8 MLP

Up to this point we have made the case that machine learning is about modelling based on a specific hypothesis function, ideally in the form of a density function that describes the data. We have illustrated this concept on simple problems, mostly linear regression or linear classification. While we have seen that causal models are not restricted to low dimensional cases, building such causal models in complex situations is certainly a challenge. We are now taking a second journey that has advanced the applicability of ML to high dimensional cases with complex nonlinearities. The idea of this journey is to make large models with high complexities and find creative ways to constrain them with data. The models we are now discussing are somewhat more generic which leaves us with many more parameters that have to be constrained. Large data collections together with some new techniques have opened such venues, and lot of recent success stories are build on this approach. We will specifically follow discuss neural network models, starting with more traditional multilayer perceptrons before exploring deep learning in the following chapter.

8.1 Neurons and the threshold perceptron

There was always a strong interest of AI researchers in **real intelligence**, that is, to understand the human mind that is enabled by the brain. The brain is made up of specialized cells including glia and neurons. Glia are thought to provide supporting functionalities to the neurons, and it is usually the neurons which are thought to enable most of the information processing functionality. A schematic example of a neuron is shown in Fig. 8.1a. The neuron has receiving arms called dendrites which other neurons contact to transmit signals. These signals are usually transmitted with the help of chemicals called neurotransmitters that are released from a presynaptic terminal and the end of a trunk called the axon. Axional release sites are tupically in close contact with dendritic receiving sites, and the neurotransmitters then open ion channels that alter the electrical charge of the neurons. The electrical signals are then transmitted to the soma of the cell. Electrical signals superimpose, meaning that the voltages of the different channels just add up. When this net input voltage reaches some sufficient values it triggers the opening of special ion channels in the axon that allow the transmission of this signal down the axon to the axonal release sites of neurotrasmitters, which completes this information transmission cycle. Since this is a model and not a real neuron it is helpful to realize that we made several simplification. For example, we ignored the description of the specific time course of opening and closing of ion channels and hence some of the more detailed dynamics of a neuron. Also, we ignored the description of the transmission of the electric signals within within the neuron; this is why such a model is commonly called a point-neuron. Nevertheless, this model

captures some important aspects of a neuron functionality, and it is the role of this functionality that we can explore further in the following.

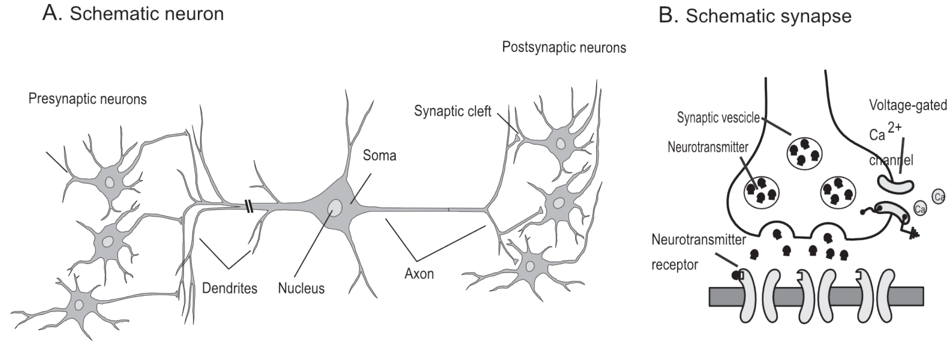

Fig. 8.1 A. Outline of the components of a neuron and its connectivity to other neurons in the brain that make up the neuronal network. B. Outline of s synaptic terminal where neurotransmitters are released that can trigger the opening of ion channels in the receiving neuron which in turn trigger the change of the electrical state of the neuron.

Warren McCulloch and Walter Pitts proposed a simple model of a neuron in 1943, called the **threshold logical unit**, now often called the McCulloch–Pitts neuron. Such a unit is shown in Fig. 8.2A 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, representing the 'strength' of a synapse. 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,

$$y(\mathbf{x};\mathbf{w}) = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i = \mathbf{w}^T\mathbf{x} > 0 \\ 0 \qquad\qquad \text{otherwise} \end{cases}. \tag{8.1}$$

This simple neuron model can also be written in a more generic form, which we will call the **perceptron**, where the output is calculated from the weighted summed input with a grain function g,

$$y(\mathbf{x};\mathbf{w}) = g(\mathbf{w}^T\mathbf{x}). \tag{8.2}$$

The threshold perceptron is then defined given by a threshold gain function, which is can also be written in a more compact way when using the notation of the Heavisde step function

$$g(x) = \theta(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{8.3}$$

The Heaviside step function is here first example of a non-linear function that transformed the sum of the weighted input. While we called this the gain function, this function is also called the transfer function or the output function in the neural network community. We will see that this is the principle way to introduce the important non-linearities in the models.

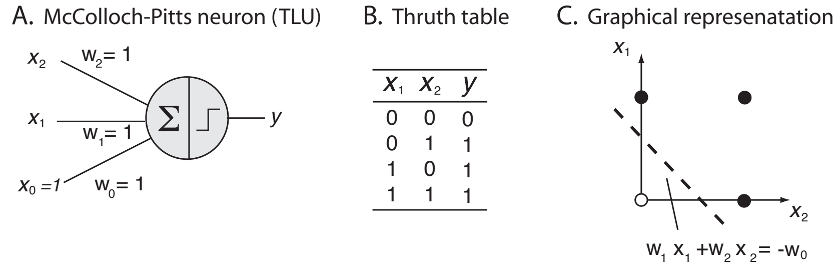

Fig. 8.2 Representation of the boolean OR function with a McCulloch-Pitts neuron (TLU).

The argument that McCulloch and Pitts brought forward with this model is that neurons can implement logical functions. This is shown illustrated in Fig. 8.2. For this we use a threshold perceptron with three inputs. One of these inputs is set to a constant so that the associated weight represents the threshold value or bias. This is equivalent to the trick used in linear regression. The boolean OR function is represented by a truth table in the figure and also with the graph on the right. If either of the inputs x_1 and x_2 has a value one, the solid dots in the graph representing true, then this function is one (true). Only if both inputs are zero is the output zero, representing false and shown with open circle in the graph. The perceptron specifies the function of eq.8.1, this means that the decision line is given by the equation

$$w_0 + w_1 x_1 + w_2 x_2 = 0. \tag{8.4}$$

As can be seen from the graph, there are many solutions to this equation, and we could discuss which one would be the best. For example, we could add the conditional constraint that the solution should be farthest away from any data point, which is called a large margin classifier. This is the approach taken in Support Vector Machines, which represents additional considerations that we will discuss in later chapters.

8.2 Sigmoidal perceptron and the delta learning rule

The next step is of course to find the parameters with an appropriate learning algorithm. However, since the gain function is not differentiable, we will wait and discuss first the revealing relation of such simple neural networks to our previous probabilistic interpretation, specifically to logistic regression. This is simply achieved by choosing as grain function the logistic function

$$g(x) = \frac{1}{1 + e^{-x}}, \tag{8.5}$$

Which is somewhat a smooth approximation of the threshold function. The **logistic perceptron** is a specific model or parameterized hypothesis functions given by

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}}} \tag{8.6}$$

A graphical representation of this model is shown in Figure 8.3A. At this point we see that we are back at a logistic regression model where the output of this function actually

represents probability of class membership. Historically it was of course the other way around. The logistic perceptron was introduced already in the late 1950s by Frank Rosenblatt who later wrote one of the first comprehensive books about perceptrons.

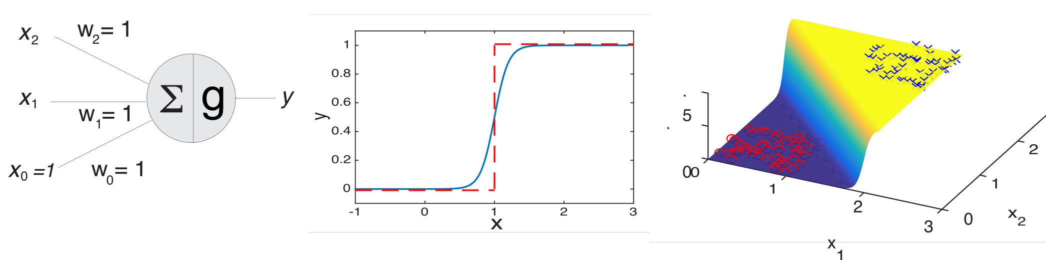

Fig. 8.3 A) Graphical representation of a perceptron with three input channels of which one is constant. B) The logistic function with different slopes and offset parameters. C) Illustration of the logistic regression problem with two input features.

At this point we know already that one solution to the learning problem is a gradient descent on a loss function, and while we have argued that the loss function should be chosen carefully from a probabilistic interpretation, we follow the older traditional derivation by using the mean square error function

$$E(\mathbf{w}) = \frac{1}{2N} \sum_i \left(y^{(i)} - y(\mathbf{x}^{(i)}; \mathbf{w}) \right)^2. \tag{8.7}$$

Using the $1/2$ in this formula is pure convention. To find this minimum we use again gradient descent, and we merely go threw these steps again as an reviewing exercise. That is update of the weight values is given by

$$w_j \leftarrow w_j - \alpha \frac{\partial E}{\partial w_j}, \tag{8.8}$$

where α is a learning parameter. We can now calculate the gradient in order to provide a formula that can be implemented with Python. For this we have to recall two rules from calculus namely that the derivative of an exponent function is

$$\frac{\mathrm{d}}{\mathrm{d}x} x^n = n x^{n-1}. \tag{8.9}$$

The derivative of the Euler function is

$$\frac{\mathrm{d}}{\mathrm{d}x} e^x = e^x, \tag{8.10}$$

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

$$\frac{\mathrm{d}}{\mathrm{d}x} f(g(x)) = \frac{\mathrm{d}f}{\mathrm{d}g} \frac{\mathrm{d}g}{\mathrm{d}x}. \tag{8.11}$$

With these rules we get

$$\frac{\partial E}{\partial w_j} = \frac{1}{N}\sum_i \left((y^{(i)} - y(\mathbf{x}^{(i)};\mathbf{w}))(-1)\frac{\partial y}{\partial w_j} \right). \tag{8.12}$$

The derivative of our model with respect to the parameters is

$$\frac{\partial y}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{1+e^{-\sum_i w_i x_i}} = \frac{e^{-\sum_i w_i x_i}}{(1+e^{-\sum_i w_i x_i})^2}\frac{\partial \sum_i w_i x_i}{\partial w_j}. \tag{8.13}$$

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

$$\frac{e^{-\sum_i w_i x_i}}{(1+e^{-\sum_i w_i x_i})^2} = y(1-y), \tag{8.14}$$

and hence

$$\frac{\partial y}{\partial w_j} = y(1-y)x_j. \tag{8.15}$$

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

$$w_j \leftarrow w_j - \alpha\frac{1}{N}\sum_i \left((y^{(i)} - y(\mathbf{x}^{(i)};\mathbf{w}))y(\mathbf{x}^{(i)};\mathbf{w})(1-y(\mathbf{x}^{(i)};\mathbf{w}))x^{(i)_j} \right) \tag{8.16}$$

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

$$\delta(\mathbf{x}^{(i)};\mathbf{w}) = (y^{(i)} - y(\mathbf{x}^{(i)};\mathbf{w}))y(\mathbf{x}^{(i)};\mathbf{w})(1-y(\mathbf{x}^{(i)};\mathbf{w})), \tag{8.17}$$

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

$$\delta = (y^{(i)} - y)y(1-y) \tag{8.18}$$

We can thus write the learning rule as

$$w_j \leftarrow w_j + \alpha\frac{1}{N}\sum_i \left(\delta(\mathbf{x}^{(i)};\mathbf{w})x_j^{(i)} \right) \tag{8.19}$$

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, we could use a one training example as a time, $(\mathbf{x}^{(i)}, y^{(i)}$, 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.

8.3 Multilayer perceptron (MLP)

We saw that a threshold perceptron can realize the Boolean OR function. However, consider the XOR function which is one only if either one of the input features are one. The XOR function is not linear separable. This has lead to the demise of perceptrons in the 1970s although it was clear that more elaborate perceptrons could. Rosenblatt did indeed start to build networks out of the simple neuron models, and he even started to build neural computers based on such perceptrons. The main problem was actually that learning was thought to be problematic with these more elaborate structures, and these models became only popular again when effective training was possible.

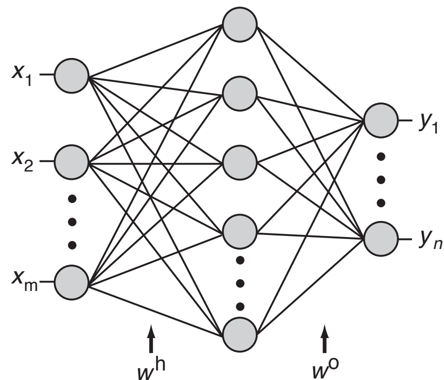

Fig. 8.4 The standard architecture of a feedforward multilayer network with one hidden layer, in which input values are distributed to all hidden nodes with weighting factors summarized in the weight matrix $\mathbf{w}^{\mathrm{h}}$. The output values of the nodes of the hidden layer are passed to the output layer, again scaled by the values of the connection strength as specified by the elements in the weight matrix $\mathbf{w}^{\mathrm{o}}$.

We will not consider networks of simple sigmoidal neurons to build up what are commonly called neural networks. We will first consider a network structure as shown

in Figure 8.4. The network as a layer of m input nodes, a layer of h hidden nodes, and a layer of n output nodes, The input layer is merely just represnting the input values, while the hidden and output layer do active calculations specified before with the sigmoidal neuron equation 8.6. 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

$$\mathbf{y} = g(\mathbf{w}^{\mathrm{o}} g(\mathbf{w}^{\mathrm{h}} \mathbf{x})). \tag{8.20}$$

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

$$\mathbf{y} = g(\mathbf{w}^{\mathrm{o}} g(\mathbf{w}^{\mathrm{h3}} g(\mathbf{w}^{\mathrm{h2}} g(\mathbf{w}^{\mathrm{h1}} \mathbf{x})))). \tag{8.21}$$

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 8.21 then simplifies to

$$\begin{aligned} \mathbf{y} &= \mathbf{w}^{\mathrm{o}} \mathbf{w}^{\mathrm{h3}} \mathbf{w}^{\mathrm{h2}} \mathbf{w}^{\mathrm{h1}} \mathbf{x} \\ &= \tilde{\mathbf{w}} \mathbf{x}. \end{aligned} \tag{8.22}$$

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 8.22 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. 8.4, 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

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

$$ \tag{8.23} $$

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

$$ \mathbf{y}^h = g(\mathbf{w}^{\mathrm{h}}\mathbf{x})). \tag{8.24} $$

Then we can continue with our derivative as,

$$ \begin{aligned} \frac{\partial E}{\partial w_{ij}^{\mathrm{out}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{out}}} \sum_k \left(\mathbf{y}^{(k)} - g(\mathbf{w}^{\mathrm{out}}\mathbf{y}^h)\right)^2 \\ &= -\sum_k g'(\mathbf{w}^h\mathbf{x}^{(k)})(y_i^{(k)} - y_i)y_j^h \\ &= \delta_i^{\mathrm{out}} y_j^{\mathrm{h}}, \end{aligned} \tag{8.25} $$

Eqn 8.25 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

$$ \begin{aligned} \frac{\partial E}{\partial w_{ij}^{\mathrm{h}}} &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{h}}} \sum_k (\mathbf{y}^{(k)} - \mathbf{y})^2 \\ &= \frac{1}{2}\frac{\partial}{\partial w_{ij}^{\mathrm{h}}} \sum_k \left(\mathbf{y}^{(k)} - g(\mathbf{w}^{\mathrm{out}} g(\mathbf{w}^{\mathrm{h}}\mathbf{x}^{(k)})\right)^2. \end{aligned} \tag{8.26} $$

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

$$ \frac{\partial E}{\partial w_{ij}^{\mathrm{h}}} = \delta_i^{\mathrm{h}} x_j, \tag{8.27} $$

when we define the delta term of the hidden term as

$$ \delta_i^{\mathrm{h}} = g^{\mathrm{h}\prime}(h_i^{\mathrm{in}}) \sum_k w_{ik}^{\mathrm{out}} \delta_k^{\mathrm{out}}. \tag{8.28} $$

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 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 8.1. Much more common with large data sets are **mini-batches** as discussed later in more detail.

Let us illustrate a basic multilayer perceptron implementation in python on the XOR problem. The program starts with defining the training problem (the training

Table 8.1 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: $\Delta w_{ij}^l = \alpha \delta_i^l y_j^{l-1}$

data set) in feature arrays `X` and desired label vector `Y`. We then introduce and initialize some variables. The variables `h` and `y` are the activations of the hidden and the output nodes, respectively, and the weight parameters are named `wh` for the weight leading into the hidden nodes and `wo` for the weights leading into the output node. `dwh` and `dwo` are the changes (gradients) of the weights, and `dh` and `do` are the respective delta terms. The variable `error` stores the absolute difference between the desired output and the actual output to plot the learning curve.

```
# MLP with backpropagation learning of XOR function
from pylab import *

# training vectors (Boolean AND function and constant input)
X=array([[0,0,1,1],[0,1,0,1],[1,1,1,1]])
Y=array([0,0,0,1])

# model specifications
Ni=3; Nh=3; No=1;

#parameter and array initialization
Ntrials=2000
h=zeros(Nh); y=zeros(No)
wh=randn(Nh,Ni); wo=randn(No,Nh)
dwh=zeros(wh.shape); dwo=zeros(wo.shape)
dh=zeros(Nh); do=zeros(No)
error=zeros(Ntrials)

for trial in range(Ntrials):
    #randomly pick training example
    pat=randint(4); x=X[:,pat]

    #calculate prediction
    h=1/(1+exp(-wh@x))
    y=1/(1+exp(-wo@h))
```

```
    # delta term for each layer (objective function error)
    do=y*(1-y)*(Y[pat]-y)
    dh=(h*(1-h))*(wo.transpose()@do)

    # update weights with momentum
    dow=0.9*dwo+outer(h,do).T
    wo=wo+0.1*dwo
    dwh=0.9*dwh+outer(dh,x)
    wh=wh+0.1*dwh

    # test all pattern
    h=1/(1+exp(-wh@X))
    y=1/(1+exp(-wo@h))
    error[trial]=error[trial]+sum(abs(y-Y))

plot(error)
```

We then iterate over trials. In each trial we pick a random example from the training set, calculate the output, calculate the delta terms, update the weights and then test all patterns. This code is very compact with matrix notations. In general is it useful to think about the layers of the neural network to perform operations such as building the dot product between the input vector and the weight matrix. This will largely help when moving to other operations in the following chapter. However, we can of course always write this operation component-wise, which can help clarify the calculation. For example the last section of testing all patterns can be calculated component wise as given below.

```
# test all pattern
for pat in range(4):
    x=X[:,pat]
    #calculate prediction
    for ih in range(Nh):  #for each hidden node
        sumInput=0
        for ii in range(Ni):  #loop over input features
            sumInput=sumInput+wh[ih,ii]*x[ii]
        h[ih]=1/(1+exp(-sumInput))
    for io in range(No):  #for each output node
        sumInput=0;
        for ih in range(Nh):  #loop over inputs from hidden
            sumInput=sumInput+wo[io,ih]*h[ih];
        y[io]=1/(1+exp(-sumInput))
```

An example learning curve is shown in Fig. 8.5A. Since this is an online version of the algorithm with weights update after a single training pattern, the error fluctuates and and rise sometimes. However, overall the error is getting smaller, indicating that some learning takes place. The error does however not reach zero. This comes from the fact that we use a sigmoid function which approaches a value of one only asymptotically. However, we can introduce another post-processing step in which we treat outputs

over 0.5 as 1 and 0 otherwise. We can simply implement this by rounding the y value of the output before determining the error. The corresponding learning curve is shown Fig. **??**B. This demonstrates that this MLP can solve the boolean XOR function.

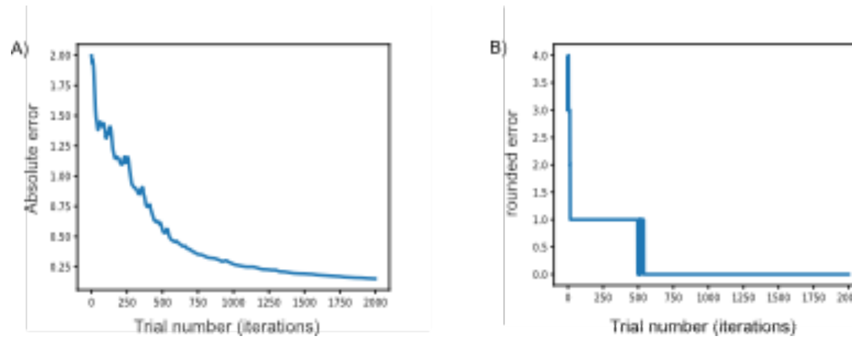

Fig. 8.5 Learning curve of an MLP trained on the boolean XOR function. The graph of on the left hand side shows the learning curve of the absolute difference between the desired output and the sigmoidal activation of the output node. The graph on the right shows an example of the performance when we use the rounded value of the output node as prediction.

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,

$$L \propto \left(y - g(\mathbf{w}^T x)\right)^2 \tag{8.29}$$

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,

$$L \propto \left(y - \mathbf{w}^T \phi(\mathbf{x})\right)^2. \tag{8.30}$$

The corresponding quadratic optimization problem can be solved much more efficiently. This line of ideas are further developed in support vector machines briefly later. I point this out here to stress that basic neural networks are not always optimal in the way they are regularly implemented and that variations are possible which we do not discuss at this point. Also, 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 representational learning discuss later.

8.4 Probabilistic and Stochastic Neural Networks

We have previously argued that machine learning is most powerful when describing uncertainties in the world and hence when using probabilistic models instead of just function models, and neural networks seems rather just being a more fancy way of

writing complex functions. Also, we argued with Bayesian models that a careful causal decomposition of a problem and hence a more specific model is a key aspect of being able to learn from data. We here try to reconcile these views by discussing how MLPs fit into this probabilistic picture. We will thereby touch on three aspects, that of generalistic versus specific models, that of representing probability functions, and that of stochastic models.

Let us start with the first issues, that of building specific functional descriptions for a specific problem that we want to model versus using neural networks that rather seems to 'fit it all'. We already discussed Wolpert's "No free lunch" theorem, and it is clear that we should not expect that high dimensional machines are optimal compared to models that are close to the true underlying world model. However, in practice there is also the problem of finding this specific model, and building a very general machine is one way of moving forward. Our strategy is thereby to use data and good regularizations (useful priors) to restrict the model from data. The recent success of deep learning is a testimony that this strategy can work in conjunction with "big data". However, the challenge of using this approach is greater in situation where there is a limited supply of data.

What we just discussed is the same if the model describes a functional relation of a probabilistic relation, so let us now move to the second important subject, how neural networks compare with our desire to build probabilistic models. We actually started to make this connection in Chapter 5 with viewing classification as a logistic regression problem. The essence there was that a logistic function describes the probability that a data point belongs to one class versus another. In the same way we are now treating the neural network itself as a high-dimensional function that describes a probability $p(y|x;w)$. To be more clear, we only need to assume that the output node represents this probability, and we will later argue that the computational layers leading up to it can be viewed as a transformation of features that lead to a simple probabilistic function on which we can base our decision of class membership in classification.

In order to train the network we have already introduced the log likelihood principle. We are now taking the opportunity to show an alternative way of derive the learning rule which is more common in the neural network community. For this we assume that the true nature of data are governed by the unknown density function $q(y|\mathbf{x})$. The neural network model represents the probability $p(y|\mathbf{x};\mathbf{w})$ which we hope to be a good approximation of $q(y|\mathbf{x})$. The negative log probability of the given label under the current model is then given by

$$H(p,q) = -\sum_y p(y) \log q(y), \tag{8.31}$$

where we omitted some arguments to see more clearly the structure. This quantity is called the **cross entropy**. The term $p \log q$ is in essence the same as the log likelihood estimate, which we want to maximize. That is, we want to maximize the log probability of the data given the labels. Since the cross entropy is the negative of this, maximizing the log probability of the data given the labels is equivalent of minimizing the cross entropy. Furthermore, the cross entropy is related to the **KL-divergence** by

$$H(p,q) = H(p) + KL(p||q), \tag{8.32}$$

and since changing model parameters do not effect the true data, minimizing the cross entropy is equivalent to minimizing the KL-divergence. Both starting points are common ion the literature, Many deviations of the learning rule in neural network start with either of these formulations, and we hope we made it clear that these are equavalent and that both are closely related to the maximum (log) likelihood principle.

Let us apply this to a 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

$$H(p, q|\mathbf{x}; \mathbf{w}) = -p(y = 0|\mathbf{x}; \mathbf{w}) \log q(y = 0|\mathbf{x}) - p(y = 1|\mathbf{x}; \mathbf{w}) \log q(y = 1|\mathbf{x}) \tag{8.33}$$

or

$$H(p, q) = -p \log q - (1 - p) \log (1 - q) \tag{8.34}$$

for short. We now assume again a sigmoidal function for the probability of the class membership around a decision point,

$$p(\hat{y} = 0|\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{xw}}} \tag{8.35}$$

$$1 - p(\hat{y} = 0|\mathbf{x}; \mathbf{w}) = \frac{1 + e^{-\mathbf{xw}} - 1}{1 + e^{-\mathbf{xw}}} = \frac{1}{1 + e^{\mathbf{xw}}}. \tag{8.36}$$

Hence, minimizing the cross-entropy between the network output, $p(\hat{y})$, and the given labels, y, is given by minimizing the Loss function

$$L = -y \log p(\hat{y}) - (1 - y) \log(1 - p(\hat{y})) \tag{8.37}$$

$$= y \log(1 + e^{-\mathbf{xw}}) + (1 - y) \log(1 + e^{\mathbf{xw}}) \tag{8.38}$$

The derivative of the function is

$$\frac{dL}{d(\mathbf{xw})} = y \frac{-e^{-\mathbf{xw}}}{1 + e^{-\mathbf{xw}}} + (1 - y) \frac{e^{\mathbf{xw}}}{1 + e^{\mathbf{xw}}} \tag{8.39}$$

$$= -y \frac{1}{1 + e^{\mathbf{xw}}} + (1 - y) \frac{1}{1 + e^{-\mathbf{xw}}} \tag{8.40}$$

$$= -y(1 - p(\hat{y})) + (1 - y)p(\hat{y}) \tag{8.41}$$

$$= -y + yp(\hat{y}) + p(\hat{y}) - yp(\hat{y}) \tag{8.42}$$

$$= p(\hat{y}) - y, \tag{8.43}$$

from which we can derive the gradient we need for the learning rule

$$\frac{dL}{d(\mathbf{w})} = \frac{dL}{d(\mathbf{xw})} \frac{d\mathbf{xw}}{d(\mathbf{w})} = (p(\hat{y}) - y)\mathbf{x}. \tag{8.44}$$

This expression has again the now familiar form of the perceptron learning rule.

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

$$p(\hat{y} = i|\mathbf{x}; \mathbf{w}) = \frac{e^{\mathbf{x}\mathbf{W}_i}}{\sum_{j=1}^{N} e^{\mathbf{x}\mathbf{W}_j}}, \tag{8.45}$$

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,

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

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

$$\frac{dL}{d(\mathbf{xW})} = p(\hat{\mathbf{y}}) - \mathbf{y} \tag{8.46}$$

$$\frac{dL}{d\mathbf{x}} = (p(\hat{\mathbf{y}}) - \mathbf{y})\mathbf{W}^T \tag{8.47}$$

$$\frac{dL}{d\mathbf{W}} = \mathbf{x}^T(p(\hat{\mathbf{y}}) - \mathbf{y}) \tag{8.48}$$

Thus, in summary. A neural network, regardless of having many or few layers, is in this case an implementation of a probabilistic model for classification or logistic regression when the output layer chosen to be a softmax function and the loss function is the negative cross entropy.

We started this chapter on a biological note, and I would like to end it on one. We have just discussed how a neural network represents a probabilistic model function. However, one can even go a step further and view the neural model itself as a stochastic system. For example, we could change the gain function of the artificial neuron to a stochastic rule, like

$$p(y=1|\mathbf{x};\mathbf{w}) = \frac{1}{1+e^{\mathbf{xw}}}. \tag{8.49}$$

While this look equivalent to eq.8.35, the difference is now that the output of the neuron is $y = 1$ with this probability and $y = 0$ with the inverse probability. Hence, this neuron is a binary neuron with only one binary states, on and off. This seems better to resemble the spiking nature of real neurons. If the neurons are stochastic nodes in a network, then we can view this network itself as a Bayesian network where the weights form represent some form of conditional probability to influence the firing of the receiving neuron. There is strong evidence that real neurons have a stochastic nature. An important example of such stochastic networks that have plaid an instrumental role in the development of deep networks is the Boltzmann machine. We will later visit this fascinating area in more detail.