A Comparative Study of the Impact of Depth in Deep Learning Architectures

Kirsten Byers
University of Texas at El Paso

A COMPARATIVE STUDY OF THE IMPACT OF DEPTH

IN DEEP LEARNING ARCHITECTURES

KIRSTEN NICOLE BYERS

Master's Program in Computer Engineering

APPROVED:

Patricia Nava, Ph.D., Chair

John Moya, Ph.D.

Monika Akbar, Ph.D.

Stephen L. Crites, Jr., Ph.D.
Dean of the Graduate School

Dedication

To Mom, Dad, Geoff, Michelle, Katie, and Luis.

A COMPARATIVE STUDY OF THE IMPACT OF DEPTH IN DEEP LEARNING ARCHITECTURES

by

KIRSTEN BYERS, B.S.E.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

THE UNIVERSITY OF TEXAS AT EL PASO

May 2020

Acknowledgements

First and foremost, I would like to thank Dr. Patricia Nava, for her help and support not only with this thesis but in regard to my entire graduate career. Although this semester has brought several transitions and incredible circumstances, she has been an immense help and I certainly could not have done it without her. Dr. Nava, please know your support is not only greatly appreciated by myself but my entire family. Secondly, I would like to express my gratitude to the Electrical and Computer Engineering Department for challenging and motivating me during my 6 years with the department. My final acknowledgement is to the UCI Machine Learning Repository for the availability of their datasets used in this thesis along with clear dataset information and relevant papers used to guide my research.

Abstract

Machine Learning continues to evolve as applications become more complex. Neural Networks, or Deep Networks, are integral to machine learning and the entire taxonomy of Artificial Intelligence [Sze17]. Intelligent structures and algorithms continue to advance, keeping pace with the complexity of data. Changes in architecture, algorithms, and parameters are necessary to keep up with computational complexity and data available. This study focuses on how changes in depth of the architecture affect performance on three distinct datasets, including one on Heart Disease. An adaptable network is created in original code, trained, and tested on these datasets. Its performance parameters are observed in order to better understand when it is necessary to add depth and what depth to add to a network for each dataset. The study compares performance, which indicates that there are tradeoffs, and that users must understand the balance between the complexity of the problem and complexity of the architecture in order to effectively use these architectures.

Table of Contents

Acknowledgements v

Abstract vi

Table of Contents vii

List of Tables ix

List of Figures x

Chapter 1: Introduction 1
- Objective 1
- Organization 1

Chapter 2: Background 4
- ANN History 4
- Basic Feed Forward Architecture 6
- Alternative Architectures 7
  - Fully Connected vs. Sparesly Connected 7
  - Reccurent Neural Networks 8
  - Convolutional Neural Networks 9
- Activation Functions 10
- Feed-Forward Algorithm 11
- Early Stopping 14

Chapter 3: Discussion of Applications 16
- Computer Vision 16
- Language 18
- Medical 19

Chapter 4: Work & Results 21
- Creation of a Shallow Network 21
- Creation of a Deep Network 22
- Additions to Basic Architecture 23
- XOR Dataset 24
  - Results 25

Iris Dataset 26

Distributed vs. Encoded Outouts 28

Distributed....28

Encoded....29

Results....29

Heart Disease Dataset 30

Dataset Pre-Processing....31

Other Work 32

Results....33

Chapter 5: Conclusion....36

Future Work 36

References....38

Appendix....40

Vita …....45

List of Tables

Table 4.1: XOR functiontable 25

Table 4.2: Performance of FDLN model on XOR data 26

Table 4.3: Distributed representation of Iris type 28

Table 4.4: Encoded representation of Iris type 29

Table 4.5: Performance of FDLN model on Iris data 30

Table 4.6: Describtion of heart disease data attributes 31

Table 4.7: Describtion of shallow MLP model 32

Table 4.8: Performance of shallow MLP model 32

Table 4.9: Performance of FDLN model on Cleveland Heart Disease data 34

List of Figures

Figure 2.1: Biological Neuron 4

Figure 2.2: XOR problem example 5

Figure 2.3: a. Single layer network, b. Multi-layer network 7

Figure 2.4: Fully connected vs. Sparsely connected architecture 8

Figure 2.5: a. Recurrent neural network, b. Time layered representation 9

Figure 2.6: Convolutional neural network example 10

Figure 3.1: Regions with CNN features 17

Figure 4.1: Plot of parameters for different types of Irises 27

Introduction

Machine learning may seem like a newly emerging technology as there has been growing interest in self-driving cars, voice assistants, and robotic tasks [Sze17]. Contrary to how it may seem, machine learning (and its associated, and sometimes interchangeable, topics such as Artificial Intelligence and Artificial Neural Networks) have been around for decades. Starting in the 1940's and after several highs and lows of interest, machine learning, or more specifically, neural networks has evolved into its newest role: Deep Learning.

Deep learning uses the "depth" of the network, the number of layers in the net, to learn more computationally expensive tasks and provide better accuracy. While deep learning is a means to gain better prediction results, the outcome of the system is dependent on the application and moreover, the complexity of the architecture should be in balance with the complexity of the data set in use for the application.

Objective

The purpose of this thesis is to understand how the depth of a network will affect the performance of several different datasets. The datasets trained and tested are the XOR dataset, the Iris dataset for pattern recognition, and a dataset to predict the presence of heart disease in a patient. These datasets are trained and tested on a flexible deep learning network (FDLN) and the results are observed for changes in accuracy.

Organization

This thesis is organized into five chapters. The remainder of the chapters describing this study are organized as follows:

Chapter 2 covers the background of deep learning, including history, architectures, and the learning algorithm. Starting with the biological system that inspired the entire field, an artificial neuron model is presented. A modified version of this model, then, serves as the primary unit for creation of the Flexible Deep Learning Network (FDLN) model that is used in the rest of this study. This chapter also covers the highs and lows of interest in machine learning and major milestones in the field, as well as machine learning's current standing. Common architectures used in building artificial neural networks are also discussed. Finally, an algorithm for training is presented, which will be one used in the FDLN to make further observations.

In Chapter 3, popular applications for deep neural networks are discussed with common architectures presented for each. Computer vision continues to be an application, alongside speech and language, that is of great interest. Several examples of each are discussed, such as image classification and speech recognition. The medical field, while currently using machine learning for a variety of applications will likely continue to grow their demand for these systems. Thus, Deep Learning, Machine Learning and Artificial Intelligence will continue to be an increasingly important aspect of the medical field.

Chapter 4 begins with discussion surrounding the creation of the Flexible Deep Learning Network (FDLN) before covering results of the datasets tested. An initial simple dataset is utilized to validate the FDLN and its operation. The effect of architecture depth is analyzed with this dataset. The second dataset, the "Iris Classification Data" set, is incrementally more complex than the first, since the distinct classes are overlapped. In the case of the iris dataset, a conversation about the data representation of output values is covered before discussing study results determining the impact of architecture depth of the FCLN in iris results. Last, a well-known dataset, the Cleveland Heart Disease dataset, is utilized for the final phase of the study. The heart disease results include dialogue concerning pre-processing of data, and its impact on the FDLN's performance with the dataset.

Chapter 5 summarizes the findings of the study, encapsulates the results detailed in Chapter 4, and draws generalized conclusions for the study. Additionally, there are some thoughts presented for the future work, for anyone interested in taking the results of this study to the next logical step in investigating the effects of deep learning architecture on performance.

Background

ANN History

Artificial neural networks (ANN) have been around since the 1940's starting with the McCulloch and Pitts model, where the goal was to create a neural network model based on the human brain. The human brain has a structure of neurons connected to one another by axons and dendrites, which are then connected by synapses to make up the biological neural network, as illustrated in figure 2.1.

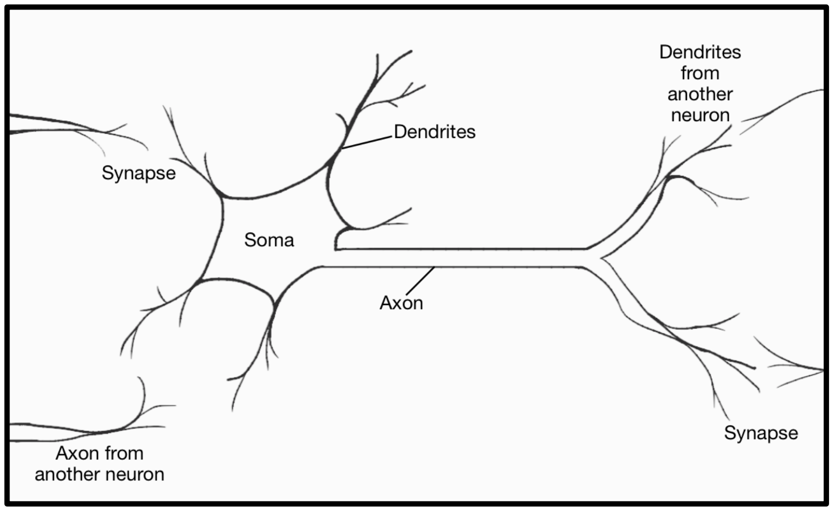


Figure 2.1: Biological neuron.

This pattern of connections makes up the architecture of the artificial neural network, whereas the adjustment of weights associated with each connection dictates how the network "learns." In biology, the soma sums the incoming signals from receiving synapses and when the soma receives sufficient input, it fires a signal through its own axon to other cells. McCulloch and Pitts first proposed a binary threshold as a computational function for their artificial neuron, this threshold would serve as functional analogy of the biological soma. The McCulloch-Pitts neuron computes a weighted sum of the inputs and compares that weighted sum to a threshold to determine its output. This method proved to work well for simple logic functions, such as the AND and OR functions [Faus94].

In the 1960's, Frank Rosenblatt introduced the perceptron which, like the McCulloch-Pitts neuron, used a threshold output function. The excitement over the perceptron came from its ability to learn via iterative weight adjustment that will eventually converge to the correct weights needed to reach the target output. However, this excitement was short lived due to the limitations of what a perceptron can learn. A major point researchers made against the perceptron, and against machine learning in general, was that the perceptron could not solve linearly separable problems, such as the simple XOR problem, shown in figure 2.2. The perceptron was made to converge reliably to zero error when the output classes are linearly separable. Minsky and Papert argued against neural networks due to the perceptron's inability to converge, due to the XOR function's linearly inseparable data, leading to the first downfall of neural networks [Faus94].

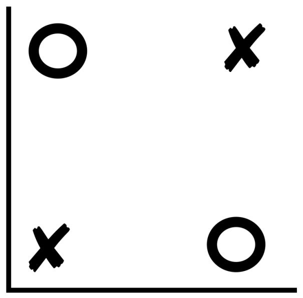


Figure 2.2: The XOR problem needs at least two decision surfaces to converge.

A secondary reason neural networks research was halted was due to an inability to train a multi-layer network. Enthusiasm returned to the study of neural networks with the addition of hidden layers, improved computational capabilities, and backpropagation, introduced by David

Rumelhart in the 1980's. Adding layers to the perceptron meant that the XOR problem could now be solved and the addition of backpropagation would make adjustments to the weights associated with connections to help the network converge.

Unfortunately, backpropagation came with its own stability and computational issues and had challenges with overfitting to data. This situation, once again, discouraged interest in neural networks. It wasn't until the early 2000's where a rise in computational power was able to aid in the computationally heavy tasks. At the same time there was more data availability to use in training networks and better methods to make adjustments during initialization. Ultimately three factors have shaped current deep learning: information availability, availability of computational capacity, and algorithmic development [Sze17]. To further aid deep learning, there are several different approaches on how to optimize training, including the use of different activation functions, pre-processing of data, learning algorithms with specific learning rates, and the use of momentum.

Basic Feed Forward Architecture

A basic neural network is comprised of a layer of inputs, a layer or layers of computational neurons called the "hidden layer(s)" and an output layer of computation neurons. A single layer network would be one comprised of just the input and output layer and so, only has one layer of connections, shown in figure 2.3a. Multilayer networks can be split into two fields, those being shallow and deep nets, with the distinction that "shallow" nets generally have one or two hidden layers and Deep Neural Networks (DNNs) have more hidden layers. DNNs are also referred to as Deep Learning Networks (DLNs). The term deep learning comes from the "depth" of a network, or how many layers a network has and anything greater than two hidden layers would be considered a deep net. The layer of inputs does no computational work and is used for the input of a single value for each node in the layer. The hidden layer consists of "feature detectors" that respond to features that appear in the input pattern [Day90]. This layer contains several neurons and may have several layers inside of it, all of which will result in some output based on a particular activation function. Finally, the output layer is another computational layer that will produce the final output value. Any of the layers, excluding the final output layer may also contain a bias used to offset the input to a layer. Once all the neurons are established in terms of their layers and functions, connections need to be made and given values to determine the weight of each connection.

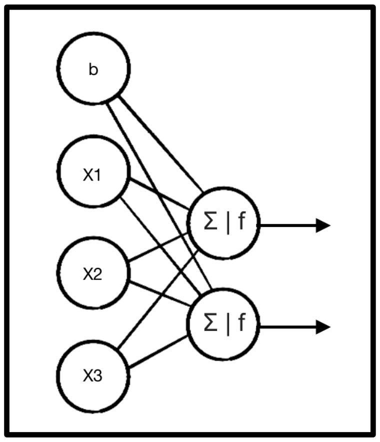


Figure 2.3a: Single Layer Network

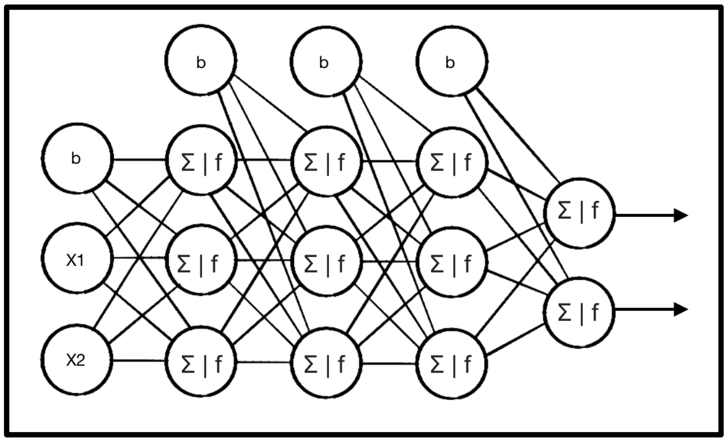


Figure 2.3b: Multi-Layer Network

Alternative Architectures

Fully Connected v. Sparsely Connected

The model previously presented is commonly called the Multilayer Perceptron (MLP) model and so has fully connected layers. However, fully connected nets require a large amount of storage and computation and, if needed, can be made into a sparsely connected network. Turning a fully connected MLP into a sparsely connected one is simply done by setting some weights to zero, this "removes" the connection and so removes the computational complexity by simply circumventing an operation whose result is known at the outset: multiplying by zero. Figure 2.4 illustrates a network where the connections between the input layer and hidden layer are fully connected but between the hidden layer and output layer, not all the hidden layer nodes are connected to every output node. In some cases, a network can be "pruned" during training to make it more efficient by removing the weights that don't give meaningful contribution to the output.

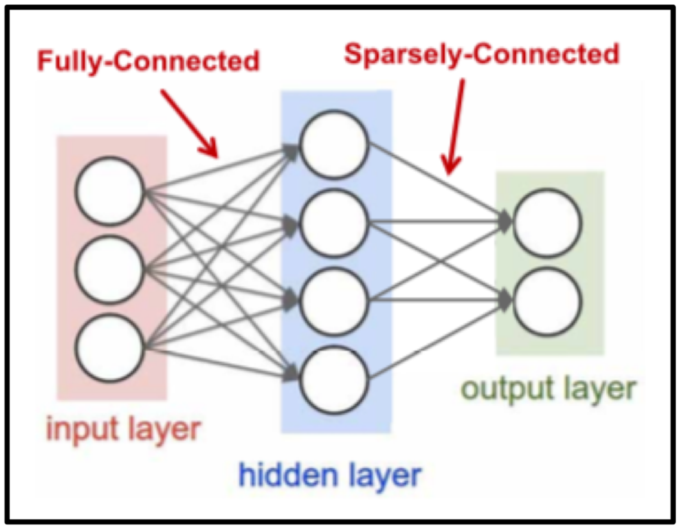


Figure 2.4: Fully connected versus sparsely connected architecture [Sze17].

Recurrent Neural Networks

Recurrent Neural Networks are designed to allow inputs to interact directly with the hidden node created from previous inputs [Agg18]. This is helpful in situations where data has a temporal component, or where data is received in sequences, and inputs are dependent on one another, such as text in a sentence or biological data. A loop is created on nodes in either the output or hidden layers to use previous output data in the next iteration. Figure 2.5(a) demonstrates an architecture that predicts the next word in a spoken phrase. The loop is easier to understand once it is unraveled, as shown in figure 2.5(b). This demonstrates how for every hidden node after the first iteration, the node will take its corresponding input as well as the output of the node from the previous iteration.

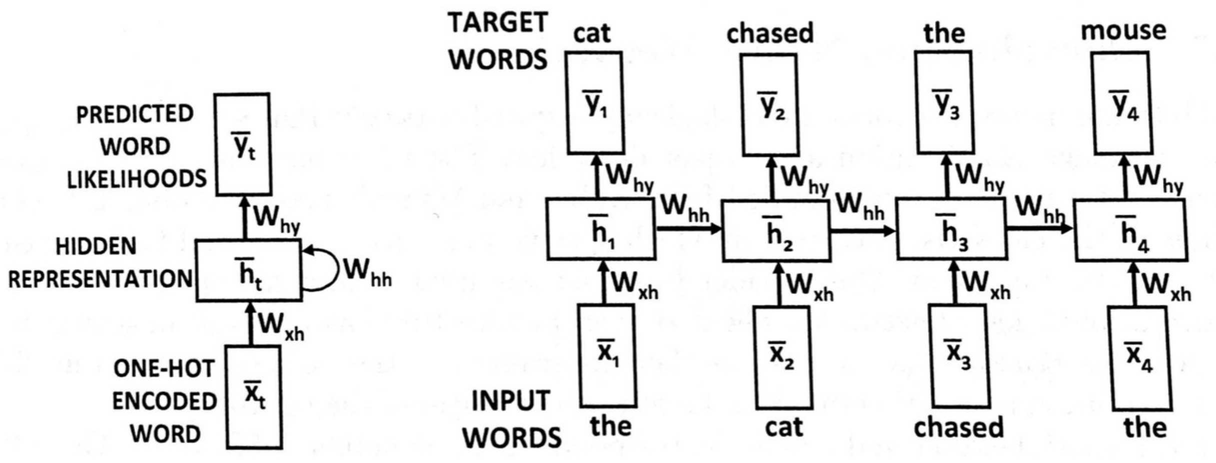


Figure 2.5: (a)Recurrent neural network and (b)time-layered representation. [Agg18]

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a family of multi-layer neural networks particularly designed for use on two-dimensional data, such as images and videos [Arel10]. More specifically CNN's are a form of DNN that have convolutional layers, and each convolutional layer is comprised of high-dimensional convolutions, shown in Figure 2.6. In the convolution, activation functions of a layer are structured as a 2-D input feature map called a channel [Sze17]. Each channel is convolved with filters that are composed of weights and the results are summed to create one channel of output feature map. The motivation behind using CNN's is that unlike traditional neural networks, CNN's have sparse connectivity that allows reduced memory requirements and better efficiency by storing fewer parameters and requiring fewer operations. The relationship between layers and spatial information allows them to work well with feature extraction and image processing. In fact, CNNs are commonly used in image processing, classification and speech recognition because of its feature maps ability to store essential information with each layer of the model.

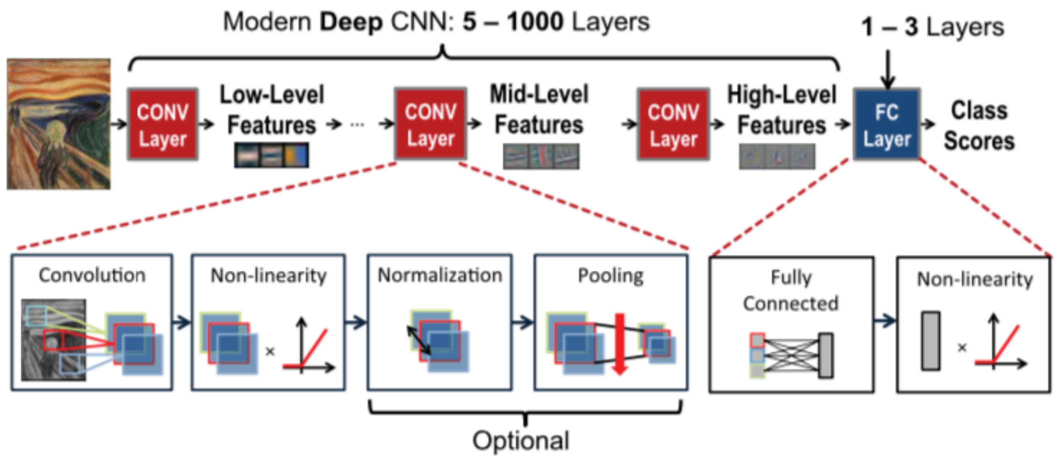


Figure 2.6: An example of a Convolutional neural network [Sze17].

Activation Functions

Activation functions used in hidden layers directly affect the possibilities for describing nonlinear systems using a feedforward neural network [Stur19]. In choosing an activation function, there are many options such as the sign function, reLU (Rectified Linear Unit), sigmoid, Gaussian, and other functions. Choice of activation function is dependent on the target variable to be predicted. The sign function is helpful in its ability to map binary outputs, reLU is useful in cases where values less than zero are made to be set to zero while values equal or greater to zero are expressed as a linear identity function [Stur19], and the sigmoid function is used to output values between 0 and 1 in situations where computations are to be interpreted as probabilities. Each computational neuron could have a different activation function however it is common practice that perceptrons in a layer would all have the same activation function [Agg18]. Using a particular activation function over others or choosing different activation functions in different layers depends mostly on its application.

Feed-Forward Algorithm

A standard approach to setting weights to the connections between nodes is to use small, random variables which will be later updated using a learning algorithm, such as backpropagation. Training a network using backpropagation is comprised of three parts, first of which is the forward processing, second is to calculate the backpropagation of error and the third is to update the weights accordingly. Forward processing is the process of moving through the layers of the network, from input layer to output layer, and using the weighted sum of each node in a layer as inputs to the neurons and the output firing depending on its activation function. The steps in forward processing are explicitly shown below, in equations 2.1 and 2.2. Equation 2.1 indicates the weighted sum of inputs is created, mathematically by ensuring the subscripts are indexed properly:

$$wsum_j = \sum_{i=0}^{N} w_{ji} x_i \tag{2.1}$$

where:

N	is the number of nodes in the previous layer,
w_{ji}	are the connections from the previous layer nodes, i's, to the current node, j, and
x_i	is the output from the previous layer.

Equation 2.2 demonstrates the use of the activation function in causing the neuron to "fire" and produce its output, y.

$$y_j = f(wsum_j) \tag{2.2}$$

where:

$wsum_j$	is the weighted sum into node j, and
f	is the selected activation function.

Once the neurons of a layer fire, the output will then be multiplied by the next weight connection as input to the next layer. This continues through each layer until there is a final output or outputs. The second stage, backpropagation of the errors, is calculating the errors of each neuron starting with the output layer and moving backwards as the output errors are needed to compute the inner layers errors. The errors for the output layer are determined by the values of the target output(s). Utilizing the gradient descent method [Faus94], the hidden layer neurons of the network use the errors of the previous layer to calculate its error, as they don't have associated targets. The ingenuity is attributed to the fact that the error for each hidden neuron is the sum of the output layer's error(s), subject to the degree to which they contributed to that error, as indicated by the weight on the connecting link. It should be noted that derivation of the equations results in utilization of the derivative of the activation function, evaluated at the current weighted sum. This, then results in a "delta," or δ, that can be used to modify the weights to achieve "learning." In equations 2.3 and 2.4, below, the error in the output layer, and δ for the output layer neurons is calculated based on the unipolar logistic (sigmoid) function:

$$\delta_j = ERROR_j \cdot f'(y_j) \tag{2.3}$$

where:

$ERROR_j$	is the RMS error between the actual output and the target output for node j, and
y_j	is the actual (computed) output for node j.

Taking the derivative of the unipolar logistic function, and algebraically manipulating the results into convenient form yields:

$$\delta_j = (t_j - y_j)(y_j)(1 - y_j) \tag{2.4}$$

where:

t_j	is the target output for node j, and
y_j	is the actual (computed) output for node j.

As for the hidden layer, the error is calculated by obtaining the sum of the all the errors in the previous layer k (working backwards) multiplied by the weights, w_{kj}, connecting it to the current node, j. That result is then multiplied by the derivative of the unipolar logistic function, evaluated at the weighted input:

$$\delta_j = (\sum_{all\ k} \delta_{out-k} * w_{kj})(z_j)(1 - z_j) \tag{2.5}$$

where:

δ_{out-k}	is the calculated "delta" for output-layer node k,
w_{kj}	is the weight connecting hidden-layer node *j* with the output layer node k, and
z_j	is the actual (computed) output for hidden-layer node j.

The last step in the process is to update the weights. After calculation of all the deltas for all the nodes, the change in weights connecting the neurons, often called the "delta weights" are calculated. This is done using the previously calculated error and delta of each neuron. The weights of the connections are updated by summing the original value of the weight with the product of the learning rate, the neuron's gradient (or delta), and the input of the node at that connection, as demonstrated in equations 2.6 and 2.7, below.

$$\Delta w_{ji} = \alpha * \delta_j * x_i \tag{2.6}$$

where:

Δw_{ji}	is the "delta weight," or change in the weight on the link connecting node *j* with input node i,

α	is the "learning rate,"
δ_j	is the gradient, or error associated with node j, and
x_i	is the input to this node from the node *i,* in the previous layer.

The data samples drive the updates, as the new weights for sample (*n+1*) are determined by the error calculated due to the processing of data sample n, as demonstrated in equation 2.7:

$$w(n+1) = w(n) + \Delta w(n) \tag{2.7}$$

This process is repeated through all of the data samples provided in the training dataset. Completion of this process on all training data is referred to as an "epoch." Typically, many iterations, or epochs, are necessary for training a backpropagation neural net until convergence [Faus94], the point at which the network has learned to generalize.

Early Stopping

Early stopping is a way of cutting off the gradient descent before convergence and is one of the most commonly used forms of regularization in deep learning [Good17]. The reason for using early stopping is to avoid overfitting during training which results in poor generalization, and poor accuracy of output for the test data. Typically, the test dataset is a distinct set of data samples that the network has never seen before. However, in cases where there is a single dataset, a portion of the data is selected for training, and the remainder of the data is withheld for validation after training. The error on the training data is monitored, and there will come a point where the error on the testing data is consistently rising instead of decreasing. This is a point where early stopping could be utilized, as it becomes evident that the network is diverging.

Another reason for early stopping would be when the error is decreasing minimally, as continuing would cause overfitting. This approach must be tested several times in order to choose the proper point to stop training as small increases in error wouldn't necessarily mean it would continue to rise and could just be a local maximum. Early stopping is used to improve performance and can be easily added to network training without significantly changing training algorithm [Agg18].

Discussion of Applications

Neural networks were first proposed in the 1940's but did not have its first practical application until the LeNet network for handwritten digit recognition in the 1980's used by ATMs for digit recognition on checks [Sze17]. Since then, applications for neural networks have expanded over several fields contributing to commercial success and expanding areas of research. A rise in popularity of Deep Learning can be attributed to three factors: the emergence of large-scale training data, quick development of high preforming parallel computing systems, and advances in network structures design and training strategies [Zhao19]. Advances in deep networks paved the way for a broad range of applications to apply machine learning, whereas previously, the computational capability wasn't sufficient. Discussion surrounding popular DNN applications are outlined in this chapter.

Computer Vision

Examples of DNNs used in everyday life are becoming more emergent. More than ever we see a push for self-driving cars, with the ability to get around while not once guiding the steering wheel, pressing gas pedal, or brakes. All operations in an autonomous vehicle rely heavily on sensors in the car and its ability to make a decision on what it is "seeing." This is an example of how DNNs are used to make complex decisions based on the information dependent on images. A primary field of computer vision is object detection, where within a bounded region, objects are labeled based on location and classification. Object detection can be divided into three stages: informative region selection, feature extraction, and classification.

Informative region selection is the selection of windows chosen for a particular object, as different objects will have varying sizes and positions. Although finding the appropriately sized window can be computationally expensive and, in using multi-scale sliding windows, if only a fixed number of sliding window templates are applied, unsatisfactory regions may be produced [Zhao19]. Feature Extraction is used to help recognize and differentiate objects; features are extracted to be used as defining attributes that set one object apart from the rest. Finally, classification is used to compare objects to a target and distinguish them from all other categories. Classification is used to predict what an object is by its features and other descriptive elements. For example, an object could be categorized to have a particular size, weight, and color, and, with the help of the trained network, could be identified.

Computer vision often employs the use of convolutional neural networks, because of their ability to create feature maps and handle complex computation needed for feature extraction and classification. Figure 3.1 illustrates how convolutional neural networks facilitate object classification.

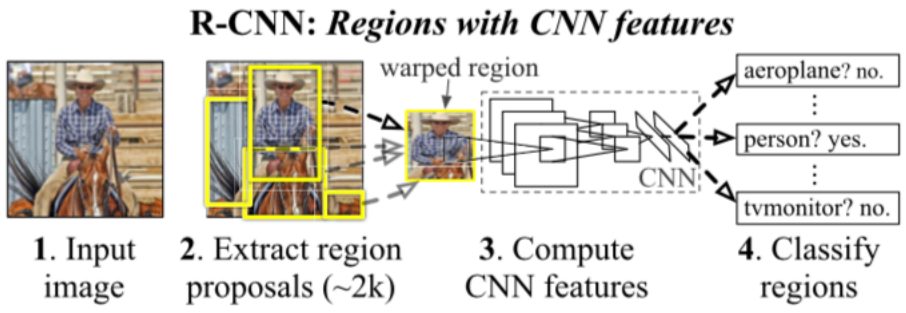


Figure 3.1: A region-based CNN 1) Input image, 2) region proposals, 3) computes features for each proposal using a CNN, and then 4) classifies each region [Zhao19].

CNNs are popular for their feature extraction abilities and are typically comprised of many layers. It must be noted, however, that these convolutional networks are not the only architecture capable of classification. A multilayer feed forward network is a simplistic architecture that is powerful enough to make correct classifications. In fact, many models can be simulated with a shallow network containing only one or two layers. Deep architectures are often created by simply stacking these simple architectures in creative ways [Agg18]. A multilayer feed forward network only needs additional layers to become a deep network, which can be easily accomplished with modifications to the architecture and training algorithm.

Language

Neural networks are useful for language with examples such as word prediction, noise reduction, and signal processing among other things. A prime example of this is emerging voice assistants such as Amazon Alexa or the Google Assistant, where these devices need to understand and associate words and their order for different sentences in order to predict the next word you're going to say. The use of neural networks in these speech recognition cases are to understand the spoken command and respond with an appropriate action with little delay.

A speech recognition system is used to understand the human voice to react accordingly, either with text or a command [Raj14]. These systems are necessary to understand a command regardless of accent or other speech characteristics. A major role in understanding and interpreting the audio is feature extraction, which can be achieved with the help of neural networks as layers can be used for different features. In a recent study to improve quality of a signal intended for speech enhancement, shallow networks used to predict the signal-to-noise (SNR) ratio could not fully learn the relationships between the noisy features and the target SNR [Xu15]. It was for this reason that a recurrent neural network was proposed in the cited study.

While several different architectures can be applied to language processing, recurrent neural networks are the most common architecture for speech and text data [Agg18]. Recurrent neural networks are generally used because of their ability to establish a relationship between inputs using “loops” in the architecture that provide “feedback,” and give a time-indexed order to the reception of inputs. It is crucial to establish this time-domain relationship to make accurate predictions for language or text because the order of words verbalized can make a big difference in the interpretation and meaning of the sentence or command.

Medical

One of the first examples of neural networks applications in medicine was developed in the 1980’s and was dubbed the “Instant Physician” [Faus94]. The net learned, through a large number of medical records, to associate symptoms and diagnoses. Thus, it could be used to determine a diagnosis and recommend a treatment for an input dataset consisting of a patient’s symptoms. Since then, the medical field has utilized machine learning (ML) in a variety of medical applications that provide consistent and efficient support for medical practitioners and personnel. For example, ML can be used to recognize and report patterns in patients’ medical history, detect illnesses, and even analyze data to create new medicines. DNNs have been used in medical imaging decision support systems to detect skin cancer, brain cancer, and breast cancer [Sze17].

Although the feed forward network may be one of the simplest architectures in the entire field of deep networks, it is capable of complex operations, such as detecting cancer through classification, as discussed in [Pei17]. However, in order to do the classification, it is widely understood that this type of application requires that the data be pre-processed, which is (at times) such an important aspect of the utility of the DNN. That is, when the appropriate data parameters are presented to a feed forward DNN, the convergence is quick, and the DNN’s accuracy very consistent and efficient. There are many aspects to “massaging” or pre-processing the raw data that have proven effective, ranging from the simplistic to the complex. There are many studies that refer to feature extraction, and utilization of statistical analyses of the data to create more efficient and powerful data, such as the calculation or extraction of Hjorth parameters [Vour00], Itakura Distance [Est05], and Detrended Fluctuation Analysis [Hard12]. However, recent applications focus on providing "raw" data directly to the system, and therefore, embedding the former pre-processing of data described above. This is extremely important, especially in medical imaging, where it could be useful to use convolutional neural networks because of their excellence in computer vision. Other cases in the medical field such as DNA sequencing might use a recurrent neural network as the order of the inputs is relevant data to consider when making predictions [Agg18].

As DNNs have been made popular for a variety of applications, the number of applications will surely grow. As we are seeing a growing demand for self-driving vehicles, voice assistants and deep learning in healthcare, these applications still require tuning in order to become more reliably and more widely used. With each addition to improve these applications, they will lead to newer strategies to train and newer additions to architecture that will certainly drive demand both deep learning and spark new applications in which to use DNNs.

Work and Results

While it is well documented that deep networks are used for training with large amounts of data and complex computation, it is difficult to gauge when a shallow network would be a better model than a deep net. As noted in the previous chapter, different applications may benefit from the use of different architectures for performance improvement. It can be difficult to determine the number of layers and nodes per layer to include in a model. Although the number of nodes in a hidden layer is tied to the complexity of the decision surface needed [Good17] and the number of layers to the complexity of the input space [Agg18], the current school of thought indicates that the ideal architecture for a problem must be found via trial and error guided by monitoring the error [Good17]. The first indicator for inclusion of a hidden layer is the data's characteristic of linear separability. If it is linearly separable, there is no need for a hidden layer at all as the output layer is enough to make accurate classification predictions. It must be stated, however, that it is rare to have a problem so simple. Even a problem as simple as the XOR problem requires at least one hidden layer to converge. In order to decide how deep a network should be, there needs to be balance between the complexity of the problem and complexity of the architecture. This chapter covers how changes in architecture affect the accuracy of different datasets and which architectures result in best performance, given the application.

Creation of a Shallow Network

Prior to initiating this study, the opportunity to build a shallow network that comprised of a single hidden layer was presented and accepted. This exercise allowed the author to familiarize herself with the basic feed forward architecture and learning algorithms typically used in neural networks. As the network simulation required the creation of original code, the XOR problem was selected as the dataset, in order to verify operation of the network. In this manner, it could be guaranteed that not only were computations through layers and back propagation were correct, but also that the network would successfully learn a linearly inseparable problem. (The XOR problem is elegantly simple, and, therefore, typically selected as the first dataset when writing original code for neural network simulation, since it is not linearly separable, yet has only four, clean and complete, data vectors. This results in easier debugging of the code, and fast verification of operation.) Once the network was coded, and could properly learn the XOR problem, a study was performed on how the number of neurons in the single hidden layer affected convergence. It was found that increasing the number of neurons in the hidden layer had a direct impact on training: it took less epochs to converge than the network with fewer neurons in the hidden layer. However, because the XOR problem is very simple, adding more neurons to a layer than necessary can overfit the problem. Performing the analysis on the one-layer network also revealed that changes in learning rate affected convergence as well: the more aggressive the learning rate, the more quickly the network would converge. However, it was also found that this, too, is at the cost of accuracy as too fast of a learning rate can cause the network to overshoot the target output.

Creation of a Deep Network

Creating a shallow network is not intuitive, there are many steps to forward processing and backpropagation, and each step is composed of complex calculations and matrix manipulation. Which is to say that understanding a set of equations and associated algorithm does not necessarily make the translation to a coding language very simple, as structures, data references, and arrangement and use of intermediate results is complex. Although the simple shallow network was difficult at first, with the possibility of working all small errors and calculations by hand, proved it fairly easy to trouble shoot, debug, and make necessary changes to the code. However, as the process began to generalize the code by converting the shallow network into a more flexible deep learning network (FDLN), the luxury of being able to do every calculation by hand was no longer an option. Besides the added computation complexity that accompanied the new depth, creating a flexible network architecture with variable number of inputs, outputs, and layers, was much more difficult than expected. It took several attempts at creating the correctly sized structures for the weights of connections between each layer, and once the proper structures were created, they needed care in traversing them to obtain correct values. Finally, when it could be guaranteed that computations were being executed correctly in each forward processing of a data sample, as well as correct values for backpropagation of error and updating weights, further additions were added to help convergence.

Additions to Basic Architecture

The architecture chosen for the Flexible Deep Learning Network (FDLN) in this study is a fully connected multi-layer perceptron model (MLP). The code is written such that the user can, at the start of the program, select the number of samples, inputs, outputs, layers and nodes per layer. This was envisioned to create a situation where further testing of the different architectures would be facilitated, without having to edit the base code. The base code, then, creates the selected architecture and connections that are initialized to small, random weights. The user also has the ability to choose the learning rate and momentum parameters utilized in the learning portion of training the network. The addition of momentum was used not only as a deterrent against overfitting but also to accelerate the learning and gradient descent. Its inclusion in the learning process is summarized by Equation 4.1.

$$\Delta w_{ji}(n+1) = \alpha * \delta_j(n) * x_i(n) + \mu \, \Delta w_{ji}(n-1) \tag{4.1}$$

where:

n	is the index, indicating the current iteration, as well as the current data sample,
$\Delta w_{ji}(k)$	is the "delta weight," or change in the weight on the link connecting node j with input node i, for interation k,
α	is the "learning rate,"
$\delta_j(n)$	is the gradient, or error associated with node j for the current data sample,
$x_i(n)$	is the input to this node from the node i, in the previous layer, and
μ	is the "momentum," or acceleration, that potentially speeds up convergence.

Momentum allows the net to make larger adjustments as long as they are in the same direction for several patterns, by using a combination of the current gradient and previous weight adjustment [Faus94].

Lastly, in order to combat overfitting, early stopping was applied to the training once the mean squared error of the network was at an acceptable value. Mean squared error (MSE) is shown in equation below where the output subtracted from the target of every sample is squared and summed, then divided by the number of samples to get the average.

$$MSE = (\Sigma\ (target - output)^2)/N \tag{4.2}$$

where:

target	is the target value, or expected output of the neuron for this particular data sample, as dictated by the training data set,
output	is the actual (computed) value of the output, and
N	is the number of data samples in the training dataset.

XOR Dataset

The XOR, or Exclusive-Or, problem is a classic problem in artificial neural networks (ANN) research. It is the problem of using a neural network to predict the outputs of XOR logic gate given two binary inputs. The XOR dataset is referred to as “clean and complete” due to the fact that it is linearly inseparable (complex), small (four data samples), has integers for inputs and expected values and has no “missing parameters.” It is widely used in ANN research because of these characteristics, and for the same reason, is often described as an “elegantly simple” dataset. It does have a storied past, as this data et was utilized by Minksy and Papert [Min69] to discourage initial interest in Perceptrons.

An XOR function should return a true value if the two inputs are not equal and a false value if they are equal. The XOR dataset consists of four samples for the different binary values to represent the TRUE-FALSE input combinations and their associated outputs, as shown in Table 4.1.

Table 4.1: XOR Function Table

x_1	x_2	**Target Output**
0	0	0
0	1	1
1	0	1
1	1	0

Results

The XOR dataset was used to explore the effect of deep-nets versus shallow-nets. The FDLN simulation code, as described in the previous sections of this chapter, was designed to allow the modification of the architecture in an efficient manner. Therefore, the model in this study was trained on the XOR dataset using a learning rate of 0.4, and momentum of 0.8, values chosen through trial and error, and two nodes in the hidden layer or layers. Therefore, the only differing factor, for this portion of the study, was in the number of hidden layers. Thus, this part of the study was intended to demonstrate the effect the number of layers imposes on the performance.

The network was trained with the XOR dataset until the mean squared error of the network was less than 1%. Presentation of the complete dataset is commonly called an "epoch" in ANN studies. Therefore, the performance parameters reported in Table 4.2 are the number of epochs to train properly (achieve an MSE of 1% or less), and "accuracy" (the percentage of accurate outputs).

Table 4.2: Performance of FDLN Model on XOR Data

Number of Hidden Layers	Epochs to Train	Accuracy
1	474	99.02%
2	1977	99.02%
3	11170	99.04%
4	26118	99.12%

It is clear that the number of layers presented to the XOR problem plays a major role in time it takes to train the model, because the XOR data is so simple, adding layers when it is not computationally necessary will only complicate the architecture and slow the learning. This larger architecture means there are more nodes, and between them, more connections and weights that need to be updated. This will ultimately take more computation cycles to update all the weights on the added connections. Accuracy, however, seems to be mostly unaffected by the number of layers in the network. This result is due to a variety of factors. First, the small amount of data that composes the XOR problem works in the favor of learning accurately. Second, the training and testing datasets are identical, which is usually discouraged because of the possibility of overfitting. Third, the data is "clean," such that the inputs never vary, nor are they vague, so that the decision surfaces are very clear. Lastly, the data is so small that once the model is trained, regardless of the number of layers, there should be little variance in accuracy results and while we may see small improvement, it may not be worth the added complexity.

Iris Dataset

The Iris dataset is a classic dataset created by R.A. Fisher for pattern recognition, including three different types of Irises and 50 samples of each. In this dataset, one type of Iris is linearly separable from the others while the other two are not linearly separable from each other, as demonstrated in Figure 4.1. This dataset is classic for use in ANN studies because it is not considered "big" in samples, and yet provides a distinction from the classic XOR problem: it has more than two classes, and the decision surface is not "clean," which is to say that two of the classes are interspersed in their attributes, and thus the decision surface is very complex. So, this is a problem that introduces a level of complexity not seen in the previous dataset.

The attributes of the Iris Dataset, listed in order to classify the type of iris, are sepal length, sepal width, petal length, and petal width. The samples in the Iris Dataset are shown in Figure 4.1, plotted to highlight the different types in the dataset.

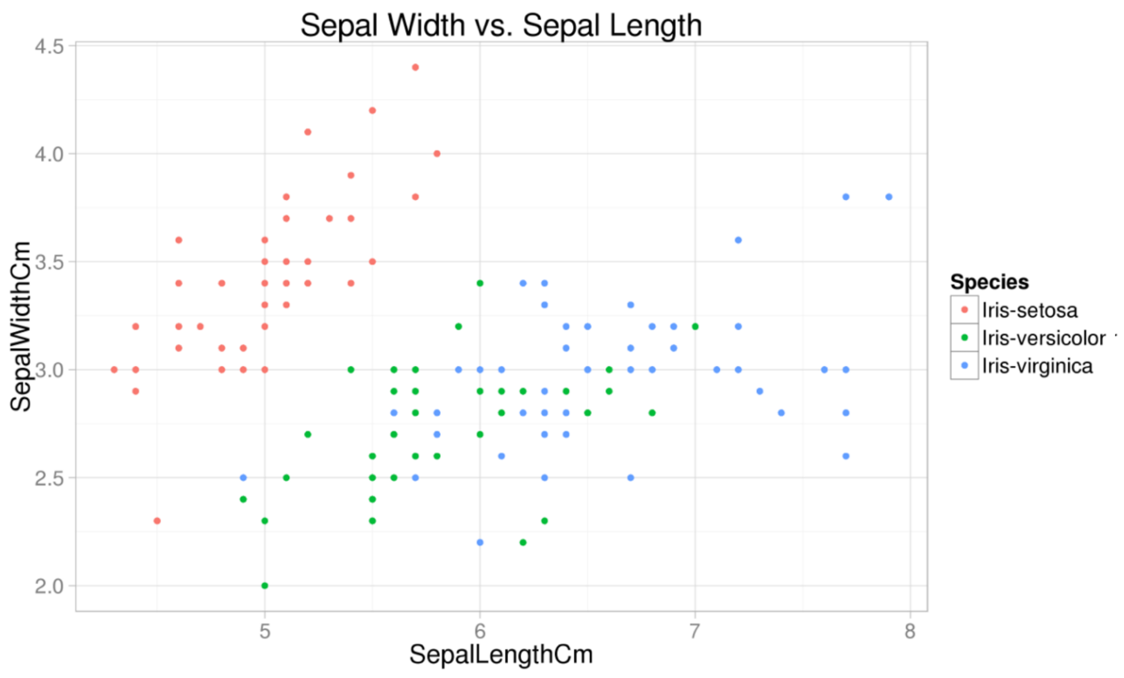


Figure 4.1 Plot of Parameters for Different Types of Irises

Illustrated above, Iris Versicolor and Iris Virginica are not linearly separable and will need at least one hidden layer to correctly classify them. The dataset was preprocessed, to intersperse examples of the different types, in order to avoid over fitting. Moreover, training and testing datasets were created out of the single dataset, where 80% of the data was used for training while the other 20% was used for testing.

Distributed vs Encoded Outputs

The XOR dataset had a single binary target output, where there was only one output node, which needed to represent the target of 0 or 1. However, the Iris dataset has three possible target outputs: Iris Setosa, Iris Versicolor, and Iris Virginica. These three types cannot be represented by a single node and so two options of output representation are proposed and tested to see which yields better results: Distributed and Encoded. In order to test which of the two provides the best results, the model will have all the same number of layers and parameters, and will be tested on the same data, the only difference being the number of output nodes and how they are interpreted.

Distributed

Distributed representation would yield three output nodes, one for each type of Iris. Once classified, the node representing the predicted iris type would output a 1 while the other two nodes output a 0, shown for clarity in Table 4.3 below.

Table 4.3: Distributed Representation of Iris Type

Iris Type	Target Output
Iris Setosa	001
Iris Versicolor	010
Iris Virginia	100

Once the representation of the expected outputs is established, the datasets must be edited to align with the selected representation. After the data was processed in this manner, the model was trained with 1 hidden layer and 4 nodes per hidden layer until an MSE of less than 5% was achieved. With a learning rate of 0.2 and momentum of 0.3, the network converged on average at epoch 70 and had an average accuracy of 93.34%.

Encoded

An encoded method of representation would only need two output nodes to represent the tree types of Iris. This method of output representation is demonstrated in Table 4.4. below.

Table 4.4: Encoded Representation of Iris Type

Iris Type	Target Output
Iris Setosa	00
Iris Versicolor	01
Iris Virginia	10

After processing the original dataset to ensure the encoded representation was utilized for all data samples, the model was trained using the Iris data and the same learning rate (0.2), momentum (0.3), and number of nodes per hidden layer (4) until an MSE of less than 5% was achieved. Using encoded outputs gave an average convergence at epoch 182 and an average system accuracy of 91.93%.

Results

Between the two types of output representation, distributed outputs yielded both faster convergence and better accuracy. For this reason, the model was only tested using distributed outputs for the remainder of the study. However, the difference between the size of the output layer is small when comparing, in cases where there are 16 targets, we may see that encoded would provide better results given we would only need 4 output nodes rather than 16. Data was then separated into two sets, training, which was comprised of 80% of the dataset, and testing which was the remaining 20%. To observe the iris dataset, a learning rate of 0.2 and momentum of 0.3 was used, these values were chosen due to a convergence issue that arose when using higher, more aggressive, values. The network constantly over-shot the target and would not converge when using higher values for the learning rate and momentum. Finally, the network was trained and tested with 4 nodes in each hidden layer trained until a mean squared error of less than 0.05 was achieved. Once the network was trained, it was then run using the testing set to observe performance. Performance values for different depths of Deep Learning architectures are shown in Table 4.5.

Table 4.5: Performance of FDLN Model on Iris Data

Number of Hidden Layers	Epochs to Train	Accuracy
1	66	93.15%
2	190	93.75%
3	198	93.82%
4	222	93.85%

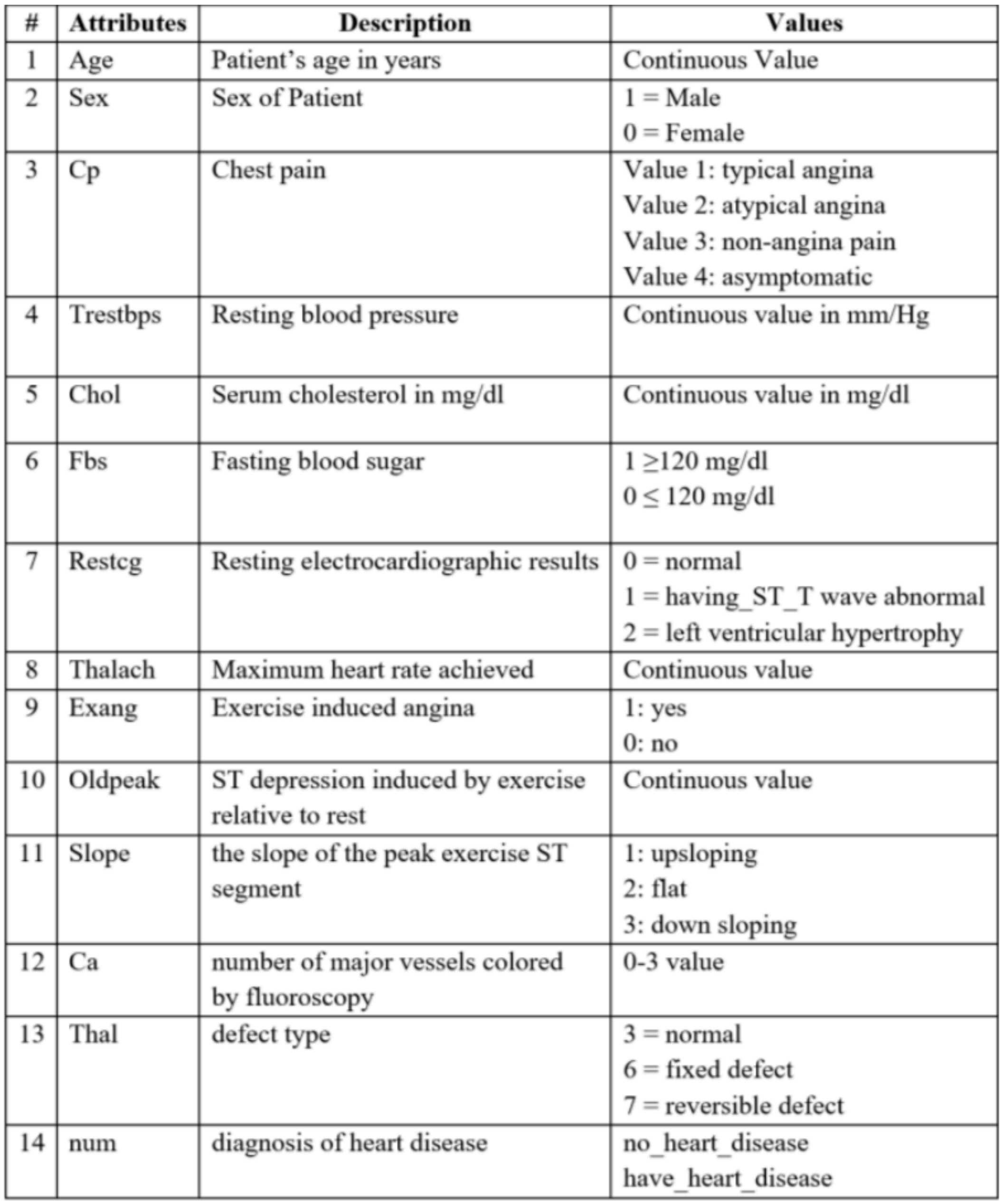
Much like the XOR dataset results, when using the iris dataset, we can see that although adding more hidden layers does improve accuracy of the classification, this improvement is so slight that it does not necessarily compensate for the time it takes to train.

Heart Disease Dataset

The Cleveland Heart Disease dataset is used to predict whether there is a presence of heart disease in a patient, based on other attributes, originally investigated by Robert Detrano [Aha88]. While there are datasets form Hungary, Switzerland, and VA Long Beach included in this database, the Cleveland dataset is that which is most used by machine learning (ML) researchers

[Aha88] and contains 297 samples. When applied to machine learning, the experiments commonly use just 14 of the 76 attributes, these attributes are detailed in Table 4.6, below.

Table 4.6: Description of Heart Disease Data Attributes [Olan18]



#	Attributes	Description	Values
1	Age	Patient's age in years	Continuous Value
2	Sex	Sex of Patient	1 = Male 0 = Female
3	Cp	Chest pain	Value 1: typical angina Value 2: atypical angina Value 3: non-angina pain Value 4: asymptomatic
4	Trestbps	Resting blood pressure	Continuous value in mm/Hg
5	Chol	Serum cholesterol in mg/dl	Continuous value in mg/dl
6	Fbs	Fasting blood sugar	1 ≥120 mg/dl 0 ≤ 120 mg/dl
7	Restcg	Resting electrocardiographic results	0 = normal 1 = having_ST_T wave abnormal 2 = left ventricular hypertrophy
8	Thalach	Maximum heart rate achieved	Continuous value
9	Exang	Exercise induced angina	1: yes 0: no
10	Oldpeak	ST depression induced by exercise relative to rest	Continuous value
11	Slope	the slope of the peak exercise ST segment	1: upsloping 2: flat 3: down sloping
12	Ca	number of major vessels colored by fluoroscopy	0-3 value
13	Thal	defect type	3 = normal 6 = fixed defect 7 = reversible defect
14	num	diagnosis of heart disease	no_heart_disease have_heart_disease

Dataset Pre-Processing

An initial trial, running the input values (as is) through the DL model would not converge. Upon careful examination of the data, it was found that some values of one parameter, being such large positive numbers (in comparison with values for the other parameters), that they were obscuring the meaning or importance of the other parameters. Stated explicitly, the gradient was being killed by the effect of this single parameter on the sigmoid function. The sigmoid function takes a real number and "squashes" it between 0 and 1, using large positive values will be nearly one and large negative values nearly 0. This makes the local gradient 0 and does not effectively update during backpropagation [Dhol18]. In order to ensure convergence, several values needed to be examined and normalized before training. The attributes that needed normalization were age (Age), resting blood pressure (Trestbps), cholesterol (Chol), and maximum heart rate achieved (Thalach). These parameters were then normalized by setting the maximum value of the type of attribute equal to 1 and scaling the rest, accordingly, as demonstrated in Equation 4.3 below. In addition to processing the input data, the output target is transcribed to a binary representation: 0 being an absence of heart disease, 1 meaning there is a presence of heart disease.

$$x_{norm}(k) = \frac{x(k)}{M} \tag{4.3}$$

where:

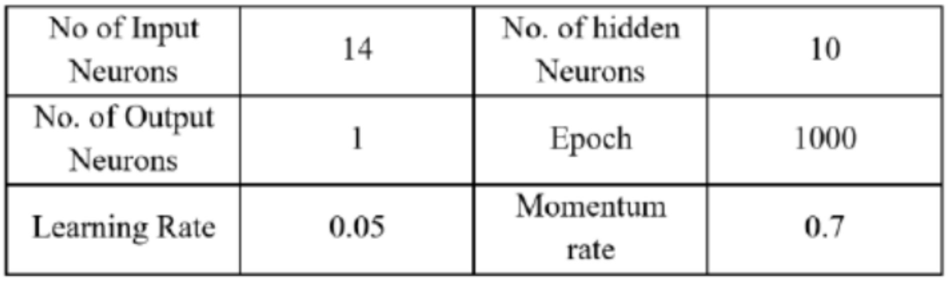
$x_{norm}(k)$	is the normalized parameter value of the kth data sample,
$x(k)$	is the original parameter value of the kth data sample, and
M	is the maximum value of the specific parameter, over the entire dataset.

Other Work

To better diagnose those who suffer from Cardiovascular Disease, Rashidah Olanrewaju, et al. proposed using a feed-forward multi-layer perceptron as a method of predicting cardiovascular disease, as only human intelligence is not enough to accurately predict cardiovascular disease

[Olan18]. In the study, the dataset was separated, 70% to train and 30% to test. Performance was measured by sensitivity, specificity, positive predicted value, negative predicted value, and accuracy. The list of parameters used to train the model are given in Table 4.7.

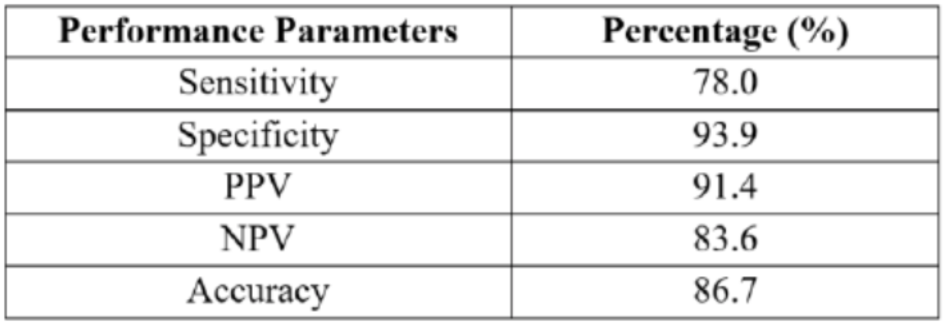
Table 4.7: Description of Shallow MLP Model [Olan18]



No of Input Neurons	14	No. of hidden Neurons	10
No. of Output Neurons	1	Epoch	1000
Learning Rate	0.05	Momentum rate	0.7

Although the number of epochs was set to 1000, the network was optimized at 250 epochs and so the training algorithm was terminated. Once the network was trained, it was then trained using the test set and obtained an accuracy of 86.7%. Thus, Olanrewaju concluded that neural networks could be applicable in determining the presence or absence of cardiovascular disease. The other performance parameters are given in Table 4.8.

Table 4.8: Performance of Shallow MLP Model [Olan18]



Performance Parameters	Percentage (%)
Sensitivity	78.0
Specificity	93.9
PPV	91.4
NPV	83.6
Accuracy	86.7

Results

In order to observe the how the FDLN network designed in this study performs on the Cleveland Heart Disease dataset, a comparison between Olanrewaju's network and the FDLN is necessary. While performance may have been measured including other parameters, in order to keep consistency with other datasets used for this study, this study is only concerned with measuring accuracy of classification, or prediction of heart disease. Once data was again split, 80% for training and 20% for testing, a single hidden layer was used and all the same parameters to train the FDLN, results showed an accuracy of 85.77%, while converging in 255 epochs, just 1% shy of Olanrewaju's network's accuracy and only taking five more epochs before reaching an MSE of less than 0.05. Making this comparison is useful for establishing a basis for how to initialize the FDLN training parameters and understand how the network will behave with just one layer. However, analysis of the heart disease dataset with a deep network is still in order.

The heart disease dataset was tested using a learning rate of 0.1 and momentum of 0.7 and used 10 nodes in each hidden layer. The values for the learning rate, momentum and node in each layer are based off of Olanrewaju's network, however, uses a slightly more aggressive learning rate to speed convergence. The dataset was separated, taking 80% of data samples for training, and the remaining 20% of data samples for testing. The FDLN was trained until a mean squared error of less than 0.05 was achieved. Experimental performance parameters are shown in Table 4.9.

Table 4.9: Performance of FDLN on Cleveland Heart Disease Data

Number of Hidden Layers	**Epochs to Train**	**Accuracy of Classification**
1	161	85.23%
2	170	77.68%
3	174	77.12%
4	224	81.79%

While the number of epochs to train becomes greater with the additional layers, much like the XOR and Iris datasets results, this is to be expected because with the addition of layers there are more connections and more weights that need to be updated. However, the heart disease results differ from the other datasets where, with the addition of layers, accuracy grew. In this case with the addition of layers there is a loss in accuracy.

Using varying datasets to observe the changes in performance caused by the depth of the network shows that it is not necessary to add layers to improve accuracy because there will be a point in which the added depth only causes training to be more difficult and take longer to reach convergence. This is apparent with the XOR datasets results where adding layers over-complicated a very simple problem and needed upwards of 10,000 epochs for minimal increase in accuracy. In the case of the heart disease dataset, adding additional layers did not improve accuracy at all, and thus there is no reason to complicate the architecture by adding more depth. This is to be expected as a feed-forward network with three layers of neurons, the input, hidden, and output layer, can represent any continuous function exactly [Faus94]. That is not to say that deep networks do not provide substantial improvement on performance but can explain why we see little, if any, improvement in these examples. The decision to add more layers to a network depends on the application and will have differing results. For instance, it would be appropriate to add hidden layers when doing analysis on a dataset with a very large number of parameters as layers will serve as feature extractors to the system. As for the datasets in this study, it was not imperative to add layers as a single layer would suffice to achieve a good accuracy.

Conclusion

Although deep neural networks have the ability to have great accuracy with problems that are very complex and handle large amounts of information, sometimes sticking with shallow networks can have greater benefits. Several parameters can affect a networks performance, such as making changes to the architecture and algorithm, but as for the datasets presented in this study using just one hidden layer yielded the best results in terms of accuracy and convergence. The XOR and Iris dataset showed that by adding layers to the architecture, better accuracy was achieved but only by a small amount. Given the number of epochs necessary for convergence on those datasets, its arguably better to compromise the additional accuracy for a simpler architecture. In regard to the Heart disease dataset, additional layers caused the network to take longer to converge and received worse accuracy than using just a single hidden layer.

There is no perfect method for determining the number of layers to be used in a neural network, but the complexity of the problem gives insight as to an estimate, and in the context, the user can make an expert's assessment about the number of layers with which to start. Additionally, it's important to keep in mind to not over-complicate a problem as seen in the XOR analysis. The architecture of a network for a given application should reflect the difficulty of the problem.

Future Work

Areas of exploration for future work in relation to this study would be, first to broaden the scope of this study by observing the impact depth has on accuracy when using much larger datasets. Secondly, would be to create an MLP network with cascading size of the hidden layers to observe if additional, but smaller, layers would yield better results. The flexible deep learning network (FDLN) only prompted users for the number of nodes in a hidden layer that was then applied for all hidden layers. The idea is that adding more layers for improved accuracy while keeping the number of weights to update low could lead to better performance. Third, the feed forward network presented is capable of deep learning, but it would be interesting to see how other architectures would measure up. Performing this same analysis across architectures such as convolutional neural networks and recurrent neural networks would provide narrative around which architectures benefit most from the addition of layers. Finally, comparing the differences in performance given through an original program and libraries available for implementing deep architectures would could demonstrate which is a better choice for future researchers before they begin the laborious process of writing machine learning code.

References

[Agg18] Aggarwal, Charu C. *Neural Networks and Deep Learning: a Textbook*. Springer, 2018.

[Aha88] David W. Aha (1988) UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[Arel10] I. Arel, D. C. Rose and T. P. Karnowski, "Deep Machine Learning - A New Frontier in Artificial Intelligence Research [Research Frontier]," in *IEEE Computational Intelligence Magazine*, vol. 5, no. 4, pp. 13-18, Nov. 2010, doi: 10.1109/MCI.2010.938364.

[Day90] Dayhoff, J. E. (1990). Neural Network Architectures: An Introduction. New York, NY: Van Nostrand Reinhold.

[Dhol18] Dholakia, Dhaval. "Activation Functions." Medium, Towards Data Science, 8 May 2018, towardsdatascience.com/activation-functions-b63185778794.

[Est05] E. Estrada*, H. Nazeran, P. Nava, K Behbehani, J. Burk, and E. Lucas, "Itakura Distance: A Useful Similarity Measure between EEG and EOG Signals in Computer- aided Classification of Sleep Stages," *Proceedings of the 2005 27th Annual International Conference of the Engineering in Medicine and Biology Society, IEEE-EMBS 2005*, pp. 1189-1192, 2005.

[Faus94] Fausett, Laurene V. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice Hall, 1994.

[Fish36] Fisher, R.A. "The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936);

[Good17] Goodfellow, Ian, et al. *Deep Learning*. The MIT Press, 2017.

[Hard12] Hardstone, Richard, et al. "Detrended Fluctuation Analysis: A Scale-Free View on Neuronal Oscillations." Frontiers in Physiology, vol. 3, 2012, doi:10.3389/fphys.2012.00450.

[Min69] Minsky, M., & Papert, S. *Perceptrons*. The M.I.T. Press, 1969.

[Olan18] Olanrewaju, Rashidah & Musa, Nasra & Mohd Paudzi, Nurul Fatihah Aina. (2018). Early Prediction of Cardiovascular Disease Level Based using Artificial Neural Networks.

[Pei17] S. Pei, L. Tong, X. Li, J. Jiang and J. Huang, "Feed-forward network for cancer detection," 2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), Guilin, 2017, pp. 697-701, doi: 10.1109/FSKD.2017.8393356.

[Raj14] N. Rajput and S. K. Verma, "Back propagation feed forward neural network approach for Speech Recognition," Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization, Noida, 2014, pp. 1-6, doi: 10.1109/ICRITO.2014.7014712.

[Stur19] D. Stursa and P. Dolezel, "Comparison of ReLU and linear saturated activation functions in neural network for universal approximation," *2019 22nd International Conference on Process Control (PC19)*, Strbske Pleso, Slovakia, 2019, pp. 146-151.

[Sze17] V. Sze, Y. Chen, T. Yang and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE, vol. 105, no. 12, pp. 2295-2329, Dec. 2017.

[Vour00] M. Vourkas, S. Micheloyannis and G. Papadourakis, "Use of ANN and Hjorth parameters in mental-task discrimination," 2000 First International Conference Advances in Medical Signal and Information Processing (IEE Conf. Publ. No. 476), Bristol, UK, 2000, pp. 327-332, doi: 10.1049/cp:20000356.

[Xu15] Y. Xu, J. Du, L. Dai and C. Lee, "A Regression Approach to Speech Enhancement Based on Deep Neural Networks," in IEEE/ACM Transactions on Audio, Speech, and Language Processing, vol. 23, no. 1, pp. 7-19, Jan. 2015.

[Zhao19] Z. Zhao, P. Zheng, S. Xu and X. Wu, "Object Detection With Deep Learning: A Review," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 11, pp. 3212-3232, Nov. 2019.

Appendix

```
/*
* Author: Kirsten Byers
* Date Created: March, 2020
* Description: A flexible, deep neural network created
* to learn and classify a variety of datasets.
*/
import java.util.Scanner;
import java.io.*;
public class thesis_code_old
{
  public static void main(String [] args)throws FileNotFoundException
  {
    /*******Prompt User for Smaples and Outputs*******/
    System.out.println("How many samples?");
    Scanner sc = new Scanner(System.in);
    int numLines = Integer.parseInt(sc.next());
    System.out.println();

    System.out.println("How many Outputs?");
    int numOutputs = Integer.parseInt(sc.next());
    System.out.println();

    /**Store Input File in 2D array**/
    String fileName = "heart_train.csv";
    Scanner scan = new Scanner(new File(fileName));
    double [][] arrayInputs = new double [numLines][];
    double [][] arrayTargets = new double [numLines][numOutputs];

    int r = 0;
    while(scan.hasNextLine()){
    String line = scan.nextLine();
    String [] stringArray = line.split(",");
    arrayInputs[r] = new double [(stringArray.length - numOutputs)+1];
      for(int c = 0; c < arrayInputs[r].length; c++){
        if(c==0)
          arrayInputs[r][c] = 1;
        else
          arrayInputs[r][c] = Double.parseDouble(stringArray[c-1]);
      }
      int outputStart = arrayInputs[r].length-1;
      for(int c = 0; c < arrayTargets[r].length; c++){
        arrayTargets[r][c] = Double.parseDouble(stringArray[outputStart]);
        outputStart++;
      }
      r++;
    }

    /*Print training inputs*/
    System.out.println("Inputs: ");
    for(int i=0; i<arrayInputs.length; i++){
      for(int j=0; j< arrayInputs[i].length; j++){
        System.out.print(arrayInputs[i][j]+" ");
      }
      System.out.println();
    }
    System.out.println();

    /*Print training targets*/
     System.out.println("Targets: ");
    for(int i=0; i< arrayTargets.length; i ++){
      for(int j=0; j< arrayTargets[i].length; j++){
        System.out.print(arrayTargets[i][j]+" ");
      }
      System.out.println();
            }
    System.out.println();

    /*Prompt for hidden layers and learning rate*/
```

```
        System.out.println("How many hidden layers?");
        Scanner s = new Scanner(System.in);
        int hid = Integer.parseInt(s.next());
        System.out.println();

        System.out.println("How many neurons in the hidden layers?");
        int neuron = Integer.parseInt(s.next());
        System.out.println();

        System.out.println("Learning Rate?");
        double rate = s.nextDouble();
        System.out.println();

        System.out.println("Momentum?");
        double momentum = s.nextDouble();
        System.out.println();

        /*create input layer weights*/
        double [][] inputWeights = createWeights(neuron, arrayInputs[0].length);

        /*create hidden layer weights only if there is more than one hidden layer*/
        double [][][] hiddenLayerWeights = new double[hid - 1][neuron][neuron+1];
        if(hid>1){
           for(int i = 0; i < hiddenLayerWeights.length; i++){
              hiddenLayerWeights[i] = createWeights(neuron, neuron+1);
           }
        }

        /*create output weights*/
        double [][] outputWeights = createWeights(numOutputs, neuron+1);

        double [][] prevDeltaInputWeights = new double [neuron][arrayInputs[0].length];
        double [][][] prevDeltaHiddenWeights = new double [hid -1 ][neuron][neuron+1];
        double [][] prevDeltaOutputWeights = new double [outputWeights.length][outputWeights[0].length];

        double [][] inputToNeurons = new double[hid][neuron+1];
        double [][] finalOutput = new double [numLines][numOutputs];
        double systemError=0;

        /*Train for 10000 epoch or until MSE is appropriate*/
        for(int i=0; i< 10000; i++)
        {
           systemError = 0;
           System.out.println("EPOCH: "+i);
           for(int j = 0; j<numLines; j++)
           {
              finalOutput [j] = forwardProcess(arrayInputs[j], inputWeights, hiddenLayerWeights, outputWeights, neuron, hid, numOutputs, input-
ToNeurons);
              updateWeights(finalOutput[j], arrayTargets[j], outputWeights, inputToNeurons, hiddenLayerWeights, inputWeights, rate, momentum,
prevDeltaInputWeights, prevDeltaHiddenWeights, prevDeltaOutputWeights, arrayInputs[j], neuron);
           }
           for(int j=0; j<finalOutput.length; j++){
              for(int k=0; k< finalOutput[j].length; k++){
                 System.out.print(finalOutput[j][k]+" ");
                 systemError += ((arrayTargets[j][k] - finalOutput[j][k]) * (arrayTargets[j][k] - finalOutput[j][k]));
              }
              System.out.println();
           }
           if((systemError/numLines) <= 0.05)
              break;
        }
        System.out.println("Learning Complete");

        /*Prompt user for testing parameters*/
        System.out.println("How many samples to test?");
        sc = new Scanner(System.in);
        int numLinesTest = Integer.parseInt(sc.next());
        System.out.println();

        String fileNameTest = "heart_test.csv";
        Scanner scanner = new Scanner(new File(fileNameTest));
        double [][] arrayInputsTest = new double[numLinesTest][];
        double [][] arrayTargetsTest = new double [numLinesTest][numOutputs];

        r = 0;
        while(scanner.hasNextLine()){
            String line = scanner.nextLine();
            String [] stringArray = line.split(",");
            arrayInputsTest[r] = new double [(stringArray.length - numOutputs)+1];
            for(int c = 0; c < arrayInputsTest[r].length; c++){
                if(c==0)
                    arrayInputsTest[r][c] = 1;
                else
                    arrayInputsTest[r][c] = Double.parseDouble(stringArray[c -1]);
            }
            int outputStartTest = arrayInputsTest[r].length-1;
            for(int c = 0; c < arrayTargetsTest[r].length; c++){
                arrayTargetsTest[r][c] = Double.parseDouble(stringArray[outputStartTest]);
                outputStartTest++;
            }
            r++;
        }

        /*Print inputs*/
        System.out.println("Inputs: ");
        for(int i=0; i<arrayInputsTest.length; i++){
            for(int j=1; j< arrayInputsTest[i].length; j++){
                System.out.print(arrayInputsTest[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println();

        /*Print test targets*/
         System.out.println("Targets: ");
        for(int i=0; i< arrayTargetsTest.length; i ++){
            for(int j=0; j< arrayTargetsTest[i].length; j++){
                System.out.print(arrayTargetsTest[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println();

        double [][] testOutputs = new double [numLinesTest][numOutputs];
        for(int i=0; i<numLinesTest; i++){
             testOutputs[i] = forwardProcess(arrayInputsTest[i], inputWeights, hiddenLayerWeights, outputWeights, neuron, hid, numOutputs, input-
ToNeurons);
        }

        //print test outputs
        for(int i=0; i<testOutputs.length; i++){
            for(int j=0; j< testOutputs[i].length; j++){
                System.out.print(testOutputs[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println();
        systemError = 0;

       /*Train on the test set and report accuracy*/
        for(int i=0; i<testOutputs.length; i++){
            for(int j=0; j< testOutputs[i].length; j++){
                systemError += ((arrayTargetsTest[i][j] - testOutputs[i][j]) * (arrayTargetsTest[i][j] - testOutputs[i][j]));
            }
        }
        System.out.println("System accuracy is: "+(1-(systemError/numLinesTest))+"% accurate.");
    }

    /*Creates and returns a 2D array with small random weight values*/
    static double [][] createWeights(int m, int n){
        double [][] weights = new double[m][n];
        for(int i=0; i<weights.length; i++){
            for(int j=0; j<weights[i].length; j++){
                weights[i][j] = Math.random()*2-1;
            }
        }
        return weights;
    }

    /*Forward propagation through the network*/
    static double [] forwardProcess(double [] arrayInputs, double [][] inputWeights, double [][][] hiddenLayerWeights, double [][] outputWeights,
int neuron, int hid, int numOutputs, double [][] inputToNeurons){
        double [] finalOutput = new double [numOutputs];
        double [] outputFromNeurons = new double [neuron];

        for(int i=0; i < outputFromNeurons.length; i++){
            outputFromNeurons[i] = sigmoidFunc(arrayInputs, inputWeights[i]);
        }
        /*new array to add bias to the first layer*/
        inputToNeurons[0][0] = 1;
        for(int i=1; i < inputToNeurons[0].length; i++){
            inputToNeurons[0][i] = outputFromNeurons[i-1];
        }
        /*forward processing through hidden layers*/
        if(hid > 1){
            int layer = 1;
            for(int i = 0; i< hiddenLayerWeights.length; i++){
                for(int j = 0; j < outputFromNeurons.length; j++){
                    outputFromNeurons[j]= sigmoidFunc(inputToNeurons[layer-1], hiddenLayerWeights[i][j]);
                }
                inputToNeurons[layer][0] = 1;
                for(int j=1; j < inputToNeurons[0].length; j++){
                    inputToNeurons[layer][j] = outputFromNeurons[j-1];
                }
                layer++;
            }
        }
        /*processing the final layer, resulting in final output*/
        for(int j=0; j< finalOutput.length; j++){
            finalOutput[j] = sigmoidFunc(inputToNeurons[hid -1], outputWeights[j]);
        }
        return finalOutput;
    }

    /*Calculates sigmoid function output for each neuron*/
    static double sigmoidFunc(double [] inputs, double [] weights){
        double sum = 0;
        double output = 0;
        for(int i=0; i < inputs.length ; i++){
            sum += (inputs[i]*weights[i]);
        }
        output = 1/(1+Math.exp(-sum));
        return output;
    }

    /*Calculate errors and update all weights*/
    static void updateWeights(double [] finalOutput, double [] targets, double [][] outputWeights, double[][] inputToNeurons, double [][][] hidden-
Weights, double [][] inputWeights, double rate, double momentum, double [][] prevDeltaInputWeights, double [][][] prevDeltaHiddenWeights,
double [][] prevDeltaOutputWeights, double [] arrayInputs, int neuron){
        double [] outError = new double [finalOutput.length];
        for(int i =0; i<finalOutput.length; i++){
            outError[i] = ((targets[i] - finalOutput[i])*(finalOutput[i])*(1 - finalOutput[i]));
        }

        /*Getting hidden errors*/
        double hiddenError [][] = new double[hiddenWeights.length][neuron];
        if(hiddenWeights.length > 0){
            hiddenError[hiddenError.length-1] = calcError(outError, outputWeights, inputToNeurons[inputToNeurons.length-1]);

            for(int i = hiddenError.length-2; i > -1; i--){
```

```
            hiddenError[i] = calcError(hiddenError[i+1], hiddenWeights[i+1], inputToNeurons[i+1]);
         }
      }
      /*Getting input errors (error for layer immediatly after input layer)*/
      double inputError [] = new double[hiddenWeights.length];
      if(hiddenWeights.length>0)
         inputError = calcError(hiddenError[0], hiddenWeights[0], inputToNeurons[0]);
      else
         inputError = calcError(outError, outputWeights, inputToNeurons[0]);

      /*Saveing delta weights and updating weights*/
      double [][] deltaInputWeights = new double [inputWeights.length][inputWeights[0].length];
      for(int i=0; i< deltaInputWeights.length; i++){
         for(int j=0; j< deltaInputWeights[i].length; j++){
            deltaInputWeights[i][j] = ((rate * inputError[i] * arrayInputs[j]) + (momentum * prevDeltaInputWeights[i][j]));
            inputWeights[i][j] = inputWeights[i][j] + deltaInputWeights[i][j];
            prevDeltaInputWeights[i][j] = deltaInputWeights[i][j];
         }
      }
      double [][][] deltaHiddenWeights = new double [hiddenWeights.length][neuron][neuron+1];
      if(hiddenWeights.length > 0){
         for(int i=0; i< deltaHiddenWeights.length; i++){
            for(int j=0; j< deltaHiddenWeights[i].length; j++){
               for(int k=0; k< deltaHiddenWeights[i][j].length; k++){
                  deltaHiddenWeights[i][j][k] = ((rate * hiddenError[i][j] * inputToNeurons[i][k]) + (momentum * prevDeltaHidden-
Weights[i][j][k]));
                  hiddenWeights[i][j][k] = hiddenWeights[i][j][k] + deltaHiddenWeights[i][j][k];
                  prevDeltaHiddenWeights[i][j][k] = deltaHiddenWeights[i][j][k];
               }
            }
         }
      }
      double [][] deltaOutputWeights = new double [outputWeights.length][outputWeights[0].length];
      for(int i =0; i< deltaOutputWeights.length; i++){
         for(int j=0; j< deltaOutputWeights[i].length; j++){
            deltaOutputWeights[i][j] = ((rate * outError[i] * inputToNeurons[inputToNeurons.length-1][j]) + (momentum * prevDeltaOutput-
Weights[i][j]));
            outputWeights[i][j] = outputWeights[i][j] + deltaOutputWeights[i][j];
            prevDeltaOutputWeights[i][j] = deltaOutputWeights[i][j];
         }
      }
   }

   static double [] calcError(double [] error, double [][] weights, double [] inputToNeurons){
      int connection = 1;
      double [] layerError = new double [weights[0].length-1];
      double [] sum = new double [weights[0].length-1];
      for(int i=0; i<sum.length; i++){
         for(int j=0; j< weights.length; j++){
            sum [i] += (error[j]*weights[j][connection]);
         }
         layerError[i] = sum[i]*inputToNeurons[connection]*(1-inputToNeurons[connection]);
         connection++;
      }
      return layerError;
   }
}
```

Vita

Kirsten Nicole Byers was born in El Paso, Texas on August 30, 1996. She graduated from Coronado High School in El Paso, Texas in June of 2014 and continued on to The University of Texas at El Paso in the Fall of that same year. While at UTEP Ms. Byers pursued a bachelor's degree in Electrical Engineering she was also engaged in organizations such as the Institute of Electrical and Electronics Engineers (IEEE) and held the officer position of Engineering Student Leadership Council (ESLC) representative for the Association for Computing Machinery (ACM).

Following her bachelor's degree, Ms. Byers continued to further her education by pursuing a Masters' of Science in Computer Engineering at UTEP. Her interest in machine learning lead her to do research with Dr. Patricia Nava, whom would later supervise Ms. Byers work as a Teaching Assistant for two courses: Topics in Soft Computing, and Advance Digital Design. While in these positions Ms. Byers would not only hold lab sessions, but also perform functionalities such as giving exams and grading assignments. During her graduate career Ms. Byers also served as a Lower Division Engineering Advisor, where she would regularly advise students not only on classes, but how to navigate their academic careers. Ms. Byers also had the opportunity of interning with Lockheed Martin Aeronautics in Fort Worth, Texas where she wrote airborne software for the T-50 trainer program and updated a familiarization training to be used in teaching new hires and interns. In addition to Ms. Byers academic and industry experience, she was awarded Spring 2020 Graduate Student Marshal of Students for the College of Engineering at The University of Texas at El Paso.