Multi-Layer Neural Networks

Steve Renals

23 February 2015

This note gives more details on training multi-layer networks.

1 Neural network architecture

Consider the simplest multi-layer network, with one hidden layer. The first layer involves M linear combinations of the d-dimension inputs:

$$b_j = \sum_{i=0}^{d} w_{ji}^{(1)} x_i \qquad j = 1, 2, \ldots, M.$$

As before $x_0 = 1$, with the weights leading out from it corresponding to the biases. The quantities b_j are called *activations*, and the parameters $w_{ji}^{(1)}$ are the weights. The superscript '(1)' indicates that this is the first layer of the network. Each of the activations is then transformed by a nonlinear activation function g, typically a sigmoid:

$$z_j = h(b_j) = \frac{1}{1 + \exp(-b_j)} \tag{1}$$

The quantities z_j are interpreted as the output of *hidden units* – so called because they do not have values specified by the problem (as is the case for input units) or target values used in training (as is the case for output units).

In the second layer, the outputs of the hidden units are linearly combined to give the activations of the K output units:

$$a_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \qquad k = 1, 2, \ldots, K. \tag{2}$$

Again $z_0 = 1$, corresponding to the bias. This transformation is the second layer of the neural network parameterized by weights $w_{kj}^{(2)}$. The output units are transformed using an activation function; again a sigmoid may be used:

$$y_k = g(a_k) = \frac{1}{1 + \exp(-a_k)}, \tag{3}$$

or for multi-class classification problems, a softmax activation function:

$$g(a_k) = \frac{\exp(a_k)}{\sum_{\ell=1}^{K} \exp(a_\ell)}$$

These equations may be combined to give the overall equation that describes the *forward propagation* through the network, and describes how an output vector is computed from an input vector, given the

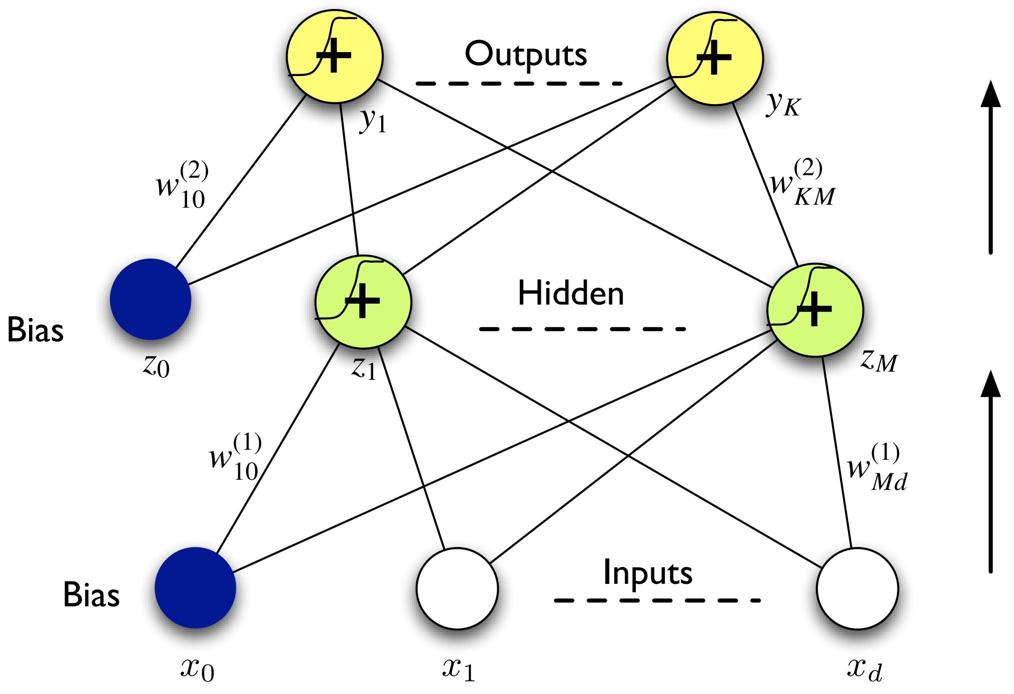

Figure 1: Network diagram for a multi-layer perceptron (MLP) with two layers of weights

weight matrices:

$$y_k = g\left(\sum_{j=0}^{M} w^{(2)}_{kj} h\left(\sum_{i=0}^{d} w^{(1)}_{ji} x_i\right)\right) \tag{4}$$

This is illustrated in figure 1.

2 Activation functions

2.1 Sigmoid

If we have a two class problem, with classes C_1 and C_2, then we can express the posterior probability of C_1 using Bayes' theorem:

$$P(C_1|\mathbf{x}) = \frac{p(\mathbf{x}|C_1)P(C_1)}{p(\mathbf{x}|C_1)P(C_1) + p(\mathbf{x}|C_2)P(C_2)}$$

If we divide top and bottom of the right hand side by $p(\mathbf{x}|C_1)P(C_1)$, then we obtain:

$$P(C_1|\mathbf{x}) = \frac{1}{1 + \frac{p(\mathbf{x}|C_2)P(C_2)}{p(\mathbf{x}|C_1)P(C_1)}} \tag{5}$$

If we define a as the ratio of log posterior probabilities (log odds):

$$a = \ln \frac{p(\mathbf{x}|C_1)P(C_1)}{p(\mathbf{x}|C_2)P(C_2)} \tag{6}$$

and substitute into (5) we obtain:

$$P(C_1|\mathbf{x}) = f(a) = \frac{1}{1 + \exp(-a)} \tag{7}$$

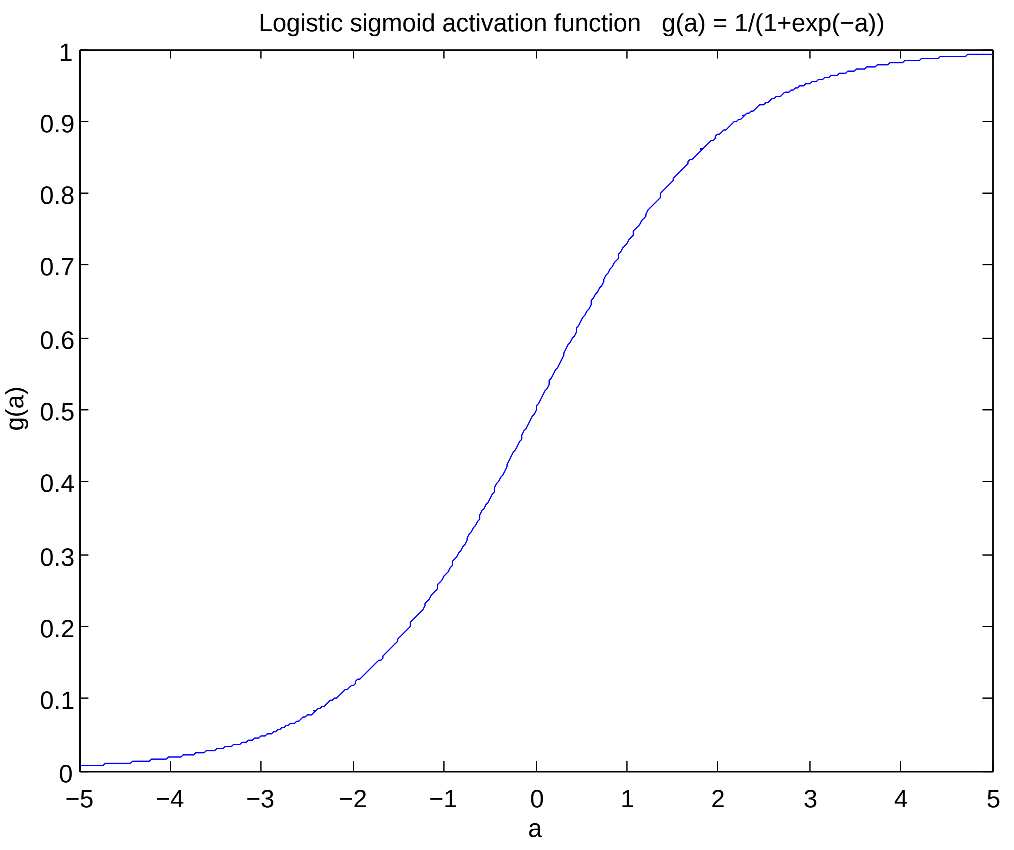

Figure 2: Sigmoid function, $g(a) = 1/(1 + \exp(-a))$

$f(a)$ is the *sigmoid activation function*, plotted in figure 2. Sigmoid means 'S'-shaped: the function maps $(-\infty, \infty)$ onto $(0, 1)$ — it is a "squashing function". If $|a|$ is small then $f(a)$ is approximately linear: so a network with logistic sigmoid activation functions approximates a linear network when the weights (and hence the inputs to the activation function) are small. As a increases, $f(a)$ saturates to 1, and as a decreases to become large and negative $f(a)$ saturates to 0.

For a single layer neural network:

$$a = \mathbf{w}^T\mathbf{x} + w_0 \tag{8}$$

If we have a single-layer neural network, with one output, and a sigmoid activation function f on the output node, then from (7) and (8) we see that the posterior probability may be written:

$$P(C_1 \mid \mathbf{x}) = f(a) = f(\mathbf{w}^T\mathbf{x} + w_0) \quad .$$

This is corresponds to a single layer neural network.

Therefore, for a two class problem (which may be represented with a single output), a single layer neural network with a sigmoid activation function on the output may be regarded as providing a posterior probability estimate.

2.2 Softmax

For more than two classes consider the following function, related to the log posterior probability of class C_k:

$$a_k = \ln p(\mathbf{x}|C_k)P(C_k)$$

a_k is the activation value of output k. If we substitute into Bayes' theorem, we find that the posterior probability is given by the following expression:

$$P(C_k|\mathbf{x}) = \frac{\exp a_k}{\sum_{\ell=1}^{K} \exp(a_\ell)}$$

This is sometimes referred to as the *softmax* or normalized exponential.

If we use this as the activation function for a multi-class single layer neural network:

$$y_k = \frac{\exp(a_k)}{\sum_\ell \exp a_\ell}$$
$$a_k = \sum_{i=0}^{d} w_{ki} x_i$$

Then it guarantees that the K output values will sum to 1, a necessary condition for probability estimates.

3 Training: Back-propagation of error

We can train a network using gradient descent. This involves defining an error function E, and then evaluating the derivatives $\partial E/\partial w_{kj}^{(2)}$ and $\partial E/\partial w_{ji}^{(1)}$. The evaluation of these error derivatives proceeds using a version the chain rule of differentiation, referred to as *back-propagation of error*, or just *backprop*.

3.1 Gradient descent

The idea of gradient descent is that to minimize an error function with respect to the weights, we want to take small steps in a *downhill* direction. We take small steps because the gradient is not uniform, and if we take too big a step we may end up going uphill again! When considering this form of optimization, we are considering the *weight space*. This is a $K \cdot (d + 1)$ dimension space, and a specific weight matrix $\mathbf{W}$ corresponds to a point in weight space. The error function evaluates the error value for a point in weight space (given the training set).

The gradient of E given $\mathbf{W}$ is written as $\nabla_{\mathbf{W}} E$, the vector of partial derivatives of E with respect to the elements of $\mathbf{W}$:

$$\nabla_{\mathbf{W}} E = \left(\frac{\partial E}{\partial w_{10}}, \ldots, \frac{\partial E}{\partial w_{ki}}, \ldots, \frac{\partial E}{\partial w_{Kd}} \right)^T \quad .$$

Descending in weight space means adjusting the weight matrix $\mathbf{W}$ by moving a small direction down the gradient, which is the direction along which E decreases most rapidly. This means adjusting the weight factor in the direction of $-\nabla_{\mathbf{W}} E$, or adjusting each weight w_{ki} by adding a factor $-\eta \cdot \partial E/\partial w_{ki}$, where η is a small constant called the *step size* or *learning rate*.

The operation of gradient descent is as follows:

1. Start with a guess for the weight matrix $\mathbf{W}$ (e.g. small randomly chosen weights)

2. Update the weights by adjusting the weight matrix by a small distance in the direction in weight space along which E decreases most rapidly: i.e. in the direction of $-\nabla_{\mathbf{W}} E$.

3. Recompute the error, and goto 2, terminating either after a fixed number of steps, or when the error stops decreasing by more than a threshold.

If we write the value of a weight at iteration τ as w_{kj}^{τ}, then its updated value is given by:

$$w_{kj}^{\tau+1} = w_{kj}^{\tau} - \eta \frac{\partial E}{\partial w_{kj}} \tag{9}$$

The learning rate η specifies how much the parameters should be adjusted along the direction of the gradient.

When training a neural network with a single hidden layer, the hidden-output weights can be trained so as to move the output values closer to the targets. However, target values are not available for hidden units, and so it is not possible to train the input-to-hidden weights in precisely the same way. This is sometimes called the *credit assignment* problem: what is the "error" of a hidden unit? how does the value of a particular input-to-hidden weight affect the overall error? The solution to this problem is found by systematically deriving expressions for the relevant derivatives using the chain rule of differentiation.

For convenience let's first consider a neural network which uses sigmoid activation functions f for the output units as well as the hidden units, so the network equation is:

$$y_k = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} f\left(\sum_{i=0}^{d} w_{ji}^{(1)} x_i\right)\right) \tag{10}$$

3.2 Error function

To train a neural network we need to define an error function (cost function). For now, we use the sum-of-squares error function, obtained by summing over a training set of N examples:

$$E = \sum_{n=1}^{N} E^n \tag{11}$$

$$E^n = \frac{1}{2} \sum_{k=1}^{K} (y_k^n - t_k^n)^2 \tag{12}$$

The values of y_k^n may be computed for each pattern using the MLP forward propagation equation (4). To avoid clutter, we'll drop the '(1)' and '(2)' superscripts when writing down weights.

To obtain the overall error gradients, we sum over the training examples:

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E^n}{\partial w_{kj}} \tag{13}$$

$$\frac{\partial E}{\partial w_{ji}} = \sum_{n=1}^{N} \frac{\partial E^n}{\partial w_{ji}} \tag{14}$$

3.3 Hidden-to-output weights

First we would like to compute the error gradients for the hidden-to-output weights, $\partial E^n / \partial w_{kj}$. Now we can write E_n in terms of these weights:

$$\begin{aligned} E^n &= \frac{1}{2} \sum_{k=1}^{K} (f(a_k^n) - t_k^n)^2 \\ &= \frac{1}{2} \sum_{k=1}^{K} \left(f\left(\sum_{j=0}^{M} w_{kj} z_j^n \right) - t_k^n \right)^2 . \end{aligned} \tag{15}$$

The derivatives of the error with respect to w_{kj} can be broken down as follows:

$$\frac{\partial E^n}{\partial w_{kj}} = \frac{\partial E^n}{\partial a_k^n} \frac{\partial a_k^n}{\partial w_{kj}} \tag{16}$$

The gradient of the error E^n with respect to the activations a_k^n is often referred to as the error signal and given the notation δ_k^n, analagous to what we had for single layer neural networks.

$$\delta_k^n = \frac{\partial E^n}{\partial a_k^n} \tag{17}$$

And since:

$$\frac{\partial a_k^n}{\partial w_{kj}} = z_j^n \tag{18}$$

we may substitute (17) and (18) into (16) to obtain:

$$\frac{\partial E^n}{\partial w_{kj}} = \delta_k^n z_j^n \tag{19}$$

where:

$$\delta_k^n = \frac{\partial E^n}{\partial y_k^n} \cdot \frac{\partial y_k^n}{\partial a_k^n} = (y_k^n - t_k^n) f'(a_k^n) \tag{20}$$

Here $\partial y_k^/ \partial a_k^n$ is the derivative of the sigmoid function $f'(a_k^n)$. It turns out that

$$f'(a) = f(a)(1 - f(a))$$

3.4 Input-to-hidden weights

Now we would like to compute the error gradients for the input-to-hidden weights, $\partial E / \partial w_{ji}$. To do this we need to make sure that we take into account all the ways in which hidden unit j (and hence weight w_{ji}) can influence the error. To do this let's look at δ_j^n, the error signal for hidden unit j:

$$\begin{aligned} \delta_j^n &= \frac{\partial E^n}{\partial b_j^n} \\ &= \sum_{k=1}^{K} \frac{\partial E^n}{\partial a_k^n} \frac{\partial a_k^n}{\partial b_j^n} \\ &= \sum_{k=1}^{K} \delta_k^n \frac{\partial a_k^n}{\partial b_j^n} \end{aligned} \tag{21}$$

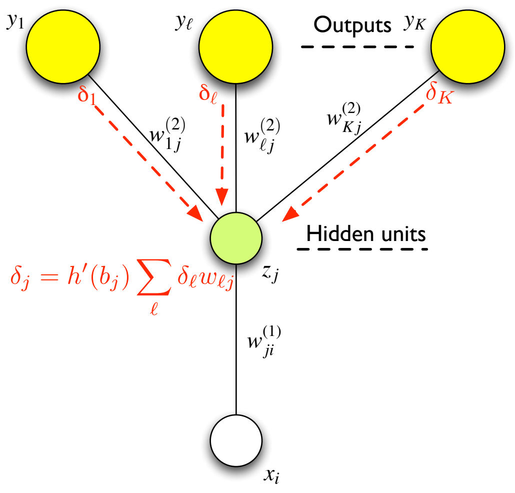

Figure 3: Back-propagation of error signals in an MLP

Since hidden unit j can influence the error through all the output units (since it is connected to all of them), we must sum over all the output units' contributions to δ_j^n. We need the expression for $\partial a_k^n / \partial b_j^n$, obtained by differentiating (2) and the sigmoid activation function:

$$\begin{aligned}\frac{\partial a_k^n}{\partial b_j^n} &= \frac{\partial a_k^n}{\partial z_j^n}\frac{\partial z_j^n}{\partial b_j^n} \\ &= w_{kj} f'(b_j^n) \end{aligned} \tag{22}$$

Substituting (22) into (21) we obtain:

$$\delta_j^n = f'(b_j^n)\sum_{k=1}^{K}\delta_k^n w_{kj}. \tag{23}$$

This is the famous *back-propagation of error* (backprop) equation. By applying the chain rule of differentiation, backprop obtains the δ values for hidden units by "back-propagating" the δ values of the outputs, weighted by the hidden-to-output weight matrix. This is illustrated in figure 3. The derivatives of the input-to-hidden weights can thus be evaluated using:

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial b_j^n}\frac{\partial b_j^n}{\partial w_{ji}} = \delta_j x_i \tag{24}$$

This approach can be recursively applied to further hidden layers.

3.5 Back-propagation algorithm

The back-propagation of error algorithm is summarised as follows:

1. Apply the N input vectors from the training set, $\mathbf{x}^n$, to the network and forward propagate using (4) to obtain the set of output vectors $\mathbf{y}^n$
2. Using the target vectors $\mathbf{t}^n$ compute the error E using (11) and (12)
3. Evaluate the error signals δ_k^n for each output unit using (20)

4. Evaluate the error signals δ_k^n for each hidden unit using back-propagation of error (23)
5. Use (19) and (24) to evaluate the derivatives for each training pattern, obtaining the overall derivatives using (13) and (14).

And the computed gradients may then be used in the gradient descent algorithm above

4 Training with softmax outputs

Consider a single-layer (no hidden layer) network with a single output y with a sigmoid activation function.

$$y = f(a) = \frac{1}{1 + \exp(-a)}$$

This is used for a two-class problem with classes C_1 (denoted by target variable $t = 1$) and C_2 (denoted by $t = 0$). We can show that in the case of a sigmoid activation function we can interpret y as the conditional probability $P(C_1 \mid \mathbf{x})$ and $(1 - y)$ as $P(C_2 \mid \mathbf{x})$, where $\mathbf{x}$ is the input vector.

The target t is a binary variable. We know that, given $\mathbf{x}$, the probability of $t = 1$ is $P(t = 1 \mid \mathbf{x}) = P(C_1 \mid \mathbf{x}) = y$; likewise we have $P(t = 0 \mid \mathbf{x}) = P(C_2 \mid \mathbf{x}) = 1 - y$. We can combine this information and write the distribution of the target t in the form:

$$P(t \mid x, \mathbf{W}) = y^t + (1 - y)^{1-t}$$

(This is called the Bernoulli distribution.) Note that we have also explicitly shown the dependence on the weights. We can write the log probability:

$$\ln P(t \mid x, \mathbf{W}) = t \ln y + (1 - t) \ln(1 - y)$$

We can use this to optimise the weights $\mathbf{W}$ to maximise the log probability – or to minimise the negative log probability. We can do this by writing the error function as follows:

$$E(\mathbf{W}) = -(t \ln y + (1 - t) \ln(1 - y)) \,.$$

This is the error for a single training example, denoted as E^n above. To avoid clutter the superscript n is ignored in this section. If we want to train by gradient descent, we need the derivative $\partial E/\partial w_i$ (where w_i connects the ith input to the single output). First we look at the single sigmoid output.

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a} \cdot \frac{\partial a}{\partial w_i} \\
\frac{\partial E}{\partial y} &= -\frac{t}{y} + \frac{1-t}{1-y} \\
&= \frac{-(1-y)t + y(1-t)}{y(1-y)} \\
\frac{\partial y}{\partial a} &= y(1-y) \quad \text{(usual sigmoid derivative)} \\
\frac{\partial a}{\partial w_i} &= x_i \quad \text{(as usual)} \\
\frac{\partial E}{\partial w_i} &= \frac{-(1-y)t + y(1-t)}{y(1-y)} \cdot y(1-y) \cdot x_i \\
&= (-(1-y)t + y(1-t))x_i = (y - t)x_i
\end{aligned}$$

So with a sigmoid activation function and the negative log probability error function, the error signal $\delta = \partial E / \partial a = (y - t)$. The derivative of the sigmoid cancels out. The sigmoid activation function corresponds to a two class posterior probability estimation, and the negative log probability is the 'natural' or consistent error function: the network estimates a posterior probability, and the error function corresponds to directly optimising the parameters to estimate that (log) probability.

Now consider the C class case in which we have a "1-from-C" coding scheme, and in which the kth output y_k is interpreted as $P(C_k \mid \mathbf{x})$. In this case the negative log probability error function is:

$$E(\mathbf{W}) = -\sum_{k=1}^{C} t_k \ln y_k$$

If the transfer function for the output units is the softmax:

$$y_k = \frac{\exp(a_k)}{\sum_{j=1}^{C} \exp(a_j)}$$

then we need an expression for the derivative $\partial E / \partial w_{ki}$. In this case the kth activation a_k – and hence the weight w_{ki} – influences the error function through all the output units, because of the normalising term in the denominator. We have to take this into account when differentiating.

$$\begin{aligned}
\frac{\partial E}{\partial w_{ki}} &= \sum_{c=1}^{C} \frac{\partial E}{\partial y_c} \cdot \frac{\partial y_c}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{ki}} \\
\frac{\partial a_k}{\partial w_{ki}} &= x_i (\text{as usual}) \\
\frac{\partial E}{\partial y_c} &= -\frac{t_c}{y_c}
\end{aligned}$$

Now to look at $\partial y_c / \partial a_k$ we look at two cases, when $c = k$, and when $c \neq k$.

First when $c = k$, we apply the quotient rule of differentiation:

$$\begin{aligned}
\frac{\partial y_c}{\partial a_c} &= \frac{\sum_j \exp(a_j) \cdot \exp(a_c) - \exp(a_c) \exp(a_c)}{(\sum_j \exp(a_j))^2} \\
&= y_c - y_c^2 = y_c(1 - y_c)
\end{aligned}$$

And, when $c \neq k$:

$$\begin{aligned}
\frac{\partial y_c}{\partial a_k} &= \frac{-\exp(a_c) \exp(a_k)}{(\sum_j \exp(a_j))^2} \\
&= -y_c y_k
\end{aligned}$$

We can combine these using the Kronecker delta δ_{ck} ($\delta_{ck} = 1$ if $c = k$, $\delta_{ck} = 0$ if $c \neq k$):

$$\frac{\partial y_c}{\partial a_k} = y_c(\delta_{ck} - y_k)$$

Putting these derivatives together in the chain rule we have:

$$\begin{aligned}
\frac{\partial E}{\partial w_{ki}} &= \sum_{c=1}^{C} \left(-\frac{t_c}{y_c} \right) (y_c(\delta_{ck} - y_k))\, x_i \\
&= \sum_{c=1}^{C} t_c (y_k - \delta_{kc}) x_i
\end{aligned}$$

Now, since we have a '1-from-C' output coding, we know that $\sum_c t_c y_k = y_k$ (in fact this holds for the weaker condition $\sum_c t_c = 1$), thus:

$$\frac{\partial E}{\partial w_{ki}} = (y_k - t_k)x_i$$

Beautiful! Once again the derivative of the transfer function cancels, and we have the error signal $\delta_k = y_k - t_k$. The softmax is the multiclass counterpart of the sigmoid and is the natural partner of the negative log probability error function.

Lots of subscripts in this section, so take care!

Some other comments:

- Sum-squared error function is not invalidated and is a good 'general-purpose error function'. But if you are doing classification, and interpret the outputs as posterior probability estimates, then it is consistent to maximise the probability (or, in practice, minimise the negative log probability). And you are rewarded by a nice derivative
- Don't be confused by the Kronecker delta which is very different from the error signal δ... sorry for overloaded notation, but both are standard
- logs to base e since we have exp in the transfer function
- Not having the transfer function derivative ($y(1 - y)$) results in larger derivative values, and experiments consistently indicate that this leads to faster gradient descent training
- Remember all the derivatives in the question are for a single training example; you would use these directly in stochastic gradient descent; otherwise you would sum over the training set.