Neural Networks

Neural networks are machine learning models inspired by the structure of the human brain. A neural network is composed of artificial neurons, weights and connections arranged into at least three layers: the input layer, the hidden layer, and the output layer. Fully-connected, feed-forward neural networks are known as multilayer perceptrons (MLPs). Models with multiple hidden layers are often referred to as deep neural networks (DNNs) . There are numerous neural network variants designed to tackle a wide array of problems. This document will focus on simple MLPs. Neural networks are compatible with all important learning paradigms: supervised, unsupervised, self-supervised, transfer and reinforcement learning, amongst others.

1 Perceptrons

A perceptron (or neuron) takes several binary inputs, $x_1, x_2, x_3, ...$, and produces a single binary output (Figure 1). For each input, x_i, there is a weight, w_i, that reflects the relative importance of the respective input to the output. The output of the neuron (zero or one) is determined by whether or not the weighted sum $\sum_{i=1} w_i x_i$ is greater than some threshold value. That is

$$\text{Output} = \begin{cases} 0 & \text{if} \quad \sum_{i=1} w_i x_i \leq \text{Threshold} \\ 1 & \text{if} \quad \sum_{i=1} w_i x_i > \text{Threshold} \end{cases} \tag{1}$$

Let's study an example of how the perceptron works. Say we are deciding whether or not to go to watch a football game. Our decision might depend on three factors:

1. If we have any friends to go with.
2. The probability our team has of winning the match.
3. The price of the ticket.

These factors can be represented by the corresponding binary variables $x_1, x_2,$ and x_3. We can say that if we do have friends to go with then $x_1 = 1$, and if we don't, then $x_1 = 0$. If our team has a probability of 0.5 or more of winning then $x_2 = 1$, and if not then $x_2 = 0$. Finally, if the price of a ticket is £50 or less then $x_3 = 1$, and if the price is higher then $x_3 = 0$. Now, let's say that the most important factor in determining whether we go to the match is the ticket price, followed by if we have friends to go with or not, and then the probability our team has of winning. We can select our weights to reflect this order of importance; let's say

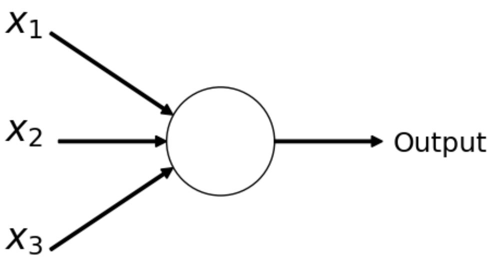

Figure 1: This perceptron has three inputs: $x_1, x_2,$ and x_3. The weights of these inputs determine their relative importance to the output.

we set $w_3 = 5, w_1 = 3,$ and $w_2 = 1$. We will now choose the threshold to be 3. In this case, the following possibilities result in an output of one (we decide to go to the game):

1. We have friends to go with, our team has a probability of 0.5 or more of winning the match, and the ticket price is £50 or less ($x_1 = x_2 = x_3 = 1$).
2. We have friends to go with, and our team has a probability of 0.5 or more of winning the match ($x_1 = x_2 = 1, x_3 = 0$).
3. The ticket price is £50 or less, and our team has a probability of 0.5 or more of winning the match ($x_1 = 0, x_2 = x_3 = 1$).
4. The ticket price is £50 or less and we have friends to go with ($x_1 = x_3 = 1, x_2 = 0$).
5. The ticket price is £50 or less ($x_1 = x_2 = 0, x_3 = 1$).

Any other possibilities result in an output of zero (we decide not to go to the game). We will decide to go the game if the ticket price is £50 or less regardless of whether or not we have friends to go with and our teams chances of winning the match. However, if the ticket price is more than £50, we will only go to the game if we have friends to go with and our team has a probability of 0.5 or more of winning the match. We can achieve different models of decision making by modifying the weights and the threshold.

We can write (1) in an alternate form:

$$\text{Output} = \begin{cases} 0 & \text{if} \quad \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if} \quad \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases} \tag{2}$$

where $\mathbf{w} \cdot \mathbf{x} = \sum_{i=1} w_i x_i$, and the bias term $b = -\text{Threshold}$.

2 Sigmoid Neurons

The sigmoid neuron has the inputs $x_1, x_2, x_3, ...,$ like the perceptron. It also has weights corresponding to each input, and the bias term b. However, unlike the perceptron, these inputs are non-binary and can take any value between zero and one. The output is also non-binary; it is given by

$$\sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + \exp[-(\mathbf{w} \cdot \mathbf{x} + b)]} \tag{3}$$

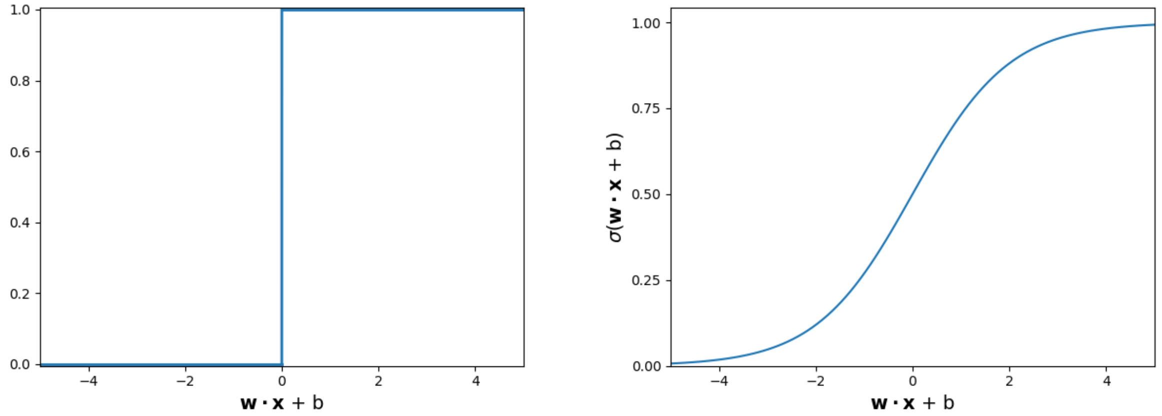

Figure 2: (left) The perceptron output. (right) The sigmoid neuron output.

The function is plotted in Figure 2. We can see that as $\mathbf{w} \cdot \mathbf{x} + b \to \infty$, $\sigma \to 1$, and as $\mathbf{w} \cdot \mathbf{x} + b \to -\infty$, $\sigma \to 0$. Hence, for large values of $|\mathbf{w} \cdot \mathbf{x} + b|$, the sigmoid function replicates the perceptron output. However, whereas the perceptron output is a step function, the sigmoid function is smooth between the values of zero and one. This smoothness is a very important feature of the sigmoid function. A small change in the weights and bias can cause the output of the standard perceptron to flip (from zero to one, or from one to zero) which makes learning very difficult in a neural network. The sigmoid output is a function of the w_i and b, so from the chain rule we have

$$\Delta Output \approx \sum_{i=1} \frac{\partial Output}{\partial w_i} \Delta w_i + \frac{\partial Output}{\partial b} \Delta b \tag{4}$$

This demonstrates explicitly that the change in output, $\Delta Output$, is a linear function of the changes Δw_i and Δb. Consequently, it is simple to make small changes to w_i and b in order to obtain a small change in the output. This is what allows a network of sigmoid neurons to learn and explains why the sigmoid function is a commonly used activation function in neural networks.

3 Rectified Linear Unit (ReLU) Neurons

The ReLU function is defined as

$$\sigma(\mathbf{w} \cdot \mathbf{x} + b) = \max\{0, \mathbf{w} \cdot \mathbf{x} + b\} \tag{5}$$

or

$$\sigma(\mathbf{w} \cdot \mathbf{x} + b) = \begin{cases} 0 & \text{if} \quad \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ \mathbf{w} \cdot \mathbf{x} + b & \text{if} \quad \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases} \tag{6}$$

The ReLU function is visualised in Figure 3. For positive values of $\mathbf{w} \cdot \mathbf{x} + b$, the derivative

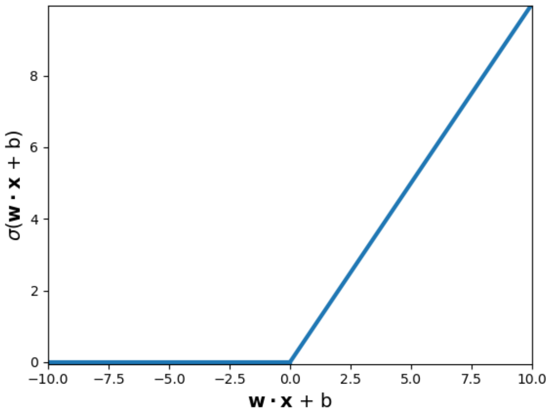

Figure 3: The ReLU activation output.

of the ReLU function is one, whereas for $\mathbf{w} \cdot \mathbf{x} + b \leq 0$ the derivative is zero.

In the majority of applications it is vital that a neural network is capable of producing non-linear decision boundaries, and this requires the use of non-linear activation functions. However, due to the gradient descent method of optimisation, neural networks are easier to optimise when their behaviour is linear. The ReLU function was developed with this in mind since it is a non-linear function that behaves mostly like a linear one. A key advantage of the ReLU function is its capability to output a true zero, something which the sigmoid function cannot do. In a neural network this sparsity is considered beneficial in limited amounts since it can accelerate learning and produce more compact and simpler models which have a greater predictive ability, and are less susceptible to overfitting. The ReLU function is also significantly less costly to compute than the sigmoid function; an important consideration in very large, deep neural networks.

4 Softmax Neurons

Say we have a layer of k neurons, with a weighted input to the i-th neuron of $\mathbf{w}_i \cdot \mathbf{x}_i + b_i$. The output of the softmax function for the i-th neuron is

$$\sigma_i = \frac{\exp(\mathbf{w}_i \cdot \mathbf{x}_i + b_i)}{\sum\limits_{j=1}^{k} \exp(\mathbf{w}_j \cdot \mathbf{x}_j + b_j)} \tag{7}$$

where the denominator represents the sum of the exponentials of the weighted inputs for all neurons in the layer. If we sum the softmax function for all neurons in the layer we get

$$\sum_{i=1}^{k} \sigma_i = \frac{\sum\limits_{i=1}^{k} \exp(\mathbf{w}_i \cdot \mathbf{x}_i + b_i)}{\sum\limits_{j=1}^{k} \exp(\mathbf{w}_j \cdot \mathbf{x}_j + b_j)} = 1 \tag{8}$$

Therefore, a softmax layer of neurons outputs a probability distribution, with the neurons having the largest weighted inputs generating the largest probabilities.

5 Neural Network Architecture

A neural network consists of at least three layers: the input layer, a hidden layer, and the output. A neuron in the hidden layer receives the inputs multiplied by their respective weights (plus the bias if there is one) and uses them as the argument to compute its activation. Similarly, a neuron in the output layer uses the activations from the hidden layer multiplied by their own respective weights (plus the bias) as the argument to compute its activation.

Figure 4 represents a simple neural network for a single-class problem. The network has an input layer containing three inputs, and a hidden layer with three neurons. The output layer contains only a single sigmoid neuron which will return an activation of > 0.5 if the test sample is predicted to belong to the class and < 0.5 if it is not. Note that in the figure, all arrows emerging from a neuron represent the same output.

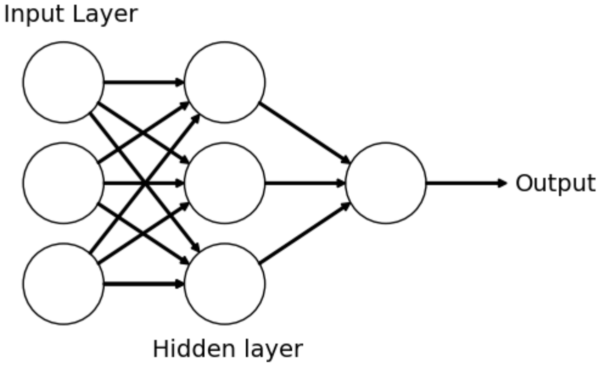

Figure 4: Representation of a simple, single-class neural network consisting of an input layer, a single hidden layer, and an output layer with one output neuron.

Figure 5 represents a multi-class neural network with two hidden layers. As before, a neuron in the first hidden layer receives the weighted inputs and computes its activation. A neuron in the second hidden layer receives the weighted first layer activations and computes its activation. Therefore, we can think of neurons in the second hidden layer as computing more abstract decisions than the neurons in the first layer. Neurons in the output layer receive the weighted activations from the second hidden layer and compute their activations which provide the output. The output layer in Figure 5 contains three neurons, which indicates that this neural network is designed to tackle a three-class classification problem.

In deep neural networks ReLU is generally employed as the activation function in the hidden layers as there are several benefits in doing so (see Appendix). We can choose to have sigmoid neurons in the output layer. These neurons will, of course, output a value between zero and one, where the neuron with the largest output specifies the predicted class of a sample. For example, if the second neuron in the output layer has the largest activation, the sample is predicted to be of the second class.

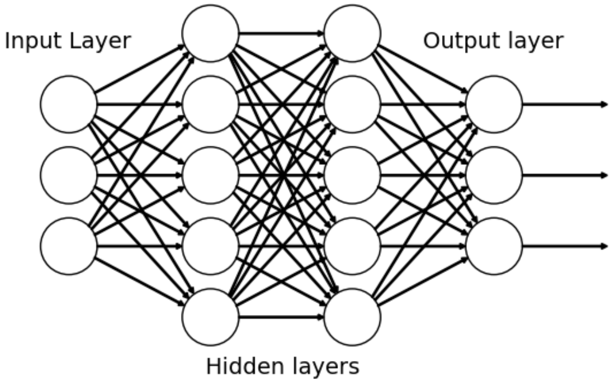

Figure 5: Representation of a simple, multi-class neural network consisting of an input layer, two hidden layers, and an output layer with three neurons.

To provide a more explicit example let's denote $\boldsymbol{y}$ as the desired output vector from the network. Then for a sample of the first class $\boldsymbol{y} = (1, 0, 0)^T$, for a sample of the second class $\boldsymbol{y} = (0, 1, 0)^T$, and a sample of the third class $\boldsymbol{y} = (0, 0, 1)^T$. Introducing the notation $\boldsymbol{a}^L$ as the actual output vector from the network of L layers, in this example we could have for a sample, say $\boldsymbol{a}^4 = (0.1, 0.8, 0.3)$, which would indicate a prediction of the second class.

We could instead utilise an output layer of softmax neurons which is usually advantageous in multi-class problems. As we saw earlier, such a layer outputs a probability distribution which makes for the most intuitive class predictions. For example, for a sample we could get the output vector $\boldsymbol{a}^4 = (0.04, 0.07, 0.89)$ which would indicate a prediction of the third class.

Up to now we have focused exclusively on classification tasks, but neural networks are frequently used for linear regression also. In this case there will only be a single neuron in the output layer (since the output is a scalar) and it will not use any activation function, since we have no desire to transform its weighted input. For example, if the dependent variable in our linear regression problem is population height, a sigmoid activation output between zero and one will not be very helpful.

6 Forward Propagation

We introduce the notation w^l_{jk} to denote the weight multiplied with the activation of the k-th neuron in the $(l-1)$-th layer as it is input into the j-th neuron in the l-th layer. For example, w^4_{12} is the weight multiplied with the activation of the second neuron in the third layer for its input into the first neuron in the fourth layer. Similarly, b^l_j represents the bias for the j-th neuron in the l-th layer, and a^l_j is the activation of the j-th neuron in the l-th layer.

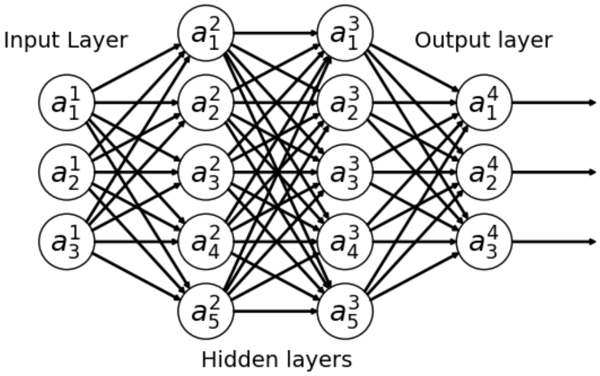

Figure 6: The neural network from Figure 5 with the neuron activations indicated.

Considering the network represented in Figure 6, and using this notation we can write, for example

$$\begin{aligned}
a_1^2 &= \sigma(w_{11}^2 a_1^1 + w_{12}^2 a_2^1 + w_{13}^2 a_3^1 + b_1^2)\\
a_2^2 &= \sigma(w_{21}^2 a_1^1 + w_{22}^2 a_2^1 + w_{23}^2 a_3^1 + b_2^2)\\
a_3^2 &= \sigma(w_{31}^2 a_1^1 + w_{32}^2 a_2^1 + w_{33}^2 a_3^1 + b_3^2)\\
a_4^2 &= \sigma(w_{41}^2 a_1^1 + w_{42}^2 a_2^1 + w_{43}^2 a_3^1 + b_4^2)\\
a_5^2 &= \sigma(w_{51}^2 a_1^1 + w_{52}^2 a_2^1 + w_{53}^2 a_3^1 + b_5^2)
\end{aligned} \tag{9}$$

or

$$\begin{pmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \\ a_4^2 \\ a_5^2 \end{pmatrix} = \sigma \left[\begin{pmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \\ w_{41}^2 & w_{42}^2 & w_{43}^2 \\ w_{51}^2 & w_{52}^2 & w_{53}^2 \end{pmatrix} \begin{pmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \end{pmatrix} + \begin{pmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \\ b_4^2 \\ b_5^2 \end{pmatrix} \right] \tag{10}$$

which is

$$\boldsymbol{a}^2 = \sigma(w^2 \boldsymbol{a}^1 + \boldsymbol{b}^2) \tag{11}$$

where $\boldsymbol{a}^2$ is the activation vector of the second layer, σ is an activation function, w^2 is the second layer weights matrix, $\boldsymbol{a}^1$ is the activation vector of the first layer, and $\boldsymbol{b}^2$ is the bias vector for the second layer. We can write an equation like this for every layer of the network. Generally, we have

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right) \tag{12}$$

and

$$\boldsymbol{a}^l = \sigma(w^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l) \tag{13}$$

where $\boldsymbol{a}^l$ is the activation vector of the l-th layer with components a_j^l, w^l is the weights matrix for layer l, and $\boldsymbol{b}^l$ is the bias vector for the l-th layer with components b_j^l.

It is also common to define

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{14}$$

so that

$$a_j^l = \sigma(z_j^l) \tag{15}$$

and

$$\boldsymbol{a}^l = \sigma(\boldsymbol{z}^l) \tag{16}$$

where $\boldsymbol{z}^l$ is the vector with components z_j^l.

With known inputs, and weights matrices between the layers, it is possible to calculate the output from the network using equation (12) (or equivalently equation (13)). The first hidden layer activations are computed from the inputs and first weights matrix. These activations and the second weights matrix are used to compute the second hidden layer activations, and so on until the output is generated. Considering Figure 6 again, data enters the network via the input layer and propagates layer by layer from left to right until the output is produced. This process is called forward propagation.

7 Cost Functions

To quantify the performance of the neural network we employ a cost function. An example would be the Mean Squared Error (MSE)(multiplied by a factor) which is

$$C(w,b) = \frac{1}{2m}\sum_{x_i} ||\boldsymbol{y} - \boldsymbol{a}^L||^2 \tag{17}$$

where w represents all the weights in the network, b all the biases, m is the total number of training samples, $\boldsymbol{a}^L$ is the vector of outputs from a network of L layers when training sample x_i is the input, and $\boldsymbol{y}$ is the corresponding vector of the desired output from the network. It is obvious that C will take on smaller values as $\boldsymbol{a}^L$ gets closer in value to $\boldsymbol{y}$. Hence, a smaller cost function value indicates a better performing network. The MSE cost function is typically used in linear regression problems in which we are modelling the relationship between a continuous dependent variable and one or more independent variables. In this case $\boldsymbol{a}^L$ and $\boldsymbol{y}$ will both be scalar values.

For classification problems, it is more common to use the cross-entropy cost function (see Appendix). For a single neuron with output a, the cross-entropy cost function is

$$C = -\frac{1}{m}\sum_{x_i}[y \ln a + (1-y)\ln(1-a)] \tag{18}$$

where y is the desired output from the neuron, x_i is the training sample, and m is the total number of training samples (for a more detailed discussion and derivation see Logistic Regression). For a particular training sample, let's say that $y = 1$. In this case the second term of the cost function will vanish and we are left with $C = -\ln a$, which is zero when $a = 1$, i.e., when the output of a and y are identical. Similarly, if $y = 0$, the first term vanishes and we are left with $-\ln(1-a)$. If the output of a is identical to y, i.e., if $a = 0$, then the cost function is zero. We can see that the cross-entropy cost function will be zero whenever a and y are identical and will increase as the discrepancy between a and y increases.

For a multi-layer network of numerous neurons the cross-entropy cost function is

$$C = -\frac{1}{m}\sum_{x_i}\sum_{j}[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)] \tag{19}$$

where y_j is the desired output from neuron j in the final layer, and a_j^L is the actual output from the same neuron. Notice how we are now also summing over all j output neurons.

Both the MSE and cross-entropy cost functions are convex, meaning they possess a global minimum (see Linear Regression and Logistic Regression). Consequently, we can employ gradient descent to minimise the cost functions. We simultaneously apply the updates

$$w_p \to w_p - \eta\frac{\partial C}{\partial w_p}, \qquad b_q \to b_q - \eta\frac{\partial C}{\partial b_q} \tag{20}$$

in terms of the weights w_p, and biases b_q in the network, where η is the learning rate.

7.1 The Regularised Cross-Entropy Cost Function

The (ridge) regularised (see Linear Regression for a discussion on regularisation) cross-entropy cost function is

$$C = -\frac{1}{m}\sum_{x_i}\sum_{j}[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)] + \frac{\lambda}{2m}\sum_{p} w_p^2 \tag{21}$$

If we denote the unregularised cross-entropy cost function (equation (19)) as C_0, we can write equation (21) as

$$C = C_0 + \frac{\lambda}{2m}\sum_{p} w_p^2 \tag{22}$$

Substituting this into (20) we see that to undertake gradient descent we simultaneously apply the updates

$$w_p \to w_p - \eta\frac{\partial C_0}{\partial w_p} - \frac{\eta\lambda w_p}{m} = \left(1-\frac{\eta\lambda}{m}\right)w_p - \frac{\partial C_0}{\partial w_p}, \qquad b_q \to b_q - \eta\frac{\partial C_0}{\partial b_q} \tag{23}$$

for all p and q.

8 The Theory and Equations of Backpropagation

In the last section we saw that we can utilise a gradient descent algorithm to fine-tune the weights and biases of the neural network to minimise its cost function. This is how the network ”learns”. The gradient of the cost function is given by

$$\boldsymbol{\nabla} C = \left(\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \frac{\partial C}{\partial w_3}, \cdots, \frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial b_2}, \frac{\partial C}{\partial b_3}, \cdots\right) \tag{24}$$

where the w_p and b_q are the weights and biases in the neural network, respectively. Backpropagation is the algorithm employed to compute the gradient of the cost function required for gradient descent. Therefore, backpropagation will compute the partial derivatives

$$\frac{\partial C}{\partial w_p}, \quad \frac{\partial C}{\partial b_q} \quad \text{for all } p \text{ and } q \tag{25}$$

For backpropagation to be correctly implemented, a necessary assumption is that the cost function, C, can be written as an average over cost functions for individual training samples, C_{x_i}, i.e.,

$$C = \frac{1}{m}\sum_{x_i} C_{x_1}$$

This is because backpropagation computes the partial derivatives $\partial C_{x_i}/\partial w_p$ and $\partial C_{x_i}/\partial b_q$ for a single training sample. We can recover $\partial C/\partial w_p$ and $\partial C/\partial b_q$ by averaging over all the training samples.

We begin by defining δ_j^L as the error in the j-th neuron in the l-th layer. Backpropagation gives a procedure to calculate δ_j^l and relate it to $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$.

Let's say that for the j-th neuron of the l-th layer, we add Δz_j^l to the weighted input, so that instead of computing the activation as $\sigma(z_j^l)$, we compute $\sigma(z_j^l + \Delta z_j^l)$. This change propagates through the layers of the network, causing the cost function to change by an amount $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$.

If $\partial C/\partial z_j^l$ has a large value, then the cost function can be lowered by choosing Δz_j^l to have the opposite sign to $\partial C/\partial z_j^l$. If $\partial C/\partial z_j^l$ is very small, then it is not possible to significantly lower the cost function, and we can conclude that the weighted input of the neuron must already be close to optimal. Therefore, we deduce that the error of the j-th neuron in the l-th layer is equal to $\partial C/\partial z_j^l$, i.e.,

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{26}$$

Using the chain rule we can write

$$\frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L} \tag{27}$$

Of course, $\frac{\partial a_k^L}{\partial z_j^L} = 0$ unless $j = k$, so this simplifies to

$$\begin{aligned} \frac{\partial C}{\partial z_j^L} &= \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} && (28) \\ &= \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) && (29) \end{aligned}$$

where we have used equation (15) to get $\frac{\partial a_j^L}{\partial z_j^L} = \sigma'(z_j^L)$. Therefore, we can write the error of the j-th neuron in the output layer as

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{30}$$

If we define $\boldsymbol{\delta}^L$ as the vector with the components δ_j^L, and $\boldsymbol{\nabla}_a C$ as the vector with the components $\frac{\partial C}{\partial a_j^L}$, then from equation (28) it follows that

$$\boldsymbol{\delta}^L = \boldsymbol{\nabla}_a C \odot \sigma'(\boldsymbol{z}^L) \tag{31}$$

where for two vectors of identical dimensions the Hadamard product produces another vector of the same dimensions which has as it elements products of the elements of the original two vectors. For example,

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \odot \begin{pmatrix} d \\ e \\ f \end{pmatrix} = \begin{pmatrix} ad \\ be \\ cf \end{pmatrix} \tag{32}$$

We can obtain the components (30) by performing the following multiplication:

$$\begin{pmatrix} \delta_1^L \\ \delta_2^L \\ \delta_3^L \\ \vdots \\ \delta_n^L \end{pmatrix} = \begin{pmatrix} \sigma'(z_1^L) & 0 & 0 & \cdots & 0 \\ 0 & \sigma'(z_2^L) & 0 & \cdots & 0 \\ 0 & 0 & \sigma'(z_3^L) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \sigma'(z_n^L) \end{pmatrix} \begin{pmatrix} \frac{\partial C}{\partial a_1^L} \\ \frac{\partial C}{\partial a_2^L} \\ \frac{\partial C}{\partial a_3^L} \\ \vdots \\ \frac{\partial C}{\partial a_n^L} \end{pmatrix} \tag{33}$$

$$= \begin{pmatrix} \frac{\partial C}{\partial a_1^L}\sigma'(z_1^L) \\ \frac{\partial C}{\partial a_2^L}\sigma'(z_2^L) \\ \frac{\partial C}{\partial a_3^L}\sigma'(z_3^L) \\ \vdots \\ \frac{\partial C}{\partial a_n^L}\sigma'(z_n^L) \end{pmatrix} \tag{34}$$

Therefore, an alternate form of equation (31) is

$$\boldsymbol{\delta}^L = M_L \boldsymbol{\nabla}_a C \tag{35}$$

where M_L is a square, diagonal matrix with the diagonal elements $\sigma'(z_j^L)$.

Having found an equation to compute the error in the output layer, we now look to find an equation that relates the errors between adjacent layers in the network, i.e., relates $\boldsymbol{\delta}^l$ and $\boldsymbol{\delta}^{l+1}$.

Using the chain rule we can write

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{36}$$

$$= \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{37}$$

Using equation (14) we have

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} \tag{38}$$

$$= \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1} \tag{39}$$

Differentiating with respect to z_j^l we get

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\sigma'(z_j^l) \tag{40}$$

Substituting into equation (37) we obtain

$$\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma'(z_j^l) \tag{41}$$

Returning to the network represented in Figure 6, we would have as the errors of the neurons in the second layer, for example

$$\begin{aligned}
\delta_1^2 &= w_{11}^3\delta_1^3\sigma'(z_1^2) + w_{21}^3\delta_2^3\sigma'(z_1^2) + w_{31}^3\delta_3^3\sigma'(z_1^2) + w_{41}^3\delta_4^3\sigma'(z_1^2) + w_{51}^3\delta_5^3\sigma'(z_1^2)\\
\delta_2^2 &= w_{12}^3\delta_1^3\sigma'(z_2^2) + w_{22}^3\delta_2^3\sigma'(z_2^2) + w_{32}^3\delta_3^3\sigma'(z_2^2) + w_{42}^3\delta_4^3\sigma'(z_2^2) + w_{52}^3\delta_5^3\sigma'(z_2^2)\\
\delta_3^2 &= w_{13}^3\delta_1^3\sigma'(z_3^2) + w_{23}^3\delta_2^3\sigma'(z_3^2) + w_{33}^3\delta_3^3\sigma'(z_3^2) + w_{43}^3\delta_4^3\sigma'(z_3^2) + w_{53}^3\delta_5^3\sigma'(z_3^2)\\
\delta_4^2 &= w_{14}^3\delta_1^3\sigma'(z_4^2) + w_{24}^3\delta_2^3\sigma'(z_4^2) + w_{34}^3\delta_3^3\sigma'(z_4^2) + w_{44}^3\delta_4^3\sigma'(z_4^2) + w_{54}^3\delta_5^3\sigma'(z_4^2)\\
\delta_5^2 &= w_{15}^3\delta_1^3\sigma'(z_5^2) + w_{25}^3\delta_2^3\sigma'(z_5^2) + w_{35}^3\delta_3^3\sigma'(z_5^2) + w_{45}^3\delta_4^3\sigma'(z_5^2) + w_{55}^3\delta_5^3\sigma'(z_5^2)
\end{aligned} \tag{42}$$

which can be expressed as

$$\begin{pmatrix}\delta_1^2\\ \delta_2^2\\ \delta_3^2\\ \delta_4^2\\ \delta_5^2\end{pmatrix} = \begin{pmatrix} w_{11}^3 & w_{21}^3 & w_{31}^3 & w_{41}^3 & w_{51}^3\\ w_{12}^3 & w_{22}^3 & w_{32}^3 & w_{42}^3 & w_{52}^3\\ w_{13}^3 & w_{23}^3 & w_{33}^3 & w_{43}^3 & w_{53}^3\\ w_{14}^3 & w_{24}^3 & w_{34}^3 & w_{44}^3 & w_{54}^3\\ w_{15}^3 & w_{25}^3 & w_{35}^3 & w_{45}^3 & w_{55}^3 \end{pmatrix} \begin{pmatrix}\delta_1^3\\ \delta_2^3\\ \delta_3^3\\ \delta_4^3\\ \delta_5^3\end{pmatrix} \odot \begin{pmatrix}\sigma'(z_1^2)\\ \sigma'(z_2^2)\\ \sigma'(z_3^2)\\ \sigma'(z_4^2)\\ \sigma'(z_5^2)\end{pmatrix} \tag{43}$$

or equivalently

$$\boldsymbol{\delta}^2 = [(w^3)^T\boldsymbol{\delta}^3]\odot\sigma'(\boldsymbol{z}^2) \tag{44}$$

More generally, this is

$$\boldsymbol{\delta}^l = [(w^{l+1})^T\boldsymbol{\delta}^{l+1}]\odot\sigma'(\boldsymbol{z}^l) \tag{45}$$

We can express equation (43) in a different form:

$$\begin{aligned}
\begin{pmatrix}\delta_1^2\\ \delta_2^2\\ \delta_3^2\\ \delta_4^2\\ \delta_5^2\end{pmatrix} &= \begin{pmatrix}\sigma'(z_1^2) & 0 & 0 & 0 & 0\\ 0 & \sigma'(z_2^2) & 0 & 0 & 0\\ 0 & 0 & \sigma'(z_3^2) & 0 & 0\\ 0 & 0 & 0 & \sigma'(z_4^2) & 0\\ 0 & 0 & 0 & 0 & \sigma'(z_5^2)\end{pmatrix} \begin{pmatrix} w_{11}^3 & w_{21}^3 & w_{31}^3 & w_{41}^3 & w_{51}^3\\ w_{12}^3 & w_{22}^3 & w_{32}^3 & w_{42}^3 & w_{52}^3\\ w_{13}^3 & w_{23}^3 & w_{33}^3 & w_{43}^3 & w_{53}^3\\ w_{14}^3 & w_{24}^3 & w_{34}^3 & w_{44}^3 & w_{54}^3\\ w_{15}^3 & w_{25}^3 & w_{35}^3 & w_{45}^3 & w_{55}^3 \end{pmatrix} \begin{pmatrix}\delta_1^3\\ \delta_2^3\\ \delta_3^3\\ \delta_4^3\\ \delta_5^3\end{pmatrix}\\
&= \begin{pmatrix}\sigma'(z_1^2) & 0 & 0 & 0 & 0\\ 0 & \sigma'(z_2^2) & 0 & 0 & 0\\ 0 & 0 & \sigma'(z_3^2) & 0 & 0\\ 0 & 0 & 0 & \sigma'(z_4^2) & 0\\ 0 & 0 & 0 & 0 & \sigma'(z_5^2)\end{pmatrix} \begin{pmatrix} w_{11}^3\delta_1^3 & w_{21}^3\delta_2^3 & w_{31}^3\delta_3^3 & w_{41}^3\delta_4^3 & w_{51}^3\delta_5^3\\ w_{12}^3\delta_1^3 & w_{22}^3\delta_2^3 & w_{32}^3\delta_3^3 & w_{42}^3\delta_4^3 & w_{52}^3\delta_5^3\\ w_{13}^3\delta_1^3 & w_{23}^3\delta_2^3 & w_{33}^3\delta_3^3 & w_{43}^3\delta_4^3 & w_{53}^3\delta_5^3\\ w_{14}^3\delta_1^3 & w_{24}^3\delta_2^3 & w_{34}^3\delta_3^3 & w_{44}^3\delta_4^3 & w_{54}^3\delta_5^3\\ w_{15}^3\delta_1^3 & w_{25}^3\delta_2^3 & w_{35}^3\delta_3^3 & w_{45}^3\delta_4^3 & w_{55}^3\delta_5^3 \end{pmatrix}
\end{aligned}$$

Carrying out the multiplication we regain the components in (42). So, we have

$$\boldsymbol{\delta}^2 = M_2(w^3)^T\boldsymbol{\delta}^3 \tag{46}$$

where M_2 is the square, diagonal matrix with the diagonal elements $\sigma'(z_j^2)$ Therefore, an alternate form of equation (45) is

$$\boldsymbol{\delta}^l = M_l(w^{l+1})^T\boldsymbol{\delta}^{l+1} \tag{47}$$

where M_l is a square, diagonal matrix with the diagonal elements $\sigma'(z_j^l)$.

This is our equation for relating the error between adjacent layers in the network. We calculate the error in the output layer with equation (31) (or (35)) and then utilise equation (45) (or (47)) to compute the error in the final hidden layer, and then the layer before that one, and so on.

In order to undertake gradient descent we require equations that enable us to compute the partial derivatives of the cost function with respect to the weights and biases in the networks in terms of quantities that we know. The rate of change of the cost function with respect to the bias b_j^l is

$$\begin{align}
\frac{\partial C}{\partial b_j^l} &= \sum_k \frac{\partial C}{\partial z_k^l}\frac{\partial z_k^l}{\partial b_j^l} \tag{48}\\
&= \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \tag{49}\\
&= \delta_j^l \frac{\partial}{\partial b_j^l}\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{50}\\
&= \delta_j^l \tag{51}
\end{align}$$

The rate of change of the cost function with respect to the weight w_{jk}^l is

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_i \frac{\partial C}{\partial z_i^l}\frac{\partial z_i^l}{\partial w_{jk}^l} \tag{52}$$

From equation (14) we can see that $\frac{\partial z_i^l}{\partial w_{jk}^l} = 0$ unless $i = j$. Therefore

$$\begin{align}
\frac{\partial C}{\partial w_{jk}^l} &= \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \tag{53}\\
&= \delta_j^l \frac{\partial}{\partial w_{jk}^l}\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{54}\\
&= \delta_j^l a_k^{l-1} \tag{55}
\end{align}$$

Hence, the equations (51) and (55) enable us to compute the partial derivatives of the cost function in terms of known quantities.

9 The Backpropagation Algorithm

1. **Input sample $\boldsymbol{x_i}$**: Set the corresponding activation $\boldsymbol{a}^1$ for the input layer.
2. **Feedforward**: For each $l = 2, 3, ..., L$ compute $\boldsymbol{z}^l = w^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$, and $\boldsymbol{a}^l = \sigma(\boldsymbol{z}^l)$, until eventually we obtain the vector of outputs $\boldsymbol{a}^L$.
3. **Output error $\boldsymbol{\delta}^L$**: Compute $\boldsymbol{\delta}^L = \boldsymbol{\nabla}_a C \odot \sigma'(\boldsymbol{z}^L)$, the vector of errors in the final layer neurons.
4. **Backpropagate the error**: For each $l = L-1, L-2, ..., 2$ compute $\boldsymbol{\delta}^l = [(w^{l+1})^T \boldsymbol{\delta}^{l+1}] \odot \sigma'(\boldsymbol{z}^l)$, the vector of errors in the neurons of the l-th layer.
5. **Output**: The components of the gradient of the cost function are $\frac{\partial C}{\partial b^l_j} = \delta^l_j$ and $\frac{\partial C}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k$.

The algorithm is named backpropagation because first the error in the output layer, $\boldsymbol{\delta}^L$, is computed, before the errors in the other layers are calculated in reverse order, i.e., $\boldsymbol{\delta}^{L-1}, \boldsymbol{\delta}^{L-2}, ..., \boldsymbol{\delta}^2$. The backpropagation algorithm above is for a single training sample, x_i. To compute the gradient of the cost function, $\boldsymbol{\nabla} C$, we would need to calculate the gradient $\boldsymbol{\nabla} C_{x_i}$ for each sample and take the mean:

$$\boldsymbol{\nabla} C = \frac{1}{m} \sum_{x_i} \boldsymbol{\nabla} C_{x_i} \tag{56}$$

where as before m is the total number of training samples. For very large datasets this is not a practical approach. A method to resolve the issue is stochastic gradient descent, which entails the random selection of a subset of the training samples, say $X_1, X_2, ..., X_N$. Provided that the number of samples in the subset, N, is large enough, we would expect the mean of the gradients of the subset samples, $\boldsymbol{\nabla} C_{X_n}$, to be roughly equal to the mean of all the samples, i.e.,

$$\boldsymbol{\nabla} C = \frac{1}{m} \sum_{x_i} \boldsymbol{\nabla} C_{x_i} \approx \frac{1}{N} \sum_{X_n} \boldsymbol{\nabla} C_{X_n} \tag{57}$$

Therefore, we simultaneously update the weights and biases for each $l = L-1, L-2, ..., 2$ as follows:

$$w^l_{jk} \to w^l_{jk} - \frac{\eta}{N} \sum_{X_n} \frac{\partial C_{X_n}}{\partial w^l_{jk}}, \quad b^l_j \to b^l_j - \frac{\eta}{N} \sum_{X_n} \frac{\partial C_{x_n}}{\partial b^l_j}$$

After gradient descent, the next iteration of the algorithm begins from step 2; forward propagation is undertaken with the updated weights and biases.

Reference

Nielsen, M. (2015). *Neural Networks and Deep Learning.* Determination Press.

Appendix

Why do we prefer the Cross-Entropy Cost Function to the MSE for Classification Problems?

To keep things as simple as possible we will consider a single neuron with one input, x. The weight applied to the input is w, and the bias b, so that $z = wx + b$. The output of the neuron is $a = \sigma(z)$. For a single training sample the MSE (multiplied by a scale factor) is

$$C = \frac{1}{2}[y - \sigma(z)]^2 \tag{58}$$

The partial derivatives with respect to the weight and the bias are

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial \sigma}\frac{\partial \sigma}{\partial z}\frac{\partial z}{\partial w} = [\sigma(z) - y]\sigma'(z)x \tag{59}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial \sigma}\frac{\partial \sigma}{\partial z}\frac{\partial z}{\partial b} = [\sigma(z) - y]\sigma'(z) \tag{60}$$

Both of the partial derivatives depend on the derivative of the neuron's sigmoid output, $\sigma'(z)$. Viewing Figure 2, we can see that as the sigmoid output approaches zero or one, the derivative becomes increasingly smaller. This leads to what is known as a learning slowdown. This is because the network learns by updating the weights and biases through gradient descent, which makes a change to a weight or bias by an amount proportional to $\partial C/\partial w$ or $\partial C/\partial b$, respectively. For example, if the desired output is $y = 1$ and our initial neuron output is $\sigma(z) \approx 0$, then learning will initially be very slow until $\sigma(z)$ gets larger and the learning speed increases.

The cross-entropy cost function for the same network of one neuron with a single input is

$$C = -[y \ln \sigma(z) + (1 - y) \ln(1 - \sigma(z)] \tag{61}$$

The partial derivative of the cost function with respect to the weight is

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial \sigma}\frac{\partial \sigma}{\partial z}\frac{\partial z}{\partial w} \tag{62}$$

$$= -\left[\frac{y}{\sigma(z)} - (1 - y)\frac{1}{1 - \sigma(z)}\right]\sigma'(z)x \tag{63}$$

$$= -\left\{\frac{y[1 - \sigma(z)] - (1 - y)\sigma(z)}{\sigma(z)[1 - \sigma(z)]}\right\}\sigma'(z)x \tag{64}$$

$$= \frac{\sigma(z) - y}{\sigma(z)[1 - \sigma(z)]}\sigma'(z)x \tag{65}$$

The sigmoid neuron output is

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

Therefore,

$$\sigma'(z) = (-1)(1 + e^{-z})^{-2}(-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2} \tag{66}$$

Performing some algebra, we find that

$$\sigma'(z) = \sigma(z)\frac{e^{-z}}{1+e^{-z}} \tag{67}$$

$$= \sigma(z)\left[1 - \frac{1}{1+e^{-z}}\right] \tag{68}$$

$$= \sigma(z)[1-\sigma(z)] \tag{69}$$

Substituting back into equation (65) we obtain

$$\frac{\partial C}{\partial w} = \frac{\sigma(z)[1-\sigma(z)]x}{\sigma(z)[1-\sigma(z)]}[\sigma(z) - y] \tag{70}$$

$$= x[\sigma(z) - y] \tag{71}$$

The partial derivative of the cost function with respect to the bias is

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial \sigma}\frac{\partial \sigma}{\partial z}\frac{\partial z}{\partial b} \tag{72}$$

$$= \sigma(z) - y \tag{73}$$

Equations (71) and (73) are identical to (59) and (60), apart from the the dependence on $\sigma'(z)$ which is not present in the former two. Consequently as the partial derivatives of the cross-entropy cost function have no dependence on the gradient of the sigmoid output, we will not suffer from the learning slowdown that is particularly evident with the MSE when we have values of $\sigma(z)$ close to one or zero. Instead the partial derivatives of the cross-entropy cost function depend on the difference between the neuron output and the desired output. This means that the partial derivatives will be greatest when the error in the neuron output is largest, i.e., the larger the neuron error, the faster it will learn.

Why do we prefer to use the ReLU activation in the hidden layers?

We saw in the last section that by using the cross-entropy cost function we can rid $\partial C/\partial w$ and $\partial C/\partial b$ of any dependence on $\sigma'(z)$. Failure to do so can result in a learning slowdown. However, if we consider a network and not just a single neuron, it is clear that the findings of the last section are only applicable to the output layer. Looking back at the equations of backpropagation, we see that the computations of the partial derivatives of C with respect to the weights and biases in the hidden layers have a dependence on $\sigma'(z)$. Therefore, if we have sigmoid neurons in these layers we could still suffer from a learning slowdown as the activations approach zero or one. If the hidden layers are utilising the ReLU activation function, there is no vanishing gradient problem because for $\mathbf{w}\cdot\mathbf{x} + b > 0$, the derivative of the ReLU function is one. We still choose to use the sigmoid activation function for the output layer for binary classification problems (or a softmax output layer for multi-class problems) because it outputs a value between zero and one which predicts the class of the label.

We mentioned earlier that the property of being able to output a zero is considered an advantage of the ReLU activation function. However, if too many neurons in the neural network are outputting zero then optimisation of the weights becomes challenging. This is because during backpropagation the gradient of the activation function is required to compute the partial derivatives of C with respect to the weights and biases, and if a ReLU activation is zero, its derivative is also zero, and gradient descent cannot make updates to the associated weights and biases. This can result in a large part of the network becoming inactive and is known as the "dying ReLU problem".

Additional Insight into Backpropagation

The goal of backpropagation is to obtain the partial derivatives $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$ required for gradient descent. If we make a small change to the weight in the network w^l_{jk}, the activation a^l_j will change by the amount

$$\Delta a^l_j \approx \frac{\partial a^l_j}{\partial w^j_{jk}} \Delta w^l_{jk} \tag{74}$$

This change in the activation, Δa^l_j will, in turn, cause a change in the activation of a neuron a^{l+1}_p in the next layer according to

$$\Delta a^{l+1}_p \approx \frac{\partial a^{l+1}_p}{\partial a^l_j} \Delta a^l_j \tag{75}$$

If the path goes through the network via the activations $a^l_j, a^{l+1}_p, a^{l+2}_q, ..., a^{L-1}_s, a^L_t$ then we can relate the change in the cost due to the change in the weight w^l_{jk} as

$$\begin{align} \Delta C &\approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk} \tag{76} \\ &\approx \frac{\partial C}{\partial a^L_t} \frac{\partial a^L_t}{\partial a^{L-1}_s} \frac{\partial a^{L-1}_s}{\partial a^{L-2}_r} \cdots \frac{\partial a^{l+2}_q}{\partial a^{l+1}_p} \frac{\partial a^{l+1}_p}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^j_{jk}} \Delta w^l_{jk} \tag{77} \end{align}$$

As we have considered only a single path through the network, this expression represents the change in C due to the changes in activations along this particular path. For the total change in C we should sum over all possible paths in the network. This is

$$\Delta C \approx \sum_{tsr...qp} \frac{\partial C}{\partial a^L_t} \frac{\partial a^L_t}{\partial a^{L-1}_s} \frac{\partial a^{L-1}_s}{\partial a^{L-2}_r} \cdots \frac{\partial a^{l+2}_q}{\partial a^{l+1}_p} \frac{\partial a^{l+1}_p}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^j_{jk}} \Delta w^l_{jk} \tag{78}$$

From which it follows that

$$\frac{\partial C}{\partial w^l_{jk}} = \sum_{tsr...qp} \frac{\partial C}{\partial a^L_t} \frac{\partial a^L_t}{\partial a^{L-1}_s} \frac{\partial a^{L-1}_s}{\partial a^{L-2}_r} \cdots \frac{\partial a^{l+2}_q}{\partial a^{l+1}_p} \frac{\partial a^{l+1}_p}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^j_{jk}} \tag{79}$$

This equation demonstrates how a change in the weight w^l_{jk} propagates layer by layer through the network. In backpropagation we start by finding the rate of change of the cost function with respect to the output (the first partial derivative in the sum above) and then work backwards layer by layer, computing all the terms in the sum from left to right until we obtain $\partial C/\partial w^j_{jk}$.