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 recursively working backwards from the output layer.

Since our loss function has scalar output, the gradient of the output layer with respect to the loss is straight forward:

$$\nabla_{x^{(n)}}(L) = \begin{bmatrix} \frac{\partial L}{\partial x_1^{(n)}} \\ \cdots \\ \frac{\partial L}{\partial x_{l_n}^{(n)}} \end{bmatrix}$$

Then for any layer $x^{(i)}$ we can consider the Jacobian of $z^{(i)}$ with respect to the output. However since $z_j^{(i)}$ only effects $x_j^{(i)}$ the Jacobian will be diagonal, so we can just write it as a gradient vector (abusing notation slightly).

$$\nabla_{z^{(i)}}(x^{(i)}) = \begin{bmatrix} \frac{\partial x_1^{(i)}}{\partial z_1^{(i)}} \\ \cdots \\ \frac{\partial x_{l_i}^{(i)}}{\partial z_{l_i}^{(i)}} \end{bmatrix}$$

Additionally we know that $x^{(i)} = \sigma(z^{(i)})$, so $\nabla_{z^{(i)}}(x^{(i)}) = \sigma'(z^{(i)})$. Next we want to conisder the Jacobian of $x^{(i-1)}$ with respect to $z^{(i)}$, but notice that $z^{(i)} = W^{(i)}x^{(i-1)} + b^{(i)}$ so $\frac{\partial z_j^{(i)}}{\partial x_k^{(i-1)}} = W_{jk}^{(i)}$. Thus:

$$J(z^{(i)})(x^{(i-1)}) = \begin{bmatrix} \frac{\partial z_1^{(i)}}{\partial x_1^{(i-1)}} & \cdots & \frac{\partial z_{l_i}^{(i)}}{\partial x_1^{(i-1)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1^{(i)}}{\partial x_{l_{i-1}}^{(i-1)}} & \cdots & \frac{\partial z_{l_i}^{(i)}}{\partial x_{l_{i-1}}^{(i-1)}} \end{bmatrix}$$

$$= (W^{(i)})^T$$

Abusing notation again we will call this $\nabla_{x^{(i-1)}}(z^{(i)})$. Now we can combine the following results to find the Jacobian of $x^{(i-1)}$ with respect to $x^{(i)}$. Observe that $\frac{\partial x_j^{(i)}}{\partial x_k^{(i-1)}} = \frac{\partial x_j^{(i)}}{\partial z_j^{(i)}} \frac{\partial z_j^{(i)}}{\partial x_k^{(i-1)}}$ and we know both of these values. Thus, The Jacobian of $x^{(i-1)}$ with respect to $x^{(i)}$ is just the Jacobian of $x^{(i-1)}$ with respect to $z^{(i)}$ with each row scaled by $\nabla_{z^{(i)}}(x^{(i)})$. So:

$$\nabla_{x^{(i-1)}}(x^{(i)}) = \nabla_{x^{i-1}}(z^{(i)}) \odot \nabla_{z^{(i)}}(x^{(i)})^T$$

Note we assume that the row vector in the equation is broadcasted by copying the rows on top of each other.

With this we have a way of recursively calculating the gradient of each layer with respect to the loss function. The final step is to now compute the gradients with respect to the weights and biases of each layer and store them.

First consider $z^{(i)}$ and $W^{(i)}$, note that we can compute the Jacobian of the flattened version of $W^{(i)}$ with respect to $z^{(i)}$ but if we observe that the jth component of $z^{(i)}$ is only affected by the jth row of $W^{(i)}$ we can construct our gradient matrix as containing the partial derivatives of each component of the jth row of $W^{(i)}$ with respect to the jth component of $z^{(i)}$ as its columns. Now since $z^{(i)} = W^{(i)}x^{(i-1)} + b^{(i)}$, $\frac{z_j^{(i)}}{W_{jk}^i} = x_k^{(i-1)}$. Thus:

$$J(z^{(i)})(W^i) = \begin{bmatrix} \frac{\partial z_1^{(i)}}{\partial W_{11}^{(i)}} & \cdots & \frac{\partial z_{l_i}^{(i)}}{\partial W_{l_i 1}^{(i)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1^{(i)}}{\partial W_{1 l_{i-1}}^{(i)}} & \cdots & \frac{\partial z_{l_i}^{(i)}}{\partial W_{l_i l_{i-1}}^{(i)}} \end{bmatrix}$$

$$= \begin{bmatrix} x_1^{(i-1)} & \cdots & x_{l_{i-1}}^{(i-1)} \\ \vdots & \ddots & \vdots \\ x_1^{(i-1)} & \cdots & x_{l_{i-1}}^{(i-1)} \end{bmatrix}$$

Thus we can succinctly write the combined gradient as an outer product:

$$\begin{aligned} \nabla_{W^{(i)}}(x^{(i)}) &= J(z^{(i)})(W^i) \odot \nabla_{z^{(i)}}(x^{(i)})^T \\ &= x^{(i-1)} \nabla_{z^{(i)}}(x^{(i)})^T \end{aligned}$$

Next for the $b^{(i)}$ term since it is just a constant term and only its jth component affects the j-th component of $z^{(i)}$, $\nabla_{b^{(i)}}(z^{(i)}) = 1_{l_i}$ and $\nabla_{b^{(i)}}(x^{(i)}) = \nabla_{z^{(i)}}(x^{(i)})$. With this, we have fully detailed the recursive algorithm steps.

Implementation

Since I am building this model in pure NumPy and Python, I want to leverage NumPy BLAS calls as much as possible, thus I will opt for storing the necessary gradients as arrays and using vectorized operations to compute each back prop step. Thus I will not create a individual neuron class and work from the layer level, storing a neuron array and a gradient array.