Artificial Neural Networks

Kristian Wichmann

July 18, 2017

1 Feed-forward artificial neural networks

An *artificial neural network* (ANN) superficially mimics the workings of the brain by being made up of *neurons* and *dendrites/axons*[1].

Here we will focus on *feed-forward neural networks*. Here, the neurons are organized into *layers*. These come in three categories:

- The *input layer*, into which the data to be processed enters.
- *Hidden layers* - which are layers between the input and output layers.
- The *output layer* in which the result of the ANN's data processing can be read.

Every neuron in each layer is connected to every neuron in the next layer. A feed-forward ANN has at least three layer: Input, output and at least one hidden layer. An ANN with two or more hidden layers is known as a *deep network*.

Figure 1 shows a neural network with three laters: 3 cells in the input layer, 4 cells in the hidden layer, and 2 cells in the output layer.

The cells in each layer are connected by dendrites/axons, which are represented as arrows in figure 1: Every cell in each layer is connected to all the cells in the next layer.

As the data put into the input layer is translated into *activity* levels in the cells. These activities then propagate via the axons through the network, causing different activity levels in each layer. The is repeated, until reaching the conclusion: The activity levels in the output layer. These are then interpreted accordingly.

[1] In real neural networks, dendrites carry inputs to the neuron, while axons carry outputs from the neuron.

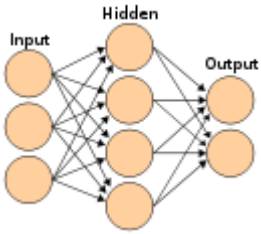

Figure 1: An ANN with three layers.

2 Neuron modelling

The activity of a neuron is affected by all dendrites pointing to the cell - for a feed-forward neural network, this means all cells in the previous layer. In other words, the activity of the neuron depends on these inputs.

2.1 Types of neurons: Deterministic

The actual activity depends on the type of neuron used. In this section we'll look at *deterministic* neurons, i.e. neurons where the inputs uniquely determine the activity. In other words, the activity is a function of the input. We call this the *activation function*.

Here, we will only consider functions that depends on a weighted sum of the inputs: All the inputs are weighted corresponding to the strength of each dendrite. In addition a *bias* term may be added - a constant term. The result, which can thus be written as $z = \sum_{i=1}^{n} \theta_i x_i + b$, is sometimes called the *pre-activation*. Here, the x_i are the inputs (n total), the θ_i the corresponding weights (strength of the dendrites) and b is the bias. The actual activity is found by applying the *activation function* to this quantity. Mathematically, this may be formulated:

$$a_{\text{neuron}} = f(z) = f\left(\sum_{i=1}^{n} \theta_i x_i + x_0\right) \tag{2.1}$$

Here, a is the activation, x_0 is the (renamed) bias term and finally f is the activation function.

The situation is outlined in figure 2. Note that the bias has been shown as an input, even if this is not really the case: It is merely a constant term.

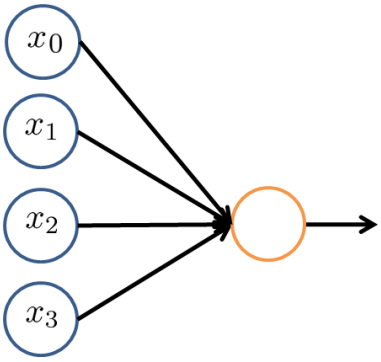

Figure 2: Neuron with three inputs and a bias term

The inputs and weights are often written in vector form:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \theta = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{pmatrix} \tag{2.2}$$

Hence, equation 2.1 may be written:

$$a_{\text{neuron}} = f\left(\theta^T x + x_0\right) \tag{2.3}$$

2.1.1 Linear neurons

Here, the activation function is simply $f(z) = z$.

2.1.2 Binary threshold neurons

For these types of neurons, the activation function is a *step function*, i.e. a function of the form:

$$f(z) = \begin{cases} 0 & \text{if } z < c \\ 1 & \text{if } z \geq c \end{cases} \tag{2.4}$$

Here c is the *activation threshold*. By adjusting the bias term, c can always be turned into zero.

This type of neuron is the basis for the learning model known as the *perceptron*.

2.1.3 Rectified linear neurons

This combines the two previous types of neurons by setting:

$$f(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases} \tag{2.5}$$

These are also sometimes known as *linear threshold neurons*.

2.1.4 Sigmoid neurons

Here, the activation function is a *sigmoid function*, which just means s-shaped. This may be any function that is convex for $z < 0$ and concave for $z > 0$, and which has finite limits for $z \to \pm\infty$.

The most famous of these functions is the *logistic function*, which is the one we will focus on in the following. In fact it's common to simply denote it as **the** sigmoid function. The next section is devoted to the details of this function.

Another example are tanh-neurons, for which the activation function unsurprisingly is:

$$f(z) = \tanh(z) \tag{2.6}$$

2.2 Stochastic binary neurons

The deterministic neurons always have the same activity given the same inputs. This is not true for stochastic binary neurons. Instead, the values of the activation function $f(z)$ are taken as *probabilities*. More precisely, it is the probability of the activity being 1. Otherwise, the activation is 0 (hence the 'binary' in the name).

This usually assumes that f only takes on values between 0 and 1, so that it may be interpreted as a probability.

3 The logistic function

As noted above, we will consider sigmoid neurons for the rest of this text. For our activation function, we will choose the *logistic function*. It is defined as:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{3.1}$$

The function is graphed in figure 3. It can be though of as a "smoothed out step function".

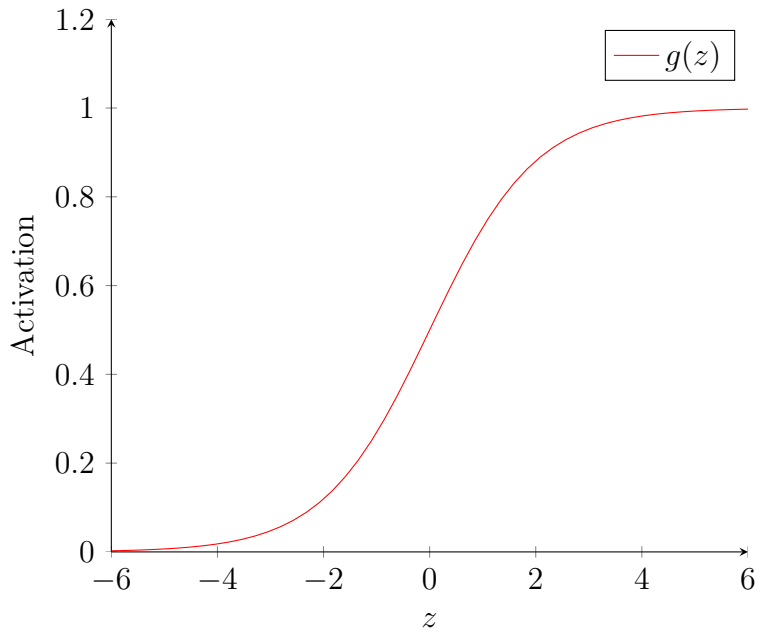

Figure 3: The sigmoid function.

The reason for the choice of this function is two-fold. First of all, it introduces *activation saturation* into the model: Even if $\theta^T x + x_0$ turns out to be very large, the corresponding activity will still be close to 1. Similarly large negative values produce activities close to 0.

However, many functions could have achieved activation saturation. But $g(z)$ also has an additional nice mathematical property. Consider its derivative:

$$g'(z) = \frac{1' \cdot (1 + e^{-z}) - 1 \cdot (1 + e^{-z})'}{(1 + e^{-z})^2} = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{e^{-z}}{1 + e^{-z}} g(z) \tag{3.2}$$

It turns out the first factor is equal to $1 - g(z)$:

$$1 - g(z) = 1 - \frac{1}{1 + e^{-z}} = \frac{1 + e^{-z}}{1 + e^{-z}} - \frac{1}{1 + e^{-z}} = \frac{1 + e^{-z} - 1}{1 + e^{-z}} = \frac{e^{-z}}{1 + e^{-z}} \tag{3.3}$$

So we have:

$$g'(z) = g(z)\,(1 - g(z)) \tag{3.4}$$

4 Matrix form of layer activities

So, given cell number i in layer L, its activation depends on the input activities from layer $L-1$, the activity is:

$$a_i^{(L)} = g\left(\left(\theta_i^{(L)}\right)^T a^{(L-1)} + b_i^{(L)}\right) \tag{4.1}$$

Here, the activities from layer $L-1$ has been collected into the vector $a^{(L-1)}$ and the biases into the vector b^L. However, this may simply be expressed through matrix multiplication:

$$a^{(L)} = g\left(\theta^{(L)} a^{(L-1)} + b^{(L)}\right) \tag{4.2}$$

Here, the matrix $\theta^{(L)}$ has rows consisting of weights for each cell. Also, g is to be applied component-wise.

4.1 Dimensionalities

If layer $L-1$ contains m_{L-1} cells and layer L contains m_L cells, this means the dimensionality of the matrices and vectors above are:

$$a^{(L-1)} \in \mathbb{R}^{m_{L-1}\times 1}, \quad a^{(L)}, b^{(L)} \in \mathbb{R}^{m_L \times 1}, \quad \theta^{(L)} \in \mathbb{R}^{m_L \times m_{L-1}} \tag{4.3}$$

4.2 Output layer as a function of input layer

If there is a total of K layers in the ANN. Then the input layer is $x \equiv a^{(1)}$ and the output layer $y \equiv a^{(K)}$. Then we may express y as a function of x through repeated application of equation 4.1. This process is known as *forward propagation*:

$$y = g\left(\theta^{(K)}\left(\cdots g\left(\theta^{(3)} g\left(\theta^{(2)} x + b^{(2)}\right) + b^{(3)}\right)\cdots\right) + b^{(K)}\right) \equiv h^{(\theta,b)} \tag{4.4}$$

Here the notation $h^{(\theta,b)}$ been introduced. Here h stands for *hypothesis*, i.e. the hypothesis that the particular choices of θ and b are appropriate.

5 ANN used for classification

As an example, consider an ANN trained for picture classification. Se we imagine an ANN which has a representation of a picture for inputs x. The

output layer is a series of n numbers between 0 and 1. We may interpret these as probabilities. Maybe an ANN has been trained to recognize cats, dogs and horses. In this case, the size of the output layer is 3, and y_1, y_2 and y_3 are interpreted as the probabilities of the input picture containing a cat, dog or a horse respectively. If we know the picture only contains one animal, we would simply pick the highest value and use the corresponding animal as prediction.

5.1 Training an ANN - overview

So the question is now how to train an ANN for classification? This is a supervised machine learning problem, as we imagine we have a large dataset to train the network on.

The actual training is done using *maximum likelihood estimation* of the weights and biases. In practice, this will be done using gradient or a similar optimization algorithm. The partial derivatives needed for this is found using what is known as the *backpropagation algorithm*.

All of these topics will be the subject of the next set of sections.

6 Likelihood of a Bernoulli process

Let's take a step back and consider a Bernoulli process, i.e. an experiment with two outcomes: success and failure. The probability of success is called p, and hence the probability of failure must be $1 - p$. This may be summed up in the equation:

$$P(x|p) = p^x(1-p)^{1-x}, \quad x = 0, 1 \tag{6.1}$$

Here $x = 1$ indicates success and $x = 0$ failure.

The "probability mindset" here is, that if we know p we know the distribution. The idea of likelihood swaps this around: Instead we ask what the parameter p might be given the outcome x:

$$\mathcal{L}(p|x) = p^x(1-p)^{1-x} \tag{6.2}$$

Based on the observation, we will then pick the value of p that makes the observation most likely. This is known as maximum likelihood estimation[2].

[2]For a single observation this means the estimate is $p = 1$ in case of success and $p = 0$ in case of failure. In other words $p = x$.

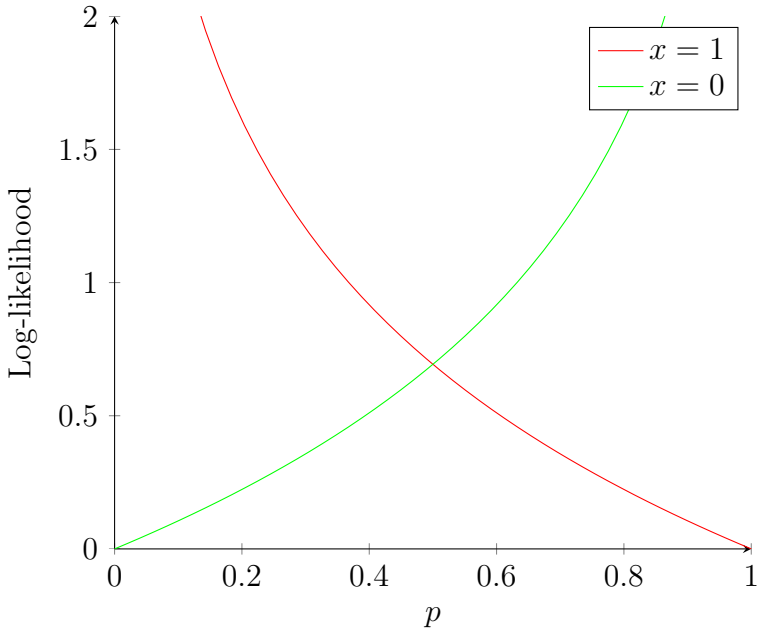

Figure 4: Log-likelihoods for the Bernoulli process.

Often, it is more convenient to minimize the *log-likelihood*, i.e. minus the logarithm of the likelihood[3]. Since log is monotone, this is equivalent. For the Bernoulli process:

$$L(p|x) = -\log \mathcal{L}(p|x) = -x \log p - (1-x) \log(1-p) \tag{6.3}$$

This may also be viewed as the *cross-entropy* between the distributions of x and p.

L can also be seen as a *cost function*, i.e. something we wish to minimize. Figure 4 graphs the function for the two values of x. Because of the connection to entropy it is sometimes knows as the *cross-entropy loss function*.

7 Single item training set

Now consider a single item training set, i.e. an input x that has been classified as option a. This corresponds to a Bernoulli process for each cell in the output y, with only the a'th one being a success. So the desired output is $y = \delta_a$.

Now, the log-likelihood of the i'th such Bernoulli process depends on the actual choice of weights and biases - more specifically, it depends on the $h^{(\theta,b)}$

[3]Not all presentations include the minus sign. It is kept here to make L a cost function.

function defined in equation 4.4:

$$L_i(\theta, b|y_i) = -y_i \log\left[h_i^{(\theta,b)}\right] - (1-y_i)\log\left[1-h_i^{(\theta,b)}\right] \tag{7.1}$$

If the processes are assumed to be independent (a big if, but let's go with that for now), the log-likelihoods add up, and the total likelihood is:

$$L(\theta, b|y) = \sum_i \left\{-y_i \log\left[h_i^{(\theta,b)}\right] - (1-y_i)\log\left[1-h_i^{(\theta,b)}\right]\right\} \tag{7.2}$$

The task is now to find the weights and biases that minimize this. To do so, we wish to use gradient descent, and so need the partial derivatives of L with respect to all weights and biases!

If α is any weight or bias, we can differentiate L with respect to it:

$$\frac{\partial L}{\partial \alpha} = \sum_i \left\{-y_i \frac{1}{h_i^{(\theta,b)}}\frac{\partial h_i^{(\theta,b)}}{\partial \alpha} - (1-y_i)\frac{1}{1-h_i^{(\theta,b)}}\left(-\frac{\partial h_i^{(\theta,b)}}{\partial \alpha}\right)\right\} \tag{7.3}$$

Collecting terms:

$$\frac{\partial L}{\partial \alpha} = \sum_i \left\{\frac{-y_i\left(1-h_i^{(\theta,b)}\right) + (1-y_1)h_i^{(\theta,b)}}{h_i^{(\theta,b)}(1-h_i^{(\theta,b)})}\frac{\partial h_i^{(\theta,b)}}{\partial \alpha}\right\} \tag{7.4}$$

$$\sum_i \left\{\frac{h_i^{(\theta,b)} - y_i}{h_i^{(\theta,b)}\left(1-h_i^{(\theta,b)}\right)}\frac{\partial h_i^{(\theta,b)}}{\partial \alpha}\right\} \tag{7.5}$$

So the problem is reduced to finding partial derivatives of h.

7.1 Partial derivatives: Output layer

The values of the output layer can be expressed through their dependence on the last hidden layer as per equation 4.1:

$$h^{(\theta,b)} = a^{(K)} = g(\theta^{(K)}a^{(K-1)} + b^{(K)}) \tag{7.6}$$

Remember that this is a vector equation - we need to consider each component. The property of the sigmoid function expressed in equation 3.4 now comes in handy when we want to calculate partial derivatives:

$$\frac{\partial h_i^{(\theta,b)}}{\partial \alpha} = g(z_i)(1-g(z_i))\frac{\partial z_i}{\partial \alpha}, \quad z_i = \sum_j \theta_{ij}^{(K)} a_j^{(K-1)} + b_i^{(K)} \tag{7.7}$$

But $g(z_i) = h_i^{(\theta,b)}$! So inserting into equation 7.5 there's a lot of cancellation:

$$\frac{\partial L}{\partial \alpha} = \sum_i \left\{ \left(h_i^{(\theta,b)} - y_i \right) \frac{\partial z_i}{\partial \alpha} \right\} = \sum_i \delta_i^{(K)} \frac{\partial z_i}{\partial \alpha} \tag{7.8}$$

Here, the *error* $\delta^{(K)} = h_i^{(\theta,b)} - y_i$ for the output layer has been introduced. This can be seen as the amount the hypothesis specified by the current weights and biases is off compared to what is desired. Be careful to distinguish between the error and Kronecker deltas in the following!

Equation 7.8 may be written in vector form:

$$\frac{\partial L}{\partial \alpha} = \left(\delta^{(K)} \right)^T \frac{\partial z}{\partial \alpha} \tag{7.9}$$

Finally, we need to find the partial derivatives of z. Again, remember that z is a vector:

$$z_i = \sum_j \theta_{ij}^{(K)} a_j^{(K-1)} + b_i^{(K)} \tag{7.10}$$

At this point we only consider weights and biases from the axons pointing at the output layer, i.e. with superscript K. But since the weights form a matrix and the biases a vector, the resulting derivative objects will be 3 and 2-dimensional respectively:

$$\frac{\partial z_i}{\partial \theta_{kl}^{(K)}} = \sum_j \delta_{ik} \delta_{jl} a_j^{(K-1)} = \delta_{ik} a_l^{(K-1)}, \quad \frac{\partial z_i}{\partial b_k^{(K)}} = \delta_{ik} \tag{7.11}$$

This means that the only non-zero derivatives are:

$$\frac{\partial z_i}{\partial \theta_{ij}^{(K)}} = a_j^{(K-1)}, \quad \frac{\partial z_i}{\partial b_i^{(K)}} = 1 \tag{7.12}$$

We may now calculate the derivatives of L:

$$\frac{\partial L}{\partial \theta_{ij}^{(K)}} = \sum_k \delta_k^{(K)} \delta_{ki} a_j^{(K-1)} = \delta_i^{(K)} a_j^{(K-1)} \tag{7.13}$$

$$\frac{\partial L}{\partial b_i^{(K)}} = \sum_k \delta_k^{(K)} \delta_{ki} = \delta_i^{(K)} \tag{7.14}$$

7.2 Partial derivatives: Hidden layers

On to the hidden layers. Luckily, it turns out that most of the calculations echo the output layer one closely.

Let's look at a weight or bias α from the last hidden layer. We need the derivative of h with respect to α. It is convenient to make the following definition:

$$z^{(N)} = \theta^{(N)} a^{(N-1)} + b^{(N)} \tag{7.15}$$

Then we may write:

$$h^{(\theta,b)} = g(z^{(K)}) \tag{7.16}$$

Now we wish to form the partial derivative with respect to a weight/bias α from the last hidden layer, i.e. layer $K-1$. Here in vector form:

$$\frac{\partial h^{(\theta,b)}}{\partial \alpha} = g(z^{(K)}) \odot (1 - g(z^{(K)}) \odot \frac{\partial z^{(K)}}{\partial \alpha} \tag{7.17}$$

Here, the symbol $\odot$ is used to mean the *Hadamard product* or *Schur product*. It simply takes two vectors of equal length and return another vector of the same length, where each component is the product of the relevant factor components:

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \odot \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_n b_n \end{pmatrix} \tag{7.18}$$

So what is the derivative?

$$\frac{\partial z^{(K)}}{\partial \alpha} = \theta^{(K)} \frac{\partial a^{(K-1)}}{\partial \alpha} \tag{7.19}$$

But $a^{(K-1)} = g\left(z^{(K-1)}\right)$:

$$\frac{\partial}{\partial \alpha} g(z^{(K-1)}) = g(z^{(K-1)}) \odot (1 - g(z^{(K-1)})) \odot \frac{\partial z^{(K-1)}}{\partial \alpha} \tag{7.20}$$

Again, the two first factors can be expressed as activations, and because α is associated with layer $K-1$, the derivative is analogous to the results from equations 7.13 and 7.14. Inserting into equation 7.5, once again, there's some cancelling out, but this time we get some extra factors:

$$\sum_i \left\{ \delta_i^{(K)} \sum_j \theta_{ij}^{(K)} a_j^{(K-1)} \left(1 - a_j^{(K-1)}\right) \frac{\partial z_j^{(K-1)}}{\partial \alpha} \right\} \tag{7.21}$$

Reorder the summation and use that $A_{ij} = A_{ji}^T$:

$$\frac{\partial L}{\partial \alpha} = \sum_j \left\{ \underbrace{\sum_i \left(\theta^{(K)}\right)_{ji}^T \delta_i^{(K)}}\ a_j^{(K-1)} \left(1 - a_j^{(K-1)}\right) \frac{\partial z_j^{(K-1)}}{\partial \alpha} \right\} \tag{7.22}$$

The underbraced part is equal to $\left[\left(\theta^{(K)}\right)^T \delta^{(K)}\right]_j$. So it makes sense to set the error of layer $K-1$ equal to:

$$\delta^{(K-1)} = \left(\theta^{(K)}\right)^T \delta^{(K)} \odot a^{(K-1)} \odot \left(1 - a^{(K-1)}\right) \tag{7.23}$$

The calculations are now essentially the same as in the last section, and we get:

$$\frac{\partial L}{\partial \theta_{ij}^{(K-1)}} = \delta_i^{(K-1)} a_j^{(K-2)} \tag{7.24}$$

$$\frac{\partial L}{\partial b_i^{(K-1)}} = \delta_i^{(K-1)} \tag{7.25}$$

If α had been associated with an earlier hidden layer, we could have followed the same steps iteratively, defining the error of layer N recursively:

$$\delta^{(N)} = \delta^{(N+1)} = \left(\theta^{(N+1)}\right)^T \delta^{(N+1)} \odot a^{(N)} \odot \left(1 - a^{(N)}\right) \tag{7.26}$$

The general formulas for the derivatives is then:

$$\frac{\partial L}{\partial \theta_{ij}^{(N)}} = \delta_i^{(N)} a_j^{(N-1)} \tag{7.27}$$

$$\frac{\partial L}{\partial b_i^{(N)}} = \delta_i^{(N)} \tag{7.28}$$

Again, N should be at least 2, reflecting the fact that there's no error, weights or biases for the input layer.

These can be summed up in matrix form:

$$\frac{\partial L}{\partial \theta^{(N)}} = \delta^{(N)} \left(a^{(N-1)}\right)^T \tag{7.29}$$

$$\frac{\partial L}{\partial b^{(N)}} = \delta^{(N)} \tag{7.30}$$

7.3 Backpropagation algorithm

As we have seen above, calculation of derivatives is a recursive process, starting from the output layer and working backwards. Hence the name *backpropagation algorithm*. Here are the steps in bullet form:

- First use forward propagation to calculate all the activations of the neurons corresponding to the input x and the current weights and biases.

- Using the y's, calculate the error of the output layer.
- Repeat the following until all derivatives are calculated:
  - Calculate derivatives for the current layer using the layer's error and the previousy calculated activations.
  - Iteratively calculate the error of the previous layer.

This is merely one step in the gradient descent process. The derivatives are used to generate a new set of weights/biases and the process is repeated until all derivatives are sufficiently small.

8 Full training set

So far, we've been dealing with the situation for a training set consisting of only one item. In practice, we ideally want to train our ANN on a large dataset.

However, since each item in the training set is independent of each other, the log-likelihood of multiple items is simply equal to the sum of all of them. By additivity of the derivative, the backpropagation process essentially remains the same, except we need to sum over the entire training set in each step of the gradient descent.

8.1 Vectorization

If there's p data points in the training set, we may collect them all into a $(n+1) \times p$ matrix X:

$$X = \begin{pmatrix} 1 & \cdots & 1 \\ | & \cdots & | \\ x_1 & \cdots & x_p \\ | & \cdots & | \end{pmatrix} \tag{8.1}$$

Here, a constant coordinate of 1 have been added to each data point in order to unify treatment of weights and biases. This is done by setting:

$$\Theta^{(L)} = \begin{pmatrix} b_1^{(L)} & \theta_{11}^{(L)} & \theta_{12}^{(L)} & \cdots & \theta_{1n_{L-1}}^{(L)} \\ b_2^{(L)} & \theta_{21}^{(L)} & \theta_{22}^{(L)} & \cdots & \theta_{2n_{L-1}}^{(L)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{n_L}^{(L)} & \theta_{n_L1}^{(L)} & \theta_{n_L2}^{(L)} & \cdots & \theta_{n_Ln_{L-1}}^{(L)} \end{pmatrix} \tag{8.2}$$

So $\Theta^{(L)} \in \mathbb{R}^{n_L \times (n_{L-1}+1)}$. So we can get the activities in the first hidden layer by:

$$A^{(2)} = \Theta^{(2)} X = \begin{pmatrix} | & \cdots & | \\ a_1^{(2)} & \cdots & a_p^{(2)} \\ | & \cdots & | \end{pmatrix} \tag{8.3}$$

So $A^{(2)} \in \mathbb{R}^{n_2 \times p}$. To get the activities in the next layer, first we must add a row of ones in the same way we did with the input layer. Using this, we can get the collection of activities in the L'th layer by:

$$A^{(L)} = g(\Theta^{(L)} A^{(L-1)*}) \tag{8.4}$$

Here, the asterisk is a shorthand for adding a row of ones, similarly to equation 8.1. In general $A^{(L)} \in \mathbb{R}^{n_L \times p}$. Then we may rewrite 4.4 simply as:

$$H = A^{(K)} = g\left(\Theta^{(K)} \left(\cdots g(\Theta^{(2)} X)\right)^* \cdots\right) \tag{8.5}$$

Here, H is a $m_K \times p$ matrix containing the hypotheses for each training set data point:

$$H = \begin{pmatrix} | & \cdots & | \\ h_1 & \cdots & h_p \\ | & \cdots & | \end{pmatrix} \tag{8.6}$$

The errors for the last layer is now:

$$\Delta^{(K)} = H - Y \tag{8.7}$$

Here Y is a matrix containing the expected outcomes according to the training set:

$$Y = \begin{pmatrix} | & \cdots & | \\ y_1 & \cdots & y_p \\ | & \cdots & | \end{pmatrix} \tag{8.8}$$

Following the pattern of equation 7.29 we calculate:

$$\Delta^{(K)} \left(A^{(K-1)*}\right)^T = \begin{pmatrix} | & \cdots & | \\ \delta_1^{(K)} & \cdots & \delta_p^{(K)} \\ | & \cdots & | \end{pmatrix} \begin{pmatrix} 1 & - & a_1^{(K-1)} & - \\ \vdots & \vdots & \vdots & \vdots \\ 1 & - & a_p^{(K-1)} & - \end{pmatrix} \tag{8.9}$$

This means that the ij-element of the product matrix is the sum over the ij-derivatives for each of the p data point. It is customary to divide by p to get an average, so that different training set sizes have comparable dynamics during gradual descent.

x_1	x_2	x_1 XOR x_2
0	0	0
1	0	1
0	1	1
1	1	0

Table 1: XOR truth table.

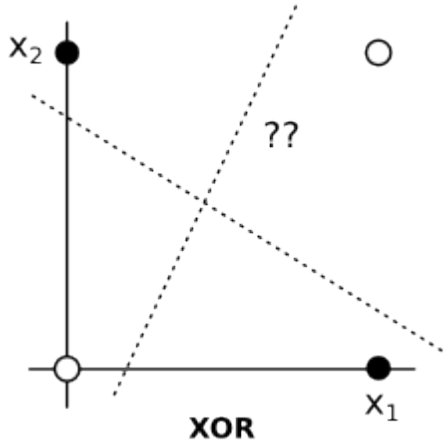

Figure 5: XOR is not linearly separable.

To backpropagate, we need to calculate the errors in the previous layers. Following equation 7.23, this is done through:

$$\Delta^{(L-1)} = \left(\theta^{(L)}\right)^T \Delta^{(K)} \odot A^{(L-1)} \odot (1 - A^{(L-1)}) \tag{8.10}$$

Note that the lower case theta is used here, as there's no bias error to propagate back to.

9 Example: Learning XOR

Exclusive or (XOR) is a logical operator with the truth table as shown in table 1. It is infamous for being linearly non-separable, as shown on figure 5.

However, we might try to learn the behaviour using a neural network. Figure 6 shows just about the simplest possible network for this task. The inputs are x_1 and x_2 and the output is the probability of the correct answer being 1.

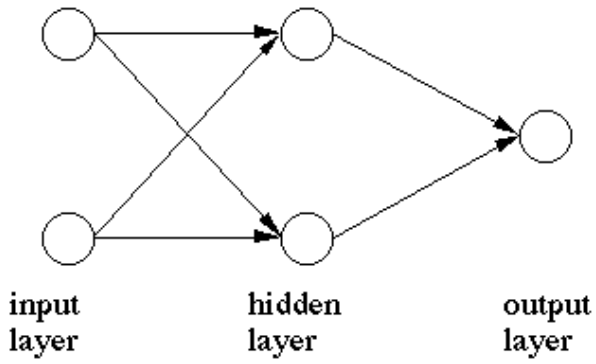

Figure 6: A simple neural net for learning XOR.

9.1 Vectorized formalism

So, we have two input neurons, one hidden layer with two neurons, and finally one output neuron. Our input and output matrices are determined by the four data point in table 1 and following equations 8.1 and 8.8:

$$X = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \in \mathbb{R}^{3\times 4}, \quad Y = \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix} \in \mathbb{R}^{1\times 4} \tag{9.1}$$

The weight matrices take on the form (with and without bias):

$$\Theta^{(1)} = \begin{pmatrix} b_1^{(1)} & \theta_{11}^{(1)} & \theta_{12}^{(1)} \\ b_2^{(1)} & \theta_{21}^{(1)} & \theta_{22}^{(1)} \end{pmatrix} \in \mathbb{R}^{2\times 3}, \quad \Theta^{(2)} = \begin{pmatrix} b_1^{(2)} & \theta_{11}^{(2)} & \theta_{12}^{(2)} \end{pmatrix} \in \mathbb{R}^{1\times 3} \tag{9.2}$$

$$\theta^{(1)} = \begin{pmatrix} \theta_{11}^{(1)} & \theta_{12}^{(1)} \\ \theta_{21}^{(1)} & \theta_{22}^{(1)} \end{pmatrix} \in \mathbb{R}^{2\times 2}, \quad \theta^{(2)} = \begin{pmatrix} \theta_{11}^{(2)} & \theta_{12}^{(2)} \end{pmatrix} \in \mathbb{R}^{1\times 2} \tag{9.3}$$

Now, forward propagation takes the form:

$$A = g(\Theta^{(1)} X) \in \mathbb{R}^{2\times 4}, \quad H = g(\Theta^{(2)} A^*) \in \mathbb{R}^{1\times 4} \tag{9.4}$$

Backpropagation is done first by:

$$\Delta^{(2)} = H - Y \in \mathbb{R}^{1\times 4}, \quad \frac{\partial L}{\partial \Theta^{(2)}} = \Delta^{(2)} (A^*)^t \in \mathbb{R}^{1\times 3} \tag{9.5}$$

Then by:

$$\Delta^{(1)} = \left(\theta^{(2)}\right)^t \Delta^{(2)} \odot A \odot (1 - A) \in \mathbb{R}^{2\times 4}, \quad \frac{\partial L}{\partial \Theta^{(1)}} = \Delta^{(1)} X^t \in \mathbb{R}^{2\times 3} \tag{9.6}$$