A Simple Understanding of Neural Networks

Introduction

Linear algebra is central to the development and execution of machine learning models (MLMs). We can initially begin to think of MLMs as a field of neurons. These neurons are realistically a value in between 0 and 1 that is stored somewhere inside of the network. A field of neurons is a group of neurons that are connected in a specific way by biases and mathematical functions explained late. These calculations in simple neural networks can be defined by a matrix vector multiplication model where the vector is the input values and the matrix is the biases of the network. The field of neurons will have different forms such as input neurons, which for our example is each pixel on the image being tested, output neurons, which represent what the neural network predicts is the output, and middle layer neurons, which are arbitrary neurons used in the calculation process. Given a certain number of input neurons, through certain mathematical linear transformation operations done by each "layer" of middle layer neurons, the output layer of neurons will correspond to the probability of each output being the correct answer from a 0-1 scale for each possible output.

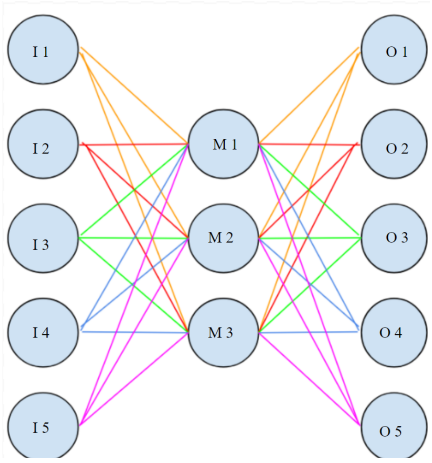


Figure 1. Models of a system

In the image to the left, the five input neurons (nodes I1-I5) would represent all the inputs to the neural network. The lines connecting these inputs to the next layer are essentially linear transformations acting on the input number and are a combination of a weight and bias value. Then, an arbitrary number of these "hidden" middle layers (nodes M1-M3) where further weights and biases are added to hopefully refine the result where finally it reaches the last output layer (Nodes O1-O5). This layer is responsible for outputting what it determines to be the probability of each possible output.

In this paper, we will be analyzing the mathematics of a 40x40 pixel set (1600 total input neurons) and passing them through a 5 neuron hidden layer to determine the probability that the number in the image represents some number from 0-9.

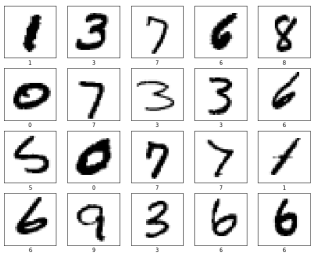


Figure 2, Examples from MNIST dataset - https://bit.ly/46lF1ai

Mechanism

We must first assign a grid of 40x40 pixels and assign them their corresponding neuron in our network where we will denote each neuron as a_i where i represents the i^{th} pixel of the canvas. Each pixel cannot be represented as a (x,y) coordinate because they must be represented in the input vector of neurons such that they are able to have operations performed on them. Each input neuron will be a decimal value from 0 to 1, where a_i=1 represents a fully

dark input (the pixel is entirely black) and a_i=0 would represent an off input (the pixel is entirely white and not colored). After this, we attach a “connection” from each of the input neurons to each of the 5 hidden neurons b_1-b_5 such that

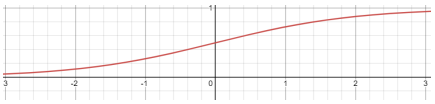
$$b_n = \sum_{i=0}^{40*40} a_i.$$

This essentially states that each neuron in the b layer is the sum of each of the a layers. However, this current model would result in each neuron in the corresponding column or layer having the same value of just the sum of all of the inputs. To fix this and introduce more variability in the outcomes, we can add a “weight” to each of the connections so that each neuron’s inputs are a weighted sum of each connected neuron before it. In addition to this, we can have a shift “bias” value that defaults each signal to a natural value, even if the weight factor is 0 (we can currently set both the weights and biases to be randomized numbers in range [0,1] that we will modify later). This then changes the equation for each node to be

$$\sum_{i=0}^{40*40} w_i a_i + b_i.$$

However, since the sum of this number can be an incredibly large number, we can compress the result to a value between 0 and 1 so it can be stored in the neuron by using the sigmoid function whose domain spans all of the real number line and favors values that are more decisive in output (this will be explained more later):

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



We hoped that the neural network would act identical to the way human brains decipher numbers. This method being that a hidden layer neuron would activate on a certain cluster of pixels. For instance, one of the hidden layer neurons would activate if there’s a loop in the top half of the image and another may activate if there’s a loop in the bottom half. Then the network decides that the image is an 8 if both of those neurons are active or in other words have a closer value to 1. If one of these neurons is off then 8 would not be “active” in the output neuron for 8.

One may note that the expression for a general total strength of a neuron of the hidden layer can be represented by the following vector equation:

$$\sigma(A\overline{w} + b)$$

This can be proved by performing vector multiplication on the first two matrices (representing the input nodes and the weights respectively) and adding it to the bias matrix. This will create a vector of the values for each node in the layer.

We can use this formula recursively on each node, getting the data from the layers behind them and performing the linear operation described above on each node in the current layer before repeating the process for the next layer. Performing this algorithm on our current model unsurprisingly yields random noise for the output probabilities currently.

How to Learn?

Since our weights and biases currently are not initialized to any meaningful numbers, we should expect randomized noise as output. This is the foundation of machine learning; optimizing these weights and biases to match an intended result. Before we can train this algorithm though, we need a numerical analysis to tell the network how "close" to the right answer it is. For this, we can implement a "cost" function where

$$C(v) = \sum_{i=0}^{k} (v_i - a_i)^2$$

Where k represents the amount of output nodes (10 in our example), v represents the output vector's result that our network supplied and a should be the actual vector result. Essentially, the cost function simply takes the difference between each output node's value and the expected value and squares that for each output before summing them all. This gives us a generalized number for how "bad" our network is; larger values of the cost function signify a worse network and a smaller cost value signifies a correct prediction. Therefore, if we can adjust the weights and biases such that it minimizes the cost function for all of our test cases, we would have trained this neural model.

Learning

Essentially, we've boiled down the problem of optimizing a neural network to a problem of finding the minimum of the cost function given the previous layer of weights and biases as arguments. Now, optimizing a function in hundreds of dimensions (since each weight/bias is a separate axis that needs optimizing) seems hard. But, if we analyze this as a study of a singular function, we have tools that make it easier.

Namely, we can start off at any point on a function. Then, we can take its derivative to calculate its slope at that point. From there, we move our point downwards in the direction of the slope continually until the slope evens out, at which point we've hit a local minimum. This is known as Newton's method.

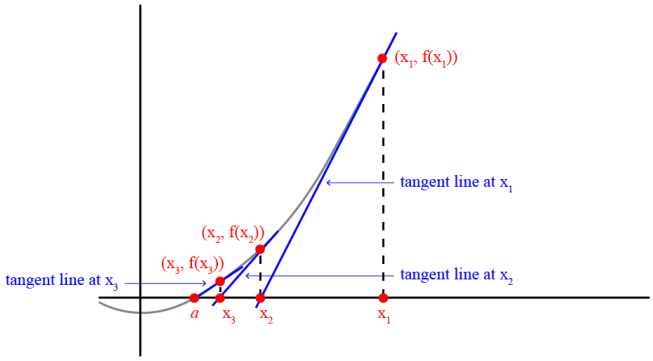


Figure 3, Calcworkshop

This same principle applies to linear vectors of greater dimensions, just in an n-dimensional space. Using this, we can try optimizing our weights and biases by finding the weights and biases that lead to the lowest possible cost value.

Calculating Gradient Descent with Vectors

In the instance of our model, we can assign a vector W to be the vector containing all of the weights and biases present in the whole model. The order of where the weights/biases are and what row they would be in doesn't matter as long as the position is correlated to the resultant vector of the operation we will perform. We will then define the Cost function C to be a function that takes in a weights/biases vector and returns the result of the neural network and the cost of the output. From here, we want to find the individual derivative of each weight/bias entry given a change in cost that we can call $\nabla C(W)$. This can be represented by the equation:

$$\nabla C(W) = [\partial C/\partial W_1, \partial C/\partial W_2, \ldots, \partial C/\partial W_n]$$

Where W_i is the ith entry of W and n is the amount of weights/biases in W. We call this value the gradient of C. Essentially, given a certain W value, it returns the vector that is the biggest possible step of C. Taking the negative of ∇C, likewise, returns the most downward step of C. So, we can say that we essentially want to compute

$$C(W) - \nabla C(W)$$

Until ∇C(W) = 0, at which point C has become a minimum. A way to compute this is to break C into its constituent dimensions, at which point we can calculate each dimension independently. To do this, we can create a matrix known as the Jacobian. This is where each entry in each column changes the W_i value of the derivative and each row value in each row changes the W_i value of the output.

$$J_C(W) = [\nabla C_1(W)^T, \nabla C_2(W)^T, \ldots, \nabla C_n(W)^T]$$

Where each entry corresponds to a different row. In this definition, $\nabla C_i(W)$ corresponds to the derivative of the ith index of W. The transpose is necessary to turn the vector columns into rows. We transposed this such that when J_C(W) is multiplied by a change in weight/bias vector ∇W, each column is $\nabla C_i(W)_i$ and will be multiplied by each entry of ∇W, producing a vector $\nabla C(W)$.

Now, we can finally construct an equation to find a weight vector W that minimizes C. We know what our initial W and $C(W)$ is. Therefore, we can calculate ∇W by noticing and rearranging the equation:

$$C(W) + J_C(W)\nabla W = 0$$

This equation is very similar to stating $y + \Delta x(\frac{dy}{dx}) = \Delta y$. We simply set Δy = 0 as a parallelism for setting ΔC to 0 to find a local minimum. We then exchange y for C and x for W. Next, we can notice that this equation can be solved for ∇W.

$$J_C(W)\nabla W = -C(W)$$

$$\nabla W = -J_C(W)^{-1} C(W)$$

Repeated iterations of $W + \nabla W$ will continue to give better values for W until locally optimized.

Limitations

We can repeat this process until the second term is zero, thereby optimizing the function. This should make sense since that second term is the n-dimensional movement towards a valley of our cost function. It should also be noted that this method will only find a local minimum.

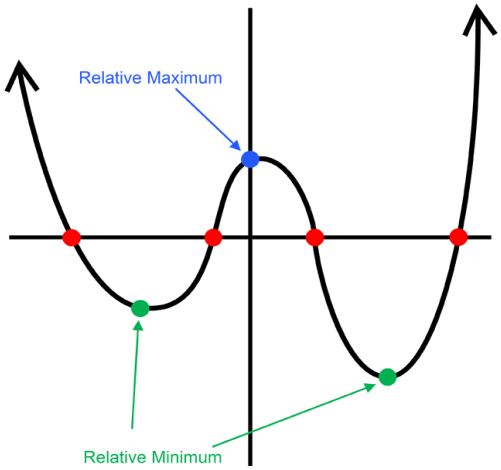


Figure 4, Calcworkshop

In the image above, the left minimum will be a valid convergence point for our cost function. Therefore, a differential initial random set of weights and biases may have to be picked or a larger jump size may have to be performed.

Conclusion

After the method of gradient descent is finished and the weights and biases are continually updated to minimize cost, this process must be repeated over lots of data sets to continually update the data set to better adjust to different pixel placements or other artifacts not accounted for in the first few data sets or iterations. More hidden layers or neurons can be added if the model fails to recognize and properly categorize objects outside of inputs closely matching other inputs. However, the tradeoff requires more processing power and time to train.

Works Cited

[1]
J. Krohn, Linear Algebra for Machine Learning. Pearson, 2020.
[2]
A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, “Compressed linear algebra for large-scale machine learning,” vol. 9, no. 12, pp. 960–971, 2016, doi: 10.14778/2994509.2994515.
[3]
A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, “Scaling Machine Learning via Compressed Linear Algebra,” vol. 46, no. 1, pp. 42–49, 2017, doi: 10.1145/3093754.3093765.