Multi-Layer Neural Networks

Steve Renals

18 January 2016

1 Intorduction

The aim of neural network modelling is to learn a system which maps an input vector **x** to a an output vector **y**.

At *runtime* the network computes the output **y** for each input **x**; when *training* the network the aim is to optimise the parameters of the system such that the correct **y** is computed for each **x**. We are most interested in the output accuracy of the system for unseen test data – *aeneralisation*.

Feed-forward networks are characterised by a number of layers of computation. A *single layer network* has a single layer of computation to map between input and output; *Multi-layer networks* had additional layers of computation using learned representations – *hidden units*

2 Single-layer networks

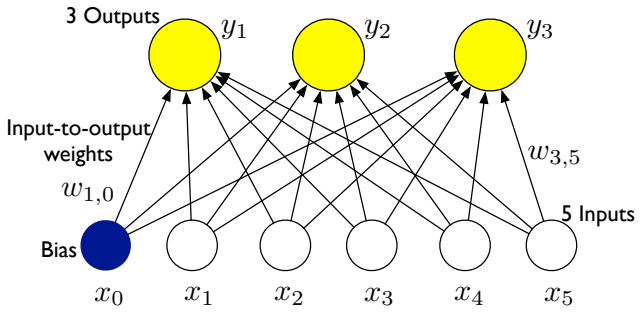

Figure 1: Schematic of a single-layer network

Figure 1 shows a single network, which may be defined as follows:

Input vector $\mathbf{x} = (x_1, x_1, \ldots, x_d)^T$

Output vector $\mathbf{y} = (y_1, \ldots, y_K)^T$

Weight matrix **W**: w_{ki} is the weight from input x_i to output y_k

Bias w_{k0} is the bias for output k

The Outputs are a weighted sum of inputs:

$$y_k = \sum_{i=1}^{d} w_{ki} x_i + w_{k0}$$

3 Training Single Layer Networks

Training set N input/output pairs $\{(\mathbf{x}^n, \mathbf{t}^n) : 1 \leq n \leq N\}$

Target vector $\mathbf{t}^n = (t_1^n, \ldots, t_K^n)^T$ – the target output for input $\mathbf{x}^n$

Training problem Set the values of the weight matrix $\mathbf{W}$ such that each input $\mathbf{x}^n$ is mapped to its target $\mathbf{t}^n$

Error function We define the training problem in terms of an error function E defined in terms of the network outputs $\mathbf{y}^n$ and the targets $\mathbf{t}^n$. Training corresponds to minimizing the error function E

This is a *supervised* learning setup - there is a target output for each input. We can also write the network output vector as $\mathbf{y}^n(\mathbf{x}^n; \mathbf{W})$ to explicitly show the dependence on the weight matrix and the input vector.

Training requires an *error function* which measures how far an output vector is from its target. The (squared) Euclidean distance – *squared error function* – is an example error function:

$$E = \frac{1}{2} \sum_{n=1}^{N} ||\mathbf{y}^n - \mathbf{t}^n||^2 = \sum_{n=1}^{N} E^n$$

$$E^n = \frac{1}{2} ||\mathbf{y}^n - \mathbf{t}^n||^2$$

E is the total error summed over the training set; E^n is the error for the nth training example. We can write E^n in terms of the errors for each training example:

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

The training process may be summarised as: set $\mathbf{W}$ to minimise E given the training set.

The notion of *weight space* is a helpful one when considering training neural networks. Weight space is a $K \times d$ dimension space – each possible weight matrix corresponds to a point in weight space. $E(\mathbf{W})$ is the value of the error at a specific point in weight space (given the training data).

To train a neural network we can use *gradient descent training*: adjust the weight matrix by moving a small direction down the gradient of $E(\mathbf{W})$ given $\mathbf{W}$, which is the direction along which E decreases most rapidly. We can write this gradient as $\nabla_{\mathbf{W}} E$, the vector of partial derivatives of E with respect to the elements of $\mathbf{W}$:

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

Gradient descent training involves updating each weight w_{ki} by adding a factor proportional to the gradient: $-\eta \cdot \partial E / \partial w_{ki}$. The *hyperparameter* η is a small constant called the *learning rate*.

1. Initialise the weight matrix with small random weights and load the training data
2. For each epoch
   (a) Initialise weight changes Δw_{ki} to zero

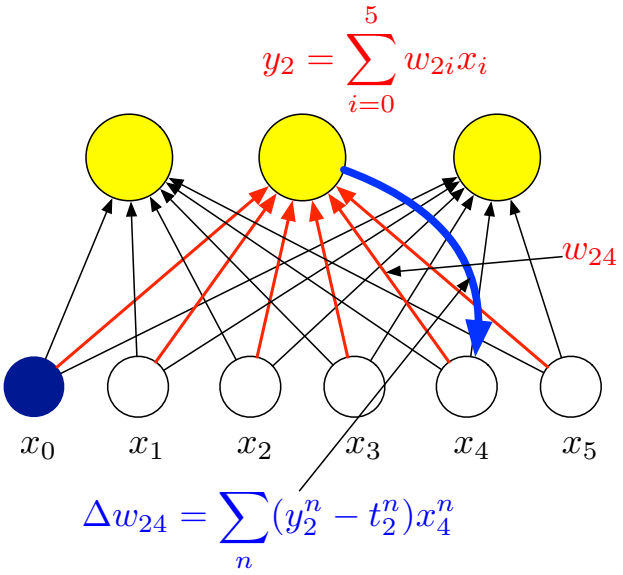

Figure 2: Applying gradient descent to train a single-layer network

(b) For each training example n:

i. Compute the error E^n

ii. Compute the gradients $\partial E^n / \partial w_{ki}$ for all k, i

iii. Update the total gradient by a small amount in the direction of $\nabla_{\mathbf{W}} E$):

$$\Delta w_{ki} \leftarrow \Delta w_{ki} + \frac{\partial E^n}{\partial w_{ki}} \quad \forall k, i$$

(c) Update weights: $w_{ki} \leftarrow w_{ki} - \eta \Delta w_{ki} \quad \forall k, i$

The procedure terminates either after a fixed number of epochs, or when the error stops decreasing by more than a threshold. (An epoch is a complete pass through the training data).

We outline how gradient descent may be applied to a single-layer network. This is illustrated in Figure 2, and pseudocode is given in Figure 3.

- Differentiate the error function with respect to each weight:

$$E^n = \frac{1}{2} \sum_{k=1}^{K} (y_k^n - t_k^n)^2 = \frac{1}{2} \sum_{k=1}^{K} \left(\sum_{i=0}^{d} w_{ki} x_i^n - t_k^n \right)^2$$

$$\frac{\partial E^n}{\partial w_{rs}} = (y_r^n - t_r^n) x_s^n = \delta_r^n x_s^n \quad ; \quad \delta_r^n = y_r^n - t_r^n$$

$$\frac{\partial E}{\partial w_{rs}} = \sum_{n=1}^{N} \frac{\partial E^n}{\partial w_{rs}} = \sum_{n=1}^{N} \delta_r^n x_s^n$$

- Weight update is

$$w_{rs} \leftarrow w_{rs} - \eta \sum_{n=1}^{N} \delta_r^n x_s^n$$

This is sometimes called the *delta rule*.

procedure GRADIENTDESCENTTRAINING($\mathbf{X}, \mathbf{T}, \mathbf{W}$)
    initialize $\mathbf{W}$ to small random numbers
    while not converged **do**
        for all k, i: $\Delta w_{ki} \leftarrow 0$
        for $n \leftarrow 1, N$ **do**
            for $k \leftarrow 1, K$ **do**
                $y_k^n \leftarrow \sum_{i=0}^{d} w_{ki} x_i^n$
                $\delta_k^n \leftarrow y_k^n - t_k^n$
                for $i \leftarrow 1, d$ **do**
                    $\Delta w_{ki} \leftarrow \Delta w_{ki} + \delta_k^n \cdot x_i^n$
                end for
            end for
        end for
        for all k, i: $w_{ki} \leftarrow w_{ki} - \eta \cdot \Delta w_{ki}$
    end while
end procedure

Figure 3: Pseudocode for gradient training of a single-layer network

In practice we use *stochastic gradient descent*. Rather than computing the exact gradient by summing per-example gradients over the complete training set ("batch gradient descent") which is very slow (only one update per epoch), the true gradient $\partial E / \partial w_{ki}$ (obtained by summing over the entire training dataset) is approximated by the gradient for a point $\partial E^n / \partial w_{ki}$. This is called stochastic gradient descent, and the weights are updated after each training example rather than after the batch of training examples. Inaccuracies in the gradient estimates are washed away by the many approximations. To prevent multiple similar data points (all with similar gradient approximation inaccuracies) appearing consecutively, the training set is presented in random order. Pseudocode for SGD is given in Figure 4.

procedure SGDTRAINING($\mathbf{X}, \mathbf{T}, \mathbf{W}$)
    initialize $\mathbf{W}$ to small random numbers
    randomize order of training examples in $\mathbf{X}$
    while not converged **do**
        for $n \leftarrow 1, N$ **do**
            for $k \leftarrow 1, K$ **do**
                $y_k^n \leftarrow \sum_{i=0}^{d} w_{ki} x_i^n$
                $\delta_k^n \leftarrow y_k^n - t_k^n$
                for $i \leftarrow 1, d$ **do**
                    $w_{ki} \leftarrow w_{ki} - \eta \cdot \delta_k^n \cdot x_i^n$
                end for
            end for
        end for
    end while
end procedure

Figure 4: Pseudocode for stochastic gradient training of a single-layer network

In practice, *minibatches* are used: the gradient is computed from a minibatch of M training examples, where $M > 1$, $M << N$. The main reason for this is computationally efficiency as it enables the best use of vectorisation, keeping processor pipelines full.

4 Classification, sigmoids and softmax

In speech recognition we are often interested in *classification* – given some acoustic observation can we classify it as a phone or a word? Classification outputs can be binary (1/0) or probabilistic – p, $1-p$, for a 2-class problem. One could train a linear single layer network as a classifier in which the output targets are 1/0; at run time if the output $y > 0.5$ classify as yes, otherwise classify as no. This will work, but it is better to use output *activation functions* to constrain the outputs to binary or probabilistic values.

4.1 Sigmoids

Consider a single-layer network with a *sigmoid* output (Figure 5). In this case the output is the weighted sum of the inputs (as before) followed by a sigmoid function f:

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

The sigmoid (plotted in Figure 6) is a squashing function with strongly negative inputs saturating to 0, and strongly positive inputs saturating to 1.

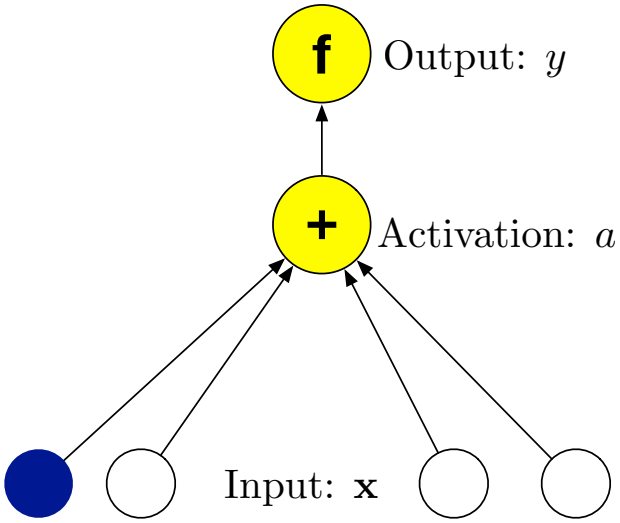

Figure 5: Single-layer network with sigmoid output

We can interpret the output of a sigmoid single layer network probabilistically (statisticians would call this logistic regression). Let a be the *activation* of the single output unit, the value of the weighted sum of inputs, before the activation function, so:

$$y = f(a) = f\left(\sum_i w_i x_i\right)$$

For two classes, we have single output y, with weights w_i

To train a sigmoid single layer network, gradient descent requires $\partial E/\partial w_i$ for all weights:

$$\frac{\partial E^n}{\partial w_i} = \frac{\partial E^n}{\partial y^n}\frac{\partial y^n}{\partial a^n}\frac{\partial a^n}{\partial w_i}$$

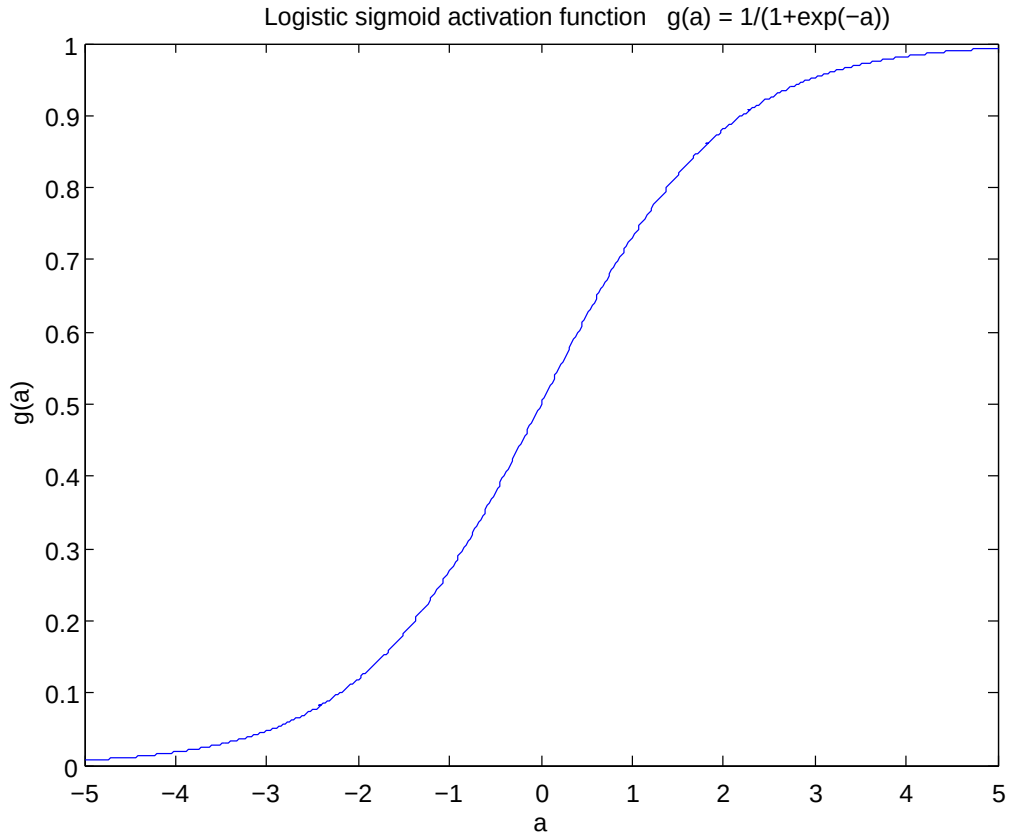

Figure 6: Sigmoid activation function

For a sigmoid:

$$y = f(a) \quad \frac{dy}{da} = f(a)(1 - f(a))$$

(Show that this is indeed the derivative of a sigmoid.) Therefore:

$$\frac{\partial E^n}{\partial w_i} = \underbrace{(y^n - t^n)}_{\delta^n} \underbrace{f(a^n)(1 - f(a^n))}_{f'(a^n)} x_i^n$$

The application of gradient descent to a sigmoid single-layer network is illustrated in Figure 7.

4.2 Cross-entropy error function

If we use a sigmoid single layer network for a two class problem (C_1 (target $t = 1$) and C_2 ($t = 0$)), then we can interpret the output as follows

$$y \sim P(C_1 \mid \mathbf{x}) = P(t = 1 \mid \mathbf{x})$$
$$(1 - y) \sim P(C_2 \mid \mathbf{x}) = P(t = 0 \mid \mathbf{x})$$

Combining, and recalling the target is binary

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

This is a Bernoulli distribution. We can write the log probability:

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

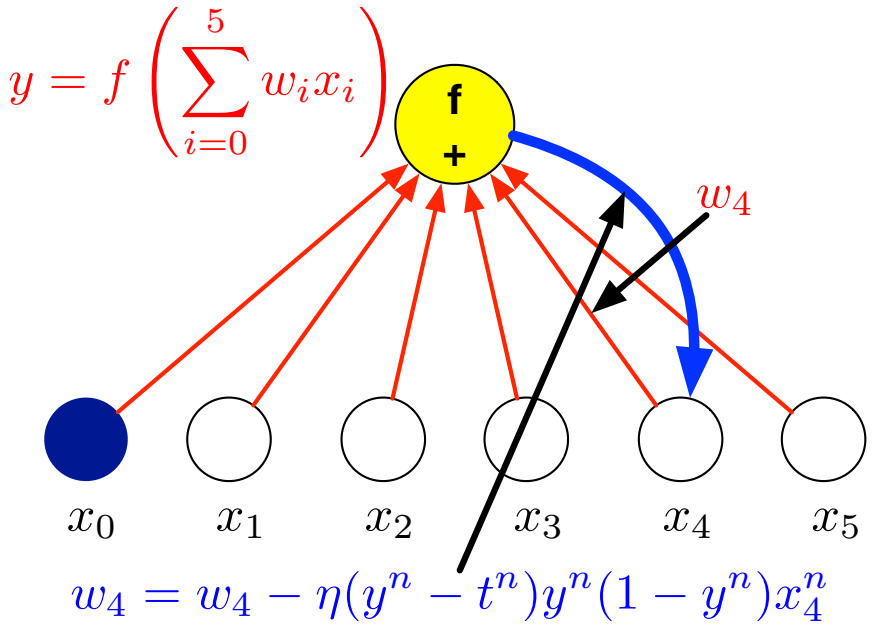

Figure 7: Training a single-layer network with sigmoid output

We optimise the weights **W** to maximise the log probability – or to minimise the negative log probability. Write the error function as follows:

$$E^n = -(t^n \ln y^n + (1 - t^n) \ln(1 - y^n)) .$$

This is called the **cross-entropy error function**

Gradient descent training requires the derivative $\partial E/\partial w_i$ (where w_i connects the ith input to the single output).

$$\begin{aligned}
\frac{\partial E}{\partial y} &= -\frac{t}{y} + \frac{1-t}{1-y} = \frac{-(1-y)t + y(1-t)}{y(1-y)} \\
\frac{\partial E}{\partial w_i} &= \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial a} \cdot \frac{\partial a}{\partial w_i} \\
&= \frac{-(1-y)t + y(1-t)}{y(1-y)} \cdot y(1-y) \cdot x_i = (y-t)x_i
\end{aligned}$$

(derivative of the sigmoid $y(1 - y)$ cancels)

4.3 Multi-class networks and Softmax

If we have K classes we can use a "one-hot" ("one-from-N") output coding – the target of the correct class is 1, all other targets are zero. It is possible to have a multi-class net with sigmoid output functions, but this is not the best approach. Using multiple sigmoids for multiple classes means that $\sum_k P(k|\mathbf{x})$ is not constrained to equal 1 – we want this if we would like to interpret the outputs of the net as class probabilities. The solution is to use an activation function with a sum-to-one constraint: **softmax**.

The softmax activation function is given as follows:

$$y_k = \frac{\exp(a_k)}{\sum_{j=1}^{K} \exp(a_j)}$$
$$a_k = \sum_{i=0}^{d} w_{ki} x_i$$

Softmax has the following properties

- Each output will be between 0 and 1
- The denominator ensures that the K outputs will sum to 1

Using softmax we can interpret the network output y_k^n as an estimate of $P(k|\mathbf{x}^n)$ – Softmax is the multiclass version of the two-class sigmoid.

To train a softmax network, we can extend the cross-entropy error function to the multiclass case

$$E^n = -\sum_{k=1}^{C} t_k^n \ln y_k^n$$

Again the overall gradient we need is

$$\frac{\partial E^n}{\partial w_{ki}} = \sum_{c=1}^{C} \frac{\partial E}{\partial y_c} \cdot \frac{\partial y_c}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{ki}}$$
$$= \sum_{c=1}^{C} -\frac{t_c}{y_c} \cdot \frac{\partial y_c}{\partial a_k} \cdot x_i$$

Note that the kth activation a_k – and hence the weight w_{ki} – influences the error function through all the output units, because of the normalising term in the denominator. We have to take this into account when differentiating. However when you do the differentiation, the resulting expression is very simple:

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

Here δ_{ck} ($\delta_{ck} = 1$ if $c = k$, $\delta_{ck} = 0$ if $c \neq k$ is called the Kronecker delta. Putting it all together we find:

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

The delta rule!