column vector, a variant of this 3×3 filter will be obtained by a larger matrix $\mathbf{W}$ where includes two more weight parameter sets in each row. Therefore, the result will be achieved by implementing the summation and multiplication applied by the matrix representation of a 2D filter. The filter response $\mathbf{y}$ would be computed similarly like the cross-correlation operation in Digital Signal Processing context which is similarly related to convolution operation (Witten *et al.*, 2016).

Now, let the filter $\mathbf{W}$ be centered by assigning -1 to the first vector entry or generally $-K$, where K denotes the filter radius. Hence, the 1D filtering operation would be expressed as (Witten *et al.*, 2016)

$$\mathbf{y}[n] = \sum_{k=-K}^{K} \mathbf{W}[k]\mathbf{x}[n+k]. \tag{3.75}$$

Under the assumption of having a 2D image $\mathbf{X}$ and weights $\mathbf{W}$, the filtering generalization would be formulated in a compact form as

$$\mathbf{Y} = \mathbf{W} * \mathbf{X}, \tag{3.76}$$

where $*$ used for convolutional operation and the cross-correlation result for the entry in row

r and column c of the matrix $\mathbf{Y}$ is computed as

$$\mathbf{Y}[r,c] = \sum_{j=-J}^{J} \sum_{k=-K}^{K} \mathbf{W}[j,k]\mathbf{X}[r+j,c+k], \tag{3.77}$$

where J and K are the filters dimensions. Thus, the convolution result will be achieved by (Witten *et al.*, 2016)

$$= \sum_{j=-J}^{J} \sum_{k=-K}^{K} \mathbf{W}[-j,-k]\mathbf{X}[r+j,c+k]. \tag{3.78}$$

Elements of a feature map are supposed to employ the similar filtering operation on different local tiles of the image (Witten *et al.*, 2016). Instead of using predefined weight filters, CNNs tend to jointly learn a large number of convolutional filters in a data-driven manner by backpropagation along with the other model parameters. Different filters could be applied on a spatial location of the image and several feature maps will be produced by sliding different filters across the image. After convolving the image with several filters, the produced feature maps are fed into the consecutive layer in the model (LeCun *et al.*, 1998a).

3.3.2 Max Pooling Layer

Each convolutional layer in a CNN architecture consists of a filter operation over the feature maps received from the preceding layer. Thus, as one operates successively the receptive fields of any unit will increase and high-level layers exploit larger features which are produced by small parts of objects in lower levels and pass much more larger features to the subsequent layers as well. Consequently, a spatial pooling layer will be incorporated within subsequent convolutional layers in a CNN architecture to represent a degree of local spatial invariance in replacement of the locations where corresponding features have been produced. The most widely-used form of the max pooling layer is made up of filters of sizes 2×2 using stride of 2. As a result, CNNs are typically composed of multiple convolution layers followed by max filter and subsampling layer applied on non-overlapping slides of the receptive fields with stride of 2 for stepping across corresponding input. The goal of applying max pooling layer is to downsample corresponding feature map by reducing its spatial size, number of parameters and computations that avoiding overfitting as well. The reduced feature maps generated from the last max pooling and subsampling layer are often passed to a fully-connected neural network, eventually (Witten *et al.*, 2016).

The original image in a numeric form is first convolved with a predetermined filter $\mathbf{W}$ that tiled across the entire image. The generated feature map contains the result of the convolution

operation in the form of matrix. Then, the maximum values within non-overlapping 2×2 regions will be detected. Finally, the results are pooled and downsampled by a factor of 2 to produce the final matrix (Witten *et al.*, 2016).

3.3.3 Loss Function and Gradients

As mentioned earlier, in CNNs instead of using predefined filters one can learn the filter values during the learning of other model parameters through backpropagation. Hence, it is necessary to compute the gradient of the loss function with respect to the entire model parameters.

Therefore, at any layer l in CNN, we have $i = 1, ..., N^{(l)}$ filtered features and their feature maps. The convolutional kernel matrices $\mathbf{K}_i$ including flipped filters regarding kernel weight matrices $\mathbf{W}_i$. Consider act(.) as activation function that operates element-wise function corresponding input matrix received from the preceding layer. For each filter i, there is a scaling parameter g_i. Then, the feature maps are defined as matrices $\mathbf{H}_i(\mathbf{A}_i(\mathbf{X}))$ that could be expressed as a set of feature maps as below (Witten *et al.*, 2016)

$$\mathbf{H}_i = g_i \text{act}(\mathbf{K}_i * \mathbf{X}) = g_i \text{act}(\mathbf{A}_i). \tag{3.79}$$

Thus, the loss function $\mathcal{L}$ at softmax output layer will be defined as a function of the $N^{(l)}$ feature maps at layer l as

$$\mathcal{L} = \mathcal{L}(\mathbf{H}_1^{(l)}, ..., \mathbf{H}_{N^{(l)}}{}^{(l)}), \tag{3.80}$$

where each feature map is related to a layer l. Having activation function with a scale factor of 1 and biases of 0, the gradient of loss function $\mathcal{L}$ with respect to the $\mathbf{X}$ of the convolutional units will be calculates using chain rule as

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{X}} &= \sum_{i=1}^{N^{(l)}} \sum_{j=1}^{J} \sum_{k=1}^{K} \frac{\partial a_{ijk}}{\partial \mathbf{X}} \frac{\partial \mathbf{H}_i}{\partial a_{ijk}} \frac{\partial L}{\partial \mathbf{H}_i} \\
&= \sum_{i=1}^{N^{(l)}} \frac{\partial a_i}{\partial \mathbf{X}} \frac{\partial \mathbf{h}_i}{\partial a_i} \frac{\partial \mathcal{L}}{\partial \mathbf{h}_i} \\
&= \sum_{i=1}^{N^{(l)}} [\mathbf{W}_i * \mathbf{D}_i],
\end{aligned} \tag{3.81}$$

where $\mathbf{h}$, $\mathbf{x}$ and $\mathbf{a}$ are the vectors with stacked columns of the matrices $\mathbf{H}$, $\mathbf{X}$ and $\mathbf{A}$, respec-

tively ; $\mathbf{D}_i = \frac{\partial \mathcal{L}}{\partial \mathbf{A}_i}$ is a matrix including the derivative of the act(.) inputs with respect to the related preactivations for the filter i that are arranged based on the spatial locations (j, k). Consequently, the result will be a summation of convolutions of the filter weights matrix $\mathbf{W}_i$ with a matrix of derivatives $\mathbf{D}_i$. Thus, the gradient of loss with respect to the softmax layer will be achieved by (Witten *et al.*, 2016)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_i} = \sum_j \sum_k \frac{\partial a_{ijk}}{\partial \mathbf{W}_i} \frac{\partial \mathbf{H}_i}{\partial a_{ijk}} \frac{\partial \mathcal{L}}{\partial \mathbf{H}_i}. \tag{3.82}$$

Finally, suppose that a pooling layer is used across a spatially arranged feature map. The input includes matrices $\mathbf{H}_i$ for each feature map with entries h_{ijk}. Therefore, the results of the max pooling operations are pooled layer matrices $\mathbf{P}_i$ with entries p_{ijk} that generated by

$$p_{ijk} = \max_{\substack{r \in R_{j,k} \\ c \in C_{j,k}}} h_{i,r,c}, \tag{3.83}$$

where $R_{j,k}$ and $C_{j,k}$ are index sets representing the pooling areas at each position j and k (Witten *et al.*, 2016). The units that hold the maximum values within each region (j, k) are discovered by

$$\{r^*, c^*\}_{j,k} = \text{argmax}_{\substack{r \in R_{j,k} \\ c \in C_{j,k}}} h_{i,j,k}. \tag{3.84}$$

For non-overlapping regions, the gradient will be backpropagated from the pooled layer $\mathbf{P}_i$ to the preceding feature map $\mathbf{H}_i$, from the p_{ijk} to $\{r^*, c^*\}_{j,k}$ in each region. As a result, the backpropagating the gradient through the max pooling layers yields (Witten *et al.*, 2016)

$$\frac{\partial \mathcal{L}}{\partial h_{i,r_j,c_k}} = \begin{cases} 0 & r_j \neq r_j^*,\ c_k \neq c_k^*, \\ \frac{\partial \mathcal{L}}{\partial P_{i,j,k}} & r_j = r_j^*,\ c_k = c_k^*. \end{cases} \tag{3.85}$$

CHAPTER 4 ADAPTIVE ACTIVATION FUNCTIONS

In Chapter 4, we aim at developing a family of asymmetric and smooth activation functions for Artificial Neural Networks (ANNs). In particular, the goal is to propose an adaptive activation function with a parameter to be controlled and fitted, along with other model parameters in the ANN, e.g. weights and biases. We also indicate that this parametric form of activation functions improve the performance of ANNs in multiple classification tasks.

In Chapter 2, we studied the logistic regression as a special case of GLMs with binomial error distribution and logit link function. Under the assumption of binary response, there are other possible alternatives to the logit link. A strictly monotonic transformation that enables to map the probabilities π_i onto a real number could be employed as a link function. Let $g(.)$ is the c.d.f. of a random variable which is defined on the real line and it is of the form

$$\pi_i = g^{-1}(\eta_i) \quad , \quad -\infty < \eta_i < \infty. \tag{4.1}$$

Now, the inverse transformation could be used as (Rodriguez, 2012)

$$\eta_i = g(\pi_i) \quad , \quad 0 < \pi_i < 1. \tag{4.2}$$

The c.d.f.'s are used as the link functions in binary responses applications because the c.d.f. is always restricted to the interval $[0, 1]$. The most widely-used choice of c.d.f.'s in the case of Bernoulli model is logistic, normal and extreme value distribution Type I. The extreme value distribution is usually referred to as Gumbel distribution.

4.1 Latent Variable Formulation

A GLMs relates the expected value of responses as mean, $\mathbb{E}(y_i)$ and the explanatory variables in a linear system :

$$\eta_i = \mathbb{E}(y_i) = \mathbf{x}_i^\top \mathbf{w}. \tag{4.3}$$

If the dependent variable is on a binary scale, then $\mathbb{E}(y_i)$ would be simply interpreted as the probability of response, π_i and the corresponding linear model could be generalized as following :

$$g(\pi_i) = g\{\mathbb{E}(y_i)\} = \mathbf{x}_i^\top \mathbf{w} \tag{4.4}$$

or simply $g(\pi_i) = \eta_i$, in such a way that $g(.)$, as the link function, is a strictly monotonic differentiable function.

Now let y_i^*, called the latent response, is a continuous random variable and it takes any value in the real line. The latent variable y_i^* is subjected to a certain threshold so that the observed outcome y_i will be one if and only if y_i^* exceeds that threshold. This process in logistic regression, as a GLMs model, leads to finding the probability and takes the form of a logistic c.d.f. The logit model will be motivated as a method of estimating the coefficients from the latent regression of y_i^* on η_i as (Rodriguez, 2012)

$$\begin{cases} 0 & y_i^* > \mathbf{x}_i^\top \mathbf{w}, \\ 1 & y_i^* < \mathbf{x}_i^\top \mathbf{w}. \end{cases} \tag{4.5}$$

Figure (4.1) demonstrates the latent variable associated to logistic distribution and the response when the threshold defined in 0.

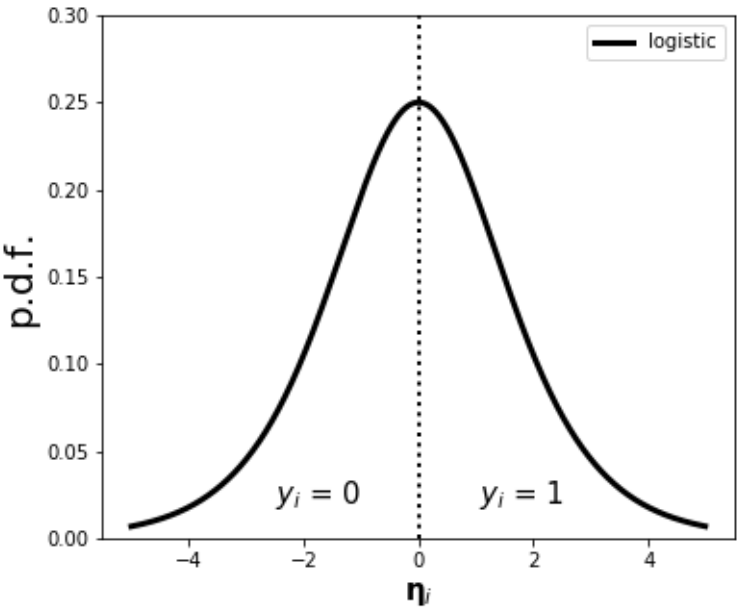


Figure 4.1 Latent variable and random response.

The way we interpret the y_i and y_i^* depends on the context. In a case that y_i viewed as a binary response, a positive outcome will be seen when the latent response exceeds a certain threshold t. Thus, the probability π_i of a positive outcome takes the form

$$\pi_i = p(y_i = 1) = p(y_i^* > t). \tag{4.6}$$

Note that no change will occur in the probability π_i by adding a constant to both y_i^* and t, or multiplying them by a constant. It is assumed that the the threshold t takes the zero value and y_i^* is standardized to have a constant standard deviation (e.g. 1).

Now, let the outcome depends on a vector of predictors $\mathbf{x}$. this dependency could be modeled by an ordinary linear model for the latent variable such that

$$y_i^* = \mathbf{x}_i^\top \mathbf{w} + \varepsilon_i. \tag{4.7}$$

where $\mathbf{w}$ is a vector of coefficients for $\mathbf{x}_i$ and *epsi* is the error supposed to have a distribution with c.d.f. $g(\varepsilon)$. Therefore, the probability of having a positive outcome will be given by :

$$\pi_i = p\{y_i < t = 0\} = p\{\varepsilon_i < \eta_i\} = g^{-1}(\eta_i) \tag{4.8}$$

where $\eta_i = \mathbf{x}_i^\top \mathbf{w}$ is the linear predictor. If the error associated with a symmetric distribution is around zero, this defines a GLMs with Bernoulli response and link shown below

$$\eta_i = g(\pi_i). \tag{4.9}$$

In the cases where the error distribution is not necessarily symmetric, the link could be written as (Rodriguez, 2012)

$$\eta_i = -g^{-1}(1 - \pi_i). \tag{4.10}$$

The common method to model the binomial response is to apply a GLMs such that the latent probability of success will be modeled through a linear function of covariates using a link function. The logit and probit links are two of the widely-used links in GLMs. Compared to other link functions, logit is simpler to interpret since it is defined as log(odds). Moreover, logit link function is a same function as the one used as the canonical link for Bernoulli distribution. Therefore, it is mathematically convenient to justify its application as a link function. The standard normal and logistic distributions usually lead to a same result while they use the same mean and variance parameters and the GLMs with these links will be expected to fit equally well.

In GLMs with Bernoulli error distribution, data could be fitted with either logit or probit transformation. Since a symmetric model is employed, we suppose that the latent probability of a given binomial response approaches 0 and 1 with the same rate. Consequently, the corresponding p.d.f. of logit and probit links is symmetric which is not reasonably a true

assumption in many real-world scenarios. A commonly-known asymmetric link function is the complementary log-log with a constant negative skewness (Rodriguez, 2012).

In next section, we will analyze the binomial response problem with complementary log-log transformation.

4.2 The complementary log-log transformation

A commonly-used choice as an alternative to logit link is complementary log-log which is defined as

$$\eta_i = g(\pi_i) = \log\{-\log(1-\pi_i)\}. \tag{4.11}$$

Similar to the logit and probit probability transformation, the complementary log-log takes a value bounded above by 1 and below by 0 and maps it onto the real line. The complementary log-log transformation is the inverse function of the c.d.f. of the extreme value Type I or Gumbel distribution with c.d.f.

$$\pi_i = g^{-1}(\eta_i) = 1 - \exp\left\{-\exp(\eta_i)\right\}. \tag{4.12}$$

The Gumbel distribution has two forms : minimum and maximum (Kececioglu, 1991). The former is based on the smallest and the latter is based on the largest extreme. Typically, the probability density function (p.d.f) of the Gumbel distribution takes the form

$$f(x) = \frac{1}{\sigma}\exp\left(\frac{x-\mu}{\sigma}\right)\exp\left\{-\exp\left(\frac{x-\mu}{\sigma}\right)\right\}, \tag{4.13}$$

where μ and σ denote the location and scale parameters, respectively. The p.d.f. with $\mu = 0$ and $\sigma = 1$ is called standard Gumbel distribution. Considering the equation (4.3), now suppose that we have

$$\eta_i = \mathbf{x}_i^\top \mathbf{w} = \mathbf{w}_0 + \mathbf{x}_i^\top \mathbf{w}_1. \tag{4.14}$$

Thus, the equation (4.12) gives us (Rodriguez, 2012)

$$\pi_i = 1 - \exp\left\{-\exp(\mathbf{w}_0 + \mathbf{x}_i^\top \mathbf{w}_1)\right\}. \tag{4.15}$$

The Gumbel distribution has an s-shaped response curve that approaches 0 moderately slowly while approaching 1 so sharply with $\mathbf{w}_1 > 0$, see following figure (left).

While the complementary log-log link function is employed to model π as the probability of a success, there is another alternative that could be used to model the failure called log-log link function which is of the form (Rodriguez, 2012)

$$\pi_i = g^{-1}(\eta_i) = \exp\left\{-\exp(\mathbf{w}_0 + \mathbf{x}_i^\top \mathbf{w}_1)\right\}. \tag{4.16}$$

In Figure (4.2), a demonstration of complementary log-log and log-log inverse of the links are presented. For $\mathbf{w}_1 > 0$, log-log link approaches 1 fairly slowly while approaching 0 quite sharply. Moreover, by increasing the explanatory variable, the log-log curve is monotonically decreasing but it is monotonically increasing for $\mathbf{w}_1 < 0$. Log-log transformation is rarely employed as a link function since it works poorly in most applications (McCullagh et Nelder, 1989). Therefore, it is excluded from our work.

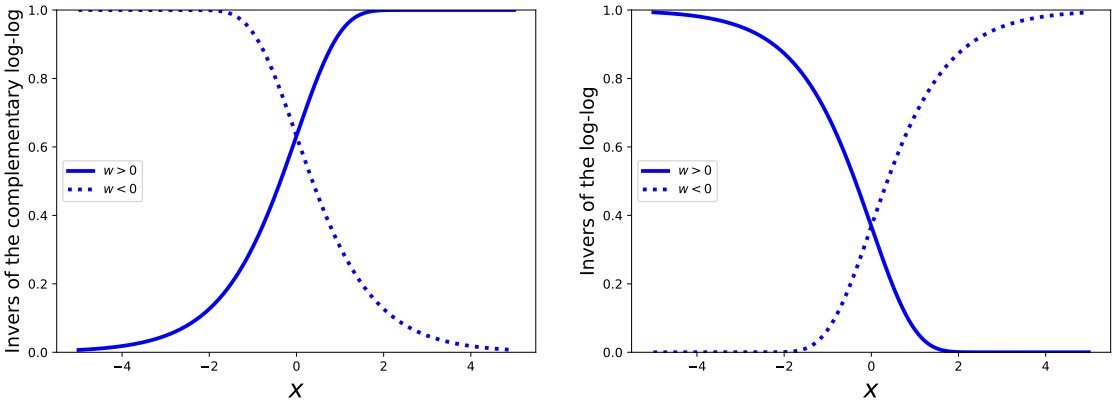


Figure 4.2 The inverse of the link functions for : (left) complementary log-log and (right) log-log.

Figure (4.3) demonstrates the comparison of the p.d.f, link and c.d.f. functions of the three commonly-used functions for binomial error distribution in GLMs : logit, probit and complementary log-log. One should keep in mind that to distinguish the difference between the behavior of logit and probit link functions in real scenarios, one needs extremely large data samples (Chambers et Cox, 1967). The logit, probit, and complementary log-log link functions are increasingly monotonic, differentiable, and one-to-one in $[0, 1]$ interval. Logit and probit link functions are symmetric around zero which means that they behave similarly when $g(x) = g(-x)$. In other words, $g(.)$ function approaches to 0 with a same rate as approaching to 1. Thus, to take the reverse function of the dependent variable, only a change in the coefficient signs gives us the result. For the complementary log-log link with a fixed negative skewness, the result will be completely different. The shape of the logistic distribution and

normal distribution are very similar, except that logistic shape has heavier tails. One may interpret the coefficients in a logit transformation as the effects of the covariates on a latent variable with the assumption that the error in linear model has a logistic distribution. To compare the link functions, the logit and probit transformations are linear functions with respect to π in the range of 0.1 and 0.9. Under the assumption that g is asymmetric in the interval $[0, 1]$, logit and probit increase slowly to a moderate value while approaching 1 sharply. By contrast, as the probability g increases, the complementary log-log approaches ∞ slower than logit and probit link functions. When g takes a small value, the complementary behaves close to logit link with significantly less heavy right tail. Unlike the logit and probit, complementary log-log is often employed when the probability π is either too small or too large (Agresti, 2003).

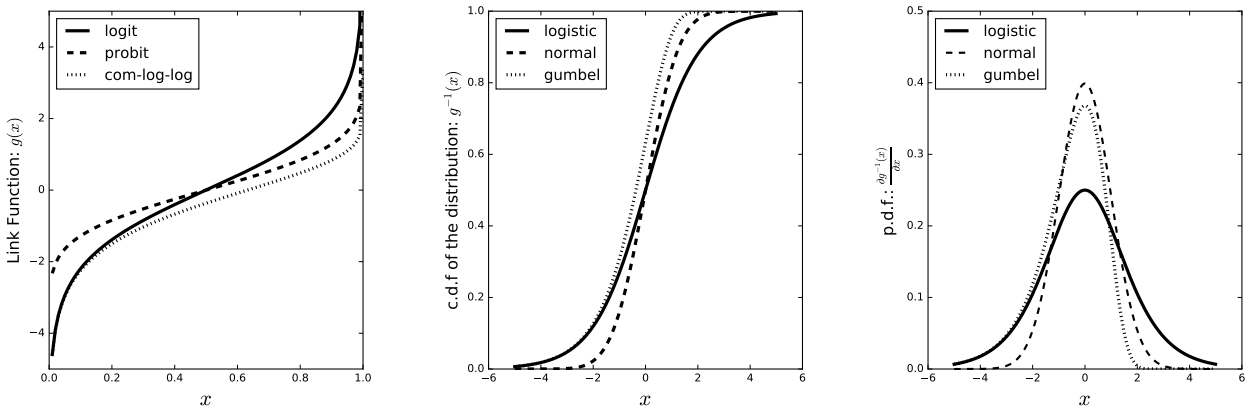


Figure 4.3 Left : a comparison of the logit, probit and comp-log-log link functions, g is shown. In middle, graph compares the c.d.f. of Logistic, Normal and Gumbel distribution as g^{-1}. Right-side graph gives a demonstration of the p.d.f. of the Logistic, Normal and Gumbel distribution.

4.2.1 Maximum Likelihood Estimates with Complementary log-log

In Section 2.2.8, a maximum likelihood estimate using the IRLS algorithm was shown for logistic regression under the assumption of having binomial error distribution with the logit link function. Similarly, an IRLS approach could be employed to fit the parameters of the Bernoulli regression model with a complementary log-log link function. Given a parameter of the estimation $\hat{\mathbf{w}}$, the linear predictor $\hat{\eta}_i = \text{complementarylog} - \log(\pi_i) = \mathbf{x}_i^\top \mathbf{w}$ and the fitted values $\hat{\mu}_i = g^{-1}(\eta_i) = \text{Gumbel}(\eta_i)$ could be obtained, consequently. Under the assumption of having these quantities, the working dependent variable, z_i, and the iteration weight, ω_i, are

identified to estimate the parameters of the Bernoulli model. Suppose the model is defined by

$$\eta_i = g(\pi_i) = \log\{-\log(1-\pi_i)\}. \tag{4.17}$$

The link function could be shown more conveniently as a function of μ_i, thus, we have :

$$\eta_i = \log\left\{-\log\left(1-\mu_i\right)\right\}. \tag{4.18}$$

Taking the derivative of η_i with respect to the parameter μ_i yields

$$\frac{\partial \eta_i}{\partial \mu_i} = \frac{-1}{(1-\mu_i)\log(1-\mu_i)}$$

$$= \frac{-1}{(1-\pi_i)\log \pi_i}, \tag{4.19}$$

Hence, the working dependent variable is

$$z_i = \hat{\eta}_i + (y_i - \hat{\mu}_i)\frac{\partial \eta_i}{\partial \mu_i}$$

$$= \hat{\eta}_i + (y_i - \pi_i)\left\{\frac{-1}{(1-\pi_i)\log \pi_i}\right\}. \tag{4.20}$$

and the iteration weight will be of the form

$$\omega_i = \frac{p_i}{b''(\theta_i)(\frac{\partial \eta_i}{\partial \mu_i})^2}$$

$$= \frac{-(1-\pi_i)\log(\pi_i)^2}{\pi_i} \tag{4.21}$$

where $p_i = 1$ and $b''(\theta_i) = \pi_i(1-\pi_i)$.

Consequently, the estimates of the parameter $\mathbf{w}$ by regressing the working dependent variable z_i on the predictors x_i will be obtained as (2.52) in an iterative fashion and the iterations with corresponding updates will be repeated until the convergence (see Chapter 2).

4.3 Adaptive Sigmoid Link Function

The complementary log-log transformation is an asymmetric link with fixed negative skewness. It suffers from the lack of flexibility to allow the data to adjust the amount of skewness required. Furthermore, there is no ability to have the positive skewness in corresponding p.d.f. As a result, binary response regression might be better fitted with an asymmetric link function and flexible and adjustable skewness might be a better alternative for modeling the binary response problem. Hence, we aim at developing such link function so that allows the data to determine how much skewness should be included, positive, negative or even zero skewness (symmetric link).

To introduce such flexibility into the link functions, we propose a one-parametric model for skewness-flexibility in the Gumbel distribution using the inverse of log Box–Cox transformation (Sakia, 1992) on the $\frac{1}{1-x}$ to form an adaptive function of standard Gumbel that could approximate both symmetric logistic and asymmetric Gumbel (with either positive or negative skewness) quite well, depends on the value of the parameter. Our most important goal is to propose an adaptive link function to allow more adoptable skewness controlled by a parameter and includes both standard logistic and Gumbel as well.

We first apply the Box-Cox transformation on variable x which obtains

$$\begin{cases} \frac{\left(\frac{1}{1-x}\right)^{\alpha}-1}{\alpha} & \alpha \neq 0, \\ \log\left(\frac{1}{1-x}\right) & \alpha = 0. \end{cases} \tag{4.22}$$

Taking the logarithm function of the Box-Cox transformation

$$\begin{cases} \log\left\{\frac{(\frac{1}{1-x})^{\alpha}-1}{\alpha}\right\} & \alpha \neq 0, 1, \\ \log(\frac{1}{1-x}-1) = \log\frac{x}{1-x} & \alpha = 1, \\ \log\{\log(1-x)^{-1}\} = \log\{-\log(1-x)\} & \alpha = 0, \end{cases} \tag{4.23}$$

or simply

$$\begin{cases} \log\left\{\frac{(\frac{1}{1-x})^{\alpha}-1}{\alpha}\right\} & \alpha \neq 0, 1 \\ \text{logit}(x) & \alpha = 1 \\ \text{complementary} - \log - \log(x) & \alpha = 0. \end{cases} \tag{4.24}$$

For $\alpha = 0$ and $\alpha = 1$, we have the standard logit and complementary log log transformation as seen before. Any value not equal to zero and one leads to a parametric transformation of

x.

Figure below compares the shape of the logit, complementary log-log, probit and Log Box-Cox link function. All functions are alike g^{-1} in the interval $[0, 1]$. For small values of g, the logarithm of Box-Cox is close to the logit and also behaves similar to the behavior of complementary log-log. As x approaches 1, the logarithm of Box-Cox function approaches infinity much more slowly than logit but more sharply than probit and complementary log-log.

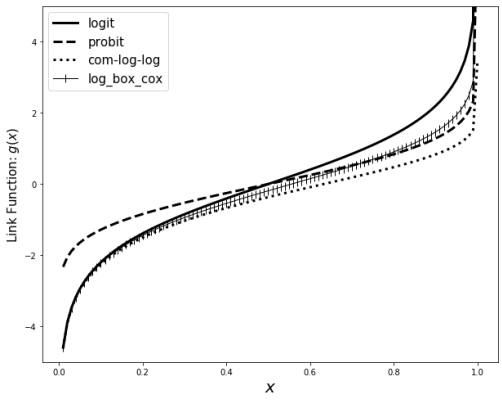


Figure 4.4 A comparison of link functions.

Now, the inverse function of (4.24) can be expressed as

$$\begin{cases} 1 - \frac{1}{\{1+\alpha \exp(x)\}^{\frac{1}{\alpha}}} & \alpha \neq 0, 1, \\ \frac{1}{\{1+\exp(-x)\}} & \alpha = 1, \\ 1 - \exp\{-\exp(x)\} & \alpha = 0. \end{cases} \tag{4.25}$$

Consequently, to define π_i we could use the definition of the adaptive-sigmoid equivalently as

$$\pi_i = g^{-1}(\eta_i) = \begin{cases} 1 - \frac{1}{\{1+\alpha \exp(\mathbf{x}_i^\top \mathbf{w})\}^{\frac{1}{\alpha}}} & \alpha \neq 0, 1, \\ \frac{1}{1+\exp(-\mathbf{x}_i^\top \mathbf{w})} & \alpha = 1, \\ 1 - \exp\Big\{ -\exp\Big(\mathbf{x}_i^\top \mathbf{w}\Big)\Big\} & \alpha = 0. \end{cases} \tag{4.26}$$

with an interesting property so that with $a = 0$ and $a = 1$ the equation converges to Gumbel and logistic distributions, respectively. This function will be called adaptive-sigmoid.

Similarly, it is easy to show that for the α close to 1, $g(.)$ approaches the logistic. Due to the fact that $\{1 + \alpha \exp(.)\}$ must necessarily be positive and the domain of the adaptive distri-

bution will be changed by choosing negative values for α, therefore, the adaptive distribution for $\alpha < 0$ will be defined as

$$\pi_i = \begin{cases} 1 - \frac{1}{\{1+a\exp(\mathbf{x}_i^\top \mathbf{w})\}^{\frac{1}{\alpha}}} & \mathbf{x}_i^\top \mathbf{w} < \log(\alpha) \ , \\ 1 & \text{otherwise.} \end{cases} \tag{4.27}$$

As a result, the adaptive distribution includes both standard forms of logistic and Gumbel distributions for $\alpha = 1$ and $\alpha = 0$, respectively. Adaptive link functions, referred to as adaptive-sigmoid, with different values of α are illustrated in Figure (4.5). In Figure (4.5), curves with solid line show a special case of the Gumbel distribution while $\alpha \to 0$ and curves in dotted style, demonstrate the standard logistic distributions when $\alpha = 1$. The other curves show the adaptive-sigmoid function with different values of parameter α. The p.d.f. of the adaptive sigmoid is illustrated in middle graph which demonstrates a nice view of the skewness of the adaptive sigmoid distribution induced by parameter α.

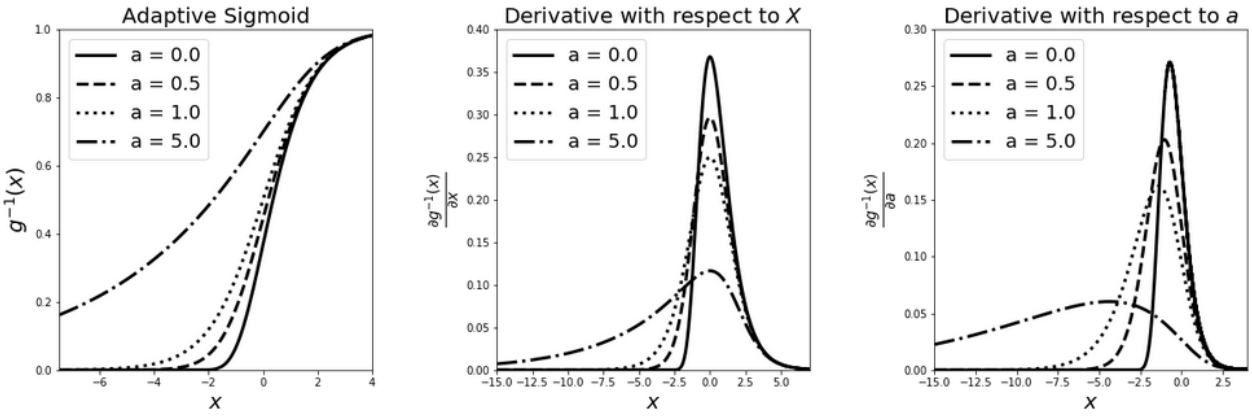


Figure 4.5 Adaptive sigmoid and its derivatives with respect to parameters x (or p.d.f.) and for different values of α.

As obviously understood from the middle plot, parameter α, appeared in the above distribution, could be used to tune the skewness parameter, as we shall show later in sequel.

4.3.1 Identifiability

Before the discussion about the maximum likelihood estimates of the model, the identifiability property of the model with respect to the parameter α must be verified. A family of distributions with probability density function $f(x;\omega), \omega \in \Omega$, is said to be identifiable if, for

any ω_1 and ω_2 in the parameter space Ω, we have (Huang, 2005) :

$$f(x, \omega_1) = f(x, \omega_2) \leftrightarrow \omega_1 = \omega_2, \tag{4.28}$$

where $\omega = (x, \alpha)$. This definition also means that distinct values of parameter ω should result in distinct functions as :

$$\omega_1 \neq \omega_2 \leftrightarrow f(x, \omega_1) \neq f(x, \omega_2), \tag{4.29}$$

or the mapping from parameters to functions is one-to-one.

Our goal is to show the identifiability of the link function in the Bernoulli model.

Lemma 4.3.1. *Under the assumption of having a Bernoulli random variable with probabilities $p(y_i = 1) = \pi_i$ and $p(y_i = 0) = 1 - \pi_i$, we have to show the identifiability of the proposed link function for the corresponding model.*

Proof. We have the following probability distribution function

$$p\{y_i\} = \pi_i^{y_i}(1 - \pi_i)^{(1-y_i)} \quad , \quad y_i = \{0, 1\},$$

where π_i is a function of parameters $\omega = (x, \alpha)$ and parameter α is supposed to represent a non-negative value. π_i is referred to as $P_{x,\alpha}$ which yields

$$y_i \left\{ \log \frac{P_{x,\alpha}}{1 - P_{x,\alpha}} - \log \frac{P_{x,\alpha'}}{1 - P_{x,\alpha'}} \right\} + \log \left\{ \frac{1 - P_{x,\alpha}}{1 - P_{x,\alpha'}} \right\} = 0. \tag{4.30}$$

Equating the above polynomial to zero for all its positive factors, requires that each coefficient be equal zero, individually. Hence, we have to show $P_{x,\alpha} = P_{x,\alpha'}$, $\forall \alpha = \alpha'$ and $P(.) = 1 - \frac{1}{\{1+\alpha \exp(x)\}^{1/\alpha}}$.

To show the $P_{x,\alpha} = P_{x,\alpha'}$, we have to solve the equation below

$$1 - \frac{1}{\{1 + \alpha \exp(x)\}^{\frac{1}{\alpha}}} = 1 - \frac{1}{\{1 + \alpha' \exp(x)\}^{\frac{1}{\alpha'}}}$$

$$\Rightarrow \{1 + \alpha \exp(x)^{\frac{1}{\alpha}}\} = \{1 + \alpha \exp(x)^{\frac{1}{\alpha'}}\}. \tag{4.31}$$

Taking the logarithm from both sides of the above expression gives us

$$\Rightarrow \frac{1}{\alpha}\log\{1+\alpha\exp(x)\} = \frac{1}{\alpha'}\log\{1+\alpha'\exp(x)\}. \tag{4.32}$$

Now, let $z = \exp(x)$. By replacing z in the above equation, we have

$$\frac{1}{\alpha}\log(1+\alpha z) = \frac{1}{\alpha'}\log(1+\alpha' z), \tag{4.33}$$

where $f(z,\alpha) = \log(1+\alpha z)$. If we could show $f(z,\alpha) = f(z,\alpha') \leftrightarrow \alpha = \alpha'$, then the proof of identifiability will be accomplished, successfully. To do this, the Taylor series expansion of both sides of this equation should be written as follows

$$\frac{1}{\alpha}\left\{\sum_{n=0}^{\infty}\frac{f_z^{(n)}(0)\alpha^n}{n!}\right\} = \frac{1}{\alpha'}\left\{\sum_{n=0}^{\infty}\frac{f_z^{(n)}(0)\alpha'^n}{n!}\right\}$$

$$\Rightarrow \frac{f_z^{(0)}(0)\alpha^0}{0!} + \left\{\sum_{n=1}^{\infty}\frac{f_z^{(n)}(0)\alpha^{n-1}}{n!}\right\} = \frac{f_z^{(0)}(0)\alpha'^0}{0!} + \left\{\sum_{n=1}^{\infty}\frac{f_z^{(n)}(0)\alpha'^{n-1}}{n!}\right\}. \tag{4.34}$$

This equation is true for almost all z only when all corresponding coefficients are equal, which is only possible when

1)

$$f_z{}'(\alpha) = f_z{}'(\alpha') \Rightarrow \frac{\partial\log(1+\alpha z)}{\partial\alpha} = \frac{\partial\log(1+\alpha' z)}{\partial\alpha'} \Rightarrow \alpha = \alpha',$$

2)

$$f_z{}''(\alpha) = f_z{}''(\alpha') \Rightarrow \frac{\partial^2\log(1+\alpha z)}{\partial\alpha^2} = \frac{\partial^2\log(1+\alpha' z)}{\partial\alpha'^2}$$

$$\Rightarrow \frac{-z}{(1+\alpha z)^2} = \frac{-z}{(1+\alpha' z)^2} \Rightarrow (1+\alpha z) = (1+\alpha' z) \Rightarrow \alpha = \alpha',$$

$\vdots$

and

n)

$$\frac{(n-1)!(-1)^{n-1}\alpha^n}{(1+az)^n} = \frac{(n-1)!(-1)^{n-1}\alpha'^n}{(1+\alpha' z)^n}$$

$$\Rightarrow \frac{\alpha}{(1+\alpha z)} = \frac{\alpha'}{(1+\alpha' z)}$$

$$\Rightarrow \alpha + \alpha\alpha' z = \alpha' + \alpha\alpha' z$$

$$\Rightarrow z(\alpha\alpha' - \alpha\alpha') + (\alpha - \alpha') = 0$$

$$\alpha = \alpha'. \tag{4.35}$$

Therefore, the the parameter α of adaptive-sigmoid is identifiable in the Bernoulli model.

4.3.2 Maximum Likelihood Estimates for Adaptive Sigmoid Link Function

Since the response variable in our problem is known and the parameter of the link function is identifiable, the method of maximum likelihood could be employed to estimate the model parameters. As described before, the errors in our desired model is in the form of Bernoulli distribution with mean zero and fixed variance σ^2. Hence, the observations $\mathbf{y}$ are from Bernoulli distribution with mean $\mathbf{X}^\top\mathbf{w}$ and variance $\sigma\mathbf{I}$. Thus the maximum likelihood function could be obtained by the joint probability distribution of the observed data. Recall that the maximum likelihood function is equivalent to maximizing the log likelihood which is of the form (2.63) which aims at estimating the values of parameters $\mathbf{w}$ and σ^2. Taking the partial derivatives of the log-likelihood against the parameters and equating to zero yields the estimator $\hat{\mathbf{w}} = (\mathbf{X}^\top\mathbf{\Omega}\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{\Omega}\mathbf{z}$.

Given a parameter of the estimation $\hat{\mathbf{w}}$, the linear predictor $\hat{\eta}_i = \mathbf{x}_i^\top\mathbf{w}$ and the fitted values $\hat{\mu}_i = \text{adaptive} - \text{sigmoid}(\eta_i)$, where the adaptive-sigmoid is defined in (4.26).

Having these quantities, the working dependent variable, z_i, and the iteration weight, w_i, are determined to estimate the parameters of the binomial regression. Suppose the model is defined by

$$\eta_i = \log\left\{\frac{(\frac{1}{1-\pi_i})^\alpha - 1}{\alpha}\right\},$$

and

$$\pi_i = g^{-1}(\eta_i) = 1 - \frac{1}{\{1 + \alpha\exp(\eta_i)\}^{\frac{1}{\alpha}}}. \tag{4.36}$$

Since it is easier to show the link function as a function of μ_i, we have :

$$\eta_i = \log(\frac{1}{1-\mu_i})^\alpha - 1 - \log\alpha. \tag{4.37}$$

Taking the derivative of the η_i with respect to parameter μ_i yields

$$\frac{\partial \eta_i}{\partial \mu_i} = \frac{\alpha(\frac{1}{1-\mu_i})^{\alpha}}{(1-\mu_i)\{(\frac{1}{1-\mu_i})^{\alpha}-1\}}$$

$$= \frac{\alpha(\frac{1}{1-\pi_i})^{\alpha}}{(1-\pi_i)\{(\frac{1}{1-\pi_i})^{\alpha}-1\}}$$

$$= \frac{\alpha}{(1-\pi_i)\left\{1-(1-\pi_i)^{\alpha}\right\}}. \tag{4.38}$$

Hence, the working dependent variable is

$$z_i = \hat{\eta}_i + (y_i - \hat{\mu}_i)\frac{\partial \eta_i}{\partial \mu_i}$$

$$= \hat{\eta}_i + (y_i - \pi_i)\{\frac{\alpha}{(1-\pi_i)[(1-\pi_i)^{\alpha}-1]}\} \tag{4.39}$$

and the iteration weight will be of the form

$$\omega_i = \frac{p_i}{b''(\theta_i)(\frac{\partial \eta_i}{\partial \mu_i})^2}$$

$$= \frac{(1-\pi_i)\left\{1-(1-\pi_i)^{\alpha}\right\}^2}{\pi_i \alpha^2}, \tag{4.40}$$

where $p_i = 1$ and $b''(\theta_i) = \pi_i(1-\pi_i)$.

Finally, the estimates of the parameter $\mathbf{w}$ by regressing the working dependent variable z_i on the predictors $\mathbf{x}_i$ using the weights ω_i could be determined as in equation (2.52) in an iterative way using the IRLS algorithm. The iterations and updating weights will be repeated until convergence (see Section 2.2.7).

4.3.3 ANNs with Adaptive-sigmoid : Loss Function and Optimization

The multiple output linear regression model with a K-dimensional output is a linear mapping from an input vector

$$\mathbf{x} = [x_0, ..., x_m]^{\top}$$

to a vector of responses $\mathbf{y}$ through a set of $(m+1) \times K$-dimensional coefficients $\mathbf{W}$ matrix, including the biases. This linear mapping is modeled by an error term ε_i normally distributed and independent that adds a Gaussian noise of unknown variance σ^2 to the linear relationship between the response variables and covareiates $\mathbf{x}$. Hence, the model will take the form below

$$\boldsymbol{\eta} = \mathbf{x}^\top \mathbf{W}$$

and

$$y_i = \eta_i + \varepsilon_i \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2), \tag{4.41}$$

where $\boldsymbol{\eta}$ denotes the systematic component of the linear model and $\boldsymbol{\varepsilon}$ is the random component.

In GLMs, the response variables are allowed for error distribution models other than a normal distribution and typically, belongs to the exponential family. In fact, in the GLMs, the linear model is related to the response variable through a link function and the variance of each measurement is defined as a function of the mean.

Thus, in GLMs, we have

$$\mathbb{E}[y_i] = \mu_i = g^{-1}(\eta_i), \tag{4.42}$$

where $g(.)$ is considered as the link function that allows us to define the parameters η_i as a function of mean μ_i, more conveniently. If the logit function is used as the link function, then the inverse link function in the definition of mean will be the logistic sigmoid function and consequently, the mean parameters corresponding to the probabilities of $\mathbf{y}$ takes either 0 or 1, under the Binomial distribution. In other words, GLMs takes the covariates $\mathbf{x}$, applying a linear combination of $\mathbf{x}$ and coefficients $\mathbf{W}$ and proceeds the result by a nonlinear transformation (the inverse link function) as a single building block.

In ANNs, the inverse link function is referred to as the activation function $\text{act}(\boldsymbol{\eta}^l)$ and the process of GLMs building block could be repeated in each hidden layer and produces the corresponding outcome $\mathbf{h}^l$ as

$$\mathbf{h}^l = \text{act}(\boldsymbol{\eta}^l), \tag{4.43}$$

where $\boldsymbol{\eta}^l = \mathbf{h}^{l-1^\top}\mathbf{w}^l$ and $\mathbf{h}^{l-1^\top}$ and $\mathbf{w}^l$ are the inputs and coefficients of the hidden layer $l-1$ and l, respectively. As a result, a recursive GLMs could be identified by applying the above building block recursively as follows :

$$\boldsymbol{\mu}^{L+1} = \mathbf{h}^{L+1} \circ \mathbf{h}^L \circ ... \mathbf{h}^l \circ ... \mathbf{h}^1 \circ \mathbf{h}^0(\mathbf{x}) \tag{4.44}$$

which is exactly the model architecture of an L-layer MLP model and an ANN could be considered as a recursive GLMs model (Ranganath *et al.*, 2015; Mohamed, 2015).

As described in Chapter 3, logistic sigmoid and hyperbolic tangent are two widely-used nonlinear functions that applied as activation functions in ANNs and play a crucial role in the convergence and performance of the ANNs. An ANN model could be employed as an alternative way to solve binomial regression models. Specifically for the binary response model which is considered as a specific case GLMs.

Our motivation is to broaden the range of activation functions for ANNs using the inverse of the adaptive-sigmoid function as the inverse of the log Box-Cox of $\frac{1}{1-x}$. Our objective is to implement the adaptive-sigmoid as a nonlinear function in the processing units of the ANNs. Adaptive-sigmoid parameter α and the ANNs model parameters will be learned simultaneously during training step and each hidden neuron would be enabled to learn its own adaptive activation function. Thus, the amount of the skewness and consequently the symmetry in each neuron could be adjusted in a data-driven learning way.

As seen in Chapter 3, the logit, probit, complementary log-log and log Box-Cox link functions are almost linearly related in the interval $[0.1, 0.9]$. Also, this figure indicates that the adaptive-sigmoid function is a nonconstant, monotonically-increasing, continuous, and differentiable function which is bounded to $[0, 1]$. Hence, adaptive-sigmoid satisfies the requirements of the UAT for being employed as an activation function in ANNs.

Now, consider a MLP using the adaptive-sigmoid function for L hidden layers and a softmax regression layer at the end. According to the formulations and notations presented in Chapter 3, the gradient of softmax loss function ($\mathcal{L}$) with respect to the the weight parameters remains same. The gradients for the k^{th} parameter vector of layer l could be calculated by

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k^l} = -\frac{\partial \boldsymbol{\eta}^l}{\partial \boldsymbol{\theta}_k} \boldsymbol{\Delta}^l,$$

where

$$\boldsymbol{\theta}_k^l = [\mathbf{W}_k^l, \mathbf{w_0}_k^l, \boldsymbol{\alpha}_k^l],$$

$$\boldsymbol{\Delta}^l = \frac{\partial \mathbf{h}^l}{\partial \boldsymbol{\eta}^l} \frac{\partial \boldsymbol{\eta}^{l+1}}{\partial \mathbf{h}^l} \boldsymbol{\Delta}^{l+1}, \tag{4.45}$$

and α is the parameter of the adaptive-sigmoid function. Therefore, the gradient for each parameter could be obtained as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_k^l} = -\mathbf{h}^{l-1} \boldsymbol{\Delta}^l,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_{0_k}^l} = -\boldsymbol{\Delta}^l$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\alpha}_k^l} = -\frac{\partial \boldsymbol{\eta}^l}{\partial \boldsymbol{\alpha}_k^l}\boldsymbol{\Delta}^l$$

$$= \frac{\mathbf{W}_k^l}{[1+\boldsymbol{\alpha}\exp(\mathbf{h}^{l-1})]^{\frac{1}{\alpha}}}\left[\frac{-\exp\{\mathbf{h}^{l-1}\}}{\boldsymbol{\alpha}\{1+\boldsymbol{\alpha}\exp(\mathbf{h}^{l-1})\}} + \frac{\log\left\{1+\boldsymbol{\alpha}\exp(\mathbf{h}^{l-1})\right\}}{\boldsymbol{\alpha}^2}\right]\boldsymbol{\Delta}^l. \tag{4.46}$$

Also, the parameters typically will be updated using (stochastic) gradient descent by the following update equations :

$$\mathbf{W}_k^l \leftarrow \mathbf{W}_k^l - \gamma\frac{\partial \mathcal{L}}{\partial \mathbf{W}_k^l},$$

$$\mathbf{w}_{0_k}^l \leftarrow \mathbf{w}_{0_k}^l - \gamma\frac{\partial \mathcal{L}}{\partial \mathbf{w}_{0_k}^l}$$

and

$$\boldsymbol{\alpha}_k^l \leftarrow \boldsymbol{\alpha}_k^l - \gamma\frac{\partial \mathcal{L}}{\partial \boldsymbol{\alpha}_k^l}, \tag{4.47}$$

where γ is the learning rate and takes a small fixed value.

Having above quantities, softmax output function and corresponding loss at logistic regression layer with/without any regularization terms, the MLP model could be formulated. Then, it could be optimized and fitted using backpropagation with (stochastic) gradient descent in an iterative manner.

4.4 Adaptive ReLu Function

In this Section, we suggest a smooth form of standard relu function that is inspired from the approximation of Heaviside function. Our aim is to apply an adaptable parameter in the form of relu activation function in such a way that it could be tuned along with other ANN model parameters in backpropagation.

Consider the Heaviside $H(x)$ function defined as

$$H(x) = \int_{-\infty}^{x} \delta(x)dx,$$

where

$$\delta(x) = \begin{cases} 1 & x = 0\ , \\ 0 & \text{otherwise} \end{cases} \tag{4.48}$$

is a step function. Furthermore, function $\tilde{H}(x)$ is a smooth approximation to step function given by

$$\tilde{H}(x) = \frac{1}{1 + e^{-\alpha x}}, \;\; a \geq 0. \tag{4.49}$$

This definition could also be considered as a parametric form of logistic function, as well.

Besides, the function $H(x)$ could be related to the Ramp function, $R(x)$ as

$$R(x) = xH(x), \tag{4.50}$$

which is comprised of a heaviside multiplied by line x. Accordingly, by considering the $\tilde{H}(x)$ rather than $H(x)$ in 4.50, we have

$$\tilde{R}(x) = x\tilde{H}(x) = \frac{x}{1 + e^{-\alpha x}}, \;\; \alpha \geq 0, \tag{4.51}$$

where for a large value of a, the Ramp function behaves like $\max(0, x)$ whose functionality is as similar as the standard relu function.

By the virtue of using Equation 4.50, $\tilde{R}(x)$ could be identified as a smooth parametric form of relu function called adaptive-relu defined as

$$\text{adaptive} - \text{relu} = \tilde{R}(x) = \begin{cases} \max(0, x) & \alpha \to \infty \;, \\ \frac{x}{1+e^{-\alpha x}} & \text{otherwise}, \end{cases} \tag{4.52}$$

where $\alpha \geq 0$.

Consequently, the first derivative of adaptive-relu with respect to x will be calculated as

$$\frac{\partial \tilde{R}(x)}{\partial x} = \frac{1}{1 + e^{-\alpha x}} + \frac{x\alpha e^{-\alpha x}}{(1 + e^{-\alpha x})^2}. \tag{4.53}$$

In the same way, the derivative of the adaptive-relu with respect to parameter α is obtained by

$$\frac{\partial \tilde{R}(x)}{\partial \alpha} = \frac{x^2 e^{-\alpha x}}{(1 + e^{-\alpha x})^2}. \tag{4.54}$$

Figure 4.6 shows the adaptive-relu function with different values of parameter α. The green curve in most left-side graph illustrates an approximation of standard relu for a large value of non-negative α. The middle and right-side graphs demonstrate the derivatives of adaptive-relu with respect to input x and its parameter α, respectively. As understood from the above

plots, parameter α appeared in the adaptive-relu definition could be employed to tune the smoothness of the function by choosing fairly small values.

4.4.1 ANNs with Adaptive-relu : Loss Function and Optimization

Suppose that we have an MLP using the adaptive-relu function for L hidden layers and a softmax regression layer at top of the hidden layers. The gradients for the k^{th} parameter vector of layer l will be obtained by

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k^l} = -\frac{\partial \boldsymbol{\eta}^l}{\partial \boldsymbol{\theta}_k} \boldsymbol{\Delta}^l,$$

where

$$\boldsymbol{\theta}_k^l = [\mathbf{W}_k^l, \mathbf{w}_{0k}^l, \boldsymbol{\alpha}_k^l],$$

$$\boldsymbol{\Delta}^l = \frac{\partial \mathbf{h}^l}{\partial \boldsymbol{\eta}^l} \frac{\partial \boldsymbol{\eta}^{l+1}}{\partial \mathbf{h}^l} \boldsymbol{\Delta}^{l+1} \tag{4.55}$$

and $\boldsymbol{\alpha}$ is the parameter of the adaptive-relu function. Similar to the formulations presented for updating the parameters of ANN with adaptive-sigmoid, the gradient of softmax loss function ($\mathcal{L}$) with respect to the the weights and biases parameters are the same. The gradient for parameter α could be written as

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\alpha}_k^l} = -\frac{\partial \boldsymbol{\eta}^l}{\partial \boldsymbol{\alpha}_k^l} \boldsymbol{\Delta}^l$$

$$= \frac{\mathbf{W}_k^l (\mathbf{h}^{l-1})^2 \exp(-\boldsymbol{\alpha} \mathbf{h}^{l-1})}{\{1 + \exp(-\boldsymbol{\alpha} \mathbf{h}^{l-1})\}^2} \boldsymbol{\Delta}^l. \tag{4.56}$$

Accordingly, the parameters will be updated in backpropagation by the following update rules :

$$\mathbf{W}_k^l \leftarrow \mathbf{W}_k^l - \gamma \frac{\partial \mathcal{L}}{\partial \mathbf{W}_k^l},$$

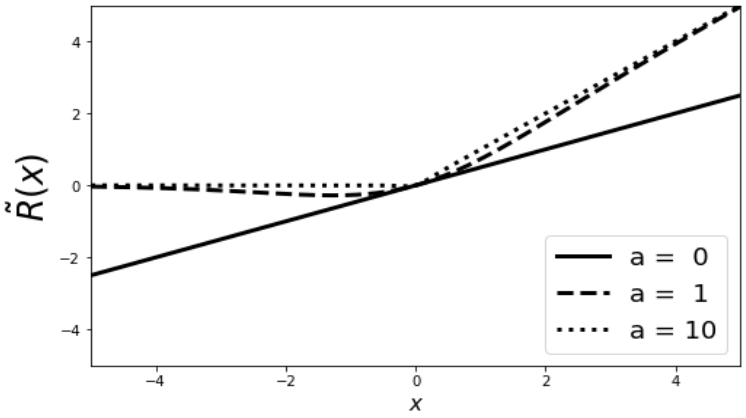
$$\mathbf{w}_{0k}^l \leftarrow \mathbf{w}_{0k}^l - \gamma \frac{\partial \mathcal{L}}{\partial \mathbf{w}_{0k}^l}$$

and

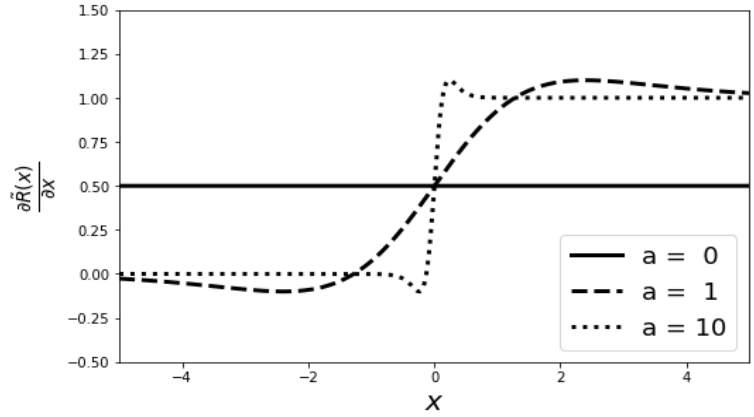
$$\boldsymbol{\alpha}_k^l \leftarrow \boldsymbol{\alpha}_k^l - \gamma \frac{\partial \mathcal{L}}{\partial \boldsymbol{\alpha}_k^l}, \tag{4.57}$$

where γ is the learning rate and takes a small fixed value.

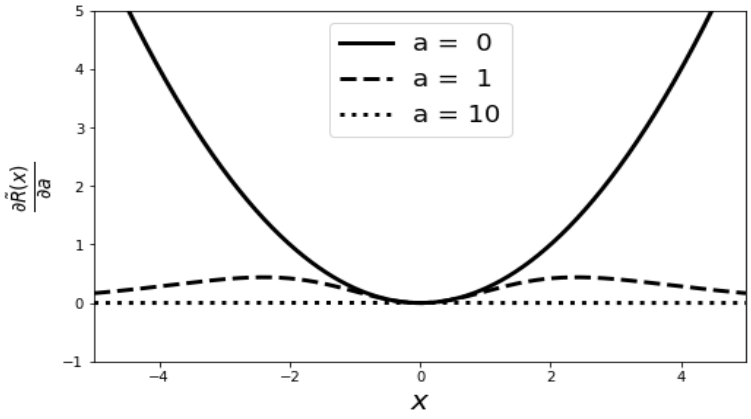
By computing the above quantities, the softmax output function and the corresponding loss



(a) adaptive-relu



(b) Derivative of adaptive-relu with respect to x



(c) Derivative of adaptive-relu with respect to α

Figure 4.6 Adaptive-relu and its derivatives with respect to parameters x and α for different values of α.

at softmax layer with/without regularization terms, the model parameters could be optimized in an iterative manner.

CHAPTER 5 EXPERIMENTS AND RESULTS

5.1 Performance Measures

In classification problems, the results on the test set are not measured directly from the classifier outputs, but from a two-dimensional confusion matrix built from these results. This matrix has one row and one column for each class ($k \leq K$). Therefore, each entry (i, j), $(1 \leq i, j \leq K)$, represents the number of examples predicted as class number i by the classifier when they actually belong to class number j. Hence, the diagonal entries (i, j) , $\forall i = j$ show the correctly classified examples, whereas off-diagonal terms, (i, j) for $i \neq j$, denote the incorrectly classified ones.

An example of a confusion matrix for a binary classification problem is illustrated in Table (5.1).

A single prediction may distinguish four different possible outcomes : true positives (TP) and false positives (FP) refers to the case where examples are correctly and incorrectly classified as positive, whereas true negatives (TN) and false negatives (FN) are examples correctly and incorrectly classified as negative, respectively. Ideally, a good result will be obtained by having large numbers on the main diagonal and small values for off-diagonal entries.

In our experiment, the performance of the classification is evaluated by the following rates that are computed from the confusion matrix as follows.

Accuracy : Accuracy is one of the most popular metrics and is defined as (Fawcett, 2006)

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN},$$

and it indicates how close measured predictions are to the actual values. In our experiments,

Table 5.1 Confusion Matrix Table

		Predicted Labels: Positive	Predicted Labels: Negative	Total
Actual Labels	Positive	TP	FN	$TP + FN$
	Negative	FP	TN	$FP + TN$
	Total	$TP + FP$	$FN + TN$	N

we also measure the performance of the classification with error which is of the form

$$\text{error} = 1.0 - \text{accuracy}.$$

In a problem where we have a large class imbalance in the data, the trained model may predict the value of the majority class for all predictions with a high accuracy (accuracy paradox).

Precision : Precision is defined as the ratio of true positives over the number of true positives plus the false positives as (Fawcett, 2006)

$$\text{precision} = \frac{TP}{TP + FP},$$

and it demonstrates how relevant the predicted values are to each other. In other words, it is the probability that an unseen retrieved data is relevant.

Recall : Recall is the ratio of true positives over the number of true positives plus the number of false negatives and takes the form (Fawcett, 2006)

$$\text{recall} = \frac{TP}{TP + FN}.$$

A low recall means that the classifier returned many false negative predictions. Precision is a measure of result relevancy, but recall is a metric which indicates how many truly related results are returned.

F1 : F1 score considers simultaneously both precision and recall in measurement. Thus both false positives and false negatives are taken into account. F1 score is defined as follows (Fawcett, 2006)

$$\begin{aligned} \text{F1} &= 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \\ &= \frac{2TP}{2TP + FP + FN} \\ &= 1.0 - \frac{FN + FP}{2TP + FN + FP}. \end{aligned}$$

F1 score could be interpreted as the harmonic mean of precision and recall. The best value for F1 score is 1 and the worst case is 0. Compared to accuracy, F1 is more useful when we have an uneven class distribution. By contrary, accuracy works better if false positives and

false negatives take small same values. If the value of false positives and false negatives are too different, one should look at F1 score.

Cost : Another metric we attempt to minimize during the network training is the negative log-likelihood of the fitted models, called the loss function. Furthermore, a weighted combination of λ_1 and λ_2 regularization terms is often added to the loss and the result is referred to as cost.

5.2 Experiment 1 : Simulated Data

Our objective in this experiment is to understand better how using the nonlinear, smooth and asymmetric activation functions have an effect on the representation and the prediction of the fitted MLP models. Suppose that we have a neural network M_1 with matrix $\mathbf{W}_1$ and bias vector $\mathbf{w}_{10}$ parameters. Given an input matrix $\mathbf{X}$, $\mathbf{W}_1$ and $\mathbf{w}_{10}$, the network model produces an output $\mathbf{Y}_1$. Now, under the assumption of having input $\mathbf{X}$ and output $\mathbf{Y}_1$, after performing a neural network training step, we aim at understanding how well the fitted model M_2 with corresponding learned parameters, $\mathbf{W}_2$, $\mathbf{w}_{20}$ and its prediction $\mathbf{Y}_2$ are approximated closely to $\mathbf{W}_1$, $\mathbf{w}_{10}$ and true labels $\mathbf{Y}_1$, respectively. While in M_1 and M_2 different initialization methods and activation functions are used.

For the primary model M_1, the first half of the weights in $\mathbf{W}_1$ are initialized randomly from a Normal distributions $\mathcal{N}_1(1, 0.5)$ and the remaining are produced randomly from $\mathcal{N}_2(-1, 0.5)$. Bias parameters $\mathbf{w}_{10}$, are also randomly initialized from a Normal distribution $\mathcal{N}(0, 0.5)$. These initialization methods were done at each layer in the primary model M_1.

For the fitted model M_2, we initialized the weights $\mathbf{W}_2$ and biases $\mathbf{w}_{20}$ parameters at each layer according to paper (LeCun *et al.*, 1998a).

Several configurations for M_1 and M_2 models were designed. These settings are shown in Table (5.2). The fitted model M_2 and corresponding primary model M_1 that are arranged in the same row, have the same number of hidden layers and same number of neurons at each layer. Also, other hyper-parameters such as $\gamma, \lambda_1, \lambda_2$, the coefficients of L_1 and L_2 regularizations, and batch-size in M_1 and M_2 were configured in a similar way. For creating the M_1 model, we used standard sigmoid and relu activation functions. For example, given $\mathbf{X}$ and $\mathbf{Y}_1$ from model M_1 with 2 hidden layers (2NN) and sigmoid as activation function, we compare the performance of four M_2 models fitting with sigmoid, adaptive-sigmoid, relu and adaptive-relu activation functions separately in terms of accuracy, predictions $\mathbf{Y}_2$ and the learned parameters of the model M_2, including $\mathbf{W}_2$ and $\mathbf{w}_{20}$.

5.2.1 Data

To produce the input $\mathbf{X}$, we first create a toy data set applying a random sample generator. The produced data includes a matrix of features and corresponding discrete binary targets such that each target value was assigned to one normally-distributed cluster of points. We also add noisy features to the feature space by using correlated, redundant and non-informative features; multiple Gaussian clusters per class; and linear transformations of the current features. The data includes 10,000 examples with 10 features which is considered as the input matrix $\mathbf{X}$.

Given $\mathbf{X}$ and feeding it into one pre-designed model M_1 with pre-defined parameters $\mathbf{W}_1$ and $\mathbf{w}_{10}$, we could obtain an output $\mathbf{Y}_1$. Now, having $\mathbf{X}$ and $\mathbf{Y}_1$, we can train a deep MLP M_2 and fit corresponding parameters $\mathbf{W}_2$ and $\mathbf{w}_{20}$ in a supervised manner. We tend to compare the fitted parameters $\mathbf{W}_2$ and $\mathbf{w}_{20}$ with the pre-defined parameters $\mathbf{W}_1$ and $\mathbf{w}_{10}$ in M_1 that the true class labels were originated from. Our goal is to understand how the trained model M_2 is successful in prediction and fitting the parameters closely to the primary ones in M_1.

5.2.2 Tools and Technologies

Training and results of the MLP models were implemented using a slightly modified implementation of original code developed by Theano development team [1].

5.2.3 Results and Evaluations

First, we want to study the effect of varying activation functions on the accuracy of the predictions produced by the fitted model M_2 compared to its original counterpart M_1 when they have the same number of hidden layers, same number of neurons at each layer, and weights and biases were initialized from different distributions. Our goal is to understand how accurately M_2 could predict the class labels at the end of training step. The accuracy results are displayed in Table (5.2).

Selecting the best values for hyper parameters including λ_1 and λ_2 parameters, are performed by designing a 5-fold cross-validation for model with two hidden layers (2NN) on simulated data. The hyper parameters in other model configurations remain fixed to only evaluate the effect of varying activation functions on final predictions.

In Table (5.2), for a given model M_1 with specific number of hidden layers and types of activation functions, the accuracy of fitted models using different activation functions are

1. http ://deeplearning.net/tutorial/code/mlp.py

Table 5.2 Accuracy table for Simulated data. M_1 : primary model with different configurations, M_2 : fitted models using different activation functions including sigmoid, adaptive-sigmoid, relu and adaptive-relu. The model M_1 could be created based on either sigmoid or relu. 1NN denotes MLP with one hidden layer and 2NN is used for MLP with two hidden layers. Both M_1 and M_2 have the same hyper parameters in all configurations such as $\lambda_1 = .001, \lambda_2 = .001, \gamma = .01$, batch size = 20, number of examples = 10000, number of features 10, number of classes = 2, number of neurons at each hidden layer = 10 and number of epochs = 2000.

	M_1	M_2			
model	activation function	sigmoid	ada-sigmoid	relu	ada-relu
1NN	sigmoid	95.8	97.5	**97.8**	97.7
	relu	96.1	97.4	**98.2**	98
2NN	sigmoid	95.1	95.5	95.1	**96.5**
	relu	92.3	96.1	96	**96.2**
4NN	sigmoid	91.8	92.1	91	**93.8**
	relu	93.8	**94.7**	94.6	94.3
8NN	sigmoid	83.7	**83.8**	81.8	81.9
	relu	57.3	88.2	89.3	**89.9**

reported. M_1 and M_2 that are being compared in one cell have exactly the same number of hidden layers. Following is the list of inferences understood from Table (5.2).

– For 1 hidden layer network (1NN), regardles of the type of activatin function in M_1, fitting M_2 with relu gives the best result. Adaptive-sigmoid and adaptive-relu outperform the sigmoid. Also, adaptive-relu obtains the better accuracy versus adaptive-sigmoid whether the primary model M_1 contains sigmoid or relu.

– For two hidden layer cases (2NN), M_2 with adaptive-relu outperforms the other counterparts whether the activation function in M_1 is sigmoid or relu. Similarly, M_2 with adaptive-sigmoid gives the better accuracy compared to M_2 with relu or sigmoid.

– Suprisingly, for four hidden layer cases (4NN), when M_1 uses sigmoid, M_2 with adaptive-relu gives the best accuracy. M_2 with adaptive-sigmoid outperforms other models when M_1 uses relu. M_2 with adaptive-sigmoid produces better accuracy than sigmoid whether M_1 uses sigmoid or relu.

– For eight hidden layer M_1 models (8NN) associated with sigmoid, M_2 with adaptive-sigmoid outperforms the other activation functions. Moreover, for M_1 created with relu, M_2 with adaptive-relu gives the best accuracy. Fitting M_2 with adaptive-relu and adaptive-sigmoid yields the better accuracy versus relu and sigmoid, separately.

– As the number of hidden layers increases, an expected drop in the performance of all M_2 is

seen. Hence, it is evident that there is no question to continue for deeper fully-connected layers.
– For models with less layers, fitting M_2 with relu or adaptive-relu almost outperforms the other activation functions. For deeper models, fitting M_2 with the adaptive-sigmoid also exhibits a good performance in competition with relu and adaptive-relu.
– In all model settings, fitting M_2 with adaptive-sigmoid outperforms the accuracy obtained by M_2 with sigmoid.

The fitted weights ($\mathbf{W}_2$) and biases ($\mathbf{w}_{20}$) curves corresponding to the first hidden layer, are illustrates in Figure 5.1 and 5.2, respectively.

As observed from Figure (5.1), when the original model is associated with standard sigmoid activation function (left histograms), the M_2 models with proposed adaptive functions outperform other models in approximating the biases. As the number of hidden layers increases, a drop in the approximation is seen. M_2 with sigmoid function could not fit biases in an eight-layer network while M_2 with adaptive-relu function is successful to fit an acceptable curve.

For the original models designed by sigmoid activation functions, fitting models with adaptive-sigmoid gives the better bias approximation compared to other activation functions in 1NN and 2NN models. By increasing the number of hidden layers, fitting model with relu outperforms other functions in approximating bias. In all models originated from sigmoid, fitting models with adaptive-relu yield poor bias approximation.

When the original model M_1 is created by standard relu (right histograms), the M_2 models with adaptive-sigmoid and adaptive-relu functions fit the biases except for the eight-layer networks where the M_2 model with standard relu approximate biases excellent. Overally, fitting models with relu gives a better approximation of bias distribution. In deeper models, although adaptive-relu exhibits a fairly good approximation, performance of the sigmoid and adaptive-sigmoid drop dramatically in approximating bias distribution.

As seen from Figure (5.2), when the original model was initialized by a mixture Gaussian distribution, none of the fitted models could approximate the original weights $\mathbf{W}_1$ properly, but cover corresponding range in 2 and 4 hidden layer models. It is probable that the reason would be due to the non-identifiability property of the neural networks. Hence, studying the fitted weight histograms dose not yield any insight regarding the performance of the adaptive activation functions being compared in terms of weight approximation.

As understood from Figure (5.3), the prediction performance of the all M_2 models associated with different activation functions are the same, except the case where the original model M_1 was designed by a standard relu. In this case, M_2 with standard sigmoid could not converge

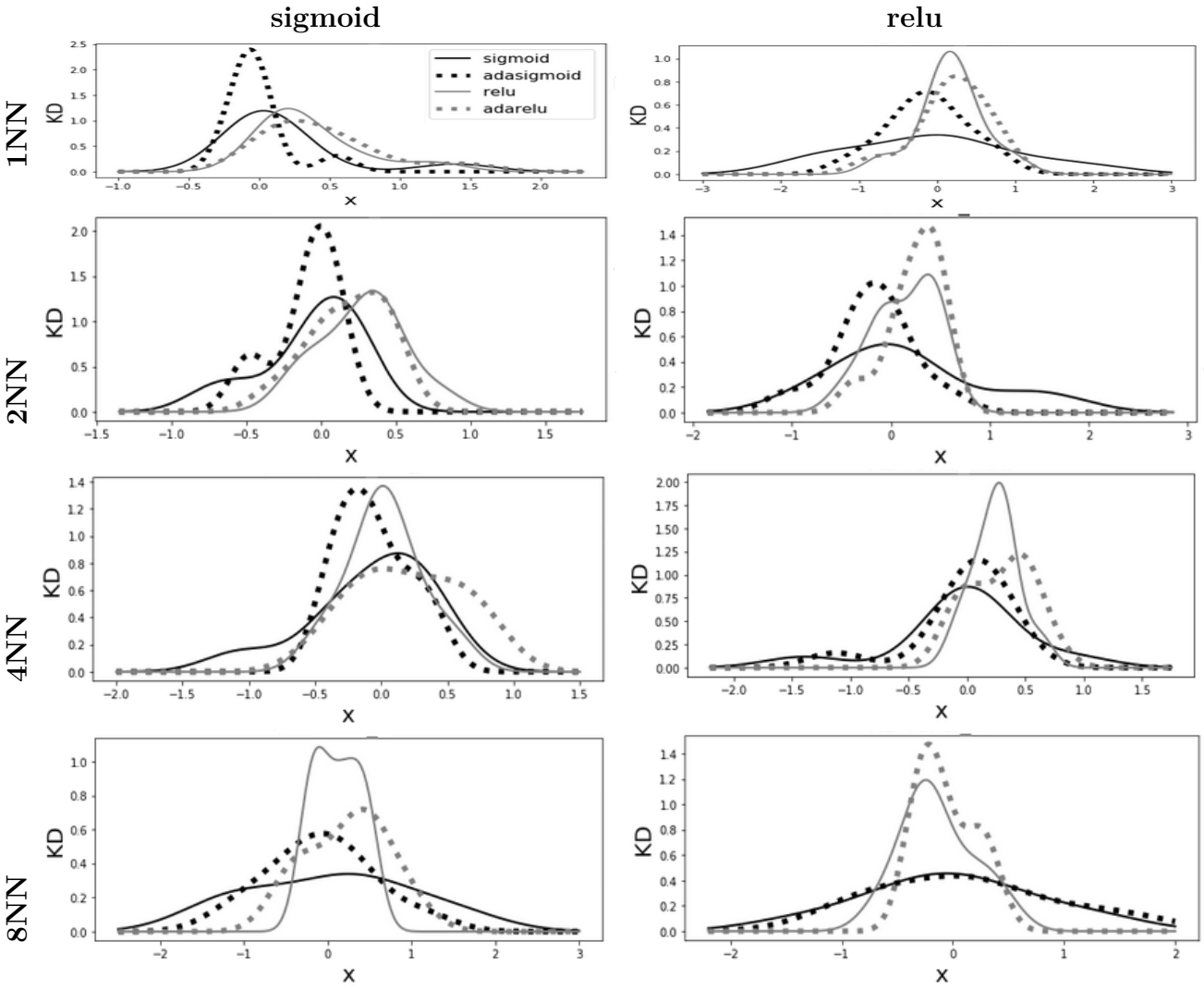


Figure 5.1 Comparing the biases $\mathbf{w}_{20}$ histogram of fitted models M_2 (in first hidden layyer) using different activation functions when the primary model M_1 uses sigmoid (left column) and relu (right column). Biases were initialized from $\mathcal{N}(0, 0.5)$ in M_1 and biases in M_2 initialized by formulation suggested in (LeCun *et al.*, 1998b). Both M_1 and M_2 models in each single graph have the same configuration in terms of number of layers and neurons at each layer. The hyper parameters in all models are set as $\lambda_1 = .001, L_2 = .001, \gamma = .01$, batch size = 20, number of examples = 10000, number of features 10, number of classes = 2, number of neurons at each hidden layer = 10 (Simulated data).

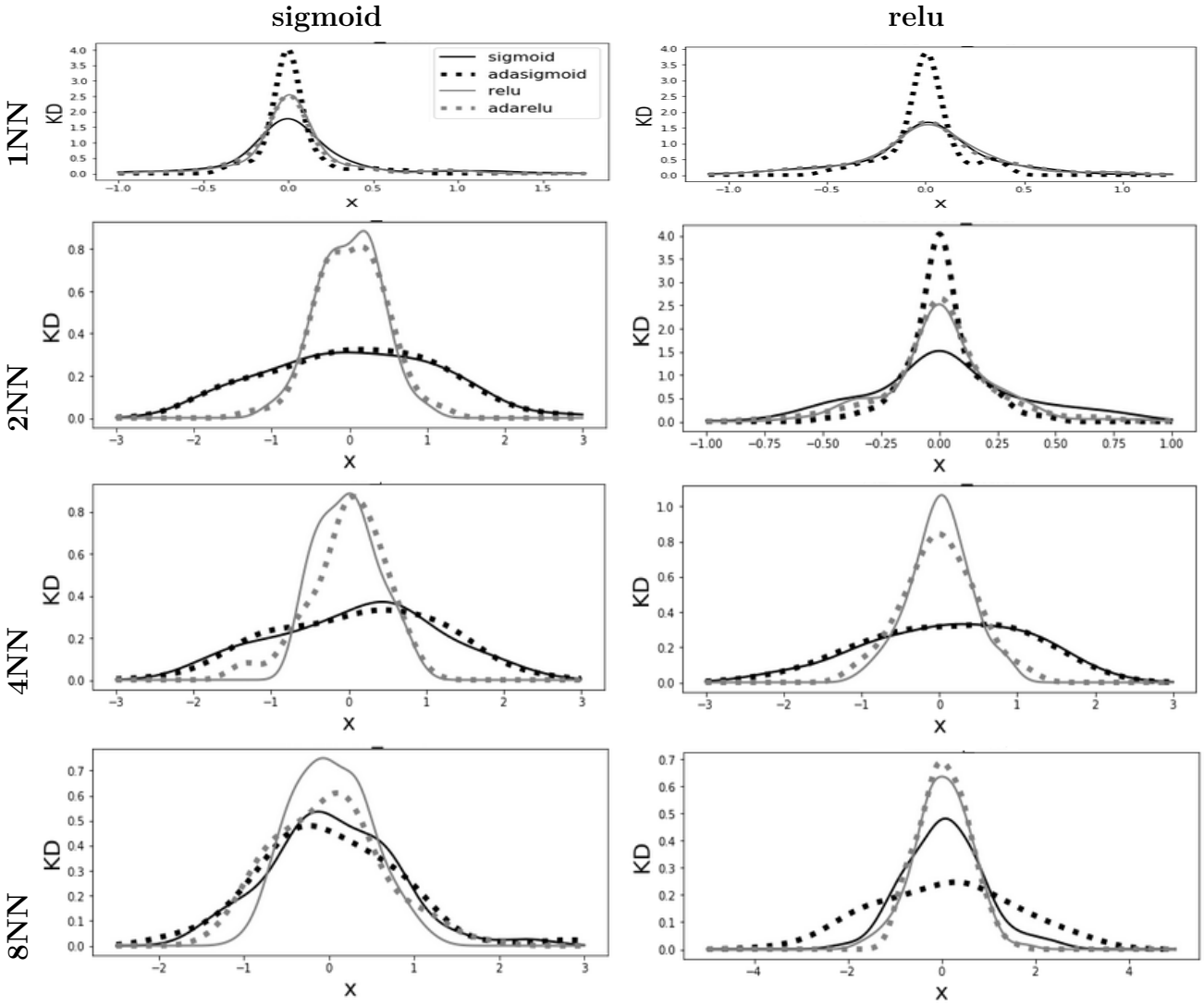


Figure 5.2 Comparing the weights $\mathbf{W}_2$ histogram of fitted models M_2 (in first hidden layer) using different activation functions when the primary model M_1 using sigmoid (left column) and relu (right column). The origin weights $\mathbf{W}_1$ in M_1 were initialized from a mixture of $\{\mathcal{N}(-1, 0.5), \mathcal{N}(1, 0.5)\}$ and weights in M_2 initialized by (LeCun *et al.*, 1998b). Both M_1 and M_2 models in each single graph have same configuration in terms of number of layers and neurons at each layer. The hyper parameters in all models are set as $\lambda_1 = .001, \lambda_2 = .001, \gamma = .01$, batch size = 20, number of examples = 10000, number of features 10, number of classes = 2, number of neurons at each hidden layer = 10 (Simulated data).

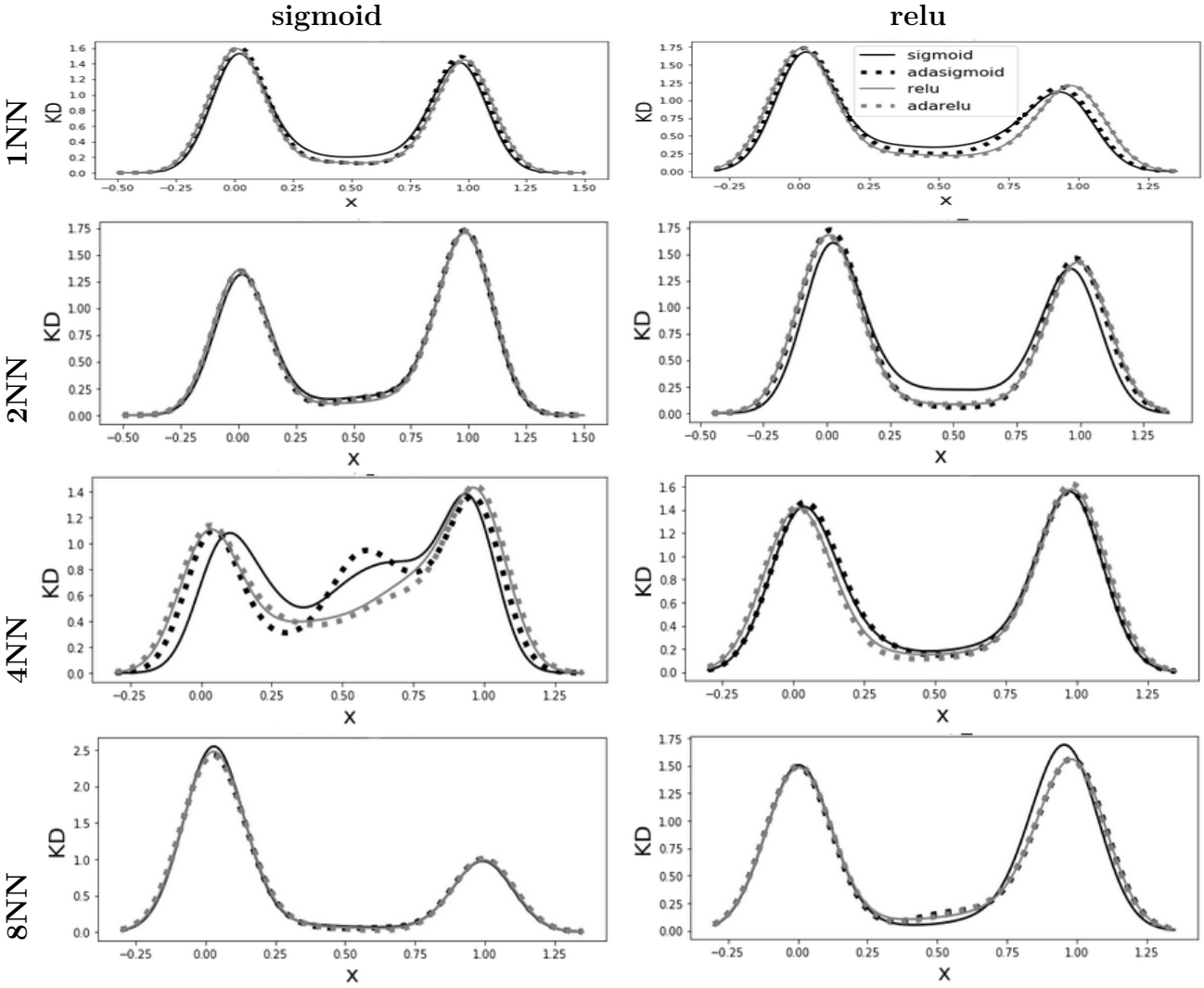


Figure 5.3 Comparing the weights difference $\mathbf{Y}_2$ histogram of fitted models M_2 using different activation functions when the primary model M_1 using sigmoid (left column) and relu (right column). Both M_1 and M_2 models in each single graph have same configuration in terms of number of layers and neurons at each layer. The hyper parameters in all models are set as $\lambda_1 = .001, \lambda_2 = .001, \gamma = .01$, batch size = 20, number of examples = 10000, number of features 10, number of classes = 2, number of neurons at each hidden layer = 10 (Simulated data).

as expected. Moreover, fitting models with standard proposed activation functions could approximate the true labels to the same extend in almost all configurations.

The final evaluation that we care for is the success of the learned models in error and cost convergence. Figures (5.4) and (5.5) demonstrate a zoom-in of the error and cost curves, respectively during the training.

These curves show the effect of choosing different activation functions in model training. Several conclusions could be inferred from the error and cost curves :

- The classical M_2 models with standard sigmoid neurons converge more slowly compared to their counterparts associated with adaptive-sigmoid in terms of error and cost.
- M_2 models with relu and adaptive-relu always converge faster than sigmoid.
- Deeper M_1 models with relu yield poorer convergence for M_2 with sigmoid.
- For approximating the M_1 models with sigmoid, fitting M_2 with adaptive functions seems to be more robust to the size of the network than the standard sigmoid and relu.

The performance of the predictions are also evaluated by Precision (Pr), Recall (Re) and F1 in each experiment.

The Pr, Re and F1 of the M_2 model per each class using various activation functions when the M_1 is designed by Sigmoid are demonstrated in Figure (5.6).

If the original model is designed by sigmoid, fitting models with adaptive-sigmoid gives the better classification performance compared to standard sigmoid in 1NN and 2NN models. On contrary, sigmoid is superior compared to adaptive-sigmoid in deeper networks. In 1NN and 2NN networks, M_2 models fitted by adaptive-sigmoid and adaptive-relu outperform models with standard sigmoid and relu activation functions, respectively. In 2NN models, M_2 models fitted with ada-sigmoid outperforms the other activation functions. In 4NN and 8NN networks, M_2 with adaptive-relu considerably results in better performance than other activation functions. Furthermore, comparing the sigmoid and adaptive-sigmoid, fitting models with adaptive-sigmoid produces poor results in terms of precision, recall and F1. It is also inferred that relu and adaptive-relu could predict the same classification results in deeper networks.

Similar experiments were performed for measuring the Pr, Re and F1 of the M_2 model per each class using various activation functions when the M_1 is designed by the standard relu. Results are illustrated in Figure (5.7).

In two-hidden layer models, M_2 with adaptive-sigmoid or adaptive-relu work better than the standard sigmoid or relu, respectively. In four hidden-layer models, it is evident that using adaptive-relu remarkably outperforms the other activation functions. In eight-layer models,

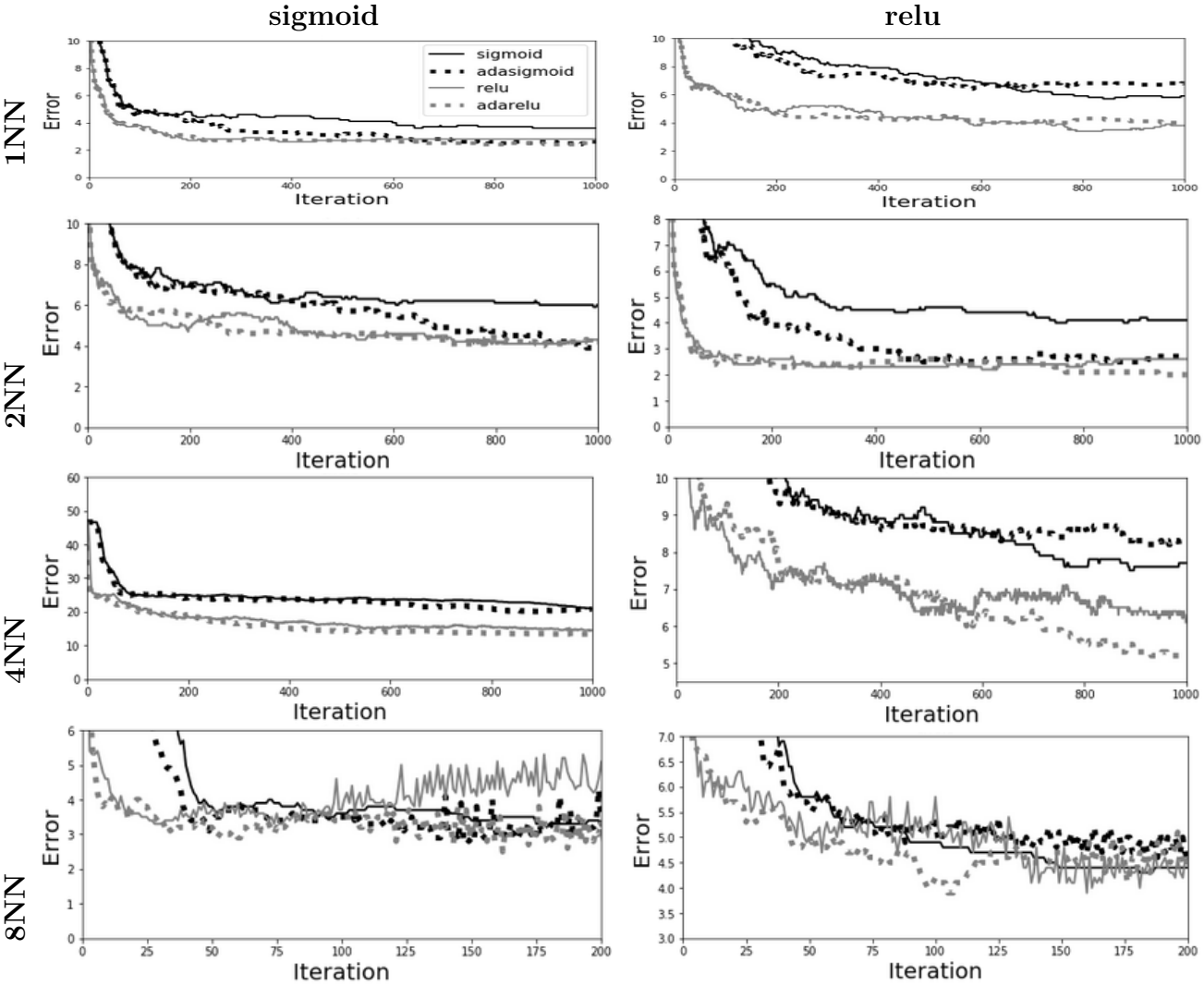


Figure 5.4 Compare the error curve of fitted models M_2 using different activation functions with the primary model M_1 using sigmoid (left column) and relu (right column). Both M_1 and M_2 models in each single graph have the same configuration in terms of number of layers and neurons at each layer. The hyper parameters in all models are set as $\lambda_1 = .001, \lambda_2 = .001, \gamma = .01$, batch size = 20, number of examples = 10000, number of features 10, number of classes = 2, number of neurons at each hidden layer = 10 (Simulated data).

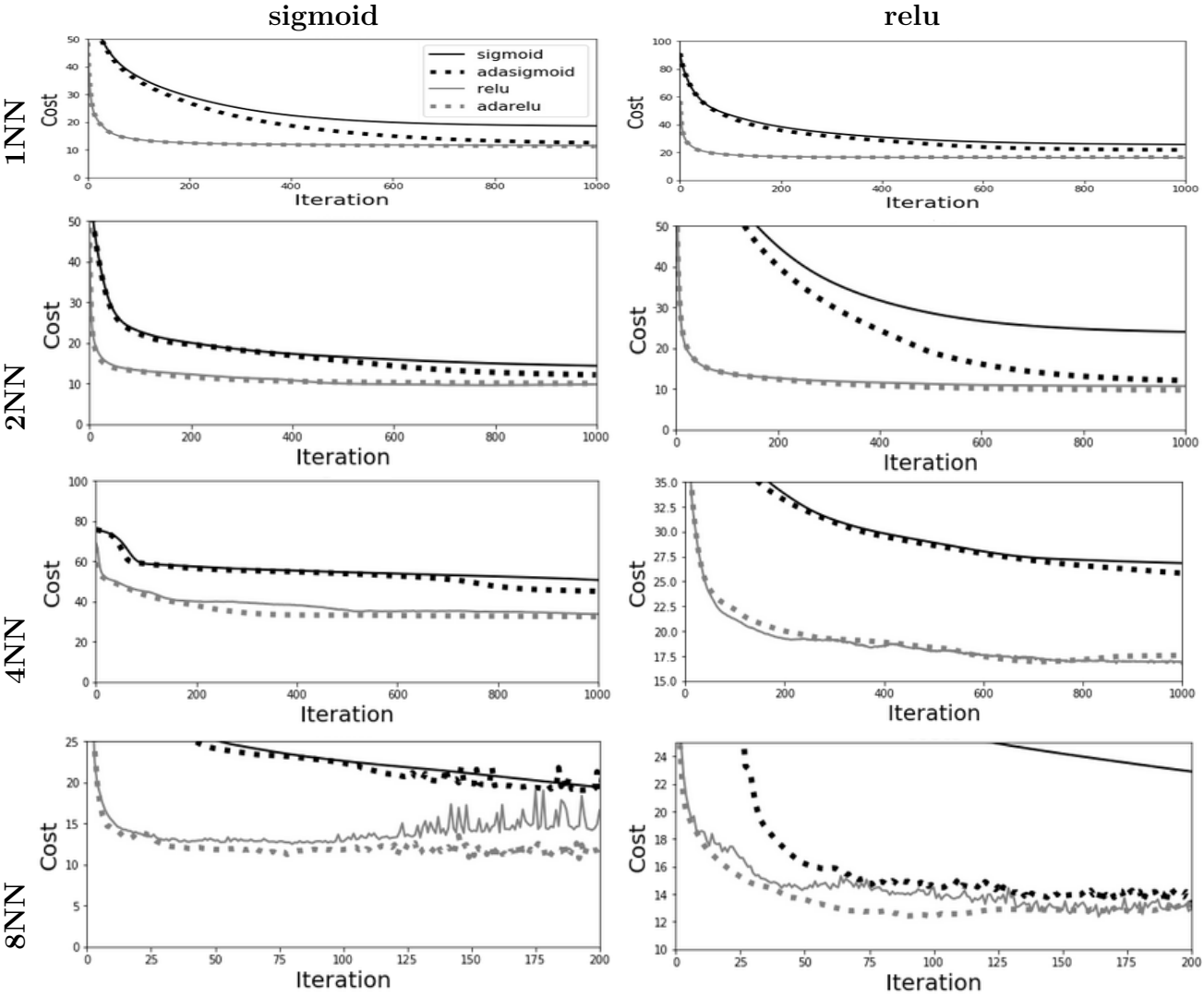
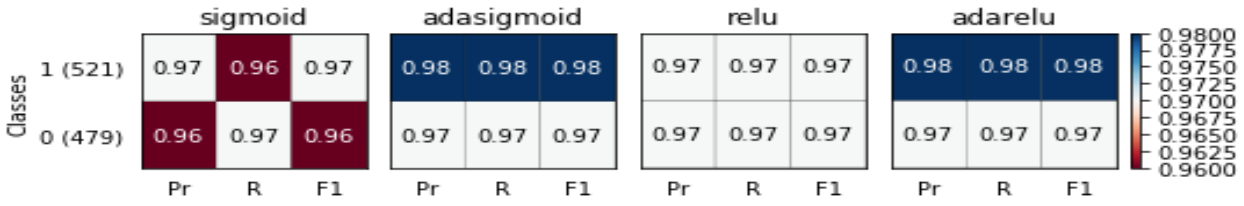
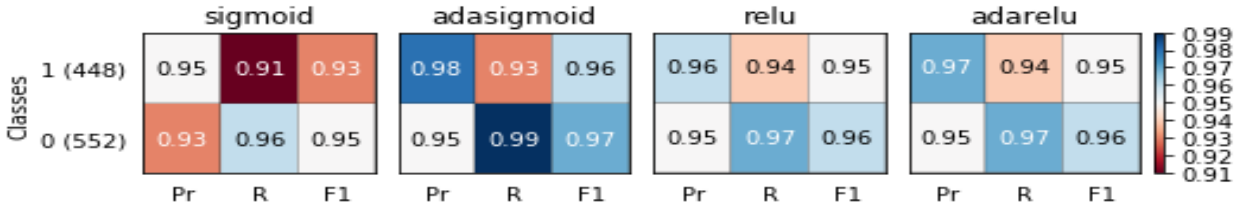


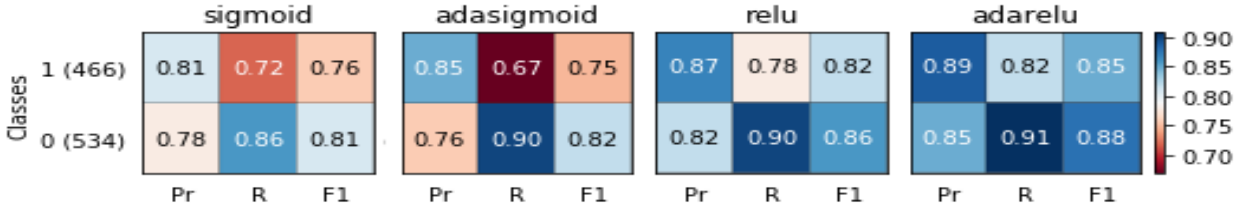
Figure 5.5 Comparing the cost curve of fitted models M_2 using different activation functions with the primary model M_1 using sigmoid (left column) and relu (right column). Both M_1 and M_2 models in each single graph have the same configuration in terms of number of layers and neurons at each layer. The hyper parameters in all models are set as $\lambda_1 = .001, \lambda_2 = .001, \gamma = .01$, batch size = 20, number of examples = 10000, number of features 10, number of classes = 2, number of neurons at each hidden layer = 10 (simulated data).



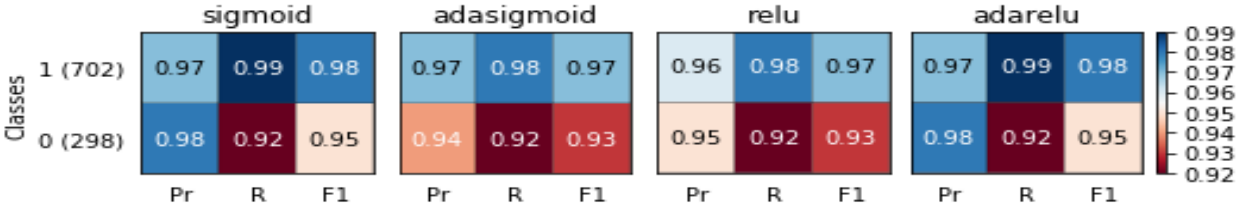
(a) 1NN



(b) 2NN



(c) 4NN



(d) 8NN

Figure 5.6 Comparing the Pr, Re and F1 of the four M_2 models fitted with sigmoid, ada-sigmoid, relu and ada-relu activation functions when their corresponding M_1 models designed with sigmoid in a) 1NN, b) 2NN, c) 4NN and d) 8NN hidden layer MLPs (Simulated data).

M_2 models with sigmoid and adaptive-relu give the better classification performance compared to relu and adaptive-sigmoid. Overally, adaptive-relu often outperforms other activation functions regardless of network size and it mostly gives the better performance compared to standard relu.

5.2.4 Conclusions

Our motivation is to compare the performance of the proposed adaptive activation functions in approximating the biases, weight representations, prediction and convergence of the MLP models, when the original simulated models are designed by standard sigmoid or relu, separately. Additionally, we aimed at understanding how accurate the deep MLP models using our proposed activation functions could approximate the targets which are originated from a simulated model.

Comparing standard sigmoid and adaptive-sigmoid, fitting MLP models with adaptive-sigmoid is the model of choice to produce more accurate predictions. Comparing the standard relu and adaptive-relu in deep networks, MLP using relu is the model of choice to produce more accurate prediction.

Comparing sigmoid and adaptive-sigmoid, fitting a model with adaptive-sigmoid is superior to approximate bias in all model architectures. Comparing relu and adaptive-relu, relu fits the better bias distribution, generally.

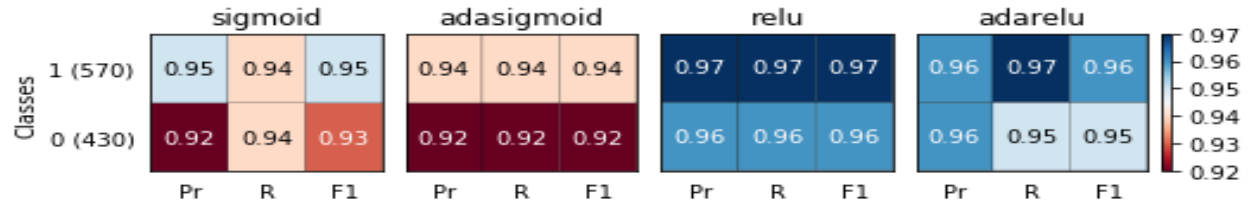
Due to the non-identifiability of neural networks, fitting models with standard and adaptive activation functions could not approximate the pre-defined weight parameters of the original models.

The empirical evidence shows that using adaptive activation functions in fitting models could approximate the distribution of true labels correctly in (deep) networks.

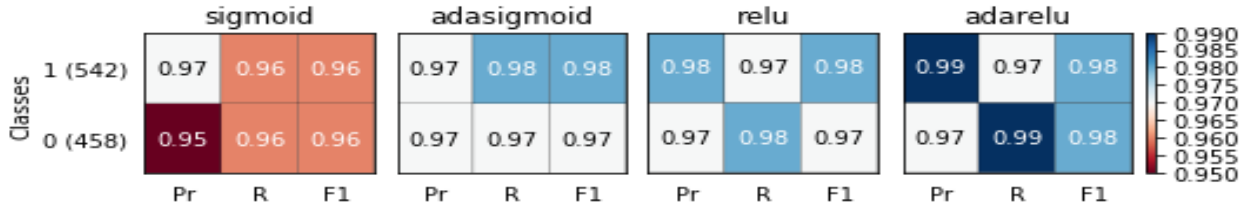
By designing a deep MLP using adaptive sigmoid or adaptive-relu, fitted models would be able to converge more rapidly than their counterparts with standard sigmoid and standard relu, respectively in terms of error and cost.

Fitting models with adaptive-relu could be the model of choice to have a better classification performance in terms of accuracy, precision, recall and F1 metric, regardless of the network size and activation functions used in original model.

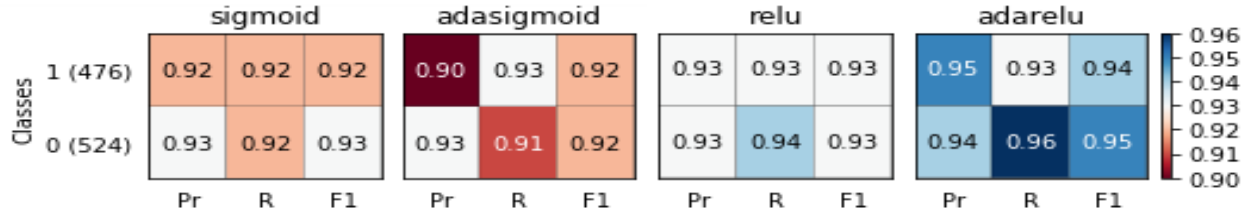
When the original model is designed with relu, model approximation with adaptive-sigmoid could not produce good results in terms of precision, recall and F1. However, by increasing the network size, adaptive-relu gives the better performance than relu in classification metrics.



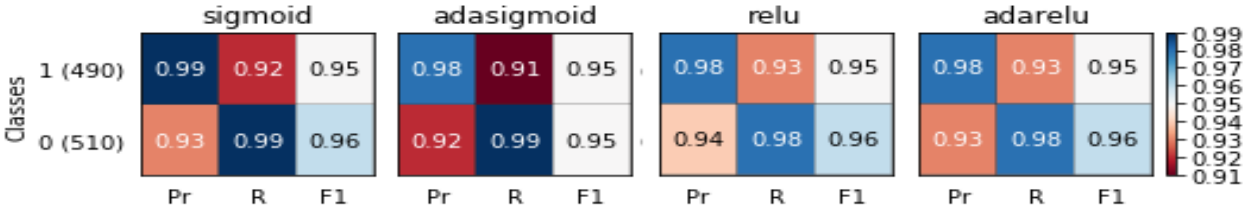
(a) 1NN



(b) 2NN



(c) 4NN



(d) 8NN

Figure 5.7 Comparison of the Pr, Re and F1 of the four M_2 models fitted with sigmoid, ada-sigmoid, relu and ada-relu activation functions when their corresponding M_1 models designed with relu in a) 1NN, b) 2NN, c) 4NN and d) 8NN hidden layer MLPs (Simulated data).

5.3 Experiment 2 : MNIST Benchmark

Motivated by the results obtained from experiment on simulated data, we tend to evaluate the performance of the MLP and CNN models accompanied with our proposed activation functions compared to standard ones, on MNIST benchmark.

5.3.1 data

The MNIST dataset [2] is a widely used benchmark for image recognition problems in the literature. Huge amount of comparative experiments have been studied on MNIST with different variants of ANNs (LeCun, 1998; Jain et Ko, 2008; Jou *et al.*, 2004; LeCun *et al.*, 1998a). This data set is a collection of 60,000 hand written digits ranging from '0' to '9', written by American high school students. The samples in this original data set are black and white and are centered and normalized to a 28×28 image size. The full MNIST dataset contains 50,000 training, 10,000 validation and 10,000 testing instances. Each image is constituted by a set of pixels represented by possible values between 0 and 255 which denote how bright that pixel is and what color it should be in grey scale. Zero represents black and 255 is taken to be white color. Other pixel values in between are taken to be different shades of gray (LeCun, 1998).

5.3.2 Tools and Technologies

Training and results of the MLP and CNN models were implemented using a modified implementation of original code developed by Theano development team [3]. After performing the data preparation step, the Theano tools can be employed for model training implementation. The results of the training step are dumped and stored as separate files including the learned parameters wherein these parameters together could be retrieved for classification of test images.

5.3.3 Models

In this work, several variants of MLP and CNN algorithms accompanied with our proposed adaptive activation functions as classifiers have been trained to test their performance on MNIST. Selecting the best values for hyper parameters, e.g. λ_1 and λ_2 are performed by designing a 5-fold cross-validation for MLP with one hidden layer and 1000 neurons on

2. http ://yann.lecun.com/exdb/mnist/
3. http ://deeplearning.net/tutorial/code/mlp.py

MNIST data. These hyper parameters in other MLP models remain fixed to only evaluate the effect of varying activation functions on final predictions. Hyper parameters for CNN models are chosen according to the primary settings of LeNet5 implementation developed by Theano development team. Similar to MLP models, we select fixed hyper parameters in all CNN models to study the effect of changing activation functions on final performance.

5.3.4 Results and Evaluations

First, the ability of some variants of fully-connected MLP models with standard and proposed adaptive activation functions are evaluated to compare how accurately they could predict digit characters. The models are fed with raw images as inputs to recognize the corresponding characters they represent. The models characterizations are different in terms of number of hidden layers, number of neurons and the type of the activation functions at each layer. The hyper-parameters of the MLP models are obtained by running a 5-fold cross-validation experiment for MLP with one hidden layer and 1000 neurons on MNIST data and remain fixed to only evaluate the effect of varying activation functions on final predictions. The hyper parameters are chosen as $\gamma = 0.01$, $\lambda_1 = 0.0$, $\lambda_2 = 0.0001$, batch-size $= 20$ and epochs $= 500$. The accuracy of each model is reported by running the corresponding algorithm on standard test set provided in MNIST data.

Several conclusions are drawn from the study of the accuracy in these MLP models from Table (5.3).

– The best accuracy is gained by the one fully-connected MLP with adaptive-relu activation function at each layer.
– By increasing the number of fully-connected layers from 1 to 2 and preserving the number of neurons fixed on 500, the performance of the MLP with sigmoid and adaptive-sigmoid functions decrease while the accuracy of models with relu and adaptive-relu improve.

Table 5.3 Accuracy table : MLP models with different number of hidden layers, number of neurons and the type of activation functions at each layer. The hyper-parameters are $\gamma = 0.01$, $\lambda_1 = 0.0$, $\lambda_2 = 0.0001$, $\text{batch} - \text{size} = 20$, epochs $= 500$ (MNIST data).

model	# of neurons	sigmoid	ada-sigmoid	relu	ada-relu
1NN	500	97.65	97.6	98.22	**98.77**
	1000	97.95	97.36	97.85	97.99
2NN	500	97.55	97.59	98.27	98.29
	1000	97.95	97.36	97.85	97.99

– By increasing the number of fully-connected layers from 1 to 2 and preserving the number of neurons fixed on 1000, no improvement was observed in performance of the MLP models.
– In one hidden-layer models (1NN), regardless of the number of neurons, the accuracy of MLP models with sigmoid, relu and adaptive-relu are always higher than MLP with adaptive-sigmoid. Also, using adaptive-relu outperforms its counterpart with relu activation function.
– In two hidden layer models, using adaptive-sigmoid almost yields poor results and using adaptive-relu always outperforms other models regardless of number of neurons.
– In almost all models, using adaptive-sigmoid reports the poorest results compared to relu and adaptive-relu.
– Using the adaptive-relu as the activation function always produces the best results in terms of accuracy. Using the adaptive-sigmoid is not recommended in fully-connected MLP models.

The potential of some variants of CNN models with standard and proposed adaptive activation functions are also studied in our MNIST experiments. Our CNN models are built based on the generic form of the LeNet5 introduced by (LeCun *et al.*, 1998a) and implemented in Theano. Each model architecture contains two convolutional layers and each of them is followed by a max-pooling layer. The model characterizations are different in terms of activation functions at convolutional and hidden layers. Number of hidden layers in a fully-connected network is 1 and includes 1000 neurons. The hyper-parameters are chosen as following : number of kernel window sizes = [20, 50], image shapes = [28, 12], filter shapes = [5, 5], pool sizes = [2, 2], $\lambda_1 = 0.0$ and $\lambda_2 = 0.0001$. For all experiments, we train our model parameters using the stochastic gradient descent with a fixed learning rate $\gamma = 0.01$, mini-batches of size 100 during 150 epochs and it takes about 25 epochs to converge. The accuracy of each model is reported by running the corresponding algorithm on standard test data in Table 5.4.

The following conclusions are inferred from the evaluation of accuracy in our CNN models (Table 5.4) :

– Applying the adaptive-sigmoid in fully-connected layers significantly improve the accuracy of the corresponding models compared to the models that use tanh function in their hidden layers, regardless of what function is used in their convolution layers.
– On the contrary, using adaptive-relu rather than standard relu in fully-connected layers drops the performance of the models considerably regardless of the choice of activation function in convolution layers.
– CNNs with adaptive-relu in fully-connected layers often give the better results compared to models with tanh in fully-connected layers.
– Using standard relu in fully-connected layers always outperforms its counterparts with

Table 5.4 CNN models with two convolution layers (each of them following a max pooling layer) and one fully-connected MLP at the end. The functions in rows represent the activation functions used in convolutional layers while the functions in columns denote the activation functions employed in fully-connected network. The hyper parameters of the all models are the same in all experiments including # of kernel window sizes = [20, 50], image shapes = [28, 12], filter shapes = [5, 5], pool sizes = [2, 2] and hyper-parameters $\gamma = 0.01$, $\lambda_1 = 0.0$, $\lambda_2 = 0.0001$, $\text{batch} - \text{size} = 100$, iterations = 150. In all CNN models batch normalization is used (MNIST data).

		Fully-connected layer			
		sigmoid	ada-sigmoid	relu	ada-relu
Convolutional layer	tanh	98.56	98.79	98.91	98.77
	ada-sigmoid	98.55	98.74	98.89	98.78
	relu	98.95	99.07	**99.09**	98.77
	ada-relu	98.75	98.76	98.85	98.9

standard sigmoid or adaptive-sigmoid in fully-connected layers.

– The best accuracy is gained by the CNN using standard relu activation function in both convolutional and fully connected layers.
– Compared to tanh, using adaptive-sigmoid in convolutional layers almost leads to a drop in accuracy performance.
– Using relu in convolutional layers almost outperforms the CNN models accompanied with sigmoid, adaptive-sigmoid or adaptive-relu except when those models are associated with adaptive-relu in fully-connected layers.
– Although applying adaptive-sigmoid in MLPs yiels poor results on MNIST data, using it as an activation function in fully-connected layers in CNN models leads to a fairly large increase in accuracy.
– For convolutional layers, using adaptive-relu is a good choice versus adaptive-sigmoid, but not relu.

In terms of convergence, as understood from Figures 5.8 and 5.9, in one-hidden layer MLPs by increasing the number of neurons, error curve for those models associated with standard sigmoid and adaptive-sigmoid converges slightly slower. A similar trend was seen in cost curves as well. Furthermore, MLP models with adaptive-sigmoid converge faster than their counterparts with sigmoid in terms of error and cost.

On the contrary, in two hidden layer MLPs, models with adaptive-sigmoid activations converge slower than those using sigmoid in terms of error. But their corresponding cost curves converge considerably faster. In all models, MLPs with relu and adaptive-relu activations converges significantly faster in competition with sigmoid and adaptive-sigmoid in terms of cost and

error.

In Figure (5.10), as understood from the comparison of the error and cost curves, CNN models with adaptive-relu in fully-connected layer work excellent compared to other models, except when relu is used in convolutional layers. Models with adaptive-sigmoid in fully-connected layer always converge slower than sigmoid, regardless of the activetion function in convolutional layers.

Classification reports according to Precision (Pr), Recall (Re) and F1 are also performed and the results are provided in appendix A.

5.3.5 Conclusions

In this experiment, we have motivated to study the effect of using proposed adaptive-sigmoid and adaptive-relu in two classical MLP and CNN models and evaluate the performance of the learned models in comparison with standard activation functions tanh and relu on MNIST data.

Using adaptive-relu is strongly recommended in MLP networks in terms of accuracy. However, applying adaptive-sigmoid results in failure compared to other activation functions.

Although using adaptive-sigmoid in MLP models exhibits poor accuracy, applying it in fully-connected layers of CNNs are recommended, but not in convolutional layers.

Compared to tanh and adaptive-sigmoid, using adaptive-relu is strongly recommended to be employed in either convolutional or fully-connected layers. However, it could not outperform the accuracy obtained by relu, regardless of where it is applied.

In terms of error and cost convergence, empirical results show that MLP models with adaptive-sigmoid converge more faster compared to the standard sigmoid while the error and cost curves for adaptive-relu remain same compared to the standard relu. Adaptive-sigmoid curves could not exceed the relu and adaptive-relu, evidently.

In CNN experiment, models with adaptive-relu converge more rapidly than adaptive-sigmoid. Adaptive-sigmoid could not exceed tanh curves in terms of error and cost convergence.

To summarize, experiments on MNIST data show that an MLP or CNN model using adaptive activation functions could perform excellent in both prediction and convergence compared to models using only standard activation functions.

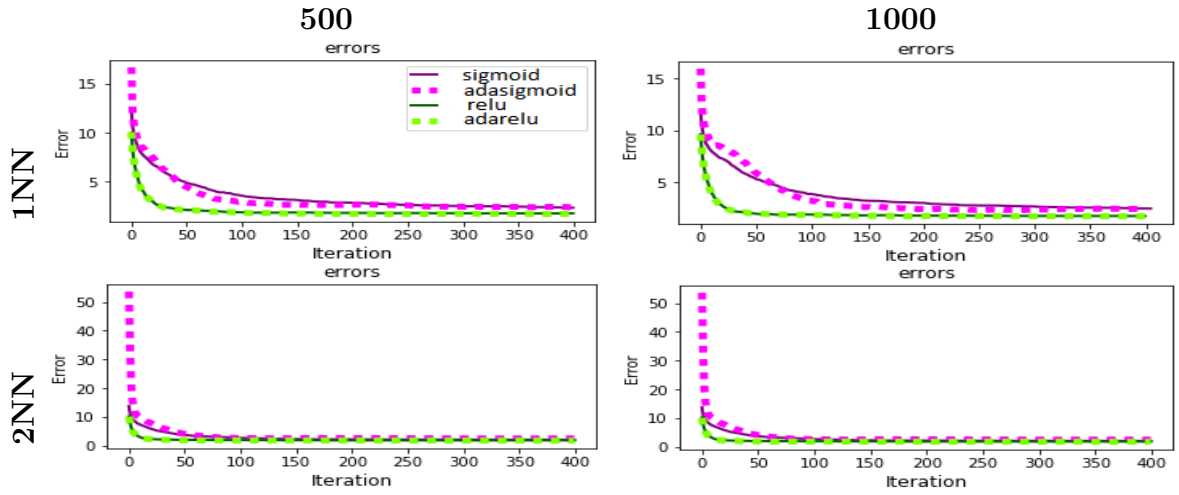


Figure 5.8 Comparison of the errors curves for MLP models with different configurations : rows represent the number of hidden layers (1NN and 2NN) and columns indicate the number of neurons at each hidden layer (500 and 1000). The comparisons are based on MLP models using four different activation functions : sigmoid, adaptive-sigmoid, relu and adaptive-relu (MNIST data).

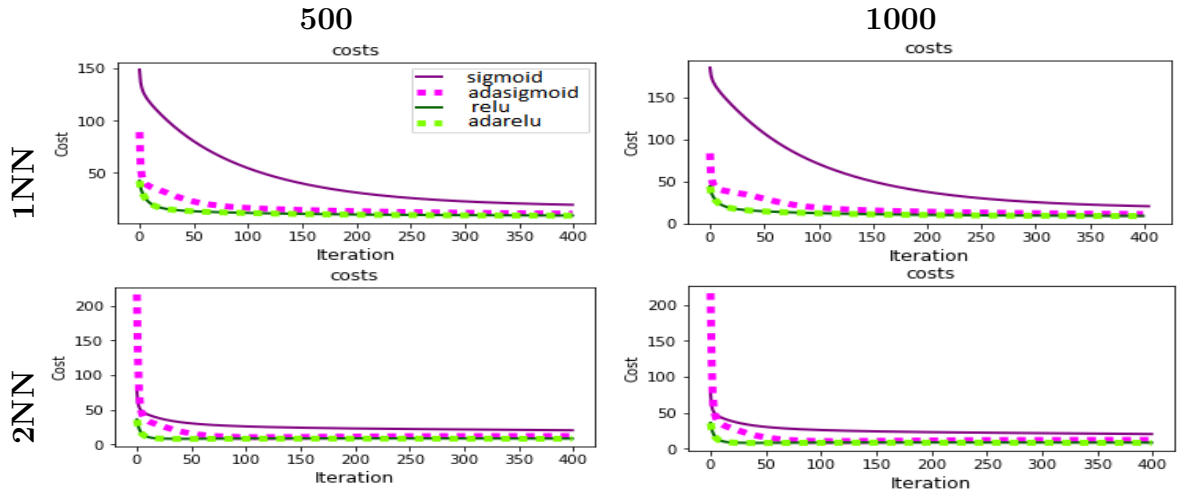


Figure 5.9 Comparison of the cost curves for MLP models with different configurations : rows represent the number of hidden layers (1NN and 2NN) and columns indicate the number of neurons at each hidden layer (500 and 1000). The comparisons are based on MLP models using four different activation functions : sigmoid, adaptive-sigmoid, relu and adaptive-relu (MNIST data).

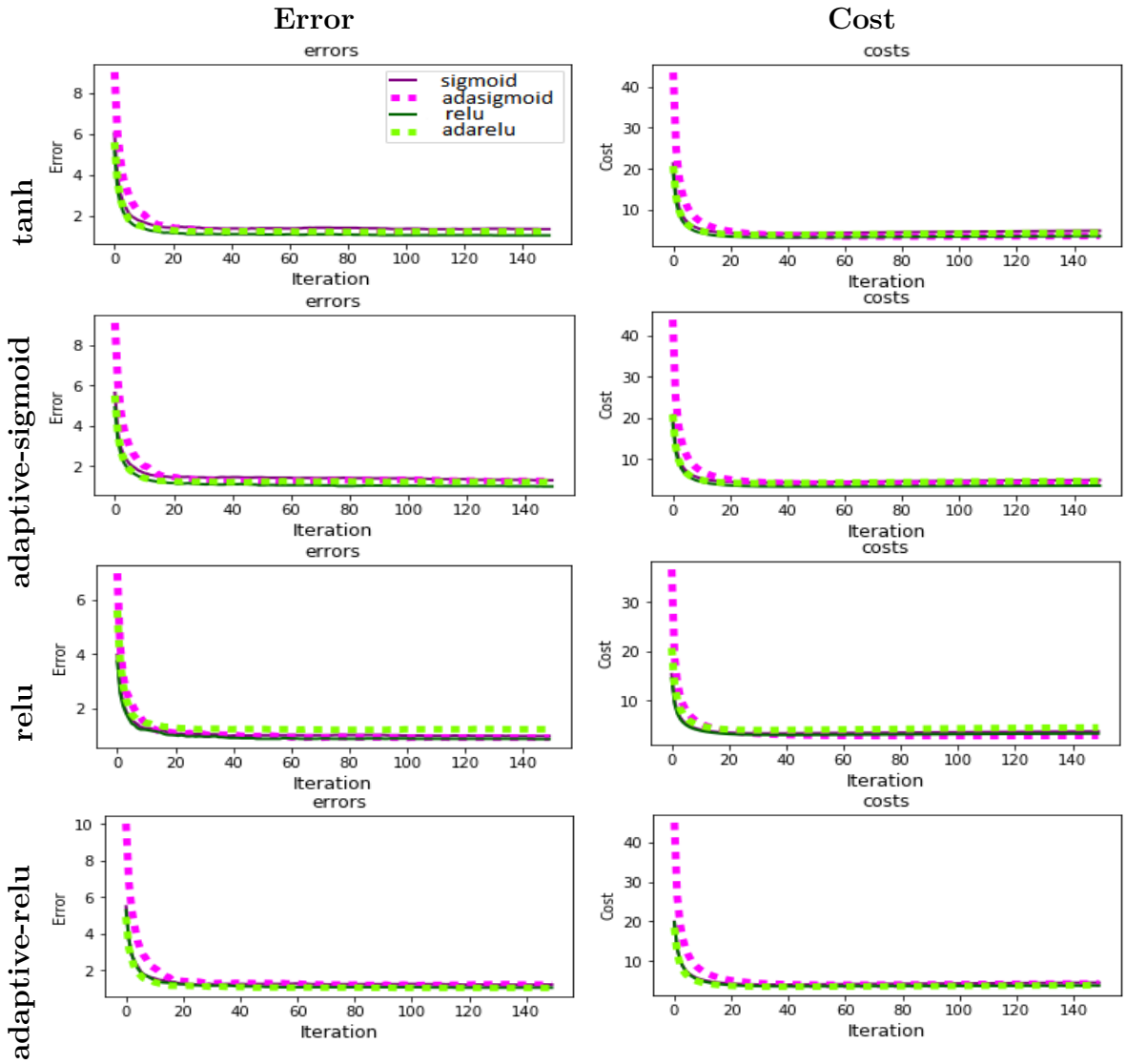


Figure 5.10 Comparison of the errors and cost curves for CNN models with different configurations. The activation functions in convolutional layers of CNN models are tanh, adaptive-sigmoid, relu and adaptive-relu. All CNN models are trained with batch normalization. The comparison of the error curve of fitted CNNs with sigmoid, adaptive-sigmoid, relu and adaptive-relu are demonstrated in each single graph (MNIST data).

5.4 Experiment 3 : Sentiment Analysis

In this work, a series of experiments using CNN-based models trained on top of pre-trained word vectors for sentiment analysis task is reported. We show that a generic form of LeNet5 CNN with adaptive activation function achieves good results on Movie Review benchmark for sentiment analysis application.

5.4.1 Data

The data in this experiment is a version of the Movie Review (MR) dataset [4] which is introduced in (Pang et Lee, 2004) and it consists of 2000 movie reviews, 1000 positive and 1000 negative reviews considered as labels all submitted before 2002. The review classification of positive or negative are extracted automatically from rating information of the original reviewer. The data only includes the reviews where the author marked the movie's rating with stars or some numerical values. The maximum number of reviews that each author is allowed to have is determined to be 20 in the original corpus. This corpus is also referred to as the polarity dataset (Pang et Lee, 2004). Since the data is not splitted separately as training, test and validation sets, one should perform a cross-validation analysis to report the average performance evaluation and compare the final results.

5.4.2 Data preparation

It should be noted that for the sentiment analysis applications a feature vector for each review is required. The desired feature vector could be obtained using Bag of Words (BOW) or Tf-idf methods. In our experiment, a typical CNN was trained with one convolution layer on top of word vectors fitted from an unsupervised neural language model. These word vectors were trained based on the word2vec method proposed in (Mikolov *et al.*, 2013) on 100 billion words of Google News which is publicly available [5]. The dimensionality of these word vectors is 300 and they are trained using the continuous BOW model. The word2vec method was used to transform each word into a vector such that the transformation should preserve the words semantics. This means that if two words have similar meaning, therefore we expect that their corresponding vectors should be close as well. If a word does not exist in the set of pre-trained words, it would be initialized randomly (Mikolov *et al.*, 2013).

4. http ://www.cs.cornell.edu/people/pabo/movie-review-data/
5. https ://code.google.com/archive/p/word2vec/

5.4.3 Models

A generic architecture of CNN in (Kim, 2014) with two convolutional layers is used to classify the sentences. This model takes concatenated word2vec vectors as input and fed them into the convolutional layers. Filter windows with fixed size and same weights in convolutional layers move across the sentence and produce the feature maps by convolving with each small part of the sentence. Convolutional layers extract the spatial features from the sequence of words arranged in a sentence almost the same way that high-order n-gram features collect locally features from a text. The CNN model could be comprised of several filters with different window sizes and weights so that each one establish a channel. Each word is a k-dimensional word vector represents the word i in the given sentence. Sentences are aligned to a specific size, such that sentences with smaller sizes will be padded by zeros.

A convolution operation is applied over a window of a small number of words to generate a new feature. The relative significance of the words are captured by applying the max-pooling layer in order to choose the maximum value from each new feature vector and down-sampled to a new sentence vector as well. The sentence vector is often fed into a one or two hidden-layer network with fully-connected weights to finally produce the output vector at softmax layer. A generic variant of CNN model for sentence classification is illustrated in Figure below.

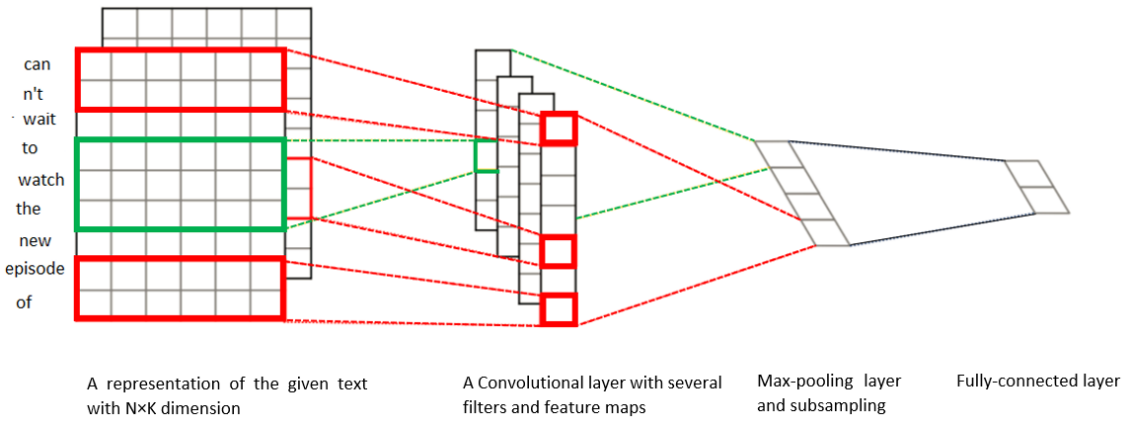


Figure 5.11 CNN architecture with two channels for sentence classification reproduced from (Kim, 2014).

A CNN model with pre-trained word2vec vectors called static in (Kim, 2014), is used in our experiments. In this variant, all known and unknown words are considered fixed and the other parameters of the model will be fitted. The static CNN we use involves two convolutional layers each of them followed by a max-pooling layer and a fully-connected layer at the end.

The fully-connected network includes one hidden layer with 100 neurons and a softmax output layer for binary text classification. Model fitting is performed using the stochastic gradient descent across the mini batches of size 50. The optimization process iterated in 50 epochs and the mini batches are shuffled at the end of each epoch. Two commonly-used penalization methods, Dropout (with probability = 0.5) and L_2 regularization are considered as well.

For consistency, same data, pre-processing and hyper-parameter settings are used as reported in (Kim, 2014) . Since MR data does not include a separate standard test set, each result in our experiments is reported by an average of accuracy with confidence intervals measured over running a 5-fold cross- validation.

The effect of choosing different CNN configurations regarding activation functions on the model performance is evaluated in terms of accuracy, error and cost convergence. We also evaluated the performance of these models in terms of precision, recall and F1 which the results are reported in appendix B.

5.4.4 Tools and Technologies

Training and results of the CNN models were implemented using a slightly modified implementation of original code [6] in Theano.

5.4.5 Results and Evaluations

The following conclusions are inferred from the evaluation of accuracy of the CNN models on MR benchmark from Table (5.5).

– Using adaptive-sigmoid in convolutional layers yields poor results versus CNN models using tanh, relu or adaptive-relu in convolutional layers, regardless of the choice of activation function for fully-connected layer.
– Using adaptive-relu in convolutional layers accompanied with any activation function in a fully-connected layer often produces fairly good results. The best result is obtained with adaptive-relu in convolutional layers and with sigmoid in fully-connected networks.
– Using adaptive-sigmoid in a fully-connected network nearly gains better accuracy than sigmoid, except when adaptive-relu employed in convolutional layers. Similarly, adptive-sigmoid in fully-connected outperforms the CNN with relu and adaptive-relu regardless of the choice of activation function in convolutional layers.
– According to our results, one could say that adaptive-sigmoid could be employed as the activation function for fully-connected layer in a CNN model. Furthermore, adaptive-relu

6. https ://github.com/yoonkim/CNN-sentence

Table 5.5 Static CNNs models with two convolution layers (each of them following a max pooling layer) and one fully-connected network at the end. Functions in rows indicate the activation functions used in convolutional layers and functions in columns show activation functions in fully-connected network. The hyper parameters in all models are same including image dimensions (img_w) = 300, filter sizes = [3, 4, 5], each have 100 feature maps, batch_size = 50, $\gamma = 0.05$, dropout = 0.5, $\lambda_2 = 3$, number of hidden layers = 1, number of neuron = 100, number of epochs = 50. The results are reported as the average accuracy from 5-fold cross-validation (MR data).

		Fully-connected layer			
		sigmoid	ada-sigmoid	relu	ada-relu
Convolutional layer	tanh	78 ± .55	78.6 ± .47	77.9 ± .37	77.7 ± .45
	ada-sigmoid	78 ± .56	78.7 ± .46	52.3 ± .71	77.4 ± .49
	relu	**79.7 ± 52**	78.9 ± .51	79.1 ± .29	79.1 ± .47
	ada-relu	**79.8 ± .62**	**79.3 ± .49**	78.9 ± .31	78.5 ± .51

works excellent when it is used as the activation function in convolutional layers.

The best results of our models versus results reported in (Kim, 2014) are presented in Table (5.6). Despite our best static CNN models with relu or adaptive-adaptive-relu in convolutional layers and sigmoid in fully-connected layer, we do not outperform the CNN-static, CNN-nonstatic and CNN-multichannel in (Kim, 2014), but they produce extremely close results to those reported in (Kim, 2014). These results suggest that our proposed activation functions work good on benchmark data and could be employed in text classification and sentiment analysis as well. Applying the proposed adaptive activation functions in nonstatic and multichannel variants of CNN, along with fine-tuning the hyper parameters may yield further improvements.

As can be observed from Figure 5.12, the error curves corresponding the CNN model with adaptive-sigmoid in fully-connected layers, converges fastest of all models, regardless of the activation function used in convolutional layers. By contrast, models with adaptive-relu in fully-connected layers always show a slow convergence. CNN model associated with ada-sigmoid in convolutional layers and relu in fully-connected layers does not converge during the time that most of the models converge.

Likewise, cost curves for models along with adaptive-sigmoid in convolution layers achieve the minimum cost in contrast to their counterparts but with different convergence speeds. Like error, the cost curve shows a poor convergence when the CNN model uses ada-sigmoid in convolutional layer and relu in fully-connected.

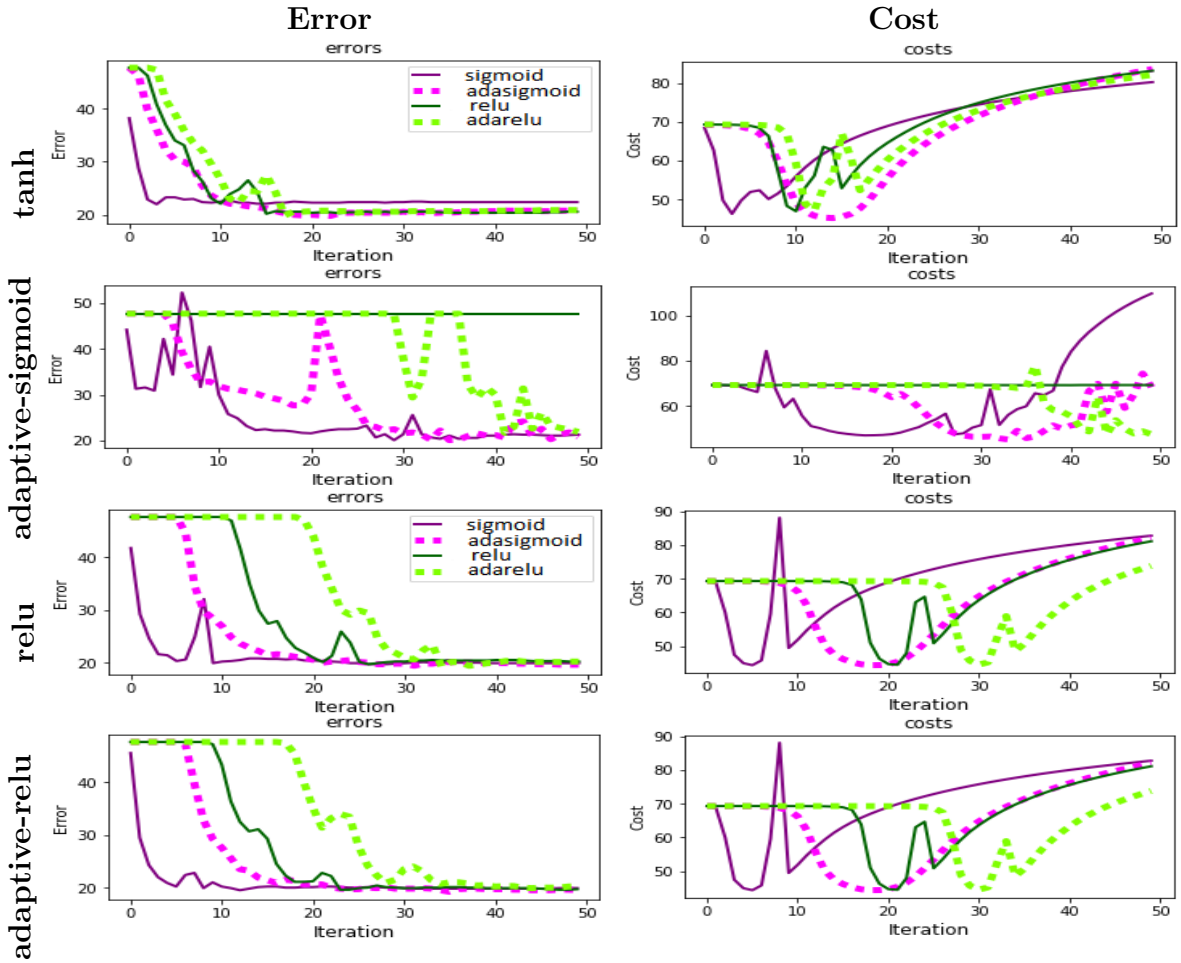


Figure 5.12 Comparison of the error and cost curves of our static CNNs models with two convolution layers (each of them following a max pooling layer) and one fully-connected network at the end. The hyper parameters in all models are same including image dimensions (img_w) = 300, filter sizes = [3, 4, 5], each have 100 feature maps, batch_size = 50, $\gamma = 0.05$, dropout = 0.5, $L_2 = 3$, number of hidden layers = 1, number of neuron = 100, number of epochs = 50. a) CNN with tanh in convolutional layer, b) CNN with adaptive-sigmoid in convolutional layer, c) CNN with relu in convolutional layer and d) CNN with adaptive-relu in convolutional layer (MR data).

Table 5.6 Comparison between our best models and the state-of-the-art results on Movie Review data. best-CNN-1 : CNN with adaptive-relu in convolutional layer and sigmoid in fully-connected network, best-CNN-2 : CNN with relu in convolutional layer and sigmoid in fully-connected network and best-CNN-3 : CNN with adaptive-relu in convolutional layer and adaptive-sigmoid in fully-connected network.

Model	Accuracy
CNN-rand (Kim, 2014)	76.1
CNN-static (Kim, 2014)	81.0
CNN-non-static (Kim, 2014)	81.5
CNN-multichannel (Kim, 2014)	81.1
best-CNN-1	79.8 $\pm$ .62
best-CNN-2	79.7 $\pm$ .52
best-CNN-3	79.3 $\pm$ .49

More information is provided to compare the performance of each model, separately in terms of precision, recall and $F1$ per each class in Appendix B.

5.4.6 Conclusions

In this experiment, our motivation is to evaluate the performance of a generic type of CNN model along with proposed adaptive activation functions compared to standard functions on a commonly-used text benchmark, Movie Review data. Furthermore, we aim at understanding how well our proposed activation functions could perform an accurate prediction compared to the state-of-the-art results in a sentiment analysis application.

Accordingly, our empirical results show that using adaptive-sigmoid, as the activation function, in fully-connected layer is a good choice. Moreover, adaptive-relu could work excellent when it is applied in convolutional layers.

A comparison between the best results of our experiments (reported from 5-fold cross-validation) and the state-of-the-art results in (Kim, 2014) on the same version of Movie Review data for sentiment analysis application shows that our adaptive activation functions perform fairly good results on Movie Review data in terms of accuracy. Applying those adaptive activation functions with fine-tuning the hyper parameters may result in further improvement.

These findings on Movie Review data motivate us to apply the proposed adaptive activation functions on URLs dataset in order to predict the user intentions.

5.5 Experiment 4 : Company Data

5.5.1 Dataset

The data set collected by an e-commercial company was mainly extracted from the answers that the anonymous visitors provided in the online surveys while they visited a commercial website over 3 months (from July 1, 2015 to September 30, 2015). In those surveys, visitors are typically asked to tell their purpose of visit from the "Browse-Search product", "Complete transaction-Purchase", "Get Order-Technical Support" or "Other" options, considered as four class labels. In this experiment, we aim at studying the relationship between user behavioral data which includes URL sequences as features and the stated purpose of the visit provided in the survey after the end of sessions. The data consists of approximately 13480 user sessions and for each, we consider a maximum of 50 page visits.

5.5.2 Problem Definition

Our goal is to construct a model which can predict the e-commerce website visitors intentions based on the URL sequence they navigated from page to page. The input is the CM data in the form of URL sequence and for each data example the output is the purpose of visit.

5.5.3 Company Data Preparation

Each visitor navigates approximately 14 pages in each visit. Thus, our data is too sparse. Although nearly 1% of the visits lead to a purchase, this number is not reflected in our raw data after preparation step and feature engineering employed in our experiments.

In the raw dataset, URL features are in the form of categorical data including 187,375 URL clicks which 29,375 URLs are unique. Due to the large number of URLs, it is impossible to classify each URL to one unique feature. This will result in an extremely large feature space, which in turn deteriorates our predicting performance. Therefore, to reduce the number of features, a URL feature categorization method is required to identify each URL to a small domain of categories. This challenge could be referred to as the Web page classification without a page content in literature (Baykan *et al.*, 2011). In the following section, several methods for the problem of URL categorization based on URL strings are described and different techniques to map a given URL to a numerical feature vector are discussed. In this project we evaluate the performance of the designed model using different feature engineering methods for URL sequences categorization including Bag of Words (BOW), Term Frequency-Inverse Document Frequency (Tf-idf), Latent Dirichlet Allocation (LDA) and k-

means Clustering. These methods are used since they were also employed in related work, and they are commonly-used algorithms in unsupervised text classification applications. URL categorization is the task of mapping a URL to a predefined category regardless of considering its content in an unsupervised learning process. The performance of the visitor intention prediction can be very sensitive to feature representation depending on the way in which URL sequences are encoded. Suppose we have the this URL as a string :

"http ://www.companyname.com/support/home/productsupport/servicetag/drivers ?s=DHS". The URL must be treated like a text string. First, a text cleaning process including word tokenizing, removing non-ascii and stop-words, standardizing words, stemming and lemmatizing is required to extract the following clean string made up of the token words : "companyname", "support", "home", "productsupport", "servicetag", "drivers", "DHS". Resulting tokens of length less than 2, "http" and "www" are removed.

In following we describe how this string of tokens could be categorized with BOW, Tf-idf, LDA and k-means.

BOW : The BOW model is a commonly-used representation used in Natural Language Processing (NLP) for document classification such that a sentence is represented as the multi-set of its words without considering grammar and word order (Sivic et Zisserman, 2009). BOW creates a vocabulary set out of all the words in the corpus and for each sentence creates a vector of word-counts pertaining to that instance. We simply put all the clean URL strings into a single file, count the frequency of each word in our file and rank them according to their frequencies and selecting the top N words (in our dataset, $N = 30$). Next, we classify each URL in a sequence according to those top N keywords. Each URL could be categorized into the deepest keyword from the top list if that keyword exists in the URL. Here, deepest means the one with the maximum frequency in the corpus. Note that if a URL cannot be categorized by using these N Keywords, it will be simply ignored. Using tokens, we could also add 1-gram and 2-gram derived from tokens to add the encoded ordinal information about the URLs into the feature space.

Tf-idf : The Tf-idf representation is an extension of BOW, provides a clever way of weighting a given word for a given example. Whereas BOW weights every word across all examples, Tf-idf provides a weight to a word based on both its frequency within the given sentence and its frequency over all sentences in the corpus. Tf-idf provides a way of statistically linking the weight of a word in the feature vector to its overall importance both within and across sentences. The Tf-idf weight for word i in sentence j is :

$$\mathrm{Tf} - \mathrm{idf}_{\mathrm{ij}} = \frac{N_{ij}}{N_j} \log(\frac{N_i}{N})$$

where N_{ij} denotes the number of times word i is in sentence j, N_j is the number of total words in sentence j, N_i represents the number of times word i is seen in the corpus and N is the number of total words in corpus (Ullman, 2011). Finally, we simply put all the clean URL strings into a single file, calculate the Tf-idf score of each word in the corpus and rank them according to their Tf-ids scores and choosing the top N words (in our dataset, $N = 30$). Each URL could be categorized into the highest-scores keyword from the top list if that keyword exists in the URL. Similar to BOW, we could add 1-gram and 2-gram features into the categorized URL features in the dataset.

LDA : LDA is a generative probabilistic model for collections of discrete data (text corpora). A three-level hierarchical Bayesian model, wherein each item of a collection is modeled as a finite mixture over an underlying set of topics (Blei *et al.*, 2003). Each topic is, in turn, modeled as an infinite mixture over an underlying set of topic probabilities. In the context of text modeling, the topic probabilities provide an explicit representation of a document. The simple idea behind LDA is that documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words. In this method, each document is considered as a mixture of different topics with an assumption of having a Dirichlet prior which yields more mixtures of topics in a document (Blei *et al.*, 2003). To categorize the URLs using LDA, we first need to parse each URL in the sequences to obtain the set of tokens. Now, we have a collection of tokens as the documents. We design a document-matrix which keeps the frequency of each word (in vocab set) in each document. Going through each URL as the document, the LDA method first assigns each token in the URL to one of the N topics randomly, (N is the number of desired topics we tend to find). This randomness provides the topic representations of all the URLs and token distributions over all the topics. By iterating this process for each URL, we go through each token in URL. For each topic, we have to calculate the probability of $p(\text{topic}|\text{URL})$ which is the proportion of tokens in that URL which are currently assigned to topic, and the probability of $p(\text{token}|\text{topic})$ which equals to the proportion of having that topic over all URLs that come from this token. Repeating the assignment of tokens to a new topic, where we pick that topic with $p(\text{topic}|\text{URL}) \times p(\text{token}|\text{topic})$ based on the Generative model.

k-means : k-means clustering is a technique of partitioning data into k groups, where each data example is assigned to the closest cluster based on its distance from the center of that cluster. This clustering method could also be used as a feature in the learning phase in unsupervised learning (Coates et Ng, 2012). In order to use k-means clustering with text documents, we have to first transform the given text corpus into a numeric format by creating the Tf-idf or BOW sparse document-term matrix. Next, the k-means classification method will be applied to create $k = N$ text categories. Finally, for each visitor in our dataset,

instead of using the URL sequence itself, we build a data set made up of the sequence of URL category features belonging to $N = 30$ categorical set. The 2-gram features could be added to the feature space to keep the ordinal information corresponding to each URL category.

The data created by each of the above categorization methods will be of the form of categorical (nominal) data. Some machine learning algorithms could work with nominal data directly without any transformation e.g. Decision Trees. Most machine learning methods cannot work on nominal data directly. To tackle this problem, a transformation technique is often applied to convert the nominal data into a numerical form using the one-hot encoding method. Using a URL categorization and one-hot encoding transformation will give us a dataset in numeric format with 13480 samples and 1536 features.

We also performed the effect of feature representation obtained by different URL categorization methods on user intention prediction based on several standard machine learning classifiers like Naive Bayes (NB), Logistic Regression (LR), Gradient Boosting (GB) and Random Forest (RF). The results indicated that using different categorization methods does not remarkably improve the final results of the baseline classifiers in our application. The input is the CM data in the form of URL categories in numeric format and for each data example the output is the purpose of visit. The class of interest in this application is "Complete transaction-Purchase" and the goal is to build a model which can closely predict the visitors with purchase intentions. Therefore, we continued our further experiments according the URL categorized data obtained by Tf-idf due to the simplicity and resources it uses.

According to Figure (5.12), since the classes are not distributed equally in our data, accuracy metric is not enough to measure the usefulness of the learned models. A good strategy is to include the other appropriate metrics in our evaluation approaches. In this experiment, we apply precision and recall metrics per class to measure the performance of the learned classifiers according to each class.

By running a 10-fold cross validation experiment for hyper parameter tuning and comparing the classification report graphs it is understood that in terms of accuracy, LR and RF algorithms give best results.

The following conclusions are inferred from the evaluation of performances of baseline classifiers based on data categorized by Tf-idf illustrated in Figure 5.14.

– In terms of average precision, GB outperforms the other counterparts by the virtue of its ensemble functionality. From a recall perspective, LR and RF demonstrate the better results compared to NB and GB.
– In terms of precision per class 0, 1, 2 and 3, algorithms LR, RF, GB and GB obtain better

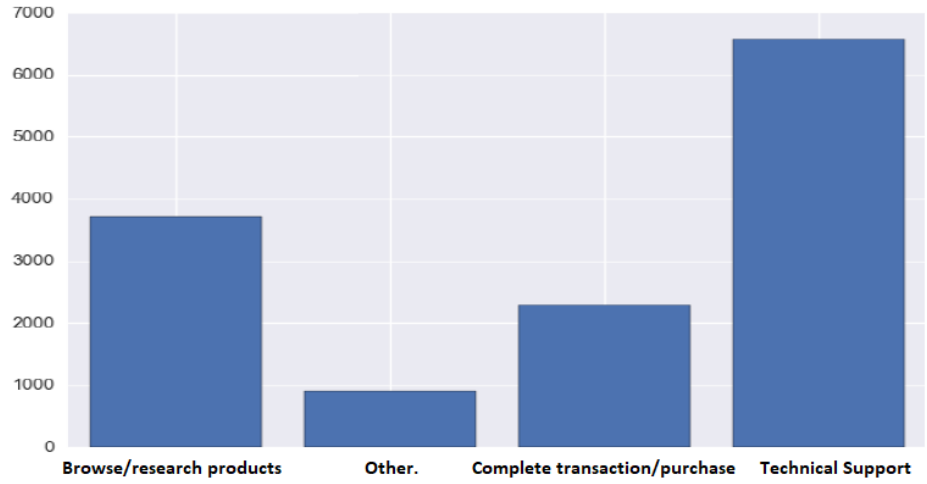


Figure 5.13 Class distribution for company data. class 0 : browse/search product, class 1 : complete transaction/purchase, class 2 : get order support/technical support and class 3 : other.

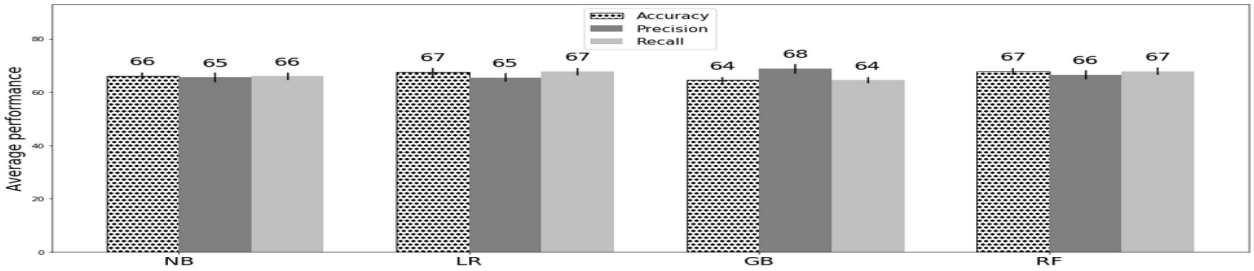
results respectively compared to other classifiers which means that they could perform more relevant prediction corresponding to each class with least false positives.

– In terms of recall per class 0, 1, 2 and 3, algorithms, GB, LR, LR and LR outperform other classifiers respectively which means that they could return more truly related results compared to their counterparts.

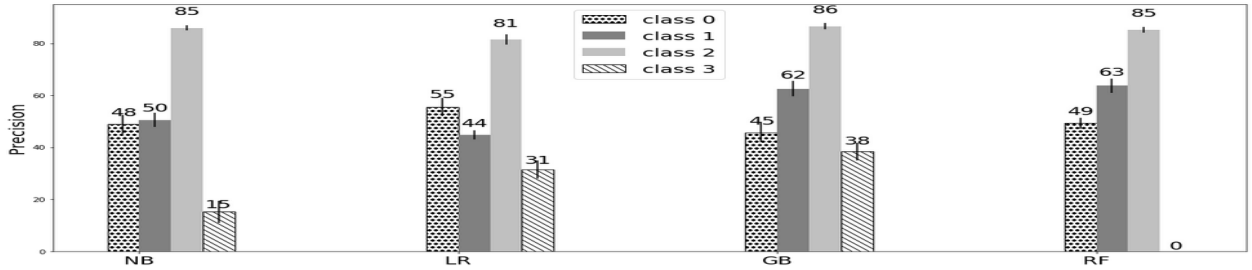
Neural Network Models :

The following conclusions are derived from the comparison of accuracy obtained by classical 1NN, 2NN and CNN models with/without dropout in Figure 5.15 and Figure 5.16 :

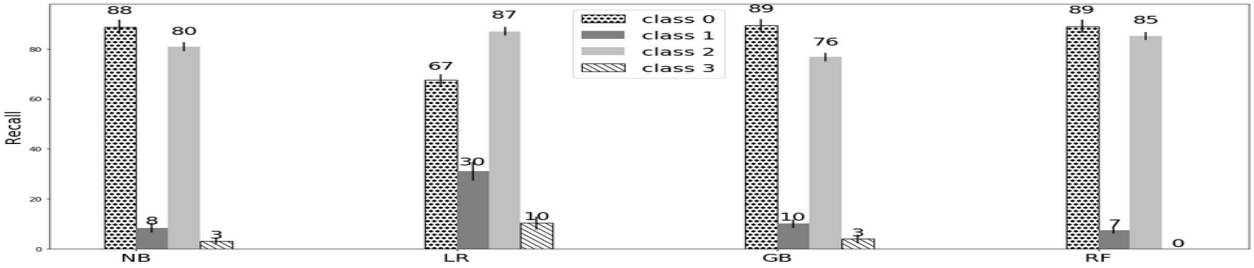
– Using dropout as regularizer in 1NN, 2NN and CNN models improve the accuracy, average precision, and average recall.
– In terms of accuracy and average precision, CNN with dropout outperforms the other models.
– In terms of average recall, 1NN with dropout gives the best results among the all models.
– Using CNN without dropout considerably improves the results in terms of average accuracy, precision and recall compared to 1NN and 2NN in Figure 5.15 (a).
– CNN with dropout improves the average accuracy and precision, but not recall compared to 1NN and 2NN with dropout in Figure 5.16 (a).
– In Figure 5.15(b), using CNN without dropout yields a considerable improvement in precision of class 1, 2 and 3.
– Comparing figures (b) in 5.15 and 5.16, it is clear that using dropout in 1NN and 2NN leads to a noticeable increase in precision per class 2 and 3.



(a) Average Performance

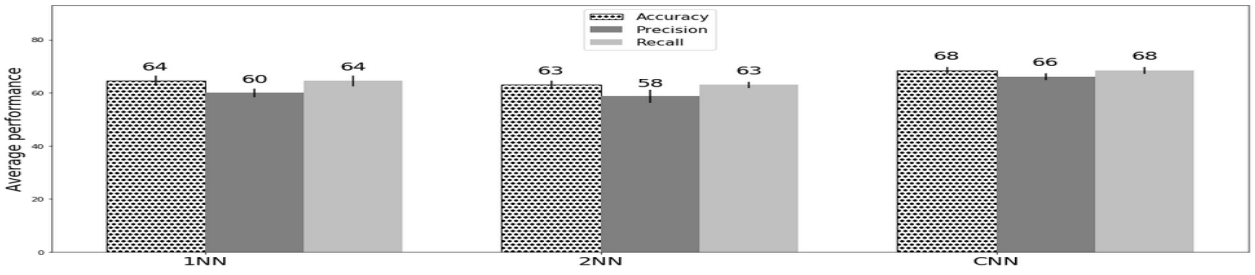


(b) Precision per class

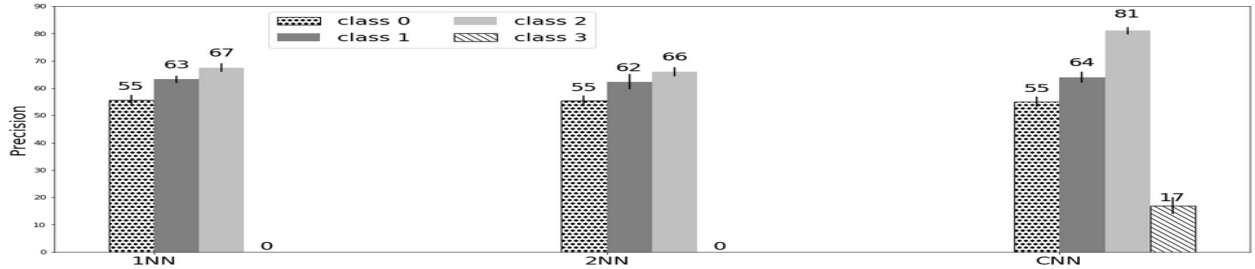


(c) Recall per class

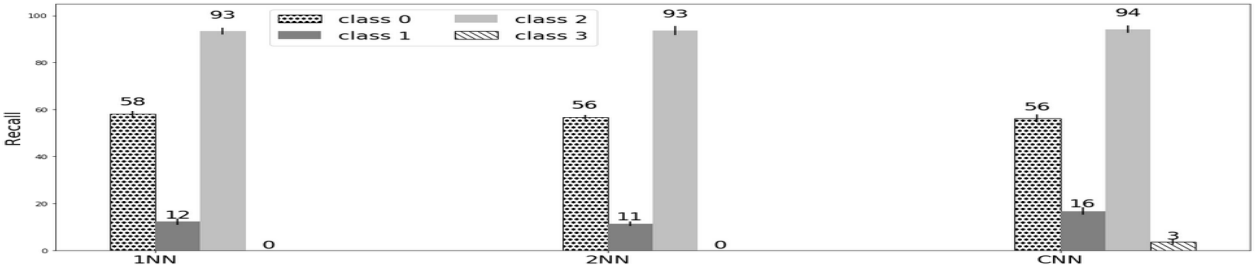
Figure 5.14 Comparison of the performance of the user intention prediction with different baseline classifiers (NB, LR, GB and RF) over URL categorized by Tf-idf. a) comparison based on average accuracy, precision and recall, b) comparison based on precision per class and c) comparison based on recall per class.



(a) Average Performance

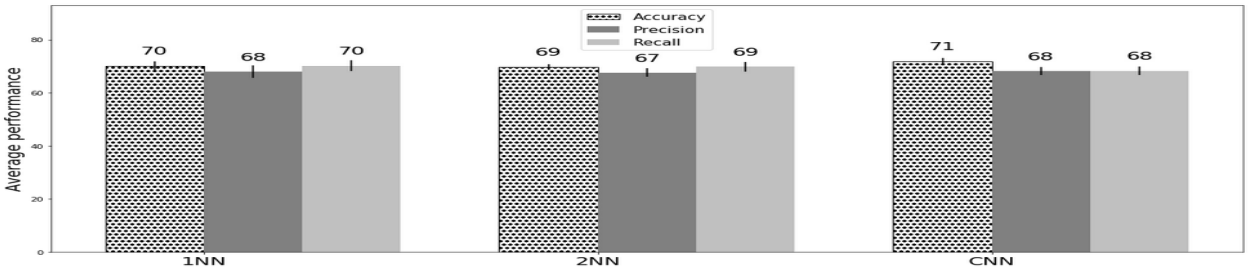


(b) Precision per class

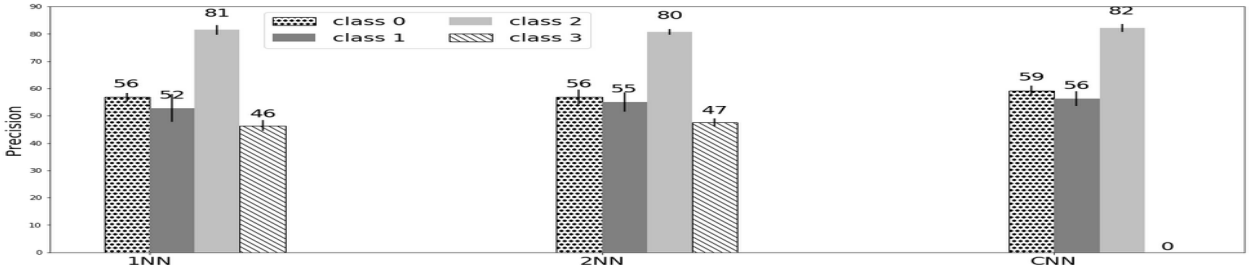


(c) Recall per class

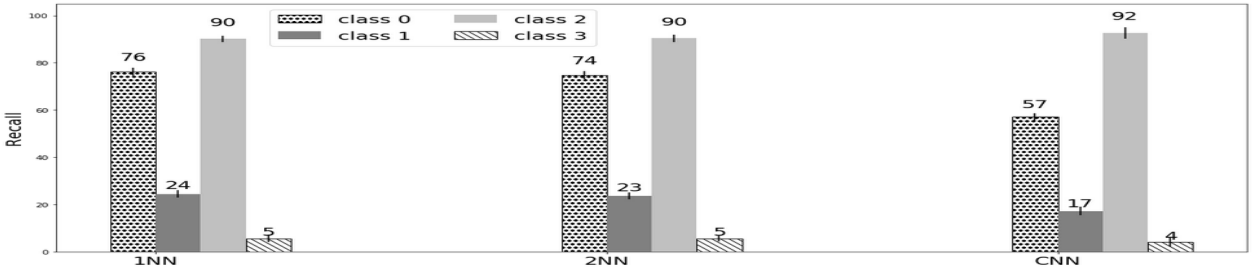
Figure 5.15 Comparison of the performance of the user intention prediction with with 1NN, 2NN and CNN without dropout over URL categorized by Tf-idf. a) comparison based on average accuracy, precision and recall, b) comparison based on precision per class and c) comparison based on recall per class.



(a) Average Performance



(b) Precision per class



(c) Recall per class

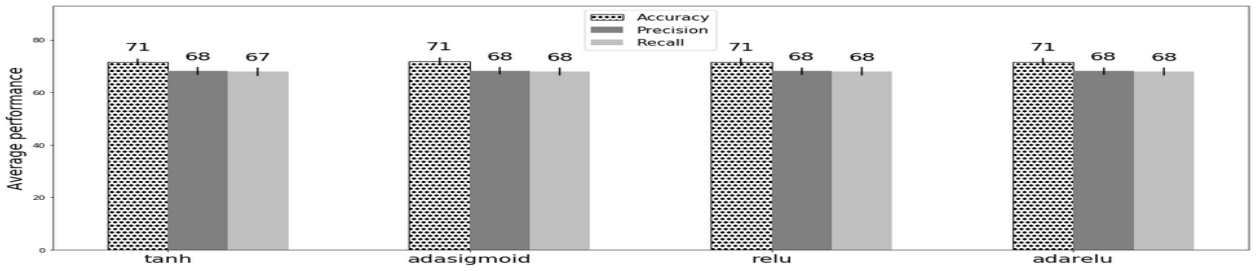
Figure 5.16 Comparison of the performance of the user intention prediction with with 1NN, 2NN and CNN with dropout ($p = .5$), over URL categorized by Tf-idf. a) comparison based on average accuracy, precision and recall, b) comparison based on precision per class and c) comparison based on recall per class.

– Comparing figures (c) in 5.15 and 5.16, it is evident that using dropout in 1NN, 2NN and CNN improves the results per class 0 and 1, but not per class 2 and 3.
– For class 0, CNN with dropout and 1NN with dropout gives the best results for precision and recall, respectively. For class 1, CNN without dropout outperforms the other models in terms of precision while 1NN with dropout gives the best recall result. For Class 2, the best result in terms of precision is obtained by CNN with dropout while the best results for recall is reported by CNN without dropout. For Class 3, for precision, 2NN with dropout reports the best performance and 2NN and 1NN both with dropout give the best recall.

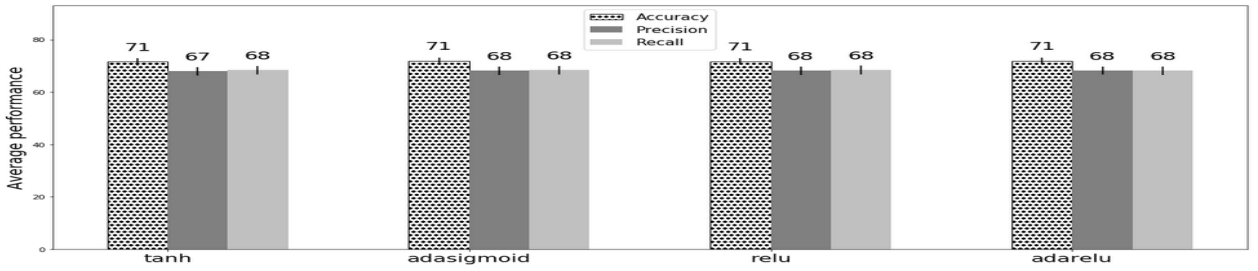
5.5.4 Proposed Models

Motivated by the best results obtained from the Experiment 3 on Movie Review data, we aim at investigating the performance of the static CNN model with proposed activation functions on categorized URL data. Therefore, after performing the preparation step and choosing the best categorization method (Tf-idf) for data representation, we feed the prepared data to the convolutional CNN models applying standard and adaptive activation functions and evaluate the performance of the fitted models on test data.

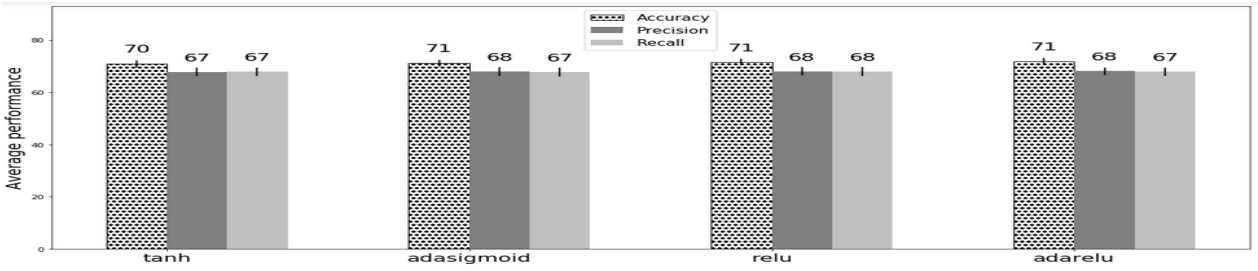
The following conclusions are inferred from comparing Figures 5.17, 5.18 and 5.19 :
– In terms of average accuracy, models employing adaptive-sigmoid in convolutional and fully-connected layers gives the best result. In terms of precision, model with tanh and adaptive-sigmoid in convolutional and fully-connected layers obtains the better result. Also, models with adaptive-sigmoid and adaptive-relu in convolutional and fully-connected layers gives similar results.
– Models using adaptie-sigmoid in convolutional layers report the best avergae recalls.
– For class 0, models with adaptive-sigmoid in both convolutional and fully-connected layers gives the best recall.
– For class 1, models with adaptive-sigmoid and adaptive-relu in convolutional and fully-connected layers and model with adaptive-relu in convolutional and relu in fully-connected produce the better precision. Best recall results are reported by models with adaptive-relu in fully-connected layers.
– For class 2, in terms of precision using tanh in convolutional layers is not recommended. Instead models with relu or adaptive-relu in convolutional layers obtain fairly good results. Having adaptive-sigmoid in convolutional layers produces fairly good results in terms of recall.



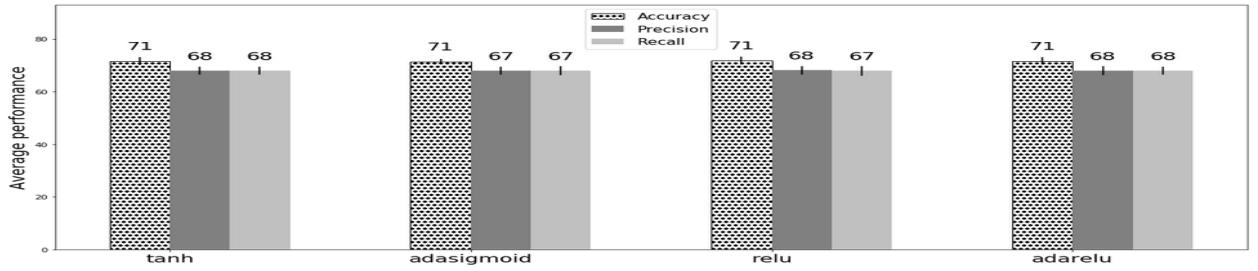
(a) tanh



(b) adaptive-sigmoid

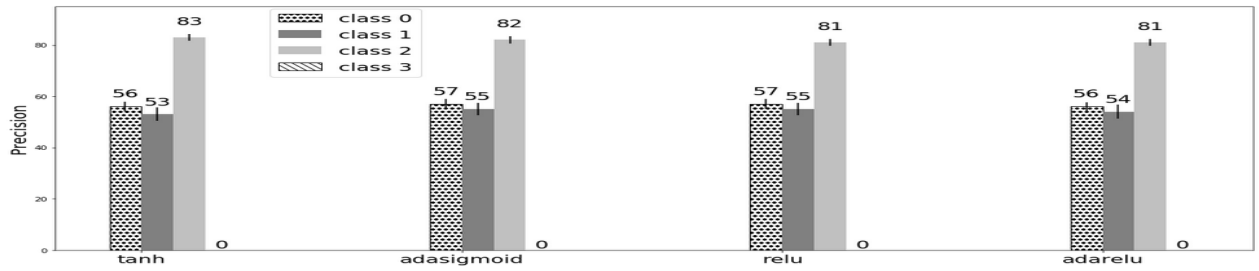


(c) relu

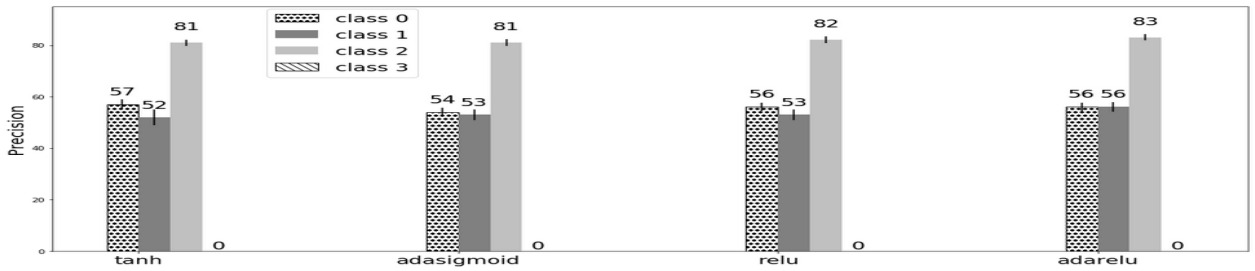


(d) adaptive-relu

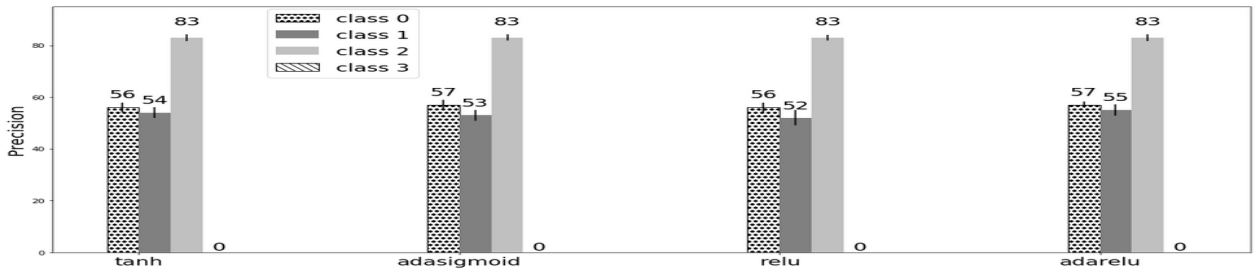
Figure 5.17 Average performance : comparison of the performance of the user intention prediction with different static CNN models fitted with different activation functions in convolutional and fully-connected layers in terms of average accuracy, precision and recall : a) static CNN with tanh in convolutional layers, b) static CNN with adaptive-sigmoid in convolutional layers, c) static CNN with relu in convolutional layers and d) static CNN with adaptive-relu in convolutional layers.



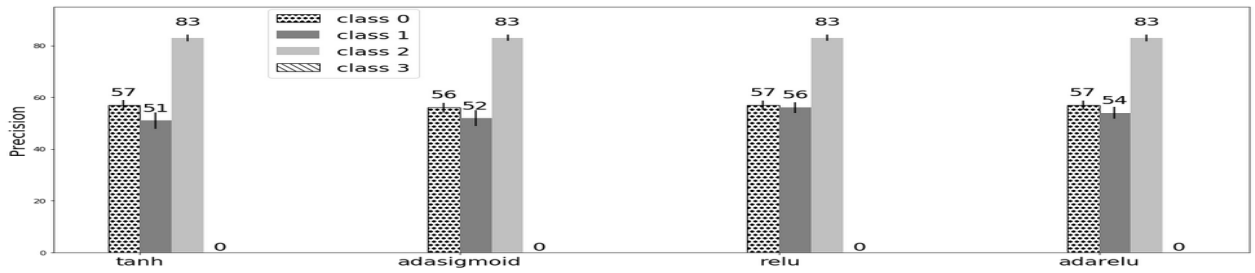
(a) tanh



(b) adaptivesigmoid

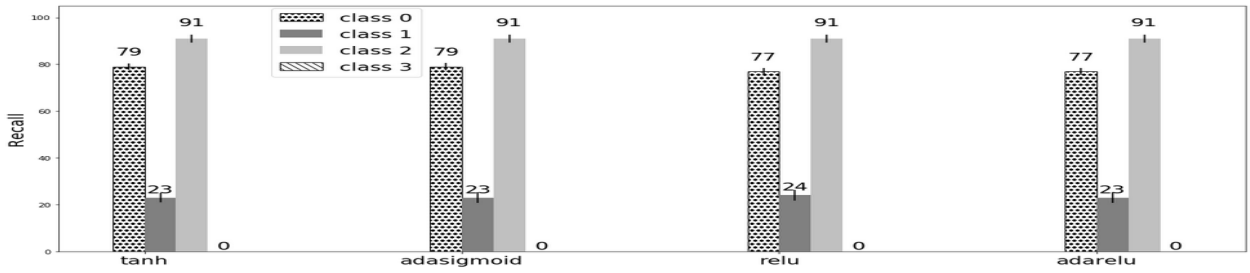


(c) relu

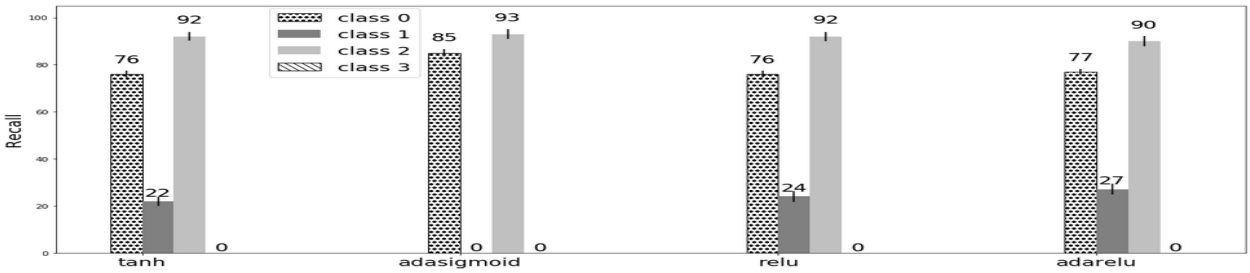


(d) adaptive-relu

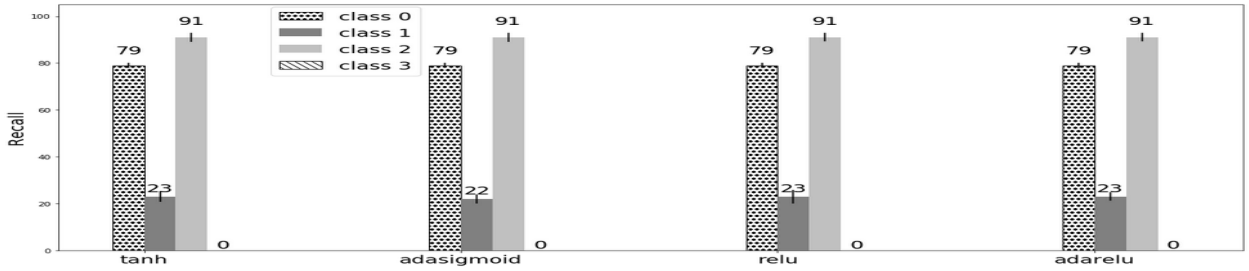
Figure 5.18 Precision per class : Comparison of the performance of the user intention prediction with different static CNN models fitted with different activation functions in convolutional and fully-connected layers : a) static CNN with tanh in convolutional layers, b) static CNN with adaptive-sigmoid in convolutional layers, c) static CNN with relu in convolutional layers and d) static CNN with adaptive-relu in convolutional layers.



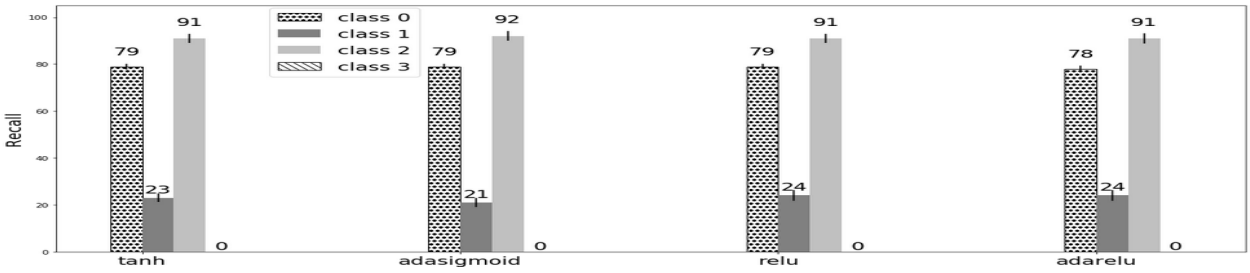
(a) tanh



(b) adaptive-sigmoid



(c) relu



(d) adaptive-relu

Figure 5.19 Recall per class : comparison of the performance of the user intention prediction with different static CNN models fitted with different activation functions in convolutional and fully-connected layers : a) static CNN with tanh in convolutional layers, b) static CNN with adaptive-sigmoid in convolutional layers, c) static CNN with relu in convolutional layers and d) static CNN with adaptive-relu in convolutional layers.

5.5.5 Tools and Technologies

For data cleaning, we use the NLTK 3.2.5 [7] package in Python 2.7 [8]. To implement the BOW, Tf-idf, and k-means, we used the scikit-learn [9] machine learning package in Python 2.7. For LDA, we implemented the codes in genesis [10] package in Python 2.7. For the implementations of the NB, LR, GB and GB classifiers we used the scikit-learn as well and we set the corresponding required hyper parameters to the best ones which are extracted by running a 10 cross-validation experiment, separately. Training and results for user intention prediction using different variants of MLP including our proposed models were implemented using a slightly modified implementation of original code developed by Theano development team [11]. The CNN-based algorithms including the proposed ones were also implemented using the modified implementation of original code [12] in Theano.

5.5.6 Conclusions

Motivated by the good results obtained from sentiment analysis application, we aim at evaluating the performance of a generic variant of CNN model using proposed activation functions on a real-world URLs dataset. Our goal is to study the effect of using adaptive activation functions in deep neural networks for user intention prediction.

The classes in our data are not distributed equally and the class of interest is "Complete-transactions-Purchase". The main objective is to develop a model which can predict excellent the visitors with purpose of purchase based on URLs they navigated at the end of the sessions.

On plus side, in terms of average accuracy, using CNN models with proposed activation functions yields a considerable increase compared to baseline classifiers. Similarly, a negligible improvement is reported in terms of average precision and average recall. Also, using adaptive activation functions in CNN models could improve the results obtained by generic CNN with standard activation functions.

On the minus side, in terms of precision and recall per class 1, class of interest in this application, CNN with proposed activation functions could not exceed the results reported by baseline classifiers. A further downside is an unexpected drop in the performance of the CNN models with adaptive activation functions in terms of precision and recall per class 3,

7. www.nltk.org
8. https ://www.python.org/download/releases/2.7
9. http ://scikit-learn.org/stable/
10. https ://pypi.python.org/pypi/genesis
11. http ://deeplearning.net/tutorial/mlp.html
12. https ://github.com/yoonkim/CNN-sentence

class with minimum number of examples, compared to baseline classifiers. This issue can be viewed from two different aspects. First reason is the lack of enough data per each class for model training (more data in training, more accurate result in test evaluation). Second, URL strings are not rich to extract enough knowledge for learning representation in our studied models. A URL string mostly includes unmeaning and senseless words which do not exist in the set of pre-trained words. Consequently, the corresponding word2vec vectors are often initialized randomly (compared to classical 1NN, 2NN and CNN models using Tf-idf and one-hot encoded features).

CHAPTER 6 CONCLUSION

The main objective in this thesis is to explore the problem of user intention prediction on URLs dataset. Particularly, we aim at proposing two adaptive form of standard activation functions, sigmoid and relu, employed in MLPs and CNNs and evaluate the performance of these suggested models on predicting the purpose of visits in our application.

In first step, we develop an asymmetric and parametrised type of sigmoid function inspired from the Box-cox transformation, called adaptive-sigmoid. Our proposed adaptive-sigmoid allows more skewness adaptablity controlled by a parameter and includes both standard sigmoid and Gumbel as well. Besides, the identifiability of the adaptive-sigmoid with respect to its parameters in a Bernouilli model is studied as well. Furthermore, a parametric smooth relu function, called adaptive-relu is proposed in Chapter 4. We also indicate how the parameters of the adaptive-sigmoid and adaptive-relu could be learned along with the other MLP parameters in backpropagation.

Next, we design a series of experiments to understand how our proposed activation functions affect on the prediction, representation, bias and convergence of the fitted models using a simulated data. The results of this experiment show that under the assumption of having a simulated deep MLP model accompanied with standard sigmoid or standard relu, fitting models with proposed activation functions approximates the bias and true labels well in original models compared to models with standard functions. Moreover, using adaptive activation functions in fitting models for prediction approximation is superior compared to standard functions in terms of accuracy, precision, recall and F1, regardless of network size and activation functions used in original model. The flaw is that models with adaptive activation functions could not approximate the weight representations in the original models due to the non-identifiability nature of neural networks.

In the next step, we aim at evaluating the performance of the typical MLP and CNN models using our proposed activation functions on image and text data. Accordingly, we design a series of experiments on MNIST as a widely-used image benchmark to understand how accurate the adaptive activation functions in MLP and CNN models classify the hand-written digit images compared to standard activation functions. The results obtained from this experiment suggest using adaptive-relu in MLP to improve the accuracy and to converge faster. Besides, compared to standard sigmoid, applying adaptive-sigmoid in fully-connected layer of the CNN models are recommended. Overally, MLP and CNN models using proposed activation functions work excellent in prediction and convergence compared to models that work

exclusively with standard functions.

Additionally, a series of experiments using CNN models trained on a top of word2vec text data is performed to evaluate the performance of the proposed activation functions in a sentiment analysis application on Moview Review data. Our empirical results imply that using adaptive-sigmoid as activation functions in fully-connected layer and adaptive-relu in convolutional layers are strongly recommended. These observations are consistent with the findings were noticed in experiments on MNIST data. Also, a comparison between our best observations and the state-of-the-art results in (Kim, 2014) indicates that despite the excellent accuracy of our static CNN with proposed activation functions, these adaptive activation functions could not gain further improvement compared to the results reported in this paper. We believe that applying more fine-tuning hyper parameters and using other variants of CNN models accompanied with our proposed activation functions could improve the existing results.

Motivated by the results obtained from neural networks using adaptive activation functions on Moview Review data, we design a series of experiments to apply our proposed activation functions in CNN models. The application of this experiment is the user intention prediction which performs on a real-world URLs data.

One of the main challenges in our application is the size of imbalanced data. Additionally, using word2vec represntation for URLs data is a bit tricky due to the fact that URL strings are too short and mostly consist of meaningless words in English dictionary. On the positive side, results show that using CNN models with adaptive activation functions improve the average accuracy, significantly compared to baseline classifiers. In the same way, an improvement in average precision and average recall is reported. The downside is that using CNN with adaptive activation functions yields a dramatic drop in performance of the prediction of visitors with purpose of purchase (class of interest) in terms of precision and recall. This issue could be explained in two ways. First, the data size is not large enough to train an accurate model. Second reason might be explained in a way that URL strings are often includes senseless tokens and not rich to extract enough knowledge for learning representation in our studied models.

To recap, our empirical experiments on two well-known image and text benchmarks imply that by virtue of using adaptive-activation functions in MLP and CNN models, we can improve the performance of the deep networks in terms of accuracy and convergence. Running deep networks with our proposed activation functions on URLs data yields a remarkable improvement in terms of average accuracy, average precision and average recall, but does not gain any improvement in terms of precision and recall for visitors with purpose of purchase (class of interest). One reason might be originated from the vector representation we used as

URL features. Another reason is the lack of enough data for training our proposed models. More efforts and experiments to verify the effectiveness of the CNN models with proposed activation functions in user intention prediction application is required.

REFERENCES

Agostinelli, Forest and Hoffman, Matthew and Sadowski, Peter and Baldi, Pierre (2014). Learning activation functions to improve deep neural networks. *arXiv preprint arXiv :1412.6830*.

Agresti, Alan (2003). Building and applying logistic regression models. *Categorical Data Analysis, Second Edition*, 211–266.

Agresti, Alan and Kateri, Maria (2011). Categorical data analysis. *International Encyclopedia of Statistical Science*, Springer. 206–208.

Basu, Dev (1955). On statistics independent of a complete sufficient statistic. *Sankhyā : The Indian Journal of Statistics (1933-1960)*, *15*(4), 377–380.

Battiti, Roberto (1989). Accelerated backpropagation learning : Two optimization methods. *Complex Systems*, *3*(4), 331–342.

Baykan, Eda and Henzinger, Monika and Marian, Ludmila and Weber, Ingmar (2009). Purely url-based topic classification. *Proceedings of the 18th International Conference on World Wide Web*. ACM, 1109–1110.

Baykan, Eda and Henzinger, Monika and Marian, Ludmila and Weber, Ingmar (2011). A comprehensive study of features and algorithms for url-based topic classification. *ACM Transactions on the Web*, *5*(3), 15.

Baykan, Eda and Henzinger, Monika and Weber, Ingmar (2008). Web page language identification based on urls. *Proceedings of the VLDB Endowment*, *1*(1), 176–187.

Bengio, Yoshua (2012). Practical recommendations for gradient-based training of deep architectures. *Neural Networks : Tricks of the Trade*, Springer. 437–478.

Bengio, Yoshua and Ducharme, Réjean and Vincent, Pascal and Jauvin, Christian (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, *3*(Feb), 1137–1155.

Bishop, Christopher M (2006). *Pattern Recognition and Machine Learning*. springer.

Blei, David M and Ng, Andrew Y and Jordan, Michael I (2003). Latent dirichlet allocation. *Journal of Machine Learning Research*, *3*(Jan), 993–1022.

Block, Hans-Dieter (1962). The perceptron : A model for brain functioning. i. *Reviews of Modern Physics*, *34*(1), 123.

Bottou, Léon (2012). Stochastic gradient descent tricks. *Neural Networks : Tricks of the Trade*, Springer. 421–436.

Bryson, Arthur E (1961). A gradient method for optimizing multi-stage allocation processes. *Proc. Harvard Univ. Symposium on Digital Computers and Their Applications*. 72.

Bryson, Arthur E and Denham, Walter F (1962). A steepest-ascent method for solving optimum programming problems. *Journal of Applied Mechanics*, *29*(2), 247–257.

Cesa-Bianchi, Nicolo and Lugosi, Gábor (2006). *Prediction, Learning, and Games*. Cambridge university press.

Chakrabarti, Soumen and Dom, Byron and Indyk, Piotr (1998). Enhanced hypertext categorization using hyperlinks. *ACM SIGMOD Record*. ACM, vol. 27, 307–318.

Chakrabarti, Soumen and Van den Berg, Martin and Dom, Byron (1999). Focused crawling : a new approach to topic-specific web resource discovery. *Computer Networks*, *31*(11), 1623–1640.

Chambers, Elizabeth A and Cox, David R (1967). Discrimination between alternative binary response models. *Biometrika*, *54*(3-4), 573–578.

Charnes, Abraham and Frome, Edward L and Yu, Po-Lung (1976). The equivalence of generalized least squares and maximum likelihood estimates in the exponential family. *Journal of the American Statistical Association*, *71*(353), 169–171.

Cho, Youngmin and Saul, Lawrence K (2010). Large-margin classification in infinite neural networks. *Neural Computation*, *22*(10), 2678–2697.

Clevert, Djork-Arné and Unterthiner, Thomas and Hochreiter, Sepp (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv :1511.07289*.

Coates, Adam and Ng, Andrew Y (2012). Learning feature representations with k-means. *Neural Networks : Tricks of the Trade*, Springer. 561–580.

Cybenko, George (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, *2*(4), 303–314.

Darken, Christian and Moody, John (1990). Fast adaptive k-means clustering : some empirical results. *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE, 233–238.

Dobson, Annette J and Barnett, Adrian (2008). *An Introduction to Generalized Linear Models*. CRC press.

Dreyfus, Stuart (1973). The computational solution of optimal control problems with time lag. *IEEE Transactions on Automatic Control*, *18*(4), 383–385.

Dushkoff, Michael and Ptucha, Raymond (2016). Adaptive activation functions for deep networks. *Electronic Imaging*, *2016*(19), 1–5.

Fausett, Laurene and Fausett, Laurene (1994). *Fundamentals of neural networks : architectures, algorithms, and applications*. No. 006.3. Prentice-Hall,.

Fawcett, Tom (2006). An introduction to roc analysis. *Pattern recognition letters*, *27*(8), 861–874.

Fukushima, Kunihiko and Miyake, Sei (1982). Neocognitron : A self-organizing neural network model for a mechanism of visual pattern recognition. *Competition and Cooperation in Neural Nets*, Springer. 267–285.

Gill, Jeff (2000). *Generalized Linear Models : A Unified Approach*, vol. 134. Sage Publications.

Glorot, Xavier and Bengio, Yoshua (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 249–256.

Glorot, Xavier and Bordes, Antoine and Bengio, Yoshua (2011). Deep sparse rectifier neural networks. *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 315–323.

Goodfellow, Ian J and Vinyals, Oriol and Saxe, Andrew M (2014). Qualitatively characterizing neural network optimization problems. *arXiv preprint arXiv :1412.6544*.

Goodfellow, Ian J and Warde-Farley, David and Mirza, Mehdi and Courville, Aaron and Bengio, Yoshua (2013). Maxout networks. *arXiv preprint arXiv :1302.4389*.

Graves, Alex and Mohamed, Abdel-rahman and Hinton, Geoffrey (2013). Speech recognition with deep recurrent neural networks. *Acoustics, speech and signal processing (icassp), 2013 IEEE International Conference*. IEEE, 6645–6649.

Harris, David and Harris, Sarah (2010). *Digital design and computer architecture*. Morgan Kaufmann.

Hashemi, Homa B and Asiaee, Amir and Kraft, Reiner (2016). Query intent detection using convolutional neural networks. *International Conference on Web Search and Data Mining, Workshop on Query Understanding*.

Hastie, Trevor and Tibshirani, Robert and Friedman, Jerome (2009). The elements of statistical learning : Data mining, inference, and prediction. *Biometrics*.

Haykin, Simon S and Haykin, Simon S and Haykin, Simon S and Haykin, Simon S (2009). *Neural Networks and Learning Machines*, vol. 3. Pearson Upper Saddle River, NJ, USA :.

He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian (2015). Delving deep into rectifiers : Surpassing human-level performance on imagenet classification. *Proceedings of the IEEE International Conference on Computer Vision*. 1026–1034.

Hebb, Donald Olding (1949). *The organization of behavior : A neuropsychological approach*. John Wiley & Sons.

Hinton, Geoffrey E and Srivastava, Nitish and Krizhevsky, Alex and Sutskever, Ilya and Salakhutdinov, Ruslan R (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv :1207.0580*.

Hornik, Kurt and Stinchcombe, Maxwell and White, Halbert (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, *2*(5), 359–366.

Hou, Le and Samaras, Dimitris and Kurc, Tahsin and Gao, Yi and Saltz, Joel (2017). Convnets with smooth adaptive activation functions for regression. *Artificial Intelligence and Statistics*. 430–439.

Hou, Le and Samaras, Dimitris and Kurc, Tahsin M and Gao, Yi and Saltz, Joel H (2016). Neural networks with smooth adaptive activation functions for regression. *arXiv preprint arXiv :1608.06557*.

Huang, Guan-Hua (2005). Model identifiability. *Wiley StatsRef : Statistics Reference Online*.

Hubel, David H and Wiesel, Torsten N (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, *160*(1), 106–154.

Ioffe, Sergey and Szegedy, Christian (2015). Batch normalization : Accelerating deep network training by reducing internal covariate shift. *International Conference on Machine Learning*. 448–456.

Ivakhnenko, Alexey Grigorevich (1971). Polynomial theory of complex systems. *IEEE Transactions on Systems, Man, and Cybernetics*, *1*(4), 364–378.

Ivakhnenko, Alekseui Grigorevich and Lapa, Valentin Grigorevich (1966). Cybernetic predicting devices. Rapport technique.

Jain, Gaurav and Ko, Jason (2008). Handwritten digits recognition. *Multimedia Systems, Project Report, University of Toronto*, 1–3.

Jarrett, Kevin and Kavukcuoglu, Koray and LeCun, Yann and others (2009). What is the best multi-stage architecture for object recognition ? *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, 2146–2153.

Jou, Chichang and Lee, Hung-Chang and others (2004). Handwritten numeral recognition based on simplified feature extraction, structural classification, and fuzzy memberships. *Lecture Notes in Computer Science*, 372–381.

Kalchbrenner, Nal and Grefenstette, Edward and Blunsom, Phil (2014). A convolutional neural network for modelling sentences. *arXiv preprint arXiv :1404.2188*.

Kan, Min-Yen (2004). Web page classification without the web page. *Proceedings of the 13th international World Wide Web Conference on Alternate Track Papers & Posters*. ACM, 262–263.

Kececioglu, Dimitri (1991). *Reliability Engineering Handbook (vol. 1)*. Prentice-Hall, Inc.

Kelley, Henry J (1960). Gradient theory of optimal flight paths. *Ars Journal*, *30*(10), 947–954.

Kim, Yoon (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv :1408.5882*.

Kingman, JFC (1961). A convexity property of positive matrices. *The Quarterly Journal of Mathematics*, *12*(1), 283–284.

Korpusik, Mandy and Sakaki, Shigeyuki and Chen, Francine and Chen, Yan-Ying (2016). Recurrent neural networks for customer purchase prediction on twitter. *CBRecSys@ RecSys*. 47–50.

Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E (2012). Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*. 1097–1105.

Kustanowitz, Jack and Shneiderman, Ben and Kules, Bill (2006). Categorizing web search results into meaningful and stable categories using fast-feature techniques. *Digital Libraries, 2006. JCDL'06. Proceedings of the 6th ACM/IEEE-CS Joint Conference on*. IEEE, 210–219.

LeCun, Yann (1998). The mnist database of handwritten digits. *http ://yann. lecun. com/exdb/mnist/*.

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

LeCun, Yann and Bottou, Léon and Orr, Genevieve B and Müller, Klaus-Robert (1998b). Efficient backprop. *Neural networks : Tricks of the Trade*, Springer. 9–50.

LeCun, Yann and Simard, Patrice Y and Pearlmutter, Barak (1993). Automatic learning rate maximization by on-line estimation of the hessian's eigenvectors. *Advances in Neural Information Processing Systems*. 156–163.

Lindsey, James K (2000). *Applying generalized Linear Models*. Springer Science & Business Media.

Lippmann, Richard (1987). An introduction to computing with neural nets. *IEEE Assp magazine*, *4*(2), 4–22.

Liu, Qiang and Yu, Feng and Wu, Shu and Wang, Liang (2015). A convolutional click prediction model. *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 1743–1746.

Lo, Caroline and Frankowski, Dan and Leskovec, Jure (2016). Understanding behaviors that lead to purchasing : A case study of pinterest. *KDD*. 531–540.

Maas, Andrew L and Hannun, Awni Y and Ng, Andrew Y (2013). Rectifier nonlinearities improve neural network acoustic models. *Proc. ICML*. vol. 30.

McCullagh, Peter (1984). Generalized linear models. *European Journal of Operational Research*, *16*(3), 285–292.

McCullagh, Peter and Nelder, James A (1989). Generalized linear models, no. 37 in monograph on statistics and applied probability.

McCulloch, Warren S and Pitts, Walter (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133.

McMahan, H Brendan and Holt, Gary and Sculley, David and Young, Michael and Ebner, Dietmar and Grady, Julian and Nie, Lan and Phillips, Todd and Davydov, Eugene and Golovin, Daniel and others (2013). Ad click prediction : a view from the trenches. *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1222–1230.

Mikolov, Tomas and Sutskever, Ilya and Chen, Kai and Corrado, Greg S and Dean, Jeff (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*. 3111–3119.

Minsky, Marvin and Papert, Seymour A and Bottou, Léon (1969). *Perceptrons : An Introduction to Computational Geometry*. MIT press.

Mohamed, Shakir (2015). A statistical view of deep learning.

Myers, Raymond H and Montgomery, Douglas C and Vining, G Geoffrey and Robinson, Timothy J (2012). *Generalized Linear Models : with Applications in Engineering and Sciences*, vol. 791. John Wiley & Sons.

Nelder, John Ashworth and Baker, R Jacob (1972). *Generalized Linear Models*. Wiley Online Library.

Ng, Andrew (2000). Cs229 lecture notes. *CS229 Lecture Notes*, *1*(1), 1–3.

Pang, Bo and Lee, Lillian (2004). A sentimental education : Sentiment analysis using subjectivity summarization based on minimum cuts. *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 271.

Qi, Xiaoguang and Davison, Brian D (2009). Web page classification : Features and algorithms. *ACM Computing Surveys*, *41*(2), 12.

Qian, Sheng and Liu, Hua and Liu, Cheng and Wu, Si and San Wong, Hau (2018). Adaptive activation functions in convolutional neural networks. *Neurocomputing*, *272*, 204–212.

Ranganath, Rajesh and Tang, Linpeng and Charlin, Laurent and Blei, David (2015). Deep exponential families. *Artificial Intelligence and Statistics*. 762–771.

Richardson, Matthew and Dominowska, Ewa and Ragno, Robert (2007). Predicting clicks : estimating the click-through rate for new ads. *Proceedings of the 16th international conference on World Wide Web*. ACM, 521–530.

Robbins, Herbert and Monro, Sutton (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, 400–407.

Rodriguez, German (2012). Generalized linear models. *Notas [acessado em 1 maio 2010]. Disponível em http ://data. princeton. edu/wws509/notes*.

Rojas, Raul (1996). The backpropagation algorithm. *Neural Networks*, Springer. 149–182.

Rojas, Raúl (2013). *Neural networks : A systematic introduction*. Springer Science & Business Media.

Rosen-Zvi, Michal and Biehl, Michael and Kanter, Ido (1998). Learnability of periodic activation functions : General results. *Physical Review E*, *58*(3), 3606.

Rosenblatt, Frank (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.

Rosenblatt, Frank (1958). The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386.

Rosenblatt, Frank (1962). Principles of neurodynamics.

Rumelhart, David E and Hinton, Geoffrey E and Williams, Ronald J and others (1988). Learning representations by back-propagating errors. *Cognitive Modeling*, *5*(3), 1.

Sakia, RM (1992). The box-cox transformation technique : a review. *The Statistician*, 169–178.

Schmidhuber, Jürgen (2015). Deep learning in neural networks : An overview. *Neural Networks*, *61*, 85–117.

Shen, Yelong and He, Xiaodong and Gao, Jianfeng and Deng, Li and Mesnil, Grégoire (2014). Learning semantic representations using convolutional neural networks for web search. *Proceedings of the 23rd International Conference on World Wide Web*. ACM, 373–374.

Simard, Patrice and LeCun, Yann and Denker, John S (1993). Efficient pattern recognition using a new transformation distance. *Advances in Neural Information Processing Systems*. 50–58.

Sivic, Josef and Zisserman, Andrew (2009). Efficient visual search of videos cast as text retrieval. *IEEE transactions on pattern analysis and machine intelligence*, *31*(4), 591–606.

Snyman, Jan (2005). *Practical mathematical optimization : an introduction to basic optimization theory and classical and new gradient-based algorithms*, vol. 97. SpringeR.

Springenberg, Jost Tobias and Riedmiller, Martin (2013). Improving deep neural networks with probabilistic maxout units. *arXiv preprint arXiv :1312.6116*.

Srivastava, Nitish (2013). Improving neural networks with dropout. *University of Toronto-Master Thesis*, *182*.

Srivastava, Nitish and Hinton, Geoffrey E and Krizhevsky, Alex and Sutskever, Ilya and Salakhutdinov, Ruslan (2014). Dropout : a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1), 1929–1958.

Sussillo, David (2014). Random walks : Training very deep nonlin-ear feed-forward networks with smart ini. *arXiv preprint arXiv*, *1412*.

Turner, Andrew James and Miller, Julian Francis (2014). Neuroevolution : evolving heterogeneous artificial neural networks. *Evolutionary Intelligence*, *7*(3), 135–154.

Ullman, Jeffrey David (2011). *Mining of Massive Datasets*. Cambridge University Press.

Vieira, Armando (2015). Predicting online user behaviour using deep learning algorithms. *arXiv preprint arXiv :1511.06247*.

Werbos, Paul J (1982). Applications of advances in nonlinear sensitivity analysis. *System Modeling and Optimization*, Springer. 762–770.

Widrow, Bernard (1962). Generalization and information storage in network of adaline'neurons'. *Self-organizing Systems-1962*, 435–462.

Widrow, Bernard and Hoff, Marcian E (1960). Adaptive switching circuits. Rapport technique.

Witten, Ian H and Frank, Eibe and Hall, Mark A and Pal, Christopher J (2016). *Data Mining : Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.

Wu, Huaiqin (2009). Global stability analysis of a general class of discontinuous neural networks with linear growth activation functions. *Information Sciences*, *179*(19), 3432–3441.

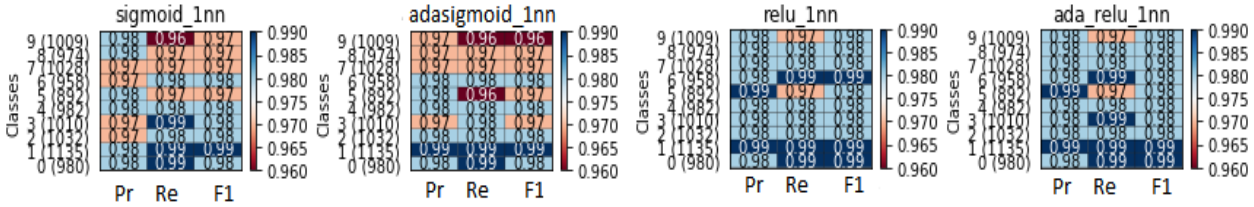
Yan, Ling and Li, Wu-jun and Xue, Gui-Rong and Han, Dingyi (2014). Coupled group lasso for web-scale ctr prediction in display advertising. *International Conference on Machine Learning*. 802–810.

Yao, Xin (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423–1447.

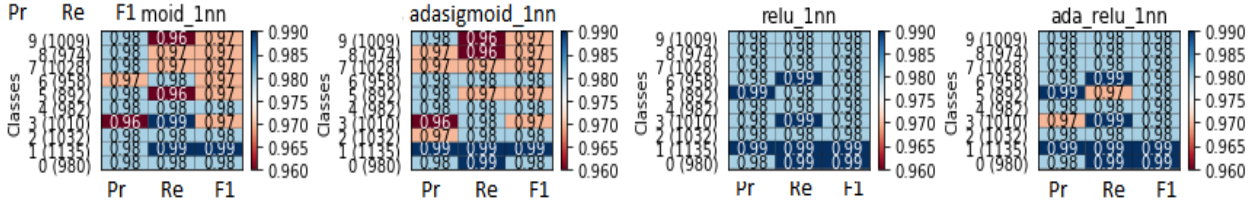
Yih, Scott Wen-tau and He, Xiaodong and Meek, Chris (2014). Semantic parsing for single-relation question answering.

Zhang, Chao and Woodland, Philip C (2015). Parameterised sigmoid and relu hidden activation functions for dnn acoustic modelling. *Sixteenth Annual Conference of the International Speech Communication Association*.

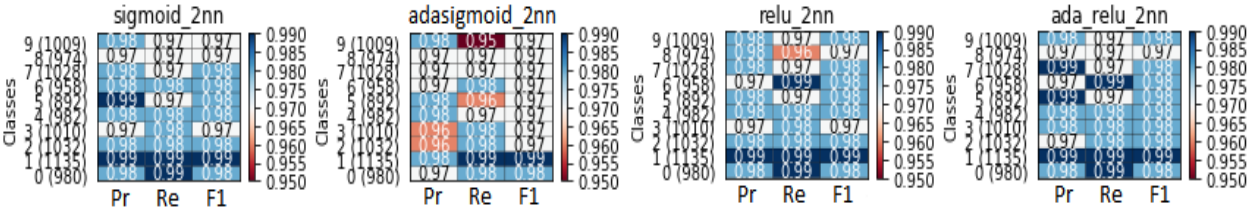
Appendix A CLASSIFICATION REPORT FOR EXPERIMENT 2 ON MNIST



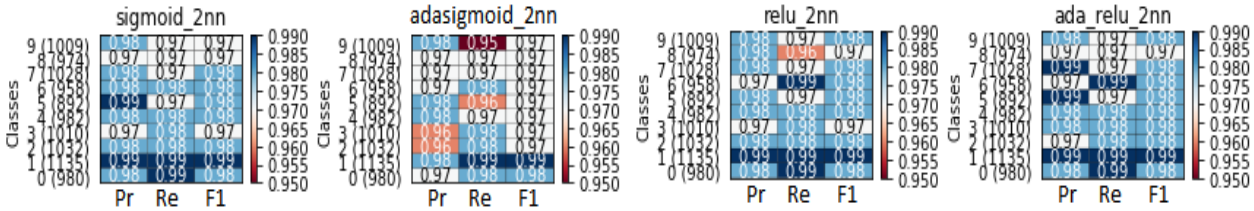
(a) 1nn_500



(b) 1nn_1000

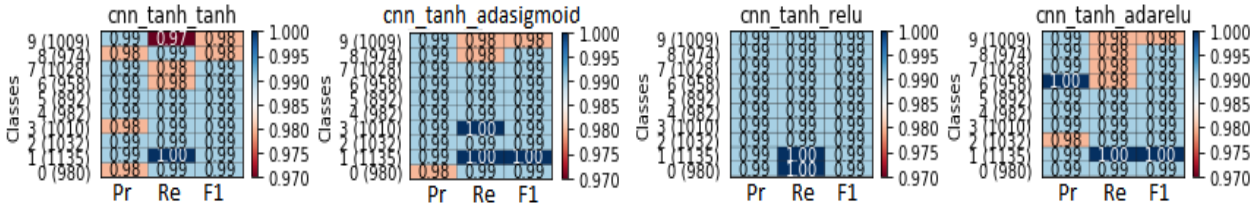


(c) 2nn_500

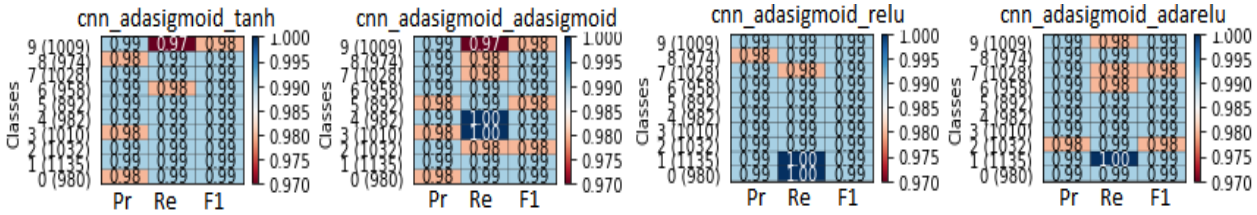


(d) 2nn_1000

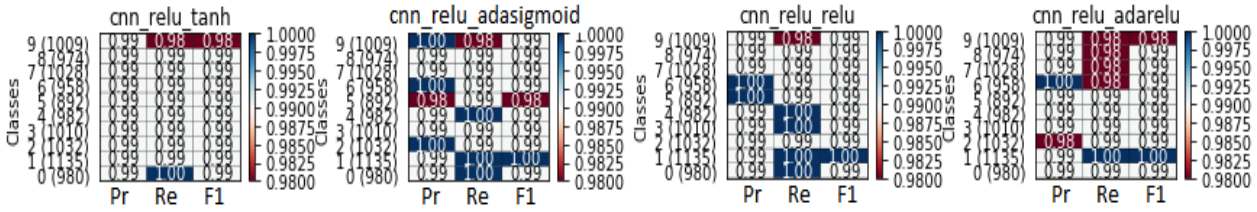
Figure A.1 MLP models with different configurations : compare the precision, recall and F1.



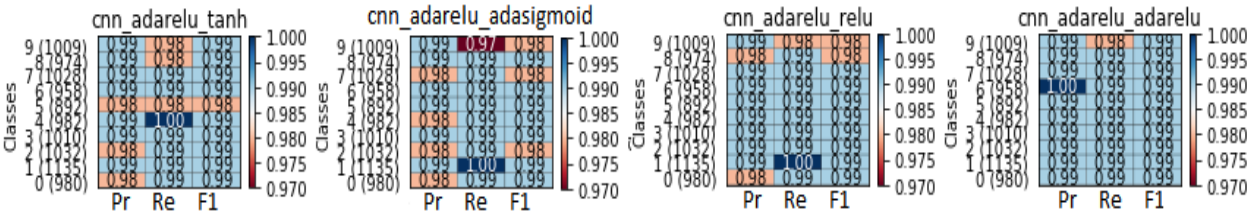
(a) cnn_tanh_bn



(b) cnn_adasigmoid_bn



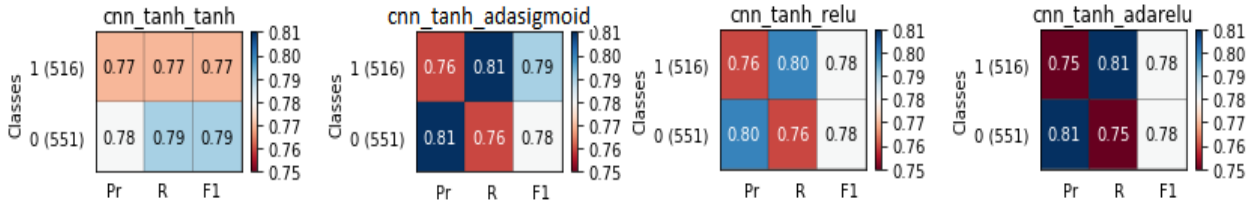
(c) cnn_relu_bn



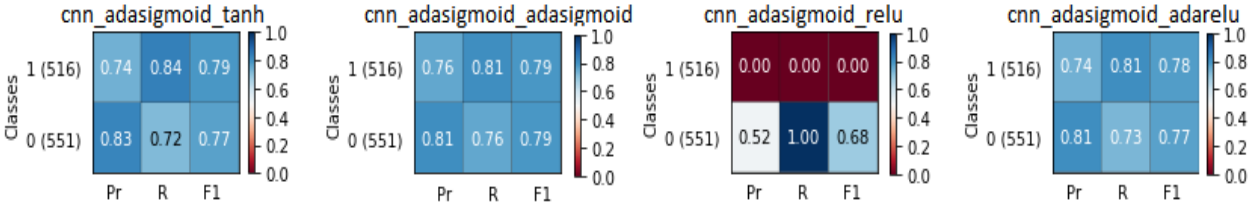
(d) cnn_adarelu_bn

Figure A.2 CNN models : Compare the precision, recall and F1.

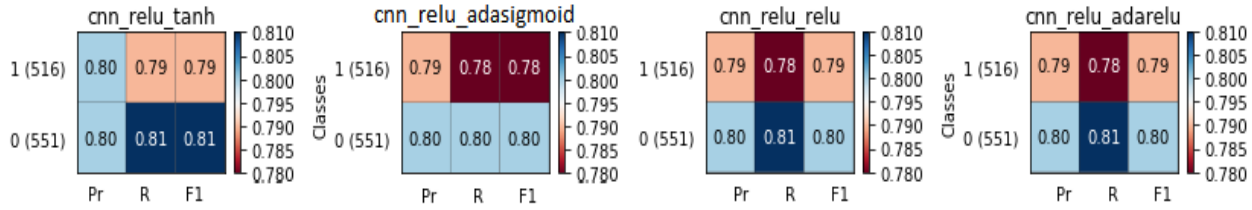
Appendix B CLASSIFICATION REPORT FOR EXPERIMENT 3 ON MOVIE REVIEW



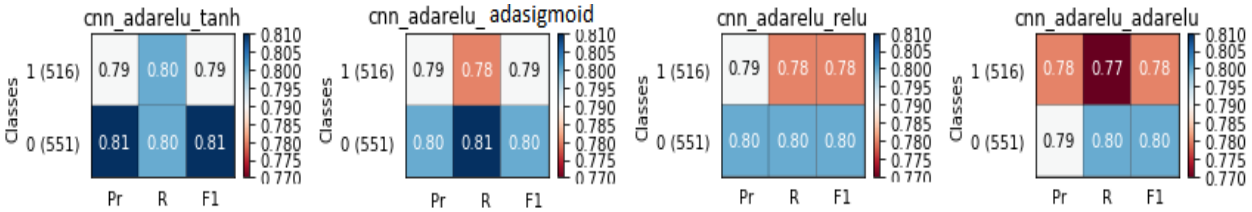
(a) cnn_tanh



(b) cnn_adasigmoid



(c) cnn_relu



(d) cnn_adarelu

Figure B.1 Compare the precision, recall and F1.