Volume 7 | 2017 Spring 2017 Article 4

2017

Diagnosing Breast Cancer with a Neural Network

John Cullen
University of South Florida

Advisors:
Arcadii Grinshpan, Mathematics and Statistics
John Cullen Sr., Principal Software Engineer, Mach7 Technologies

Problem Suggested By: John Cullen Sr.

Diagnosing Breast Cancer with a Neural Network

Abstract

Fine needle aspiration (FNA) is a minimally invasive biopsy technique that can be used to successfully diagnose types of cancer, including breast cancer. Immediately, it is difficult for a human to spot any trends in the cell level data gathered during a fine needle aspiration procedure. One way to predict the type of tumor a patient has, is to use a computer to develop a mathematical model based on known data. This project utilizes the Diagnostic Wisconsin Breast Cancer Database (DWBCDB) to create an accurate mathematical model that predicts the type of a patient's tumor (Malignant or Benign). A neural network model is created in a two step-process. It is first created with random parameters, and is then refined using the data set, with known tumor types. A model with a success rate of 98% is created, which suggests that there is a high level of correlation between FNA data and the type of tumor a patient had. This approach was not capable of producing a perfect model that could be used in clinical applications.

Keywords

breast cancer, machine learning, neural network

Creative Commons License

PROBLEM STATEMENT

Using a large set of Fine Needle Aspirate (FNA) breast cancer biopsy data, develop a model that can accurately predict whether a new patient's tumor is malignant or benign.

MOTIVATION

The issue of finding noninvasive techniques for the diagnosis of diseases is extremely important. Generally, surgical biopsies can be very anxiety provoking in patients, and may even lead them to put off the procedure or not get it done at all. This of course, will reduce their chance of survival if they do have the disease. Techniques like FNA are less threatening to patients, safer, and faster. Techniques like this often do not provide as much information to a doctor as a full biopsy would. In this difficult situation, a computer can be used to recognize and diagnose patients based on a mathematical model that it has developed based on previous patients with known tumor types.

Given a large data set, we know some basic common sense information about the trends it contains, but we don't have a way to quantify this, or the complex interactions each variable has with one another. The field of machine learning allows computers to quantify these interactions for us, and is very important to the computer science field. There are applications for machine learning in the medical field, artificial intelligence, marketing, speech recognition, and many other areas of study. Neural networks can be used successfully on a diverse set of problems, and have started to become very powerful as computer hardware improves. Models that are trained by a computer, or more specifically, neural networks, can pick up very difficult to spot trends that humans often miss. This allows us to develop very accurate prediction systems without needing to exhaustively analyze data.

The objective of this project is to develop an accurate neural network based model with machine learning techniques, that is capable classifying a patient's tumor type based on the data found in the DWBCDB.

MATHEMATICAL DESCRIPTION AND SOLUTION APPROACH

I. MODELING THE DATA

Table 1 covers each data point in the DWBCDB, as well as the known type of the patient's tumor. We will be using this data set to develop a mathematical model to predict a new patient's tumor type.

Variables	Description
1. Mean (Fields 1-10) a. Radius b. Texture c. Perimeter d. Area e. Smoothness f. Compactness g. Concavity h. Concave Points i. Symmetry j. Fractal Dimension	The means of the calculated cell values from the sample.
2. Standard error (Fields 11-20) a. Radius b. Texture c. Perimeter d. Area e. Smoothness f. Compactness g. Concavity h. Concave Points i. Symmetry j. Fractal Dimension	The standard errors of the calculated cell values from the sample.
3. Largest (Fields 21-30) a. Radius b. Texture c. Perimeter	The averages of the three largest values in the sample of the values calculated for each cell.

d. Area e. Smoothness f. Compactness g. Concavity h. Concave Points i. Symmetry j. Fractal Dimension	
4. Diagnosis (Field 31)	The type of the tumor that has developed in the patient's breast. M (Malignant) or B (Benign)

Table 2: A table of the parameters provided in the DWBCDB.

In a situation with many possibly correlated variables, one of the best options is to use a neural network to model the data and to make a prediction. A neural network is simply a mathematical model that can fit many shapes, unlike a low degree polynomial. It is based on the neuron structure of the human brain, but it much less powerful. A neural network can be 'taught' to recognize trends by adjusting its parameters so that the difference between what it predicts and the actual value is minimal.

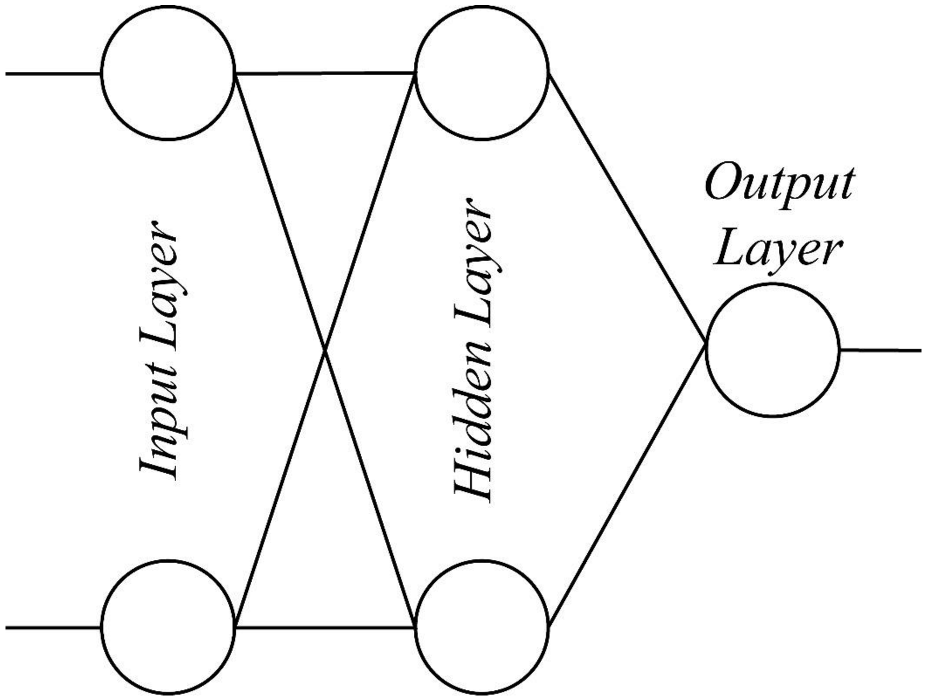


Figure 1: A simple neural Network.

Figure 1 is a visual representation of a simple neural network. Each line represents a *weight* and each circle or *node* represents the weighted sum of the outputs of the previous layer that is run through a non-linear function called an *activation* function. The last layer, or the *output layer,* represents what the model has predicted based on the values entered in the first layer, or *input layer.* Although it is possible to have many output nodes, this situation will only require one. We will consider an output value of 1 to represent 'Malignant' and a value of 0 to represent 'Benign'. We can consider the network to be a multivariate function with a range of (0,1).

$$P = f_{network}(I_1, I_2, I_3, \dots)$$

When each node in the input layer is set to an activation value representing the input of the function, each subsequent node in the network will receive an activation value based on the ones in the previous layer, and the weights. The activation values in the first layer are simply the inputs of the network function, shown above. The activation value of the node in the last layer will be the prediction of the network. The activation for the jth node in the ℓ^{th} layer is denoted by:

$$a_j^\ell = \sigma(z_j^\ell) \tag{1}$$

$\sigma(x)$ is a non-linear function. The function must be non-linear for the network to fit non-linear data. We will use the sigmoid function because it closely resembles how a human neuron would be activated (Wikipedia).

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

$$\lim_{x\to\infty} \sigma(x) = 1$$

$$\lim_{x\to-\infty} \sigma(x) = 0$$

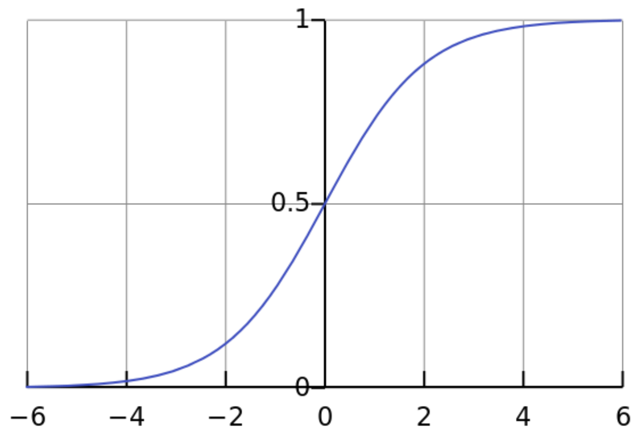


Figure 2: The logistic curve, $\sigma(x)$ *(Wikipedia)*

The limits of the sigmoid function are useful because we can consider an output close to 1 to represent an activated neuron and an output close to 0 to represent a non-activated neuron. They also scale all outputs to our desired range of (0,1). In the output layer an activated neuron will mean that the data satisfies the criteria of the category that the node represents.

The term z_j^ℓ is the weighted sum of the inputs plus a bias term b_j^ℓ. The bias constant is necessary here because we want the sigmoid function to be able to shift to the left or right, which makes fitting the function to the data easier. Later, both the weights and biases will be updated to make the network fit the data more closely, but they will be initialized to random constants.

$$z_j^\ell = \sum_i^{n_{\ell-1}} \omega_{ij}^\ell a_i^{\ell-1} + b_j^\ell \tag{2}$$

Here we are summing over each node i in the previous layer and multiplying its weight by its activation value. The upper limit of the sum, $n_{\ell-1}$, is the number of nodes in the previous layer. The term ω_{ij}^ℓ represents the weight from the ith node in the previous layer (ℓ-1) to the jth node in the layer ℓ. $a_i^{\ell-1}$ is the activation value of the ith node in the previous layer.

With equations (1) and (2), we can calculate the output of the neural network given the inputs. A completed application of these equations is in **Appendix I**.

II. Training the Model: Adjusting the Weights and Biases

To improve the weights and biases of the network, we can come up with a function of these parameters that returns a value that represents how far off the real value the network's prediction is. The parameters of the function can then be altered in a way that minimizes it. We will use the one half squared error function which is commonplace in machine learning applications. y_i is the expected activation value for node i in the last layer, L. Note that in this equation the activations of the final layer are constants, as are the expected activations, because we will iterate through each training example and minimize the error for each example.

$$E(\omega_{1,1}^{1}, \dots \omega_{ij}^{\ell}, b_1^1, \dots b_j^{\ell}) = \frac{1}{2}\sum_{i}^{n_{\mathrm{L}}}(y_i - a_i^{\mathrm{L}})^2 \tag{3}$$

While it is theoretically possible to minimize the function in closed form, it would be very difficult and time consuming. Since we will be using a computer to do the math, we can use an iterative algorithm called gradient descent. Gradient descent uses the derivative of a function to find a downward slope, and then takes a step down that slope, much like the way a sled would go down a hill. The derivative is multiplied by a constant η, to determine how large the step should be. Note that this technique will only find a local minimum, so the process is repeated multiple times starting at random values, and the best result (Closest to absolute minimum) is used. There is an example of this process in **Appendix II**. Here is the equation that will be used to update each weight and bias using gradient descent:

$$\omega_{ij}^{\ell} := \omega_{ij}^{\ell} - \eta \cdot \frac{\partial E}{\partial \omega_{ij}^{\ell}} \tag{4}$$

$$b_j^{\ell} := b_j^{\ell} - \eta \cdot \frac{\partial E}{\partial b_j^{\ell}} \tag{5}$$

Now all that is needed is the partial derivatives of E with respect to each weight and bias. These could be calculated using only the chain rule, but the process would be lengthy. A simple algorithm called Backpropagation will make the process faster. Backpropagations states (Nielsen):

$$\delta_j^\ell = \frac{\partial E}{\partial z_j^\ell} \tag{6}$$

This alias for the partial derivative of E with respect to z_j^ℓ is called the error or delta term, and can be used to calculate the error terms in the layer before it. To incorporate (6) into equations (4) and (5), the chain rule is applied:

$$\frac{\partial E}{\partial \omega_{ij}^\ell} = \frac{\partial E}{\partial z_j^\ell} \cdot \frac{\partial z_j^\ell}{\partial \omega_{ij}^\ell} = \delta_j^\ell \frac{\partial z_j^\ell}{\partial \omega_{ij}^\ell}$$

$$\frac{\partial E}{\partial b_j^\ell} = \frac{\partial E}{\partial z_j^\ell} \cdot \frac{\partial z_j^\ell}{\partial b_j^\ell} = \delta_j^\ell \frac{\partial z_j^\ell}{\partial b_j^\ell}$$

The final term of each equation is easy to calculate, we take the partial derivatives of (2) with respect to the weights and biases. The summation may be removed because the term will be zero unless k is equal to i:

$$\frac{\partial z_j^\ell}{\partial \omega_{ij}^\ell} = \frac{\partial}{\partial \omega_{ij}^\ell}\left(\sum_k^{n_{\ell-1}} \omega_{kj}^\ell a_k^{\ell-1} + b_j^\ell\right) = \frac{\partial}{\partial \omega_{ij}^\ell}\left(\omega_{ij}^\ell a_i^{\ell-1} + b_j^\ell\right) = a_i^{\ell-1}$$

$$\frac{\partial z_j^\ell}{\partial b_j^\ell} = \frac{\partial}{\partial b_j^\ell}\left(\sum_k^{n_{\ell-1}} \omega_{kj}^\ell a_k^{\ell-1} + b_j^\ell\right) = 1$$

The partial derivatives of the error function, (3), may be revised to:

$$\frac{\partial E}{\partial \omega_{ij}^\ell} = \delta_j^\ell \frac{\partial z_j^\ell}{\partial \omega_{ij}^\ell} = \delta_j^\ell \cdot a_i^{\ell-1} \tag{7}$$

$$\frac{\partial E}{\partial b_j^\ell} = \delta_j^\ell \frac{\partial z_j^\ell}{\partial b_j^\ell} = \delta_j^\ell \tag{8}$$

To find delta in terms of the delta in the layer $\ell + 1$, the chain rule can be applied to (6) to find how the z value of one node affects all the z values in the next layer, and subsequently how those z values affect the one-half squared error function.

$$\delta_j^\ell = \frac{\partial E}{\partial z_j^\ell} = \sum_k^{n_{\ell+1}} \frac{\partial E}{\partial z_k^{\ell+1}} \cdot \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_k^{n_{\ell+1}} \delta_k^{\ell+1} \cdot \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell}$$

The summation of the values for each node in the next layer is required because the change of a node in a previous layer will affect all nodes in the next layer, and all layers after that. The delta alias can be substituted for the first partial derivative, representing the error of each node in the next layer. The partial derivative of $z_k^{\ell+1}$ can be calculated like so, using a slightly modified version of (2):

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \frac{\partial}{\partial z_j^\ell}\left(\sum_i^{n_\ell} \omega_{ik}^{\ell+1} a_i^\ell + b_k^{\ell+1}\right)$$

The summation can be removed because all terms where i $\neq$ j will be zero. We also substitute $\sigma(z_i^\ell)$ in for a_i^ℓ using (1).

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \frac{\partial}{\partial z_j^\ell}\left(\omega_{jk}^{\ell+1}\sigma(z_j^\ell) + b_k^{\ell+1}\right) = \omega_{jk}^{\ell+1}\sigma'(z_j^\ell)$$

Based on the above equations, we can use the following equation for the delta of an arbitrary node j in layer ℓ, where k represents a node in the next layer, $\ell + 1$:

$$\delta_j^\ell = \sum_k^{n_{\ell+1}} \delta_k^{\ell+1} \cdot \omega_{jk}^{\ell+1}\sigma'(z_j^\ell) \tag{9}$$

Now we just need to determine the actual value of δ_j^ℓ in the last layer, so we can use it to solve the rest of the delta terms. This can be solved simply by substituting $\sigma\left(z_i^{\mathrm{L}}\right)$ in for a_i^{L} using (1), and applying the chain rule. The summation can be eliminated again because if i $\neq$j the derivative will be zero.

$$\delta_j^{\mathrm{L}} = \frac{\partial E}{\partial z_j^{\mathrm{L}}} = \frac{\partial}{\partial z_j^{\mathrm{L}}}\left(\frac{1}{2}\sum_{i}^{n_{\mathrm{L}}}(y_i - \sigma(z_i^{\mathrm{L}}))^2\right) = \frac{\partial}{\partial z_j^{\mathrm{L}}}\left(\frac{1}{2}(y_j - \sigma(z_j^{\mathrm{L}}))^2\right)$$

$$= (y_j - \sigma(z_j^{\mathrm{L}}))(0 - \sigma'(z_j^{\mathrm{L}}))$$

$$= -(y_j - a_j^{\mathrm{L}})\sigma'(z_j^{\mathrm{L}})$$

$$= (a_j^{\mathrm{L}} - y_j)\sigma'(z_j^{\mathrm{L}}) \tag{10}$$

The last layer's delta term can be used to calculate all other error terms, and then the partial derivatives of each weight and bias, which gives us enough information to complete an update of the parameters to minimize the error function. A worked example of a weight update can be found in **Appendix III**.

III. IMPLEMENTING THE MODEL

Fully training the model to fit the dataset will require thousands of iterations of gradient descent with hundreds of training examples. For this reason, the most efficient way to implement the model is by creating a computer program that will make the calculations without the possibility of human error. The following is an explanation of the program that I implemented, the full source code can be found in **Appendix IV.**

1. Before writing the program, I scaled all the data from the DWBCDB into values between 0 and 1. This makes the input activation levels fall in the same range as the rest of the

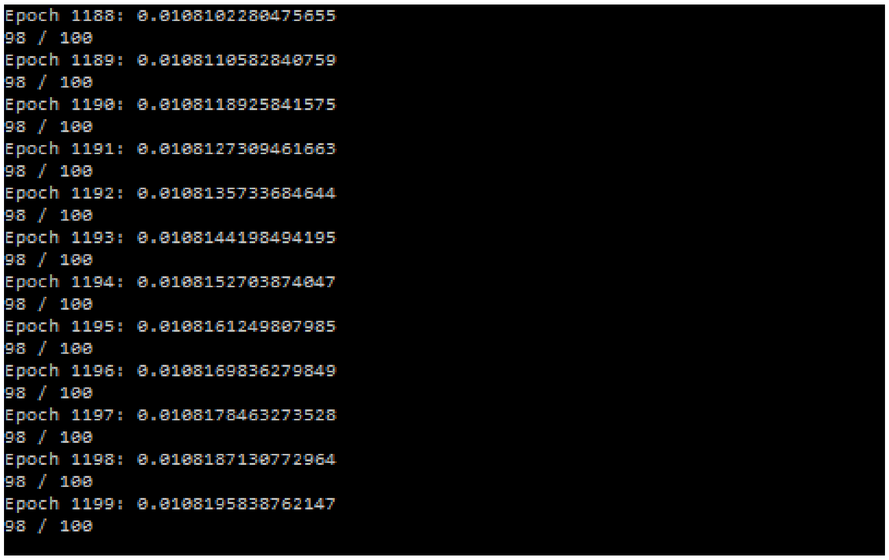
activation values. This was done using the following formula, where x is the column's value for a row of data: $\frac{x-x_{min}}{x_{max}-x_{min}}$.

2. Separate the data set into two distinct sets, one to use for training, and the other to test the model as it is trained. The DWBCDB has 569 different patients in it. I chose a training set of 469 rows, so that the testing set would be 100 rows. This way the test results can easily be converted to a percentage.
3. A neural network with 3 layers is created. The first layer has 30 nodes, the number of the inputs. The second layer has 30 more nodes, and the output has one node where any value over 0.5 is 'Malignant' and any value under that threshold is 'Benign'. All the weights and biases are initialized to random values. Anything more than this number of layers makes the process extremely slow due to the increase in biases and weights that need to be updated.
4. For each row in the training set, an update of each bias and weight is performed, using the gradient descent equations. I found through trial and error that $\eta = 0.02$ was a good value for the constant in the gradient descent equation. It takes steps big enough that the model learns quickly, but does not overshoot.
5. The model is tested using the 100-row training set. The network is fed the inputs for those rows, and the value of the output node is compared to the real output. The number correct guesses, as well as the mean of the error function (calculated for each row) is logged.
6. Steps 4-5 are repeated 1200 times. I found that this number provides the best results. If the process is run more than this, the model becomes too specific to the training set and begins to start failing to recognize items in the testing set.

7. The entire process is repeated multiple times and the best result is used, in case the gradient descent algorithm finds a local minimum and gets stuck.

DISCUSSION

Running the program multiple times, I found a set of weights and biases that could predict the type of tumor in 98 of the 100 testing rows. This gives the model a 98% success rate. The exhaustive list of optimal weights and biases can be found in **Appendix V.** This is a higher success rate than anticipated and is indicative that at least some of the input parameters used have a very strong correlation with the output. Unfortunately, the model abstracts away a lot of information about how things are correlated so it is very difficult for a human to understand exactly what the machine is doing. It does however, provide a way to classify information when humans do not know anything about how it correlates to the results.



```
Epoch 1188: 0.0108102280475655
98 / 100
Epoch 1189: 0.0108110582840759
98 / 100
Epoch 1190: 0.0108118925841575
98 / 100
Epoch 1191: 0.0108127309461663
98 / 100
Epoch 1192: 0.0108135733684644
98 / 100
Epoch 1193: 0.0108144198494195
98 / 100
Epoch 1194: 0.0108152703874047
98 / 100
Epoch 1195: 0.0108161249807985
98 / 100
Epoch 1196: 0.0108169836279849
98 / 100
Epoch 1197: 0.0108178463273528
98 / 100
Epoch 1198: 0.0108187130772964
98 / 100
Epoch 1199: 0.0108195838762147
98 / 100
```

Figure 3: The output of the neural network program, found in Appendix IV.

It is also clear in **Figure 3** that overfitting was a problem that was encountered during the training process. The error function for the 100 testing rows started to increase even as the machine should be perfecting its model. This means that the model was starting to specifically fit the training dataset and becoming less generic, thus making it unable to deal with items it has not seen before. This indicates the training set may need to be larger than it is, with a more diverse set of examples.

Conclusion and Recommendations

Using a neural network system, a 98% accurate model was reached. This suggests that there is a strong correlation between FNA data and the type of a patient's tumor. This model however was not 100% accurate which can pose some serious problems when diagnosing patients with cancer. The consequences of a false positive or a false negative are both potentially catastrophic to the patients' health. This model should not be used in a clinical setting. The model is also very abstract and doesn't allow humans to see what trends the computer is seeing.

Improvements in accuracy would be necessary to use this model in a clinical environment. Several techniques could improve the accuracy of the model. A wide array of activation functions could be tested and the best selected. A gradient descent technique that incorporates momentum, much like the real world, could help the model to reach a global minimum. A larger training data set could provide the system with more possible cases. Finally, other machine learning models could be considered. Some other models, like decision trees, or grouping allow humans to better interpret trends.

Nomenclature

Symbol	Description
a_j^ℓ	The activation value of the node j in the layer ℓ
z_j^ℓ	The weighted sum of the activations of the previous layer, plus the bias term for node j in layer ℓ
n_ℓ	The number of nodes in the layer ℓ
$\sigma(x)$	The sigmoid activation function
ω_{ij}^ℓ	The weight between the node i in layer $\ell - 1$ and the node j in ℓ
b_j^ℓ	The bias term for node j in layer ℓ
E	The error function
y_i	The expected output of node i in the output layer, based on the training set
L	The last layer
η	The 'learning rate' constant for the gradient descent equation
δ_j^ℓ	The backpropagation error term. Same as $\frac{\partial E}{\partial z_j^\ell}$
$:=$	The assignment operator. Sets a variable to the specified value.

References

Johnson, Justin and Andrej Karpathy. "CS231n: Convolutional Neural Networks for Visual Recognition." 2017. *Github.* Course Notes. May 2017. <http://cs231n.github.io/>.

Lichman, M. *UCI Machine Learning Repository*. 2013. University of California, School of Information and Computer Science. Database. May 2017. <http://archive.ics.uci.edu/ml/>.

Mazur, Matt. "A Step by Step Backpropagation Example." 17 March 2015. *Matt Mazur.* Blog Post. May 2017. <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>.

Neural Networks Demystified [Part 3: Gradient Descent]. Welch Labs. 2014. Short Video. <https://www.youtube.com/watch?v=5u0jaA3qAGk>.

Neural Networks Demystified [Part 4: Backpropagation]. Welch Labs. 2014. Short Video. <https://www.youtube.com/watch?v=GlcnxUlrtek&t=2s>.

Neural Networks Demystified [Part 5: Numerical Gradient Checking]. Welch Labs. 2014. Short Video. <https://www.youtube.com/watch?v=pHMzNW8Agq4>.

Nielsen, Michael. *Neural Networks and Deep Learning*. Michael Nielsen , 2017. E-Book.

Rashid, Tariq. *Make Your Own Neural Network*. CreateSpace Independant Publishing, 2016. PDF.

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." 19 January 2016. Blog Post. May 2017. <http://sebastianruder.com/optimizing-gradient-descent/>.

Shiffman, Daniel. *The Nature of Code: Simulating Natural Systems with Processing*. Daniel Shiffman, 2012. E-Book.

Wikipedia. "Sigmoid function." 1 May 2017. *Wikipedia*. Wiki. May 2017. <https://en.wikipedia.org/wiki/Sigmoid_function>.

Wolberg, Dr. William H., W. Nick Street and Olvi L. Mangasarian. "Breast Cancer Wisconsin (Diagnostic) Data Set." *UCI Machine Learning Repository*. UCI Machine Learning Repository, 01 11 1995. CSV. <http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)>.

APPENDIX

I. FORWARD PROPAGATION

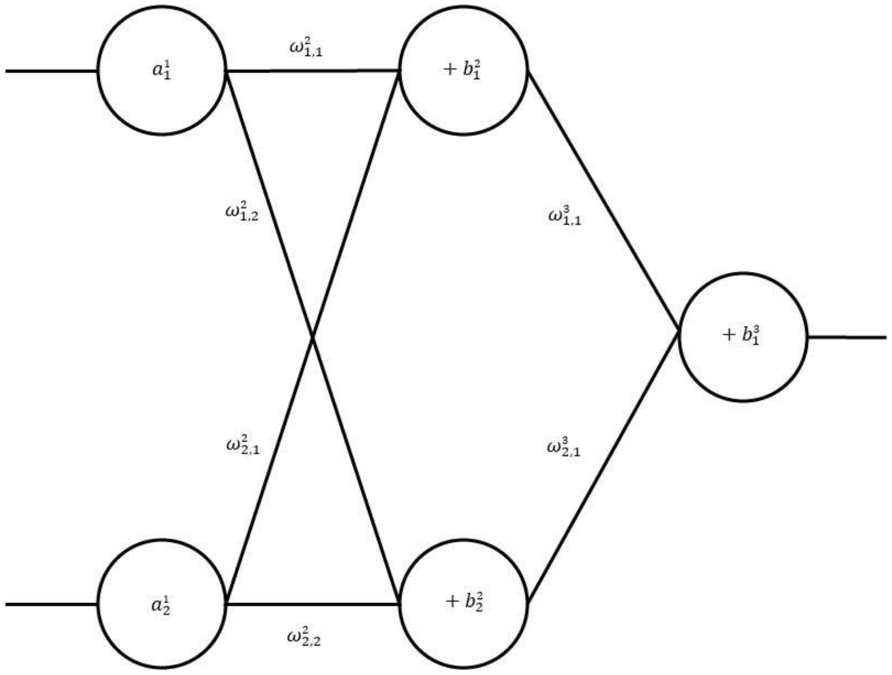


Figure 4: A three-layer network with 2 inputs, 2 nodes in the second layer, and one output.

Consider a neural network with 2 input nodes, 2 nodes in the second layer, and 1 node in the output layer. The output can be calculated as follows when the inputs are both equal to 1, and the following weights and biases.

$$a_1^1 = 1, a_2^1 = 1$$

$$\omega_{1,1}^2 = 0.5, \omega_{1,2}^2 = 0.5, \omega_{2,1}^2 = 0.5, \omega_{2,2}^2 = 0.5$$

$$\omega_{1,1}^3 = 0.5, \omega_{2,1}^3 = 0.5$$

$$b_1^2 = 0.5, b_2^2 = 0.5, b_1^3 = 0.5$$

The goal is to find the activation value of the node in the last layer, but first we must find the activation values in the second layer, using (1) and (2):

$$a_1^2 = \sigma\left(\sum_i^2 \omega_{i,1}^2 a_i^1 + b_1^2\right) = \sigma(0.5(1) + 0.5(1) + 0.5) = \sigma(1.5) = 0.818$$

$$a_2^2 = \sigma\left(\sum_i^2 \omega_{i,2}^2 a_i^1 + b_2^2\right) = \sigma(0.5(1) + 0.5(1) + 0.5) = \sigma(1.5) = 0.818$$

Then the activation value of the last node can be calculated, again using (1) and (2):

$$a_1^3 = \sigma\left(\sum_i^2 \omega_{i,1}^3 a_i^2 + b_1^3\right) = \sigma(0.5(0.818) + 0.5(0.818) + 0.5) = \sigma(1.32) = 0.789$$

So for the inputs of 1, and 1 our network predicts an output value of 0.789.

II. Gradient Descent

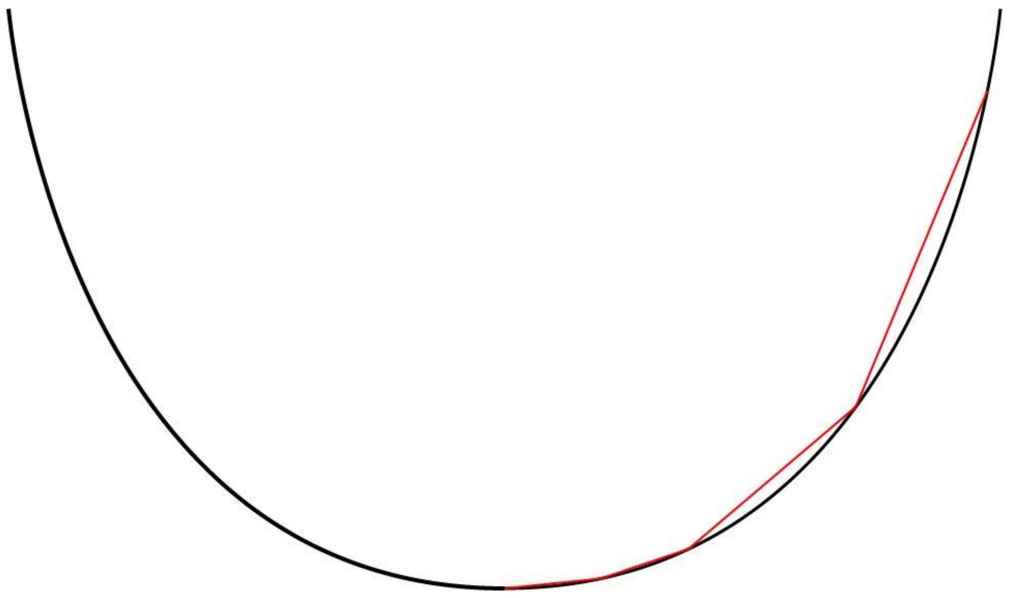


Figure 5: A sketch of a gradient descent path (red) in a parabola (black).

We will apply gradient descent to the function $f(x) = x^2$, starting at x = 1 using $\eta = 0.4$.

We can write the update as follows:

$$x := x - \eta \cdot f'(x)$$

Or

$$x := x - 0.8x$$

The update can be repeated until close to a minimum value.

$$x := 1 - 0.8 = 0.2$$

$$x := 0.2 - 0.8(0.2) = 0.04$$

$$x := 0.04 - 0.8(0.04) = 0.008$$

As you can see, each iteration of gradient descent is getting closer to the minimum value of the function, 0.

III. Backpropagation

To update the weights of the 3-layer neural network created in **Appendix I,** there are a couple more pieces of information that are necessary. The first is the η term for gradient descent, we will use $\eta = 0.1$. The next is the expected output given the inputs of 1, and 1. We will use 1 as the expected output. Again, we have a neural network with a 2 node first layer, a 2 node second layer, and a 1 node output layer, with the following weights, biases, and inputs:

$$a_1^1 = 1, a_2^1 = 1$$

$$\omega_{1,1}^2 = 0.5, \omega_{1,2}^2 = 0.5, \omega_{2,1}^2 = 0.5, \omega_{2,2}^2 = 0.5$$

$$\omega_{1,1}^3 = 0.5, \omega_{2,1}^3 = 0.5$$

$$b_1^2 = 0.5, b_2^2 = 0.5, b_1^3 = 0.5$$

$$y_1 = 1$$

$$\eta = 1$$

We start by forward propagating the network, which is already done in **Appendix I.** We know that:

$$z_1^2 = z_2^2 = 1.5$$

$$a_1^2 = a_2^2 = 0.818$$

$$z_1^3 = 1.32$$

$$a_1^3 = 0.789$$

Now the delta term for the last node can be calculated, using (10):

$$\delta_1^{\mathrm{L}} = (a_1^{\mathrm{L}} - y_1)\sigma'(z_1^{\mathrm{L}}) = (0.789 - 1)\sigma(x)(1 - \sigma(x)) = -0.211(0.789)(1 - 0.789)$$

$$= -0.035127069$$

Then the delta terms for the nodes in the second layer may be calculated, using (9):

$$\delta_1^2 = \sum_k^1 \delta_k^3 \cdot \omega_{1,k}^3 \sigma'(z_1^2) = (-0.035127069)(0.5)(0.818)(1 - 0.818) = -0.00261478876$$

$$\delta_2^2 = \sum_k^1 \delta_k^3 \cdot \omega_{2,k}^3 \sigma'(z_2^2) = (-0.035127069)(0.5)(0.818)(1 - 0.818) = -0.00261478876$$

Now the delta values can be used to calculated the derivatives of each weight and bias, using (7) and (8):

$$\frac{\partial E}{\partial b_1^2} = -0.00261478876$$

$$\frac{\partial E}{\partial b_2^2} = -0.00261478876$$

$$\frac{\partial E}{\partial b_1^3} = -0.035127069$$

$$\frac{\partial E}{\partial \omega_{1,1}^2} = \delta_1^2 \cdot a_1^1 = -0.00261478876(1) = -0.00261478876$$

$$\frac{\partial E}{\partial \omega_{2,1}^2} = \delta_1^2 \cdot a_2^1 = -0.00261478876(1) = -0.00261478876$$

$$\frac{\partial E}{\partial \omega_{1,2}^2} = \delta_2^2 \cdot a_1^1 = -0.00261478876(1) = -0.00261478876$$

$$\frac{\partial E}{\partial \omega_{2,2}^2} = \delta_2^2 \cdot a_2^1 = -0.00261478876(1) = -0.00261478876$$

$$\frac{\partial E}{\partial \omega_{1,1}^{3}} = \delta_1^3 \cdot a_1^2 = (-0.035127069)(0.818) = -0.02873394244$$

$$\frac{\partial E}{\partial \omega_{2,1}^{3}} = \delta_1^3 \cdot a_2^2 = (-0.035127069)(0.818) = -0.02873394244$$

Finally, gradient descent is used to update each weight and bias, using (4) and (5):

$$b_1^2 := b_1^2 - \eta \cdot \frac{\partial E}{\partial b_1^2} = 0.5 - (1)(-0.00261478876) = 0.50261478876$$

$$b_2^2 := b_2^2 - \eta \cdot \frac{\partial E}{\partial b_2^2} = 0.5 - (1)(-0.00261478876) = 0.50261478876$$

$$b_1^3 := b_1^3 - \eta \cdot \frac{\partial E}{\partial b_1^3} = 0.5 - (1)(-0.035127069) = 0.535127069$$

$$\omega_{1,1}^2 := \omega_{1,1}^2 - \eta \cdot \frac{\partial E}{\partial \omega_{1,1}^2} = 0.5 - (1)(-0.00261478876) = 0.50261478876$$

$$\omega_{2,1}^2 := \omega_{2,1}^2 - \eta \cdot \frac{\partial E}{\partial \omega_{2,1}^2} = 0.5 - (1)(-0.00261478876) = 0.50261478876$$

$$\omega_{1,2}^2 := \omega_{1,2}^2 - \eta \cdot \frac{\partial E}{\partial \omega_{1,2}^2} = 0.5 - (1)(-0.00261478876) = 0.50261478876$$

$$\omega_{2,2}^2 := \omega_{2,2}^2 - \eta \cdot \frac{\partial E}{\partial \omega_{2,2}^2} = 0.5 - (1)(-0.00261478876) = 0.50261478876$$

$$\omega_{1,1}^3 := \omega_{1,1}^3 - \eta \cdot \frac{\partial E}{\partial \omega_{1,1}^3} = 0.5 - (1)(-0.02873394244) = 0.52873394244$$

$$\omega_{2,1}^3 := \omega_{2,1}^3 - \eta \cdot \frac{\partial E}{\partial \omega_{2,1}^3} = 0.5 - (1)(-0.02873394244) = 0.52873394244$$

IV. Source Code

The following is the raw C# code that I wrote to calculate the weights and biases, as well as create the model that was arrived at as a solution to the problem.

```
// Program.cs
```

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Calc_Project
{
    class Program
    {
        static int[] Layers = { 30, 30, 1 };

        static double[][,] Weights;

        static double[][] Biases;

        static Program()
        {
            Random r = new Random();

            Weights = new double[Layers.Length][,];

            for (int i = 1; i < Layers.Length; i++)
            {
                Weights[i] = new double[Layers[i-1], Layers[i]];
            }

            Biases = new double[Layers.Length][];

            for (int i = 1; i < Layers.Length; i++)
            {
                Biases[i] = new double[Layers[i]];
            }

            // Initialize to random doubles
            for (int i = 1; i < Weights.Length; i++)
            {
                for (int j = 0; j < Weights[i].GetLength(0); j++)
                {
                    for (int k = 0; k < Weights[i].GetLength(1); k++)
                    {
                        Weights[i][j, k] = r.NextDouble() * 2 - 1;
                    }
                }
            }

            for (int i = 1; i < Biases.Length; i++)
            {
                for (int j = 0; j < Biases[i].Length; j++)
                {
                    Biases[i][j] = r.NextDouble() * 2 - 1;
                }
            }
        }

        // Calculate the output of the net given an input
        static double[] ForwardPropagate(double[] inputActivations)
        {
            double[] lastA = inputActivations;
            for (int i = 1; i < Layers.Length; i++)
            {
                lastA = A(Z(i, lastA));
            }

            return lastA;
        }

        // Get the gradients of each weight and bias
        static Tuple<double[][,], double[][]> BackwardPropagate(double[]
inputActivations, double[] expectedOutputs)
        {
            // Forward propagate first
            double[][] activations = new double[Layers.Length][];
            activations[0] = inputActivations;

            for (int i = 1; i < activations.Length; i++)
            {
                activations[i] = A(Z(i, activations[i - 1]));
            }

            // Calculate Error terms
            double[][] deltas = new double[Layers.Length][];

            // Calculate last layer
            deltas[Layers.Length - 1] = new double[Layers[Layers.Length - 1]];
            for (int i = 0; i < Layers[Layers.Length - 1]; i++)
            {
                deltas[Layers.Length - 1][i] = (activations[Layers.Length - 1][i] -
expectedOutputs[i]) * (activations[Layers.Length - 1][i] * (1 - activations[Layers.Length
- 1][i]));
            }

            // Calculate the rest of the layers
            for (int i = deltas.Length - 2; i > 0; i--)
            {
                deltas[i] = new double[Layers[i]];
                for (int j = 0; j < deltas[i].Length; j++)
                {
                    double sum = 0;

                    for (int k = 0; k < Layers[i + 1]; k++)
                    {
                        sum += deltas[i + 1][k] * Weights[i + 1][j, k];
                    }

                    deltas[i][j] = sum * (activations[i][j] * (1 - activations[i][j]));
                }
            }

            // Calculate the derivatives
            double[][] d_biases = deltas;

            // Weights
            double[][,] d_weights = new double[Layers.Length][,];
            for (int l = 1; l < d_weights.Length; l++)
            {
                d_weights[l] = new double[Layers[l-1],Layers[l]];

                for (int i = 0; i < d_weights[l].GetLength(0); i++)
                {
                    for (int j = 0; j < d_weights[l].GetLength(1); j++)
                    {
                        d_weights[l][i, j] = deltas[l][j] * activations[l - 1][i];
                    }
                }
            }

            return Tuple.Create(d_weights, d_biases);
        }

        // Estimate the values of the derivatives, to check the math done in the other
BackPropagate function
        static Tuple<double[][,], double[][]> BackwardPropagateEstimate(double[]
inputActivations, double[] expectedOutputs, double epsilon)
        {
            double[][] d_biases = new double[Layers.Length][];
            double[][,] d_weights = new double[Layers.Length][,];

            // Estimate bias derivatives
            for (int l = 1; l < d_biases.Length; l++)
            {
                d_biases[l] = new double[Layers[l]];
                for (int j = 0; j < d_biases[l].Length; j++)
                {
                    // For each bias estimate the gradient when changed slightly
                    Biases[l][j] += epsilon;
                    double loss1 = Loss(ForwardPropagate(inputActivations),
expectedOutputs);

                    Biases[l][j] -= 2*epsilon;
                    double loss2 = Loss(ForwardPropagate(inputActivations),
expectedOutputs);

                    // Set it back
                    Biases[l][j] += epsilon;

                    d_biases[l][j] = (loss1 - loss2) / (2 * epsilon);
                }
            }

            // Estimate Weights
            for (int l = 1; l < d_weights.Length; l++)
            {
                d_weights[l] = new double[Layers[l-1], Layers[l]];

                for (int i = 0; i < d_weights[l].GetLength(0); i++)
                {
                    for (int j = 0; j < d_weights[l].GetLength(1); j++)
                    {
                        Weights[l][i, j] += epsilon;
                        double loss1 = Loss(ForwardPropagate(inputActivations),
expectedOutputs);

                        Weights[l][i, j] -= 2*epsilon;
                        double loss2 = Loss(ForwardPropagate(inputActivations),
expectedOutputs);

                        Weights[l][i, j] += epsilon;

                        d_weights[l][i,j] = (loss1 - loss2) / (2 * epsilon);
                    }
                }
            }

            return Tuple.Create(d_weights, d_biases);
        }

        // Verify that all derivatives are calculated properly
        static bool CheckGradients(Tuple<double[][,], double[][]> calculated,
Tuple<double[][,], double[][]> estimated)
        {
            bool pass = true;

            // Weights
            for (int l = 1; l < calculated.Item1.Length; l++)
            {
                for (int i = 0; i < calculated.Item1[l].GetLength(0); i++)
                {
                    for (int j = 0; j < calculated.Item1[l].GetLength(1); j++)
                    {
                        // Percent diff
                        double dif = Math.Abs(estimated.Item1[l][i, j] -
calculated.Item1[l][i, j]) / Math.Abs(calculated.Item1[l][i, j]) * 100;

                        if (dif > 5)
                        {
                            pass = false;
                            Console.WriteLine($"Gradient Check failed: Calculated -
{calculated.Item1[l][i, j]} Estimated - {estimated.Item1[l][i, j]}");
                        }
                    }
                }
            }

            return pass;
        }

        // Calculate the Loss for the expected outputs and the actual outputs
        public static double Loss(double[] actualOutputs, double[] expectedOutputs)
        {
            double errSum = 0;

            for (int i = 0; i < expectedOutputs.Length; i++)
            {
                errSum += Math.Pow(expectedOutputs[i] - actualOutputs[i], 2);
            }

            return .5 * errSum;
        }

        // Train the network
        static void Train(double[][] trainingInputs, double[][] trainingOutputs, int
iters, double lr, double[][] testDataInputs = null, double[][] testDataOutputs = null)
        {
            for (int epoch = 0; epoch < iters; epoch++)
            {
                // Run a test data set if provided
                if (testDataInputs != null && testDataOutputs != null)
                {

                    double error = 0;

                    int totalCorrect = 0;

                    for (int i = 0; i < testDataInputs.Length; i++)
                    {
                        double[] predicted = ForwardPropagate(testDataInputs[i]);

                        error += Loss(predicted, testDataOutputs[i]);

                        double[] predictedWhole = predicted.Select(p => p >= 0.5 ? 1.0 :
0).ToArray();

                        totalCorrect += Enumerable.SequenceEqual(predictedWhole,
testDataOutputs[i]) ? 1 : 0;
                    }

                    error /= testDataInputs.Length;

                    Console.WriteLine($"Epoch {epoch}: {error}");

                    Console.WriteLine($"{totalCorrect} / {testDataOutputs.Length}");
                }

                // Use each example to train the net
                for (int i = 0; i < trainingInputs.Length; i++)
                {
                    // Backprop
                    Tuple<double[][,], double[][]> gradient =
BackwardPropagate(trainingInputs[i], trainingOutputs[i]);

                    // GD to update weights and biases
                    GradientDescent(gradient, lr);
                }
            }
        }

        // Update the weights and biases of each layer using gradient descent
        static void GradientDescent(Tuple<double[][,], double[][]> gradient, double lr)
        {
            // Update weights
            for (int l = 1; l < gradient.Item1.Length; l++)
            {
                for (int i = 0; i < gradient.Item1[l].GetLength(0); i++)
                {
                    for (int j = 0; j < gradient.Item1[l].GetLength(1); j++)
                    {
                        Weights[l][i, j] -= lr * gradient.Item1[l][i, j];
                    }
                }
            }

            // Update Biases
            for (int l = 1; l < gradient.Item2.Length; l++)
            {
                for (int i = 0; i < gradient.Item2[l].Length; i++)
                {
                    Biases[l][i] -= lr * gradient.Item2[l][i];
                }
            }
        }

        // The weighted sum of the inputs Z
        static double[] Z(int layer, double[] lastA)
        {
            double[] output = new double[Layers[layer]];

            for (int j = 0; j < output.Length; j++)
            {
                double weightedSum = 0;

                for (int i = 0; i < lastA.Length; i++)
                {
                    weightedSum += Weights[layer][i, j] * lastA[i];
                }

                output[j] = weightedSum + Biases[layer][j];
            }

            return output;
        }

        // The sigmoid activation function
        static double[] A(double[] z)
        {
            double[] output = new double[z.Length];

            for (int i = 0; i < output.Length; i++)
            {
                output[i] = 1 / (1 + Math.Exp(-z[i]));
            }

            return output;
        }

        static void Main(string[] args)
        {
            // Read in dataset
            List<double[]> inputs = new List<double[]>();
            List<double[]> outputs = new List<double[]>();

            using (var fs = File.OpenRead(@"C:\Users\jtcul\Desktop\training-set.csv"))
            using (var reader = new StreamReader(fs))
            {
                while (!reader.EndOfStream)
                {
                    var line = reader.ReadLine();
                    var values = line.Split(',');

                    var ins = new double[values.Length - 1];
                    for (int i = 0; i < ins.Length; i++)
                    {
                        ins[i] = double.Parse(values[i], 
System.Globalization.NumberStyles.Float);
                    }

                    inputs.Add(ins);
                    outputs.Add(new double[] { double.Parse(values[values.Length - 1], 
System.Globalization.NumberStyles.Float) });
                }
            }

            // Training Data
            double[][] trainingInputs = inputs.Take(469).ToArray();
            double[][] trainingOutputs = outputs.Take(469).ToArray();

            // Testing Data
            double[][] testingInputs = inputs.Skip(469).ToArray();
            double[][] testingOutputs = outputs.Skip(469).ToArray();

            Train(trainingInputs, trainingOutputs, 1200, 0.02, testingInputs, 
testingOutputs);

            Console.ReadLine();
        }
    }
}
```

V. The Optimal Weights and Biases

The neural network developed in this paper may be reproduced with the following weight and bias values. The DWBCDB can be used for input values, when each attribute is scaled using the process previously defined. The DWBCDB can be found at (http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29).

b^2_{1-30}

$= -0.84, -1.29, 0.57, -0.25, 0.18, -0.19, 1.77, -1.68, -0.5, 0.4, -0.2, -0.67, -0.94, 2.01, 1.27, 0.2,$

$-0.82, 2.31, -0.31, 1.01, -1.66, -0.12, 0.58, 0.58, -0.72, 0.93, -0.22, 0.26, 0.11, 1.43$

$b^3_1 = -0.92$

$\omega^2_{1,1-30}$

$= -1.02, -0.69, -0.79, 0.56, -0.13, 0.52, 0.09, 0.27, 0.1, -0.6, -0.49, 0.25, 0.26, 0.48, -0.07, 0.76, -0.54, 0.08,$

$0.89, -0.01, -0.36, 0.61, 0.48, 0.44, -1.11, 0.57, 0.12, 0.97, 0.61, -0.95$

$\omega^2_{2,1-30}$

$= 0.86, 0.1, 0.26, -0.67, 0.98, 1.06, -1.21, -0.27, -0.42, 0.55, 0.74, -0.26, 0.01, -1.09, -0.31, 0.07,$

$-0.71, 0.04, 0.08, -0.35, 0.17, 0.74, -0.05, 0.56, 0.44, -0.54, -0.25, -0.1, 0.24, 0.14$

$\omega^2_{3,1-30}$

$= -0.78, 0.28, 0.16, 0.32, -0.58, 0.37, 0.41, 0.29, 0.39, 0.05, 0.82, 1.12, -0.44, -0.66, 0.5, 0.89, -0.25, 0.32,$

$-0.27, -0.1, 0.51, -0.91, -0.16, -0.15, -0.17, -0.79, -0.89, 0.8, -0.38, 0.43$

$\omega^2_{4,1-30}$

$= 0.13, 0.33, -0.01, 0.67, 0.23, -0.63, -0.15, 1.15, 0.67, 0.52, 0.64, 0.46, 0.66, -1.09, -0.16, 0.07, 0.35, -0.77,$

$-0.93, 0.38, 1.1, 0.4, -0.94, -0.76, 0.65, -0.82, -0.59, -0.52, -0.93, -0.96$

$\omega^2_{5,1-30}$

$= 0.1, 0.53, -0.69, -0.82, 0.73, -0.48, -0.28, 0.11, 0.12, 0.19, -0.15, 0.04, 0.68, 0.27, 0.63, -0.91, -0.42,$

$-0.03, 0.81, 0.91, 0.03, 0.53, 0.7, -0.04, 0.54, 0.75, 0, -0.02, 0.53, -0.28$

$\omega^2_{6,1-30}$

$= 0.85, -0.78, 0.79, 0.52, -0.67, 0.85, 0.14, 0.2, -0.81, -0.43, -0.6, -0.86, 0.2, 0.26, 0.31, -0.09, 0.54, 0.5,$

$-0.26, 0.95, 0.48, 0.67, -0.15, -0.93, 0.17, 0.87, -0.91, 0.37, -0.43, 1.04$

$\omega^2_{7,1-30}$

$= 0.44, 1.09, -0.02, -0.34, 0.3, -0.91, -0.55, -0.15, 0.27, 0.88, 0.45, 0.49, -0.32, -0.49, -0.11, -0.3, -0.58,$

$-1.1, -0.96, -1.05, -0.14, -0.79, -1.15, 0.91, -0.1, -1.15, -0.95, 0.64, -0.71, -0.95$

$\omega^2_{8,1-30}$

$= 0.57, 0.81, -0.5, -0.02, -0.01, -0.18, -1.01, 0.95, 0.8, -0.09, 0.86, 0.88, 1.19, -0.85, -1.12, -0.5, 0.68,$

$-0.64, -0.41, -1.41, 1.32, 0.32, -0.07, 0.78, 0.29, 0.08, 0.38, 0.74, -0.06, -0.45$

$\omega^2_{9,1-30}$

$= -1, 0.29, 0.81, -0.52, -0.91, 0.88, -0.76, 0.73, -0.73, -0.31, -0.91, -0.13, 0.13, 0.41, 0.34, -0.28, -0.73,$

$0.76, 0.3, -0.27, -0.21, -0.24, 0.03, 0.99, -0.02, -0.51, -0.94, 0.34, -0.17, 0.46$

$\omega^2_{10,1-30}$

$= 0.42, -0.3, -0.66, -0.56, -0.29, -1.03, 0.87, 0.17, -0.16, -0.45, -0.48, 0.46, -0.16, 1.1, -0.4, -0.26, 0.94,$

$-0.4, -0.38, 0.12, -0.02, -0.92, -0.54, -0.62, -0.68, 0.32, -0.25, 0.82, 0.4, 0.55$

$\omega^2_{11,1-30}$

$= 0.77, 0.24, -0.42, 0.11, 0.26, 0, -0.98, 0.36, -0.21, 0.08, 0.24, 0.14, -0.18, -0.99, -1.52, 0.29, 1.04, -0.88,$

$0.64, -1.31, 0.04, 1.06, -0.69, 0.96, -0.88, -0.01, -0.32, 0.92, -0.84, -0.06$

$\omega^2_{12,1-30}$

$= 0.28, 0.55, 1.05, 0.19, -0.22, 0.84, -0.33, -0.88, -0.52, 0.21, -0.92, 0.2, 0.43, 0.79, 1.12, -0.64, 0.52, 0.98,$

$-0.71, 0.03, 0.37, 0.71, -0.78, 0.11, 0.11, 0.9, -0.96, 0.04, -0.55, 0.96$

$\omega^2_{13,1-30}$

$= -0.7, 0.01, -0.74, 0.31, 0.74, 0.25, -0.47, 0.52, 1, -0.68, -0.04, 0.7, -0.46, -0.88, -1.09, 0.25,$

$-0.38, -0.29, -1.01, -1.25, 0.99, 0.12, 0.38, 0.45, 0.7, 0.41, 0.64, -0.14, -0.91, -0.29$

$\omega^2_{14,1-30}$

$= 0.73, 0.74, 0.5, -0.09, -0.12, 0.28, -0.71, -0.21, 1.01, -0.32, 0.63, -0.41, 0.93, -1.46, 0.57, -1.01,$

$0.55, 0.1, -0.03, 0.43, 0.75, -0.01, -0.16, -0.53, -0.99, 0.03, 0.22, 0.1, 0.21, -1.12$

$\omega^2_{15,1-30}$

$= 0.55, 1.2, -0.21, 0.06, -0.57, 0.39, 0.22, 1.18, 0.55, -0.17, -0.29, -0.62, 0.59, -0.05, -0.9, 0.31, 0.5,$

$-0.05, 0.42, 0.28, -0.56, 0.99, 0.43, -0.66, 0.32, -0.52, -0.69, 0.34, -0.61, 0.15$

$\omega^2_{16,1-30}$

$= -0.93, 0.05, 0.32, 0.41, 0.08, 0.64, 0.72, -0.72, 0.06, -0.49, -0.8, 0.52, -1.09, -0.14, 0.91, 0.3, 0.42, 1.63,$

$0.96, 0.29, 0.23, 0.64, 0.89, 0.38, 1.03, 0.98, -0.82, -0.34, 0.63, 1.04$

$\omega^2_{17,1-30}$

$= 0.16, -0.65, 0.15, 0.88, -0.22, -0.09, -0.05, 0.6, -0.78, -0.04, -0.87, -0.25, -0.91, 0, 0.55, -0.48,$

$0.77, 0.76, 0.02, -0.73, 0.05, -0.42, 0.41, 0.7, 0.75, -0.63, -0.93, 0.57, 0.71, 0.36$

$\omega^2_{18,1-30}$

$= 0.37, -0.78, 0.2, 0.75, 0.35, 0.95, -0.46, -0.68, 0.11, -0.64, -0.72, -0.8, 0.43, 0.17, -0.1, 0.13, 0.56,$

$-0.12, 0.98, 0.47, 0.71, -0.58, 0.76, 0.14, -0.57, -0.38, -0.02, 0.38, -0.82, -0.64$

$\omega^2_{19,1-30}$

$= -0.15, -0.02, 0.9, -0.19, 0.56, 0.64, 0.69, -0.91, 0.47, 0.56, -1.07, 0.39, -0.7, -0.01, -0.62, 0.04,$

$0.06, -0.13, 0, -0.19, -0.48, 0.4, 0.69, -0.74, 0.52, 0.1, -0.31, -0.94, 0.3, -0.2$

$\omega^2_{20,1-30}$

$= -0.04, -0.23, -0.75, -0.21, 0.49, -0.12, -0.59, -0.67, 0.23, 0.05, 0.67, 0.3, -0.04, -0.31, 0.67,$

$-0.6, -1.09, 0.51, 0.15, 0.72, 0.17, -0.52, -0.42, 0.46, 0.69, 0.55, 0.8, 0.52, 0.47, -0.33$

$\omega^2_{21,1-30}$

$= -0.13, 0.87, -0.47, 0.29, -0.38, 0.01, -0.45, 1.26, 0.36, 0.27, 0, 0.07, 0.4, 0.14, -0.69, 0.28, -0.28,$

$-0.76, -0.18, -0.58, 0.66, 0.66, -0.44, 0.82, -0.09, -0.88, -0.46, -0.27, 0.15, -1.44$

$\omega^2_{22,1-30}$

$= -0.02, 0.25, -0.44, 0.35, -0.97, -0.7, -1.71, 0.11, 1.05, -0.54, 0.32, 0.09, 1.37, -0.19, -0.98,$

$-0.31, 0.9, -1.54, -0.49, -1.53, 0.96, -0.33, -0.73, -0.58, 0.26, -0.05, -0.65, 0.39, 0.34, -1.52$

$\omega^2_{23,1-30}$

$= -0.67, -0.01, -1.15, 0.47, -0.9, -0.86, -0.02, -0.24, 0.9, -0.32, 0.25, -0.53, 0.71, -1.05,$

$-1.3, -0.71, 0.87, -0.68, -0.08, -1, 0.48, -0.9, -1.16, 0.07, -0.35, -0.66, 0.93, -0.16, 0.19, -0.65$

$\omega^2_{24,1-30}$

$= 0.84, 0.52, -0.74, 0.94, 0.97, -0.65, -1.31, 0.45, 1.08, -0.21, 0.48, 0.62, 0.03, -1.68, -0.39, -0.13,$

$0.38, -1.65, -0.81, -0.39, 1.42, 0.56, -0.99, 0.17, -0.11, -0.89, -0.39, 0.26, -0.93, 0.15$

$\omega^2_{25,1-30}$

$= -0.96, -0.17, -0.31, 0.5, -0.37, -0.99, 0.06, 1.3, -0.35, 0.74, -0.82, 0.13, 0.19, -1.34, -0.1, 0.63,$

$0.55, -1.29, 0.79, -1.46, 0.92, 0.78, -0.66, 1.08, -0.38, -0.87, -0.3, 0.63, -0.74, 0.05$

$\omega^2_{26,1-30}$

$= 0.67, -0.21, -0.26, -0.65, -0.37, -0.32, -0.51, 0.26, 0.2, 0.21, 0.45, 0.13, 0.54, -0.25, 0.58, -0.01,$

$-0.67, -0.34, 0.93, 0.64, -0.08, -0.43, 0.57, -0.88, -0.9, -0.36, -0.82, -0.1, -0.51, -0.63$

$\omega^2_{27,1-30}$

$= 0.25, 1.13, 0.23, 0.25, -0.98, -0.15, -1.36, 0.11, 0.85, -0.3, 0.53, 1.07, -0.14, -0.9, -0.84, -0.79,$

$1.2, -1.42, -0.24, 0.35, -0.52, -0.3, 0.47, -0.01, -0.72, -0.07, 0.82, -0.49, -0.81, -1.22$

$\omega^2_{28,1-30}$

$= -0.72, 0.43, -0.18, 0.56, 0.88, 0.72, 0.17, 0.34, -0.43, -0.49, -0.06, 1.02, -0.48, -0.23, -1.15,$

$0.19, 1.1, -0.74, 0.64, 0.29, 0.02, -0.82, -1.02, -0.43, -1.17, -0.47, -0.78, -0.36, -0.55, -0.48$

$\omega^2_{29,1-30}$

$= -0.04, 0.83, -0.8, 0.8, -0.81, -1.03, -0.77, 0.92, 0.08, -0.1, -0.28, 0.26, 0.75, -1.12, -0.25,$

$-0.7, -0.53, -1.05, -0.21, -0.24, 0.73, 0.26, -0.42, -0.46, 0.47, -0.89, 0.17, -0.21, -0.16, -0.28$

$\omega^2_{30,1-30}$

$= 0.38, -0.01, -0.6, 0.71, 0.36, 0.09, 0.02, -0.14, 0.19, -0.95, -0.29, -0.64, 0.8, -0.41, 0.14, 0.41,$

$-0.41, 0.81, 0.46, 0.68, -0.37, 0.89, 0.77, 0.27, 0.6, 0.77, -0.08, -0.92, 0.89, 0.71$

$\omega^{3}_{1-30,1}$

$= 0.02, 2.9, -1.88, 1.57, -0.38, -0.89, -3.35, 2.86, 2.03, -0.03, 0.97, 1.31, 2.12, -3.58, -3.33, -0.36, 1.68,$

$-4.54, -0.28, -3.12, 2.89, 0.52, -2.06, 0.88, -0.83, -2.08, -0.35, 0.94, -1.03, -2.66$