University of Calgary

ENEL 525 F2020 - Final Project

Jathniel Ong

December 2020

TABLE OF CONTENTS

INTRODUCTION

The main goal for this project is to create a neural network that can identify patients who have SARS-CoV-2 based on their blood count information. For this project, we were given a dataset with the patient's age, SARS-CoV-2 test result, and other blood markers. The dataset had 598 data points overall. Among these 598 data points, 81 had positive SARS-CoV-2 test results.

For this project, I followed a similar strategy as described in Chapter 22 and 25 of the textbook [1][2]. This strategy goes through the different stages of creating a neural network: preprocessing data, selecting network architecture, selecting training algorithms, initializing weights and training, and analyzing network performance.

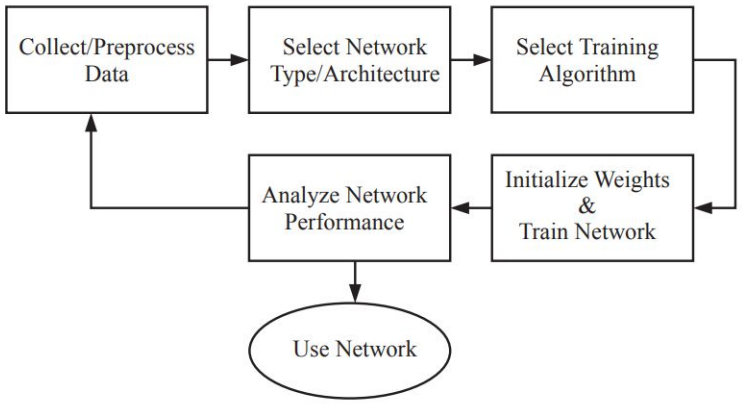


Figure 22.1 Flowchart of Neural Network Training Process

METHODOLOGY

Problem Setup (Preprocessing Data)

For neural networks, the first step is to normalize input data [1]. However, the data provided has already been standardized to 0 mean with unit standard deviation. Thus, the only input that needed to be normalized was the patient's age quantile. Next, unnecessary columns were removed, this allows the network to train more efficiently and effectively. For this particular dataset, column A (Patient Number) was removed. Ideally, more research could have been done to identify if any blood markers are unnecessary or could be combined. However, due to inexperience with blood data, the other columns were left untouched for this project.

For this project, the target output is the SARS-CoV-2 exam result. This column was provided as a string ("positive" or "negative"). Thus, the textual data was converted to numerical data. 0.76 (positive) and -0.76 (negative) was chosen instead of (1,-1). This would prevent the algorithm from oversaturating the tan-sigmoid function[1] that was chosen for our output's activation function (further described below in 'Network Design').

Training Testing Scheme

In order to test our network, the provided dataset was divided with a split of 90%-10%. 90% of the data would be used for training and 10% would be used for testing. The program kept the ratio of positive-negative test cases equivalent between the split (86% negative - 14% positive). Furthermore, the split was chosen randomly every time the network was trained.

The provided dataset also skewed towards negative test results (86% negative). In order to compensate for this while training, the positive test results were duplicated. This would ensure that our network is trained with a relatively equal number of positive and negative tests[2]. For our training, we duplicated the positive tests cases 6 times and ended up with a (465 : 504) split.

Network Design

For the network design, a 2-layer system (1 hidden, 1 output) was chosen because it would be easier to train and make adjustments quickly. Furthermore, it would also be possible to extend the network's structure if deemed necessary.

Since this problem is a classification problem, I decided to use a tansig-tansig activation function. This function was chosen because the input data is normalized to 0 mean with unit standard deviation. Thus, this function would be the best to evaluate the data. Furthermore, the textbook described the tansig-tansig function as a "good place to start" for classification problems such as this.[2]

Learning Rule

Ultimately, my system is similar to the backpropagation systems described in Chapter 11[3]. The learning rule derivations are adapted from equations 11.41 - 11.47[3] and specifically calculated for my tansig-tansig network.

Propagate inputs through the network:

$$\mathbf{a}^1 = \text{tansig}(\mathbf{W}^1\mathbf{p} + \mathbf{b}^1),\ p = \text{input from dataset}$$

$$a^2 = \text{tansig}(\mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2)$$

$$e = t - a^2$$

Propagate sensitivities backward:

derivative for tansig:

$$f(n) = \frac{df(n)}{dn} = \frac{d}{dn}\left(\frac{e^n - e^{-n}}{e^n + e^{-n}}\right) = 1 - \left(\frac{(e^n - e^{-n})^2}{(e^n + e^{-n})^2}\right) = 1 - (a)^2$$

$$\mathbf{s}^2 = -2\,(1 - (a^2)^2)\,e$$

$$\mathbf{s}^1 = (1 - (a^1)^2)\,\mathbf{W}^2\,s^2$$

Update weights and biases:

$$\mathbf{W}^2(k+1) = \mathbf{W}^2(k) - \alpha\,\mathbf{s}^2\,(\mathbf{a}^1)^T, \quad \mathbf{b}^2(k+1) = \mathbf{b}^2(k) - \alpha\,\mathbf{s}^2$$

$$\mathbf{W}^1(k+1) = \mathbf{W}^1(k) - \alpha\,\mathbf{s}^1(\mathbf{p})^T, \quad \mathbf{b}^1(k+1) = \mathbf{b}^1(k) - \alpha\,\mathbf{s}^1$$

Determining Number of Neurons

For the number of neurons, 1 neuron was used in the output layer as the network is only expected to produce a single value result (0.76, -0.76). For the hidden layer, I decided to start with a high number of neurons to ensure that the system is powerful enough to classify the problem. The challenge with this is having to stop training early to make sure that the network is not overtrained.

First, I started with an arbitrarily large-20-neuron hidden layer. This system is too powerful and would tend to overtrain the network quickly. As such, the network was trained with different iterations (epochs) in order to find the best classification system.

Test Results - Iteration v Accuracy for 20 Nodes					
constants: 0.01 alpha, 0.00005 threshold, 10x runs per iteration					
Iterations	**Correctly Identified Negative (avg 10 runs)**	**Correctly Identified Positive (avg 10 runs)**	**Incorrectly Identified Negative (avg 10 runs)**	**Incorrectly Identified Positive (avg 10 runs)**	**Total Accuracy (avg 10 runs)**
3000	43	4.1	9	4.9	77.2%
1000	42.9	3.4	9.1	5.6	75.9%
500	45.5	3.3	6.5	5.7	80.0%
300	45.1	3.4	6.9	5.6	79.5%
250	46.3	4.3	5.7	4.7	82.9%
225	44.7	4.2	7.3	4.8	80.1%
200	44	3.5	8	5.5	77.8%

Based on the results, the 20-node-network performs best around 250 iterations, this is considerably low, and the system tends to overtrain quickly, as such I tested it out with a significantly lower number of neurons (6 neurons).

Test Results - Iteration v Accuracy for 6 Nodes					
constants: 0.01 alpha, 0.00005 threshold, 10x runs per iteration					
Iterations	**Correctly Identified Negative (avg 10 runs)**	**Correctly Identified Positive (avg 10 runs)**	**Incorrectly Identified Negative (avg 10 runs)**	**Incorrectly Identified Positive (avg 10 runs)**	**Total Accuracy (avg 10 runs)**
10000	45	4.8	7	4.2	81.6%
7500	44.8	3.1	7.2	5.9	78.5%
5000	45.8	4.3	6.2	4.7	82.1%
4000	44.5	4.2	7.5	4.8	79.8%
3000	45.5	4.5	6.6	4.4	81.9%
2500	44.1	5.3	7.9	3.7	80.9%
1000	44.9	3.4	6.9	5.6	79.5%

Based on the results, the 6-node-network performs best around 5000 iterations, this is acceptable; however, I wanted to try a system that is slightly more powerful to see if I can train it with less iterations. As such, I tested it again with 10 nodes.

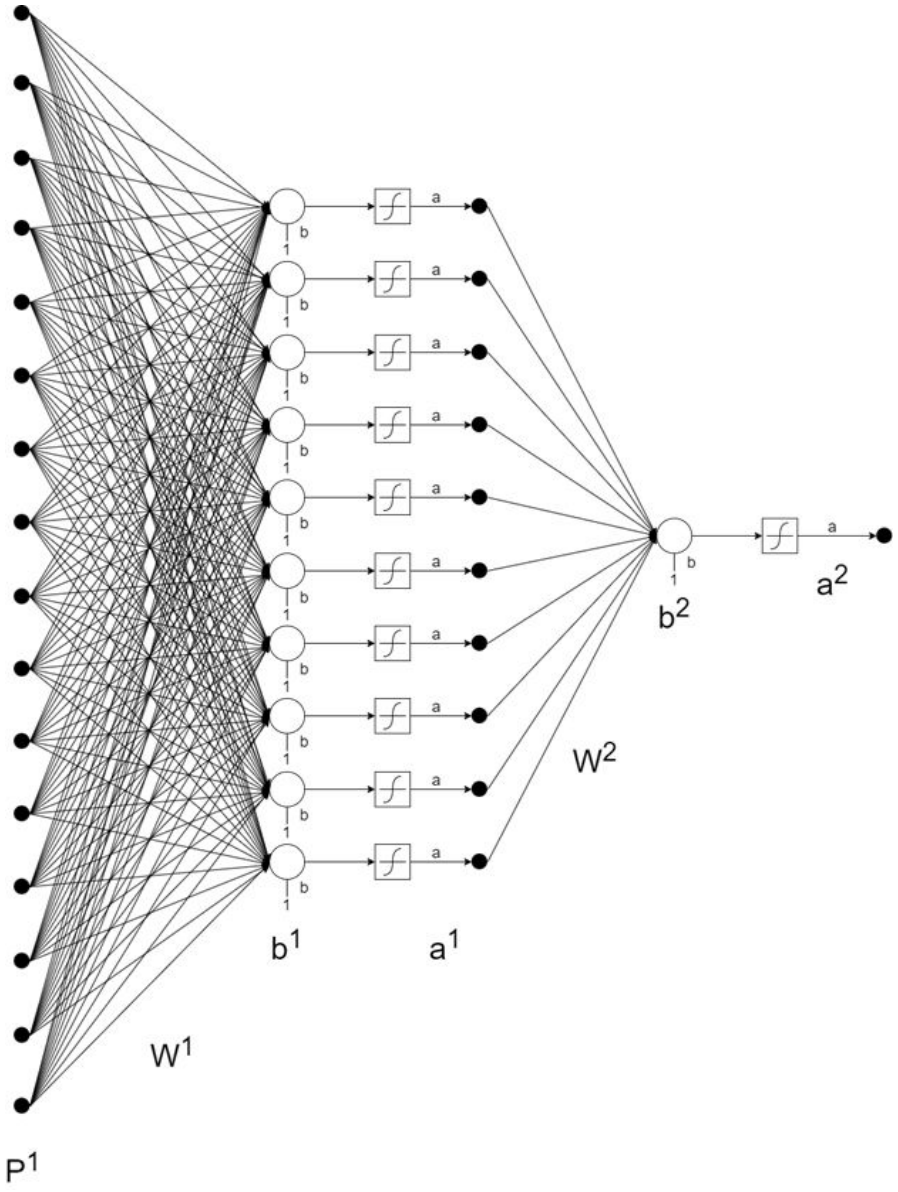
Test Results - Iteration v Accuracy for 10 Nodes					
constants: 0.01 alpha, 0.00005 threshold, 10x runs per iteration					
Iterations	**Correctly Identified Negative (avg 10 runs)**	**Correctly Identified Positive (avg 10 runs)**	**Incorrectly Identified Negative (avg 10 runs)**	**Incorrectly Identified Positive (avg 10 runs)**	**Total Accuracy (avg 10 runs)**
3000	44.8	3.5	7.2	5.5	79.2%
1500	44.9	3.7	7.1	5.3	79.6%
1000	45.9	4.1	5.8	5	82.2%
850	46.3	3.4	5.7	5.6	81.5%
750	45.5	4.1	6.5	4.9	81.3%
500	45.6	3.3	6.4	5.7	80.2%
300	44.4	4.2	7.6	4.8	79.7%

Ultimately, I ended up with 10 hidden nodes as it proved to be quick to train and powerful enough for the problem's complexity. However, as seen above, the other numbers worked as well (given the right number of iterations and not overtraining).

Note: The raw data for all (70x3=210) test runs are available in the attached excel file.

Final Network Diagram

Based on the explanations described in the sections above, I ended up with a network with 16 inputs, 10 hidden nodes, 1 output node, with a tansig-tansig activation function.



RESULTS AND DISCUSSION

Training Parameters

For the initial variables of the system, the alpha rate and error threshold were set to considerably low numbers (0.01, 0.00002). This would ensure that the network converges to a minimum[3]. Next, the weights were also set to small numbers (using randn). This would ensure that the system's transfer function does not become saturated too quickly[3]. These weights were also randomized on every training run to ensure that the system converges to the global minimum for some of the test runs.

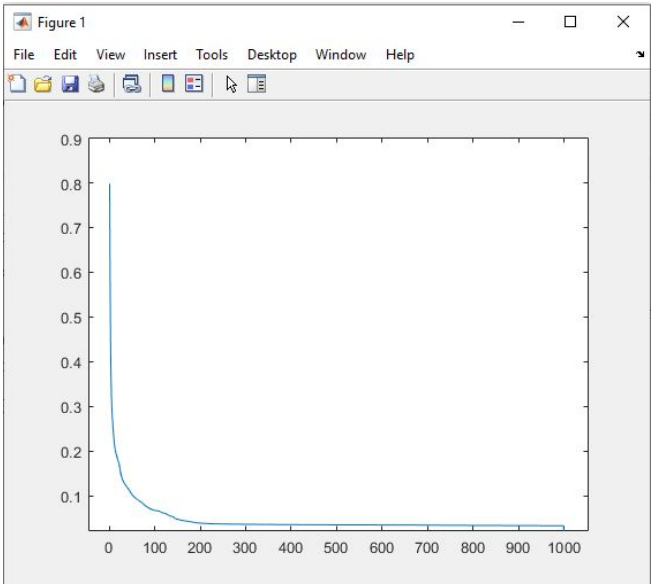
Training Process and Iterations

As described in the section "Determining Number of Neurons", I ended up choosing a system with more nodes than necessary and had to stop training early to prevent overtraining. Thus, for the 10-node system, I ran through 1000 iterations (epochs) to get the best results.

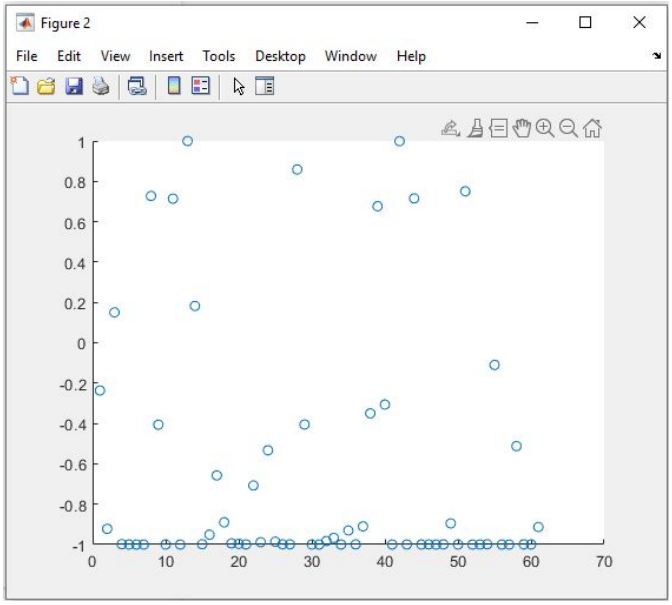
The training was repeated 10 times with randomized weights and a randomized test case selection (described in Methodology). Repeating the test multiple times ensured that results were accurate and would account for variations from different initial variables. This would also account for some tests not finding the global minimum.

Variables	**Value**
Alpha	0.01
Error Threshold	0.00002
Iterations (Epochs)	1000x
Weights	randn()
Train - Test Split	90% (train) - 10% (test)
Training data (adjusted)	969 - (465 : 504)
Training Attempts	10x
Test data	61 total - (52:9) split

Final Results and Performance



MSE training curve for one of the ten training attempts



Result Chart for one of the ten training attempts (1 - Covid Positive, -1 - Covid Negative)

10 Run Average	Positive Test (expected)	Negative Test (expected)	Percentage
Positive Test (actual result)	4.1 6.7%	5.8 9.5%	41.41%
Negative Test (actual result)	5 8.2%	45.9 75.5%	98.23%
Percentage	45.05%	88.78%	82.24%

Confusion Table - average of 10 runs

During training, my network was able to quickly minimize the MSE as expected. The MSE fell quickly during the initial training process and after 250 epochs, the improvement slowed significantly. Based on this result, the network seemed to have found the minimum point successfully and rather effectively.

The result chart shows the system's prediction for each of the test points. The system considers everything above a 0 as a positive test result. Consequently, it considers everything that falls under 0 as negative. Based on this graph, it seems that the system was able to get the correct ratio of negative to positive tests. The test data had 52 negative tests and 9 positive tests and the system classified 51 points as negative and 10 points as positive.

On average, the neural network was able to classify 82.24% of the test points correctly. The system is effective in correctly identifying negative tests (98.23%), however, it has a difficult time correctly identifying positive test cases (45.05%). The system also has a 9.5% false positive rate and a 8.2% false negative rate.

Effectiveness

Overall, the neural network's performance is adequate. However, it leaves much room for improvement. To improve on this performance there are several factors that should be considered.

First, the network needs more training data. Overall, the accuracy rate would be significantly improved with more training. Specifically, the system also needs more data

points that have positive test results. This would help the training process as the system would be exposed to more variability. Furthermore, this would help it identify positive test cases more accurately and efficiently.

Next, the data could be further pre-processes. Ideally, more research could have been done to identify blood markers that are unnecessary. Furthermore, it might be beneficial to check if any of the inputs can be combined and or transformed. Giving the system better training data would allow the system to train more effectively and accurately.

Next, the network's overall design could be improved. Currently, the system is using a two layer network. For further improvement, a 3 or even 4 layer system could be tested. Furthermore, different activation functions could be tried as well. For example, some nodes could have a linear or softmax activation function. Currently, the number of nodes for the network was determined through a trial-and-error basis. This is not ideal, and a more systematic approach should be used if available.

Finally, further analysis could have been done to spot areas of improvement. For example, given enough data, it might be helpful to do a sensitivity analysis to identify any inputs that are unnecessary.

CONCLUSION

In summary, for this project, the goal was to create a neural network that identifies SARS-CoV-2 results based on blood markers. A dataset with 598 data points was provided, among which, 81 data points had positive SARS-CoV-2 test results.

First, the data was preprocessed. Some columns were removed, some were normalized, and others were modified. Additionally, the dataset was split into training and testing data (90%:10%). Since there were more negative test results than positive test results, the positive test results were duplicated for the training data.

A 2-layer tansig-tansig network was chosen for the final network design. This decision was based on the input provided and information from the textbook[2]. In order to determine the number of nodes for the hidden layer, multiple values were tested with different iterations. Ultimately, 10 nodes were used in the final network design. This system is powerful enough to classify the inputs and very quick to train (~1000 epochs).

With the network design determined, 10 randomized training runs were done with the provided data. The network achieved adequate results that have room for improvement.

Overall, the network was able to correctly identify 82.24% of the results. It was able to identify most of the negative test cases (98%), however, it struggled to identify positive test cases (45%).

Finally, some improvement ideas were identified for future iterations. First, more data should be collected. This would help the training process. Next, the design of the neural network should be reconsidered (layers, activation function, number of nodes). Finally, the evaluation methods should also be analyzed further to see if there are any areas for improvement.

REFERENCES

[1] M. Hagan, H. Demuth, M. Beale and O. De Jesús, Neural network design. [S. l.: s. n.], 2016. Chapter 22

[2] M. Hagan, H. Demuth, M. Beale and O. De Jesús, Neural network design. [S. l.: s. n.], 2016. Chapter 25

[3] M. Hagan, H. Demuth, M. Beale and O. De Jesús, Neural network design. [S. l.: s. n.], 2016. Chapter 14