Deriving the Training Equations for a Multilayer Linear-ReLU-Softmax Neutral Network with Cross-entropy loss

Guille Cornejo

June 12, 2024

Math is hard. There are many excelent books out there about Neural Networks and Deep Learning, but most of them present the derivation of the training equations in a very general way. Sometimes, it comes handy to see the derivation applied to a very specific use case to finally *get it*.

This document presents a detailed derivation of the training equations (weights and biases update equations) for a multi-layer Linear-ReLU-SoftMax neural network, using cross-entropy as loss function.

Although, in the end, we present a set of equations that are general for any neural network (and they should, more or less, match the equations in any book you read), the derivation is motivated with the very specific use case of training a neural network to classify the FashionMNIST dataset [Xiao et al., 2017].

Figure 1 presents a diagram of the neural network. Heads up! It's used just as a motivation, other network architectures are possible and probably would perform better (convolutional neural networks, for instance).

Section 1 presents a brief recapitulation of the idea of gradient descent, section 2 presents mathematical preliminaries to help you understand the following sections without the need to look outside this document, just a friendly refresh, in case you need it. Nonetheless, some basic knowledge of calculus is required to understand the derivation. The remaining sections present the derivation itself.

Also, it's assumed that you have some idea of what a neural network is. If that isn't the case, we recommend you to first read the introduction and chapter 1 (at the minimum) of [Nielsen, 2015]. It's a free e-book, just Google it. It's quite good! We strongly recommend it, even if you'd already read something else about Deep Learning.

1 Recap of Gradient Descent

Have you ever wondered how a boulder knows which way to move to roll down a hill? Sounds like a naive question, it's not like the boulder has its own mind or anything. You can rephrase it into whatever makes you feel better (read, less dumb): how does the force gravity act over the boulder? How does it know how to follow a geodesic curve on space-time? The fundamental idea doesn't change: there has to be some measurable, physical quantity involved.

Without introducing any math, just following the intuition, if we shift our perspective to that of the boulder, we would *see* that it naturally follows the terrain' slope around it. In math, we have a very general word for that: the *gradient* (derivatives with respect to the terrain).

Figure 2 shows our boulder, rolling down the hill. We could approximate the hill as (yeah, just a regular $y = x^2$ parabola):

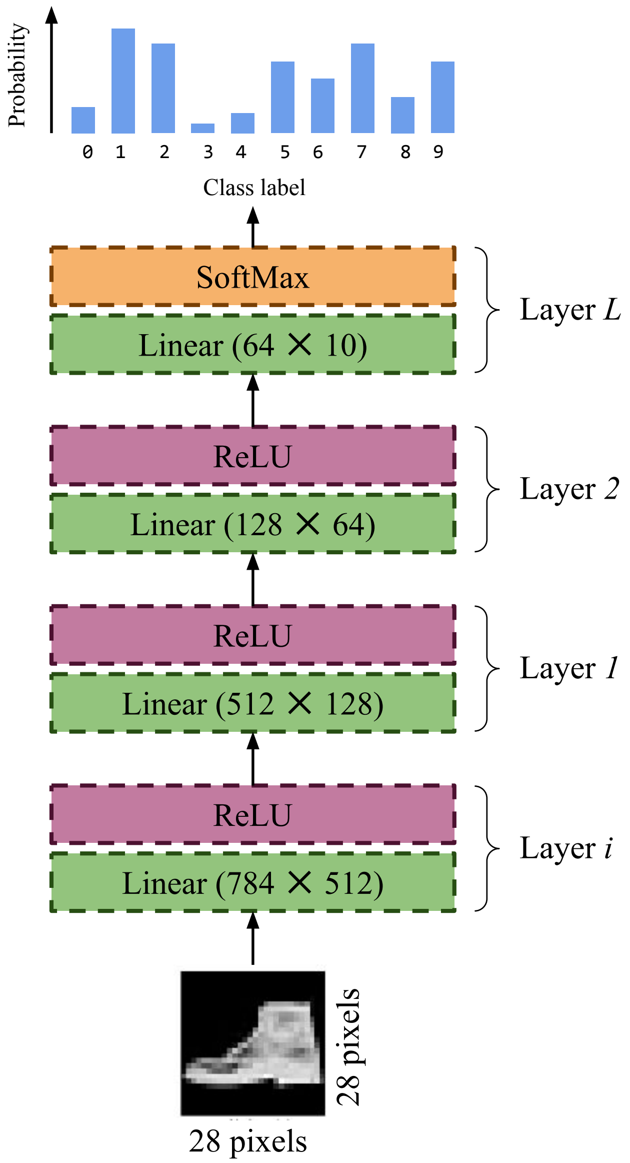

Figure 1: Diagram of the neural network

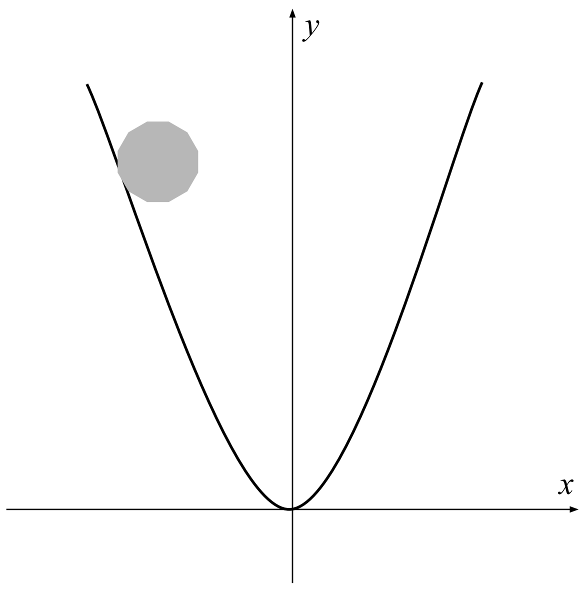

Figure 2: Illustration of a boulder rolling down a hill, it serves as motivation for the gradient descent optimization algorithm

$$Hill(x, y) = x^2 - y$$

Guided by the idea that what the boulder does is to follow the terrain, from one moment to the next, what the update rules should do is to move the coordinates a little, following the *Hill* function, like this:

$$y \longleftarrow y + \eta \frac{\partial Hill}{\partial y} \tag{1}$$

$$x \longleftarrow x + \eta \frac{\partial Hill}{\partial x} \tag{2}$$

Where η is a very small number, meaning that it's just a change from one moment to the next. Solving the partial derivatives:

$$\frac{\partial Hill}{\partial y} = \frac{\partial x^2 - y}{\partial y} = -1$$

$$\frac{\partial Hill}{\partial x} = \frac{\partial x^2 - y}{\partial x} = 2x$$

Thus, the update rules (1) become:

$$y \longleftarrow y - \eta$$
$$x \longleftarrow x + 2\eta x$$

And that makes a lot of sense! As the boulder rolls just a little bit down the hill, y becomes a little smaller and x moves a little to the right. This simple idea is the fundament of the Gradient Descent optimization algorithm.

We'll use that idea to train the neural network. The loss function is going to play the role of the terrain, while the network's weight and biases are the quantities we can change to minimize the loss.

2 Math preliminaries

Friendly refresher about math you (maybe) learned a long time ago. You can skip it if you like. Also, if you're super sensitive about people brushing some math details under the carpet, please go for ice-cream or something :)

2.1 Chain Rule for derivatives

You have functions f and g

$$\frac{df(g(x))}{dx} = \frac{df}{dx}(g(x)) \cdot \frac{dg}{dx}(x)$$

which means: first, calculate the derivative of f, evaluate the resulting function at $g(x)$, then, calculate the derivative of g and evaluate the resulting function at x. Finally, multiply both.

Also, if we have a variable z that depends on variable y, which itself depends on x, we can say:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

2.2 Derivative of a quotient

$$\frac{df(x)}{dg(x)} = \frac{\frac{df(x)}{dx} \cdot g(x) - f(x) \cdot \frac{dg(x)}{dx}}{(g(x))^2}$$

2.3 Nabla (Del) notation

Let's say you have a function f that takes n parameters, as in $f(x_1, x_2, \ldots, x_n)$. You want to calculate how much f changes with respect to each of its parameters, like:

$$\left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

We call that resulting vector *the gradient*, usually, we use ∇ to denote it:

$$\nabla_x f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

Notice the x as subindex in ∇, it indicates with respect to whom to derivate.

2.4 Kronecker Delta function

Defined as:

$$d_{i,j} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

2.5 Absolute value

Denote as $|x|$ is defined as:

$$|x| = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

In other words, it returns the *positive* value of a number. Its derivative is given by the sign function:

$$\frac{d\ |x|}{dx} = sgn(n) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

2.6 Cross-entropy loss function

Defined as:

$$L = -\sum_{k=0}^{n_L} y_k\ log(a_k^L) = \sum_{k=0}^{n_L} y_k\ log\left(\frac{1}{a_k^L}\right)$$

Where:

- y_k is the k-th entry of the one-hot encoded class indicator, $\mathbf{y}$
- a_k^L is the activation of unit k in the output layer (numbered as L)
- n_L is the number of units at layer L (the output layer)

Here is the reason why this function make sense as a loss for a classification task: we want $\mathbf{a^L}$ to be as similar to $\mathbf{y}$ as possible, so they indicate the same class. $\mathbf{y}$ is a one-hot encoded vector, that means the product $y_k log(1/a_k^L)$ is always zero except for the entry indicating the class. Hence, to minimize the loss we need a_k^L to be as large as possible, in other words, we minimize the loss by making the entry corresponding to that class very large, so it matches $\mathbf{y}$.

2.7 Linear Layer

Just in case, this is what I mean when I say *Linear* in figure 1.

$$\mathbf{z^l} = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$

Where:

- $\mathbf{z}^l$ is called the *logits* of layer l
- $\mathbf{W}^l$ is the weights matrix of layer l
- $\mathbf{a}^{l-1}$ is the activations from layer $l-1$. This is, the "output" from the ReLU function from the previous layer. If we're at the first layer, then it means the actual data input (the "x")
- $\mathbf{b}^l$ biases for layer l

We can also write that in index notation:

$$z_i^l = \sum_{j=0}^{n_l} w_{ij}^l a_j^{l-1} + b_j^l$$

Same meaning, with the addition of:

- The index i
- n_l means the size of layer l

2.8 Softmax function

Defined as:

$$Softmax(k, L) = \frac{e^{z_k^L}}{\sum_{j=0}^{n_L} e^{z_j^L}}$$

Where:

- z_k^L is the logit of unit k at layer L
- z_j^L is the logit of unit j at layer L

2.9 ReLU activation function

Defined as:

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

This functions is commonly implemented as $max(0, x)$. The derivative of ReLU is also a function in parts

$$\frac{d\ ReLU(x)}{dx} = ReLU'(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x \neq 0 \end{cases}$$

2.10 Hadamard product

Also known as the element-wise product, defined as:

$$(A \odot B)_{ij} = A_{ij} B_{ij}$$

If you're comfortable with Numpy, this is the regular * operation. A couple of examples:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \odot \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 6 & 8 \\ 15 & 18 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \odot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 5 & 12 \\ 21 & 48 \end{pmatrix}$$

2.11 Outer product

Also know as the Tensor product. Given two vectors, **u** and **v**:

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix}, \ \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

$$(\mathbf{u} \ \otimes \ \mathbf{v})_{i,j} = u_i v_j$$

In matrix form:

$$\mathbf{u} \ \otimes \ \mathbf{v} = \begin{bmatrix} u_1v_1 & u_1v_2 & \dots & u_1v_n \\ u_2v_1 & u_2v_2 & \dots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \dots & u_mv_n \end{bmatrix}$$

3 Training the last layer

When we say *training*, what we mean is decreasing the average loss by tweaking the model's parameters. Let's start with the last layer then! First, the weights. Following the ideas from gradient descent, we want to calculate how much the loss (the "Hill") changes with respect to an individual weight, in other words:

$$\frac{\partial Loss}{\partial w_{ij}^L} = \frac{\partial Loss}{\partial z_i^L}\frac{\partial z_i^L}{\partial w_{ij}^L} \tag{3}$$

I used the Chain rule to split by the logits at layer L. We can now tackle each partial derivative individually.

3.1 $\partial Loss/\partial z_i^L$

Here we want to say how much the loss changes when z_i^L changes a little. First, substitute the definition of the cross-entropy loss:

$$\frac{\partial Loss}{\partial z_i^L} = \frac{\partial}{\partial z_i^L}\left(-\sum_{k=0}^{n_L} y_k \; log(a_k^L)\right)$$

The sum and the partial derivative are linear with respect to each other, so, we can move the derivative into the sum

$$= -\sum_{k=0}^{n_L}\left(y_k \; \frac{\partial}{\partial z_i^L} \; log(a_k^L)\right)$$

Following the chain rule for the composition of functions

$$= -\sum_{k=0}^{n_L}\left(y_k \cdot \frac{1}{a_k^L} \cdot \frac{\partial a_k^L}{\partial z_i^L}\right) \tag{4}$$

3.2 $\partial a_k^L/\partial z_i^L$

Trust me, we're going to need a full section for this derivative. The good news is, it's kinda the half of the work for this derivation. Which is a little unexpected, as at first glance, it seems to have little to do with the back-propagation algorithm itself. Remember that the activation is produced by taking the logits of the layer and applying the *Softmax* function:

$$\begin{aligned}\frac{\partial a_k^L}{\partial z_k^L} &= \frac{\partial}{\partial z_i^L} Softmax(k, L) \\ &= \frac{\partial}{\partial z_i^L}\left(\frac{e^{z_k^L}}{\sum_{j=0}^{n_L} e^{z_j^L}}\right)\end{aligned}$$

Then, applying the quotient rule (take a deep breath, here we go):

$$= \frac{\left(\frac{\partial e^{z_k^L}}{\partial z_i^L} \cdot \sum_{j=0}^{n_L} e^{z_j^L}\right) - \left(e^{z_k^L} \cdot \frac{\partial}{\partial z_i^L}\sum_{j=0}^{n_L} e^{z_j^L}\right)}{\left(\sum_{j=0}^{n_L} e^{z_j^L}\right)^2} \tag{5}$$

Slowly but surely, let's calculate each derivative individually. For this one, notice that if $k = i$, then

$$\frac{\partial e^{z_i^L}}{\partial z_i^L} = e^{z_i^L}$$

Otherwise, if $k \neq i$, then

$$\frac{\partial e^{z_k^L}}{\partial z_i^L} = 0$$

This behaviour obviously calls for the introduction of a Kronecker delta somewhere:

$$\frac{\partial e^{z_k^L}}{\partial z_i^L} = \delta_{k,i} e^{z_i^L}$$

Now, for this other one, we can do the trick of pulling the partial derivative into the sum:

$$\frac{\partial}{\partial z_i^L} \sum_{j=0}^{n^L} e^{z_j^L} = \sum_{j=0}^{n_L} \frac{\partial}{\partial z_i^L} e^{z_j^L}$$

Similarly to the derivative above, all terms for which $j \neq i$ are zero, hence

$$\frac{\partial}{\partial z_i^L} \sum_{j=0}^{n^L} e^{z_j^L} = e^{z_i^L}$$

Plugging them back into equation 5

$$\begin{aligned}
\frac{\partial a_k^L}{\partial z_k^L} &= \frac{\left(\delta_{k,i} e^{z_i^L} \cdot \sum_{j=0}^{n_L} e^{z_j^L}\right) - \left(e^{z_k^L} \cdot e^{z_i^k}\right)}{\left(\sum_{j=0}^{n_L} e^{z_j^L}\right)^2} \\
&= \frac{e^{z_i^L}}{\sum_{j=0}^{n_L} e^{z_j^L}} \cdot \frac{\delta_{k,i} \sum_{j=0}^{n_L} e^{z_j^L} - e^{z_k^L}}{\sum_{j=0}^{n_L} e^{z_j^L}}
\end{aligned}$$

That last step was a little bit of algebraic re-arrangement: factor out $e^{z_i^L}$, split in two fractions. Hopefully, you can see that the first fraction is the definition of the *Softmax* function. Also, we can split the second fraction by the minus. The sums cancel each other, leaving only the delta. The rightmost expression is another *Softmax*. (remember, $a_j^L = Softmax(j, L)$).

$$\frac{\partial a_k^L}{\partial z_i^L} = a_i^L(\delta_{k,i} - a_k^L) \tag{6}$$

3.3 Back to $\partial Loss / \partial z_i^L$

Plugging (eq 6) into (eq 4):

$$\begin{aligned}
\frac{\partial Loss}{\partial z_i^L} &= -\sum_{k=0}^{n_L} \left(y_k \cdot \frac{1}{a_k^L} \cdot \frac{\partial a_k^L}{\partial z_i^L}\right) \\
&= -\sum_{k=0}^{n_L} \left(y_k \cdot \frac{1}{a_k^L} \cdot a_i^L(\delta_{k,i} - a_k^L)\right)
\end{aligned}$$

What follows is a bunch of algebra, if you get lost, don't worry! Slow down, go to the previous line a check. There is nothing hard, only symbols' carpentry. We can "extract" the $k = i$ from the previous expression:

$$
\begin{aligned}
&= -(y_i \cdot \frac{1}{a_i^L} \cdot a_i^L(\delta_{i,i} - a_i^L)) - \sum_{k=0,k\neq i}^{n_L} y_k \cdot \frac{1}{a_k^L} \cdot a_i^L(\delta_{k,i} - a_k^L) \\
&= -y_i(1 - a_i^L) - \sum_{k=0,k\neq i}^{n_L} y_k \cdot \frac{1}{a_k^L} \cdot a_i^L(0 - a_k^L) \\
&= -y_i + y_i a_i^L - \sum_{k=0,k\neq i}^{n_L} y_k \cdot -a_i^L \\
&= -y_i + a_i^L y_i + a_i^L \sum_{k=0,k\neq i}^{n_L} y_k \\
&= -y_i + a_i^L(y_i + \sum_{k=0,k\neq i}^{n_L} y_k) \\
&= -y_i + a_i^L \sum_{k=0}^{n_L} y_k
\end{aligned}
$$

Still here? Good :) Now, $\mathbf{y}$ is a one-hot encoded vector, therefore $\sum_{k=0}^{n_L} y_k = 1$, hence:

$$\frac{\partial Loss}{\partial z_i^L} = a_i^L - y_i \tag{7}$$

3.4 $\partial z_i^L / \partial w_{ij}^L$

This one is easy, substituting the definition of the logit:

$$
\begin{aligned}
\frac{\partial z_i^L}{\partial w_{ij}^L} &= \frac{\partial}{\partial w_{ij}^L}\left(\sum_{k=0}^{n_L} w_{ik}^L a_k^{L-1} + b_k^L\right) \\
&= \sum_{k=0}^{n_L} \frac{\partial}{\partial w_{ij}^L}\left(w_{ik}^L a_k^{L-1} + b_k^L\right) \\
&= a_j^{L-1}
\end{aligned} \tag{8}
$$

To understand that last line, notice that the partial derivative evaluates to zero for all cases except for $k = j$

3.5 $\partial Loss / \partial w_{ij}^L$

Substituting (eq 7) and (eq 8):

$$
\begin{aligned}
\frac{\partial Loss}{\partial w_{ij}^L} &= \frac{\partial Loss}{\partial z_i^L}\frac{\partial z_i^L}{\partial w_{ij}^L} \\
&= (a_i^L - y_i)\ a_j^{L-1}
\end{aligned} \tag{9}
$$

I've been using index notation for simplicity, but "in real life", we will use vector and matrix operations. It could be a little hard to see how to transform from index notation into matix notation, but we can use a very simple example with "numbers" to figure out the pattern. For instance, we can restrict indexes to $i = 0, 1, 2, 3$ and $j = 0, 1, 2$:

$$
\begin{bmatrix}
\frac{\partial Loss}{\partial w_{00}^L} & \frac{\partial Loss}{\partial w_{01}^L} & \frac{\partial Loss}{\partial w_{02}^L} \\
\frac{\partial Loss}{\partial w_{10}^L} & \frac{\partial Loss}{\partial w_{11}^L} & \frac{\partial Loss}{\partial w_{12}^L} \\
\frac{\partial Loss}{\partial w_{20}^L} & \frac{\partial Loss}{\partial w_{21}^L} & \frac{\partial Loss}{\partial w_{22}^L} \\
\frac{\partial Loss}{\partial w_{30}^L} & \frac{\partial Loss}{\partial w_{31}^L} & \frac{\partial Loss}{\partial w_{32}^L}
\end{bmatrix}
$$

$$
= \begin{bmatrix}
(a_0^L - y_0)a_0^{L-1} & (a_0^L - y_0)a_1^{L-1} & (a_0^L - y_0)a_2^{L-1} \\
(a_1^L - y_1)a_0^{L-1} & (a_1^L - y_1)a_1^{L-1} & (a_1^L - y_1)a_2^{L-1} \\
(a_2^L - y_2)a_0^{L-1} & (a_2^L - y_2)a_1^{L-1} & (a_2^L - y_2)a_2^{L-1} \\
(a_3^L - y_3)a_0^{L-1} & (a_3^L - y_3)a_1^{L-1} & (a_3^L - y_3)a_2^{L-1}
\end{bmatrix}
$$

$$
= \begin{bmatrix} (a_0^L - y_0) \\ (a_1^L - y_1) \\ (a_2^L - y_2) \\ (a_3^L - y_3) \end{bmatrix} \otimes \begin{bmatrix} a_0^{L-1} \\ a_1^{L-1} \\ a_2^{L-1} \end{bmatrix}
$$

Writing (eq 9) in full matrix form:

$$
\nabla_{\mathbf{W^L}} Loss = (\mathbf{a^L} - \mathbf{y}) \otimes \mathbf{a^{L-1}} \tag{10}
$$

4 What about the last layer's biases?

We're almost done there!

$$
\frac{\partial Loss}{\partial b_i^L} = \frac{\partial Loss}{\partial z_i^L} \frac{\partial z_i^L}{\partial b_i^L}
$$

We already know what $\partial Loss/\partial z_i^L$ is (eq 7), we just have to figure out $\partial z_i^L/\partial b_i^L$. Substituting with the definition for logits:

$$
\begin{aligned}
\frac{\partial z_i^L}{\partial b_i^L} &= \frac{\partial}{\partial b_i^L} \left(\sum_{k=0}^{n_L} w_{kj}^L a_k^{L-1} + b_k^L \right) \\
&= \sum_{k=0}^{n_L} \left[\frac{\partial}{\partial w_{ij}^L} \left(w_{kj}^L a_k^{L-1} + b_k^L \right) \right] \\
&= 1
\end{aligned}
$$

For all cases when $k \neq i$ the derivative is zero, for $k = i$ it's 1. Hence:

$$
\frac{\partial Loss}{\partial b_i^L} = \frac{\partial Loss}{\partial z_i^L} \frac{\partial z_i^L}{\partial b_i^L} = a_i^L - y_i \tag{11}
$$

In matrix form:

$$
\nabla_{\mathbf{b^L}} Loss = \mathbf{a^L} - \mathbf{y} \tag{12}
$$

5 Training the other layers

Ok, let's say that, after training the last layer (L), we move back one layer ($L-1$) to attempt to derive training equations for it. Following a definition analogous to (eq 3):

$$\frac{\partial Loss}{\partial w_{ij}^{L-1}} = \frac{\partial Loss}{\partial z_i^{L-1}} \frac{\partial z_i^{L-1}}{\partial w_{ij}^{L-1}} \tag{13}$$

Now, this is totally fine. It's a definition, it's true in that sense. We could go on and derive the expression in terms of the activations and weights, as we did for the last layer. But then, what about $L-2$? What if the network has 24 layers or something like that? We can do better.

Instead of deriving a set of expressions for each layer, we can write those training equations "recursively". We can claim that, for any layer, (eq 3) transforms into:

$$\frac{\partial Loss}{\partial w_{ij}^{l}} = \frac{\partial Loss}{\partial z_i^{l}} \frac{\partial z_i^{l}}{\partial w_{ij}^{l}} \tag{14}$$

This obviously reduces to (eq 3) when $l = L$, but also works for all the other layers! We're doing good progress here. The goal is then to figure out how those partial derivatives look like in *recursive* form.

5.1 $\partial z_i^l / \partial w_{ij}^l$

The easy one first. Using a similar reasoning, there is nothing special about (eq 8), we can swap L by l and we're done:

$$\frac{\partial z_i^{l}}{\partial w_{ij}^{l}} = a_j^{l-1} \tag{15}$$

5.2 $\partial Loss / \partial z_i^l$

Hold your horses! Before you apply the chain rule here and say something silly like

$$\frac{\partial Loss}{\partial z_i^{L-1}} = \frac{\partial Loss}{\partial z_i^{L}} \frac{\partial z_i^{L}}{\partial z_i^{L-1}}$$

Think why that is **NOT** true. You don't need a bunch of math, just intuition. $\partial L / \partial z_i^{L-1}$ means by how much the loss changes due to a little wiggle in z_i^{L-1}. Yeah, we want to bridge that using something from layer L, obviously we need the chain rule somehow. But notice that changing z_i^{L-1} would change ALL the logits in layer L, not only z_i^L (I mean, not only the logit with index i).

Luckily, we can just add together all those little wiggles!

$$\frac{\partial Loss}{\partial z_i^{L-1}} = \sum_{k=0}^{n_L} \frac{\partial Loss}{\partial z_k^{L}} \frac{\partial z_k^{L}}{\partial z_i^{L-1}} \tag{16}$$

We already know that $\partial L / \partial z_k^L = a_k^L - y_i$ from (eq 7), but in general, we'll always know $\partial L / \partial z_k^l$ when calculating the gradients for $l-1$. So, we can focus on the second derivative. Substituting with the definition of the logits:

$$\frac{\partial z_k^{L}}{\partial z_i^{L-1}} = \frac{\partial}{\partial z_i^{L-1}} \left(\sum_{j=0}^{n_L} w_{kj}^{L} a_j^{L-1} + b_j^{L} \right)$$

Now, remember that the activations are simply the logits passed through the *ReLU* function:

$$= \frac{\partial}{\partial z_i^{L-1}} \left(\sum_{j=0}^{n_L} w_{kj}^L ReLU(z_j^{L-1}) + b_j^L \right)$$
$$= \sum_{j=0}^{n_L} \frac{\partial}{\partial z_i^{L-1}} \left(w_{kj}^L ReLU(z_j^{L-1}) + b_j^L \right)$$

As in other situations, the derivative is non-zero only if $j = i$

$$= w_{ki}^L ReLU'(z_i^{L-1})$$

Therefore:

$$\frac{\partial Loss}{\partial z_i^{L-1}} = \sum_{k=0}^{n_L} \frac{\partial Loss}{\partial z_k^L} w_{ki}^L ReLU'(z_i^{L-1})$$

Substituting both derivatives in (eq 13):

$$\begin{aligned} \frac{\partial Loss}{\partial w_{ij}^{L-1}} &= \frac{\partial Loss}{\partial z_i^{L-1}} \frac{\partial z_i^{L-1}}{\partial w_{ji}^{L-1}} \\ &= \left(\sum_{k=0}^{n_L} \frac{\partial L}{\partial z_k^L} w_{ki}^L ReLU'(z_i^{L-1}) \right) \; a_j^{L-2} \\ &= ReLU'(z_i^{L-1}) \; \left(\sum_{k=0}^{n_L} w_{ki}^L \frac{\partial L}{\partial z_k^L} \right) \; a_j^{L-2} \end{aligned}$$

That last line was just terms re-arrangement. Remember that our goal here is to write general equations for any layer, so, we can transform that into a "recursive" equation, in the same spirit of (eq 14):

$$\frac{\partial Loss}{\partial w_{ij}^l} = ReLU'(z_i^l) \; \left(\sum_{k=0}^{n_{l+1}} w_{ki}^{l+1} \frac{\partial Loss}{\partial z_k^{l+1}} \right) \; a_j^{l-1} \tag{17}$$

Also, we want to work with these equations in matrix form. Hopefully, you can see that the $ReLU$ and a_j^{l-1} are Hadamard product and outer product, respectively, in analogy to the derivation of (eq 10). But the sum is harder to see. If you have any difficulty seeing how the sum transforms into matrix multiplication in the expression below, I advise you to use the same trick I used to figure out the pattern while deriving (eq 10). Without further ado, (eq 17) in matrix form reads as:

$$\nabla_{W^l} Loss = [ReLU'(z^l) \odot ((\mathbf{W}^{l+1})^T \nabla_{z^{l+1}} L)] \otimes \mathbf{a^{l-1}} \tag{18}$$

6 What about the biases?

Dejavu:

$$\frac{\partial Loss}{\partial b_i^{L-1}} = \frac{\partial Loss}{\partial z_i^{L-1}} \frac{\partial z_i^{L-1}}{\partial b_i^{L-1}}$$

We just calculated $\partial Loss/\partial z_i^{L-1}$. also, there is nothing special about the last layer's biases regarding how the logits are calculated, then:

$$\frac{\partial z_i^{L-1}}{\partial b_i^{L-1}} = 1$$

That means we can go ahead and claim that:

$$\nabla_{\mathbf{b}^l} Loss = ReLU'(z^l) \odot ((\mathbf{W}^{l+1})^T \nabla_{z^{l+1}} L) \tag{19}$$

7 Update rules (for any layer!)

Ladies and Gentlemen, with you, following the ideas from the Gradient Descent optimization algorithm, the updates rules for any layer in a multi-layered Linear-ReLU-Softmax neural network using Cross-entry as loss function:

$$\mathbf{W^l} \leftarrow \mathbf{W^l} - \eta \nabla_{\mathbf{W^l}} L$$
$$\mathbf{b^l} \leftarrow \mathbf{b^l} - \eta \nabla_{\mathbf{b^l}} L$$

Together with equations 10, 12, 18 and 19.

8 Adding L1 regularization

L1 regularisation adds a new term to the loss:

$$Loss_{L1} = Loss_v + \lambda \sum_{ijl} |w_{ij}^l| \tag{20}$$

Where $Loss_v$ is the *vanilla* loss from previous sections (cross-entropy) and λ is the regularization parameter.

This new term has no effect on the update rule for the biases, because the partial derivative vanishes (the new term doesn't depend on the biases):

$$\begin{aligned}\frac{\partial Loss_{L1}}{\partial b_i^l} &= \frac{\partial}{\partial b_i^l}\left(Loss_v + \lambda \sum_{ijl} |w_{ij}^l| \right) \\ &= \frac{\partial Loss_v}{\partial b_i^l} + \lambda \frac{\partial}{\partial b_i^l} \sum_{ijl} |w_{ij}^l| \\ &= \frac{\partial Loss_v}{\partial b_i^l}\end{aligned} \tag{21}$$

For the weights, it adds a quantity proportional to the regularization parameter:

$$\begin{aligned}\frac{\partial Loss_{L1}}{\partial w_{ij}^l} &= \frac{\partial}{\partial w_{ij}^l}\left(Loss_v + \lambda \sum_{ijl} |w_{ij}^l| \right) \\ &= \frac{\partial Loss_v}{\partial w_{ij}^l} + \lambda \frac{\partial}{\partial w_{ij}^l} \sum_{ijl} |w_{ij}^l| \\ &= \frac{\partial Loss_v}{\partial w_{ij}^l} + \lambda \sum_{ijl} \frac{\partial |w_{ij}^l|}{\partial w_{ij}^l} \\ &= \frac{\partial Loss_v}{\partial w_{ij}^l} + \lambda sgn(w_{ij}^l)\end{aligned} \tag{22}$$

References

[Nielsen, 2015] Nielsen, M. (2015). *Neural Networks and Deep Learning*. Determination Press.

[Xiao et al., 2017] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.