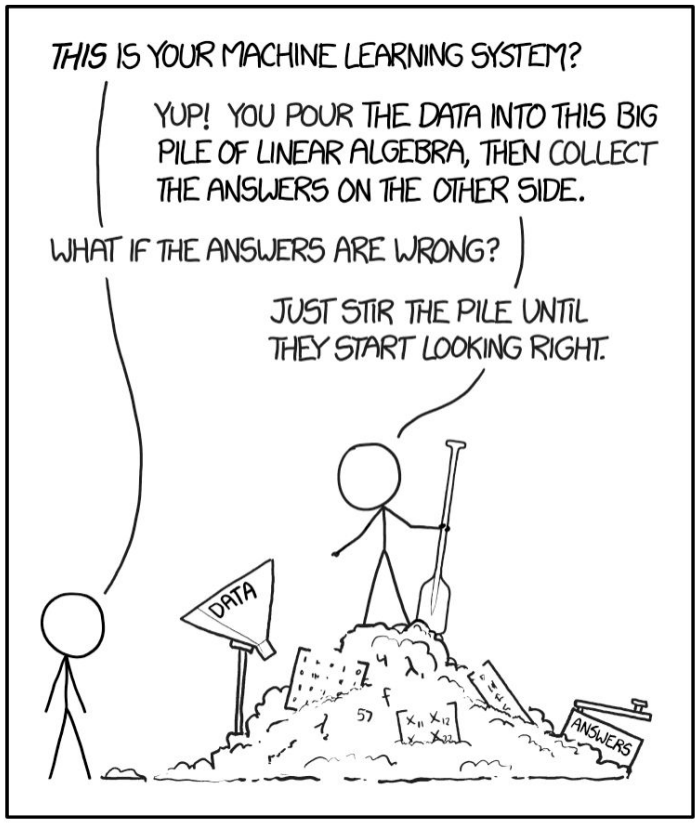
Machine Learning with Neural Networks and Backpropagation



Jane Sieving & Katie Thai-Tang

December 10, 2018

Olin College of Engineering

Linearity II

Abstract

Machine learning has risen to the forefront of discussions surrounding technology and computer science in recent years, but it is uncommon for the average person to understand what makes machine learning possible. In this paper, we explore the math behind back propagation as a method of machine learning. We also explain how we modified existing code to create a simple game where the computer learns to classify patterns of squares based on a rule made up by the player.

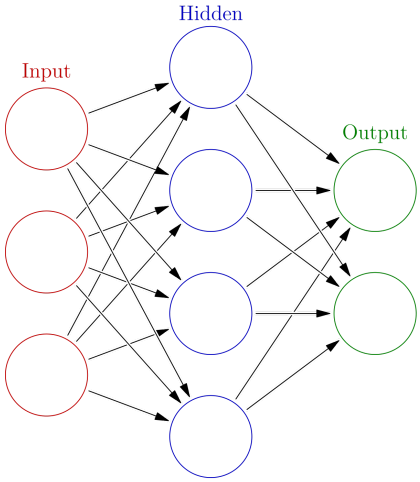
Introduction

Google searches for machine learning have increased dramatically in the last few years. There's no denying that terms like 'machine learning', 'algorithms', and 'neural networks' are becoming more and more common as tech industry buzzwords. Even though the terms are becoming more well known, most people don't really know how they work. Going into this project, Jane referred to our topic of backpropagation as, "a hand-wavey computer science concept," which seems like an accurate way to describe the average person's understanding of the topic. Our goal for this project was to be able to understand the math behind the concept of backpropagation as machine learning and modify existing code in order to teach a computer to play a simple game.

Background

To understand backpropagation, we first have to get a baseline understanding of artificial neural networks. Artificial neural networks refer to a computer system, inspired by the human brain, that functions as a framework for machine learning algorithms to process large amounts of data.

An artificial neural network consists of an interconnected group of nodes that can be separated into the input layer, output layer, and hidden layers. Each of the connections between the nodes has a specific weight and bias that indicates how important that decision is to the output. In order for the neural network to make an accurate decision, it must be trained with a dataset of known inputs and outputs; this is what the term 'machine learning' refers to. Backpropagation is the method used to train the neural network on the training set of data.



Backpropagation is a shortened version of "the backward propagation of errors", and calculates the gradient of error in order to adjust the weights of a neural network. This section will explain the general process of backpropagation, and the math will be explained in more depth in the following section. Conceptually, here's what's happening: With the provided training example, the designated input values are fed forward through the neural network and assigned weights that are essentially random. When an output value is reached, it is compared to the known output for that training set and a total error value is calculated. Using this value and partial derivatives of the error function, we can find a new set of weights which attempt to minimize the error between the intended and resulting outputs. This process is repeated until the neural network is able to give the correct output for a given input with acceptably little error. That means the neural network should be able to correctly determine outputs for new data that doesn't have a known output.

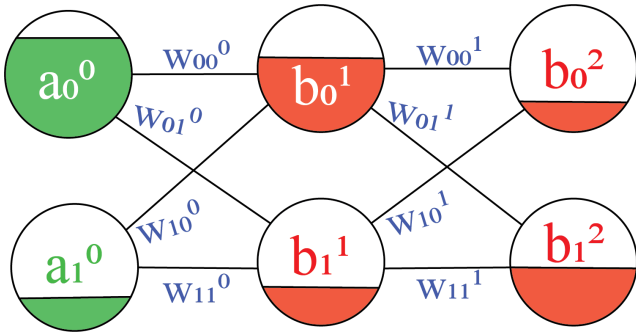
Neural Network Mechanics

A neural network consists of several layers of nodes: an input layer, one or more hidden layers, and an output layer. The input layer has exactly as many nodes as input values (in small-scale image processing, the input layer is generally the number of pixels). The hidden layer is less strictly defined. Generally, each node in a hidden layer can act as a "feature detector" neuron, and more complex classification tasks require more layers, although a lot can be done with one or two. The output layer has as many nodes as the number of possible outputs, with the result often considered to correspond to the node with the highest output value.

The input that each node passes on to the next layer is called the activation. The input data is taken as the activation of the first layer. To get the activation of the next layer, the previous activation from each node is multiplied by a corresponding weight. The sum of the weighted activations then has the receiving node's bias added to it. A node with a negative bias is biased against firing, creating a smaller activation, and vice versa.

$$a_j^L = b_j^L + \sum_{i=0}^{nL-1} (a_i^{L-1} * w_{ij})$$

Translation: The activation of a node equals its bias plus the sum of weighted preceding activations.

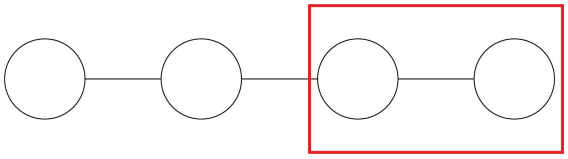


In the example network on the right, each layer has 2 nodes. a_0^0 and a_1^0 are the inputs.

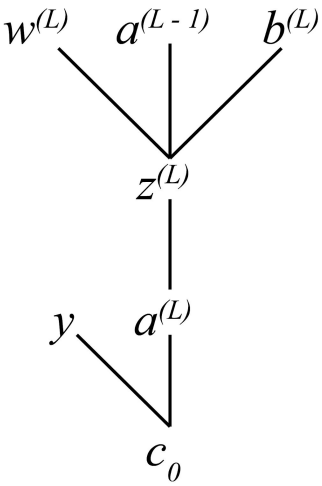
a_0^1 is computed *as* $a_0^0 * w_{00}^0 + a_1^0 * w_{10}^0 + b_0^1$. a_0^2 is computed as $a_0^1 * w_{00}^1 + a_1^1 * w_{10}^1$.

Backpropagation

In order to explain the math behind the backpropagation algorithm, we will be looking at a highly simplified example of a neural network with four layers, and one neuron in each layer. This is the method used in Chapter 4 of 3Blue1Brown's video series on the topic of machine learning and backpropagation. Since there are four neurons, there are three connections between them, indicating that this neural network is based on three biases, b_1, b_2, b_3, and three weights, w_1, w_2, w_3. The goal of backpropagation is to determine how much of an effect each of these elements has on the cost function, $C(w_1, b_1, w_2, b_2, w_3, b_3)$.



Focusing on only the last neuron, we can label the activation as $a^{(L)}$ to mark which layer it is in,



and the activation of the neuron before can be marked as $a^{(L-1)}$. In our training example, we can call our desired or expected output y. To find the cost for this one training example we would have $C_0(...) = (a^{(L)} - y)^2$. To get the activation of the last neuron, $a^{(L)}$, we calculate the weighted sum $z^{(L)}$ by multiplying the activation of the previous neuron by some weight, and add on some bias, giving us: $z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$. Then we run that weighted sum through some special non-linear function like the sigmoid, represented by σ. The purpose of the sigmoid function is to map any output value into a range from 0 to 1. So, $a^{(L)} = \sigma(z^{(L)})$. We can think of this in the form of this graphic, where the previous activation, weight, and bias influence the weighted sum, which influences the new activation, which, with the

desired input, gives you the cost. Our goal, is to figure out how sensitive the cost is to small changes in our weight, which can be written as, $\partial C_0/\partial w^{(L)}$, or the derivative of the cosrt with respect to the weight. We can break this down using the chain rule, so that $\partial C_0/\partial w^{(L)} = \partial z^{(L)}/\partial w^{(L)} * \partial a^{(L)}/\partial z^{(L)} * \partial C_0/\partial a^{(L)}$. Now we can work out the derivatives that make up the chain rule, which come out to:

$C_0 = (a^{(L)} - y)^2 \Rightarrow \partial C_0/\partial a^{(L)} = 2(a^{(L)} - y)$

$a^{(L)} = \sigma(z^{(L)}) \Rightarrow \partial a^{(L)}/\partial z^{(L)} = \sigma'(z^{(L)})$

$z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)} \Rightarrow \partial z^{(L)}/\partial w^{(L)} = a^{(L-1)}$

Which when combined using the chain rule give you,

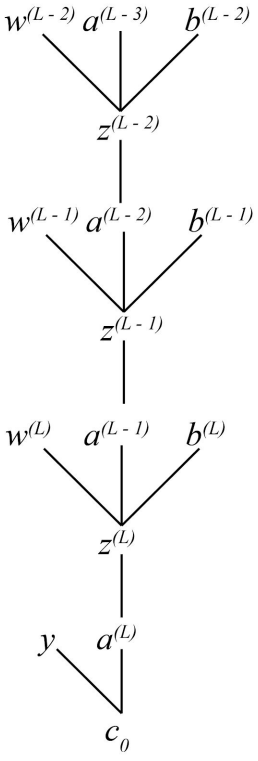
$\partial C_0/\partial w^{(L)} = 2(a^{(L)} - y) * a^{(L-1)} * \sigma'(z^{(L)})$

We also need to find the derivative of the cost with respect to the bias, which can be done in a very similar process, replacing the $\partial w^{(L)}$ term with $\partial b^{(L)}$, which in the end comes out to

$\partial C_0/\partial b^{(L)} = 2(a^{(L)} - y) * \sigma'(z^{(L)})$

When solving for the derivative of the cost with respect to the previous activation, we get

$\partial C_0/\partial a^{(L-1)} = 2(a^{(L)} - y) * w^{(L)} * \sigma'(z^{(L)})$, which we cannot directly change, but tells us that we can continue to iterate this chain rule concept backwards through each layer to see how sensitive the function is to previous weights and biases. This is what backpropagation refers to, working backwards through the layers of the neural network, determining the sensitivity of each weight and bias on the overall cost. In a more complex example, where



each layer of the network has more than one neuron, the process is almost exactly the same. The main change that is made is that there are more indexes to follow. For example, we can index layer L with j so the activation of a neuron in the last layer would be $a_j^{(L)}$. Similarly, we can index the second to last layer with k, giving us an activation of $a_k^{(L-1)}$ for a neuron in the second to last layer. Therefore, the new cost function would be $C_0 = \sum_{j=0}^{nL-1} (a_j^{(L)} - y_j)^2$, the sum of the squares of the differences of the layer activations and the desired outputs. Since there are now many more weights and biases between each layer, we also have to add indexes to each of their labels. For example, the weight between the kth and jth neuron would be $w^{(L)}_{jk}$. In addition, finding the derivative of the cost with respect to the previous activation changes because the previous neuron now influences the layer through multiple paths. Therefore, when finding that derivative you have to take the sum over layer L.

$$\partial C_0/\partial a_k^{(L-1)} = \sum_{j=0}^{nL-1} (\partial z_j^{(L)}/\partial a_k^{(L-1)} * \partial a_j^{(L)}/\partial z_j^{(L)} * \partial C_0/\partial a_j^{(L)}$$

Once again, the chain rule process is repeated, but this time taking into account each of the neurons in each layer. Each of the derivatives we found that tell us the sensitivity to each weight and bias, is a component of the gradient that, through gradient descent, helps minimize the cost of the overall network.

Algorithm Structure

The machine learning algorithm that we used has 3 important functions, which will be described in the following paragraphs. The particular algorithm we examined uses Stochastic Gradient Descent (SGD), which is distinct from regular gradient descent because it randomly distributes

the training data into mini-batches and computes the gradient for each piece of training data, rather than computing the gradient for the whole of the data at once. Specifically, this algorithm creates mini-batches of data, computes the gradient for each pair in a batch, averages these, and "descends" by one step after each batch has been processed. This type of algorithm is good for processing large data sets, and our grid game is simple enough that reducing computational load in this way is not required. We chose to examine it because it was straightforward and designed as a teaching example, so it was easy for us to dissect the code and connect it directly to the math we were learning.

The SGD() function takes a body of training data (a list of x, y tuples where x is an input vector and y is an output value), a number of cycles, a batch size, a learning rate, and optionally test data. It breaks the training data into batches and passes that to the Update Weights & Biases function to improve the weights and biases in the network. If test data was supplied, it calls the Self-Evaluation function on the test data after each training cycle and prints the number correct.

The batch_update() function takes a batch of training data and a learning rate (eta) as input. It initializes the gradient for every weight and bias value as zero, then adds the gradient calculated by the Backpropagation function for every input/output pair in the current batch. When it has totaled all the gradients, it divides the result by the batch size to get the average gradient. It multiplies it by the learning rate to get a small step in the right direction, then subtracts it from the current weights and biases to descend the gradient.

The backprop() function calculates the gradient of the cost function with respect to the weights and biases. Gradient with respect to weights and gradient with respect to biases are treated separately for convenience of data handling, but this has no effect since every weight and bias is essentially an independent dimension. It starts with the input data and the current weights and biases and computes the z values and activations for every node as the data is fed through the network (this is the feed-forward part of the process, although it is done without using the previous Feed Forward function.) Once the final outputs have been computed, the function has everything it needs to compute the gradient for that piece of data. It iterates from the final output layer backwards, using the chain rule as described previously to compute the gradient of the cost with respect to each weight and bias value. Instead of calculating with a single weight or bias, in each layer it is calculating with a vector of biases and a matrix of weights, but the math remains the same for each one. Backpropagation returns the new gradients for each weight and bias, which will be averaged within the batch.

These are the most important mathematical functions that make up the network() object, which has other functions for simple things such as the sigmoid and displaying the current accuracy of the network. The code we wrote uses pygame to display a 3x3 grid of squares. First, 6 random patterns are shown, and the user is asked to mark them as 0 or 1. With this as the first bit of training data, the computer then asks the user to draw patterns and tries to guess their classification. The user is asked to correct the computer after each guess, so the computer can update the weights and biases of the neural network to make gradually better guesses.

References:

1. "Artificial Neural Network". *Wikipedia.org*. https://en.wikipedia.org/wiki/Artificial_neural_network
2. "Backpropagation". *Brilliant.org*. https://brilliant.org/wiki/backpropagation/
3. Martin, Carlos & Schuermann, Lucas. February 27, 2016. "The Math behind Backpropagation". *Big Theta*.
4. Mazur, Matt. March 17, 2015. "A Step by Step Backpropagation Example". https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
5. Munroe, Randall. "Machine Learning". *Xkcd*. https://xkcd.com/1838/
6. Nielsen, Michael A. 2015. *Neural Networks and Deep Learning*. Determination Press.
7. Skalski, Piotr. August 17, 2018. "Deep Dive into Math Behind Deep Networks". *Towards Data Science*. https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba
8. 3Blue1Brown. November 3, 2017. "Backpropagation Calculus". https://www.youtube.com/watch?v=tIeHLnjs5U8&t=189s

Appendix:

Rather than weigh this section down with code, the link to our GitHub repository with the relevant code is: https://github.com/jsieving/Lin-2-Backpropagation