Neural Network Model of Heart Disease Prediction

Joseph O'Connor, M.S. Chemistry jpoconnor1961@gmail.com

December 21, 2023

Abstract

This report describes investigation and development of fully connected feedforward neural network (FNN) models to classify patients by their heart health related data into heart disease or healthy heart outcome groups. Comparison of the FNN models to a logistic regression model was also included as a benchmark reference point. Performance of the models were evaluated in terms of the general type of model (GLM or FNN), the type of gradient descent algorithm (backprop, rprop–, rprop+), different seeds & initialized start weights, and threshold levels for convergence. In addition, simplified loss landscape type plots were evaluated to compare models of interest. In head-to-head testing, the RPROP+ adaptive gradient descent algorithm converged more than 80 times faster than a standard backpropagation gradient descent algorithm, with 100% Test Accuracy.

Several successful FNN models were developed that scored 100% on the test dataset. From these, an overall best model was determined, with inferred evidence for better generalization, to be a FNN with 217 parameters across three hidden layers with 10 nodes, 5 nodes, and 3 nodes respectively. More importantly, the model used the RPROP+ adaptive gradient descent algorithm and a set of start weights that conformed to the LeCun et al. (1998) guidelines on weight initialization.[1] The best model (10,5,3 FNN nnTrained.2c) confirmed that not only is proper conditioning of the input data important, but so is proper conditioning of the start weights.

Keywords: Feedforward Neural Network (FNN), Resilient Backpropagation (RPROP), neuralnet
Repository: https://github.com/jpoconnor1961/HeartDiseaseFNNmodel

1 Introduction

The perceptron is the earliest model of an artificial neuron that also has the ability to learn, and was proposed by Frank Rosenblatt in 1958.[2] Rosenblatt's perceptron was inspired by the connection of the retina and neurons in the visual cortex of the brain. Rosenblatt proposed that the perceptron models the encoding of visual stimuli by the retina and transmission of that information to the visual cortex, where the input signals are interpreted and classified. Furthermore, he proposed a mathematical model by which a perceptron can learn from experience, just like the brain, by adjusting the weights of the connections based on feedback from the environment (see Figure 1).

In Rosenblatt's model, the Sensory Points/Projection Area are analogous to cells in the retina, which receive input patterns of light that are transformed and encoded into neural signals. Next, the input signals are reorganized through Association units that restrict the receptive field of inputs to each Response unit [Haykin 2009, pp. 29-30]. The Response Units, which are analogous to neurons in the visual cortex, compute weighted sums of the input signals and apply threshold/activation functions, such as a step function, to produce binary outputs. In its simplest reductionistic form, however, the *basic unit* of the perceptron is a set of inputs that

[1] LeCun, Yann A., Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. (1998). "Efficient BackProp" In: Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller (Editors). *Neural Networks: Tricks of the Trade, 2nd Edition,* Lecture Notes in Computer Science 7700, pp. 9–48, Springer-Verlag Berlin Heidelberg ©2012.

[2] Frank Rosenblatt (1958) "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain" *Psychological Review,* Vol. 65, No. 6, pp. 386-408

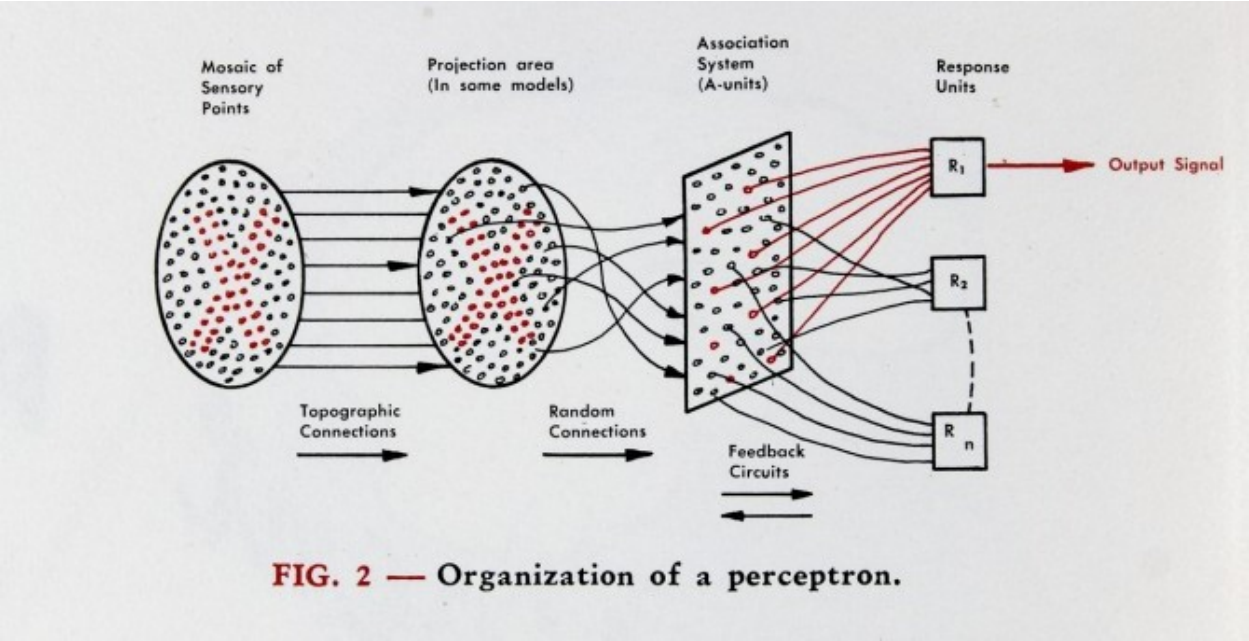

Figure 1: **An image of the perceptron from Rosenblatt's "The Design of an Intelligent Automaton," in *Research Trends*, a Cornell Aeronautical Laboratory publication, Vol. VI, No. 2, 1958.**
Source: https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon

are connected by learned weights to a response node that computes their weighted sum and applies an activation function to that sum, which produces the output signal (see Figure 2).[3]

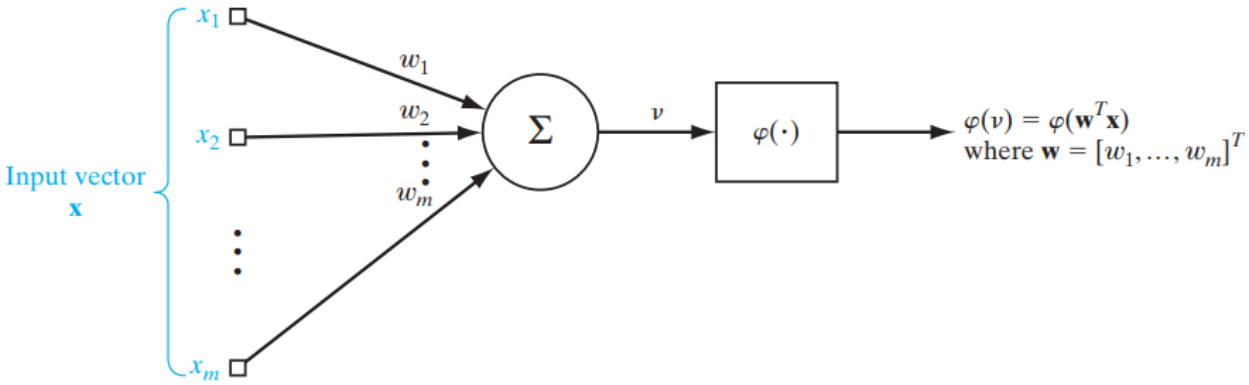

Figure 2: **Basic Unit of the Perceptron: a set of inputs connected by learned weights to a Response Node that computes the weighted sum and applies the activation function to that sum, which produces the output signal. Response Node is represented by the sequential mapping of the Sigma summation function (Σ) followed by the phi activation function ($\phi(\cdot)$).**
Source: Haykin, Simon (2009) *Neural Networks and Learning Machines — 3rd Edition*, page 538.

Furthermore, Rosenblatt's perceptron learned to classify linearly separable patterns by a supervised learning algorithm, which adjusted the weights of the sensory input connections based on the error between the

[3]Wikipedia (2023) Artificial Neuron https://en.wikipedia.org/wiki/Artificial_neuron

desired output and the actual output of the artificial neuron. The basic algorithm iterated over n input sample vectors $\mathbf{x}(n)$, and updated each of the input weights w_i for the next iteration $n+1$ according to the following equation:

$$w_i(n+1) = w_i(n) + \eta\,[d(n) - y(n)]\,x_i(n)$$

where $w_i(n)$ is the weight of the i-th input $x_i(n)$ of the current sample vector $\mathbf{x}(n) = \langle x_1, x_2, \cdots, x_i, \cdots, x_m \rangle$ (which has m dimensions), η is the learning rate, $d(n)$ is the desired output of the current sample, and $y(n)$ is the actual output of the current sample [Haykin 2009, pp. 47-55]. Rosenblatt also rigorously proved the perceptron convergence algorithm as a linearly separable pattern classifier in finite time-steps [Rosenblatt 1962, pp. 97-127].

Rosenblatt's perceptron was a groundbreaking model that led the way to artificial neural networks and artificial intelligence. However his model was limited to solving only linearly separable problems, because it lacked nonlinear activation functions.[4] These limitations were later overcome by more advanced models that were built upon the basic perceptron unit, such as multilayer perceptrons, which in turn led the way to even more advanced deep neural networks.[5]

2 Datasets

As of September 2023 only four individuals had submitted notebooks at kaggle.com with models trained on the Heart Disease Prediction dataset, however none of the models were neural network models. Therefore this dataset was appropriate for me to use in my Harvard edX Capstone project assignment, since my intent was to train neural network models in order to broaden the scope of my experience in the Harvard edX Data Science program to an important area of machine learning. Furthermore, I could make a unique contribution to the notebooks of models submitted for the Heart Disease Prediction dataset at kaggel.com.

The dataset is the Heart Disease Prediction dataset located at kaggel.com,[6] which contains data related to cardiac health, with 1,025 patient sample entries for model training (train.csv) and 303 patient samples entries for model testing (test.csv). Each patient sample includes 14 attributes, where the first 13 are variables for prediction of heart disease, and the last one is the actual outcome (training or test target) for that patient:

- Age: The age of the patient in years, represented as an integer value.
- Sex: The gender of the patient, represented by a binary value of 0 or 1.
- Chest Pain (cp) type: A categorical attribute indicating the type of chest pain experienced by the patient. It has four possible values: 0, 1, 2 , or 3.
- Resting Blood Pressure (trestbps): The resting blood pressure of the patient in millimeters of mercury (mm Hg), represented as an integer value.
- Serum Cholesterol (chol): The serum cholesterol level in mg/dl of the patient, represented as an integer value.
- Fasting Blood Sugar (fbs): Indicates whether the patient's fasting blood sugar is greater than 120 mg/dl, and is represented by a binary value of 0 or 1.
- Resting Electrocardiogram (restecg): A categorical attribute representing the results of the resting electrocardiogram (ECG). It has three possible values: 0, 1, or 2.
- Maximum Heart Rate (thalach): The maximum heart rate achieved by the patient in beats per minute (bpm), represented as an integer value.
- Exercise Induced Angina (exang): Indicates whether the patient experienced angina (chest pain) induced by exercise, and is represented by a binary value of 0 or 1.
- Oldpeak: Depression of Sinoatrial (ST) segment below baseline in ECG during recovery after exercise, which indicates myocardial ischemia and is represented by a numerical value in units of mm Hg.

[4] Wikipedia (2023) Perceptron https://en.wikipedia.org/wiki/Perceptron
Wikipedia (2023) Artificial Neuron https://en.wikipedia.org/wiki/Artificial_neuron
[5] Wikipedia (2023) Multilayer Perceptron https://en.wikipedia.org/wiki/Multilayer_perceptron
[6] https://www.kaggle.com/datasets/moazeldsokyx/heart-disease/data

- Slope: The slope of the peak exercise ST segment in ECG. It has three possible values: 0, 1, or 2.
- Number of Major Vessels (ca): Represents the number of major blood vessels colored by fluoroscopy. It has five possible values: 0, 1, 2 , 3, or 4.
- Thalassemia (thal): A categorical attribute indicating the thalassemia type (hemoglobin-defect anemia) of the patient. It has four possible values: 0 for normal, 1 for fixed defect, 2 for reversible defect, and 3 for irreversible defect.
- Target: The outcome variable which indicates the presence of heart disease in the patient, where a value of 0 signifies the absence of heart disease, and a value of 1 indicates the presence of heart disease.[7]

2.1 Data Conditioning

According to LeCun et al. (1998) in the classic paper "Efficient BackProp" of *Neural Networks: Tricks of the Trade, 2nd Edition*, convergence in neural network training is faster if the variables are centered with zero means and are scaled to have the same covariance, where the covariance value should match the sigmoid activation function, i.e., covariance = 1. In practice this is easily accomplished using the scale() function from the base package in R, with the center and scale arguments set to TRUE. The function centers each column (variable) by subtracting the column mean, and then scales (normalizes) the values by the standard deviation in each column. This way the binary variables with values of 0 or 1 will have the same influence on training the network as the blood pressure, cholesterol, and heart rate variables with values that range from 94 to 200 mm Hg, 126 to 564 mg/dl, and 71 to 202 bpm respectively (compare Tables 1 & 2).

Table 1: **Sample of Raw Data from the Training Dataset**

```
train[1:14,]
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
1	52	1	0	125	212	0	1	168	0	1.0	2	2	3	0
2	53	1	0	140	203	1	0	155	1	3.1	0	0	3	0
3	70	1	0	145	174	0	1	125	1	2.6	0	0	3	0
4	61	1	0	148	203	0	1	161	0	0.0	2	1	3	0
5	62	0	0	138	294	1	1	106	0	1.9	1	3	2	0
6	58	0	0	100	248	0	0	122	0	1.0	1	0	2	1
7	58	1	0	114	318	0	2	140	0	4.4	0	3	1	0
8	55	1	0	160	289	0	0	145	1	0.8	1	1	3	0
9	46	1	0	120	249	0	0	144	0	0.8	2	0	3	0
10	54	1	0	122	286	0	0	116	1	3.2	1	2	2	0
11	71	0	0	112	149	0	1	125	0	1.6	1	0	2	1
12	43	0	0	132	341	1	0	136	1	3.0	1	0	3	0
13	34	0	1	118	210	0	1	192	0	0.7	2	0	2	1
14	51	1	0	140	298	0	1	122	1	4.2	1	3	3	0

Table 2: **Sample of Conditioned Data from the Training Dataset**

```
scaledTrain[1:14,]
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
[1,]	-0.26830561	0.6611813	-0.91530860	-0.37745126	-0.65901038	-0.4186735	0.890820	0.8209198	-0.7119396	-0.06085868	0.9949476	1.2086307	1.0893199	-1.0261968
[2,]	-0.15807986	0.6611813	-0.91530860	0.47887354	-0.83345431	2.3861656	-1.003559	0.2558430	1.4032432	1.72629436	-2.2425804	-0.7316143	1.0893199	-1.0261968
[3,]	1.71575790	0.6611813	-0.91530860	0.76431513	-1.39555140	-0.4186735	0.890820	-1.0481803	1.4032432	1.30078173	-2.2425804	-0.7316143	1.0893199	-1.0261968
[4,]	0.72372615	0.6611813	-0.91530860	0.93558009	-0.83345431	-0.4186735	0.890820	0.5166477	-0.7119396	-0.91188394	0.9949476	0.2385082	1.0893199	-1.0261968
[5,]	0.83395190	-1.5109689	-0.91530860	0.36469690	0.93036760	2.3861656	0.890820	-1.8740617	-0.7119396	0.70506405	-0.6238164	2.1787531	-0.5218676	-1.0261968
[6,]	0.39304890	-1.5109689	-0.91530860	-1.80465926	0.03876532	-0.4186735	-1.003559	-1.1785826	-0.7119396	-0.06085868	-0.6238164	-0.7316143	-0.5218676	0.9735213
[7,]	0.39304890	0.6611813	-0.91530860	-1.00542278	1.39555140	-0.4186735	2.785199	-0.3961686	-0.7119396	2.83262719	-2.2425804	2.1787531	-2.1330550	-1.0261968
[8,]	0.06237164	0.6611813	-0.91530860	1.62063993	0.83345431	-0.4186735	-1.003559	-0.1788314	1.4032432	-0.23106374	-0.6238164	0.2385082	1.0893199	-1.0261968
[9,]	-0.92966011	0.6611813	-0.91530860	-0.66289286	0.05814798	-0.4186735	-1.003559	-0.2222989	-0.7119396	-0.23106374	0.9949476	-0.7316143	1.0893199	-1.0261968
[10,]	-0.04785411	0.6611813	-0.91530860	-0.54871622	0.77530633	-0.4186735	-1.003559	-1.4393873	1.4032432	1.81139688	-0.6238164	1.2086307	-0.5218676	-1.0261968
[11,]	1.82598365	-1.5109689	-0.91530860	-1.11959942	-1.88011786	-0.4186735	0.890820	-1.0481803	-0.7119396	0.44975647	-0.6238164	-0.7316143	-0.5218676	0.9735213
[12,]	-1.26033736	-1.5109689	-0.91530860	0.02216698	1.84135255	2.3861656	-1.003559	-0.5700384	1.4032432	1.64119183	-0.6238164	-0.7316143	1.0893199	-1.0261968
[13,]	-2.25236912	-1.5109689	0.05590394	-0.77706950	-0.69777570	-0.4186735	0.890820	1.8641384	-0.7119396	-0.31616626	0.9949476	-0.7316143	-0.5218676	0.9735213
[14,]	-0.37853136	0.6611813	-0.91530860	0.47887354	1.00789824	-0.4186735	0.890820	-1.1785826	1.4032432	2.66242214	-0.6238164	2.1787531	1.0893199	-1.0261968

[7] The 1,025 target outcomes in train.csv showed a fairly balanced dataset with 499 healthy and 526 heart disease patients.

3 Analysis

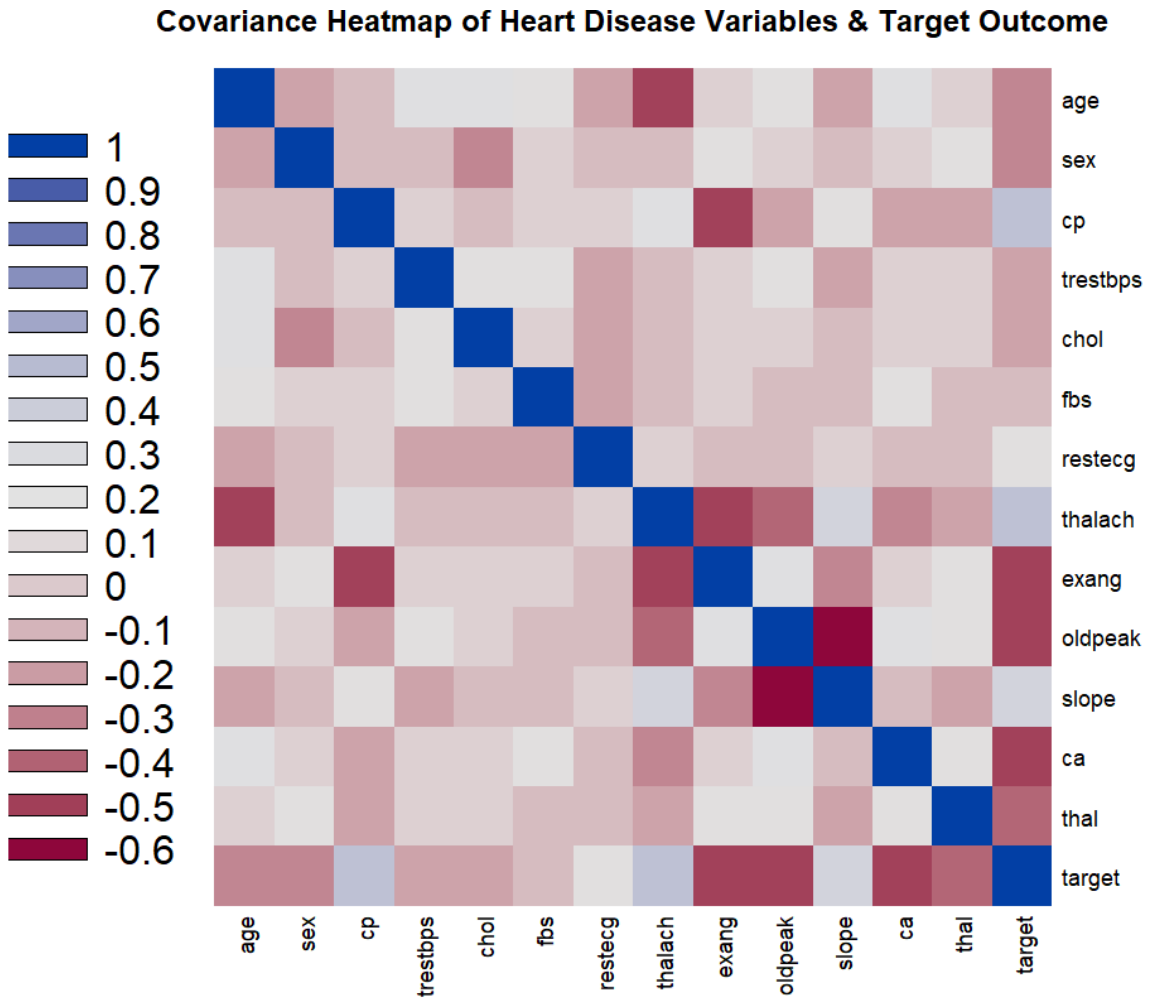

Figure 3: **Covariance Heatmap of Heart Disease Variables & Target Outcomes from Patient Training Data**

Figure 3 shows a heatmap of the covariances between the 14 attributes of the patient training data. The most relevant covariances for model prediction are between the 13 variables of heart disease and the actual heart health outcomes of the patients, i.e., the training targets. Thus, the target row (bottom) and the target column (far right) show that Chest Pain, Maximum Heart Rate, Exercise Induced Angina, Oldpeak, and Number of Major Vessels all have the strongest covariance with the patient outcome. Furthermore, Table 3 shows that these strongest variable covariances with the target are all about ± 0.4 in value.

4 Methods

4.1 Modeling Approach – Feedforward Neural Network

Multi-Layer Perceptrons (MLP) are a generalization and extension of Rosenblatt's perceptron model, which overcame some of its limitations. MLPs are networks of the basic perceptron unit that are organized in two or more layers of basic units, also called nodes, where the output of a layer of nodes serves as the input to the next layer of nodes. Typically, the output from each node of a layer is fully connected as weighted inputs

Table 3: **Covariance Values of Heart Disease Variables & Target Outcomes from Patient Training Data**

covTrain

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
age	1.00000000	-0.10324030	-0.07196627	0.27112141	0.21982253	0.121243479	-0.13269617	-0.390227075	0.08816338	0.20813668	-0.16910511	0.27155053	0.07229745	-0.22932355
sex	-0.10324030	1.00000000	-0.04111909	-0.07897377	-0.19825787	0.027200461	-0.05511721	-0.049365243	0.13915681	0.08468656	-0.02666629	0.11172891	0.19842425	-0.27950076
cp	-0.07196627	-0.04111909	1.00000000	0.03817742	-0.08164102	0.079293586	0.04358061	0.306839282	-0.40151271	-0.17473348	0.13163278	-0.17620647	-0.16334148	0.43485425
trestbps	0.27112141	-0.07897377	0.03817742	1.00000000	0.12797743	0.181766624	-0.12379409	-0.039264069	0.06119697	0.18743411	-0.12044531	0.10455372	0.05927618	-0.13877173
chol	0.21982253	-0.19825787	-0.08164102	0.12797743	1.00000000	0.026917164	-0.14741024	-0.021772091	0.06738223	0.06488031	-0.01424787	0.07425934	0.10024418	-0.09996559
fbs	0.12124348	0.02720046	0.07929359	0.18176662	0.02691716	1.000000000	-0.10405124	-0.008865857	0.04926057	0.01085948	-0.06190237	0.13715626	-0.04217732	-0.04116355
restecg	-0.13269617	-0.05511721	0.04358061	-0.12379409	-0.14741024	-0.104051244	1.00000000	0.048410637	-0.06560553	-0.05011425	0.08608609	-0.07807235	-0.02050406	0.13446821
thalach	-0.39022708	-0.04936524	0.30683928	-0.03926407	-0.02177209	-0.008865857	0.04841064	1.000000000	-0.38028087	-0.34979616	0.39530784	-0.20788842	-0.09806817	0.42289550
exang	0.08816338	0.13915681	-0.40151271	0.06119697	0.06738223	0.049260570	-0.06560553	-0.380280872	1.00000000	0.31084376	-0.26733547	0.10784854	0.19720104	-0.43802855
oldpeak	0.20813668	0.08468656	-0.17473348	0.18743411	0.06488031	0.010859481	-0.05011425	-0.349796163	0.31084376	1.00000000	-0.57518854	0.22181603	0.20267203	-0.43844127
slope	-0.16910511	-0.02666629	0.13163278	-0.12044531	-0.01424787	-0.061902374	0.08608609	0.395307843	-0.26733547	-0.57518854	1.00000000	-0.07344041	-0.09409006	0.34551175
ca	0.27155053	0.11172891	-0.17620647	0.10455372	0.07425934	0.137156259	-0.07807235	-0.207888416	0.10784854	0.22181603	-0.07344041	1.00000000	0.14901387	-0.38208529
thal	0.07229745	0.19842425	-0.16334148	0.05927618	0.10024418	-0.042177320	-0.02050406	-0.098068165	0.19720104	0.20267203	-0.09409006	0.14901387	1.00000000	-0.33783815
target	-0.22932355	-0.27950076	0.43485425	-0.13877173	-0.09996559	-0.041163547	0.13446821	0.422895496	-0.43802855	-0.43844127	0.34551175	-0.38208529	-0.33783815	1.00000000

to all of the nodes in the next layer. To introduce nonlinearity into the network, the nodes in each layer of modern day MLPs also use nonlinear activation functions, such as logistic or tanh sigmoidal functions. This allows the network to model and learn nonlinear and complex functions and patterns. Modern day MLPs are better known as Feedforward Neural Networks (FNN), because like MLPs, the data flows in one direction only, from the input layer to the output layer of the network, however FNNs use nonlinear activation functions, unlike traditional MLPs (see Figure 4).[8]

Compared to Figure 4, Figures 2 & 7 use a similar but more compact nomenclature, which allows for cleaner and less cumbersome equations. This cleaner nomenclature does not use superscripts to indicated layer IDs, instead the subscript i, j or k indicates the layer ID with the understanding that there are an arbitrary m node IDs in a layer. The cleaner nomenclature also uses the two subscripts convention with the weights, where the first subscript indicates the receiving layer and the second subscript indicates the sending layer. In this cleaner nomenclature, when a number replaces an i, j or k subscript, it indicates the node ID within that layer. **The following equations (1-18) were adapted from reference Haykin (2009), Chapter 4 - Multilayer Perceptrons (pp. 122-137), from Wikipedia (2023) Backpropagation,[9] and from reference Deisenroth (2023), Chapter 5 - Vector Calculus (pp. 139-164).**

For sample n, the output signal $y_k(n)$ of a node in layer k of the network is computed with respect to $v_k(n)$, its sum of weighted $w_{kj}(n)$ input signals $y_j(n)$ from layer j:

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{1}$$

where m is the total number of inputs (excluding bias) from the preceding layer j. For the sake of a homogeneous representation in the equation, the bias b_k applied to the node in layer k is substituted by a weight w_{k0} that is assigned to a fixed input $y_0 = +1$ from layer j at position $m = 0$ (see Figure 7). Thus, the output signal $y_k(n)$ of a node in layer k is:

$$y_k(n) = \phi_k(v_k(n)) \tag{2}$$

where ϕ_k is an arbitrary Activation Function applied to $v_k(n)$. In this model all the hidden layers in the neural network (e.g., layers i & j in Figure 7) used a non-linear sigmoidal activation function known as the Standard Logistic-Growth Distribution-Function, which is shown in Figure 5. However, the output layer in this model (e.g., layer k in Figure 7) generally used a linear activation function where $\phi_k = 1$, although some testing was also done using the logistic activation function in the output layer (see Future Work for details).

[8] Wikipedia (2023) Feedforward Neural Network https://en.wikipedia.org/wiki/Feedforward_neural_network
[9] Wikipedia (2023) Backpropagation https://en.wikipedia.org/wiki/Backpropagation

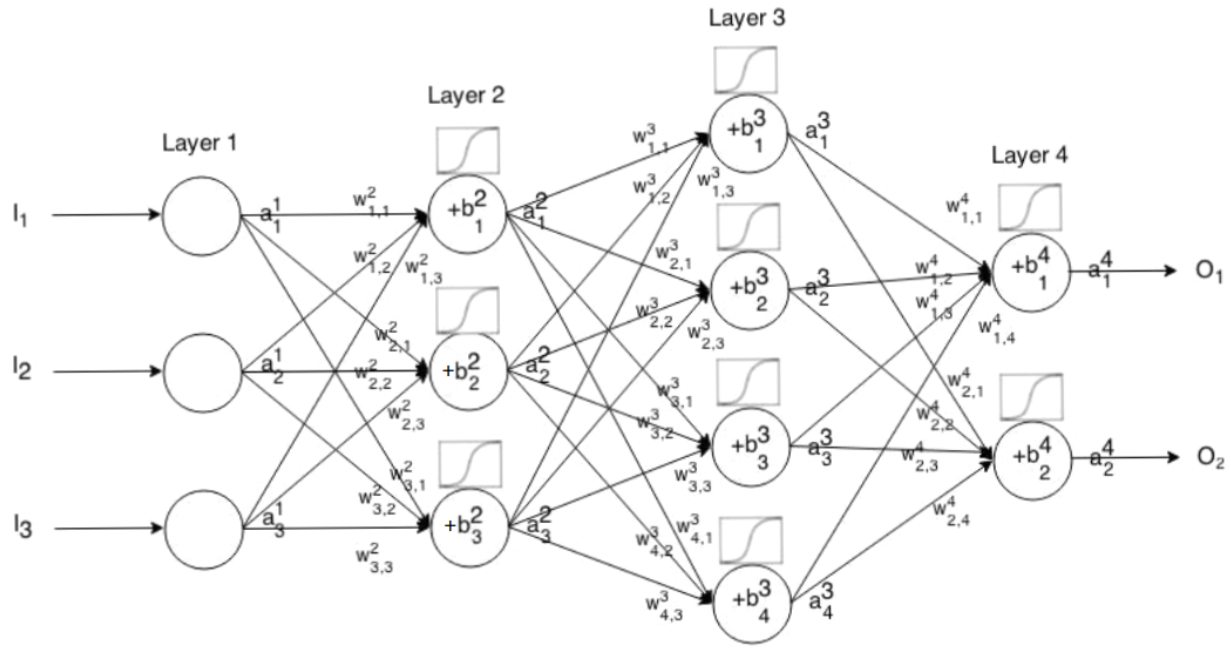

Figure 4: **Fully connected Feedforward Neural Network with two Hidden Layers: Layer 2 & Layer 3. Layer 1 = Input Layer, Layer 4 = Output Layer, I = Input Value, O = Output Value, a = Activated Output from Node, b = Bias, w = Weight Factor, Superscript = Layer ID, One Subscript = Node ID, Two Subscripts = Receiving Node ID & Sending Node ID, and Sigmoid indicates that node output was generated with a sigmoidal activation function.**
Source: https://th.bing.com/th/id/R.367bc64e39794b1108280e56ddcbb353?rik=W%2b5ugX8AzvZ%2bPQ&riu=http%3a%2f%2fi.stack.imgur.com%2f76Kuo.png&ehk=U0vtGsGE9i298foJYYgZkOIuFe7IVTlcrRdVA6uM1xM%3d&risl=&pid=ImgRaw&r=0
with some annotations added for clarity.

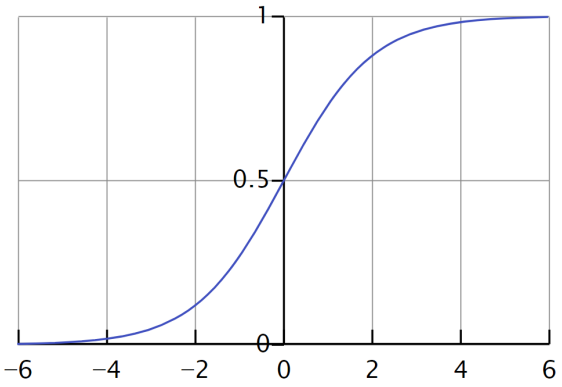

Figure 5:
Logistic Function - Standard Distribution Plot
Source: https://en.wikipedia.org/wiki/Logistic_function

The horizontal axis in Figure 5 represents the numerical values of the $v_k(n)$ term in the right hand side of Equation 3:

$$\phi_k(v_k(n)) = \frac{1}{1 + e^{-v_k(n)}} \tag{3}$$

and the vertical axis in Figure 5 represents all the values between 0 and 1 for the activated output $\phi_k(v_k(n))$ of the node, and is equivalent to $y_k(n)$ by Equation 2. Furthermore, the Logistic Function is a continuous and differentiable function, which is important for training a neural network. The derivative of Equation 3 is conveniently expressed as:

$$\phi'_k(v_k(n)) = y_k(n)\left[1 - y_k(n)\right] = \frac{\partial y_k(n)}{\partial v_k(n)} \tag{4}$$

and is equivalent to the partial derivative of Equation 2 with respect to $v_k(n)$, as shown in Equation 4. However, when $\phi_k = 1$ then the partial derivative of Equation 2 with respect to $v_k(n)$ is expressed as:

$$\frac{\partial y_k(n)}{\partial v_k(n)} = 1 \tag{5}$$

Also, the partial derivative of Equation 1 with respect to $w_{kj}(n)$ is expressed as:

$$\frac{\partial v_k(n)}{\partial w_{kj}(n)} = y_j(n) \quad \text{where all the other weights in the summation of Equation 1} \tag{6}$$

are treated as constants with zero derivatives.

4.2 Model Training – Backpropagation

To train a Neural Network (NN), a generalization of Rosenblatt's supervised learning algorithm for a perceptron is needed. One of the most widely used algorithms for training NNs is backpropagation (BP), which was popularized by Rumelhart, Hinton & Williams through their experimental analysis of the technique in 1985,[10] which was also published in 1986.[11] BP is the main algorithm in gradient descent (GD), which reduces the error by iteratively updating the weights in the opposite direction of the error gradient. The GD iterative process consists of a two phase cycle: forward propagation (FP) and backpropagation (BP). In the FP phase, GD computes the network output for a given amount of training input and calculates the error of the output. In the BP phase, GD calculates the gradient of the error at the weighted links to the output layer in order to update those weights, and then at the weighted links to each preceding hidden layer to update those respective weights, which finally ends with calculation of the error gradient and the updates at the weighted links from the network's input layer. Repeated GD iterations of the FP/BP cycle updates the network's weights to descend the error gradient over many small steps and ideally arrive at the global minimum of the error landscape, which is also called the Loss Landscape (for example see Figure 6).

A neural network model in this project was trained in *batch mode*, meaning that the entire dataset of training samples was used in each iteration of learning, where learning refers to an update of the model's network parameters, which reduces the error between the model's output of classifications and the target classifications of the training samples. The batch error in this model was calculated with a Half of the Sum of Squared Errors (HSSE) function on the entire batch of training samples and the respective classifications generated through one output node:

$$E = \frac{1}{2} \sum_{n=1}^{N} \left[t(n) - y_k(n) \right]^2 \tag{7}$$

where n is the n-th sample in the batch of N total samples in the training dataset, $t(n)$ is the target classification of the n-th training sample, and $y_k(n)$ is the neural network model's output classification of the n-th training sample. In practice, the network is trained to a small degree by each sample n, such that the output error and error gradients at the weights for each training sample are added to running totals that are used to update the learning parameters after all N samples of the batch have been processed. Thus we also need to define the error function of a single training sample:

$$E(n) = \frac{1}{2} \left[t(n) - y_k(n) \right]^2 \tag{8}$$

The one-half factor in the error function is a convenience term that simplifies the partial derivative of the error function with respect to the output $y_k(n)$:

[10] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. (1985) "Learning Internal Representations by Error Propagation." Institute of Cognitive Science, Report 8506, September 1985, University of California - San Diego, La Jolla, CA.

[11] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. (1986) "Learning Internal Representations by Error Propagation" in *Parallel Distributed Processing, Volume 1: Explorations in the Microstructure of Cognition: Foundations*, by David E. Rumelhart, James L. McClelland, and PDP Research Group (Editors), MIT Press, ©1986, pp. 318-362.
Rumelhart, Hinton & Williams (1986) "Learning Representations by Back-Propagating Errors" *Nature,* Vol. 323, pp. 533–536.

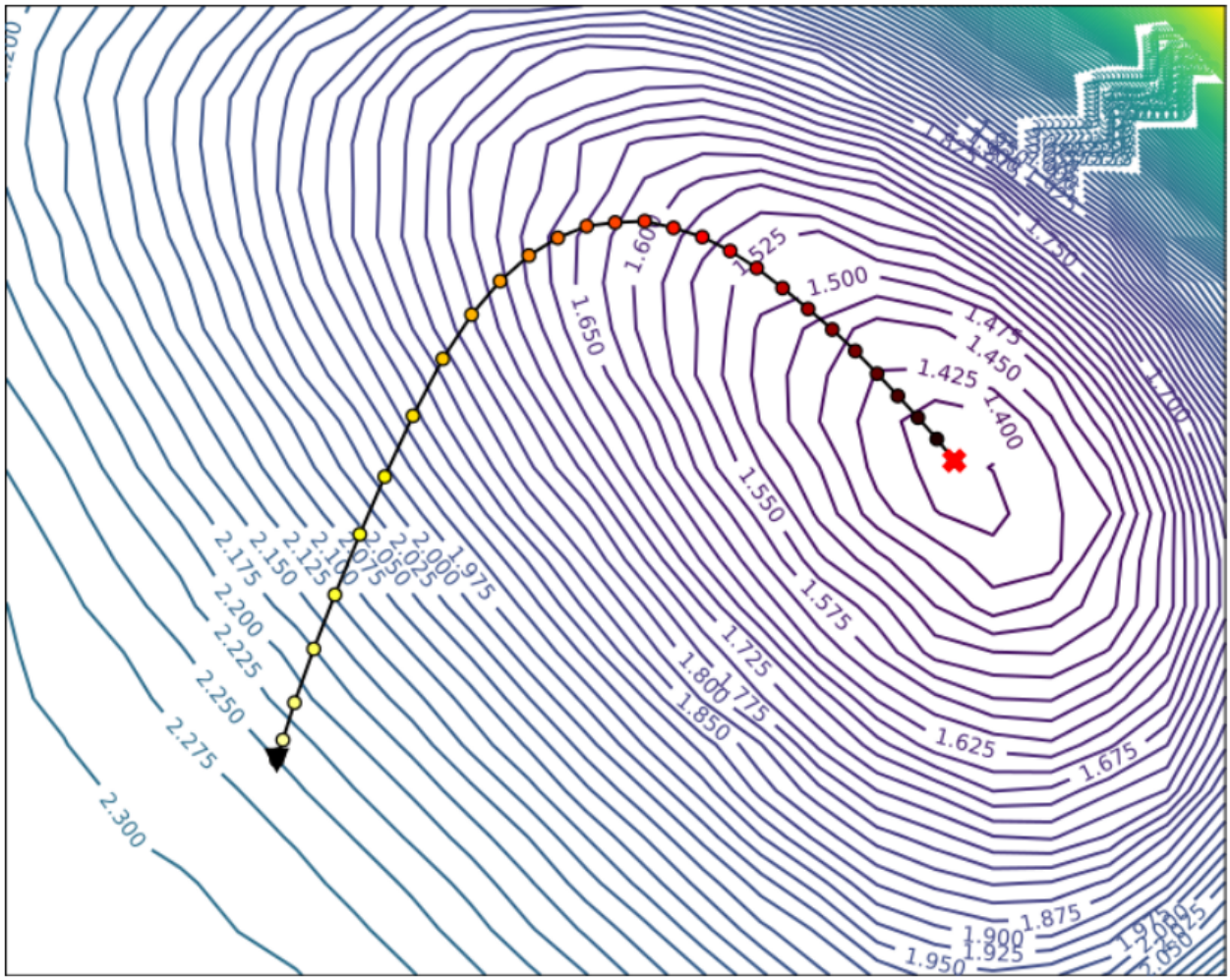

Figure 6: **Loss Landscape with minimization trajectory by gradient descent. (Example for illustration.)** Source: https://sc19.supercomputing.org/proceedings/workshops/workshop_files/ws_mlhpce111s1-file1.pdf

$$\frac{\partial E(n)}{\partial y_k(n)} = y_k(n) - t(n) \tag{9}$$

The partial derivatives in Equations 4/5, 6 & 9 have overlapping dependencies and therefore can be combined using the *chain rule* of differentiation to express the partial derivative of the error with respect to the weight $w_{kj}(n)$:

$$\frac{\partial E(n)}{\partial w_{kj}(n)} = \frac{\partial E(n)}{\partial y_k(n)} \cdot \frac{\partial y_k(n)}{\partial v_k(n)} \cdot \frac{\partial v_k(n)}{\partial w_{kj}(n)} \tag{10}$$

Equation 10 expresses the *key mathematical relationship of backpropagation (BP), where repeated application of the chain rule may be used to successively compute the gradient of the error at each weight and bias parameter in each of the preceding layers of the network, hence backpropagation of the error.* In this case, Equation 10 may be used to compute the gradient of the error at weight $w_{kj}(n)$ in Figure 7, and then by repeated application of the chain rule, the equation may be extended to compute the gradient of the error at weight $w_{ji}(n)$ in the preceding layer of the network:

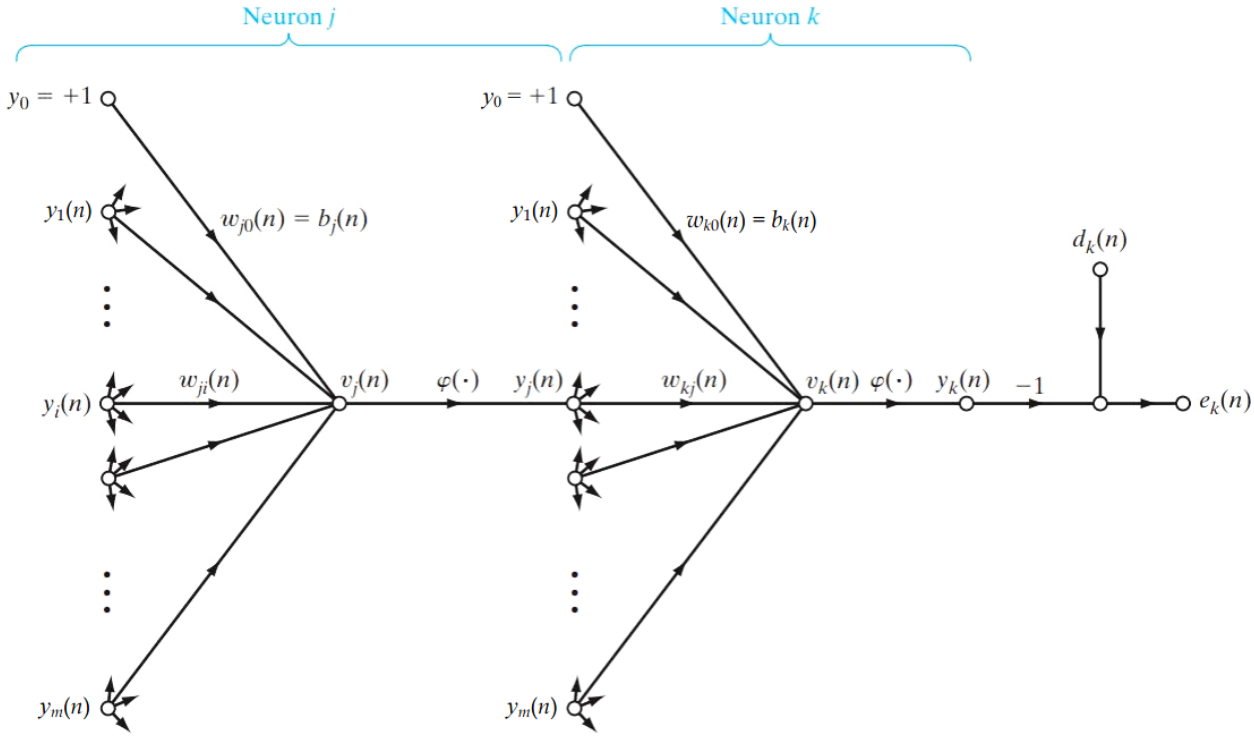

Figure 7: **Signal flow graph of output neuron k connected to hidden neuron j, which is connected to hidden neuron i. (Note that bias weight $w_{k0}(n) = b_k(n)$ is different for each of the m bias connections in layer k, where in actual use, a number replaces the k subscript to indicate the node ID within that layer. Likewise for the $w_{j0}(n) = b_j(n)$ bias weights. See Figure 4, Appendix B.4, and Appendix B.5 for examples.)**
Source: Haykin, Simon (2009) *Neural Networks and Learning Machines — 3rd Edition*, page 132, with some annotations added for clarity.

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial y_k(n)} \cdot \frac{\partial y_k(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{11}$$

where

$$\frac{\partial y_k(n)}{\partial y_j(n)} = \frac{\partial y_k(n)}{\partial v_k(n)} \cdot \frac{\partial v_k(n)}{\partial y_j(n)} \tag{12}$$

and the partial derivative of Equation 1 with respect to $y_j(n)$ is expressed as:

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{13}$$

Also, hidden layer j only uses the Logistic Function for activation, thus analogous to Equations 3 & 4:

$$\phi_j'(v_j(n)) = y_j(n)\,[1 - y_j(n)] = \frac{\partial y_j(n)}{\partial v_j(n)} \tag{14}$$

and analogous to Equation 6 for layer j:

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \quad \text{because} \quad v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \quad \text{is analogous to Equation 1 for layer } j. \tag{15a, 15b}$$

Finally, the values of

$\frac{\partial E(n)}{\partial y_k(n)}$ and $\frac{\partial y_k(n)}{\partial v_k(n)}$ in Equation 11 and Equation 12 are recycled from Equation 10, the previous step of backpropagation. *This efficiency is another key advantage of BP, where an increasing number of partial derivative values get reused as the BP progresses through each preceding layer of the network.*

Equations 10 & 11 describe the two steps of backpropagation down a single arbitrary path of a generic network, shown in Figure 7, with one output node in layer k preceded by two hidden layers j & i. However as implied by Figure 7 and clearly shown in Figure 4, there may be multiple paths of backpropagation from an output node to a given weight of a hidden layer, where all such paths must be summed together to compute the gradient of the error at the given weight. *In order to account for all the paths to each weight in a network, it is most efficient to write and calculate with a vectorized version of the chain rule for backpropagation,* where all of the weights of a given layer are represented by a matrix $\mathbf{W}_\ell$, the weighted sums of inputs to each node of a given layer are represented by a vector $\mathbf{v}_\ell$, and all the activated outputs of a given layer are represented by a vector $\mathbf{y}_\ell$:

$$\frac{\partial E(n)}{\partial \mathbf{W}_\ell(n)} = \frac{\partial E(n)}{\partial \mathbf{y}_\ell(n)} \cdot \frac{\partial \mathbf{y}_\ell(n)}{\partial \mathbf{v}_\ell(n)} \cdot \frac{\partial \mathbf{v}_\ell(n)}{\partial \mathbf{W}_\ell(n)} \tag{16}$$

where under the hood, these partial derivative terms are multivariable Jacobian matrices and Jacobian vectors.[12] However, the error remains a scalar value because the error between the output vector $\mathbf{y}_\ell(n)$ and the target vector $\mathbf{t}(n)$ of a sample (n) is the modulus squared (i.e., Euclidean norm squared) of the *difference vector*, and furthermore, the partial derivative of the vectorized error function with respect to the output vector is also a scalar value:

$$E(n) = \frac{1}{2}||\mathbf{t}(n) - \mathbf{y}_\ell(n)||^2 \quad \text{and} \quad \frac{\partial E(n)}{\partial \mathbf{y}_\ell(n)} = ||\mathbf{y}_\ell(n) - \mathbf{t}(n)|| \tag{17a, 17b}$$

Note that the vectorized form of the error function is more generalized because it implies an arbitrary number of output nodes, where the output vector $\mathbf{y}_\ell(n)$ and the target vector $\mathbf{t}(n)$ each have the same number of vector elements as output nodes. Furthermore, for the gradient of the error at the output node(s) of a network with an arbitrary number of L layers, with respect to the weights of an arbitrary layer $\mathbf{W}_\ell$ in the network:

$$\frac{\partial E(n)}{\partial \mathbf{W}_\ell(n)} = \frac{\partial E(n)}{\partial \mathbf{y}_{L}(n)} \cdot \frac{\partial \mathbf{y}_{L}(n)}{\partial \mathbf{y}_{L-1}(n)} \cdot \frac{\partial \mathbf{y}_{L-1}(n)}{\partial \mathbf{y}_{L-2}(n)} \cdots \frac{\partial \mathbf{y}_{\ell+1}(n)}{\partial \mathbf{y}_\ell(n)} \cdot \frac{\partial \mathbf{y}_\ell(n)}{\partial \mathbf{v}_\ell(n)} \cdot \frac{\partial \mathbf{v}_\ell(n)}{\partial \mathbf{W}_\ell(n)} \tag{18}$$

completely describes the backpropagation from the last layer L to any internal layer ℓ, and is how most computer program packages handle BP to take advantage of the efficiency of linear algebra algorithms, automatic (numerical) differentiation, and GPU processors (if available).[13]

4.3 Modeling Tools – neuralnet & RPROP+

The *neuralnet* package[14] is an R package that was used in this project for creating fully connected feedforward neural networks (FNN) with an arbitrary number of hidden layers and arbitrary numbers of nodes in the layers. Choice of training methods were: backpropagation, resilient backpropagation (RPROP) with or without weight backtracking, and a globally convergent version of resilient backpropagation (GRPROP). Choice of start weights were: random (may be used with a seed), and a custom set of weights supplied as a vector. Choice of activation functions were: logistic, tanh, and linear. Choice of error functions were: sum of squared errors, and cross-entropy. The output is a model of fitted weights, the error, threshold reached,

[12] Wikipedia (2023) Jacobian Matrix & Determinant https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant
[13] Deisenroth (2023), Chapter 5 - Vector Calculus (pp. 139-164).
[14] neuralnet R-Documentation https://www.rdocumentation.org/packages/neuralnet/versions/1.44.2/topics/neuralnet

and number of training steps needed. The trained neuralnet model can then be used as the first argument of a predict function to generate predictions for new or test data.[15]

In this project, the neuralnet training for most of the models was done with backpropagation (BP) in a fast adaptive optimization algorithm of gradient descent (GD) known as Resilient Backpropagation with Backtracking (RPROP+). The main idea behind RPROP+ is to use only the sign of the gradient of the error function to update the weights of the network, not the magnitude, and increase or decrease the weights by set factors based on the gradient sign changes. This makes the algorithm insensitive to the scale of the gradient, which avoids the problems of slow convergence and getting stuck in flat regions or saddle points of the error function, where the gradient is small or zero.[16] RPROP+ also incorporates a weight backtracking mechanism that restores the previous weights if the current weights caused the error to increase. Furthermore, the update values for those restored weights are also decreased by the set factor. This mechanism helps the algorithm to avoid oscillations and escape from local minima. The weight backtracking mechanism also ensures that the error function decreases monotonically at each iteration, which is a desirable property for optimization algorithms [Riedmiller & Braun 1993].

The RPROP+ algorithm can be summarized as follows [Igel & Hüsken 2003; also reference Footnote #14]:

- Initialize the weights randomly.
- Set the initial update values of the weights (default = 0.1). This value is used only after the first backpropagation because there is no history.
- Repeat the following on the full batch of training data samples (i.e., learning by epoch) until convergence of error gradient below the set threshold (default = 0.01) or until maximum number of iterations (default = 100,000 steps) – whichever comes first:
  - Compute the gradient of the error function with respect to the weights. If gradient is above the threshold setting then continue to next step, otherwise report the final error, final gradient and final weights.
  - Look at the signs of the last and the current gradients for the weights.
    - If they have the same sign, that means the weight updates are going in the right direction, and the step-size should be increased by a set factor (default = 1.2).
    - If they have different signs, that means the weight update was too large and jumped over a local minima, thus restore the previous weights and decrease the update value for those restored weights by a set factor (default = 0.5).
  - Increment the number of epoch trained iterations by one step. If less than stepmax, train another epoch, otherwise report the final weights and compute the final error and final gradient.

5 Results & Discussion

Table 4 summarizes the performance results for fully connected Feedforward Neural Networks (FNN) that were trained & tested on the Heart Disease Prediction datasets. Performance was evaluated in terms of basic network architecture, such as number of hidden layers and number of nodes within each layer. Furthermore, performance was evaluated in terms of the general type of model (GLM or FNN), the type of gradient descent algorithm, different seeds & initial (start) weights, and threshold levels.

Understanding the logic behind the taxonomy of the FNN model names will help with making sense out of the discussions on interpretations of the data. The basic name for a FNN model is the prefix "nnTrained." followed by an *integer* that indicates which *hidden layer/node architecture* it uses. (Note that the integer to architecture correspondence followed no particular order.) The basic name will never change for FNNs that use the same NN architecture.[17] If the basic name does not also have a suffix letter attached to it, then the FNN model used only the default settings of this project which are as follows:

[15] Package 'neuralnet' - Training of Neural Networks https://cran.r-project.org/web//packages/neuralnet/neuralnet.pdf
[16] Wikipedia (2023) Rprop https://en.wikipedia.org/wiki/Rprop
[17] Due to the limited scope of this paper, FNN may be used interchangeably with NN.

- Start Weights generated by set.seed(1) and neuralnet function with rnorm(n=TNIM, mean=0, sd=1)
- Activation Function of Hidden Layers = logistic *
- Activation Function of Output Layer = linear *
- Gradient Descent Algorithm = RPROP+
- Threshold Setting = 0.01
- Learning Rate Adjustment Factors of 0.5 for Decrease and 1.2 for Increase *
- Error Function = Sum of Squared Errors *
- where TNIM indicates the Total number of Nodes In the feedforward neural network Model
- * indicates Default Setting was constant for all models types and variations throughout the project

If the basic name of a FNN does have a suffix letter attached to it, then that indicates one or more of the default settings listed above (i.e., Seed/Start Weights, Algorithm, and/or Threshold) were changed as indicated in Table 4:

Table 4: **Summary of Logistic Regression Model (GLM), Feedforward Neural Network (FNN) Models, and Performance Results**

	Model	Seed_StartWeights	Hidden	Hid_Act	Out_Act	Algorithm	Steps	Threshold	Error	MCC	Accuracy	Acc.pValue	Sensitivity	Specificity	BA
1	fit_logistic	<NA>	<NA>	<NA>	logistic	glm-logistic	1	NA	NA	0.713	0.8581	2.0e-16	0.8848	0.8261	0.8555
2	nnTrained.0a	1	16,8,4,2	logistic	linear	backprop	107607	0.01	0.01832	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000
3	nnTrained.0b	1	16,8,4,2	logistic	linear	rprop-	1295	0.01	0.00191	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000
4	nnTrained.0	1	16,8,4,2	logistic	linear	rprop+	1315	0.01	0.00536	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000
5	nnTrained.3	1	14,7,3	logistic	linear	rprop+	778	0.01	0.00612	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000
6	nnTrained.4	1	13,7,3	logistic	linear	rprop+	12887	0.01	0.00561	0.993	0.9967	2.0e-16	0.9939	1.0000	0.9970
7	nnTrained.4a	4	13,7,3	logistic	linear	rprop+	1627	0.01	0.00070	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000
8	nnTrained.5	1	13,6,3	logistic	linear	rprop+	604	0.01	0.01015	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000
9	nnTrained.1	1	12,6,3	logistic	linear	rprop+	1137	0.01	0.00448	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000
10	nnTrained.2	1	10,5,3	logistic	linear	rprop+	2045	0.01	0.01425	0.987	0.9934	2.0e-16	0.9879	1.0000	0.9939
11	nnTrained.2h	1	10,5,3	logistic	linear	rprop+	7443	0.001	0.00205	0.987	0.9934	2.0e-16	0.9879	1.0000	0.9939
12	nnTrained.2a	4	10,5,3	logistic	linear	rprop+	8260	0.01	0.02325	0.993	0.9967	2.0e-16	0.9939	1.0000	0.9970
13	nnTrained.2i	4	10,5,3	logistic	linear	rprop+	70931	0.001	0.00142	0.993	0.9967	2.0e-16	0.9939	1.0000	0.9970
14	nnTrained.2b	startweights10.5.3	10,5,3	logistic	linear	rprop+	1335	0.01	0.00216	0.993	0.9967	2.0e-16	0.9939	1.0000	0.9970
15	nnTrained.2c	startweights10.5.3	10,5,3	logistic	linear	rprop+	2557	0.001	0.00074	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000

5.1 Table 4 Attribute Details

In Table 4, the Steps, Threshold, and Error attributes are measures of the *Training Data performance* of the FNN models, whereas the MCC, Accuracy, Sensitivity and Specificity attributes are measures of the *Test Data performance* of all the models in the table (including GLM). The details of all the Table 4 attributes are as follows:

Seed_StartWeights: "Seed" refers to an integer value in the seed argument of the set.seed() function in R base package. "StartWeights" refers to a vector of weight values for the startweights argument in the neuralnet() function in the R 'neuralnet' package. Both of these methods were used for controlling the Initial Weights in the training of the neural network models. If the startweights argument is NULL, then *neuralnet() uses random initialization of the rnorm() function to create the initial weights drawn from a normal distribution with mean = 0 and standard deviation = 1.* Furthermore, the random seed that the neuralnet() function uses with the rnorm() function can be controlled by running set.seed() immediately prior to executing the neuralnet() function. Hence, the Seed numbers in Table 4 are the random seed values that were run immediately prior to the neuralnet function.

Hidden: Vector of integer values that indicate the number of nodes in each hidden layer of the neural network model. For example, (4,2,1) indicates 3 hidden layers with 4 nodes in the first hidden layer (closest to the input layer), 2 nodes in the second hidden layer, and 1 node in the third (last) hidden layer (closest to the output layer).

Hid_Act: The activation function used in each of the hidden layers of a neural network model. For example, "logistic" refers to the Standard Logistic-Growth Distribution-Function (see Figure 5 and Equation 3).

Out_Act: The activation function used in the output layer of a neural network model. For example, "linear" refers to the Linear Activation Function where $\phi_k = 1$ (as previously described in Section 4.1 Modeling Approach).

Algorithm: The type of error minimization or gradient descent algorithm used to calculate and update parameters or weights to train the model. The following types of algorithms were used: glm-logistic, backprop, rprop–, and rprop+. glm-logistic refers to Generalized Linear Model Logistic Regression, backprop refers to standard/vanilla (no frills) Backpropagation, rprop– and rprop+ refer to Resilient Backpropagation without (–) and with (+) Weight Backtracking.[18]

Steps: The number of full batch training iterations required for a neural network model to converge with an error gradient less than the Threshold setting. (Logistic Regression Model "Steps" is number of fits to GLM.)

Threshold: A numeric value setting that specifies the stopping criteria for training a neural network model, when the error gradient falls below the set value.

Error: The error of the last batch trained when the threshold was reached, where the error is calculated as previously described in Equation 7 with a Half of the Sum of Squared Errors (HSSE) function iterated over all N samples of the training batch.

MCC: Matthews Correlation Coefficient (MCC) is a robust measure of the true and false positives and the true and false negatives in binary classification. MCC remains valid even when the classes are of very different sizes. MCC returns a value between -1 and +1, where a coefficient of +1 indicates a positive correlation and perfect agreement between observations & predictions, a coefficient of 0 indicates no correlation and predictions no better than random, and a coefficient of -1 indicates a negative correlation and total disagreement between observations & predictions. MCC is calculated from the *confusion matrix* by the following formula:[19]

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

where TP is the number of True Positives, TN is the number of True Negatives, FP is the number of False Positives, and FN is the number of False Negatives.

Accuracy: Combined proportion of True Positives & True Negatives correctly classified by the method. Calculated from the *confusion matrix*, where Accuracy = $(TP + TN)/(TP + TN + FP + FN)$.[20]

Acc.pValue: Probability of getting, by chance, the reported Accuracy when it is no greater than the naive classifier which always predicts the most common class in the data, i.e., the No Information Rate (NIR) = the proportion of the most frequent class.[21] For the Test Data in this project, NIR = 165/(165+138) = 0.54

Sensitivity: Proportion of True Positives correctly classified by the method. Calculated from the *confusion matrix*, where Sensitivity = $TP/(TP + FN)$. Also known as True Positive Rate (TPR) or Recall.[22]

Specificity: Proportion of True Negatives correctly classified by the method. Calculated from the *confusion matrix*, where Specificity = $TN/(TN + FP)$. Also known as True Negative Rate (TNR) or Selectivity.[23]

BA: Balanced Accuracy (BA) is the average of the Sensitivity (TPR) and the Specificity (TNR), where the BA = (TPR + TNR)/2.[24]

5.2 Model Performance Details

In all models, the Positive Class is the Heart Disease Patient with a Target Value of 1, and conversely, the Negative Class is the Healthy Patient with a Target Value of 0. The Specificity was 100% for all the FNN

[18] See Footnote #14
[19] See MCC in Machine Learning section at https://en.wikipedia.org/wiki/Phi_coefficient
[20] See Confusion Matrix at https://en.wikipedia.org/wiki/Confusion_matrix
[21] See p. 26 in confusionMatrix pp.24-27 of Package 'caret' at https://cran.r-project.org/web/packages/caret/caret.pdf
[22] See Footnote #20
[23] See Footnote #20
[24] See Footnote #20

models regardless of Accuracy, etc., because there were no False Positives (no healthy patients misclassified as heart disease patients) in any of the NN models. Thus the Sensitivity confirms that all accuracy problems for the NN models were the misclassification of only one or two heart disease patients (positives) as healthy patients (false negatives) in the test dataset.

The parameters tested in Table 4 were the general type of model (GLM or NN), the type of training algorithm, numbers of layers & nodes, different seeds & start weights, and threshold levels. What do performance metrics of the model parameters tell us:

- **fit_logistic**: Logistic Regression was used as a performance reference model and achieved only 85.81% Accuracy, 88.48% Sensitivity, 82.61% Specificity, and 85.55% Balance Accuracy. Which means that the model was about 6% better at correctly predicting heart disease patients than at predicting healthy patients, but that overall it was at best only 86% accurate. Even the worst performing FNN model, nnTrained.2, had better accuracy (99.34%) and performance than the Logistic Regression reference model (85.81%).

- **nnTrained.0a, nnTrained.0b, nnTrained.0**: The FNN models with 16,8,4,2 hidden layer/nodes architecture showed that the standard backpropagation algorithm took more than 80 times longer to train than the much faster RPROP– or RPROP+ resilient backpropagation algorithms. Not surprisingly, the trained weights of the slower standard backpropagation algorithm were generally smaller than the trained weights of the faster RPROP– or RPROP+ algorithms (see Figures 12, 13 & 14 in Section 9.2 Appendix B).

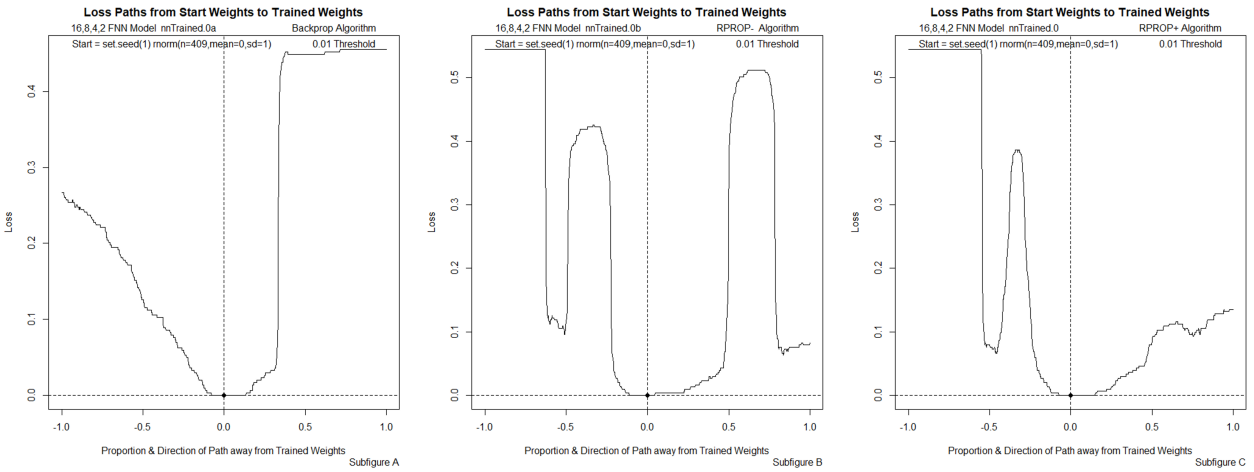

Figure 8: **Loss Paths to Trained Weights for 16,8,4,2 FNN Models based on Different Algorithms**
All models used Startweights = set.seed(1) rnorm(n=409,mean=0,sd=1) & Threshold=0.01
Subfigure A: nnTrained.0a Algorithm = Backpropagation
Subfigure B: nnTrained.0b Algorithm = RPROP–
Subfigure C: nnTrained.0 Algorithm = RPROP+

- Figure 8 shows that in the relatively large 16,8,4,2 FNN models, the Backpropagation algorithm has a simple error landscape but takes much longer to train, whereas the RPROP– and RPROP+ algorithms have more complex convoluted error landscapes, and yet still train very fast due to the much greater efficiency of the Resilient Backpropagation algorithms' handling of error gradient information.

- **nnTrained.3**: Downsizing from (nnTrained.0) a 16,8,4,2 to a 14,7,3 FNN remained at 100% Test Accuracy but took only about 0.59 times the training steps that ended with about 20% more Training Error, compared to the 16,8,4,2 model.

- **nnTrained.4**: Whereas, downsizing from a 16,8,4,2 to a 13,7,3 FNN decreased Test Accuracy to 99.67% and took about 9.8 times the training steps that ended with about 5% more Training Error, compared to the 16,8,4,2 model. Furthermore, Sensitivity decreased to 99.39% while Specificity remained at 100% due to misclassification of only one heart disease patient as a healthy patient (i.e., a false negative), which also explains the 99.67% Test Accuracy. Thus, the loss of one more node than the 100% accurate 14,7,3 downsize, resulted in over-training of the 13,7,3 model, which caused the loss of generalization to one heart disease patient.

- **nnTrained.4a**: In contrast, downsizing from a 16,8,4,2 to a 13,7,3 FNN along with using a different set.seed value of 4 (instead of seed = 1 as in the previous 13,7,3 NN model) kept Test Accuracy at 100% and took only about 1.24 times the training steps that ended with about 87% less Training Error, compared to the 16,8,4,2 model. Thus, different initial weight values can have a significant effect on the subsequent FNN training efficiency and success.

- **nnTrained.5**: In further contrast, downsizing from a 16,8,4,2 to a 13,6,3 FNN, but continuing to use the same set.seed value (of 1), kept Test Accuracy at 100% but took only about 0.46 times the training steps that ended with about 89% more Training Error, compared to the 16,8,4,2 model. Thus, the loss of 8 nodes resulted in less training that ended with more training error, but no loss of generalization that kept Test Accuracy at 100%. Furthermore, comparing the downsizing from a 16,8,4,2 to a 13,7,3 FNN (nnTrained.4), which lost 7 nodes and lost performance, with the downsizing from a 16,8,4,2 to a 13,6,3 FNN (nnTrained.5), which lost 8 nodes and kept 100% performance, had completely opposite outcomes. This counterintuitive performance by the nnTrained.5 model may be explained by a shift in the assignment of the seed-1 initial weights by one node (13,7,3 versus 13,6,3) in the second and third layers, which seems to be another example of the profound effect of the initial weights.

- **nnTrained.1**: Downsizing from a 16,8,4,2 to a 12,6,3 FNN remained at 100% Test Accuracy and took only about 0.86 times the training steps that ended with about 16% less Training Error, compared to the 16,8,4,2 model.

- **nnTrained.2**: Whereas, downsizing from a 16,8,4,2 to a 10,5,3 FNN decreased Test Accuracy to 99.34% and took about 1.56 times the training steps that ended with about 266% more Training Error, compared to the 16,8,4,2 model. Furthermore, Sensitivity decreased to 98.79% while Specificity remained at 100% due to misclassification of two heart disease patients as healthy patients (i.e., false negatives), which also explains the 99.34% Test Accuracy. Thus with all the other model factors unchanged, the change to 12 less nodes (-40%) resulted in a worse model with significantly more training error (+226%) and loss of 100% test accuracy by -0.66%.

- **nnTrained.2h**: Decreasing the error gradient Threshold an order of magnitude (from 0.01 to 0.001) in the 10,5,3 FNN did not improve Test Accuracy which remained at 99.34%, and took about 3.6 times the training steps to meet the lower 0.001 threshold but ended with about 86% less Training Error, compared to the previous 10,5,3 FNN model that had a 0.01 threshold (nnTrained.2). Note that the nnTrained.2h model used the default set.seed value of 1 so that it could be directly compared with the nnTrained.2 model which used a set.seed value of 1 by default. In summary, it seems that lowering the training threshold in this case had a net effect of over training because the training error greatly improved while the test accuracy remained deficient.

- **nnTrained.2a**: In contrast, downsizing from a 16,8,4,2 to a 10,5,3 FNN along with using a different set.seed value of 4 (instead of seed = 1 as in the nnTrained.2 model) decreased Test Accuracy to only 99.67% and took about 6.3 times the training steps that ended with about 434% more Training Error, compared to the 16,8,4,2 model. Furthermore, Sensitivity decreased to only 99.39% while Specificity remained at 100% due to misclassification of only one heart disease patient as a healthy patient (i.e., a false negative), which also explains the 99.67% Test Accuracy. Also note that the seed = 4 for the initial weights was the only difference in the nnTrained.2a model compared to the seed = 1 of the nnTrained.2 model which misclassified two heart disease patients – demonstrating that initial weights affect accuracy of an FNN model, this time by improving generalization of the model.

- **nnTrained.2i**: Keeping the set.seed value of 4, but decreasing the error gradient Threshold an order of magnitude (from 0.01 to 0.001) in the 10,5,3 FNN did not improve Test Accuracy which remained at 99.67%, and took about 8.6 times the training steps to meet the lower 0.001 threshold which ended with about 94% less Training Error, compared to the previous 10,5,3 FNN model that had a 0.01 threshold and a set.seed value of 4 (nnTrained.2a). In this case, it seems that lowering the training threshold had, yet again, a net effect of over training because the training error greatly improved while the test accuracy remained deficient.

- **nnTrained.2b**: Alternatively, downsizing from a 16,8,4,2 to a 10,5,3 FNN along with using a vector of initial weights (startweights10.5.3, described below) also decreased Test Accuracy to only 99.67%, which only took about 1.02 times the training steps and ended with about 60% less Training Error, compared to the 16,8,4,2 model. Furthermore, Sensitivity decreased to only 99.39% while Specificity remained at 100% due to misclassification of only one heart disease patient as a healthy patient (i.e., a false negative), which also explains the 99.67% Test Accuracy. Also note that the startweights10.5.3 vector was the only difference in the nnTrained.2b model compared to the seed = 1 of the nnTrained.2 model which misclassified two heart disease patients – again demonstrating that initial weights affect accuracy of an FNN model, in this case by again improving generalization of the model.

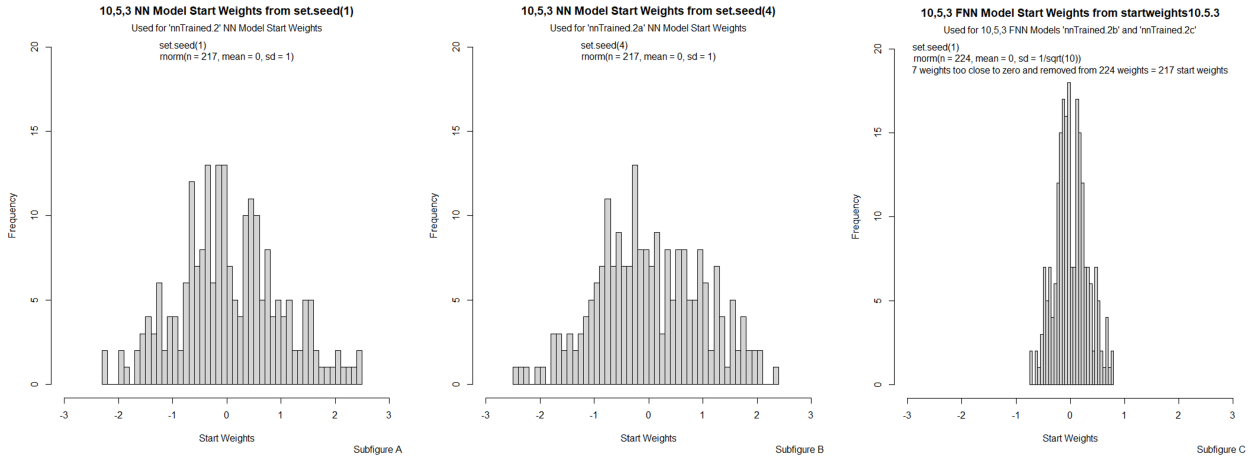

Figure 9: **Start Weight Distributions for 10,5,3 FNN Models based on set.seed & standard deviation**
Subfigure A: set.seed(1) rnorm(n = 217, mean = 0, sd = 1) used for nnTrained.2 & nnTrained.2h
Subfigure B: set.seed(4) rnorm(n = 217, mean = 0, sd = 1) used for nnTrained.2a & nnTrained.2i
Subfigure C: set.seed(1) rnorm(n = 224, mean = 0, sd = 1/sqrt(10)) where 7 weights were too close to zero and were removed, which resulted in 217 weights dubbed as startweights10.5.3 that were used for 10,5,3 FNN models nnTrained.2b and nnTrained.2c

- **startweights10.5.3** was created specifically for further development of the 10,5,3 model which needs 217 values to initialize the start weights of the 217 connections in the model. The set.seed(1) function was used in tandem with the rnorm() function in R to randomly select values from a normal distribution with mean = 0 and standard deviation = $(1/\sqrt{10})$, where 10 was selected as a representative number of connections feeding into a node of the 10,5,3 FNN model.[25] As previously described in **Seed_StartWeights**, *neuralnet uses random initialization of the rnorm function to create the initial weights drawn from a normal distribution with mean = 0 and standard deviation = 1*, and furthermore, the random seed that the neuralnet function uses with the rnorm function can be controlled by running set.seed immediately prior to executing neuralnet. Therefore, *the key difference* between the

[25]Start weights drawn from a normal distribution with mean = 0 and standard deviation = $(1/\sqrt{10})$ ensures that the standard deviation of the sum of the weights into a node, with 10 input connections, is approximately 1. See p. 20 of LeCun et al. (1998) and Appendix A for more details.

first 10,5,3 FNN model that used seed = 1 (nnTrained.2) and the 10.5.3 FNN model that first used startweights10.5.3 (nnTrained.2b) is *the value of the standard deviation used in the rnorm function* that drew the weights from normal distributions centered at mean = 0. See Figure 9 for histograms showing the start weight distributions used in the 10,5,3 NN models. See Section 9.1 Appendix A for more details on creating the weights in 'startweights10.5.3' and a complete list of the start weights in matrix form, as well as complete lists of the seed 1 and seed 4 start weights, in matrix form, generated by the neuralnet function for the respective 10,5,3 FNN models.

- **nnTrained.2c**: Keeping the startweights10.5.3, but decreasing the error gradient Threshold an order of magnitude (from 0.01 to 0.001) in the 10,5,3 FNN improved Test Accuracy to 100% in model nnTrained.2c, which took about 1.9 times the training steps to meet the lower 0.001 Threshold and achieved it with 65% less Training Error, compared to the previous 10,5,3 FNN model nnTrained.2b, which had 99.67% Test Accuracy with one false negative heart disease patient. In this case, more training by lowering the threshold *improved both training error and test accuracy* in the 10,5,3 FNN model with the *properly conditioned start weights* (startweights10.5.3).

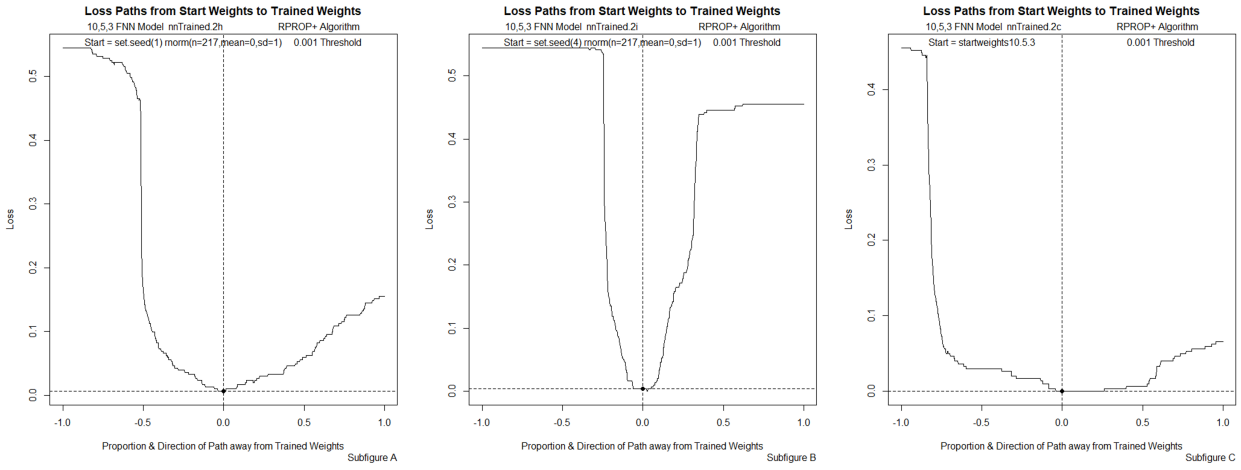

Figure 10: **Loss Paths to Trained Weights for 10,5,3 FNN Models based on Different Start Weights**
Subfigure A: nnTrained.2h Start Weights = set.seed(1) rnorm(n = 217, mean = 0, sd = 1)
Subfigure B: nnTrained.2i Start Weights = set.seed(4) rnorm(n = 217, mean = 0, sd = 1)
Subfigure C: nnTrained.2c Start Weights = startweights10.5.3 (See Figure 9 for more details.)

- Figure 10 shows that with smaller 10,5,3 FNN models, the RPROP+ algorithms have simple error landscapes (with no convolutions). In contrast, Figure 8 shows that in larger 16,8,4,2 FNN models, the RPROP– and RPROP+ algorithms have more complex and convoluted error landscapes. This suggests that the 16,8,4,2 FNN model over-represents the features of the data and may be more susceptible to over training. Furthermore, the more wide open valley in the loss landscape of Subfigure C in Figure 10 suggests that the 10,5,3 FNN model with properly conditioned start weights (startweights10.5.3) generalizes the best to the features of the data.

Best Model: nnTrained.2c with 10,5,3 hidden layers and startweights10.5.3 trained with the RPROP+ algorithm down to a gradient threshold of less than 0.001. This was chosen over the more easily trained FNN models with more nodes, precisely because the 10,5,3 model has fewer nodes and therefore should generalize better, because of less resources available for over training on noise in the data. Figure 15 and Table 9 in Appendixes B.4 and B.5 shows the complete map and weights of the nnTrained.2c model.

6 Conclusions

The best model, 10,5,3 FNN nnTrained.2c, confirmed that not only is conditioning of the input data important, but so is conditioning of the start weights. Weights that are too large get transformed into very shallow gradients at the asymptotic ends of the sigmoid activation function. Weights that are too close to zero also get transformed into very shallow gradients. Per the guidelines of LeCun et al. (1998), start weights that were randomly drawn from a normal distribution with zero mean and standard deviation of $m^{-1/2}$, where m is the number of connections feeding into a node, worked very well in this project as confirmed by the best model. Furthermore the 10,5,3 FNN nnTrained.2c model has fewer nodes and therefore generalizes better, because of less resources available for over training on noise in the data. This appears to be confirmed by the more wide open valley of its simple loss landscape, as well as by its 100% Test Accuracy and very low 0.00074 Training Error.

In head-to-head testing, the RPROP+ algorithm converged more than 80 times faster than standard backpropagation, with 100% Test Accuracy. The RPROP+ algorithm achieves this by individual adaptation of the weight-updates for each node according to the temporal behavior of the error function sign. This is an efficient and transparent adaptation process, which helps to overcome inherent disadvantages of pure gradient descent backpropagation (GD-BP). In addition to faster convergence, RPROP+ has several advantages over standard GD-BP, such as not affected by flat-regions and saddle-points, and has better avoidance of local minima and oscillations. Furthermore, there are several more advantages to using RPROP+ over traditional backpropagation algorithms:

- RPROP+ is more robust to noisy data. This is because RPROP+ adjusts the weights based on the sign of the gradient, rather than the actual value of the gradient, which makes RPROP less affected by noise in the data and very suitable to numerical estimation of the gradient [Igel & Hüsken 2003].
- Likewise, RPROP+ is less sensitive to the initial values of the weights, because RPROP+ adjusts the weights based on the sign of the gradient, rather than the actual value of the gradient. Whereas, traditional backpropagation algorithms use the actual value of the gradient which is highly dependent on the initial values of the weights [Igel & Hüsken 2003].
- Furthermore, RPROP+ is very robust with respect to its internal parameter settings, which makes their optimization relatively easy (if not the default values) [Igel & Hüsken 2003].
- RPROP+ is more computationally efficient than conjugate gradient algorithms. This is because RPROP+ does not require the computation of the second-order derivatives of the error function, which can be computationally expensive. Instead, RPROP+ only requires the computation of the first-order derivatives of the error function, which are relatively fast to compute [Riedmiller 1994-a]. Furthermore, because RPROP+ is a first-order method, "the time and space complexity scales only linearly with the number of parameters to be optimized" [Igel & Hüsken 2003].
- Finally, RPROP+ is not affected by vanishing gradients. Deep neural networks have several hidden layers, which can cause vanishing gradients toward the input layer due to slope limitations from the sigmoidal activation function at each layer. RPROP+ adjusts learning rates for each weight in the network based on the sign of the gradient, which is independent of the magnitude of the slopes. Thus, weight updates and learning remains consistent across the entire network, which makes it much better for training deep neural networks than traditional backpropagation algorithms [Riedmiller 1994-b].

7 Limitations

Rprop is limited to training by full batch size epochs, which is not practical for extremely large datasets. Rprop can accumulate very large weight values if the gradients are large, which is not a problem when working with the full batch. Large weight accumulations are a problem if trying to use mini-batches because the batches will violate the central limit theorem and can't properly average out the gradients. The key to stochastic gradient descent is a small enough learning rate that allows gradients to average out over successive mini-batches.[26]

Thus, a related adaptive GD-BP algorithm called RMSprop (Root Mean Square propagation) addresses the Rprop problem by keeping a moving average of the squared gradients for each weight, which is used to control the magnitude of the weight's update by dividing the new gradient of each weight by the square root of its mean square moving average. This mean square moving average will be similar for adjacent mini-batches, which allows the mini-batches to conform to the central limit theorem.[27]

8 Future Work

The activation function of the output layer has a big impact on the error results and subsequent weight corrections, and only a minor impact on the non-linear modeling complexity of the model. Thus it seems that better results should be achieved using a linear activation function in the output layer. Study the effects of a logistic versus a linear activation function in the output node of binary classification FNN models to confirm or refute this hypothesis.

Further study on initializing/conditioning the start weights of a FNN, particularly for use with logistic activation functions which has a mean = 0.5 and range 0 to 1. Guidance from LeCun et al. (1998) suggests that there should be no negative values in the start weights for logistic activation functions. However, all start weights and final trained weights for the models in this project had equal proportions of positive and negative weights. Furthermore, preliminary investigation showed very poor training results when model start weights were drawn from a normal distribution with mean = 0.5 and standard deviation = $1/\sqrt{m}$.

[26] Vitaly Bushaev (2018) "Understanding RMSprop — faster neural network learning" https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a

[27] Geoffrey Hinton (2012) "Neural Networks for Machine Learning - Lectures 6a-6e" https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

9 Appendixes

9.1 Appendix A

Appendix A explains the rationale and details on creating the initial weights 'startweights10.5.3' for the 10,5,3 FNN model. Furthermore, Appendix A.1 shows the complete list of the 'startweights10.5.3' weights in matrix form, which was used first in the 10,5,3 FNN model nnTrained.2b (which had Threshold = 0.01), and then in another 10,5,3 FNN model nnTrained.2c (which had a lower Threshold = 0.001). In addition, Appendix A.2 shows the complete lists of the seed-1 start weights, in matrix form, generated by the neuralnet function for the 10,5,3 FNN models nnTrained.2 (which had Threshold = 0.01) and nnTrained.2h (which had a lower Threshold = 0.001). Furthermore, Appendix A.3 shows the complete lists of the seed-4 start weights, in matrix form, generated by the neuralnet function for the 10,5,3 FNN models nnTrained.2a (which had Threshold = 0.01) and nnTrained.2i (which had a lower Threshold = 0.001). Finally, Figure 11 in Appendix A.4 shows histograms of the three start weight distributions used in the 10,5,3 FNN models, and is an expanded view of the same Figure 9 displayed in the Results & Discussion section 5.

The rationale for creating the initial weights 'startweights10.5.3' for the 10,5,3 FNN model is best explained in the words of LeCun et al. (1998):

> "The starting values of the weights can have a significant effect on the training process. Weights should be chosen randomly but in such a way that the sigmoid is primarily activated in its linear region. If weights are all very large then the sigmoid will saturate resulting in small gradients that make learning slow. If weights are very small then gradients will also be very small. Intermediate weights that range over the sigmoid's linear region have the advantage that (1) the gradients are large enough that learning can proceed and (2) the network will learn the linear part of the mapping before the more difficult nonlinear part."[p. 20]

Furthermore, LeCun et al. (1998) explain that it is assumed that the training set has been normalized (as described in this report, see Section 2.1 Data Conditioning) and a sigmoid activation function is being used. Then the initializing weights should be randomly drawn from a distribution (ideally uniform) with a zero mean and standard deviation of $m^{-1/2}$, where m is the number of connections feeding into a node.

Table 5: **Calculation of Start Weights for 10,5,3 FNN Model – The startweights10.5.3 Vector:**

```
Based on LeCun et al. (1998), use rnorm function with mean = 0 and standard deviation = 1/sqrt(number of connections to node)

1/sqrt(13)     # 13 inputs feeding into each node in 1st hidden layer.
[1] 0.2773501

1/sqrt(10)     # 10 connections feeding into each node in 2nd hidden layer.
[1] 0.3162278  # Use this value for start weights as representative of the 4 levels of connections            <------------
               # into the nodes of the 3 hidden layers and the output node.
1/sqrt(5)      # 5 connections feeding into each node in 3rd hidden layer.
[1] 0.4472136

1/sqrt(3)      # 3 connections feeding into the output node.
[1] 0.5773503

Need 217 start weights for 10.5.3 NN model, and weights that are not too close to zero,
i.e., absolute value of weight >= 0.0100000000

set.seed(1)
startweights10.5.3 <- rnorm(n = 217, mean = 0, sd = 1/sqrt(10))
sum(abs(startweights10.5.3)<0.0100000000)
[1] 6          # 6 weights are too small

set.seed(1)    # add 6 to 217 = 223 and rerun rnorm
startweights10.5.3 <- rnorm(n = 223, mean = 0, sd = 1/sqrt(10))
sum(abs(startweights10.5.3)<0.0100000000)
[1] 7          # 7 weights are too small

set.seed(1)    # add 7 to 217 = 224 and rerun rnorm                                                           <------------
startweights10.5.3 <- rnorm(n = 224, mean = 0, sd = 1/sqrt(10))
sum(abs(startweights10.5.3)<0.0100000000)
[1] 7          # 7 out of 224 weights are too small, so then select for the 217 weights that are NOT too small:

startweights10.5.3 <- startweights10.5.3[abs(startweights10.5.3)>=0.0100000000]
startweights10.5.3                                                          # Flat Vector Form of startweights10.5.3
  [1] -0.19810209  0.05807312 -0.26424897  0.50447208  0.10419951 -0.25945488  0.15413860  0.23347877  0.18207805 -0.09657229
 [11]  0.47806718  0.12327926 -0.19645352 -0.70034960  0.35573439 -0.01420925  0.29846722  0.25969294  0.18780809  0.29060616
 [21]  0.24733322  0.02357952 -0.62908824  0.19600611 -0.01774947 -0.04926687 -0.46509274 -0.15120432  0.13216473  0.42965220
 [31] -0.03250433  0.12259253 -0.01701465 -0.43546447 -0.13123280 -0.12468543 -0.01875654  0.34785857  0.24133736 -0.05202693
 [41] -0.08012000  0.22039917  0.17603236 -0.21780367 -0.22372961  0.11529094  0.24303145 -0.03552699  0.27863073  0.12589213
 [51] -0.19353974  0.10787152 -0.35713597  0.45316188  0.62625744 -0.11612563 -0.33018436  0.18016117 -0.04270802  0.75945822
 [61] -0.01240878  0.21811474 -0.23504363  0.05970137 -0.57077803  0.46344914  0.04846296  0.68704013  0.15036932 -0.22450477
 [71]  0.19312863 -0.29538761 -0.39643369  0.09216339 -0.14018120  0.02350879 -0.18642289 -0.17982884 -0.04274723  0.37254382
 [81] -0.48179413  0.18782228  0.10528815  0.33618169 -0.09619140  0.11701022  0.08446405 -0.17155990  0.38196134  0.36695153
 [91]  0.22142700  0.50180080  0.17660891 -0.40369390 -0.18128244 -0.38725651 -0.14970243 -0.19617717  0.01331821 -0.28805872
[101]  0.04997309 -0.20699784  0.55886531  0.22664280  0.28782236  0.12149008  0.53195078 -0.20103752 -0.14598488  0.45292741
[111] -0.20576825 -0.06557955 -0.12421677 -0.10119063 -0.08826338  0.15627607 -0.05607682 -0.15999780  0.42470617 -0.06785597
[121] -0.05678076 -0.03168309  0.22536487 -0.02326311 -0.01190097 -0.21555997 -0.10254326  0.01902440 -0.18622479  0.16807385
[131] -0.48015837  0.09694211 -0.48586810 -0.09517701 -0.16705677 -0.20621048 -0.01799234 -0.60537360  0.37206831 -0.52651051
[141] -0.14658118 -0.35288492 -0.23742982  0.66002001 -0.40676394 -0.51880502  0.14236166 -0.10058205 -0.29389012 -0.47037625
[151] -0.34000566  0.31623687 -0.19646178 -0.43779421  0.59112160  0.13442854 -0.07546684  0.33472173  0.28031145 -0.19582185
[161]  0.69763085 -0.08064663 -0.45046476 -0.04566316  0.06562939  0.72984685  0.03345765  0.14451571 -0.02439790 -0.10562034
[171] -0.01098133  0.24907351  0.65625009  0.32489002  0.38197417 -0.38937865  0.31113510  0.06954633 -0.46398520  0.16476186
[181] -0.05020261  0.46314317 -0.24225640 -0.13604490 -0.29286154 -0.05600519  0.12712729 -0.23139909  0.26258705 -0.38202932
[191] -0.33140177  0.45573408 -0.32123917  0.13027784 -0.12050683  0.12946423  0.53406863  0.50172332 -0.10464223 -0.72265493
[201]  0.78982994  0.21094484  0.17118273  0.16131045 -0.05198020  0.13303533 -0.12656913 -0.43329778  0.31238189  0.48058557
[211] -0.09763234 -0.39632502  0.20309453 -0.01413827 -0.54809178 -0.19931847 -0.10782373
```

9.1.1 Appendix A.1

Table 6: **10,5,3 FNN Models nnTrained.2b & nnTrained.2c Start Weights Matrix created by neuralnet function from startweights10.5.3 Vector**

```
# startweights10.5.3 vector generated using seed = 1 with rnorm mean = 0 and std dev = 1/sqrt(10)
# where weights filtered out if too close to zero, i.e., absolute value of weight >= 0.0100000000

nnTrained.2b$startweights  # generated by neuralnet function using startweights argument = startweights10.5.3 vector
nnTrained.2c$startweights  # generated by neuralnet function using startweights argument = startweights10.5.3 vector
[[1]]
[[1]][[1]]
             [,1]        [,2]        [,3]        [,4]        [,5]        [,6]        [,7]        [,8]        [,9]       [,10]
 [1,] -0.19810209  0.35573439  0.13216473  0.17603236 -0.33018436  0.19312863 -0.09619140  0.01331821 -0.12421677 -0.10254326
 [2,]  0.05807312 -0.01420925  0.42965220 -0.21780367  0.18016117 -0.29538761  0.11701022 -0.28805872 -0.10119063  0.01902440
 [3,] -0.26424897  0.29846722 -0.03250433 -0.22372961 -0.04270802 -0.39643369  0.08446405  0.04997309 -0.08826338 -0.18622479
 [4,]  0.50447208  0.25969294  0.12259253  0.11529094  0.75945822  0.09216339 -0.17155990 -0.20699784  0.15627607  0.16807385
 [5,]  0.10419951  0.18780809 -0.01701465  0.24303145 -0.01240878 -0.14018120  0.38196134  0.55886531 -0.05607682 -0.48015837
 [6,] -0.25945488  0.29060616 -0.43546447 -0.03552699  0.21811474  0.02350879  0.36695153  0.22664280 -0.15999780  0.09694211
 [7,]  0.15413860  0.24733322 -0.13123280  0.27863073 -0.23504363 -0.18642289  0.22142700  0.28782236  0.42470617 -0.48586810
 [8,]  0.23347877  0.02357952 -0.12468543  0.12589213  0.05970137 -0.17982884  0.50180080  0.12149008 -0.06785597 -0.09517701
 [9,]  0.18207805 -0.62908824 -0.01875654 -0.19353974 -0.57077803 -0.04274723  0.17660891  0.53195078 -0.05678076 -0.16705677
[10,] -0.09657229  0.19600611  0.34785857  0.10787152  0.46344914  0.37254382 -0.40369390 -0.20103752 -0.03168309 -0.20621048
[11,]  0.47806718 -0.01774947  0.24133736 -0.35713597  0.04846296 -0.48179413 -0.18128244 -0.14598488  0.22536487 -0.01799234
[12,]  0.12327926 -0.04926687 -0.05202693  0.45316188  0.68704013  0.18782228 -0.38725651  0.45292741 -0.02326311 -0.60537360
[13,] -0.19645352 -0.46509274 -0.08012000  0.62625744  0.15036932  0.10528815 -0.14970243 -0.20576825 -0.01190097  0.37206831
[14,] -0.70034960 -0.15120432  0.22039917 -0.11612563 -0.22450477  0.33618169 -0.19617717 -0.06557955 -0.21555997 -0.52651051

[[1]][[2]]
            [,1]        [,2]        [,3]        [,4]        [,5]
 [1,] -0.1465812  0.31623687 -0.45046476  0.32489002 -0.29286154
 [2,] -0.3528849 -0.19646178 -0.04566316  0.38197417 -0.05600519
 [3,] -0.2374298 -0.43779421  0.06562939 -0.38937865  0.12712729
 [4,]  0.6600200  0.59112160  0.72984685  0.31113510 -0.23139909
 [5,] -0.4067639  0.13442854  0.03345765  0.06954633  0.26258705
 [6,] -0.5188050 -0.07546684  0.14451571 -0.46398520 -0.38202932
 [7,]  0.1423617  0.33472173 -0.02439790  0.16476186 -0.33140177
 [8,] -0.1005821  0.28031145 -0.10562034 -0.05020261  0.45573408
 [9,] -0.2938901 -0.19582185 -0.01098133  0.46314317 -0.32123917
[10,] -0.4703763  0.69763085  0.24907351 -0.24225640  0.13027784
[11,] -0.3400057 -0.08064663  0.65625009 -0.13604490 -0.12050683

[[1]][[3]]
            [,1]       [,2]        [,3]
[1,]  0.1294642  0.2109448 -0.43329778
[2,]  0.5340686  0.1711827  0.31238189
[3,]  0.5017233  0.1613104  0.48058557
[4,] -0.1046422 -0.0519802 -0.09763234
[5,] -0.7226549  0.1330353 -0.39632502
[6,]  0.7898299 -0.1265691  0.20309453

[[1]][[4]]
            [,1]
[1,] -0.01413827
[2,] -0.54809178
[3,] -0.19931847
[4,] -0.10782373
```

9.1.2 Appendix A.2

Table 7: **10,5,3 FNN Models nnTrained.2 & nnTrained.2h Start Weights Matrix created by neuralnet with rnorm function and set.seed(1)**

```
nnTrained.2$startweights  # generated by neuralnet using seed = 1 with rnorm mean = 0 and rnorm standard deviation = 1
nnTrained.2h$startweights  # generated by neuralnet using seed = 1 with rnorm mean = 0 and rnorm standard deviation = 1
[[1]]
[[1]][[1]]
            [,1]        [,2]        [,3]       [,4]        [,5]         [,6]       [,7]        [,8]       [,9]       [,10]
 [1,] -0.6264538  1.12493092 -0.47815006  0.6969634 -0.36722148  0.475509529  0.5939462 -1.22461261  1.4322822 -0.07356440
 [2,]  0.1836433 -0.04493361  0.41794156  0.5566632 -1.04413463 -0.709946431  0.3329504 -0.47340064 -0.6506964 -0.03763417
 [3,] -0.8356286 -0.01619026  1.35867955 -0.6887557  0.56971963  0.610726353  1.0630998 -0.62036668 -0.2073807 -0.68166048
 [4,]  1.5952808  0.94383621 -0.10278773 -0.7074952 -0.13505460 -0.934097632 -0.3041839  0.04211587 -0.3928079 -0.32427027
 [5,]  0.3295078  0.82122120  0.38767161  0.3645820  2.40161776 -1.253633400  0.3700188 -0.91092165 -0.3199929  0.06016044
 [6,] -0.8204684  0.59390132 -0.05380504  0.7685329 -0.03924000  0.291446236  0.2670988  0.15802877 -0.2791133 -0.58889449
 [7,]  0.4874291  0.91897737 -1.37705956 -0.1123462  0.68973936 -0.443291873 -0.5425200 -0.65458464  0.4941883  0.53149619
 [8,]  0.7383247  0.78213630 -0.41499456  0.8811077  0.02800216  0.001105352  1.2078678  1.76728727 -0.1773305 -1.51839408
 [9,]  0.5757814  0.07456498 -0.39428995  0.3981059 -0.74327321  0.074341324  1.1604026  0.71670748 -0.5059575  0.30655786
[10,] -0.3053884 -1.98935170 -0.05931340 -0.6120264  0.18879230 -0.589520946  0.7002136  0.91017423  1.3430388 -1.53644982
[11,]  1.5117812  0.61982575  1.10002537  0.3411197 -1.80495863 -0.568668733  1.5868335  0.38418536 -0.2145794 -0.30097613
[12,]  0.3898432 -0.05612874  0.76317575 -1.1293631  1.46555486 -0.135178615  0.5584864  1.68217608 -0.1795565 -0.52827990
[13,] -0.6212406 -0.15579551 -0.16452360  1.4330237  0.15325334  1.178086997 -1.2765922 -0.63573645 -0.1001907 -0.65209478
[14,] -2.2146999 -1.47075238 -0.25336168  1.9803999  2.17261167 -1.523566800 -0.5732654 -0.46164473  0.7126663 -0.05689678

[[1]][[2]]
             [,1]        [,2]       [,3]        [,4]       [,5]
 [1,] -1.91435943 -0.01855983  1.0584830 -0.07715294  0.5210227
 [2,]  1.17658331 -0.31806837  0.8864227 -0.33400084 -0.1587546
 [3,] -1.66497244 -0.92936215 -0.6192430 -0.03472603  1.4645873
 [4,] -0.46353040 -1.48746031  2.2061025  0.78763961 -0.7660820
 [5,] -1.11592011 -1.07519230 -0.2550270  2.07524501 -0.4302118
 [6,] -0.75081900  1.00002880 -1.4244947  1.02739244 -0.9261095
 [7,]  2.08716655 -0.62126669 -0.1443996  1.20790840 -0.1771040
 [8,]  0.01739562 -1.38442685  0.2075383 -1.23132342  0.4020118
 [9,] -1.28630053  1.86929062  2.3079784  0.98389557 -0.7317482
[10,] -1.64060553  0.42510038  0.1058024  0.21992480  0.8303732
[11,]  0.45018710 -0.23864710  0.4569988 -1.46725003 -1.2080828

[[1]][[3]]
           [,1]       [,2]        [,3]
[1,] -1.0479844  1.6888733  0.54132734
[2,]  1.4411577  1.5865884 -0.01339952
[3,] -1.0158475 -0.3309078  0.51010842
[4,]  0.4119747 -2.2852355 -0.16437583
[5,] -0.3810761  2.4976616  0.42069464
[6,]  0.4094018  0.6670662 -0.40024674

[[1]][[4]]
           [,1]
[1,] -1.3702079
[2,]  0.9878383
[3,]  1.5197450
[4,] -0.3087406
```

9.1.3 Appendix A.3

Table 8: **10,5,3 FNN Models nnTrained.2a & nnTrained.2i Start Weights Matrix created by neuralnet with rnorm function and set.seed(4)**

```
nnTrained.2a$startweights  # generated by neuralnet using seed = 4 with rnorm mean = 0 and rnorm standard deviation = 1
nnTrained.2i$startweights  # generated by neuralnet using seed = 4 with rnorm mean = 0 and rnorm standard deviation = 1
[[1]]
[[1]][[1]]
             [,1]        [,2]       [,3]        [,4]       [,5]        [,6]        [,7]        [,8]       [,9]       [,10]
 [1,]  0.21675486  0.03435191 -0.9280281  1.29251234 -0.5525072  0.15867690 -1.03273843  0.96847913  0.8652208 -0.9412566
 [2,] -0.54249257  0.16902677  1.2401808 -1.68804858  0.6959666 -0.48566507 -1.30652486 -0.27753563  0.3053288  0.3491856
 [3,]  0.89114465  1.16502684  0.1534642 -0.82099358 -0.1556640 -0.95890607 -0.83825241  0.68480194 -0.1140228 -0.5944187
 [4,]  0.59598058 -0.04420400  1.0519326 -0.86214614  1.3488982  0.18051729 -1.13065368 -0.11511351  0.4236522 -2.3822428
 [5,]  1.63561800 -0.10036844 -0.7542112  0.09884369 -1.0685231  0.72173428  0.36874818 -0.35647518 -0.7977097  1.0780190
 [6,]  0.68927544 -0.28344457 -1.4821891 -0.37565514  1.0644507 -0.36954048 -0.20180302 -0.10577161 -0.6041972  0.6682451
 [7,] -1.28124663  1.54081498  0.8611319  0.72390416 -1.3127218  0.23753831 -1.27765990  0.04488279  1.7150106 -0.9646257
 [8,] -0.21314452  0.16516902 -0.4045198 -1.79738202  2.0636947 -0.66592211 -0.79801248 -1.72617323 -0.7159483 -1.9752373
 [9,]  1.89653987  1.30762236 -0.2274054 -0.66374314  0.1313830 -0.79680751  0.15908242  1.55578702 -0.1332356 -0.5847739
[10,]  1.77686321  1.28825688  0.9340962 -0.62372649 -0.2316884 -0.05169693  0.61479763  0.77641269 -0.9997651  0.9692770
[11,]  0.56660450  0.59289694 -0.4658959 -0.07963243 -0.3973555  1.28692833  0.68794796 -1.09850751  1.8737601  0.5522923
[12,]  0.01571945 -0.28294368 -0.6375435  0.43562476  0.8894321 -0.21414966 -0.04705101 -1.72801975 -0.3373884 -0.0821555
[13,]  0.38305734  1.25588403  1.3437086  1.97090097  0.5261690 -0.57474546  2.33032168  0.42763822  0.9732703 -1.6767138
[14,] -0.04513712  0.90983915  0.1815354 -0.59675867 -0.1712732 -1.47072704 -0.57756599  0.74456465  0.9878279  1.2126074

[[1]][[2]]
            [,1]       [,2]        [,3]        [,4]          [,5]
 [1,]  1.0004999 -0.8161958 -0.93199903 -2.25793822  1.9005426699
 [2,]  0.7193290  1.5392933 -1.42789198  1.68260721 -0.7161791664
 [3,] -0.8443642  1.3745257  0.97576509  0.07229068  0.3804596689
 [4,]  0.6219854 -0.4832487 -1.54634119 -0.44002409  0.4408428474
 [5,] -0.7226138  0.5503500  0.01770348  0.62657339  0.2573258583
 [6,] -0.4494786 -0.8573657 -0.77471740 -0.79979606 -0.1794485371
 [7,] -1.1955061 -0.7069614 -0.22934229 -1.12798602 -0.6901276793
 [8,]  0.3904724 -2.0970775 -0.27438210 -1.02501605 -0.0004228025
 [9,] -0.5163766  1.0994368  1.79606378  0.07107173  0.5655808964
[10,]  0.9098690  0.3420341 -0.47811290  0.38171116 -1.2087470098
[11,]  0.8769847  0.4908295 -0.59476285 -1.62258832 -0.3461711560

[[1]][[3]]
           [,1]       [,2]       [,3]
[1,] -0.6501970 -1.5478003  0.5563283
[2,] -0.8895917 -0.3022460  1.1055339
[3,]  1.4770299  1.0392077  0.1664368
[4,] -1.1954751 -0.7678417 -0.2254623
[5,]  1.7504948  1.5246726 -0.2284123
[6,]  1.2147301 -2.4220873 -0.2531892

[[1]][[4]]
           [,1]
[1,]  2.0682733
[2,]  1.5831899
[3,] -1.0425907
[4,] -0.0083874
```

9.1.4 Appendix A.4

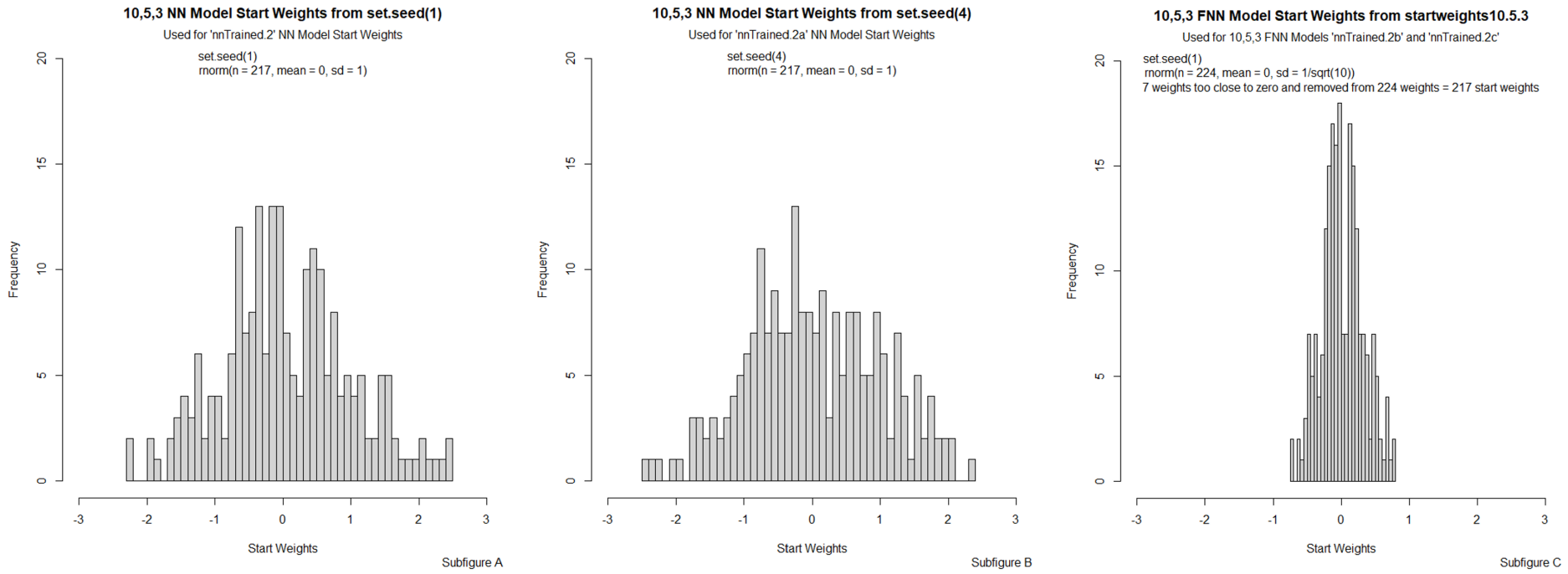

Figure 11: **Start Weight Distributions for 10,5,3 FNN Models based on set.seed and standard deviation**
Subfigure A: set.seed(1) rnorm(n = 217, mean = 0, sd = 1) used for nnTrained.2 & nnTrained.2h models
Subfigure B: set.seed(4) rnorm(n = 217, mean = 0, sd = 1) used for nnTrained.2a & nnTrained.2i models
Subfigure C: set.seed(1) rnorm(n = 224, mean = 0, sd = 1/sqrt(10)), where 7 weights were too close to zero and were removed, which resulted in 217 weights designated as startweights10.5.3 that were used for 10,5,3 FNN models nnTrained.2b and nnTrained.2c

9.2 Appendix B

9.2.1 Appendix B.1

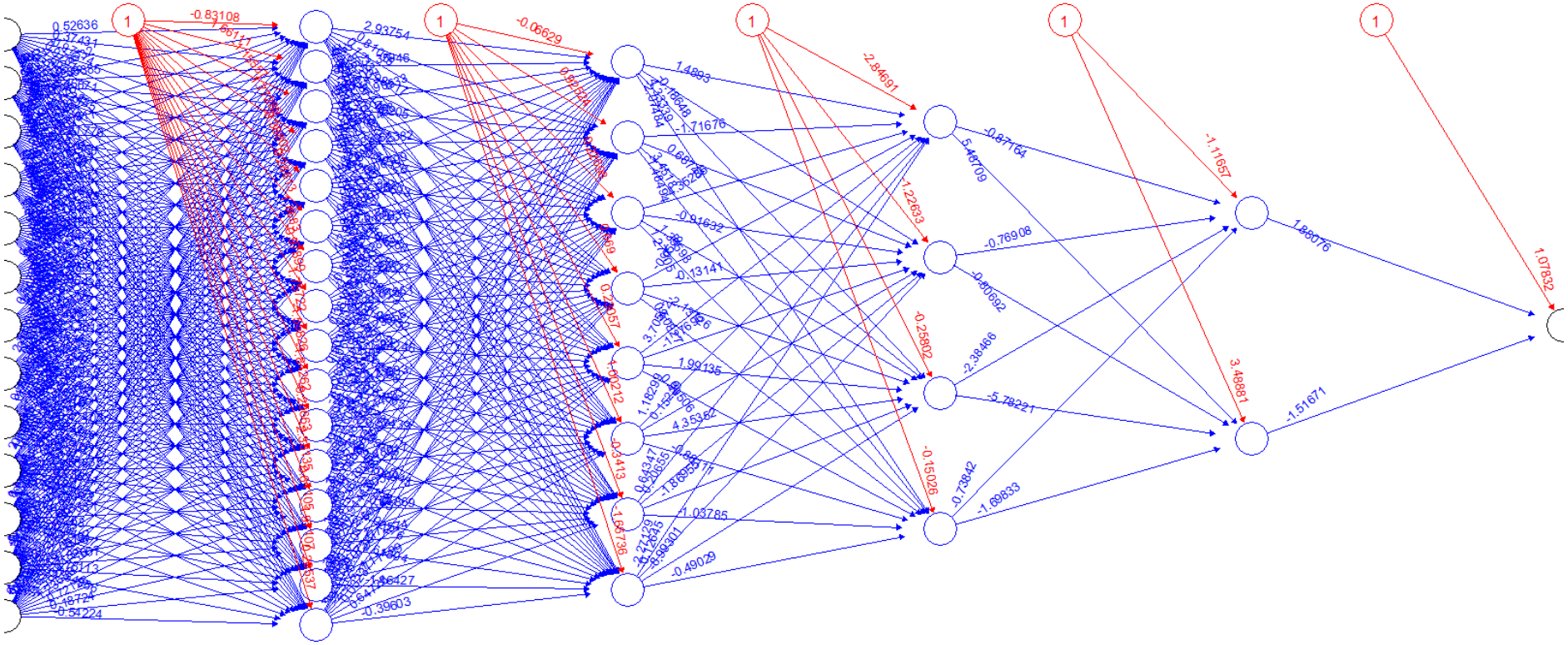

Figure 12: **Layers, Nodes, Trained Final Weights & Biases of Fully Connected Feedforward Neural Network Model: nnTrained.0a**

	Model	Seed	Hidden	Hid_Act	Out_Act	Algorithm	Steps	Threshold	Error	MCC	Accuracy	Acc.pValue	Sensitivity	Specificity	BA
nnTrained.0a		1	16,8,4,2	logistic	linear	backprop	107607	0.01	0.018320	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000

Bias Inputs = 1 (Circled)
Bias Weights = Red Connections
Node Weights = Blue Connections

9.2.2 Appendix B.2

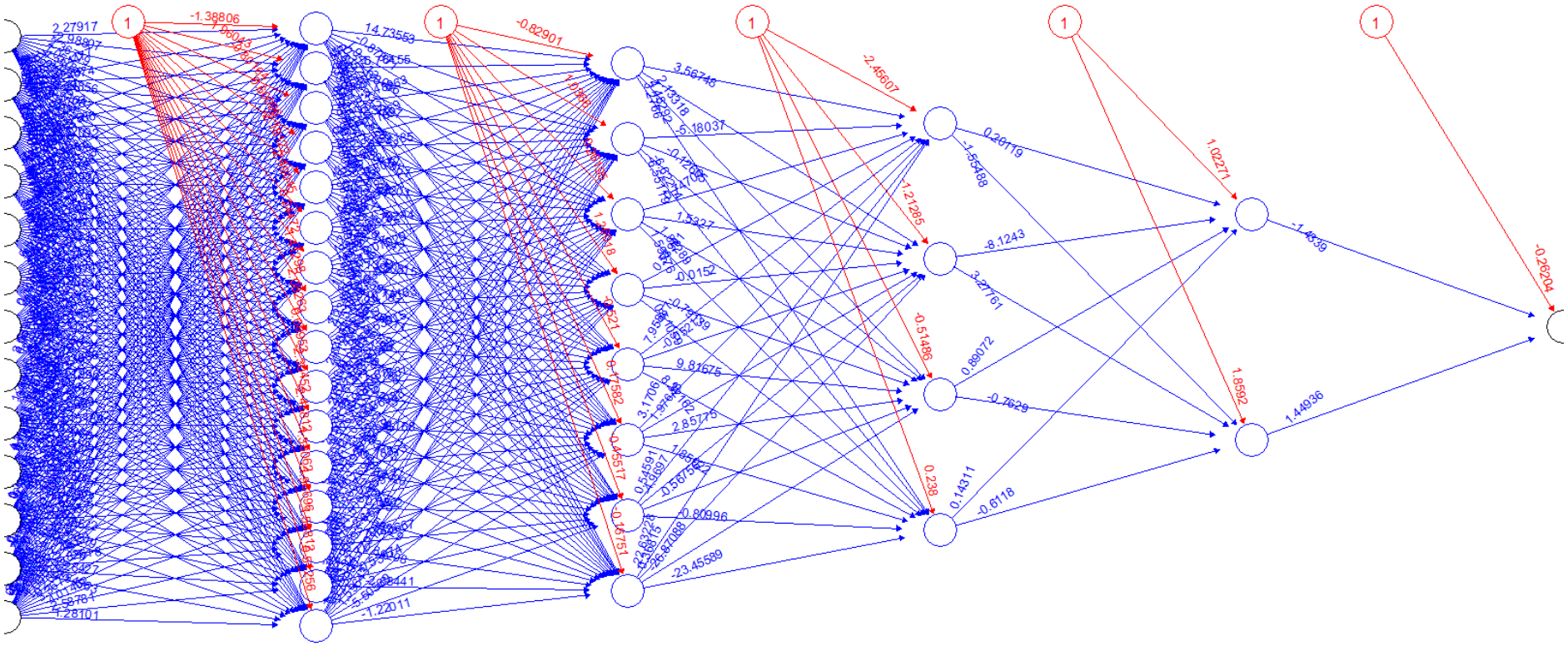

Figure 13: **Layers, Nodes, Trained Final Weights & Biases of Fully Connected Feedforward Neural Network Model: nnTrained.0b**

Model	Seed	Hidden	Hid_Act	Out_Act	Algorithm	Steps	Threshold	Error	MCC	Accuracy	Acc.pValue	Sensitivity	Specificity	BA
nnTrained.0b	1	16,8,4,2	logistic	linear	rprop-	1295	0.01	0.001910	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000

Bias Inputs = 1 (Circled)
Bias Weights = Red Connections
Node Weights = Blue Connections

9.2.3 Appendix B.3

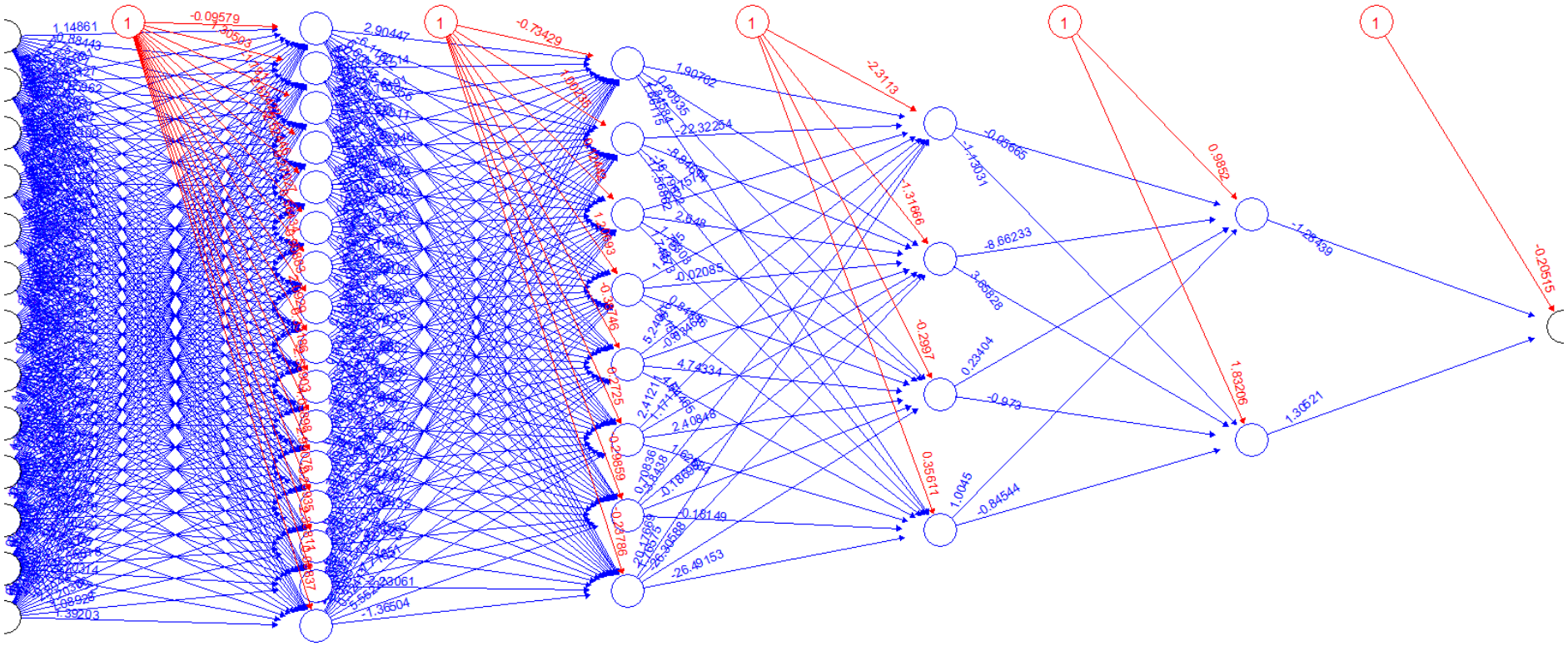

Figure 14: **Layers, Nodes, Trained Final Weights & Biases of Fully Connected Feedforward Neural Network Model: nnTrained.0**

Model	Seed	Hidden	Hid_Act	Out_Act	Algorithm	Steps	Threshold	Error	MCC	Accuracy	Acc.pValue	Sensitivity	Specificity	BA
nnTrained.0	1	16,8,4,2	logistic	linear	rprop+	1315	0.01	0.005360	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000

Bias Inputs = 1 (Circled)
Bias Weights = Red Connections
Node Weights = Blue Connections

9.2.4 Appendix B.4

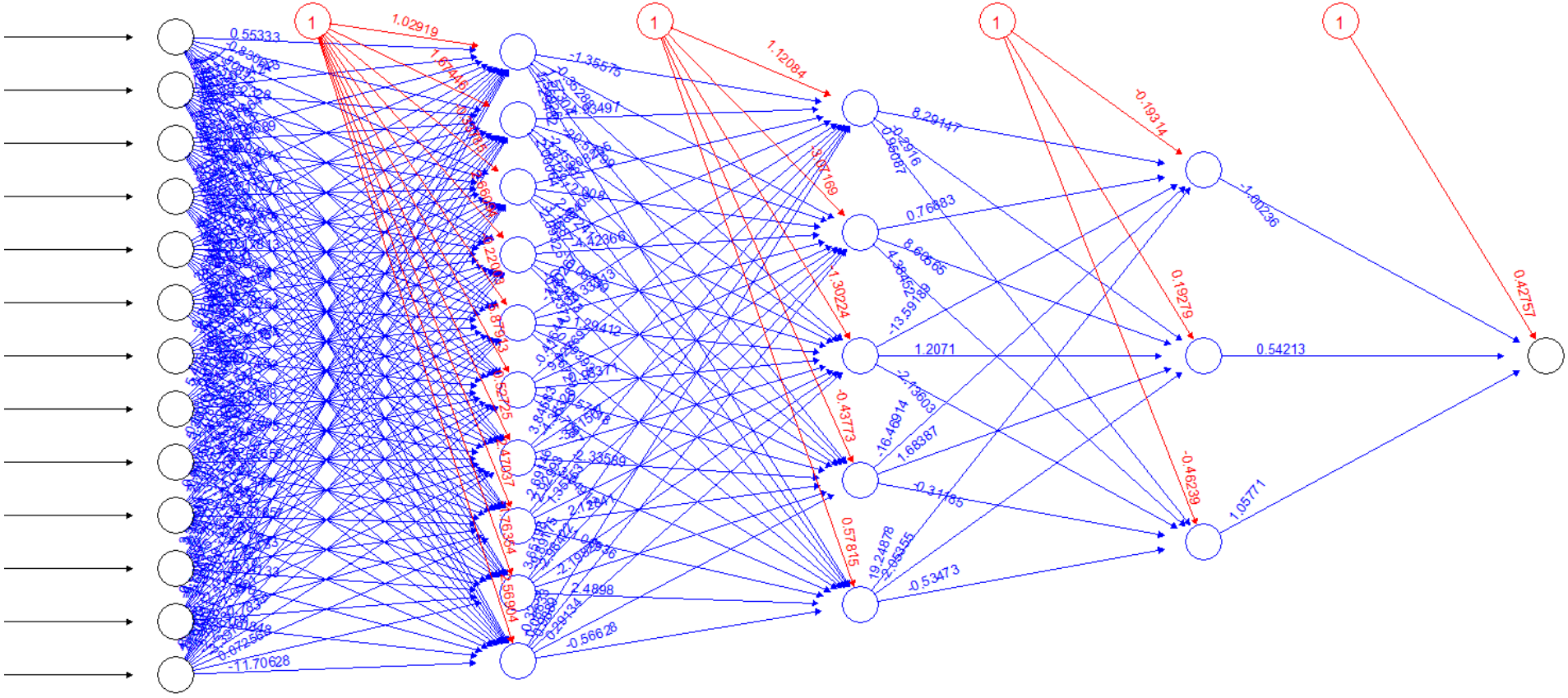

Figure 15: **Layers, Nodes, Trained Final Weights & Biases of Fully Connected Feedforward Neural Network Model: nnTrained.2c**

Model	Seed_StartWeights	Hidden	Hid_Act	Out_Act	Algorithm	Steps	Threshold	Error	MCC	Accuracy	Acc.pValue	Sensitivity	Specificity	BA
nnTrained.2c	startweights10.5.3	10,5,3	logistic	linear	rprop+	2557	0.001	0.00074	1.000	1.0000	2.2e-16	1.0000	1.0000	1.0000

Bias Inputs = 1 (Circled)
Bias Weights = Red Connections
Node Weights = Blue Connections

9.2.5 Appendix B.5

Table 9: **10,5,3 FNN Model nnTrained.2c – Final Model Trained Weights & Biases Matrix (Created by neuralnet with startweights10.5.3, RPROP+ and Threshold = 0.001)**

```
nnTrained.2c$weights  # First Row of Each Submatrix are the Biases to the Receiving Layer of Nodes (See Figure 15 for Node Map)
[[1]]                 # Each subsequent Row are the Weights from each Sending Node (top to bottom) to all the Receiving Nodes
[[1]][[1]]  # Sending (Rows) = Input Nodes       Receiving (Columns) = 1st Hidden Layer Nodes
            [,1]       [,2]        [,3]       [,4]       [,5]        [,6]       [,7]       [,8]        [,9]        [,10]
 [1,]  1.0291890  1.6744631 -0.33784811  2.6609420 -0.2200303   5.8791265  0.5272507  2.4703711 -1.76354317   2.56903566
 [2,]  0.5533283 -0.8306217  1.80812168  0.7536240  1.2358774   1.9125936  2.7899542  4.9849459  1.18102363  -0.22932231
 [3,] -0.3424295  1.2327978 -2.90799508  1.0787607 -2.1598595   0.4355095 -0.7950251 -1.6442900  0.24497098   2.02437005
 [4,]  1.8766446  1.0668927 -0.94016250  2.3424906  5.3788912   0.6906697  0.4388523 -0.1063527 -2.93621269   1.96699122
 [5,] -4.4053524  0.1149726 -1.11276875 -4.3487910 -2.6945742  -0.1220513 -0.5901387  3.6414669 -0.27582613  -4.80761884
 [6,]  2.8364228 -1.3059377 -0.73482642  0.1781274  0.2009560   3.2382630  1.4558434 -1.4232118 -0.17996187   5.14842142
 [7,]  3.6580388 -1.6631980  0.28616585  0.6946398  1.3895443  -5.4814624 -1.8209023 -2.9408910  1.08605535   3.38755135
 [8,] -0.2696267 -2.5292748 -1.96647304 -1.0914340 -0.4013389   4.0755187  1.1835213 -4.1522522 -3.03698618   0.22015648
 [9,]  5.4758443  4.6441503  0.04459092  6.1894602  4.5547276   4.5899845 -2.2728454  7.8950310  2.87121966   5.25403307
[10,] -0.1786079  5.3390273  1.77860871  2.4292404 -2.3834309   0.7245750 -0.5265571  1.9425031  1.18448093   0.43333627
[11,] -0.8877558 -0.5275597  2.77124759 -4.8935428 -3.0544084 -11.0499072 -0.7877233 -2.8185655  2.27084463  -2.90126448
[12,]  3.5281401  2.1644102  2.48526480  2.6777862  2.7827257   5.1386119 -1.3698173 -2.2543274  0.34133247   2.04028571
[13,] -0.7414821 -4.0952885 -6.76254260 -1.6407046 -2.3986805   3.1119169  2.4540945 -7.0800995  0.78300263   0.08479699
[14,] -8.3093044  1.2859014  2.23671879 -7.6709142 -7.9599125  -4.9688061 -2.2962953 -3.3976905  0.07256479 -11.70628335

[[1]][[2]]  # Sending (Rows) = 1st Hidden Layer Nodes       Receiving (Columns) = 2nd Hidden Layer Nodes
            [,1]         [,2]        [,3]       [,4]       [,5]
 [1,]  1.1208363  -3.07169485 -1.30224213 -0.4377251  0.5781482
 [2,] -1.3557457  -0.35287622  1.57304301  1.5592617 -1.2948220
 [3,]  4.9349668 -20.51798992 -2.45387455 -3.8129076  2.0006427
 [4,] -4.0820481   2.00800413  2.67240863  3.1686985 -2.0932518
 [5,] -1.6640940  -4.42366259  0.05738715  0.5441271 -0.2237250
 [6,] -1.4026693 -38.33312871  1.29412436  0.3939327 -1.1672937
 [7,] -0.4164355   9.82769355  0.98371409  0.5711845 -0.7086956
 [8,]  3.8458322  -4.36328460 -3.61507388 -2.3358856  3.0491157
 [9,] -2.6914620   2.92992632  1.35753316  2.1284101 -1.0693632
[10,]  3.6519808  -9.89175382 -2.98422221 -2.1982344  2.4898005
[11,] -0.3239046   0.08833452  0.98889661  0.2913403 -0.5662825

[[1]][[3]]  # Sending (Rows) = 2nd Hidden Layer Nodes       Receiving (Columns) = 3rd Hidden Layer Nodes
            [,1]       [,2]       [,3]
[1,]  -0.1931371  0.1927932 -0.4623860
[2,]   8.2914654 -0.2915983  0.9508704
[3,]   0.7688276  8.6056502  4.3845220
[4,] -13.5918895  1.2071036 -2.1360321
[5,] -16.4691428  1.6838683 -0.3118462
[6,]  19.2487794 -2.0535477 -0.5347250

[[1]][[4]]  # Sending (Rows) = 3rd Hidden Layer Nodes       Receiving (Column) = Output Node
          [,1]
[1,]  0.4275667
[2,] -1.0023582
[3,]  0.5421307
[4,]  1.0577133
```

10 References

Anonymous (2019) "GradVis: Visualization and Second Order Analysis of Optimization Surfaces during the Training of Deep Neural Networks" https://sc19.supercomputing.org/proceedings/workshops/workshop_files/ws_mlhpce111s1-file1.pdf

Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong. (2023). *Mathematics for Machine Learning.* Cambridge: Cambridge University Press, ©2020.
ISBN: 9781108455145
Free Online 2023-02-15 Draft: https://mml-book.github.io/book/mml-book.pdf

Haykin, Simon. (2009) *Neural Networks and Learning Machines — 3rd Edition*
Pearson Prentice Hall, ©2009
ISBN: 978-0-13-147139-9
https://dai.fmph.uniba.sk/courses/NN/haykin.neural-networks.3ed.2009.pdf

Igel, Christian, and Michael Hüsken. (2003) "Empirical Evaluation of the Improved Rprop Learning Algorithms."
Neurocomputing, Vol. 50, pp. 105-123
https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=3185398b36c0d2be6a18c0407ad57cfe0f9b8667
https://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.102.1273

Igel, Christian, Marc Toussaint, and Wan Weishui. (2005). "Rprop Using the Natural Gradient."
In: Mache, D.H., Szabados, J., de Bruin, M.G. (eds) *Trends and Applications in Constructive Approximation.* ISNM International Series of Numerical Mathematics, vol 151. Birkhäuser Basel. ©2005.
https://link.springer.com/chapter/10.1007/3-7643-7356-3_19
https://doi.org/10.1007/3-7643-7356-3_19

LeCun, Yann A., Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. (1998). "Efficient BackProp"
In: Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller (Editors). *Neural Networks: Tricks of the Trade, 2nd Edition,*
Lecture Notes in Computer Science 7700, pp. 9–48, Springer-Verlag Berlin Heidelberg ©2012.
https://link.springer.com/content/pdf/10.1007/978-3-642-35289-8_3?pdf=chapter%20toc
https://cseweb.ucsd.edu//classes/wi08/cse253/Handouts/lecun-98b.pdf

Riedmiller, Martin, and Heinrich Braun. (1993) "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm."
In *Proceedings of 1993 IEEE International Conference on Neural Networks (ICNN'93)*, Vol. 1, pp. 586-591
https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2016/pdfs/Rprop.pdf

Riedmiller, Martin. (1994-a) "Advanced Supervised Learning in Multi-Layer Perceptrons — From Backpropagation to Adaptive Learning Algorithms."
Computer Standards & Interfaces, Vol. 16, No. 3, pp. 265-278
https://staff.fmi.uvt.ro/~daniela.zaharie/dm2020/RO/Proiecte/Biblio/MLP/AdaptiveBackpropagation.pdf

Riedmiller, Martin. (1994-b) "Rprop - Description and Implementation Details"
Institut fur Logik, Komplexitat und Deduktionssyteme, Technical Report, January 1994, University of Karlsruhe, W-76128 Karlsruhe, FRG https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a8815421205d3a8938d78db974db1d4f3584ffb6 https://citeseerx.ist.psu.edu/doc/10.1.1.21.3428

Rosenblatt, Frank. (1958) "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain"
Psychological Review, Vol. 65, No. 6, pp. 386-408
https://perceptrondemo.com/assets/rosenblatt-perceptron-probabilistic-1958-9c2228ee.pdf

Rosenblatt, Frank. (1962) *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms* Vol. 55, Washington, DC: Spartan Books, 1962. https://babel.hathitrust.org/cgi/pt?id=mdp.39015039846566&seq=9

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. (1985) "Learning Internal Representations by Error Propagation"
Institute of Cognitive Science, Report 8506, September 1985, University of California - San Diego, La Jolla, CA 92093
https://apps.dtic.mil/sti/pdfs/ADA164453.pdf

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. (1986-a) "Learning Internal Representations by Error Propagation"
in *Parallel Distributed Processing, Volume 1: Explorations in the Microstructure of Cognition: Foundations.* by David E. Rumelhart, James L. McClelland, and PDP Research Group (Editors), MIT Press, ©1986, pp. 318-362
https://direct.mit.edu/books/monograph/chapter-pdf/2163044/9780262291408_cah.pdf

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. (1986-b) "Learning Representations by Back-Propagating Errors"
Nature, Vol. 323, No. 6088, pp. 533-536
https://www.cs.toronto.edu/~hinton/absps/naturebp.pdf

א