Investigating How The Change of Layer Count in Feed Forward Neural Networks Effects Performance

215758
Acquired Intelligence & Adaptive Behaviour (G6042)

Feburary 24, 2021

Abstract

A neural network is a set of linked together nodes that are modeled after biological neurons. Neural networks have several layers, however, it is difficult to figure out how many you need for a given problem. We will be trying to replicate an XOR gate with varying amounts of layers to see the effect on loss, using ReLU as our activation function. Knowing the right amount of layers for an XOR problem, seeing what happens when we only use one layer, and looking at general trends which could be generalized is what we are aiming to do. Here we show that having four layers for the XOR problem trains the fastest over the epoch, that one layered neural networks are still impossible, and that this problem isn't complex enough to show general trends throughout applications of neural networks. We found that increase of layers could either increase the time by a small margin or drastically decrease it, however, once we reached five layers errors occurred where it couldn't solve the XOR problem correctly. We need to next look into how having a more complex model affects the performance with layers due to the errors being found being problem-dependent.

1 Introduction

Finding the optimum count of layers for a problem using neural networks can be rather difficult and a guessing game. We will be using several neural networks with a differing layer count, seeing how the performance increases or decreases with added layers. We will be testing this on replicating an XOR gate, to correctly give the output depending on the two input bits. Table 1 shows the possible inputs and correct outputs. Additionally, we will discuss in further detail the problems of a one-layer neural network, and the importance of having an additional layer to mitigate the problems of the aforementioned. Having one layer to correctly replicate an XOR gate is impossible[4], investigating the loss over epochs will be able to show where it stagnates and to see the reason why we need hidden layers to correctly solve it. An epoch where you have iterated through all your training data once.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: XOR truth table

A node in a neural network is modeled after how a biological neuron functions, and a neural network is many of these nodes linked together as one. In figure 1, it displays how the input propagates and what influences the values when this is happening. Each input and weight is multiplied with an added bias, the values are summed together; the value is fed into the activation function which squishes it to be in between zero and one, this value is then outputted. If this was a multi-layered neural network, the outputted value would be sent to the connecting neurons, and the process repeats[2]. The activation could be a multitude of different options, such as sigmoid, tanh, or ReLU[5]. We will be using the RelU[1] (see figure 2 for visualization) activation function, as it's popularly utilized within neural networks, and using backpropagation[3] to adjust the weights and biases of the neural network to make it learn.

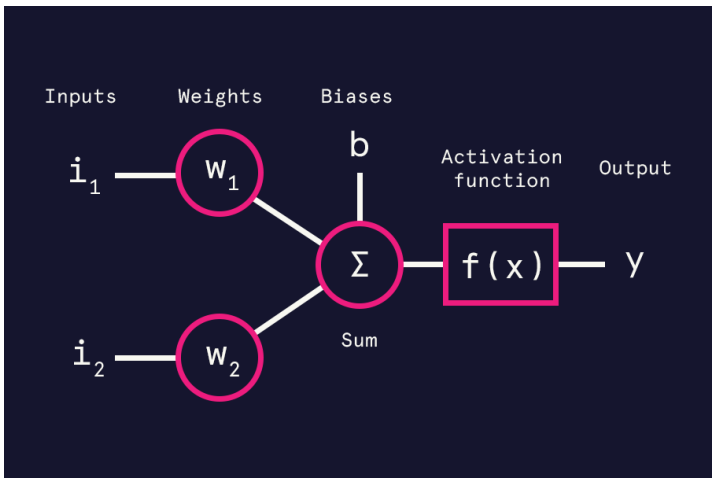


Figure 1: Visualization of how a node with two inputs would function

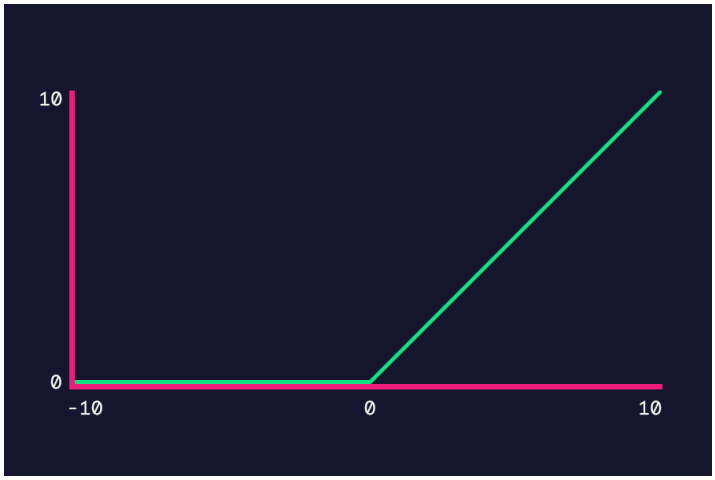


Figure 2: Visualization of relu activation function

Our hypothesis is that for one layer it will fail, due to it being mathematically impossible, but continue to work as well as the epochs will decrease, until some point where the layers needed become too complex for the problem space and errors will arise.

2 Methods

Initially, we thought about hand-crafting the weights and biases of each neural network, however, it would just create redundant code; instead, we generated them for the neural networks which had more than one layer. We did this by creating matrices of shape 3x3 for up to the size we wanted, after, we changed the first matrix to be of shape 2x3, and edited the last matrix to be 3x1, in total this created a valid set of matrices that can all be multiplied together. The biases were also generated, each layer was set as a vector of size 3. This cut algorithm 1 down to a fraction of what it potentially could have been; a visualization of the neural network can be seen in figure 3. For one layer, a pre-defined weight and bias are returned differing from the patterns found with multiple layers within algorithm 1 also.

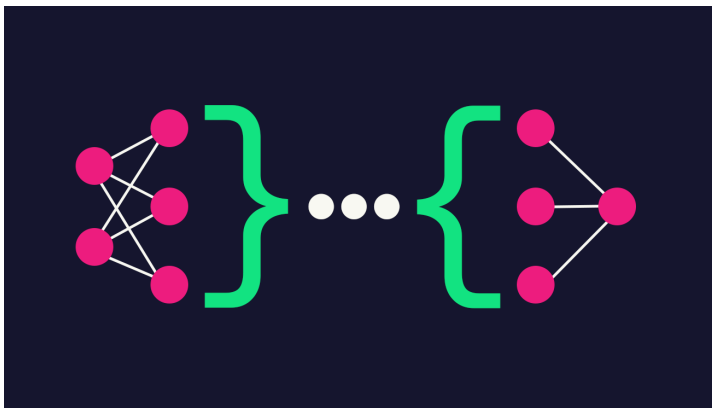


Figure 3: visualization of the neural network structure using the generation algorithm

Next, we needed a predict function to train the neural with; as we didn't hardcode each weight and bias, the code will have to be appropriately adjusted. The predict function implemented the use of the squishification mathematical function ReLU, and applies it for each layer; this is displayed in algorithm 3

Finally, there is the training algorithm itself, it sets up the input data, which is all combinations of two bits, and the output data is the associated correct result which you try to aim for. As we wanted to analyze increasingly larger neural networks, we looped through each size we desired to gather data for, the weights and biases are regenerated for the specific size and the optimizer is created. In each iteration, we looped through the number of epochs specified; within the epoch loop, we had to iterate through the dataset, as that is what an epoch is. For each iteration, we get the data for the position of the input and output array; we then found the prediction the neural network made on that data, calculate the loss and then backpropagate to alter the weights in the network. In each loop, we gathered the loss and error margin for plotting.

We trained the 15 different neural networks and increased the layer count by one over each network, during this process we will collect information about the loss and error margin for every epoch, this will then be plotted onto seperate graphs to display trends.

We found that changing layer count can vary your results, more could make it faster, or just completely ruin the model. However, this investigation was used on a very simple problem which can be easily solved with two layers, so the results shown could be inaccurate with a more complex problem which may need many layers. The learning rate may of also affected it, as at four layers, it hardly needed any epochs to solve the problem, but from there, excepting two cases, there was no learning.

3 Results

We expected that having only one layer will fail the task and it has done so; in figure 4 it shows us that it took below 500 epochs for it to stagnate and not progress, never reaching zero;

With two layers (figure: 5), it worked perfectly, all types of inputs convolved on zero at $\approx$ a thousand epochs.

At three layers (figure: 6), it differed from what we thought what have happened, it took more than a thousand epochs for no loss but not a drastic increase from the stats of two layers.

What we expected to happen at three layers (figure: 7), ended up happening at four, however at a much higher rate. We thought it would increase in performance more slowly over added layers, however, it only needed $\approx$ 250 epochs.

Next is five layers (figure: 8), this is where everything stagnates (for the most part), there is no learning happening; it seems that it cannot correctly predict for inputs with 0 1, or 1 0; it acts better than a one layered neural network as it can correctly classify two inputs unlike the aforementioned, but as a whole not representing an XOR gate correctly.

This continues, until we reach layer seven, and layer ten (in between these layers, before and after, have the same results as figure 8) these two graphs show that there is still an attempt to get to zero loss, but it gets caught between 0.4 and 0.2 and never progresses, the difference is that there is more of a dip before convolving for ten layers. Looking at figure 4, it gets caught at the same position of loss.

We see in figure 10 that there still is no progress happening to get back to zero loss.

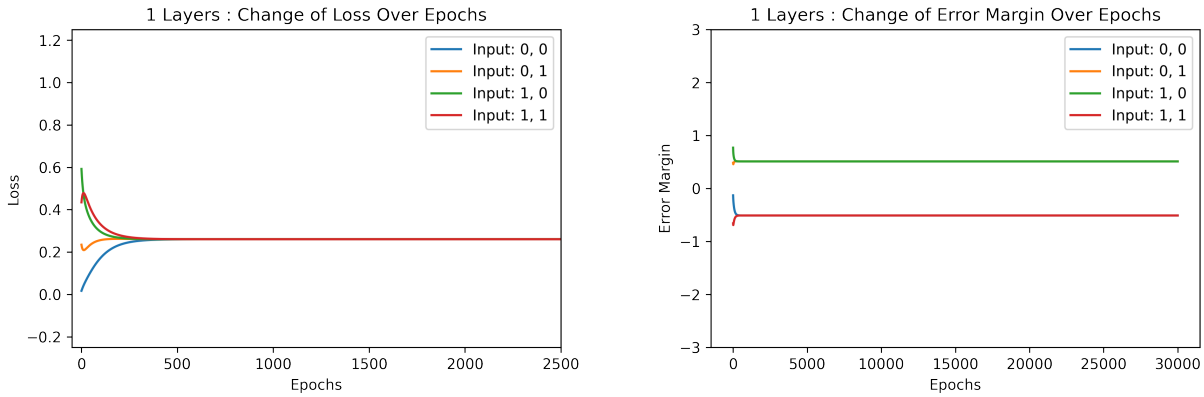


Figure 4: One layered neural network change in error margin and loss

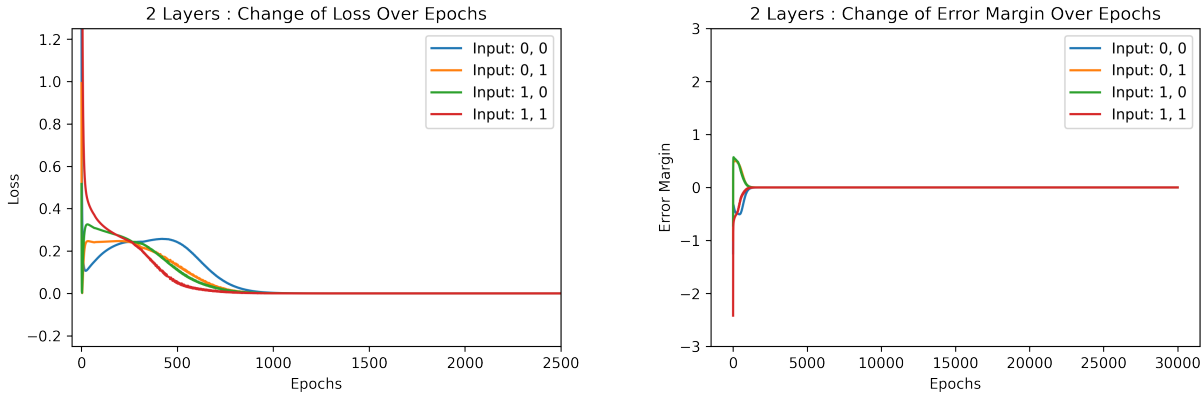


Figure 5: Two layered neural network change in error margin and loss

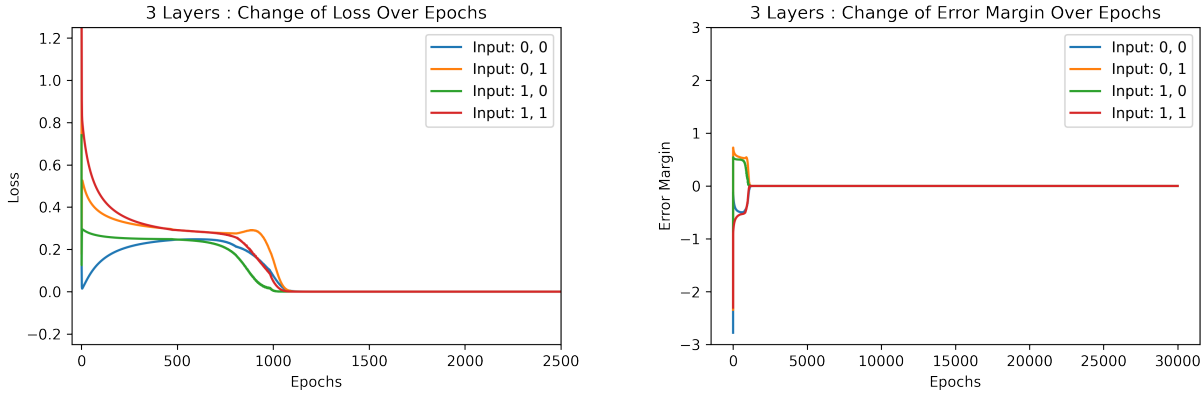
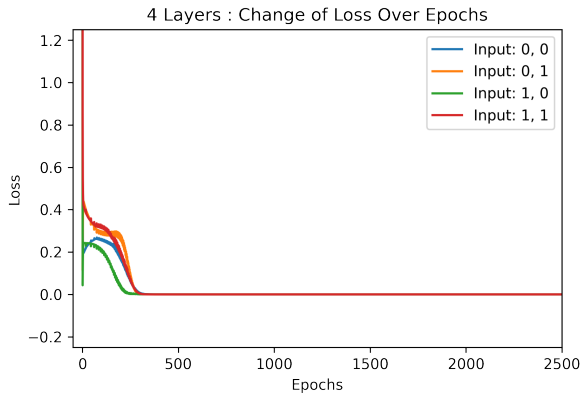


Figure 6: Three layered neural network change in error margin and loss



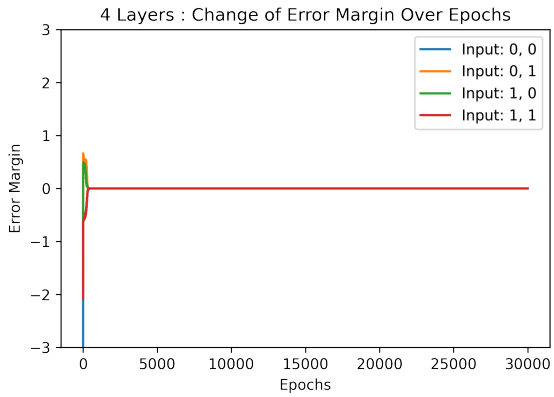
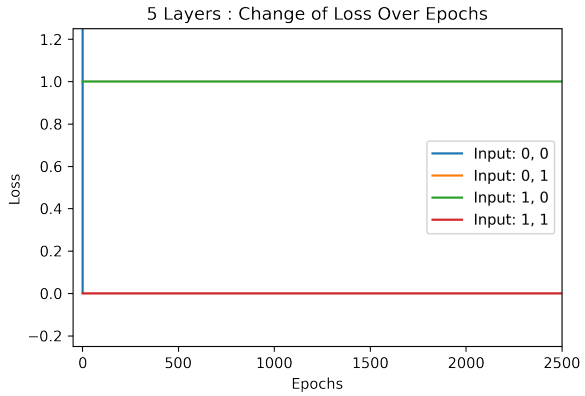


Figure 7: Four layered neural network change in error margin and loss



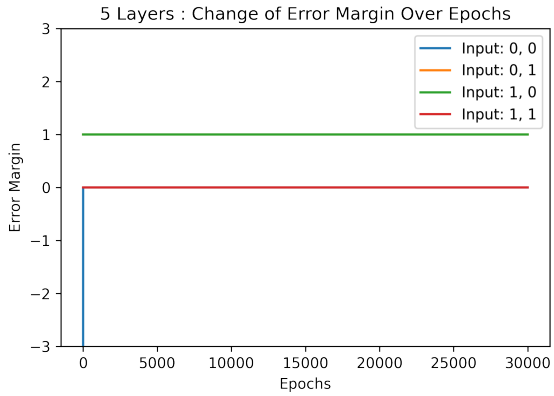
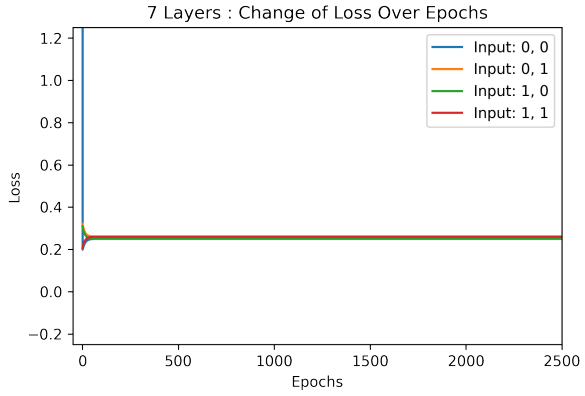


Figure 8: Five layered neural network change in error margin and loss



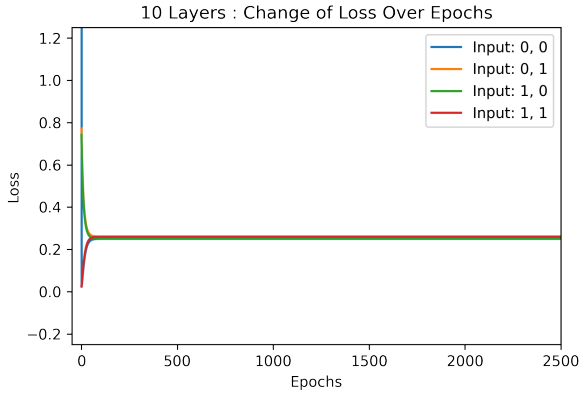


Figure 9: Seven and ten layered neural network change in loss

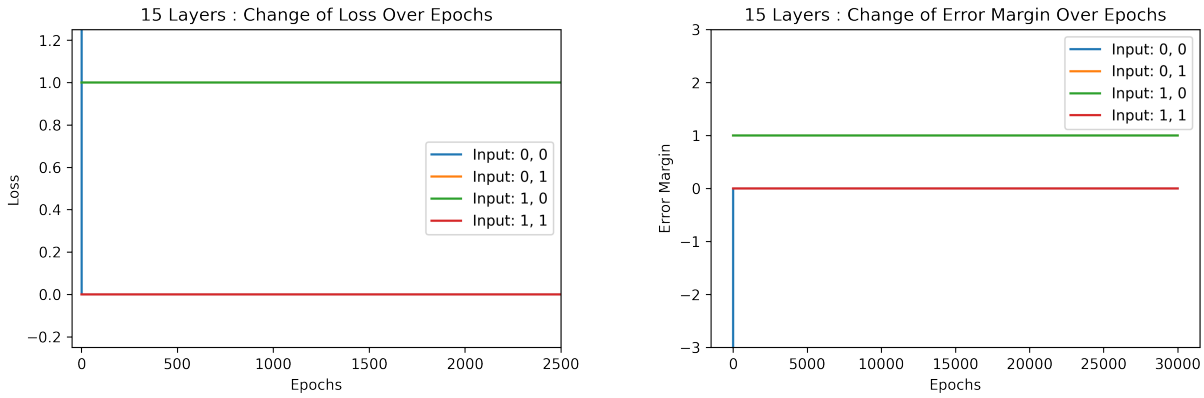


Figure 10: Fifteen layered neural network change in error margin and loss

4 Discussion

The need for two layers is clearly shown when comparing the results of figure 4 and figure 5, as having only one layer it just completely doesn't work. The predict function can be interpreted as a straight line, instead of $y = mx + b$ it is $\hat{y} = wx + b$ where weights is w, biases is b and x is an input (either a previous node or the initial fed in data). A one-layered neural network can be visualized as a single linear line on a plot, while two would be two linear lines and so forth. With trying to replicate an XOR gate, having one line to try and split the data into separate sections only containing their type is impossible; with two lines it is not. See figure 11 to see how this works. Having the second layer makes this problem solvable, and it shows this in the generated data.

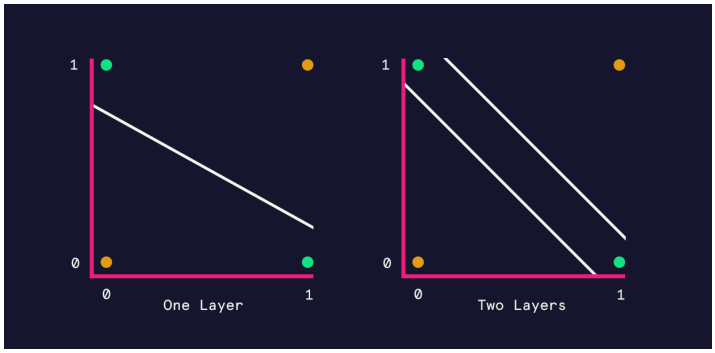


Figure 11: Visualization of the one-layer plot and two layer plot on the problem space

From layer two to four, the learning was continuing and working as intended. When we reached four, it learned extremely fast over a small number of epochs, this could be a potential cause of our main error. Four layers were the best for the least epochs while training.

There were issues discovered when increasingthe layer count past four, there are two possible ideas we had. The first is that the problem doesn't need to have a large number of layers for a very small problem; going back to the linear line idea, possibly with five lines and forward, it splits it further so the groups are further split. However, with seven layers, and ten layers, there was still some learning, and had the same change of loss as one layer. This could mean that it got to a point where it could split it in a similar way like the one layer does or that this isn't the correct analysis of what could have happened. Another option could be the fact that it was learning so fast, that at layer five there is no opportunity to actually learn anything, but looking at past five, ignoring seven and ten, they were still able to correctly predict inputs one-layerthat were both of the same value which is more than what a one-layer neural network can do. In the future, we should do similar tests, but for increaslingly more complex problems to see if this is a layer or learning issue.

5 Conclusion

One layer for an XOR problem is still impossible, having more than one layer for XOR will solve it for the most part. If you have too many layers or have a learning rate that is too fast it could cause the model to not solve it correctly. Four layers seem to be the best for this problem. In general, you would want to try

and have the appropriate amount of layers for your problem space, however, this is still tricky as you need to know what it is, and a learning rate which is also in the sweet spot for your model. In the future, we would retry this experiment on a more complex problem to see how the data changes.

6 References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[2] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[4] Ashish Sabharwal and Bart Selman. S. russell, p. norvig, artificial intelligence: A modern approach, third edition. *Artif. Intell.*, 175:935–937, 04 2011.

[5] Tomasz Szandała. Review and comparison of commonly used activation functions for deep neural networks. In *Bio-inspired Neurocomputing*, pages 203–224. Springer, 2020.

7 Appendix

Algorithm 1 Generate weights and biases
Input: *size* as int
Output: *weightList*[1... *size*] as an List of matrices, *biasList*[1... *size*] as an List of vectors

```
1:  procedure GenerateWeightsAndBiases
2:      weightList ← new List
3:      biasList ← new List
4:      if size = 1 then
5:          weightList[0] ← randMatrix(2, 1)
6:          biasList[0] ← randVertex(1)
7:      else
8:          for i ← 0 to size do
9:              weightList[i] ← randMatrix(3, 3)
10:             biasList[i] ← randVertex(3)
11:         weightList[0] ← randMatrix(2, 3)
12:         biasList[0] ← randVertex(3)
13:         weightList[size - 1] ← randMatrix(3, 1)
14:         biasList[size - 1] ← randVertex(1)
15:     return weightList, biasList
```

Algorithm 2 Predict output
Input: x as float, *weightList*[1... n] as an List of matrices, *weightList*[1... m] as an List of vectors
Output: $\hat{y}$ as float

```
1:  procedure Predict
2:      current ← ReLU(x × weightList[0]) + biasList[0]
3:      weightList ← weightList.remove(0)
4:      biasList ← biasList.remove(0)
5:      for i ← 0 to n do
6:          current ← ReLU(current × weightList[0]) + biasList[0]
7:      return current
```

Algorithm 3 Iteratively train up to 15 Layers

```
1: procedure TRAIN
2:     numEpochs ← 20000
3:     inputData ← Tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
4:     outputData ← Tensor([[0.], [1.], [1.], [0.]])
5:     lossFn ← MSELoss()
6:     for j ← 1 to 15 do
7:         weights, biases ← GenerateWeightsAndBiases(j)
8:         optimizer ← SGD(concat(weights, biases))
9:         for epoch ← 1 to numEpochs do
10:            for i ← 1 to 4 do
11:                x, y ← unsqueezeData(inputData, outputData, i)
12:                optimizer.zeroGrad()
13:                ŷ ← Predict(x, weights, biases)
14:                loss ← lossFn(ŷ, y)
15:                lossFn.backprop()
16:                oprimizer.step()
```