Artificial Neural Networks and Back-Propagation

Outline

Notation ................................................................. 2

Artificial Neural Networks ................................. 3
  The Artificial Neuron ...................................................... 3
  The Neural Network model ........................................ 5
  Backpropagation ........................................................... 8
  Summary of Backpropagation ...................................... 11
  Back-propagation Algorithm as Gradient Descent .......... 12

Class notes on the web :
http://www-prima.inrialpes.fr/Prima/Homepages/jlc/Courses/2016/ENSI2.SIRR/ENSI2.SIRR.html
Notation

\(x_d\)  
A feature. An observed or measured value.

\(\tilde{X}\)  
A vector of D features.

D  
The number of dimensions for the vector \(\tilde{X}\)

\(\{\tilde{X}_m\}\)  
Training samples for learning.

\(M\)  
The number of training samples.

\(\{y_m\}\)  
An indicator variable for each training sample, \(y_m \in \{0,1\}\)

\(\{\tilde{y}_m\}\)  
A vector of K indicator variables, one for each of K classes.

K  
The number of output classes.

L  
The number of layers (number of non-linear activation layers)

l  
The layer index. \(l\) ranges from 0 (input layer) to \(L\) (output layer)

\(N^{(l)}\)  
The number of units in layer \(l\). \(N^{(1)} = D\)

\(a_j^{(l)}\)  
The activation output of the \(j\)th neuron of the \(l\)th layer.

\(w_{ji}^{(l)}\)  
The weight from the unit \(i\) of layer \(l-1\) to the unit \(j\) of layer \(l\).

\(b_j^{(l)}\)  
The bias term for \(j\)th neuron of the \(l\)th layer.

\(f(z)\)  
A non-linear activation function, such as a sigmoid, tanh, or soft-max

Key Equations:

Feed Forward from Layer \(i\) to \(j\):  
\[ a_j^{(l)} = f\left( \sum_{i=1}^{N^{(l-1)}} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)} \right) \]

Feed Forward from Layer \(j\) to \(k\):  
\[ a_k^{(l+1)} = f\left( \sum_{j=1}^{N^{(l)}} w_{kj}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} \right) \]

Back Propagation from Layer \(j\) to \(i\):  
\[ \delta_{i,m}^{(l-1)} = \frac{\partial f(z_i^{(l-1)})}{\partial z_i^{(l-1)}} \sum_{j=1}^{N^{(l)}} w_{ji}^{(l)} \delta_{j,m}^{(l)} \]

Back Propagation from Layer \(k\) to \(j\):  
\[ \delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w_{kj}^{(l+1)} \delta_{k,m}^{(l+1)} \]

Weight and Bias Corrections for layer \(j\):  
\[ \Delta w_{ji}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)} \]
\[ \Delta b_j^{(l)} = \delta_{j,m}^{(l)} \]

Network Update Formulas:  
\[ w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \cdot \Delta w_{ji}^{(l)} \]
\[ b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_j^{(l)} \]
Artificial Neural Networks

The Artificial Neuron

The simplest possible neural network is composed of a single neuron.

A “neuron” is a computational unit that integrates information from a vector of $D$ features, $\tilde{X}$, to the likelihood of a hypothesis, $h_{w,b}(\tilde{X})$

$$a = h_{w,b}(\tilde{X})$$

The neuron is composed of a weighted sum of input values

$$z = w_1 x_1 + w_2 x_2 + ... + w_D x_D + b$$

followed by a non-linear “activation” function, $f(z)$ (sometimes written $\phi(z)$)

$$a = h_{w,b}(\tilde{X}) = f(\tilde{w}^T \tilde{X} + b)$$

Many different activation functions are used.

A popular choice for activation function is the sigmoid:

$$f(z) = \frac{1}{1 + e^{-z}}$$

This function is useful because the derivative is:

$$\frac{df(z)}{dz} = f(z)(1 - f(z))$$

This gives a decision function:

if $h_{w,b}(\tilde{X}) > 0.5$ POSITIVE else NEGATIVE

Other popular decision functions include the hyperbolic tangent and the softmax.
The hyperbolic Tangent: 

\[ f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]

The hyperbolic tangent is a rescaled form of sigmoid ranging over \([-1, 1]\)

We can also use the step function: 

\[ f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \]

Or the sgn function: 

\[ f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \]

Plot of Sigmoid (red), Hyperbolic Tangent (Blue) and Step Function (Green)

The softmax function is often used for multi-class networks. For K classes:

\[ f(z_k) = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} \]

The rectified linear function is popular for deep learning because of a trivial derivative:

Relu: 

\[ f(z) = \max(0, z) \]

While Relu is discontinuous at \( z=0 \), for \( z > 0 \): 

\[ \frac{df(z)}{dz} = 1 \]

Note that the choice of decision function will determine the target variable “y” for supervised learning.
**The Neural Network model**

A neural network is a multi-layer assembly of neurons of the form. For example, this is a 2-layer network:

![Neural Network Diagram](image)

The circles labeled +1 are the bias terms. The circles on the left are the input terms. Some authors, notably in the Stanford tutorials, refer to this as Level 1. We will NOT refer to this as a level (or, if necessary, level L=0). The rightmost circle is the output layer, also called L. The circles in the middle are referred to as a “hidden layer”. In this example there is a single hidden layer and the total number of layers is L=2.

The parameters carry a superscript, referring to their layer.

We will use the following notation:

- $L$ The number of layers (Layers of non-linear activations).
- $l$ The layer index. $l$ ranges from 0 (input layer) to $L$ (output layer).
- $N(l)$ The number of units in layer $l$. $N(0) = D$
- $a_j^{(l)}$ The activation output of the $j^{th}$ neuron of the $l^{th}$ layer.
- $w_{ji}^{(l)}$ The weight from the unit $i$ of layer $l-1$ for the unit $j$ of layer $l$.
- $b_j^{(l)}$ The bias term for $j^{th}$ unit of the $l^{th}$ layer.
- $f(z)$ A non-linear activation function, such as a sigmoid, tanh, or soft-max.

For example: $a_1^{(2)}$ is the activation output of the first neuron of the second layer. $W_{13}^{(2)}$ is the weight for input for of neuron 3 in the second level.

The above network would be described by:

- $a_1^{(1)} = f(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3 + b_1^{(1)})$
- $a_2^{(1)} = f(w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{23}^{(1)} x_3 + b_2^{(1)})$
- $a_3^{(1)} = f(w_{31}^{(1)} x_1 + w_{32}^{(1)} x_2 + w_{33}^{(1)} x_3 + b_3^{(1)})$
- $h_{w,b} (\hat{X}) = a_1^{(2)} = f(w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + w_{13}^{(2)} a_3^{(1)} + b_1^{(2)})$
This can be generalized to multiple layers. For example:

\[ h(X_m) \] is the vector of network outputs (one for each class).

Each unit is defined as follows:

\[ \begin{align*}
    a^{(l-1)}_i &= \sum_{j} w_{ji}^{(l)} \cdot a^{(l-1)}_j + b^{(l)}_j \\
    z^{(l)}_j &= f(z^{(l)}_j) \\
    a^{(l)}_j &= f(z^{(l)}_j)
\end{align*} \]

The notation for a multi-layer network is:
- \( \tilde{a}^{(0)} = \tilde{X} \) is the input layer. \( a^{(0)}_i = X_d \)
- \( l \) is the current layer under discussion.
- \( N^{(l)} \) is the number of activation units in layer \( l \). \( N^{(0)} = D \)
- \( i,j,k \) Unit indices for layers \( l-1, l \) and \( l+1: i \rightarrow j \rightarrow k \)
- \( w_{ji}^{(l)} \) is the weight for the unit \( i \) of layer \( l-1 \) feeding to unit \( j \) of layer \( l \).

(We use the subscript is \( j,i \) to respect matrix notation convention. )

\( a^{(l)}_j \) is the activation output of the \( j^{th} \) unit of the layer \( l \)
\( b^{(l)}_j \) the bias term feeding to unit \( j \) of layer \( l \).

\( z^{(l)}_j = \sum_{i=1}^{N^{(l+1)}} w_{ji}^{(l)} a^{(l-1)}_i + b^{(l)}_j \) is the weighted input to \( j^{th} \) unit of layer \( l \)
\( f(z) \) is a non-linear decision function, such as a sigmoid, tanh(), or soft-max
\( a^{(l)}_j = f(z^{(l)}_j) \) is the activation output for the \( j^{th} \) unit of layer \( l \)

Note that the Stanford tutorials call the input vector \( l=1 \), and label the weights as belonging to the earlier layer. There does not seem to be a single convention.
In deriving the back-propagation algorithm for learning, we will use

\[ z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)} \]
\[ \tilde{z}_{k}^{(l+1)} = \sum_{j=1}^{N^{(l)}} w_{kj}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} \]
\[ a_j^{(l)} = f\left( \sum_{i=1}^{N^{(l-1)}} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)} \right) \]
\[ a_{k}^{(l+1)} = f\left( \sum_{j=1}^{N^{(l)}} w_{kj}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} \right) \]

It can be more convenient to represent this using vectors:

\[ \tilde{z}^{(l)} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{N_l}^{(l)} \end{bmatrix} \quad a^{(l)} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{N_l}^{(l)} \end{bmatrix} \]

and to write the weights and bias at each level \( l \) as a \( k \) by \( j \) Matrix,

\[
W^{(l)} = \begin{bmatrix}
    w_{11}^{(l)} & \cdots & w_{1j}^{(l)} & \cdots & w_{1N^{(l-1)}} \\
    \vdots & \ddots & \vdots & \ddots & \vdots \\
    w_{j1}^{(l)} & \cdots & w_{jj}^{(l)} & \cdots & w_{jN^{(l-1)}} \\
    \vdots & \ddots & \vdots & \ddots & \vdots \\
    w_{N_l1}^{(l)} & \cdots & w_{N_lj}^{(l)} & \cdots & w_{N_lN^{(l-1)}}
\end{bmatrix}
\]
\[ \tilde{b}^{(l)} = \begin{bmatrix} b_1^{(l)} \\ \vdots \\ b_{j}^{(l)} \\ \vdots \\ b_{N^{(l-1)}}^{(l)} \end{bmatrix} \]

We can see that the weights are a 3\(^{rd}\) order Tensor (vector of matrices), and that the biases are a matrix (vector of vectors).

\[ \tilde{z}^{(l)} = W^{(l)} \tilde{a}^{(l-1)} + \tilde{b}^{(l)} \quad \text{and} \quad \tilde{a}^{(l)} = f(\tilde{z}^{(l)}) = f(W^{(l)} \tilde{a}^{(l-1)} + \tilde{b}^{(l)}) \]

We can assemble the set of matrices \( W^{(l)} \) into an 3rd order Tensor (Vector of matrices), \( W \), and represent \( \tilde{a}^{(l)} \), \( \tilde{z}^{(l)} \) and \( \tilde{b}^{(l)} \) as matrices (vectors of vectors): \( A \), \( Z \), \( B \).

So how to do we learn the weights \( W \) and biases \( B \)?

We could train a 2-class detector from a labeled training set \( \{X_m, y_m\} \) using gradient descent. For more than two layers, we will need to use the more general “back-propagation” algorithm.
Backpropagation

Backpropagation adjusts the network the weights $w_{ji}^{(l)}$ and biases $b_{j}^{(l)}$ so as to minimize an error function between the network output $\tilde{h}(\tilde{x}_m;W,B) = \tilde{a}^{(L)}$ and the target value $\tilde{y}_m$ for the M training samples $\{\tilde{x}_m\}, \{\tilde{y}_m\}$.

This is an iterative algorithm that propagates an error term back through the hidden layers and computes a correction for the weights at each layer so as to minimize the error term.

This raises two questions:
1) How do we initialize the weights?
2) How do we compute the error term for hidden layers?

1) How do we initialize the weights?

A natural answer for the first question is to initialize the weights to 0.

By experience this causes problems. If the parameters all start with identical values, then the algorithm can end up learning the same value for all parameters. To avoid this, we initialize the parameters with a small random variable that is near 0, for example computed with a normal density with variance $\epsilon$ (typically 0.01).

$$\forall w_{ji}^{(l)} = \mathcal{N}(0;\epsilon) \quad \text{and} \quad \forall b_{j}^{(l)} = \mathcal{N}(0;\epsilon)$$

where $\mathcal{N}$ is a sample from a normal density.

An even better solution is provided by Xavier GLORIOT’s technique (see course web site on Xavier normalization). However that solution is too complex for today’s lecture.

2) How do we compute the error term?

Backpropagation propagates the error term back through the layers, using the weights. We will present this for individual training samples. The algorithm can easily be generalized to learning from sets of training samples (Batch mode).

Given a training sample, $\tilde{x}_m$, we first propagate the $\tilde{x}_m$ through the $L$ layers of the network (Forward propagation) to obtain a hypothesis $\tilde{h}(\tilde{x}_m;W,B) = \tilde{a}^{(L)}$. 
We then compute an error term. In the case, of a multi-class network, this is a vector, with k components, one output for each hypothesis. In this case the indicator vector would be a vector, with one component for each possible class:

\[ \tilde{\delta}^{(L+1)}_m = (a^{(L+1)}_m - \tilde{y}_m) \quad \text{or for each class } k: \quad \delta^{(L+1)}_{k,m} = (a^{(L)}_{k,m} - y_{k,m}) \]

This error term tells how much the unit was responsible for differences between the activation of the network \( h(X_m; w_{kj}^{(l)}, b_{k}^{(l)}) \) and the target value \( \tilde{y}_m \).

To keep things simple, let us consider the case of a two class network, so that \( \tilde{\delta}^{(L+1)}_m \), \( h(X_m) \), \( a^{(L+1)}_m \), and \( y_m \) are scalars. The results are easily generalized to vectors for multi-class networks. At the output layer, the “error” for each training sample is:

\[ \delta^{(L+1)}_m = (a^{(L)}_m - y_m) \]

For the hidden units in layers \( l \leq L \) the error \( \delta^{(l)}_j \) is based on a weighted average of the error terms for \( \delta^{(l+1)}_k \).

We compute error terms, \( \delta^{(l)}_j \) for each unit \( j \) in layer \( l \) back to \( l = l-1 \) using the sum of errors times the corresponding weights times the derivative of the activation function.

\[ \delta^{(l)}_{j,m} = \frac{\partial f(z^{(l)}_j)}{\partial z^{(l)}_j} \sum_{k=1}^{N^{(l+1)}} w_{kj}^{(l+1)} \delta^{(l+1)}_{k,m} \]

For the sigmoid activation function. \( f(z) = \frac{1}{1+e^{-z}} \) the derivative is: \( df(z) = f(z)(1-f(z)) \)

For \( a^{(l)}_j = f(z^{(l)}_j) \) this gives:

\[ \delta^{(l)}_{j,m} = a^{(l)}_{j,m}(1-a^{(l)}_{j,m}) \cdot \sum_{k=1}^{N^{(l+1)}} w_{kj}^{(l+1)} \delta^{(l+1)}_{k,m} \]
This error term can then used to correct the weights and bias terms leading from layer $j$ to layer $i$.

\[
\Delta w_{ji,m}^{(l)} = a_i^{(l-1)} \delta_j^{(l)}_m \\
\Delta b_{j,m}^{(l)} = \delta_j^{(l)}_m
\]

Note that the corrections $\Delta w_{ji,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are NOT applied until after the error has propagated all the way back to layer $l=1$, and that when $l=1$, $a_i^{(0)} = x_i$.

For “batch learning”, the corrections terms, $\Delta w_{ji,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are averaged over $M$ samples of the training data and then only an average correction is applied to the weights.

\[
\Delta w_{ji}^{(l)} = \frac{1}{M} \sum_{m=1}^{M} \Delta w_{ji,m}^{(l)} \\
\Delta b_j^{(l)} = \frac{1}{M} \sum_{m=1}^{M} \Delta b_{j,m}^{(l)}
\]

then

\[
w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \cdot \Delta w_{ji}^{(l)} \\
b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_j^{(l)}
\]

where $\eta$ is the learning rate.

Back-propagation is equivalent to computing the gradient of the loss function for each layer of the network. A common problem with gradient descent is that the loss function can have local minimum. This problem can be minimized by regularization. A popular regularization technique for back propagation is to use “momentum”

\[
w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \cdot \Delta w_{ji}^{(l)} + \mu \cdot w_{ji}^{(l)} \\
b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_j^{(l)} + \mu \cdot b_j^{(l)}
\]

where the terms $\mu \cdot w_{ji}^{(l)}$ and $\mu \cdot b_j^{(l)}$ serves to stabilize the estimation.

The back-propagation algorithm may be continued until all training data has been used. For batch training, the algorithm may be repeated until all error terms, $\delta_j^{(l)}_m$, are a less than a threshold.
Summary of Backpropagation

The Backpropagation algorithm can be summarized as:

1) Initialize the network and a set of correction vectors:

\[
\forall_{i,j,l} w^{(l)}_{ji} = \mathcal{N}(0; \varepsilon)
\]
\[
\forall_{i,l} b^{(l)}_{j} = \mathcal{N}(0; \varepsilon)
\]
\[
\forall_{i,j,l} \Delta w^{(l)}_{ji} = 0
\]
\[
\forall_{i,l} \Delta b^{(l)}_{j} = 0
\]

where \(\mathcal{N}\) is a sample from a normal density, and \(\varepsilon\) is a small value.

2) For each training sample, \(\tilde{X}_m\), propagate \(\tilde{X}_m\) through the network (forward propagation) to obtain a hypothesis \(\hat{h}(\tilde{X}_m; W, B)\). Compute the error and propagate this back through the network:

a) Compute the error term:

\[
\delta^{(L+1)}_m = (h(\tilde{X}_m) - y_m) = (a^{(L)}_m - y_m)
\]

b) Propagate the error back from \(l=L\) to \(l=1\):

\[
\delta^{(l)}_{jm} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w^{(l+1)}_{kj} \delta^{(l+1)}_{km}
\]
\[
\delta^{(l-1)}_{jm} = \frac{\partial f(z_j^{(l-1)})}{\partial z_j^{(l-1)}} \sum_{j=1}^{N^{(l)}} w^{(l)}_{ji} \delta^{(l)}_{jm}
\]

c) Use the error to set a vector of correction weights.

\[
\Delta w^{(l)}_{ji,m} = a^{(l-1)}_i \delta^{(l)}_{jm} \quad \Delta b^{(l)}_{j,m} = \delta^{(l)}_{jm}
\]

3) For all layers, \(l=1\) to \(L\), update the weights and bias using a learning rate, \(\eta\)

\[
w^{(l)}_{ji} \leftarrow w^{(l)}_{ji} - \eta \cdot \Delta w^{(l)}_{ji,m} + \mu \cdot w^{(l)}_{ji}
\]
\[
b^{(l)}_{j} \leftarrow b^{(l)}_{j} - \eta \cdot \Delta b^{(l)}_{j,m} + \mu \cdot b^{(l)}_{j}
\]

Note that this last step can be done with an average correction matrix obtained from many training samples (Batch mode), providing a more efficient algorithm.
Back-propagation Algorithm as Gradient Descent

Write the weights and bias at each level $l$ as a matrix with $i$ columns and $j$ rows.

$$W_l = \begin{pmatrix} w_{11}^{(l)} & \cdots & w_{ij}^{(l)} & \cdots & w_{1N^{(l-1)}}^{(l)} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{N^{(l-1)}1}^{(l)} & \cdots & w_{N^{(l-1)}j}^{(l)} & \cdots & w_{N^{(l-1)}N^{(l-1)}}^{(l)} \end{pmatrix}$$

$$b_l = \begin{pmatrix} b_1^l \\ \vdots \\ b_{N^{(l-1)}}^l \end{pmatrix}$$

This assumes that the number of hidden units is constant, which is rarely the case.

We can see that the weights are a 3rd order Tensor (vector of matrices) and that the biases are a matrix (vector of vectors). We can write these as the tensor $W$ and the matrix $B$.

Each of the $M$ training samples $\{\tilde{x}_m\}, \{y_m\}$ contributes an average loss to a cost function that can be defined as the square of the error between the activation and the target.

$$L(W, B; \tilde{x}_m, y_m) = \frac{1}{2} (a_m^{(L)} - y_m)^2$$

The $\frac{1}{2}$ term will simplify the algebra when we compute a derivative.

For each training sample $\tilde{x}_m$, $y_m$, and for each unit $j$ in layer $l$ we use this loss function to update the weights and bias using partial derivatives.

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \frac{\partial}{\partial w_{ji}^{(l)}} L(W, B; \tilde{x}_m, y_m)$$

and

$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \frac{\partial}{\partial b_j^{(l)}} L(W, B; \tilde{x}_m, y_m)$$

Where $\eta$ is a learning rate. This is fine for the final level $L$. However, to propagate to earlier levels we need to apportion the error to different units. We need to determine what part of this loss is due to each coefficient $w_{ji}^{(l)}$. 
To keep the notation simple, let us define the Loss at from sample \( m \) as

\[
L_m = L(W, B; \widetilde{X}_m, y_m)
\]

The contribution of each coefficient \( w_{ji}^{(l)} \) to the loss is

\[
L_m \bigg| w_{ji}^{(l)}
\]

Using the chain rule, we can rewrite this as

\[
\frac{\partial L_m}{\partial w_{ji}^{(l)}} = \frac{\partial L_m}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}}
\]

The error term for unit \( j \) of level \( l \) is:

\[
\delta_{j,m}^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ji}^{(l)} a_i^{(l-1)}
\]

Thus \( \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}} = a_i^{(l-1)} \) and so \( \frac{\partial L_m}{\partial w_{ji}^{(l)}} = \delta_{j,m}^{(l)} a_i^{(l-1)} \)

To propagate from layer \( k \) down to layer \( j \) we note that the same formula applies at level \( l+1 \).

\[
\frac{\partial L_m}{\partial w_{kj}^{(l+1)}} = \delta_{k,m}^{(l+1)} a_j^{(l)}
\]

By the chain rule. \( \delta_{j,m}^{(l)} = \frac{\partial L_m}{\partial z_j^{(l)}} = \frac{\partial L_m}{\partial f(z_j^{(l)})} \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \)

and

\[
\frac{\partial L_m}{\partial f(z_j^{(l)})} = \sum_{k=1}^{N^{(l+1)}} w_{kj}^{(l+1)} \delta_{k,m}^{(l+1)}
\]

Giving:

\[
\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w_{kj}^{(l+1)} \delta_{k,m}^{(l+1)}
\]