Backpropagation Calculus in Neural Networks

Matthew Sears

December 5, 2017

Abstract

For my final project in Multivariable Calculus with Professor Rachel Maitra, I chose to study the backpropagation algorithm in the context of a simple feed-forward neural network with one hidden layer. In short, backpropagation is a machine learning technique that uses gradient descent to allow neural networks perform non-linear classification tasks. I will use this learning technique to show how such a neural network is capable of learning the XOR logical gate, a problem that is only solvable for such a network through the use of backpropagation.

1 Introduction

To begin with, the XOR classification task is non-linear because basically it's impossible to use a single line to separate our two groups. In this case of XOR, the gate outputs a 1 if and only if one of the two inputs is a 1, and 0 otherwise; referencing the right side of Figure 1, the two groups we have to classify for this case is when it should output a 1, labeled as the green dots, and when it should output a 0, labeled as the red dots. If XOR were linear, we would be able to separate the green dots from the red dots with one single line.

As a side note, focusing on the XOR case allows us to ignore the confusing notation that naturally comes with the generalized form of neural networks, and although I love linear algebra, it is only a distraction in a case as simple as this; therefore, the use of explicit notation, and the stripping of linear algebra, will make the learning curve towards understanding the heart of such a profound machine learning technique a lot less steep. Additionally, the XOR problem is a particularly famous one in neural networks, and I recommend anyone who's interested to read how XOR helped develop (and almost destroyed) the field of neural networks. [See "Explained: Neural Networks" or Minsky & Papert]

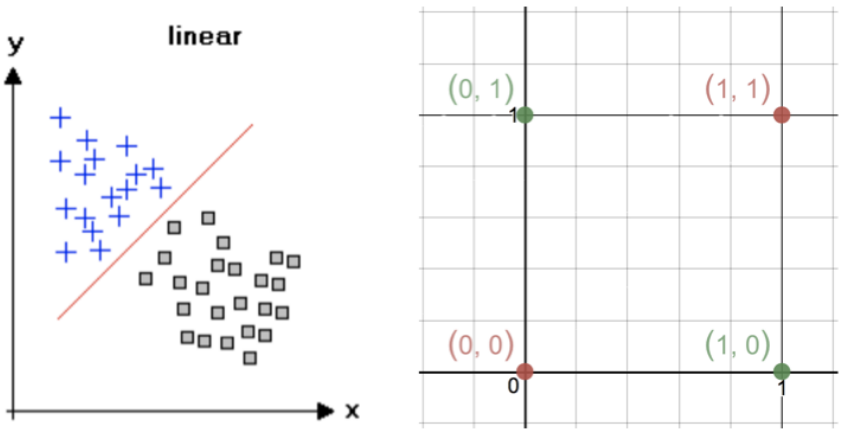

Figure 1: The left side shows a group of data which can be classified linearly, whereas the right side (XOR) is impossible to classify linearly. For XOR, the cartesian coordinates represent the two inputs into the gate.

2 Backpropagation

2.1 Activation

Since this situation isn't linear (as with most natural phenomena), we introduce a non-linearity, or activation function, into the network to avoid it calculating linear combinations of values as if it were a linear classification task. This is what allows our neural network to successfully begin learning such non-linear phenomena, thus giving it the capability to perform non-linear classification. We use the sigmoidal activation function, as seen in Figure 2, which squashes and maps our inputs to the range $(0, 1)$.

This function also serves as an abstract representation of the action potential process that takes place during the firing in a neuron:

$$\varphi_x = \frac{1}{1 + e^{-x}}; \quad \varphi(x) \coloneqq \varphi_x$$

which has a neat derivative of:

$$\varphi_x(1 - \varphi_x).$$

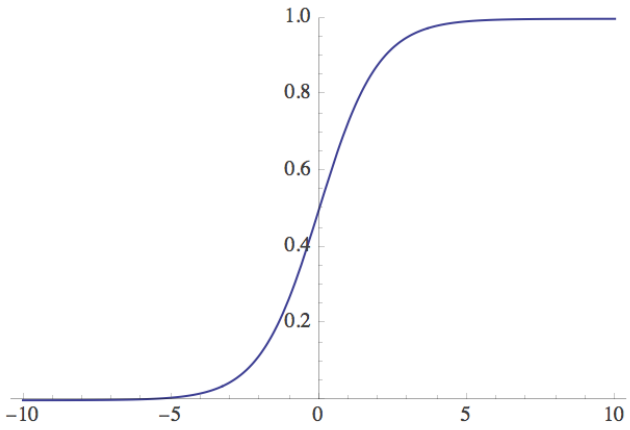

Figure 2: The sigmoid activation function plotted in Mathematica.

2.2 Multivariable Calculus & Interpretation

The cost function, or error of this network, which we want to minimize is:

$$E = \frac{1}{2}(y - \hat{y})^2 \tag{1}$$

where y is the correct output and $\hat{y}$ is the network's output. To minimize our error, we have to adjust each of the network's weights until we get the desired output. This is done through backpropagation, using the negative gradient to find the steepest descent towards the local minimum of the error for each weight. Referencing Figure 3, this gradient is defined as a vector whose indices are the rate of change in the direction of steepest descent for every weight:

$$-\nabla E \equiv -\left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \frac{\partial E}{\partial w_{11}}, \ldots, \frac{\partial E}{\partial w_{22}}\right] \tag{2.1}$$

We use the training rule defined as follows to update all of our weights (defined here generally as w) using the corresponding gradient index which holds that specific weight:

$$\Delta w = -\alpha \frac{\partial E}{\partial w}; \quad \alpha \in (0, 1)$$

where α is the learning rate, or how fast the algorithm will try to approach the local minimum.

Referencing Figure 3, this entire network can be defined as follows:

$$\begin{aligned} \hat{y} &= \varphi(\text{net}_{\hat{y}}) \\ \text{net}_{\hat{y}} &= \varphi(\text{net}_1)w_1 + \varphi(\text{net}_2)w_2 \\ \text{net}_1 &= i_1 w_{11} + i_2 w_{21} \\ \text{net}_2 &= i_1 w_{12} + i_2 w_{22} \end{aligned}$$

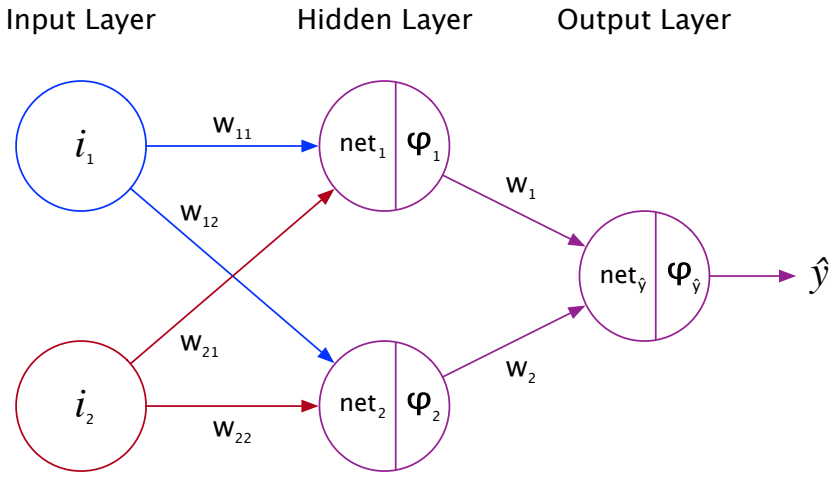

Figure 3: XOR Neural Network Diagram that I made in OmniGraffle.

This notation will be used when computing the partial derivatives, as to avoid confusion between what is a function and what is multiplication:

$$\begin{aligned} \varphi(\text{net}_{\hat{y}}) &:= \varphi_{\hat{y}} \\ \varphi(\text{net}_1) &:= \varphi_1 \\ \varphi(\text{net}_2) &:= \varphi_2 \end{aligned}$$

Since this is a multivariable function, we have to apply the chain rule to take into account everything that's a function of the hidden layer weights:

$$\Delta w_{\hat{y}} = -\alpha \frac{\partial E}{\partial w_{\hat{y}}} = -\alpha \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \text{net}_{\hat{y}}} \frac{\partial \text{net}_{\hat{y}}}{\partial w_{\hat{y}}} \tag{2.2}$$

Equation 2.2 will be used for the proposed adjustments of these two weights, $w_{\hat{y}} \in \{w_1, w_2\}$, which are directly connected to the output node where the partials are:

$$\begin{aligned} \frac{\partial E}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}} \left(\frac{1}{2}(y - \hat{y})^2 \right) = -(y - \hat{y}) \\ \frac{\partial \hat{y}}{\partial \text{net}_{\hat{y}}} &= \frac{\partial}{\partial \hat{y}} \left(\varphi_{\hat{y}} \right) = \varphi_{\hat{y}}(1 - \varphi_{\hat{y}}) \\ \frac{\partial \text{net}_{\hat{y}}}{\partial w_{\hat{y}}} &= \frac{\partial}{\partial w_{\hat{y}}} \left(\varphi_1 w_1 + \varphi_2 w_2 \right) = \varphi_1 \quad \text{or} \quad \varphi_2 \end{aligned}$$

$$\therefore$$

$$\begin{aligned} \Delta w_1 &= \alpha(y - \hat{y})\varphi_{\hat{y}}(1 - \varphi_{\hat{y}})\varphi_1 \\ \Delta w_2 &= \alpha(y - \hat{y})\varphi_{\hat{y}}(1 - \varphi_{\hat{y}})\varphi_2 \end{aligned}$$

The only noticeable difference between these two weight changes is which net input was activated prior to it. Thus begins our backpropagation!

For the four weights directly connected to the input nodes we have to extend the chain rule even further to account for everything that's a function of the inputs:

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = -\alpha \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \text{net}_{\hat{y}}} \frac{\partial \text{net}_{\hat{y}}}{\partial w_{\hat{y}}} \frac{\partial w_{\hat{y}}}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial w_{ij}}$$

where w_{ij} represents the j^{th} weight directly connected to the i^{th} input. Using Equation 2.2 simplifies this to:

$$\Delta w_{ij} = -\Delta w_{\hat{y}} \frac{\partial w_{\hat{y}}}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial w_{ij}} \tag{2.3}$$

Equation 2.3 will be used for the proposed adjustments of these four weights, $w_{ij} \in \{w_{11}, w_{12}, w_{21}, w_{22}\}$ which are directly connected to the input node. Basically, we're looking at each of the four paths that need to be taken in order to reach the original inputs with our starting point being our output (as color-coded in Figure 3). Recall that:

$$\begin{aligned} \text{net}_1 &= i_1 w_{11} + i_2 w_{21} \\ \text{net}_2 &= i_1 w_{12} + i_2 w_{22} \end{aligned}$$

Additionally, since the weights connected to the output neuron are influenced by their respective activated net inputs:

$$\frac{\partial w_{\hat{y}}}{\partial \text{net}_i} = \frac{\partial}{\partial \text{net}_i} (\varphi_i) = \varphi_i (1 - \varphi_i)$$

$$\therefore$$

$$\begin{aligned} \Delta w_{11} &= -\Delta w_1 \ \varphi_1 (1 - \varphi_1) i_1 \\ \Delta w_{12} &= -\Delta w_2 \ \varphi_2 (1 - \varphi_2) i_1 \\ \Delta w_{21} &= -\Delta w_1 \ \varphi_1 (1 - \varphi_1) i_2 \\ \Delta w_{22} &= -\Delta w_2 \ \varphi_2 (1 - \varphi_2) i_2 \end{aligned}$$

The noticeable differences between these four weight changes are the actual input itself in conjunction with which hidden node the weight's attached to.

3 Conclusion

Focusing on the XOR case, as opposed to tackling the generalized form of neural networks, provides a more easily-digestible intuition for how backpropagation works in neural networks. The network literally propogates backwards along every possible path connecting the output node to the input nodes in order to discover the right values for each individual weight. Over multiple gradient descent iterations, often referred to as epochs, this network will successfully converge towards a local minimum of the multivariable error function, therefore learning the XOR gate. This means that you could input any combination of two binary numbers and it would successfully return the correct XOR response to such inputs without looking up a hard-coded truth table. This non-linear classification task was previously impossible for feed-forward networks due to their inherently linear structure. Without the invention of the backpropagation algorithm, neural networks would've been completely disregarded only a few years after their inception. Within the last decade, the immense strides in neural networks gave birth to a technique called deep learning, now implemented within almost every cutting-edge artificial intelligence system imaginable.