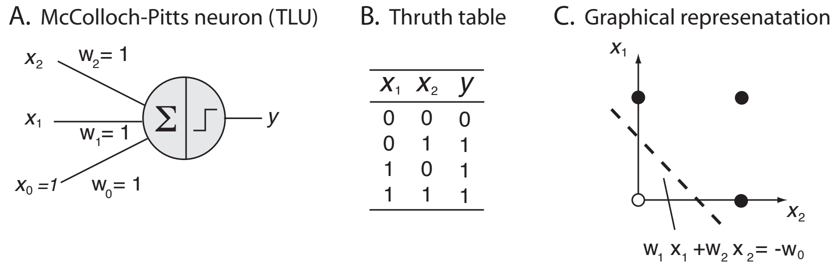
1 Multilayer perceptron

This manuscript introduces basic feedforward neural networks, so called multiplayer perceptrons (MLPs).

1.1 The single layer perceptron


A. McColloch-Pitts neuron (TLU)
x_2
$w_2 = 1$
x_1
$w_1 = 1$
$x_0 = 1$
$w_0 = 1$
Σ
y
B. Thruth table
C. Graphical represenatation
x_1
x_2
$w_1 x_1 + w_2 x_2 = -w_0$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 1.1 Representation of the boolean OR function with a McCulloch-Pitts neuron (TLU).

We will now explore the computational capabilities of networks of simple neuron-like elements. The neuron model used here is that of a McCulloch–Pitts neuron or that of a population rate node. Such a unit is shown in Fig. 1.1A with three input channels, although it could have an arbitrary number of input channels. Input values are labeled by x with a subscript for each channel. Each channel has a corresponding **weight parameter**, w_i, representing synaptic weights or the effect thereoff.

Each node operates in the following way. Each input value is multiplied with the corresponding weight value, and these weighted values are summed. The weighted sum is then supplied as argument to the **transfer function** or **gain function** g to produce the output. Hence, the output of each node is governed by the

$$\textbf{update rule} \quad y(\mathbf{x};\mathbf{w}) = g(\sum_{i=0}^{n} w_i x_i) = g(\mathbf{w}^T\mathbf{x}) \tag{1.1}$$

The notation of the function on the left hand side is worth noting; it tell us that the output y is calculated from the input $\mathbf{x}$ and that this function has parameters $\mathbf{w}$ listed after the semicolon. The right hand side then specifies this function.

In case of the threshold unit representing the McCulloch-Pitts neurone, the grain function is given by

$$g(x) = \theta(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} . \tag{1.2}$$

McCulloch and Pitts introduced this unit as a simple neuron model, and they argued that such a unit can perform computational tasks resembling boolean logic. This is demonstrated in Fig. 1.1 for a threshold unit that implements the Boolean OR function.

Another important grain function is the **sigmoid function** or **logistic function**

$$g(x) = \frac{1}{1 + e^{-ax+b}}. \tag{1.3}$$

A graphical representation of this model is shown in Figure 1.2A. We will mainly use this gain function in the following and note that the threshold function correspond to the limit of infinite slope a. While this has been an important historical gain functions for neural networks, it is now argued that rectified-linear units

$$g(x) = x * \theta(x). \tag{1.4}$$

have advantages as discussed later.

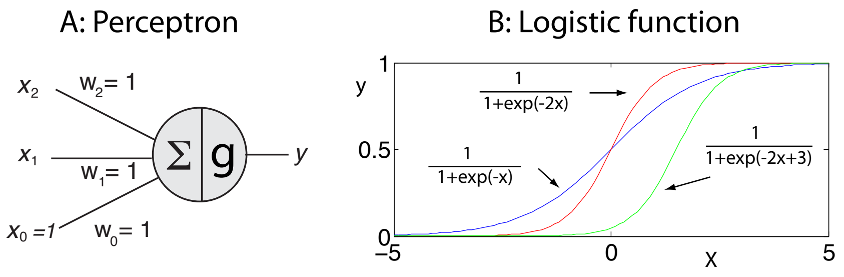


Fig. 1.2 A) Graphical representation of a perceptron with three input channels of which one is constant. B) The logistic function with different slopes and offset parameters.

The main question is now how to adjust the parameters of the model given by the connection weights w_i, so that the perceptron would perform a task correctly. The procedure we use is to provide to the system a number of examples, let's say m input data, $\mathbf{x}^{(i)}$ and the corresponding desired outputs, $y^{(i)}$. Such a task where a teacher provides the desired output or label $y^{(i)}$ for some feature vectors $\mathbf{x}^{(i)}$ is called **supervised learning**. The learning rule that we are using is given by

$$\textbf{learning rule} \quad w_j \leftarrow w_j + \alpha y' \left(y^{(i)} - y(\mathbf{x}^{(i)}) \right) x_j^{(i)}, \tag{1.5}$$

which is also related to the Widrow-Hoff learning rule, the Adaline rule, and the delta rule. These learning rules are nearly identical but differ with respect to the term y'. In the case of the logistic perceptron the term y' is given by

$$y' = y(1 - y). \tag{1.6}$$

The learning rule it is often called the delta rule because the difference between the desired and actual output. When multiplying out the difference with the inputs results in two product terms with components that represent the values of nodes framing the connection between them. A learning rule that depends on the activities of the pre- and post-synamptic neurone is called a Hebbian rule. Thus the delta rule is a Hebbian rule (after the famous NovaScotian Donald Hebb) which learned according to the desired output and unlearns the actual output,

$$\Delta w_j \propto (y^{(i)} x_j^{(i)} - y x_j^{(i)}). \tag{1.7}$$

In other words, the learning rule reinforces the correlation between the input and the desired output and reduces the correlation between input and the inaccurate actual output until the actual output is equal to tyne desired output. As stated before, such a learning rule is also called Hebbian learning.

There was a lot of excitement during the 1960s in the AI and psychology community about such a learning system that resemble some brain functions. However, Minsky and Peppert showed in 1968 that such perceptrons can not represent all possible boolean functions (sometimes called the XOR problem). While it was known at this time that this problem could be overcome by a layered structure of such perceptrons (called **multilayer perceptrons**), a learning algorithms was not widely known at this time. This nearly abolished the field of learning machines, and the AI community concentrated on rule-based systems in the following years.

1.1.1 Derivation of the batch learning rule

The logistic perceptron is given by

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}'\mathbf{x}}} = \frac{1}{1 + e^{-\sum_i w_i x_i}}. \tag{1.8}$$

The next step is to specify how we will search for good parameters. The first step is to quantify how we measure a good fit. We chose here to measure this with what we will call the **objective function** or **error function**. We choose this for now to be the mean square error function

$$E(\mathbf{w}) = \frac{1}{2N} \sum_i \left(y^{(i)} - y(\mathbf{x}^{(i)}; \mathbf{w}) \right)^2. \tag{1.9}$$

Using the $1/2$ in this formula is a useful convention. Note that this is a function of the parameters as the output of our model depends on the weight values. The minimum of this function correspond to the weight values that best describe the training data. To find this minimum we use an iterative method called gradient descent. In this method we start with a guess of the weight values and improve our prediction iteratively according to the change of the error function. More formally, the update of the weight values is

$$w_j \leftarrow w_j - \alpha \frac{\partial E}{\partial w_j}, \tag{1.10}$$

where α is is a learning rate, often called a hyper-parameter since it is a parameter of the learning process rather than the resulting model. We can now calculate the gradient in order to provide a formula that can be implemented with Matlab. For this we have to recall two rules from calculus namely that the derivative of an exponent function is

$$\frac{\mathrm{d}}{\mathrm{d}x}x^n = nx^{n-1}, \tag{1.11}$$

and the other is the chain rule

$$\frac{\mathrm{d}}{\mathrm{d}x}f(g(x)) = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x}. \tag{1.12}$$

With these rules we get

$$\frac{\partial E}{\partial w_j} = \frac{1}{N}\sum_i \left((y^{(i)} - y(\mathbf{x}^{(i)};\mathbf{w}))(-1)\frac{\partial y}{\partial w_j} \right). \tag{1.13}$$

The derivative of our model with respect to the parameters is

$$\frac{\partial y}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{1+e^{-\sum_i w_i x_i}} = -\frac{e^{-\sum_i w_i x_i}}{(1+e^{-\sum_i w_i x_i})^2}\frac{\partial \sum_i w_i x_i}{\partial w_j}. \tag{1.14}$$

In the remaining derivative over the sum, only the term containing w_j survives. Hence this derivative is x_j. We can also write the other portion of this equation in terms of the original function, namely

$$\frac{e^{-\sum_i w_i x_i}}{(1+e^{-\sum_i w_i x_i})^2} = y(1-y), \tag{1.15}$$

and hence

$$\frac{\partial y}{\partial w_j} = y(1-y)x_j. \tag{1.16}$$

We can now collect all the pieces and write the whole update rule for the weight values as

$$w_j \leftarrow w_j + \alpha\frac{1}{N}\sum_i \left((y^{(i)} - y(\mathbf{x}^{(i)};\mathbf{w}))y(\mathbf{x}^{(i)};\mathbf{w})(1-y(\mathbf{x}^{(i)};\mathbf{w}))x_j^{(i)} \right) \tag{1.17}$$

The first part of the sum is called the delta term

$$\delta(\mathbf{x}^{(i)};\mathbf{w}) = (y^{(i)} - y(\mathbf{x}^{(i)};\mathbf{w}))y(\mathbf{x}^{(i)};\mathbf{w})(1-y(\mathbf{x}^{(i)};\mathbf{w})), \tag{1.18}$$

or, if we write this without the arguments to better see the structure, it is

$$\delta = (y^{(i)} - y)y(1-y) \tag{1.19}$$

We can thus write the learning rule as

$$w_j \leftarrow w_j + \alpha\frac{1}{N}\sum_i \left(\delta(\mathbf{x}^{(i)};\mathbf{w})x_j^{(i)} \right) \tag{1.20}$$

Note that the learning stops when $y = 1$ or $y = 0$. This should of course be the case when when the correct answer is reached. However, it is even zero when the the wrong answer was reached, and learning also slows down when the y values approach these values. Such a slow down of learning together with the distribution of the gradient in deep networks leads to the problem of **vanishing gradients** that have been a major problem of deep networks in the 19990s. We will come back to this point later.

1.1.2 Batch versus online learning rule

We have derived here the learning rule based on the mean square error over all the training points. This corresponds to applying all the training examples and calculating the average gradient before updating the weight values based on this average. This is called **batch learning** since we use the whole batch of training examples for each weight update step.

In contrast, in class we have derived the learning rule for one example. That is, we could just apply one training tuple $(\mathbf{x}^{(i)}, y^{(i)})$ and calculate the gradient for this point, and use this gradient to update the weight value after the application of each data point. This is called **online learning** since the idea is that we could use each incoming data point for one update and don't have to store anything. Of course, in reality we want to do several iterations so we have anyhow to keep each training point. This method is also called **stochastic gradient descent** if we assume that the training points are randomly chosen.

(Include figure showing the difference)

What is the advantage or disadvantage of the different methods? The batch algorithm guarantees that the average training error goes down. So if you plot this curve and you see that the training error is rising than there must be something wrong. In contrast, when we change the weights based only on the last training example it is expected that the performance on other training points gets worse, so we have to make sure to keep the learning rate small. Note that we might at first think that making the average training error small is hence much better, but also keep in mind that we are after good generalization and that making the average training error very small might indeed lead to overfitting. It is hence good to monitor the generalization (test or validation) error. The advantage of the stochastic method is that it is less likely to get stuck in shallow areas of the error manifold. With big data it is now common to use a method with **mini batches** in which we divide the data into small batches and use a stochastic gradient over these sub batches.

1.2 Multilayer perceptron (MLP)

A multilayer perceptron with a layer of m input nodes, a layer of h hidden nodes, and a layer of n output nodes, is shown in Figure 1.3. The input layer is merely just relaying the inputs, while the hidden and output layer do active calculations. Such a network is thus called a 2-layer network. The term hidden nodes comes from the fact that these nodes do not have connections to the external world such as the input and output nodes.

The network is a graphical representation of a nonlinear function of the form

$$\mathbf{y} = g(\mathbf{w}^{\mathrm{o}} g(\mathbf{w}^{\mathrm{h}}\mathbf{x})). \tag{1.21}$$

It is easy to include more hidden layers in this formula. For example, the operation rule for a four-layer network with three hidden layers and one output layer can be written as

$$\mathbf{y} = g(\mathbf{w}^{\mathrm{o}} g(\mathbf{w}^{\mathrm{h3}} g(\mathbf{w}^{\mathrm{h2}} g(\mathbf{w}^{\mathrm{h1}}\mathbf{x})))). \tag{1.22}$$

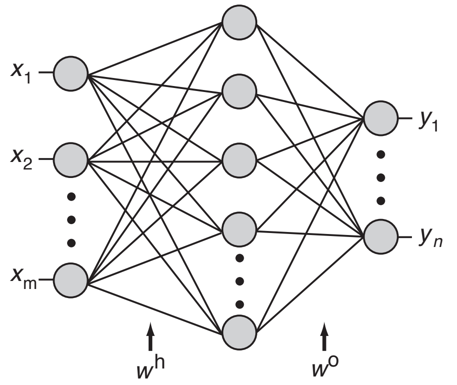


Fig. 1.3 The standard architecture of a feedforward multilayer network with one hidden layer, in which input values are distributed to all hidden nodes with weighting factors summarized in the weight matrix $\mathbf{w}^{\mathrm{h}}$. The output values of the nodes of the hidden layer are passed to the output layer, again scaled by the values of the connection strength as specified by the elements in the weight matrix $\mathbf{w}^{\mathrm{o}}$.

Table 1.1 Summary of error-back-propagation algorithm

```
Initialize weights arbitrarily
Repeat until error is sufficiently small
    Apply a sample pattern to all input nodes: $x_i$
    Propagate input through the network by calculating the rates of
        nodes in successive layers $l$: $y_i^l = g(\sum_j w_{ij}^l y_j^{l-1})$
    Compute the delta term for the output layer:
        $\delta_i^{\mathrm{out}} = g'(y_i^{\mathrm{out}-1})(y_i^{\mathrm{desired}} - y_i^{\mathrm{out}})$
    Back-propagate delta terms through the network:
        $\delta_i^{l-1} = g'(y_i^{l-1}) \sum_j w_{ji}^l \delta_j^l$
    Update weight matrix by adding the term: $\Delta w_{ij}^l = \alpha \delta_i^l y_j^{l-1}$
```

Learning these networks is a bit more challenging since the original delta rule requires desired values of some nodes. While a teacher can supply these values for the output nodes, we do not have this values for the hidden nodes. In the generalized delta rule called the **error backpropagation algorithm** we will use delta term of the output and back propagate this values to the hidden nodes with the appropriate multiplications of the corresponding weights between the output and the hidden nodes. The online version of this algorithm is summarized in Table 1.1.

Which functions can be approximated by multilayer perceptrons? The answer is, in principle, any. A multilayer feedforward network is a **universal function approximator**. This means that, given enough hidden nodes, any mapping functions can be approximated with arbitrary precision by these networks. The remaining problems are to know how many hidden nodes we need, and to find the right weight values. Also, the general approximator characteristics does not tell us if it is better to use more hidden layers or just to increase the number of nodes in one hidden layer. These are important concerns for practical engineering applications of those networks.

1.2.1 Derivation of error-backpropagation

To derive the learning rule we consider again minimizing MSE. The learning rule is then given by a gradient descent on this error function. Specifically, the gradient of the MSE error function with respect to the output weights is given by

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\mathrm{out}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}}\sum_k (\mathbf{y}^{(k)} - \mathbf{y})^2 \\ &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}}\sum_k \left(\mathbf{y}^{(k)} - g(\mathbf{w}^{\mathrm{out}} g(\mathbf{w}^h \mathbf{x}^{(k)})\right)^2\end{aligned} \tag{1.23}$$

Let's call the activation of the hidden nodes $\mathbf{y}^h$,

$$\mathbf{y}^h = g(\mathbf{w}^{\mathrm{h}}\mathbf{x})). \tag{1.24}$$

Then we can continue with our derivative as,

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\mathrm{out}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}}\sum_k \left(\mathbf{y}^{(k)} - g(\mathbf{w}^{\mathrm{out}}\mathbf{y}^h)\right)^2 \\ &= -\sum_k g'(\mathbf{w}^h\mathbf{x}^{(k)})(y_i^{(k)} - y_i)y_j^h \\ &= \delta_i^{\mathrm{out}} y_j^{\mathrm{h}},\end{aligned} \tag{1.25}$$

Eqn 1.25 is just the delta rule as before because we have only considered the output layer. The calculation of the gradients with respect to the weights to the hidden layer again requires the chain rule as they are more embedded in the error function. Thus we have to calculate the derivative

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\mathrm{h}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{h}}}\sum_k (\mathbf{y}^{(k)} - \mathbf{y})^2 \\ &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{h}}}\sum_k \left(\mathbf{y}^{(k)} - g(\mathbf{w}^{\mathrm{out}} g(\mathbf{w}^{\mathrm{h}}\mathbf{x}^{(k)})\right)^2.\end{aligned} \tag{1.26}$$

After some battle with indices (which can easily be avoided with analytical calculation programs such as MAPLE or MATHEMATICA), we can write the derivative in a form similar to that of the derivative of the output layer, namely

$$\frac{\partial E}{\partial w_{ij}^{\mathrm{h}}} = \delta_i^{\mathrm{h}} x_j, \tag{1.27}$$

when we define the delta term of the hidden term as

$$\delta_i^{\mathrm{h}} = g^{\mathrm{h}\prime}(h_i^{\mathrm{in}})\sum_k w_{ik}^{\mathrm{out}}\delta_k^{\mathrm{out}}. \tag{1.28}$$

The error term δ_i^{h} is calculated from the error term of the output layer with a formula that looks similar to the general update formula of the network, except that a signal is propagating from the output layer to the previous layer. This is the reason that the algorithm is called the **error-back-propagation algorithm**.

In this derivation we used the MSE over all the training patterns. Since all the training patterns are used at once, this algorithm is again a batch algorithm. This is generally a good idea, but it also takes up a lot of memory with large training sets. However, we can also use a similar algorithm with one training pattern at a time as shown in Table 1.1. Much more common with large data sets are **mini-batches**.