5 MLP

This chapter starts with a brief historical introduction to neural networks and the basic perceptron, basically a neural network without hidden layers. We also discuss the multilayer perceptron and training with gradient decent.

5.1 The historical threshold perceptron

There was always a strong interest of AI researchers in **real intelligence**, that is, to understand the human mind. For example, both Alan Turing and John von Neumann worked more directly on biological systems in their last years before their early passing, and human behaviour and the brain have always been of interest to AI researchers.

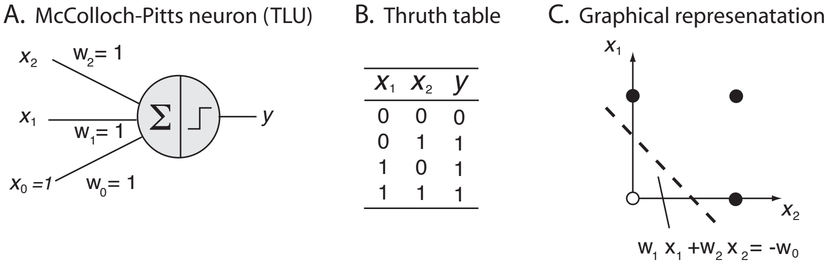

Fig. 5.1 Representation of the boolean OR function with a McCulloch-Pitts neuron (TLU).

A seminal paper, which has greatly influenced the development of early learning machines, is the 1943 paper by Warren McCulloch and Walter Pitts. In this paper, they proposed a simple model of a neuron, called the **threshold logical unit**, now often called the **McCulloch–Pitts neuron**. Such a unit is shown in Fig. 5.1A 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 also a **weight parameter**, w_i. The McCulloch–Pitts neuron operates in the following way. Each input value is multiplied with the corresponding weight value, and these weighted values are then summed. If the weighted summed input is larger than a certain threshold value, $-w_0$, then the output is set to one, and zero otherwise, that is,

$$y(\mathbf{x};\mathbf{w}) = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i = \mathbf{w}^T\mathbf{x} > 0 \\ 0 \qquad\quad \text{otherwise} \end{cases} . \tag{5.1}$$

Note the notation for the function on the left hand side; 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. We can also write the formula more compact as

$$y(\mathbf{x}; \mathbf{w}) = \theta(\mathbf{w}^T \mathbf{x}) \tag{5.2}$$

where the function θ is called a threshold function of the Heavisde step function

$$\theta(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} . \tag{5.3}$$

The Heaviside step function is here first example of a non-linear function that transformed the sum of the weighted input. This function is often called transfer function or grain function in the neural network community.

The model above resembles, to some extend, a neuron in that a neuron is also summing synaptic inputs and fires (has a spike in its membrane potential) when the membrane potential reaches a certain level that opens special voltage-gated ion channels. 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. 5.1 for a threshold unit that implements the Boolean OR function. The symbol h is used in these lecture notes since the output of this neuron represents the **hypothesis** that this neuron implements given the parameters $\mathbf{w}$. Also note that the non-linear step-function used in this neuron model corresponds to hypothesis for classification.

The next major developments in this area were done by Frank Rosenblatt and his engineering colleague Charles Wightman (Fig. 5.2), using such elements to build a machine that Rosenblatt called the **perceptron**. As can be seen in the figures, they worked on a machine that can perform letter recognition, and that the machine consisted of a lot of cables, forming a network of simple, neuron-like elements.

The most important challenge for the team was to find a way how to adjust the parameters of the model, the connection weights w_i, so that the perceptron would perform a task correctly. The procedure was to provide to the system a number of examples, let's say m input data, $\mathbf{x}^{(i)}$ and the corresponding desired outputs, $y^{(i)}$. The procedure they used thus resembles supervised learning. The learning rule they used is called the **perceptron learning rule**,

$$w_j := w_j + \alpha \left(y^{(i)} - y(\mathbf{x}^{(i)}) \right) x_j^{(i)}, \tag{5.4}$$

which is also related to the Widrow-Hoff learning rule, the Adaline rule, and the delta rule. These learning rules are nearly identical, but are sometimes used in slightly different contexts It is often called the delta rule because the difference between the desired and actual output (difference between actual (training) data and hypothesis) to guide the learning. When multiplying out the difference with the inputs results in two product term, where the components are the values of nodes framing the connection. 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,

$$\Delta w_j \propto (y^{(i)} x_j^{(i)} - y x_j^{(i)}). \tag{5.5}$$

In other words, the learning rule reinforces the correlation between the input and the desired output and reduces the correlation between input and the actual output. Such

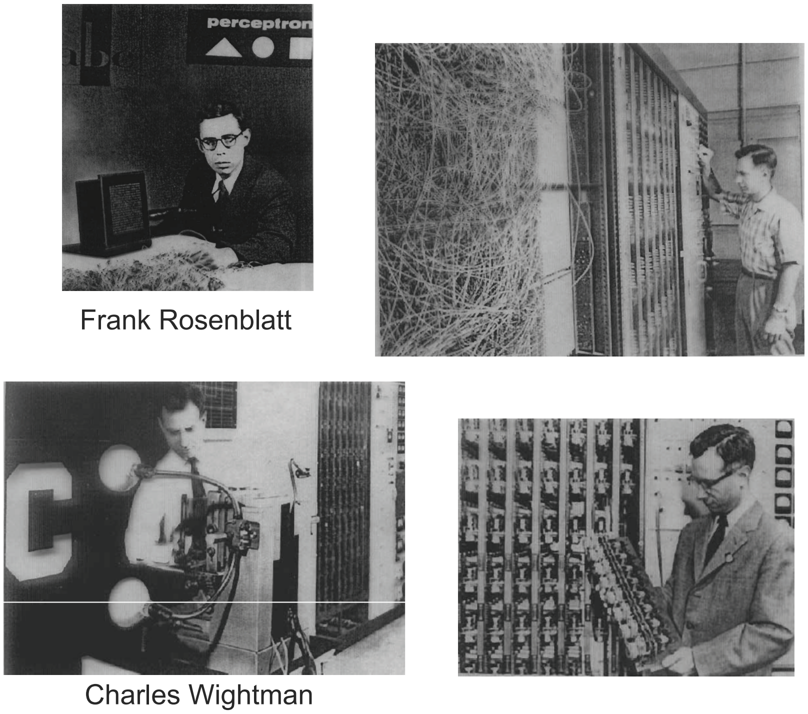

Fig. 5.2 Neural Network computers in the late 1950s.

a learning rule is also called Hebbian . We will see later that learning rule as a special case of a gradient descent rule for a linear hypothesis function. Although this rule is not ideal for a perceptron with non-linear functions, it turned out that it still works in same cases since it corresponds to taking a step towards minimizing MSE, albeit with a wrong gradient.

There was a lot of excitement during the 1960s in the AI and psychology community about such 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.

5.2 The sigmoid perceptron

5.2.1 Derivation of the batch learning rule

The logistic perceptron is a specific **model** or **parameterized hypothesis functions** given by

$$y(\mathbf{x};\mathbf{w}) = \frac{1}{1+e^{-\mathbf{w}'\mathbf{x}}} = \frac{1}{1+e^{-\sum_i w_i x_i}}. \qquad (5.6)$$

A graphical representation of this model is shown in Figure 5.3A. This is a perceptron which sums up the weighted inputs and puts the resulting value through the logistic function

$$g(x) = \frac{1}{1+e^{-ax+b}}. \qquad (5.7)$$

We have included there the parameters a and b which determines the slope and offset of this sigmoidal function. Some examples of this function with different values for this parameter is show in Figure 5.3B. In the perceptron the slope parameter represents the weights for the different input channels, and the offset is represented by a weight to a fixed input that is always set to 1. This little trick in representations simplifies the notation considerably.

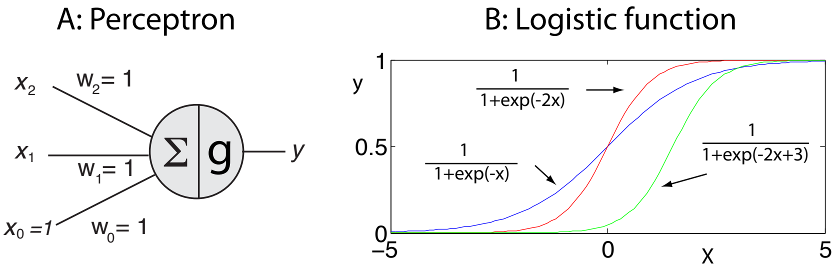

Fig. 5.3 A) Graphical representation of a perceptron with three input channels of which one is constant. B) The logistic function with different slopes and offset parameters.

Supervised learning means that we are fitting this function to our data points of the training set $\{\mathbf{x}^{(i)}, y^{(i)}\}$. The superscript (i) labels the individual training datum. We do this by comparing the desired label $y^{(i)}$ with the label predicted by the model $y(\mathbf{x}^{(i)};\mathbf{w})$. Note the difference between these two y-values. The first is a given number for each training datum, the other one is calculated from the feature values $\mathbf{x}^{(i)}$ according to the model with for a given set of parameters $\mathbf{w}$

The next step is to specify how we will search for good parameters. The first step for this 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

$$E(\mathbf{w}) = \frac{1}{2N}\sum_i \left(y^{(i)} - y(\mathbf{x}^{(i)};\mathbf{w})\right)^2. \qquad (5.8)$$

Using the $1/2$ in this formula is pure convention. Note that this is a function of the parameters as the output of our model represents 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 the 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

$$w_j \leftarrow w_j - \alpha \frac{\partial E}{\partial w_j}, \tag{5.9}$$

where α is a learning parameter. 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

$$\frac{\mathrm{d}}{\mathrm{d}x} x^n = n x^{n-1}. \tag{5.10}$$

The derivative of the Euler function is

$$\frac{\mathrm{d}}{\mathrm{d}x} e^x = e^x, \tag{5.11}$$

which means that this function at every point is equal to its slope. Finally we need the chain rule

$$\frac{\mathrm{d}}{\mathrm{d}x} f(g(x)) = \frac{\mathrm{d}f}{\mathrm{d}g} \frac{\mathrm{d}g}{\mathrm{d}x}. \tag{5.12}$$

With these rule we get

$$\frac{\partial E}{\partial w_j} = \frac{1}{N} \sum_i \left((y^{(i)} - y(\mathbf{x}^{(i)}; \mathbf{w}))(-1) \frac{\partial y}{\partial w_j} \right). \tag{5.13}$$

The derivative of our model with respect to the parameters is

$$\frac{\partial y}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{1 + e^{-\sum_i w_i x_i}} = \frac{e^{-\sum_i w_i x_i}}{(1 + e^{-\sum_i w_i x_i})^2} \frac{\partial \sum_i w_i x_i}{\partial w_j}. \tag{5.14}$$

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

$$\frac{e^{-\sum_i w_i x_i}}{(1 + e^{-\sum_i w_i x_i})^2} = y(1 - y), \tag{5.15}$$

and hence

$$\frac{\partial y}{\partial w_j} = y(1 - y) x_j. \tag{5.16}$$

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

$$w_j \leftarrow w_j - \alpha \frac{1}{N} \sum_i \left((y^{(i)} - y(\mathbf{x}^{(i)}; \mathbf{w})) y(\mathbf{x}^{(i)}; \mathbf{w})(1 - y(\mathbf{x}^{(i)}; \mathbf{w})) x^{(i)_j} \right) \tag{5.17}$$

The first part of in the sum is after called the delta term

$$\delta(\mathbf{x}^{(i)}; \mathbf{w}) = (y^{(i)} - y(\mathbf{x}^{(i)}; \mathbf{w})) y(\mathbf{x}^{(i)}; \mathbf{w})(1 - y(\mathbf{x}^{(i)}; \mathbf{w})), \tag{5.18}$$

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

$$\delta = (y^{(i)} - y)y(1 - y) \tag{5.19}$$

We can thus write the learning rule as

$$w_j \leftarrow w_j + \alpha \frac{1}{N} \sum_i \left(\delta(\mathbf{x}^{(i)}; \mathbf{w}) x_j^{(i)} \right) \tag{5.20}$$

5.2.2 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 training** 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 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 an **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 that we have anyhow keep each training point. This method is also called **stochastic gradient descent** is 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 is guarantied that the average training error goes down. So if you plot this curve and you see that the training error is raising than there must be something wrong. In contrast, when we change the weights based on the last training example it is expected that the performance to the other training points get worse and 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 god to monitor the generalization (test or validation) error. The advantage of the stochastic nature 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.

5.2.3 The threshold perceptron

The original perceptron we discussed was a threshold perceptron that was based on a grain function

$$g(x) = \begin{cases} 1 & \text{if } x > \theta \\ 0 & \text{otherwise} \end{cases} . \tag{5.21}$$

We can not apply gradient descent directly to this function as it is not differentiable at $g(\theta)$. Even if we take the left or right limit would not work as the gradient of $g(x)$

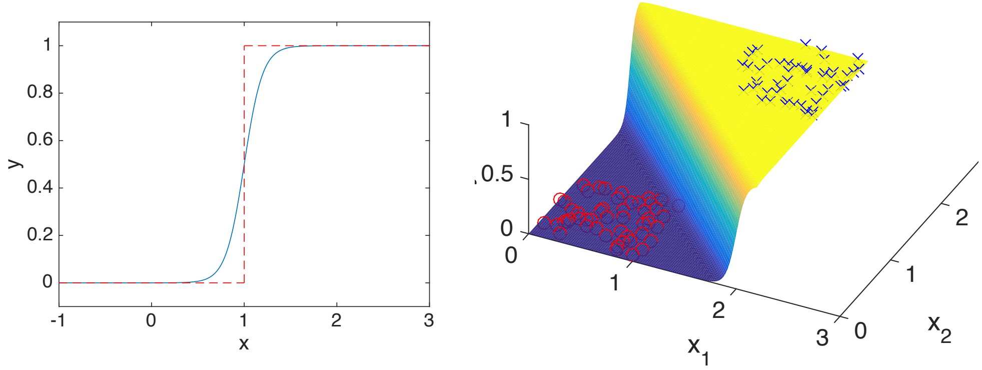

Fig. 5.4 A) logist function versus threshold function. B) Logistic regression in two dimensional feature space.

for $x \neq 0$ is always zero. However, we can see the the sigmoid function as a smooth approximation of the step function.

We could however also think about just using a linear perceptron with transfer function

$$g(x) = x, \tag{5.22}$$

and just use a threshold function after training in a post processing step. The gradient of the linear function is just x so that we end up with the perceptron learning rule as originally stated.

5.3 Multilayer perceptron (MLP)

The generalization of a delta rule, known as error-backpropagation, was finally introduced and popularized by Rumelhart, Hinton and Williams in 1986, although many others including Paul Werbos and Sunichi Amari have used it before. This popularization resulted in the explosion of the field of **Artificial Neural Networks**.

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 5.5. 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. Instead of the step function used in the McCulloch-Pitts model above, most such networks use a sigmoidal non-linearity,

$$g(x) = tanh(\beta x) = 2\frac{1}{1+e^{-\beta x}} - 1, \tag{5.23}$$

to allow for continuous values of the nodes. The network is thus a graphical representation of a nonlinear function of the form

$$\mathbf{y} = g(\mathbf{w}^{\mathrm{o}} g(\mathbf{w}^{\mathrm{h}}\mathbf{x})). \tag{5.24}$$

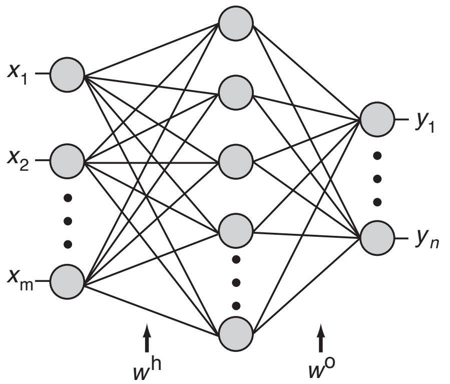

Fig. 5.5 The standard architecture of a feedforward multilayer network with one hidden layer, in which input values are distributed to all hidden nodes with weighting factors summarized in the weight matrix $\mathbf{w}^{\mathrm{h}}$. The output values of the nodes of the hidden layer are passed to the output layer, again scaled by the values of the connection strength as specified by the elements in the weight matrix $\mathbf{w}^{\mathrm{o}}$.

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

$$\mathbf{y} = g(\mathbf{w}^{\mathrm{o}} g(\mathbf{w}^{\mathrm{h3}} g(\mathbf{w}^{\mathrm{h2}} g(\mathbf{w}^{\mathrm{h1}} \mathbf{x})))). \tag{5.25}$$

Let us discuss a special case of a multilayer mapping network where all the nodes in all hidden layers have linear activation functions ($g(x) = x$). Eqn 5.24 then simplifies to

$$\begin{aligned} \mathbf{y} &= \mathbf{w}^{\mathrm{o}} \mathbf{w}^{\mathrm{h3}} \mathbf{w}^{\mathrm{h2}} \mathbf{w}^{\mathrm{h1}} \mathbf{x} \\ &= \tilde{\mathbf{w}} \mathbf{x}. \end{aligned} \tag{5.26}$$

In the last step we have used the fact that the multiplication of a series of matrices simply yields another matrix, which we labelled $\tilde{\mathbf{w}}$. Eqn 5.25 represents a single-layer network as discussed before. It is therefore essential to include non-linear activation functions, at least in the hidden layers, to make possible the advantages of hidden layers that we are about to discuss. We could also include connections between different hidden layers, not just between consecutive layers as shown in Fig. 5.5, but the basic layered structure is sufficient for the following discussions.

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. These questions are related to the bias-variance tradeoff in non-linear regression as discussed later.

To train the these networks we consider again minimizing MSE which would be appropriate for Gaussian noisy data around the mean described by the model. 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

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\mathrm{out}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}}\sum_k (\mathbf{y}^{(k)} - \mathbf{y})^2 \\ &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}}\sum_k \left(\mathbf{y}^{(k)} - g(\mathbf{w}^{\mathrm{out}} g(\mathbf{w}^h \mathbf{x}^{(k)}))\right)^2\end{aligned} \tag{5.27}$$

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

$$\mathbf{y}^h = g(\mathbf{w}^{\mathrm{h}}\mathbf{x})). \tag{5.28}$$

Then we can continue with our derivative as,

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\mathrm{out}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}}\sum_k \left(\mathbf{y}^{(k)} - g(\mathbf{w}^{\mathrm{out}} \mathbf{y}^h)\right)^2 \\ &= -\sum_k g'(\mathbf{w}^h \mathbf{x}^{(k)})(y_i^{(k)} - y_i)y_j^h \\ &= \delta_i^{\mathrm{out}} y_j^{\mathrm{h}},\end{aligned} \tag{5.29}$$

Eqn 5.28 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

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\mathrm{h}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{h}}}\sum_k (\mathbf{y}^{(k)} - \mathbf{y})^2 \\ &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{h}}}\sum_k \left(\mathbf{y}^{(k)} - g(\mathbf{w}^{\mathrm{out}} g(\mathbf{w}^{\mathrm{h}} \mathbf{x}^{(k)}))\right)^2.\end{aligned} \tag{5.30}$$

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

$$\frac{\partial E}{\partial w_{ij}^{\mathrm{h}}} = \delta_i^{\mathrm{h}} x_j, \tag{5.31}$$

when we define the delta term of the hidden term as

$$\delta_i^{\mathrm{h}} = g^{\mathrm{h}\prime}(h_i^{\mathrm{in}})\sum_k w_{ik}^{\mathrm{out}}\delta_k^{\mathrm{out}}. \tag{5.32}$$

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

Table 5.1 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: $\Delta w_{ij}^l = \alpha \delta_i^l y_j^{l-1}$
```

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 called 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. This is called an **online algorithm**, and this algorithm is summarized in Table 5.1. Much more common with large data sets are **mini-batches** as discussed later in more detail.

Such multilayer perceptrons were able to learn nonlinear relations in data and had some success in application. However, the general problem of overfitting and the question of optimality, and well as the applicability to large data problems with more complex functions, has hampered the field since the early successes. This area of artificial neural networks become now absorbed into the more general are of machine learning with new methods that have clarified field and have led to more applicable methods that we will discuss below. We therefore follow in the next sections a more contemporary path.

Before leaving this area it is useful to point out some more general observations. Artificial neural networks have certainly been one of the first successful methods for nonlinear regression, implementing nonlinear hypothesis of the form $h(\mathbf{x};\mathbf{w}) = g(\mathbf{w}^T x)$. The corresponding mean square loss function,

$$L \propto \left(y - g(\mathbf{w}^T x)\right)^2 \tag{5.33}$$

is then also a general nonlinear function of the parameters. Minimizing such a function is generally difficult. However, we could consider instead hypothesis that are linear in the parameters, $h(\mathbf{x};\mathbf{w}) = \mathbf{w}^T \phi(\mathbf{x};$, so that the MSE loss function is quadratic in the parameters,

$$L \propto \left(y - \mathbf{w}^T \phi(\mathbf{x})\right)^2 . \tag{5.34}$$

The corresponding quadratic optimization problem can be solved much more efficiently. This line of ideas are further developed in support vector machines discussed next. An interesting and central further issue is how to chose the non-linear function ϕ. This will be an important ingredient for nonlinear support vector machines and unsupervised learning discussed below.

5.4 Stochastic MLPs and Cross-Entropy Loss for sigmoidal classification

We have previously argued about the stochastic nature of the problem by building specific parameterized probabilistic models. But what if we don't know the functional form of the underlying probabilistic nature and how do MLPs fit into this view. Bayesian people would say that this must be suboptimal, so here we will relate this to our of using neural networks. These learning machine do somewhat try to compensate for the unknown by building large and encompassing functional models, in particular when it comes to deep networks with a large number of parameters.

For a given set of parameters, such a model gives us a specific response to each possible input. In order to make such a model stochastic we need to introduce some stochastic component into the model. There are different types of generalizations of such models. For example, we could include some noise into the response of each node of the network or include stochasticities in the parameters (weight values) of the model. In the past, different versions of stochastic gain functions have been used, and drop-out, which sets the response of a node to zero in random trials is another more recent example. With such stochastic modifications we get a model that represents a density function $p(\hat{y}|x)$.

Now we need a measure how good this model is in comparison to the true nature of data that are described by the unknown density function $q(y|x)$. One specific measure of the distance or divergence between two probability distributions is given by the **Cross-entropy**

$$H(p,q) = -\sum_x p(x) \log q(x) \tag{5.35}$$

In other words, this is the negative log probability of the given labels under the current model. Since we want to maximize the probability of the data under the model, we want to minimize the cross entropy. The cross entropy is related to the **KL-divergence** by

$$H(p,q) = H(p) + KL(p||q), \tag{5.36}$$

and since changing model parameters do not effect the true data, minimizing the cross entropy is equivalent to minimizing the KL-divergence.

Let us apply this to binary classification model which is described by Bernoulli variables that take the value 0 or 1. For this density function, the cross entropy is given by

$$\begin{aligned} H(p,q) &= -p(x=0)\log q(x=0) + -p(x=1)\log q(x=1) \\ &= -p(x)\log q(x) - (1-p(x))\log(1-q(x)) \end{aligned}$$

where $p(x)$ is shorthand for $p(x=1)$. The natural way for a neural networks to represent a Bernoulli variable is with a sigmoid output.

$$p(\hat{y}|x;w) = \frac{1}{1+e^{-\mathbf{x}\mathbf{w}}} \tag{5.37}$$

$$1 - p(\hat{y}|x;w) = \frac{1+e^{-\mathbf{x}\mathbf{w}}-1}{1+e^{-\mathbf{x}\mathbf{w}}} = \frac{1}{1+e^{\mathbf{x}\mathbf{w}}} \tag{5.38}$$

$$\log p(\hat{y}|x; w) = -\log(1 + e^{-\mathbf{xw}}) \tag{5.39}$$

$$\tag{5.40}$$

Note that the output of the network is the probability of the label being y=1 and not directly the label.

Maximizing the log-probability of the observed data is given by minimizing the cross-entropy between the network output, $p(\hat{y})$, and the given labels, y, or mathematically minimizing the Loss function

$$\begin{aligned}
L &= -y \log p(\hat{y}) - (1-y)\log(1 - p(\hat{y})) && (5.41)\\
&= y \log(1 + e^{-\mathbf{xw}}) + (1-y)\log(1 + e^{\mathbf{xw}}) && (5.42)\\
\frac{dL}{d(\mathbf{xW})} &= y\frac{-e^{-\mathbf{xw}}}{1+e^{-\mathbf{xw}}} + (1-y)\frac{e^{\mathbf{xw}}}{1+e^{\mathbf{xw}}} && (5.43)\\
&= -y\frac{1}{1+e^{\mathbf{xw}}} + (1-y)\frac{1}{1+e^{-\mathbf{xw}}} && (5.44)\\
&= -y(1 - p(\hat{y})) + (1-y)p(\hat{y}) && (5.45)\\
&= -y + yp(\hat{y}) + p(\hat{y}) - yp(\hat{y}) && (5.46)\\
&= p(\hat{y}) - y && (5.47)
\end{aligned}$$

For multi-class problems, the equivalent of the sigmoid is the **softmax function**

$$p(\hat{y} = i|x) = \frac{e^{\mathbf{xW}_i}}{\sum_{j=1}^{N} e^{\mathbf{xW}_j}}, \tag{5.48}$$

where N is the number of classes. You can easily see the equivalence to the sigmoid in the case of having two classes where one of them has input 0,

$$\begin{aligned}
p(\hat{y} = 0) &= \frac{e^0}{e^0 + e^{\mathbf{xw}}} = \frac{1}{1 + e^{\mathbf{xw}}} = 1 - \frac{1}{1 + e^{-\mathbf{xw}}}\\
p(\hat{y} = 1) &= \frac{e^{\mathbf{xw}}}{e^0 + e^{\mathbf{xw}}} = \frac{e^{\mathbf{xw}}}{1 + e^{\mathbf{xw}}} = \frac{1}{1 + e^{-\mathbf{xw}}}
\end{aligned}$$

The derivation of the gradient for this multi class case works out to the same as the binary classification,

$$\begin{aligned}
\frac{dL}{d(\mathbf{xW})} &= p(\hat{\mathbf{y}}) - \mathbf{y} && (5.49)\\
\frac{dL}{d\mathbf{x}} &= (p(\hat{\mathbf{y}}) - \mathbf{y})\mathbf{W}^T && (5.50)\\
\frac{dL}{d\mathbf{W}} &= \mathbf{x}^T(p(\hat{\mathbf{y}}) - \mathbf{y}) && (5.51)
\end{aligned}$$