SPAF-network with Saturating Pretraining Neurons

A Thesis Submitted to the Committee of Graduate Studies
in Partial Fulfillment of the Requirements for the Degree of Master of Science
in the Faculty of Arts and Science

TRENT UNIVERSITY
Peterborough, Ontario, Canada

© Copyright by Hasham Burhani 2014

Applied Modeling and Quantitative Methods
M.Sc. Graduate Program
October 5, 2015

Abstract

SPAF-network with Saturating Pretraining Neurons

Hasham Burhani

In this work, various aspects of neural networks, pre-trained with denoising autoencoders (DAE) are explored. To saturate neurons more quickly for feature learning in DAE, an activation function that offers higher gradients is introduced. Moreover, the introduction of sparsity functions applied to the hidden layer representations is studied. More importantly, a technique that swaps the activation functions of fully trained DAE to logistic functions is studied, networks trained using this technique are reffered to as SPAF-networks. For evaluation, the popular MNIST dataset as well as all 3 sub-datasets of the Chars74k dataset are used for classification purposes. The SPAF-network is also analyzed for the features it learns with a logistic, ReLU and a custom activation function. Lastly future roadmap is proposed for enhancements to the SPAF-network.

Keywords: Unsupervised Pretraining, Machine Learning, Neural Networks, Denoising Auto-Encoders, SPAF-network, Artificial Neurons.

Acknowledgements

I would like to thank my supervisor Dr. Wenying Feng for supporting and guiding me over the last 2 years. Thank you for being a great mentor and placing your confidence in me. Not to mention, keeping my goals in check.

I would also like to thank my supervisory committee members Dr. Kenzu Abdella and Dr. Richard Hurley. Your advice and input in regards to my research have been tremendously helpful.

I would also like to thank Dr. Hugo Larochelle, for taking the time to make tremendous deep learning videos available online. Not only that, that all of my questions pertaining to the said videos as well as research focused questions were personally answered. Without you I wouldn't have been able to come this far in this research. Thank you.

I would like to especially thank and dedicate this thesis to my family for the love, support and encouragement they have given me over the years.

Table of Contents

Abstract **ii**

Acknowledgements **iii**

Table of Contents **iv**

List of Figures **ix**

List of Tables **xiii**

1 Introduction **1**

1.1 Introduction 1

1.2 Data, Datasets and Functions 5

1.3 Major Contribution 6

1.4 Thesis Outline 6

2 Background **8**

2.1 Artificial Neural Networks 8

2.1.1 Artificial Neuron 9

2.1.2 A Network of Neurons 11

2.1.3 Hidden Layers 13

2.1.4 Intermediate Representation 15

2.2 Applications for ANNs 20

2.2.1 Multi-Class Classification 21
2.2.2 Document Classification 23
2.2.3 Image Classification 25
2.3 Feature Selection 25
2.3.1 Existing Methodologies 27
2.3.2 Deep Learning 29

3 Sparse and Saturating Neurons 31

3.1 Preliminaries 31
3.1.1 Cost Function 31
3.1.2 Backpropagation 33
3.1.3 Batch & Mini Batch Learning 36
3.1.4 Initializing Parameters 38
3.1.5 Overfitting, Generalization & Regularization 40
3.1.6 Measuring Sparsity 42
3.1.7 Unsupervised Feature Learning & Autoencoders 44
3.1.8 Denoising AutoEncoder 46
3.2 The Model 49
3.2.1 New Activation Function 51
3.2.2 Sparse favouring Loss Function 52
3.3 Conclusion 54

4 Implementation & Experiments 56

4.1 Implementation 56
4.2 Datasets & Experiments 60
4.2.1 MNIST 61
4.2.2 Chars74k 63

4.2.3 Learned Features 72

4.2.4 Images from Weights 72

4.3 Conclusion 77

5 Conclusion 83

5.1 Conclusion 83

5.2 Limitations & Future Work 84

References 91

A Appendix 92

A.1 MNIST 200-200 Sigmoidal Network No Penalty Term 92

A.2 MNIST 200-200 Sigmoidal Network log penalty 93

A.3 MNIST 200-200 Sigmoidal Network L1 penalty 93

A.4 MNIST 200-200 SPAF-Network No Penalty Term 94

A.5 MNIST 200-200 SPAF-Network log penalty 94

A.6 MNIST 200-200 SPAF-Network L1 penalty 94

A.7 Hnd dataset 200-200 Sigmoidal-Network L1 penalty 95

A.8 Hnd dataset 200-200 Sigmoidal-Network log penalty 96

A.9 Hnd dataset 200-200 Sigmoidal-Network No penalty 96

A.10 Hnd dataset 200-200 SPAF-Network L1 penalty 97

A.11 Hnd dataset 200-200 SPAF-Network log penalty 97

A.12 Hnd dataset 200-200 SPAF-Network No penalty 98

A.13 Img dataset 500-400 Sigmoidal-Network No penalty 99

A.14 Img dataset 500-400 Sigmoidal-Network L1 penalty 99

A.15 Img dataset 500-400 Sigmoidal-Network log penalty 99

A.16 Img dataset 500-400 SPAF-Network No penalty 100

A.17 Img dataset 500-400 SPAF-Network L1 penalty 100
A.18 Img dataset 500-400 SPAF-Network log penalty 100
A.19 Fnt dataset 50-50 SPAF-Network 101
A.20 Fnt dataset 100-100 SPAF-Network 101
A.21 Fnt dataset 200-200 SPAF-Network 102
A.22 Fnt dataset 400-400 SPAF-Network 102
A.23 Fnt dataset 700-700 SPAF-Network 103
A.24 Fnt dataset 1000-1000 SPAF-Network 104
A.25 Fnt dataset 50-50 Sigmoidal-Network 104
A.26 Fnt dataset 100-100 Sigmoidal-Network 105
A.27 Fnt dataset 200-200 Sigmoidal-Network 105
A.28 Fnt dataset 300-300 Sigmoidal-Network 106
A.29 Fnt dataset 400-400 Sigmoidal-Network 106
A.30 Fnt dataset 700-700 Sigmoidal-Network 107
A.31 Fnt dataset 1000-1000 Sigmoidal-Network 107
A.32 10% training Fnt dataset 200-200 Spaf-Network 108
A.33 20% training Fnt dataset 200-200 Spaf-Network 108
A.34 50% training Fnt dataset 200-200 Spaf-Network 109
A.35 70% training Fnt dataset 200-200 Spaf-Network 109
A.36 90% training Fnt dataset 200-200 Spaf-Network 110
A.37 10% training Fnt dataset 200-200 Sigmoidal-Network 110
A.38 20% training Fnt dataset 200-200 Sigmoidal-Network 111
A.39 50% training Fnt dataset 200-200 Sigmoidal-Network 111
A.40 70% training Fnt dataset 200-200 Sigmoidal-Network 112
A.41 90% training Fnt dataset 200-200 Sigmoidal-Network 112

B Appendix **114**

B.1 MNIST Features Layer-1 114

B.2 MNIST Features Layer-2 117

B.3 Fnt Features Layer-1 120

B.4 Fnt Features Layer-2 123

B.5 Hnd Features Layer-1 126

B.6 Hnd Features Layer-2 129

List of Figures

1.1 Line of best fit attempts applied to house square footage (x) versus price (y) . . . 2

2.1 Artificial neuron with multiple inputs will produce one output. However this output can be connected to multiple neurons, much like how biological neurons work. . . . 9

2.2 Some common activation functions that can be found in neural networks 10

2.3 OR Network . . . 12

2.4 AND Network . . . 13

2.5 Feed-Forward neural network has input nodes that simply take the value of each input, and the following layers up until the last layer (Output Layer) are called hidden layers. . . . 14

2.6 A network with three input neurons, connected to two hidden neurons all feeding output to a final output neuron. . . . 15

2.7 OR Function displayed in a scatter plot. Blue stars marks indicate an output of one, and red squares indicate an output of zero. The OR function can be linearly separated as indicated by the black line. . . . 16

2.8 AND Function displayed in a scatter plot. Blue stars marks indicate an output of one, and red squares indicates an output of zero. The AND function can be linearly separated as indicated by the black line. 17

2.9 XOR Function displayed in a scatter plot. Blue marks indicate an output of one, and red indicates an output of zero. The XOR function cannot be linearly separated by a line. . . . 18

2.10 XOR Network with OR and AND networks . . . 18

2.11 All possible values of H1 and H2 from 2.10 plotted on a scatter plot. The problem is linearly separable. . . . 19

2.12 In a classification problem, neural networks are designed to have multiple output neurons, each representing a category. In this case, the neural network is designed to categorise input into n categories. . . . 21

2.13 Unlike humans, computers cannot process the images with the complexity of all of its features. It simply sees pixels that make up the image. 26

3.1 Derivatives of a logistic and ReLU activation functions 39

3.2 Comparison of the L1 norm in blue versus the log-penalty in red proposed for this work. 43

3.3 AutoEncoder transforms the input into a hidden representation, then reconstructs the input back from the hidden representation. 45

3.4 Denoising AutoEncoder gets a corrupted input and is expected to reconstruct the original. (Dotted neurons automatically are set to a value of 1). 47

3.5 Comparison of the modified Elliot function in red versus the sigmoidal activation function in blue 52

3.6 Comparison of the proposed activation function in red versus the ReLU activation function in blue 53

3.7 Derivatives of a logistic and the newly proposed activation function . 54

4.1 A neural network that follows pre-training paradigm trains each hidden layer with the input of the preceeding layer. Encoding weights are kept, and the decoding weights are removed once pre-training is complete. . 57

4.2 Sample images from the MNIST dataset, as can be seen, there is a wide variation in writing styles 62

4.3 Comparison of Sigmoidal activation function versus the SPAF-network, along with the affects of L1 and the newly proposed log-penalty term. 62

4.4 Comparison of the average activation in the first layer of Sigmoidal-model versus the SPAF-network, along with the affects of L1 and the newly proposed log-penalty term. 63

4.5 Comparison of the average activation in the second layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly propsed log-penalty term. 64

4.6 The individual dataset in Chars74k were formatted in a tree folder structure, with the top level folder indicating one of the 3 folders, and each subfolder indicating a category. 65

4.7 Comparison of Sigmoidal-model versus SPAF function, along with the affects of L1 and the newly propsed log-penalty term on the Img Dataset. 67

4.8 Comparison of the average activation in the first layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly propsed log-penalty term on the Img Dataset 67

4.9 Comparison of the average activation in the second layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly proposed log-penalty term on the Img dataset. 68

4.10 Comparison of Sigmoidal activation function versus the newly proposed swapped activation function, along with the affects of L1 and the newly propsed log-penalty term on the Hnd Dataset. 69

4.11 Comparison of the average activation in the first layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly propsed log-penalty term on the Hnd Dataset 69

4.12 Comparison of the average activation in the second layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly propsed log-penalty term on the Hnd dataset. 70

4.13 Comparing the regular Sigmoidal-model over the SPAF-network for their performance over different number of neurons in the hidden layers 70

4.14 Comparing the regular Sigmoidal-model over the SPAF-network for their performance given a percent of the total dataset for training. . . 71

4.15 Converting Weights of neurons in the first layer to feature images. . . 73

4.16 Features of the first layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, in that respective order. Trained on the MNIST dataset. 77

4.17 Features of the second layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, all in that respective order. Trained on the MNIST dataset. 78

4.18 Features of the first layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, in that respective order. Trained on the Fnt dataset. 79

4.19 Features of the second layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, all in that respective order. Trained on the Fnt dataset. 80

4.20 Features of the first layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, in that respective order. Trained on the Hnd dataset. 81

4.21 Features of the second layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, in that respective order. Trained on the Hnd dataset. 82

List of Tables

2.1 A and B represent two binary features, followed up by output for an OR, AND and a XOR function. 11

2.2 The text This is an example broken into n-grams. 24

Chapter 1

Introduction

1.1 Introduction

In our modern era, we are increasingly relying on intelligent machines to help us carry out various tasks. These tasks range from mundane tasks such as vacuuming the house to complex security threatening problems such as automating the task of recognizing suspects in thousands of images. These intelligent machines are possible due to the advances in the field of Artificial Intelligence. Artificial Intelligence broadly speaking is a field in Computer Science and Engineering, as such it can be further divided into various distinct sub-fields. Some of these fields pertain to vision, speech, object detection, scene detection, motion detection, navigation and dozens of others. Many of these fields and in particular, speech and vision recognition belong to an area called Machine Learning.

Machine Learning in essence is using computational methods to study data for patterns, then using these learned patterns to make predictions on new data. At

a very basic level, it can be equated to a line of best fit on a scatter plot. Where the x-axis are the inputs and y-axis are outputs, by drawing a line that best fits the pattern of x to y, we can then extrapolate and make predictions on undocumented (x,y) points.

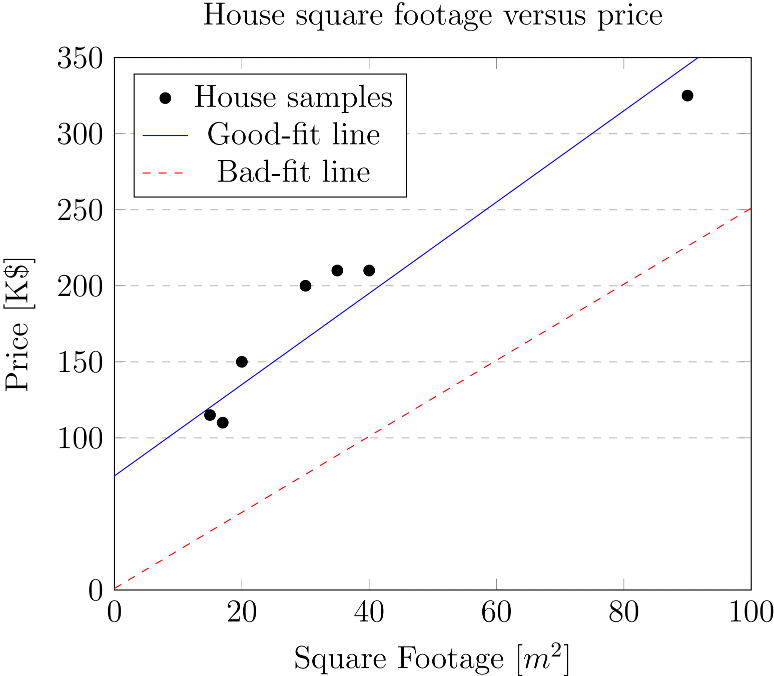


Figure 1.1: Line of best fit attempts applied to house square footage (x) versus price (y)

An example of when this is useful can be found in Fig. 1.1 where the square footage of houses are plotted against the prices at which they were sold at. If an individual wanted to use this information to price a new house, then a line of best fit can be drawn and used as a guidance for pricing a new house given its square footage. The process of drawing a line that fits the data best can also require some effort, and can have an effect on how reliable of an estimate one may recieve on the price of the new house. In Fig. 1.1, the solid line would be selected as the line of best fit. While the task may seem trivial in this specific example, since it is visibly clear that the solid line is closer to all the points than the dashed line; this however may

not always be the case. One can imagine that simply taking one variable (size of the house) maybe insufficient for pricing. Other features or attributes of the house are also important when pricing, for instance location, number of rooms, bathrooms and etc are all very important factors. As such it especially becomes hard when dealing with multiple dimensions and thousands upon thousands of points, to a) draw lines and b) select the line of best fit. The math and comparison of different lines become extremely difficult, as such needing an algorithm to automate the selection of a line that fits the dataset best becomes important.

Learning Algorithms

Another way to look at the previous example is to think of the line of best fit as a function that takes house square footage and produces a price. The solid line in Fig. 1.1, would have the form outlined in Eq. (1.1). So instead of manually plotting new points on the line of best fit, one can use the equation of the line to compute the prices. This whole process in essence is what a Learning Algorithm in Machine Learning would do. In fact, this exact process of line generation and the selection of the line that fits best is called Linear Regression. In Linear Regression an equation in the form of Eq. (1.2) is manipulated till an equation that approximates the data points best is computed. This new equation is different from a simple line such as the one in Eq. (1.1) in that it can take multiple features. The bold variables namely $\mathbf{w}$ and $\mathbf{x}$ are vectors. The $\mathbf{w}$ and b are manipulated in the direction that minimizes an error. There are various ways to measure the error, a simple method is equivalent to measuring the sum of all distances between each point and the line. The line that provides the smallest error then is selected as the best-fit line. There are a variety of other Learning Algorithms that one can use when needing to learn from data. Other learning algorithms include, Logistic Regression, (SVM) Support

Vector Machines and Neural Networks. Selecting which one to use is dependent on a variety of factors, some of which are: availability of resources on a system, type of system/platform, data and even type of Machine Learning problem. This example falls under a regression problem, simply put, the prices for the houses can be any real number. Problems such as number recognition in images fall under classification problems, classification problems have to classify input into a distinct category. For instance, an image with a handwritten nine should be classified by the algorithm as a nine. Logistic Regression, SVM and Neural Networks are capable of solving classification problems.

$$f(x) = 3x + 75 \tag{1.1}$$

$$f(\mathbf{x}) = \mathbf{x}\mathbf{w}^T + b \tag{1.2}$$

The focus of this thesis will be on Neural Networks. The ideas for modern neural networks stemmed out of research focused on the computational model of biological neurons since the early 1940s, see the work of McCulloch and Pitts in (1). Since then neural networks have gone through ups and downs, especially when it comes to academic research focus. Neural networks after the advent of Support Vector Machines took the backseat in the machine learning community. Research continued in various labs even during this low period. For example neural networks were used in experiments with complex input with the idea that it can learn its own features, as seen in (2). It wasnt until the early 2000s, when SVM plateaued on many problems, that Neural Networks once more gained popular support. Neural Networks can be topologically designed such that they can learn better features from the input. It

was exactly progress in regards to automatic feature learning, that brought neural networks back to the spot light (3). In one of the more popular cases, Google was able to train a network to recognize the face of a cat from Youtube videos (4). They can now be found in Apple, Google and Microsoft products that rely on speech and image recognition.

1.2 Data, Datasets and Functions

In the previous sections, Machine Learning was defined to be the study of patterns in data, then using those patterns to make intelligent decisions, which is not completely accurate in all Machine Learning problems. In some problems, given an input x and output y combinations, a function needs to be constructed that can accurately produce the output given the input. In other words, a function that can map $x \to y$, lets call this function $f()$. This function is often an approximation of the true function $F()$. Approximating $F()$ is useful as it allows us to compute outputs y for previously unseen inputs x. This is the case in the house pricing example in the previous section. Historical data was used as samples of the true $F()$, and the line equation constructed from this historical data can now be used to price houses that haven't been sold yet. A note worth making, is that while no mathematical procedure for learning structure and patterns have been discussed yet, combining the two types of learning, the learning of patterns from x and then using that knowledge to better map $x \to y$ can be very powerful. This idea will further be explored in Chapter 3.

In Machine Learning, the historical data collected is called a dataset, the features are used as inputs, and that which needs to be predicted is called an output. Any complete (x, y) datasets used to compute a function $f()$ that can map $x \to y$, is called a training set. And often, to test the effectiveness of $f()$, samples set aside and not

used in the training set are used, this set is called the Testing set. In some cases, a separate Validation set is used to direct the Learning Algorithm towards the true $F()$.

1.3 Major Contribution

The goal of this work was to explore methods that could accelerate pre-training with autoencoders a variant of neural networks, and improve hidden layer representations. As such, a total of three separate areas were covered in the overall work. A new modified activation function was introduced during pre-training with autoencoders, this method was picked to increase gradients and therefore increase the rate of neuron saturation. Secondly, to the best of our knowledge, for the first time with pre-training and fine-tuning methodologies, a different activation function was used in the pre-training stage versus the fine-tuning stage, networks trained with this method we call SPAF-networks (Swapped Pre-training Activation Functions). Finally, methods that induced sparse representations in the hidden layers of denoising autoencoders were explored, this was done by using an L1 and log based term of the hidden representation and added to the cost function as a regularization term. A paper using this work has also been published (5).

1.4 Thesis Outline

The rest of this thesis will be divided as follows: In Chapter 2, Neural Networks will be covered in more detail, various application areas will be discussed along with various feature selection techniques.

In Chapter 3, the new work and how it relates to and helps unsupervised feature learning in denoising autoencoders will be discussed. Chapter 3 will begin with mathematical preliminaries will followed by the detailed explanations of supervised and unsupervised networks.

In Chapter 4, the datasets used to test the algorithm will be discussed in detail, followed by implementation details of the algorithms. The end of the chapter will be reserved for mentioning experimental results on the datasets.

Finally, Chapter 5 will be used to outline the conclusion of the work as well as ideas and suggestions for future work.

Chapter 2

Background

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are biologically inspired machine learning algorithms, that are capable of approximating very complex functions. Networks that are considered to meet a certain requirements (networks with at least one-hidden layer), are in fact universal function approximators (6). Meaning they are capable of approximating any function given enough samples of required function. Due to the biological inspiration, the architecture and naming conventions of the building blocks of an artificial neuron mimic those of a biological neuron. A biological neuron is composed of three distinct parts, the dendrite (incoming signals), the cell body and finally the axon (outgoing signals). They receive a signal through the dendrite, decide on whether to propagate it forward in the soma (body) and finally propagate the signals forward through the axon (7).

2.1.1 Artificial Neuron

An artificial neuron similarly can have a node with incoming (dendrite) and outgoing (axon) connections as seen in Fig. 2.1. Each connection has an associated weight, which is used to multiply the incoming value by, the artificial neuron then takes the value and will alter it in some way, before propagating the output further on as final output, or an input to other neurons.

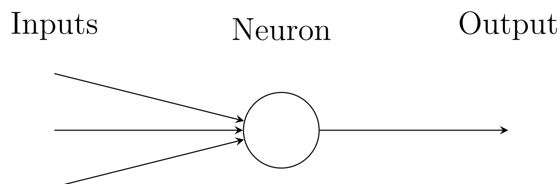


Figure 2.1: Artificial neuron with multiple inputs will produce one output. However this output can be connected to multiple neurons, much like how biological neurons work.

Each neuron can have different functions to apply to its incoming signal. These functions are called activation functions, there are many different kinds of activation functions, a few of the popular ones are: Logistic function (2.1), Step function (2.2), Hyperbolic Tangent (2.3), and Rectified Linear Unit (ReLU) (2.4). These functions are in place to mimic the action potential found in biological neurons. When an action potential is reached the neurons fire a signal, which is propagated forward. While these activations functions are not as complex as the real biological ones, they still regulate the strength of the output.

$$\phi(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

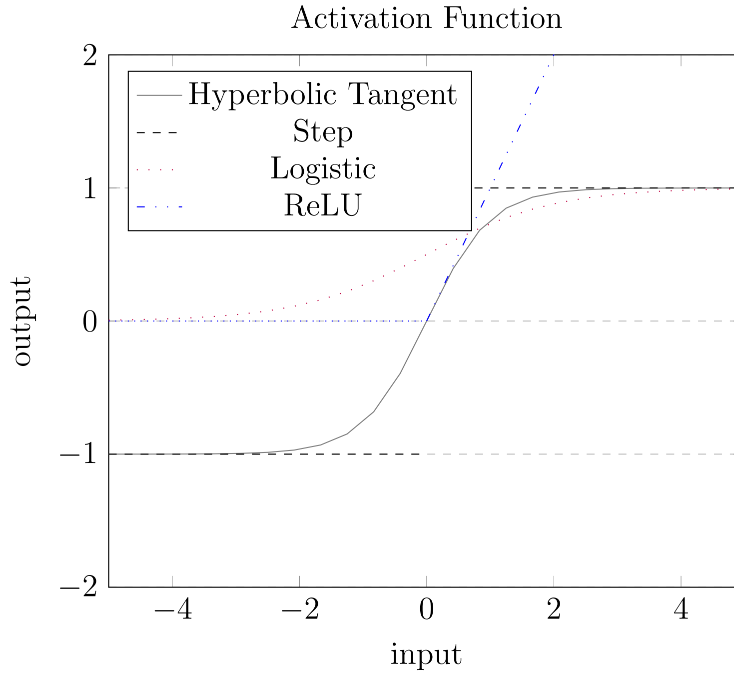


Figure 2.2: Some common activation functions that can be found in neural networks

$$\phi(x) = \begin{cases} 1 & \text{if } x \text{ =¿ } 0 \\ -1 & \text{if } x \text{ ¡ } 0 \end{cases} \tag{2.2}$$

$$\phi(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{2.3}$$

$$\phi(x) = \begin{cases} x & \text{if } x \text{ ¿ } 0 \\ 0 & \text{if } x \text{ ¡= } 0 \end{cases} \tag{2.4}$$

Like a biological neuron, an artificial neuron can have multiple incoming connections as seen in Fig. 2.1, it deals with this by simply adding up all the signals prior to applying an activation function. This can be seen in Eq. (2.5); Where $f_j(x)$ represents the jth neuron and ϕ, x, w and b represents the activation function, input, weight and bias values respectively. Inputs are first multiplied by the weights of the

connections before being summed and potentially squashed by an activation function. It is also worth noting, Eq. (2.5) is identical in every aspect but the addition of the activation function ϕ, to the Linear Regression equation seen in Eq. (1.2) discussed in Chapter 1.

$$f_j(x) = \phi(b_j + \sum_{i=0}^{|x|-1} x_i w_{ij}) \tag{2.5}$$

2.1.2 A Network of Neurons

This building block, like the biological neuron, can be connected with others to create complex topologies that can compute different functions. To illustrate this point, an OR, AND and XOR functions will be constructed using neural networks and explained. These functions are popular in the industry to showcase the ability of neural networks in separating non-linear problems. Richard Bland outlines the functions in more detail in (8). To better understand the functions, Table 2.1 can be analyzed. Two binary variables are taken as input and a single binary output is produced.

A	B	OR	AND	XOR
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Table 2.1: A and B represent two binary features, followed up by output for an OR, AND and a XOR function.

An OR will produce a 1, if any of the two features is set to a 1. To produce the right output for an OR function using a neural network, weights of value 1 can be

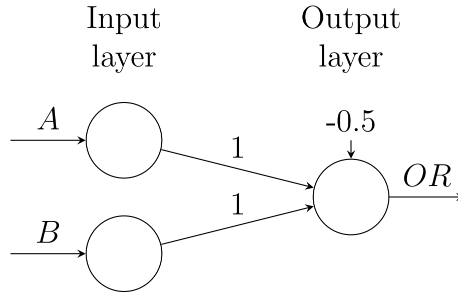


Figure 2.3: OR Network

used with each connection to one output node, this is illustrated in Fig. 2.3.

To assure that the right output is produced, an activation function that simply outputs zero or one needs to be put in place. A custom activation function of the form in Eq. (2.6) can be used.

$$\phi(x) = \begin{cases} 1 & \text{if } x \text{ ¿ } 0 \\ 0 & \text{if } x \text{ ¡= } 0 \end{cases} \tag{2.6}$$

With this last piece in place, Eq. (2.5) can be used to compute an output using the network in Fig. 2.3 as follows:

$$\begin{aligned} Input : [0,1], Weights : [1,1], Bias : -0.5 \\ f([0,1]) &= \phi(0 + \sum_{i=0}^{1} x_i w_{ij}) \\ &= \phi(-0.5 + ((0 * 1) + (1 * 1))) \\ &= \phi(0.5) \\ &= 1 \end{aligned}$$

This equation can be used again for all the inputs and it will produce the right output. To construct an AND function, the bias will have to be adjusted to a -1 on the output node. Doing this will increase the required activation rate to a 2 before the bias is applied. Therefore both inputs (A and B) will need a value of 1 for the neuron to activate. This network can be seen in Fig. 2.4.

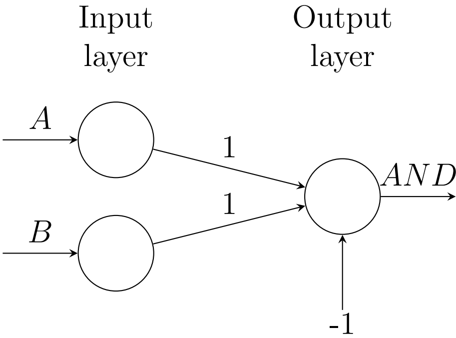


Figure 2.4: AND Network

2.1.3 Hidden Layers

Before the network for the XOR function can be created, new terms for more complex topologies will need to be mentioned. In ANNs, the majority of the topologies follow the feed-forward neural network topology. Neurons are grouped such that, they are layered with lower layers feeding output to the next layer sequentially, this process is called feed forward. In regards to the number of layers, there is no limit on how many there can be in a network.

The first layer is called an input layer, the layers immediately following the input layer are called hidden layers, and is often simply clamped with the values from the input vector, as has been the case in previous examples. Fig. 2.5 is a 1-hidden layer network. As is evident, there are no intra-layer connections between neurons. This

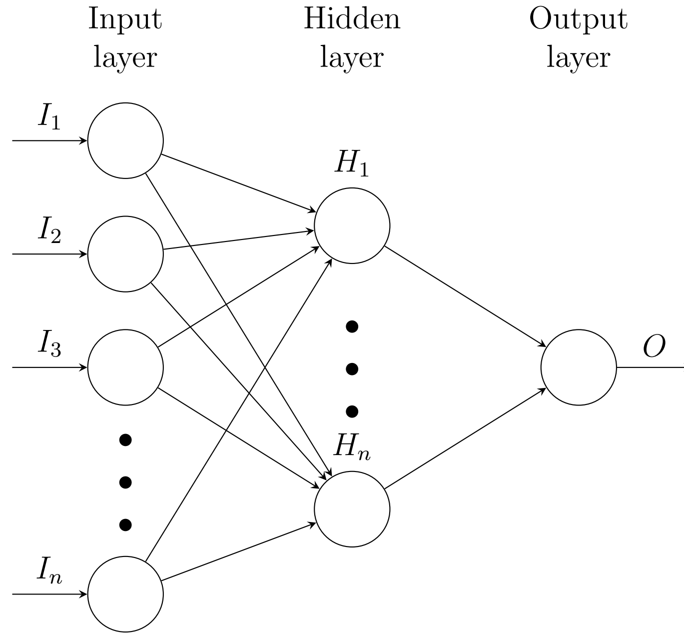


Figure 2.5: Feed-Forward neural network has input nodes that simply take the value of each input, and the following layers up until the last layer (Output Layer) are called hidden layers.

architecture produces a simple sequential process to compute values for each layer. For example, given values for the input layer (simply clamp the values of inputs, or use the input vector), the first hidden layer can be computed using the following equation:

$$f(x) = \phi(\mathbf{x}\mathbf{W} + \mathbf{b}) \tag{2.7}$$

Where $\mathbf{x}$, $\mathbf{W}$ and, $\mathbf{b}$ and the input vector, weight matrix that represents all the connections between input neurons and hidden neurons, and finally the bias vector representing biases for each hidden neuron respectively. In fact this equation can be applied to any layer in the network, where the input, $\mathbf{x}$ is simply the output of the previous layer. An important point to make here, is the representation of connection weights between layers in the form of a matrix. By doing so, the summation from

Eq. (2.5), can be replaced by a simple dot product between **x** and **W** in Eq. (2.7). The matrix is constructed by simply placing indexes of neurons in one layer as rows, and the next layer as columns. The values in each group of index is the weight of the connection between the neurons that the indexes represent. Fig. 2.6 is a network with 3 input neurons, 2 hidden neurons and 1 output neuron. Its corresponding matrix for the connections between the input layer and hidden layer is as follows:

$$\begin{bmatrix} 0.1 & 0.1 \\ 0.2 & 0.2 \\ 0.3 & 0.3 \end{bmatrix}$$

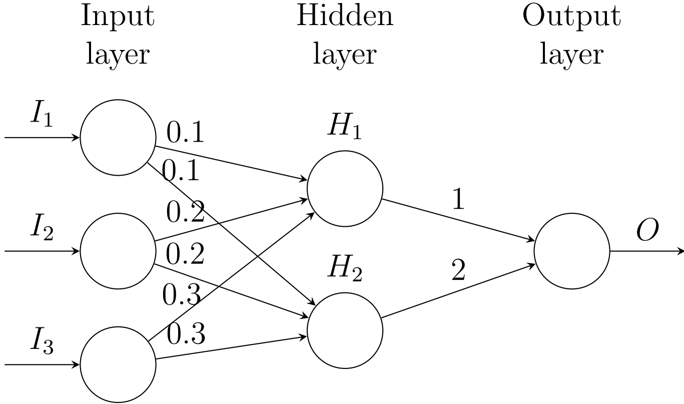


Figure 2.6: A network with three input neurons, connected to two hidden neurons all feeding output to a final output neuron.

2.1.4 Intermediate Representation

Both the OR and AND function can be separated by a simple line as seen in Fig. 2.7, and Fig. 2.8, this is unlike the XOR function seen in Fig. 2.9. To construct a network that can mimic the XOR function, the input needs to be modified to an

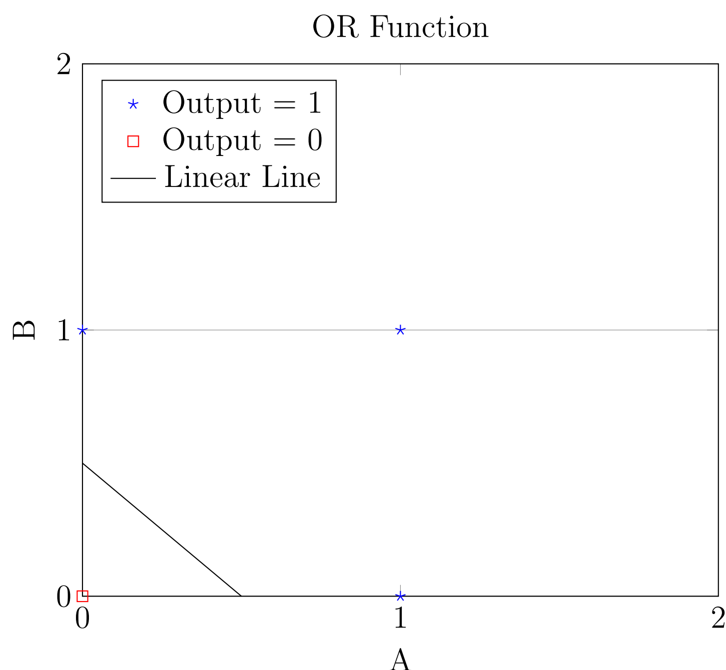


Figure 2.7: OR Function displayed in a scatter plot. Blue stars marks indicate an output of one, and red squares indicate an output of zero. The OR function can be linearly separated as indicated by the black line.

intermediate representation that is separable. One way to achieve this, is to simply add a hidden layer that can represent the intermediate representations.

By the simple definition of a XOR network, or by analyzing Table 2.1, one can see that the XOR function simply activates for problems where an OR function activates but not an AND function. By this definition it can be easily seen that the two networks seen in Fig. 2.3 and 2.4 if combined as input to another network will give this new network the simple ability to check for an OR and not an AND; to ultimately allow for the introduction of XOR logic into a network. What this means is that the outputs from the OR and AND networks will represent the neurons in the hidden layer of a XOR network. These hidden neurons, then become intermediate features for the XOR function. Following the logic discussed earlier, the outgoing connection from the AND neuron can have a weight of -1, which means if the AND

neuron activates a value of -1 will be produced (dislike of an AND presence). If it does not activate, a value of 0 will be produced. The OR neuron can have an outgoing connection of 1, if it activates it will produce a 1, if not a 0. Hence if both are on, $-1+1$, will produce a 0 at the output, and if only the OR neuron is on, then $0+1$, will produce a 1. Simply, what the output node checks is for the presence of an OR neuron (weight of 1 from OR neuron) and not an AND neuron (weight of -1 from AND neuron) to activate; this is represented by the red and green lines in Fig. 2.10.

It is important to note, that a neural network produces a function, as can be seen by analyzing the inputs and weights of the various networks constructed above. Consider the OR function, with weight values of $[1, 1]$, we get the equation:

$$A + B - 0.5 = 0$$

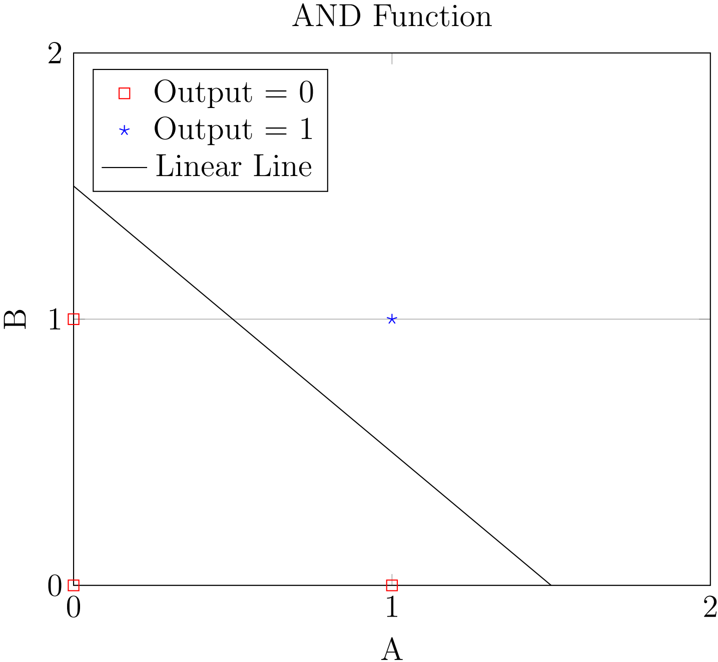


Figure 2.8: AND Function displayed in a scatter plot. Blue stars marks indicate an output of one, and red squares indicates an output of zero. The AND function can be linearly separated as indicated by the black line.

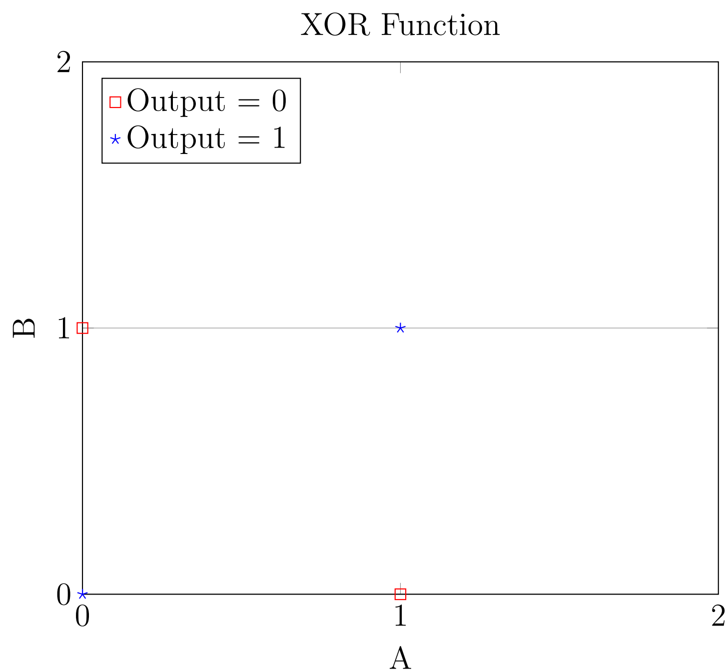


Figure 2.9: XOR Function displayed in a scatter plot. Blue marks indicate an output of one, and red indicates an output of zero. The XOR function cannot be linearly separated by a line.

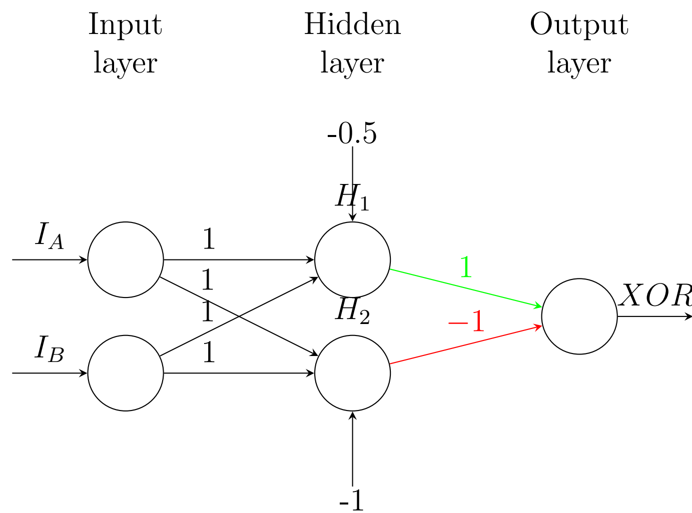


Figure 2.10: XOR Network with OR and AND networks

at the output node. This equation is simply the equation of a line, the activation function changes this to:

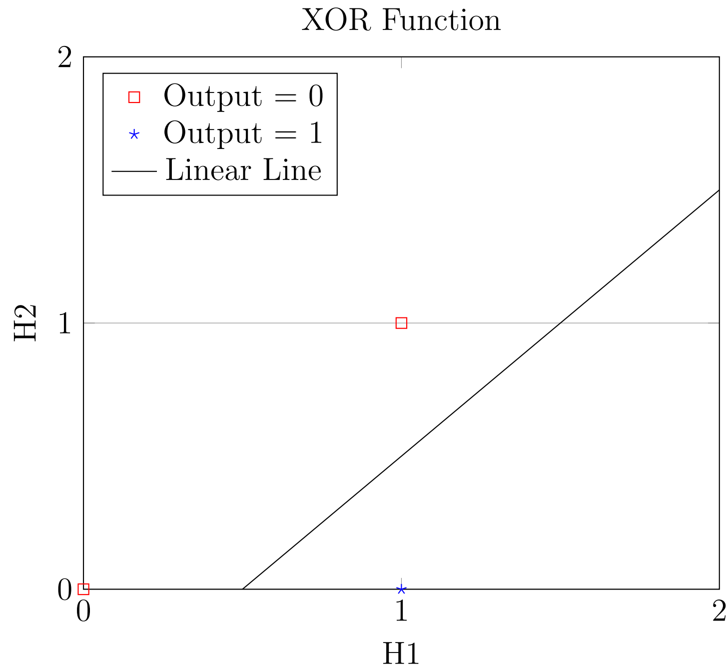


Figure 2.11: All possible values of H1 and H2 from 2.10 plotted on a scatter plot. The problem is linearly separable.

$$\phi(x) = \begin{cases} 1 & \text{if } A + B - 0.5 \text{ ¿ } 0 \\ 0 & \text{if } A + B - 0.5 \text{ ¡= } 0 \end{cases} \tag{2.8}$$

Of course constructing more and more complex networks enable us to mimic more and more complex functions, as was the case with the XOR function. While these networks were hand constructed, in Chapter 3, a method called Back Propagation will be discussed, which enables the automatic learning of weights for neural networks. This makes neural networks extremely powerful in their ability to approximate and build all kinds of complex functions. Another important point to make, the addition of hidden layers, enabled us to add logic gates within the network, which were then used as features in the final output layer. One can imagine, adding more layers, which would then create even more complex features built on top of features to feed to the final output layer.

2.2 Applications for ANNs

ANNs are unique in that they can be applied to many kinds of problems. In the industry they can be found in a wide variety of areas, from Robots that serve coffee (9) all the way to powering speech recognition software in Android phone devices (10). It is worth mentioning however, that finding ANNs in the indusrty is a recent phenomenon. It is due to the recent progress in neural networks in the academic field that pushed the industry to realize their potential. Much of the chain of developments can be attributed to a new training methodology developed by Geoff Hinton at the University of Toronto, which was soon discovered to work well in initializing the weights of neural networks. This sparked a series of applications in the industry and also produced new record successes on image recognition tasks such as ImageNet. In fact ImageNet over the past 5 years has seen a consistent improvement over its classification accuracy. ImageNet has seen a trend of more and more teams utilizing neural networks. In 2005 no teams used a neural network, in 2012 every team that participated in the ImageNet competition had used a neural network (11). Looking at the details of some of the applications, outlines a new kind of problem. Previously architectures and examples pertaining to the regression problem were discussed. These industry examples all have an aspect of classification in them. A classification problem in Machine Learning, takes an input and assigns it into a category, the OR, AND and XOR functions constructed in the previous section were all also classification problems. Those problems had a possible output of two categories or classes. In the case of speech recognition, taking audio as input would be run through a network, which would in turn assign the audio into one of several outputs that could represent words.

2.2.1 Multi-Class Classification

Multi-Class classification refers to machine learning classification tasks that have higher than two categories. In neural networks, two categories can be classified by a network that has one output, this can easily be seen in the examples discussed in the previous section. In those scenarious an output of greater than 0 was set to one class and an output of 1 of the same neuron was considered another class. To enable multi-class classification, more than one output neuron can be used. This difference can be seen by comparing Fig. 2.5 versus 2.12.

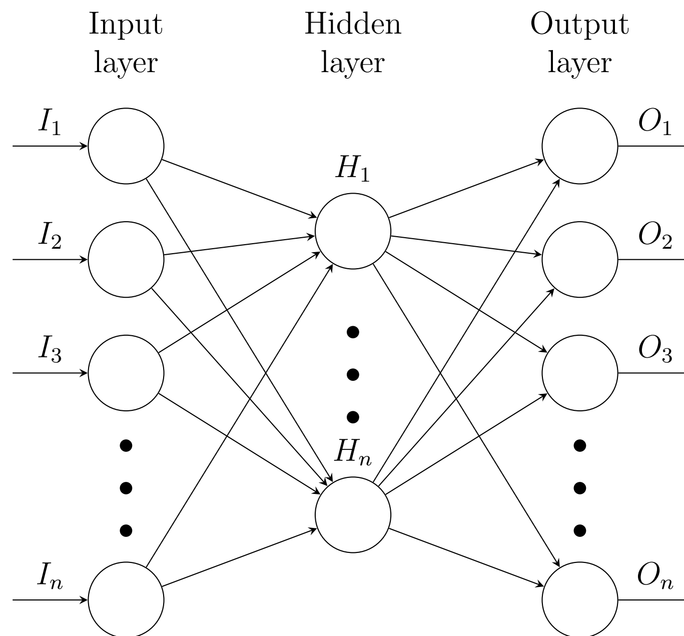


Figure 2.12: In a classification problem, neural networks are designed to have multiple output neurons, each representing a category. In this case, the neural network is designed to categorise input into n categories.

However with multiple outputs a method to select one of the many outputs is needed. One such method is called Softmax. Softmax is a two step process, the first step is applying the Logistic Activation function seen in Eq. (2.1) in all output neurons, the values are then taken and converted to probabilities using the following equation:

$$Pr(z)_j = \frac{z_j}{\sum_{i=0}^{n} z_i} \tag{2.9}$$

Where z is the output vector from a given network, and z_j is simply the value in the jth index of the vector. Softmax is a popular function that is used in multiclass neural network topologies (12). The function Pr(), will produce a new vector with probabilities, this in fact is more powerful than the hard classifier that was used in previous examples. With softmax, we can attribute probabilities to input belonging to a certain class, we can also choose to attribute multiple classes to an input with probabilities as certainties. A class or category is represented by the index of the output vector, this needs to be predefined, and often the dataset with input and output groupings are adjusted such that each class is represented by the proper index. Consider the following example with a random array representing a cat, dog and a wolf, in [Input], [Output] format.

$$[0.1, 0.3, 0.4, 0.9], [cat]$$
$$[0.3, 0.9, 0.1, 0.2], [dog]$$
$$[0.6, 0.1, 0.7, 0.3], [wolf]$$

The first step to converting this to be compatible for a softmax neural network, would be to count the number of different classes. Which in this case is three, a vector of size three is then created to represent the classes, and each index represents a unique class. In this case, position zero for cat, one for dog and two for a wolf. This information is then used to construct a one-hot vector. Which simply means to set a value of 1 in the corresponding position to represent a class in the vector. The output vectors

would then look like this:

$$[0.1, 0.3, 0.4, 0.9], [1, 0, 0]$$

$$[0.3, 0.9, 0.1, 0.2], [0, 1, 0]$$

$$[0.6, 0.1, 0.7, 0.3], [0, 0, 1]$$

In most problems, the norm is to assign the input the class with the highest probability. This simply translates to selecting the neuron with the highest probability, returning the index of that neuron as the class. From the previous examples if the network returned 2, the class wolf would be assigned to the input that produced that output.

2.2.2 Document Classification

An example of where multi-class neural networks can be applied is document classification. The goal of the network is to classify a page into one of several categories. This is useful in a wide variety of ways, in a search engine this can be used to return webpages that are directly relevant to the topic the user is researching. Similarly the technology is being used for delivering a much more relevant online advertisement in the Targeted Advertisement Industry. This is done, by first classifying the web page the user is on, into one of potentially thousands of categories, then matching an advertisement to the category of the web page. This is called Contextual Targeting (13).

Another name for document classification is text classification, and as such, to classify documents or text, the individual words need to be formatted such that neural networks can process them. To humans, understanding documents is simply

the task of reading the words, for neural networks that can only process numbers, words will need to be converted to numbers. The process of generating inputs for machine learning is called feature engineering, in other words, it means document representation (14). A popular method for document representation is called bag of words. In this method a document is represented as a set of words together with their associated frequency or occurrence in the document (14). Bag of words is also often referred to as n-grams, where n stands for the number of words in a single feature. An example of n-grams can be seen in Table 2.2.

Feature	1-gram	2-gram	3-gram
0	This	This is	This is an
1	is	is an	is an example
2	an	an example	
3	example		

Table 2.2: The text This is an example broken into n-grams.

The bag of words method can be broken into the following steps:

(1) Create the desired n-gram table using the whole corpus (dataset). This often means, an n-gram table representing every word found in the vocabulary of the corpus.

(2) For each feature z from step one, create a vector for each document where the index z has the frequency of feature z in the document. This vector then becomes the input $\mathbf{x}$.

(3) For each input $\mathbf{x}$ (representing document), create an associated 1-hot output vector $\mathbf{y}$, to represent the category of the document input $\mathbf{x}$ represents.

Once this is complete, documents will be represented in the format $(\mathbf{x}, \mathbf{y})$, ready to be further processed.

2.2.3 Image Classification

Image Classification or Image Recognition is the task of identifying the objects or persons in an image. This is useful in a wide variety of ways, consider image recognition at security institutions, Facebook also employs Image recognition for its tagging tips (15). Unlike humans, computers see an image as pixels, as such the input to a learning algorithm is the individual pixels. Unlike document classification, the processing of images as input is a little simpler, and can be processed as follows:

(1) Convert each image to a matrix, where each position in the matrix directly corresponds to the pixel intensity in the image.

(2) Convert the matrix to row-major vector $\mathbf{x}$, by appending all of the rows from the matrix together in the order of first to last.

(3) For each input $\mathbf{x}$ (representing each iamge), create an associated 1-hot output vector $\mathbf{y}$, to represent the category of the image input $\mathbf{x}$ represents.

Once this is complete, images will be in the right format for a classification algorithm, that takes $\mathbf{x}$ as input and expects $\mathbf{y}$ as output.

2.3 Feature Selection

Feature selecting is an integral part of machine learning, it is often used to reduce computational complexity and in many cases to improve performance. Feature selection is the process of selecting a subset of the features used in a learning algorithm. Consider the case of document classification, if a large dataset is ever used,

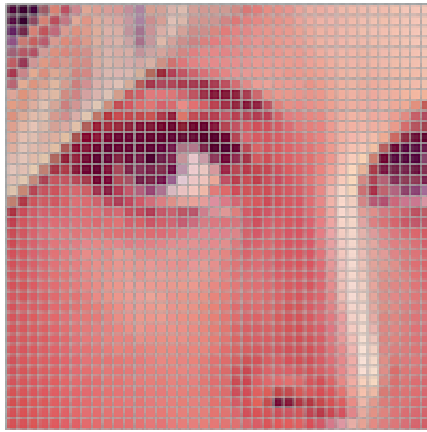


Figure 2.13: Unlike humans, computers cannot process the images with the complexity of all of its features. It simply sees pixels that make up the image.

then the size of the vocabulary might get large enough to increase the size of the input vectors to the point where it is infeasible to store, use and process in a Learning Algorithm. This is easy to see when considering that a website like wikipedia has over 2.6 billion words (16), using a 1-gram bag of words method will give us a vector of size 2.6 billion for each wikipedia page! Feature selection is also used to reduce the noise in a dataset which will ultimately increase the classification accuracy of a model. Noise can be thought of as features that reduce the accuracy of a model in its ability to map unseen inputs $\mathbf{x}$ to $\mathbf{y}$ (17). Consider training a document model by using documents written by a few individuals, each individual's name appears at the end of each document. If for instance, the documents that belong to the category sports were all written by Joe Smith, then the model could erroneously assume that the feature Joe Smith is very telling of the category sports. If the model were to ever get deployed and be asked to classify a politics document written by Joe Smith, it might erroneously classify it as a sports document.

2.3.1 Existing Methodologies

There are many feature selection algorithms, two of the popular methods are TF-IDF (term frequency inverse document frequency) and χ^2 (Chi-Square). TF-IDF measures the relevance of a word to a document in a corpus, this has widely been useful in machine learning as well as query retrieval, where it has increased performance as seen in (18). As indicated by the name, TF-IDF is a multi-step algorithm and can be broken down into the following steps:

(1) The first part of TF-IDF is TF or term frequency. The simplest approach is to count the frequency of word w in document d, denoted as $tf(w, d)$.

(2) The second step is computing IDF or inverse document frequency and it is measured by dividing the total number of documents in the corpus, by the total number of documents that have the word w. IDF is logarithmically scaled, so the final form of the equation is as follows:

$$idf(w, D) = \log \frac{N}{(1 + |d \in D : w \in d|)} \tag{2.10}$$

Where D denotes the corpus and N is the total number of documents in D. one is added to the denominator to avoid a division by zero.

(3) The final step is combining the two parts to compute TF-IDF, and that is done by simply multiplying the two functions:

$$tfidf(w, D) = tf(w, d) \times idf(w, D) \tag{2.11}$$

tf-idf is computed for every feature in a corpus and is then used as a dimensionality reduction tool. This is done by keeping features with high values and discarding

features with lower values. Similarly χ^2 can be used to lower dimensionality by scoring all of the features in a dataset. Its different in that, while TF-IDF measures the relevance of a word to a document, χ^2 measures independence of two variables. In document classification, a words independence can be measured in regards to a specific category. The first step of χ^2 is calculating a total of four variables:

(1) A = The number of times w is found in the category c (co-occurance).

(2) B = The number of times w occurs outside of category c.

(3) C = The number of times c occurs without w, (number of documents without the word w).

(4) D = Number of times neither c nor w occurs (documents without c or w).

(5) N = Total number of documents in corpus.

The scoring is then calculated for each word in the corpus by the following formula:

$$\chi^2 = \frac{N \times (AD - CB)^2}{(A + C) \times (B + D) \times (A + B) \times (C + D)} \tag{2.12}$$

Higher scores in χ^2 mean that the two features w, c are not independent, in other words they should be retained in the final feature list. This is why χ^2 is often described as a measure of the lack of independence between two variables (19).

Considering that machine learning is applied to a wide array of different areas, there are similarly a wide array of feature selection algorithms that have been developed. What is more interesting to mention is the idea of abstracted features to

different fields. The methods described so far, work well with the Document classification and potentially many other problems. Where it is lacking is Image Classification, where instead of high level features like words, the individual pixels are provided. Words are high level human features that have meaning. These meanings translate well to categories, so applying independence analysis is logical, doing the same on pixels would produce very noisy results on images. Even if pixels were good features, any small variation in the image would cause drastic differences in the vectors. Consider an image with a human face, pixels that are found in the eye region would be very telling of the presence of an eye; a slightly different pose of the same individual would translate the effect of that pixel to a different area causing noise in the model. Word level features in images would translate to concepts such as an eye, mouth, brow and etc for a human face. Window, door, and bell for a house. Of course feature engineering at this level for images can be extremely difficult, these will have to be hand constructed which is a futile task due to the sheer number of different features that can be found for all objects. What has been done is the introduction of mathematical methods that can create a list of features from images (20), similarly in Speech Recognition there are many different kinds of feature selection algorithms that are employed (21). What would be far more interesting is a method that can automatically be applied on the most basic building block features (pixels) and learn high level features that can enhance accuracy levels in classification tasks.

2.3.2 Deep Learning

With recent breakthroughs in our understanding of how the human brain works, new development methodologies have been proposed and developed. One such breakthrough falls under Deep Learning, it borrows a central idea from the human brain that deals with multiple levels of representation of the sensory input. For instance the

visual stimuli is converted to various levels of representation with increasing complexity before it reaches the visual cortex (22). This phenomenon makes Deep Learning very interesting when considering the various problems with the conventional methods discussed earlier. Deep Learning will accept a raw input and transform them to a better representative feature space of the problem. It also can be used in a wide array of fields, Deep Learning has been applied in Text Classification, Image Recognition/Classification, and Speech recognition. In fact in Image and Speech recognition it has replaced all conventional methods and is widely being used in the iPhone, Android, and Windows phone platforms. Deep Learning can further be divided into several different algorithms, a few of the popular algorithms are Autoencoders, Deep Belief Networks and Deep Convolutional Networks. The focus of this work will be on AutoEncoders, which will be discussed in detail in the next Chapter.

Conventional methods like TF-IDF and χ^2 have proven to be useful in some areas; they however are not a universal solution. They are also inherently a linear transformation tool, and as the size of the datasets increases we will increasingly need non-linear tools to represent high dimensional data in a useful way. It is also clear, that while hand-engineered features have helped various fields, they are not the most effective way to move forward. As such, recent research has heavily been focused on deep learning solutions that are inherently non-linear. It is thus very reasonable that research will continue to be focused on these areas, and inspiration will be drawn from the multi-level representational ability of the human brain to develop complex self-learning feature selection algorithms.

Chapter 3

Sparse and Saturating Neurons

3.1 Preliminaries

3.1.1 Cost Function

In the previous chapter, a few different neural networks were created, each due to their topology and weights produced a different function. As such, if the weights can be intelligently or heuristically modified towards a desired function, learning can be achieved. An important piece for this to work is to assess the impact of a change in weights on the final output of the function. Given the definition of Machine Learning in Chapter 1, the impact can be measured in terms of the actual function F, and the prediction of the learning algorithm f. The lower the difference between the two, the closer we are in regards to learning. Given that the actual function F is unknown, observations of F can be taken (from the dataset), and the performance of f on that dataset can be compared. The formula that measures the performance of f is called a cost function. A popular cost function is known as Mean Squared Error (MSE) and

is measured as follows:

$$f_{error} = \frac{1}{D}\sum_{j=0}^{D}\sum_{i=0}^{N}(f_{net}(x_j)_i - y_{ji})^2 \tag{3.1}$$

Where f_{net} is the output of the network, x is a given sample from the dataset and y is its associated expected output value and N is the demensionality of the output vector, which is also the same as the number of classes. The equation can be used to compute one error at a time, or the error of a network on the whole dataset. In the case of computing the error for anything larger than one sample, the mean of the errors is taken; D in that case represents the total number of samples used to compute an error. MSE will produce a lower score when f_{net} makes close predictions to y. In other words, the higher the classification accuracy of the network, the lower the score of MSE. Given a method to modify weights, weights can be updated such that f_{error} is minimized, in this manner gradual learning can be achieved. An algorithm that mimics this process is outlined below.

(1) Initialize weights randomly.

(2) Compute f_{error} with (x, y) and f_{net}

(3) If f_{error} is acceptably low, stop. Return current f_{net}.

(4) Else update weights towards a better estimate.

(5) Repeat from step two.

Another cost function that can be used in classification problems is the Mean Negative Log-Likelihood (NLL), this function deals with increasing the probability of producing the correct class. Recall that in Chapter 2, a softmax funciton was

introduced and explained to be akin to producing probability for every possible class. That is to say: $P(y|x;\theta) = softmax(f_{net}(x;\theta))$. Where y, x is the class and input from the dataset, and f_{net} is the full network function and θ are its parameters. NLL works towards increasing the probability of the correct class y for every x. To work in a minimization environment the log of the probability is taken, then converted to a negative, thus when minimizing NLL, it is equivalent to maximizing the log probability. The final equation for NLL is as follows:

$$NLL = -\sum_{j=0}^{D}\sum_{i=0}^{N} 1_{\{y_j=i\}} log(P(y_i|x_j;\theta)) \tag{3.2}$$

Due to the different types of cost functions, in the rest of the chapters f_{cost} will be used to refer to a generic cost function. In this work, the MSE cost function is used in the denoising autoencoders (DAE) and NLL in the final network fine-tuning. All of these will be discussed in the next few sections.

3.1.2 Backpropagation

Given that different weights produce different functions, for learning to take place, a learning algorithm can randomly generate weights until weights that meet a certain minimum on a cost function is attained. However the effort for finding the right weights will increase substantially as the size of the network grows. An algorithm that can more intelligently adjust the weights is needed for network sizes where an exhaustive search method is impractical. One such method is called Backpropagation. Backpropagation is one of the most popular learning algorithms for training neural networks. Backpropagation utilizes gradient descent, which is the process of adjusting the parameters of the network in the direction of achieving a minimum for the cost

function. This is achieved by taking the partial derivative (gradients) of the cost function with respect to the parameters that need to be adjusted in the network. As such, for back-propagation to work, the activation function used needs to be differentiable (23), the activation function referred in Eq. (2.6) cannot be used with back-propagation. Other activation functions that are differentiable, such as the ones found in Eq. (2.1) and Eq. (2.3) will work. Back-propagation can be broken down into these steps:

(1) Given a network and input data, propagate forward to produce an output.

(2) Compute a cost given the output from step 1 and actual expected output from dataset. Eq. (3.1) can be used in this step to produce an error. We will define f_{cost} and for the time being, can be equated as such: $f_{cost} = f_{error}$ It is possible to enhance a cost function to do more than just reducing the error. This will be further discussed in another section.

(3) Take the derivative of the cost function with respect to all the parameters using the following equation.

$$\Delta\theta = \frac{\partial f_{cost}}{\partial \theta} \tag{3.3}$$

Where $\Delta\theta$ stands for the change in θ, and θ stands for all the parameters in the network that need to be changed and f_{cost} is simply f_{error}

(4) Update all the parameters using the following equation:

$$\theta = \theta - \gamma\Delta\theta \tag{3.4}$$

Where the added γ is a configurable parameter that controls the rate of change in the parameters. This variable is called the learning rate.

Taking the gradients of the cost function can become fairly complex and will need the application of the chain-rule. In fact the name backpropagation comes from reusing gradients of nested functions in the lower order of the network. However in programmatic implementation, due to the availability of automatic differentiation, the process is fairly simple. One such library that offers automatic differentiation with chain rule is Theano (24). With backpropagation a complete Learning Algorithm can be seen in Algorithm 1. The function $propagateforward$ in the algorithm refers to computing the full output of a neural network given an input x.

Algorithm 1: Backpropagation

Data: $TrainingData = [(x, y)..., (x, y)]$ where x is an input vector and y is its corresponding 1-hot category vector.
Data: $ValidationData = [(x, y)..., (x, y)]$
begin
- $Predictions \longleftarrow Array()$
- $Actual \longleftarrow Array()$
- $W \longleftarrow Random(min = -1, high = 1)$
- $b \longleftarrow Random(min = -1, high = 1)$
- $epoch \longleftarrow 0$
- $accuracy \longleftarrow 0$
- **while** $epoch{+}{+} < 1000$ **and** $accuracy < acceptable$ **do**
  - **for** $sample \in TrainingData$ **do**
    - $x \longleftarrow sample[0]$
    - $y \longleftarrow sample[1]$
    - $\hat{y} \longleftarrow Propagateforward(x)$
    - $cost \longleftarrow f_{cost}(\hat{y}, y)$
    - $\Delta W, \Delta b \longleftarrow gradients(cost, w.r.t = (W, b))$
    - $W \longleftarrow W + \gamma \Delta W$
    - $b \longleftarrow b + \gamma \Delta b$
  - **for** $sample \in ValidationData$ **do**
    - $x \longleftarrow sample[0]$
    - $y \longleftarrow sample[1]$
    - $AppendToPredictions(Propagateforward(x))$
    - $AppendToActual(y)$
  - $accuracy \longleftarrow ComputeAccuracy(Actual, Predictions)$

3.1.3 Batch & Mini Batch Learning

Up until now, a method called stochastic gradient descent (also called online learning) has been discussed for the purpose to training neural networks. In stochastic gradient descent weights are updated after one sample iteration through the network, this can be seen in the the backpropagation algorithm discussed in the previous section.

An alternative to stochastic gradient descent is to first compute the errors of the network on all available samples before making an update. This method is called batch learning, where batch refers to the whole training dataset. In batch learning the updates gathered point in the right direction, however doing so means to tally up updates for each sample before updating the parameters, as such selecting the right value for the learning rate is very important. One way to look at this is to imagine that the cost function outlines a terrain and the parameters are the coordinates. Taking the gradient of the cost function tells the learning algorithm of the direction towards parameters that will produce a lower cost, but it never provides magnitude. Using a small learning rate will cause the algorithm to converge to an acceptable minimum very slowly (this is due to the need to compute gradients for every sample before an update). Using a large learning rate will cause the algorithm to potentially hop around from hill to hill in the terrain.

Stochastic gradient descent on the other hand, deals with one sample at a time, the process is much quicker to get direction then move in that direction, however because only one sample is used at a time, the path taken towards a minimum (valley) will form a zig-zag (Because each sample will pull towards different parameters that will produce an optimal for that sample only). Which essentially translates to more parameter updates before we reach an acceptable minimum. Stochastic gradient

descent is also prone to the very same large and low learning rate problems of batch learning, however it suffers less so, this is because unlike batch learning the gradients are not accumulated before an update.

Both methods are well understood, but often researchers may lean towards one more than the other, this is mainly due to which one they believe is faster in regards to convergence. An extensive survey and study by Wilson and Martinez in (25) shows that stochastic gradient descent is capable of learning just as well and is faster than batch learning. Of course there would have to be a lower bound on the number of samples in a dataset. A dataset with just ten samples will be much faster with batch learning, and still provide gradients in the true direction.

In this study, a hybrid of the two will be used by taking advantage of a method called mini-batch learning. Mini-batch learning divides the original batch (dataset) by the required number of samples in a smaller batch referred to as mini-batch. In this work, a mini-batch of size ten will be used. The gradients are computed and accumulated for ten samples before parameter updates are done. This algorithm will still produce a zig-zag path, but one that will have fewer direction changes than a fully stochastic algorithm. Due to the nature of various libraries and optimizations for matrix-matrix computations available on modern computers, using mini-batch will produce far faster forwardpropagation results. An algorithm that deals specifically with mini-batch gradient descent is outlined in Algorithm 2.

Algorithm 2: Mini batch Algorithm

Data: $TrainingData = [(x, y)..., (x, y)]$

begin

  $W \longleftarrow$ initiatilize W

  $b \longleftarrow$ initialze b

  while *stopping criterion not met* **do**

    $\Delta W, \Delta b \longleftarrow 0$

    $minibatch \longleftarrow$ Get next ten samples from TrainingData

    for $sample \in minibatch$ **do**

      $x \longleftarrow sample[0]$

      $y \longleftarrow sample[1]$

      $\hat{y} \longleftarrow Propagateforward(x)$

      $cost \longleftarrow f_{cost}(\hat{y}, y)$

      $\Delta W, \Delta b \longleftarrow (\Delta W, \Delta b) + gradients(cost, w.r.t = (W, b))$

    $W \longleftarrow W + \gamma \Delta W$

    $b \longleftarrow b + \gamma \Delta b$

3.1.4 Initializing Parameters

To understand how the parameters need to be initialized, activation functions need to be discussed in a different light. Ultimately throughout the learning process, weights are updated such that neurons are forced to produce a representation that will make it easier for the last layer to produce the right answers. As discussed in Chapter 2, the last step before a neuron value is set is the application of an activation function over the scaled and summed inputs. As such, when computing gradients for weights, the derivative of the activation function will have an impact over the size of the update step in backpropagation. Considering that, if the initial values of the neurons are very high, and the region of the activation function is flat, then likewise the update steps will be fairly small. To achieve higher update steps, the neuron output will need to be in a region that produces high derivatives. Fig. 3.1 shows the gradients of a logistic and ReLU function. ReLU consistently produce the same derivative for all values equal to or above zero, where as a logistic function will produce its highest value at zero, and smoothly degrade as the input moves away

from zero in both directions.

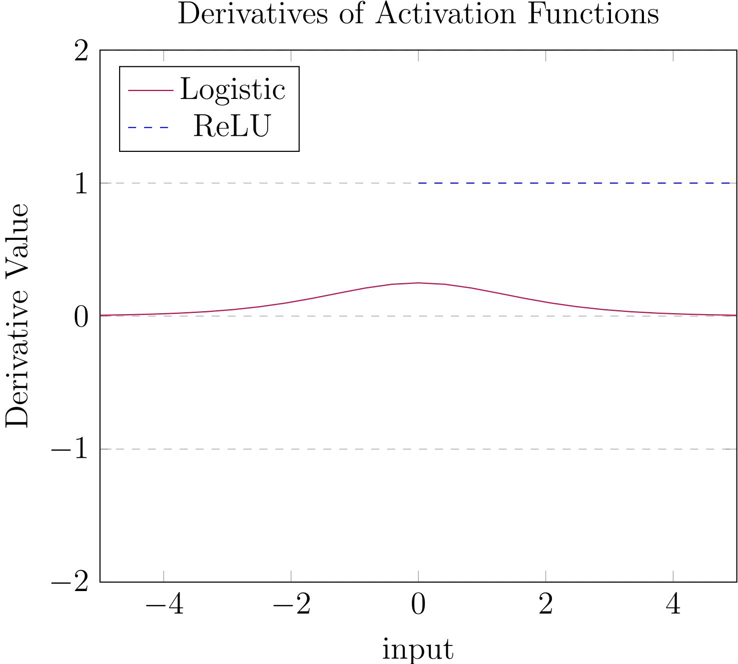


Figure 3.1: Derivatives of a logistic and ReLU activation functions

Therefore, given a logistic based neural network, to saturate the neurons towards an optimal, weights needs to be initialized such that the input to the activation function is close to zero. A simple method to achieve that, would be to generate numbers for weights associated with each neuron that have a mean of zero, and are small, for the sake of this example assume $(-0.1, 0.1)$. This will force all incoming inputs to first be scaled down to really small signals, and the summation will force to cancel out signals of the same strength that are positive and negative. Thus ensuring that most neurons will produce values that are around zero, and ensuring saturation through the use of higher gradients. The specific method that was used in this work comes from the work of Xavier Glorot and Yoshua Bengio in (26), where they received good results for using the initialization technique that depends on the number of columns and rows in the weight matrix (corresponding to number of visible and hidden neurons). A variation of that technique is outlined in Eq. (3.5). This

equation is very popular and is also used in the Theano tutorial neural networks (27). The biases are set to a value of zero. Setting the biases to anything but a zero would cause the input to the activation function to move away from zero.

$$W \sim U\left(-\sqrt{\frac{6}{v+h}}, \sqrt{\frac{6}{v+h}}\right) \tag{3.5}$$

Where U refers to a random uniform number generator which sets the first input as a minimum possible value and the second input as a maximum. v and h refer to the number of visible units and hidden units.

3.1.5 Overfitting, Generalization & Regularization

Like other learning algorithms, neural networks can produce functions that are over complex for the data, what this produces is a great accuracy on the data that it was trained on, but will have a low accuracy on unseen data. What the model ends up learning in this case is the noise that is found in the training data, but that noise maybe absent in unseen data, As such it is able to successfully learn the training data and often performs very well but fails when the model is asked to perform on data that wasn't used to train it. This is called overfitting.
One way to overcome overfitting, is to track the performance of the network on a validation set, once the performance starts to degrade, learning can be stopped (28). The process of preventing overfitting is closely related to two other concepts, generalization and regularization.

Generalization happens when the model performs just as well on previously unseen data as it did on training data (12). When a model overfits, it often ends up learning the noise and bias in the data. These may not be present in other data that

may come from the same distribution as the training data. This causes the model to perform much worse on previously unseen data. The goal of a learning algorithm is to achieve high generalization.

Regularization on the other hand deals with model complexity; it is related to overfitting, in that a model that overfits is often very complex in order to conform to the contours of the training data. A popular way to apply regularization to neural networks, is to add another term to the cost function. This is besides the error term responsible for increasing classification accuracy. An example of regularization is when weights are constrained to be small, this is often achieved by simply adding the value of the weights to the cost function. Other more complex methods also exist. A popular term can be seen in Eq. (3.6), the γ term in the cost function dictates the extent of regularization. A higher γ will force the learning algorithm to favour smaller weights over learning to classify properly.

$$f_{cost} = f_{error} + \gamma |W|^2 \tag{3.6}$$

Because of the close relationship between these three concepts, affecting one of these will have effects on the others. Neural networks that are large, and especially deep networks (networks with more than one hidden layer) are prone to overfitting, as such regularization methods become increasingly important as the size of the network grows. A major method that helps in that regard is Dropout and more recently DropConnect (29; 30). Both methods incorporate the dropping of values at random. In Dropout hidden neuron values are dropped and in DropConnect individual weight connections are dropped. More specifically, in forwardpropagation, neurons or weights are selected at random, and their values are set to zero. In a sense, this cancels a percent of the information flow in the network, thereby simplifying the model. The

official explanation behind the success of Dropout is attributed to the averaging of thinned networks (29). However it can also very easily be seen that these thinned networks are sparse large networks. Consider the case where 50% of the inputs in a hidden layer are dropped, then what is really happening is that 50% of the values are set to zero, irrespective of the original values. This causes the hidden layer to have at-least 50% of its values set to zero, which will cause the layer to be far more sparse than what it started with. Sparsity will be explained in the next section.

3.1.6 Measuring Sparsity

Sparsity formally is the number of zeros in a vector. Consider the two vectors $x = [1, 1, 0, 1, 0, 1, 1]$ and $y = [0, 0, 1, 0, 1, 0, 0]$, if sparsity was measured on both, y would have a higher sparsity score, since it has more zeros. A simple method of measuring sparsity would be to simply count the number of zeros. In a neural network where the goal is to measure the sparsity in the hidden layer, a sequential code that would need to check every index in the hidden layer and count the number of zeros would be required. That however maybe impractical and slow for large neural networks, hence a more simple and mathematical equation is needed. There are various such equations as can be seen in (31), in Machine learning l^1 and l^2 referred to as L1 and L2 going forward, are common. Both can be computed by the same general equation, this equation can be seen in Eq. (3.7). L2 is often applied on the weights of a neural network as a regularization method. Another equation that can easily be added to the cost function would be log-penalty as seen in Eq. (3.8). When applying L1 and log-penalty to vectors x and y, we get the values 5.0 and 2.0 for L1 and 1.41 and 0.70 for log-penalty. Normally the negatives of these equations are taken to signify higher sparsity for higher numbers (31), for our purpose since we are minimizing a cost function, high numbers for lower sparsity is preferred, therefore

the negatives of these equations will not be taken. This will lead gradient descent to increase sparsity while minimizing cost.

$$l^p = \left(\sum_j c_j^p \right)^{1/p} \quad \text{for } p > 0 \tag{3.7}$$

where vector $\mathbf{c} = \{c_1, c_2, \cdots, c_n\}$. L2 is often applied to the weights of a neural network as a regularization method.

$$f_s = \log \left(1 + \sum_{i=1}^{n} (y_i^2) \right) \tag{3.8}$$

where y_i is the output from the ith neuron in the hidden layer.

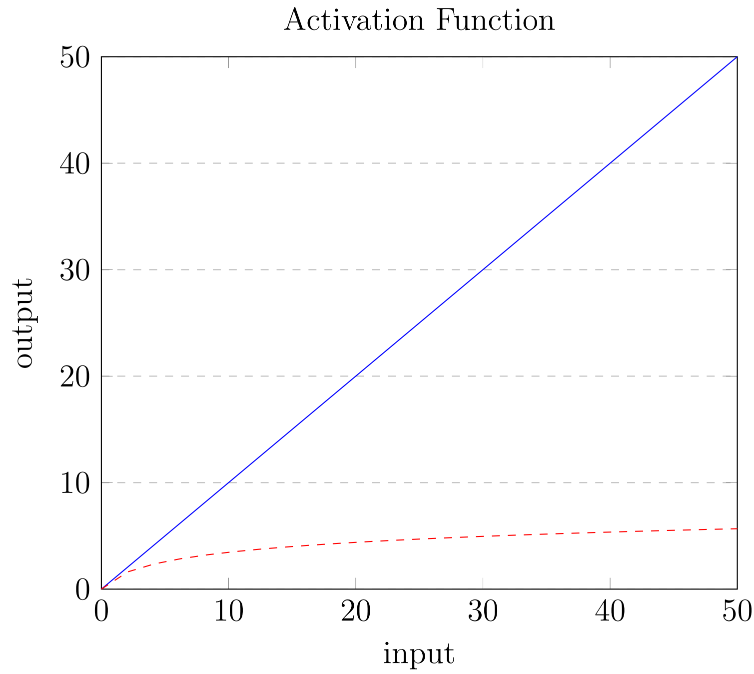


Figure 3.2: Comparison of the L1 norm in blue versus the log-penalty in red proposed for this work.

3.1.7 Unsupervised Feature Learning & Autoencoders

In Chapter 2, to solve a non-linear problem that was inseparable, a hidden layer was added (Section 2.1.4). In a sense, the hidden layer transformed the input to become linearly separable. The idea of unsupervised pre-training is to build hidden layers that have been automatically trained to pick up features using unsupervised learning algorithms. An easy way to force neural networks to learn abstract features of the input, is to rethink of the expectation. So far, classification neural networks that focused on mapping input x to output or category y have been discussed. In those cases the objective of the neural network is to simply classify correctly, irrespective of how its done, this can potentially make the network prone to overfitting due the presence of the possible bias and noise in the dataset. If the objective of the network is to learn representation, then the objective will need to be better aligned to achieve that. A simple method, that may direct us in the right direction, would be to simply ask the network to reconstruct its input from a hidden representation.

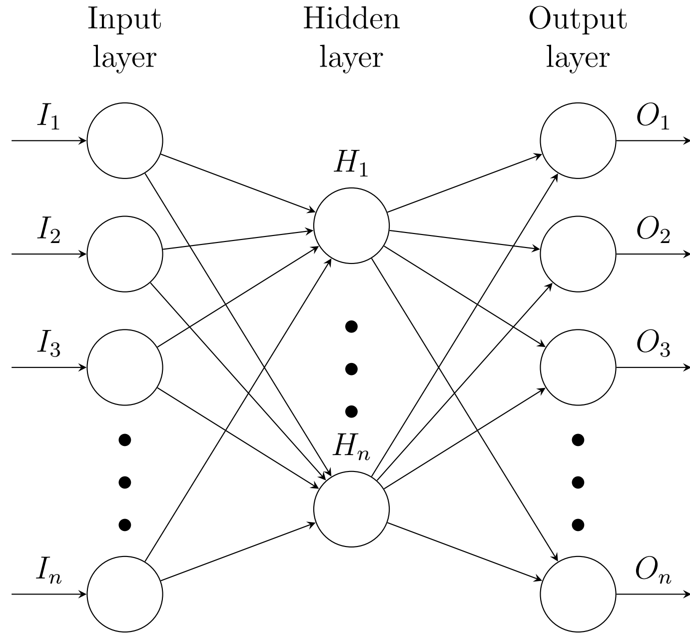


Figure 3.3: AutoEncoder transforms the input into a hidden representation, then reconstructs the input back from the hidden representation.

To do this, the network first need to propagate its input forward to a hidden state, this can be achieved by using Eq. (3.9). The output of this propagation, is what we will refer to as hidden representations, this can be followed by propagating back using Eq. (3.10). This output produced will be an array that will be in the same dimensionality as the input. These two functions, lay the foundation for the ability of a neural network to learn to reconstruct the original image from the hidden representation. If the network can be forced not to simply copy the inputs to the hidden layers, then we can guarantee a different representation in the hidden layer by simply minimizing a reconstruction criterion where x is the input, and $\hat{x}$ is the reconstruction from the hidden representation. MSE can be used as the cost function for lowering the difference between the original and the reconstruction.

$$h(\mathbf{x}) = \phi(\mathbf{x}\mathbf{W} + \mathbf{b}) \tag{3.9}$$

$$\hat{x} = \phi(h(\mathbf{x})\mathbf{U} + \mathbf{c}) \tag{3.10}$$

This particular neural network topology is referred to as an autoencoder. Autoencoders have successfully been used to learn hidden representations of its inputs, these were also then successfully placed in a regular classification neural network (32). The process of first learning hidden representations using unsupervised methods such as an AutoEncoder is called unsupervised pre-training. Which is followed by classification using a regular feedforward network, this network retains the hidden representations from pre-training, this stage is referred to as the fine-tuning stage. When multiple layers of a neural network are pretrained in this manner, it is referred to as layer-wise pre-training. Pseudo-code for functions that will mimic activations of an autoencoder are as follows:

```
import numpy
#numpy imported to perform a dot product
def Encode(x, W, b) :
    return activation(numpy.dot(x,W) + b)
def Decode(x, U, c) :
    return activation(numpy.dot(x,U) + c)
```

3.1.8 Denoising AutoEncoder

Denoising AutoEncoders unlike regular AutoEncoders learn abstract features much like those found in RBM models (33). This is achieved by corrupting the input before it is fed into the model, thus the model is forced to reconstruct the original from a corrupted input. Fig. 3.4 showcases how the corrupted dotted inputs are expected to be rebuilt as the original. In denoising autoencoders the input or x is corrupted

before being passed to Eq. (3.9), this can be achieved by simply multiplying x by a randomly generated binary vector of the same size as x, as shown in Eq. (3.11). The level of corruption can be controlled by adjusting the probability that each value at index i in the vector m is set to 1. For instance a corruption level of 30% will produce a 1 for index i in the vector m with probability 0.7. If this is applied to an image; which is first converted from matrix form to row-major vector form; 30% of the pixels in that image will be corrupted, receiving a 0 in place of the original value. Abstracting away the mathematical details, this procedure will produce a network that resembles Fig. 3.4.

$$\bar{x} = x \times m \tag{3.11}$$

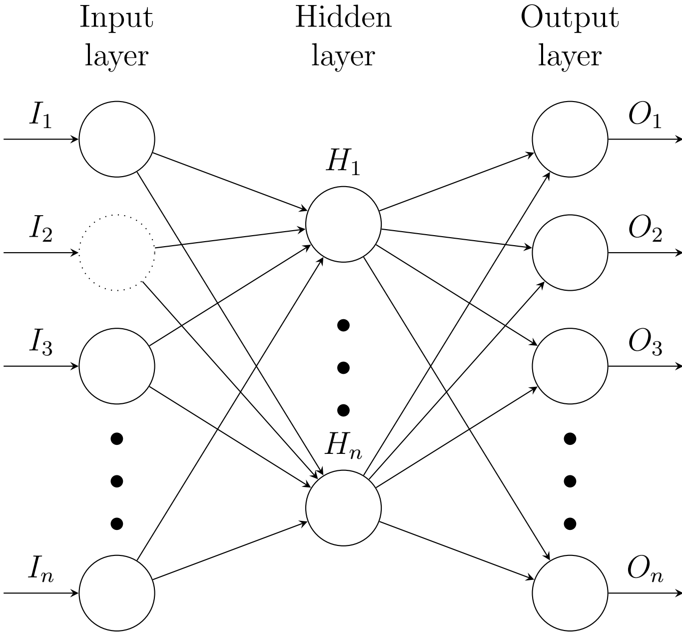


Figure 3.4: Denoising AutoEncoder gets a corrupted input and is expected to reconstruct the original. (Dotted neurons automatically are set to a value of 1).

Layer-wise pre-training

Recent successes in the deep learning field have been attributed to early unsupervised layerwise pre-raining methods which helped deep architectures learn layerwise representations (34; 35). This was largely attributed to the ability of these unsupervised networks to learn abstract features, one of these unsupervised methods, a variation of autoencoders, called denoising autoencoders have been found to learn interesting features of the input (33). Given Eq. (3.9), it can easily be seen that the encoding process is highly dependant on the weights $\mathbf{W}$ and the biases $\mathbf{b}$. As such, once an autoencoder has been successfully trained to reconstruct the input, it can be assumed that the encoding parameters of the network $(\mathbf{W}, \mathbf{b})$ can successfully be used as building blocks for the first layer of a neural network, and this process will bring along the information encoded in the parameters $(\mathbf{W}, \mathbf{b})$. Interestingly enough, the hidden representation from this network, can be used as input to another autoencoder, this autoencoder will learn features of features. This process can be repeated as many times as the number of desired hidden layers for a feedforward neural network. The process of throwing away decoding parameters $(\mathbf{U}, \mathbf{c})$, and instead using hidden representation as input to subsequent layers are all a part of layerwise pre-training. This process has been widely successful, and has not been limited to autoencoders (36; 32; 33). Once the desired number of hidden layers are trained layer-wise, the final network can be topped off with a softmax layer and fine-tuned using regular backpropagation. The main difference here, is that the final network parameters are initialized using autoencoders.

3.2 The Model

Sparsity has been seen as an advantageous characteristic in Machine Learning, this can be seen through various other works that forced networks to learn sparse representations. A popular method where this can be found is in Sparse Coding models (37), where the model is forced to prefer sparse representation by adding a sparsity term to the loss function, these methods can also be applied to AutoEncoders. Aside from adding a direct sparsity term to the cost function, using certain activation functions have also been found to help with training neural networks. ReLU, or rectified linear units that take the max between zero and the input x, have been found to decrease training time and sometimes even increase classification accuracy (38).

Other major methods that are widely used today can also be seen to increase sparsity, namely Dropout and DropConnect (29; 30). Both of these methods have overtaken the layer-wise pretrainig methodologies, the main reason is the ability for the network to perform just as well, without the need to waste time pre-training. However as explained in previous sections in this chapter, unsupervised feature learning can be integral to creating representations that are robust to noise, as such models that are far superior in their ability to classify can be created. As such, the main work done in this research focuses on enhancing unsupervised pre-training methodology in denoising autoencoders. The exact areas are as follows:

(1) New Activation Function

(2) Addition of log-penalty sparsity term to the cost function

(3) Swapping of activation function back to logistic function once the DAE is trained.

The final complete model will be a neural network with sigmoidal activation neurons throughout. Training the model will follow the layerwise pre-training paradigm, which falls into two parts. The first is the unsupervised stage also known as unsupervised pre-training. The second stage is the supervised training using the weights learned in the unsupervised stage. Conventionally many types of learning algorithms have been used for the unsupervised pretrainig stage, in this model a modified Denoising AutoEncoder will be used.

The focus of this work will also be on the unsupervised pre-training stage. The modifications applied to DAE can also be divided into two areas, first of which is in regards to the activation function used in the hidden layer. The second modification, is in regards to the cost function. To reduce the total training time and still maintain the advantages provided by unsupervised pre-training, a model that can saturate the neurons quickly will be preferred. One way to achieve this is through using a high learning rate. However a more natural method is to use an activation function that has higher gradients around zero. The reasoning behind this is to simply increase the gradient size, hence the change in weights with each update. In essence enabling the fast saturation of neurons.

The adjustment to the cost function is in regards to restricting the information flow between the layers. In other words, restricting the amount of potential copying of information. Consider images as input, since the objective of an AutoEncoder is to reconstruct the original image from the hidden layer representation, it will be fairly easy for the model to simply copy the input directly into the hidden layer (provided there are sufficient neurons in the hidden layer), however this would give us no better representation to apply a classifier unto. As such, a neural network that can force the model to focus on the more salient, structure related information will inevitably give us a much better representation. One such way to reduce the flow of information is to

force a large percentage of the neurons to be close to zero, or in other words force the model to prefer sparse representations in the hidden layer. Doing this mathematically is to use a sparsity term along with the reconstruction term in the cost function. The main work is in regards to the swapping of functions before fine-tuning begins. This particular methodology we refer to as Swapped pre-training Activation Functions (SPAF), networks trained in this method are referred to as SPAF-networks.

The rest of this chapter will outline the details of modifications in the Unsupervised layer, an explanation of the transfer of learned knowledge from the pre-training phase to the supervised fine-tuning phase and finally the details of the fine-tuning phase.

3.2.1 New Activation Function

To assure that all of the neurons are well saturated in the pre-training phase, an activation function with high gradients near zero is preferred. The function used in this work can be seen in Eq. (3.12). A comparison between Sigmoid and this new activation can be seen in Fig. 3.5, as is evident from that graph, the proposed function is much steeper around zero, and also reaches its threshold at much lower values of its input. A comparison of the new activation function and ReLU can be seen in Fig. 3.6, the ReLU does offer a similar gradient to the new activation function, but it has no gradients in the negative spectrum for its input.

As will be seen in Chapter 4, ReLU causes the network to learn mostly excitatory features, where as the new activation function allows for inhibitory neurons just like a logistic function. The exact gradient differences between a logistic function and the newly proposed function can be seen in Fig. 3.7. The higher gradients enable the

network to saturate quickly, the swapping of activation function before fine-tuning, which will be explained in SPAF-network section, enables us to go back to regular gradients using a logistic function for smooth tuning of the weights for classification.

$$Activation(x) = \frac{x}{\sqrt{1+x^2}} + 0.5 \tag{3.12}$$

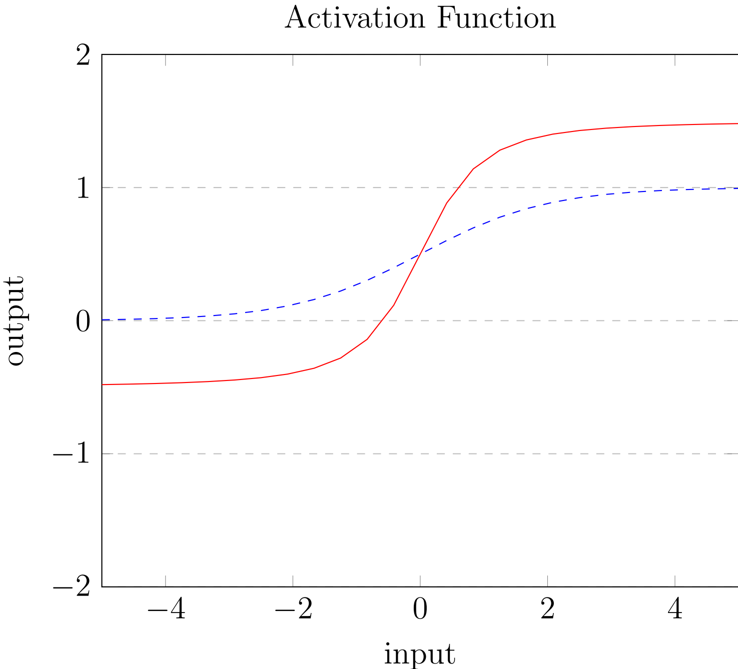


Figure 3.5: Comparison of the modified Elliot function in red versus the sigmoidal activation function in blue

3.2.2 Sparse favouring Loss Function

As outlined in the previous section, sparsity in the hidden layer representations will be preferred. This will divide the loss function into two distinct parts. First, the model needs to reduce the reconstruction error. Second, the model needs to prefer sparse representations in the hidden layer over non-sparse representations. The first

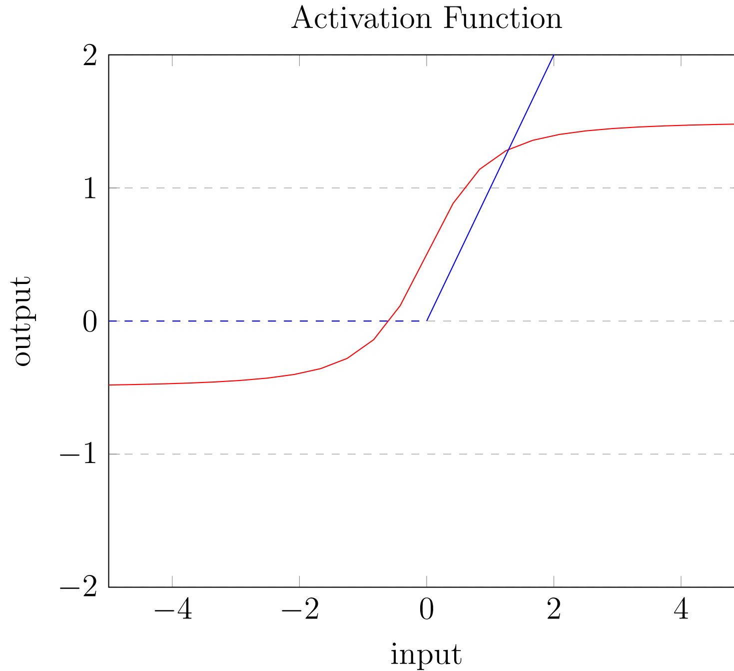


Figure 3.6: Comparison of the proposed activation function in red versus the ReLU activation function in blue

part is easily achieved by using a mean-squared error function. Minimizing MSE will lead to minimizing the reconstruction error. The second part can be achieved by using one of the sparsity functions mentioned in the previous section, Measuring Sparsity.

Previously L1 and L2 regularization techniques have utilized 1^1 and l^2 norms to force the weights of neural networks to remain sparse. In this work, the focus is on the representation of inputs, hence sparsity will be forced on the hidden layer neurons. This model will use the log-pentaly formula to force regularization, the main reason for this selection is that it is easily computable, and the values also do not explode for possibly very large inputs as is the case with L1. If the values become very large, the loss function will be competing against itself in regards to favouring sparsity versus reconstruction error. This is also the reason why L1 is often used with a small weight penalty, to force the model to prefer reconstruction over sparsity. The

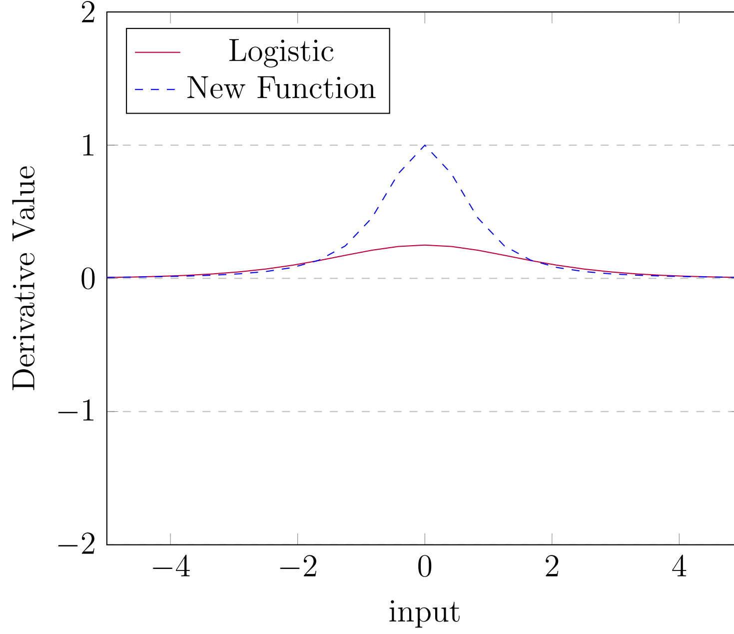


Figure 3.7: Derivatives of a logistic and the newly proposed activation function

final loss function with the two pieces can be seen in Eq. (3.13).

$$f_{cost} = \sum_{i=0}^{N}(f(x)_i - y_i)^2 + \log(\sum_{i=0}^{M}(1 + |h_i|)^2) \tag{3.13}$$

Where the **h** holds the hidden representation in an autoencoder, N is the dimensionality of y and M is the dimensionality of hidden layer **h**.

3.3 Conclusion

Conventionally the weights of the supervised network in the fine-tuning stage are constructed using all of the hidden layer representations generated in the unsupervised pre-training stage. The parameters that are borrowed are the weights, the biases and the activations of each hidden layer. In the SPAF-model, only the weights and

biases are kept from the unsupervised pre-training stage. The activations used in the pre-trainig stage are all replaced with logistic activation functions. The network will still resemble the topology created in the pre-training stage, minus the decoding weights on the top-most layer. For supervised training to be able to work, a softmax layer with as many units as the number of classes is added to the topmost hidden layer. This is the final step for the supervised fine-tuning stage, before conventional backpropagation is applied to finetune the weights for a classification task.

The reasoning behind why this specific technique was chosen is as follows: Since most of the feature learning occurs in the pre-training stage, the process is accelerated by introducing saturating neurons. Once the neurons are saturated, the logistic is used to only finetune the weights in a smooth manner for classification. The goal is to try and find the closest minimas that reduce a classification error given the pretrained parameters. Since the pretrained parameters have learned abstract and hopefully useful features, then doing so will be advantageous towards achieving a better classification result.

Finally, the newly proposed model offers high gradients with no algorithmic or topological differences in comparison to the conventional unsupervised pre-training technique. The higher gradients come in the form of a different activation function, which in regards to its complexity, is still constant. As such one output from both models with the same number of neurons in all of its layers will be: $\mathcal{O}(Mn)$, where M is the matrix multiplication part, and n is the number of matrix multiplications.

Chapter 4

Implementation & Experiments

4.1 Implementation

The overall problem can be divided into two distinct parts, the unsupervised pre-training using autuencoders, and the final fine-tuning using a regular feedforward neural network. The second part borrows parameters from the first. An intuitive way of programming this problem, is to simply create two modules, the first taking care of unsupervised training and the final stage copying weights and fine-tuning. However, it is far more efficient to create one network, that has the ability to pretrain its individual hidden layers, before finally fine-tuning. Fig. 4.1 outlines the final solution. The program was coded in Python to take advantage of Theano, a library that enables automatic differentiation (24). This simplifies the backpropagation algorithm in regards to computing gradients with respect to the the parameters in the network. More importantly modules for an autoencoder and feed-forward neural networks are available on the Theano website as tutorials (27), these were taken as starting points and extensively modified to create the final program. Another important part to

mention is that the denoising autoencoder designed, used the transpose of the weight matrix W for decoding, this produces better features and also simplifies the design of the autoencoder.

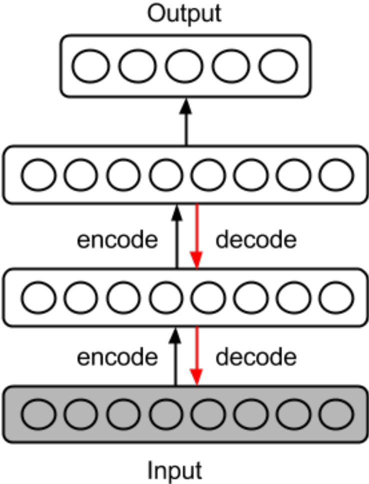


Figure 4.1: A neural network that follows pre-training paradigm trains each hidden layer with the input of the preceeding layer. Encoding weights are kept, and the decoding weights are removed once pre-training is complete.

The main computation falls under two files, the first of which deals with a feedforward network with a softmax output layer (fine-tuning file), the second deals with unsupervised pre-training using autoencoders. The final algorithms for pre-training and fine-tuning can be seen in Algorithm 3 and 4 respectively.

Algorithm 3: pre-training Algorithm

Data: $TrainingData = [(x, y)..., (x, y)]$ where x is an input vector and y is its corresponding 1-hot category vector.

begin

$\gamma \longleftarrow pre - traininglearningrate$

for $hiddenlayer \in HiddenLayers$ **do**

$W \longleftarrow hiddenlayer.W$

$b \longleftarrow hiddenlayer.b$

$c \longleftarrow hiddenlayer.c$

Initialize W

$b \longleftarrow 0$

$c \longleftarrow 0$

for $pre - trainingepochs = 0; pre - trainingepochs < 10; pre - trainingepochs{+}{+}$ **do**

for $minibatch \in TrainingData$ **do**

$\nabla W, \nabla b, \nabla c, \longleftarrow 0$

for $sample \in minibatch$ **do**

$x \longleftarrow sample[0]$

$y \longleftarrow sample[1]$

$h \longleftarrow Encode(W, b, x)$

$\hat{x} \longleftarrow Decode(W^T, c, h)$

$cost \longleftarrow f_{error}(\hat{x}, x) + logPenalty(h)$

$\nabla W, \nabla b, \nabla c, \longleftarrow (\nabla W, \nabla b, \nabla c) + gradients(cost, w.r.t = (W, b, c))$

$W \longleftarrow W + \gamma \nabla W$

$b \longleftarrow b + \gamma \nabla b$

$c \longleftarrow c + \gamma \nabla b$

Algorithm 4: Fine-tuning Algorithm

Data: $TrainingData = [(x, y)..., (x, y)]$
Data: $ValidationData = [(x, y)..., (x, y)]$
begin
- $\gamma \longleftarrow learningrate$
- $patience \longleftarrow 10 * trainingbatches$
- $patienceincrease \longleftarrow 2$
- $improvementthreshold \longleftarrow 0.995$
- $donetraining \longleftarrow false$
- $epoch \longleftarrow 0$
- $iter \longleftarrow 0$
- **while** $epoch{+}{+} < 300$ *and* $(!donetraining)$ **do**
  - **for** $minibatch \in TrainingData$ **do**
    - $iter + +$
    - $\nabla\theta \longleftarrow 0$
    - **for** $sample \in minibatch$ **do**
      - $x \longleftarrow sample[0]$
      - $y \longleftarrow sample[1]$
      - $\hat{y} \longleftarrow f_{net}(x; \theta)$
      - $cost \longleftarrow f_{cost}(\hat{y}, y)$
      - $\nabla\theta \longleftarrow \nabla\theta + gradients(cost, w.r.t = \theta)$
    - $\theta \longleftarrow \theta + \gamma\nabla\theta$
- $validationscore = TestModel(f_{net}, ValidationData)$
- **if** $validationscore > bestvalidationscore$ **then**
  - $bestvalidationscore \longleftarrow validationscore$
  - **if** $validationscore < bestvalidationscore * improvementthreshold$ **then**
    - $patience \longleftarrow max(patience, iter * patienceincrease)$
- **if** $patience < iter$ **then**
  - $donetraining \longleftarrow True$

4.2 Datasets & Experiments

To test the algorithms, four datasets will be used, namely MNIST and Img, Hnd and the Fnt datasets from the bigger Chars74k dataset (39). A total of six models divided into two groups will be trained and compared for three of the four datasets. The first group we will refer to as the SPAF-network, and the second as Sigmoidal group. The Sigmoidal group offers sigmoidal activation functions with no function swapping during fine-tuning. The SPAF-network employs the newly proposed activation function, and will also incorprate function swapping to sigmoidal activations during the fine-tuning stage. Both of these groups will offer models with these variations:

(1) No sparsity term in the cost function.

(2) L1 sparsity term.

(3) Newly propsed log-penalty sparsity term.

For each of the six models, depending on the dataset 30 networks were trained, tested, averaged and 95% confidence intervals were calculated and graphed. For the last dataset Fnt, two models will be compared across two types of model and data variations. First being, varying the number of neurons in each layer and the second being the percent of dataset used for training. The first experiment will be used to see the effect of increasing the number of available neurons on the final performance for each model. The second experiment to measure the affect of increasing data samples over learning. It is also worth mentioning that all of the statistics gathered in the experiment were done so programmatically using the scipy and numpy libraries (40). Every network was allowed to run for a minimum of ten pre-training epochs

and a maximum of 300 finetuning epochs with early stopping if the network stops to improve after training for a minimum of $10 * trainingbatches$ iterations over the data, where *trainingbatches* is measured by how many mini-batches the full dataset gets divided into. Along with a minimum wait, there is also a patience variable, that increases as we get better results, this is to better follow the performance curve as it approaches its best performance threshold. The Fnt and Img dataset was allowed to run 40 pre-training epochs and 100 for Hnd dataset.

4.2.1 MNIST

The MNIST dataset is a fairly popular dataset for neural networks for various reasons. The main reason is that the algorithm is preprocessed and formatted, and the data is also real world handwritten digits (41). They were originally collected from a larger database of handwritten digits, but hand crafted to be a much more reasonable dataset. The digits were collected from Census Bureau employees and high school students. The training set and testing set are designed such that, the writers of the training set do not have any samples in the testing set. Thus the two datasets are written by completely different people (41). Each individual sample is a $28 * 28$ size vector, representing a square image in row-major format. There are a total of 60,000 samples in the training dataset, and 10,000 samples in the testing set. A two hidden layered network with 200 neurons each will be used to test this dataset, we will refer to this network as a 200-200 network. The network will have $28 * 28$ input neurons (corresponding to the number of pixels in each image), and a final softmax classification layer with ten neurons (corresponding to the total number of digits in the dataset, zero to nine). A subset of 50,000 from the training dataset will be used to train the network, another 10,000 will be kept for validation purposes. The final test set will be used to measure the final performance of the network. A learning

rate of 0.1 was used for both the supervised and unsupervised training stages. The L1-models had a γ value of 0.01, while the log-penalty models had a value of 1. This is because L1-models would produce sparsity terms that were too large for proper learning to be effective, hence the small γ value. A total of 30 networks were trained to get proper statistics.

Figure 4.2: Sample images from the MNIST dataset, as can be seen, there is a wide variation in writing styles

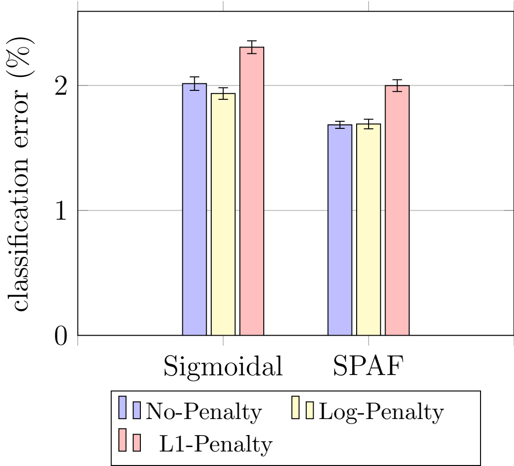


Figure 4.3: Comparison of Sigmoidal activation function versus the SPAF-network, along with the affects of L1 and the newly proposed log-penalty term.

The error percent of the various models and their respective confidence intervals can be seen in Fig. 4.3. The L1-model was worse off in both the standard Sigmoidal-

model and the SPAF-network. In regards to the success of the new model, it produced better average results when compared with its corresponding conventional Sigmoidal-models. The log-penalty on the other hand produced about the same performance as the regular models. When it comes to sparsity, the results were drastically different. The L1-models produced the most sparse representations in the first hidden layer where they achieved $0.152 \pm 0.003, 0.286 \pm 0.008$ for the sigmoidal and the SPAF-network respectively, this can be seen in Fig. 4.4. The newly proposed log-penalty based models reduced sparsity in all layers and models except for the second layer in the Sigmoidal-model, as can be seen in Fig. 4.5. We can conclude that the L1 induces sparsity at the expense of accuracy, while the log-penalty was ineffective at both inducing sparsity and offering better accuracy performance.

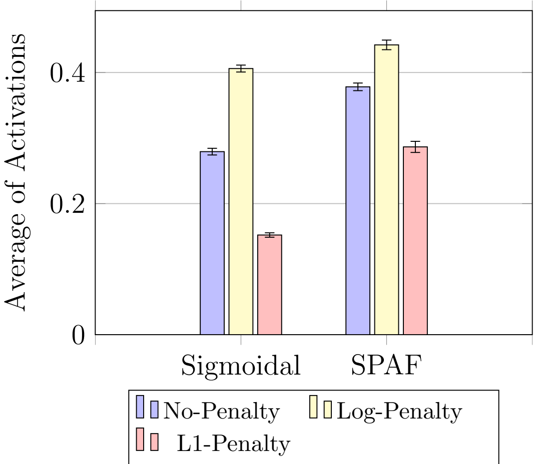


Figure 4.4: Comparison of the average activation in the first layer of Sigmoidal-model versus the SPAF-network, along with the affects of L1 and the newly proposed log-penalty term.

4.2.2 Chars74k

Similar to MNIST the Chars74k dataset has images of the arabic numerals for classification. It however also has the english alphabet for classification tasks, this

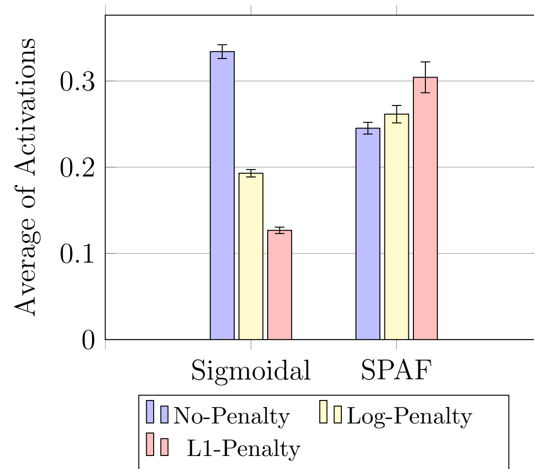


Figure 4.5: Comparison of the average activation in the second layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly propsed log-penalty term.

makes this dataset much more difficult than the MNIST dataset, by bringing the total possible classes to 62 versus the ten total categories in MNIST. Chars74k comes in three folders, the first of which is called Fnt and has images of numbers and the alphabet generated from computer fonts. The second folder called Bmp, has images that were generated from natural images of various signs which were taken using a variety of cameras. The last folder is called Hnd and has images of handwritten letters generated through a tablet device (39). The dimensionality of each image varies in Chars74k, as such they were scaled and feature scaled to $[0, 1]$ to reduce dimensionality and also make them better suited for neural network classification. The exact process was divided into two steps, the first step involved processing each image in the following manner:

(1) Reading image into program.

(2) Convert image to greyscale

(3) Color invert each image

(4) Save image to disk

All this was done through a python library called Pillow (42).

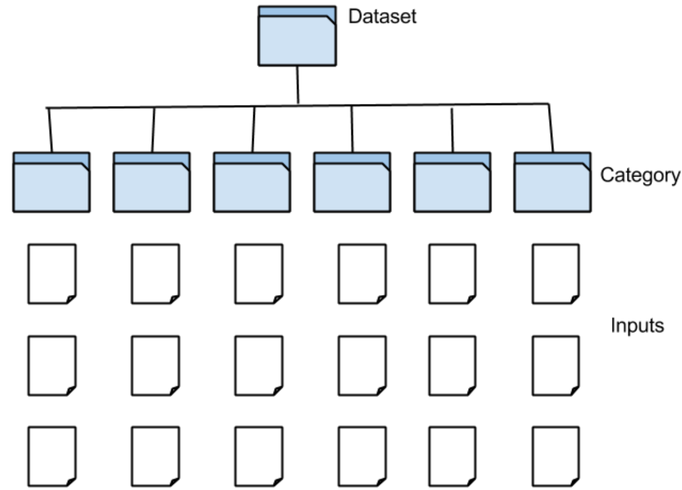


Figure 4.6: The individual dataset in Chars74k were formatted in a tree folder structure, with the top level folder indicating one of the 3 folders, and each subfolder indicating a category.

The second step involved getting the images ready in standard (X, Y) dataset format that can be fed to a neural network. The original dataset was stored on disk in a tree folder structure with images of the same cateogry or class saved in the same folder as seen in Fig. 4.6. These folders were traversed and processed in the following manner:

(1) In each folder open every image and let folder name be expected output.

(2) Select each image and resize to $28 * 28$ pixels.

(3) Store pixel values in array x and folder name to varibale y.

(4) Normalize values in x using Eq. (4.1)

Once this process was completed for all possible images, all values of x were appended to X and all y to Y. Scikit-learn (43) was then used to generate unique numbers for each category name, these numbers were then converted to one-hot vectors at run time. Each subfolder was converted to its own dataset using the aforementioned method. Models were trained and built on each of these subfolders and their associated results are mentioned in their own respective sections. Unlike MNIST, chars74k does not have a designated test set, as such the repeated random sub-sampling cross-validation was used.

$$x_{scaled} = \frac{x - X_{min}}{X_{max} - X_{min}} \tag{4.1}$$

Bmp Dataset

The available samples per class varies widely in this dataset, starting at 33 samples per class all the way to 554 samples. For this dataset, a learning rate of 0.2 for both pre-training and fine-tuning stages was used, this was selected based on test network training starting with a learning rate of 0.8, 0.2 showed promising descend in cost. The gamma values for the sparse models were kept the same as the MNIST models. This dataset was randomly split 75/25 for training and test sets a total of 10 times and their average and standard deviations were calculated. The results on training the models on the BMP dataset can be seen in Fig. 4.7. The associated average activation values in each hidden layer can be seen in Fig. Similar to the results in MNIST, the L1-model offered higher sparsity at the expense of accuracy performance, and the log-penalty again decreased sparsity in both models except the second layer in the standard Sigmoidal-model as can be seen in Fig. 4.8 and Fig. 4.9.

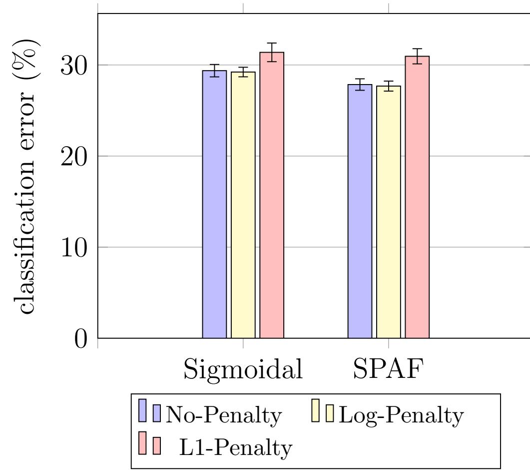


Figure 4.7: Comparison of Sigmoidal-model versus SPAF function, along with the affects of L1 and the newly propsed log-penalty term on the Img Dataset.

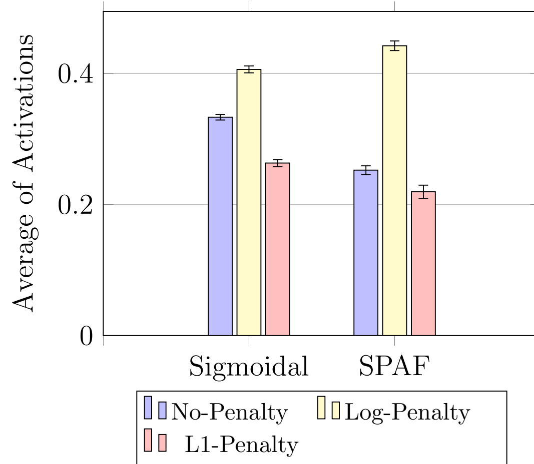


Figure 4.8: Comparison of the average activation in the first layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly propsed log-penalty term on the Img Dataset

Hnd Dataset

This dataset only has a total of 3,410 samples, it was collected through tablet pc from a total of 55 individuals. Similar to the original work in (39), approximately 15 samples from each class were randomly placed in a testing set, the remaining were placed in a training set. A total of 30 networks again were trained in this manner

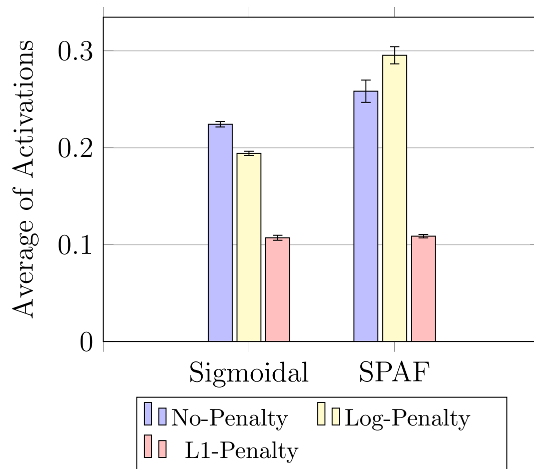


Figure 4.9: Comparison of the average activation in the second layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly proposed log-penalty term on the Img dataset.

to produce the results shown in Fig. 4.10. The results for Hnd dataset were not as pronounced as the previous two datsets discussed, both in performance and also sparsity induction for both L1 and the log-penalty. This may be related to the lack of samples for this particular dataset despite that, the SPAF-network performed better on average than the standard Sigmoidal-model. What is worth mentioning is that the log-penalty model continued to increase sparsity in the second hidden layer of the Sigmoidal-model. The sparsity results in the first and second layers can be seen in Fig. 4.11 and Fig. 4.12 respectively.

Fnt Dataset

This dataset was by far the biggest out of all. A total of $62,992$ samples were collected from 254 different fonts in four styles (39). Due to the size of this dataset and the higher classification complexity (due to the increase in available classes over MNIST), two sets of experiments were conducted to test two models. First being

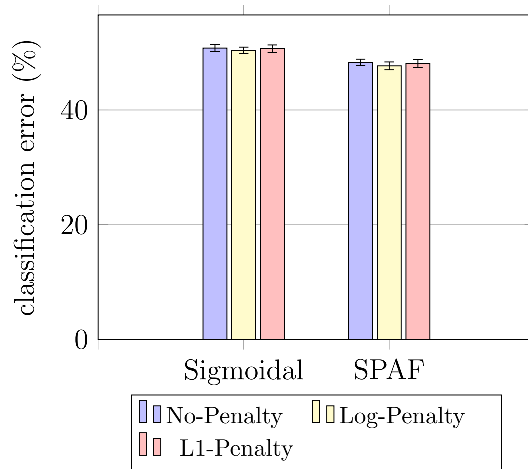


Figure 4.10: Comparison of Sigmoidal activation function versus the newly proposed swapped activation function, along with the affects of L1 and the newly propsed log-penalty term on the Hnd Dataset.

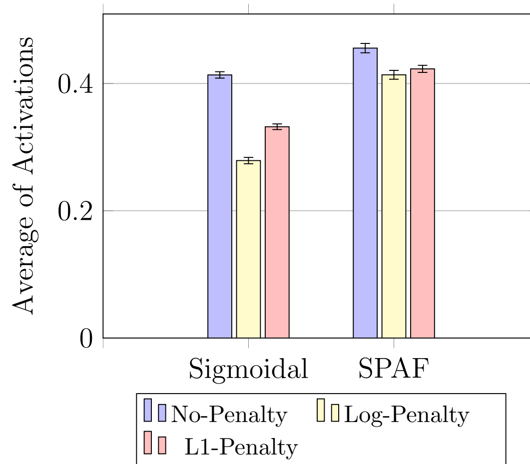


Figure 4.11: Comparison of the average activation in the first layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly propsed log-penalty term on the Hnd Dataset

the conventional sigmodial model, where the normal logistic activation functions are used in both pre-training and fine-tuning stages. The second model is the proposed SPAF-network model. These models in the first experiment are trained for their classification performance over the number of neurons used in each hidden layer. These results can be seen in Fig. 4.13. In the second experiment, the models were tested for their performance given a percent of the overall dataset. All the models in this experiment utilized a network with two hidden layers of 200 neurons each.

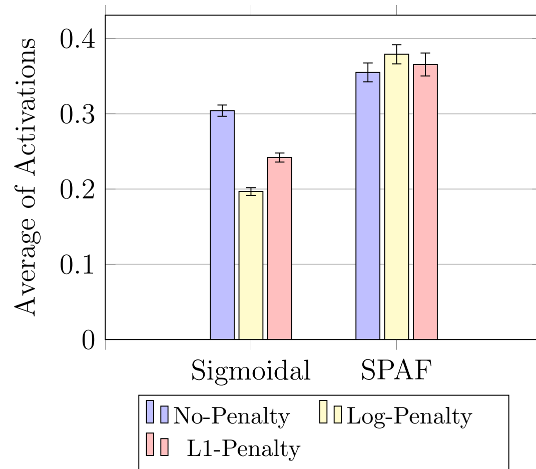


Figure 4.12: Comparison of the average activation in the second layer of Sigmoidal-model versus SPAF, along with the affects of L1 and the newly propsed log-penalty term on the Hnd dataset.

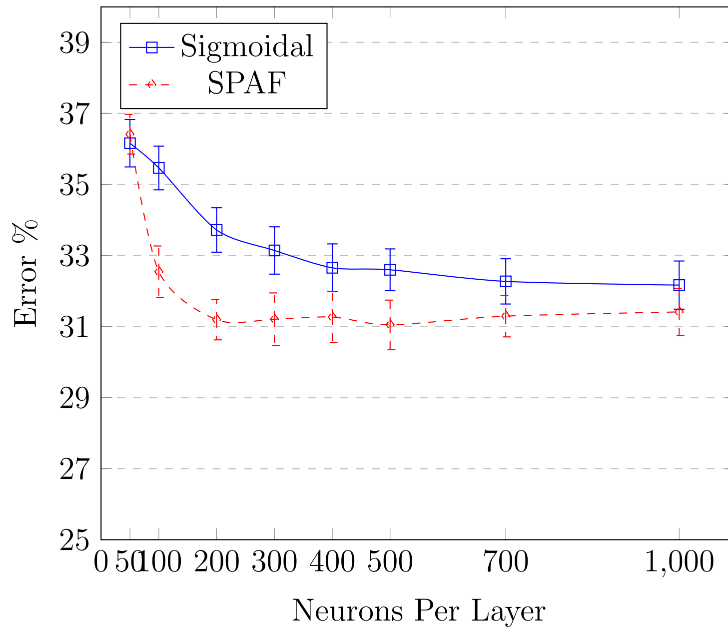


Figure 4.13: Comparing the regular Sigmoidal-model over the SPAF-network for their performance over different number of neurons in the hidden layers

The results of the second experiment can be seen in Fig. 4.14. Both models utilized the random sampling of approximately 15 samples from the dataset for the testing set. The first experiment used the same number of samples in the training set as the testing set (15).

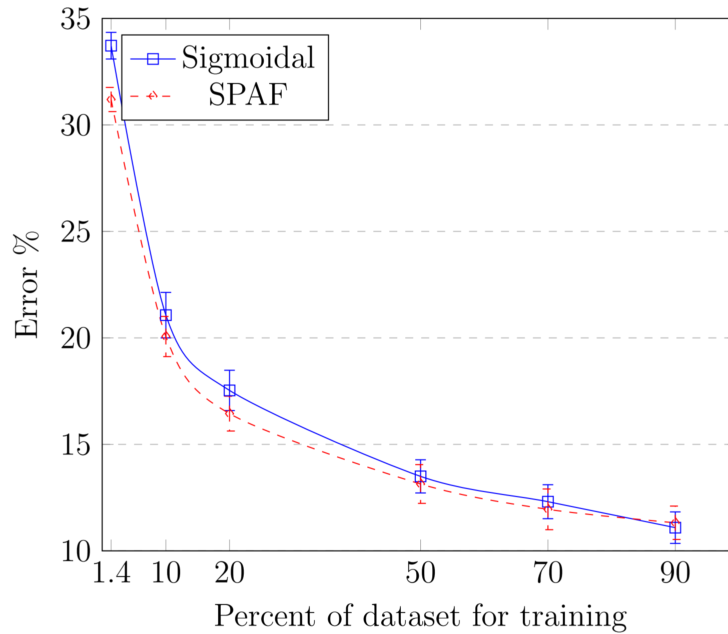


Figure 4.14: Comparing the regular Sigmoidal-model over the SPAF-network for their performance given a percent of the total dataset for training.

The first experiment that studies the affect of the number of neurons in the hidden layers over performance showed promising results. The experiment results can be seen in Fig. 4.13. The proposed model starts at about the same result as the Sigmoidal-model at 50 neurons, but quickly outperforms it until about 700 neurons, where both models start to plateau to about the same performance average. The second experiment seen in 4.14 shows that there is no real difference between the two models when we increase the number of training samples. This can be related to the fact that only 15 samples per class were used for testing, as the two models approach the maximum possible performance, the difference in their learning ability diminishes.

4.2.3 Learned Features

In this section, a method for how to reconstruct images from weights will be discussed, and using this method images of features learned across two SPAF-networks and the regular logistic DAE pretrained based network. These three models will be trained on three of the four datasets from the previous experiments, namely MNIST, Fnt and Hnd. For the sake of simplicity, the same network topology of 200-200 and learning rate of 0.1 will be used. In regards to pre-training epochs, ten epochs were used for MNIST and 100 for Fnt and MNIST. The MNIST splits were kept the same, so $50,000$ samples were used for training, where as for Fnt and Hnd 50% of their respective datasets were used.

4.2.4 Images from Weights

Recall that in Chapter 2, the XOR function was constructed from the OR and AND networks. The weights of the XOR network going from the hidden layer to the classification layer were described in the manner of 'presence' of and 'absence' of features; where features were the OR and AND functions. In regards to images, weights can be described in the following fashion. Weights that are positive, coming from a pixel can be thought of, as weights that 'like' the pixel. Given that a neuron will have connections to every single pixel in the image, the weights can be thought of as the 'like' and 'dislike' variables of a neuron. Given a single layer, the process of converting the 'like' and 'dislike' variable intensities to pixel intensities for every neuron requires the conversion of every column in the weight matrix $\mathbf{W}$ to a row-major vector that follows the same technique as the original image. Given this vector, a matrix of pixel intensities can be created by first feature scaling the vector to values between $(0, 1)$ using Eq. (4.1). This matrix can then be converted to an image using

a generic library that can take pixel intensities in a matrix form. One such library that was used in this study is Pillow (42). In this manner, values that are high, will mean particular pixel areas that the neuron likes in an image, and pixels that are dark are ones that the neuron dislikes.

Another way to understand this, is to consider the range of possible values that each scaled vector can take. The lower values will get closer to zero, and the higher values will get closer to 1. Absolute 0 meaning complete inhibition of that particular pixel, and 1 meaning excitation of that pixel. A collection of these can indicate the excitation of the neuron for the presence of a feature that is more complex than a pixel. Fig. 4.15 outlines this process, where the like and dislike of pixel values is referred to as pixel strength. This method of generating images from weights is widely used in the industry (33; 34; 35; 44).

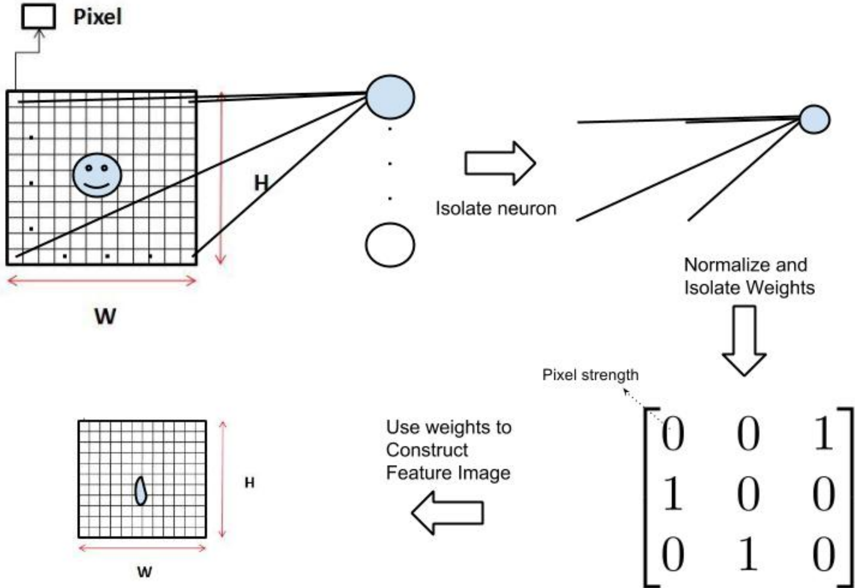


Figure 4.15: Converting Weights of neurons in the first layer to feature images.

The process can repeated for as many layers as needed. The calculation become

more complicated, however the intuition remains the same. In this experiment, the first layer features were generated by code provided in the Theano tutorials (27). An algorithm was developed for generating matrices for layers above layer-1, and can be seen in the pseudocode below.

```
#where vh is the visible-hidden matrix
#can also be a recursive call to drawWeights
#thereby representing a preceding hidden-hidden matrix
#and hh is the next level hidden-hidden matrix.
def drawWeights(vh,hh,layer) :
    new = vh.copy()
    new.fill(0)
    for master,k in enumerate(hh.T) :
        #tally up all visible unit connections for this neuron.
        for idx,value in enumerate(k) :
            new[master,:] = new[master,:] + vh[idx,:] * value
    image = PIL.Image.fromarray(tile_raster_images(
        X=numpy.asarray(new),
        img_shape=(28, 28), tile_shape=(20, 10),tile_spacing=(1, 1)))
    image.save(name+'layer-'+str(layer)+'.png')
    return new
```

In essence what that algorithm does, is traverse the connections from the top, all the way down to the original pixels. This is done in such a fashion, that a final matrix that has dimensions of (v, h) is attained, where v and h refers to the number of neurons in the visible and hidden layer in question. This final matrix can then easily be used to save an image with patterns which translate back to the inhibition and excitation of the various neurons. To better understand the calculations being made in a neural network, consider that the weights of every neuron in the first layer will generate as many images as there are neurons in that layer. These images represent the features that this first layer has learned. As such the layers above will be learning

features of the features in the first layer. Therefore when wanting to get features for every neuron in subsequent layers, the associated weights of the neurons in the respective layer will be used as scaling units for the images of individual neurons of the preceding layer, these images once scaled can be summed, to produce a single image per neuron in layers above the first.

Feature Learning Results

When trained on MNIST and the given network configuration, the features learned in the first hidden layer of the newly proposed based SPAF network produces more inhibitory and excitatory neurons than those learned by the regular Sigmoidal-network. The ReLU based SPAF network surprisngly only offers excitatory and neutral neurons, this is explained by the fact that the ReLU activation only offers gradients on the positive spectrum of the inputs. Inhibitory neurons are defined as neurons, that have predominantly lower values (indicated by the black in the images) and excitatory neurons are defined by the predominantly large values (indicated by the white in the images).

The second layer was far more interesting, as almost all the neurons produced predominantly inhibitory neurons. The sigmoidal produced the most excitatory neurons, however as can be seen in Fig. 4.16, the patterns still mimic the features learned in layer-1, as they are mostly basic detectors for a cluster of neighbourly pixels. The ReLU based SPAF-model fails at producing any discernible features for half of its neurons, and the rest of the neurons that do learn features are like those learned in the regular sigmoidal network; predominantly simple. The first block in question in Fig. 4.17, is the representations of the newly proposed function based SPAF-model. As can be seen almost all of the neurons with the exception of two, sixth from top

right, and second from bottom left) produce complex features that mimic line and curve detection.

The Features learned for the Fnt dataset as seen in Fig. 4.18, follow similar patterns to those from the MNIST dataset. The ReLU based SPAF network again produces predominantly excitatory neurons with a minority being neutral. The newly proposed activation based SPAF network again produced more excitatory and inhibitory neurons than the regular Sigmoidal-model. The second layer as seen in Fig. 4.19, showed interesting results, as some of the neurons can be seen to have copied individual images. This is not necessarily great for classification, as these particular neurons have over-fit to one particular sample. The other two models continued to produce good features in its second layer.

The Hnd based model learned very few interesting features in all of the models. The fewest can be seen in the regular Sigmoidal-model. The trend of ReLU producing excitatory neurons in layer-1 continued here as well. Overall in the second layer, while the newly proposed activation function based SPAF and regular sigmoidal-network produced interesting features, there was however not much complexity when compared to previous datasets in the type of features learned. This can be attributed to the limited number of samples used for each individual class, infact by using 50% of the total $3,410$ samples available in the dataset, each individual class had roughly 27 samples in the training dataset. The respective features from layer-1 and layer-2 can be seen in Fig. 4.20 and Fig. 4.21.

Lastly, what is important to take away from these results is that each network learned unique features, especially in layer-2 of the networks. To see the features of all five trained networks, see Appendix B.

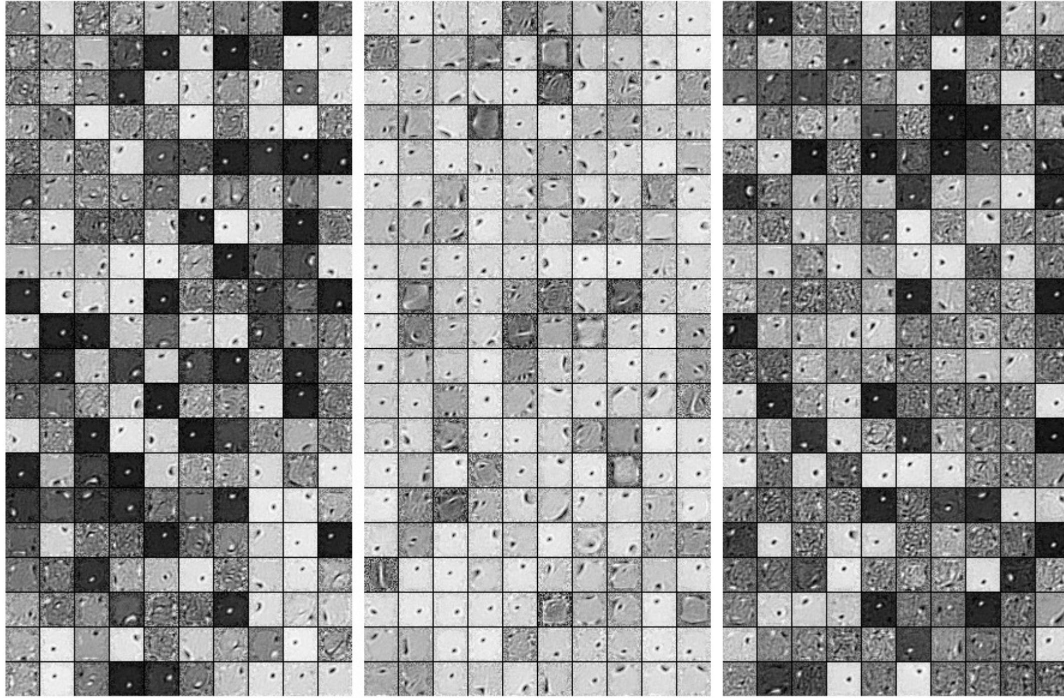


Figure 4.16: Features of the first layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, in that respective order. Trained on the MNIST dataset.

4.3 Conclusion

The SPAF-model showed promising results throughout, with 16.4%, 12.6% and 13.3% improvement over the Sigmoidal-model in the no-penalty, L1 penalty and log-penalty models respectively. Improvements were also seen in the BMP dataset, however they were less pronounced with 5.2%, 5.3% and 1.4% for the no-penalty, L1 penalty and log-penalty comparisons in favour of the SPAF-model. The Hnd dataset was more consistent with 4.9%, 5.4% and 5.1% improvements over the Sigmoidal-model.

In regards to sparsity, the log-penalty due to it's small values was not able to induce very high sparse representations in the first hidden layer. However, it consistently produced better results in the second layer of the Sigmoidal-model over

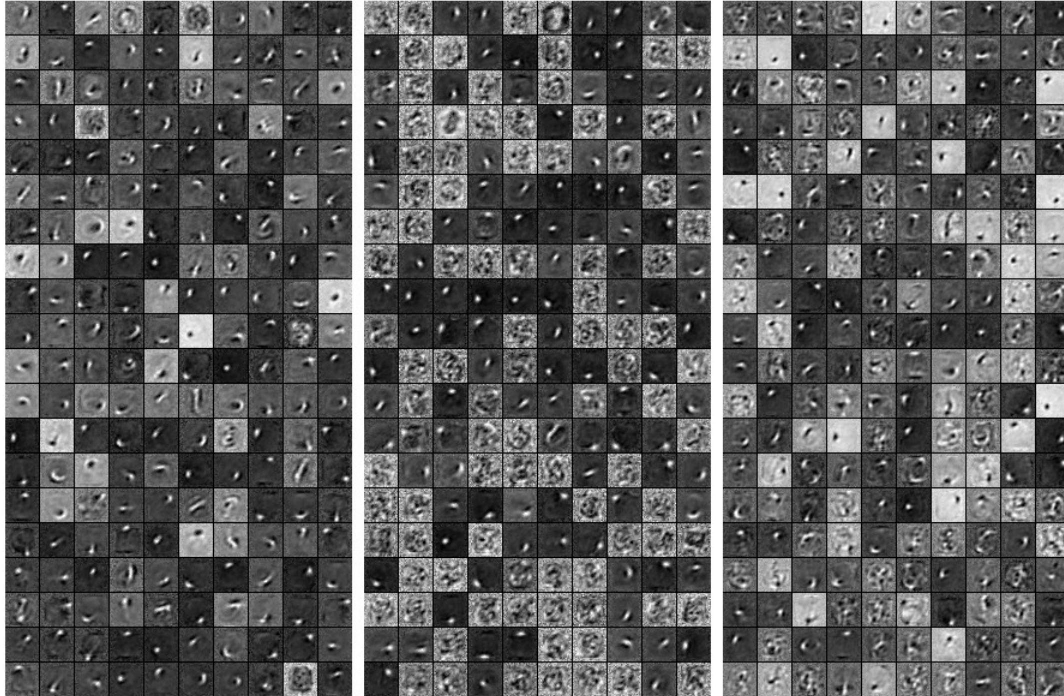


Figure 4.17: Features of the second layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, all in that respective order. Trained on the MNIST dataset.

the SPAF-model, this can be seen in Fig. 4.12, 4.9 and 4.5. This maybe related to the type of features that are learned when applying log-penalty on a Sigmoidal-model versus a SPAF-model. Further investigation with analysis of features of each respective model might shed more light into this phenomenon.

In regards to features learned, the SPAF-model produced a higher number of fuller features; both excitatory and inhibitory. This is especially pronounced in the second layer of the SPAF-model trained on the MNIST dataset seen in Fig. 4.17, where the majority of the features end up with strokes, line and edge detectors that can be thought of as building blocks on top of the simple circles learned in the previous layer.

Lastly, the most important idea is in regards to the variation in the type of

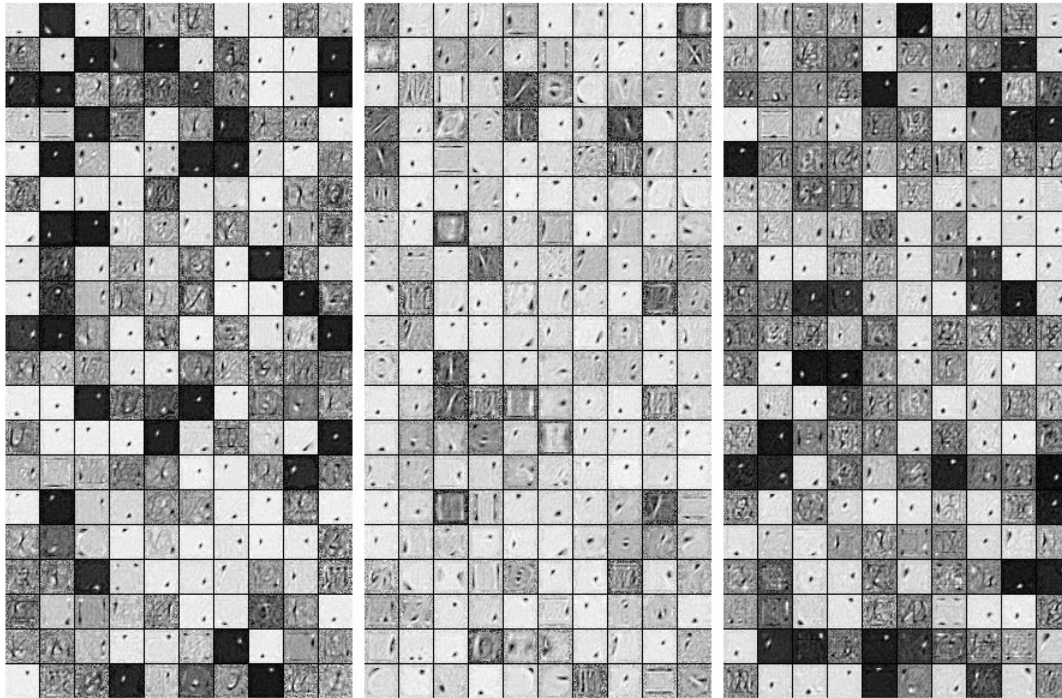


Figure 4.18: Features of the first layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, in that respective order. Trained on the Fnt dataset.

features learned when using different activation functions. Our results have further strengthened the idea that deep neural networks are filled with many local minima, some of which perform better than others, this is discussed in the work of Erhan et al. in (45). The ReLU features learned are specially indicative of this phenomenon, where the weights are clearly in a very different region than all the others. This work can further be extended by varying the activation functions further and measuring the performance.

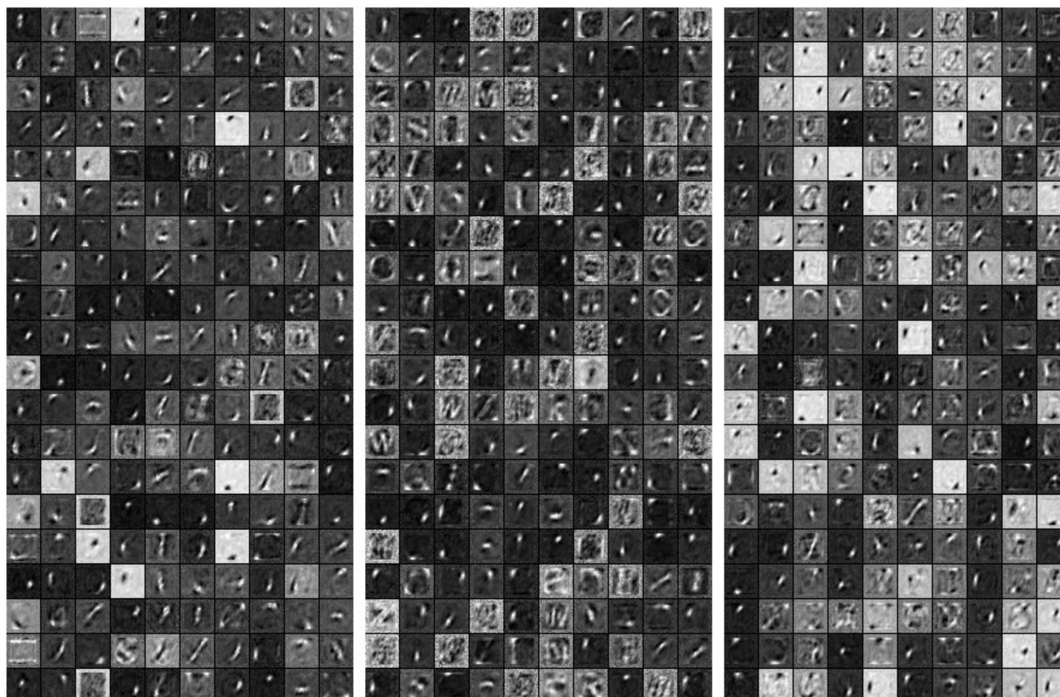


Figure 4.19: Features of the second layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, all in that respective order. Trained on the Fnt dataset.

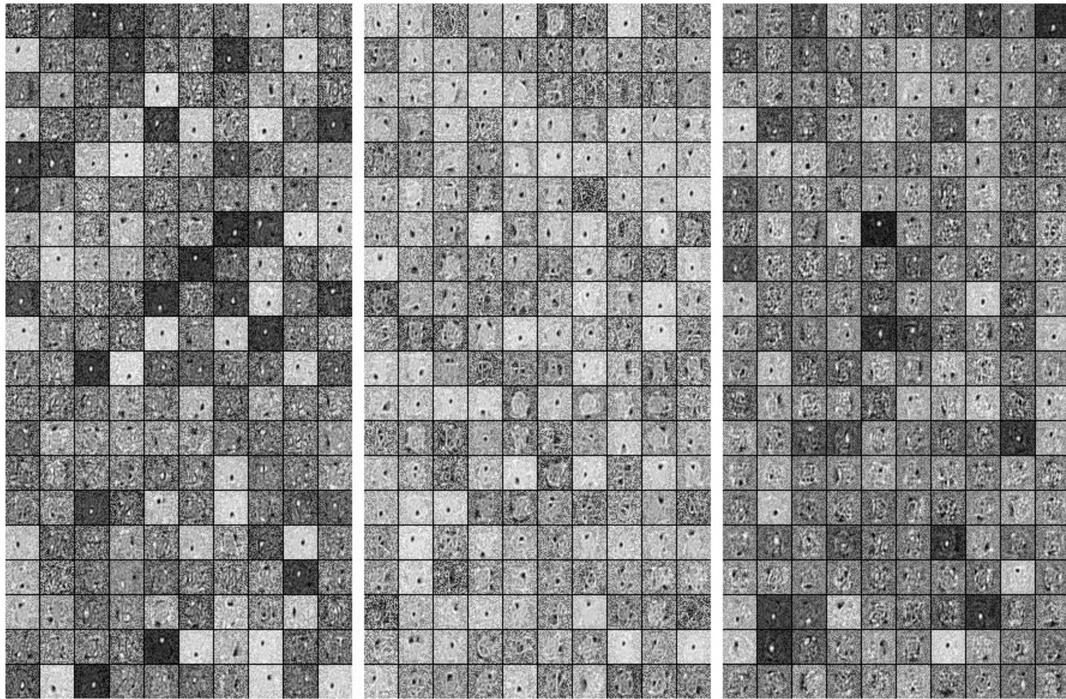


Figure 4.20: Features of the first layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, in that respective order. Trained on the Hnd dataset.

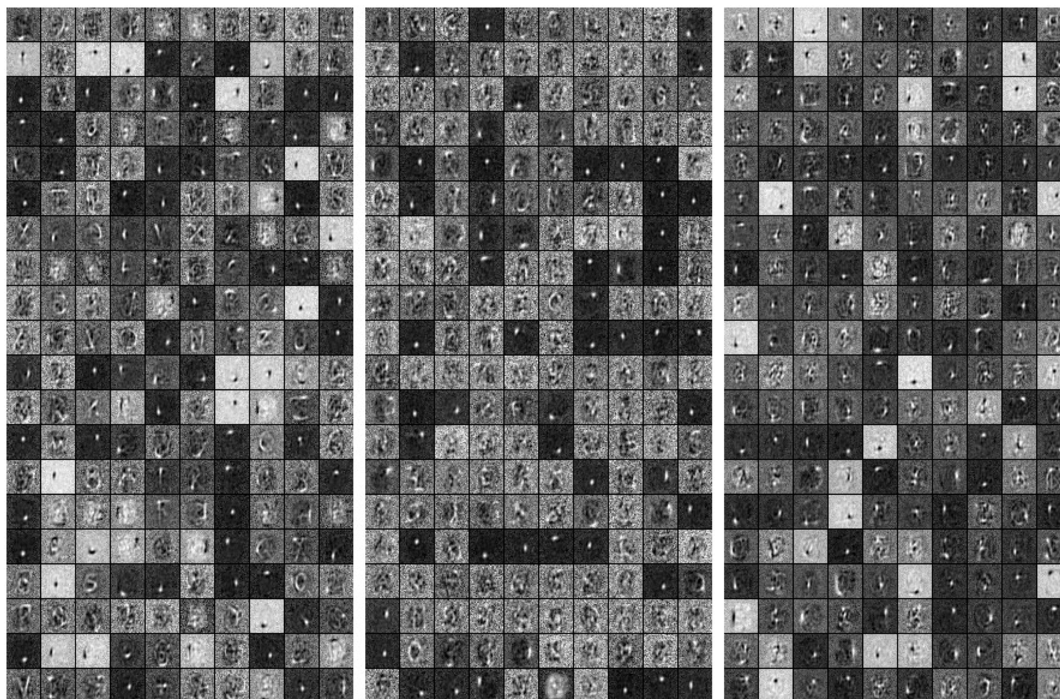


Figure 4.21: Features of the second layer of SPAF-networks with the proposed activation function and ReLU along with a regularly DAE pretrained network with logistic function, in that respective order. Trained on the Hnd dataset.

Chapter 5

Conclusion

5.1 Conclusion

In this work various ideas were explored to enhance the pre-training stage in supervised networks whose weights are initialized using unsupervised autoencoders. More specifically, a new activation function was introduced to the hidden layer of denoising autoencoders. This activation function was selected specifically to offer higher gradients and also offer negative output values much like a tanh function, but with a focus on positive values. The goal was to accelerate neuron saturation in the hidden layers, and results have shown that this new method does translate to better results. More specifically, the new activation function showed promising results, with better average error rates over 30, or ten networks for MNIST and Chars74k datsets.

The second idea explored was the introduction of the log based sparsity penalty on the hidden layer representation in the cost function. This model was compared and contrasted with an L1 penalty on the hidden representations and was found to

show that the log penalty offered no improvements in the first layer of the models, but showed modest improvements in the second layer of Sigmoidal models. The reasoning behind why improvements were only seen in the second layer of Sigmoidal models and not the newly proposed models, can be linked to the kind of features that are learned in layer-1 of the SPAF-model, that make layer-2 much more resilient to sparsity changes. The L1 penalty showed improvements over every layer, except for the second layer in the model with the new activation function trained on MNIST as well as the Hnd Dataset. The log penalty did not affect the classification performance in any significant way, the L1 on the other hand, increased classification error in more than one dataset understandably so, as the reduction in average activation in the hidden layers was significant.

The final and unique method for this work was the introduction of the SPAF-network model, which incorporated the swapping of activation functions from the unsupervised to supervised training stages. This is the first research work that has explored this concept and the results are promising. While the model did perform better on average on several of the datasets, what is more important is that this model is capable of learning. The swapping of activation function can open a new research area into combining models with unique feature detection abilities. These unique features in the SPAF-model can be combined and finetuned using the standard Sigmoidal or recent ReLU activations for fine-tuning.

5.2 Limitations & Future Work

In regards to limitations, a few of them revolve around the various parameters used in the models, namely the learning rate as well as the γ parameter for sparsity in the cost function. It would be interesting to see if further reduction of γ, in the

L1-model would lead to a more comparable performance in error rates yet still offer good sparsity improvements. Likewise a comparison over all the models with multiple reasonable learning rates can offer more insight over the differences in the learning abilities of the various models. Another major area, is the max number of allowed epochs for training in both the unsupervised and supervised stages. Varying the number of epochs and studying the effects of the length of training time over the features learned as well as the final performance will shed more light into the effects of swapping activation functions as well as varying penalty terms.

In regards to potential future work, there are a few avenues. One major avenue is the introduction of other activation functions during activation swapping from pre-training to funetuning stages. For instance, this work tested a new activation function that closely resembled a tanh but with a $y - axis$ translation and a higher gradient. In this form as was seen, both excitatory and inhibitory features are produced in the hidden layer. Other activation functions may produce different proportions of excitatory, inhibitory and neutral neurons. The ReLU activation function for instance, only provides gradients for positive output, hence pushing the weights towards positive weights; this gives us a model that only ever produces excitatory neurons, as was seen in the results. Other activation function, that may focus more on producing negative values will likewise produce predominantly inhibitory neurons. It would be interesting to compare and contrast the performance of both these models. That can further be extended by creating a hybrid model that randomly selects a percent of the neurons from models trained with vastly different activation function to create hybrid layers with drastically different features. Incorporating some of this knowledge in Convolutional neural networks may also produce interesting results. The SPAF-model can further extend the ability of the conventional layers to create a larger variation in its feature maps; thus opening the potential for further improvements in classification performance.

Bibliography

[1] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943. [Online]. Available: http://dx.doi.org/10.1007/BF02478259

[2] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[3] T. Tieleman, "Training restricted boltzmann machines using approximations to the likelihood gradient," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1064–1071.

[4] "Google neural network teaches itself to identify cats," accessed: 2015-04-22. [Online]. Available: http://www.eetimes.com/document.asp?doc$_id = 1266579$

[5] H. Burhani, W. Feng, and G. Hu, "Denoising autoencoder in neural networks with modified elloitt activation function and sparsity-favoring cost function," in *In Proceedings of the 2nd ACIS International Conference on Computational Science and Intelligence*. Okayama, Japan: IEEE Computer Society, Jul. 2015, pp. 79–84.

[6] K. J. D. . K. J. M. Hartman, E., "Layered neural networks with gaussian hidden units as universal approximation," *Neural computation*, vol. 2, pp. 210–215, 1989.

[7] S. H. Cardoso, "Parts of the nerve cell and their functions," accessed: 2015-05-09. [Online]. Available: http://www.cerebromente.org.br/n07/fundamentos/neuron/parts$_i$*.htm*

[8] R. Bland, U. of Stirling. Department of Computing Science, and Mathematics, *Learning XOR: Exploring the Space of a Classic Problem*, ser. Technical report (University of Stirling. Dept. of Computing Science and Mathematics). Department of Computing Science and Mathematics, University of Stirling, 1998. [Online]. Available: https://books.google.ca/books?id=T-YJMwEACAAJ

[9] "'robobarista' can figure out your new coffee machine," accessed: 2015-05-06. [Online]. Available: http://www.news.cornell.edu/stories/2015/04/robobarista-can-figure-out-your-new-coffee-machine

[10] "How google retooled android with help from your brain," accessed: 2-15-05-05. [Online]. Available: http://www.wired.com/2013/02/android-neural-network/

[11] Y. B. S. B. P. V. Dmitru Erhan, Peirre-Antoine Manzagol, "The difficulty of training deep architectures and the effect of unsupervised pre-training," *AISTATS*, pp. 153–160, 2009.

[12] Y. Bengio, I. J. Goodfellow, and A. Courville, "Deep learning," 2015, book in preparation for MIT Press. [Online]. Available: http://www.iro.umontreal.ca/ bengioy/dlbook

[13] "Contextual targeting: Why its awesome," accessed: 2015-05-06. [Online]. Available: http://acuityads.com/contextual-targeting-why-its-awesome/

[14] C. C. Aggarwal and C. X. Zhai, *Mining Text Data*. Springer Publishing Company, Incorporated, 2012.

[15] "How does facebook suggest tags?" accessed: 2015-05-06. [Online]. Available: https://www.facebook.com/help/122175507864081

[16] "Wikipedia statistics english," accessed: 2015-05-09. [Online]. Available: http://stats.wikimedia.org/EN/TablesWikipediaEN.htm

[17] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* New York, NY, USA: Cambridge University Press, 2008.

[18] J. Ramos, "Using tf-idf to determine word relevance in document queries," accessed: 2015-05-09. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.1424 rep=rep1type=pdf

[19] Y. Yang and J. O. Pedersen, "A comparative study on feature selection in text categorization," in *Proceedings of the Fourteenth International Conference on Machine Learning*, ser. ICML '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 412–420. [Online]. Available: http://dl.acm.org/citation.cfm?id=645526.657137

[20] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ser. ICCV '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 1150–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850924.851523

[21] U. Shrawankar and V. M. Thakare, "Techniques for feature extraction in speech recognition system : A comparative study," *CoRR*, vol. abs/1305.1145, 2013. [Online]. Available: http://arxiv.org/abs/1305.1145

[22] G. E. Hinton, "Learning multiple layers of representation," in *Trends in Cognitive Sciences, Vol. 11*, 2007, pp. 428–434.

[23] R. Rojas, *Neural Networks: A Systematic Introduction.* New York, NY, USA: Springer-Verlag New York, Inc., 1996.

[24] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio, “Theano: new features and speed improvements,” Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[25] D. R. Wilson and T. R. Martinez, “The general inefficiency of batch training for gradient descent learning,” *Neural Netw.*, vol. 16, no. 10, pp. 1429–1451, Dec. 2003. [Online]. Available: http://dx.doi.org/10.1016/S0893-6080(03)00138-2

[26] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[27] “Theano tutorial,” accessed: 2-15-05-05. [Online]. Available: http://deeplearning.net/software/theano/tutorial/index.htmltutorial

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[29] ——, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[30] L. Wan, M. D. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, “Regularization of neural networks using dropconnect,” in *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, ser.

JMLR Proceedings, vol. 28. JMLR.org, 2013, pp. 1058–1066. [Online]. Available: http://jmlr.org/proceedings/papers/v28/wan13.html

[31] N. Hurley and S. Rickard, "Comparing Measures of Sparsity," *Information Theory, IEEE Transactions on*, vol. 55, no. 10, pp. 4723–4741, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1109/tit.2009.2027527

[32] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems 19 (NIPS'06)*, B. Schölkopf, J. Platt, and T. Hoffman, Eds. MIT Press, 2007, pp. 153–160. [Online]. Available: http://www.iro.umontreal.ca/ lisa/pointeurs/BengioNips2006All.pdf

[33] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, W. W. Cohen, A. McCallum, and S. T. Roweis, Eds. ACM, 2008, pp. 1096–1103.

[34] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring strategies for training deep neural networks," *J. Mach. Learn. Res.*, vol. 10, pp. 1–40, Jun. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1577069.1577070

[35] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1162/neco.2006.18.7.1527

[36] G. E. Hinton and S. Osindero, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, p. 2006, 2006.

[37] R. Jenatton, J. Mairal, G. Obozinski, and F. Bach, "Proximal methods for hierarchical sparse coding," *J. Mach. Learn. Res.*, vol. 12, pp. 2297–2334, Jul. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2021074

[38] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving Deep Neural Networks for LVCSR using Rectified Linear Units and Dropout," in *Proc. ICASSP*, 2013.

[39] T. E. D. Campos, B. R. Babu, and M. Varma, "Character recognition in natural images." [Online]. Available: http://research.microsoft.com/en-us/um/people/manik/pubs%5CdeCampos09.pdf

[40] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed 2015-07-07]. [Online]. Available: http://www.scipy.org/

[41] Y. Lecun and C. Cortes, "The MNIST database of handwritten digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[42] S. H. Cardoso, "Parts of the nerve cell and their functions," accessed: 2015-05-18. [Online]. Available: https://python-pillow.github.io/

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[44] M. Côté and H. Larochelle, "An infinite restricted boltzmann machine," *CoRR*, vol. abs/1502.02476, 2015. [Online]. Available: http://arxiv.org/abs/1502.02476

[45] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *J. Mach. Learn. Res.*, vol. 11, pp. 625–660, Mar. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756025

Appendix A

Appendix

A.1 MNIST 200-200 Sigmoidal Network No Penalty Term

```
average:  2.015  std: +- 0.138076065993
LayerOne Activations
0.406032316542  std: +- 0.00528142770508
LayerTwo Activations
0.334163289287  std: +- 0.00800826895513
[ 2.03  1.95  2.08  2.01  2.12  2.27  1.76  1.74  1.89  1.96  2.02  2.14
  1.88  2.04  2.08  1.93  2.28  1.83  2.18  2.03  1.98  2.31  1.83  2.11
  2.02  1.97  2.05  2.07  1.98  1.91]
```

A.2 MNIST 200-200 Sigmoidal Network log penalty

```
average:  1.93566666667  std: +- 0.118678370209
LayerOne Activations
0.279282081424  std: +- 0.00510171660576
LayerTwo Activations
0.193012877292  std: +- 0.0042919473522
[ 2.01  2.06  1.9   1.96  1.89  1.97  1.99  1.96  2.2   1.85  1.87  1.95
  1.82  1.68  1.87  2.03  2.01  1.95  2.29  1.87  1.93  1.76  1.88  1.73
  1.93  1.95  1.9   1.92  2.01  1.93]
```

A.3 MNIST 200-200 Sigmoidal Network L1 penalty

```
average:  2.30633333333  std: +- 0.130651784867
LayerOne Activations
0.152066204159  std: +- 0.00335341178362
LayerTwo Activations
0.126760358864  std: +- 0.0036927543879
[ 2.17  2.33  2.3   2.4   2.27  2.11  2.51  2.15  2.3   2.43  2.14  2.32
  2.4   2.46  2.46  2.32  2.71  2.18  2.28  2.3   2.22  2.11  2.36  2.41
  2.17  2.31  2.26  2.32  2.17  2.32]
```

A.4 MNIST 200-200 SPAF-Network No Penalty Term

```
average:  1.685  std: +- 0.0726521392206
average activations Layer 1:
average:  0.442140897767  std: +- 0.00733054683157
average activations Layer 2:
average:  0.245269342607  std: +- 0.00680615541421
```

A.5 MNIST 200-200 SPAF-Network log penalty

```
average:  1.69166666667  std: +- 0.0976757674941
LayerOne Activations
0.378140412225  std: +- 0.0058100137639
LayerTwo Activations
0.261579565372  std: +- 0.0100946314887
[ 1.8   1.73  1.79  1.66  1.76  1.55  1.6   1.62  1.6   1.62  1.6   1.65
  1.74  1.64  1.67  1.66  1.64  1.59  1.92  1.82  1.61  1.69  1.92  1.74
  1.84  1.66  1.55  1.73  1.62  1.73]
```

A.6 MNIST 200-200 SPAF-Network L1 penalty

```
average:  1.999  std: +- 0.120784932835
LayerOne Activations
0.286514413471  std: +- 0.00848186569934
```

```
LayerTwo Activations
0.30432592998  std: +- 0.017904758007
[ 1.88  1.78  2.05  2.13  1.94  1.87  2.02  1.94  1.88  1.81  2.14  2.11
  1.98  2.02  2.06  2.02  2.07  2.03  1.81  2.09  1.93  2.06  2.21  1.78
  2.04  2.04  1.92  1.96  2.17  2.23]
```

A.7 Hnd dataset 200-200 Sigmoidal-Network L1 penalty

```
average:  50.6884057971  std: +- 1.74530907283
LayerOne Activations
0.332040828486  std: +- 0.00443763731482
LayerTwo Activations
0.241891155534  std: +- 0.00601108034783
[ 52.93478261  48.15217391  50.97826087  47.93478261  53.15217391
  51.73913043  50.32608696  51.30434783  49.02173913  49.45652174
  52.06521739  50.76086957  51.19565217  51.41304348  48.91304348
  54.7826087   49.89130435  53.47826087  50.54347826  50.43478261
  52.2826087   47.82608696  51.52173913  49.89130435  49.56521739
  48.47826087  51.95652174  49.23913043  52.60869565  48.80434783]
```

A.8 Hnd dataset 200-200 Sigmoidal-Network log penalty

```
average:  50.4130434783  std: +- 1.46003254144
LayerOne Activations
0.278877926569  std: +- 0.00496434754724
LayerTwo Activations
0.196698051929  std: +- 0.00514898714236
[ 51.19565217  50.43478261  51.73913043  49.67391304  52.06521739
  50.2173913   50.65217391  50.76086957  49.02173913  48.36956522
  51.30434783  50.          50.10869565  51.95652174  48.58695652
  54.13043478  49.45652174  49.89130435  48.15217391  49.89130435
  51.63043478  47.82608696  50.10869565  50.65217391  50.76086957
  49.13043478  53.47826087  48.80434783  52.06521739  50.32608696]
```

A.9 Hnd dataset 200-200 Sigmoidal-Network No penalty

```
average:  50.7934782609  std: +- 1.69629075675
LayerOne Activations
0.413474922148  std: +- 0.00503668606661
LayerTwo Activations
0.304151464329  std: +- 0.00752188960895
[ 52.39130435  51.08695652  49.45652174  49.34782609  50.86956522
  51.19565217  49.7826087   53.15217391  49.89130435  49.67391304
```

```
  51.52173913  50.97826087  54.23913043  51.84782609  48.91304348
  55.65217391  50.10869565  48.58695652  49.56521739  49.7826087
  50.97826087  46.63043478  51.08695652  51.19565217  51.63043478
  50.43478261  52.60869565  50.32608696  50.76086957  50.10869565]
```

A.10 Hnd dataset 200-200 SPAF-Network L1 penalty

```
average:  48.0688405797  std: +- 1.87021461028
LayerOne Activations
0.422996403438  std: +- 0.00555557375641
LayerTwo Activations
0.36545346276  std: +- 0.015222697638
[ 48.80434783  48.69565217  48.58695652  45.65217391  48.69565217
  47.60869565  44.23913043  49.56521739  44.56521739  45.86956522
  48.36956522  47.93478261  49.45652174  50.          47.60869565
  51.63043478  48.91304348  46.73913043  46.08695652  47.2826087
  51.63043478  46.52173913  49.34782609  48.47826087  47.82608696
  48.58695652  51.08695652  45.32608696  49.7826087   47.17391304]
```

A.11 Hnd dataset 200-200 SPAF-Network log penalty

```
average:  47.6920289855  std: +- 1.83342283679
LayerOne Activations
0.413656007672  std: +- 0.00695700445938
LayerTwo Activations
```

```
0.379042815457  std: +- 0.012717972992
[ 48.69565217  45.86956522  47.93478261  47.60869565  47.2826087
  49.67391304  46.08695652  49.23913043  44.7826087   49.34782609
  45.2173913   47.39130435  47.82608696  47.93478261  45.2173913
  50.43478261  47.82608696  49.13043478  47.2826087   46.73913043
  49.02173913  43.58695652  47.7173913   49.45652174  48.47826087
  48.26086957  50.54347826  45.10869565  51.19565217  45.86956522]
```

A.12 Hnd dataset 200-200 SPAF-Network No penalty

```
average:  48.2898550725  std: +- 1.52353241472
LayerOne Activations
0.455611277365  std: +- 0.00735349748563
LayerTwo Activations
0.354987800169  std: +- 0.0125050934264
[ 50.97826087  46.95652174  49.13043478  46.84782609  47.39130435
  47.82608696  45.86956522  48.36956522  47.17391304  49.7826087
  47.2826087   47.7173913   48.80434783  49.23913043  45.86956522
  49.56521739  49.67391304  47.5         49.34782609  50.10869565
  48.69565217  46.19565217  48.47826087  47.5         48.36956522
  46.19565217  51.30434783  48.26086957  51.41304348  46.84782609]
```

A.13 Img dataset 500-400 Sigmoidal-Network No penalty

```
average:  29.375  std: +- 0.952996917624
average activations Layer 1:
average:  0.333074501102  std: +- 0.00429321012854
average activations Layer 2:
average:  0.224190098619  std: +- 0.00276707496876
```

A.14 Img dataset 500-400 Sigmoidal-Network L1 penalty

```
average:  31.3854166667  std: +- 1.43285026986
average activations Layer 1:
average:  0.116259871115  std: +- 0.00452441823107
average activations Layer 2:
average:  0.107110430289  std: +- 0.00258336032168
```

A.15 Img dataset 500-400 Sigmoidal-Network log penalty

```
average:  29.2239583333  std: +- 0.733783745438
average activations Layer 1:
average:  0.263147223218  std: +- 0.00536961512996
```

average activations Layer 2:

average: 0.194161940109 std: +- 0.0021697675218

A.16 Img dataset 500-400 SPAF-Network No penalty

average: 27.8489583333 std: +- 0.882670373269

average activations Layer 1:

average: 0.252317078138 std: +- 0.00661106141532

average activations Layer 2:

average: 0.258295582197 std: +- 0.0114883552225

A.17 Img dataset 500-400 SPAF-Network L1 penalty

average: 30.953125 std: +- 1.16463038441

average activations Layer 1:

average: 0.116478605554 std: +- 0.00231731625868

average activations Layer 2:

average: 0.10877348584 std: +- 0.00174710700659

A.18 Img dataset 500-400 SPAF-Network log penalty

average: 27.6822916667 std: +- 0.767323795598

average activations Layer 1:

average: 0.219429005757 std: +- 0.00998341604752

```
average activations Layer 2:
average:  0.295352717604  std: +- 0.0088516773689
```

A.19 Fnt dataset 50-50 SPAF-Network

```
average:  36.4157706093  std: +- 1.46708424177
LayerOne Activations
0.487105370405  std: +- 0.00913897135714
LayerTwo Activations
0.443889845583  std: +- 0.0134994356665
[ 35.16129032  36.77419355  38.27956989  37.41935484  36.02150538
  36.66666667  36.12903226  36.55913978  34.19354839  36.34408602
  35.05376344  36.12903226  36.4516129   37.09677419  36.12903226
  35.69892473  36.34408602  37.84946237  34.7311828   39.89247312
  36.88172043  33.87096774  34.51612903  37.52688172  34.19354839
  37.95698925  38.49462366  38.92473118  36.66666667  34.51612903]
```

A.20 Fnt dataset 100-100 SPAF-Network

```
average:  32.5448028674  std: +- 1.90848014822
LayerOne Activations
0.469396633084  std: +- 0.00770533720206
LayerTwo Activations
0.406812487123  std: +- 0.0140451630365
[ 32.25806452  34.51612903  32.3655914   33.97849462  30.64516129
```

```
  35.2688172   30.96774194  32.58064516  31.50537634  32.79569892
  30.21505376  31.72043011  35.69892473  32.25806452  30.86021505
  31.39784946  31.1827957   32.25806452  29.56989247  37.95698925
  31.39784946  32.04301075  30.75268817  34.94623656  29.78494624
  32.79569892  35.2688172   33.33333333  33.87096774  32.15053763]
```

A.21 Fnt dataset 200-200 SPAF-Network

```
average:  31.1935483871  std: +- 1.49312732917
LayerOne Activations
0.446805943969  std: +- 0.00603755095091
LayerTwo Activations
0.377071201389  std: +- 0.00867185986223
[ 30.96774194  31.61290323  31.93548387  30.64516129  29.56989247
  32.79569892  30.75268817  31.82795699  30.          30.86021505
  29.24731183  30.53763441  33.65591398  32.04301075  30.10752688
  30.96774194  31.50537634  31.61290323  29.03225806  34.08602151
  30.10752688  30.86021505  28.49462366  33.11827957  29.24731183
  31.93548387  34.62365591  32.90322581  30.          30.75268817]
```

A.22 Fnt dataset 400-400 SPAF-Network

```
average:  31.2688172043  std: +- 1.87713727573
LayerOne Activations
0.413123027841  std: +- 0.00368036066158
```

```
LayerTwo Activations
0.357652071463  std: +- 0.0105627227114
[ 31.1827957   31.72043011  31.29032258  34.08602151  29.35483871
  33.5483871   33.5483871   32.79569892  28.92473118  31.29032258
  28.49462366  30.75268817  33.5483871   31.61290323  31.1827957
  29.03225806  31.07526882  31.1827957   28.27956989  34.83870968
  29.46236559  31.82795699  28.17204301  33.01075269  29.13978495
  31.07526882  34.19354839  32.90322581  31.1827957   29.35483871]
```

A.23 Fnt dataset 700-700 SPAF-Network

```
average:  31.29390681  std: +- 1.53816902587
LayerOne Activations
0.379521640686  std: +- 0.00593606415207
LayerTwo Activations
0.334115013532  std: +- 0.0159624770869
[ 30.75268817  32.68817204  31.39784946  33.11827957  29.56989247
  33.76344086  31.50537634  32.58064516  29.46236559  32.04301075  30.
  29.78494624  33.65591398  30.64516129  31.61290323  29.89247312
  31.93548387  30.75268817  28.49462366  34.51612903  30.10752688
  30.64516129  29.67741935  33.11827957  29.35483871  31.61290323
  33.65591398  31.93548387  30.10752688  30.43010753]
```

A.24 Fnt dataset 1000-1000 SPAF-Network

```
average:  31.4121863799  std: +- 1.75123261973
LayerOne Activations
0.353617485583  std: +- 0.0060085445815
LayerTwo Activations
0.294775072224  std: +- 0.0170667138442
[ 29.78494624  32.47311828  31.1827957   33.33333333  29.56989247
  34.08602151  31.72043011  33.11827957  29.67741935  32.25806452
  29.46236559  29.78494624  34.51612903  31.07526882  29.13978495
  30.43010753  31.82795699  31.72043011  29.13978495  35.69892473
  30.75268817  31.82795699  29.13978495  33.01075269  28.92473118
  32.15053763  33.44086022  31.93548387  30.21505376  30.96774194]
```

A.25 Fnt dataset 50-50 Sigmoidal-Network

```
average:  36.1612903226  std: +- 1.75352356018
LayerOne Activations
0.486510907291  std: +- 0.0101482232436
LayerTwo Activations
0.443726148972  std: +- 0.0131973051622
[ 34.40860215  35.69892473  38.06451613  38.17204301  33.76344086
  37.31182796  38.06451613  38.17204301  34.30107527  37.20430108
  34.51612903  33.76344086  38.06451613  36.55913978  34.30107527
  37.41935484  34.19354839  36.66666667  34.83870968  40.53763441
  35.59139785  35.69892473  34.83870968  38.38709677  33.87096774
```

```
  35.59139785  37.84946237  36.55913978  36.23655914  34.19354839]
```

A.26 Fnt dataset 100-100 Sigmoidal-Network

```
average:  35.4659498208  std: +- 1.62055136109
LayerOne Activations
0.441502943289  std: +- 0.0053899264678
LayerTwo Activations
0.456047860258  std: +- 0.00604340809631
[ 34.7311828   38.17204301  35.59139785  34.08602151  35.69892473
  38.17204301  32.90322581  38.92473118  34.62365591  34.94623656
  33.65591398  34.40860215  36.88172043  35.48387097  33.44086022
  35.91397849  35.80645161  36.34408602  32.58064516  37.20430108
  35.16129032  34.40860215  33.44086022  36.66666667  33.01075269
  35.80645161  37.6344086   36.23655914  36.4516129   35.59139785]
```

A.27 Fnt dataset 200-200 Sigmoidal-Network

```
average:  33.7204301075  std: +- 1.64662478537
LayerOne Activations
0.406774377753  std: +- 0.00402400993644
LayerTwo Activations
0.428018714356  std: +- 0.004377557142
[ 33.11827957  33.97849462  34.40860215  33.76344086  33.76344086
  35.2688172   32.25806452  34.19354839  33.44086022  33.5483871
```

```
  32.25806452  33.01075269  35.48387097  31.93548387  31.29032258
  32.25806452  36.12903226  33.5483871   31.50537634  36.98924731
  32.47311828  33.97849462  32.15053763  35.59139785  31.1827957
  33.33333333  38.17204301  33.5483871   35.48387097  33.5483871 ]
```

A.28 Fnt dataset 300-300 Sigmoidal-Network

```
average:  33.1433691756  std: +- 1.75451964326
LayerOne Activations
0.378371809088  std: +- 0.00371037806331
LayerTwo Activations
0.401575450276  std: +- 0.00448703144657
[ 31.82795699  34.30107527  33.11827957  35.69892473  32.79569892
  36.55913978  33.44086022  34.19354839  32.25806452  32.58064516
  29.89247312  32.25806452  34.83870968  31.93548387  31.29032258
  32.90322581  33.44086022  34.30107527  30.10752688  36.55913978
  30.53763441  32.25806452  31.50537634  35.2688172   31.29032258
  33.87096774  36.02150538  33.33333333  32.79569892  33.11827957]
```

A.29 Fnt dataset 400-400 Sigmoidal-Network

```
average:  32.6559139785  std: +- 1.77036624222
LayerOne Activations
0.356917132398  std: +- 0.00346345997655
LayerTwo Activations
```

```
0.377263443836  std: +- 0.00406335149786
[ 30.96774194  34.83870968  33.65591398  32.25806452  30.64516129
  34.62365591  32.68817204  33.65591398  31.93548387  33.11827957  30.
  31.93548387  36.02150538  30.96774194  30.53763441  30.75268817
  33.01075269  31.50537634  30.10752688  36.02150538  31.1827957
  33.22580645  30.75268817  35.37634409  30.96774194  35.2688172
  34.08602151  33.65591398  33.11827957  32.79569892]
```

A.30 Fnt dataset 700-700 Sigmoidal-Network

```
average:  31.3440860215  std: +- 1.67795919996
LayerOne Activations
0.376454839656  std: +- 0.00728535358898
LayerTwo Activations
0.329812907765  std: +- 0.0186937725578
[ 30.53763441  32.25806452  30.96774194  33.65591398  28.49462366
  33.97849462  31.1827957   33.44086022  29.03225806  32.68817204
  29.78494624  30.86021505  33.76344086  31.50537634  30.64516129
  30.64516129  31.50537634  31.07526882  29.13978495  35.05376344
  29.35483871  31.1827957   28.92473118  32.79569892  28.8172043
  31.1827957   33.11827957  32.3655914   31.07526882  31.29032258]
```

A.31 Fnt dataset 1000-1000 Sigmoidal-Network

```
average:  32.1684587814  std: +- 1.785525521
```

```
LayerOne Activations
0.28455873079  std: +- 0.00355159054912
LayerTwo Activations
0.235650879549  std: +- 0.00323752780479
[ 31.07526882  33.76344086  33.11827957  32.90322581  30.          34.40860215
  31.61290323  33.87096774  30.64516129  33.33333333  29.13978495
  30.75268817  34.30107527  30.64516129  29.56989247  30.53763441
  31.93548387  31.72043011  31.1827957   35.37634409  30.75268817
  33.44086022  30.75268817  35.48387097  29.78494624  32.68817204
  35.59139785  32.79569892  31.72043011  32.15053763]
```

A.32 10% training Fnt dataset 200-200 Spaf-Network

```
average:  20.064516129  std: +- 1.64428256805
LayerOne Activations
0.435749242699  std: +- 0.00518925923181
LayerTwo Activations
0.32336094079  std: +- 0.00583672563172
[ 18.70967742  22.68817204  21.07526882  20.43010753  18.60215054
  22.47311828  20.96774194  21.61290323  17.09677419  19.24731183
  19.35483871  17.41935484  21.1827957   19.24731183  20.86021505]
```

A.33 20% training Fnt dataset 200-200 Spaf-Network

```
average:  16.4444444444  std: +- 1.42567651963
```

```
LayerOne Activations
0.429997410885  std: +- 0.00663367833362
LayerTwo Activations
0.305065258082  std: +- 0.00565056299617
[ 15.91397849  17.31182796  17.31182796  16.23655914  15.48387097
  20.32258065  16.77419355  17.52688172  15.2688172   15.2688172
  15.69892473  13.97849462  15.69892473  16.23655914  17.6344086 ]
```

A.34 50% training Fnt dataset 200-200 Spaf-Network

```
average:  13.1397849462  std: +- 1.58819788711
LayerOne Activations
0.415633734291  std: +- 0.0075062541002
LayerTwo Activations
0.281921876871  std: +- 0.00558575636773
[ 13.65591398  15.69892473  15.37634409  12.68817204  11.29032258
  13.97849462  14.62365591  14.40860215  10.          13.01075269
  13.22580645  10.43010753  13.01075269  13.33333333  12.3655914 ]
```

A.35 70% training Fnt dataset 200-200 Spaf-Network

```
average:  11.9498207885  std: +- 1.66758752069
LayerOne Activations
0.430604636595  std: +- 0.00619601230456
LayerTwo Activations
```

```
0.286950898226  std: +- 0.00383594349772
[ 12.3655914   13.44086022  13.65591398  13.87096774  10.10752688
  13.5483871   12.04301075  13.76344086   9.35483871  10.21505376
  13.5483871    8.92473118  12.04301075  10.53763441  11.82795699]
```

A.36 90% training Fnt dataset 200-200 Spaf-Network

```
average:  11.3189964158  std: +- 1.36772991286
LayerOne Activations
0.415176183216  std: +- 0.00598043664498
LayerTwo Activations
0.343000039861  std: +- 0.00644888652744
[ 12.58064516  11.61290323  13.01075269  14.19354839   9.03225806
  12.47311828  10.53763441  11.82795699   9.13978495  11.72043011
  11.07526882  10.32258065  10.21505376  10.53763441  11.50537634]
```

A.37 10% training Fnt dataset 200-200 Sigmoidal-Network

```
average:  21.0681003584  std: +- 1.86219947882
LayerOne Activations
0.415751518501  std: +- 0.0064212645445
LayerTwo Activations
0.381890000269  std: +- 0.00703455751795
[ 20.          23.76344086  21.61290323  22.25806452  17.74193548
```

```
  23.87096774  21.39784946  22.90322581  18.38709677  20.64516129
  20.86021505  18.06451613  22.15053763  20.21505376  22.15053763]
```

A.38 20% training Fnt dataset 200-200 Sigmoidal-Network

```
average:  17.5340501792  std: +- 1.64974256903
LayerOne Activations
0.411789855691  std: +- 0.00492026615895
LayerTwo Activations
0.35993252115  std: +- 0.00690541178238
[ 16.98924731  18.92473118  18.70967742  17.74193548  15.80645161
  22.15053763  18.38709677  17.84946237  15.80645161  16.4516129
  16.88172043  16.02150538  16.88172043  15.69892473  18.70967742]
```

A.39 50% training Fnt dataset 200-200 Sigmoidal-Network

```
average:  13.4982078853  std: +- 1.35947699339
LayerOne Activations
0.415952832776  std: +- 0.00741936162213
LayerTwo Activations
0.33557353736  std: +- 0.00655986624061
[ 14.51612903  15.2688172   15.2688172   12.68817204  11.82795699
```

```
  14.19354839  14.19354839  16.12903226  11.39784946  13.22580645
  12.25806452  12.58064516  13.65591398  12.04301075  13.22580645]
```

A.40 70% training Fnt dataset 200-200 Sigmoidal-Network

```
average:  12.3082437276  std: +- 1.3914198237
LayerOne Activations
0.412280607496  std: +- 0.00553717608909
LayerTwo Activations
0.352144577669  std: +- 0.00628866036513
[ 13.5483871   13.33333333  13.44086022  12.79569892  10.64516129
  14.51612903  13.44086022  13.33333333  10.10752688  10.75268817
  11.93548387   9.67741935  13.01075269  12.04301075  12.04301075]
```

A.41 90% training Fnt dataset 200-200 Sigmoidal-Network

```
average:  11.0896057348  std: +- 1.29084019885
LayerOne Activations
0.412105776905  std: +- 0.00598382952077
LayerTwo Activations
0.344297876639  std: +- 0.00808914513865
[ 12.47311828  12.25806452  13.11827957  12.25806452  10.          12.58064516
```

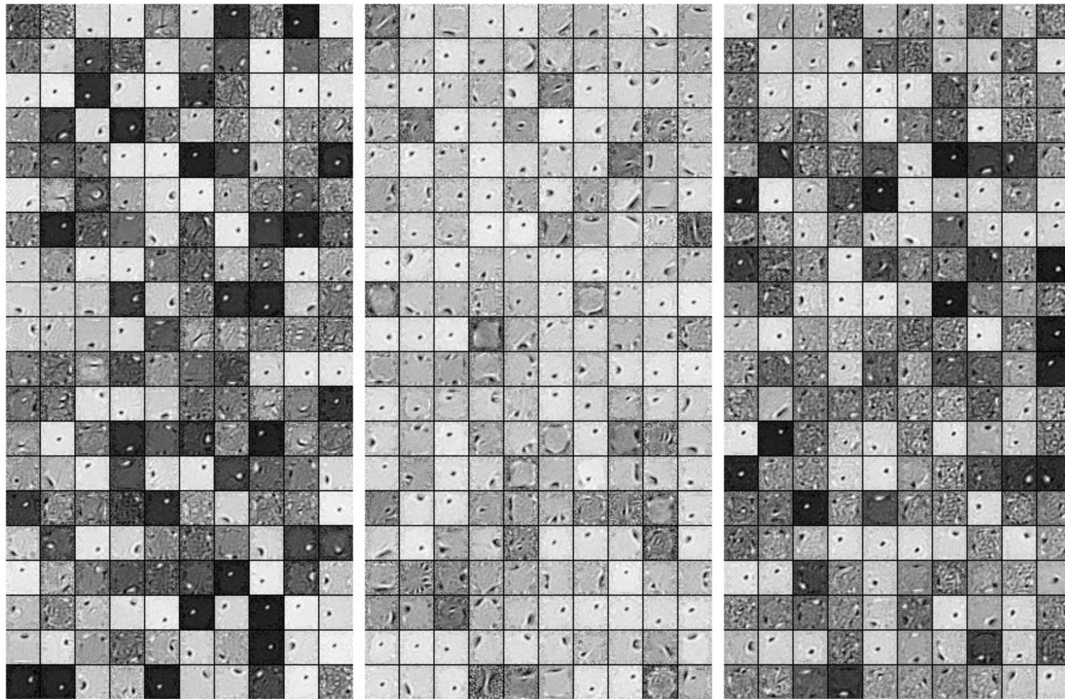
```
12.68817204  10.75268817   9.46236559  10.53763441  10.53763441
 9.35483871  10.21505376  10.75268817   9.35483871]
```

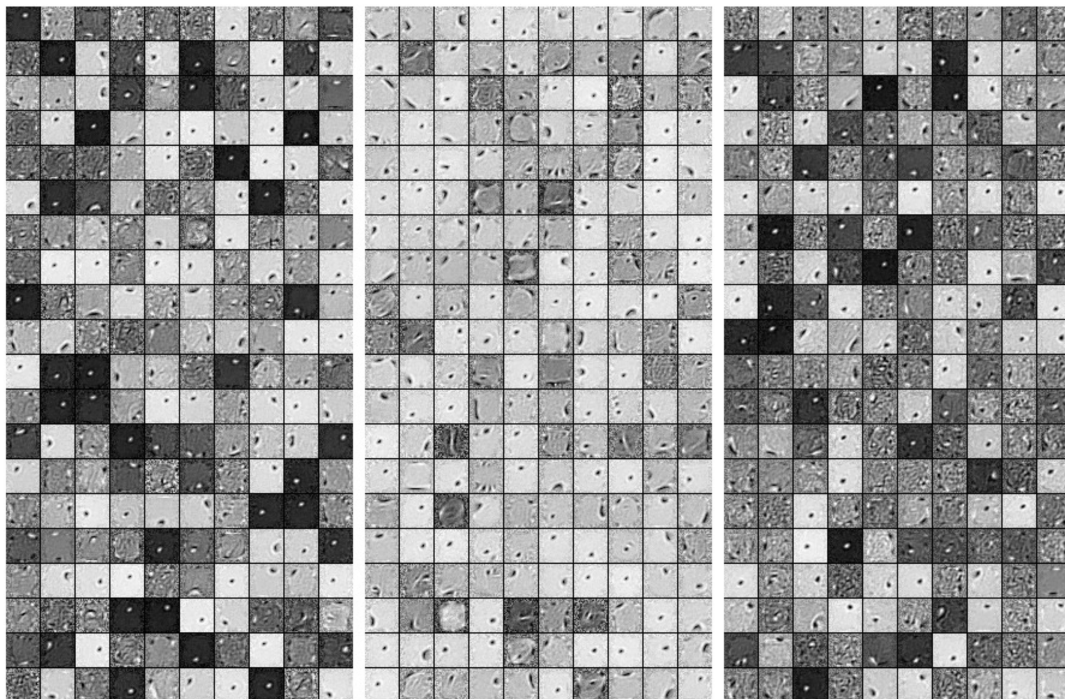
Appendix B

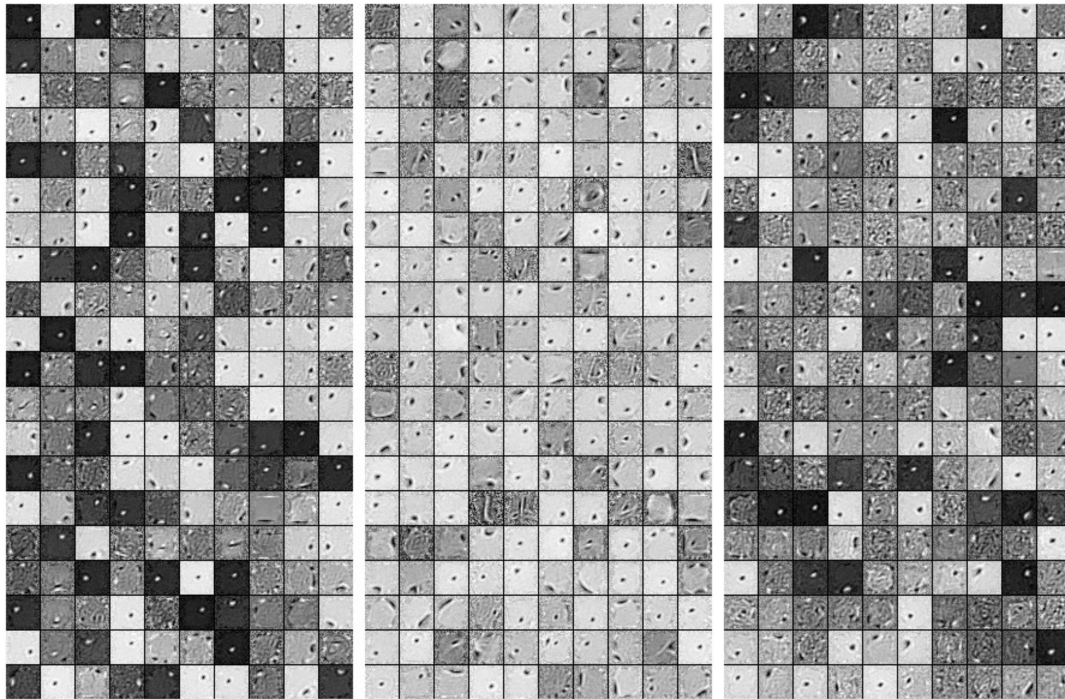
Appendix

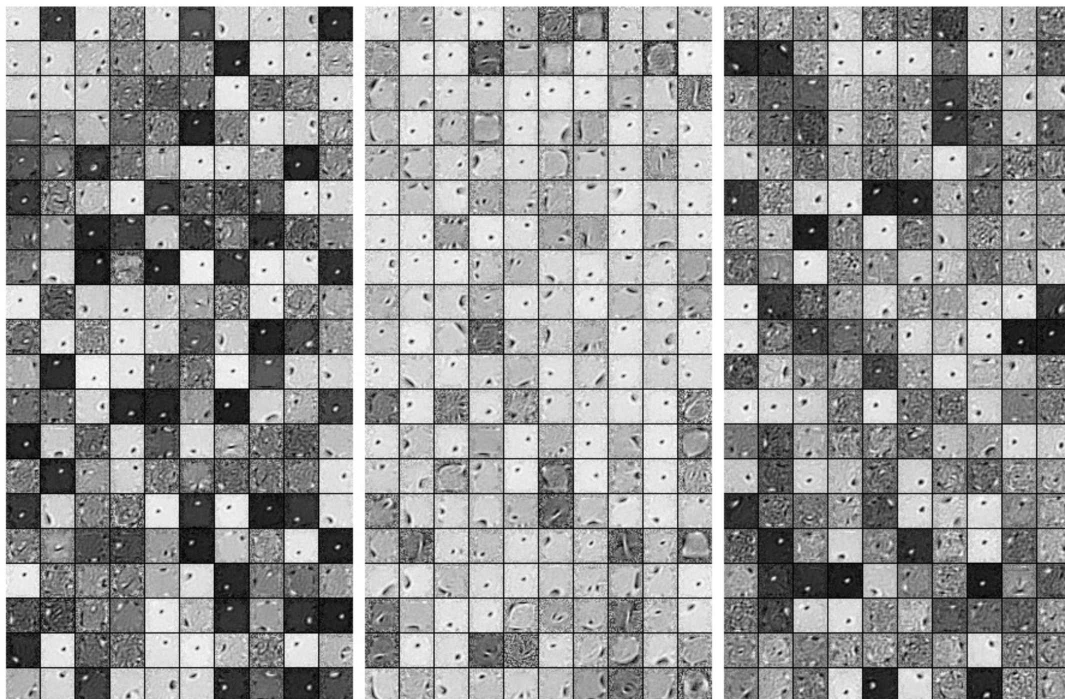
All feature images are in the order of: Newly proposed activation function, Relu SPAF networks followed by the normal sigmoidal networks.

B.1 MNIST Features Layer-1

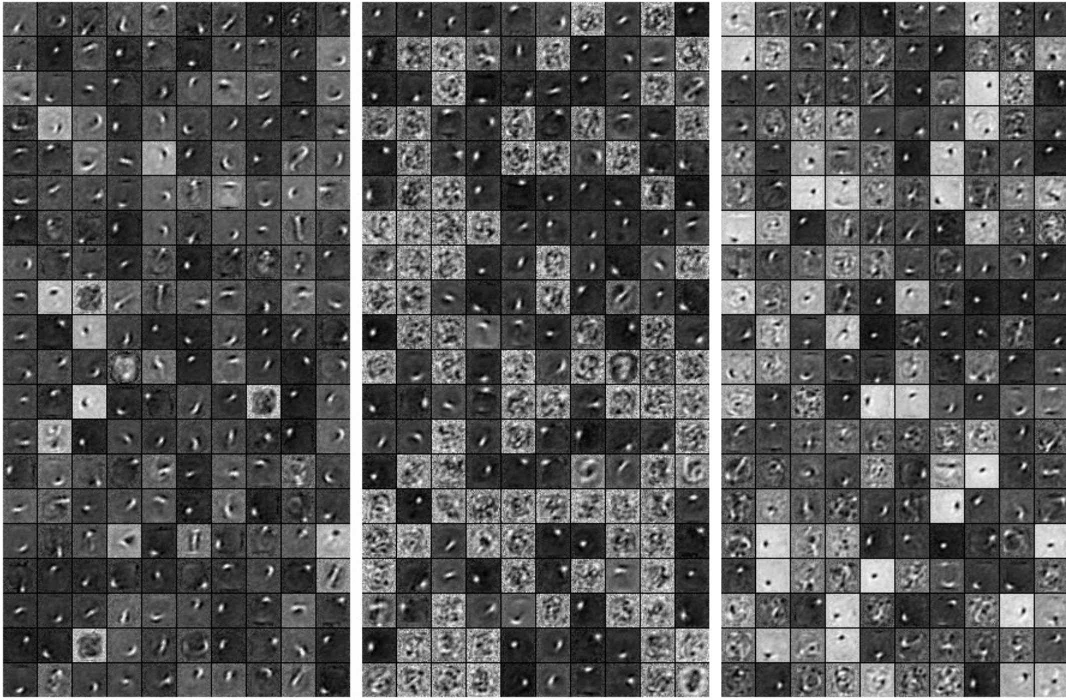


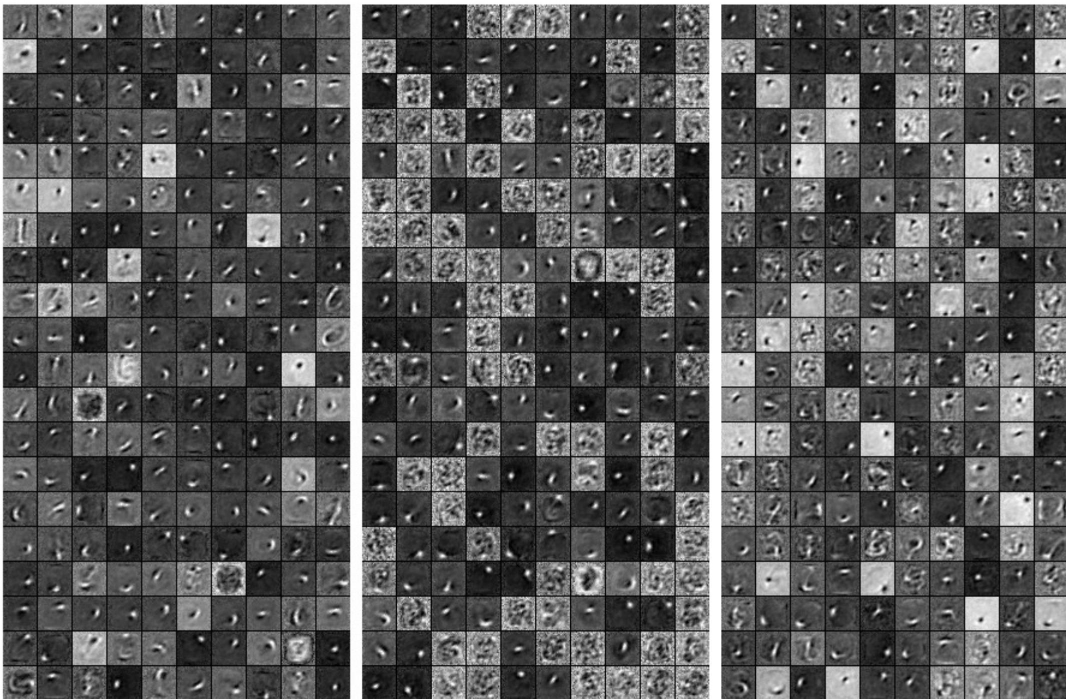


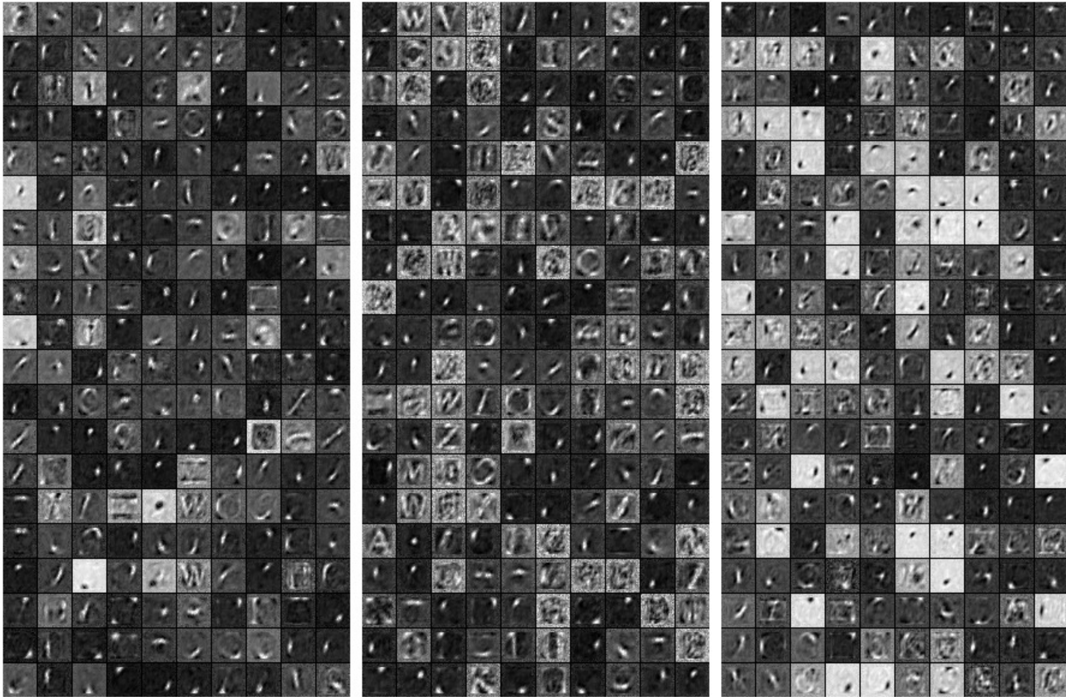


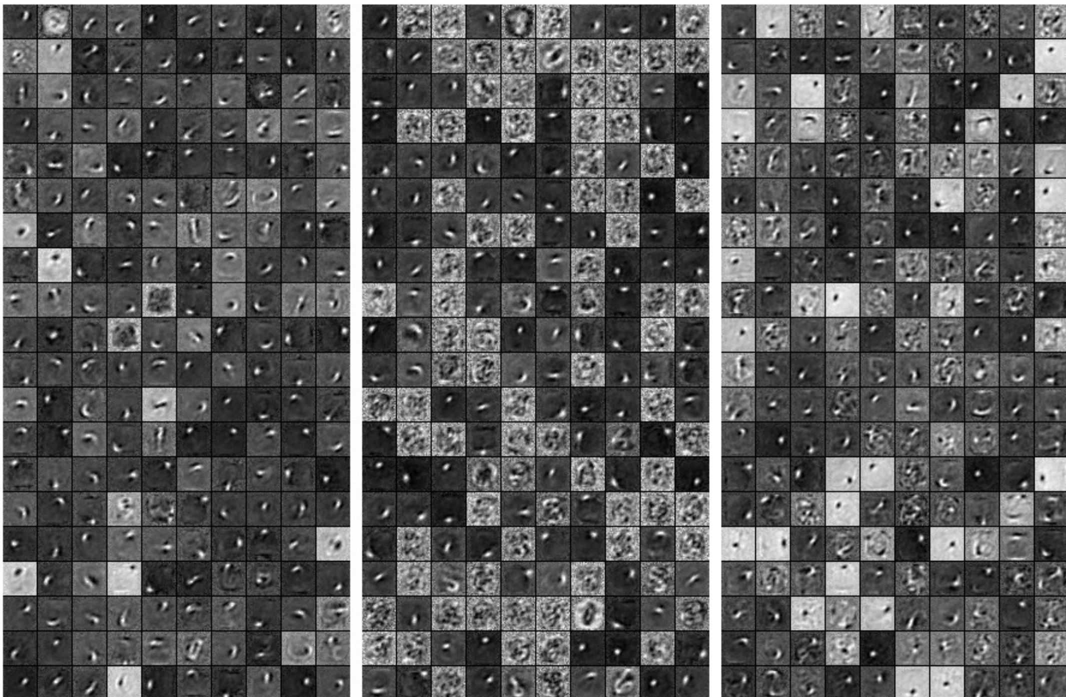


B.2 MNIST Features Layer-2

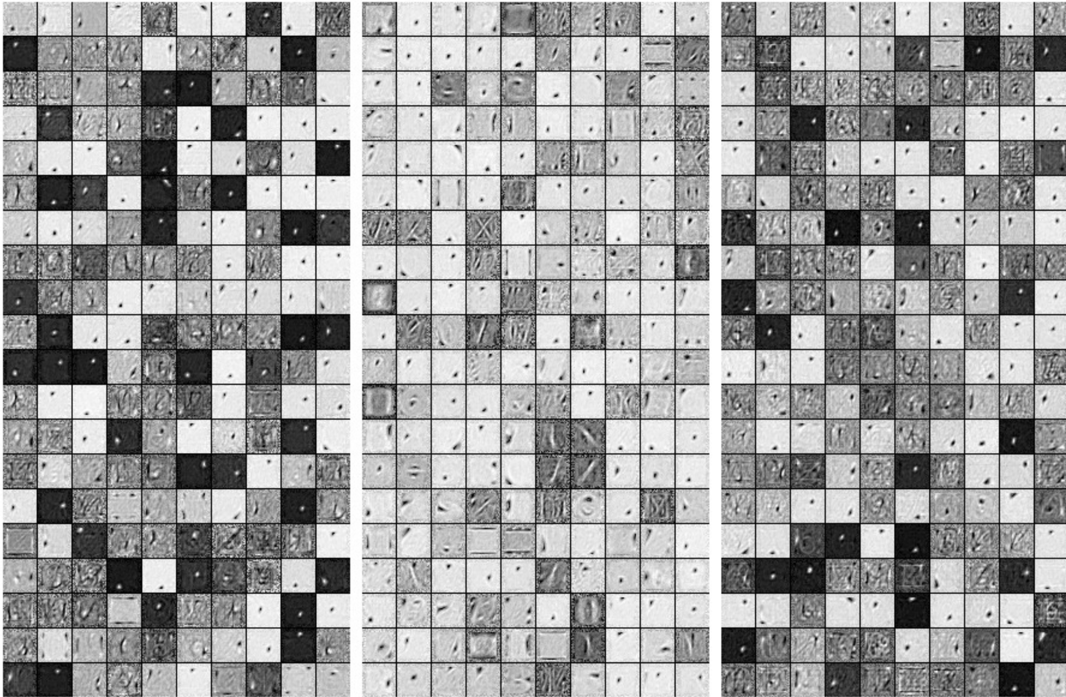


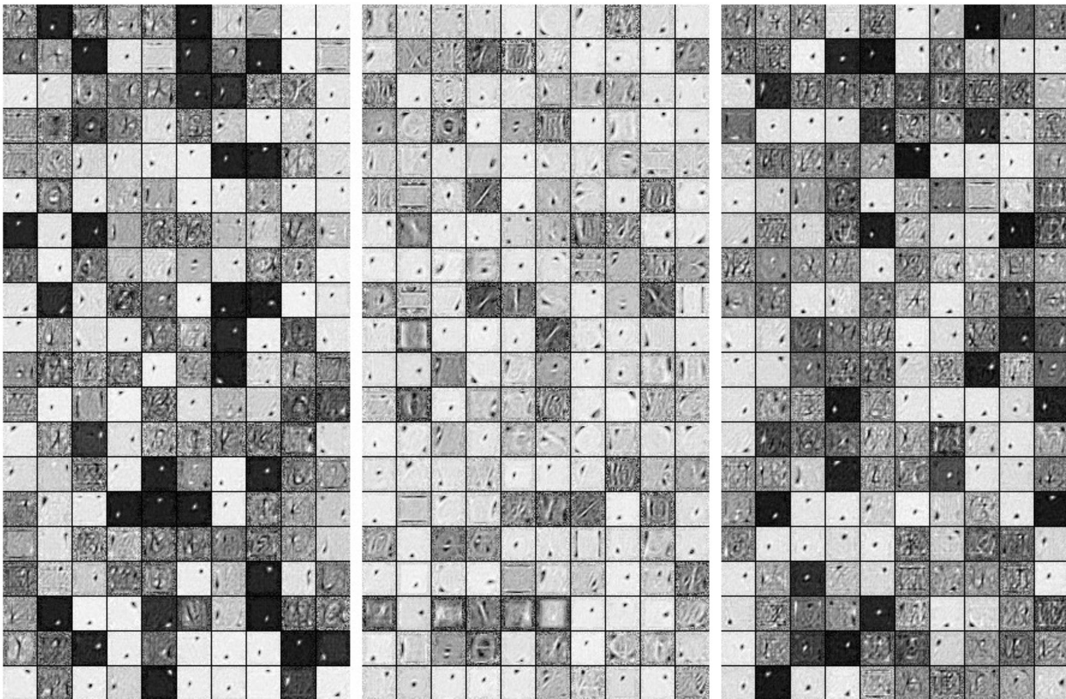


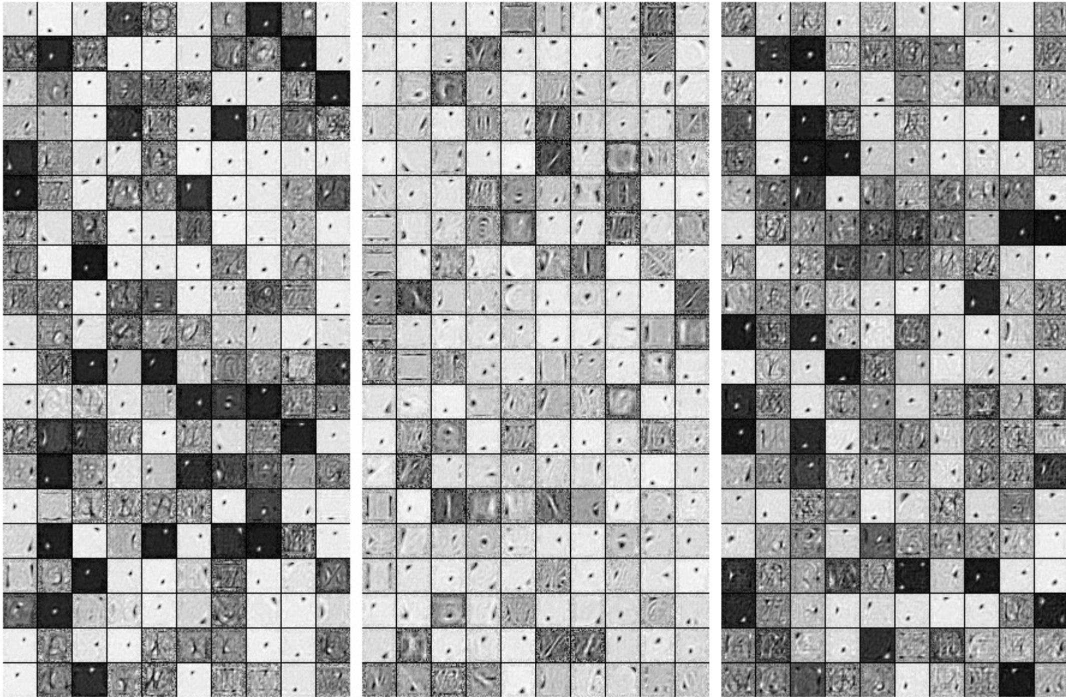


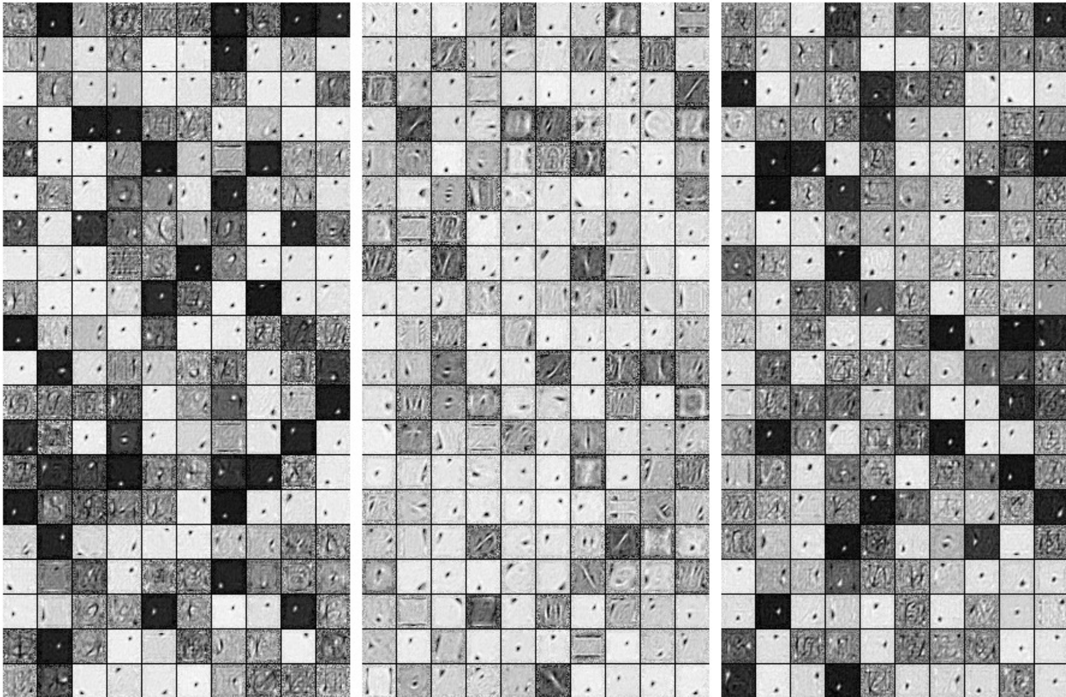


B.3 Fnt Features Layer-1

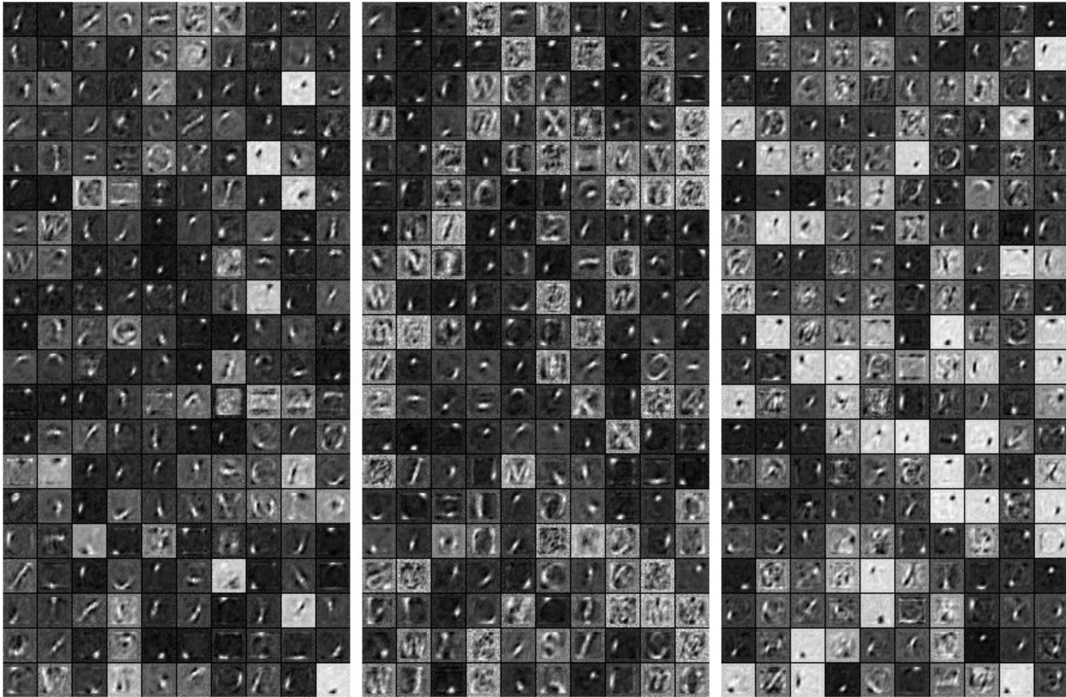


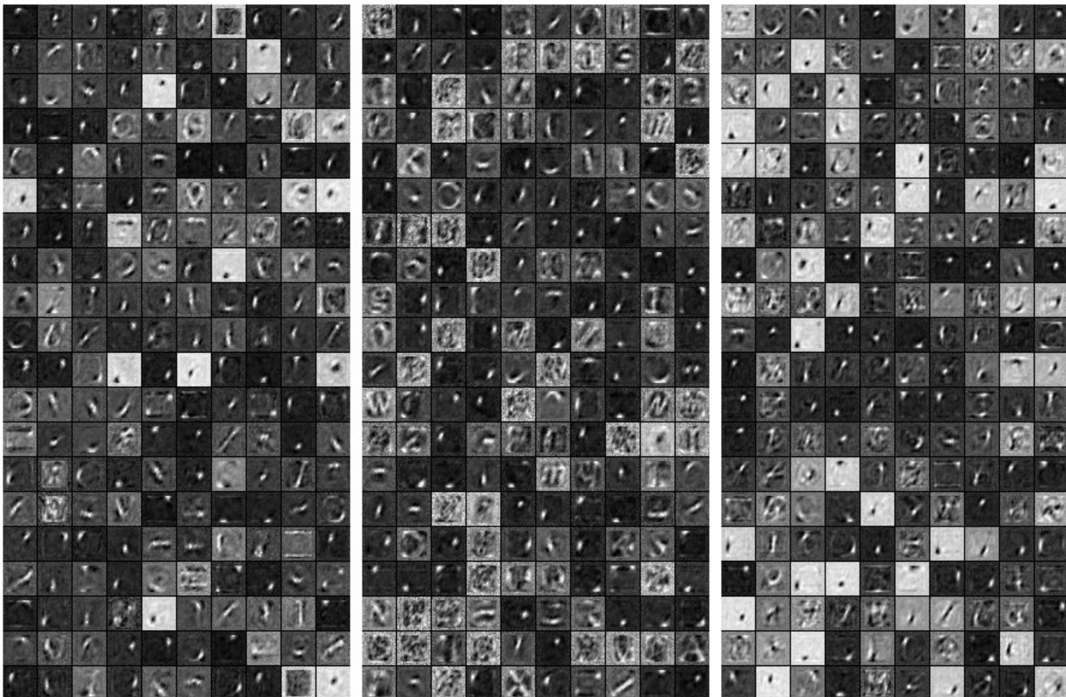


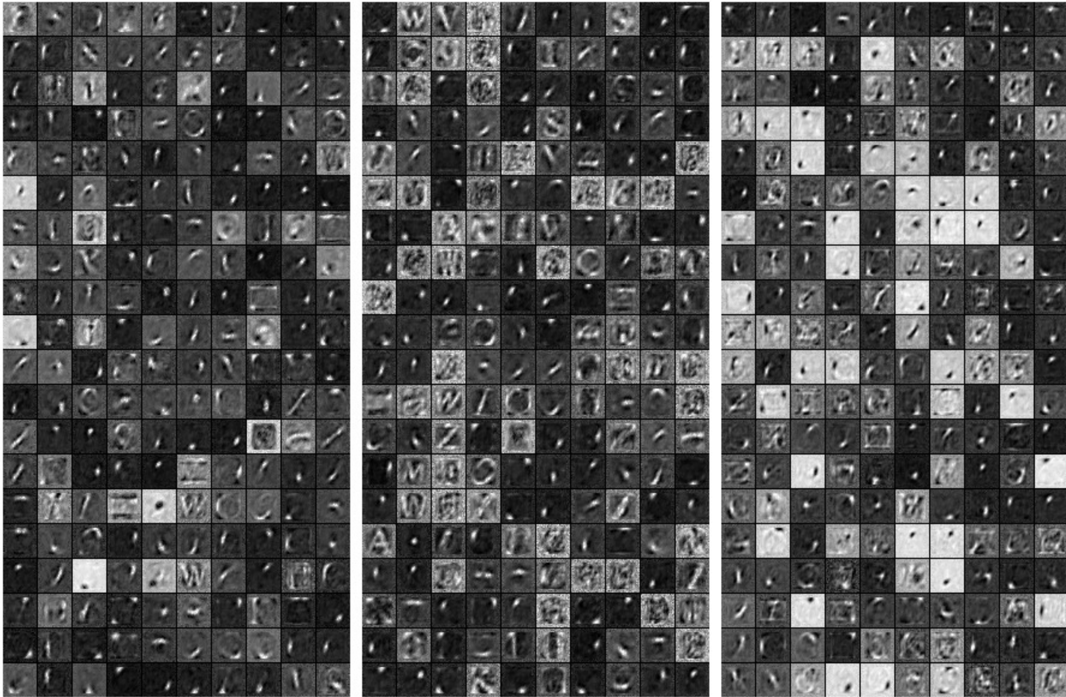


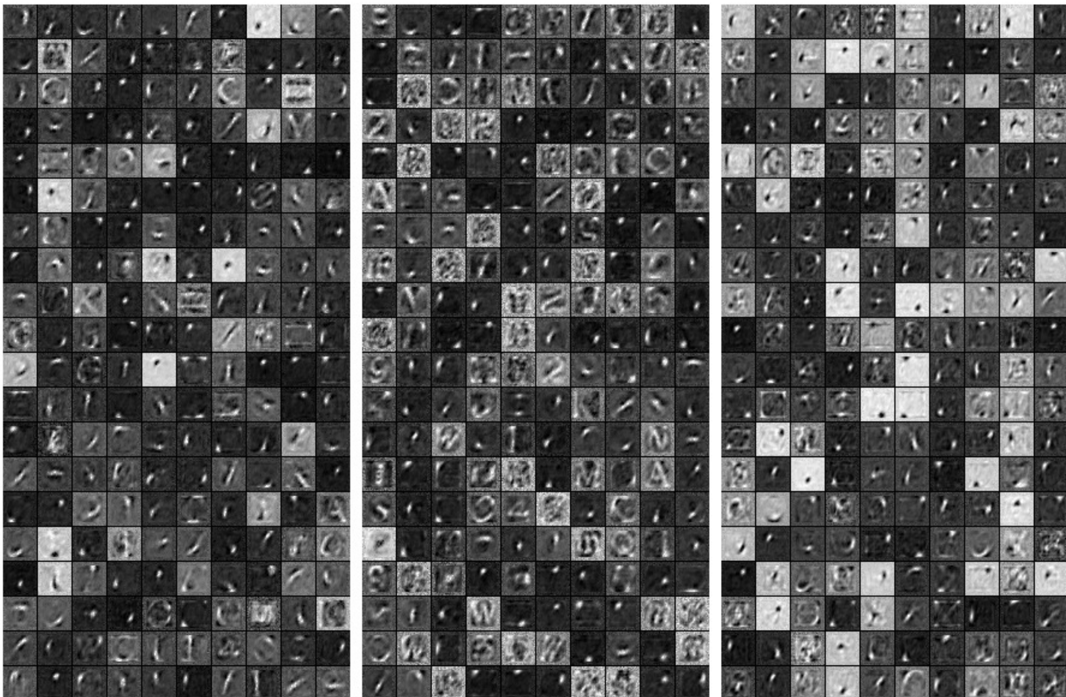


B.4 Fnt Features Layer-2

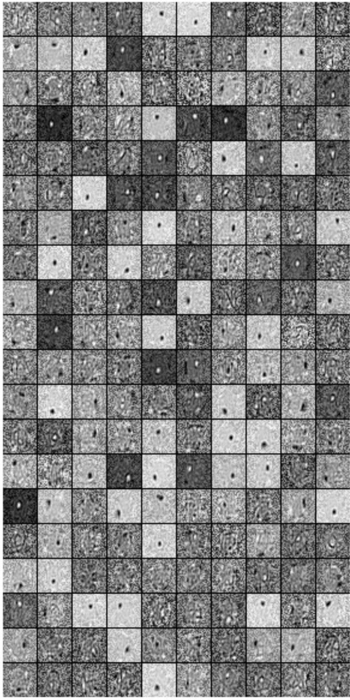
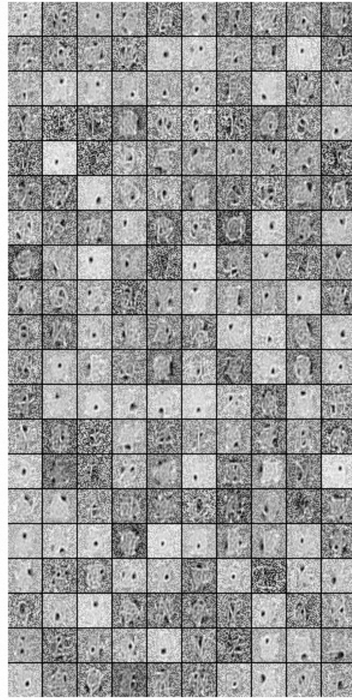
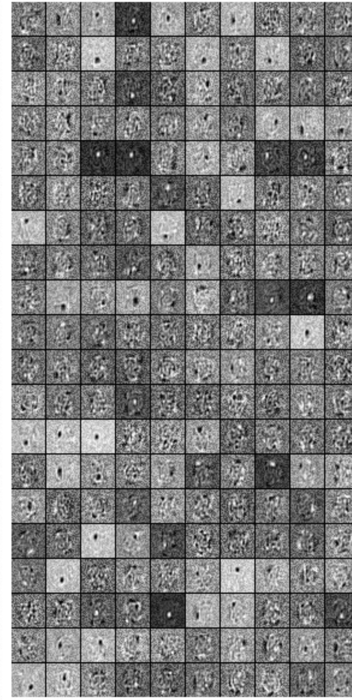
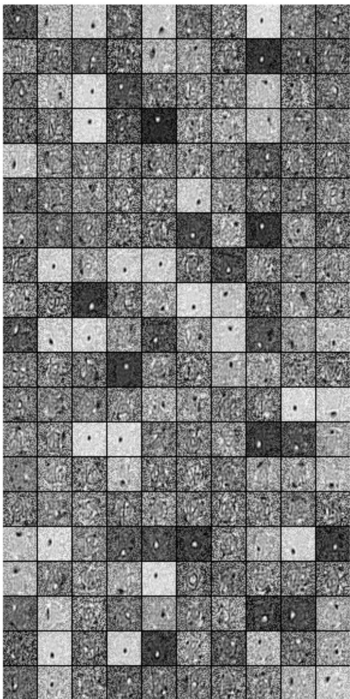
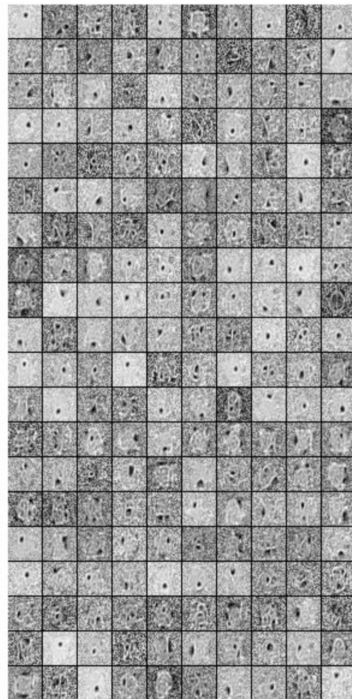
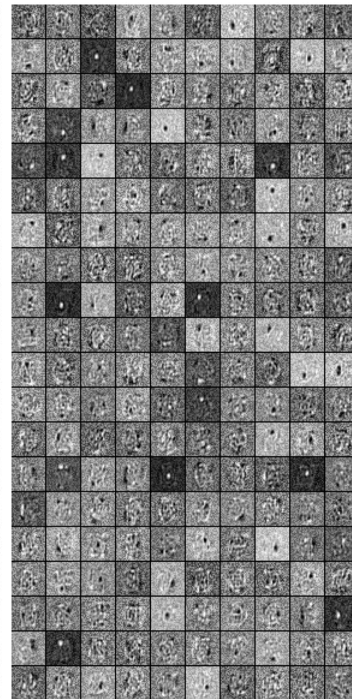


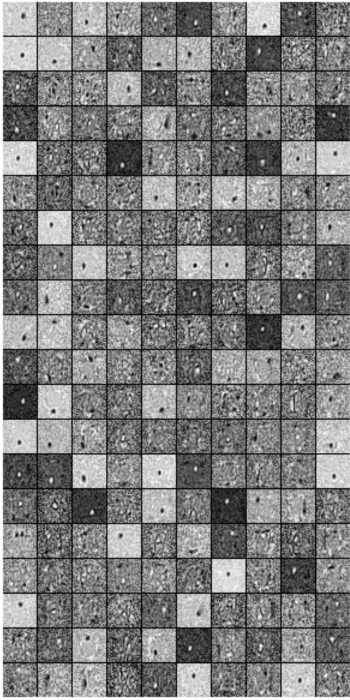
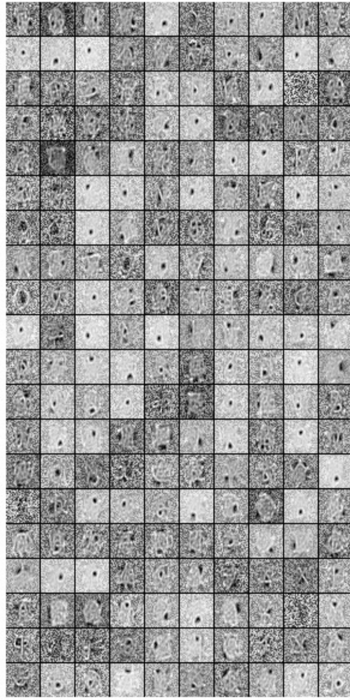
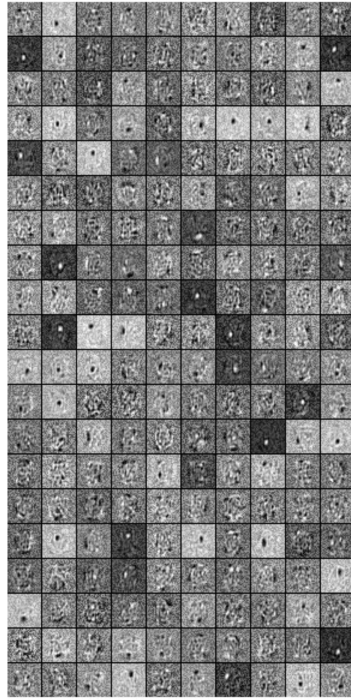
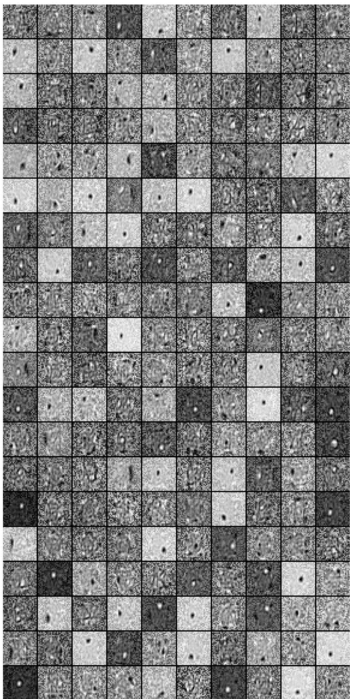
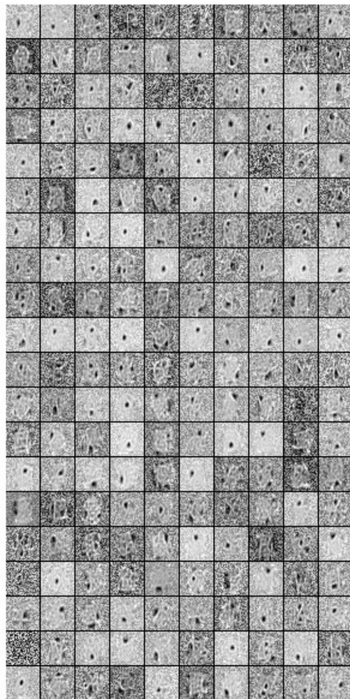
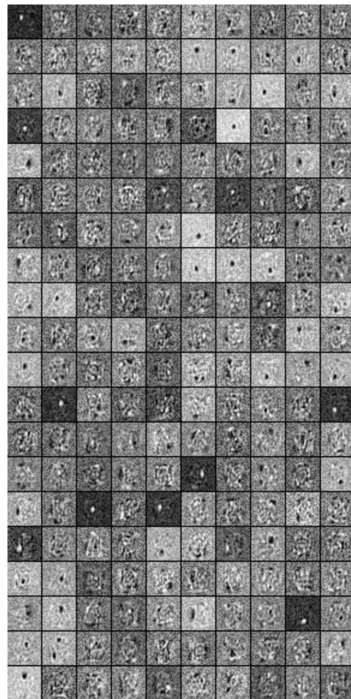






B.5 Hnd Features Layer-1

B.6 Hnd Features Layer-2