Contents

Chapter 1

Basic Algorithm

1.1 Notation

Consider a dense network with L layers, with L^{th} being the output layer with K neurons:

> **t_k:** Target output of the k^{th} neuron of the output layer.
>
> **c_k:** Assumed cost of the output error of k^{th} neuron of the output layer.
>
> **c:** Assumed cost of the output errors across all neurons of the output layer.

Consider l^{th} layer having I neurons and preceding layer, $l-1^{th}$, having J neurons. For the i^{th} neuron on this layer:

> **a_i^l:** Output
>
> **w_{ij}^l:** Weight applied to a_j^{l-1}
>
> **b_i^l:** Bias
>
> **z_i^l:** Sum of products
>
> **σ^l:** Activation function, generally the same for all i, and often for all l as well.
>
> **λ:** Cost function that takes in the vector $\mathbf{a}^L$ and output the scalar cost c.
>
> **s_i^l:** Sensitivity (or error gradient) of c w.r.t. z_i^l.

Notes:

> *Lower case bold symbols are used for vectors while uppercase bold symbols are used for matrices. Appropriate indices are dropped in these cases.*
>
> *The weights' subscript order is used both ways in texts. Above is chosen partly for C++ array calculations to be a little faster.*

1.2 Forward Pass

$$z_i^l = \sum^{J} w_{ij}^l a_j^{l-1} + b_i^l \tag{1.1}$$

$$a_i^l = \sigma^l(z_i^l) \tag{1.2}$$

In matrix notation:

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \tag{1.3}$$

$$\mathbf{a}^l = \sigma^l(\mathbf{z}^l) \tag{1.4}$$

From (1.1), we have:

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = a_j^{l-1} \tag{1.5}$$

and

$$\frac{\partial z_i^l}{\partial a_j^{l-1}} = w_{ij}^l \tag{1.6}$$

We will use a common activation function, the *logistic function*:

$$logistic(x) = \frac{L}{1+e^{k(x_0-x)}} \tag{1.7}$$

where L is the maximum, x_0 is the midpoint and k is the growth rate at the midpoint. Usually we set $L = 1$, $x_0 = 0$ and $k = 1$ to get:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z} \tag{1.8}$$

with

$$\sigma'(z) = \frac{\partial \sigma}{\partial z} = \frac{e^x}{(1+e^x)^2} = \sigma(z)(1-\sigma(z)) \tag{1.9}$$

When applied to neuron i in layer l, substituting (1.2):

$$\sigma'(z_i^l) = a_i^l(1-a_i^l) \tag{1.10}$$

1.3 Cost Function

The **prediction error** at the output layer is $t_k - a_k^L$. The cost of the error is given by a cost function. The simplest is the *quadratic cost function* and we will use it in the rest of this chapter. It is defined as:

$$c_k = (t_k - a_k^L)^2 \tag{1.11}$$

Note:

$$\frac{\partial c_k}{\partial a_k^L} = -2(t_k - a_k^L) = 2(a_k^L - t_k) \tag{1.12}$$

In case of multiple outputs we need to represent the net cost by a scalar value so that we can attempt to minimize it by adjusting network weights. In case of quadratic cost we use:

$$c = (\mathbf{t} - \mathbf{a}^L)^T(\mathbf{t} - \mathbf{a}^L) \tag{1.13}$$

or

$$c = \sum_{k=1}^{K}(t_k - a_k^L)^2 \tag{1.14}$$

and still we have:

$$\frac{\partial c}{\partial a_k^L} = -2(t_k - a_k^L) = 2(a_k^L - t_k) \tag{1.15}$$

Note that this is independent of all $a_{n \neq k}$ and $t_{n \neq k}$ only because the way quadratic cost for multiple output case is defined and it is not the general for a cost function.

1.4 Backpropagation

1.4.1 Error Gradients

By definition:

$$s_i^l = \frac{\partial c}{\partial z_i^l} \tag{1.16}$$

By simple **chain rule**:

$$s_i^l = \frac{\partial c}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l} \tag{1.17}$$

For a hidden layer $l-1$ with J neurons:

$$s_j^{l-1} = \frac{\partial c}{\partial a_j^{l-1}} \frac{\partial a_j^{l-1}}{\partial z_j^{l-1}} \tag{1.18}$$

$c = f(z_1^l, z_2^l, ..., z_I^l,)$ and each $z_i^l = g_{ij}(a_j^{l-1})$ for some f_k and g_{ij}. c depends on a_j^{l-1} only via these functions. Hence, by **chain rule for multiple variables**:

$$\frac{\partial c}{\partial a_j^{l-1}} = \sum_{i=1}^{I} \frac{\partial c}{\partial z_i^l} \frac{\partial z_i^l}{\partial a_j^{l-1}} \tag{1.19}$$

Substituting (1.16) and (1.6):

$$\frac{\partial c}{\partial a_j^{l-1}} = \sum_{i=1}^{I} s_i^l w_{ij}^l \tag{1.20}$$

If we consider j^{th} column of the weight matrix we notice that we can write this as:

$$\frac{\partial c}{\partial a_j^{l-1}} = \mathbf{w}_j^{l\,T} \mathbf{s}^l \tag{1.21}$$

Substituting (1.2) and (1.20) in (1.18):

$$s_j^{l-1} = \sigma'(z_j^{l-1}) \sum_{i=1}^{I} s_i^l \; w_{ij}^l \tag{1.22}$$

In matrix notation:

$$\mathbf{s}^{l-1} = \sigma'(\mathbf{z}^{l-1}) \; \odot (\mathbf{W}^{l^T} \mathbf{s}^l) \tag{1.23}$$

where $\odot$ is the Hadamard product or element-wise product.

With our chosen activation function, using (1.10)

$$s_j^{l-1} = a_j^{l-1}(1 - a_j^{l-1}) \sum_{i=1}^{I} s_i^l \; w_{ij}^l \tag{1.24}$$

for hidden layers. And for output layer, with our chosen cost function, substituting (1.15) and (1.10) in (1.17):

$$s^L = 2(a_k^L - t_k)a_k^L(1 - a_k^L) \tag{1.25}$$

Last two equations in matrix form:

$$\mathbf{s}^{l-1} = (\mathbf{W}^{l^T} s^l) \odot \mathbf{a}^{l-1} \odot (1 - \mathbf{a}^{l-1}) \tag{1.26}$$

$$\mathbf{s}^L = 2(\mathbf{a}^L - \mathbf{t}) \odot \mathbf{a}^L \odot (1 - \mathbf{a}^L) \tag{1.27}$$

1.4.2 Gradient Descent

For any w_{ij}^l,

$$\frac{\partial c}{\partial w_{ij}^l} = \frac{\partial c}{\partial z_{ij}^l} \frac{\partial z_{ij}^l}{\partial w_{ij}^l} \tag{1.28}$$

Using (1.5) and (1.16):

$$\frac{\partial c}{\partial w_{ij}^l} = s_i^l a_j^{l-1} \tag{1.29}$$

For small changes of w_{ij}^l:

$$\Delta c = s_i^l a_j^{l-1} \Delta w \tag{1.30}$$

The goal of gradient descent is to slowly reduce the cost and thus the prediction error by making small changes to the weights during each pass. These changes are made proportional to the sensitivity of the cost to each weight so that the weights with the ability to affect the error are changed most. I.e.

$$w_{ij}^l(t+1) = w_{ij}^l(t) - \eta s_i^l a_j^{l-1} \tag{1.31}$$

where η is *learning rate* which is kept small to make gradient descent smooth but large enough to not to get trapped in a local minimum of c. In matrix notation:

$$\mathbf{W}^l(t+1) = \mathbf{W}^l(t) - \eta\, \mathbf{s}^l \mathbf{a}^{l-1^T} \tag{1.32}$$

1.5 Common Activation Functions

Most activation functions are of the form (1.2) and not (1.4). That is, each output depends only on the sum of that neuron. Common such activation functions are listed in table 1.1. An exception, *softmax*, is discussed in section 2.1.

1.6 Weight Initialization

For layer l, the number of inputs to one neuron, *fan in*, is $J = I^{l-1}$ and the number of outputs, fan out, is I^{l+1}.

	Function $a(z)$	Range	Derivative	Range
relu	$\begin{cases} 0 & z \leq 0 \\ z & z > 0 \end{cases}$	$[0, +\infty]$	$\begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases}$	$\{0, 1\}$
elu	$\begin{cases} \alpha(e^z - 1) & z < 0 \\ z & z \geq 0 \end{cases}$	$[-\infty, +\infty]$	$\begin{cases} \alpha e^z & z < 0 \\ 1 & z \geq 0 \end{cases}$	$[0, 1]$
logistic	$\frac{1}{1+e^{-z}}$ More generally, $\frac{L}{1+e^{k(z_0-z)}}$ with $L, k > 0$	$[0, 1]$	$\frac{e^{-z}}{(1+e^{-z})^2} = a(z)(1-a(z))$	$[0, 0.25]$
tanh	$\frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z}-1}{e^{2z}+1}$	$[-1, 1]$	$\frac{(e^z+e^{-z})^2 - (e^z - e^{-z})^2}{(e^z+e^{-z})^2} = (1 - a(z)^2)$	$[0, 1]$

Table 1.1: Common Activation Functions

1.6.1 He Initialization

This applies to relu and elu layers. Weights are initialized with values drawn from:

$$\mathcal{U}\left(-\frac{6}{\sqrt{I^{l-1}}}, \frac{6}{\sqrt{I^{l-1}}}\right) \tag{1.33}$$

1.6.2 Xavier Initialization

Tanh neurons are initialized from:

$$\mathcal{U}\left(-\sqrt{\frac{6}{I^{l-1}+I^{l+1}}}, \sqrt{\frac{6}{I^{l-1}+I^{l+1}}}\right) \tag{1.34}$$

And logistic neurons from:

$$\mathcal{U}\left(-4\sqrt{\frac{6}{I^{l-1}+I^{l+1}}}, 4\sqrt{\frac{6}{I^{l-1}+I^{l+1}}}\right) \tag{1.35}$$

Chapter 2

Selected Topics

2.1 Softmax Activation Function

2.1.1 Softmax Definition and Derivative

Softmax operates on a vector $\mathbf{z}$ to produce a vector $\mathbf{a}$. i^{th} element of the resulting vector is given by:

$$a_i = softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{p=1}^{I} e^{z_p}} \tag{2.1}$$

Note that here p is a dummy variable and not a part of our notation defined in section 1.1.

Commonly used in the output layer of multi-class classifiers, *softmax* attempt to predict the probability of the input being each of the classes. I.e. individual outputs are in range [0,1] and they sum to 1. For the sum to be 1, the outputs need to be normalized and thus, even to calculate one output, it needs weighted sums of all neurons in the layer. Hence a vector function.

Using **quotient rule** of differentiation:

$$\frac{\partial a_i}{\partial z_k} = \begin{cases} \dfrac{e^{z_i}\sum_{p=1}^{I} e^{z_p} - e^{z_i}e^{z_k}}{(\sum_{p=1}^{I} e^{z_p})^2} & i = k \\ -\dfrac{e^{z_i}e^{z_k}}{(\sum_{p=1}^{I} e^{z_p})^2} & i \neq k \end{cases} \tag{2.2}$$

or by substituting (2.1):

$$\frac{\partial a_i}{\partial z_k} = \begin{cases} a_i(1 - a_k) & i = k \\ -a_i a_k & i \neq k \end{cases} \tag{2.3}$$

using **Kronecker delta** we can simplify this to:

$$\frac{\partial a_i}{\partial z_k} = a_i(\delta_{ik} - a_k) \tag{2.4}$$

All such derivatives at level l forms the **Jacobian matrix** of the softmax vector function:

$$\mathbf{J}_{\mathbf{a}=softmax(\mathbf{z})} = \mathbf{J} \begin{pmatrix} a_1(1-a_1) & -a_1a_2 & \cdots & -a_1a_I \\ -a_2a_1 & a_2(1-a_2) & \cdots & a_2a_I \\ \vdots & \vdots & \ddots & \vdots \\ -a_Ia_1 & -a_Ia_2 & \cdots & a_I(1-a_I) \end{pmatrix} \tag{2.5}$$

which is a $I \times I$ symmetric matrix.

2.1.2 Softmax Forward Pass

When softmax is used as an activation function, (1.2) changes to:

$$a_i^l = \sigma^l(\mathbf{z}^l) \tag{2.6}$$

or with outputs in vector form as well,

$$\mathbf{a}^l = \boldsymbol{\sigma}^l(\mathbf{z}^l) \tag{2.7}$$

While this is the same as (1.4), the computation is different because each a_i depend on all z_i's. I.e. $\boldsymbol{\sigma}$ here is a vector function, not a scalar function which maps each value of the input vector to one value of the output vector. This is computationally more intensive compared to other activation functions.

2.1.3 Softmax Backpropagation

The vector nature of softmax complicates backpropagation more than the forward pass. Every z_i affect every c_k via each a_p^l. So, in general cost, c is affected by a z_i via multiple paths or intermediary variables and we need to apply multivariable chain rule to expand (1.16). Then instead of (1.17) we get:

$$s_i^l = \frac{\partial c}{\partial z_i^l} = \sum_{p=1}^{I} \frac{\partial c}{\partial a_p^l} \frac{\partial a_p^l}{\partial z_i^l} \tag{2.8}$$

As mentioned before, usually softmax is used in the final layer. In that case, we need to plug in the derivative of the cost function and (2.4) into (2.8) to get s_i^L. We will analyze a specific case in 2.4.

For layer $l-1$ other than the final layer, after changing subscripts and the order of terms:

$$s_j^{l-1} = \sum_{p=1}^{J} \frac{\partial a_p^{l-1}}{\partial z_j^{l-1}} \frac{\partial c}{\partial a_p^{l-1}} \tag{2.9}$$

(2.4) and (1.21) give first and second parts of the RHS we get:

$$s_j^{l-1} = \sum_{p=1}^{J} \left(a_p^{l-1}(\delta_{pj} - a_j^{l-1}) \mathbf{w}_p^{l\,T} \mathbf{s}^l \right) \tag{2.10}$$

Notice that for a given j we are using j_{th} row (or column) of (2.5). It is not hard to see that:

$$s_j^{l-1} = \mathbf{j}_j^{l-1} \mathbf{W}^{l^T} \mathbf{s}^l \tag{2.11}$$

where $\mathbf{j}_j$ denotes j_{th} row of (2.5). And the full sensitivity vector is given by:

$$\mathbf{s}^{l-1} = \mathbf{J}^{l-1} \mathbf{W}^{l^T} \mathbf{s}^l \tag{2.12}$$

2.2 Generalized Backpropagation Algorithm

In 2.1.3 we could have skipped (2.10) and instead used the general partial derivative of $\sigma(a_p^{l-l})$ w.r.t. z_j^{l-1} to get:

$$s_j^{l-1} = \sum_{p=1}^{J} \left(\sigma'_{z_k^{l-1}}(a_p^{l-l}) \mathbf{w}_p^{l\,T} \mathbf{s}^l \right) \tag{2.13}$$

and still continue on to (2.11) and (2.12). I.e. (2.12) is the general equation for backpropagation of sensitivities when $\mathbf{J}$ is the Jacobian of the activation function.

For example, if we consider $\frac{\partial a_i}{\partial z_k}$ of logistic function we notice that its Jacobian's none diagonal elements are zero while the diagonal is the same as that of softmax. So (2.12) is still valid, though (1.26) is computationally efficient in case of logistic activation.

When (2.8) is applied to the output layer:

$$s_i^L = \sum_{p=1}^{I} \sigma'_{a_p^L}(z_i^L)\lambda'_{a_p^L} \tag{2.14}$$

or in vector form

$$s_i^L = \mathbf{j}_i^L \boldsymbol{\lambda}' \tag{2.15}$$

where $\mathbf{j}_i^L$ is the i^{th} row of the Jacobian of the activation function and $\boldsymbol{\lambda}'$ is the vector of partial derivatives of λ w.r.t. each a_p^L. Or in matrix form:

$$\mathbf{s}^L = \mathbf{J}^L \boldsymbol{\lambda}' \tag{2.16}$$

In summary (2.12) and (2.16) along with (1.32) give general backpropagation algorithm.

2.3 Categorical Cross Entropy (CCE) Cost Function

In information theory, for a discrete random variable x, the **cross entropy** between two probability distributions $p(x)$ and $q(x)$ is:

$$H(p,q) = -\sum_x p(x)\log q(x) \tag{2.17}$$

When target output vector $\mathbf{t}$ and last layer output vector $\mathbf{a}$ each sum to 1 and each element is in the range $[0,1]$ then we can consider these to be probability distributions and use (2.17) as a cost function. Recall that this is indeed the case with softmax activation and the problem is a classification problem with $\mathbf{t}$ being all 0s except one element being a 1. In our familiar notation:

$$c = -\sum_k t_k \log a_k \tag{2.18}$$

By way of an explanation of why this is a good cost function, note that: a) $\log a_k$ is always -ve making each term positive. b) if t_k is large and a_k is small then we get a larger cost. According to Gibbs' inequality $\mathbf{a} \equiv \mathbf{t}$ minimizes c for a given $\mathbf{t}$.

Notice

$$\frac{\partial c}{\partial a_k^L} = -\frac{t_k}{a_k} \tag{2.19}$$

In classification problems $t_q = 1$ for some q and $t_{k\neq q} = 0$, only $\frac{\partial c}{\partial a_q^L}$ will be non-zero.

When implemented as a computer program, if a_k is very small or zero, (2.19) can yield **inf** or **NaN** and cause problems. As shown in next section, 2.4, if the derivative of the activation function cancel out a_k then the problem is solved.

2.4 CCE with Softmax Output Layer

If our output layer has softmax activation then we can use (2.19) and (2.4) in (2.8) while paying attention to subscripts to get:

$$s_i^L = \sum_{p=1}^{I} -\frac{t_p}{a_p^L} a_p^L(\delta_{ip} - a_i^L) = \sum_{p=1}^{I} -t_p(\delta_{ip} - a_i^L) = a_i^L \sum_{p=1}^{I} t_p - t_i \tag{2.20}$$

or, since t_ps add up to 1,

$$s_i^L = a_i^L - t_i \tag{2.21}$$

With this it is possible to eliminate the matrix multiplication (2.16) for last layer sensitivity calculation. Also division-by-zero or very small numbers in (2.19).

2.5 Binary Cross Entropy (BCE) with Logistic Activation

If we are doing a binary classification then (2.18) becomes:

$$c = -t_1 \log a_1 - t_2 \log a_2 \tag{2.22}$$

But, since $t_1 + t_2 = 1$ and $a_1 + a_2 = 1$,

$$c = -t_1 \log a_1 - (1 - t_1) \log(1 - a_1) \tag{2.23}$$

Obviously we are just predicting one boolean variable or a bit. And a single output neuron is sufficient for this.

If we try to apply softmax here (2.1) becomes:

$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2}} \quad \text{and} \quad a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2}} = 1 - a_1 \tag{2.24}$$

Moreover, by dividing the first equation with e^{z_1}

$$a_1 = \frac{e^{z_1 - z_2}}{e^{z_1 - z_2} + 1} \tag{2.25}$$

and substituting $z = z_1 - z_2$ and if we have only a single output we have:

$$a = \frac{e^z}{e^z + 1} \tag{2.26}$$

which is the (1.8), logistic activation function. And (1.10) give $\frac{\partial a}{\partial z}$.

From (2.23) and dropping the redundant subscript:

$$\frac{\partial c}{\partial a} = -\frac{t}{a} - \frac{(1-t)(-1)}{1-a} = \frac{-t + at + a - at}{a(1-a)} = \frac{a - t}{a(1-a)} \tag{2.27}$$

And using (2.8) the sensitivity of the final layer neuron is:

$$s^L = a^L(1 - a^L)\frac{a^L - t}{a^L(1 - a^L)} = a^L - t \tag{2.28}$$

Which is, unsurprisingly the same as (2.23). In short this whole section is a special case of 2.3 and 2.4.

Again, combining BCE with logistic activation helps to avoid divide-by-zero scenario that could occur in

Note that the last layer can have multiple logistic neurons each using BCE when we are trying to predict binary variables which as not mutually exclusive. For example, predicting tomorrow will be sunny/cloudy and warm/cold will have two logistic neurons each using BCE. Alternatively we can have four classes for the four combinations and use CCE with softmax.