Multilayer Perceptrons

December 12, 2025

Contents

Chapter 1

The Basics of Multilayer Perceptrons

A **multilayer perceptron** (otherwise known as a *neural network*) is a set of nodes called **neurons**, each containing an activation – a number (usually) between 0 and 1 – which designates how active that specific neuron is.

This is loosely analogous to how our brains work, where each of our neurons can fire to send an electrical signal which, somewhere down the chain, might move our arm or let us perceive the smell of some food.

The image people usually associate with neural networks is the following:

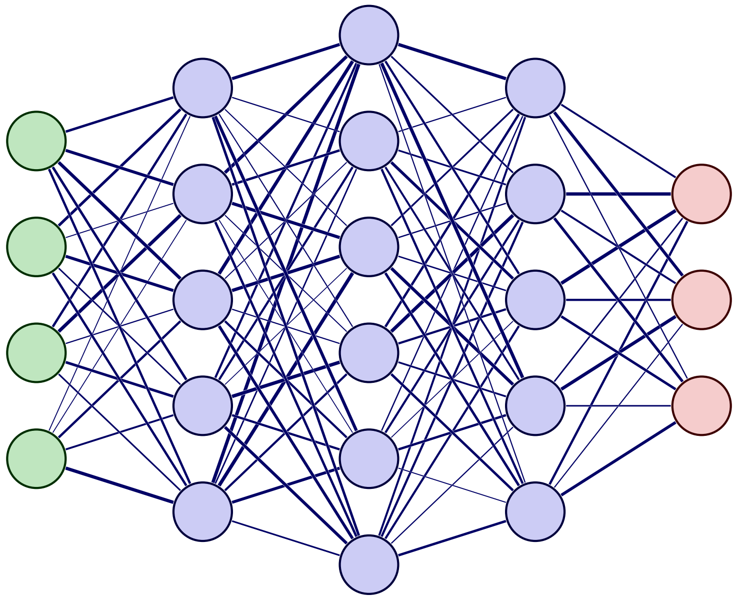

FIGURE 1.1: A simple neural network.

Figure 1.1 shows the structure of a neural network. At its core, a neural network has an **input layer** (colored in green), analogous to some sensory input such as our eyes capturing the light around us, an **output layer** (colored in red), analogous to the output of our vision, which is a colored image of our environment, and then some "hidden layers," which are basically the meat of the network and encode everything about how the inputs and corresponding outputs are related to one another.

The actual math behind these networks is surprisingly simple. We'll focus on a single connection for now:

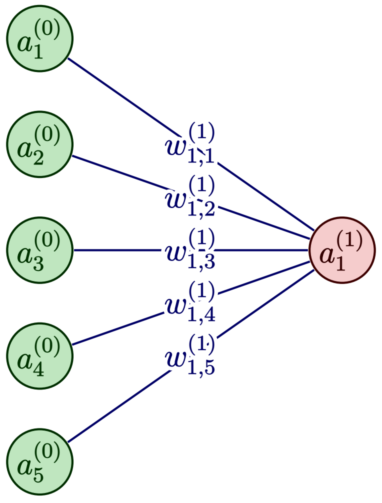

FIGURE 1.2: The activation of two neurons and the weight between them.

Let's decypher these symbols one-by-one:

- $a_n^{(0)}$: The activations of the nodes within the first layer.
- $w_{1,n}^{(1)}$: The **weights** of each of the connections. Essentially, this encodes how strongly connected the neurons are (so how much they influence each-other).
- $a_1^{(1)}$: The output activation.

In order to determine the output activation $a_1^{(1)}$ from the input activations $a_i^{(0)}$, we simply take a weighted sum whose weights are the coefficients $w_{1,i}^{(1)}$; that is,

$$a_1^{(1)} = w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + \cdots + w_{1,5}^{(1)} a_5^{(0)}$$

Usually we also add in what's called a **bias** term, which is just a value that we add to this sum. We can interpret the bias as the "sensitivity" of the neuron. That means that the whole expression in the general case (so, for n neurons in the input layer) is

$$a_1^{(1)} = b_1^{(1)} + \sum_{i=1}^{n} w_{1,i}^{(1)} a_i^{(0)}$$

There's only one issue: Remember we said that the activations must be between 0 and 1, but with a weighted sum like this, we can essentially get any number. What we need now is some kind of way to squish this value into the range $[0, 1]$. The most common function for this purpose is the **sigmoid** function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The graph of this function is shown below:

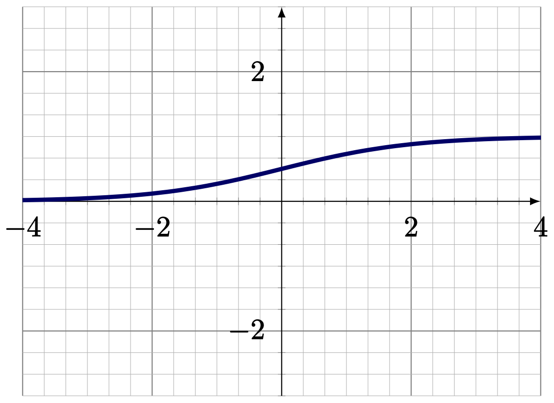

FIGURE 1.3: The graph of $\sigma(x)$.

From Figure 1.3, we can see how the function approaches 0 when x is negative, and approaches 1 when x is positive. This is exactly what we needed! Putting all of this together, we have the following expression for $a_1^{(1)}$:

$$a_1^{(1)} = \sigma\left(b_1^{(1)} + \sum_{i=1}^{n} w_{1,i}^{(1)} a_i^{(0)}\right)$$

Remember, this is just for the activation of a *single* neuron, so for a whole layer, we need to do this many times. There is a way we can simplify this, but we'll need to consider the more general case in order to do so.

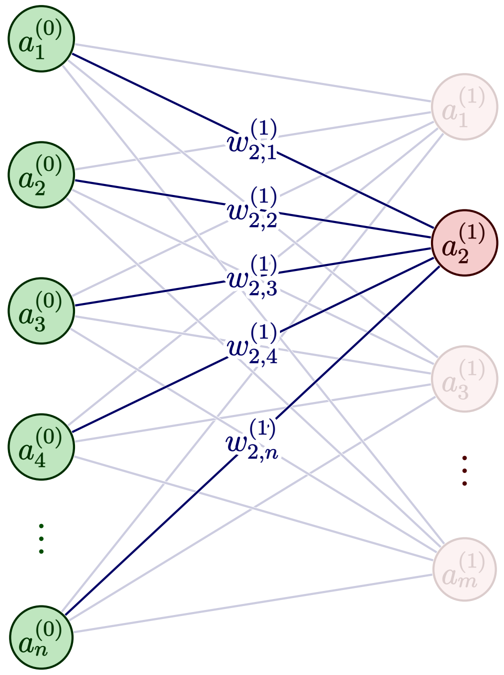

FIGURE 1.4: The general case of the connections between two layers.

The key insight here is that we can simplify this process immensely with the tools of linear algebra.

In particular, we can group all the activations in the first layer (layer 0) into a column vector:

$$\mathbf{a}^{(0)} = \begin{bmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix}$$

We can do the same thing for the output activations:

$$\mathbf{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix}$$

Similarly, we can group the biases up into a vector as well:

$$\mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_m^{(1)} \end{bmatrix}$$

Finally, the weights simply become a matrix which we multiply with $\mathbf{a}^{(0)}$:

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} & \cdots & w_{1,n}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} & \cdots & w_{2,n}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} & \cdots & w_{3,n}^{(1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(1)} & w_{m,2}^{(1)} & w_{m,3}^{(1)} & \cdots & w_{m,n}^{(1)} \end{bmatrix}$$

Now, if we imply that the function $\sigma(x)$ applies component-wise to vectors and matrices, we get the following incredibly neat formula for the activations in layer $L+1$ given those in layer L:

$$\mathbf{a}^{(L+1)} = \sigma\Big(\mathbf{b}^{(L+1)} + \mathbf{W}^{(L+1)}\mathbf{a}^{(L)}\Big)$$

Other Activation Functions

The sigmoid is not the only function we can use to remap our values. In fact, there are infinitely many functions we can use. Another common choice is the tanh function (which remaps values between -1 and 1). Beyond that, we sometimes don't want to remap between finite values at all. This is why other (arguably more common) activation functions include the ReLU, SiLU, and, more recently, GELU functions.

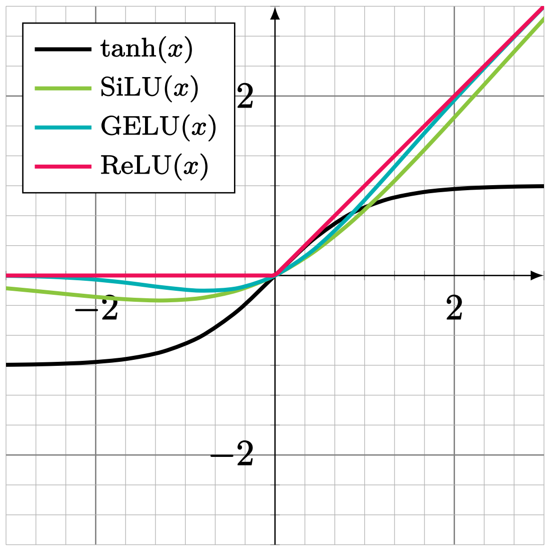

FIGURE 1.5: The graphs of the different activation functions.

ReLU is very common because it's very strict for model behavior; the activation simply gets clamped to zero if it's negative. The expressions for these functions are the following:

$$\begin{aligned} \mathrm{ReLU}(x) &= \max(x, 0) \\ \mathrm{SiLU}(x) &= x\sigma(x) \\ \mathrm{GELU}(x) &\approx 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right) \text{ [1]} \end{aligned}$$

The expression for GELU is simply an approximation because it is actually derived from the cumulative distribution function of the normal distribution, which has no elementary form. In fact, the *true* expression defining GELU(x) is

$$\mathrm{GELU}(x) = x\Phi(x) = x\frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right],$$

where

$$\mathrm{erf}(x) = \int_0^x e^{-s^2} ds$$

is the error function.

Chapter 2

Backpropagation: How Neural Networks Learn

This chapter is all about one of the most important algorithms in deep learning: backpropagation. Essentially, this is how a neural network is able to “learn” from its training data. At a high level, the way it works is that it takes the prediction of the model, compares it with the expected output, and uses how far they are from each other to adjust the weights and biases of the model.

2.1 Cost or Loss Functions

At the core of backpropagation is the idea of a **cost function** or **loss function**. This is the function which, given the model's prediction and the target value, returns some number indicating how badly the model performed. Then, once we have this mysterious function, backpropagation comes down to a simple optimization problem – we'll want to minimize whatever this function is.

There are tons of loss functions out there, but the two most common ones are MSE (mean squared error) loss and cross-entropy loss. MSE loss is good for general purpose models, whilst cross-entropy is specifically designed for models whose objective is some sort of classification (such as for the classic MNIST model).

The definition for MSE loss is quite straightforward. Given a model output $\mathbf{y}$ and a target output $\hat{\mathbf{y}}$, the MSE loss is simply the average of their squared difference. Mathematically, assuming that $\mathbf{y}, \hat{\mathbf{y}} \in \mathbb{R}^n$,

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

The main reason this is such a popular choice is that it's incredibly easy to differentiate, since it's just a power. It's also very intuitive: the amount the model deviates from the target is just the average difference between its output and the target, and we square it so that the values are all positive while keeping the function differentiable.

Cross-entropy, on the other hand, has a few nuances that we'll dig into later. As we said earlier, cross-entropy is used as a loss function for classification models. These are models whose outputs are vectors of probabilities (so, values between 0 and 1) indicating which of the "classes," as they're called, the model believes the input belongs to. Suppose that our model has n classes and an output vector $\mathbf{p}$ of its predicted probabilities for each of the classes. Then, the cross-entropy loss for this output is

$$\mathrm{CE} = -\sum_{c=1}^{n} y_c \log(p_c)$$

The vector $\mathbf{y}$ here is the vector of expected probabilities. It may well be one-hot (meaning that only one component is equal to 1 while the others are zero), in which case this simplifies down to $\mathrm{CE} = -\log(p_c)$, where c is the correct class, but that is not always the case. In some classification tasks, labels are not 100% in one class, they might be in some kind of superposition state.

The nuances with cross-entropy come down to the activation functions used. We can use whichever activation function we want in our model, but we have to use a very specific activation function on the final layer before calculating the cross-entropy. This function is called the **softmax** function. It is defined as

$$\mathrm{softmax}(\mathbf{x}) = \left[\frac{e^{x_i}}{\sum_j e^{x_j}}\right]$$

Essentially, we exponentiate each component of the vector, and divide it by the sum of all the other exponentiated components. This gives us a vector with values between 0 and 1 which also adds up to 1 (thus making it a valid probability distribution). The exact reason why we're exponentiating is that it handles negative values properly, by making them very small but also positive. If we just took each component over the sum of the vector, we'd get negative values, which are not valid probabilities.

2.2 Deriving Backpropagation

Now that we know what cost functions are, how can we use them to determine how much we need to change each of our parameters by to increase the result. This is effectively just a calculus problem: we want to minimize some function $C(\mathbf{a}^{(n_L-1)}, \mathbf{y})$ of the activations in the last layer[1] and the target output.

We'll also introduce a common piece of notation: we use $\mathbf{z}^{(L)}$ to denote the input to the activation function of the model; that is, $\mathbf{z}^{(L)} = \mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}$. This notation will be useful later on.

2.2.1 Weights in the last layer

Our goal is essentially to adjust the weights and biases so as to decrease the value of the loss function. That means that, putting aside rigor for a moment, the variation in our model's weights W and biases B should look something like this:

$$\Delta W \propto -\frac{\partial C}{\partial W}$$
$$\Delta B \propto -\frac{\partial C}{\partial B}$$

[1] We use $n_L - 1$ as the layer number for the last layer since n_L is the number of layers and the convention is that layers start at 0, not 1.

We'll start by looking at the output layer at a specific weight $w_{ij}^{(n_L-1)}$. From the equations above,

$$\Delta w_{ij}^{(n_L-1)} \propto -\frac{\partial C}{\partial w_{ij}^{(n_L-1)}}$$

However, the cost function is not directly related to the weight, so we need to apply the chain rule until we reach the input of the function. Recall that

$$a_i^{(n_L-1)} = \sigma\left(b_i^{(n_L-1)} + \sum_j w_{ij}^{(n_L-1)} a_j^{(n_L-2)}\right) = \sigma\left(z_i^{(n_L-1)}\right)$$

This means that, according to the chain rule,

$$\frac{\partial C}{\partial w_{ij}^{(n_L-1)}} = \frac{\partial C}{\partial a_i^{(n_L-1)}} \frac{\partial a_i^{(n_L-1)}}{\partial z_i^{(n_L-1)}} \frac{\partial z_i^{(n_L-1)}}{\partial w_{ij}^{(n_L-1)}}$$

We'll evaluate each of these derivatives one-by-one until we get an exact expression for $\Delta w_{ij}^{(n_L-1)}$. Firstly, since we're working generally, we can't really simplify the derivative of the cost function relative to the activation (as that will ultimately depend on the cost function used). However, we can somewhat reduce $\dfrac{\partial a_i^{(n_L-1)}}{\partial z_i^{(n_L-1)}}$:

$$\frac{\partial a_i^{(n_L-1)}}{\partial z_i^{(n_L-1)}} = \sigma'\left(z_i^{(n_L-1)}\right)$$

Unfortunately, since we want to be as general as possible, we can't really reduce the expression any more since we don't know what the activation function is.

Finally, we can get an actual exact expression for $\dfrac{\partial z_i^{(n_L-1)}}{\partial w_{ij}^{(n_L-1)}}$. The crucial thing to notice is that only one of the terms in the sum that defines z will have a nonzero derivative with respect to the weight, and that's the term with that specific weight in it:

$$z_i^{(n_L-1)} = \underbrace{b_i^{(n_L-1)}}_{\text{constant}} + \underbrace{w_{i,1}^{(n_L-1)} a_1^{(n_L-2)}}_{\text{constant}} + \underbrace{w_{i,2}^{(n_L-1)} a_2^{(n_L-2)}}_{\text{constant}} + \cdots + \underbrace{w_{ij}^{(n_L-1)} a_j^{(n_L-2)}}_{\text{not constant}} + \cdots + \underbrace{w_{in}^{(n_L-1)} a_n^{(n_L-2)}}_{\text{constant}}$$

Therefore,

$$\frac{\partial z_i^{(n_L-1)}}{\partial w_{ij}^{(n_L-1)}} = a_j^{(n_L-2)}$$

Thus, putting this all together,

$$\Delta w_{ij}^{(n_L-1)} = -\varepsilon \frac{\partial C}{\partial a_i^{(n_L-1)}} \sigma'\left(z_i^{(n_L-1)}\right) a_j^{(n_L-2)},$$

where ε is a chosen proportionality constant (usually referred to as the **learning rate**) of the model. For the sake of readability (and consistency with the widely accepted equations), we'll factor out some terms into a new variable, which we'll call δ:

$$\delta_i^{(n_L-1)} = \frac{\partial C}{\partial a_i^{(n_L-1)}} \sigma'\left(z_i^{(n_L-1)}\right)$$

This indexing is just begging us to turn δ into a vector, and we can do so by introducing the *Hadamard product* (that is, the multiplication of two vectors that is done component-wise), which is denoted with $\odot$. This cleans things up significantly:

$$\delta^{(n_L-1)} = \nabla_{\mathbf{a}} C \odot \sigma'\left(\mathbf{z}^{(n_L-1)}\right)$$

The **a** in the subscript of the gradient indicates that we're differentiating with respect to the activation (and not the target). For notational simplicity, it is implied that **a** is the activation in the last layer ($\mathbf{a}^{(n_L-1)}$ formally).

With this new notation, the expression for $\Delta\mathbf{W}^{(n_L-1)}$ gets considerably simplified:

$$\Delta\mathbf{W}^{(n_L-1)} = -\varepsilon\delta^{(n_L-1)}\left(\mathbf{a}^{(n_L-2)}\right)^T$$

A few things are worth clarifying in this equation. First, the order here *is* important, and second, the transpose is used because we need an *outer product*; that is, a product between two vectors that returns a matrix. In our case, the expression $\delta^{(n_L-1)}\left(\mathbf{a}^{(n_L-2)}\right)^T$ gives us a matrix of the following form:

$$\begin{bmatrix} \delta_1 a_1 & \delta_1 a_2 & \delta_1 a_3 & \cdots & \delta_1 a_n \\ \delta_2 a_1 & \delta_2 a_2 & \delta_2 a_3 & \cdots & \delta_2 a_n \\ \delta_3 a_1 & \delta_3 a_2 & \delta_3 a_3 & \cdots & \delta_3 a_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \delta_m a_1 & \delta_m a_2 & \delta_m a_3 & \cdots & \delta_m a_n \end{bmatrix},$$

where m is the number of neurons in the last layer (layer $n_L - 1$) and n is the number of neurons in the layer before that (layer $n_L - 2$).

Bibliography

[1] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs)*. 2023. arXiv: `1606.08415 [cs.LG]`. URL: `https://arxiv.org/abs/1606.08415`.