Scientific Machine Learning 04
Cross-Entropy, Softmax, and Negative Log-Likelihood

Donghyun Ko

January 4, 2026

This post is a story about *why* the usual **'sigmoid + quadratic/MSE'** pairing can make learning crawl, and *how* two better pairings—**'sigmoid + cross-entropy'** and **'softmax + negative log-likelihood'**—tackle this problematic situation. We will start with intuition you can hold in your head, then walk all the way through the mathematical proofs so you can see why **'sigmoid + MSE'** is a trouble-maker and how the two smarter couples make the speed-up line by line.

Contents

1 A quick motivation: where the slowdown comes from

Imagine a single sigmoid neuron with scalar input x and label $y \in \{0, 1\}$. With weight w, bias b,

$$z = wx + b, \qquad a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Under the quadratic (MSE) cost function for a single example such that:

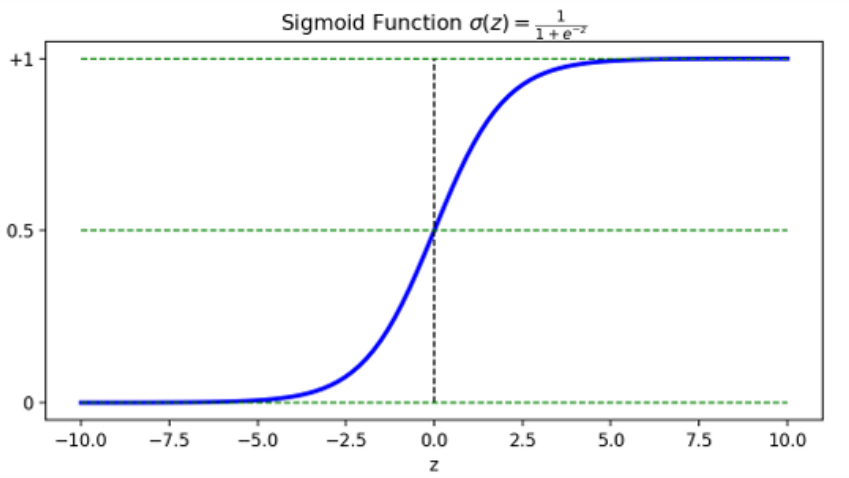
$$C_x = \frac{1}{2}(a - y)^2$$

The backpropagation chain rule gives the following:

$$\frac{\partial C_x}{\partial w} = \frac{\partial C_x}{\partial a}\,\frac{\partial a}{\partial z}\,\frac{\partial z}{\partial w} = (a - y)\,\sigma'(z)\,x, \qquad \frac{\partial C_x}{\partial b} = (a - y)\,\sigma'(z)$$

The derivation of the sigmoid activation function, $\sigma(z) = \frac{1}{1+e^{-z}}$, is:

$$\sigma'(z) = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1 + e^{-z} - 1}{(1 + e^{-z})^2} = \sigma(z)\Big(1 - \sigma(z)\Big)$$



Numerically, if we initialize around the center, say $w_{\text{init}} = b_{\text{init}} = 0$, then $z_{\text{init}} = 0$,

$$\sigma(z_{\text{init}}) = 0.5, \qquad \sigma'(z_{\text{init}}) = 0.25$$

But if we initialize in a high-activation regime, e.g. $w_{\text{init}} = b_{\text{init}} = 2$ so that $z_{\text{init}} = 4$, then

$$\sigma(z_{\text{init}}) \approx 0.982, \qquad \sigma'(z_{\text{init}}) \approx 0.0177,$$

which is already almost flat. Why does flatness matter? In back-propagation, the output-layer error is

$$\delta^L \;=\; \nabla_{a^L} C \;\odot\; \sigma'(z^L),$$

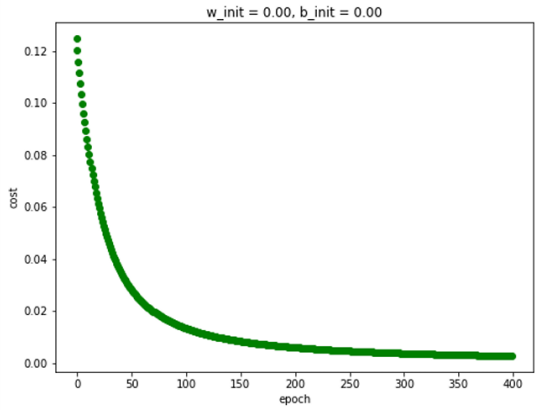
so whenever $\sigma'(z^L)$ is tiny (i.e., the neuron is saturated at a value near 0 or 1), the error signal δ^L becomes tiny, *even if* the prediction error $\nabla_{a^L} C$ is not small. That small δ^L then multiplies into all downstream gradients such that:

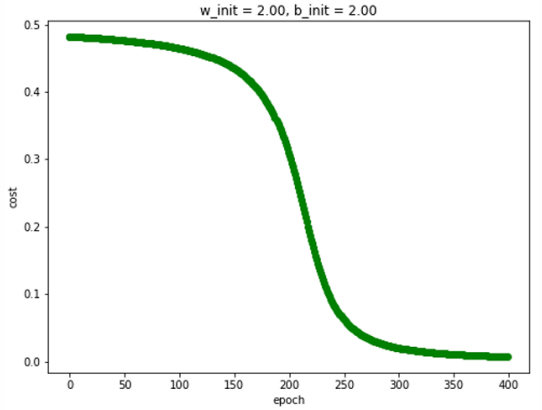
$$\frac{\partial C}{\partial w^\ell_{jk}} = a^{\ell-1}_k \, \delta^\ell_j, \qquad \frac{\partial C}{\partial b^\ell_j} = \delta^\ell_j,$$

This is the origin of learning slowdown by shrinking the parameter updates to nearly nothing:

$$w \leftarrow w' = w - \eta \frac{\partial C}{\partial w}, \qquad b \leftarrow b' = b - \eta \frac{\partial C}{\partial b}$$

In other words, this network is "trying" to learn, but each step is microscopic.

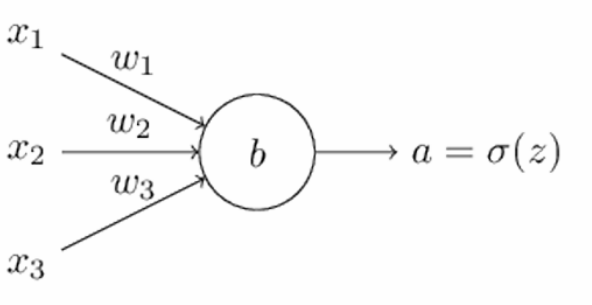




This slowdown is not an artifact of a toy with a single neuron. In deeper networks, the same mechanism repeats layer after layer, and the small factors can compound as the error signal is pushed backward—one face of the vanishing–gradient phenomenon. This is the practical motivation for pairing sigmoid outputs with a loss that *does not* multiply the error by $\sigma'(z)$ (cross-entropy), or for using softmax with negative log-likelihood at the multiclass output. The math for those pairings will show why the gradient remains healthy even when activations sit near 0 or 1.

2 Cross-entropy with a sigmoid: “faster learning"

Consider a neuron with several input variables $x_1, x_2, \ldots$, with corresponding weights $w_1, w_2, \ldots$, a bias b, and the activation function of the sigmoid. As before, the output of this neuron is $a = \sigma(z)$ where $z = \sum_j w_j x_j + b$ is the weighted sum of the inputs.



By replacing the quadratic cost(MSE) with the cross-entropy cost fucntion, learning slowdown can be avoided. By definition, cross-entropy cost function for this single neuron is defined as:

$$C_x \;=\; -\Big[\, y \ln a + (1-y)\ln(1-a) \,\Big], \quad C \;=\; \frac{1}{n}\sum_x C_x \;=\; -\frac{1}{n}\Big[\, y \ln a + (1-y)\ln(1-a) \,\Big]$$

where ’n’ is the number of training samples, ’x’ and ’y’ represent the training inputs and outputs, and the sum(C) is over all training inputs.

Is the cross-entropy cost a legitimate cost function?

Two requirements for a legitimate cost function.

1. It must be non-negative.
2. It should reduce to (or minimized) 0 when the network's prediction is close to target.

Check for cross-entropy (with a sigmoid output). Let $a = \sigma(z) \in (0, 1)$ and

$$C_x \;=\; -\Big[y \ln a + (1-y)\ln(1-a)\Big], \qquad C \;=\; \frac{1}{n}\sum_x C_x$$

- *Non-negativity.* Since $a \in (0, 1)$, both $\ln a$ and $\ln(1-a)$ are always negative. Hence, it's always true to say: $y \ln a + (1-y)\ln(1-a) < 0$ and the outer minus sign flips it to a nonnegative number: $C_x \geq 0$. The dataset cost $C = \frac{1}{n}\sum_x C_x$ is an average of those nonnegative terms, so $C \geq 0$ as well.
- *For classification problem, the true label 'y' takes the values of 0 or 1.*

$$\begin{aligned} y = 0 \text{ and } a \approx 0 &\implies C_x \approx -\ln(1-a) \approx 0, \\ y = 1 \text{ and } a \approx 1 &\implies C_x \approx -\ln(a) \approx 0 \end{aligned}$$

  Hence, when predictions are correct, the dataset cost $C = \frac{1}{n}\sum_x C_x$ is close to 0.

The minimum of the cross-entropy cost function

Fix a target $y \in [0, 1]$ and view the cross entropy per sample as a function of the predicted probability a:

$$C_x(a) \;=\; -\Big[y \ln a + (1-y)\ln(1-a)\Big], \qquad a = \sigma(z) \in (0, 1)$$

Differentiating gives:

$$\frac{\partial C_x}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a} = \frac{a-y}{a(1-a)}$$

The critical point solves $\partial C_x/\partial a = 0$, i.e. $a = y$. The second derivative gives:

$$\frac{\partial^2 C_x}{\partial a^2} = \frac{y}{a^2} + \frac{1-y}{(1-a)^2} = \frac{a^2 - 2ay + y}{a^2(1-a)^2} \geq \frac{a^2 - 2ay + y^2}{a^2(1-a)^2} = \frac{(a-y)^2}{a^2(1-a)^2} \;\geq\; 0$$

is nonnegative on $(0, 1)$ and vanishes only at $a = y$. Hence, C_x is *convex in* a and the stationary point is the *unique* global minimizer:

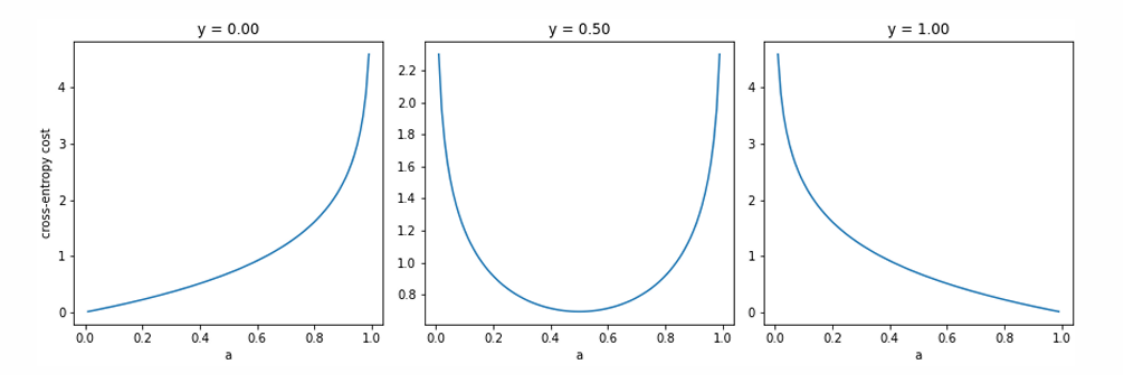
$$a^\star = y$$

Evaluating at the minimum,

$$C_x(a^\star) = -\Big[y \ln y + (1-y)\ln(1-y)\Big],$$

which is called the *binary entropy*. It is strictly positive for $y \in (0,1)$ and becomes 0 only at the classification corners, $y \in \{0,1\}$. In other words, with 0/1 classification labels, the loss can be driven nearly to zero, but when the target is a real value between 0 and 1, the loss never reaches zero—it has a positive lower bound. This indicates that the cross-entropy cost function can be close to 0 only for classification problems. However, it does not mean that we cannot use it for regression problems, just that we cannot expect the cost to be reduced to 0.

Boundary behavior (useful intuition). When $y = 1$, $C_x(a) = -\ln a$ decreases monotonically as $a\uparrow 1$ and blows up as $a\downarrow 0$; when $y = 0$, $C_x(a) = -\ln(1-a)$ decreases as $a\downarrow 0$ and blows up as $a\uparrow 1$. The loss, therefore, heavily penalizes *confident but wrong* predictions, and rewards *confident and correct* ones.



How does the cross-entropy cost avoid learning slowdown?

Recall that, for the sigmoid, we have:

$$z = \sum_j w_j x_j + b, \qquad a = \sigma(z) \in (0,1)$$

$$\sigma'(z) = \sigma(z)\Big(1 - \sigma(z)\Big) = a(1-a)$$

For the averaged cross-entropy

$$C = \frac{1}{n}\sum_x C_x, \text{ where } \quad C_x = -\Big[y \ln a + (1-y)\ln(1-a)\Big]$$

$$\frac{\partial C_x}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a} = \frac{a-y}{a(1-a)}$$

Taking the derivative of the cross-entropy(CE) cost function between the weights gives:

$$\begin{aligned}
\frac{\partial C}{\partial w_j} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial w_j} &= \frac{1}{n}\sum_x \frac{\partial C_x}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_j} \\
&= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right) \underbrace{\sigma'(z)}_{=\, a(1-a)} x_j \\
&= \frac{1}{n}\sum_x \frac{\sigma'(z)}{\sigma(z)\Big(1-\sigma(z)\Big)}\Big(\sigma(z) - y\Big)\, x_j \;=\; \frac{1}{n}\sum_x x_j \Big(\sigma(z) - y\Big)
\end{aligned}$$

This expression says that the rate at which we learn is now controlled only by $(\sigma(z) - y)$, i.e., by the error in the output, not coupled with $\sigma'(z)$. The larger the error, the faster the neuron will learn. By recalling that backpropagation in the case of 'MSE cost with sigmoid' gave us:

$$\frac{\partial C_x}{\partial w} = \frac{\partial C_x}{\partial a}\,\frac{\partial a}{\partial z}\,\frac{\partial z}{\partial w} = (\sigma(z) - y)\,\sigma'(z)\,x, \qquad \frac{\partial C_x}{\partial b} = (\sigma(z) - y)\,\sigma'(z),$$

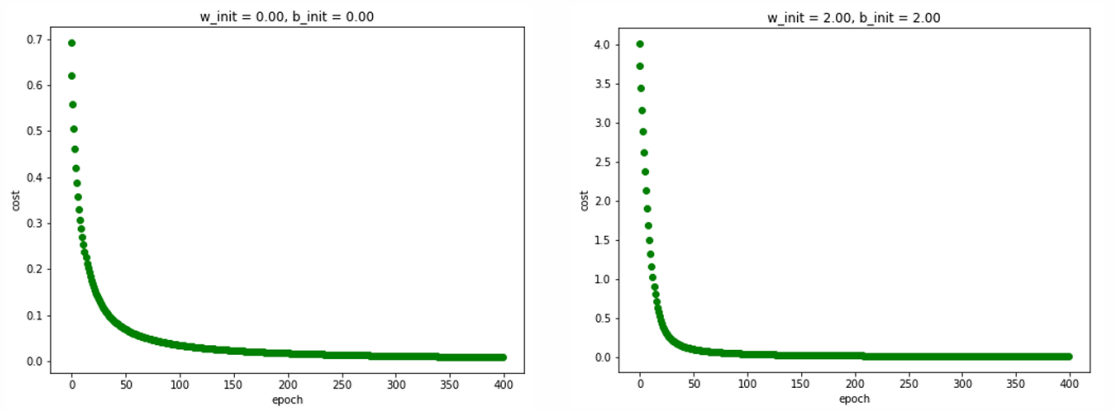
$$\text{where} \quad \sigma'(z) = \sigma(z)\Big(1 - \sigma(z)\Big) = a(1-a),$$

We can see that $\sigma'(z)$, which is the root-cause of learning slowdown in the 'MSE + sigmoid' pair, is now canceled out in the pair of 'CE + sigmoid'. The same cancellation removes $\sigma'(z)$ in the bias gradient of 'CE + sigmoid' as well:

$$\begin{aligned}
\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\,\frac{\partial a}{\partial b} &= \frac{1}{n}\sum_x \frac{\partial C_x}{\partial a}\,\frac{\partial a}{\partial z}\,\frac{\partial z}{\partial b} \\
&= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right) \underbrace{\sigma'(z)}_{=\,a(1-a)}\,\underbrace{\frac{\partial z}{\partial b}}_{=1} \\
&= \frac{1}{n}\sum_x \frac{\sigma'(z)}{\sigma(z)\Big(1-\sigma(z)\Big)}\Big(\sigma(z) - y\Big)\cdot 1 \\
&= \frac{1}{n}\sum_x \Big(\sigma(z) - y\Big)
\end{aligned}$$

Thus, as in the weight case, the $\sigma'(z) = a(1-a)$ term is canceled out, leaving the clean residual of $(a - y)$ only. This is precisely why 'sigmoid + CE' avoids the slowdown seen with 'sigmoid + quadratic/MSE', where $\sigma'(z)$ becomes tiny once the output saturates.

Revisit the toy example using the cross-entropy cost function. In the single–neuron toy example where a single neuron that takes input x = 1 and expects an output y = 0, these gradients drive the cross-entropy down rapidly whether we initialize near the origin ($w_{\text{init}} = b_{\text{init}} = 0$) or in a saturated regime ($w_{\text{init}} = b_{\text{init}} = 2$): both loss curves plunge in the first few dozen epochs and then gently taper, as shown in the figures.



Generalize the CE cost with sigmoid to many-neuron with multi-layer networks: BP1–BP4

Consider an L-layer feedforward network. For a given input x,

$$z^l = W^l a^{l-1} + b^l, \qquad a^l = \sigma(z^l) \text{ for all } l,$$

With $a^0 = x$ at the output layer $l = L$, each unit uses a sigmoid activation $a_j^L = \sigma(z_j^L) = \frac{1}{1+e^{-z_j^L}}$. For multiple layers and multiple output neurons:

$$C \;=\; \frac{1}{n}\sum_x \sum_{j=1}^{m}\Big[-y_j \ln a_j^L \;-\; (1-y_j)\ln\Big(1-a_j^L\Big)\Big],$$

where we are only adding $\sum_j$ for summing over all the neurons in the output layer, and $y_j \in \{0,1\}$ is the one-hot encoded target for output unit j.

Recall from BP1 to BP4

BP1. Error at the output layer

$$\delta_j^L \;=\; \frac{\partial C}{\partial a_j^L}\,\sigma'(z_j^L), \qquad \delta^L \;=\; \nabla_{a^L} C \;\odot\; \sigma'(z^L)$$

BP2. Error at a hidden layer

$$\delta_j^l \;=\; \sigma'(z_j^l)\sum_k w_{kj}^{l+1}\,\delta_k^{l+1}, \qquad \delta^l \;=\; \Big((W^{l+1})^\top \delta^{l+1}\Big) \;\odot\; \sigma'(z^l), \quad l = L-1,\ldots,1$$

BP3. Gradient with respect to biases

$$\frac{\partial C}{\partial b_j^l} \;=\; \frac{1}{n}\sum_x \delta_j^l, \qquad \frac{\partial C}{\partial B^l} \;=\; \frac{1}{n}\sum_x \delta^l$$

BP4. Gradient with respect to weights

$$\frac{\partial C}{\partial w_{jk}^l} \;=\; \frac{1}{n}\sum_x a_k^{l-1}\,\delta_j^l, \qquad \frac{\partial C}{\partial W^l} \;=\; \frac{1}{n}\sum_x \delta^l\,(a^{l-1})^\top$$

BP1. Fix a sample x and an output index j. For the BCE(binary CE) term of unit j,

$$C_{x,j} \;=\; -\,y_j \ln a_j^L \;-\; (1-y_j)\ln\Big(1-a_j^L\Big)$$

Then,

$$\frac{\partial C_x}{\partial a_j^L} = -\left(\frac{y_j}{a_j^L} - \frac{1-y_j}{1-a_j^L}\right)$$

Using $a_j^L = \sigma(z_j^L)$ and $\sigma'(z) = \sigma(z)\Big(1-\sigma(z)\Big)$,

$$\delta_j^L \;\equiv\; \frac{\partial C_x}{\partial z_j^L} = \frac{\partial C_x}{\partial a_j^L}\cdot\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C_x}{\partial a_j^L}\cdot\sigma'(z) = -\left(\frac{y_j}{a_j^L} - \frac{1-y_j}{1-a_j^L}\right) a_j^L(1-a_j^L) = a_j^L - y_j$$

Averaging over the dataset (multiplying by $\frac{1}{n}\sum_x$) preserves the same form:

$$\boxed{\delta_j^L = a_j^L - y_j}$$

BP2. For $l = L-1, L-2, \ldots, 1$, start from the definition:

$$\delta_j^l \;=\; \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\,\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1}\,\frac{\partial}{\partial z_j^l}\Big(\sum_r w_{kr}^{l+1} a_r^l + b_k^{l+1}\Big)$$

Only the term with $r = j$ in a_r^l depends on z_j^l, hence

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\,\frac{\partial a_j^l}{\partial z_j^l} = w_{kj}^{l+1}\,\sigma'(z_j^l)$$

Therefore,

$$\boxed{\delta_j^l \;=\; \sigma'(z_j^l)\sum_k w_{kj}^{l+1}\,\delta_k^{l+1}}$$

BP3. By definition,

$$\frac{\partial C}{\partial b_j^l} = \frac{1}{n}\sum_x \frac{\partial C_x}{\partial b_j^l} = \frac{1}{n}\sum_x \frac{\partial C_x}{\partial z_j^l}\,\frac{\partial z_j^l}{\partial b_j^l} = \frac{1}{n}\sum_x \delta_j^l\cdot 1 = \frac{1}{n}\sum_x \delta_j^l$$

At the output layer $l = L$ with sigmoids and BCE, BP1 gives $\delta_j^L = a_j^L - y_j$, hence

$$\boxed{\frac{\partial C}{\partial b_j^L} = \delta_j^L = \tfrac{1}{n}\sum_x (a_j^L - y_j)}$$

BP4. Similarly,

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{1}{n}\sum_x \frac{\partial C_x}{\partial w_{jk}^l} = \frac{1}{n}\sum_x \frac{\partial C_x}{\partial z_j^l}\,\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{1}{n}\sum_x \delta_j^l\cdot a_k^{l-1},$$

so

$$\frac{\partial C}{\partial w_{jk}^l} \;=\; \frac{1}{n}\sum_x a_k^{l-1}\,\delta_j^l$$

At the output layer $l = L$ with sigmoids and BCE, BP1 gives $\delta_j^L = a_j^L - y_j$, hence

$$\boxed{\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum_x a_k^{L-1}\,(a_j^L - y_j)}$$

When to use it? With sigmoid outputs for binary tasks, pairing with cross-entropy yields $\delta^L = a^L - y$. This avoids the vanishing factor $\sigma'(z)$ at the output and typically trains faster/more stably than using MSE with sigmoids.

3 Softmax Activation Function

For K classes, let the last layer produce logits with the weighted inputs defined as $z^L \in \mathbb{R}^K$:

$$a_j^L = \text{softmax}(z^L)_j = \frac{e^{z_j^L}}{\sum_{k=1}^K e^{z_k^L}}, \quad \text{where} \quad z_j^L \;=\; \sum_k w_{jk}^L \, a_k^{L-1} \;+\; b_j^L$$

Then, $a_j^L \in (0,1)$ and $\sum_j a_j^L = 1$, so the outputs form a discrete probability distribution over classes. Because a softmax layer outputs a probability distribution, we can interpret the output activation a_j^L as the network's estimate of the probability that the correct class is j.

Properties of the softmax activation function

The softmax is *non-local* since each a_j^L depends on all logits of z_k^L. By letting $S \overset{\text{def}}{=} \sum_{k=1}^K e^{z_k^L}$,

$$a_j^L = \frac{e^{z_j^L}}{\sum_{k=1}^K e^{z_k^L}} \;=\; \frac{e^{z_j^L}}{S} \quad (j = 1, \ldots, K)$$

We now want to seek $\dfrac{\partial a_j^L}{\partial z_k^L}$ for arbitrary 'j' and 'k'. To do this, we will use the quotient rule by setting $N(z) = e^{z_j^L}$ and $D(z) = S = \sum_k e^{z_k^L}$. Then,

$$\frac{\partial a_j^L}{\partial z_k^L} = \frac{\big(\partial N/\partial z_k^L\big) D \;-\; N\big(\partial D/\partial z_k^L\big)}{D^2} = \frac{\big(\partial/\partial z_k^L\big) e^{z_j^L} \cdot S \;-\; e^{z_j^L} \cdot \big(\partial/\partial z_k^L\big) S}{S^2} \tag{1}$$

Here, each derivative explicitly shows:

$$\frac{\partial}{\partial z_k^L} e^{z_j^L} = e^{z_j^L} \frac{\partial z_j^L}{\partial z_k^L} = e^{z_j^L} \delta_{jk} = \begin{cases} e^{z_j^L}, & k = j, \\ 0, & k \neq j, \end{cases} \quad \text{hence} \quad \frac{\partial S}{\partial z_k^L} = e^{z_k^L} \tag{2}$$

By substituting (2) into (1):

$$\frac{\partial a_j^L}{\partial z_k^L} = \frac{\delta_{jk} \, e^{z_j^L} \, S \;-\; e^{z_j^L} \, e^{z_k^L}}{S^2}$$

Since $a_j^L = e^{z_j^L}/S$ and $a_k^L = e^{z_k^L}/S$,

$$\frac{\partial a_j^L}{\partial z_k^L} = \frac{e^{z_j^L}}{S} \left(\delta_{jk} - \frac{e^{z_k^L}}{S} \right) = a_j^L \big(\delta_{jk} - a_k^L \big)$$

Equivalently, we can split this into two cases:

$$\boxed{\frac{\partial a_j^L}{\partial z_k^L} = \begin{cases} a_j^L \big(1 - a_j^L\big), & j = k \\ -\, a_j^L \, a_k^L, & j \neq k \end{cases}}$$

Therefore, increasing z_j^L is guaranteed to increase the corresponding output activation a_j^L and to decrease all other activations a_k^L for $k \neq j$. To show this, because $a_j^L \in (0,1)$,

$$\frac{\partial a_j^L}{\partial z_j^L} = a_j^L(1 - a_j^L) > 0 \quad \Rightarrow \quad a_j^L \text{ increases monotonically with } z_j^L$$

For $k \neq j$,

$$\frac{\partial a_j^L}{\partial z_k^L} = -\, a_j^L a_k^L < 0,$$

so increasing z_k^L will decrease a_j^L.

Where does the name "softmax" come from?

Consider the scaled family with a positive constant $c > 0$:

$$a_j^L(c) \;=\; \frac{e^{c\, z_j^L}}{\sum_k e^{c\, z_k^L}} \;=\; \frac{1}{\sum_k e^{\,c\,(z_k^L - z_j^L)}}$$

The standard softmax corresponds to $c = 1$. As we let $c \to +\infty$, two cases hold:

- **If z_j^L is not the largest value**, then there exists k with $z_k^L - z_j^L > 0$, so the denominator contains a term $e^{c\,(z_k^L - z_j^L)} \to +\infty$. Hence, $a_j^L(c) \to 0$.
- **If z_j^L is the largest value**, then for all k, $z_k^L - z_j^L \leq 0$. The $k = j$ term equals $e^0 = 1$ and every other term $e^{c\,(z_k^L - z_j^L)} \to 0$. Thus, $a_j^L(c) \to 1$.

As the scale 'c' grows, the exponential amplifies large logits much more than small ones, so almost all probability mass concentrates on the largest logit. The indicator $\mathbb{1}\{\text{statement}\}$ equals 1 when the statement is true and 0 otherwise. Thus,

$$\lim_{c\to\infty} a_j^L(c) = \mathbb{1}\Big\{\, j = \arg\max_k z_k^L \,\Big\} = \begin{cases} 1, & \text{if } j = \arg\max_k z_k^L \\ 0, & \text{otherwise} \end{cases}$$

In other words, when $c \to \infty$, softmax becomes a *hard max selector* (the winner gets probability 1, all others 0). At $c = 1$, softmax is a smooth, differentiable "softened" version of this max, which is suitable for gradient-based learning. This is where the name "softmax" comes from.

4 Negative log-likelihood (NLL) cost function

With a one-hot encoded label of $y \in \{1, \ldots, K\}$, define the *per-example* NLL cost

$$C_x \;=\; -\ln a_y^L,$$

where the last-layer logits are $\boldsymbol{z}^L = (z_1^L, \ldots, z_K^L)$ and the softmax outputs are

$$a_j^L \;=\; \text{softmax}(\boldsymbol{z}^L)_j \;=\; \frac{e^{z_j^L}}{\sum_{k=1}^K e^{z_k^L}}$$

The softmax output a_y^L is the model's predicted probability that the input belongs to the true class y. If the network is confident and correct ($a_y^L \approx 1$), then $-\ln a_y^L \approx 0$ (small cost). If it assigns a low probability to the true class ($a_y^L \ll 1$), the cost is large.

- **Confident & correct:** If $a_y^L \approx 1$, then the loss is

$$-\ln a_y^L \approx 0 \quad (\text{e.g., } a_y^L = 0.99 \Rightarrow -\ln 0.99 \approx 0.010),$$

  which is small because the model gave a high probability to the correct class.

- **Not confident & wrong:** If $a_y^L \ll 1$, the loss becomes large

$$a_y^L = 0.01 \Rightarrow -\ln 0.01 \approx 4.605,$$

  which penalize the model for assigning low probability to the truth.

This loss $-\ln a_y^L$ is exactly the *negative log-likelihood* (NLL) of the correct class. With a one-hot encoded label y and predicted distribution $\boldsymbol{a}^L$, it is equal to the *cross-entropy* between y and $\boldsymbol{a}^L$:

$$\text{NLL}(y, \boldsymbol{a}^L) = -\ln a_y^L \;=\; H(y, \boldsymbol{a}^L) \;=\; -\sum_{j=1}^{K} y_j \ln a_j^L$$

As an example, if we train on MNIST and the input image is "7", the NLL for this is:

$$C_x \;=\; -\ln a_7^L$$

- When the network is doing a good job (it is confident the image is a 7), the predicted probability a_7^L is close to 1, so $-\ln a_7^L$ is small.
- When the network is not doing a good job (it is unsure or favors other digits), a_7^L is smaller, so $-\ln a_7^L$ is larger.
- Thus, the NLL behaves exactly as a sensible cost should: it *rewards* high probability on the true class and *penalizes* low probability on the true class.

Key takeaway. For softmax output layers, pair them with the NLL cost. This combination yields clean output-layer gradients ($\nabla_{\boldsymbol{z}^L} C_x = \boldsymbol{a}^L - \boldsymbol{y}$), avoids the learning slowdown caused by saturated sigmoids, and directly encourages the model to place high probability on the correct class.

How does "Softmax + NLL" avoid the learning slowdown?

Let $\boldsymbol{a}^L = \text{softmax}(\boldsymbol{z}^L)$ and $C_x = -\ln a_y^L$. We now derive the output layer error.
Given $S = \sum_{k=1}^{K} e^{z_k^L}$. Then, $a_k^L = e^{z_k^L}/S$. For any pair (k, j),

$$\frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}\left(\frac{e^{z_k^L}}{S}\right) = \frac{\delta_{kj} e^{z_k^L}\, S - e^{z_k^L}\, \frac{\partial S}{\partial z_j^L}}{S^2} = \frac{\delta_{kj} e^{z_k^L}\, S - e^{z_k^L}\, e^{z_j^L}}{S^2}$$

$$= \frac{e^{z_k^L}}{S}\left(\delta_{kj} - \frac{e^{z_j^L}}{S}\right) = a_k^L\left(\delta_{kj} - a_j^L\right) \tag{3}$$

Hence, the (well-known) softmax Jacobian is

$$\boxed{\frac{\partial a_k^L}{\partial z_j^L} = a_k^L(\delta_{kj} - a_j^L)}$$

Since $C_x = -\ln a_y^L$, we have

$$\frac{\partial C_x}{\partial a_k^L} = -\frac{1}{a_y^L}\frac{\partial a_y^L}{\partial a_k^L} = -\frac{1}{a_y^L}\delta_{ky} \tag{4}$$

By combining (3) and (4),

$$\frac{\partial C_x}{\partial z_j^L} = \sum_{k=1}^{K}\frac{\partial C_x}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L} = \sum_{k=1}^{K}\left(-\frac{1}{a_y^L}\delta_{ky}\right)a_k^L(\delta_{kj} - a_j^L) \tag{A}$$

$$= -\frac{1}{a_y^L}\sum_{k=1}^{K}\delta_{ky}\,a_k^L(\delta_{kj} - a_j^L) \tag{B}$$

$$= -\frac{1}{a_y^L}\Big[\underbrace{\sum_{k=1}^{K}\delta_{ky}\,a_k^L\delta_{kj}}_{\text{term (i)}} - \underbrace{\sum_{k=1}^{K}\delta_{ky}\,a_k^L\,a_j^L}_{\text{term (ii)}}\Big] \tag{C}$$

Computing each term separately:

$$\sum_{k=1}^{K}\delta_{ky}\,\delta_{kj}\,a_k^L = a_y^L\sum_{k=1}^{K}\delta_{ky}\,\delta_{kj} = a_y^L\,\delta_{yj} \quad \text{(product of Kronecker deltas is 1 only when } k = y = j)$$

Since a_j^L does not depend on k, pull it outside the sum:

$$a_j^L\sum_{k=1}^{K}\delta_{ky}\,a_k^L = a_j^L\,a_y^L \quad \text{(the delta keeps only the } k = y \text{ component)}$$

Substituting these two into (C) gives

$$\frac{\partial C_x}{\partial z_j^L} = -\frac{1}{a_y^L}\Big[a_y^L\delta_{yj} - a_j^L a_y^L\Big] = -(\delta_{yj} - a_j^L) = a_j^L - \delta_{yj} \tag{D}$$

With a one-hot encoded label, $y_j = \delta_{yj}$, hence

$$\boxed{\frac{\partial C_x}{\partial z_j^L} = a_j^L - y_j}$$

The pre-activation at unit j is

$$z_j^L \;=\; \sum_{k=1}^{m} w_{jk}^L\,a_k^{L-1} \;+\; b_j^L, \qquad \text{so} \qquad \frac{\partial z_j^L}{\partial w_{jk}^L} = a_k^{L-1}, \quad \frac{\partial z_j^L}{\partial b_j^L} = 1$$

For a *single* training example x,

$$\frac{\partial C_x}{\partial w_{jk}^L} = \frac{\partial C_x}{\partial z_j^L}\,\frac{\partial z_j^L}{\partial w_{jk}^L} = (a_j^L - y_j)\,a_k^{L-1},$$

$$\frac{\partial C_x}{\partial b_j^L} = \frac{\partial C_x}{\partial z_j^L}\,\frac{\partial z_j^L}{\partial b_j^L} = (a_j^L - y_j)\cdot 1 = a_j^L - y_j$$

Averaging over a mini-batch of size n,

$$\boxed{\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum_x a_k^{L-1}\,(a_j^L - y_j), \qquad \frac{\partial C}{\partial b_j^L} = \frac{1}{n}\sum_x (a_j^L - y_j)}$$

Equivalently, in matrix form:

$$\nabla_{W^L} C \;=\; \frac{1}{n}\sum_x (\boldsymbol{a}^L - \boldsymbol{y})\,(\boldsymbol{a}^{L-1})^\top, \qquad \nabla_{\boldsymbol{b}^L} C \;=\; \frac{1}{n}\sum_x (\boldsymbol{a}^L - \boldsymbol{y})$$

Why this avoids learning slowdown. With MSE+*sigmoid*, the output gradient contains a factor $\sigma'(z_j^L)$, which becomes tiny when z_j^L saturates, leading to very small updates ("learning slowdown"). By contrast, with *softmax+NLL* the output gradient is *exactly* $a_j^L - y_j$, with no extra saturation term. For the true class $j = y$, $\partial C_x/\partial z_y^L = a_y^L - 1 \in (-1, 0]$; for $j \neq y$, $\partial C_x/\partial z_j^L = a_j^L \in [0, 1)$, so informative gradients remain even when logits are large in magnitude.

Guideline. Use *softmax + NLL* when you want to output probabilities over disjoint classes (standard multiclass). Use *sigmoid + cross-entropy* for binary or multi-label one-vs-rest. In both cases, the output layer gradient reduces to $(a - y)$, which minimizes the output layer saturation that causes learning slowdown.

This is a posting that I summarized with study-purpose and is adapted from lecture notes of NE-795(Scientific Machine Learning), given by professor Xu Wu, NC State University.