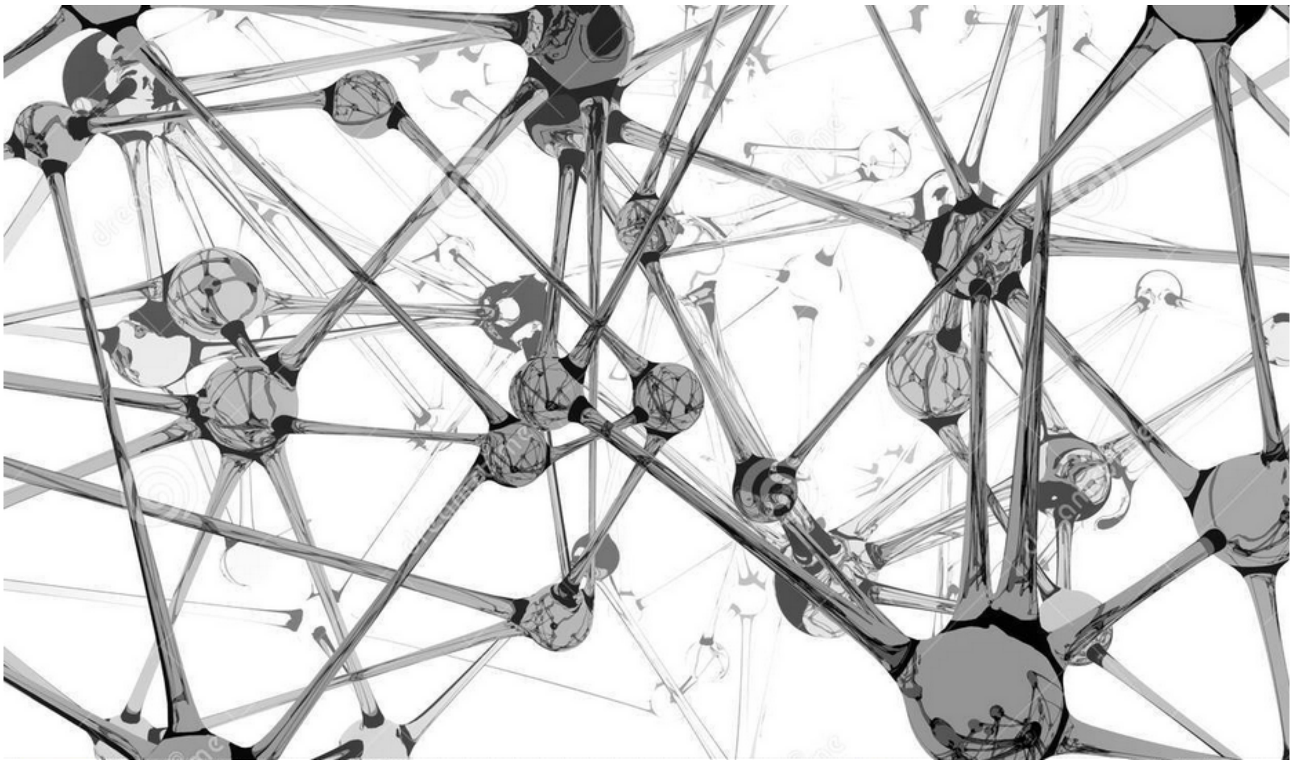


Artificial Neural Networks

Notes on feed forward artificial neural networks

Michael Wedgwood

March, 2015

Contents

1 Introduction

This document shows and derives the workings of the feed forward neural network algorithms in an explicit and comprehensive manner particularly with respect to the vectorisations which are useful for computation efficiency.

Information is motivated by mostly these sources:

Machine Learning	Coursera	Andrew Ng
Neural Networks	Coursera	Geoffrey Hinton
UFLDL	Standford Online	Andrew Ng et al
Neural Networks and Deep Learning	Online Book	Michael Nielson

2 Notation

This section outlines the various notations used throughout this document. Notations are also described in the text but this can be handy for referencing.

2.1 Indices

I have tried not to overload any indices i.e. indices should, as far as possible, always refer to the same thing.

Index number of an input feature	$n = 1 : N$ features
Index number of a training example	$m = 1 : M$ training examples
Index number of an output class	$k = 1 : K$ output classes
Index number of a layer	$\ell = 1 : L$ layers
Index number of a node in layer ℓ	$i = 1 : s_\ell$ layers (s_ℓ means size of layer ℓ e.g. $s_1 = N$ and $s_L = K$)
Index number of a node in layer $\ell - 1$	$j = 1 : s_{\ell-1}$ layers

Superscripts

Superscripts denote the training example number and / or the layer number - should be obvious from context which is being used. If both need to be used (which is rare in this document) the layer number is shown first. E.g.

$x^{(4)}$	training example 4,
$a^{(3)}$	layer number 3, and
$a^{(3)(4)}$	layer number 3 of training example 4.

Subscripts

Single subscript usually denotes the node number (in a particular layer). In a weight matrix there are two subscripts - the first subscript denotes the number of the node in the layer ahead, and the second subscript denotes the number of the node in the layer behind. Note that the number of input features is equivalent to the number of nodes in layer 1 so n is also used as subscript. Note that the numbers of output classes is equivalent to the number of nodes in layer L so k is also used as subscript. E.g.

$a_k^{(L)}$	value of the k th node in the output layer (layer L), and
$w_{ij}^{(\ell)}$	weight parameter connecting node j in layer $\ell - 1$ to node i in layer ℓ

2.2 Variables and Parameters

There are a number of variables and parameters

$x^{(m)}$	input vector containing the features of training example m
X	input matrix containing a batch (full or mini) of training examples $x^{(m)}$
$y^{(m)}$	output or "truth" of training example m - can be a vector or number
Y	output matrix (truth matrix) containing a batch (full or mini) of training examples $y^{(m)}$
$w_{ij}^{(\ell)}$	weight unit $w_{ij}^{(\ell)}$ connects node j in layer $\ell - 1$ to node i in layer ℓ
$W^{(\ell)}$	weight matrix $W^{(\ell)}$ connects layer $\ell - 1$ to layer ℓ
$b^{(\ell)}$	vector with each element the bias unit $b_i^{(\ell)}$ connecting to node i in layer ℓ
$B^{(\ell)}$	broadcasted matrix containing a batch size number of vectors $b^{(\ell)}$
$z^{(\ell)}$	vector of weighted sums of the nodes in layer $\ell - 1$ plus bias i.e. $z_i^{(\ell)} = b_i^{(\ell)} + \Sigma_j w_{ij}^{(\ell)} a_j^{(\ell-1)}$
$Z^{(\ell)}$	matrix containing vectors $z^{(\ell)}$ for a batch of training examples
$a^{(\ell)}$	vector of values of each node in layer ℓ with elements $a_i^{(\ell)} = g(z_i^{(\ell)})$
$A^{(\ell)}$	matrix containing vectors $a^{(\ell)}$ for a batch of training examples
h_k	$= a_k^{(L)}$ - helps simplify the notation
H	matrix containing vectors h for a batch of training examples
$\delta_i^{(\ell)}$	"error" at node i in layer ℓ or partial derivative of the cost function wrt $z_i^{(\ell)}$ i.e. $\partial J / \partial z_i^{(\ell)}$
λ	regularisation parameter
α	learning rate used in gradient descent algorithm
μ	momentum rate used in gradient descent algorithm

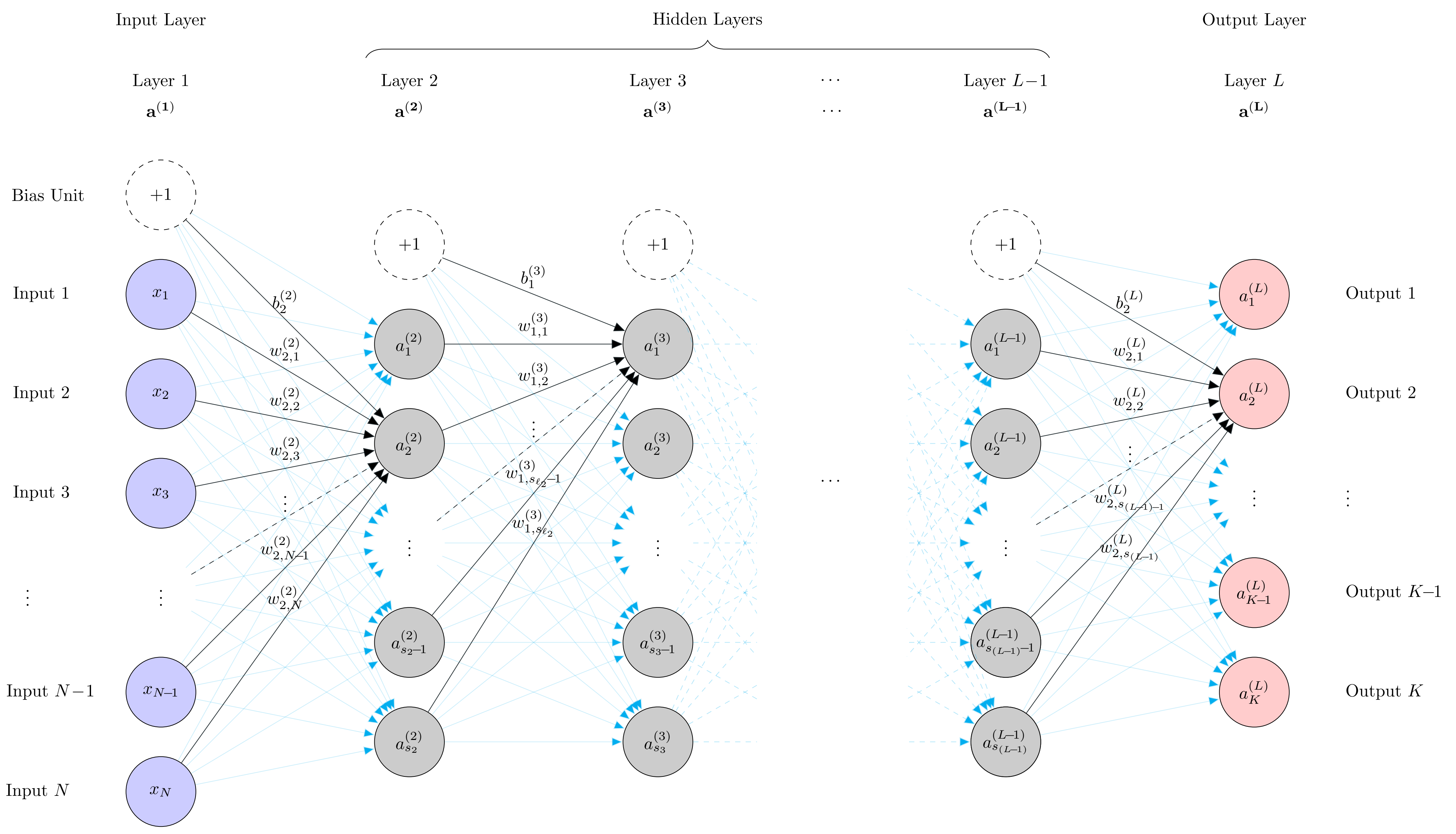
2.3 Functions

Note the g and h functions when applied to an array are applied elementwise. This isn't standard notation but should be straightforward from the text.

g	activation function of the hidden layers - typically sigmoid but can be indicator or tanh
h	hypothesis function of the output layer - sigmoid, indicator or softmax
J	cost function quantifying the disagreement between hypothesis and "truth" - quadratic (sum of squares), logistic cross entropy or softmax cross entropy

2.4 Graphical display of notation

The picture on the next sheet is an overview of the entire network structure.


Input Layer
Hidden Layers
Output Layer
Layer 1
$\mathbf{a^{(1)}}$
Layer 2
$\mathbf{a^{(2)}}$
Layer 3
$\mathbf{a^{(3)}}$
Layer $L-1$
$\mathbf{a^{(L-1)}}$
Layer L
$\mathbf{a^{(L)}}$
Bias Unit
+1
Input 1
Input 2
Input 3
Input $N-1$
Input N
x_1
x_2
x_3
x_{N-1}
x_N
$b_2^{(2)}$
$w_{2,1}^{(2)}$
$w_{2,2}^{(2)}$
$w_{2,3}^{(2)}$
$w_{2,N-1}^{(2)}$
$w_{2,N}^{(2)}$
$a_1^{(2)}$
$a_2^{(2)}$
$a_{s_2-1}^{(2)}$
$a_{s_2}^{(2)}$
$b_1^{(3)}$
$w_{1,1}^{(3)}$
$w_{1,2}^{(3)}$
$w_{1,s_{\ell_2}-1}^{(3)}$
$w_{1,s_{\ell_2}}^{(3)}$
$a_1^{(3)}$
$a_2^{(3)}$
$a_{s_3-1}^{(3)}$
$a_{s_3}^{(3)}$
$a_1^{(L-1)}$
$a_2^{(L-1)}$
$a_{s_{(L-1)}-1}^{(L-1)}$
$a_{s_{(L-1)}}^{(L-1)}$
$b_2^{(L)}$
$w_{2,1}^{(L)}$
$w_{2,2}^{(L)}$
$w_{2,s_{(L-1)}-1}^{(L)}$
$w_{2,s_{(L-1)}}^{(L)}$
$a_1^{(L)}$
$a_2^{(L)}$
$a_{K-1}^{(L)}$
$a_K^{(L)}$
Output 1
Output 2
Output $K-1$
Output K

3 Input Data

Design matrix $X \in \mathbb{R}^{N\times M}$ i.e. a matrix where each row is an individual feature (e.g. pixel value in an image) and each column is a single training example (e.g. single image in an image recognition problem).

$$X = \left.\overbrace{\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} & \cdots & x_1^{(M)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} & \cdots & x_2^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \cdots & x_n^{(m)} & \cdots & x_n^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_N^{(1)} & x_N^{(2)} & \cdots & x_N^{(m)} & \cdots & x_N^{(M)} \end{bmatrix}}^{m=1:M \text{ training examples}}\right\} \; n = 1:N \text{ features}, \quad \xrightarrow[\text{network}]{\text{feed into}} \quad \overbrace{\begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_n^{(1)} \\ \vdots \\ a_N^{(1)} \end{bmatrix}}^{a^{(1)} \text{ (layer 1)}}$$

4 Parameters - Weights and Biases

Each node in the hidden and output layers is some function of the *weighted input* (z) which is the sum of the bias unit plus the weighted sum of all the previous layer nodes i.e. $z_i^{(\ell)} = b_i^{(\ell)} + \Sigma_j w_{1j}^{(\ell)} x_j$. Thus each layer has a bias vector containing the individual biases applied to each node and a weight matrix containing the individual weights applied to each node.

$$\begin{array}{llll}
a_j^{(\ell-1)} & \xrightarrow{w_{ij}^{(\ell)}} & a_i^{(\ell)} & \text{weight unit } w_{ij}^{(\ell)} \text{ connects node } j \text{ in layer } \ell-1 \text{ to node } i \text{ in layer } \ell. \\
a^{(\ell-1)} & \xrightarrow{W^{(\ell)}} & a^{(\ell)} & \text{weight matrix } W^{(\ell)} \text{ connects layer } \ell-1 \text{ to layer } \ell. \\
+1 & \xrightarrow{b_i^{(\ell)}} & a_i^{(\ell)} & \text{bias unit } b_i^{(\ell)} \text{ connects to node } i \text{ in layer } \ell. \\
+1 & \xrightarrow{b^{(\ell)}} & a^{(\ell)} & \text{bias vector } b^{(\ell)} \text{ connects to layer } \ell.
\end{array}$$

In vector and matrix form this becomes

$$b^{(\ell)} = \begin{bmatrix} b_1^{(\ell)} \\ b_2^{(\ell)} \\ \vdots \\ b_i^{(\ell)} \\ \vdots \\ b_{s_\ell}^{(\ell)} \end{bmatrix}, \quad W^{(\ell)} = \left.\overbrace{\begin{bmatrix} w_{11}^{(\ell)} & w_{12}^{(\ell)} & \cdots & w_{1j}^{(\ell)} & \cdots & w_{1s_{\ell-1}}^{(\ell)} \\ w_{21}^{(\ell)} & w_{22}^{(\ell)} & \cdots & w_{2j}^{(\ell)} & \cdots & w_{2s_{\ell-1}}^{(\ell)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{i1}^{(\ell)} & w_{i2}^{(\ell)} & \cdots & w_{ij}^{(\ell)} & \cdots & w_{is_{\ell-1}}^{(\ell)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{s_\ell 1}^{(\ell)} & w_{s_\ell 2}^{(\ell)} & \cdots & w_{s_\ell j}^{(\ell)} & \cdots & w_{s_\ell s_{\ell-1}}^{(\ell)} \end{bmatrix}}^{j=1:s_\ell \text{ (size of layer } \ell-1)}\right\} \; i = 1 : s_{\ell+1} \text{ (size of layer } \ell)$$

5 Forward Propagation

Each node in the hidden layers is some "activation" function (g) of the *weighted input* of that layer i.e. $z^{(l)}$. And each node in the output layer is some "hypothesis" function (h) of the *weighted input* of the final layer i.e. $z^{(L)}$.

Most typically g is the logistic (sigmoid) function i.e. $g(z) = 1/(1+e^{-z})$ but can also be the tanh function i.e. $g(z) = \tanh(z)$ or the rectified linear function. More on these in a later section.

For classification problems h is often the logistic function i.e. $h(z) = 1/(1+e^{-z})$ or softmax function i.e. $h(z)_i = e^{z_i}/(\Sigma_{k=1}^{K} e^{z_k})$ for $i = 1, ..., K$. For regression problems h can be the identity function i.e $h(z) = z$. More on these in a later section.

5.1 Feed forward a single training example

Layer 2 is calculated from layer 1 as follows:

$$\begin{aligned}
a_1^{(2)} &= g(b_1^{(2)} + w_{11}^{(2)} x_1 + \cdots + w_{1j}^{(2)} x_j + \cdots + w_{1s_1}^{(2)} x_{s_1}) \\
a_2^{(2)} &= g(b_2^{(2)} + w_{21}^{(2)} x_1 + \cdots + w_{2j}^{(2)} x_j + \cdots + w_{2s_1}^{(2)} x_{s_1}) \\
&\vdots \\
a_i^{(2)} &= g(b_i^{(2)} + w_{i1}^{(2)} x_1 + \cdots + w_{ij}^{(2)} x_j + \cdots + w_{is_1}^{(2)} x_{s_1}) \\
&\vdots \\
a_{s_2}^{(2)} &= g(b_{s_2}^{(2)} + w_{s_21}^{(2)} x_1 + \cdots + w_{2j}^{(2)} x_j + \cdots + w_{s_2s_1}^{(2)} x_{s_1})
\end{aligned}
\quad or \quad
\begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ \vdots \\ a_i^{(2)} \\ \vdots \\ a_{s_2}^{(2)} \end{bmatrix} = a^{(2)} = g(b^{(2)} + W^{(2)} a^{(1)})$$

Continuing through the layers on each training example $x^{(m)}$.

$$\begin{aligned}
a^{(1)} &= x^{(m)} & & \in \mathbb{R}^{s_1} \\
a^{(2)} &= g(b^{(2)} + W^{(2)} a^{(1)}) &= g(z^{(2)}) & \in \mathbb{R}^{s_2} \\
a^{(3)} &= g(b^{(3)} + W^{(3)} a^{(2)}) &= g(z^{(3)}) & \in \mathbb{R}^{s_3} \\
&\vdots \\
a^{(\ell)} &= g(b^{(\ell)} + W^{(\ell)} a^{(\ell-1)}) &= g(z^{(\ell)}) & \in \mathbb{R}^{s_\ell} \\
&\vdots \\
a^{(L)} &= h(b^{(L)} + W^{(L)} a^{(L-1)}) &= h(z^{(\ell)}) & \in \mathbb{R}^{s_L}
\end{aligned}
\Bigg\downarrow$$

i.e. the hypothesis is generated by forward propagating each training example through the network.

So the key Forward Propagation equations for a single training example are:

Hidden Layers:

$$\boldsymbol{a^{(\ell)} = g(b^{(\ell)} + W^{(\ell)} a^{(\ell-1)})} \tag{FP1.1}$$

Final Layer:

$$\boldsymbol{a^{(L)} = h(b^{(L)} + W^{(L)} a^{(L-1)})} \tag{FP1.2}$$

5.2 Feed forward a batch of training examples

It can be more efficient to run a full batch (i.e. all training examples) or a mini batch (i.e. a subset of training examples) through the network. Instead of doing this one by one we can vectorise the calculations.

Layer 2 is calculated from layer 1 as follows:

$$\begin{aligned}
& \begin{bmatrix} b_1^{(2)} & b_1^{(2)} & \cdots & b_1^{(2)} & \cdots & b_1^{(2)} \\ b_2^{(2)} & b_2^{(2)} & \cdots & b_2^{(2)} & \cdots & b_2^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ b_i^{(2)} & b_i^{(2)} & \cdots & b_i^{(2)} & \cdots & b_i^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{s_2}^{(2)} & b_{s_2}^{(2)} & \cdots & b_{s_2}^{(2)} & \cdots & b_{s_2}^{(2)} \end{bmatrix} + \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & \cdots & w_{1j}^{(2)} & \cdots & w_{1s_1}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} & \cdots & w_{2j}^{(2)} & \cdots & w_{2s_1}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{i1}^{(2)} & w_{i2}^{(2)} & \cdots & w_{ij}^{(2)} & \cdots & w_{is_1}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{s_21}^{(2)} & w_{s_22}^{(2)} & \cdots & w_{s_2j}^{(2)} & \cdots & w_{s_2s_1}^{(2)} \end{bmatrix} \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} & \cdots & x_1^{(M)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} & \cdots & x_2^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \cdots & x_n^{(m)} & \cdots & x_n^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_N^{(1)} & x_N^{(2)} & \cdots & x_N^{(m)} & \cdots & x_N^{(M)} \end{bmatrix} \\
\in\ & \mathbb{R}^{s_2 \times M} + \mathbb{R}^{s_2 \times s_1} \ \mathbb{R}^{s_1 \times M} \\
=\ & B^{(2)} + W^{(2)} \ X \\
=\ & \begin{bmatrix} b_1^{(2)} & b_1^{(2)} & \cdots & b_1^{(2)} & \cdots & b_1^{(2)} \\ b_2^{(2)} & b_2^{(2)} & \cdots & b_2^{(2)} & \cdots & b_2^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ b_i^{(2)} & b_i^{(2)} & \cdots & b_i^{(2)} & \cdots & b_i^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{s_2}^{(2)} & b_{s_2}^{(2)} & \cdots & b_{s_2}^{(2)} & \cdots & b_{s_2}^{(2)} \end{bmatrix} + \begin{bmatrix} \Sigma_j w_{1j}^{(2)} x_j^{(1)} & \Sigma_j w_{1j}^{(2)} x_j^{(2)} & \cdots & \Sigma_j w_{1j}^{(2)} x_j^{(m)} & \cdots & \Sigma_j w_{1j}^{(2)} x_j^{(M)} \\ \Sigma_j w_{2j}^{(2)} x_j^{(1)} & \Sigma_j w_{2j}^{(2)} x_j^{(2)} & \cdots & \Sigma_j w_{2j}^{(2)} x_j^{(m)} & \cdots & \Sigma_j w_{2j}^{(2)} x_j^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \Sigma_j w_{ij}^{(2)} x_j^{(1)} & \Sigma_j w_{ij}^{(2)} x_j^{(2)} & \cdots & \Sigma_j w_{ij}^{(2)} x_j^{(m)} & \cdots & \Sigma_j w_{ij}^{(2)} x_j^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \Sigma_j w_{s_2j}^{(2)} x_j^{(1)} & \Sigma_j w_{s_2j}^{(2)} x_j^{(2)} & \cdots & \Sigma_j w_{s_2j}^{(2)} x_j^{(m)} & \cdots & \Sigma_j w_{s_2j}^{(2)} x_j^{(M)} \end{bmatrix} \\
\in\ & \mathbb{R}^{s_2 \times M} + \mathbb{R}^{s_2 \times M} \\
=\ & \begin{bmatrix} | & | & \cdots & | & \cdots & | \\ z^{(2)(1)} & z^{(2)(2)} & \cdots & z^{(2)(m)} & \cdots & z^{(2)(M)} \\ | & | & \cdots & | & \cdots & | \end{bmatrix} \\
\in\ & \mathbb{R}^{s_2 \times M} \\
=\ & Z^{(2)}
\end{aligned}$$

And finally, $A^{(2)} = g(Z^{(2)})$.

Note that $B^{(2)}$ is the “broadcasted” (i.e. repeated) $b^{(2)}$ vector. When programming this can be accomplished by a repmat or (more efficiently) broadcasting.

Continuing through the network we have:

$$\begin{array}{lclcll}
A^{(1)} & = & X & & \in & \mathbb{R}^{s_1 \times M} \\
A^{(2)} & = & g(B^{(2)} + W^{(2)}A^{(1)}) & = \; g(Z^{(2)}) & \in & \mathbb{R}^{s_2 \times M} \\
A^{(3)} & = & g(B^{(3)} + W^{(3)}A^{(2)}) & = \; g(Z^{(3)}) & \in & \mathbb{R}^{s_3 \times M} \\
\vdots & & & & & \\
A^{(\ell)} & = & g(B^{(\ell)} + W^{(\ell)}A^{(\ell-1)}) & = \; g(Z^{(\ell)}) & \in & \mathbb{R}^{s_\ell \times M} \\
\vdots & & & & & \\
A^{(L)} & = & h(B^{(L)} + W^{(L)}A^{(L-1)}) & = \; h(Z^{(L)}) & \in & \mathbb{R}^{K \times M}
\end{array} \Bigg\downarrow$$

i.e. the hypothesis is generated by forward propagating a batch of training examples through the network.

So the key Forward Propagation equations for a batch of training examples are:

Hidden Layers:

$$\boldsymbol{A^{(\ell)} = g(B^{(\ell)} + W^{(\ell)}A^{(\ell-1)})} \tag{FP2.1}$$

Final Layer:

$$\boldsymbol{A^{(L)} = h(B^{(L)} + W^{(L)}A^{(L-1)})} \tag{FP2.2}$$

6 Backward Propagation

In order to “learn” a well performing model we need to calculate the derivatives of the error / cost function $J = J(h(x), y)$ (which measures disagreement between the hypothesis h and the “truth” y) with respect to the parameters $w_{ij}^{(\ell)}$ and $b_i^{(\ell)}$ $(\forall\, i, j, l)$. I.e. we need to calculate $\partial J/\partial w_{ij}^{(\ell)}$ and $\partial J/\partial b_i^{(\ell)}$ $(\forall\, i, j, l)$.

The derivation is done by repeated use of the chain rule backwards through the network. And it is made simpler by defining and calculating what is often called the “error” of each node: $\delta_i^{(\ell)} = \partial J/\partial z_i^{(\ell)}$. [TODO - Write up of intuition about node ”error”.]

Note the following hypothesis notations that simplify the text:

$h_k^{(m)}$	$\equiv$	$h(z_k^{(L)(m)})$	$\in$	$\mathbb{R}$	The hypothesis of output class k for training example m
$h^{(m)}$	$\equiv$	$h(z^{(L)(m)})$	$\in$	$\mathbb{R}^K$	The vector of h_k’s for training example m
H	$\equiv$	$h(Z^{(L)})$	$\in$	$\mathbb{R}^{K\times M}$	The matrix of h’s for a batch of training examples

6.1 Backward propagating a single training example

Let’s initially derive the backward propagation equations for a single training example.

6.1.1 δ of final layer L

The δ of each node in the final layer L is:

$$\begin{aligned}
\delta_k^{(L)} &= \frac{\partial J}{\partial z_k^{(L)}} \\
&= \frac{\partial J}{\partial a_k^{(L)}}\frac{\partial a_k^{(L)}}{\partial z_k^{(L)}} \quad = \quad \frac{\partial J}{\partial h_k}\frac{\partial h_k}{\partial z_k^{(L)}} \qquad \text{(since } a_k^{(L)} = h(z_k^{(L)}) \equiv h_k\text{)} \\
&= \frac{\partial J}{\partial h_k}\, h'(z_k^{(L)})
\end{aligned}$$

Vectorising across all output classes $(k = 1, ..., K)$ gives

$$\begin{array}{ccccccc}
& \begin{bmatrix} \delta_1^{(L)} \\ \delta_2^{(L)} \\ \vdots \\ \delta_k^{(L)} \\ \vdots \\ \delta_K^{(L)} \end{bmatrix} & = & \begin{bmatrix} \partial J/\partial h_1 \\ \partial J/\partial h_2 \\ \vdots \\ \partial J/\partial h_k \\ \vdots \\ \partial J/\partial h_K \end{bmatrix} & \bigodot & \begin{bmatrix} h'(z_1^{(L)}) \\ h'(z_2^{(L)}) \\ \vdots \\ h'(z_k^{(L)}) \\ \vdots \\ h'(z_K^{(L)}) \end{bmatrix} \\
& \mathbb{R}^K & = & \mathbb{R}^K & \odot & \mathbb{R}^K \\
\Rightarrow & \delta^{(L)} & = & \nabla_h J & \odot & h'(z^{(L)})
\end{array}$$

So the first key Back Propagation equation for a single training example is:

$$\boldsymbol{\delta^{(L)} = \nabla_h J \odot h'(z^{(L)})} \tag{BP1.1}$$

6.1.2 $\boldsymbol{\delta}$ of each hidden layer $\boldsymbol{\ell}$

$$\begin{aligned}
\delta_j^{(\ell)} &= \frac{\partial J}{\partial z_j^{(\ell)}} \\
&= \frac{\partial J}{\partial z_1^{(\ell+1)}} \frac{\partial z_1^{(\ell+1)}}{\partial a_j^{(\ell)}} \frac{\partial a_i^{(\ell)}}{\partial z_j^{(\ell)}} \quad + \quad \frac{\partial J}{\partial z_2^{(\ell+1)}} \frac{\partial z_2^{(\ell+1)}}{\partial a_j^{(\ell)}} \frac{\partial a_j^{(\ell)}}{\partial z_j^{(\ell)}} \quad + \quad \dots \quad + \quad \frac{\partial J}{\partial z_{s_{\ell+1}}^{(\ell+1)}} \frac{\partial z_{s_{\ell+1}}^{(\ell+1)}}{\partial a_j^{(\ell)}} \frac{\partial a_j^{(\ell)}}{\partial z_j^{(\ell)}} \\
&= \delta_1^{(\ell+1)} w_{1j}^{(\ell+1)} g'(z_j^{(\ell)}) \quad + \quad \delta_2^{(\ell+1)} w_{2j}^{(\ell+1)} g'(z_j^{(\ell)}) \quad + \quad \dots \quad + \quad \delta_{s_{\ell+1}}^{(\ell+1)} w_{s_{\ell+1}j}^{(\ell+1)} g'(z_j^{(\ell)}) \\
&= \left(\sum_i \delta_i^{(\ell+1)} w_{ij}^{(\ell+1)} \right) g'(z_j^{(\ell)})
\end{aligned}$$

Vectorising across all nodes ($j = 1, ..., s_\ell$) gives

$$\begin{array}{ccccccc}
\begin{bmatrix} \delta_1^{(\ell)} \\ \delta_2^{(\ell)} \\ \vdots \\ \delta_j^{(\ell)} \\ \vdots \\ \delta_{s_\ell}^{(\ell)} \end{bmatrix} & = & \begin{bmatrix} w_{11}^{(\ell)} & w_{21}^{(\ell)} & \cdots & w_{i1}^{(\ell)} & \cdots & w_{s_{\ell+1}1}^{(\ell)} \\ w_{12}^{(\ell)} & w_{22}^{(\ell)} & \cdots & w_{i2}^{(\ell)} & \cdots & w_{s_{\ell+1}2}^{(\ell)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{1j}^{(\ell)} & w_{2j}^{(\ell)} & \cdots & w_{ij}^{(\ell)} & \cdots & w_{s_{\ell+1}j}^{(\ell)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{1s_\ell}^{(\ell)} & w_{2s_\ell}^{(\ell)} & \cdots & w_{is_\ell}^{(\ell)} & \cdots & w_{s_{\ell+1}s_\ell}^{(\ell)} \end{bmatrix} & \begin{bmatrix} \delta_1^{(\ell+1)} \\ \delta_2^{(\ell+1)} \\ \vdots \\ \delta_i^{(\ell+1)} \\ \vdots \\ \delta_{s_{\ell+1}}^{(\ell+1)} \end{bmatrix} & \odot & \begin{bmatrix} g'(z_1^{(\ell)}) \\ g'(z_2^{(\ell)}) \\ \vdots \\ g'(z_j^{(\ell)}) \\ \vdots \\ g'(z_{s_\ell}^{(\ell)}) \end{bmatrix} \\
\mathbb{R}^{s_\ell} & = & \mathbb{R}^{s_\ell \times s_{\ell+1}} & \mathbb{R}^{s_{\ell+1} \times 1} & \odot & \mathbb{R}^{s_\ell} \\
\Rightarrow \quad \delta^{(\ell)} & = & (W^{(\ell+1)})^T & \delta^{(\ell+1)} & \odot & g'(z^{(\ell)})
\end{array}$$

So the second key Back Propagation equation for a single training example is:

$$\boldsymbol{\delta^{(\ell)} = (W^{(\ell+1)})^T \delta^{(\ell+1)} \odot g'(z^{(\ell)})} \tag{BP2.1}$$

6.1.3 $\boldsymbol{\partial J / \partial w_{ij}^{(\ell)}}$

Applying the chain rule we need to calculate

$$\frac{\partial J}{\partial w_{ij}^{(\ell)}} = \frac{\partial J}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial w_{ij}^{(\ell)}}$$

We already have $\dfrac{\partial J}{\partial z_i^{(\ell)}} = \delta_i^{(\ell)}$. So we just need $\dfrac{\partial z_i^{(\ell)}}{\partial w_{ij}^{(\ell)}}$. And recall that $z_i^{(\ell)} = b_i + \sum_j w_{ij}^{(\ell)} a_j^{(\ell-1)}$ giving

$$\frac{\partial z_i^{(\ell)}}{\partial w_{ij}^{(\ell)}} = a_j^{(\ell-1)} \ .$$

I.e.

$$\frac{\partial J}{\partial w_{ij}^{(\ell)}} \;=\; \frac{\partial J}{\partial z_i^{(\ell)}}\frac{\partial z_i^{(\ell)}}{\partial w_{ij}^{(\ell)}} \;=\; \delta_i^{(\ell)} a_j^{(\ell-1)}$$

Vectorising across all nodes $(i = 1, ..., s_\ell)$ and $(j = 1, ..., s_{\ell-1})$ gives

$$\begin{bmatrix} \delta_1^{(\ell)}a_1^{(\ell-1)} & \delta_1^{(\ell)}a_2^{(\ell-1)} & \cdots & \delta_1^{(\ell)}a_j^{(\ell-1)} & \cdots & \delta_1^{(\ell)}a_{s_{\ell-1}}^{(\ell-1)} \\ \delta_2^{(\ell)}a_1^{(\ell-1)} & \delta_2^{(\ell)}a_2^{(\ell-1)} & \cdots & \delta_2^{(\ell)}a_j^{(\ell-1)} & \cdots & \delta_2^{(\ell)}a_{s_{\ell-1}}^{(\ell-1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \delta_i^{(\ell)}a_1^{(\ell-1)} & \delta_i^{(\ell)}a_2^{(\ell-1)} & \cdots & \delta_i^{(\ell)}a_j^{(\ell-1)} & \cdots & \delta_i^{(\ell)}a_{s_{\ell-1}}^{(\ell-1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \delta_{s_\ell}^{(\ell)}a_1^{(\ell-1)} & \delta_{s_\ell}^{(\ell)}a_2^{(\ell-1)} & \cdots & \delta_{s_\ell}^{(\ell)}a_j^{(\ell-1)} & \cdots & \delta_{s_\ell}^{(\ell)}a_{s_{\ell-1}}^{(\ell-1)} \end{bmatrix} = \begin{bmatrix} \delta_1^{(\ell)} \\ \delta_2^{(\ell)} \\ \vdots \\ \delta_i^{(\ell)} \\ \vdots \\ \delta_{s_\ell}^{(\ell)} \end{bmatrix} \begin{bmatrix} a_1^{(\ell-1)} & a_2^{(\ell-1)} & \cdots a_j^{(\ell-1)} & \cdots & a_{s_{\ell-1}}^{(\ell-1)} \end{bmatrix}$$

$$\in \quad \mathbb{R}^{s_\ell \times s_{\ell-1}} \qquad\qquad \in \quad \mathbb{R}^{s_\ell \times 1} \quad \mathbb{R}^{1 \times s_{\ell-1}}$$

$$\text{or} \quad \nabla_{W^{(\ell)}} J \qquad = \quad \delta^{(\ell)} \quad (a^{(\ell-1)})^T$$

So the third key Back Propagation equation for a single training example is:

$$\boldsymbol{\nabla_{W^{(\ell)}} J = \delta^{(\ell)} (a^{(\ell-1)})^T} \tag{BP3.1}$$

6.1.4 $\partial J / \partial b_i^{(\ell)}$

Applying the chain rule we need to calculate

$$\frac{\partial J}{\partial b_i^{(\ell)}} \;=\; \frac{\partial J}{\partial z_i^{(\ell)}}\frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}}$$

We already have $\frac{\partial J}{\partial z_i^{(\ell)}} = \delta_i^{(\ell)}$. So we just need $\frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}}$. And recall that $z_i^{(\ell)} = b_i + \sum_j w_{ij}^{(\ell)} a_j^{(\ell-1)}$ giving

$$\frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} \;=\; 1\ .$$

I.e.

$$\frac{\partial J}{\partial b_i^{(\ell)}} \;=\; \frac{\partial J}{\partial z_i^{(\ell)}}\frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} \;=\; \delta_i^{(\ell)} 1$$

Vectorising across all nodes $(i = 1, ..., s_\ell)$ and gives

$$\begin{array}{ccccc} & \begin{bmatrix} \delta_1^{(\ell)} \\ \delta_2^{(\ell)} \\ \vdots \\ \delta_i^{(\ell)} \\ \vdots \\ \delta_{s_\ell}^{(\ell)} \end{bmatrix} & = & \begin{bmatrix} \delta_1^{(\ell)} \\ \delta_2^{(\ell)} \\ \vdots \\ \delta_i^{(\ell)} \\ \vdots \\ \delta_{s_\ell}^{(\ell)} \end{bmatrix} & \begin{bmatrix} 1 \end{bmatrix} \\ \in & \mathbb{R}^{s_\ell} & \in & \mathbb{R}^{s_\ell} & \mathbb{R} \\ \text{or} & \nabla_{b^{(\ell)}} J & = & \delta^{(\ell)} & \end{array}$$

So the fourth key Back Propagation equation for a single training example is:

$$\boldsymbol{\nabla_{b^{(\ell)}} J = \delta^{(\ell)}} \tag{BP4.1}$$

6.1.5 Implementation

Putting this all together we have:

$$\boldsymbol{\delta^{(L)} = \nabla_h J \odot h'(z^{(L)})} \tag{BP1.1}$$

$$\boldsymbol{\delta^{(\ell)} = (W^{(\ell+1)})^T \delta^{(\ell+1)} \odot g'(z^{(\ell)})} \tag{BP2.1}$$

$$\boldsymbol{\nabla_{W^{(\ell)}} J = \delta^{(\ell)} (a^{(\ell-1)})^T} \tag{BP3.1}$$

$$\boldsymbol{\nabla_{b^{(\ell)}} J = \delta^{(\ell)}} \tag{BP4.1}$$

So to calculate the final layer gradients we need:

$\nabla_h J$	–	available from forward pass
$h'(z^{(L)})$	–	available from forward pass
$a^{(L-1)}$	–	available from forward pass

And to calculate the hidden layer gradients we need:

$\delta^{(\ell+1)}$	–	available from backward propagation
$g'(z^{(\ell)})$	–	available from forward pass
$a^{(\ell-1)}$	–	available from forward pass

6.2 Backward propagate a batch of training examples

As noted in the section on forward propagation it can be more efficient to train over a batch of training examples at a time instead of one-by-one. And when doing this it can be simpler to vectorise the implementation.

Each of the cost functions used averages the cost over all the training examples (or, if not the average i.e. $\times 1/M$, at least sums up the cost over all the training examples) i.e.

$$J(\text{Batch}) \;=\; \sum_{m=1}^{M} J(\text{Single Example})$$

So the gradient matrix $\nabla_{W^{(\ell)}} J$ will have elements

$$\partial J/w_{ij}^{(\ell)} = \sum_{m=1}^{M} \delta_i^{(\ell)(m)} a_j^{(\ell-1)(m)} \text{ instead of } \partial J/w_{ij}^{(\ell)} = \delta_i^{(\ell)} a_j^{(\ell-1)}.$$

And the gradient vector $\nabla_{b^{(\ell)}} J$ will have elements

$$\partial J/b_i^{(\ell)} = \sum_{m=1}^{M} \delta_i^{(\ell)(m)} \text{ instead of } \partial J/b_i^{(\ell)} = \delta_i^{(\ell)}.$$

The sections below show a method of accomplishing this.

6.2.1 δ of final layer L

Vectorising across all output classes ($k = 1, ..., K$) and across a batch of training examples ($m = 1, ..., M$) gives

$$\begin{array}{lcccccc}
& \begin{bmatrix} \delta_1^{(L)(1)} & \cdots & \delta_1^{(L)(M)} \\ \delta_2^{(L)(1)} & \cdots & \delta_2^{(L)(M)} \\ \vdots & \vdots & \vdots \\ \delta_k^{(L)(1)} & \cdots & \delta_k^{(L)(M)} \\ \vdots & \vdots & \vdots \\ \delta_K^{(L)(1)} & \cdots & \delta_K^{(L)(M)} \end{bmatrix} & = & \begin{bmatrix} \partial J/\partial h_1^{(1)} & \cdots & \partial J/\partial h_1^{(M)} \\ \partial J/\partial h_2^{(1)} & \cdots & \partial J/\partial h_1^{(M)} \\ \vdots & \vdots & \vdots \\ \partial J/\partial h_k^{(1)} & \cdots & \partial J/\partial h_1^{(M)} \\ \vdots & \vdots & \vdots \\ \partial J/\partial h_K^{(1)} & \cdots & \partial J/\partial h_1^{(M)} \end{bmatrix} & \odot & \begin{bmatrix} h'(z_1^{(L)(1)}) & \cdots & h'(z_1^{(L)(M)}) \\ h'(z_2^{(L)(1)}) & \cdots & h'(z_2^{(L)(M)}) \\ \vdots & \vdots & \vdots \\ h'(z_k^{(L)(1)}) & \cdots & h'(z_k^{(L)(M)}) \\ \vdots & \vdots & \vdots \\ h'(z_K^{(L)(1)}) & \cdots & h'(z_K^{(L)(M)}) \end{bmatrix} \\
& \mathbb{R}^{K\times M} & = & \mathbb{R}^{K\times M} & \odot & \mathbb{R}^{K\times M} \\
\Rightarrow & \Delta^{(L)} & = & \nabla_H J & \odot & h'(Z^{(L)})
\end{array}$$

I.e. the first key Back Propagation equation for a batch of training examples is:

$$\boldsymbol{\Delta^{(L)} = \nabla_H J \odot h'(Z^{(L)})} \qquad \text{(BP1.2)}$$

6.2.2 δ of each hidden layer ℓ

Vectorising across all nodes ($j = 1, ..., s_\ell$) and training examples ($m = 1, ..., M$) gives

$$
\begin{array}{lccc}
& \begin{bmatrix}
w_{11}^{(\ell)} & w_{21}^{(\ell)} & \cdots & w_{i1}^{(\ell)} & \cdots & w_{s_{\ell+1}1}^{(\ell)} \\
w_{12}^{(\ell)} & w_{22}^{(\ell)} & \cdots & w_{i2}^{(\ell)} & \cdots & w_{s_{\ell+1}2}^{(\ell)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
w_{1j}^{(\ell)} & w_{2j}^{(\ell)} & \cdots & w_{ij}^{(\ell)} & \cdots & w_{s_{\ell+1}j}^{(\ell)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
w_{1s_\ell}^{(\ell)} & w_{2s_\ell}^{(\ell)} & \cdots & w_{is_\ell}^{(\ell)} & \cdots & w_{s_{\ell+1}s_\ell}^{(\ell)}
\end{bmatrix}
&
\begin{bmatrix}
\delta_1^{(\ell+1)(1)} & \cdots & \delta_1^{(\ell+1)(M)} \\
\delta_2^{(\ell+1)(1)} & \cdots & \delta_2^{(\ell+1)(M)} \\
\vdots & \ddots & \vdots \\
\delta_j^{(\ell+1)(1)} & \cdots & \delta_j^{(\ell+1)(M)} \\
\vdots & \ddots & \vdots \\
\delta_{s_\ell+1}^{(\ell+1)(1)} & \cdots & \delta_{s_\ell+1}^{(\ell+1)(M)}
\end{bmatrix}
\odot
&
\begin{bmatrix}
g'(z_1^{(\ell)(1)}) & \cdots & g'(z_1^{(\ell)(M)}) \\
g'(z_2^{(\ell)(1)}) & \cdots & g'(z_2^{(\ell)(M)}) \\
\vdots & \ddots & \vdots \\
g'(z_i^{(\ell)(1)}) & \cdots & g'(z_i^{(\ell)(M)}) \\
\vdots & \ddots & \vdots \\
g'(z_{s_\ell}^{(\ell)(1)}) & \cdots & g'(z_{s_\ell}^{(\ell)(M)}))
\end{bmatrix} \\
\in & \mathbb{R}^{s_\ell \times s_{\ell+1}} & \mathbb{R}^{s_{\ell+1} \times M} \quad \odot & \mathbb{R}^{s_\ell \times M} \\
= & (W^{(\ell+1)})^T & \Delta^{(\ell+1)} \quad \odot & g'(Z^{(\ell)})
\end{array}
$$

$$
\begin{array}{lcc}
= & \begin{bmatrix}
\left(\sum_i \delta_i^{(\ell+1)(1)} w_{i1}^{(\ell+1)}\right) g'(z_1^{(\ell)(1)}) & \cdots & \left(\sum_i \delta_i^{(\ell+1)(1)} w_{i1}^{(\ell+1)}\right) g'(z_1^{(\ell)(M)}) \\
\left(\sum_i \delta_i^{(\ell+1)(1)} w_{i2}^{(\ell+1)}\right) g'(z_2^{(\ell)(1)}) & \cdots & \left(\sum_i \delta_i^{(\ell+1)(1)} w_{i2}^{(\ell+1)}\right) g'(z_2^{(\ell)(M)}) \\
\vdots & \ddots & \vdots \\
\left(\sum_i \delta_i^{(\ell+1)(1)} w_{ij}^{(\ell+1)}\right) g'(z_j^{(\ell)(1)}) & \cdots & \left(\sum_i \delta_i^{(\ell+1)(1)} w_{ij}^{(\ell+1)}\right) g'(z_j^{(\ell)(M)}) \\
\vdots & \ddots & \vdots \\
\left(\sum_i \delta_i^{(\ell+1)(1)} w_{is_\ell}^{(\ell+1)}\right) g'(z_{s_\ell}^{(\ell)(1)}) & \cdots & \left(\sum_i \delta_i^{(\ell+1)(1)} w_{is_\ell}^{(\ell+1)}\right) g'(z_{s_\ell}^{(\ell)(M)})
\end{bmatrix}
= &
\begin{bmatrix}
\delta_1^{(\ell)(1)} & \cdots & \delta_1^{(\ell)(M)} \\
\delta_2^{(\ell)(1)} & \cdots & \delta_2^{(\ell)(M)} \\
\vdots & \ddots & \vdots \\
\delta_j^{(\ell)(1)} & \cdots & \delta_j^{(\ell)(M)} \\
\vdots & \ddots & \vdots \\
\delta_{s_\ell}^{(\ell)(1)} & \cdots & \delta_{s_\ell}^{(\ell)(M)}
\end{bmatrix} \\
\in & \mathbb{R}^{s_\ell \times M} & \mathbb{R}^{s_\ell \times M} \\
= & \Delta^{(\ell)} &
\end{array}
$$

I.e. the second key Back Propagation equation for a batch of training examples is:

$$
\boldsymbol{\Delta^{(\ell)} = (W^{(\ell+1)})^T \Delta^{(\ell+1)} \odot g'(Z^{(\ell)})} \tag{BP2.2}
$$

6.2.3 $\partial J / \partial w_{ij}^{(\ell)}$

Vectorising across all nodes $(i = 1, ..., s_\ell)$ and $(j = 1, ..., s_{\ell-1})$ and training examples $(m = 1, ..., M)$ gives

$$
\begin{array}{lcc}
& \begin{bmatrix}
\delta_1^{(\ell)(1)} & \delta_1^{(\ell)(2)} & \cdots & \delta_1^{(\ell)(m)} & \cdots & \delta_1^{(\ell)(M)} \\
\delta_2^{(\ell)(1)} & \delta_2^{(\ell)(2)} & \cdots & \delta_2^{(\ell)(m)} & \cdots & \delta_2^{(\ell)(M)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\delta_i^{(\ell)(1)} & \delta_i^{(\ell)(2)} & \cdots & \delta_i^{(\ell)(m)} & \cdots & \delta_i^{(\ell)(M)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\delta_{s_\ell}^{(\ell)(1)} & \delta_{s_\ell}^{(\ell)(2)} & \cdots & \delta_{s_\ell}^{(\ell)(m)} & \cdots & \delta_{s_\ell}^{(\ell)(M)}
\end{bmatrix}
&
\begin{bmatrix}
a_1^{(\ell-1)(1)} & a_2^{(\ell-1)(1)} & \cdots & a_j^{(\ell-1)(1)} & \cdots & a_{s_\ell-1}^{(\ell-1)(1)} \\
a_1^{(\ell-1)(2)} & a_2^{(\ell-1)(2)} & \cdots & a_j^{(\ell-1)(2)} & \cdots & a_{s_\ell-1}^{(\ell-1)(2)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
a_1^{(\ell-1)(m)} & a_2^{(\ell-1)(m)} & \cdots & a_j^{(\ell-1)(m)} & \cdots & a_{s_\ell-1}^{(\ell-1)(m)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
a_1^{(\ell-1)(M)} & a_2^{(\ell-1)(M)} & \cdots & a_j^{(\ell-1)(M)} & \cdots & a_{s_\ell-1}^{(\ell-1)(M)}
\end{bmatrix} \\
\in & \mathbb{R}^{s_\ell \times M} & \mathbb{R}^{M \times s_{\ell-1}} \\
= & \Delta^{(\ell)} & (A^{(\ell-1)})^T
\end{array}
$$

$$
\begin{aligned}
&= \begin{bmatrix}
\sum_m \delta_1^{(\ell)(m)} a_1^{(\ell-1)(m)} & \sum_m \delta_1^{(\ell)(m)} a_2^{(\ell-1)(m)} & \cdots & \sum_m \delta_1^{(\ell)(m)} a_j^{(\ell-1)(m)} & \cdots & \sum_m \delta_1^{(\ell)(m)} a_{s_{\ell-1}}^{(\ell-1)(m)} \\
\sum_m \delta_2^{(\ell)(m)} a_1^{(\ell-1)(m)} & \sum_m \delta_2^{(\ell)(m)} a_2^{(\ell-1)(m)} & \cdots & \sum_m \delta_2^{(\ell)(m)} a_j^{(\ell-1)(m)} & \cdots & \sum_m \delta_2^{(\ell)(m)} a_{s_{\ell-1}}^{(\ell-1)(m)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\sum_m \delta_i^{(\ell)(m)} a_1^{(\ell-1)(m)} & \sum_m \delta_i^{(\ell)(m)} a_2^{(\ell-1)(m)} & \cdots & \sum_m \delta_i^{(\ell)(m)} a_j^{(\ell-1)(m)} & \cdots & \sum_m \delta_i^{(\ell)(m)} a_{s_{\ell-1}}^{(\ell-1)(m)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\sum_m \delta_{s_\ell}^{(\ell)(m)} a_1^{(\ell-1)(m)} & \sum_m \delta_{s_\ell}^{(\ell)(m)} a_2^{(\ell-1)(m)} & \cdots & \sum_m \delta_{s_\ell}^{(\ell)(m)} a_j^{(\ell-1)(m)} & \cdots & \sum_m \delta_{s_\ell}^{(\ell)(m)} a_{s_{\ell-1}}^{(\ell-1)(m)}
\end{bmatrix} \\
&\in \quad \mathbb{R}^{s_\ell \times s_\ell - 1} \\
&= \quad \nabla_{W^{(\ell)}} J
\end{aligned}
$$

And we've arrived in index hell. But we have accomplished our goal of calculating the matrix $\nabla_{W^{(\ell)}} J$.

I.e. the third key Back Propagation equation for a batch of training examples is:

$$\boldsymbol{\nabla_{W^{(\ell)}} J = \Delta^{(\ell)} (A^{(\ell-1)})^T} \tag{BP3.2}$$

6.2.4 $\partial J / \partial b_i^{(\ell)}$

Vectorising across all nodes ($i = 1, ..., s_\ell$) and training examples ($m = 1, ..., M$) gives

$$
\begin{array}{cccc}
\begin{bmatrix}
\delta_1^{(\ell)(1)} & \delta_1^{(\ell)(2)} & \cdots & \delta_1^{(\ell)(m)} & \cdots & \delta_1^{(\ell)(M)} \\
\delta_2^{(\ell)(1)} & \delta_2^{(\ell)(2)} & \cdots & \delta_2^{(\ell)(m)} & \cdots & \delta_2^{(\ell)(M)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\delta_i^{(\ell)(1)} & \delta_i^{(\ell)(2)} & \cdots & \delta_i^{(\ell)(m)} & \cdots & \delta_i^{(\ell)(M)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\delta_{s_\ell}^{(\ell)(1)} & \delta_{s_\ell}^{(\ell)(2)} & \cdots & \delta_{s_\ell}^{(\ell)(m)} & \cdots & \delta_{s_\ell}^{(\ell)(M)}
\end{bmatrix}
&
\begin{bmatrix}
1^{(1)} \\ 1^{(2)} \\ \vdots \\ 1^{(m)} \\ \vdots \\ 1^{(M)}
\end{bmatrix}
& = &
\begin{bmatrix}
\sum_m \delta_1^{(\ell)(m)} \\ \sum_m \delta_2^{(\ell)(m)} \\ \vdots \\ \sum_m \delta_i^{(\ell)(m)} \\ \vdots \\ \sum_m \delta_{s_\ell}^{(\ell)(m)}
\end{bmatrix} \\
\in \quad \mathbb{R}^{s_\ell \times M} & \mathbb{R}^{M \times 1} & \in & \mathbb{R}^{s_\ell} \\
\Rightarrow \quad \Delta^{(\ell)} & \overrightarrow{1} & = & \nabla_{b^{(\ell)}} J
\end{array}
$$

I.e. the fourth key Back Propagation equation for a batch of training examples is:

$$\boldsymbol{\nabla_{b^{(\ell)}} J = \Delta^{(\ell)} \overrightarrow{1}} \tag{BP4.2}$$

7 Hidden Layer Activation Functions

An idea behind the initial activation functions (e.g. see step function below) is that the neuron either "fires" (i.e. delivers a pulse to the next layer) or does not "fire" (i.e. does not deliver a pulse to the next layer). These have been modified to sigmoid shaped functions (e.g. see logistic and tanh functions below) which output in a range of completely "firing" to completely not "firing" and are continuous and differentiable and therefore able to provide meaningful information as to whether small changes in the parameters are helping or hurting the model.

7.1 Step

The step function is the activation function of the original perceptron model. The neuron outputs either 1 (i.e. it fires) or 0 (does not fire). The problem with the step function is that it has a jump discontinuity at 0 and therefore does not differentiate. I.e. we are not able to see whether small changes in input result in a higher cost (worse model) or lower cost (better model). For this reason it's not often used.

$$g(z_i^{(\ell)}) = \begin{cases} 1 & \text{if } z_i^{(\ell)} >= 0 \\ 0 & \text{if } z_i^{(\ell)} < 0 \end{cases}$$

7.2 Logistic

The logistic function outputs in the range $(0, 1)$ and is both continuous and differentiable.

$$g(z_i^{(\ell)}) = \frac{1}{1 + e^{-z_i^{(\ell)}}}$$

$$\begin{aligned} g'(z_i^{(\ell)}) &= g(z_i^{(\ell)})(1 - g(z_i^{(\ell)})) && \text{(See appendix for derivation)} \\ &= a_i^{(\ell)}(1 - a_i^{(\ell)}) \end{aligned}$$

Or completely vectorised,

$$g'(Z^{(\ell)}) = A^{(\ell)} \odot (1 - A^{(\ell)})$$

Note that the matrix $A^{(\ell)}$ is available from the forward pass through the network.

7.3 Tanh

The tanh function outputs in the range $(-1, 1)$ and is both continuous and differentiable.

$$g(z_i^{(\ell)}) = \tanh(z_i^{(\ell)}) = \frac{e^{z_i^{(\ell)}} - e^{-z_i^{(\ell)}}}{e^{z_i^{(\ell)}} + e^{-z_i^{(\ell)}}}$$

$$\begin{aligned} g'(z_i^{(\ell)}) &= 1 - (g(z_i^{(\ell)}))^2 && \text{(See appendix for derivation)} \\ &= 1 - (a_i^{(\ell)})^2 \end{aligned}$$

Or completely vectorised,

$$g'(Z^{(\ell)}) = 1 - (A^{(\ell)} \odot A^{(\ell)}) \qquad \text{(i.e. } 1- \text{ elementwise squared)}$$

Note that the matrix $A^{(\ell)}$ is available from the forward pass through the network.

7.4 Rectifier

The rectifier function is not sigmoid and instead outputs either 0 or a positive real. It is not smooth at 0 but the gradient can practically be calculated by setting to 0 at that point. A neuron applying the rectifier function is called a rectified linear unit or ReLu. It has been argued that they are more biologically plausible than the sigmoid functions. And it also has the benefit of being (compared to the two sigmoid functions above) more efficient to calculate both the function and the gradient.

$$g(z_i^{(\ell)}) = \max(0, z_i^{(\ell)})$$

$$\begin{aligned} g'(z_i^{(\ell)}) &= \begin{cases} 0 & \text{if } z_i^{(\ell)} <= 0 \\ 1 & \text{if } z_i^{(\ell)} > 0 \end{cases} \\ &= \begin{cases} 0 & \text{if } a_i^{(\ell)} = 0 \\ 1 & \text{if } a_i^{(\ell)} > 0 \end{cases} \end{aligned}$$

8 Final Layer Hypothesis and Cost Functions

The final layer neuron hypothesis / activation function provides the output of the model i.e. the model's predictions.

The cost function measures the disagreement between the model predictions and the "truth". The cost is, of course, a function of the inputs x and outputs y but for the purposes of learning we can think of cost as a function of the parameters w (the collection of network weights) and b (the collection of network biases) i.e. $J = J(w, b)$.

Once the cost function is defined, the method of learning is to take steps (of w and b) in the direction of descent of the cost function with the goal of finding a local minimum that performs sufficiently well. For this reason we calculated (in the back propagation algorithm) the gradients $\partial J/\partial w_{ij}^{(\ell)}$ and $\partial J/\partial b_i^{(\ell)}$ $(\forall\, i, j, l)$. So the cost functions need to be smooth and differentiable.

Instead of looking at the hypothesis and cost functions separately I find it more helpful to look at them together. The back propagation goal for the final layer δ is to calculate:

$$\delta_k^{(L)} = \frac{\partial J}{\partial h_k} \frac{\partial h_k}{\partial z_k^{(L)}}$$

And instead of separately calculating $\partial J/\partial h_k$ and $\partial h_k/\partial z_k^{(L)}$ it can be cleaner to calculate:

$$\delta_k^{(L)} = \frac{\partial J}{\partial z_k^{(L)}}$$

The reason it can be simpler is that typically the hypothesis function and cost function are used in pairs that are a consequence of the maximum likelihood derivation of GLMs. The table below shows the common pairings and their typical uses.

Hypothesis	**Cost**	**Typical Use**
Identity	Quadratic	Regression
Logistic	Cross Entropy	Binary classification or one-vs-all multi-class classification
Softmax	Log Loss	Multi-class classification

In the sections below we'll go through each of these pairs and see that they result in very similar δ functions.

8.1 Identity Hypothesis and Quadratic Cost

The identity function outputs any real value and so is useful for regression problems. The quadratic cost is often also called mean squared error. A network with zero hidden layers using the identity hypothesis function and quadratic cost function is equivalent to linear regression.

Identity Hypothesis

$$\begin{aligned} h(z_k^{(L)}) &= z_k^{(L)} \\ h'(z_k^{(L)}) &= 1 \end{aligned}$$

Quadratic Cost

$$J(w,b) \quad = \quad \frac{1}{2M}\sum_{m=1}^{M}\sum_{k=1}^{K}\left(h_k^{(m)} - y_k^{(m)}\right)^2$$

Final Layer δ

For a batch of training examples:

$$\begin{aligned}
\frac{\partial J}{\partial z_k^{(L)}} &= \frac{\partial J}{\partial h_k} h'(z_k^{(L)}) \\
&= \frac{1}{M}\sum_{m=1}^{M}(h_k^{(m)} - y_k^{(m)})
\end{aligned}$$

Or for a single training example:

$$\frac{\partial J}{\partial z_k^{(L)(m)}} \quad = \quad \frac{1}{M}(h_k^{(m)} - y_k^{(m)})$$

So the matrix of partial derivatives $\dfrac{\partial J}{\partial a_k^{(L)(m)}}$ across all output neurons ($k = 1,...,K$) and across all training examples ($m = 1,...,M$) is

$$\nabla_H J = \frac{1}{M}\begin{bmatrix}
h_1^{(1)} - y_1^{(1)} & h_1^{(2)} - y_1^{(2)} & \cdots & h_1^{(m)} - y_1^{(m)} & \cdots & h_1^{(M)} - y_1^{(M)} \\
h_2^{(1)} - y_2^{(1)} & h_2^{(2)} - y_2^{(2)} & \cdots & h_2^{(m)} - y_2^{(m)} & \cdots & h_2^{(M)} - y_2^{(M)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
h_k^{(1)} - y_k^{(1)} & h_k^{(2)} - y_k^{(2)} & \cdots & h_k^{(m)} - y_k^{(m)} & \cdots & h_k^{(M)} - y_k^{(M)} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
h_K^{(1)} - y_K^{(1)} & h_K^{(2)} - y_K^{(2)} & \cdots & h_K^{(m)} - y_K^{(m)} & \cdots & h_K^{(M)} - y_K^{(M)}
\end{bmatrix} = \frac{1}{M}(H-Y) = \frac{1}{M}(A^{(L)}-Y)$$

Note that K here are not called output classes as typically this will be a regression problem. And typically with a regression problem there will only be one output neuron i.e. if y is a row vector of dimension $1 \times M$ we'll have:

$$\nabla_H J = \frac{1}{M}\begin{bmatrix} h_1^{(1)} - y_1^{(1)} & h_1^{(2)} - y_1^{(2)} & \cdots & h_1^{(m)} - y_1^{(m)} & \cdots & h_1^{(M)} - y_1^{(M)} \end{bmatrix} = \frac{1}{M}(H-Y) = \frac{1}{M}(A^{(L)}-Y)$$

8.2 Logistic Hypothesis and Cross Entropy Cost

The logistic function is useful in binary classification problems (or as one-vs-all in multi-class classification problems). It outputs in the range $(0,1)$ with the output interpreted as the probability of the class (although in the one-vs-all case the probabilities don't sum to 1). It is both continuous and differentiable.

The logistic cross entropy (or logistic log loss) function is the consequence / result of the maximum likelihood derivation of logistic regression i.e. where the distribution of $y \in \{0,1\}$ given x and parameterised by w is assumed to be Bernoulli i.e. $y|x;w \sim \text{Bernoulli}(\phi)$. Intuitively the log error works better than quadratic cost as, if the prediction is similar to the "truth" the cost is also low, but if the prediction is incorrect with high certainty the cost should be very high (e.g. if $y = 1$ and $h \approx 0$ then $\log(h)$ (the cost) is very high).

A network with zero hidden layers using the logistic hypothesis function and cross entropy cost function is equivalent to logistic regression.

Logistic Hypothesis

$$\begin{aligned} h(z_k^{(L)}) &= \frac{1}{1+e^{-z_k^{(L)}}} \\ h'(z_k^{(L)}) &= h(z_k^{(L)})(1-h(z_k^{(L)})) && \text{(See appendix for derivation)} \\ &= a_k^{(L)}(1-a_k^{(L)}) \end{aligned}$$

Cross Entropy Cost

$$J(w,b) = -\frac{1}{M}\sum_{m=1}^{M}\sum_{k=1}^{K}\Big(y_k^{(m)}\log(h_k^{(m)}) + (1-y_k^{(m)})\log(1-h_k^{(m)})\Big)$$

Final Layer δ

For a batch of training examples:

$$\begin{aligned} \frac{\partial J}{\partial z_k^{(L)}} &= \frac{\partial J}{\partial h_k} h'(z_k^{(L)}) \\ &= \frac{1}{M}\sum_{m=1}^{M}(h_k^{(m)} - y_k^{(m)}) && \text{(See appendix for derivation)} \end{aligned}$$

Or for a single training example:

$$\frac{\partial J}{\partial z_k^{(L)(m)}} = \frac{1}{M}(h_k^{(m)} - y_k^{(m)})$$

Note the similarity with the linear regression output above. This is a consequence of them both being GLMs and this is a standard form of the cost function gradient for a GLM.

So the matrix of partial derivatives $\frac{\partial J}{\partial a_k^{(L)(m)}}$ across all output classes ($k = 1, ..., K$) and across all training examples ($m = 1, ..., M$) is

$$\nabla_H J = \frac{1}{M}\begin{bmatrix} h_1^{(1)} - y_1^{(1)} & h_1^{(2)} - y_1^{(2)} & \cdots & h_1^{(m)} - y_1^{(m)} & \cdots & h_1^{(M)} - y_1^{(M)} \\ h_2^{(1)} - y_2^{(1)} & h_2^{(2)} - y_2^{(2)} & \cdots & h_2^{(m)} - y_2^{(m)} & \cdots & h_2^{(M)} - y_2^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ h_k^{(1)} - y_k^{(1)} & h_k^{(2)} - y_k^{(2)} & \cdots & h_k^{(m)} - y_k^{(m)} & \cdots & h_k^{(M)} - y_k^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ h_K^{(1)} - y_K^{(1)} & h_K^{(2)} - y_K^{(2)} & \cdots & h_K^{(m)} - y_K^{(m)} & \cdots & h_K^{(M)} - y_K^{(M)} \end{bmatrix} = \frac{1}{M}(H - Y) = \frac{1}{M}(A^{(L)} - Y)$$

Note that for a multi-class classification problem K will be greater than 2. But if the problem is a binary classification model then K can be 1 i.e. we can have a single output neuron. In this case y will be a row vector of dimension $1 \times M$ outputting either 0 or 1 and we'll have:

$$\nabla_H J = \frac{1}{M}\begin{bmatrix} h_1^{(1)} - y_1^{(1)} & h_1^{(2)} - y_1^{(2)} & \cdots & h_1^{(m)} - y_1^{(m)} & \cdots & h_1^{(M)} - y_1^{(M)} \end{bmatrix} = \frac{1}{M}(H - Y) = \frac{1}{M}(A^{(L)} - Y)$$

8.3 Softmax Hypothesis and Log Loss

The softmax function is useful in multi-class classification problems. It outputs in the range $(0, 1)$ with the output interpreted as the probability of the class with the additional benefit that the probabilities of each class sum to 1. This is helpful information for the model if the classes are mutually exclusive.

The softmax log loss function is the consequence / result of the maximum likelihood derivation of softmax regression i.e. where the distribution of $y \in \{1, 2, ..., K\}$ given x and parameterised by w is assumed to be Multinomial i.e. $y|x; w \sim \text{Multinomial}(\phi_1, ..., \phi_K)$.

A network with zero hidden layers using the softmax hypothesis function and log loss function is equivalent to softmax regression.

Softmax Hypothesis

$$\begin{aligned} h(z_k^{(L)}) &= \frac{e^{z_k^{(L)}}}{\sum_{i=1}^{K} e^{z_i^{(L)}}} \\ h'(z_k^{(L)}) &= h(z_k^{(L)})(1 - h(z_k^{(L)})) && \text{(See appendix for derivation)} \\ &= a_k^{(L)}(1 - a_k^{(L)}) \end{aligned}$$

Log Loss Cost

$$J(w, b) = -\frac{1}{M} \sum_{m=1}^{M} \sum_{k=1}^{K} 1\{y^{(m)} = k\} \log(h_k^{(m)})$$

Final Layer δ

For a batch of training examples:

$$\begin{aligned} \frac{\partial J}{\partial z_k^{(L)}} &= \frac{\partial J}{\partial h_k} h'(z_k^{(L)}) \\ &= \frac{1}{M} \sum_{m=1}^{M} (h_k^{(m)} - y_k^{(m)}) && \text{(See appendix for derivation)} \end{aligned}$$

Or for a single training example:

$$\frac{\partial J}{\partial z_k^{(L)(m)}} = \frac{1}{M}(h_k^{(m)} - y_k^{(m)})$$

Note again the similarity with the linear regression and logistic classification output above. This is, again, a consequence of this being a standard form of the cost function gradient for a GLM.

So the matrix of partial derivatives $\dfrac{\partial J}{\partial a_k^{(L)(m)}}$ across all output classes $(k = 1, ..., K)$ and across all training examples $(m = 1, ..., M)$ is

$$\nabla_H J = \frac{1}{M} \begin{bmatrix} h_1^{(1)} - y_1^{(1)} & h_1^{(2)} - y_1^{(2)} & \cdots & h_1^{(m)} - y_1^{(m)} & \cdots & h_1^{(M)} - y_1^{(M)} \\ h_2^{(1)} - y_2^{(1)} & h_2^{(2)} - y_2^{(2)} & \cdots & h_2^{(m)} - y_2^{(m)} & \cdots & h_2^{(M)} - y_2^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ h_k^{(1)} - y_k^{(1)} & h_k^{(2)} - y_k^{(2)} & \cdots & h_k^{(m)} - y_k^{(m)} & \cdots & h_k^{(M)} - y_k^{(M)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ h_K^{(1)} - y_K^{(1)} & h_K^{(2)} - y_K^{(2)} & \cdots & h_K^{(m)} - y_K^{(m)} & \cdots & h_K^{(M)} - y_K^{(M)} \end{bmatrix} = \frac{1}{M}(H-Y) = \frac{1}{M}(A^{(L)}-Y)$$

9 Regularisation Methods

TODO - [Commentary on overfitting]

9.1 Addressing Overfitting

Two main methods used to address overfitting:

i) Reduce the number of features

- manual selection
- model selection (heuristic e.g. mutual information, or algorithmic e.g. forward search)

ii) Regularisation

- Introduce a penalty for complexity (Occam's razor for machine learning)
- Keep all features but constrain parameters
- Intuition: penalise (in the cost function) the weight parameters with a regularisation term
- λ is "tuned" empirically (e.g. by cross validation)
- L1 and L2 regularisation is equivalent to a Bayesian method of imposing prior distributions (Laplace for L1 and Gaussian for L2) on model parameters and estimating the maximum posterior (MAP) vs. the frequentist method of maximum likelihood (ML).

9.2 L2 Regularisation

Regularisation term is $\frac{\lambda}{2M}\sum_w w^2$ with derivative $\frac{\mathrm{d}}{\mathrm{d}w}\frac{\lambda}{2M}\sum_w w^2 = \frac{\lambda}{M}w$.

9.3 L1 Regularisation

Regularisation term is $\frac{\lambda}{M}\sum_w |w|$ with derivative $\frac{\mathrm{d}}{\mathrm{d}w}\frac{\lambda}{M}\sum_w |w| = \frac{\lambda}{M}\operatorname{sign}(w)$.

9.4 Dropout

TODO

10 Optimisation / Learning

We need a method to "learn" weights that minimise cost function. The main idea is to start at an initial guess and follow a gradient descent update rule.

Without deriving the gradient descent algorithm and it's variants we'll just show them using the notation of this document.

10.1 Batch Gradient Descent

This is applied to all training examples at once.

```
Loop (until convergence or some chosen number of iterations) {
    $w_{ij}^{(\ell)} := w_{ij}^{(\ell)} - \alpha \frac{\partial}{\partial w_{ij}^{(\ell)}} J \qquad \forall\, i, j, \ell$
    $b_i^{(\ell)} := b_i^{(\ell)} - \alpha \frac{\partial}{\partial b_i^{(\ell)}} J \qquad \forall\, i, \ell$
}
```

Or vectorised (and cleaner):

```
Loop (until convergence or some chosen number of iterations) {
    $W^{(\ell)} := W^{(\ell)} - \alpha \nabla_{W^{(\ell)}} J \qquad \forall\, \ell$
    $b^{(\ell)} := b^{(\ell)} - \alpha \nabla_{b^{(\ell)}} J \qquad \forall\, \ell$
}
```

10.2 Stochastic Gradient Descent (Online Gradient Descent)

This is applied to each training example one at a time.

```
Loop (until convergence or some chosen number of iterations) {
    for $m = 1, ..., M$ {
        $W^{(\ell)} := W^{(\ell)} - \alpha \nabla_{W^{(\ell)}} J \qquad \forall\, \ell$
        $b^{(\ell)} := b^{(\ell)} - \alpha \nabla_{b^{(\ell)}} J \qquad \forall\, \ell$
    }
}
```

10.3 Mini Batch Gradient Descent

This is applied to training examples in batches (or mini batches).

```
Loop (until convergence or some chosen number of iterations) {
    for each batch {
        $W^{(\ell)} := W^{(\ell)} - \alpha \nabla_{W^{(\ell)}} J \qquad \forall \ell$
        $b^{(\ell)} := b^{(\ell)} - \alpha \nabla_{b^{(\ell)}} J \qquad \forall \ell$
    }
}
```

10.4 Momentum

TODO

A Derivatives of Activation and Hypothesis Functions

A.1 Sigmoid / Logistic

$$\begin{aligned}
g(z) &= \frac{1}{1+e^{-z}} \\
g'(z) &= \frac{\mathrm{d}}{\mathrm{dz}}\frac{1}{1+e^{-z}} \\
&= \frac{-e^{-z}}{-(1+e^{-z})^2} \\
&= \frac{1+e^{-z}-1}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}}{(1+e^{-z})^2} - \left(\frac{1}{(1+e^{-z})^2}\right) \\
&= g(z) - (g(z))^2 \\
&= g(z)(1-g(z))
\end{aligned}$$

A.2 Tanh

$$\begin{aligned}
g(z) &= \tanh(z) \\
g'(z) &= \frac{\mathrm{d}}{\mathrm{dz}}\frac{\sinh(z)}{\cosh(z)} \\
&= \frac{\cosh(z)\frac{\mathrm{d}}{\mathrm{dz}}\sinh(z) - \sinh(z)\frac{\mathrm{d}}{\mathrm{dz}}\cosh(z)}{\cosh^2(z)} \\
&= \frac{\cosh^2(z)-\sinh^2(z)}{\cosh^2(z)} \\
&= 1-\tanh^2(z) \\
&= 1-(g(z))^2
\end{aligned}$$

A.3 Softmax

$$g(z_r) = \frac{e^{z_r}}{\Sigma_i e^{z_i}}$$

For the derivative there are 2 cases:

$$\begin{cases}
\text{if } r = k & \frac{\partial}{\partial z_k}g(z_r) = \frac{e^{z_r}}{\Sigma_i e^{z_i}} - \frac{e^{z_r}e^{z_k}}{(\Sigma_i e^{z_i})^2} = \frac{e^{z_r}}{\Sigma_i e^{z_i}}\left(1-\frac{e^{z_k}}{\Sigma_i e^{z_i}}\right) = g(z_r)(1-g(z_r)) \\
\text{if } r \neq k & \frac{\partial}{\partial z_k}g(z_r) = - \frac{e^{z_r}e^{z_k}}{(\Sigma_i e^{z_i})^2} = \frac{e^{z_r}}{\Sigma_i e^{z_i}}\left(-\frac{e^{z_k}}{\Sigma_i e^{z_i}}\right) = g(z_r)(-g(z_k))
\end{cases}$$

This is sometimes shortened to:

$$\frac{\partial}{\partial z_k}g(z_r) = g(z_r)(\delta_{rk} - g(z_r))$$

where δ_{rk} is the Kronecker delta function returning 1 if $r = k$.

B Derivatives of Cost Functions / Final Layer Delta

B.1 Logistic Cross Entropy

With cost function J:

$$J(h) = -\frac{1}{M}\sum_{m=1}^{M}\sum_{k=1}^{K}\left(y_k^{(m)}\log(h_k^{(m)}) + (1-y_k^{(m)})\log(1-h_k^{(m)})\right)$$

And hypothesis function h:

$$h(z_k^{(L)}) = \frac{1}{1+e^{-z_k^{(L)}}}$$

The final layer δ is:

$$\begin{aligned}
\delta_k^{(L)} &= \frac{\partial J}{\partial h_k}\frac{\partial h_k}{\partial z_k^{(L)}} \\
&= \frac{\partial}{\partial h_k}\left[-\frac{1}{M}\sum_{m=1}^{M}\sum_{k=1}^{K}\left(y_k^{(m)}\log(h_k^{(m)}) + (1-y_k^{(m)})\log(1-h_k^{(m)})\right)\right]\frac{\partial h_k}{\partial z_k^{(L)}} \\
&= -\frac{1}{M}\sum_{m=1}^{M}\left(y_k^{(m)}\frac{1}{h_k^{(m)}} - (1-y_k^{(m)})\frac{1}{(1-h_k^{(m)})}\right)\left(h_k^{(m)}(1-h_k^{(m)})\right) \\
&= -\frac{1}{M}\sum_{m=1}^{M}\left(y_k^{(m)}(1-h_k^{(m)}) - (1-y_k^{(m)})h_k^{(m)}\right) \\
&= -\frac{1}{M}\sum_{m=1}^{M}\left(y_k^{(m)} - h_k^{(m)}\right) \\
&= \frac{1}{M}\sum_{m=1}^{M}\left(h_k^{(m)} - y_k^{(m)}\right)
\end{aligned}$$

B.2 Softmax Log Loss

With cost function J:

$$J(h) = -\frac{1}{M}\sum_{m=1}^{M}\sum_{k=1}^{K} 1\{y^{(m)} = k\}\log(h_k^{(m)})$$

And hypothesis function h:

$$h(z_k^{(L)}) = \frac{e^{z_k^{(L)}}}{\sum_{i=1}^{K} e^{z_i^{(L)}}}$$

The final layer δ is:

$$\begin{aligned}
\delta_k^{(L)} &= \frac{\partial J}{\partial h_k}\frac{\partial h_k}{\partial z_k^{(L)}} \\
&= \frac{\partial}{\partial z_k^{(L)}}\left[-\frac{1}{M}\sum_{m=1}^{M}\sum_{k=1}^{K} 1\{y^{(m)} = k\}\log(h_k^{(m)})\right] \\
&= -\frac{1}{M}\sum_{m=1}^{M}\Bigg[1\{y^{(m)} = 1\}\frac{1}{h_1^{(m)}}h_1^{(m)}(-h_k^{(m)}) + ... + 1\{y^{(m)} = k\}\frac{1}{h_k^{(m)}}h_k^{(m)}(1 - h_k^{(m)}) + ... \\
&\qquad + 1\{y^{(m)} = K\}\frac{1}{h_K^{(m)}}h_K^{(m)}(-h_k^{(m)})\Bigg] \\
&= -\frac{1}{M}\sum_{m=1}^{M}\left[1\{y^{(m)} = 1\}(-h_k^{(m)}) + ... + 1\{y^{(m)} = k\}h_k^{(m)}(1 - h_k^{(m)}) + ... + 1\{y^{(m)} = K\}(-h_k^{(m)})\right] \\
&= -\frac{1}{M}\sum_{m=1}^{M}\left[1\{y^{(m)} = k\} - h_k^{(m)}\right] \\
&= \frac{1}{M}\sum_{m=1}^{M}\left[h_k^{(m)} - 1\{y^{(m)} = k\}\right]
\end{aligned}$$

If y is a vector of length K outputting 1 for the “true” class and 0 otherwise then we have:

$$\delta^{(L)} = \frac{1}{M}\sum_{m=1}^{M}\left[h^{(m)} - y^{(m)}\right]$$