Back propagation for Deep Feedforward Neural Networks

Aaron Avram

June 10 2025

Introduction

In this write up I will go through the derivation of the back propagation algorithm for deep feed-foward neural networks and discuss how I will implement it using NumPy.

Set up

We are working with a classic neural net set up where we have $n+1$ layers $\{x^{(0)}, \ldots, x^{(n)}\}$ with the $i-th$ layer having l_i neurons. We then define a weight matrix and bias vector for each of the n non-input neurons. $W^{(i)}_{jk}$ is the weight of the k-th neuron in the $i-1$th on the jth neuron in the ith layer. And, $b^{(i)}_j$ is the bias associated with the jth neuron in the ith layer. Denote the activation function by σ (which could be tanh for example). Note that when σ is applied to a vector we do so element wise. Now we can represent the forward pass between each layer succinctly where

$$x^{(i)} = \sigma(W^{(i)}x^{(i-1)} + b^{(i)})$$

It will be convenient later to denote $z^{(i)} := W^{(i)}x^{(i-1)} + b^{(i)}$, so that $x^{(i)} = \sigma(z^{(i)})$. Note, that there may be some transformation on the output layer so that it represents a probability distribution (I.e. applying the softmax), but this is trivial in the context of back propagation calculations so it won't be included here.

Now when training this model on some input, we optimize its performance by minimizing some loss function, which measures some numerical representation of how well the model performs on the training data. Formally: $L : \mathbb{R}^{(l_n)} \to \mathbb{R}$. In the following calculations we will take gradients and derivatives of the loss function, and so we will assume that all of the mathematical functions involved are differentiable, which is practically always the case. Now, to train our model we want to find a method for computing the gradient of the weights and biases with respect to our loss function then perform gradient descent. Back propagation is such a method.

The Plan

The back propagation algorithm is essentially a fancy application of the chain rule from calculus. Namely the chain rule states that for a composition of functions f, g, h we have:

$$\frac{df(g(h))}{dx} = \frac{df}{dg}\frac{dg}{dh}\frac{dh}{dx}$$

In the multivariable case we apply this to partial derivatives for each component of input and output respectively.

Clearly our neural network is just a very large and complicated function and to get the gradients of the weights and biases, we can apply the above concept. This looks similar to dynamic programming where we break down the task of finding the complicated gradient of the weights of the network into a composition of the gradients of directly connected components of the network.

Back Prop

Now, we will build up our gradients step by step, working backwards from the output layer.