Simple Neural Network

Neural Network

Neural networks can be used for three main tasks: ***classification***, ***regression*** (prediction of a scalar), clustering (ordering unstructured data into groups), and more.

A neural network has a vast number of adjustable parameters, represented by weights and biases. The goal of a neuron network is to train it to fit the data instead of just memorizing it – this is called ***generalization***. Training is done by slowly adjusting weights and biases on many examples of data, and calculating the loss (error) of the output.

Overfitting occurs when the network only learns to fit the training data but is incapable of relating input-output dependencies, and thus the parameters are not tweaked correctly.

Layers

A neural network consists of a number of layers: the input layer, the output layer, and so-called hidden layers.

The ***input layer*** represents the input samples, like pixel values or sensor data. This input needs to be preprocessed so that it is normalized, scaled, and in numeric form.

The ***output layer*** represents the output value. With classification the output layer typically has as many neurons as the amount of classes, but it can also be binary classification (true or false).

Any layers between these two are called ***hidden layers***. A neural network is ***deep*** when there are two or more such layers. Hidden layers consist of neurons with a vast number of tweakable parameters: the weights and biases.

Dense layers are those where each neuron in the layer is connected to each neuron in the previous and next layer. This type of layer is also called ***linear*** or ***fully-connected***.

Parameters

When training, the weights of a layer are initialized randomly to a small non-zero value, and the biases set to zero. Otherwise, in a pre-trained model the parameters are initialized to the previously determined output of that model.

If a neuron no longer affects the model's result on any input and returns 0 on most inputs, then the neuron or even the network is essentially untrainable, or "dead". In this case, tweaking the bias to some non-zero value will lead to more favorable results.

Layer inputs use data batches because the output of a layer should be sample-related and not neuron-related. Samples are passed further through the network, and the next layer will also expect a batch of inputs.

- $\mathbf{X}$: matrix of input samples
- $\boldsymbol{y}$: target classification labels
- $\hat{\boldsymbol{y}}$: predicted classification (the values of the output layer)
- $\boldsymbol{Z}$: matrix of outputs of a hidden layer

Arrays

An n-dimensional array is homologous if all arrays along the 'row' dimension are of equal length.

The shape of an array is determined by the dimensions, so a 3D array with 3 matrices of 2 lists of 4 elements has a shape of (3, 2, 4). Shape is an important factor for mathematical operations on arrays.

- An array is an ordered homologous container for numbers. - A vector is a list or 1-dimensional array. - A matrix is a 2-dimensional array. Each element of the array can be accessed using a tuple of indices as a key. - A tensor is an n-dimensional object that can be represented as an array.

Forward pass

The goal of the forward pass is to run the input samples through the neural network and to calculate the loss, which is a representation of the error of the model.

A dense layer applies the following linear formula:

$$\mathbf{Z}_i = \mathbf{Z}_{i-1} \cdot \mathbf{W}_i + \boldsymbol{b}_i$$

where $\mathbf{Z}_i$ is the output matrix of a layer i, $\mathbf{W}$ is the weight matrix of this layer, and $\boldsymbol{b}$ is a vector of biases, one for each neuron in the layer.

Matrix $\mathbf{Z}_{i-1}$ can be either output from a previous hidden layer, or the input sample matrix $\mathbf{X}$ in case of the first layer.

Activation

After calculating the linear formula above, an activation function is applied to its output to simulate whether a neuron is activated for a certain input. Just like weights and biases, activation functions are essential parts of each neuron of a layer.

A neural network typically has two different activation functions; one used in the hidden layers, and a different one in the output layer.

If an activation function is nonlinear, it will allow neural networks with two or more hidden layers to map nonlinear functions. Some examples of nonlinear activation functions are:

- Sigmoid
- Rectified Linear Unit (ReLU)
- Softmax

Sigmoid

The Sigmoid or logistic function is one of the original granular activation functions used in neural networks:

$$\text{Sigmoid}(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

It is less common in modern networks, where it has been largely replaced by the Rectified Linear activation function.

Rectified Linear Unit (ReLU)

The rectified function is simple but powerful when using them in two (or more) successive layers because it can then approximate nonlinear functions. While simple, efficient, and nearly linear, it remains nonlinear due to the bend after 0.

$$\text{ReLU}(x) = \max(0, x)$$

Softmax

The Softmax activation function is used for multiclass classification problems and is applied to the output layer. It can take in non-normalized, or uncalibrated, inputs and produce a normalized distribution of probabilities for each class that will add up to 1. The predicted class is associated with the output neuron that returned the largest confidence score.

The function takes as input a vector $\boldsymbol{z} = (z_1, \ldots, z_K) \in \mathbb{R}$ of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers:

$$\text{Softmax}(\boldsymbol{z})_i = \frac{\exp(\boldsymbol{z}_i)}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_k)} \text{ for } i = 1, \ldots, K.$$

Exponentiation leads to non-negative values for each of the inputs. Due to the monotonic nature of the exponential function, higher input values also give higher outputs, so the predicted class will not change after applying exponentiation. Converting these numbers to a probability distribution that describe a vector of confidences is done via normalization in the denominator.

Softmax also includes a subtraction of the largest of the inputs before doing the exponentiation step. This mitigates two pervasive challenges with neural networks: dead neurons and exploding values, which render a network useless over time. The exponential function in the Softmax activation is actually one of the sources of exploding values. By subtracting the highest input value from all inputs, the maximum value will be 0, and thus the exponentiation will only produce outputs between 0 and 1.

Loss

A ***loss function*** or ***cost function*** quantifies the error of a model and the objective is to reduce it to 0 as closely as possible. By calculating loss, we strive to increase correct confidence and decrease misplaced confidence.

Neural networks that do regression have a loss function that calculate the mean squared error.

Categorical Cross-Entropy

For classification, the output of the network is a probability distribution, and a useful function when training a classification problem with classes is ***categorical cross-entropy***. It compares a ground-truth probability vector $\boldsymbol{y}$ and some predicted distribution $\hat{\boldsymbol{y}}$. It is one of the most commonly used loss functions with a softmax activation on the output layer.

The formula for calculating the categorical cross-entropy of $\boldsymbol{y}$ (actual/desired distribution) and $\hat{\boldsymbol{y}}$ (predicted distribution) is:

$$L_i = -\sum_j \boldsymbol{y}_{i,j} \log(\hat{\boldsymbol{y}})$$

Where L_i denotes sample loss value, i is the i-th sample in the set, j is the label/output index, $\boldsymbol{y}$ denotes the target values, and $\hat{\boldsymbol{y}}$ denotes the predicted values.

By encoding the target values $\boldsymbol{y}_{i,j}$ as one-hot encoded vectors, this equation can be simplified. One-hot means that one of the values in a vector is "hot" with a value of 1, and the rest are "cold" with values of 0. As a result, the summation over the targets $\hat{\boldsymbol{j}}$ zeroes out to just one multiplication by 1, simplifying the equation to:

$$L_i = -\log(\hat{\boldsymbol{y}}_{i,k})$$

Where L_i denotes sample loss value, i is the i-th sample in a set, k is the index of the target label (ground-true label), $\hat{\boldsymbol{y}}$ denotes the predicted values.

There is a problem with the categorical cross-entropy when a confidence value equals 0; this will lead to the invalid calculation of $\log(0)$, which is undefined. To prevent loss from being exactly 0 or negative, the value is clipped from both sides by the same infinitesimally small number.

Accuracy

While loss is a useful metric for optimizing a model, accuracy describes how often the largest confidence is the correct class in terms of a fraction. To compute the accuracy, the argmax values from the Softmax outputs $\hat{\boldsymbol{y}}$ are compared to the target classification $\boldsymbol{y}$:

$$\text{Accuracy}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{\|\hat{\boldsymbol{y}}\|} \sum_{i=1}^{\|\hat{\boldsymbol{y}}\|} (1 \text{ if } \text{argmax}(\hat{\boldsymbol{y}}_{\boldsymbol{i}}) = \boldsymbol{y}_i)$$

In other words, it is an arithmetic mean of all the predictions $\hat{\boldsymbol{y}}_i$ that correspond to the correct classification $\boldsymbol{y}_i$ (in which the result is 1).

Backpropagation

Calculating the partial derivatives of all the functions from the loss function back through all the layers in the neural network is called backpropagation. In this backwards pass, ***gradient descent*** is performed by calculating derivatives of single-parameter functions and gradients of multivariate functions. The end goal is to reduce the loss metric as much as possible by optimizing the model.

A partial derivative measures how much impact a single input has on a function's output. The gradient (denoted with a ∇ symbol) is a vector of the size of inputs containing all of the partial derivative solutions with respect to each of the inputs and shows the direction of the steepest ascent. By applying a negative fraction to a gradient the loss can be decreased.

In a dense layer, a single neuron in the current layer is connected to all neurons in the next layer. During backpropagation, each neuron from the current layer will receive a vector of partial derivatives for all the neurons in the next layer. Each neuron will output a gradient of the partial derivatives with respect to all of their inputs.

Rectified Linear Unit (ReLU) Derivative

The forward pass with the ReLU function for one neuron with inputs $\mathbf{X}$, weights $\mathbf{W}$, and biases $\boldsymbol{b}$ is:

$$
\begin{aligned}
&y = \text{ReLU}(\mathbf{X} \cdot \mathbf{W} + \boldsymbol{b}) \text{ or} \\
&y = \text{ReLU}(x_1 w_1 + x_2 w_2 + x_3 w_3 + \boldsymbol{b})
\end{aligned}
$$

This equation contains three nested functions: max, a summation and multiplications. The partial derivative to any input x_i (or any other of the inputs) is obtained by applying the ***chain rule*** to the nested max, sum, and mul functions:

$$
\begin{aligned}
\frac{\partial}{\partial x_i}[\text{ReLU}(\text{sum}(\text{mul}(x_1, w_1), \text{mul}(x_i, w_i), \text{mul}(x_n, w_n)))] = \\
\frac{d\text{ReLU}(\dots)}{d\text{sum}(\dots)} \cdot \frac{\partial \text{sum}(\dots)}{\partial \text{mul}(x_i, w_i)} \cdot \frac{\partial \text{mul}(x_i, w_i)}{\partial x_i}
\end{aligned}
$$

Firstly, the partial derivative of the sum operation is always 1, no matter the inputs:

$$
\begin{aligned}
f(x, y) = x + y \quad \rightarrow \quad & \frac{\partial}{\partial x} f(x, y) = 1 \\
& \frac{\partial}{\partial y} f(x, y) = 1
\end{aligned}
$$

Secondly, the derivative for a product (mul operation) is whatever the input is being multiplied by:

$$
\begin{aligned}
f(x, y) = x \cdot y \quad \rightarrow \quad & \frac{\partial}{\partial x} f(x, y) = y \\
& \frac{\partial}{\partial y} f(x, y) = x
\end{aligned}
$$

Thirdly, derivative of ReLU with respect to input x is 1 if $x > 0$, and 0 otherwise:

$$
\frac{\partial}{\partial \text{ReLU}(x)} = \frac{\partial}{\partial x} \max(0, x) = 1 \, (x > 0)
$$

And thus finally – following the chain rule – the full derivative of the ReLU function with respect to input x_i becomes:

$$\frac{\partial}{\partial x_i}\text{ReLU}(x_i) = \partial_v \cdot 1\,(z > 0) \cdot w_i, \text{ with } z = \boldsymbol{x} \cdot \boldsymbol{w} + b$$

Where ∂_v is the derivative calculated in the next layer, and z is the weighted sum of the inputs and the bias.

Combining all the partial derivatives above for each input $x_i \in X$ make up the gradients and can be represented as ∇_x for the inputs, ∇_w for the weights, and ∇_b for the bias:

$$\nabla_x = \begin{bmatrix} \frac{\partial}{\partial x_1}\text{ReLU}(x_1) \\ \frac{\partial}{\partial x_i}\text{ReLU}(x_i) \\ \frac{\partial}{\partial x_n}\text{ReLU}(x_n) \end{bmatrix}$$

$$\nabla_w = \begin{bmatrix} \frac{\partial}{\partial w_1}\text{ReLU}(w_1) \\ \frac{\partial}{\partial w_i}\text{ReLU}(w_i) \\ \frac{\partial}{\partial w_n}\text{ReLU}(w_n) \end{bmatrix}$$

$$\nabla_b = \begin{bmatrix} \frac{\partial}{\partial b}\text{ReLU}(b) \end{bmatrix}$$

Categorical Cross-Entropy Derivative

To calculate the gradient we need to use the partial derivative of the full forward function, which is:

$$L_i = -\sum_j \boldsymbol{y}_{i,j} \log(\hat{\boldsymbol{y}}_{i,j})$$

Where L_i denotes samples loss value, i is the i-th sample in a set, j is the label index, $\boldsymbol{y}$ are the target values, and $\hat{\boldsymbol{y}}$ are the predicted values.

Let's define the gradient equation as the partial derivative of the loss function with respect to each of its inputs and solve it using the chain rule:

$$\begin{aligned}
\frac{\partial L_i}{\partial \hat{\boldsymbol{y}}_{i,j}} &= \frac{\partial}{\partial \hat{\boldsymbol{y}}_{i,j}}[-\sum_j \boldsymbol{y}_{i,j} \log(\hat{\boldsymbol{y}}_{i,j})] \\
&= -\sum_j \boldsymbol{y}_{i,j} \cdot \frac{\partial}{\partial \hat{\boldsymbol{y}}_{i,j}} \log(\hat{\boldsymbol{y}}_{i,j}) \\
&= -\sum_j \boldsymbol{y}_{i,j} \cdot \frac{1}{\hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial}{\partial \hat{\boldsymbol{y}}_{i,j}} \hat{\boldsymbol{y}}_{i,j} \\
&= -\sum_j \boldsymbol{y}_{i,j} \cdot \frac{1}{\hat{\boldsymbol{y}}_{i,j}} \cdot 1 \\
&= -\sum_j \frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}}
\end{aligned}$$

The summation can be omitted because the partial derivative is calculated with respect to the $\boldsymbol{y}$, predicted values at the i-th sample and given index j. Thus the sum is performed over a single element at j. The derivative of this loss function with respect to its inputs then equals the negative ground-truth vector ($\boldsymbol{y}$), divided by the vector of the predicted values ($\hat{\boldsymbol{y}}$):

$$\frac{\partial L_i}{\partial \hat{\boldsymbol{y}}_{i,j}} = -\frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}}$$

Softmax Derivative

The definition of the partial derivative of the Softmax function is:

$$\mathbf{S}_{i,j} = \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \quad \rightarrow \quad \frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} = \frac{\partial \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}}{\partial \boldsymbol{z}_{i,m}}$$

Where $\mathbf{S}_{i,j}$ denotes j-th Softmax's output of the i-th sample, $\boldsymbol{z}$ is the input array which is a list of output vectors from the previous layer, $\boldsymbol{z}_{i,j}$ is the j-th Softmax's input of the i-th sample, K is the number of classifications, and $\boldsymbol{z}_{i,m}$ is the m-th Softmax's input of the i-th sample.

The Softmax function equals the exponentiated input divided by the sum of all exponentiated inputs, so we need to exponentiate all of the values first and then divide each of them by the sum of all of them to perform the normalization. Because each input of the Softmax impacts the each of the outputs, each partial derivative of each output with respect to each input has to be calculated.

Applying the chain rule and solving with respect to input $\boldsymbol{z}_{i,k}$ yields:

$$\begin{aligned}
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} &= \frac{\partial \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}}{\partial \boldsymbol{z}_{i,m}} \\
&= \frac{\frac{\partial}{\partial \boldsymbol{z}_{i,m}} \exp(\boldsymbol{z}_{i,j}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,j}) \cdot \frac{\partial}{\partial \boldsymbol{z}_{i,m}} \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2} \\
&= \frac{\frac{\partial}{\partial \boldsymbol{z}_{i,m}} \exp(\boldsymbol{z}_{i,j}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,j}) \cdot \frac{\partial}{\partial \boldsymbol{z}_{i,m}} \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2}
\end{aligned}$$

Now there are two possibilities for the partial derivative on the left side of the subtraction operation that lead to different results: one where $j = m$ and one wehere $j \neq m$.

In the case of $j = m$:

$$
\begin{aligned}
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} &= \frac{\exp(\boldsymbol{z}_{i,j}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,j}) \cdot \exp(\boldsymbol{z}_{i,m})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2} \\
&= \frac{\exp(\boldsymbol{z}_{i,j}) \cdot (\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,m}))}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \\
&= \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \cdot \frac{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,m})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \\
&= \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \cdot \frac{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} - \frac{\exp(\boldsymbol{z}_{i,m})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}
\end{aligned}
$$

Looking closely, both the left part and right part of the equation are just the definitions of the Softmax function itself:

$$
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} = \mathbf{S}_{i,j} \cdot (1 - \mathbf{S}_{i,m}) \quad \text{when } j = m
$$

In the case of $j \neq m$:

$$
\begin{aligned}
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} &= \frac{\frac{\partial}{\partial \boldsymbol{z}_{i,m}} \exp(\boldsymbol{z}_{i,j}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,j}) \cdot \frac{\partial}{\partial \boldsymbol{z}_{i,m}} \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2} \\
&= \frac{0 \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(z_{i,j}) \cdot \exp(\boldsymbol{z}_{i,m})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2} \\
&= \frac{-\exp(z_{i,j}) \cdot \exp(\boldsymbol{z}_{i,m})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \\
&= -\frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \cdot \frac{\exp(\boldsymbol{z}_{i,m})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}
\end{aligned}
$$

For this case the Softmax function can also be identified in both the left and right parts of the equation:

$$
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} = -\mathbf{S}_{i,j} \cdot \mathbf{S}_{i,m} \quad \text{when } j \neq m
$$

Thus, the two derivatives calculated above can be represented as:

$$
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} = \begin{cases} \mathbf{S}_{i,j} \cdot (1 - \mathbf{S}_{i,m}) & \text{when } \ j = m \\ \mathbf{S}_{i,j} \cdot (0 - \mathbf{S}_{i,m}) & \text{when } \ j \neq m \end{cases}
$$

By applying the ***Kronecker delta function*** the equation simplifies even further:

$$
\mathbf{S}_{i,j}\delta_{j,k} - \mathbf{S}_{i,j}\mathbf{S}_{i,k} \ \text{ where } \ \delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}
$$

The result of this is a Jacobian matrix; an array of partial derivatives in all of the of both input vectors. This is then done for every output of the Softmax function with respect to each input separately (a batch of samples), and thus the result becomes a list of Jacobian matrices, or a 3D matrix. This result then needs to be condensed to a single partial derivative value by taking each row of the Jacobian matrix and multiplying it with the corresponding value in the loss function's gradient.

Softmax Cross-Entropy

By combining the ***Softmax activation*** and the ***Categorical Cross-Entropy loss*** functions, their derivative can be greatly simplified, leading to a significant performance boost. The simplification is possible due to the the inputs of the cross-entropy function being equal to the outputs of the Softmax function, and using the properties of the one-hot encoded vector to reduce a summation to a scalar value.

The chain rule can be applied to calculate the partial derivative of the Categorical Cross-Entropy loss function with respect to the Softmax's inputs:

$$
\begin{aligned}
\frac{\partial L_i}{\partial \boldsymbol{z}_{i,m}} &= \frac{\partial L_i}{\partial \hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} \\
&= \frac{\partial L_i}{\partial \hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial \hat{\boldsymbol{y}}_{i,j}}{\partial \boldsymbol{z}_{i,m}} \\
&= -\sum_j \frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial \hat{\boldsymbol{y}}_{i,j}}{\partial \boldsymbol{z}_{i,m}} \\
&= -\frac{\boldsymbol{y}_{i,m}}{\hat{\boldsymbol{y}}_{i,m}} \cdot \frac{\partial \hat{\boldsymbol{y}}_{i,m}}{\partial \boldsymbol{z}_{i,m}} - \sum_{j \neq m} \frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial \hat{\boldsymbol{y}}_{i,j}}{\partial \boldsymbol{z}_{i,m}} \\
&= -\frac{\boldsymbol{y}_{i,m}}{\hat{\boldsymbol{y}}_{i,m}} \cdot \hat{\boldsymbol{y}}_{i,j} \cdot (1 - \hat{\boldsymbol{y}}_{i,m}) - \sum_{j \neq m} \frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}} (-\boldsymbol{y}_{i,j} \boldsymbol{y}_{i,m}) \\
&= -\boldsymbol{y}_{i,m} \cdot (1 - \hat{\boldsymbol{y}}_{i,m}) + \sum_{j \neq m} \boldsymbol{y}_{i,j} \hat{\boldsymbol{y}}_{i,m} \\
&= -\boldsymbol{y}_{i,m} + \boldsymbol{y}_{i,m} \hat{\boldsymbol{y}}_{i,m} + \sum_{j \neq m} \boldsymbol{y}_{i,j} \hat{\boldsymbol{y}}_{i,m} \\
&= -\boldsymbol{y}_{i,m} + \sum_j \boldsymbol{y}_{i,j} \hat{\boldsymbol{y}}_{i,m} \\
&= -\boldsymbol{y}_{i,m} + \hat{\boldsymbol{y}}_{i,m} \\
&= \hat{\boldsymbol{y}}_{i,m} - \boldsymbol{y}_{i,m}
\end{aligned}
$$

After applying the chain rule to the partial derivatives of cross-entropy loss function L_i and of the Softmax function $\mathbf{S}_{i,j}$, the whole equation simplifies significantly to the subtraction of the predicted labels $\hat{\boldsymbol{y}}$ and the ground-truth labels $\boldsymbol{y}$.