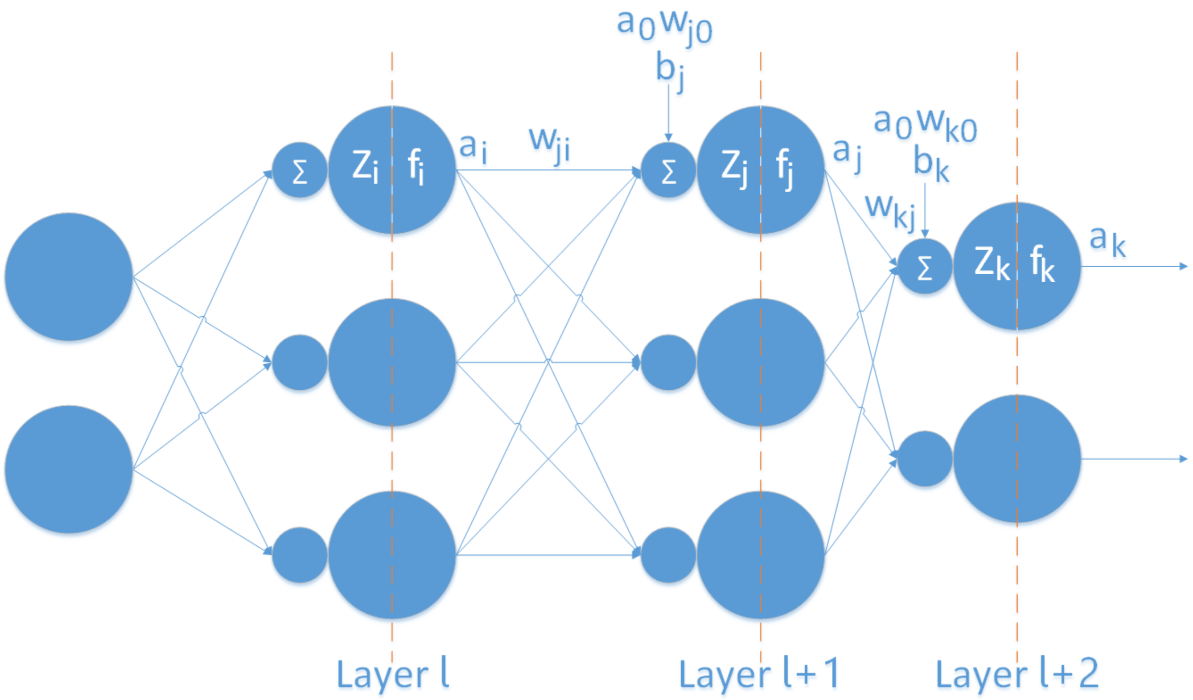
Backpropagation Algorithm



Quick notations:

1. K is the number of units (class) in the output layer of the neural network
2. n_l is the number of layers in the network
3. s_l is the number of unit in the in the l^{th} layer
4. $W_{ji}^{(l)}$ is the weight from the i^{th} unit in the l^{th} layer to the j^{th} unit in the $(l+1)^{th}$ layer
5. $z_i^{(l)}$ is the input to the i^{th} unit in the l^{th} layer
6. $a_i^{(l)}$ is the activation of the i^{th} unit in the l^{th} layer
7. $b_j^{(l)}$ is the bias to the j^{th} unit in the $(l+1)^{th}$ layer from the l^{th} layer, equal $a_0 * w_{jo}^{(l)} = w_{jo}^{(l)} = b_j^{(l)}$
8. $\delta_i^{(l)}$ is the "error term" of the i^{th} unit in the l^{th} layer, used in backpropagation
9. $A \bullet B$ is the element-wise product, which for $m \times n$ matrices *A* and *B* yields the $m \times n$ matrix $C = A \bullet B$ such that $C_{mn} = A_{mn} . B_{mn}$
10. $f^{(l)}$ is the activation function for units in the l^{th} layer

Suppose we have a fixed training set $\left\{\left(x^{(1)},y^{(1)}\right),\cdots,\left\{x^{(m)},y^{(m)}\right\}\right\}$ of m training examples. We can train our neural network using batch gradient descent. In detail, for a single training example $\left(x^{(i)},y^{(i)}\right)$, we define the cost function with respect to that single example to be:

$$J\left(W,b;x^{(i)},y^{(i)}\right)=\frac{1}{2}\left\|h_{W,b}\left(x^{(i)}\right)-y^{(i)}\right\|^{2}$$

This is a (one-half) squared-error cost function. Given a training set of m examples, we then define the overall cost function to be:

$$\begin{aligned}J(W,b)&=\left[\frac{1}{m}\sum_{i=1}^{m}J\left(W,b;x^{(i)},y^{(i)}\right)\right]+\frac{\lambda}{2m}\sum_{l=1}^{n_l-1}\sum_{j=1}^{s_{l+1}}\sum_{i=1}^{s_l}\left(W_{ji}^{(l)}\right)^2\\&=\left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}\left(x^{(i)}\right)-y^{(i)}\right\|^2\right)\right]+\frac{\lambda}{2m}\sum_{l=1}^{n_l-1}\sum_{j=1}^{s_{l+1}}\sum_{i=1}^{s_l}\left(W_{ji}^{(l)}\right)^2\\&=\left[\frac{1}{m}\sum_{i=1}^{m}\left(\sum_{k=1}^{K}\left(\frac{1}{2}\left(a_k^{(i)}-y_k^{(i)}\right)^2\right)\right)\right]+\frac{\lambda}{2m}\sum_{l=1}^{n_l-1}\sum_{j=1}^{s_{l+1}}\sum_{i=1}^{s_l}\left(W_{ji}^{(l)}\right)^2\end{aligned}$$

The first term in the definition of $J(W,b)$ is an average sum-of-squares error term. The second term is a regularization term (also called a **weight decay** term) that tends to decrease the magnitude of the weights, and helps prevent overfitting.

[**Note**: Usually weight decay is not applied to the bias terms $b_i^{(l)}$, as reflected in our definition for $J(W,b)$. Applying weight decay to the bias units usually makes only a small difference to the final network, however. If you've taken CS229 (Machine Learning) at Stanford or watched the course's videos on YouTube, you may also recognize this weight decay as essentially a variant of the Bayesian regularization method you saw there, where we placed a Gaussian prior on the parameters and did MAP (instead of maximum likelihood) estimation.]

The **weight decay parameter** λ controls the relative importance of the two terms. Note also the slightly overloaded notation: $J(W,b;x,y)$ is the squared error cost with respect to a single example; $J(W,b)$ is the overall cost function, which includes the weight decay term.

This cost function above is often used both for classification and for regression problems. For classification, we let $y=0\ or\ 1$ represent the two class labels (recall that the sigmoid activation function outputs values in $[0,1]$; if we were using a $tanh$ activation function, we would instead use -1 and $+1$ to denote the labels). For regression problems, we first scale our outputs to ensure that they lie in the $[0,1]$ range (or if we were using a $tanh$ activation function, then the $[-1,+1]$ range).

Our goal is to minimize $J(W,b)$ as a function of W and b. To train our neural network, we will initialize each parameter $W_{ji}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero (say according to a $Normal(0,\varepsilon^2)$ distribution for some small ε, say 0.01), and then apply an optimization algorithm such as batch gradient descent. Since $J(W,b)$ is a non-convex function, gradient descent is susceptible to local optima; however, in practice gradient descent usually works fairly well. Finally, note that it is important to initialize the parameters randomly, rather than to all 0's. If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input (more formally, $W_{ji}^{(1)}$ will be the same for all values of i, so that $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \cdots$ for any input x). The random initialization serves the purpose of **symmetry breaking**.

One iteration of gradient descent updates the parameters W, b as follows:

$$W_{ji}^{(l)} = W_{ji}^{(l)} - \alpha \frac{\partial}{\partial W_{ji}^{(l)}} J(W,b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b)$$

where α is the learning rate. The key step is computing the partial derivatives above. We will now describe the **backpropagation** algorithm, which gives an efficient way to compute these partial derivatives.

We will first describe how backpropagation can be used to compute $\frac{\partial}{\partial W_{ji}^{(l)}} J(W,b;x,y)$ and $\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y)$, the partial derivatives of the cost function $J(W,b;x,y)$ defined with respect to a single example (x,y). Once we can compute these, we see that the derivative of the overall cost function $J(W,b)$ can be computed as:

$$\frac{\partial}{\partial W_{ji}^{(l)}} J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial W_{ji}^{(l)}} J\left(W,b;x^{(i)},y^{(i)}\right)\right] + \frac{\lambda}{m} W_{ji}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J\left(W,b;x^{(i)},y^{(i)}\right)$$

The two lines above differ slightly because weight decay is applied to W but not b.

The intuition behind the backpropagation algorithm is as follows: Given a training example (x,y), we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_{W,b}(x)$. Then, for each node i in layer l, we would like to compute an "error term" $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in our output. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer). How about hidden units? For those, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

In detail, here is **the backpropagation algorithm** to compute $\frac{\partial}{\partial W_{ji}^{(l)}} J(W, b; x, y)$ and $\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y)$ of a single training example (x, y):

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer L_{n_l} (the output layer), set:
$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \left(\frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 \right) = -\left(y_i - a_i^{(n_l)} \right) . f'(z_i^{(n_l)})$$
3. For $l = n_l - 1,\ n_l - 2,\ n_l - 3, \cdots, 2$. For each node i in layer l, set
$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) . f'\left(z_i^{(l)} \right)\ obviously, i \neq 0$$
4. Compute the desired partial derivatives, which are given as:
$$\frac{\partial}{\partial W_{ji}^{(l)}} J(W, b; x, y) = a_i^{(l)} . \delta_j^{(l+1)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}$$
with $\forall i\ and\ a_{i=0}^{(l)} = 1$:
$$\frac{\partial}{\partial W_{ji}^{(l)}} J(W, b; x, y) = a_i^{(l)} * \delta_j^{(l+1)}$$

Finally, we can also re-write the algorithm using matrix-vectorial notation. We will use $\bullet$ to denote the element-wise product operator (denoted "**.***" in Matlab or Octave), so that $if\ a = b \bullet c, then\ a_i = b_i c_i$. Similar to how we extended the definition of $f(\cdot)$ to apply element-wise to vectors, we also do the same for $f'(\cdot)$ (so that $f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$).

The backpropagation algorithm in matrix-vectorial notation for one training example:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l} using the equations defining the forward propagation steps
2. For the output layer (layer L_{n_l}), set
$$\delta^{(n_l)} = -\left(y - a^{(n_l)} \right) \bullet f'\left(z^{(n_l)} \right)$$
3. For $l = n_l - 1,\ n_l - 2,\ n_l - 3, \cdots, 2$ set
$$\delta^{(l)} = \left((W^{(l)})^T \times \delta^{(l+1)} \right) \bullet f'(z^{(l)})$$
Here matrix $W^{(l)}$do not contain the first column, i.e. $W_{j0}^{(l)}$
4. Compute the desired partial derivatives:
$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \times \left(a^{(l)} \right)^T$$
$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}$$
with $\forall i\ and\ a_{i=0}^{(l)} = 1,\ a^{(l)} \ni a_{i=0}^{(l)}$:
$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \times \left(a^{(l)} \right)^T$$

The backpropagation algorithm in matrix-vectorial notation for a training set of m example: Matrix input X with the first column of all-one and each row corresponds to each input vector $x \ni x_0$

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l} using the equations defining the forward propagation steps
2. For the output layer (layer L_{n_l}), set
$$\delta^{(n_l)} = -\left(y - a^{(n_l)}\right) \bullet f'\left(z^{(n_l)}\right)$$
3. For $l = n_l - 1,\ n_l - 2,\ n_l - 3, \cdots, 2$ set
$$\delta^{(l)} = \left((W^{(l)})^T \times \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$
Here matrix $W^{(l)}$do not contain the first column, i.e. $W_{j0}^{(l)}$
4. Compute ***the sum of the desired partial derivatives matrix of all training example:***
for $l = n_l - 1,\ n_l - 2,\ n_l - 3, \cdots, 2$:
$$\nabla_{W^{(l)}} \sum J(W, b; x, y) = \delta^{(l+1)} \times \left(a^{(l)}\right)^T$$
$$\nabla_{b^{(l)}} \sum J(W, b; x, y) = \delta^{(l+1)}$$
$a^{(l)} \ni a_0^{(l)}$:
$$\nabla_{W^{(l)}} \sum J(W, b; x, y) = \delta^{(l+1)} \times \left(a^{(l)}\right)^T$$
$l = 1$ then:
$$\nabla_{W^{(l)}} \sum J(W, b; x, y) = \delta^{(l+1)} \times X$$
Note: $\nabla_{W^{(l)}} \sum J(W, b; x, y)$ is a matrix that each element is the sum of the partial derivatives at a specific weight of all training example. More specifically, each element is $\sum_{i=1}^{m} \frac{\partial}{\partial W_{ji}^{(l)}} J\left(W, b; x^{(i)}, y^{(i)}\right)$. Similar for $\nabla_{b^{(l)}} \sum J(W, b; x, y)$. This is different with the above vectorial form for 1 training example.

Implementation note: In steps 2 and 3 above, we need to compute$f'\left(z_i^{(l)}\right)$ for each value of i. Assuming $f(z)$ is the sigmoid activation function, we would already have $a_i^{(l)}$ stored away from the forward pass through the network. Thus, using the expression that we worked out earlier for $f'(z)$, we can compute this as

$$f'\left(z_i^{(l)}\right) = a_i^{(l)}(1 - a_i^{(l)})$$

Finally, we are ready to describe the full gradient descent algorithm. In the pseudo-code below, $\Delta W^{(l)}$ is a matrix (of the same dimension as $W^{(l)}$), and $\Delta b^{(l)}$ is a vector (of the same dimension as $b^{(l)}$). Note that in this notation, "$\Delta W^{(l)}$" is a matrix, and in particular it isn't "Δ times $W^{(l)}$."

We implement **one iteration of batch gradient descent** as follows:

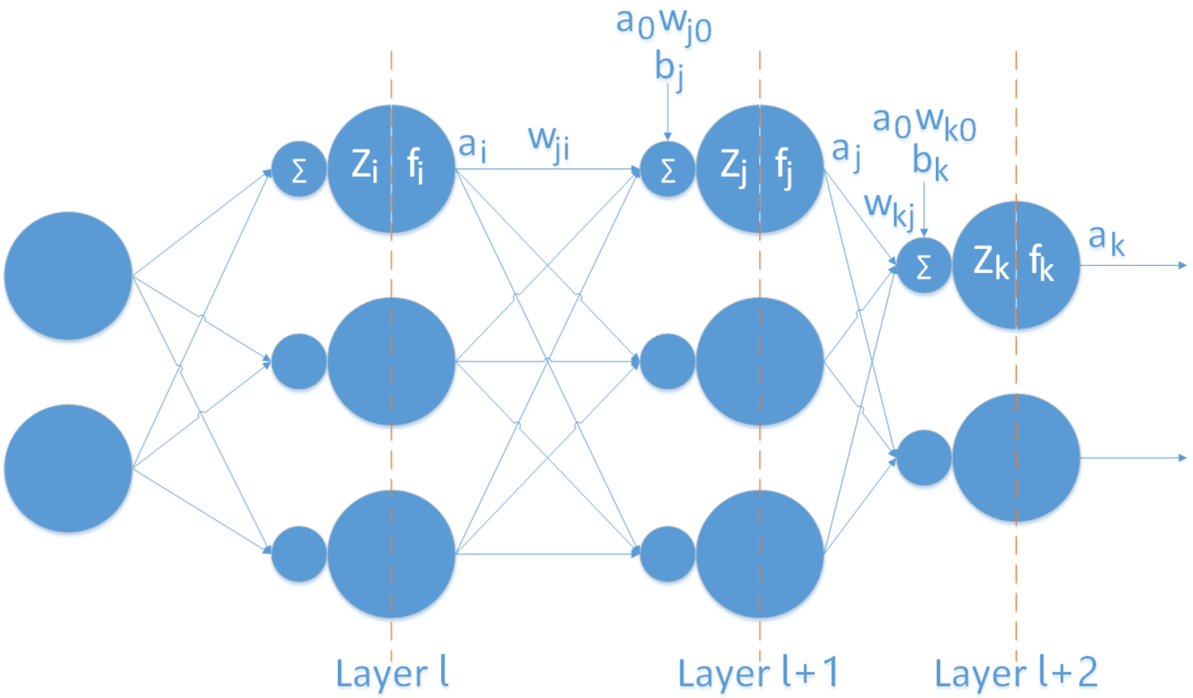
1. Set $\Delta W^{(l)} = 0$, $\Delta b^{(l)} = 0$ (matrix/vector of zeros) for all l.

2. For $i = 1$ to m,
   a. Use backpropagation to compute $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$
   b. Set $\Delta W^{(l)} = \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$
   c. Set $\Delta b^{(l)} = \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$
3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \frac{\lambda}{m} W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta b^{(l)} \right) \right]$$

To train our neural network, we can now repeatedly take steps of gradient descent to reduce our cost function $J(W, b)$.



Prove formula in backpropagation algorithm:

we have:

$$J(W, b; x, y) = \sum_{k=1}^{K} \left(\frac{1}{2} (a_k - y_k)^2 \right)$$

then,

$$\frac{\partial}{\partial W_{kj}^{(l)}} J(W, b; x, y) = \frac{\partial}{\partial W_{kj}^{(l)}} \left(\sum_{k=1}^{K} \left(\frac{1}{2} \left(a_k^{(l+1)} - y_k^{(l+1)} \right)^2 \right) \right)$$

Note that we add the superscript to denote the layer l and $l + 1$

Since we calculate the derivative for specific k, then:

$$\frac{\partial}{\partial W_{kj}^{(l)}} J(W,b;x,y) = \frac{\partial}{\partial W_{kj}^{(l)}} \left(\frac{1}{2} \left(a_k^{(l+1)} - y_k^{(l+1)} \right)^2 \right) = \left(a_k^{(l+1)} - y_k^{(l+1)} \right) \frac{\partial}{\partial W_{kj}^{(l)}} a_k^{(l+1)}$$

with:

$$a_k^{(l+1)} = f\left(z_k^{(l+1)} \right) = f\left(\sum_{j=0}^{s_l} a_j^{(l)} W_{kj}^{(l)} \right) = f\left(W_{k0}^{(l)} + \sum_{j=1}^{s_l} a_j^{(l)} W_{kj}^{(l)} \right)$$

then if $j \neq 0$ then

$$\frac{\partial}{\partial W_{kj}^{(l)}} J(W,b;x,y) = \left(a_k^{(l+1)} - y_k^{(l+1)} \right) \frac{\partial f\left(z_k^{(l+1)} \right)}{\partial z_k^{(l+1)}} a_j^{(l)} = \delta_k^{(l+1)} a_j^{(l)}$$

if $j = 0$ then

$$\frac{\partial}{\partial W_{k0}^{(l)}} J(W,b;x,y) = \left(a_k^{(l+1)} - y_k^{(l+1)} \right) \frac{\partial f\left(z_k^{(l+1)} \right)}{\partial z_k^{(l+1)}} = \delta_k^{(l+1)}$$

Overall, with $\forall j \; and \; a_{j=0}^{(l)} = 1$, $(l+1)$ is the output layer, then

$$\frac{\partial}{\partial W_{kj}^{(l)}} J(W,b;x,y) = \delta_k^{(l+1)} a_j^{(l)}$$

$$\delta_k^{(l+1)} = \left(a_k^{(l+1)} - y_k^{(l+1)} \right) \frac{\partial f\left(z_k^{(l+1)} \right)}{\partial z_k^{(l+1)}}$$

Now for the hidden layer nearest to the output layer:

$$\begin{aligned} \frac{\partial}{\partial W_{ji}^{(l-1)}} J(W,b;x,y) &= \frac{\partial}{\partial W_{ji}^{(l-1)}} \left(\sum_{k=1}^{K} \left(\frac{1}{2} \left(a_k^{(l+1)} - y_k^{(l+1)} \right)^2 \right) \right) \\ &= \sum_{k=1}^{K} \left(\left(a_k^{(l+1)} - y_k^{(l+1)} \right) * \frac{\partial}{\partial W_{ji}^{(l-1)}} a_k^{(l+1)} \right) \\ &= \sum_{k=1}^{K} \left(\left(a_k^{(l+1)} - y_k^{(l+1)} \right) * \frac{\partial f\left(z_k^{(l+1)} \right)}{\partial z_k^{(l+1)}} * \frac{\partial}{\partial W_{ji}^{(l-1)}} z_k^{(l+1)} \right) \end{aligned}$$

with:

$$z_k^{(l+1)} = W_{k0}^{(l)} + \sum_{j=1}^{s_l} \left(a_j^{(l)} W_{kj}^{(l)} \right) = W_{k0}^{(l)} + \sum_{j=1}^{s_l} \left(f\left(z_j^{(l)} \right) * W_{kj}^{(l)} \right)$$

$$= W_{k0}^{(l)} + \sum_{j=1}^{s_l} \left(f\left(W_{j0}^{(l-1)} + \sum_{i=1}^{s_{l-1}} a_i^{(l-1)} W_{ji}^{(l-1)} \right) W_{kj}^{(l)} \right)$$

then if $i \neq 0$ then

$$\frac{\partial}{\partial W_{ji}^{(l-1)}} z_k^{(l+1)} = \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} * \frac{\partial a_j^{(l)}}{\partial W_{ji}^{(l-1)}} = \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} * \frac{\partial f\left(z_j^{(l)} \right)}{\partial z_j^{(l)}} * \frac{\partial z_j^{(l)}}{\partial W_{ji}^{(l-1)}}$$

$$= W_{kj}^{(l)} * \frac{\partial f\left(z_j^{(l)} \right)}{\partial z_j^{(l)}} * a_i^{(l-1)}$$

$$\frac{\partial}{\partial W_{ji}^{(l-1)}} J(W, b; x, y) = \sum_{k=1}^{K} \left(\left(a_k^{(l+1)} - y_k^{(l+1)} \right) * \frac{\partial f\left(z_k^{(l+1)} \right)}{\partial z_k^{(l+1)}} * \frac{\partial}{\partial W_{ji}^{(l-1)}} z_k^{(l+1)} \right)$$

$$= \sum_{k=1}^{K} \left(\left(a_k^{(l+1)} - y_k^{(l+1)} \right) * \frac{\partial f\left(z_k^{(l+1)} \right)}{\partial z_k^{(l+1)}} * W_{kj}^{(l)} * \frac{\partial f\left(z_j^{(l)} \right)}{\partial z_j^{(l)}} * a_i^{(l-1)} \right)$$

$$= \frac{\partial f\left(z_j^{(l)} \right)}{\partial z_j^{(l)}} * a_i^{(l-1)} * \sum_{k=1}^{K} \left(\left(a_k^{(l+1)} - y_k^{(l+1)} \right) * \frac{\partial f\left(z_k^{(l+1)} \right)}{\partial z_k^{(l+1)}} * W_{kj}^{(l)} \right)$$

$$= \frac{\partial f\left(z_j^{(l)} \right)}{\partial z_j^{(l)}} * a_i^{(l-1)} * \sum_{k=1}^{K} \left(\delta_k^{(l+1)} * W_{kj}^{(l)} \right) = a_i^{(l-1)} * \delta_j^{(l)}$$

if $i = 0$ then

$$\frac{\partial}{\partial W_{j0}^{(l-1)}} z_k^{(l+1)} = W_{kj}^{(l)} * \frac{\partial f\left(z_j^{(l)} \right)}{\partial z_j^{(l)}}$$

$$\frac{\partial}{\partial W_{j0}^{(l-1)}} J(W,b;x,y) = \sum_{k=1}^{K}\left(\left(a_k^{(l+1)} - y_k^{(l+1)}\right) * \frac{\partial f\left(z_k^{(l+1)}\right)}{\partial z_k^{(l+1)}} * \frac{\partial}{\partial W_{j0}^{(l-1)}} z_k^{(l+1)}\right)$$

$$= \sum_{k=1}^{K}\left(\left(a_k^{(l+1)} - y_k^{(l+1)}\right) * \frac{\partial f\left(z_k^{(l+1)}\right)}{\partial z_k^{(l+1)}} * W_{kj}^{(l)} * \frac{\partial f\left(z_j^{(l)}\right)}{\partial z_j^{(l)}}\right)$$

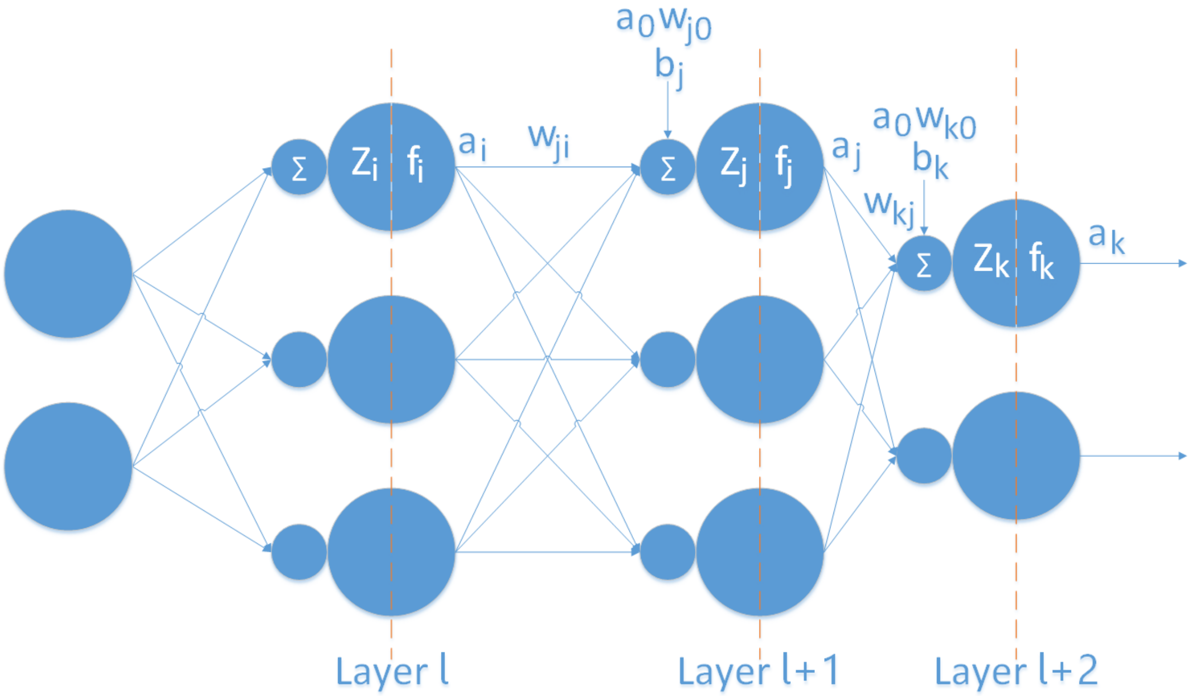
$$= \frac{\partial f\left(z_j^{(l)}\right)}{\partial z_j^{(l)}} * \sum_{k=1}^{K}\left(\left(a_k^{(l+1)} - y_k^{(l+1)}\right) * \frac{\partial f\left(z_k^{(l+1)}\right)}{\partial z_k^{(l+1)}} * W_{kj}^{(l)}\right)$$

$$= \frac{\partial f\left(z_j^{(l)}\right)}{\partial z_j^{(l)}} * \sum_{k=1}^{K}\left(\delta_k^{(l+1)} * W_{kj}^{(l)}\right) = \delta_j^{(l)}$$

Overall, with $\forall i \text{ and } a_{i=0}^{(l-1)} = 1$, then

$$\frac{\partial}{\partial W_{ji}^{(l-1)}} J(W,b;x,y) = a_i^{(l-1)} * \delta_j^{(l)}$$

In summary, we have for the following neural network:



For the output layer: with $\forall j \text{ and } a_{j=0}^{(l)} = 1$, $(l+1)$ is the output layer, then

$$\frac{\partial}{\partial W_{kj}^{(l)}} J(W,b;x,y) = \delta_k^{(l+1)} a_j^{(l)}$$

$$\delta_k^{(l+1)} = \left(a_k^{(l+1)} - y_k^{(l+1)}\right)\frac{\partial f\left(z_k^{(l+1)}\right)}{\partial z_k^{(l+1)}}$$

For the hidden layer nearest the output layer: Overall, with $\forall i \text{ and } a_{i=0}^{(l-1)} = 1$, then

$$\frac{\partial}{\partial W_{ji}^{(l-1)}} J(W, b; x, y) = a_i^{(l-1)} * \delta_j^{(l)}$$

Then similarly, for arbitrary hidden layer, with $\forall i \text{ and } a_{i=0}^{(l)} = 1$:

$$\delta_i^{(l)} = \frac{\partial f\left(z_i^{(l)}\right)}{\partial z_i^{(l)}} * \sum_{j=1}^{s_{l+1}} \left(\delta_j^{(l+1)} * W_{ji}^{(l)}\right)$$

$$\frac{\partial}{\partial W_{ji}^{(l)}} J(W, b; x, y) = a_i^{(l)} * \delta_j^{(l+1)}$$

References:

https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/

http://deeplearning.stanford.edu/wiki/index.php/Backpropagation_Algorithm