Backpropagation Calculus

Nick Rui

December 2024

Description

A written introduction to the mathematical theory behind backpropagation algorithms. We start by exploring the calculus behind finding partial derivatives of the cost function (or lost function) of neural networks. We then develop a worked example of a neural network using a cross-entropy/softmax implementation on two layers. Finally, we generalize the equations to implement a similar type of neural network with an arbitrary number of layers.

This paper was written primarily for the purposes of the author's own learning. Hence, there may be minor mistakes.

Introducing Neural Networks

We first define a neuron (in a neural network) by a singular value, called its activation. A neuron's activation is simply a numerical value. A neural network takes in an input of neurons (think a vector of numerical values) and outputs neurons (another vector of values). Simply put, a neural network is a vector-valued function that maps an input vector to an output vector.

The complexity behind a neural network comes within its hidden layers, which can be considered subfunctions within the function itself. The composition of these subfunctions make up the function itself.

As an example, say our neural network takes in an input size of n neurons and outputs k neurons. Let us define a function $f : \mathbb{R}^n \to \mathbb{R}^k$ to represent this neural network. To do this, the n input neurons are first mapped onto a hidden layer of ℓ neurons. Then, those ℓ neurons of the hidden layer are mapped to the output set of k neurons. We can define functions $f_1 : \mathbb{R}^n \to \mathbb{R}^\ell$ and $f_2 : \mathbb{R}^\ell \to \mathbb{R}^h$ to represent these mappings. Thus, our network can be described as $f = f_2 \circ f_1$ (so we map from $\mathbb{R}^n \to \mathbb{R}^\ell \to \mathbb{R}^k$).

Before we discuss how such mappings should be defined, let's establish notation. Let $a_i^{(L)} \in \mathbb{R}$ denote the activation of the i-th neuron in the L-th layer of our network (the 0-th layer is the input layer). Let n_L represent the number of neurons in the L-th layer.

So, how should we define such mappings? For simplicity, we can consider only using linear mappings so that the neurons in one layer are linear combinations of neurons in the previous layer. To do this, we need to define weights $w_1^{(L)}, w_2^{(L)}, \ldots$ which are the scalars of the linear combination. So, as a general outline we will have

$$a_i^{(L)} = \sum_{j=1}^{n_{L-1}} w_{ji}^{(L)} a_j^{(L-1)}.$$

For some added complexity, we also add a bias $b_i^{(L)}$ to each neuron, which is an added constant independent of whatever inputs we receive in the prior layer. So, we have

$$a_i^{(L)} = \sum_{j=1}^{n_{L-1}} w_{ji}^{(L)} a_j^{L-1} + b_i^{(L)}.$$

We can represent this function in vector notation. Let $\mathbf{a}^{(L)}$ denote the vector representing neurons in the L-th layer, $\mathbf{b}^{(L)}$ represent the vector of biases associated with the L-th layer, and $W^{(L)}$ the $n_L \times n_{L-1}$ matrix representing the weights associated with the L-th layer (so $w_{ji}^{(L)}$ represents the weight the i-th neuron in the L-th layer has assigned to the j-th neuron in the $L-1$-th row).

$$\mathbf{a}^{(L)} = \begin{bmatrix} a_1 \\ \dots \\ a_{n_L} \end{bmatrix}, \quad \mathbf{b}^{(L)} = \begin{bmatrix} b_i^{(L)} \\ \dots \\ b_{n_L}^{(L)} \end{bmatrix}, \quad W^{(L)} = \begin{bmatrix} w_{11}^{(L)} & \dots & w_{1n_L}^{(L)} \\ \dots & \dots & \dots \\ w_{n_{L-1},1}^{(L)} & \dots & w_{n_{L-1},n_L}^{(L)} \end{bmatrix}$$

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

And for some more added complexity, we apply an additional function to ensure that each neuron's resulting activation lies between some bound (for example, strictly positive). This is what makes the layers of our neural network valuable, since without this function every neuron would be a linear combination of the previous neurons and thus the output layer would also be a linear combination of the input layer. Examples of such functions are the sigmoid function σ or the much more common "rectified linear unit", or $ReLU$. We will elaborate more on how these functions work later on. For now, denote this mystery function as μ. We have

$$\mathbf{a}^{(L)} = \mu(W^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}).$$

Essentially, we think of these weights and biases as parameters for our function f. To train our neural network, we need to tweak these parameters until our neural network spits out the most accurate results. To do this, we must compare the result of the current model with the results we expect.

We define a cost function C to do this. Say we expect the output of our neural network to be the vector $\mathbf{y}$. We want our cost function to tell us how much our prediction was off. This is similar to the least-squares method of regression where we want to minimize the least squared difference between the neurons our network outputs and the expected output. In vector language, the cost function takes the form

$$C = ||\mathbf{a}^{(L)} - \mathbf{y}||^2.$$

When we train our neural network, we do so on large training sets where we have an input/output pair already matched up. Thus, the cost function should be the sum of all the squared errors across all test trials. In addition, the inputs into our neural network can be thought of as "constant", since these training data points are set in stone. and we can think of our parameters (weights and biases) to be the variables of our cost function. If we have N data points to train off of, our cost function is

$$C(W^{(1)}, \mathbf{b}^{(1)}, \dots, W^{(L)}, \mathbf{b}^{(L)}) = \sum_{j=1}^{N} ||\mathbf{a}_j^{(L)} - \mathbf{y}_j||^2$$

And so, the question we face is what parameters to choose to minimize the cost function.

Reviewing Polynomial Regression

The idea behind backpropagation is for neural networks is similar to the gradient descent idea that results from polynomial regression. Say we have a collection of data points in $\mathbb{R}^2$ and seek to find a polynomial $p(x)$ of degree d that "best fits" the data. By that, we mean that the polynomial minimizes the squared difference between the predicted value and actual value.

We know our polynomial will take the form of

$$p(x) = k_0x^0 + k_1x^1 + k_2x^2 + \cdots + k_dx^d.$$

Here, we think of the scalars $k_0, \ldots k_d$ as parameters for our cost function, i.e. changing these parameters will change the shape of our polynomial and thus change the sum of the squared errors. Denote the actual value of each input x_i as y_i and the number of points N. Our cost function is

$$\begin{aligned} C(k_0, \ldots k_d) &= \sum_{i=1}^{N} (p(x_i) - y_i)^2 \\ &= \sum_{i=1}^{N} ((k_0x_i^0 + \cdots + k_dx_i^d) - y_i)^2 \end{aligned}$$

For this model to "learn", we first set our parameters to random values. Then, we harness the power of calculus to calculate the gradient of C. Recall the gradient of a function "points" in the direction of greatest ascent. Since we seek to minimize C, we shift our parameters in the direction of the negative gradient. Iterating this process, we should arrive at a local minimum (hence, "gradient descent").

Most of the mathematics involved has to do with calculating the partial derivatives that make up the gradient. Recall the gradient takes the form

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial k_0} \\ \frac{\partial C}{\partial k_1} \\ \cdots \\ \frac{\partial C}{\partial k_d}. \end{bmatrix}$$

Taking partial derivatives, we have (by the chain rule)

$$\frac{\partial C}{\partial k_j} = \sum_{i=1}^{N} 2(p(x_i) - y_i)x_i^j.$$

Now, let's take a moment to verify that we can actually compute a numerical value for each partial derivative. Recall x_i, y_i are provided from the data set, and $k_0, \ldots k_d$ (found in $p(x_i)$) are the parameters we are evaluating this partial derivative at. So, we are able to actually compute this partial derivative and thus the gradient of C.

Backpropagation: Brute-force Approach with Chain Rule

We take a similar approach to the polynomial regression to calculate the partial derivatives of our cost function for our neural network.

Recall the earlier notation and our cost function given by

$$C(W^{(1)}, \mathbf{b}^{(1)}, \ldots, W^{(L)}, \mathbf{b}^{(L)}) = \sum_{j=1}^{N} ||\mathbf{a}_j^{(L)} - \mathbf{y}_j||^2$$

We will start from the simplest case and build our way up to the complexity of this cost function. First, we analyze a neural network with one layer of each one neuron. Then, we analyze a network with arbitrary layers of each one neuron. Then, a network with arbitrary layers each with an arbitrary number of neurons. Finally, we consider the fact that multiple training data points need to be analyzed.

One Layer of One Neuron

Let's analyze the simplest case of a neural network, one with only one layer (the output) with only one neuron. Denote the input and output neurons $a^{(0)}, a^{(1)}$, and the weight and bias associated with the first layer as w, b. The neurons hold the following relationship.

$$a^{(1)} = \mu(wa^{(0)} + b)$$

For simplicity, let's denote $\mathbf{z}^{(L)} = W^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}$. In this simple case, we have $z^{(1)} = wa^{(0)} + b$, and so $a^{(1)} = \mu(z^{(1)})$.

Denote y the expected output (i.e. what $a^{(1)}$ should be). The cost function is

$$\begin{aligned} C(w, b) &= (a^{(1)} - y)^2 \\ &= (\mu(z^{(1)}) - y)^2 \\ &= (\mu(wa^{(0)} + b) - y)^2 \end{aligned}$$

Calculating partial derivatives, we have (by the chain rule)

$$\begin{aligned} \frac{\partial C}{\partial w} &= \frac{\partial C}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial w} = 2(a^{(1)} - y)\mu'(z^{(1)})a^{(0)} \\ \frac{\partial C}{\partial b} &= \frac{\partial C}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial b} = 2(a^{(1)} - y)\mu'(z^{(1)}) \end{aligned}$$

Now is a good time to revisit the μ function, since we need to know its derivative. Recall this μ function could be any function, but for the sake of example let's use the rectified linear unit function, which takes the form

$$ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

(Essentially, $ReLU$ ensures all neuron activations are non-negative by collapsing any negative value to zero.)

And so, its derivative is easy to compute.

$$ReLU'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

And so, our partial derivatives become

$$\frac{\partial C}{\partial w} = 2(a^{(1)} - y)ReLU'(z^{(1)})a^{(0)} = \begin{cases} 2(a^{(1)} - y)a^{(0)} & \text{if } z^{(1)} > 0 \\ 0 & \text{if } z^{(1)} \leq 0 \end{cases}$$

$$\frac{\partial C}{\partial b} = 2(a^{(1)} - y)ReLU'(z^{(1)}) = \begin{cases} 2(a^{(1)} - y) & \text{if } z^{(1)} > 0 \\ 0 & \text{if } z^{(1)} \leq 0 \end{cases}$$

For now, we return to the more general case of any arbitrary function μ, but note that its derivative is known.

As we did before, let's verify that we can actually compute these partial derivatives. These partial derivatives are evaluated at our parameters w, b, which are used in calculating $z^{(1)}$ and $a^{(1)}$. Also, y is given, so we are able to calculate these partial derivatives.

L layers of one neuron

Let's expand this approach into more complex neural networks. Take a neural network of L layers, each with a single neuron. Let's explore how the partial derivative of our cost function with respect to an arbitrary weight and bias $w^{(j)}, b^{(j)}$ can be calculated.

Our cost function is

$$\begin{aligned} C(w^{(1)}, b^{(1)}, \ldots, w^{(L)}, b^{(L)}) &= (a^{(L)} - y)^2 \\ &= (\mu(z^{(L)}) - y)^2 \\ &= (\mu(w^{(L)} a^{(L-1)} + b^{(L)}) - y)^2 \\ &= (\mu(w^{(L)} \mu(w^{(L-1)} a^{(L-2)} + b^{(L-1)}) + b^{(L)}) - y)^2 \\ &= \ldots \end{aligned}$$

Consider the added complexity here. Tweaking $w^{(1)}, b^{(1)}$ will change $a^{(1)}$, which will change $a^{(2)}$, which will change $a^{(3)}$, and so on until $a^{(L)}$ is changed. And so, every neuron following $a^{(i)}$ is influenced by $w^{(i)}, b^{(i)}$ also. Therefore, it's helpful to also consider the partial derivative of one neuron with respect to the activation of the previous neuron.

Since

$$a^{(L)} = \mu(z^{(L)}) = \mu(w^{(L)} a^{(L-1)} + b^{(L)}),$$

we have

$$\frac{\partial a^{(L)}}{\partial a^{(L-1)}} = \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} = \mu'(z^{(L)}) w^{(L)}.$$

And so, finding the partial derivatives of C with respect to $w^{(L-1)}, b^{(L-1)}$ involves first computing the partial derivatives of $a^{(L-1)}$ with respect to $w^{(L-1)}, b^{(L-1)}$, then applying the chain rule.
To help with computation, note that

$$a^{(L-1)} = \mu(z^{(L-1)}) = \mu(w^{(L-1)} a^{(L-2)} + b^{(L-1)}).$$

So we have

$$\begin{aligned} \frac{\partial C}{\partial w^{(L-1)}} &= \frac{\partial C}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} = 2(a^{(L)} - y) \mu'(z^{(L)}) w^{(L)} \mu'(z^{(L-1)}) a^{(L-2)} \\ \frac{\partial C}{\partial b^{(L-1)}} &= \frac{\partial C}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial b^{(L-1)}} = 2(a^{(L)} - y) \mu'(z^{(L)}) w^{(L)} \mu'(z^{(L-1)}). \end{aligned}$$

We can continue this process of finding the partial derivative of C with respect to any $(L-j)$-th layer's weight and bias by utilizing the partial derivative of $a^{(L-j+1)}$ with respect to $a^{(L-j)}$, which is

$$\frac{\partial a^{(L-j+1)}}{\partial a^{(L-j)}} = \mu'(z^{(L-j+1)}) w^{(L-j+1)}$$

Now, we can generalize the partial derivatives of C with respect to $w^{(L-j)}, b^{(L-j)}$ as

$$\begin{aligned}
\frac{\partial C}{\partial w^{(L-j)}} &= \frac{\partial C}{\partial a^{(L)}} \left(\prod_{i=1}^{j} \frac{\partial a^{(L-i+1)}}{\partial a^{(L-i)}} \right) \frac{\partial a^{(L-j)}}{\partial z^{(L-j)}} \frac{\partial z^{(L-j)}}{\partial w^{(L-j)}} \\
&= 2(a^{(L)} - y) \left(\prod_{i=1}^{j} \mu'(z^{(L-i+1)}) w^{(L-i+1)} \right) \mu'(z^{(L-j)}) a^{(L-j-1)} \\
\frac{\partial C}{\partial b^{(L-j)}} &= \frac{\partial C}{\partial a^{(L)}} \left(\prod_{i=1}^{j} \frac{\partial a^{(L-i+1)}}{\partial a^{(L-i)}} \right) \frac{\partial a^{(L-j)}}{\partial z^{(L-j)}} \frac{\partial z^{(L-j)}}{\partial b^{(L-j)}} \\
&= 2(a^{(L)} - y) \left(\prod_{i=1}^{j} \mu'(z^{(L-i+1)}) w^{(L-i+1)} \right) \mu'(z^{(L-j)})
\end{aligned}$$

Again, let us take a step back and question whether we have enough information to compute these partial derivatives. We evaluate these partial derivatives at the weights and biases we set, which determine the values of all the z and a terms. Also, y is given, so we are able to compute these partial derivatives and calculate the gradient.

L Layers with Arbitrary Neurons

To avoid unnecessary confusion, we won't dive deep into the full process and derive an equation for the desired partial derivatives. Instead, let's focus on what added complexities having arbitrary neurons in each layer adds, and how we might conquer those complexities.

First, notation. Let n_L represent the number of neurons in the L-th layer, and let a subscript on any a, z, b terms denote the neuron it corresponds to. For weights, let $w_{ji}^{(L)}$ denote the weight the j-th neuron on the $(L-1)$-th layer has on the i-th neuron on the L-th layer (such indexing is used for simplicity later with matrices).

First, consider the activation of the i-th neuron in an arbitrary ℓ-th layer.

$$a_i^{(\ell)} = \mu(z_i^{(\ell)}) = \mu \left(\sum_{j=1}^{n_{\ell-1}} (w_{ij}^{(\ell)} a_j^{(\ell-1)} + b_i^{(\ell)}) \right)$$

Now, the partial derivative of $a_i^{(\ell)}$ with respect to the j-th neuron of the previous layer $a_j^{(\ell-1)}$ is

$$\frac{\partial a_i^{(\ell)}}{\partial a_j^{(\ell-1)}} = \frac{\partial a_i^{(\ell)}}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial a_j^{(\ell-1)}} = \mu'(z_i^{(\ell)}) w_{ij}.$$

Recall that the neuron $a_j^{(\ell-1)}$ influences every neuron of the ℓ-th layer. So, the overall influence of this neuron over the ℓ-th layer must consider its partial derivatives over all i.

What does our cost function look like? Say we've reached the last layer L. Let $\mathbf{y} = (y_1, \ldots, y_{n_L})$ be the expected output. Recall our cost function takes the form of

$$C = ||\mathbf{a}^{(L)} - \mathbf{y}||^2.$$

For now, it's easier to analyze this expression without any fancy vector notation. So, we have

$$C = \sum_{j=1}^{n_L} (a_j^{(L)} - y_j)^2.$$

The cost function's partial derivative with respect to the i-th neuron in the last layer is then

$$\frac{\partial C}{\partial a_i^{(L)}} = 2(a_i^{(L)} - y_i),$$

since all other elements in our summation vanish.

So, the effect of the last layer's weights and biases on the cost function can be described by

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}} = 2(a_i^{(L)} - y_i)\mu'(z_i^{(L)})a_j^{(L-1)}$$
$$\frac{\partial C}{\partial b_i^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial b_i^{(L)}} = 2(a_i^{(L)} - y_i)\mu'(z_i^{(L)})$$

Much more complexity is revealed when we analyze how the second to last layers' weights and biases impact the cost function. Since each neuron in the $L-1$-th layer influences each neuron in the L-th row, each neuron in the $L-1$-th layer influences the cost function through n_L different paths. So, we need to sum up n_L partial derivatives.

$$\frac{\partial C}{\partial w_{ij}^{(L-1)}} = \sum_{k=1}^{n_L} \left(\frac{\partial C}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial a_i^{(L-1)}} \right) \frac{\partial a_i^{(L-1)}}{\partial z_i^{(L-1)}} \frac{\partial z_i^{(L-1)}}{\partial w_{ij}^{(L-1)}}$$
$$\frac{\partial C}{\partial b_i^{(L-1)}} = \sum_{k=1}^{n_L} \left(\frac{\partial C}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial a_i^{(L-1)}} \right) \frac{\partial a_i^{(L-1)}}{\partial z_i^{(L-1)}} \frac{\partial z_i^{(L-1)}}{\partial b_i^{(L-1)}}$$

(Ensure the indices are clear. Neurons in layer L are indexed by k throughout the summation. Neurons in layer $L-1$ are indexed by i. Neurons in layer $L-2$ (which play a part for the weights in layer $L-1$) are indexed by j.)

Verify again that we are able to explicitly calculate these partial derivatives. If we consider the $L-2$-th layer, the partial derivatives are possible, but even more heinous to compute. (Since each

neuron in the $L-2$-th layer influences each neuron in the $L-1$-th layer, which each influences each neuron in the L-th layer, which all influence the cost function. Essentially, each neuron in the $L-2$-th layer influences the cost function through $n_{L-1}n_L$ paths, making the partial derivative computation require nested summations. In fact, each neuron in the $L-j$-th layer influences the cost function through $\prod_{i=0}^{j-1} n_{L-i}$ paths.)

Considering Multiple Training Data Points

So far, we've only considered what backpropagation looks like for a single training data point, $(\mathbf{a}^{(0)}, \mathbf{y})$. How might we generalize this to multiple training points?

We need to update our cost function so that it sums up the costs of multiple training trials. Say we have N data points to train off of, each data point being $(\mathbf{a}_l^{(0)}, \mathbf{y}_l)$ for $l \in [N]$. We have

$$C = \sum_{l=1}^{N} ||\mathbf{a}_l^{(L)} - \mathbf{y}_l||^2 = \sum_{l=1}^{N} \sum_{j=1}^{n_L} ((a_l)_j^{(L)} - (y_l)_j)^2.$$

And we update our partial derivatives accordingly. Though the more complex indexing makes the equation seem more menacing, remember that we are only just summing over all training trials.

$$\begin{aligned}
\frac{\partial C}{\partial w_{ij}^{(L)}} &= \sum_{l=1}^{N} \frac{\partial C}{\partial (a_l)_i^{(L)}} \frac{\partial (a_l)_i^{(L)}}{\partial (z_l)_i^{(L)}} \frac{\partial (z_l)_i^{(L)}}{\partial w_{ij}^{(L)}} \\
\frac{\partial C}{\partial b_i^{(L)}} &= \sum_{l=1}^{N} \frac{\partial C}{\partial (a_l)_i^{(L)}} \frac{\partial (a_l)_i^{(L)}}{\partial (z_l)_i^{(L)}} \frac{\partial (z_l)_i^{(L)}}{\partial b_i^{(L)}} \\
\frac{\partial C}{\partial w_{ij}^{(L-1)}} &= \sum_{l=1}^{N} \sum_{k=1}^{n_L} \left(\frac{\partial C}{\partial (a_l)_k^{(L)}} \frac{\partial (a_l)_k^{(L)}}{\partial (a_l)_i^{(L-1)}} \right) \frac{\partial (a_l)_i^{(L-1)}}{\partial (z_l)_i^{(L-1)}} \frac{\partial (z_l)_i^{(L-1)}}{\partial w_{ij}^{(L-1)}} \\
\frac{\partial C}{\partial b_i^{(L-1)}} &= \sum_{l=1}^{N} \sum_{k=1}^{n_L} \left(\frac{\partial C}{\partial (a_l)_k^{(L)}} \frac{\partial (a_l)_k^{(L)}}{\partial (a_l)_i^{(L-1)}} \right) \frac{\partial (a_l)_i^{(L-1)}}{\partial (z_l)_i^{(L-1)}} \frac{\partial (z_l)_i^{(L-1)}}{\partial b_i^{(L-1)}}
\end{aligned}$$

Backpropagation: Utilizing Matrix/Vector Notation

We explore how to calculate such partial derivatives with matrix operations.

First, we handle notation. Define the following for layer ℓ:

- $W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$: A matrix of weights $w_{ij}^{(\ell)}$ that each neuron j in the previous layer has on the neuron i.
- $\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$: A vector of biases $b_i^{(\ell)}$ for each neuron i.
- $\mathbf{z}^{(\ell)} \in \mathbb{R}^{n_\ell}$: A vector of pre-activation values $z_i^{(\ell)}$ for each neuron i.
- $\mathbf{a}^{(\ell)} \in \mathbb{R}^{n_\ell}$: A vector of neuron activations, where $\mathbf{a}^{(\ell)} = \mu(\mathbf{z}^{(\ell)})$.

So, the following relationships hold:

$$\begin{bmatrix} z_1^{(\ell)} \\ z_2^{(\ell)} \\ \vdots \\ z_{n_\ell}^{(\ell)} \end{bmatrix} = \begin{bmatrix} w_{11}^\ell & w_{12}^\ell & \dots & w_{1n_{\ell-1}}^\ell \\ w_{21}^\ell & w_{22}^\ell & \dots & w_{2n_{\ell-1}}^\ell \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_\ell 1}^\ell & w_{n_\ell 2}^\ell & \dots & w_{n_\ell n_{\ell-1}}^\ell \end{bmatrix} \begin{bmatrix} a_1^{(\ell-1)} \\ a_2^{(\ell-1)} \\ \vdots \\ a_{n_{\ell-1}}^{(\ell-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(\ell)} \\ b_2^{(\ell)} \\ \vdots \\ b_{n_\ell}^{(\ell)} \end{bmatrix} \Rightarrow \mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

$$\begin{bmatrix} a_1^{(\ell)} \\ a_2^{(\ell)} \\ \vdots \\ a_{n_\ell}^{(\ell)} \end{bmatrix} = \mu\left(\begin{bmatrix} z_1^{(\ell)} \\ z_2^{(\ell)} \\ \vdots \\ z_{n_\ell}^{(\ell)} \end{bmatrix} \right) \Rightarrow \mathbf{a}^{(\ell)} = \mu(\mathbf{z}^{(\ell)}).$$

In addition, let

$$\frac{\partial C}{\partial W^{(\ell)}} = \begin{bmatrix} \frac{\partial C}{\partial w_{11}^{(\ell)}} & \frac{\partial C}{\partial w_{12}^{(\ell)}} & \dots & \frac{\partial C}{\partial w_{1n_{\ell-1}}^{(\ell)}} \\ \frac{\partial C}{\partial w_{21}^{(\ell)}} & \frac{\partial C}{\partial w_{22}^{(\ell)}} & \dots & \frac{\partial C}{\partial w_{2n_{\ell-1}}^{(\ell)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial w_{n_\ell 1}^{(\ell)}} & \frac{\partial C}{\partial w_{n_\ell 2}^{(\ell)}} & \dots & \frac{\partial C}{\partial w_{n_\ell n_{\ell-1}}^{(\ell)}} \end{bmatrix}, \quad \frac{\partial C}{\partial \mathbf{b}^{(\ell)}} = \begin{bmatrix} \frac{\partial C}{\partial b_1^{(\ell)}} \\ \frac{\partial C}{\partial b_2^{(\ell)}} \\ \vdots \\ \frac{\partial C}{\partial b_{n_\ell}^{(\ell)}} \end{bmatrix},$$

and all other partial derivatives written similarly have a similar matrix-ized/vectorized definition.

We seek an explicit equation for such partial derivatives. For simplicity, let's only consider a single test input/output pair (as generalizing to many test trials is simple since it only requires adding a summation).

From the previous section, we've seen the importance of the partial derivative of C with respect to the pre-activation values of each neuron $z_i^{(\ell)}$ is an essential term used to compute all the partial derivatives we seek. We will denote this partial derivative as

$$\delta_i^\ell = \frac{\partial C}{\partial z_i^{(\ell)}}, \quad \boldsymbol{\delta}^\ell = \frac{\partial C}{\partial \mathbf{z}^{(\ell)}}.$$

And so, we have

$$\frac{\partial C}{\partial w_{ij}^{(\ell)}} = \frac{\partial C}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \delta_i^{(\ell)} \frac{\partial z_i^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \delta_i^{(\ell)} a_j^{(\ell-1)}$$
$$\frac{\partial C}{\partial b_i^{(\ell)}} = \frac{\partial C}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} = \delta_i^{(\ell)} \frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} = \delta_i^{(\ell)}.$$

Before we discuss how to calculate $\delta_i^{(\ell)}$, let's assume we have already calculated it. Our partial derivatives become simple, as noted above. For the partial derivatives with respect to $w_{ij}^{(\ell)}$, note that the first index corresponds to the index of the $\delta^{(\ell)}$ term and the second index corresponds to the index of the $a^{(\ell-1)}$ term. Similarly, the same can be said for the partial derivatives with respect to $b_i^{\ell)}$. So, we can generalize the above equations with matrix/vector notation.

$$\frac{\partial C}{\partial W^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} (\mathbf{a}^{(\ell-1)})^T$$
$$\frac{\partial C}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}$$

Note how this looks much cleaner than the result for $\ell = L - 1$ derived in the earlier section, and this holds for all ℓ !

Now we face the menacing question of how to actually calculate $\boldsymbol{\delta}^{(\ell)}$. Recall from the chain rule we have

$$\delta_i^{(\ell)} = \frac{\partial C}{\partial z_i^{(\ell)}} = \frac{\partial C}{\partial a_i^{(\ell)}} \frac{\partial a_i^{(\ell)}}{\partial z_i^{(\ell)}}.$$

In the case of $\ell = L$ (the last layer), this is an easy calculation we've done before.

$$\delta_i^{(L)} = \frac{\partial C}{\partial z_i^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} = 2(a_i^{(L)} - y_i)\mu'(z_i^{(L)}).$$

To generalize this in vector form, note that we have two vectors being multiplied element-wise. Thus, we utilize the Hadamard product (denoted by $\odot$), which is an operator on two matrices (or vectors) of the same size that returns a matrix (vector) of the corresponding multiplied elements (think of this as a naive matrix multiplication, as compared to how matrix multiplication (and the vector dot product) actually operate). So, we have

$$\boldsymbol{\delta}^{(L)} = \frac{\partial C}{\partial \mathbf{z}^{(L)}} = 2(\mathbf{a}^{(L)} - \mathbf{y}) \odot \mu'(\mathbf{z}^{(L)})$$

.

With this formula for $\ell = L$, we can generalize this to all other ℓ recursively, i.e., knowing $\boldsymbol{\delta}^{(\ell)}$ allows us to determine $\boldsymbol{\delta}^{(\ell-1)}$. Remember that every neuron in layer ℓ is a function of every neuron in layer $\ell - 1$, so we need to include a summation over all those partial derivatives (as we show below). For arbitrary $\ell \in [L]$, the chain rule tells us that

$$\delta_i^{(\ell-1)} = \frac{\partial C}{\partial z_i^{(\ell-1)}} = \sum_{k=1}^{n_\ell} \frac{\partial C}{\partial z_k^{(\ell)}} \frac{\partial z_k^{(\ell)}}{\partial z_i^{(\ell-1)}} = \sum_{k=1}^{n_\ell} \delta_k^\ell \frac{\partial z_k^{(\ell)}}{\partial z_i^{(\ell-1)}}$$

Consider the latter term $\frac{\partial z_k^{(\ell)}}{\partial z_i^{(\ell-1)}}$. We find this partial derivative through the relation

$$z_k^{(\ell)} = \sum_{j=1}^{n_\ell} w_{kj}^{(\ell)} a_j^{(\ell-1)} + b_k^{(\ell)} = \sum_{j=1}^{n_\ell} w_{kj}^{(\ell)} \mu(z_j^{(\ell-1)}) + b_k^{(\ell)}$$

Differentiating, we see that only the i-th term in the summation survives, yielding

$$\frac{\partial z_k^\ell}{\partial z_i^{\ell-1}} = w_{ki}\mu'(z_i^\ell).$$

Substituting into our equation for $\delta_i^{(\ell-1)}$, we have

$$\delta_i^{(\ell-1)} = \sum_{k=1}^{n_\ell} \delta_k^\ell w_{ki} \mu'(z_i^\ell) = \mu'(z_i^\ell) \sum_{k=1}^{n_\ell} \delta_k^\ell w_{ki}.$$

The summation can be thought of as a dot product between $\boldsymbol{\delta}^{(\ell)}$ and the i-th column of $W^{(\ell)}$, or alternatively, the i-th element of the product $(W^{(\ell)})^T \boldsymbol{\delta}^{(\ell)}$ (verify this!). So, in vector notation we have

$$\boldsymbol{\delta}^{(\ell-1)} = \mu'(\mathbf{z}^{(\ell)}) \odot (W^{(\ell)})^T \boldsymbol{\delta}^{(\ell)}$$

Conceptually, multiplying $\boldsymbol{\delta}^{(\ell)}$ by the transpose of the weight matrix serves to "backpropagate the error" from one layer to the previous.

With this, we can determine $\boldsymbol{\delta}^{(\ell)}$ for all layers ℓ recursively. For example, for the second to last layer $L - 1$ we have

$$\begin{aligned}\boldsymbol{\delta}^{(L-1)} &= \mu'(\mathbf{z}^{(L)}) \odot (W^{(L)})^T \boldsymbol{\delta}^{(L)} \\ &= \mu'(\mathbf{z}^{(L)}) \odot \left(2(W^{(L)})^T((\mathbf{a}^{(L)} - \mathbf{y}) \odot \mu'(\mathbf{z}^{(L)}))\right),\end{aligned}$$

which can be used to compute $\boldsymbol{\delta}^{(L-2)}$, etc. This is the heart of backpropagation.

With the knowledge of computing these terms $\boldsymbol{\delta}^{(\ell)}$, we can easily compute the partial derivatives of the cost function with respect to the weights and biases of any layer.

We make one last remark. Recall that these equations are for only one test trial, when in reality our cost function is defined as summing over N trials. So, as a final summary, for some test $l \in [N]$ we have

$$\begin{aligned}\mathbf{a}_l^{(\ell)} &= \mu(\mathbf{z}_l^{(\ell)}) \\ \mathbf{z}_l^{(\ell)} &= W^{(\ell)}\mathbf{a}_l^{(\ell-1)} + \mathbf{b}^{(\ell)}.\end{aligned}$$

Our cost function is then given by

$$C(W^{(1)}, \mathbf{b}^{(1)}, \ldots, W^{(L)}, \mathbf{b}^{(L)}) = \frac{1}{N}\sum_{l=1}^{N} ||\mathbf{a}_l^{(L)} - \mathbf{y}_l||^2.$$

And its partial derivatives as

$$\begin{aligned}\frac{\partial C}{\partial W^{(\ell)}} &= \sum_{l=1}^{N} \boldsymbol{\delta}_l^{(\ell)} (\mathbf{a}_l^{(\ell-1)})^T \\ \frac{\partial C}{\partial \mathbf{b}^{(\ell)}} &= \sum_{l=1}^{N} \boldsymbol{\delta}_l^{(\ell)},\end{aligned}$$

where

$$\boldsymbol{\delta}_l^{(\ell)} = \frac{\partial C}{\partial \mathbf{z}_l^{(\ell)}} = \frac{\partial C}{\partial \mathbf{a}_l^{(\ell)}} \frac{\partial \mathbf{a}_l^{(\ell)}}{\partial \mathbf{z}_l^{(\ell)}},$$

which has a recursive formula given by

$$\begin{aligned}\boldsymbol{\delta}_l^{(L)} &= 2(\mathbf{a}_l^{(L)} - \mathbf{y}_l) \odot \mu'(\mathbf{z}_l^{(L)}) \\ \boldsymbol{\delta}_l^{(\ell-1)} &= \mu'(\mathbf{z}_l^{(\ell)}) \odot (W^{(\ell)})^T \boldsymbol{\delta}_l^{(\ell)}.\end{aligned}$$

We conclude our exploration here.

Worked Example: Basic Handwritten Digit Classifier

We will describe a simple example of a neural network of two layers: one hidden layer and one output layer.

We utilize the MNIST (Modified National Institute of Standards and Technology) database, which consists of a pairing of a 28×28 grid of grayscale pixels (the handwritten digit) and the associated digit it is supposed to represent. Black is the background color and white is the foreground (i.e. the digits are drawn on a black canvas with a white pen). These pixels have an associated numerical value from 0 (black) to 1.00 (white). So, for each testing data point, we have an input size of $28 \times 28 = 784$ neurons each with an activation between $[0, 1]$.

We can construct our neural network in the following way. The first and second (output) layer consists each of 10 neurons. We might expect to simply output a single neuron, since we want our model to output a single predicted digit. However, we note that this model is classifying an input as one of ten different digits, not running some fancy calculation that mathematically finds its way to a number between 0 and 9 (the number itself is just a category). So, the output layer of neurons will represent the probabilities that the model thinks the image can belong to. For example, the fourth neuron $a_4^{(2)}$ represents the probability that the input image is the digit 3 (recall we are using a 1-indexed system).

So, the activations of the last layer must add to 1. We can do this by utilizing the softmax function as our mystery activation function μ for the last layer. The softmax function inputs a vector and outputs a vector of probabilities and is defined element-wise as

$$(\text{Softmax}(\mathbf{v}))_i = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}.$$

For the first layer, we do not have this constraint. So, let's use the *ReLU* function instead. (We prefer *ReLU* since backpropogating requires taking the derivative of this function, which for *ReLU* is always either 1 or 0.)

A perfect neural network would output 1 for the digit the input image represents and 0 for all other digits. We call such a vector where its i-th element is 1 and every other element 0 as the i-th "one-hot" vector. So, say the handwritten digit we input is a 3, so our expected output vector should be the fourth "one-hot" vector $(0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$. Let us denote the i-th "one-hot" vector as e_i (a reference to the canonical basis vectors), and so we have that for all $\mathbf{y}_l$, $l \in [N]$, we have $\mathbf{y}_l \in \{e_i : i \in [10]\}$.

Forward Propagation

We first perform forward propagation, i.e., running our neural network through all test trials to obtain the necessary values of $\mathbf{a}_l^{(1)}, \mathbf{z}_l^{(1)}, \mathbf{a}_l^{(2)}, \mathbf{z}_l^{(2)}$ (which we use alongside the given input-output

pair $\mathbf{a}_l^{(0)}, \mathbf{y}_l$ and the current parameters $W^{(1)}, \mathbf{b}^{(1)}, W^{(2)}, \mathbf{b}^{(2)}$) for our backpropagation calculations.

We utilize the numpy library in python to perform matrix operations. First, we ensure our data is correctly formatted. Our input vector is a vertical vector of size 784 and each output vector is the corresponding "one-hot" e_i.

Next, we initialize a random assortment of weights and biases. The input 784 neurons are propagated forward into the 10 neurons in the hidden layer, which are then propagated into the 10 neurons in the output layer. So, our weights and biases have the following dimensions:

- $W^{(1)}$: A 10×784 matrix.
- $W^{(2)}$: A 10×10 matrix.
- $\mathbf{b}^{(1)}$: A (vertical) vector of size 10.
- $\mathbf{b}^{(2)}$: A (vertical) vector of size 10.

(This means our cost function has 7960 parameters we need to optimize!)

We now calculate the following and store them as vectors to use later for calculation.

$$\begin{aligned}
\mathbf{z}_l^{(1)} &= W^{(1)}\mathbf{a}_l^{(0)} + \mathbf{b}_l^{(1)} \\
\mathbf{a}_l^{(1)} &= ReLU(\mathbf{z}_l^{(1)}) \\
\mathbf{z}_l^{(2)} &= W^{(1)}\mathbf{a}_l^{(1)} + \mathbf{b}_l^{(2)} \\
\mathbf{a}_l^{(2)} &= \text{Softmax}(\mathbf{z}_l^{(2)})
\end{aligned}$$

Backpropagation

For the majority of this paper we have used a "squared-difference" cost function, which gives only a single example of what a cost function should do. In fact, a cost function, like our activation function μ, can be arbitrary, as long as it generally accomplishes its intended purpose (to somehow quantify an error as a differentiable function that we can minimize with calculus).

Because of our choice to use the softmax activation function in the last layer, determining our $\boldsymbol{\delta}_l^{(2)}$ term will force us to compute the derivative of the softmax function, since

$$\boldsymbol{\delta}_l^{(2)} = \frac{\partial C}{\partial \mathbf{a}_l^{(2)}} \odot \text{Softmax}'(\mathbf{z}_l^{(2)}).$$

Attempts at taking the derivative of the softmax function take us down a heinous path of computation. Specifically, we obtain the Jacobian matrix defined element-wise as

$$\text{Softmax}'(\mathbf{z})_{ij} = \begin{cases} \text{Softmax}(z_i)(1 - \text{Softmax}(z_i)) & \text{if } i = j, \\ -\text{Softmax}(z_i)\text{Softmax}(z_j) & \text{if } i \neq j. \end{cases}$$

In these scenarios, recall we have free will in choosing the cost function. More importantly, the actual cost function itself is not explicitly implemented (in code) in our algorithm. So, we might as well pick a cost function with a desirable derivative. Instead of a squared-distance cost function, consider specially engineered "cross-entropy" cost function, which for a single test trial takes the form

$$C = -\sum_{i=1}^{n_L} y_i \log a_i^{(L)}.$$

In context, our cost function is

$$C(W^{(1)}, \mathbf{b}^{(1)}, W^{(2)}, \mathbf{b}^{(2)}) = \frac{1}{N} \sum_{l=1}^{N} \left(-\sum_{i=1}^{10} (y_l)_i \log(a_l)_i^{(2)} \right).$$

Verity that such a cost function makes sense in the context of this problem. Recall vectors $\mathbf{y}_l$ are "one-hot", so when $(y_l)_i$ is zero there is no contribution to the sum. When $(y_l)_i$ is one and $((a_l)_i^{(2)}$ is one (as desired), the logarithm collapses the contribution to zero. The further $((a_l)_i^{(2)}$ is from one when it should be one, the more negative the logarithm becomes contribution, hence the negative sign in the equation. Essentially, this cross-entropy cost function measures how far the element in $\mathbf{a}_l^{(2)}$ that should be one is from one.

The advantage of choosing such a cost function is that it's derivative with respect to $\mathbf{a}_l^{(2)}$ cancels out nicely with the softmax derivative to obtain

$$\boldsymbol{\delta}_l^{(2)} = \mathbf{a}_l^{(2)} - \mathbf{y}_l$$

The proof of this is omitted here for simplicity. If interested, a computation-heavy proof is in the appendix.

This result also allows us to calculate $\boldsymbol{\delta}^{(1)}$. Remember that in the first layer, we are using *ReLU*, not Softmax. So, we have

$$\boldsymbol{\delta}_l^{(1)} = ReLU'(\mathbf{z}_l^{(1)}) \odot (W^{(1)})^T \boldsymbol{\delta}_l^{(2)}.$$

And the derivative of *ReLU* is a simple computation, since

$$ReLU'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}.$$

Now we can calculate the desired partial derivatives:

$$\frac{\partial C}{\partial W^{(1)}} = \frac{1}{N}\sum_{l=1}^{N} \boldsymbol{\delta}_l^{(1)} (\mathbf{a}_l^{(0)})^T$$
$$\frac{\partial C}{\partial W^{(2)}} = \frac{1}{N}\sum_{l=1}^{N} \boldsymbol{\delta}_l^{(2)} (\mathbf{a}_l^{(1)})^T$$
$$\frac{\partial C}{\partial \mathbf{b}^{(1)}} = \frac{1}{N}\sum_{l=1}^{N} \boldsymbol{\delta}_l^{(1)}$$
$$\frac{\partial C}{\partial \mathbf{b}^{(2)}} = \frac{1}{N}\sum_{l=1}^{N} \boldsymbol{\delta}_l^{(2)}.$$

One final note. Mathematically, computing the necessary partial derivatives requires summing over all test cases. We can again simplify this process with matrix operations. With powerful computational libraries like numpy, matrix operations can be done quickly and easily. So, we define the following:

- $Z^{(\ell)}$: a matrix with columns $(\mathbf{z}_1^{(\ell)}, \ldots, \mathbf{z}_N^{(\ell)})$
- $A^{(\ell)}$: a matrix with columns $(\mathbf{a}_1^{(\ell)}, \ldots, \mathbf{a}_N^{(\ell)})$
- $Y^{(\ell)}$: a matrix with columns $(\mathbf{y}_1, \ldots, \mathbf{y}_N)$
- $\Delta^{(\ell)}$ to be a matrix with columns $(\boldsymbol{\delta}_1^{(\ell)} \ldots, \boldsymbol{\delta}_N^{(\ell)})$
- $B^{(\ell)}$: a matrix of N columns, each being the bias vector $\mathbf{b}^{(\ell)}$.

.
Our forward propagation computations can be summarized as

$$Z^{(1)} = W^{(1)} A^{(0)} + B^{(1)}$$
$$A^{(1)} = ReLU(Z^{(1)})$$
$$Z^{(2)} = W^{(1)} A^{(1)} + B^{(2)}$$
$$A^{(2)} = \text{Softmax}(Z^{(2)}),$$

and backpropagataion as

$$
\begin{aligned}
\Delta^{(2)} &= A^{(2)} - Y \\
\Delta^{(1)} &= ReLU'(Z^{(1)}) \odot (W^{(1)})^T \Delta^{(2)} \\
\frac{\partial C}{\partial W^{(1)}} &= \frac{1}{N} \Delta^{(1)} (A^{(0)})^T \\
\frac{\partial C}{\partial W^{(2)}} &= \frac{1}{N} \Delta^{(2)} (A^{(1)})^T \\
\frac{\partial C}{\partial \mathbf{b}^{(1)}} &= \frac{1}{N} \sum (\text{columns of } \Delta^{(1)}) \\
\frac{\partial C}{\partial \mathbf{b}^{(2)}} &= \frac{1}{N} \sum (\text{columns of } \Delta^{(1)}).
\end{aligned}
$$

Finally, we update our parameters in the direction opposite of these partial derivatives (a nudge in the direction of the negative gradient). How much should we nudge by? We define a learning rate α (how to choose optimal values of α, a hyperparameter, is outside the scope of this paper) so that we update our parameters in the following way:

$$
\begin{aligned}
W^{(1)}_{\text{new}} &= W^{(1)} - \alpha \frac{\partial C}{\partial W^{(1)}} \\
W^{(2)}_{\text{new}} &= W^{(2)} - \alpha \frac{\partial C}{\partial W^{(2)}} \\
\mathbf{b}^{(1)}_{\text{new}} &= \mathbf{b}^{(1)} - \alpha \frac{\partial C}{\partial \mathbf{b}^{(1)}} \\
\mathbf{b}^{(2)}_{\text{new}} &= \mathbf{b}^{(2)} - \alpha \frac{\partial C}{\partial \mathbf{b}^{(2)}}.
\end{aligned}
$$

We iterate this process over and over again until our neural network is trained to our liking.

Implementation of these ideas can get around 90% accuracy with around 1000 iterations of the gradient decent algorithm with $\alpha = 0.1$. We can improve our results with more hidden layers.

Summary: Multi-Layered Handwritten Digit Classifier

With the recursive formula for $\boldsymbol{\delta}$, adding another layer in our neural network adds an additional step in our forward/backpropagation processes. The following equations summarize an L-layered neural network using a cross-entropy cost function (for arbitrary $\ell \in [L]$):

Notation

- $A^{(0)}$: A column-wise matrix of input data.
- Y: A column-wise matrix of desired outputs (one-hot vectors).
- $A^{(\ell)}$: A column-wise matrix of neuron activations in the ℓ-th layer.
- $Z^{(\ell)}$: A column-wise matrix of pre-activations in the ℓ-th layer.
- $W^{(\ell)}$: A matrix of weights for the ℓ-th layer, where w_{ij} represents the weight the j-th neuron of the $\ell - 1$-th layer has on the i-th neuron of the ℓ-th layer.
- $\mathbf{b}^{(\ell)}$: A vector of biases for the ℓ-th layer.
- $B^{(\ell)}$: A column-wise matrix where each column is $\mathbf{b}^{(\ell)}$.
- $\Delta^{(\ell)}$: A matrix corresponding to the partial derivatives of the cost function with respect to the pre-activation values of the ℓ-th layer, i.e. $\frac{\partial C}{\partial Z^{(\ell)}}$.
- $\mu^{(\ell)}$: The activation function of the ℓ-th layer, defined by

$$\mu^{(\ell)}(x) = \begin{cases} ReLU(x) & \text{for } \ell < L \\ \text{Softmax}(x) & \text{for } \ell = L \end{cases}$$

Forward Propagation

$$\begin{aligned} Z^{(\ell)} &= W^{(\ell)} A^{(\ell-1)} + B^{(\ell)} \\ A^{(\ell)} &= \mu^{(\ell)}(Z^{(\ell)}) \end{aligned}$$

Backpropagation

$$\begin{aligned} \Delta^{(L)} &= A^{(L)} - Y \\ \Delta^{(\ell-1)} &= ReLU'(Z^{(1)}) \odot (W^{(1)})^T \Delta^{(\ell)} \\ \frac{\partial C}{\partial W^{(\ell)}} &= \frac{1}{N} \Delta^{(\ell)} (A^{(\ell-1)})^T \\ \frac{\partial C}{\partial \mathbf{b}^{(\ell)}} &= \frac{1}{N} \sum (\text{columns of } \Delta^{(\ell)}) \end{aligned}$$

Appendix

Proof of $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - \mathbf{y}_l$ for a cross-entropy cost function

Recall our cross-entropy cost function is

$$C = -\sum_{i=1}^{n_L} y_i \log a_i^{(L)}$$

and $\mathbf{a}_l^{(L)}$ given by

$$\mathbf{a}^{(2)} = \text{Softmax}(\mathbf{z}^{(2)}) = \frac{e^{z_i}}{\sum_{j=1}^{n_L} e^{z_j}}.$$

Substituting, we have

$$\begin{aligned} C &= -\sum_{i=1}^{n_L} y_i \log \left(\frac{e^{z_i}}{\sum_{j=1}^{n_L} e^{z_j}} \right) \\ &= -\sum_{i=1}^{n_L} y_i \left(z_i - \log \sum_{j=1}^{n_L} e^{z_j} \right) \\ &= -\sum_{i=1}^{n_L} y_i z_i + \sum_{i=1}^{n_L} y_i \log \sum_{j=1}^{n_L} e^{z_j} \end{aligned}$$

Since $\sum_{i=1}^{n_L} y_i = 1$ (due to the one-hot encoding of $\mathbf{y}_l$), the second term simplifies:

$$C = -\sum_{i=1}^{n_L} y_i z_i + \log \sum_{j=1}^{n_L} e^{z_j}$$

We seek the partial derivative $\boldsymbol{\delta} = \frac{\partial C}{\partial \mathbf{z}^{(L)}}$. For the first term, we have

$$\frac{\partial}{\partial z_k} \left(-\sum_{i=1}^{n_L} y_i z_i \right) = -y_k.$$

For the second term,

$$\frac{\partial}{\partial z_k} \left(\log \sum_{j=1}^{n_L} e^{z_j} \right) = \frac{\partial}{\partial z_k} (\log S) = \frac{1}{S} \frac{\partial S}{\partial z_k},$$

where $S = \sum_{j=1}^{n_L} e^{z_j}$. The derivative of S with respect to z_k is

$$\frac{\partial S}{\partial z_k} = e^{z_k}.$$

Thus

$$\frac{\partial}{\partial z_k}\left(\log\sum_{j=1}^{n_L} e^{z_j}\right) = \frac{e^{z_k}}{\sum_{j=1}^{n_L} e^{z_j}} = a_k^{(L)}.$$

Combine these results, we have

$$\frac{\partial C}{\partial z_k} = -y_k + a_k^{(L)}.$$

In vector form, this is

$$\frac{\partial C}{\partial \mathbf{z}_l^{(2)}} = \mathbf{a}_l^{(2)} - \mathbf{y}_l. \quad \blacksquare$$