Backpropagation Explanation for a Scalable Multilayer Perceptron

Callum Frederiksen

May 2025

1 Introduction

This is the explanation behind the math for my MLP project.

2 Model Overview

2.1 MLP Architecture

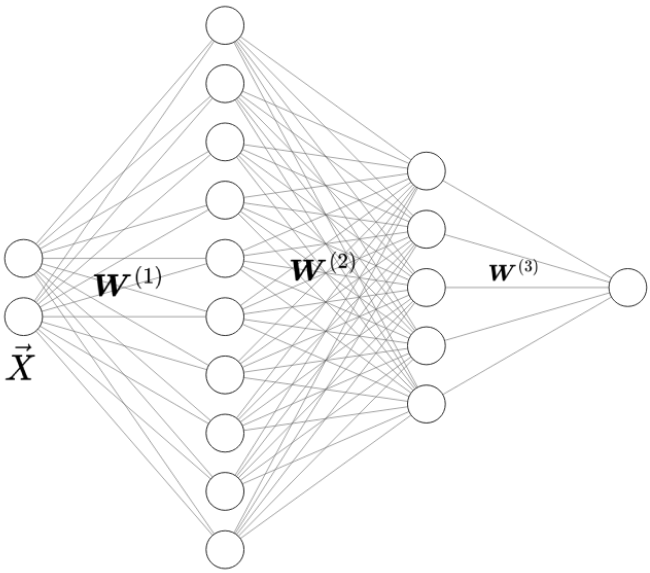

Figure 1: SPEND HR ON THIS

The underlying activation for this model is the sigmoid activation function, as it will scale output values between 0 and 1, as well as being non-linear and

continuous for $\sigma(x) \in [0, 1]$ for $x \in \mathbb{R}$. our activation $\sigma(x)$ is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}},\ x \in \mathbb{R}$$

. Interestingly, when you take the derivative of $\sigma(x)$ with respect to x,

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

This will be used extensively when backpropagating throughout the algorithm to determine the derivative of the Loss function with respect to each set of parameters (defined as weights, and biases).

2.2 The Goal of backpropogation

The end goal of backpropogation is to use the gradient descent algorithm

$$\theta_{i+1} := \theta_i - \alpha \nabla \mathcal{L}(\theta_i)$$

Where θ_i represents the collective paramters (weights and biases) for the ith iteration, and α represents the learning rate of the model.

Backpropogation can be used to achieve this, by iterating the gradient descent algorithm until convergence, in the best case scenario.

2.3 The Forward Pass

The forward pass is where the deep neural network can take the inputs $\vec{X}$, can be passed throughout the neural network, and compute the final binary prediction $\hat{y}$.

This is computed throughout the network by the linear formula

$$\vec{z}^{(n)} = \boldsymbol{W}^{(n)} \cdot \vec{a}^{(n-1)} + b^{(n)}$$

Where n refers to the layer for the activation vectors, and the layer the weight is being passed into for the weights.

The non-linear sigmoid activation function can be used to normalise the values between 0 and 1.

$$\therefore \vec{a}^{(n)} = \sigma(\vec{z}^{(n)})$$

This is repeated across the network, throughout each layer until the *final layer*. This has then computed the model's prediction $\hat{y}$ *(note as there is only one value, this will be a scalar, but is represented as a* 1×1 *matrix to simplify the numpy implementation in the code)*

2.4 The loss function

To calculate the loss of the model, we will use the sum of squared errors loss (SSE). This allows us to determine how incorrect the model is (i.e. if the model

has a high loss, it is inaccurate, therefore minimizing the loss maximises its accuracy) The SSE formula is defined below:

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

(for a single training example)

3 The backpropagation algorithm

3.1 The high-level intuition

Essentially, the backpropogation model for a DNN is to use of series of chain rules in order to derive the gradient of the model w.r.t. the parameters to calculate $\nabla\mathcal{L}$, which is the main focus for gradient descent.

To simplify notation, we will use $\delta^{(n)}$ to represent the derivative of the loss function w.r.t. the nth layer, *before applying the activation function (we will see later that this simplifies the process).*

$$\therefore \delta^{(n)} = \frac{\partial \mathcal{L}}{\partial \vec{z}^{(n)}}$$

Ideally, my program would be fully vectorised, however currently it is only partially vectorised, so calculating $\delta^{(n)}$ will fall into two methods:

- Finding $\delta^{(n)}$ for every layer *except the last layer*
- Finding $\delta^{(L)}$, where L represents the number of the model's layers (therefore calculating the last layer's delta)

3.2 Deriving $\delta^{(L)}$

In an ideal world, this model will be in a fully vectorised format, and perhaps this is one of the major drawbacks of my implementation of the model - it is dynamic in the number individual layers and the size of each corresponding layer, however it is static in returning a scalar value, scaled between 0 and 1, in a 1×1 numpy array.

Apart from it being only able to classify between two labels currently, the other major drawback is that as the output $\hat{y}$ is being treated as a scalar, dot products do not technically work.

In order to overcome this, we must manually implement the final layer's delta, which as explained below, is the starting point in calculating the gradient for the rest of the model. We will use the notation L to represent the number of layers in the model, therefore the Lth layer in the model is the final layer:

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \vec{z}^{(L)}}$$

As we can see from section 2.4, we can partially derive the loss function $\mathcal{L}(\hat{y}, y)$ w.r.t. $\hat{y}$ to calculate the "effect" on the loss.

As previously defined in section 2.3, if $\hat{y} = \vec{a}^{(L)} \implies \hat{y} = \sigma(\vec{z}^{(L)})$, therefore

$$\frac{\partial \hat{y}}{\partial \vec{z}^{(L)}} = \sigma\prime(\vec{z}^{(L)})$$

If we put all of these together, we can derive the final layer's delta:

$$\delta^{(L)} = (\hat{y} - y) \times \sigma\prime(\vec{z}^{(L)})$$

3.3 Deriving $\delta^{(n-1)}$

To start, recall that backpropogation works backwards, following the pattern $n = L, L-1, ..., 1$ iteratively. Using this knowledge, we can first compute $\delta^{(L)}$, and then work backwards.

$$\delta^{(n-1)} = \frac{\partial L}{\partial \vec{z}^{(n)}} \frac{\partial \vec{z}^{(n)}}{\partial \vec{z}^{(n-1)}} = \delta^{(n)} \frac{\partial \vec{z}^{(n)}}{\partial \vec{z}^{(n-1)}}$$

As $\vec{z}^{(n)} = \boldsymbol{W}^{(n)} \cdot \vec{a}^{(n-1)} + b^{(n)}$, we can derive the output vector $\vec{z}$ w.r.t. the activation passed into the layer. Then, as we are using the sigmoid function, we can take the derivative of that, and use the hadamard product *as the dimensions of $\vec{a}$ will be the same as $\vec{z}$.* We will donate the hadamard product here as $\odot$

$$\therefore \delta^{(n-1)} = (\boldsymbol{W}^{(n)} \cdot \delta^{(n)}) \odot \sigma\prime(\vec{z}^{(n-1)})$$

This process can be continued iteratively until the program hits the first hidden layer. Once this is complete we can start to derive the layer's weights and biases w.r.t. the layer output $\vec{z}$

3.4 Deriving weights and biases

Each $\delta^{(n)}$ represents the importance of the linear sum of the weights, biases and previous activations on the final loss of the model. Since $\delta^{(n)} = \frac{\partial L}{\partial \vec{z}^{(n)}}$, we can find $\frac{\partial L}{\partial \boldsymbol{W}}$ and $\frac{\partial L}{\partial b}$ as below:

$$\frac{\partial L}{\partial \boldsymbol{W}^{(n)}} = \frac{\partial L}{\partial \vec{z}^{(n)}} \frac{\partial \vec{z}^{(n)}}{\partial \boldsymbol{W}^{(n)}} = \delta^{(n)} \frac{\partial \vec{z}^{(n)}}{\partial \boldsymbol{W}^{(n)}}$$

And a similar approach to finding the derivative of the loss w.r.t. the biases. If we use the formula shown before in the feed-forward subsection:

$$\vec{z}^{(n)} = \boldsymbol{W}^{(n)} \cdot \vec{a}^{(n-1)} + b^{(n)}$$

Then we can derive the weights and biases with respect to the linear function $\vec{z}^{(n)}$, with respect to the weights and biases, allowing us to get the gradient of the loss w.r.t. every parameter θ in the model to be updated collectively through the gradient descent algorithm:

$$\boxed{\frac{\partial \vec{z}^{(n)}}{\partial \boldsymbol{W}^{(n)}} = \vec{a}^{(n-1)\top}} \qquad \boxed{\frac{\partial \vec{z}^{(n)}}{\partial b^{(n)}} = 1}$$

Note that the activation of the previous layer is transposed to ensure that the matrix / vector multiplication can occur and works