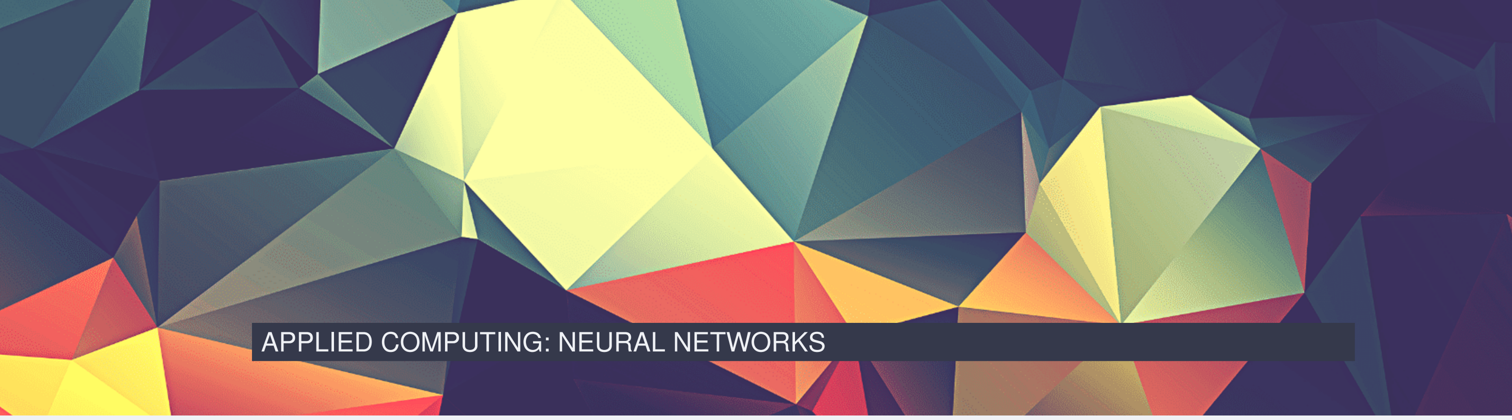


APPLIED COMPUTING: NEURAL NETWORKS

How Machines Learn from Backpropagation

Neural networks in simple terms

Areeb Shoaib
Y12 CSSOC member

The idea of neural networks roots deeply in the mathematical study multivariable calculus, making it inaccessible to most A-level students. This text studies a special case of neural networks, which allows concepts to be explained in terms of basic derivatives. This article provides an overview on the internals of a neural network, and how backpropagation can be used to train a network.

WHAT IS A NEUTRAL NETWORK

All a neural network is a collection of neurons that take in some input and give an output. The output can be simply calculated by multiplying the input a by some weight W, and adding on some bias b.

- The weight tells us how strong the given input will be.
- The bias is assigned to each output neuron and offsets the decision of a neuron, similar to how we add constants in a straight line graph $y = mx + c$ to better fit the model.

$$O_n = W_n a_{n-1} + b_n \tag{1}$$

$$\begin{tikzcd} a_{n-1} \arrow[r, "\times W_n"] & O_n \\ & b_n \arrow[u] \end{tikzcd}$$

Here is a simple example of a network with just one input neuron and one output neuron. However, we can have an entire network of these neutrons with multiple inputs and outputs, we can also have multiple layers.

Understanding neural networks

For networks with large numbers of layers and neurons, it is useful to use matrices to represent the values of inputs, outputs, weights and biases, as matrices are convenient for computers to computer.

Representation of layers

Each layer can be represented by 3 matrices.

- For inputs, we use an $N \times 1$ matrix, where N is number of neuron activations.
- For weights, we use an $M \times N$ matrix, where M is the number of neurons in the next layer.
- For biases, we can use a $M \times 1$ matrix.

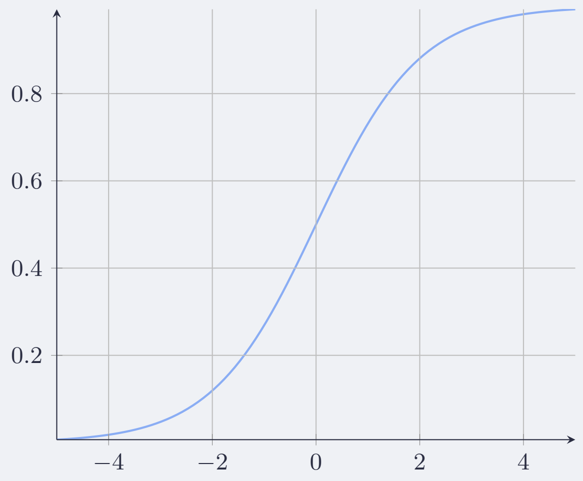
Each weight and bias will initially be randomly set, but as we train the network, they will approach some optimal value.

Using the formula $O = Wa + b$, we can see that O can be represented by an $M \times 1$ matrix.

The values for these simple calculations can vary widely, so we need a *step function* that allow the outputs O to be with a suitable range. For example, we can let any $O > 1.5$ to be equal to 1, and 0 otherwise. We will use the sigmoid function as it is smooth, which has a range between 0 and 1.

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

Changes in large inputs values have little effect on the output value, whereas changes in inputs values close to 0 have a greater effect on the output.



The sigmoid function can be differentiated, which will be important later, the derivation is left as an exercise for the reader.

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

Using the sigmoid function, inputs to the next layer can be defined as

$$a_n = \sigma(O_n) = \sigma(W_n a_{n-1} + b_n) \tag{2}$$

The cost function

The cost function tells us how wrong our neural network is when performing a particular task. When we train the network, we want to minimize this cost function as much as possible. This function outputs a value related to the difference between the produced output and the desired output.

For simplicity, this text will only consider neural networks where each layer contains only one neuron, such that each layer has only one input and one output. If you wish to build a complex neural network from scratch, you are recommended to learn about partial derivatives then watch the 3blue1brown series.

$$\begin{tikzcd}
a_0 \arrow[r, "\times W_1"] & O_1 \arrow[r, "\sigma(O_1)"] & a_1 \arrow[r, "\times W_2"] & O_2 \arrow[r, "\sigma(O_2)"] & a_2 \arrow[r, "\times W_L"] & O_L \arrow[r, "\sigma(O_L)"] & a_L \\
b_1 \arrow[ur] & & b_2 \arrow[ur] & & b_L \arrow[ur] & &
\end{tikzcd}$$

In this example we use the sum of squared differences between the final output of the network and the desired output. Squaring the differences gets rid of the negatives, and amplifies the large differences in the output. The cost function can be differentiated, which will be important later.

$$C = (a_L - y)^2 \tag{3}$$

where y is the desired output, and a_L is the final output after the sigmoid function was applied.

BACKPROPAGATION

Backpropagation is a method to minimise the cost function. It uses *stochastic gradient decent*, taking small steps to approach some local minimum for each individual weight and bias in the network. Where the steeper the gradient, the greater the step we take.

The idea follows that the cost function C is affected directly by a_L, and a_L is directly affected by O_L, which is affected by W_L, a_{L-1} and b_L. We can visualise this idea by constructing a cause-and-effect tree.

$$\begin{tikzcd}
W_1 \arrow[dr] & & W_2 \arrow[dr] & & W_L \arrow[dr] & & & \\
a_0 \arrow[r] & O_1 \arrow[r] & a_1 \arrow[r] & O_2 \arrow[r] & a_2 \arrow[r] & O_L \arrow[r] & a_L \arrow[r] & C \\
b_1 \arrow[ur] & & b_2 \arrow[ur] & & b_L \arrow[ur] & & &
\end{tikzcd}$$

Tracing backwards, we can see all tweakable parameters - the weights and the biases have an affect on the cost C. Some changes will a greater effect on C than others. For instance, a small change in W_L will have a greater effect on O_L when a_2 is large, since any change on O_L is multiplied by a_2.

We can use calculus to estimate what changes, and how much of that change has the greatest effect on reducing C.

From equation (3), we see the change in the cost function depends on the activation values passed from the last layer. To reduce the cost function we want to change a_L.

$$\frac{dC}{da_L} = 2(a_L - y)$$

From equation (2), we see the change in activation value depends on the output from the last layer. To change a_L we want to change O_L.

$$\frac{da_L}{dO_L} = \sigma'(O_L)$$

From equation (1), we see the output layer depends on both the weights of the last layer. To change O_L we want to change W_L.

$$\frac{dO_L}{dW_L} = a_{L-1}$$

By the chain rule, we can find gradient of the cost function and change the weights accordingly. The visual picture here is that we are ‘taking a small step’ towards the local minimum. Gradient for the bias b_L can be found using a similar equation.

$$\begin{aligned}
\frac{dC}{dW_L} &= \frac{dC}{da_L} \times \frac{da_L}{dO_L} \times \frac{dO_L}{dW_L} \\
&= 2(a_L - y) \times \sigma'(O_L) \times a_{L-1}
\end{aligned}$$

From equation (1), all W_n, b_n and a_{n-1} have an effect on O_n, the meat of backpropagation comes when we try to find out what changes need to be made to the activation value a_{n-1} from the previous layer. Since we cannot directly manipulate the activation values, we instead 'propagate backwards' and find how should the weights and biases from the previous layer be changed.

$$\frac{dC}{dW_{L-1}} = \frac{dC}{da_{L-1}} \times \frac{da_{L-1}}{dO_{L-1}} \times \frac{dO_{L-1}}{dW_{L-1}}$$

This process can be repeated for all layers. Essentially, *stochastic gradient decent* is a function that takes in a batch of training data and all the weights and biases as input, and outputs the changes needed to be made.

The process of repeatedly running backpropagation on training data, and applying the changes to the weights and biases will likely reduce the cost function when running on never-before-seen data. Training can be continued until reaching close to a local minimum where the neural network is able to consistently produce accurate outputs.

Editor's note

Recommended further readings are

- 3blue1brown's series on neural networks.
- Openstax Calculus, which is a great place to learn calculus from scratch.