Evaluation of the Smoothing Activation Function in Neural Networks for Business Applications

by

Jun Siong Ang

B. Eng. (Hons) Mechanical Engineering
National University of Singapore, 2009

Submitted to the MIT System and Design Management Program
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management
at the
Massachusetts Institute of Technology

June 2019

© 2019 Jun Siong Ang. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature redacted

Signature of Author____

Jun Siong Ang
MIT System and Design Management Program
May 23, 2019

Signature redacted

Certified by____

Robert Freund
Theresa Seley Professor in Management Science at MIT Sloan School of Management
Thesis Supervisor

Signature redacted

Accepted by____

Joan Rubin
Executive Director, System Design and Management Program

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 27 2019
LIBRARIES
ARCHIVES

This page is intentionally left blank

Evaluation of the Smoothing Activation Function in Neural Networks for Business Applications
by

Jun Siong Ang

Submitted to the MIT System and Design Management (SDM) program on May 23, 2019 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Engineering and Management.

Abstract

With vast improvements in computational power, increased accessibility to big data, and rapid innovations in computing algorithms, the use of neural networks for both engineering and business purposes was met with a renewed interest beginning in early 2000s. Amidst substantial development, the Softplus and Rectified Linear Unit (ReLU) activation functions were introduced in 2000 and 2001 respectively, with the latter emerging as the more popular choice of activation function in neural networks. Notably, the ReLU activation function maintains a high degree of gradient propagation while presenting greater model sparsity and computational efficiency over Softplus. As an alternative to the ReLU, a family of a modified Softplus activation function – the "Smoothing" activation function of the form g(z) = $\mu \log(1 + e^{z/\mu})$ has been proposed. Theoretically, the Smoothing activation function will leverage the high degree of gradient propagation and model simplicity characteristic of the ReLU function, while eliminating possible issues associated with the non-differentiability of ReLU about the origin. In this research, the performance of the Smoothing family of activation functions vis-à-vis the ReLU activation function will be examined.

Thesis Supervisor: Robert Freund
Title: Theresa Seley Professor in Management Science at MIT Sloan School of Management

This page is intentionally left blank

Acknowledgements

I owe a great deal of gratitude to many who have encouraged, supported and inspired my academic pursuit and research during my time at MIT.

The most important among them is my devoted and adoring wife Clara, without whom I am unquestionably a lesser man. The guilt of subjecting her to loneliness for the larger part of the past 2 years (as I maneuvered past the intensive workload in MIT) weighs heavy on me. I am perpetually indebted to her.

I also remember my first 2 children – Mitkia and Mitten. Conceived in MIT, they parted from this physical world before I had any opportunity to love them. They shall never know of the joy and hope they have given me.

I will also like to thank Mr. Matthew Soh, my Junior College teacher who sponsored my studies during my impoverished high school years, and Mr. Sean Tan, my ex-superior who so fervently believed in my capabilities when I was a junior officer in the Singapore Army. I am truly blessed.

I am forever grateful to my thesis supervisor Professor Robert Freund, whose generosity and patience in mentoring this amateur had inspired my resolve within the field of business analytics and machine learning. The sheer investment of his time into my research in spite of his hectic schedule speaks of his passion, dedication, and conviction as an academic. I am privileged to receive his tutelage.

I would like to thank Joan Rubin, Executive Director, and Bryan Moser, Academic Director, as well as faculty members and staffs of the MIT System Design and Management Program for putting together this exceptional and world-class program. I would also like to thank my organization – the Singapore Armed Forces for sponsoring my pursuit in MIT.

Last but not least, I thank my dear friends Samuel Lee and Raymond Tan. I have learnt a great deal of lessons from these guys – in both academics and life. Surely I will miss the late nights we spent at Muddy Charles, as well as the even-later nights polishing our assignments in the student lounge. I am truly honored to know you.

This page is intentionally left blank

Table of Contents

Abstract **3**
Acknowledgements **5**
Table of Contents **7**
List of Figures **10**
List of Tables **12**
Chapter 1: Introduction **16**
1.1 Neural Networks Overview 16
1.2 Feedforward Neural Networks, Cost Gradient Descent Algorithm 19
1.2.1 Network Models and Forward Propagation 19
1.2.2 Cost and Loss 21
1.2.3 Cost Gradient Descent Algorithm 22
1.2.4 Basic Theory on Deriving Cost Gradients using Back Propagation 25
1.3 Softplus, ReLU and the Smoothing Activation Functions 29
1.3.1 The Softplus Activation Function, $gz = ln(1 + ez)$ 29
1.3.2 The ReLU Activation Function, $gz = max\{0, z\}$ 30
1.3.3 The Smoothing Activation Function, $gz = \mu\, ln(1 + ez\mu)$ 31
Chapter 2: Feedforward Neural Network Model Architecture **33**
2.1 Architecture Overview 33
2.2 Matrix Implementation 34
2.3 Additional Augmented Features 37
2.3.1 Mini-Batch Training with Replacement 37
2.3.2 Optimized Learning Rates (Fixed and Variables) 37
2.3.3 Cost Calculation with Average of Parameters over Past 100 Iterations 38
2.3.4 Data Standardization 38
Chapter 3: Scope of Experiment **39**
3.1 Overview of Datasets 39
3.1.1 Framingham Heart Study Dataset 39
3.1.2 Bank Churn Modelling Dataset 39
3.1.3 German Credit Risk Dataset 39
3.2 Experimental Design 40
Chapter 4: Analysis of Experimental Results **43**

4.1 Performance of Smoothing Activation Functions 43
4.1.1 Analysis of Results for Shallow (2-Layer) Neural Networks 43
4.1.2 Analysis of Results for Deep (3-Layer) Neural Networks 54
4.2 Computational Time Across Activation Functions 66
Chapter 5: Assessing the Suitability of Neural Networks in Business Applications 68
5.1 Benchmarking Logistic Regression for Business Applications 68
5.2 Model Performance: Logistic Regression versus Neural Network 71
5.3 Considerations Beyond AUC and Cost Metrics 73
Chapter 6: Summary, Way Ahead, and Conclusion 75
6.1 Key Experimental Findings 75
6.2 Areas for Further Study 75
6.2.1 Activation Function Selection Through Topological Examination 75
6.2.2 Averaging Parameters over Iterations for Optimization 76
Appendix A: Code for Feedforward Neural Network A-1
Appendix B: Code for Logistic Regression Network B-1
References R-1

This page is intentionally left blank

List of Figures

Figure 1-1: Neural Networks as a Machine Learning Technique. 16
Figure 1-2: Neural Networks as Artificial Intelligence Method [2]. 17
Figure 1-3: Representation Learning from Neural Networks - Image Recognition [3]. 17
Figure 1-4: Learning with Linear Regression. 18
Figure 1-5: Learning With Simple Neural Net. 18
Figure 1-6: Example 3-Layer Feedforward Neural Network. 19
Figure 1-7: Operations Within a Node. 20
Figure 1-8: Overview of Lost and Cost Functions. 21
Figure 1-9: Cost Gradient Descent Overview. 22
Figure 1-10: Cost Profiles with Multiple Minimums. 23
Figure 1-11: Bivariate (3-D) Cost Profile Representation. 24
Figure 1-12: The Sigmoid Activation Function. 29
Figure 1-13: The Softplus Activation Function. 29
Figure 1-14: The Tanh Activation Function. 29
Figure 1-15: The ReLU Activation Function. 31
Figure 1-16: The Leaky ReLU Activation Function. 31
Figure 1-17: The Smoothing Activation Function (With ReLU for Comparison). 32
Figure 2-18: Architecture for Feedforward Neural Networks (Binary Classification). 33
Figure 2-19: Representation of Observation Data in Matrix Form. 34
Figure 2-20: Linear Multiplication Operation in Matrix Operation for 1st Hidden Layer. 34
Figure 2-21:Generalized Linear Multiplication in Matrix Operations. 35
Figure 2-22: Backward Propagation in Matrix Representation. 35

This page is intentionally left blank

List of Tables

Table 1: Learning Rates for Different Learning Model Parameters. 37
Table 2: Number of Hidden Layers Used for Each Dataset..40
Table 3: Permutation of Hyperparameters for Neural Networks. 41
Table 4: Shallow Networks - Minimum Cost Achieved on Training Set (Batch Learning, Fixed α Models). 44
Table 5: Cost Profiles (Framingham Heart Study Dataset, Batch Learning, Fixed α). 44
Table 6: Shallow Networks - Minimum Cost Achieved on Training Set (Batch Learning, Variable α Models). 46
Table 7: Cost Profiles (German Credit Risk Dataset, Batch Learning, Variable α). 47
Table 8: Shallow Networks - Minimum Cost Achieved on Training Set (Mini-Batch Learning, Fixed α Models). ...48
Table 9: Cost Profiles (Bank Churn Modelling Dataset, Mini-Batch Learning, Fixed α). 49
Table 10: Cost Profiles (German Credit Risk Dataset, Mini-Batch Learning, Fixed α). 50
Table 11. Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Fixed α)....................................51
Table 12: Shallow Networks - Minimum Cost Achieved on Training Set (Mini-Batch Learning, Variable α). 52
Table 13: Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Variable α). 53
Table 14: Deep Networks - Minimum Cost Achieved on Training Set (Batch Learning, Fixed α Models)...............54
Table 15: Cost Profiles (Bank Churn Modelling Dataset, Batch Learning, Fixed α) – (A). 55
Table 16: Cost Profiles (Bank Churn Modelling Dataset, Batch Learning, Fixed α) – (B). 56
Table 17: Deep Networks - Minimum Cost Achieved on Training Set (Batch Learning, Variable α Models). 57
Table 18: Cost Profiles (Framingham Heart Study Dataset, Batch Learning, Variable α). 58
Table 19: Cost Profiles (Bank Churn Modelling Dataset, Batch Learning, Variable α)..59
Table 20: Deep Networks - Minimum Cost Achieved on Training Set (Mini-Batch Learning, Fixed α Models).......60
Table 21: Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Fixed α), c=400. 61
Table 22: Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Fixed α) , c=600. 62
Table 23: Deep Networks – Minimum Cost Achieved on Training Set (Mini-Batch Learning, Variable α Models). 63
Table 24: Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Variable α). 64
Table 25: Cost Profiles (Bank Churn Modelling Dataset, Mini-Batch Learning, Variable α)....................................65
Table 26: Average Computational Time for 2-Layer, 3-Layer Neural Networks...66
Table 27: Average Computational Time for 1-Layer, 2-Layer Neural Networks...69
Table 28: Minimum Cost Performance for Logistic Regression and Neural Network Models. 71
Table 29: First Example of Averaged Parameters Outperformed Actual Parameters for Iteration. 76
Table 30: Second Example of Averaged Parameters Outperformed Actual Parameters for Iteration. 77

This page is intentionally left blank

Chapter 1: Introduction

This thesis seeks to evaluate the performance of the Smoothing family of activation functions vis-à-vis the ReLU activation function for neural networks. This chapter will first provide a brief overview of the history of neural networks hitherto while highlighting the importance of neural networks as a machine learning technique. Thereafter, the chapter delves into the mathematical theory between behind a neural net employing the gradient descent algorithm, upon which the role of the activation function will be expounded upon. The chapter closes with an introduction of the ReLU activation function and the Softplus / Smoothing activation function.

1.1 Neural Networks Overview

A neural network is a machine learning technique capable of handling both supervised (with continuous or categorical responses) and unsupervised learning tasks (See **Figure 1-1**).

Figure 1-1: Neural Networks as a Machine Learning Technique.

Supervised Machine Learning

Continuous Response

Linear Regression
Regression Trees
K – Nearest Neighbors
Neural Nets

Categorical Response

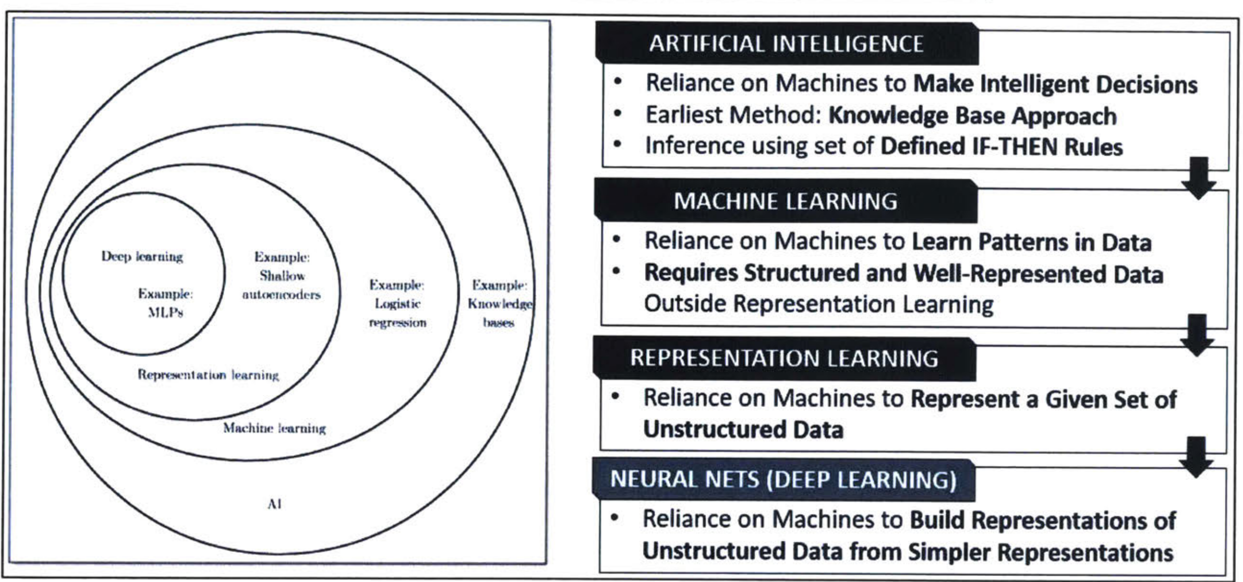
Logistics Regression
Classification Trees
K – Nearest Neighbors
Neural Nets

Unsupervised Machine Learning

K-Means Clustering
Hierarchical Clustering
Neural Nets

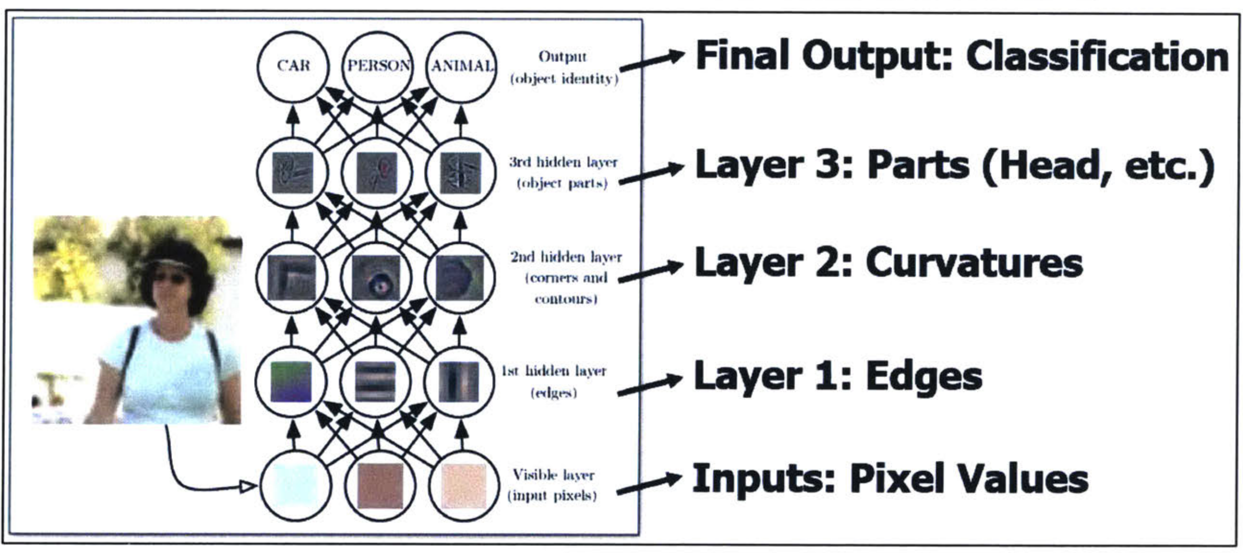
Unlike conventional machine learning techniques, however, neural networks are capable of building representations of unstructured data from simpler representations (See **Figure 1-2**) [1]. This empowers neural networks to recognize complex, nonlinear patterns from unstructured data (such as images, sounds, and text). Indeed, variants of neural networks algorithms have found widespread adoption in image recognition, voice-to-text translation and textual recognition purposes, across applications such as autonomous driving, voice-activation, machine translation and gaming behavior within the engineering, healthcare and finance industries.

Figure 1-2: Neural Networks as Artificial Intelligence Method [2].



An indicative example of how neural networks developed progressively complex representations from simpler representations within unstructured datasets can be conveyed in the case of image recognition. Consider the case of an image recognition neural network with three hidden layers (See **Figure 1-3**):

Figure 1-3: Representation Learning from Neural Networks - Image Recognition [3].



- From the visible / input layer, the first hidden layer of the neural network extracts abstract features such as edges and corners.

- Feeding the abstract features (i.e. edges and corners) into the next hidden layer, the second hidden layers captures the more complex features of corners and contours.

- The corners and contours are in turn fed into the third hidden layer, where object parts are captured by the neural net model. The data collected by the third layer is then fed into the final layer where the image gets classified into one of several possible classes.

Beyond engineering applications tackling unstructured data (such as image recognition as shown), the concept of deriving complex representations from simpler structured data is equally applicable within business domains. A simple example cited by Andrew Ng [4] discusses how property prices might be more accurately predicted using a neural network as compared to linear regression. Critically, the neural network captures more complex, non-linear interactions between the original representation of the dataset within its hidden layers, thereby giving a more robust prediction of the property price (See **Figure 1-4 and Figure 1-5**).

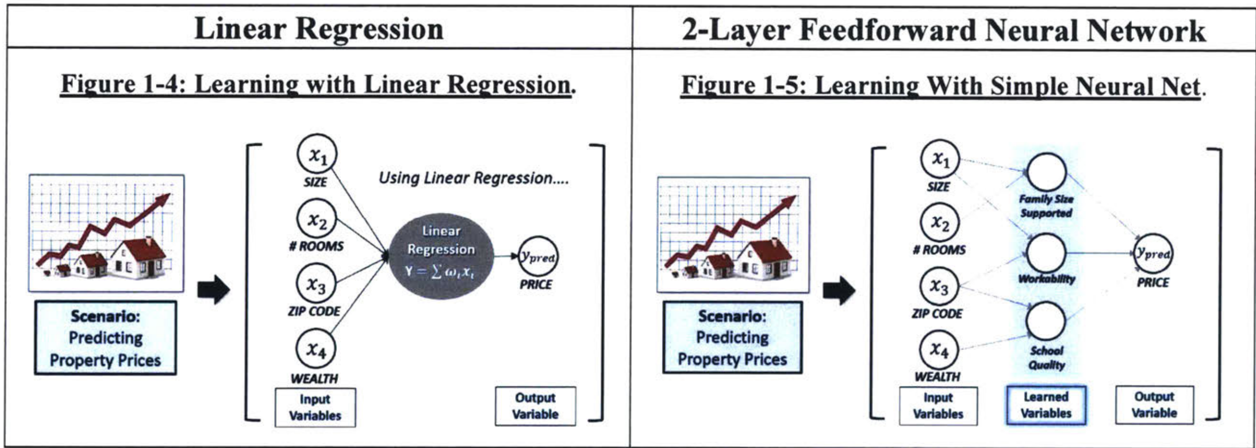


Figure 1-4: Learning with Linear Regression.

Figure 1-5: Learning With Simple Neural Net.

It is interesting to note that the earliest neural networks – or "cybernetics", originated as early as the 1940s with the intention to mimic biological intelligence. This first wave of interest in neural networks tapered off in the 1960s due to the limited capabilities of available algorithms. Enthusiasm in employing neural networks to understand human cognition was revived in the 1980s before unrealistic expectations about the capabilities of the technique caused its second decline.

In the early 2000s, with vast improvements in computational power, increased accessibility to big data, and rapid innovations in computing algorithms, the third era of neural networks – commonly termed the "Deep Learning" era, saw the adaptation of neural networks for both engineering and

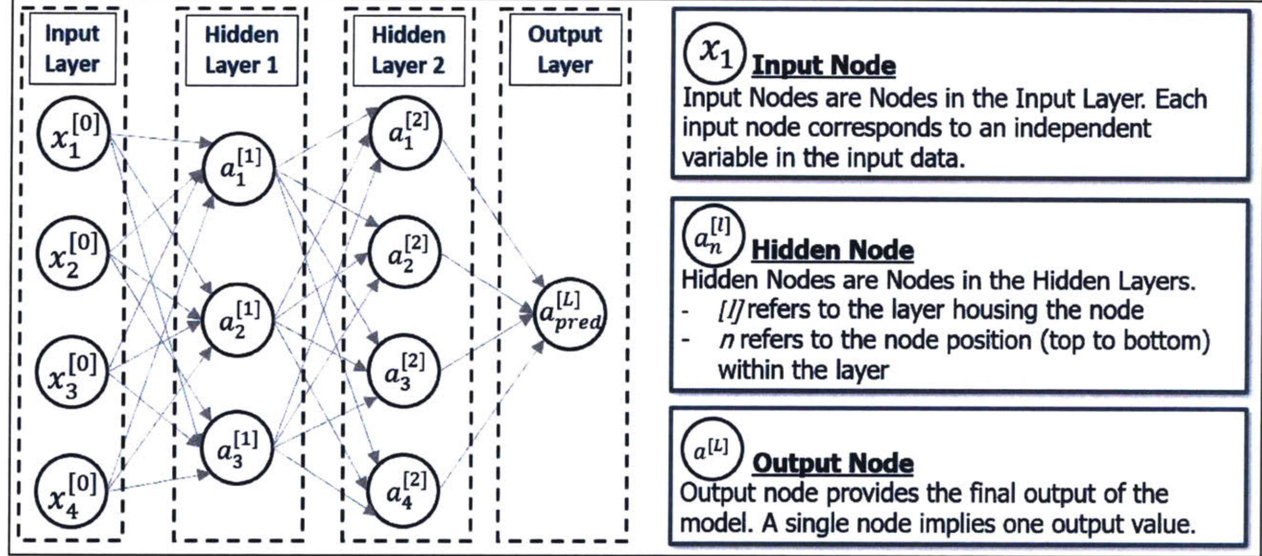
business purposes. Indeed, while neural networks are likewise capable of managing structured data, neural network proved to be particularly robust in handling unstructured data. Modern deep learning research is strongly motivated by the development of machines that can improve on human judgment in processing both structured and unstructured data by being faster, more consistent, more accurate, and less biased.

1.2 Feedforward Neural Networks, Cost Gradient Descent Algorithm

1.2.1 Network Models and Forward Propagation

For the purpose of studying neural networks for business applications[1], this study will focus on feedforward neural networks. A feedforward neural network is effectively presented using a network diagram. **Figure 1-6** presents the network of a 3-layer (conventionally, the input layer is not counted) feedforward neural network and the common nomenclature used to describe it:

Figure 1-6: Example 3-Layer Feedforward Neural Network.



- The Zeroth Layer is known as the Input Layer. The number of nodes corresponds to the number of independent variables in our dataset. In this case, the 4 nodes imply that the

[1] Variants of deep learning networks, such as convolutional neural networks or recurrent neural networks, would be more appropriate for image recognition or machine translation / speech recognition tasks, respectively. For this study, business applications entail prediction / classification tasks using a structured dataset, such as predicting property prices or estimating propensity of credit default.

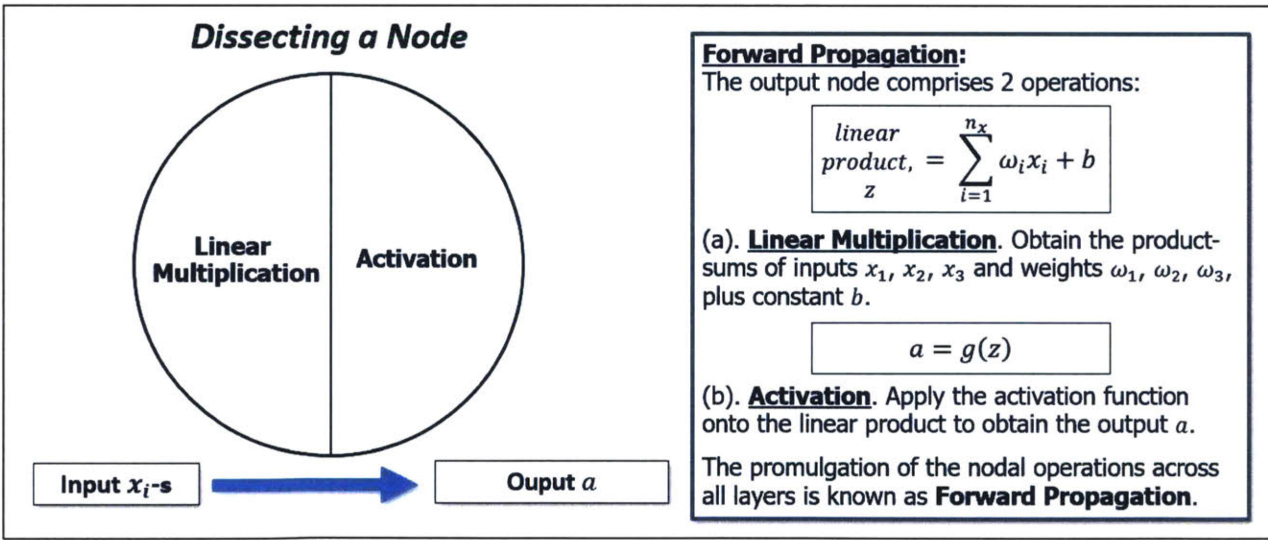
dataset contains 4 independent variables. Note that we do not count the input layer when describing the number of layers in a neural network.

- The First and Second layers are known as the Hidden Layers. The number of hidden layers, as well as the number of nodes within each hidden layer, are hyperparameters set prior to running of the model. A neural network model with 2 or more hidden layers is considered a "deep learning network".

- The Third Layer is the Output Layer. In the case of a binary classification problem (which shall be the case going forward in this example), the single node in the output layer highlights that we have a single output value (i.e., 1 or 0 for a binary classification).

Each node in the hidden and output layers holds a set of parameters (a vector **ω**) and a bias term (a constant **b**), as well as an activation function. Each node performs two operations (See **Figure 1-7**):

Figure 1-7: Operations Within a Node.



- Linear Multiplication and Addition of a Bias Term. The product sum between the vector **ω** for the node and the vector of inputs from the previous layer is first obtained (*For the first layer, the "vector of inputs from the previous layer" refers to the vector of the observation **x** from the input layer; For the second layer, this refers to the output of the nodes from the previous layer.)*. Thereafter, the product sum is added to the bias term ***b*** of

the node. We label the output from the linear multiplication and addition of the bias term as **z**.

- Application of the Activation Function onto the Product **z**. The product is then put through an activation function, from which we obtain the output for the node **a**, i.e. **a = g(z)**, where g(z) is the activation function.

A forward propagation entails passing the initial observation (the vector **x)** through nodes of every layer in the neural network model to obtain a final output from the neural network model.

1.2.2 Cost and Loss

Cost and Loss. Critically, training a neural network involves attaining the best values for the parameters **ω, b** for each of the nodes in the model. This way, the model produces predictions / classifications that are closest to the true label of a given observation. The metric used in determining the "goodness" of the parameters is **Loss** (denoted by ***L***; used to consider the goodness of parameters for 1 observation) **or Cost** (denoted by ***J***; the average of costs over a set of observations, used to consider the goodness of parameters over a set of observations) (See **Figure 1-8**). For this study in particular, we will use the **cross-entropy logistic loss** for our cost calculations.

Figure 1-8: Overview of Lost and Cost Functions.

The calculation of cross-entropy logistic cost is given as:

$$Loss,\ L(y, a^{[L]}) = -[y.\log(a^{[L]}) + (1-y).\log(1-a^{[L]})]$$

where y is the true label (1 or 0 for a binary classification task) and $\mathbf{a}^{[L]}$ is the output value of the model for a given observation.

Over **m** observations, the calculation for **Cross-Entropy Logistic Loss, J** is given:

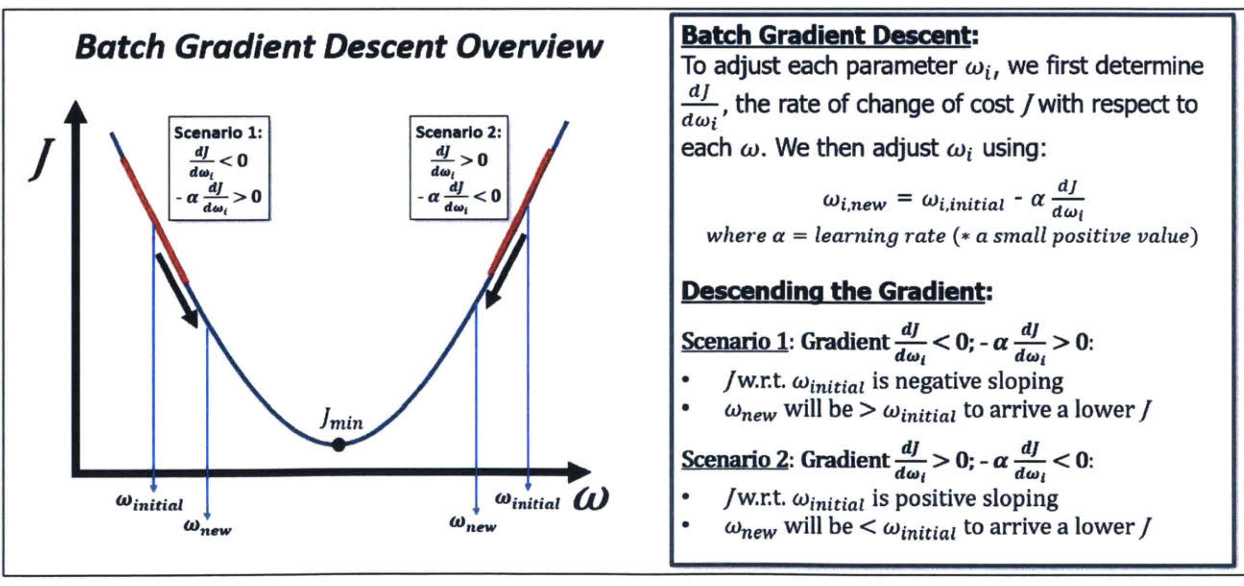
$$Cost,\ J(y, a^{[L]}) = -\frac{1}{m}\sum[y.\log(a^{[L]}) + (1-y).\log(1-a^{[L]})]$$

To summarize, the cost function penalizes a wrong prediction by the model (i.e. cost value is higher when $\boldsymbol{a}^{[L]}$ is further away while actual label **y**). In achieving a model that can best predict the true label, training a model will therefore involve obtaining the set of parameters **ω, b** for each node in the model which will minimize the cost incurred across observations within the training set.

1.2.3 Cost Gradient Descent Algorithm

The cost gradient descent algorithm is used to optimize the parameters **ω, b** for each node in the model. The concept of cost gradient descent works by considering the tangential gradient, $\frac{dJ}{d\omega}$ of the cost profile with respect to a parameter $\boldsymbol{\omega_i}$ ***or b***. **Figure 1-9** considers a univariate case: If the current value of $\boldsymbol{\omega_i}$ corresponds to a point where the cost gradient is negative, a step is taken in the positive direction. By increasing the size of $\boldsymbol{\omega_i}$, we arrive at a lower cost; Conversely, if $\boldsymbol{\omega_i}$ corresponds to a point where the cost gradient is positive, we decrease $\boldsymbol{\omega_i}$ arrive at a lower cost.

Figure 1-9: Cost Gradient Descent Overview.



The size of the steps taken when optimizing the parameters is dependent on (1) the size of the cost gradient: the larger the cost gradient, the larger the step size, and (2) the learning rate (denoted by $\boldsymbol{\alpha}$): a hyperparameter set prior to running the model. In implementing a neural network, each parameter is continuously adjusted via iterations of the cost gradient descent method using the equation below:

$$\omega_{i,new} = \omega_{i,old} - \alpha . \frac{dJ}{d\omega_i}$$

In practice, the cost profile is more complicated than depicted in **Figure 1-9**: (1) The cost profile might contain local minimums which do not reflect the true global minimum value of the cost profile, see **Figure 1-10**. (2) Moreover, over multiple independent variables, the cost profile is often a function of more than 1 or 2 parameters (**Figure 1-11**).

Generally, an optimization process can be considered robust if the local minimum of the cost profile is close to global minimum point. However, in the event that the local minimum is much higher than the global minimum point, the optimization will require further enhancements.

Figure 1-10: Cost Profiles with Multiple Minimums.

Example A: Local Minimum ≈ Global Minimum

J, J_{min}, *global*, J_{min}, *local*, ω

Example B: Local Minimum ≠ Global Minimum

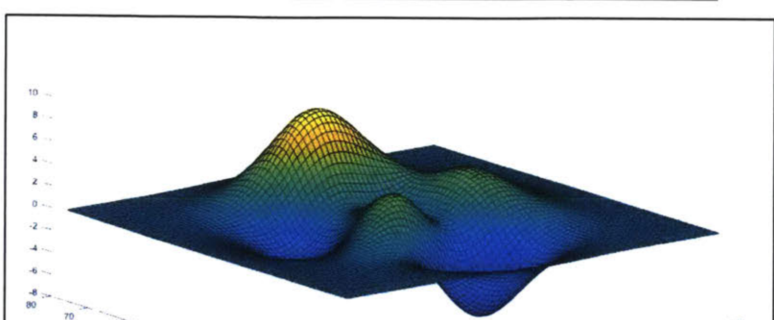
J, J_{min}, *local*, J_{min}, *global*, ω

(a) Loss Function with Multiple Minima

In practice, the loss function might have many local minima. A gradient descent process might lead us into a local minimum instead of a global minimum.

- **(Example A)** If the local minimum is close to global minimum, the optimization can still be considered robust.
- **(Example B)** If the local minimum is not close to global minimum, we require further enhancement to the optimization process.

Figure 1-11: Bivariate (3-D) Cost Profile Representation.



A specific technique used to enhance the location of the global minimum for large datasets involves utilizing a stochastic gradient descent (or mini-batch training) approach, whereby only a number of observations randomly selected from the training set are used for each iteration of the cost gradient descent. (In contrast, a cost gradient descent using all examples in the training set for each iteration is known as batch training.) Empirically, mini-batch training allows the parameters to "escape" local minimums more quickly, thus enabling the neural network to arrive at the global minima of the cost profile more effectively.

1.2.4 Basic Theory on Deriving Cost Gradients using Back Propagation

Referencing the example in **Figure 1-6**: The calculation of the cost gradients $\frac{dJ}{d\omega}$ involves using the chain rule. In general, for any layer *l*, the cost gradient can be expressed as follows:

$$\frac{dJ}{d\omega^{[l]}} = \frac{dJ}{da^{[l]}} \cdot \frac{da^{[l]}}{dz^{[l]}} \cdot \frac{dz^{[l]}}{d\omega^{[l]}}$$

$$\frac{dJ}{db^{[l]}} = \frac{dJ}{da^{[l]}} \cdot \frac{da^{[l]}}{dz^{[l]}} \cdot \frac{dz^{[l]}}{db^{[l]}}$$

Particularly, for parameters in the third (output) layer (i.e. $l = L$ *or 3*),

$$\frac{dJ}{d\omega^{[L]}} = \frac{dJ}{da^{[L]}} \cdot \frac{da^{[L]}}{dz^{[L]}} \cdot \frac{dz^{[L]}}{d\omega^{[L]}}$$

$$\frac{dJ}{db^{[L]}} = \frac{dJ}{da^{[L]}} \cdot \frac{da^{[L]}}{dz^{[L]}} \cdot \frac{dz^{[L]}}{db^{[L]}}$$

- Given Cross-entropy Logistic Cost, **J** $= -\frac{1}{m}\sum[y.\log(a^{[L]}) + (1-y).\log(1-a^{[L]})]$

$$\frac{dJ}{da^{[L]}} = -\frac{1}{m}\sum\left(\frac{y}{a^{[L]}} - \frac{1-y}{1-a^{[L]}}\right)$$

$$= -\frac{1}{m}\sum\left(\frac{y(1-a^{[L]})-(1-y)(a^{[L]})}{a^{[L]}(1-a^{[L]})}\right)$$

$$= -\frac{1}{m}\sum\left(\frac{y-ya^{[L]}-a^{[L]}+ya^{[L]}}{a^{[L]}(1-a^{[L]})}\right)$$

$$= -\frac{1}{m}\sum\left(\frac{y-a^{[L]}}{a^{[L]}(1-a^{[L]})}\right)$$

- The activated value is given as **a** = g(z), whereby g(z) is the activation function for the output layer node. In the case of a binary classification problem, we use the Sigmoidal activation function, g(z) = $\frac{1}{1+e^{-z}}$ for the output layer node. Therefore,

$$\frac{da^{[L]}}{dz^{[L]}} = (1+e^{-z})^{-2}.(-1).(e^{-z}).(-1)$$

$$= \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}}$$

$$= \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right) \cdot a^{[L]}$$

$$= \left(1 - a^{[L]}\right) . a^{[L]}$$

- The node output $z^{[L]}$ is given as $z^{[L]} = \omega^{[L]T} a^{[2]} + b^{[L]}$. Of note, $\omega^{[L]T}$ refers to the transpose of the vector of parameters for the output layer node. The matrix multiplication $\omega^{[L]T} a^{[2]}$ therefore represents the product sum between the elements of the vector $\omega^{[L]}$and $a^{[2]}$. Concurrently, $a^{[L-1]}$ *or* $a^{[2]}$ refers to the activated value from the previous layer while $b^{[L]}$ refers to the bias term for the output layer node. Given these, we obtain

$$\frac{dz^{[L]}}{d\omega^{[L]}} = a^{[2]}$$

$$\frac{dz^{[L]}}{db^{[L]}} = 1$$

Piecing everything together, we obtain:

$$\frac{dJ}{d\omega^{[L]}} = -\frac{1}{m}\sum\left(\frac{y - a^{[L]}}{a^{[L]}(1 - a^{[L]})}\right) . \left(1 - a^{[L]}\right) . a^{[L]} . a^{[2]}$$

$$= -\frac{1}{m}\sum[(y - a^{[L]}) . a^{[2]}] \text{ , and}$$

$$\frac{dJ}{db^{[L]}} = -\frac{1}{m}\sum\left(\frac{y - a^{[L]}}{a^{[L]}(1 - a^{[L]})}\right) . \left(1 - a^{[L]}\right) . a^{[L]}$$

$$= -\frac{1}{m}\sum[(y - a^{[L]})]$$

For parameters in the second (hidden) layer (i.e. $l = 2$), we build on the chain rule to calculate the cost gradients:

$$\frac{dJ}{d\omega^{[2]}} = \frac{dJ}{da^{[2]}} \cdot \frac{da^{[2]}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{d\omega^{[2]}}$$

$$\frac{dJ}{db^{[2]}} = \frac{dJ}{da^{[2]}} \cdot \frac{da^{[2]}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{db^{[2]}}$$

- The component $\frac{dJ}{da^{[2]}}$ can be expressed as $\frac{dJ}{da^{[2]}} = \frac{dJ}{da^{[L]}} \cdot \frac{da^{[L]}}{dz^{[L]}} \cdot \frac{dz^{[L]}}{da^{[2]}}$, whereby $\frac{dJ}{da^{[L]}} \cdot \frac{da^{[L]}}{dz^{[L]}}$ have already been derived previously. For $\frac{dz^{[L]}}{da^{[2]}}$, we get $\frac{dz^{[L]}}{da^{[2]}} = \boldsymbol{\omega}^{[L]}$.

- As with the node in the output layer, we consider the activation function used in the second layer to calculate $\frac{da^{[2]}}{dz^{[2]}}$. In this study, we consider both the ReLU and the Smoothing activation functions:

(a) For ReLU, a = g(z) = max{0, z}= $\begin{cases} z, if\ z > 0 \\ 0, if\ z \leq 0 \end{cases}$

$$\frac{da^{[2]}}{dz^{[2]}} = \begin{cases} 1, if\ z > 0 \\ 0, if\ z \leq 0 \end{cases}$$

(b) For Smoothing, a = g(z) = $\boldsymbol{\mu} \log\left(1 + e^{\frac{z}{\mu}}\right)$

$$\frac{da^{[2]}}{dz^{[2]}} = \frac{\mu}{1+e^{\frac{z}{\mu}}}\left(e^{\frac{z}{\mu}}\right)\left(\frac{1}{\mu}\right)$$

$$= \frac{e^{\frac{z}{\mu}}}{1+ e^{\frac{z}{\mu}}}$$

$$= \frac{1}{1+ e^{-\frac{z}{\mu}}}$$

- Finally, $\boldsymbol{a}^{[2]} = \boldsymbol{\omega}^{[2]T}\boldsymbol{a}^{[1]} + \boldsymbol{b}^{[2]}$

$$\frac{dz^{[2]}}{d\omega^{[2]}} = \boldsymbol{a}^{[1]}$$

$$\frac{dz^{[2]}}{db^{[2]}} = 1$$

Likewise, we piece the components together to derive the cost gradients. The cost gradients obtained will then be used to derive the step adjustments for the parameters of nodes within the second hidden layer. Using this approach, we can further build on the chain rule to derive the cost gradients for parameters in the first hidden layer using:

$$\frac{dJ}{d\omega^{[1]}} = \frac{dJ}{da^{[1]}} \cdot \frac{da^{[1]}}{dz^{[1]}} \cdot \frac{dz^{[1]}}{d\omega^{[1]}}$$

$$\frac{dJ}{db^{[1]}} = \frac{dJ}{da^{[1]}} \cdot \frac{da^{[1]}}{dz^{[1]}} \cdot \frac{dz^{[1]}}{db^{[1]}}$$

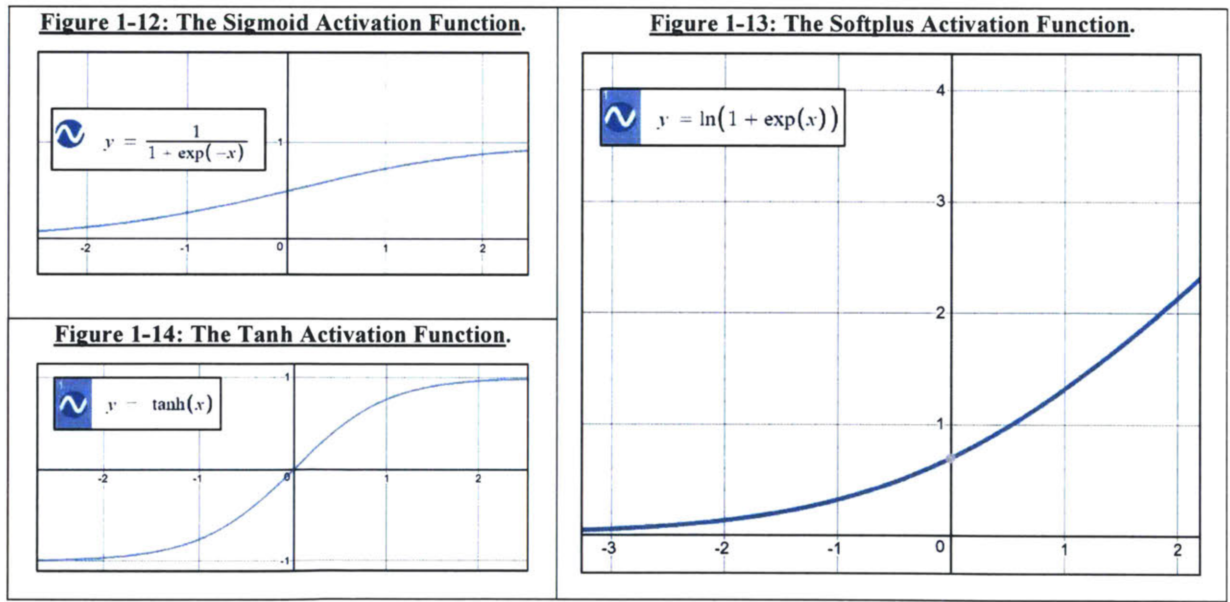
As seen, the derivation of the cost gradients for the second layer $\frac{dJ}{da^{[2]}}$ involves the computation of $\frac{dJ}{da^{[L]}} \cdot \frac{da^{[L]}}{dz^{[L]}}$ from the output layer, and the derivation of the cost gradients for the first layer $\frac{dJ}{da^{[1]}}$ require the computation of $\frac{dJ}{da^{[2]}} \cdot \frac{da^{[2]}}{dz^{[2]}}$ from the second layer. In a model with more hidden layers, the process can be repeated to derive cost gradients for every layer of the feedforward neural network. This process of working out the cost gradients beginning from the output layer and all the way to the first layer is known as back propagation.

1.3 Softplus, ReLU and the Smoothing Activation Functions

With the intention of mimicking biological intelligence through neural networks, activation functions serve as decision gates determining if a "neuron" (or a node) should be "triggered". The activation function of a node considers the linear multiplication product (which we derive by obtaining the product sum between a vector of parameters and a vector of inputs and then adding a bias term for the node) and controls the final output from the node accordingly. In back propagation, the derivative of the activation function with respect to its input (i.e., $\frac{da}{dz}$) must be calculated to determine the cost gradients.

1.3.1 The Softplus Activation Function, $g(z) = ln(1 + e^z)$

Introduced in 2000, the Softplus [5] activation function represents an improvement over the activation functions of choice thitherto[2]. Comparatively, the Softplus activation allows for a larger rate of learning for larger input values, vis-à-vis the sigmoid / tanh activation function which saturates (i.e. gradient becomes near zero) when the input value becomes too positive or negative.



[2] Namely, the sigmoid function and tanh function. Nevertheless however, the sigmoid function retains its usefulness as an activation function for the final output layer in the case of binary classification neural network model.

1.3.2 The ReLU Activation Function, $g(z) = max\{0, z\}$

The ReLU activation function was introduced in 2001. It became the default choice of activation function for neural networks[6][7][8]. In a landmark 2011 study [9], it was empirically established that the ReLU, rather unintuitively, performed better than the Softplus activation function. This was despite the Softplus function being differentiable at every single point, which was necessary for the back propagation to work. The relatively superior performance of ReLU was attributed to the sparse nature of the function, which (1) reduces non-positive elements within the input matrix to zero, and (2) computes both the output values and cost gradients in a simple manner. This contrasts with the Softplus function, whereby each forward / backward propagation computation involves the use of an exponential and/or a logarithmic function which adds to the computational burden.

Concurrently, while the ReLU is not differentiable at the point x = 0, the use of ReLU as an activation function is not necessarily hindered as (1) the computation of $\frac{dz}{da}$ rarely takes place at the origin of the activation-input graph, (2) and in the event that computation of $\frac{dz}{da}$ does occur at the origin, an arbitrary value of 0 or 1 (the gradient for a positive or non-positive input value, respectively) can be assigned. Nevertheless, one notable problem with the ReLU lies in the flatness of the activation function for negative input values. To overcome this, a Leaky RELU was proposed in 2013 [10]. The Leaky RELU assigns a small $\frac{dz}{da}$ value of 0.01 for negative input values, thus ensuring that the model continues to train even with negative input values.

However, the superiority of ReLU is not unchallenged. In a separate study by Zheng, Yang, and Liu [11], it was shown that Deep Neural Networks using the Softplus activation function outperforms the ReLU activation function for deep neural networks under certain hyper-parametrizations.

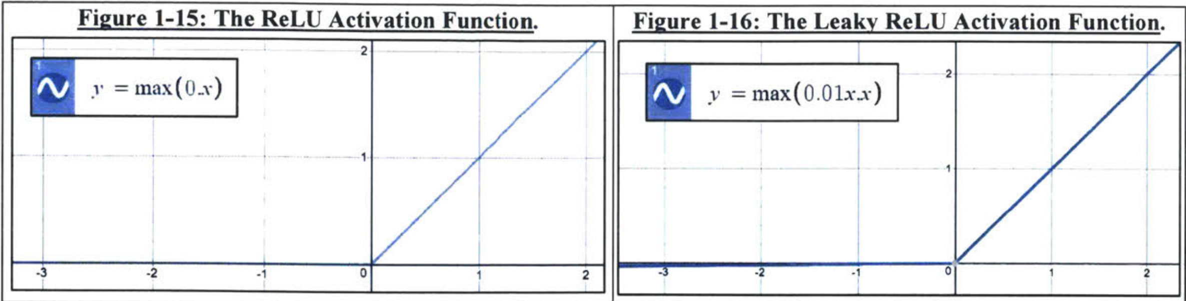


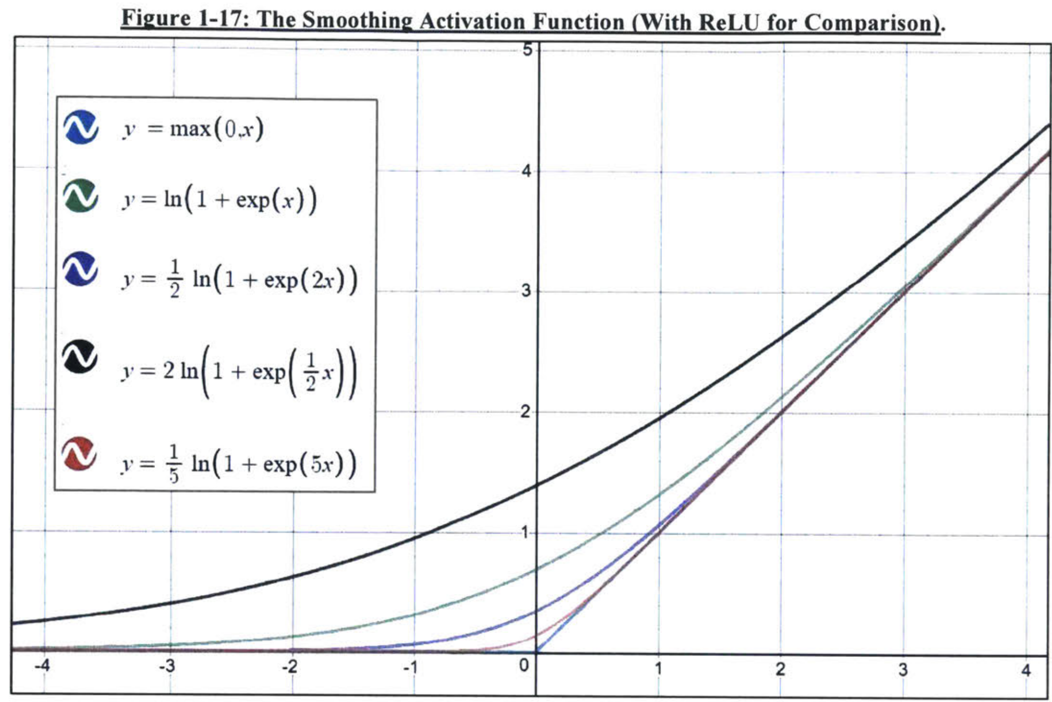
Figure 1-15: The ReLU Activation Function. **Figure 1-16: The Leaky ReLU Activation Function.**

1.3.3 The Smoothing Activation Function, $g(z) = \mu\, ln(1 + e^{\frac{z}{\mu}})$

The Smoothing activation function is the subject of interest for this study. Of note, (1) the Smoothing activation function is equivalent to the Softplus activation function for $\mu = 1$; (2) As μ approaches 0, the Smoothing activation function approaches the ReLU activation function (See **Figure 1-17**).

Importantly, the Smoothing activation is differentiable (with a non-zero gradient) across its entire domain, thus allowing the model to continue training across all input values. This property might enable a neural network model employing the Smoothing activation function to train at a faster rate when compared to a neural network employing a ReLU activation function. However, using a Smoothing activation function for the hidden layers still incurs the additional computational load highlighted for the Softplus model.

Figure 1-17: The Smoothing Activation Function (With ReLU for Comparison).



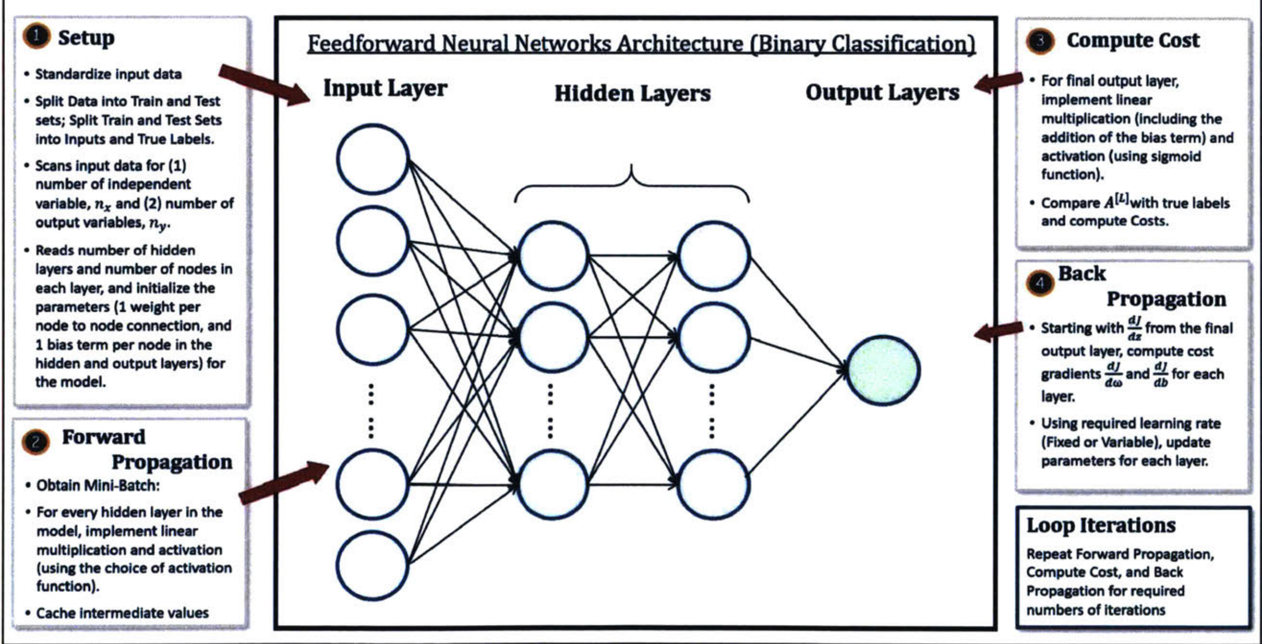
Chapter 2: Feedforward Neural Network Model Architecture

This chapter will guide readers through the code architecture for a feedforward neural network – the neural network model used for our experiments. Importantly, tailored features have been augmented into the code to enhance the data manipulation and collection process. For reproducibility, a copy of the code used has been appended in **Appendix A**.

2.1 Architecture Overview

The code architecture of the neural network model is presented in **Figure 2-18**. The initial setup standardizes the input data and initializes the parameters (depending on the number of hidden layers and the number of nodes in each hidden layer as required) in the model. Then, the modules performing the function of (1) forward propagation, (2) computing of the cost using the values of the parameters at the end of every iteration, and (3) back propagation over a certain number of iterations are developed separately. At the same time, intermediate values during the forward propagation will be captured to compute cost gradients during the back propagation process. Finally, the performance of the model is assessed by tracking cost values at the end of each iteration.

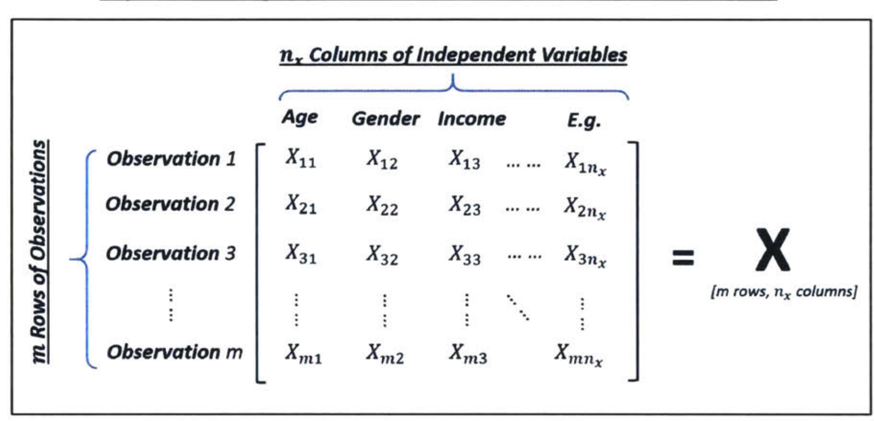
Figure 2-18: Architecture for Feedforward Neural Networks (Binary Classification).



2.2 Matrix Implementation

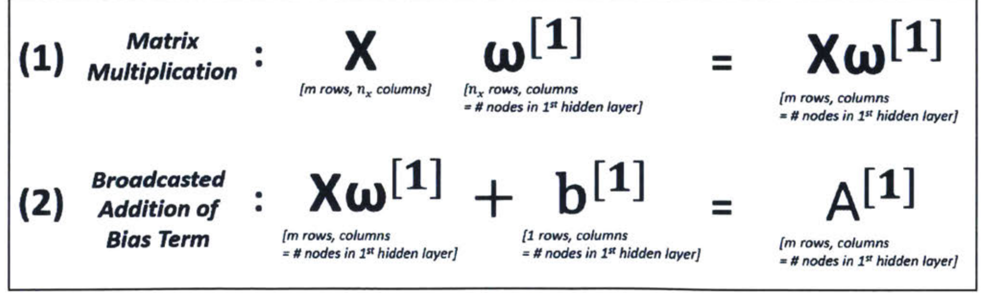
To leverage the efficiency of parallel computing, input values, intermediate linear multiplication, activation values, parameters, and cost gradients are represented and processed in their matrix forms. We begin with the observed values of all independent variables. These values are presented in a matrix form denoted as **X**. Each column in **X** represents an independent variable while each row entry in **X** corresponds to an observation.

Figure 2-19: Representation of Observation Data in Matrix Form.



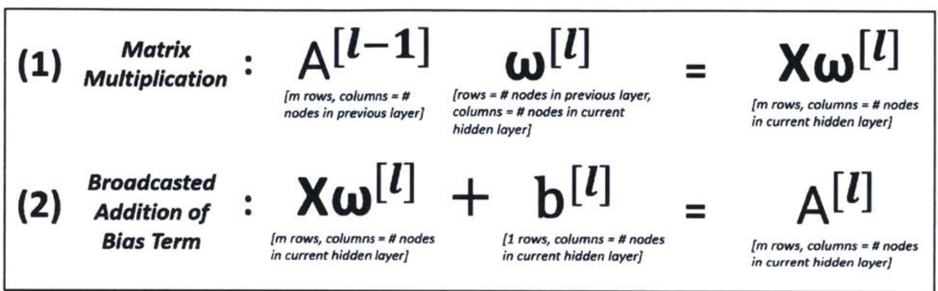
In the first hidden layer, the linear multiplication operation is performed using a matrix multiplication between input values **X** and the matrix of parameters $\boldsymbol{\omega}^{[1]}$, followed by a broadcasted addition of the product $\boldsymbol{X\omega}^{[1]}$ by the vector of bias terms $\boldsymbol{b}^{[1]}$. Of note, the dimensionality of the matrix of the parameters for first hidden layer $\boldsymbol{\omega}^{[1]}$, and $\boldsymbol{b}^{[1]}$ is such that: (1) For $\boldsymbol{\omega}^{[1]}$, the number of rows is equal to the number of nodes in previous layer (i.e. number of independent variables or columns in the input data **X**), while the number of columns is equal to the number of nodes in the first layer, and (2) For $\boldsymbol{b}^{[1]}$, the number of elements is equal to the number of nodes in the first layer.

Figure 2-20: Linear Multiplication Operation in Matrix Operation for 1st Hidden Layer.



To complete the forward propagation process, the operation in the previous paragraph is repeated for each successive layer. Likewise, the dimensionality of the matrix of the parameters for each layer $\boldsymbol{\omega}^{[l]}$, and $\boldsymbol{b}^{[l]}$ is such that: (1) For $\boldsymbol{\omega}^{[l]}$, the number of rows is equal to the number of nodes in previous layer, while the number of columns is equal to the number of nodes in the current layer, and (2) For $\boldsymbol{b}^{[l]}$, the number of elements is equal to the number of nodes in the current layer[3].

Figure 2-21:Generalized Linear Multiplication in Matrix Operations.



The use of matrix operations is likewise applied for the backward propagation process[4]. Beginning with the final layer, we compute the cost gradients $\frac{dJ}{d\omega^{[L]}}, \frac{dJ}{db^{[L]}}$. Since we are using the sigmoid function, the cost gradients were found to be $\frac{dJ}{d\omega} = \frac{1}{m}\sum(A^{[L]} - Y) \odot (A^{[L-1]})$, $\frac{dJ}{db} = \frac{1}{m}\sum(A^{[L]} - Y)$, where $\odot$ indicates an element-multiplication between the two matrices.

Figure 2-22: Backward Propagation in Matrix Representation.

(1) $$\frac{dJ}{d\omega^{[L]}} = \frac{1}{m}\left(A^{[L-1]}\right)^{T}\left(A^{[L]} - Y\right)$$

[row = # nodes in previous layer, columns = # nodes in current layer = 1] *[rows = # nodes in previous hidden layer, m columns]* *[m rows, 1 column]*

(2) $$\frac{dJ}{db^{[L]}} = \frac{1}{m}\sum\left(A^{[L]} - Y\right)$$

[1 row, columns = # nodes in current layer = 1] *[m rows, 1 column]*

---

[3] Given that the matrix operation AB for the 2-Dimensional matrices A and B is only possible if the 2nd dimension (number of columns) of A is equal to the first dimension (number of rows) of B, the matrix operations will only be possible if the dimensions of the parameter matrices have been initialized correctly as intended. This is a common technique used in ensuring that a neural network has been set up correctly.

[4] As with forward propagation, a common technique used to ensure that the neural net was properly set up lies in lies in ensuring that the cost gradient $\frac{dJ}{d\omega}$ and $\frac{dJ}{db}$ for each layer have the same dimensions as the matrix of parameters $\boldsymbol{\omega}$ and b for that layer.

The process is repeated (using the appropriate $\frac{dJ}{d\omega^{[l]}}, \frac{dJ}{db^{[l]}}$ for the choice of activation function in the hidden layers, and building upon the chain rule.) as we back propagate across the layers.

2.3 Additional Augmented Features

In addition to the essentials discussed above, the following were augmented into the code:

2.3.1 Mini-Batch Training with Replacement

This mini-batch creator module enables mini-batch training with replacement. This module takes in a batch size and outputs a random mini-batch (of the required size) from the training set. The mini-batch is randomized at every iteration to ensure that a different set of observations is used to train the model.

Batch training can be enabled using a batch size equal to the number of observations in the training set.

2.3.2 Optimized Learning Rates (Fixed and Variables)

The model will leverage a study by Freund, Grigas, and Mazumder [12] to initialize the optimal learning rates **α**: For the set of experimental runs conducted in the study, we will use both constant learning rates (i.e. the same learning rate across iterations) and variable learning rates (i.e. the learning rate changes with each iteration). Under the recommendations in [12], the optimal learning rates are given in **Table 1**:

Table 1: Learning Rates for Different Learning Model Parameters.

	Fixed α	**Variable α**
Batch Learning	$\alpha = \frac{4nc}{(\Sigma_{max})^2}$ **n** = Number of Observations in Training Set Σ_{max} = Maximum Singular Value for the Matrix of Input Values augmented with a column of 1-s (as the left most column). c = Learning Factor, an Arbitrary Constant Used to Refine the Learning Rate *(a range of values will be explored)*	$\alpha = \frac{4nc}{\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2 * (\Sigma_{max})^2}$ **n** = Number of Observations in Training Set Σ_{max} = Maximum Singular Value for the Matrix of Input Values augmented with a column of 1-s (as the left most column) $\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2$ is the *l*-2 Norm for the Cost Gradients at the Respective Iteration c = Learning Factor, an Arbitrary Constant Used to Refine the Learning Rate *(a range of values will be explored)*

Mini-Batch Learning	$\alpha = \frac{c.log2}{R^2.\sqrt{I+1}}$ R^2 = Maximum Value of the Row Sums of the Square of the Input Training Set Matrix I = Number of Training Iterations c = Learning Factor, an Arbitrary Constant Used to Refine the Learning Rate *(a range of values will be explored)*	$\alpha = \frac{c.log2}{R^2.\sqrt{i+1}}$ R^2 = Maximum Value of the Row Sums of the Square of the Input Training Set Matrix i = Current Iteration c = Learning Factor, an Arbitrary Constant Used to Refine the Learning Rate *(a range of values will be explored)*

2.3.3 Cost Calculation with Average of Parameters over Past 100 Iterations

The code is augmented with an exploratory module that calculates the cost associated with the average of parameter values over the past 100 iterations[5].

2.3.4 Data Standardization

The data standardization process, which serves to enhance the training process, is described as follows: For each independent variable, each observed value is deducted by the mean of all observations (for that independent variable), and then divided them by the standard deviation of all observations (for that independent variable). This converts each observed value into number of standard deviations away from the observed mean value.

[5] The exploration yielded an interesting finding which we further discuss in Chapter 6 – The study examining the use of "averaged parameters" to arrive at a more optimal set of model parameters was recommended for a separate study.

Chapter 3: Scope of Experiment

This chapter provides an overview of the datasets used in the study and a description of the experimental design for this study.

3.1 Overview of Datasets

The experiment uses 3 datasets. These datasets were chosen for their relevance in the business domains, particularly healthcare, banking, and finance.

3.1.1 Framingham Heart Study Dataset

The Framingham Heart Study [13] began in 1948 as an initiative to identify factor contributing to coronary heart disease. The dataset comprises a study population of 3,658 observations, and captures 15 independent variables detailing the demographics (such as age, gender, educational level, etc.) and biometrics (such as Body Mass Index, cholesterol levels, heart rate, etc.) of each individual in the study population. The response variable "TenYearCHD" indicates the onset of coronary heart diseases within a ten year period.

3.1.2 Bank Churn Modelling Dataset

The Bank Churn Modelling dataset [14] was obtained from Kaggle, an online community with open datasets for data scientists. This dataset comprises 10,000 observations collated over different banks in Europe, and captures 10 independent variables covering demographics (such as the country, gender, and age) and banking related details (such as length of tenure with the bank, the number of products with the bank, whether the client owns a credit line, etc.) of each individual in the study population. The response variable "Exited" indicates if a particular individual eventually leaves the bank as a client.

3.1.3 German Credit Risk Dataset

The German Credit Risk Dataset [15] was collated by the Department of Statistics and Econometrics at the University of Hamburg. Covering a study population of 1000 individuals, the dataset collects 30 independent variables detailing the demographics (Gender, marital status, age,

type of employment, etc.) and economic background (car ownership, amount of savings in bank, property ownership) for each individual in the dataset. The response variable "RESPONSE" indicates if the individual has a good or bad credit rating at the time of the study.

3.2 Experimental Design

Each dataset is split into a training set and a testing set comprising 75% and 25% of the observations respectively. In assessing the performances of the Smoothing activation function in a holistic manner, we will train neural networks encompassing a range of the following hyperparameters:

- Number of Hidden Layers. Both shallow neural networks (with 1 hidden layer) and deep neural networks will be used in the study. The number of nodes in the hidden layers used for each of the datasets is shown in **Table 2**.

Table 2: Number of Hidden Layers Used for Each Dataset.

Dataset	Number of Independent Variables	Number of Nodes in Hidden Layers	
		Shallow Neural Network	Deep Neural Network
Framingham Heart Study	15 *(*18 after conversion of categorical variables into binary variables)*	13	13, 8
Bank Churn Modelling	10 *(*13 after conversion of categorical variables into binary variables)*	8	8, 5
German Credit Risk	30 *(*51 after conversion of categorical variables into binary variables)*	35	35, 24

- Type of Activation Function. The Smoothing activation functions and the ReLU activation function will be used to train the neural networks. For the Smoothing activation function $g(z) = \mu\, ln(1 + e^{\frac{z}{\mu}})$ in particular, we will examine activation functions for $\mu = 0.5, \mu = 1,$ and $\mu = 2.$

To recap, the Smoothing activation function for $\boldsymbol{\mu} = \mathbf{1}$ *is equivalent to the Softplus activation function.*

- Batch Training or Mini-Batch Training. The neural networks developed will include models using batch learning and models using mini-batch learning. A batch size of 16 will be used for mini-batch training.

- Learning Rate Type. As highlighted in **Table 1**, neural networks employing both fixed and variable learning rates will be used in this study.

- Learning Rate Multiple. A range of learning rate multiple will be used to examine the characteristics of the activation functions over a range of learning values.

- Number of Iterations. Each model will undergo 10,000 iterations.

A summary of the possible permutations for the hyperparameters is shown in **Table 3**. About 176 models were trained for each dataset.

Table 3: Permutation of Hyperparameters for Neural Networks.

Hyperparameters	Option 1	Option 2	Option 3	Option 4
Number of Hidden Layers	1	2		
Type of Activation Function	Softplus	ReLU	Smoothing ($\mu = 0.5$)	Smoothing ($\mu = 2$)
Batch / Mini-Batch Training	Batch Learning	Mini-Batch of Size 16		
Learning Rate Type	Fixed	Variable		
Learning Rate Multiple	**Covers a range of values:** • Between 0.000125 to 2 for Batch Learning • Between 1 to 800 for Mini-Batch Learning			
Number of Iterations	10,000			

Using the cost profile generated by the neural networks, the relative performances of the different activation functions will be examined and presented in the next chapter. Beyond ascertaining the

performance of the Smoothing activation function in comparison with the ReLU activation functions, interesting observations and insights will be noted for further studies.

Chapter 4: Analysis of Experimental Results

This chapter highlights and discusses findings from out experimental results.

4.1 Performance of Smoothing Activation Functions

We used 2 measures in evaluating the relative performances of the different neural network models: (1) The minimum cost on the training set achieved at the end of 10,000 iterations will be used as a proxy of how well each neural network model was able to fit the observations; (2) By further examining the cost descent profile by each model where appropriate, we also qualitatively evaluate the rate which a neural network model was trained.

4.1.1 Analysis of Results for Shallow (2-Layer) Neural Networks

I. Batch Learning with Fixed Learning Rates. Under batch learning conditions with fixed learning rates, the performances of 2-layer neural network models using the Smoothing activation functions relative to models using the ReLU activation function varied across datasets. **Table 4** presents the minimum cost on the training set achieved by each of the model at the end of 10,000 iterations:

- Neural network models using the ReLU activation function were able to achieve the lowest costs on the respective training sets of the Bank Churn Modelling and German Credit Risk datasets.

- Neural network models using the Softplus activation function achieved the lowest cost on the training set for the Framingham Heart Study dataset, albeit neural networks using other activation functions were also able to achieve costs that were nearly as low (i.e. within 10%) as what the Softplus activation model achieves.

- The performance of the Smoothing activation functions (including Softplus) across the datasets are generally inconsistent. While models employing the Smoothing activation functions were able to achieve near-minimum costs on the training set of the Framingham Heart Study dataset, the costs achieved for other datasets were generally inferior to what models using the ReLU activation functions were able to achieve.

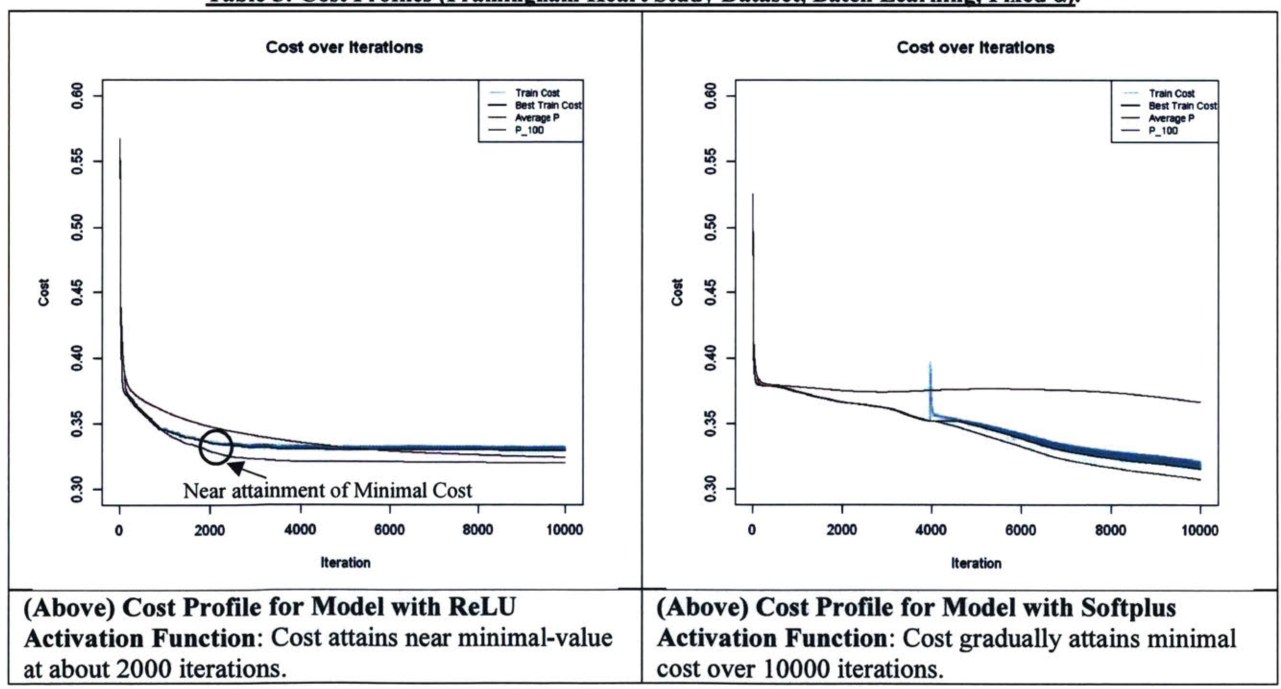
Table 4: Shallow Networks - Minimum Cost Achieved on Training Set (Batch Learning, Fixed α Models).

Dataset	Learning Rate, α	Minimum Cost Achieved on the Training Set (Batch Learning)			
		ReLU	**Softplus**	**Smoothing, μ = 0.5**	**Smoothing, μ = 2**
Framingham Heart Study	$\frac{4nc}{(\Sigma_{max})^2}$, c =1	0.330	**0.315**	0.328	0.342
	$\frac{4nc}{(\Sigma_{max})^2}$, c =2	0.329	**0.315**	0.326	0.319
Bank Churn Modelling	$\frac{4nc}{(\Sigma_{max})^2}$, c =1	0.324	0.328	0.390	0.398
	$\frac{4nc}{(\Sigma_{max})^2}$, c =2	**0.323**	0.395	0.397	0.394
German Credit Risk	$\frac{4nc}{(\Sigma_{max})^2}$, c=$\frac{1}{32}$	**0.014**	0.134	0.048	0.305
	$\frac{4nc}{(\Sigma_{max})^2}$, c=$\frac{1}{64}$	0.061	0.356	0.264	0.407

* **Blue** indicates best performing model among models with the same set of hyperparameters (i.e., for each row); **Green** indicates model with comparable cost performance with best performing model; **Red** indicates marked inferiority in cost performance against best performing model; **Bold** indicates best result for dataset.

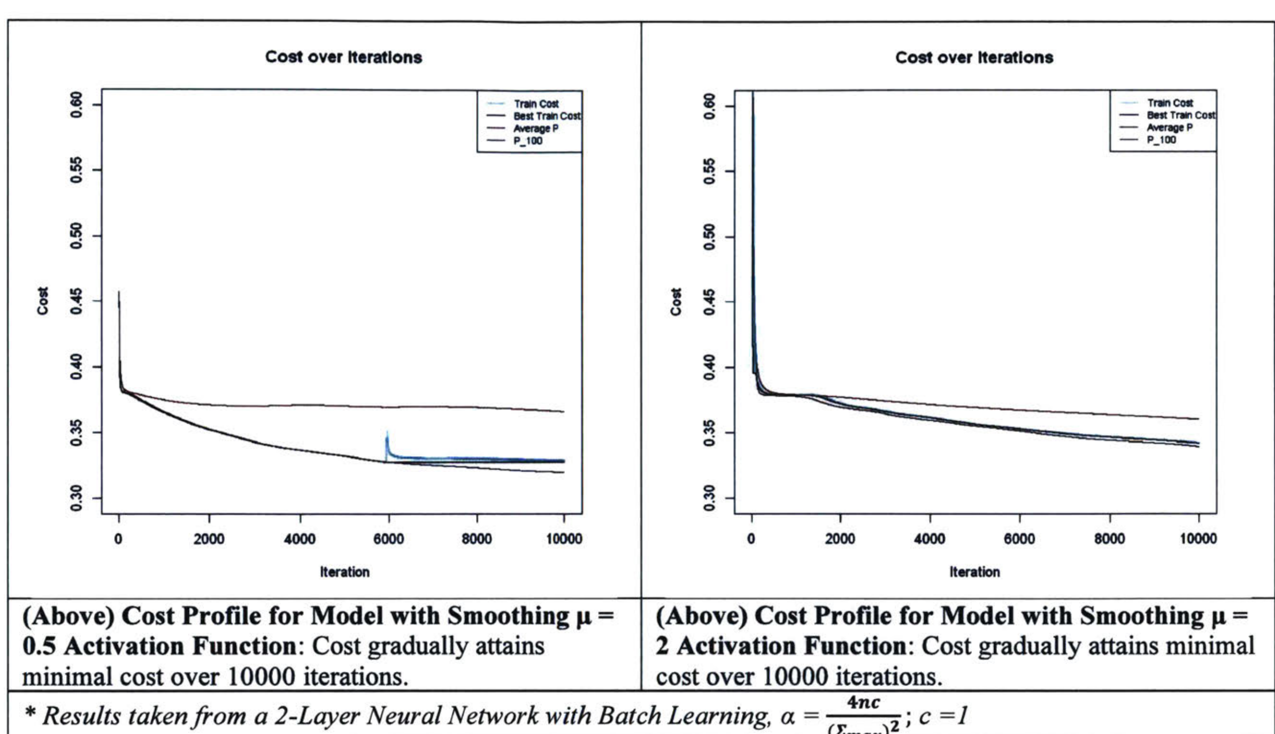
A closer inspection of the cost profiles generated by the different models on the Framingham Dataset (where models using the different activation functions were able to achieve similar performances) further revealed that the model using the ReLU activation function was able to converge towards its eventual minimal cost value more rapidly (See **Table 5**).

Table 5: Cost Profiles (Framingham Heart Study Dataset, Batch Learning, Fixed α).



(Above) Cost Profile for Model with ReLU Activation Function: Cost attains near minimal-value at about 2000 iterations.

(Above) Cost Profile for Model with Softplus Activation Function: Cost gradually attains minimal cost over 10000 iterations.



(Above) Cost Profile for Model with Smoothing μ = 0.5 Activation Function: Cost gradually attains minimal cost over 10000 iterations.	**(Above) Cost Profile for Model with Smoothing μ = 2 Activation Function**: Cost gradually attains minimal cost over 10000 iterations.

** Results taken from a 2-Layer Neural Network with Batch Learning,* $\alpha = \frac{4nc}{(\Sigma_{max})^2}$; *c =1*

II. <u>Batch Learning with Variable Learning Rates</u>. The results are largely similar to results from batch learning models with fixed learning rates: Performances of 2-layer neural network models using the Smoothing activation functions relative to models using the ReLU activation function varied across datasets. **Table 6** presents the minimum cost on the training set achieved by each of the models at the end of 10,000 iterations:

- In all cases, neural network models using the ReLU activation function were able to achieve the lowest or near-lowest (within 10% of lowest) costs on the training set of all 3 datasets.

- The performance of neural network models using the Smoothing activation function were more inconsistent: Models using the Softplus and the Smoothing ($\mu = 0.5$) activation functions were able to achieve the lowest cost on the training set on the Framingham Heart Study dataset. On the Bank Churn Modelling and German Credit datasets however, models using the Smoothing activation functions (including the Softplus activation function) were generally observed to achieve training costs that were significantly inferior to models using the ReLU activation function.

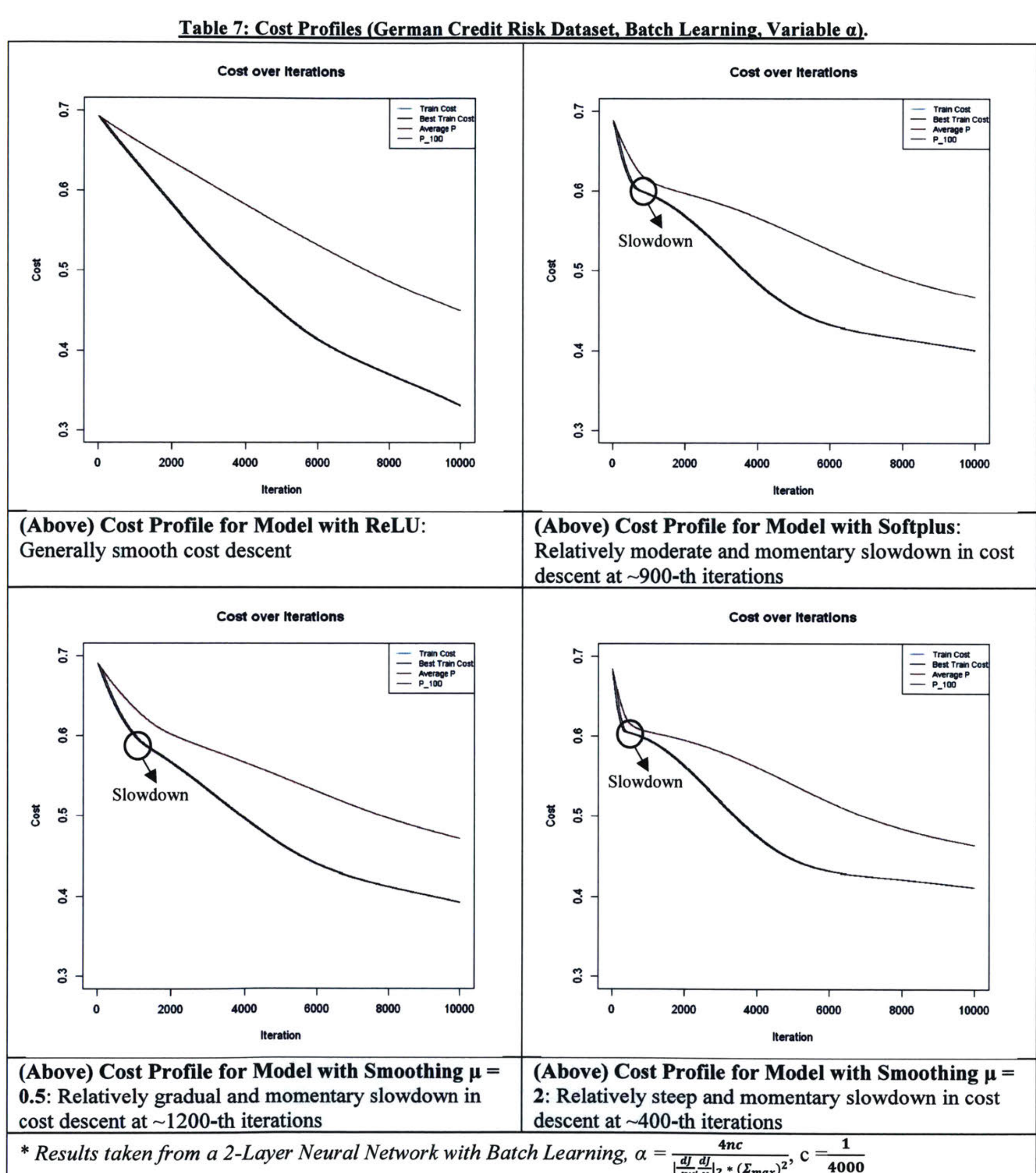
Table 6: Shallow Networks - Minimum Cost Achieved on Training Set (Batch Learning, Variable α Models).

		Minimum Cost Achieved on the Training Set (Batch Learning)			
Dataset	**Learning Rate, α**	**ReLU**	**Softplus**	**Smoothing, $\mu = 0.5$**	**Smoothing, $\mu = 2$**
Framingham Heart Study	$\frac{4nc}{\|\frac{dJ}{dW}\frac{dJ}{db}\|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{16}$	0.307	**0.302**	0.318	0.323
	$\frac{4nc}{\|\frac{dJ}{dW}\frac{dJ}{db}\|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{32}$	0.308	0.308	0.306	0.340
Bank Churn Modelling	$\frac{4nc}{\|\frac{dJ}{dW}\frac{dJ}{db}\|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{16}$	0.335	0.413	0.386	0.372
	$\frac{4nc}{\|\frac{dJ}{dW}\frac{dJ}{db}\|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{8}$	**0.333**	0.395	0.395	0.353
German Credit Risk	$\frac{4nc}{\|\frac{dJ}{dW}\frac{dJ}{db}\|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{8000}$	0.446	0.451	0.463	0.445
	$\frac{4nc}{\|\frac{dJ}{dW}\frac{dJ}{db}\|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{4000}$	0.331	0.401	0.392	0.412
	$\frac{4nc}{\|\frac{dJ}{dW}\frac{dJ}{db}\|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{2000}$	**0.140**	0.292	0.245	0.331

* **Blue** indicates best performing model among models with the same set of hyperparameters (i.e., for each row); **Green** indicates model with comparable cost performance with best performing model; **Red** indicates marked inferiority in cost performance against best performing model; **Bold** indicates best result for dataset.

On the German Credit Risk Dataset with a learning factor of c = $\frac{1}{4000}$ particularly, (whereby the family of Smoothing activation functions were observed to register significantly inferior costs on the training set), it was observed from the cost profiles from models using the family of Smoothing activation functions experienced a momentary slowdown in the training process (See **Table 7**).

Table 7: Cost Profiles (German Credit Risk Dataset, Batch Learning, Variable α).



(Above) Cost Profile for Model with ReLU: Generally smooth cost descent

(Above) Cost Profile for Model with Softplus: Relatively moderate and momentary slowdown in cost descent at ~900-th iterations

(Above) Cost Profile for Model with Smoothing μ = 0.5: Relatively gradual and momentary slowdown in cost descent at ~1200-th iterations

(Above) Cost Profile for Model with Smoothing μ = 2: Relatively steep and momentary slowdown in cost descent at ~400-th iterations

* *Results taken from a 2-Layer Neural Network with Batch Learning,* $\alpha = \frac{4nc}{|\frac{dJ}{dW'}\frac{dJ}{db}|_2 * (\Sigma_{max})^2}$, c = $\frac{1}{4000}$

III. Mini-Batch Learning with Fixed Learning Rates. Except for scenarios with a lower learning factor, models using the ReLU activation functions generally achieved the lowest cost on the training sets across all datasets. Nevertheless, it should be noted that (1) The Smoothing family of activation functions generally comparable performances, if not the best, against the ReLU activation function on the Framingham Heart Study dataset, and (2) On the Bank Churn Modelling dataset, neural networks using the Smoothing $\mu = 2$ activation function further achieved comparable performances when compared with neural networks using the ReLU activation function (**Table 8**).

Table 8: Shallow Networks - Minimum Cost Achieved on Training Set (Mini-Batch Learning, Fixed α Models).

		Minimum Cost Achieved on the Training Set (Batch Learning)			
Dataset	**Learning Rate, α**	**ReLU**	**Softplus**	**Smoothing, $\mu = 0.5$**	**Smoothing, $\mu = 2$**
Framingham Heart Study	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 1	0.669	0.564	0.636	0.460
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 200	0.392	0.387	0.383	0.399
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 400	0.376	0.381	0.381	0.381
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 600	**0.373**	0.380	0.380	0.378
Bank Churn Modelling	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 1	0.587	0.510	0.544	0.505
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 200	0.338	0.423	0.412	0.415
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 400	0.336	0.410	0.406	0.345
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 600	**0.335**	0.409	0.404	0.345
German Credit Risk	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 1	0.675	0.609	0.633	0.607
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 200	0.216	0.409	0.393	0.422
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 400	0.058	0.357	0.256	0.408
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 600	**0.021**	0.246	0.124	0.385

* **Blue** indicates best performing model among models with the same set of hyperparameters (i.e., for each row); **Green** indicates model with comparable cost performance with best performing model; **Red** indicates marked inferiority in cost performance against best performing model; **Bold** indicates best result for dataset.

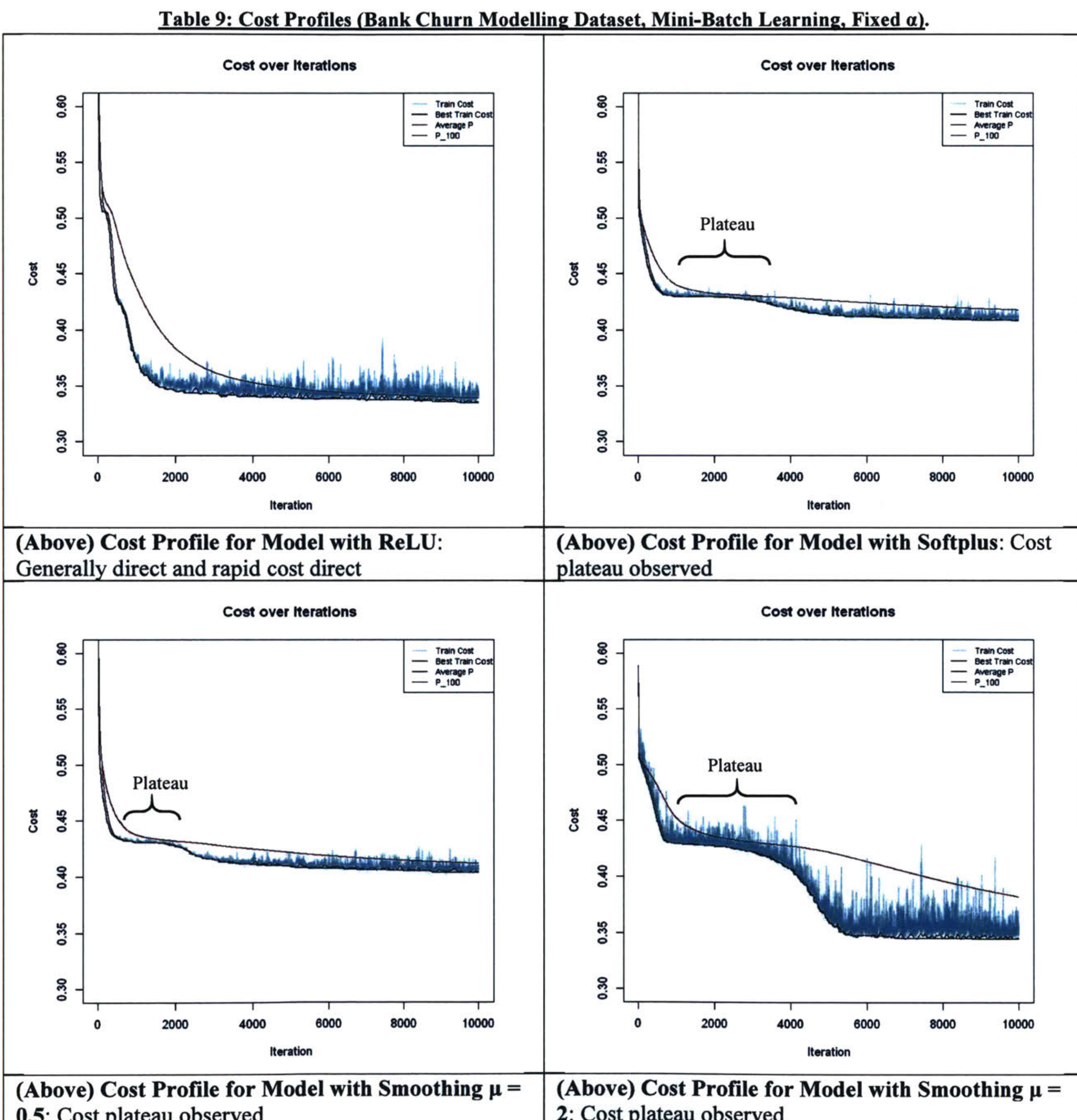
In addition, an examination of the cost profiles generated by the neural networks across the datasets reveals the following:

- On both the Bank Churn Modelling (**Table 9**) and German Credit Risk (**Table 10**) Modelling Dataset, the cost descent was direct for neural networks using the ReLU

activation function. This was not the case for neural networks using Smoothing activation functions, where a plateau in the cost descent could be observed.

- On the Framingham Heart Study dataset (See **Table 11**), the cost descent process was observed to be slower for neural networks employing the ReLU activation function. Significantly, the neural networks using the Smoothing activation functions for $\mu = 1$ (i.e., the Softplus activation functions) and $\mu = 0.5$ achieved a comparable minimum train cost with a lower number of iterations.

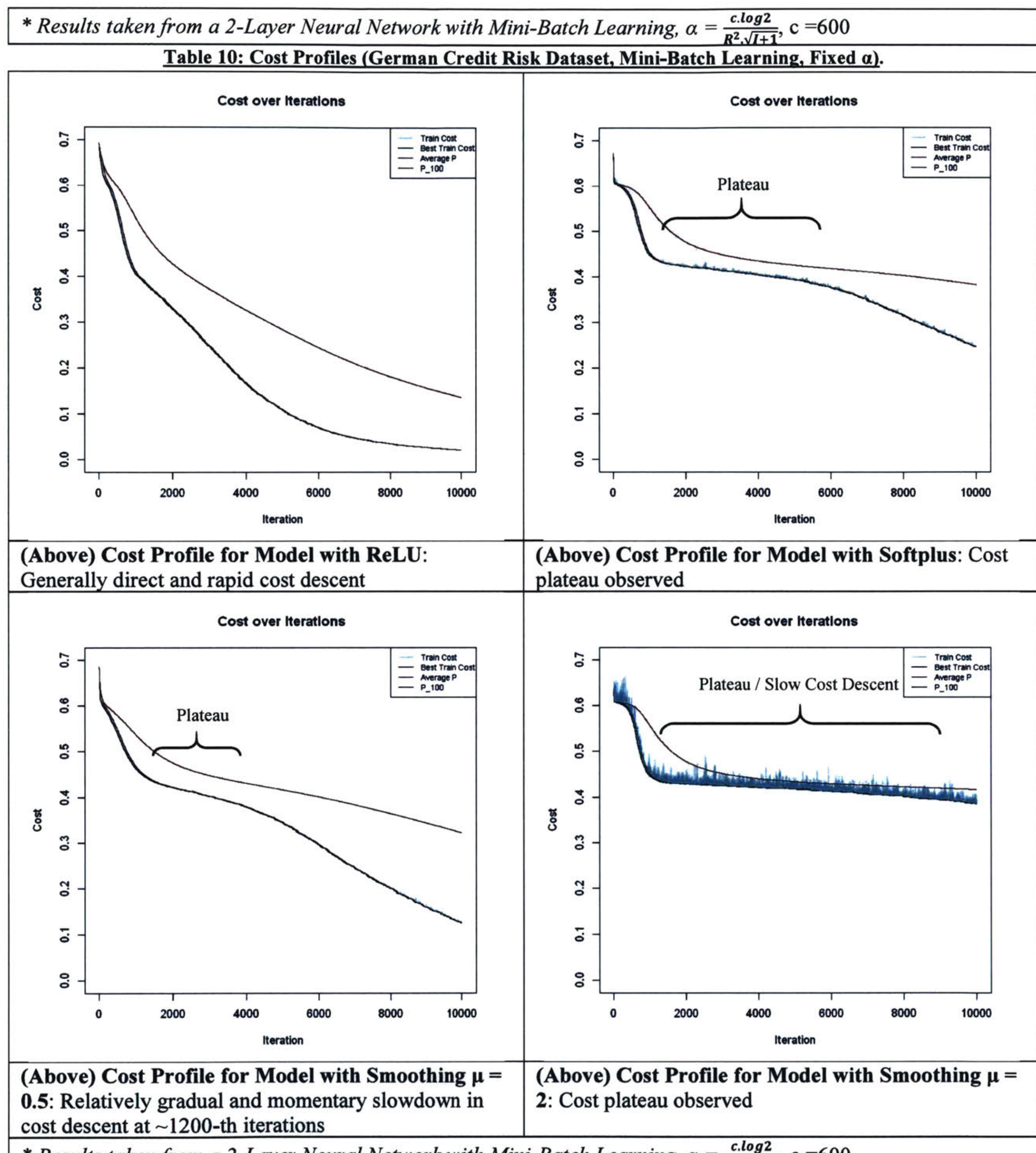
Table 9: Cost Profiles (Bank Churn Modelling Dataset, Mini-Batch Learning, Fixed α).



(Above) Cost Profile for Model with ReLU: Generally direct and rapid cost direct	**(Above) Cost Profile for Model with Softplus**: Cost plateau observed
(Above) Cost Profile for Model with Smoothing μ = 0.5: Cost plateau observed	**(Above) Cost Profile for Model with Smoothing μ = 2**: Cost plateau observed

* *Results taken from a 2-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{c.log2}{R^2.\sqrt{I+1}}$, c =600

Table 10: Cost Profiles (German Credit Risk Dataset, Mini-Batch Learning, Fixed α).



(Above) Cost Profile for Model with ReLU: Generally direct and rapid cost descent

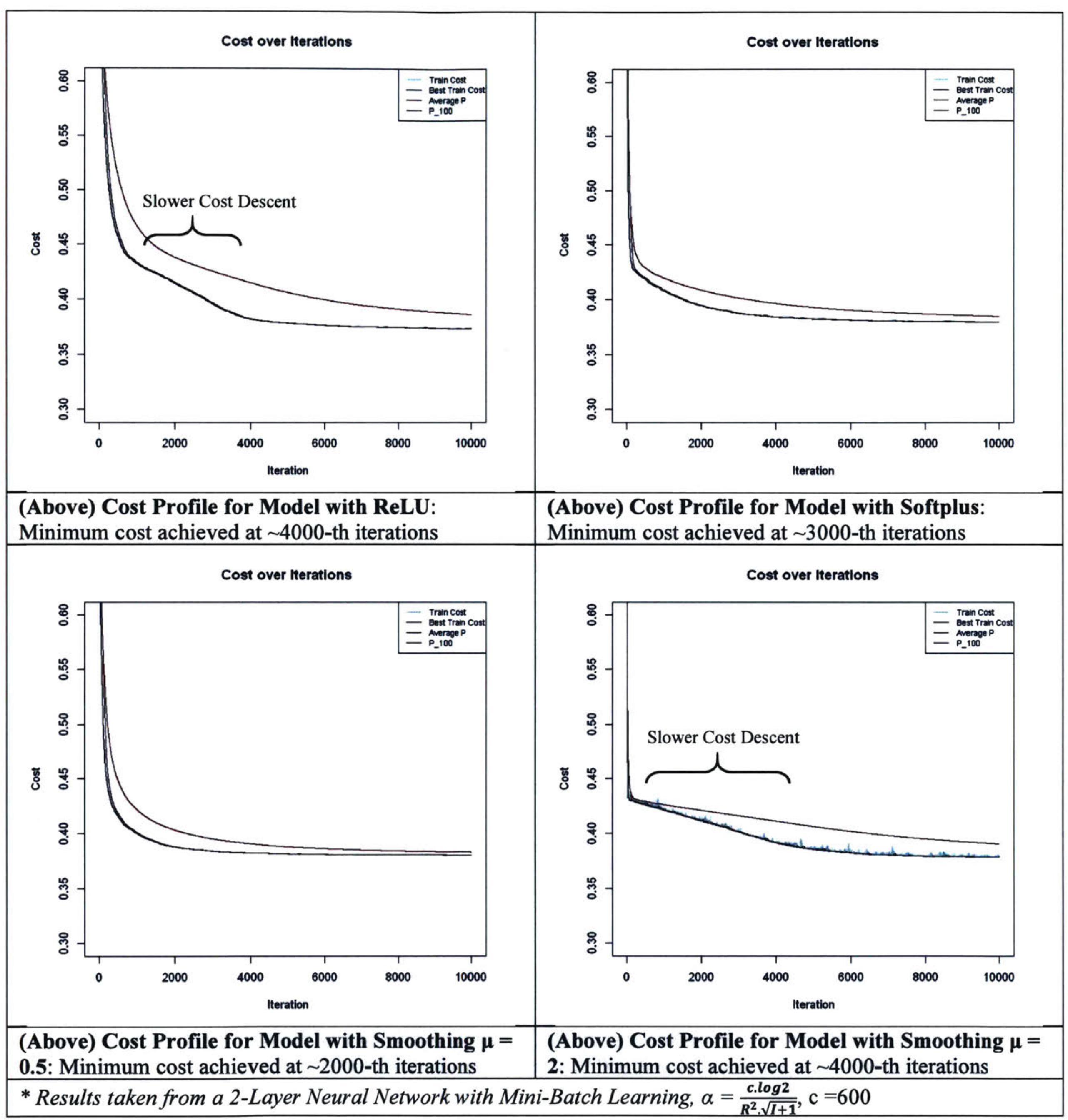
(Above) Cost Profile for Model with Softplus: Cost plateau observed

(Above) Cost Profile for Model with Smoothing μ = 0.5: Relatively gradual and momentary slowdown in cost descent at ~1200-th iterations

(Above) Cost Profile for Model with Smoothing μ = 2: Cost plateau observed

* *Results taken from a 2-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{c.log2}{R^2.\sqrt{I+1}}$, c =600

Table 11. Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Fixed α).



(Above) Cost Profile for Model with ReLU: Minimum cost achieved at ~4000-th iterations

(Above) Cost Profile for Model with Softplus: Minimum cost achieved at ~3000-th iterations

(Above) Cost Profile for Model with Smoothing μ = 0.5: Minimum cost achieved at ~2000-th iterations

(Above) Cost Profile for Model with Smoothing μ = 2: Minimum cost achieved at ~4000-th iterations

* *Results taken from a 2-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{c.log2}{R^2.\sqrt{I+1}}$, c =600

IV. Mini-Batch Learning with Variable Learning Rates. As with the case for Mini-Batch Training with fixed learning rate, the performance of neural nets employing the ReLU activation function were found to be superior in all cases across all datasets with the exception of scenarios with very low learning rates (i.e., c =1) (See **Table 12**). Said superiority, however, was more pronounced for the Bank Churn Modelling and German Credit Modelling datasets – Neural network models using the Smoothing activation functions were able to achieve comparable cost values for the Framingham Heart Study dataset.

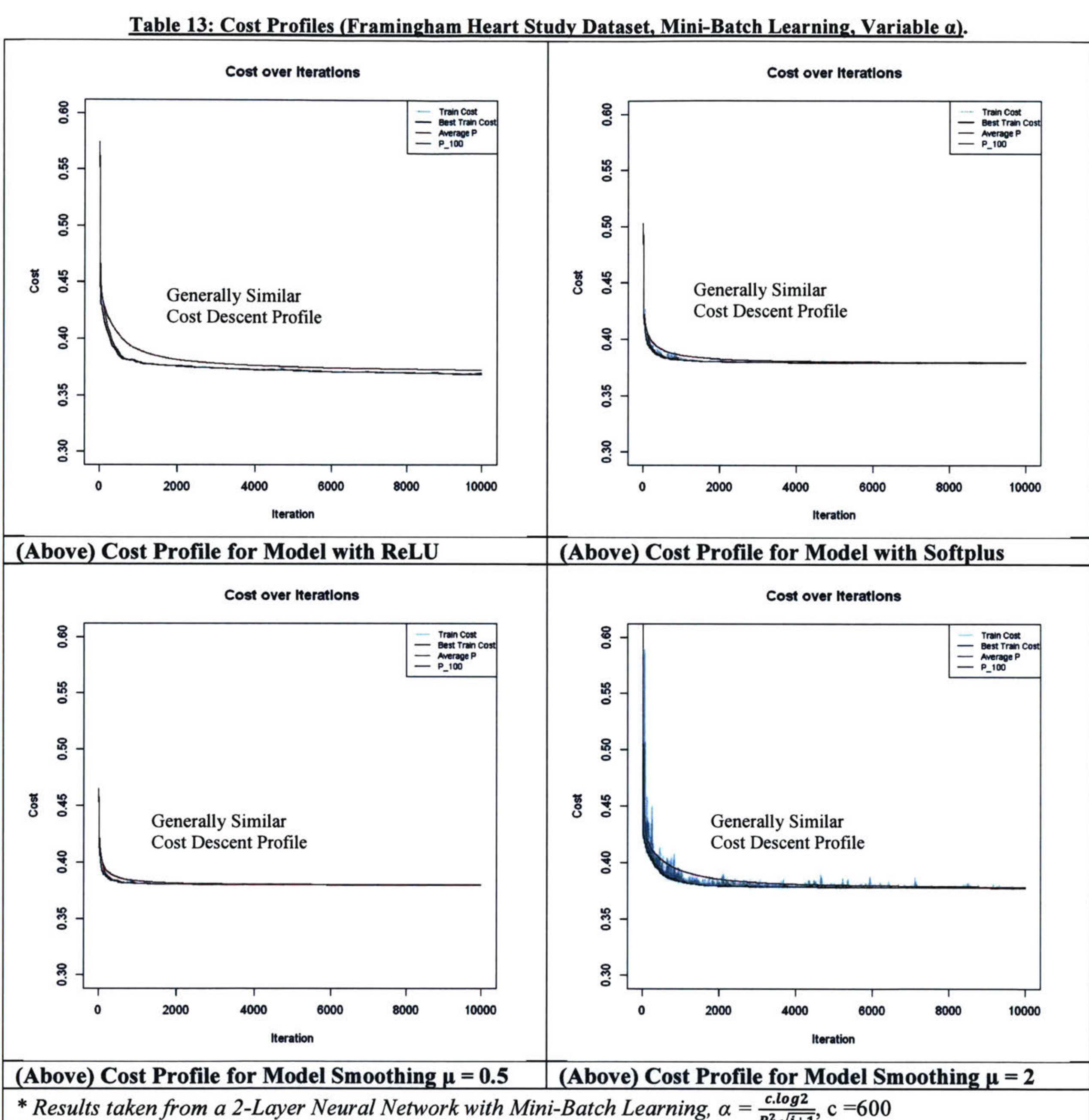
Table 12: Shallow Networks - Minimum Cost Achieved on Training Set (Mini-Batch Learning, Variable α).

		Minimum Cost Achieved on the Training Set (Batch Learning)			
Dataset	**Learning Rate, α**	**ReLU**	**Softplus**	**Smoothing, μ = 0.5**	**Smoothing, μ = 2**
Framingham Heart Study	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 1	0.647	0.503	0.593	0.436
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 200	0.376	0.381	0.381	0.381
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 400	0.371	0.379	0.380	0.378
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 600	**0.367**	0.379	0.379	0.378
Bank Churn Modelling	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 1	0.544	0.501	0.511	0.504
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 200	0.336	0.410	0.406	0.420
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 400	**0.330**	0.408	0.402	0.417
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 600	0.336	0.409	0.403	0413
German Credit Risk	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 1	0.692	0.682	0.689	0.664
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 200	0.068	0.354	0.268	0.402
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 400	**0.017**	0.209	*NA (Unable to Converge)*	*NA (Unable to Converge)*

* **Blue** indicates best performing model among models with the same set of hyperparameters (i.e., for each row); **Green** indicates model with comparable cost performance with best performing model; **Red** indicates marked inferiority in cost performance against best performing model; **Bold** indicates best result for dataset.

Unlike the case for Mini-Batch Training with fixed learning rate, the rate of convergence for neural networks using the ReLU activation function were found to be largely similar to the rate convergence for neural networks using the Smoothing activation functions (See **Table 13**). Taking into account that neural networks using the Smoothing activation functions attained a train cost that was comparable to that attained by neural networks using the ReLU activation function, the performances of models using either activation function are deemed to be comparable.

Table 13: Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Variable α).



(Above) Cost Profile for Model with ReLU

(Above) Cost Profile for Model with Softplus

(Above) Cost Profile for Model Smoothing $\mu = 0.5$

(Above) Cost Profile for Model Smoothing $\mu = 2$

* *Results taken from a 2-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{c.\log 2}{R^2.\sqrt{i+1}}$, c =600

4.1.2 Analysis of Results for Deep (3-Layer) Neural Networks

I. Batch Learning with Fixed Learning Rates. Using batch learning with fixed learning rates, neural networks using the ReLU activation functions achieved the lowest for both the Bank Churn Modelling and German Credit Risk datasets. In addition,

- For the Framingham dataset, when using a learning factor of c =2, the neural network model using the Softplus activation function was able to fit the training set data better. At the same time, however, the family of Smoothing activation functions were not consistent in achieving a train cost that is near the minimum achieved by the Softplus activation function.

- For the Bank Churn Modelling dataset, neural networks using the Smoothing activation functions achieved minimum costs on the training set that were comparable to the minimum achieved by the neural networks using the ReLU activation function (**Table 14**).

Table 14: Deep Networks - Minimum Cost Achieved on Training Set (Batch Learning, Fixed α Models).

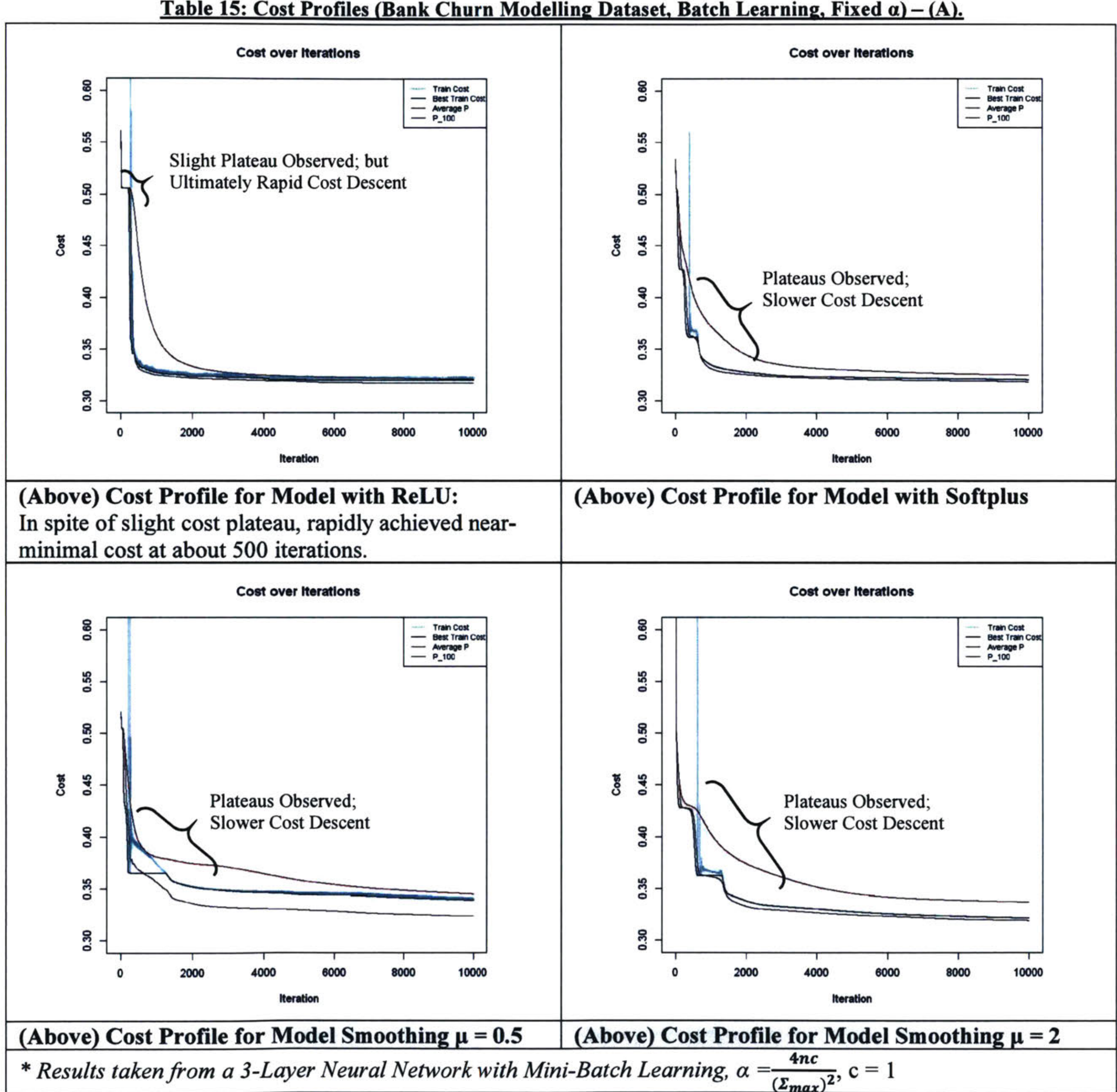
		Minimum Cost Achieved on the Training Set (Batch Learning)			
Dataset	**Learning Rate, α**	**ReLU**	**Softplus**	**Smoothing, $\mu = 0.5$**	**Smoothing, $\mu = 2$**
Framingham Heart Study	$\frac{4nc}{(\Sigma_{max})^2}$, c =1	0.221	0.250	0.288	0.346
	$\frac{4nc}{(\Sigma_{max})^2}$, c =2	*NA (Unable to Converge)*	**0.210**	*NA (Unable to Converge)*	0.268
Bank Churn Modelling	$\frac{4nc}{(\Sigma_{max})^2}$, c =1	0.320	0.320	0.337	0.321
	$\frac{4nc}{(\Sigma_{max})^2}$, c =2	**0.311**	0.319	0.322	0.314
German Credit Risk	$\frac{4nc}{(\Sigma_{max})^2}$, $c = \frac{1}{32}$	**0.002**	0.170	*NA (Unable to Converge)*	0.391
	$\frac{4nc}{(\Sigma_{max})^2}$, $c = \frac{1}{64}$	0.070	0.415	0.380	0.577

* **Blue** indicates best performing model among models with the same set of hyperparameters (i.e., for each row); **Green** indicates model with comparable cost performance with best performing model; **Red** indicates marked inferiority in cost performance against best performing model; **Bold** indicates best result for dataset.

In noting that the neural networks using each of the different activation functions achieved comparable results on the Bank Churn Modelling dataset, the cost profiles generated by these models were further analyzed.

- Using a learning factor of c = 1, an analysis of the costs profiles revealed that the cost descent for the neural network model using the ReLU activation functions was faster, while the neural networks using the various Smoothing activation functions suffered a plateau in their cost descent during the training process (See **Table 15**).

Table 15: Cost Profiles (Bank Churn Modelling Dataset, Batch Learning, Fixed α) – (A).



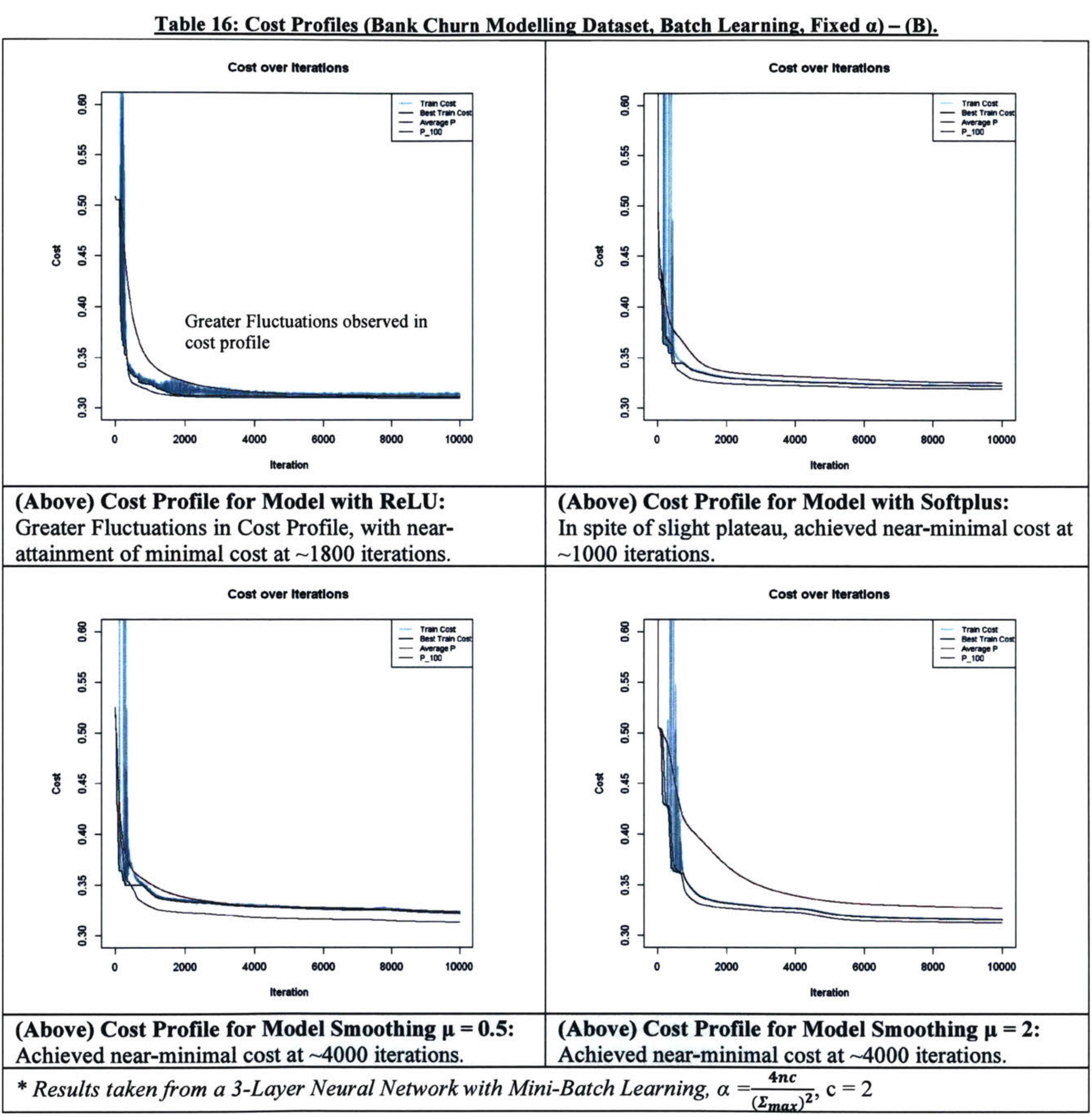
(Above) Cost Profile for Model with ReLU: In spite of slight cost plateau, rapidly achieved near-minimal cost at about 500 iterations.	**(Above) Cost Profile for Model with Softplus**
(Above) Cost Profile for Model Smoothing μ = 0.5	**(Above) Cost Profile for Model Smoothing μ = 2**

* *Results taken from a 3-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{4nc}{(\Sigma_{max})^2}$, c = 1

- Using a learning factor of c = 2 (where each model respectively produced their lowest cost on the training set for the Bank Churn Modelling dataset) however, we observe that the cost descent for the neural network model using the Softplus activation function attained a

faster cost descent, and with less fluctuations in the cost profile. This highlights a very specific instance whereby the Softplus function performed better than the ReLU activation function (See **Table 16**).

Table 16: Cost Profiles (Bank Churn Modelling Dataset, Batch Learning, Fixed α) – (B).



(Above) Cost Profile for Model with ReLU: Greater Fluctuations in Cost Profile, with near-attainment of minimal cost at ~1800 iterations.	**(Above) Cost Profile for Model with Softplus:** In spite of slight plateau, achieved near-minimal cost at ~1000 iterations.
(Above) Cost Profile for Model Smoothing μ = 0.5: Achieved near-minimal cost at ~4000 iterations.	**(Above) Cost Profile for Model Smoothing μ = 2:** Achieved near-minimal cost at ~4000 iterations.

* *Results taken from a 3-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{4nc}{(\Sigma_{max})^2}$, c = 2

II. Batch Learning with Variable Learning Rates. With batch learning and variable learning rates, neural network models using the ReLU activation function continue to dominate in terms of performance, being able to yield the lowest cost across all 3 different datasets. On the Framingham Heart Study and Bank Churn Modelling datasets, neural networks using a Smoothing activation function was able produce minimum costs that were comparable with neural network models employing the ReLU activation function. Particularly,

- On the Framingham Heart Study Dataset, the Softplus and Smoothing ($\mu = 0.5$) activation functions were able to achieve minimum costs on the training sets that were within 10% of what the ReLU activation function could achieve.

- On the Bank Churn Modelling, each of the Smoothing activation functions were able to achieve minimum costs on the training sets that were within 10% of what the ReLU activation function could achieve.

- On the Bank Churn Modelling dataset, none of the neural networks using a Smoothing activation function produced a minimum cost that could compete with the neural network using a ReLU activation function (See **Table 17**).

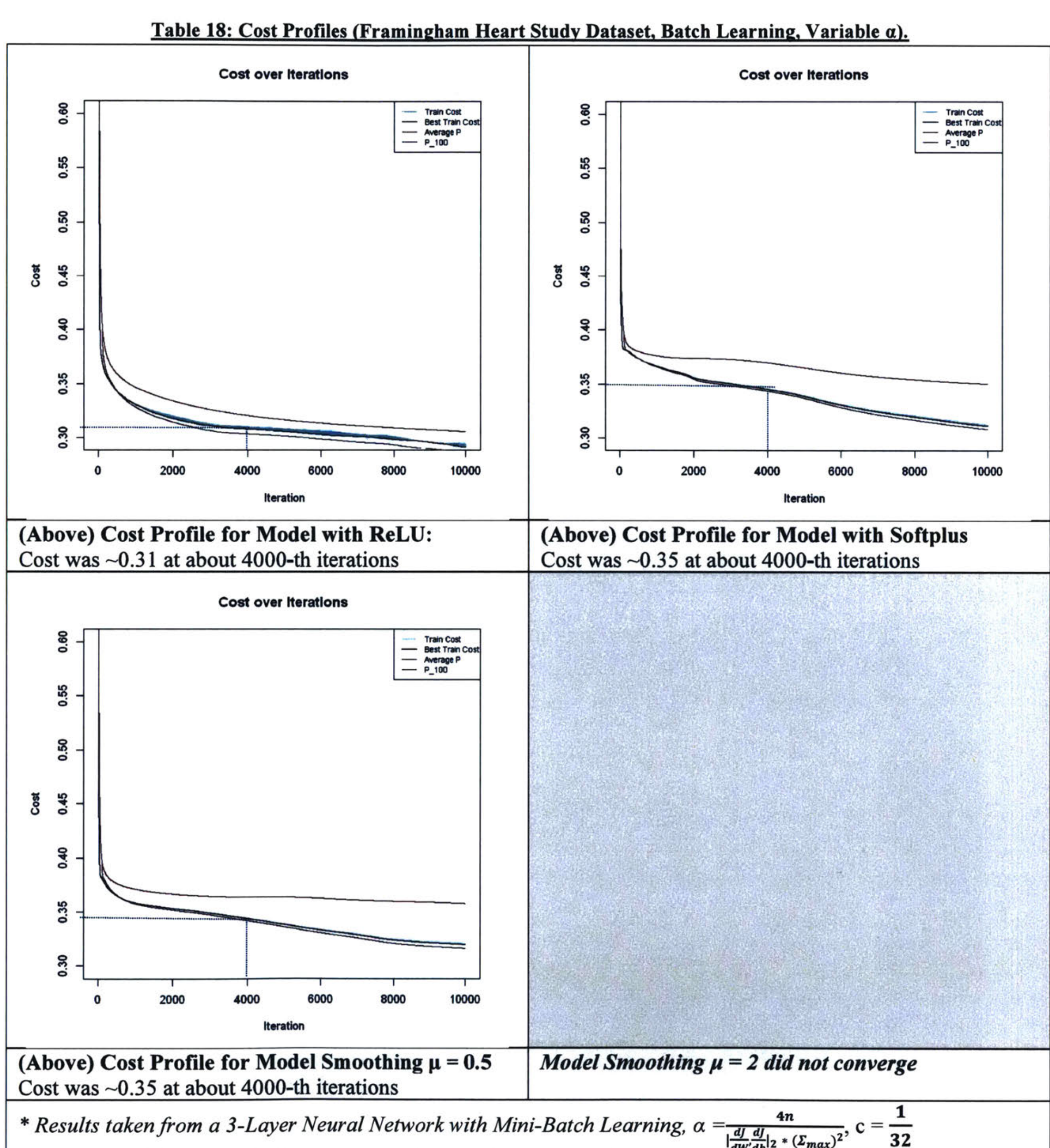
Table 17: Deep Networks - Minimum Cost Achieved on Training Set (Batch Learning, Variable α Models).

		Minimum Cost Achieved on the Training Set (Batch Learning)			
Dataset	**Learning Rate, α**	**ReLU**	**Softplus**	**Smoothing, $\mu = 0.5$**	**Smoothing, $\mu = 2$**
Framingham Heart Study	$\frac{4n}{\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2 * (\Sigma_{max})^2}$, $c = \frac{1}{32}$	**0.291**	0.311	0.319	*NA (Unable to Converge)*
	$\frac{4n}{\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2 * (\Sigma_{max})^2}$, $c = \frac{1}{64}$	0.295	0.323	0.329	0.346
Bank Churn Modelling	$\frac{4n}{\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2 * (\Sigma_{max})^2}$, $c = 1$	*NA (Unable to Converge)*	0.484	*NA (Unable to Converge)*	0.370
	$\frac{4n}{\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2 * (\Sigma_{max})^2}$, $c = \frac{1}{4}$	*NA (Unable to Converge)*	0.334	0.356	0.339
	$\frac{4n}{\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2 * (\Sigma_{max})^2}$, $c = \frac{1}{32}$	**0.319**	0.325	0.329	*NA (Unable to Converge)*
German Credit Risk	$\frac{4n}{\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2 * (\Sigma_{max})^2}$, $c = \frac{1}{32}$	**0.300**	0.415	0.413	0.421
	$\frac{4n}{\lvert\frac{dJ}{dW},\frac{dJ}{db}\rvert_2 * (\Sigma_{max})^2}$, $c = \frac{1}{64}$	0.464	0.479	0.488	0.465

* **Blue** indicates best performing model among models with the same set of hyperparameters (i.e., for each row); **Green** indicates model with comparable cost performance with best performing model; **Red** indicates marked inferiority in cost performance against best performing model; **Bold** indicates best result for dataset.

Examining the cost profiles generated off the Framingham Heart Study dataset (for $c = \frac{1}{32}$, where ReLU, Softplus, and Smoothing $\mu = 0.5$ models produced their best cost results), it was observed that the rate of cost descent for the ReLU model was slightly faster than the models using the Softplus and Smoothing ($\mu = 0.5$) activation functions (See **Table 18**).

Table 18: Cost Profiles (Framingham Heart Study Dataset, Batch Learning, Variable α).



(Above) Cost Profile for Model with ReLU:
Cost was ~0.31 at about 4000-th iterations

(Above) Cost Profile for Model with Softplus
Cost was ~0.35 at about 4000-th iterations

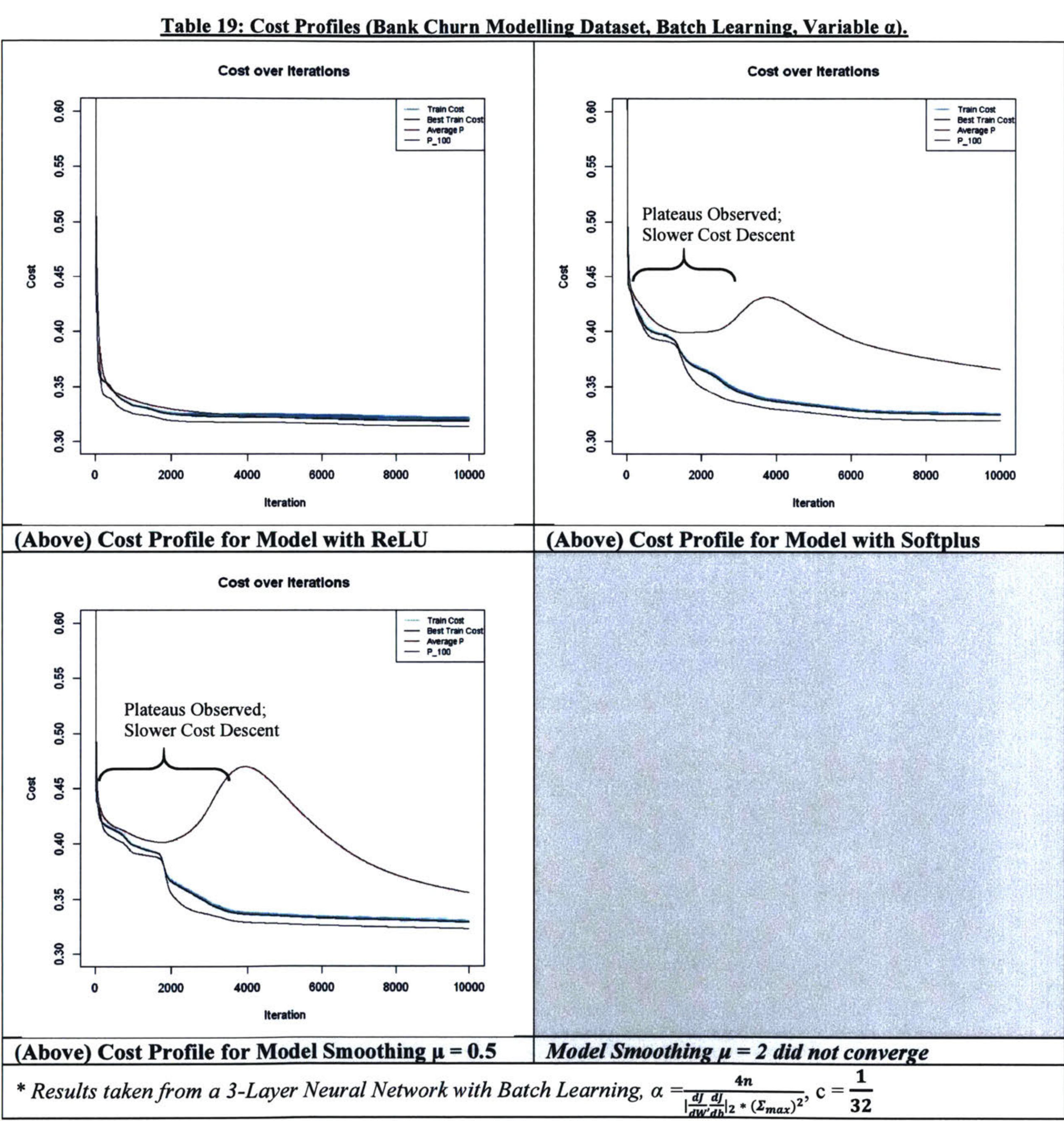
(Above) Cost Profile for Model Smoothing $\mu = 0.5$
Cost was ~0.35 at about 4000-th iterations

Model Smoothing $\mu = 2$ did not converge

* *Results taken from a 3-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{4n}{|\frac{dJ}{dW'}\frac{dJ}{db}|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{32}$

Likewise, for the Bank Churn Modelling dataset (for $c = \frac{1}{32}$, where similarly the ReLU, Softplus, and Smoothing $\mu = 0.5$ models produced their best cost results[6]), it was shown that rate of cost descent was noticeably slower for neural networks employing the Smoothing activation functions (See **Table 19**).

Table 19: Cost Profiles (Bank Churn Modelling Dataset, Batch Learning, Variable α).



(Above) Cost Profile for Model with ReLU

(Above) Cost Profile for Model with Softplus

(Above) Cost Profile for Model Smoothing $\mu = 0.5$

Model Smoothing $\mu = 2$ did not converge

* *Results taken from a 3-Layer Neural Network with Batch Learning,* $\alpha = \frac{4n}{|\frac{dJ}{dW}\frac{dJ}{db}|_2 * (\Sigma_{max})^2}$, $c = \frac{1}{32}$

---

[6] As an interesting side note, a bump was observed in the training costs associated with the averaged parameters for neural networks using the Smoothing activation function ($\mu = 0.5$ and $\mu = 1$). The investigation of how averaging parameters from past iterations is broadly proposed as an area for further study in Chapter 6.

III. Mini-Batch Learning with Fixed Learning Rates. Focusing on the learning factors of c = 400 and c = 600 (whereby each mode produced better results relative to c = 1 and c = 200),

- The neural networks using the ReLU activation function were able to achieved the lowest cost on the training set on both the Bank Churn Modelling and German Credit Risk datasets.

- On the Framingham Study dataset however, the neural networks using the Softplus and Smoothing (μ = 0.5) activation functions outperformed the ReLU activation function for c = 400 and c = 600 respectively (See **Table 20**).

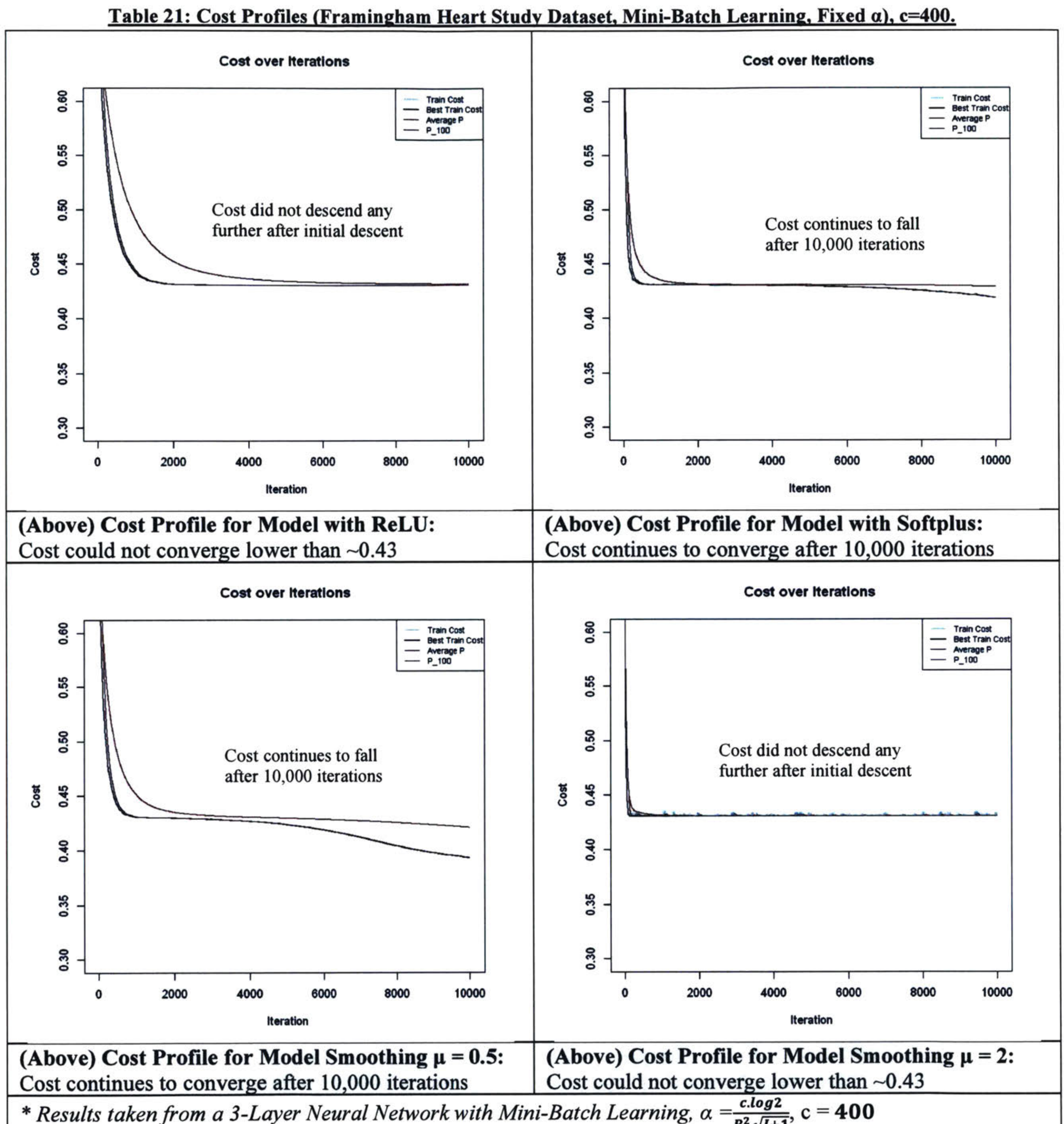
Table 20: Deep Networks - Minimum Cost Achieved on Training Set (Mini-Batch Learning, Fixed α Models).

		Minimum Cost Achieved on the Training Set (Batch Learning)			
Dataset	**Learning Rate, α**	**ReLU**	**Softplus**	**Smoothing, μ = 0.5**	**Smoothing, μ = 2**
Framingham Heart Study	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 1	0.669	0.588	0.642	0.485
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 200	0.431	0.430	0.424	0.431
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 400	0.431	0.418	0.393	0.430
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 600	0.431	0.394	**0.385**	0.429
Bank Churn Modelling	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 1	0.586	0.517	0.554	0.506
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 200	0.495	0.429	0.418	0.428
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 400	0.343	0.412	0.407	0.425
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 600	**0.333**	0.353	0.405	0.408
German Credit Risk	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 1	0.676	0.613	0.643	0.607
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 200	0.605	0.606	0.519	0.607
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 400	0.048	0.417	0.380	0.475
	$\frac{c.log2}{R^2.\sqrt{I+1}}$, c = 600	**0.003**	0.382	0.178	0.428

* **Blue** indicates best performing model among models with the same set of hyperparameters (i.e., for each row); **Green** indicates model with comparable cost performance with best performing model; **Red** indicates marked inferiority in cost performance against best performing model; **Bold** indicates best result for dataset.

Examining the cost profiles on the neural network models on the Framingham Heart Study dataset for the learning factors c = 400 and c = 600, it was observed that the models using the ReLU activation function and the Smoothing (μ = 2) activation functions encountered a plateau in their respective training processes (See **Table 21, Table 22**).

Table 21: Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Fixed α), c=400.



(Above) Cost Profile for Model with ReLU: Cost could not converge lower than ~0.43

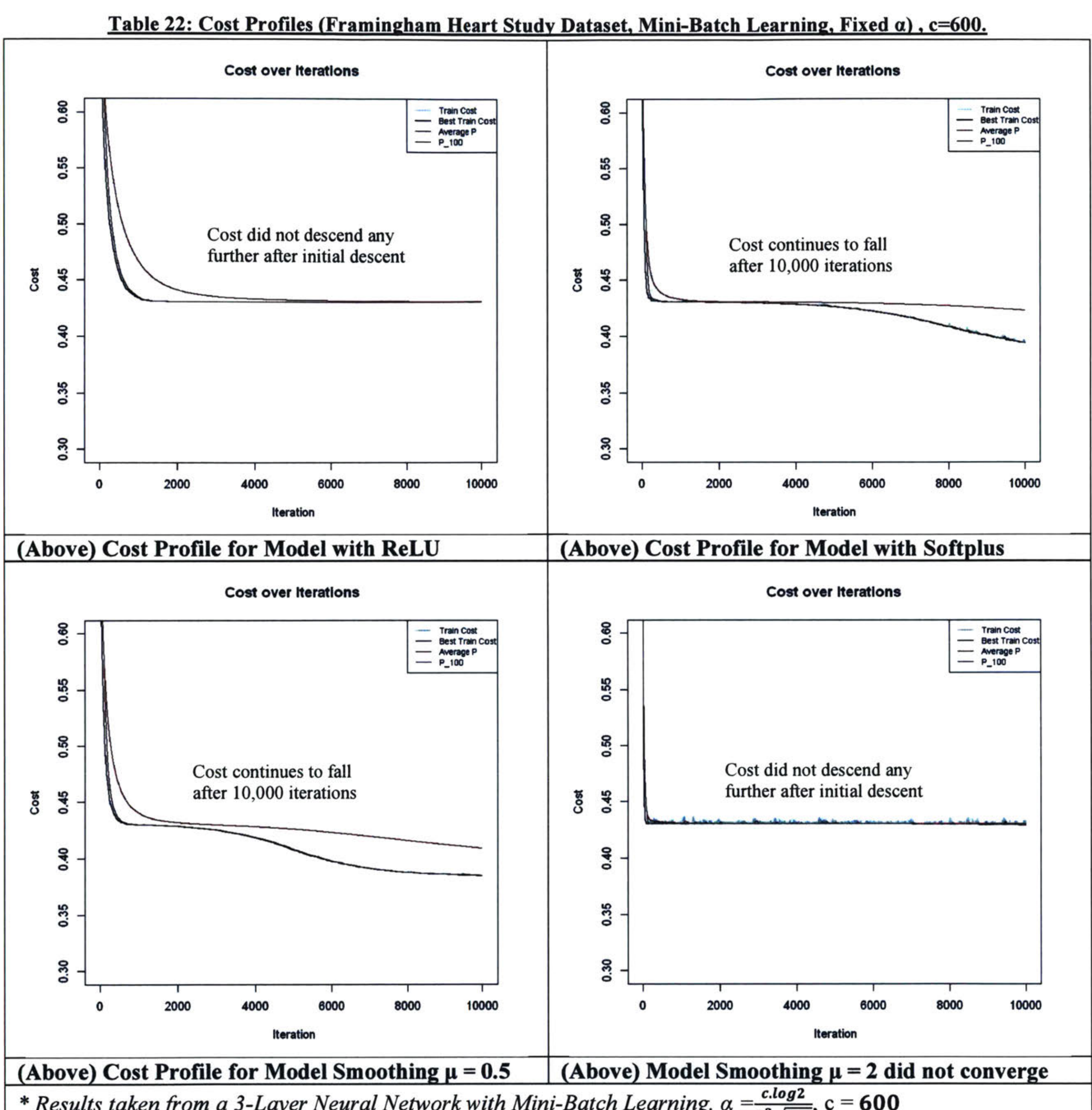
(Above) Cost Profile for Model with Softplus: Cost continues to converge after 10,000 iterations

(Above) Cost Profile for Model Smoothing μ = 0.5: Cost continues to converge after 10,000 iterations

(Above) Cost Profile for Model Smoothing μ = 2: Cost could not converge lower than ~0.43

** Results taken from a 3-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{c.log2}{R^2.\sqrt{I+1}}$, c = **400**

Table 22: Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Fixed α) , c=600.



(Above) Cost Profile for Model with ReLU	(Above) Cost Profile for Model with Softplus
(Above) Cost Profile for Model Smoothing $\mu = 0.5$	(Above) Model Smoothing $\mu = 2$ did not converge

* *Results taken from a 3-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{c.log2}{R^2.\sqrt{I+1}}$, c = **600**

IV. Mini-Batch Learning with Variable Learning Rates. Using mini-batch learning with variable learning rates,

- The neural network with the Softplus and Smoothing (μ = 0.5) activation functions achieved the lowest cost on the training sets of the Framingham Heart Study and Bank Churn Modelling datasets respectively.

- Nevertheless, the models using the ReLU functions was still able to achieved comparable training cost values on the Bank Chun Modelling dataset.

- Neural Networks using the ReLU activation function achieved, by far, the lowest cost on the training set of the German Credit Risk dataset (**Table 23**).

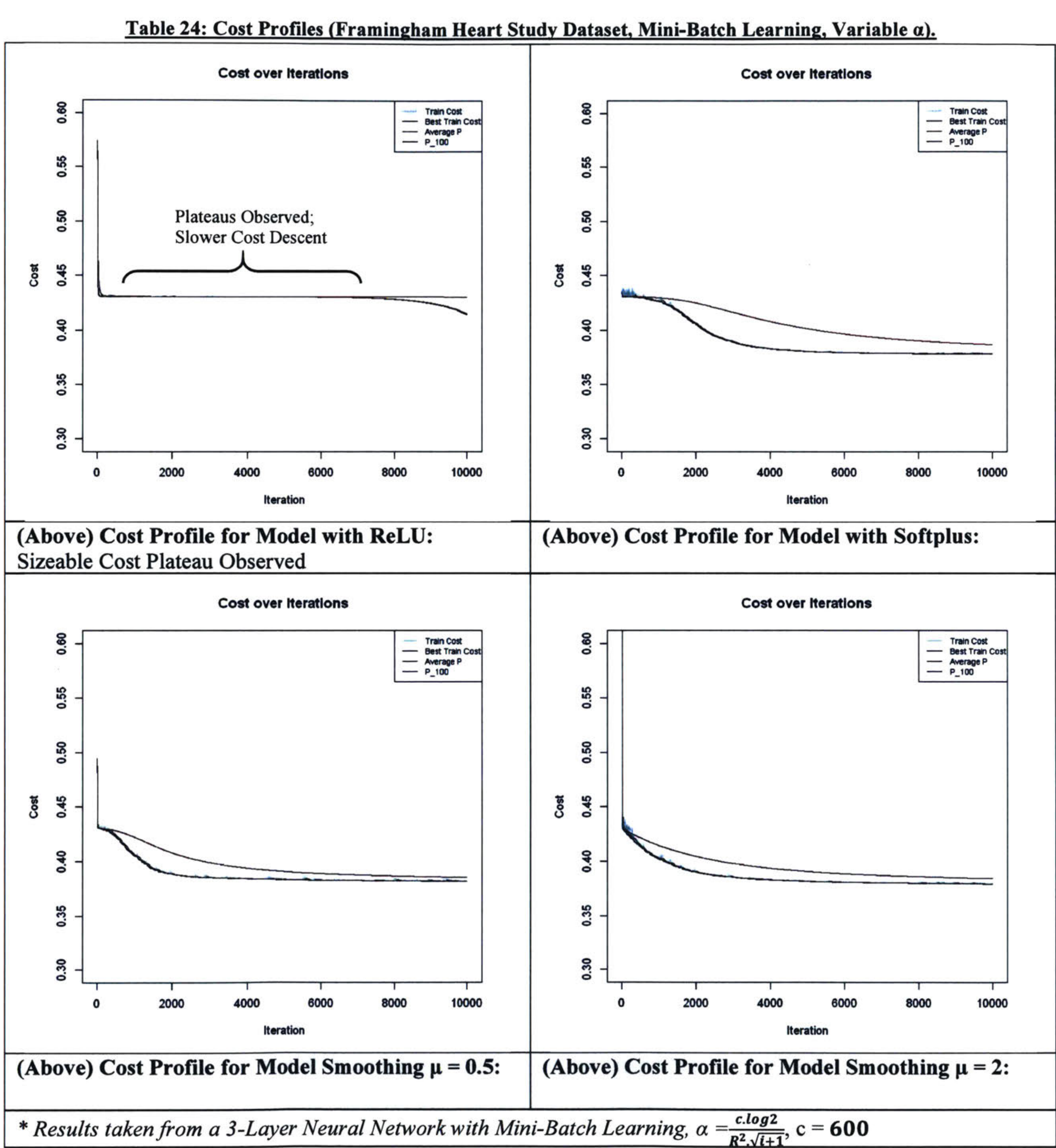
Table 23: Deep Networks – Minimum Cost Achieved on Training Set (Mini-Batch Learning, Variable α Models).

		Minimum Cost Achieved on the Training Set (Batch Learning)			
Dataset	**Learning Rate, α**	**ReLU**	**Softplus**	**Smoothing, μ = 0.5**	**Smoothing, μ = 2**
Framingham Heart Study	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 1	0.647	0.532	0.607	0.447
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 200	0.431	0.422	0.394	0.431
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 400	0.431	0.384	0.383	0.386
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 600	0.415	**0.378**	0.382	0.379
Bank Churn Modelling	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 1	0.543	0.507	0.519	0.506
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 200	0.341	0.363	0.358	0.426
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 400	0.339	0.336	0.329	0.505
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 600	0.337	0.345	**0.332**	0.365
German Credit Risk	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 1	0.693	0.687	0.691	0.674
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 200	0.055	0.437	0.402	0.440
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 400	0.002	0.427	0.189	0.432
	$\frac{c.log2}{R^2.\sqrt{i+1}}$, c = 600	**0.001**	0.388	NA (Unable to Converge)	0.432

* **Blue** indicates best performing model among models with the same set of hyperparameters (i.e., for each row); **Green** indicates model with comparable cost performance with best performing model; **Red** indicates marked inferiority in cost performance against best performing model; **Bold** indicates best result for dataset.

Examining the cost profiles generated off the Framingham Heart Study, it was revealed that the training process using a neural network with the ReLU activation function was characterized by a plateau in the cost profile (See **Table 24**).

Table 24: Cost Profiles (Framingham Heart Study Dataset, Mini-Batch Learning, Variable α).



(Above) Cost Profile for Model with ReLU: Sizeable Cost Plateau Observed

(Above) Cost Profile for Model with Softplus:

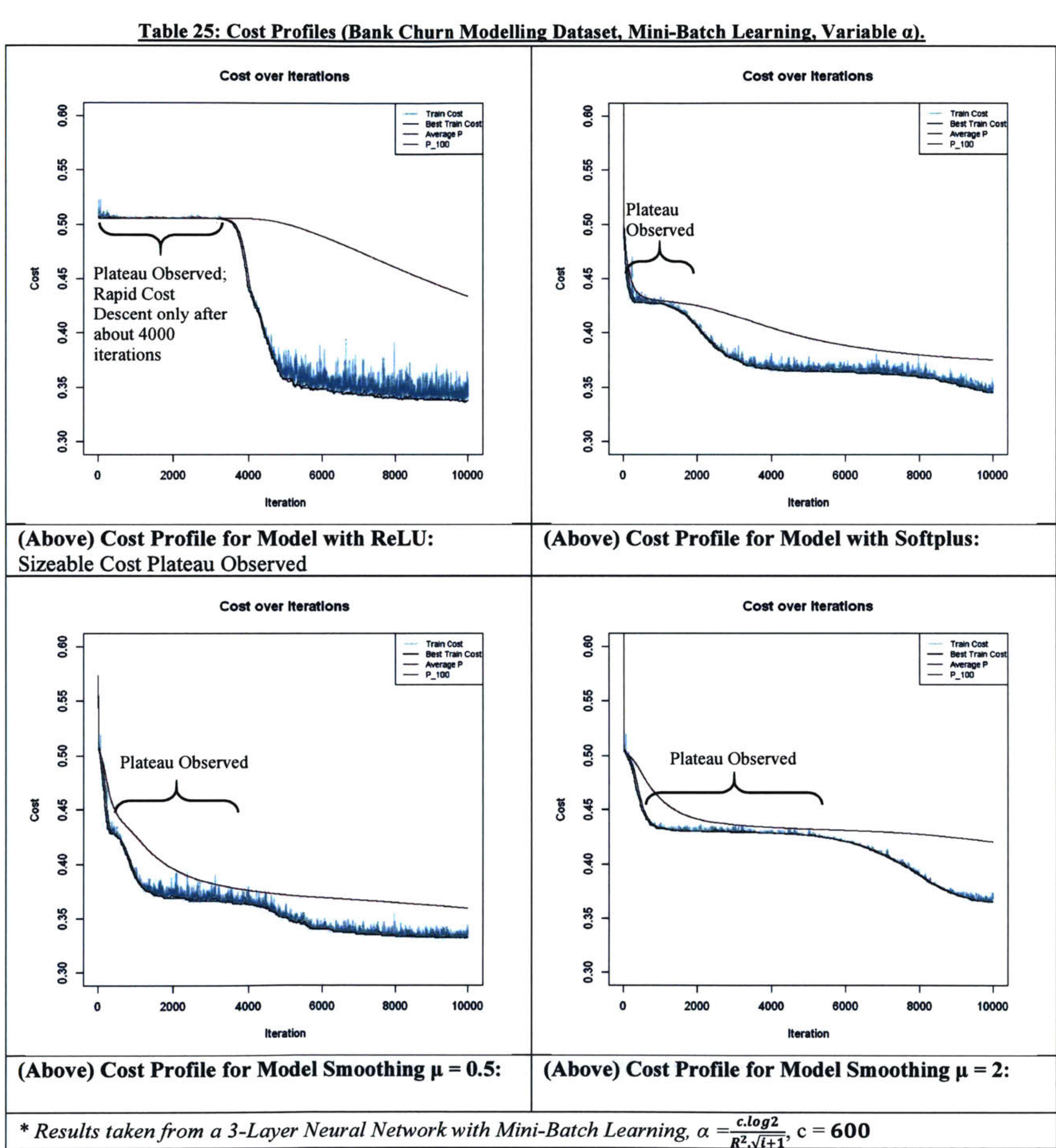
(Above) Cost Profile for Model Smoothing μ = 0.5:

(Above) Cost Profile for Model Smoothing μ = 2:

* *Results taken from a 3-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{c.log2}{R^2.\sqrt{i+1}}$, c = **600**

On the Bank Churn Modelling dataset, a sizeable cost plateau in the initial iterations of the training process can be observed for the neural network using the ReLU activation function. While plateaus were also observed for other neural networks, it is noteworthy to point out that plateau for the ReLU-activated network occurred at a much higher cost value (See **Table 25**).

Table 25: Cost Profiles (Bank Churn Modelling Dataset, Mini-Batch Learning, Variable α).



(Above) Cost Profile for Model with ReLU: Sizeable Cost Plateau Observed

(Above) Cost Profile for Model with Softplus:

(Above) Cost Profile for Model Smoothing μ = 0.5:

(Above) Cost Profile for Model Smoothing μ = 2:

* *Results taken from a 3-Layer Neural Network with Mini-Batch Learning,* $\alpha = \frac{c.log2}{R^2.\sqrt{i+1}}$, c = **600**

4.2 Computational Time Across Activation Functions

Beyond analyzing relative cost performances, we further consider the computational time associated with each activation function. **Table 26** compares the time[7] required to train a neural network model for 10,000 iterations using the various activation functions.

Table 26: Average Computational Time for 2-Layer, 3-Layer Neural Networks.

Model	Dataset Average Computational Time for 10,000 Iterations (Mins)		
	Framingham Heart Study	**Bank Churn Modelling**	**German Credit Risk**
2-Layer Neural Network			
ReLU	**3.48**	**4.13**	**2.46**
Softplus	6.62	8.94	5.13
Smoothing $\mu = 0.5$	6.84	8.72	4.49
Smoothing $\mu = 2$	6.77	8.86	4.15
3-Layer Neural Network			
ReLU	**5.04**	**6.44**	**5.11**
Softplus	12.04	12.67	10.62
Smoothing $\mu = 0.5$	11.60	13.38	10.98
Smoothing $\mu = 2$	10.31	12.72	8.02

It was shown that for all cases, neural network using the ReLU activation function required about half the time to complete 10,000 iterations as compared to neural networks using any of the Smoothing activation functions. As such, it can be seen that the ReLU activation function is computationally faster.

As covered in the literature review, the superiority of the ReLU activation function in terms of computational efficiency might be attributed to the sparse nature of the model: During forward propagation, the ReLU activation function reduces all negative values within a input matrix into zero, thus reducing the resources required to perform further computations onto the output

[7] The computational time required were measured off a computing device with an Intel i7-8550U CPU @ 1.80GHz with 16 GB RAM on a 64 bit Windows 10 operating system (No GPU),

matrices. Concurrently, during backward propagation, the calculation of the $\frac{da}{dz}$ component of the cost gradient only requires a simple assignment of the values 0 or 1 (depending on whether the respective input matrix value is positive or negative) as opposed to working with the logarithmic and exponential functions. The simplicity of the mathematical operations involved in the forward and backward propagations when using a ReLU activation function theoretically allows neural networks to complete their computations within a shorter time.

Chapter 5: Assessing the Suitability of Neural Networks in Business Applications

This chapter performs an additional analysis: The performance and suitability of neural networks within the context of business applications will be evaluated.

5.1 Benchmarking Logistic Regression for Business Applications

As a machine learning technique for business applications, logistic regression serves as an appropriate benchmark for the evaluation of neural networks. Critically, a logistic regression can be interpreted as a "1-layer feedforward neural network". In using a logistic regression model as benchmark therefore, we ascertain if there is value in introducing additional hidden layers into a 1-layer feedforward neural network for common business applications.

Importantly, against substantial development in the deep learning field, logistic regression remains a popular machine technique used in modelling common business scenarios. The reasons are as follows:

- Modelling Efficiency. Unlike logistic regression, the deployment of a neural network requires a deliberate hyper-parameterization process. Optimizing the performance of a neural network entails exploring a range of values for the numerous hyperparameters such as learning rates, type of activation functions, number of iterations (to prevent over or undertraining while ensuring that we economize the computational load), the number of hidden layers, and the number of nodes in each hidden layer. Comparatively, the development of a logistic regression model is relatively simpler and less time consuming.

- Computational Simplicity. A logistic regression model is computationally simpler. This leads to shorter training and computational time when deploying a logistic regression model. Case in point, the training of a logistic regression models for each of the 3 datasets used in our study took less than 10 seconds in each instance. This contrasts favorably against the time required to train a neural network model (See **Table 27**): For the 3 datasets used in the study, the average time required to train a 2-layer neural network lies between

4 to 6 minutes[8], while the average time required to train a 3-layer neural network lies between 8 to 12 minutes. The computational time increases by an order of 2 to 3 when we use a 2 or 3-layer neural network model instead of a logistic regression model.

Table 27: Average Computational Time for 1-Layer, 2-Layer Neural Networks.

Model	Dataset Average Computational Time for 10,000 Iterations (Mins)		
	Framingham Heart Study	**Bank Churn Modelling**	**German Credit Risk**
2-Layer Neural Network	5.93	7.68	4.09
3-Layer Neural Network	9.77	11.28	8.55

- Performance. Fundamentally, the inclusion of hidden layers within a neural network (which entails the use of nonlinear activation functions) allows a neural network model to capture nonlinear patterns. This empowers the model to formulate complex, nonlinear relationships commonly observed in unstructured data. Nevertheless, despite being a linear model, logistic regression has shown to be robust even when managing certain unstructured data. (Particularly, logistic regression is known to achieve an accuracy of more than 90% on the MNIST image recognition dataset.) Coupled with the fact that a logistic regression model is computationally simpler, a logistic regression model might be deemed as "good enough" when applied to business applications (whereby we commonly deal with structured and/or binary data).

- Interpretability. In the case of using logistic regression model for a binary classification problem, the coefficient assigned to an independent variable can be readily interpreted as the increase in odds (i.e., the relative probability of a positive observation versus a negative observation) given a unit increase in the value of the independent variable, assuming all other independent variables remain constant.

  Particularly for business applications, the interpretability provided by a logistic regression model is critical towards convincing business owners and stakeholders towards justifying business decisions. Comparatively, interpreting and making sense of nodal parameters across the hidden layers of a neural network might entail a more labored and involved

[8] The computational time required were measured off a computing device with an Intel i7-8550U CPU @ 1.80GHz with 16 GB RAM on a 64 bit Windows 10 operating system (No GPU),

effort. Indeed, deep learning networks have found wider applications in engineering applications such as image recognition and machine translation, whereby the interpretability of model parameters is less critical than the final accuracy provided by the model.

To benchmark the performance of the neural networks developed in this study against that of a logistic regression model, a code for a logistic regression network was developed (See **Appendix B**[9]). This logistic regression network was then applied to the 3 datasets used in the study and the results analyzed in the next section.

[9] The logistics regression network was able to reproduce outputs generated by a Generalized Linear Model (GLM) function in base R.

5.2 Model Performance: Logistic Regression versus Neural Network

Table 28 presents performance metrics – Area Under the Receiving Operating Characteristic Curve (AUC) and minimum costs achieved (for both the training and testing sets, at the end of 10000 iterations), using a shallow neural network model, a deep neural network (a 3-layer neural network model), and a logistic neural network. Of note, the performance metrics presented for the shallow neural network and the deep neural network are indicative of neural networks with hyperparameters that gave the near-lowest cost and near-highest AUC values, i.e., neural networks using hyperparameters that produced inferior performance metrics are not shown here.

Table 28: Minimum Cost Performance for Logistic Regression and Neural Network Models.

Model	Observation Set	Metric	Dataset and Minimum Test Cost Achieved		
			Framingham Heart Study	**Bank Churn Modelling**	**German Credit Risk**
Logistic Regression *(1-Layer Neural Network)*	Training Set	Minimum Cost	0.3772843	0.4287479	0.4283163
		AUC	0.7465124	0.7658065	0.8472478
	Testing Set	Corresponding Test Cost	0.3761477	0.4268085	0.5384435
		AUC	0.7145916	0.7759654	0.7596154
Shallow Neural Network *(2-Layer Neural Network)*	Training Set	Minimum Cost	0.3736599	0.3185538	0.3918045
		AUC	0.7536119	0.8811262	0.8718605
	Testing Set	Corresponding Test Cost	0.3752478	0.3517152	0.534829
		AUC	0.7205689	0.8542265	0.7717651
	Hyperparameters		ReLU, Minibatch Training, Variable Learning Rate x300	Softplus, Batch Learning, Fixed Learning Rate x 2	Smoothing μ = 0.5, Minibatch Learning, Variable Learning Rate x 100
Deep Neural Network *(3-Layer Neural Network)*	Training Set	Minimum Cost	0.3792733	0.331755	0.4269322
		AUC	0.7453476	0.8674127	0.8525287
	Testing Set	Corresponding Test Cost	0.3749311	0.3431166	0.5394947
		AUC	0.7170619	0.8529271	0.7647585
	Hyperparameters		Smoothing μ = 2, Minibatch Training, Variable Learning Rate x 600	Smoothing μ = 0.5, Minibatch Training, Variable Learning Rate x 300	Softplus, Minibatch Training, Variable Learning Rate x 400

For all datasets, the neural network models were able to achieve a superior AUC and lower cost values relative to the logistic regression model. The degree of improvement, however, varies across datasets.

Notably, the use of a neural network on the Bank Churn Modelling dataset produced mark improvements in both AUC and costs values: (1) The AUC value for the training and testing sets were 0.766 and 0.776 respectively using the logistic regression model; The same values for a 2-layer neural network were 0.881 and 0.854 respectively. (2) The training set and testing set cost values using a logistic regression network were 0.429 and 0.427 respectively; The same values of a 2-layer neural network were 0.319 and 0.352 respectively.

The improvements were less significant for the German Credit Risk dataset: (1) The AUC value for the training and testing sets were 0.847 and 0.760 respectively using the logistic regression model; The same values for a 2-layer neural network were 0.872 and 0.772 respectively. (2) The training set and testing set cost values using a logistic regression network were 0.428 and 0.538 respectively; The same values of a 2-layer neural network were 0.392 and 0.535 respectively.

Compared to the neural network models used on the other datasets, the neural network model gave negligible improvements for the Framingham Heart Study dataset: (1) The AUC value for the training and testing sets were 0.741 and 0.715 respectively using the logistic regression model; The same values for a 2-layer neural network were 0.753 and 0.721 respectively. (2) The training set and testing set cost values using a logistic regression network were 0.377 and 0.376 respectively; The same values of a 2-layer neural network were 0.374 and 0.375 respectively.

Comparing the performances between the 2-layer neural network and a 3-layer neural network, the latter model produced slightly inferior AUC and cost values across all datasets. However, it was opined that the degradation in performance metrics from the addition of a hidden layer into the network were not significant enough for us to conclude that a 3-layer neural network model is definitely inferior to a 2-layer neural network model.

Importantly, the results suggest that the structural properties of the independent variables of the dataset might play an important role in determining if AUC and cost performances can improve using a neural network model. Theoretically, the independent variables in the Bank Churn

Modelling dataset might hold complex, nonlinear relationships between them – relationships that a neural network was able to capture while a linear logistic regression model was not able to. This accounts for the marked improvement in performance when applying a neural network model onto the Bank Churn Modelling dataset. Conversely, the independent variables in the Framingham Heart Study dataset and the German Credit Risk dataset might not have the same degree on nonlinearities among them. This might account for the lack of improvement in performance when implementing a neural network model onto the said datasets.

5.3 Considerations Beyond AUC and Cost Metrics

In spite of the observed improvements in both AUC and costs metrics, it is premature to recommend the use of a neural network over a logistic regression model within the finite experimentation encompassed by this study:

- Deploying a neural network capable of registering a superior improvement requires an extensive and tedious process of using the correct hyperparameters (learning rates, type of activation functions, batch size, number of iterations, the number of hidden layers and the number of nodes in each hidden layer). In comparison, the use of a logistic regression is significantly simpler.

- As highlighted, the computational time required to train a neural network model is significantly longer for a neural network model as compared to the training of a logistic regression model. Depending on the time-sensitivity and nature of the business application as well as the computational power available, it might or might not be appropriate to recommend a neural network model over a logistic regression model.

- The interpretability of the model parameters is comparatively more intuitive for a logistic regression model as compared to a neural network model. The interpretation process for the latter, in particular, is further convoluted by the sheer number of possible combinations for the hyperparameters (1) number of hidden layers and (2) number of nodes in each hidden layers.

- The degree of improvement that can be yielded through using a neural network model might depend on the structural properties of the independent variables of the dataset, i.e., whether the independent variables between the independent variables exhibit complex, nonlinear relationships them. The utility of deploying a neural network over a simpler machine learning technique (i.e., logistic regression model) therefore hinges on the nature of the dataset being used.

Chapter 6: Summary, Way Ahead, and Conclusion

6.1 Key Experimental Findings

For most cases, neural networks using the ReLU activation function were able to fit the observations across all datasets with comparable, if not better, cost results when compared to neural networks using the Smoothing activation functions. Coupled with the observation that (1) neural networks using the ReLU activation function take a shorter amount of time to train and (2) neural networks using a Smoothing activation function could not consistently produce better, if not comparable, results for most cases, we cannot recommend the Smoothing activation functions over the ReLU activation function at this juncture.

Nevertheless, in noting that a Smoothing activation function can indeed outperform the ReLU activation function under certain hyper-parameterization and for certain dataset (i.e., the 3-layer neural network models with mini-batch learning, on the Framingham Heart Study dataset), further efforts might be invested into profiling the type of data that would be suitable for neural networks using a Smoothing activation function.

In the broader scheme of things, it is opined that we remain cautious towards the application of neural networks for business applications (particularly those encompassed within the scope of this study). While neural networks have proven to be capable of improving predictive / classification capabilities, the interpretability, computational efficiency, and hyper-parameterization efficiency of neural networks remain inferior to simpler Machine Learning techniques such as logistic regression.

6.2 Areas for Further Study

It is recommended that the following areas receives further investigation.

6.2.1 Activation Function Selection Through Examination of Independent Variables

As highlighted, the Smoothing activation function was observed to outperform the ReLU activation function for certain datasets, and under certain hyper-parameterization. It is thus recommended that a further study characterizing the structural properties of the independent

variables of the datasets be conducted, towards selecting the optimal choice of activation function (and/or hyper parameters, including the value of μ when using a Smoothing activation function).

6.2.2 Averaging Parameters over Iterations for Optimization

The feedforward neural network code used for the study included a module to compute the training and testing set costs associated with the average of model parameters for the past 100 iterations. While originally intended to register averaged values off fluctuating cost profiles, the module has yielded unexpected results – for a very narrow set of hyperparameters, the cost performances associated with the average of model parameters for the past 100 iterations is more optimal than the cost performances associated with actual model parameters for each respective iteration.

Table 29 and **Table 30** highlights two such instances whereby the cost values associated with the average of all model parameters for the past 100 iterations performs better than the actual cost value for each respective iteration.

Table 29: First Example of Averaged Parameters Outperformed Actual Parameters for Iteration.

Hyperparameters: (Bank Churn Modelling Dataset, Smoothing μ = 0.5, Batch Learning, Fixed Learning Rate x 1)

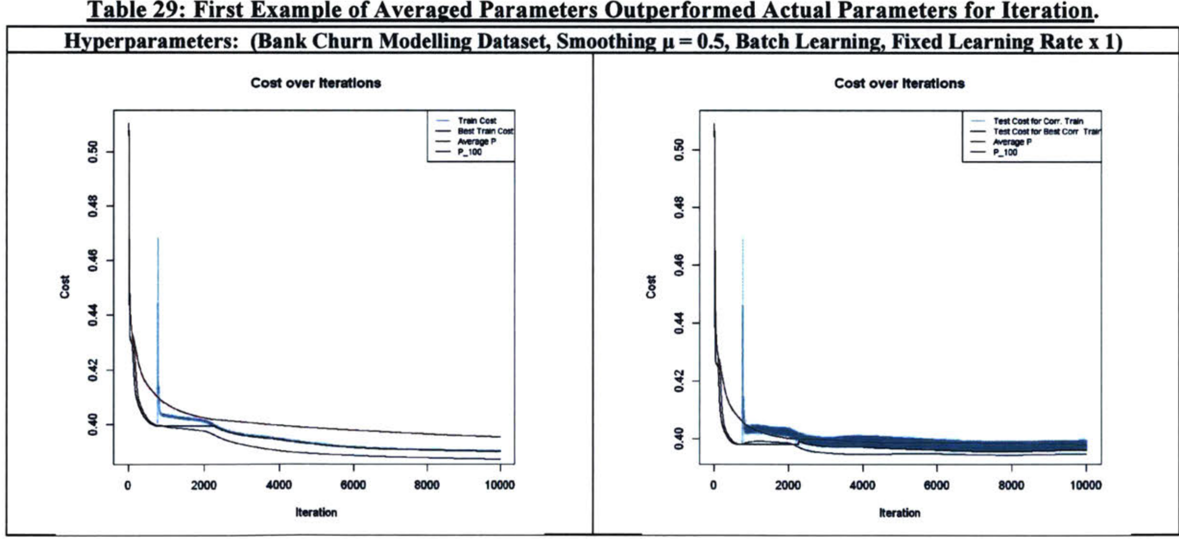
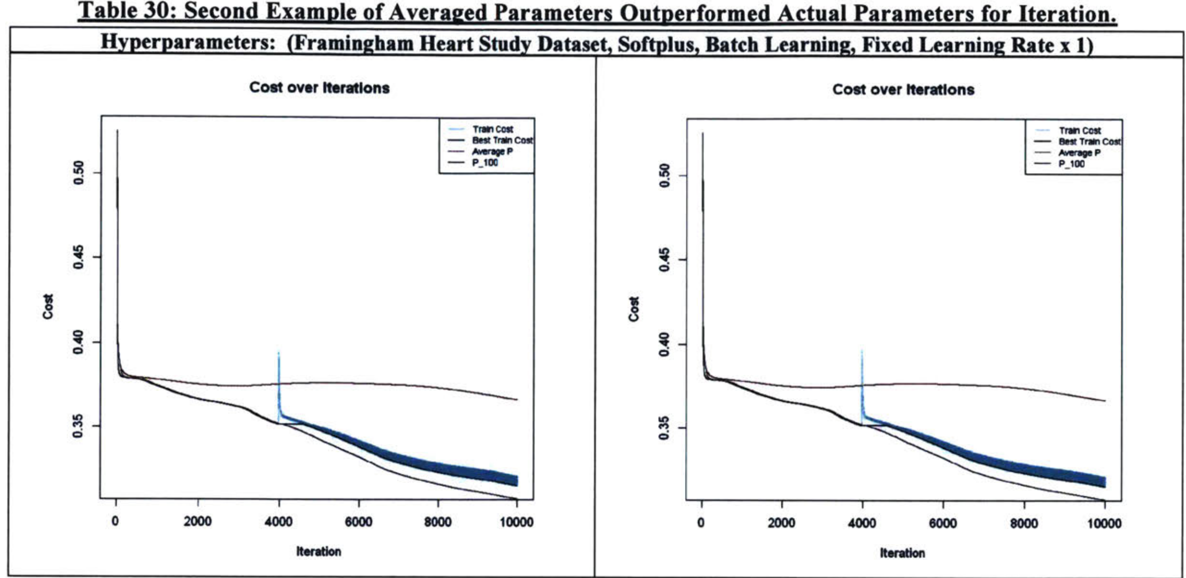


Table 30: Second Example of Averaged Parameters Outperformed Actual Parameters for Iteration.

Hyperparameters: (Framingham Heart Study Dataset, Softplus, Batch Learning, Fixed Learning Rate x 1)



It is noteworthy to point out that the phenomenon could only be observed on the Framingham Heart Study and the Bank Churn Modelling datasets under batch learning conditions, suggesting that the structural properties of the independent variables of the dataset and the batch size might play a part in yielding such an observation.

Ultimately however, the underlying explanation for the observation could not be established within the finite scope of this study. It is recommended that we invest further effort into the investigation of said observation, towards possibly unveiling a novel approach in optimizing model parameters.

This page is intentionally left blank

Appendix A: Code for Feedforward Neural Network

(Setup) Call Libraries for Functions to be Used

In [ ]:

```
# Call on the library caret for data splitting functions
library(caret)

# Call on the library ROCR For ROC/AUC analysis
library(ROCR)
```

(1) DATA PREPARATION AND STANDARDIZATION

The Data presentation and Standardization tasks include:

(a) Import Data / Modify Structure of Data.
Beyond importing the data, we examine the dataset for categorical values and convert them in a binary form (i.e. of 1s and 0s). This is the form compatible with neural nets.

(b) Split Data into *Train* and *Test* Set.
In this instance, we adopt a 75% / 25% split for the train and test set.

(c) Split *Train* and _Test_Data into *Input* and *Label* Sub-Sets.
For the Train and Test set, we separate the independent variables from the true labels.

(d) Standardize Data.
We standardize the dataset (i.e for train set, deduct each value in a particular column by the mean of all values for that column, and dividing the resultant value by the standard deviation of all values for that column)

In [ ]:

```
## (1)(a) Import Data / Modify Structure of Data
data = read.csv("framingham.csv")

# Examine data structure
str(data)
head(data)

# What is the naive model accuracy?
print(paste0("Naive Model Accuracy: ", 100*round(1 - (sum(data$TenYearCHD)/
nrow(data)), 4), "%"))
```

In [ ]:

```
## We note that the independent variable "Education" is categorical
# We need to convert these into binary dummy variables (one for each factor
level) to make it compatible with neural nets
# To do this, we use the command "model.matrix" on the education variable
# Note: We convert the dataset object into dataframe to make the dataset ea
sier to manipulate

education_variables = as.data.frame(model.matrix(~ 0 + education, data = da
ta))
head(education_variables)
```

In [ ]:

```
## We then replace the education variable with the binary dummy variables

# Obtain index of columns before/after the education column
index_before = c(1:(which(colnames(data)=="education")-1))
index_after =  c((which(colnames(data)=="education")+1):ncol(data))

# We do a cbind or column bind to get back the original dataset in the form
we want
# (i.e. With all categorical data converted into dummy variables)
data = cbind(data[,c(index_before)],education_variables,data[,c(index_after
)])
head(data)
```

In [ ]:

```
# (1)(b) Split Data Set in Train and Test Set
set.seed(15071)
split = createDataPartition(data$TenYearCHD, p = 0.75, list = FALSE)

train = data[split,]
test = data[-split,]

head(train,6)
head(test,6)

# Examine dimensions of Train / Test data
cat("Number of Training Examples in Train Set:", dim(train)[1])
cat("\nNumber of Training Examples in Test Set:", dim(test)[1])
```

In [ ]:

```
#(1)(c) Split Data into Independent Variables and Labels
# This is because we only feed the independent variables into the network
# We will isolate the actual label column (Column name of "TenYearCHD") sep
arately
# Thereafter, we convert all to matrixes with as.matrix

train_input = as.matrix(train[,-which(colnames(train)=="TenYearCHD")])
train_labels = as.matrix(train[,which(colnames(train)=="TenYearCHD")])

test_input = as.matrix(test[,-which(colnames(test)=="TenYearCHD")])
test_labels = as.matrix(test[,which(colnames(test)=="TenYearCHD")])
```

In [ ]:

```
# 1(d) *Data Standardization
# We convert the training data into a standard normal distribution using th
e mean and variance

# train mean is the vector of means across the 17 input variables
train_mean = colMeans(x = train_input, na.rm = FALSE)
train_mean

# train_variance is the vector of SAMPLE variance across the 17 input varia
bles
train_variance = apply(train_input, 2, var)
train_variance

# We standardize the input datasets
# We use sweep to broadcast in R
# MARGIN = 2 means we "sweep" across the columns (every value is deducted b
y column mean)

# First, we deduct away the mean
train_input = sweep(train_input, MARGIN = 2, train_mean, "-")
test_input = sweep(test_input, MARGIN = 2, train_mean, "-")

# Next, we divide by the sample std dev
train_input = sweep(train_input, MARGIN = 2, sqrt(train_variance), "/")
test_input = sweep(test_input, MARGIN = 2, sqrt(train_variance), "/")

head(train_input)
head(test_input)
```

(2) Feeder Modules

The feeder modules are integrated in (3) to give the complete, integrated feedforward neural network function,

(a) Initialize Correct Number of Parameters for Model

(i) Extract n_x (# of Independent Variables) and n_y (# of Output Variables) from Train Data.
(ii) Initialize Parameters.__

(b) Create Modules to Perform Forward Propagation

(i) Forward Propagation Part 1: Linear Multiplication Function
(ii) With 2(b)(i), perform Forward Propagation (Linear Multiplication + Activation)
(iii) Implement 2(b)(ii) across all layers

(c) Create Module to Compute Cost

(i) Compute Cost

(d) Create Modules to Perform Backward Propagation

(i) Back Propagation Part 1: Backward Linear Multiplication Function
(ii) With 2(d)(i), perform Backward Propagation (Backward Linear Multiplication + Activation) across all layers
(iii) Implement 2(d)(ii) across all layers

(e) Create Modules to Update Parameters and Create Batch for Mini-Batch Training

(i) Update Parameters
(ii) Create Mini-Batch for current iteration

(f) Create Module Predict Labels using the Final, Optimized parameters

(i) Predict Labels

In [ ]:

```
## (2)(a)(i) Extract n_x (number of independent variables) and n_y (number
 of output variables) from train data
# This function measures:
# The dimensions of the training examples "n_x" (i.e. number of independent
variables)
# The dimensions of the output layer, "n_y" (= 1 for logistic regression /
 classification problem)

First_Last_Layers_Dimensions = function(X, Y){

    n_x = dim(X)[2]
    n_y = dim(Y)[2]

    first_last_layers_dimensions = list("n_x" = n_x, "n_y" = n_y)

    cat("\ndimension of independent variables in X is",n_x)
    cat("\ndimension of output in Y is", n_y)

    return(first_last_layers_dimensions)
}
```

In [ ]:

```
## (2)(a)(ii) "Initialize_Parameters" creates matrixes of parameters (Wl, b
l) of randomized values for each layer
# Using the rnorm function, we generate random numbers from the distributio
n N(0,1)

Initialize_Parameters = function(first_last_layers_dimensions, hidden_layer
s_dimensions){

    # Initialize the list of parameters
    parameters = list()

    # Retrieve n_x and n_y from first_last_layers_dimensions
    n_x = first_last_layers_dimensions[["n_x"]]
    n_y = first_last_layers_dimensions[["n_y"]]

    # hidden_layers_dimensions is the vector of numbers indicating the numb
er of nodes in each of the layers
    # The seed will be set in the main integrated function

    # We first add the number of nodes in the zeroth-layer and final layer
 to the list
    layers_dimensions = c(n_x, hidden_layers_dimensions)
    layers_dimensions = c(layers_dimensions, n_y)

    # length(hidden_layers_dimensions) is thus the number of layers (exclud
ing zeroth layer, but including output layer)
    L = length(layers_dimensions)

    for (l in 1:(L-1)){

        # Create parameters W for layer l
        assign(paste0('W', l), matrix(rnorm(layers_dimensions[l] * layers_d
imensions[l+1], mean=0, sd=1) * 0.01,
                                      nrow = layers_dimensions[l],
                                      ncol = layers_dimensions[l+1]))

        # Create parameters b for layer l
        assign(paste0('b', l), matrix(0, nrow = 1, ncol = layers_dimensions
[l+1]))

        # Paste the created paramters onto the list "parameters"
        parameters[[paste0('W', l)]] = eval(parse(text = paste0('W', l)))
        parameters[[paste0('b', l)]] = eval(parse(text = paste0('b', l)))

    }

    return (parameters)
}
```

In [ ]:

```
## (2)(b)(i) Forward Propagation part 1 - Linear Multiplication Component
# This function performs the linear multiplication portion of a node

Forward_Linear_Multiplication = function (A_prev, W, b){

    cache=list()

    Z = sweep(A_prev %*% W, MARGIN = 2, b, FUN="+")

    # We store the following values in a cache
    # (because the backward propagation might require them, or data might b
e used for further analyses)
    cache[["Z"]] = Z
    cache[["A_prev"]] = A_prev
    cache[["W"]] = W
    cache[["b"]] = b

    return (cache)
}
```

In [ ]:

```
## (2)(b)(ii) Forward Propagation part 2 - Linear Multiplication + Activati
on Component
# This functions does exactly what a node does
# Note that it calls on the previous function "Forward_Linear_Multiplicatio
n"

Forward_Linear_and_Activation = function(A_prev, W, b, activation, miu){
    if (activation == "sigmoid"){

        cache = Forward_Linear_Multiplication(A_prev, W, b)
        Z = cache[["Z"]]

        A = 1/(1 + exp(-Z))

    } else if (activation == "relu"){

        cache = Forward_Linear_Multiplication(A_prev, W, b)
        Z = cache[["Z"]]

        A = Z * (Z > 0)

    } else if (activation == "smoothing"){

        cache = Forward_Linear_Multiplication(A_prev, W, b)
        Z = cache[["Z"]]

        A = miu * log(1 + exp(Z/miu))
    }

    cache[["A"]] = A

    return(cache)
}
```

In [ ]:

```
# (2)(b)(iii) Implement Forward Propagation across all layers

Forward_Model = function(X, parameters, activation, miu){

    # L refers to the number of layers in the model (this excludes the 0-th
layer)
    L = floor(length(parameters)/2)
    caches = list()

    # Initialize X, the original train_input as the first A_prev
    A_prev = X

    # For L-1 times, we use the smoothing/relu function as activation
    for (l in 1:(L-1)){
        # At the beginning of every loop, the output of a layer A
        # becomes the A_prev of the next layer

        cache = Forward_Linear_and_Activation(A_prev,
                                              parameters[[paste0("W",l)]],
                                              parameters[[paste0("b",l)]],
                                              activation,
                                              miu)

        A_prev = cache[["A"]]

        caches[[paste0("Layer_",l)]] = cache
    }

    # For the final L-th layer, we use a sigmoid function
    A = A_prev

    cache = Forward_Linear_and_Activation(A_prev,
                                          parameters[[paste0("W",L)]],
                                          parameters[[paste0("b",L)]],
                                          "sigmoid",
                                          miu)

    caches[[paste0("Layer_",L)]] = cache

    return(caches)
}
```

In [ ]:

```
# (2)(c) Compute Cost

# A_L refers to the activation of the final layer
# Y refers to true labels
# We will call A_L from the "caches" generated off the forward propagation
 module 2(b)(iii)
# i.e. caches[["Layer_L"]][["A"]]

Compute_Cost = function(A_L, Y){

    # m ins the number of training observations
    m = dim(A_L)[1]

    cost = (-1/m) * (sum(Y*log(A_L) + (1-Y)*log(1-A_L)))

    return(cost)
}
```

In [ ]:

```
# (2)(d)(i) Backward Propagation Function Part 1: Linear Portion
# dZ is the cost gradient with respect to the Z (i.e. dJ/dZ, to be obtained
from the next module)
# Recall that dJ/dZ = dJ/dA x dA/dZ, whereby dA/dZ will depend on the activ
ation function, since A = g(Z)

# To obtain cache in this module, we call upon"caches" generated
# from the Forward_Model 2(b)(iii) for each layer

Backward_Linear_Multiplication = function(dZ, cache){
    A_prev = cache[["A_prev"]]
    W = cache[["W"]]

    # Initialize list for output
    gradients=list()

    # Obtain number of training samples
    m = dim(A_prev)[1]

    # Calculate cost gradients dW, db and dA_prev for current layer

    dW = (1/m) * (t(A_prev) %*% dZ)
    db = (1/m) * colSums(dZ)
    dA_prev = dZ %*% t(W)

    gradients[["dW"]] = dW
    gradients[["db"]] = db
    gradients[["dA_prev"]] = dA_prev

    return(gradients)
}
```

In [ ]:

```
# (2)(d)(ii) Backward Propagation Function Part 2: Linear_Backward and Acti
vation Backward Portion
# cache refers to the output of our forward_activation function 2(b)
# we obtain cache by calling "caches" (to be covered in our main integrated
function

Backward_Linear_and_Activation = function(Y, dA, cache, activation, miu){

    # Note dJ/dZ = dJ/dA * dA/dZ
    # dA/dZ will depend on what function you use

    if (activation == "relu"){
        Z = cache[["Z"]]
        dZ = dA * (1 * (Z > 0))
        gradients = Backward_Linear_Multiplication(dZ, cache)

    } else if (activation == "sigmoid"){
        A = cache[["A"]]
        dZ = A - Y
        gradients = Backward_Linear_Multiplication(dZ, cache)

    } else if (activation == "smoothing"){
        Z = cache[["Z"]]
        dZ = dA * 1 / (1 + exp(-Z/miu))
        gradients = Backward_Linear_Multiplication(dZ, cache)

    }

    return(gradients)
}
```

In [ ]:

```
# (2)(d)(iii) Implement Backward Propagation Function 2(d)(ii) across all L
ayers

Backward_Model = function(A_L, Y, caches, activation, miu){

    # Calculate number of layers L (this excludes the zero-th layer)
    L = length(caches)

    # Initialize list of gradients for output
    gradients_library = list()

    # Initiate backward propagation for the L-th layer (i.e. output layer)
    # The initial dA can be set to zero here: Since the L-th layer uses the
sigmoid function,
    # we will compute dZ using dZ = A - Y directly

    cache = caches[[paste0("Layer_",L)]]
    gradients = Backward_Linear_and_Activation(Y, dA = 0, cache, activation
= "sigmoid", miu)

    # Store the gradients into gradients_library
    gradients_library[[paste0("Layer_",L)]] = gradients
    #gradients_library[[Layer_L]]

    # Backpropagate for remaining layers
    for (l in rev(1:(L-1))){
        # Update the cache to be used
        cache = caches[[paste0("Layer_",l)]]

        # Recall that the single Backward_Linear_Multiplication step spews
 out your dA_prev
        # We pump this dA into the next cycle of backward_propagation
        dA = gradients[["dA_prev"]]
        gradients = Backward_Linear_and_Activation(Y, dA, cache, activation
, miu)

        # Store gradients into gradients_library
        gradients_library[[paste0("Layer_",l)]] = gradients
    }

    return(gradients_library)
}
```

In [ ]:

```
# (2)(e)(i) Update Parameters

Update_Parameters = function(parameters, gradients_library, learning_rate,
i, weighted_counts, method){
    # str(gradients_library)
    L = floor(length(parameters)/2)

    #str(gradients_library)

    for (l in (1:L)){
        # Retrieve gradients from gradients_library
        gradients = gradients_library[[paste0("Layer_",l)]]

        dW = gradients[["dW"]]
        #cat("\nDim dW:",dim(dW))
        db = gradients[["db"]]
        #cat("\ndb:", db)

        # Update Parameters
        parameters[[paste0("W",l)]] = parameters[[paste0("W",l)]] - learnin
g_rate * dW
        parameters[[paste0("b",l)]] = parameters[[paste0("b",l)]] - learnin
g_rate * db
    }

    return(parameters)
}
```

In [ ]:

```
# (2)(e)(ii) Minibatch Creator

Create_Batches = function (X, Y, batch_size){
    # X is the original training examples
    # batch_size refers to the number of training examples in each batch

    # m is the total number of training examples
    m = dim(X)[1]

    # We first shuffle the training examples in X randomly
    # drop=FALSE prevents R from converting shuffled_Y, a matrix into a low
er dimensional object
    permutation = sample(m)
    shuffled_X = X[c(permutation),,drop=FALSE]
    shuffled_Y = Y[c(permutation),,drop=FALSE]

    # We hardcode the number of batches to be just 1
    # i.e. Applying mini-batch training with replacement
    # Alternatively, we can set number_of_batches to be floor(nrow(X)/batch
_size)
    # (on top of the necesssary adjustment to correctly capture the "spillo
ver" batch)
    number_of_batches = 1

    mini_batch_X = shuffled_X[1 : batch_size,,drop=FALSE]
    mini_batch_Y = shuffled_Y[1 : batch_size,,drop=FALSE]

    # After creating the mini batches, we a new list "mini_batches" to stor
e both X_mini_batches and Y_mini_batches
    mini_batches = list("number_of_batches" = number_of_batches,
                        "mini_batch_X" = mini_batch_X,
                        "mini_batch_Y" = mini_batch_Y)
    return (mini_batches)
}
```

In [ ]:

```
# (2)(f)(i) Function to predict labels using the trained parameters
# We use back the forward propagation function 2(b) help us generate A_L

Predict_Label = function(parameters, X, n_y, activation, L, threshold = 0.5
, miu){
    Output = Forward_Model(X, parameters, activation, miu)
    A = Output[[paste0("Layer_",L)]][["A"]]

    Y_prediction = matrix(0, nrow = dim(X)[1], ncol = n_y)

    # predictions is a string of TRUE/FALSE
    for (i in 1:dim(X)[1]){
        if (A[i,1] > threshold){
            Y_prediction[i,1] = 1
        }
    }

    prediction_data = list("Y_prediction" = Y_prediction, "A" = A)

    return(prediction_data)
}
```

(3) Create Integrated Function to Create Feedforward NN Model

In [ ]:

```
Deep_Neural_Net = function (X_train, Y_train, X_test, Y_test, hidden_layers
_dimensions, learning_rate,
                            num_iterations, batch_size, activation, miu){

    # Set seed to replicate results
    set.seed(1)

    # Helper 2(a): Find n_x, n_y (dimensions for input/output data) and Ini
tialize parameters
    first_last_layers_dimensions = First_Last_Layers_Dimensions(X_train, Y_
train)
    parameters = Initialize_Parameters(first_last_layers_dimensions, hidden
_layers_dimensions)

    # We also initialize a set of parameters for our average_P calculations
    # Note these are just "placeholders" to make sure the dimensions of the
parameters are correct
    # The initial values are inconsequential since their weight (towards ca
lculating average P)
    # will zero for the first loop
    parameters_average = Initialize_Parameters(first_last_layers_dimensions
, hidden_layers_dimensions)

    # Initialize Cost vectors for train and test set
    Costs_train = c()
    Costs_test = c()

    # Initialize "BEST" Cost vectors for train and test set
    Best_Costs_train = c()
    Best_Costs_test = c()

    # Initialize "Average Parameters" Cost vectors for train and test set
    Average_P_Costs_train = c()
    Average_P_Costs_test = c()

    # Initialize "Weighted Parameters" Cost vectors for train and test set
    # Here we used the average of the parametrs for the last 150 iterations
as an example
    P_150_Costs_train = c()
    P_150_Costs_test = c()

    # Initialize list for parameters storage
    parameters_store = list()

    # Note we store number of layers in a variable L first
    L = floor(length(parameters)/2)

    # Set up loop for each iterations
```

```
    for (i in 1:num_iterations){

        # Helper 2(e)(ii): Create mini-batches for each loop
        Batches_Store = Create_Batches(X_train, Y_train, batch_size)

        # Retrieve number of batches created
        # Note that for mini-batch training with replacement, we hard-fix t
he number of batches to be 1
        number_of_batches = Batches_Store$number_of_batches
        # retrieve the store of X_mini_batches and Y_minibatches
        X_mini_batch = Batches_Store$mini_batch_X
        Y_mini_batch = Batches_Store$mini_batch_Y

        # Helper 2(k): Create mini-batches for each loop
        Batches_Store = Create_Batches(X_train, Y_train, batch_size)

        # Retrieve number of batches created
        number_of_batches = Batches_Store$number_of_batches
        # retrieve the store of X_mini_batches and Y_minibatches
        X_mini_batch = Batches_Store$mini_batch_X
        Y_mini_batch = Batches_Store$mini_batch_Y

        # Helper 2(e): Perform Forward Propagation
        caches = Forward_Model(X_mini_batch, parameters, activation, miu)

        # Helper 2(i): Perform Backward Propagation
        # We use the A_L from the cache to initiate the backward propagatio
n
        gradients_library = Backward_Model(A_L, Y_mini_batch, caches, activ
ation, miu)

        # Helper 2(j): Update Parameters
        # If using a variable learning rate, we can additionally insert a l
ine to modify the learning rate
        # For instance, when using a variable learning rate for minibatch l
earning,
        # We can set "learning_rate" = learning_rate / sqrt(i+1) <<As per t
he expression>>
        parameters = Update_Parameters(parameters, gradients_library, learn
ing_rate, i, weighted_counts, method)
        parameters_store[[paste0("Iter:",i)]] = parameters

        ### Helper 2(h): Compute Cost

        ## (I) For Regular Gradient Descent
        # Compute Cost for train set
        Train_Run = Forward_Model(X_train, parameters, activation, miu)
        A_L_train = Train_Run[[paste0("Layer_",L)]][["A"]]

        Cost_train = Compute_Cost(A_L_train, Y_train)
```

```
        Costs_train = c(Costs_train, Cost_train)

        # Compute Cost for test set
        Test_Run = Forward_Model(X_test, parameters, activation, miu)
        A_L_test = Test_Run[[paste0("Layer_",L)]][["A"]]

        Cost_test = Compute_Cost(A_L_test, Y_test)
        Costs_test = c(Costs_test, Cost_test)

        ## (II) Keep Track of "BEST" Cost for Train Set, and calculate corr
esponding test cost
        Best_Costs_train = c(Best_Costs_train, min(Cost_train, Best_Costs_t
rain[length(Best_Costs_train)]))
        # BE CAREFUL - we don't simply take the minimum of Cost_test and Be
st_Costs_test[length(Best_Costs_test)] here
        # We are finding out the test cost for the parameters that give the
lowest cost for the train set thus far

        if (Best_Costs_train[length(Best_Costs_train)] != Cost_train){
            # If Cost_train did not perform better, we maintain the last Be
st_Costs_test value
            Best_Costs_test = c(Best_Costs_test, Best_Costs_test[length(Bes
t_Costs_test)])

        } else {
            # If Cost_train performs better, we add corresponding test_cost
value to list Best_Costs_test
            Best_Costs_test = c(Best_Costs_test, Cost_test)
        }

        ## (III) Compute Cost for Average of parameters for all past iterat
ions

        # Calculate the Average of Parameters for all past iterations
        for (l in 1:L){
            parameters_average[[paste0("W",l)]] = ((i-1)/i) * parameters_av
erage[[paste0("W",l)]] + (1/i) * parameters[[paste0("W",l)]]
            parameters_average[[paste0("b",l)]] = ((i-1)/i) * parameters_av
erage[[paste0("b",l)]] + (1/i) * parameters[[paste0("b",l)]]

        }

        # Compute Cost for train set
        Average_P_Train_Run = Forward_Model(X_train, parameters_average, ac
tivation, miu)
        Average_P_A_L_train = Average_P_Train_Run[[paste0("Layer_",L)]][[
"A"]]

        Average_P_Cost_train = Compute_Cost(Average_P_A_L_train, Y_train)
        Average_P_Costs_train = c(Average_P_Costs_train, Average_P_Cost_tra
```

```
in)

        # Compute Cost for test set
        Average_P_Test_Run = Forward_Model(X_test, parameters_average, acti
vation, miu)
        Average_P_A_L_test = Average_P_Test_Run[[paste0("Layer_",L)]][["A"
]]

        Average_P_Cost_test = Compute_Cost(Average_P_A_L_test, Y_test)
        Average_P_Costs_test = c(Average_P_Costs_test, Average_P_Cost_test)

        ###

        ## (IV) Compute Cost for Average of parameters for past 150 iterati
ons

        # For the first 150 iterations, the parameters are no different fro
m (C)
        if (i <= 150){
            parameters_P_150 = parameters_average

        } else {

            # ...From 151th loop onwards, we take away 1/150 of the (i-150)
-th iteration and add back
            for (l in 1:L){

                parameters_remove = parameters_store[[paste0("Iter:",(i-150
))]]

                parameters_P_150[[paste0("W",l)]] = (parameters_P_150[[past
e0("W",l)]] -
                                                    (1/150) * parameters_r
emove[[paste0("W",l)]] +
                                                    (1/150) * parameters[
[paste0("W",l)]])
                parameters_P_150[[paste0("b",l)]] = (parameters_P_150[[past
e0("b",l)]] -
                                                    (1/150) * parameters_r
emove[[paste0("b",l)]] +
                                                    (1/150) * parameters[
[paste0("b",l)]])

            }
        }

        # Compute Cost for train set
        P_150_Train_Run = Forward_Model(X_train, parameters_P_150, activati
on, miu)
        P_150_A_L_train = P_150_Train_Run[[paste0("Layer_",L)]][["A"]]
```

```
        P_150_Cost_train = Compute_Cost(P_150_A_L_train, Y_train)
        P_150_Costs_train = c(P_150_Costs_train, P_150_Cost_train)

        # Compute Cost for test set
        P_150_Test_Run = Forward_Model(X_test, parameters_P_150, activation
, miu)
        P_150_A_L_test = P_150_Test_Run[[paste0("Layer_",L)]][["A"]]

        P_150_Cost_test = Compute_Cost(P_150_A_L_test, Y_test)
        P_150_Costs_test = c(P_150_Costs_test, P_150_Cost_test)

        }

    # Compile Output: We capture needed for ROC/AUC and Cost Analyses
    output=list()

    # Helper 2(f)(i): With the optimized parameters, predict both the train
and test set with threshold 0.5
    #(i) For train data
    prediction_data_train = Predict_Label(parameters,
                                          X_train,
                                          first_last_layers_dimensions[["n_
y"]],
                                          activation,
                                          L, threshold = 0.5,
                                          miu)
    Y_prediction_train = prediction_data_train[["Y_prediction"]]

    #(ii) For test data
    prediction_data_test = Predict_Label(parameters,
                                         X_test,
                                         first_last_layers_dimensions[["n_
y"]],
                                         activation,
                                         L, threshold = 0.5,
                                         miu)
    Y_prediction_test = prediction_data_test[["Y_prediction"]]

    # Print a simple train/test error
    cat("\nTrain accuracy: ", (1-((sum(abs(Y_prediction_train - Y_train)))/
length(Y_prediction_train)))*100,"%")
    cat("\nTest accuracy: ", (1-((sum(abs(Y_prediction_test - Y_test)))/len
gth(Y_prediction_test)))*100,"%")

    # Store output in a list - might be useful for further analysis
    output=list("Costs_train" = Costs_train,
                "Costs_test" = Costs_test,
                "Best_Costs_train" = Best_Costs_train,
                "Best_Costs_test" = Best_Costs_test,
```

```
                "Average_P_Costs_train" = Average_P_Costs_train,
                "Average_P_Costs_test" = Average_P_Costs_test,
                "P_150_Costs_train" = P_150_Costs_train,
                "P_150_Costs_test" = P_150_Costs_test,
                "A_train" = prediction_data_train[["A"]],
                "A_test" = prediction_data_test[["A"]],
                "parameters" = parameters,
                "parameters_store" = parameters_store)

    return(output)
}
```

(4) Run Model!

In [ ]:

```
## Depending on the type of model (batch, or mini-batch) and
## type of learning rate (fixed or variable, and with what learning factor)
## We compute our learning rate differently
## Here, we cited the example of calculating fixed learning rate for batch
 learning

# (A) Calculate fixed learning rate for Batch learning

# Create train_augment (add column of 1-s beside original training set matr
ix)
train_augment = as.data.frame(train_input)
train_augment$intersect = rep(1, nrow(train_input))
train_augment = train_augment[,c(19,1:18)]

# Obtain max singular value
max_d = max((svd(train_augment))$d)

# Determine learning rate
alpha_batch_A1 = (4 * nrow(train_input)) / max_d**2
cat("\nLearning Rate for Batch Learning:", alpha_batch_A1)
```

In [ ]:

```
# Set Hidden Layer Dimensions - Here we set a single hidden layer with 13 h
idden nodes
hidden_layers_dimensions = c(13)

# Run Model
start_time = Sys.time()

Results_A1 = Deep_Neural_Net(train_input, train_labels, test_input, test_la
bels, hidden_layers_dimensions,
                          learning_rate = alpha_batch_A1, num_iterations =
10000,
                          batch_size = 16, activation = "smoothing", miu =
3)


end_time = Sys.time()
Time_Taken = difftime(end_time, start_time, units ="mins")

cat("\nTime_Taken: ", Time_Taken, "min", "\n")
```

In [ ]:

```
## (Optional) Plot ROC curve for Train and Test Data Set
# To plot the ROC for the model, we use the prediction function under the R
OCR library

# (1) Train Data (Note that we don't use A_L_train directly because the ord
er is jumbled up)
A_train = Results_A1[["A_train"]]

# Now for the prediction function
pred_train = prediction(A_train, train_labels)

roc.perf_train = performance(pred_train, measure = "tpr", x.measure = "fpr"
)
plot(roc.perf_train)
abline(a=0, b= 1)

# We display the AUC Value
auc.perf_train = performance(pred_train, measure = "auc")
cat("\nTrain AUC: ", auc.perf_train@y.values[[1]])

# (2) Test Data
A_test = Results_A1[["A_test"]]

# Now for the prediction function
pred_test = prediction(A_test, test_labels)

roc.perf_test = performance(pred_test, measure = "tpr", x.measure = "fpr")
plot(roc.perf_test)
abline(a=0, b= 1)

# We display the AUC Value
auc.perf_test = performance(pred_test, measure = "auc")
cat("\nTest AUC: ", auc.perf_test@y.values[[1]])
```

In [ ]:

```
# Plot loss over iterations
Costs_train = Results_A1[["Costs_train"]]
Costs_test = Results_A1[["Costs_test"]]
Best_Costs_train = Results_A1[["Best_Costs_train"]]
Best_Costs_test = Results_A1[["Best_Costs_test"]]
Average_P_Costs_train = Results_A1[["Average_P_Costs_train"]]
Average_P_Costs_test = Results_A1[["Average_P_Costs_test"]]
P_150_Costs_train = Results_A1[["P_150_Costs_train"]]
P_150_Costs_test = Results_A1[["P_150_Costs_test"]]

# Plot Costs for Train Set
plot(Costs_train, type="l", lty=1, lwd=1, main = "Cost over Iterations",
     xlab = "Iteration", ylab = "Cost", col = "cyan")
lines(Best_Costs_train, col="black", type="l", lty=1, lwd=1)
lines(Average_P_Costs_train, col="red", type="l", lty=1, lwd=1)
lines(P_150_Costs_train, col="brown", type="l", lty=1, lwd=1)


legend("topright",
       legend = c("Train Cost", "Best Train Cost", "Average P", "P_150"),
       col = c("cyan", "black", "red", "brown"),
       lty = c(1,1,1,1), cex=0.7)

# Plot Costs for Test Set
plot(Costs_test, type="l", lty=1, lwd=1, main = "Cost over Iterations",
     xlab = "Iteration", ylab = "Cost", col = "cyan")
lines(Best_Costs_test, col="black", type="l", lty=1, lwd=1)
lines(Average_P_Costs_test, col="red", type="l", lty=1, lwd=1)
lines(P_150_Costs_test, col="brown", type="l", lty=1, lwd=1)


legend("topright",
       legend = c("Test Cost for Corr. Train", "Test Cost for Best Corr. Tr
ain", "Average P","P_150"),
       col = c("cyan", "black", "red", "brown"),
       lty = c(1,1,1,1), cex=0.7)
```

In [ ]:

```
# (Optional) Investigate Minimum Cost for Test and Train Set

cat("\nMinimum Train Cost: ",min(Costs_train), "@", which(Costs_train == mi
n(Costs_train)), "iteration.")
cat("\n The Corresponding Test Cost is ", Costs_test[which(Costs_train == m
in(Costs_train))])
cat("\n")

cat("\nMinimum Test Cost: ",min(Costs_test), "@", which(Costs_test == min(C
osts_test)), "iteration.")
cat("\n The Corresponding Train Cost is ", Costs_train[which(Costs_test ==
min(Costs_test))])
cat("\n")
```

Appendix B: Code for Logistic Regression Network

(Setup) Call Libraries for Functions to be Used

In [ ]:

```
# Call on caret for data splitting and ggplot2 functions
library(caret)

# Call on ROCR for ROC/AUC analysis
library(ROCR)
```

(1) DATA PREPARATION

(a) Import Data / Modify Structure of Data.
This includes converting categorical data into binary (1s and 0s) form.
(b) Split Data into *Train* and *Test* Set.
(c) Split Data into *Input* and *Label* Sets.
(d) Standardize Data.

In [ ]:

```
# (1)(a) Import Data / Modify Structure of Data
data = read.csv("framingham.csv")

str(data)
head(data)

# What is the naive model accuracy?
cat("Naive Model Accuracy is", 1 - sum(data$TenYearCHD)/nrow(data))
```

In [ ]:

```
# We note that the independent variable "Education" is categorical
# We need to convert these into binary dummy variables (one for each factor
level) to make it compatible with neural nets
# To do this, we use the command "model.matrix" on the education variable
# Note: We convert the into dataframe makes it easier to manipulate

education_variables = as.data.frame(model.matrix(~ 0 + education, data = da
ta))
head(education_variables)
```

In [ ]:

```
# We then replace the education variable with the binary dummy variables cr
eated
# Obtain index of columns before/after the education column
index_before = c(1:(which(colnames(data)=="education")-1))
index_after =  c((which(colnames(data)=="education")+1):ncol(data))

# We do a cbind or column bind to get the data in the form we want
data = cbind(data[,c(index_before)],education_variables,data[,c(index_after
)])
```

In [ ]:

```
# (1)(b) Split Data Set in Train and Test Set
set.seed(15071)
split = createDataPartition(data$TenYearCHD, p = 0.75, list = FALSE)

train = data[split,]
test = data[-split,]

head(train,6)
head(test,6)

# Examine dimensions of Train / Test data
cat("Number of Training Examples in Train Set:", dim(train)[1])
cat("\nNumber of Training Examples in Test Set:", dim(test)[1])
```

In [ ]:

```
#(1)(c) Split Data into Input Variables and Labels
# Recall that we will only feed input variables into the network
# We will isolate the actual label "TenYearCHD" separately
# Convert all to matrixes with as.matrix

train_input = as.matrix(train[,-which(colnames(train)=="TenYearCHD")])
train_labels = as.matrix(train[,which(colnames(train)=="TenYearCHD")])

test_input = as.matrix(test[,-which(colnames(test)=="TenYearCHD")])
test_labels = as.matrix(test[,which(colnames(test)=="TenYearCHD")])

cat("Dimensions of train_input: ", as.character(dim(train_input)), "\n")
cat("Dimensions of train_labels: ", as.character(dim(train_labels)), "\n")

cat("Dimensions of test_input: ", as.character(dim(test_input)), "\n")
cat("Dimensions of test_input: ", as.character(dim(test_labels)), "\n")
```

In [ ]:

```
# 1(d) Data Standardization
# We convert the training data into a standard normal distribution using th
e training mean and variance

# train mean is the vector of means across the 17 input variables
train_mean = colMeans(x = train_input, na.rm = FALSE)
train_mean

# train_variance is the vector of SAMPLE variance across the 17 input varia
bles
train_variance = apply(train_input, 2, var)
train_variance

# We standardize the input datasets
# We use sweep to broadcast in R
# Sweep = 2 means we sweep by columns(everyvalue is deducted by column mea
n)

# First, we deduct away the mean
train_input = sweep(train_input, MARGIN = 2, train_mean, "-")
train_input

test_input = sweep(test_input, MARGIN = 2, train_mean, "-")

# Next, we divide by the sample std dev
train_input = sweep(train_input, MARGIN = 2, sqrt(train_variance), "/")
test_input = sweep(test_input, MARGIN = 2, sqrt(train_variance), "/")
```

(2) HELPER FUNCTIONS

- (a) Initialize Parameters **w**
- (b) Calculate Sigmoid
- (c) Calculate **y_pred**
- (d) Run Single Iteration of **Forward & Backward Propagation**
- (e) **Iterate** through (2)(d) and extract **cost** value over iterations

In [ ]:

```
# (2)(a) "Initialize_Parameters" creates a vector (or matrix with 1 column)
of n zeros
# where n is eventually the number of input variables in our training examp
le
# we use "b" for w_0, the bias term here

Initialize_Parameters = function (no_of_input_variables){
    set.seed(1)
    w = matrix(0, nrow = no_of_input_variables, ncol = 1)
    b = 0

    parameters = list("w" = w, "b" = b)

    return (parameters)
}
```

In [ ]:

```
# (2)(b) "Sigmoid" applies the sigmoid function onto an given input

Sigmoid = function(z){
    return (1 / (1 + exp(-z)))
}
```

In [ ]:

```
# (2)(c) "Predict" calculates the predicted label (0 or 1) from the finaliz
ed parameters after training
# The initial threshold is set to 0.5

Predict_Label = function (w, b, X, threshold = 0.5){
    # Note: This creates a matrix that stores the predicted label (0 or 1)
 in a single column
    # Note: The size of the column, dim(X)[1] is the number of training exa
mples, or rows in X
    Y_prediction = matrix(0, nrow = dim(X)[1],ncol = 1)

    # Compute the vector A that stores the output from the sigmoid function
    A = Sigmoid(X %*% w + b)

    # We run a for loop to compare with the threshold
    for (i in 1:dim(X)[1]){
        if (A[i,1] > threshold){
            Y_prediction[i,1] = 1
        }
    }

    return (Y_prediction)
}
```

In [ ]:

```
# 2(d) Propagate runs a single iteration of the forward and backward propag
ation across m training examples

Propagate = function(w, b, X, Y){

    # We store the number of training examples in variable m
    m = dim(X)[1]

    # Forward Propagation - We calculate A and cost across the m training e
xamples
    # "*" gives element wise multiplication
    A = Sigmoid(X %*% w + b)
    cost = -1/m * (sum(Y*log(A) + (1-Y)*log(1-A)))

    # Backward Propagation
    # For simplicity, we designate dJ/dw as dw, dJ/db as db
    dw = 1/m * (t(X) %*% (A-Y))
    db = 1/m * (sum(A-Y))

    gradients_and_cost = list("cost" = cost, "dw" = dw, "db" = db)

    return (gradients_and_cost)
}
```

In [ ]:

```
# 2(e) Iterations the propagate function for num_iterations times
# We also keep track of costs to study how the process of optimizing the pa
rameters

gradient_descent = function (w, b, X, Y, num_iterations, learning_rate, pri
nt_cost = TRUE){

    # Initialize list that will store cost values
    costs = c()

    for (i in 1:num_iterations){
        # Perform 1 iteration and retrieve cost gradients
        gradients_and_cost = Propagate(w, b, X, Y)
        dw = gradients_and_cost[["dw"]]
        db = gradients_and_cost[["db"]]
        last_cost = gradients_and_cost[["cost"]]

        # Update parameters for iteration
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # Record the costs every 100 iterations
        if (i %% 10 == 0){
            costs = c(costs, last_cost)
        }

        # If print_cost is True then we print the cost every 10 iterations
        if (i %% 10 == 0 & print_cost == TRUE){
            print(paste0("Cost after ", i, " iterations: ", last_cost))
        }
    }

    parameters_and_costs = list("w" = w, "b" = b, "costs" = costs)
    return (parameters_and_costs)
}
```

(3) Perform Logistic Regression Model with Integrated Function

In [ ]:

```
log_reg_model = function(X_train, Y_train, X_test, Y_test, num_iterations = 
2000, learning_rate = 0.5, print_cost = False){

    # Initialize parameters w and b with helper function 2(a)
    parameters = Initialize_Parameters(dim(X_train)[2])
    w = parameters[["w"]]
    b = parameters[["b"]]

    # Perform gradient descent over num_iterations using helper function 2
(e)
    parameters_and_costs = gradient_descent(w, b, X_train, Y_train, num_ite
rations, learning_rate, print_cost = TRUE)
    w_final = parameters_and_costs[["w"]]
    b_final = parameters_and_costs[["b"]]
    costs = parameters_and_costs["costs"]

    # With the optimized parameters, we use helper function 2(c) to predict
both the train and test set with threshold 0.5
    Y_prediction_train = Predict_Label(w_final, b_final, X_train, threshold
= 0.5)
    Y_prediction_test = Predict_Label(w_final, b_final, X_test, threshold =
0.5)

    # Print a simple train/test error
    cat("Train accuracy: ", (1-((sum(abs(Y_prediction_train - Y_train)))/le
ngth(Y_prediction_train)))*100,"%")
    cat("\nTest accuracy: ", (1-((sum(abs(Y_prediction_test - Y_test)))/len
gth(Y_prediction_test)))*100,"%")

    # Store output in a list - might be useful for further analysis
    model_output=list("costs" = costs,
                      "Y_prediction_train" = Y_prediction_train,
                      "Y_prediction_test" = Y_prediction_test,
                      "w_final" = w_final,
                      "b_final" = b_final,
                      "learning_rate" = learning_rate,
                      "num_iterations" = num_iterations)


    return (model_output)
}
```

Run Model and See Results!

In [ ]:

```
# Calculate fixed learning rate for

# Create train_augment (add column of 1-s beside original training set matr
ix)
train_augment = as.data.frame(train_input)
train_augment$intersect = rep(1, nrow(train_input))
train_augment = train_augment[,c(19,1:18)]

# obtain max singular value
max_d = max((svd(train_augment))$d)

# determine learning rate (We use a learning factor of 1 here)
alpha = (4 * nrow(train_input)) / max_d**2
```

In [ ]:

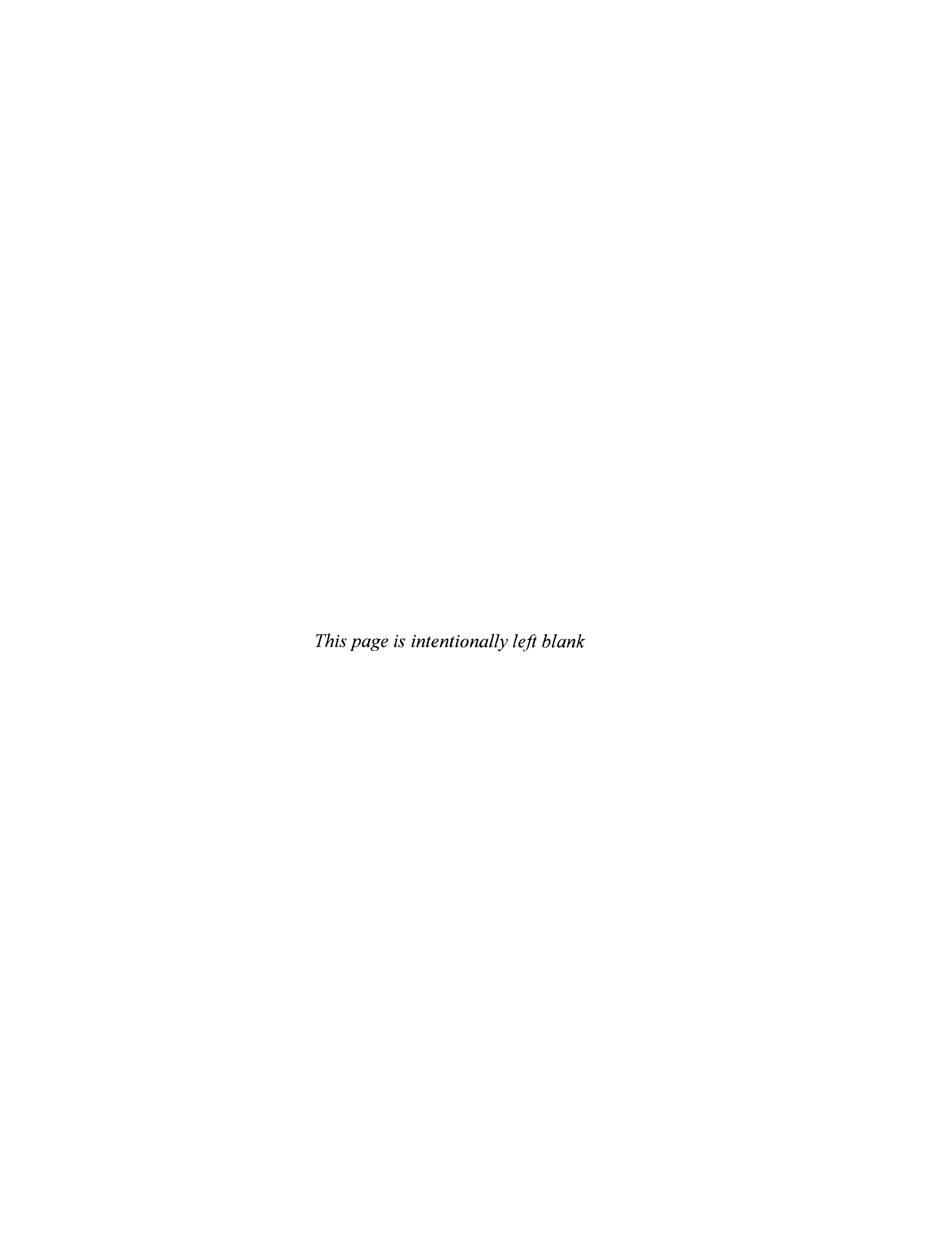
```
# This considers the result for the default case, whereby we use a threshol
d of 0.5
answer = log_reg_model(train_input, train_labels, test_input, test_labels,
num_iterations = 100, learning_rate = alpha, print_cost = False)
```

In [ ]:

```
# To plot the ROC for the model, we use the prediction function under the R
OCR library
# Essentially, we leave behind the last step of the predict function, such
 that we only calculate A

# The A is a value between 0 to 1 suggesting the "propensity of contracting
CHD within 10 years"
# A will be fed into the first argument of the "prediction function"
# The second argument to be fed is the actual prediction test_labels

w_final = answer[["w_final"]]
b_final = answer[["b_final"]]

A_test = (test_input %*% w_final) + b_final

# Now for the prediction function
pred = prediction(A_test, test_labels)

roc.perf = performance(pred, measure = "tpr", x.measure = "fpr")
plot(roc.perf)
abline(a=0, b= 1)

# This quantifies AUC value = ~0.714 (same value as using a glm() function
 in R)
# Note that this is  consistent with the assignment on logistics regression
back in 15.071 (0.71459)
auc.perf = performance(pred, measure = "auc")
cat("AUC Value:", as.character(auc.perf@y.values))
```

In [ ]:

```
# Plot loss over iterations
costs = answer$costs
plot(costs$costs, type = "l", main = "Cost over Iterations", xlab = "Iterat
ion x 10", ylab = "Cost")
```



This page is intentionally left blank

References

[1]	Goodfellow, Bengio, and Courville. "Deep Learning", *Chapter 1*: *Introduction*, pp. 1-11. The MIT Press, 2016.
[2]	Goodfellow, Bengio, and Courville. "Deep Learning", *Chapter 1*: *Introduction: Who should read this book?*, pp. 9. The MIT Press, 2016.
[3]	Goodfellow, Bengio, and Courville. "Deep Learning", *Chapter 1*: *Introduction: Illustration of a deep learning model"*, pp. 6. The MIT Press, 2016.
[4]	Andrew Ng. Deeplearning.ai. *Module 1: Neural Networks and Deep Learning. From Deep Learning Specialization In " What is a Neural Network?".* Retrieved from https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning.
[5]	Dugas, Bengio, Beslisle, Nadeau, Garcia. *Incorporating Second-Order Functional Knowledge for Better Option Pricing*, CIRANO & University of Montreal, 2001.
[6]	Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best *multi-stage architecture for object recognition?* In ICCV'09 .
[7]	Nair, V. and Hinton, G. (2010). *Rectified linear units improve restricted Boltzmann machines*. In CML'2010 .
[8]	Glorot, X., Bordes, A., and Bengio, Y. (2011a). *Deep sparse rectifier neural networks*. In AISTATS'2011 .
[9]	Sun, Yi, Wang, Xiaogang, and Tang, Xiaoou. Deeply learned face representations are sparse, selective, and robust. arXiv preprint arXiv:1412.1265, 2014.
[10]	Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). *Rectifier nonlinearities improve neural network acoustic models.* In ICML Workshop on Deep Learning for Audio, Speech, and Language Processing.
[11]	Zheng, Yang, Liu (2015). *Improving Deep Neural Networks Using Softplus Units*. National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Sciences, Bejing. International Joint Conference on Neural Networks IJCNN. 2015.
[12]	Freund R., Grigas P., Mazumder R. (2018). *Condition Number Analysis of Logistic Regression, and its Implications for Standard First-Order Solution Methods*. MIT Sloan School of Management.
[13]	Framingham Heart Study (2019). *National Heart Institute (now known as the National Heart, Lung, and Blood Institute or NHLBI). Retrieved as part of Course Materials for 15.071 The Analytics Edge, MIT Sloan School of Management, "Logistics Regression".*
[14]	Kaggle Online Community(2017). *Predicting Churn for Bank Customers*. Retrieved from https://www.kaggle.com/adammaus/predicting-churn-for-bank-customers

[15]	Hofmann H. (1994) *German Credit Data*, Institut f"ur Statistik und "Okonometrie Universit"at Hamburg. Retrieved from https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data)