Chapter 3

Feed-forward Neural Networks

This chapter will deal with the feed-forward type of artificial neural networks. We will start with the simple perceptron and then move onto multilayer perceptrons.

3.1 The Perceptron

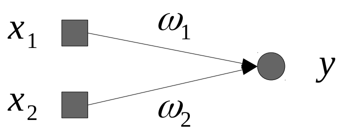
The simple perceptron has no hidden layer, only an input layer and one output node. It is illustrated in fig. (2.2). For the output node, we call the activation function φ_o. Given an input vector $\mathbf{x}$ we can then compute the output y,

$$y(\mathbf{x}, \boldsymbol{\omega}, b) = \varphi_o(\sum_{k=1}^{K} \omega_k x_k + b) = \varphi_o(\boldsymbol{\omega}^T \mathbf{x} + b). \tag{3.1}$$

The perceptron is a simple mapping $y = F(\mathbf{x}, \boldsymbol{\omega}, b)$ where the weights $\boldsymbol{\omega}$, the bias b, and the (output) activation function $\varphi_o()$ determine the type of mapping.

3.1.1 Interpretations

The argument a to φ_o depends on the input vector $\mathbf{x}$ only through a scalar product $\boldsymbol{\omega}^T\mathbf{x}$. Thus, the countour surfaces of a become parallel hyperplanes in $\mathbf{x}$-space. The direction of $\boldsymbol{\omega}$ defines the orientation of the contour surfaces, and the magnitude of $\boldsymbol{\omega}$ determines the distance between surfaces for two different values of a. Finally, the bias b determines which contour surface that corresponds to $a = 0$. This is illustrated for a simple 2-dimensional case in fig. (3.1).



$$a = \omega_1 x_1 + \omega_2 x_2 + b, \quad y = \varphi_o(a)$$

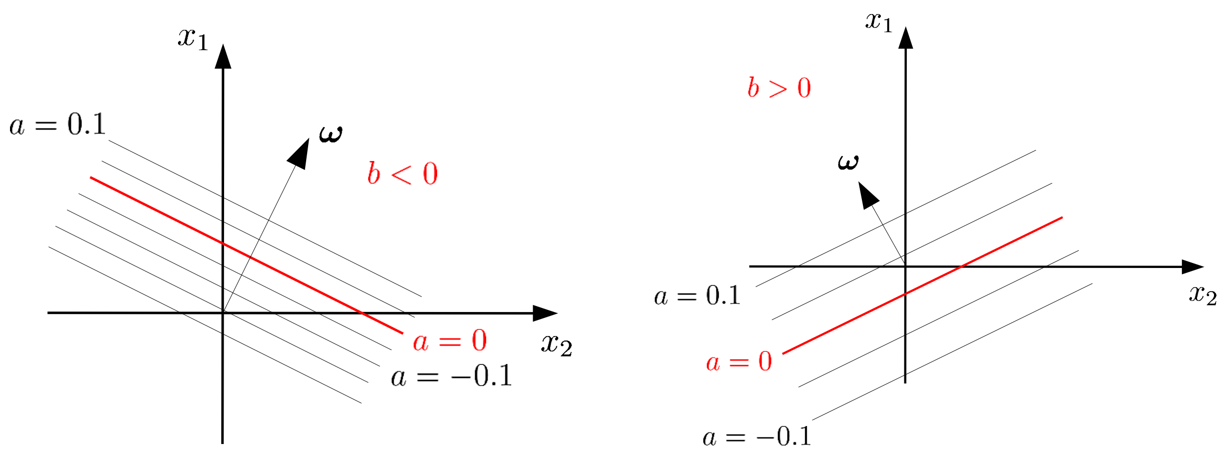


Figure 3.1: The countour surfaces for the output argument a. Large $\boldsymbol{\omega}$ and negative bias to the left, small $\boldsymbol{\omega}$ and positive bias to the right.

This is of course not the only possible way to define contour surfaces for a (see for example the part on "radial basis functions" later). However, one appealing property of neural networks is that in order to solve more complex tasks, you do not need to make your nodes more complicated – instead you add more nodes. We will therefore restrict our discussion to what very simple perceptrons can do.

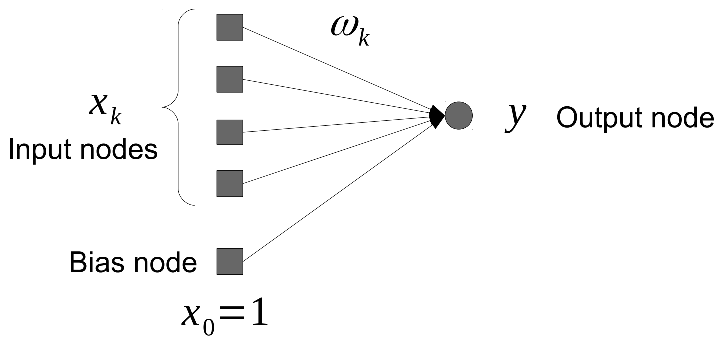


Figure 3.2: Including the bias term as an extra weight ω_0.

It is often convenient to add an extra "bias input" $x_0 = 1$ to all samples, and represent the bias parameter b as a weight ω_0, as shown in fig. (3.2). If so, ω_0 and x_0 are usually absorbed into the vecctors $\boldsymbol{\omega}$ and $\mathbf{x}$, respectively, and the output function is written

$$y(\mathbf{x}, \boldsymbol{\omega}) = \varphi_o(\sum_{k=1}^{K} \omega_k x_k + \omega_0) = \varphi_o(\sum_{k=0}^{K} \omega_k x_k) = \varphi_o(\boldsymbol{\omega}^T \mathbf{x}). \tag{3.2}$$

Be mindful that the vectors $\mathbf{x}$ and $\boldsymbol{\omega}$ therefore may have slightly different meaning in different context, for example if bias terms are explicitly added as arguments to activation functions or not.

3.1.2 Regression (function approximation)

We start to look at a perceptron with a single linear unit, i.e.

$$\varphi_o(a) = a.$$

The output from the perceptron is then just a linear combination of the inputs, $y = \boldsymbol{\omega}^T \mathbf{x}$ (including bias). This unit is often used in function approximation, where we hope to find a linear function $y(\mathbf{x})$ such that $d_n = y(\mathbf{x}_n)$ for our available patterns n. In general, there is no such linear function. Instead, it is common to look for an approximation that minimizes the mean squared error (MSE), where by convention we introduce an extra factor $\frac{1}{2}$,

$$E(\boldsymbol{\omega}) = \frac{1}{2N} \sum_{n=1}^{N} (d_n - y(\mathbf{x}_n))^2. \tag{3.3}$$

Before we discuss how our perceptron could solve this, we note that there actually is an exact solution. The error has a stagnation point if all $\frac{\partial E}{\partial \omega_k} = 0$, which implies

$$0 = \frac{1}{N} \sum_{n=1}^{N} (y(\mathbf{x}_n) - d_n) \frac{\partial y(\mathbf{x}_n)}{\partial \omega_k} = \frac{1}{N} \sum_{n=1}^{N} (\sum_{i=0}^{K} x_{ni} \omega_i - d_n) x_{nk}, \quad \forall k. \tag{3.4}$$

If we treat x_{nk} as elements in a matrix $\mathbf{X}$, while d_n and ω_k are elements in column matrices $\mathbf{d}$ and $\boldsymbol{\omega}$, respectively, this can be written as $0 = \mathbf{X}^T \mathbf{X} \boldsymbol{\omega} - \mathbf{X}^T \mathbf{d}$. Typically, $\mathbf{X}$ does not have an inverse (it need not even be square), but $\mathbf{X}^T \mathbf{X}$ is square and symmetric, so the solution can most often be found as

$$\boldsymbol{\omega} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{d}.$$

Remark 1: $\frac{1}{N} \mathbf{X}^T \mathbf{X}$ is called the *correlation matrix* for the inputs $\mathbf{x}_n$.
Remark 2: $(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T$ is called the *pseudo inverse* of $\mathbf{X}$.
Remark 3: If $\mathbf{X}^T \mathbf{X}$ lacks an inverse, there are still solutions, but they are not unique. Finding one of them typically involves diagonalizing $\mathbf{X}^T \mathbf{X}$.

3.1.3 Gradient Descent Learning

A linear regression problem can be formulated as finding a weight vector $\boldsymbol{\omega}$ that mimimizes the mean squared error $E(\boldsymbol{\omega})$, given a dataset with N patterns. This can be achieved by starting with random weights and change them in small steps towards the solution, as follows:

1. Initiate all ω_k with small random numbers.
2. Define a small "learning rate" η.
3. Repeat until convergence
   - (a) For each pattern n, compute the output
     $y_n = y(\mathbf{x}_n)$
     and the difference
     $\delta_n = \frac{1}{N}(d_n - y_n)$.
   - (b) Update the weights ω_k according to
     $\omega_k \to \omega_k + \eta \ \sum_n \delta_n x_{nk}$.

Figure 3.3: Gradient descent learning for the linear perceptron with a summed squared error function.

Since

$$\frac{\partial E}{\partial \omega_k} = \sum_n \frac{\partial E}{\partial y_n} \cdot \frac{\partial y_n}{\partial a_n} \cdot \frac{\partial a_n}{\partial \omega_k} = \sum_n \delta_n \cdot 1 \cdot x_{nk},$$

the gradient descent minimization can be summarized with the following update rule for a given weight ω_k:

$$\Delta\omega_k = -\eta \frac{\partial E}{\partial \omega_k} \quad \Rightarrow \quad \Delta\boldsymbol{\omega} = -\eta \nabla_{\boldsymbol{\omega}} E.$$

Thus, the error is minimized by taking small steps in negative gradient direction (see figure 3.4).

Stochastic Gradient Descent

The mean squared error is just the mean of errors E_n for individual patterns,

$$E = \frac{1}{N} \sum_n E_n, \quad E_n(\boldsymbol{\omega}) = (d_n - y(\mathbf{x}_n))^2.$$

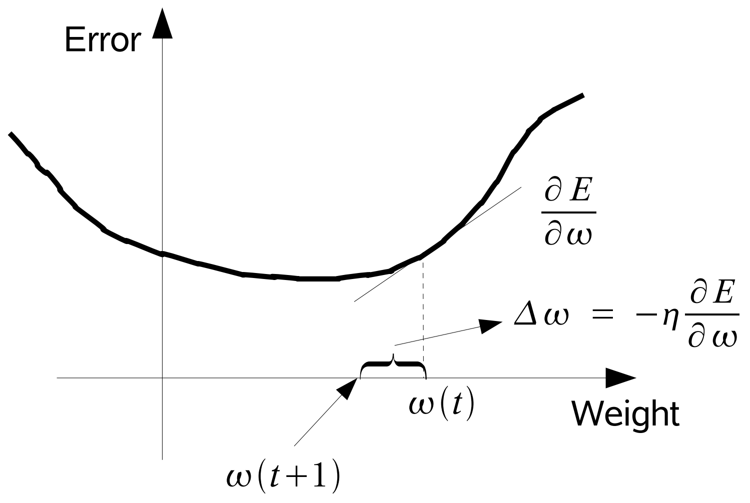

Error
$\frac{\partial E}{\partial \omega}$
$\Delta\omega = -\eta \frac{\partial E}{\partial \omega}$
$\omega(t)$
Weight
$\omega(t+1)$

Figure 3.4: Illustration of the gradient descent method.

This means that the weight update can be written as

$$\Delta\omega_k = \frac{1}{N}\sum_{n=1}^{N} \Delta\omega_{nk}, \qquad \text{with} \qquad \Delta\omega_{nk} = -\eta \frac{\partial E_n}{\partial \omega_k}.$$

When we average the individual weight updates over all patterns in the training dataset we call it *batch updating.* A more common method, however, is to update the weights using a small number of patterns,

$$\Delta\omega_k = \frac{1}{P}\sum_{p=1}^{P} \Delta\omega_{pk},$$

where P is called the minibatch size and is typically between $10-50$. This version of gradient descent is known under the name of *stochastic gradient descent* (SGD). If the minibatch size is large enough to give a fair approximation of the gradient $\nabla_{\boldsymbol{\omega}} E$, SGD is much quicker than batch updating.

We may worry that SGD prevents us from finding the perfect solution to $\nabla_{\boldsymbol{\omega}} E = 0$, since stochastic selection of minibatches will cause our weights to wobble around in the neighbourhood of the correct solution. Later, when we discuss the problem of overtraining, we will learn that this need not be an issue, and may even be an asset!

Every time we have used one minibatch to update the weights, we have performed one *iteration.*

Typically all patterns are partitioned into minibatches so that all patterns are included in precisely one part. When all patterns have been used for weight updates once, we have peformed one *epoch.* After each epoch, a new random partitioning is usually made.

The extreme case of $P = 1$, when weights are updated after each pattern, is called *online updating.*

3.1.4 Data Pre-processing and Perceptron Initialization

There are now two small uncertanties to resolve to get started with an ANN application: what is a good random distribution for initial weights, and what is a good learning rate? The answer to these questions may depend on data, and therefore we will give a suggestion that involves some simple data transformation.

Input normalization

Usually we want to normalize the input values. Strictly this is not necessary since we can always rescale the weights to deal with different input ranges, but practically it is always a good idea. Consider the inputs x_1 and x_2 and their distributions in figure 3.5.

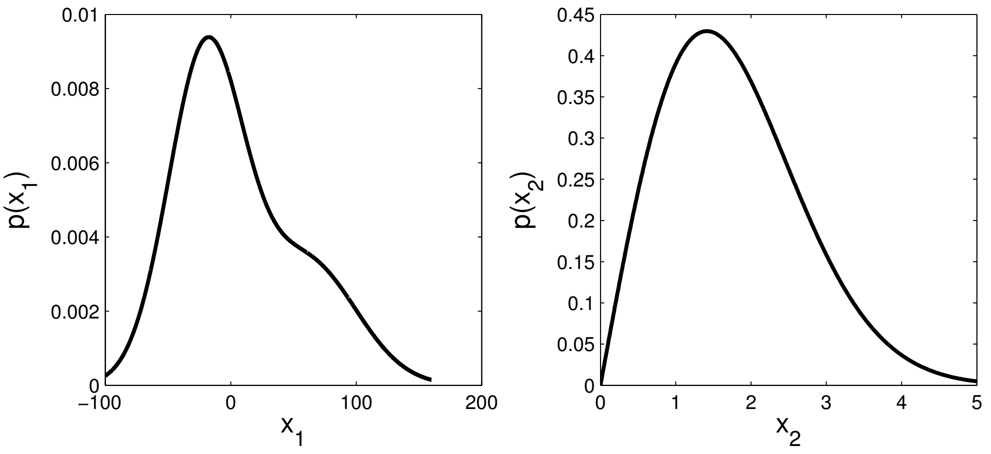


Figure 3.5: Distribution of input vectors x_1 and x_2.

If we start with small random weights and then minimize the error function it will be difficult to see the effect of the x_2 inputs since, they will "drown" in the x_i input. To avoid having to initiate the weights different for different inputs it is always better to normalize the inputs and have a common procedure for the weight initialization. There are many ways of doing input normalization, but the most common one is rescaling to zero mean and unit standard deviation for each input variable:

Compute the sample mean and (some estimate of) the standard deviation

$$\mu_k = \frac{1}{N}\sum_{n=1}^{N} x_{nk}$$

$$\sigma_k = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(x_{nk} - \mu_k\right)^2}$$

Do the transformation

$$x_{nk} \to \frac{x_{nk} - \mu_k}{\sigma_k} \quad \forall\ n, k$$

For binary inputs we make sure the two values are small integers, typically 0 and 1, or ± 1. We avoid strange values like 100 and 2000, even if they represent something for the creator of the data set.

Initial weight values

If we have performed a linear mapping of inputs to zero mean and unit variance, it is a good idea to initalize the weights from a distribution with zero mean and variance $1/K$, where K is the number of inputs. We then have typical magnitudes $|x_{nk}| \sim 1$ and $|\omega_k| \sim 1/\sqrt{K}$, so that $\sum_{k=1}^{K} \omega_k x_{nk}$ can be seen as a random walk with terms of typical size $1/\sqrt{K}$. Since we add K such terms, the sum is hopefully about 1.

If we now let the distribution for a random initial bias term ω_0 to have mean 0 and variance about 1, we can expect that many samples will have an argument to the output function, $a_n = \sum_{k=1}^{K} \omega_k x_{nk} + \omega_0$, that will be of order 1.

Target normalization and learning rate

It is a good idea to also normalize continuous target values to zero mean and unit variance, and to let binary target values take on values 0 and 1. Then, we can expect a good solution to the task to have $\mathcal{O}(1)$ outputs. As can be seen in fig. (2.3), all the output functions we consider will then need $\mathcal{O}(1)$ arguments, which we have achieved with our input transformation and weight selection.

With all these decisions, it seems that weight updates $\Delta\omega_k$ noticeably smaller than 1 are a good idea, while gradient terms $\delta_n x_{nk}$ can be expected to (at least in the beginning) be $\mathcal{O}(1)$. A good learning should then be smaller than 1, say in the range 0.01-0.1, or so.

Surely, all this is very vague and hand-waving. The random walk argument fails if strong correlations exist between inputs. Formally, this is not a problem: finding a solution is not very sensitive to our initialization, it may just take some more training. A good learning rate can be found by some preliminary trial-and-error. If the error during SGD wobbles drastically between epochs, the learning rate seems high. If the error reduction is steady but annoyingly slow, we should just try a larger learning rate.

A second virtue of unit variance target distribution is that we immediately can get a feeling for what a really bad mean squared error (MSE) is. If there are no available inputs, then all outputs y_n will be a function of just the bias to the output node, and hence equal. The common y that minimizes the MSE, $E = \frac{1}{N}\sum_n(d_n - y)^2$, is then the mean of targets $y = \langle d \rangle = \frac{1}{N}\sum_n d_n$ and the MSE essentially becomes the variance of target values $E = \frac{1}{N}\sum_n(d_n - \langle d \rangle)^2$. Thus, if we get an MSE close to 1 even when we have inputs, we may suspect that the inputs carry no useful information.

Non-linear transformations

If some data distributions are heavliy skewed, data values may become large even after transformation to zero mean and unit variance. Then, one may consider a non-linear transformation. Fig. (3.6) shows an example of the effect of a simple log transformation of a skewed input variable.

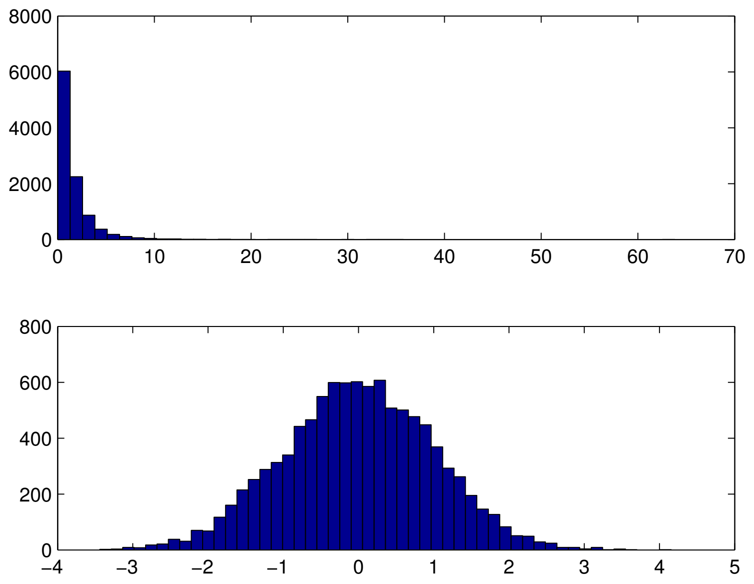


Figure 3.6: Upper plot. The original input variable distribution. Lower plot. The new distribution after a log() operation.

Notice that if you train a linear perceptron but perform nonlinear transformations first, you are formally no longer performing linear regression on the original data!

Non-linear preprocessing of input data is **very** problem dependent. Sometimes preprocessing is a very large part of working with a machine learning problem, sometimes it is only a matter the simple a normalization step.

3.1.5 Binary Classification

In binary classification, the task is to classify each input pattern $\mathbf{x}_n$ into one of two classes C_1 or C_2. An illustration of a two-class classification problem can be found in figure 3.7.

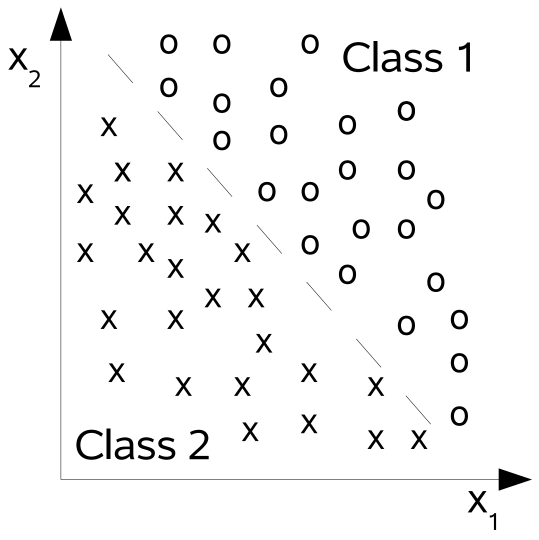


Figure 3.7: Data points represented by two coordinates (x_1 and x_2). There are two classis in this classification problem, denoted by 'o' and 'x' symbols. It turns out that this problem is solvable by the simple perceptron.

Each training pattern $\mathbf{x}_n$ we have a corresponding target d_n, which we can define to be

$$d_n = \begin{cases} 1 & \text{if} \quad \mathbf{x}_n \in C_1 \\ 0 & \text{if} \quad \mathbf{x}_n \in C_2 \end{cases} \tag{3.5}$$

We continue to discuss the single perceptron, and the goal is to find a weight vector $\boldsymbol{\omega}$ such that $y(\mathbf{x}_n) = d_n$, $\forall n$. From fig. (2.3) we see that a threshold function or a logistic function is suitable choice. In fig. (3.1) we looked at contour surfaces of the argument a to a node. In fig. (3.8) we illustrate the contour surfaces for the output of a threshold node and logistic node, respectively.

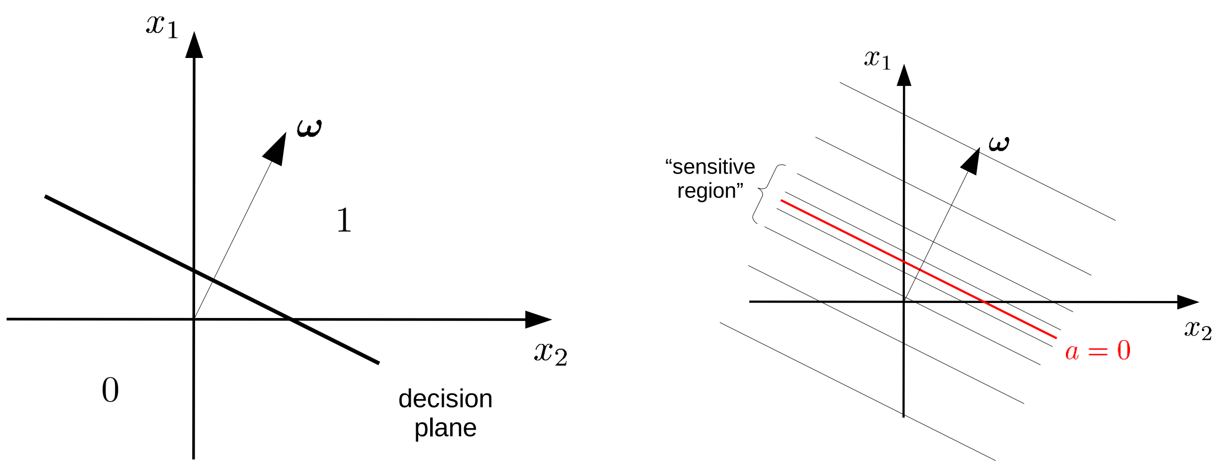


Figure 3.8: Left: For a threshold function, the are no countour surfaces, but a "decision plane" at $a = 0$. Right: The countour surfaces for the logistic node are close near $a = 0$, where the function is sensitive to the argument. Further away the output is essentially constant. In the limit of infinite $|\boldsymbol{\omega}|$, the logistic node becomes the a threshold node, with the sensitive region compressed to a single decision plane.

Linear Separability

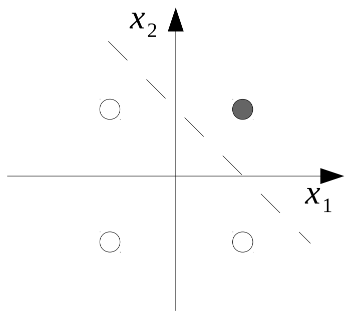
We can say that a classification problem is linearly separable if we can find a weight vector $\boldsymbol{\omega}$ such that the threshold function $\theta(a)$ solves

$$\theta(\boldsymbol{\omega}^T \mathbf{x}_n) = d_n \ \ \forall n$$

which can be rephrased as: A problem is linearly separable if we can find a hyperplane that separates the two classes.

A classical example of what the simple perceptron can handle is the 2-dimensional AND problem. It is defined as follows (in two dimensions),

x_1	x_2	d
1	1	1
1	-1	0
-1	1	0
-1	-1	0



A perceptron that solves this problem is given by $(\omega_1, \omega_2, \omega_0) = (1, 1, -1)$ (see figure 3.9).

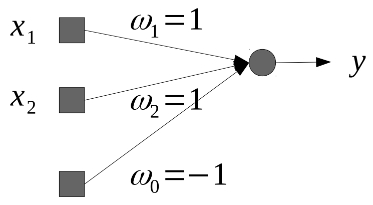


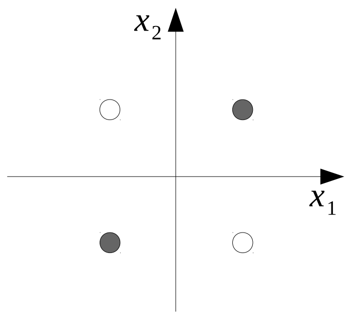
Figure 3.9: A perceptron that can solve the 2-dimensional AND problem.

For a linearly separable problem, we can use the threshold activation function and use the McCullogh-Pitts model for perceptron training, dating back to 1946:

```
1. Initiate all $\omega_k$ with small random numbers
2. Define a small "learning rate" $\eta$.
3. Repeat until all $y(\mathbf{x}_n) = d_n$
    (a) Present an input pattern $\mu$ from the training set
    (b) Compute the output $y(\mathbf{x}_\mu)$
    (c) If $y(\mathbf{x}_\mu) \neq d_\mu$ update $\boldsymbol{\omega}$ according to
        $\boldsymbol{\omega} \to \boldsymbol{\omega} + \eta\big(d_\mu - y(\mathbf{x}_\mu)\big)\mathbf{x}_\mu$
```

A problem that *cannot* be solved by a single perceptron is the XOR problem.

x_1	x_2	d
1	1	0
1	-1	1
-1	1	1
-1	-1	0



If we try to solve it using a single perceptron with a threshold function $y = \theta(x_1\omega_1+x_2\omega_2+\omega_0)$, we get the following set of equations

d		
0	$\omega_1 + \omega_2 + \omega_0 < 0$	(1)
1	$\omega_1 - \omega_2 + \omega_0 > 0$	(2)
1	$-\omega_1 + \omega_2 + \omega_0 > 0$	(3)
0	$-\omega_1 - \omega_2 + \omega_0 < 0$	(4)

Combining expression (2) and (3) results in $\omega_0 > 0$ but expressions (1) and (4) give $\omega_0 < 0$! So, there is no solution to this problem. It turns out that the XOR problem is not linearly separable and cannot be solved by a single perceptron. The single perceptron is a linear classifier!

Gradient descent for non-linear output

Even if a classification problem is not linearly separable, we may want to train a perceptron to do "as well as possible". We can then define and error that measures what we mean by "not well", and train our perceptron with gradient descent.

We can then generalize our linear regression training method to a more versatile gradient descent algorithm:

```
1. Initiate all $\omega_k$ with small random numbers
2. Define a small "learning rate" $\eta$.
3. Repeat until convergence
    (a) For each pattern $n$, compute the output
        $y_n = y(\mathbf{x}_n)$
        and the difference
        $\delta_n = -\frac{\partial E}{\partial y_n}\varphi_o'(a_n)$.
    (b) Update the weights $\omega_k$ according to
        $\omega_k \to \omega_k + \eta \sum_n \delta_n x_{nk}$.
```

Figure 3.10: Gradient descent learning (batch) for a perceptron with a differentiable error function $E(\{y_n\}, \{d_n\})$.

Again, we can use $\frac{\partial E}{\partial \omega_k} = \sum_n \frac{\partial E}{\partial y_n}\frac{\partial y_n}{\partial a_n}\frac{\partial a_n}{\partial \omega_k}$ to verify that the weight vector is updated with $\Delta\boldsymbol{\omega} = -\eta\nabla_{\boldsymbol{\omega}} E$, as it should in the gradient descent method.

It must be possible to find the derivative of the error. Thus, the "number of misclassified patterns" will not work, but we could still use the mean squared error. Most popular is the "cross-entropy error" that we will discuss later.

The activation function $\varphi_o(a)$ must also have a derivative. The threshold function cannot be used, but the logistic function $\varphi_o(a) = 1/(1+\exp(-a))$ is very suitable. The reader is encouraged to verify that the derivative then can be handily written $\varphi_o' = (1-\varphi_o)\varphi_o$.

Most often, the error E is a sum of errors for each pattern n, $E = \frac{1}{N}\sum_n E_n$. If so, we can do stochastic gradient descent, and update weights after calculating the gradient from a minibatch of samples.

3.1.6 Summary

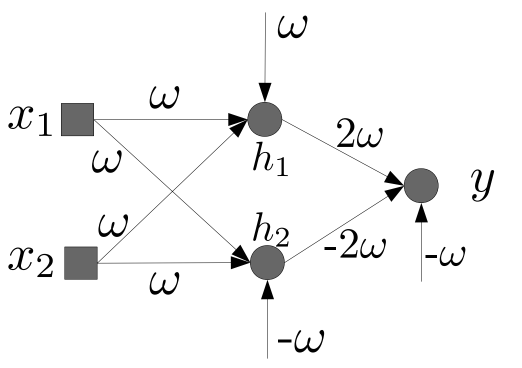
So, for the simple perceptron we can deal with a threshold activation function for classification problems, but it is more common to use the logistic activation function together with gradient descent minimization of an error. For regression problems (linear activation function) we can directly solve for the optimal weight vector, but again, we can also use gradient descent learning. A small summary of the findings for the perceptron!

Output unit	Learning algorithm	Problem type
threshold	perceptron learning algorithm	classification
logistic	gradient descent learning	classification
linear	exact solution	regression
linear	gradient descent learning	regression

3.2 The multilayer perceptron (MLP)

3.2.1 The XOR solution

The multilayer perceptron is constructed by adding one or more hidden layers of nodes. Before we continue we will show that this approach can solve the XOR problem. We will use threshold neurons both in the hidden layer and for the output node.



$$h_1 = \theta(\omega x_1 + \omega x_2 + \omega)$$
$$h_2 = \theta(\omega x_1 + \omega x_2 - \omega)$$
$$y = \theta(2\omega h_1 - 2\omega h_2 - \omega)$$

x_1	x_2	d	h_1	h_2	y
+1	+1	0	1	1	0
+1	-1	1	1	0	1
-1	+1	1	1	0	1
-1	-1	0	0	0	0
			$\overbrace{OR}$	$\overbrace{AND}$	

We can see that this one-hidden-layer MLP can solve the XOR problem for any positive value of the parameter ω. The two hidden nodes implement the AND and the OR logic, resulting in the final XOR logic.

3.2.2 The MLP architecture

The MLP can have an arbitrary number of hidden layers, but most common are one or possibly two hidden layers of nodes (see figure 3.11).

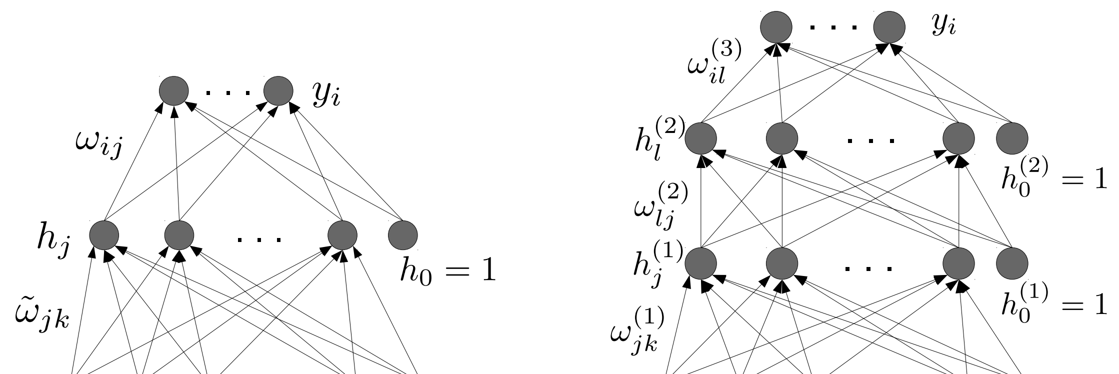


Figure 3.11: (left) A standard MLP with one hidden layer. (right) Two hidden layers.

The activation functions for the hidden nodes are usually non-linear. In fact there is no meaning in having linear activation functions in many hidden layers. The argument to the output would then become a linear combination of linear combinations, which is just one big linear combination, possible to represent with a simple perceptron.

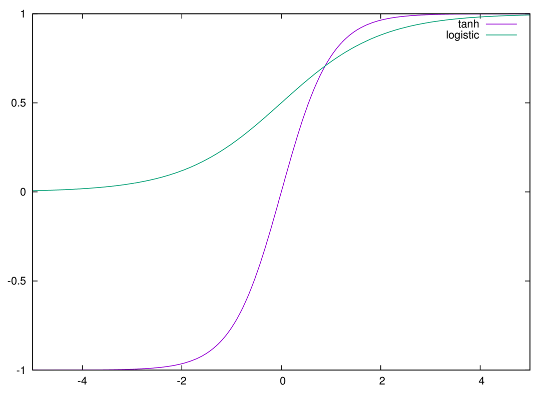
Three common non-linear activation functions are:

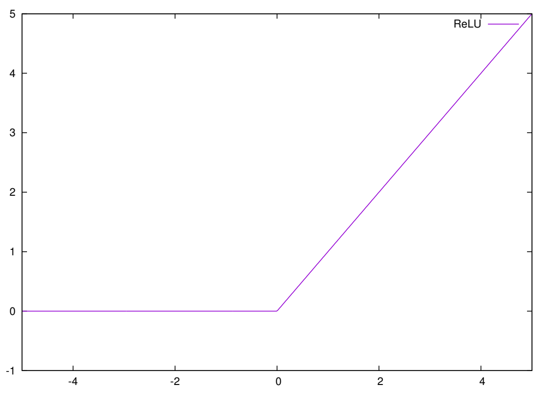
The logistic activation function

$$\varphi(a) = \frac{1}{1+e^{-a}}$$

and the tanh activation function

$$\varphi(a) = \tanh(a)$$





The rectifier activation function

$$\varphi(a) = \max(0, a)$$

For networks with a few number of hidden layers both the logistic and the tanh functions works well. However, the rectifier function is the most popular for larger networks. The output node activation function is either linear (for function approximation problems), logistic (for two-class classification problems) or a function called softmax (for multi-class classification problems, more about this later). Assuming one hidden layer the output $y_i(\mathbf{x}_n)$, given an input pattern $\mathbf{x}_n$, is calculated through a forward pass

$$\begin{aligned} y_i(\mathbf{x}_n) &= \varphi_o\Big(\sum_j \omega_{ij} h_{nj}\Big) = \varphi_o\Big(\boldsymbol{\omega}_i^T \mathbf{h}_n\Big) \quad \text{where} \\ h_{nj} &= \varphi_h\Big(\sum_k \tilde{\omega}_{jk} x_{nk}\Big) = \varphi_h\Big(\tilde{\boldsymbol{\omega}}_j^T \mathbf{x}_n\Big) \end{aligned}$$

In one expression,

$$y_i(\mathbf{x}_n) = \varphi_o\bigg(\sum_j \omega_{ij}\varphi_h\Big(\sum_k \tilde{\omega}_{jk} x_{nk}\Big)\bigg)$$

Note that the threshold neurons that are present in every layer are not explicitly shown in the equations. It is assumed that one of the input/hidden nodes serves as the threshold node (see figure 3.11).

3.2.3 How does the MLP work?

The hidden nodes, in the first hidden layer, act as a kind of feature detectors. Each hidden node has its own decision plane or sensitive region in input space. The argument to a node in the nex layer will then depend on different hidden nodes in different regions, allowing for non-flat contour surfaces to the output. See figure 3.12 for an illustration and figure 3.13 for two numerical examples.

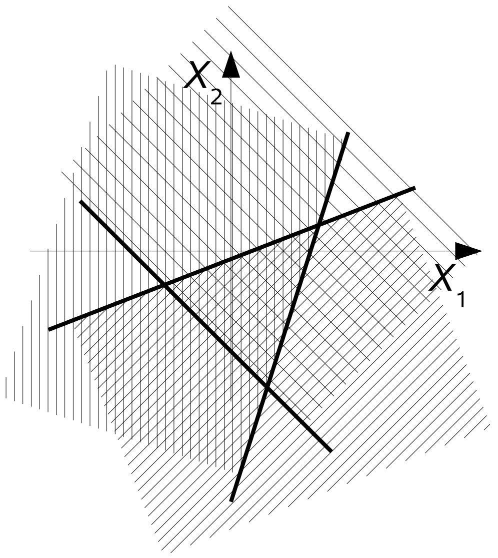
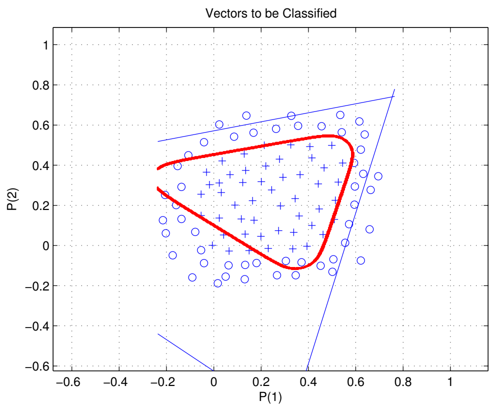


Figure 3.12: Illustration of the role played by the hidden nodes. Each thick line represent a hyperplane where the argument to the node is 0. This is the decision plane or the center of the sensitive region for that node. The shaded regions show an example of where a hidden node has positive output.

3.2.4 Learning algorithm: Back-propagation

We will now introduce back-propagation. It is often used to as a name for training an neural networks. We will however see below that back-propagation is just the way we compute gradients in an MLP. As before we will need a training dataset,

$$D = \{\mathbf{x}_n, \mathbf{d}_n\}_{n=1\cdots N}.$$



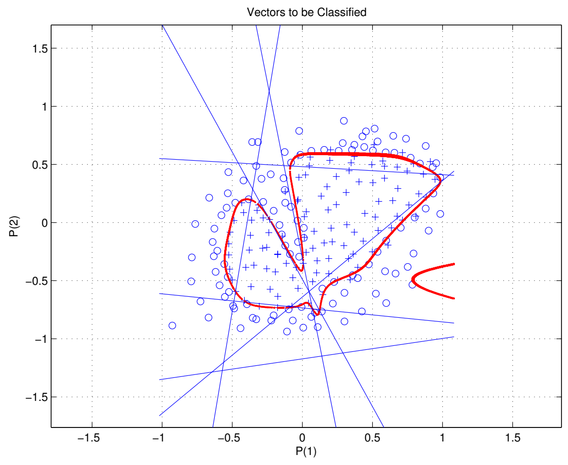


Figure 3.13: Examples of the role played by the hidden nodes. The straight line represents "hyperplanes" implemented by different hidden nodes, while the curved line is the output boundary. (Left) A 3-hidden node network. Here we can see how the 3 nodes create a closed boundary. (Right) A more complicated problem with more hidden nodes.

Here we now allow for multiple outputs with targets $\mathbf{d}_n$, where we denote d_{ni} as the target value for output i for pattern n. We also need a differentiable error function $E(\boldsymbol{\omega})$ to minimize. It could for example be a function based on the sum of mean squared errors for all outputs,

$$E(\boldsymbol{\omega}) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{i}\left(d_{ni} - y_i(\mathbf{x}_n)\right)^2 . \tag{3.6}$$

To minimize the error E (as a function of the weights) we will use the gradient descent method. Compared to the gradient descent for a simple perceptron, we now have to keep track of more indices and exploit the chain rule even further. Other than that, there is nothing new.

For the one hidden layer MLP there are two sets of weights to compute updates for, namely

$$\begin{array}{ll} \text{input-to-hidden weights} & \Delta\tilde{\omega}_{jk} = -\eta\dfrac{\partial E}{\partial \tilde{\omega}_{jk}} \\ \text{hidden-to-output weights} & \Delta\omega_{ij} = -\eta\dfrac{\partial E}{\partial \omega_{ij}} \end{array}$$

We start with the hidden-to-output weights $\Delta\omega_{ij}$. First we notice that

$$\frac{\partial E}{\partial \omega_{ij}} = \sum_n \sum_{i'} \frac{\partial E}{\partial y_{ni'}}\frac{\partial y_{ni'}}{\partial \omega_{ij}} = \sum_n \frac{\partial E}{\partial y_{ni}}\frac{\partial y_{ni}}{\partial \omega_{ij}},$$

since $\partial y_{ni'}/\partial \omega_{ij} = 0$ except when $i' = i$.

With $\frac{\partial y_{ni}}{\partial \omega_{ij}} = \frac{\partial y_{ni}}{\partial a_{ni}} \frac{\partial a_{ni}}{\partial \omega_{ij}} = \varphi_o'(a_{ni}) h_{nj}$, we can put all together as

$$\frac{\partial E}{\partial \omega_{ij}} = \sum_n \underbrace{\frac{\partial E}{\partial y_{ni}} \varphi_o' \Big(\sum_{j'} \omega_{ij'} h_{nj'} \Big)}_{\equiv -\delta_{ni}} h_{nj} \tag{3.7}$$

$$= -\sum_n \delta_{ni} h_{nj} \qquad \text{(Just like the perceptron!)}$$

To compute the input-to-hidden weights we start with

$$\begin{aligned}
\frac{\partial E}{\partial \tilde{\omega}_{jk}} &= \sum_n \frac{\partial E}{\partial h_{nj}} \frac{\partial h_{nj}}{\partial \tilde{\omega}_{jk}} = \\
&= \sum_n \left(\sum_i \frac{\partial E}{\partial a_{ni}} \frac{\partial a_{ni}}{\partial h_{nj}} \right) \frac{\partial h_{nj}}{\partial a_{nj}} \frac{\partial a_{nj}}{\partial \tilde{\omega}_{jk}} = \\
&= -\sum_n \left(\sum_i \delta_{ni} \omega_{ij} \right) \varphi_h'(a_{nj}) x_{nk}.
\end{aligned} \tag{3.8}$$

As seen the values δ_{ni} calculated for the output layer can be used to find the gradients in the earlier layer. If our MLP consists of more than one hidden layer, it is therefore convenient to define

$$\tilde{\delta}_{nj} = \varphi_h' \Big(\sum_{k'} \tilde{\omega}_{jk'} x_{nk'} \Big) \sum_i \delta_{ni} \omega_{ij}$$

and use those in a similar way for the next, even earlier layer.

Considering many patterns n in our weight update, we get as a final expression for the weight updates

$$\Delta \omega_{ij} = -\eta \frac{\partial E}{\partial \omega_{ij}} = \eta \sum_n \delta_{ni} h_{nj}$$
$$\Delta \tilde{\omega}_{jk} = -\eta \frac{\partial E}{\partial \tilde{\omega}_{jk}} = \eta \sum_n \tilde{\delta}_{nj} x_{nk}$$

The above way of updating the weights is called *back-propagation*, because once you have made a forward pass to compute the outputs, deltas are back-propagated through the layers to compute the weight updates. Back-propagation is nothing but minimizing the error function using the gradient descent method.

Just as for the simple perceptron, as long as the error is a sum of sample-specific terms, $E = \frac{1}{N} \sum_n E_n$, we can do weight-update using a sub-set of patterns.

In the literature back-propagation is often the standard method for training MLP's. We will see later that there are alternatives.

3.2.5 The universal approximation theorem

The following theorem states that a MLP can approximate any continuous function. It is important since it means that there is no fundamental constraint built into the MLP. It is an existence proof, meaning that no information is given concerning the details of the MLP. We only state the theorem here and leave the proof to a more rigorous mathematics course.

Let $\varphi(\cdot)$ be a sigmoidal function and let $f(x) \in \mathcal{C}(I_m)$ where $\mathcal{C}(I_m)$ be the set of all continuous functions defined over the m-dimensional hypercube $I_m = [0,1]^m$. For any $\varepsilon > 0$ there exists an integer N_h and a set of real constants: $\omega_j, \tilde{\omega}_{jk}, b_j$ $(j = 1, \cdots, N_h,\ k = 1, \cdots, m)$ such that

$$|F(x_1, x_2, \cdots, x_m) - f(x_1, x_2, \cdots, x_m)| < \varepsilon \quad \forall x \in I_m$$

where

$$F(x_1, x_2, \cdots, x_m) = \sum_{j}^{N_h} \omega_j \varphi\left(\sum_{k=1}^{m} \tilde{\omega}_{jk} x_k + b_j\right).$$

Note 1: $F(x)$ is a 2-layer network with $\varphi(\cdot)$ as the activation function for the hidden layer and a linear output node.

Note 2: The theorem does not tell me how many nodes that I need for a specific problem.

Note 3: The theorem also applies for classification problems.

Note 4: There exists an updated version of this theorem where also rectifier activiation functions are included.

Note 5: While an MLP with a large enough hidden layer can solve a complicated task presented by training data, the results of the MLP outside the range of training patterns can be counter-intuitive. As an example, notice the extra decision boundary outside availiable data in the right plot in fig. (3.13)!

3.2.6 Data Pre-processing and MLP Initialization

The recipe vaguely motivated for a perceptron can be inhereted for an MLP, and can be summarized:

- It is often a good idea to linearly transform all continuous inputs and outputs to zero mean and unit variance.
- If some data distributions are heavily skewed, non-linear transformations could be considered.

- It is often a good idea to let binary variables take on small integer values.
- For initialization of bias parameters, a random distribution with zero mean and about unit variance is suitable.
- For initialization of weights into a node with K input weights (sometimes called "fan-in" K), a random distribution with zero mean and variance roughly $1/K$ is suitable.
- Trial-and-error may be needed to find a good learning rate, but a value smaller than 1, but not vanishingly small, could be a good starting guess.

3.3 Target Distributions and Network Outputs

Right after we have learned about the universal approximation theorem, we should note that our data with may not represent a well-defined function $\mathbf{d}(\mathbf{x})$. The extreme case would be that we have many patterns with identical inputs $\mathbf{x}_{p_1} = \mathbf{x}_{p_2}$ but different targets $\mathbf{d}_{p_1} \neq \mathbf{d}_{p_2}$. We take a look at what a large network, able to fit a complicated function, would give in this case, if we try to minimize the mean squared error. For simplicity, we restrict to a single target value per pattern $\mathbf{d}_n \to d_n$. With the network large enough to fit all other $y(\mathbf{x}_n) = d_n$ perfectly, the only non-zero terms in the error are those with conflicting targets d_m for the same input $\mathbf{x}^*$,

$$E \propto \sum_{m=1}^{M} (d_m - y(\mathbf{x}^*))^2. \tag{3.9}$$

Our training will be completed if for every weight ω we have

$$0 = \frac{\partial E}{\partial \omega} = \frac{\partial y^*}{\partial \omega} \frac{\partial E}{\partial y^*} \propto \frac{\partial y^*}{\partial \omega} \sum_m (y^* - d_m), \tag{3.10}$$

with solution

$$y(\mathbf{x}^*) = \frac{1}{M} \sum_m d_m = \overline{d(\mathbf{x}^*)}. \tag{3.11}$$

In the limit of infinitely large dataset, the sample mean $\overline{d(\mathbf{x})}$ for every $\mathbf{x}$ approaches the conditional expectation value

$$\overline{d(\mathbf{x})} \to \langle d|\mathbf{x} \rangle, \quad N \to \infty. \tag{3.12}$$

Thus, while we cannot be sure that there is a function $d(\mathbf{x})$ for a network to find, we can always expect there to be a conditional probability distribution $p(d|\mathbf{x})$ and give the network the task to find the conditional expectation value $\langle d|\mathbf{x} \rangle$. This is illustrated in figure 3.14.

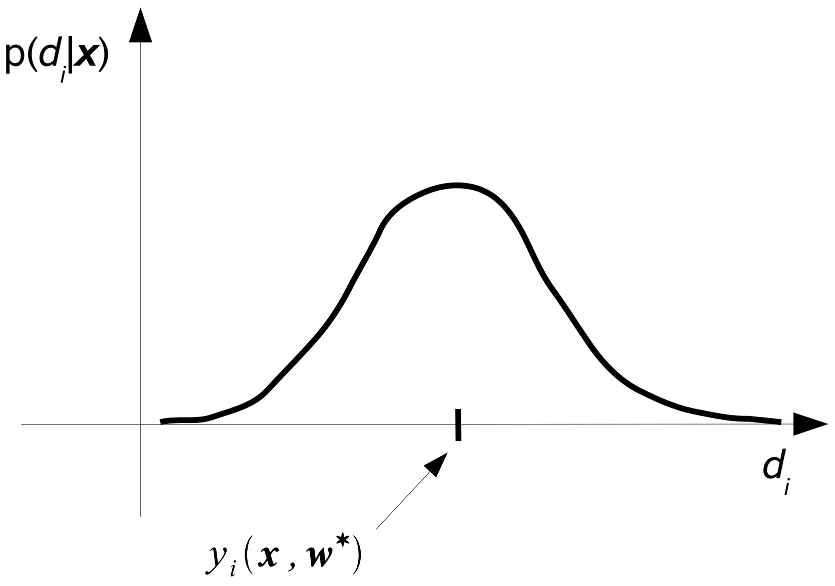

p(d_i|$\boldsymbol{x}$)
d_i
$y_i(\boldsymbol{x}, \boldsymbol{w}^*)$

Figure 3.14: Illustration of the conditional distribution $p(d_i|\mathbf{x})$ and the corresponding network mapping found by weight $\boldsymbol{\omega}^*$ that minimize the error function.

The notion that there could be a distribution $p(d|\mathbf{x})$ will be discussed in two ways. First, we will use it to motivate a set of output activation functions and error functions for regression and classification. Then, we will discus *generalization*, which could be viewed as an ambition to find $y(\mathbf{x}) = \langle d|\mathbf{x}\rangle$ even when the data set is not all that large.

3.4 Error functions

The use of the word "error" as the name of the function that we want minimize to find the values of the weights in a network is not universal. It is quite common to separate between the function you minimize and other functions (more interpretable) that might be used to measure the difference between the targets and the network prediction. Sometimes "loss function" is used for the former and simply "error" for the rest. In these lecture notes we use the word "error" for both situations, i.e. for the actual function that we minimize and for other measures of the difference between targets and predictions.

We will now explore how the idea that our data is picked from a distribution with some conditional probability $p(\mathbf{d}|\mathbf{x})$ can be used to motivate suitable error functions for different types of problems. The underlying principle that we will use is called *Maximum Likelihood.* Here follows a brief introduction followed by a derivation of two common error functions. For a bit more in depth discussion on Maximum Likelihood see chapter 5.5 (Machine Learning Basics) of the DLB.

We now consider our MLP as a model with the parameters $\boldsymbol{\omega} = \{\omega_1, \dots, \omega_M\}$. Our model should try to realize a training set,

$$\begin{aligned} X &= \{\mathbf{x}_1, \dots, \mathbf{x}_N\} \quad \text{inputs} \\ D &= \{\mathbf{d}_1, \dots, \mathbf{d}_N\} \quad \text{targets} \end{aligned}$$

Our training data comes from an underlying generator that we want to model. The most complete description is

$$p(X, D) = \text{ the joint probability density of } X \text{ and } D$$

If we assume that data points in the dataset are independent of each other we can write down the probability density for the full training set as

$$p(X, D) = \prod_{n=1}^{N} p(\mathbf{x}_n, \mathbf{d}_n) = \prod_{n=1}^{N} p(\mathbf{d}_n|\mathbf{x}_n)p(\mathbf{x}_n), \tag{3.13}$$

where $p(\mathbf{d}|\mathbf{x})$ is the conditional probability distribution[1] for $\mathbf{d}$ given a particular $\mathbf{x}$. Note that $p(\mathbf{d}|\mathbf{x})$ is useful if we want to make new predictions.

We call the full probability the *Likelihood* ($\mathcal{L}$) of the dataset and we view this a as function of the weights, $\boldsymbol{\omega}$, for a fixed X and D. The maximum of $\mathcal{L}(\boldsymbol{\omega})$ (with respect to $\boldsymbol{\omega}$) is by definition the model that maximizes $p(X, D)$ i.e. the most likely model given the training set

[1]If you are not familiar with basic concepts from probability theory, chapter 3 (Probability and Information Theory) of the DLB provides such an overview.

(X, D). Instead of maximizing $\mathcal{L}(\boldsymbol{\omega})$ we minimize its negative logarithm. We can therefore define our error function $E(\boldsymbol{\omega})$ as

$$\begin{aligned} E(\boldsymbol{\omega}) &= -\log \mathcal{L}(\boldsymbol{\omega}) \\ &= -\sum_{n=1}^{N} \log p(\mathbf{d}_n|\mathbf{x}_n) - \underbrace{\sum_{n=1}^{N} \log p(\mathbf{x}_n)}_{\text{Independent of } \boldsymbol{\omega}} \end{aligned}$$

Now the last term is independent of $\boldsymbol{\omega}$, which means that we can neglect it since we are only interested of the position of minimum not the value of the minimum. This leads finally to our error function

$$E(\boldsymbol{\omega}) = -\sum_{n} \log p(\mathbf{d}_n|\mathbf{x}_n) \tag{3.14}$$

It is at this moment perhaps somewhat difficult to see where the MLP (i.e. the weights $\boldsymbol{\omega}$) enter into the formalism. The section that follows will help!

3.4.1 Summed square error and regression problems

Let us now see how the summed square error function arises naturally for regression type problems. Suppose we have c target variables

$$\mathbf{d}_n = \big(d_{n1}, \dots, d_{nc}\big)$$

We assume that these outputs are independent of each other,

$$p(\mathbf{d}|\mathbf{x}) = \prod_{i=1}^{c} p(d_i|\mathbf{x}) \tag{3.15}$$

Furthermore, we assume the following form for the observable d_i,

$$\underset{\underset{\text{what we observe}}{\nearrow}}{d_i(\mathbf{x})} = \underset{\underset{\text{the unknown function}}{\uparrow}}{f_i(\mathbf{x})} + \underset{\underset{\text{noise}}{\nwarrow}}{\varepsilon_i} \tag{3.16}$$

where the noise term ε_i is Gaussian distributed with mean = 0 and standard deviation σ.

$$p(\varepsilon_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\varepsilon_i^2/2\sigma^2} \tag{3.17}$$

We will now model the unknown $f_i(\mathbf{x})$ by our network $y_i(\mathbf{x}, \boldsymbol{\omega})$. Since $\mathbf{d}$ is not bounded we cannot restrict the output from the network using a bounded activation function. We therefore use a linear output activation function when dealing with regression type problems.

From Eqs. 3.16 and 3.17 it follows that we can write the distribution of the observable d_i as,

$$p(d_i|\mathbf{x}, \boldsymbol{\omega}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(d_i - y_i(\mathbf{x}, \boldsymbol{\omega}))^2/2\sigma^2} \tag{3.18}$$

Using eqn. 3.15 we get

$$p(\mathbf{d}_n|\mathbf{x}_n, \boldsymbol{\omega}) = \Big(\frac{1}{\sqrt{2\pi\sigma^2}}\Big)^c e^{-\frac{1}{2\sigma^2}\sum_i (d_{ni} - y_i(\mathbf{x}_n, \boldsymbol{\omega}))^2}$$

We can now write down the error function for this system as $E = -\sum_n \log p(\mathbf{d}_n|\mathbf{x}_n, \boldsymbol{\omega})$ (see eqn. 3.14). We obtain,

$$E(\boldsymbol{\omega}) = \underbrace{\frac{1}{2\sigma^2}}_{\text{scale factor}} \sum_n \sum_i (d_{ni} - y_i(\mathbf{x}_n, \boldsymbol{\omega}))^2 + \underbrace{Nc \ln(\sigma) + \frac{Nc}{2} \ln(2\pi)}_{\text{constant}} \tag{3.19}$$

We can ignore the constant and the scale factor σ^2 to finally obtain

$$E(\boldsymbol{\omega}) = \frac{1}{2} \sum_n \sum_i (d_{ni} - y_i(\mathbf{x}_n, \boldsymbol{\omega}))^2 \tag{3.20}$$

Often a factor $\frac{1}{N}$ is included to make the error independent of the number of samples. To conclude, using a Gaussian noise model (eqn. 3.16) and the maximum likelihood principle we should minimize the summed squared error function in order to find the optimal weight vector.

One could argue that we have made rather bold assumptions about the σ parameter: we assume it independent of $\mathbf{x}$ and we assume it equal for all outputs i. Luckily, right now our goal is only to find $\langle \mathbf{d}|\mathbf{x} \rangle$, which is independent of σ. The error function resulting from the simplifying σ-assumptions serves that purpose.

3.4.2 Classification problems

We want our MLP to model $P(C_k|\mathbf{x})$, the probability for class C_k given the observable $\mathbf{x}$. This interpretation of the network output gives us many advantages, which will be discussed later.

Two classes

For the two class problem we will consider an MLP with a single output $y(\mathbf{x})$. The coding scheme we will use for two classes is $d = 1$ for class C_1 and $d = 0$ for class C_2. Then $\langle d \rangle$ is just the probability of being in class C_1, and we want our output $y(\mathbf{x})$ to model $P(C_1|\mathbf{x})$.

The conditional probability for a target is then

$$p(d|\mathbf{x}) = \left\{ \begin{array}{rl} y(\mathbf{x}), & d=1 \\ 1-y(\mathbf{x}), & d=0 \end{array} \right\} = y(\mathbf{x})^d (1-y(\mathbf{x}))^{1-d} \quad \text{(Bernoulli distribution).}$$

The likelyhood-based error function in eq. (3.14) becomes

$$E(\boldsymbol{\omega}) = -\sum_{n=1}^{N} \Big(d_n \log y(\mathbf{x}_n) + \big(1-d_n\big) \log\big(1-y(\mathbf{x}_n)\big) \Big). \tag{3.21}$$

This error is called the *cross-entropy* error. We often divide by N for normalization purposes.

Now, what happens to the backpropagation algorithm? First let's compute the derivative of $E_n = -d_n \ln(y_n) - (1-d_n)\ln(1-y_n)$ with respect to the output y_n

$$\frac{\partial E_n}{\partial y_n} = -\left(\frac{d_n}{y_n} - \frac{1-d_n}{1-y_n}\right) = \frac{y_n - d_n}{y_n(1-y_n)}$$

We may feel worried that the gradient diverges as y approaches 0 or 1. Luckily, this can be compensated with a good activation function!

Since $P(C_1|\mathbf{x})$ is bounded between zero and one, we want that property for our output. The logistic activation function

$$\varphi_o(a) = \frac{1}{1+e^{-a}}$$

not only has that property, it also satisfies

$$\varphi_o' = (1-\varphi_o)\varphi_o,$$

so that

$$\frac{\partial E_n}{\partial a_n} = \varphi_o'(a_n)\frac{\partial E_n}{\partial y_n} = (1-y_n)y_n\frac{y_n-d_n}{y_n(1-y_n)} = y_n - d_n,$$

which is exactly the same as we got the linear output function and an MSE-based error!

Using this we get the following expression for the hidden-to-output weights

$$\begin{aligned} \frac{\partial E}{\partial \omega_{ij}} &= -\frac{1}{N}\sum_n \delta_{ni} h_{nj}, \\ \delta_{ni} &= d_{ni} - y_i(\mathbf{x}_n), \end{aligned}$$

both for a linear output i optimized by a MSE error, and for a logistic output i optimized by a cross-entropy error. After that, back-propagation continues the same in both cases.

Multiple classes

For classification problems with $c > 2$ classes one often use an MLP with c outputs. The coding scheme for the classes is called "one-hot encoding" and is given by,

$$d_{ni} = \begin{cases} 1 & \text{if pattern } n \in \text{class } C_l \text{ and } i = l \\ 0 & \text{if pattern } n \in \text{class } C_l \text{ and } i \neq l \end{cases}$$

Below is an example with four classes,

1 0 0 0	class 1
0 1 0 0	class 2
0 0 1 0	class 3
0 0 0 1	class 4

As before we want our MLP to model $\langle d_i | \mathbf{x} \rangle = P(C_i|\mathbf{x})$, i.e.

$$y_i(\mathbf{x}_n) = P(C_i|\mathbf{x}_n)$$

Assuming as before that data points are independent of each other we can write the full conditional distribution as

$$p(\mathbf{d}_n|\mathbf{x}_n) = \prod_{i=1}^{c} \big(y_i(\mathbf{x}_n)\big)^{d_{ni}}$$

Again, using the definition of E (eq. (3.14)) we obtain,

$$E(\boldsymbol{\omega}) = -\sum_{n=1}^{N} \sum_{i=1}^{c} d_{ni} \log y_i(\mathbf{x}_n) \tag{3.22}$$

Again we often divide the error (equation 3.22) by N for normalization purposes.

What output activation function should we have for multiple classes? If we want to be able to interpret the output as probabilities we want $y_i(\mathbf{x}_n) \in [0,1]$ and $\sum_i y_i(\mathbf{x}_n) = 1$. A generalization of the logistic function, called softmax or smooth winner-takes-all, is suitable,

$$y_i(\mathbf{x}_n) = \frac{e^{a_{ni}}}{\sum_{i'} e^{a_{ni'}}}$$

where a_{ni} is the net input to output node i for pattern n. This activation function has the correct properties. Now the question as before, what happens to the backpropagation algorithm for these new error and activation functions?

Since one argument a_{ni} now influences all outputs $y_{ni'}$ we must start with

$$\frac{\partial E_n}{\partial a_{ni}} = \sum_{i'} \frac{\partial y_{ni'}}{\partial a_{ni}} \frac{\partial E_n}{\partial y_{ni'}} = -\sum_{i'} \frac{\partial y_{ni'}}{\partial a_{ni}} \frac{d_{ni'}}{y_{ni'}}.$$

To simplify this, we note that $\frac{1}{y}\frac{\partial y}{\partial a} = \frac{\partial \ln y}{\partial a}$ and

$$\frac{\partial \ln(y_{ni'})}{\partial a_{ni}} = \frac{\partial a_{ni'}}{\partial a_{ni}} - \frac{\partial}{\partial a_{ni}} \ln\left(\sum_{i''} \exp(a_{ni''})\right) = \frac{\partial a_{ni'}}{\partial a_{ni}} - \frac{\exp(a_{ni})}{\sum_{i''} \exp(-a_{ni''})} = \delta^{\mathrm{kr}}_{ii'} - y_{ni},$$

where we have introduced the Kronecker delta $\delta^{\mathrm{kr}}_{ij}$ as

$$\delta^{\mathrm{kr}}_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

We therefore get

$$\frac{\partial E_n}{\partial a_{ni}} = -\sum_{i'} (\delta^{\mathrm{kr}}_{ii'} - y_{ni}) d_{ni'} = -d_{ni} + y_{ni} \sum_{i'} d_{ni'} = y_{ni} - d_{ni},$$

where the last step follows from the design of the one-hot vectors, giving $\sum_i d_{ni} = 1$.

At the end of all calculations, we get the same result as for the entropy error for two classes!

As a semantic side-note, if want to solve a multi-class problem without a hidden layer, we can introduce multiple output nodes with the softmax output function. That would be considered a "simple perceptron", despite more than one output node.

3.4.3 A brief conclusion

A small conclusion regarding the choice of activation and error functions for different problem types

Problem	# of classes	φ_{output}	Error function
regression	-	linear	summed square error
classification	2	logistic	cross entropy
classification	> 2	softmax	cross entropy

3.5 Generalization

The central challenge of machine learning is to perform well on new previously unseen data, not part of the training. A model with such ability is said to *generalize*. We can define generalization performance as,

Definition Generalization performance = the performance of a trained ANN model on new data, i.e. data that was not part of the training procedure.

One obviously have to define the precise meaning of "new" data. Without going into mathematical details it is reasonable to require that new data most come from the same process that generated the training data.

One complication that can limit the generalization performance of an ANN model is *overfitting*. We can say that overfitting is a result of having a model that conforms, in too much detail, to the training set. Figure 3.15 is an illustrations of the concept of overfitting.

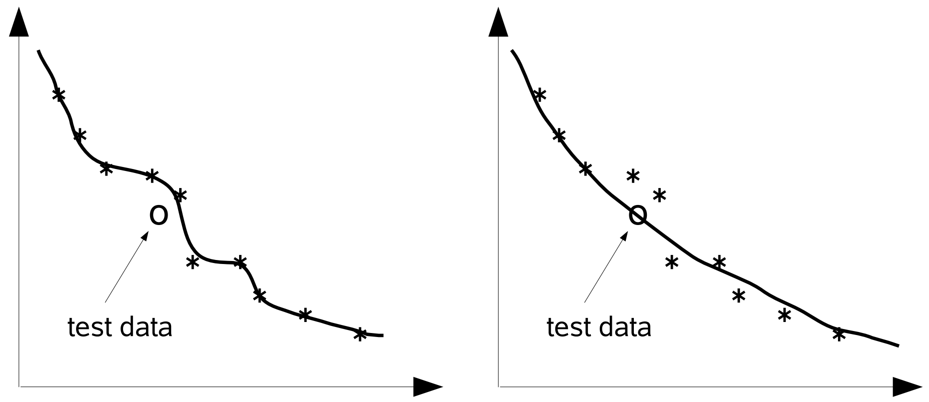


Figure 3.15: Left: Very good fit to the training data, but not that good generalization performance - overfitting. Right: Better generalization performance as a result of worse training performance - no overfitting.

We will continue this section on generalization by looking at the bias-variance tradeoff which will help us understand the problem of overfitting (and the opposite, underfitting). Later in this section we will look at methods that can be used to estimate the true generalization performance of an ANN model.

3.5.1 Bias-Variance tradeoff

Consider a dataset $\mathcal{D}$ of size N. Also assume that we have many such N-sized datasets $\mathcal{D}$. A measure of how close the ANN function is to the target is

$$(y_{\mathcal{D}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 \qquad \text{Note: Only one output}$$

where $y_{\mathcal{D}}(\mathbf{x})$ denotes the ANN function one obtains using dataset $\mathcal{D}$. To avoid the dependence on a particular dataset $\mathcal{D}$ we take the expectation value over all possible $\mathcal{D}$'s.

$$\mathrm{E}\left[(y_{\mathcal{D}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2\right]$$

where $\mathrm{E}[\cdot]$ denotes the expectation over all $\mathcal{D}$'s. Using the well-known formula for variance, $\mathrm{Var}[z] = \mathrm{E}[z^2] - \mathrm{E}[z]^2$, we let $z = y_{\mathcal{D}} - \langle d\rangle$, and find

$$\mathrm{E}\left[(y_{\mathcal{D}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2\right] = \underbrace{\Big(\mathrm{E}\left[y_{\mathcal{D}}(\mathbf{x})\right] - \langle d|\mathbf{x}\rangle\Big)^2}_{\text{bias}^2} + \mathrm{Var}\left[y_{\mathcal{D}}(\mathbf{x})\right]. \tag{3.23}$$

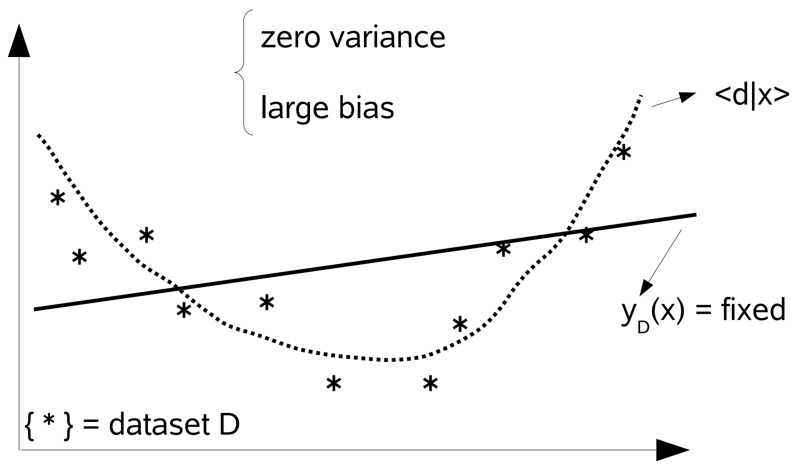
Here, we have exploited that $\langle d\rangle$ is independent of dataset $\mathcal{D}$, so that $\mathrm{E}[y_{\mathcal{D}} - \langle d\rangle] = \mathrm{E}[y_{\mathcal{D}}] - \langle d\rangle$ and $\mathrm{Var}[y_{\mathcal{D}} - \langle d\rangle] = \mathrm{Var}[y_{\mathcal{D}}]$.

- The bias term measures (in average) how much the different ANN models differs from the target function $\langle d|\mathbf{x}\rangle$.
- The variance term measures the sensitivity of $y_{\mathcal{D}}(\mathbf{x})$ for different dataset $\mathcal{D}$.

The bias-variance tradeoff is illustrated in figure 3.16. The above expression for the bias and the variance still have an $\mathbf{x}$ dependence. To get the overall tradeoff we need to integrate over $\mathbf{x}$ taking into account a distribution $p(\mathbf{x})$ generating the inputs of the datasets,

$$\begin{aligned} \text{bias}^2 &= \int \Big(\mathrm{E}\left[y_{\mathcal{D}}(\mathbf{x})\right] - \langle d|\mathbf{x}\rangle\Big)^2 p(\mathbf{x})d\mathbf{x} \\ \text{variance} &= \int \mathrm{Var}\left[y_{\mathcal{D}}(\mathbf{x})\right] p(\mathbf{x})d\mathbf{x} \end{aligned}$$

The bias-variance tradeoff means that there is a conflict between the bias term and the variance term. For ANN models this usually means that in order to have low bias one need a flexible model, e.g. a large number of hidden nodes. A flexible model however means that we can fit the ANN model very well to a specific dataset $\mathcal{D}$, i.e. the variance increases. In order to have a low variance we can restrict the ANN model by e.g. having very few hidden nodes, but this also means that we may increase the bias since the ANN model may be too limited. In the next section we will try to control the bias-variance tradeoff using *regularization*.



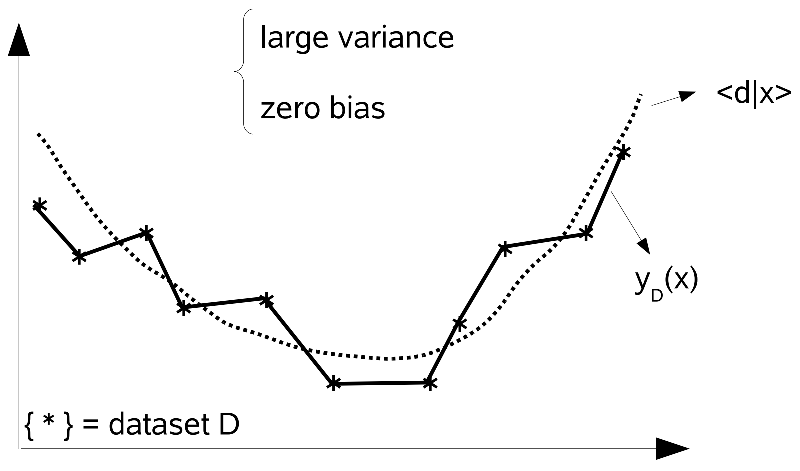


Figure 3.16: Illustration of the bias-variance tradeoff. In both figures the target function is illustrated using a dotted line and a particular dataset $\mathcal{D}$ with star symbols. **Top figure:** The network function $y_{\mathcal{D}}(\mathbf{x})$ is fixed, regardless of the data $\mathcal{D}$. Hence we can conclude that the variance is zero, however the bias is large since it is far from the target function. We call this situation underfitting. **Bottom figure:** The network now perfectly adapted to any instance of $\mathcal{D}$. We can imagine that the average of all such $y_{\mathcal{D}}$'s is very close to the target function, hence zero bias. However we now have a large variance because the extreme dependence that $y_{\mathcal{D}}(\mathbf{x})$ has on $\mathcal{D}$. We call this situation overfitting. In both cases we have a large total error.

3.5.2 Network regularization

In order to control the bias-variance tradeoff neural networks we are going to use network regularization. The first types of regularization we are going to look at modifies the error function according to,

$$\tilde{E}(\boldsymbol{\omega}) = E(\boldsymbol{\omega}) + \alpha\Omega$$

where Ω is the regularization term and α controls the amount of regularization. We are going to look at three Ω terms.

L2 norm regularization (weight decay)

The weight decay term is as follows,

$$\Omega = \frac{1}{2}\sum_i \omega_i^2$$

where the sum runs over all weights in the network (except thresholds/biases). This term will force excess weights to zero while keeping necessary weights to a non-zero value. To summarize:

+ Easy to implement
+ Often increases the generalization performance
+ "Theoretical" interpretation
- α needs tuning
- Sometimes large weights are necessary

Modified L2 norm (weight elimination)

The modified L2 norm is as follows,

$$\Omega = \frac{1}{2}\sum_i \frac{(\omega_i/\omega_o)^2}{1+(\omega_i/\omega_o)^2}$$

where ω_o is a new parameter. A plot of this term is shown in figure 3.17. Weight elimination takes care of the last "minus" for the weight decay term, with the cost of having to tune an extra parameter.

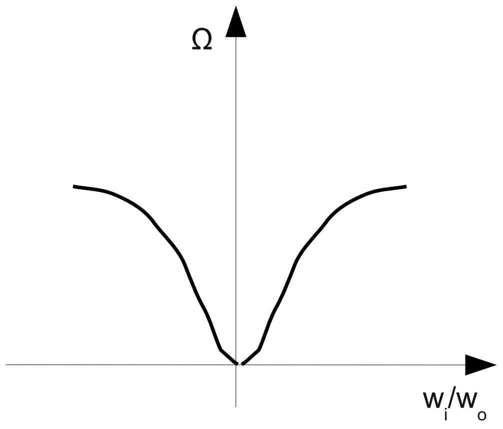


Figure 3.17: The weight elimination regularization term.

L1 norm regularization (lasso)

The lasso regularization is as follows,

$$\Omega = \frac{1}{2}\sum_{i} |\omega_i|$$

A plot of this term is shown in figure 3.18. A difference compared to weight decay is that gradient of the lasso term is constant and does not go to zero as the weight approaches zero. This means that weights that are not needed will be forced to zero. Lasso can therefore be used as *feature selection* method!

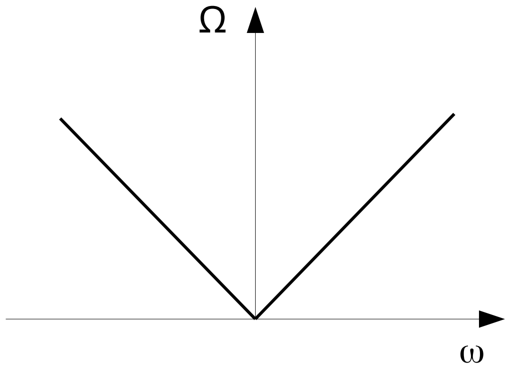


Figure 3.18: The lasso regularization term.

Max-norm regularization

The max-norm method puts a constraint on individual node weight vectors. Denote by $\boldsymbol{\omega}_i$ the vector of all weights feeding into node i. Max-norm then puts the following constraint on $\boldsymbol{\omega}_i$,

$$|\boldsymbol{\omega}_i| \leq c$$

where c is a parameter, typically of size 2-4. This means that node weight vectors are not allowed to grow large. The constraint is enforced during training after each weight update, by simply multiply $\boldsymbol{\omega}_i$ with $c/|\boldsymbol{\omega}_i|$ if the constraint is violated.

Early stopping

Another alternative to regularization as a way of controlling the complexity of a network is the procedure of *early stopping.* A typical behavior during the minimization process can be seen in figure 3.19. If we estimate the generalization error using a separate validation set we

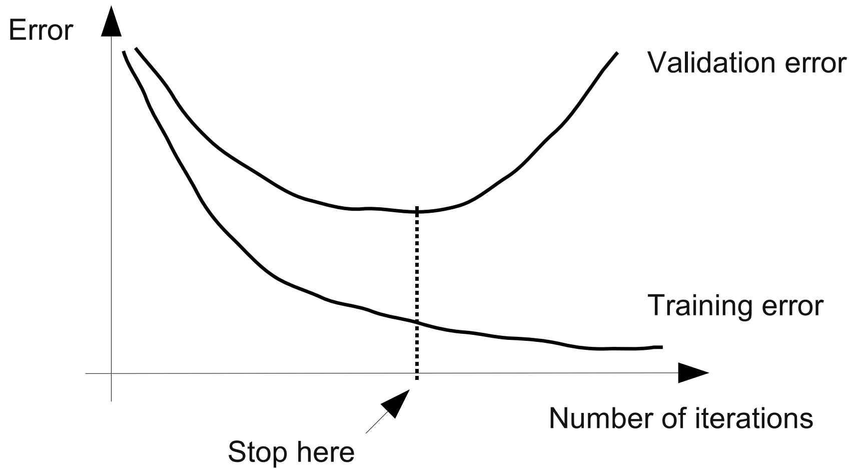


Figure 3.19: The training and the validation error during the minimization process. The “early stop” procedure halts the training when the validation error have reached a minimum.

can stop the training when the validation error starts to increase. This procedure is called early stop.

Other methods

There exists other regularization techniques and to mention a few:

- Training with noise
- Soft weight sharing
- Data augmentation

However there is still one very interesting method to discuss, the dropout method.

3.5.3 Dropout regularization

The dropout method was introduced 2013, by Hinton et al. It can be seen as an efficient regularization technique to avoid overtraining. It can also be seen as a huge ensemble of many different networks[2]. The term dropout refers to temporarily removing nodes in a network. This means that all weights feeding into and out from the node are temporarily removed along with the node. Figure 3.20 shows the idea of dropout. The nodes to drop are

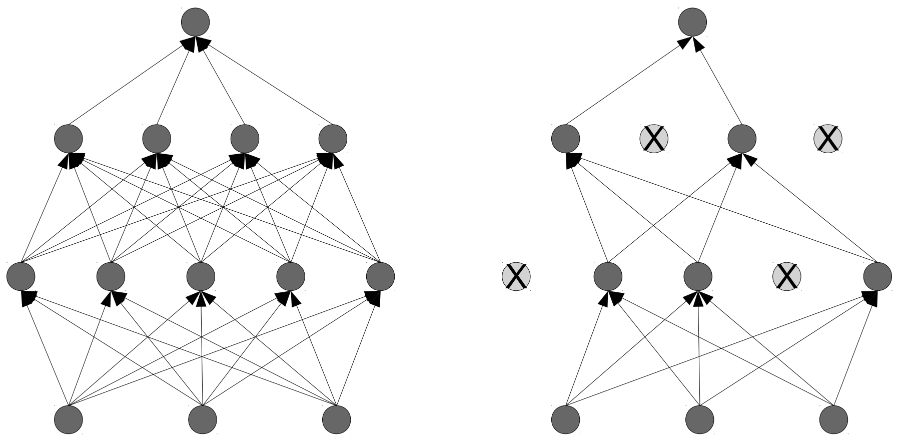


Figure 3.20: (left) An MLP with two hidden layers. (right) The same network but this time a few of the hidden nodes have been (temporarily) removed along with all of the associated weights.

selected at random, usually with a fixed parameter p that is the probability to *keep* nodes. So if $p = 0.5$ it means that, in average, half of the nodes are dropped. We can use dropout in all layers of the network, except the output layer. It is common to have one p parameter for each layer and these probabilities are usually set using a validation dataset. The dropout technique is only used during training of the network, when testing the network all of the nodes are kept as usual.

[2]Ensemble methods are described later.

Training with dropout Using dropout means that each time we are going the use the network during training we will apply the dropout procedure, meaning that each node will be subject to a removal with a probability of $1 - p$. This results in a thinned network that will be used during the forward and backpropagation steps. It is important here that a new thinned network will be sampled for each new data point that is used during training. One can summarize the dropout procedure for stochastic gradient descent as,

1 Set mini-batch size M for SGD

2 Select M random patterns for the mini-batch.

3 For each of the M patterns do,

  a Sample a thinned network, ie. apply the dropout procedure by dropping random nodes according to p.

  b Use this thinned network to compute all weight updates for this pattern and add to total weight update. Weights that are not used because associated nodes are dropped are omitted.

4 Average the weight update by the number of times a given weight was part of the thinned networks and perform the update. Go back to item 2.

Using a network trained with dropout If we have P nodes in our network then a thinned network is just one of the 2^P possible such thinned networks. All of our thinned networks share weights. We can therefore see dropout as training a collection of 2^P networks with extensive weight sharing. Each of the thinned networks are trained rarely, if at all. So, during testing should we make an average of all these thinned networks? This does not seem feasible! Instead a simple weight averaging technique is used. The weights of the final network (used for testing) are rescaled using the probability p, as illustrated in figure 3.21. We are only going to use one network, but with all nodes present. If a node is present during

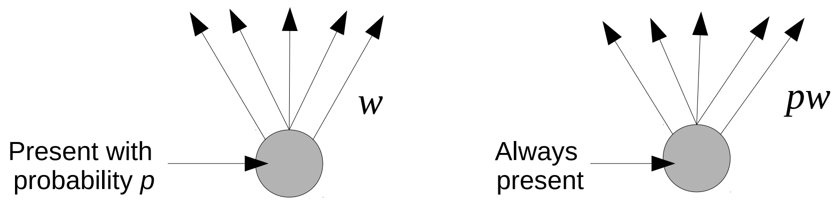


Figure 3.21: (left) During training a node is present with probability p. (right) During testing the node is always present, and to make the expected output to agree in both cases the outgoing weights are multiplied by p.

training with probability p, then all of the weights feeding out from this node is multiplied with p. This is intuitively correct since the expected number of weights feeding into a node during training is less than during test. We compensate for that by rescaling the weights of an outgoing node by the factor p.

An example Figure 3.22 shows two examples of varying the number of hidden nodes for the Pima Indians classification problem. In the left graph we see the expected behavior of an increasing validation error as we increase the number of hidden nodes. At the same time the training error goes down. A typical example of overtraining! In the right graph we have the same network setup except we are now using $p = 0.5$ dropout on the hidden layer. First note the different scale and the fact that regardless of the number of hidden nodes the validation error approximately stays the same.

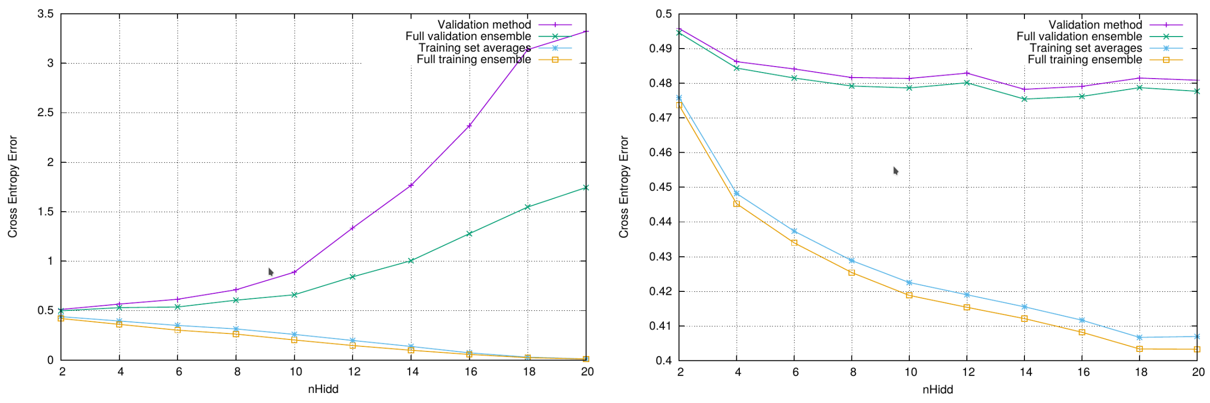


Figure 3.22: Training and validation error as a function of the number of hidden nodes for the Pima Indians classification problem. The left experiment is without dropout and the right is with dropout ($p = 0.5$) for the hidden layer. There are four curves, two validation curves and two training curves, representing an ensemble or no ensemble. We can see in the left graph that the ensemble prevents some overtraining, but not all of it.

3.6 Model selection and estimation of generalization performance

We have seen that there are an number of parameters involved when training a neural network, e.g. number of hidden nodes, learning rate, possible value of the momentum parameter, weight decay parameter etc. Most of these parameters are difficult to estimate before training starts. The procedure of selecting all parameters that define the neural network and the construction of it is usually called *model selection*. In order to select good models we need a way of estimating the performance of a model. We have seen that neural networks are powerful non-linear learning machines that can overfit a training dataset, meaning that the performance as measured on the training data is usually very biased. We need a procedure to estimate the *generalization performance*.

3.6.1 Holdout method

Split the dataset into two parts, a training set and a validation set (typically 1/3 to validation and 2/3 for training). Training is accomplished using the training set and the generalization

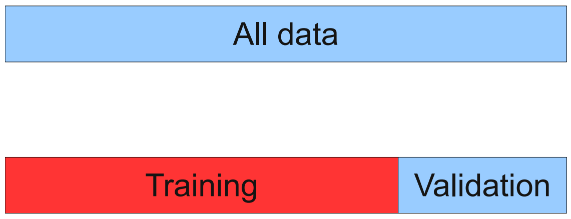


Figure 3.23: The holdout method. One part is for training and one is for validation.

performance is estimated using the validation set. This is (still) a very common method.

One drawback with this method is that we do not use all the information in the dataset since part of it is left as a validation set. Also, there may be a difference between the training and the validation set, which will bias the estimation. Next method will try to overcome these issues.

3.6.2 K-fold cross validation

Split (randomly) the dataset into K parts of (approximate) equal size. We will now train K models on K slightly different training sets and validate on K different validation sets. Figure 3.24 illustrates how the different training and validation sets are defined. The final estimate of the generalization performance is given by the average of the K validation results. Let P_i denote the performance measured on validation set i when training on the corresponding training set i. Then $\hat{P}$, the estimate of the true P is given by,

$$\hat{P} = \frac{1}{K}\sum_{i=1}^{K} P_i$$

Often the procedure is repeated N times with different K-fold splits each time in order to further enhance the estimation. The estimation then becomes,

$$\hat{P} = \frac{1}{NK}\sum_{n=1}^{N}\sum_{i=1}^{K} P_{in}$$

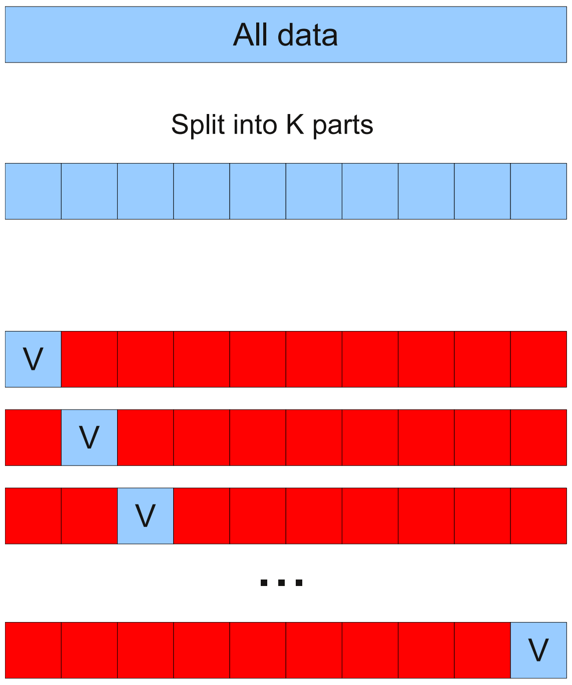


Figure 3.24: The K-fold cross validation method. Each of the K validation parts (V) are used as a validation set when training on the remaining parts.

where P_{in} being the validation performance on split i for the n:th repetition.

3.6.3 Bootstrap

A bootstrap sample of the dataset (of size N) means resampling the data set, **with replacement**, to produce a new dataset of size N. There will hence be duplicate points in the bootstrap sample. Let P_b^* be the performance measured on the samples **not** part of the b:th bootstrap sample. The bootstrap estimate is then given by

$$\hat{P}^{boot} = \frac{1}{B}\sum_{b=1}^{B} P_b^*$$

One can argue the this estimate is too pessimistic and an adjustment has been suggested, which is called the *bootstrap 632* rule, where

$$\hat{P}^{632} = 0.368\hat{P}^{a} + 0.632\hat{P}^{boot}$$

where $\hat{P}^a$ is the training performance averaged over all bootstrap samples. The 632 rule have some problems with very overfitted models where it often gives a to optimistic estimation.

The .632 factor comes from the fact that for large N this is the probability that a given data point is part of the bootstrap sample. Can you show this?

3.6.4 Model selection

Now to perform a model selection let's say for the α parameter that controls the weight decay regularization term we can produce a plot like the one in Figure 3.25. Here the validation

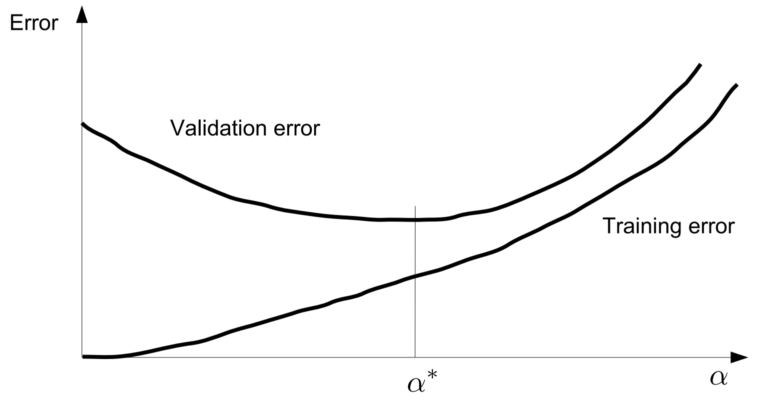


Figure 3.25: Let α be a regularization parameter, e.g. the weight decay parameter. We train and validate a number of networks, each with a different value of α. Since the validation error is an estimate of the generalization error, we pick the α that minimizes the validation error.

error can be estimated using one of the above techniques.

Having obtained the best set of parameters for our neural network we are still faced with the problem of estimating the generalization performance for this model. It might be tempting to use the validation performance that we obtained during the model selection procedure. However this estimate is biased since it has implicitly been part of the training procedure. We cannot use that estimate! A common way is to introduce yet another dataset, often called an independent test set. It is also possible to have two cross validation loops, one outer loop for estimating the generalization performance and one inner loop that takes care of model selection. Figure 3.26 is illustrating this. Here each "construction" dataset is subject to a model section M-fold cross validation where model parameters for the construction data is determined. Once that is finished a new model is trained on all construction data and that model is then used on the corresponding test dataset from the outer most K-fold cross "testing" loop. The test sets in the K-fold cross "testing" loop are never part of any training of the networks, hence we have a unbiased estimate of the generalization performance.

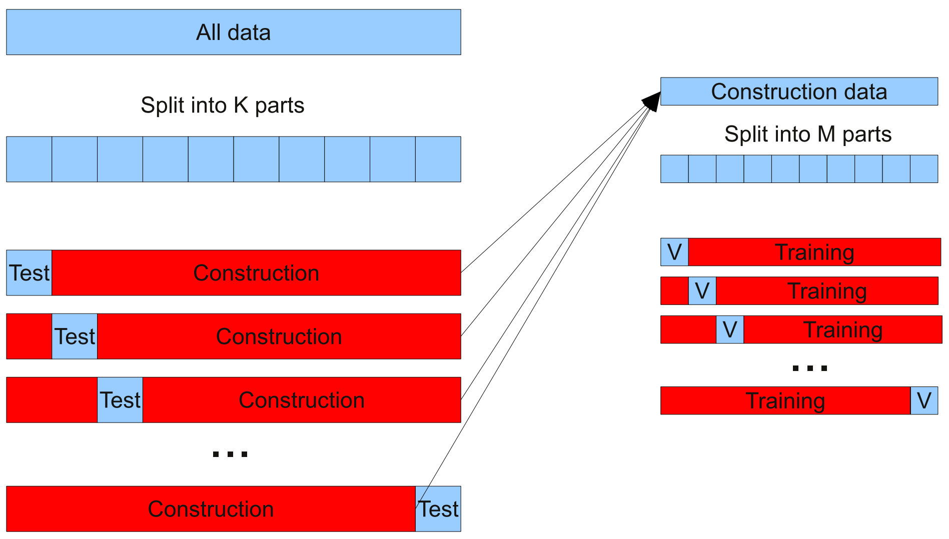


Figure 3.26: Procedure for estimating the generalization performance using two cross validation loops.

3.7 Bayesian learning of neural networks

(This is a brief introduction to the subject)

Bayesian approach to learning offers some new important features:

- Regularization can be given a natural interpretation in the Bayesian framework.
- For regression problems, confidence intervals and error bars can be assigned to the predictions generated by the network.
- Regularization coefficients *can* be selected using the training set only. No need for a validation set to select these parameters.
- The problem of overtraining is reduced.
- The relative importance of different inputs can be determined using the Bayesian technique of ARD (Automatic Relevance Determination).

3.7.1 Prior distribution of weights

As usual we have

$\mathbf{X} = \{\mathbf{x}_1 \dots \mathbf{x}_N\}$	input data
$\mathbf{D} = \{\mathbf{d}_1 \dots \mathbf{d}_N\}$	target data
$\mathcal{D} = \{\mathbf{D}, \mathbf{X}\}$	the data set
$\boldsymbol{\omega} = \{\omega_1 \dots \omega_M\}$	all the weights as a single vector

Before we look at data, we have no real clue what the weights $\boldsymbol{\omega}$ should be, but even so, we always start with some restrictions. Given the size of our network, we claim that a longer $\boldsymbol{\omega}$-vector, representing more nodes, is not needed. If we apply some regularization, it can be seen as a claim that small ω_i-values are more likely. In the Bayesian framework, such "claims" or "beliefs" are represented by a *prior distribution* $p(\boldsymbol{\omega})$. This idea can seem bold, but one argument is that whenever we start to build a model, we implicitly introduce a lot of assumptions about $\boldsymbol{\omega}$, and the prior distribution $p(\boldsymbol{\omega})$ helps us to keep track of the consequences.

Given a prior and a dataset $\mathcal{D}$, we can calculate the conditional probability $p(\mathcal{D}|\boldsymbol{\omega})$, and also the total probability for the set $\mathcal{D}$,

$$p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\omega})p(\boldsymbol{\omega})d\boldsymbol{\omega}.$$

However, what we really want to formulate is how the dataset $\mathcal{D}$ influences our belief in different $\boldsymbol{\omega}$: we want a *posterior distribution* $p(\boldsymbol{\omega}|\mathcal{D})$. Bayes' theorem allows us to write

$$p(\boldsymbol{\omega}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\omega})p(\boldsymbol{\omega})}{p(\mathcal{D})} \tag{3.24}$$

So in order to compute $p(\boldsymbol{\omega}|\mathcal{D})$ we need to...

formulate the prior	$p(\boldsymbol{\omega})$
compute the likelihood	$p(\mathcal{D}|\boldsymbol{\omega})$

Note that for a large dataset $\mathcal{D}$, the ratio $p(\mathcal{D}|\boldsymbol{\omega})/p(\mathcal{D})$ will essentially vanish, except for weights $\boldsymbol{\omega}$ near the $\boldsymbol{\omega}^*$ that maximizes the likelihood $p(\mathcal{D}|\boldsymbol{\omega})$. As long as the prior is non-zero for $\boldsymbol{\omega}^*$, the posterior becomes a very narrow distribution around $\boldsymbol{\omega}^*$. Thus, we can regard the maximum likelihood approach discussed earlier as a Bayesian strategy, with the simplifying assumption that the dataset is large enough to allow us to neglect the width in the posterior.

With our network models, we focus on the relations between inputs and targets. Changing notation from dataset $\mathcal{D}$ to targets $\mathbf{D}$ and inputs $\mathbf{X}$, where $\mathbf{X}$ are independent of $\boldsymbol{\omega}$ so that $p(\mathbf{X}|\boldsymbol{\omega}) = p(\mathbf{X})$, we can write Bayes' relation in eq. (3.24) as

$$p(\boldsymbol{\omega}|\mathcal{D}) = \frac{p(\mathbf{D}|\mathbf{X}, \boldsymbol{\omega})p(\boldsymbol{\omega})}{p(\mathbf{D}|\mathbf{X})}, \tag{3.25}$$

where the normalization in the denominator is

$$p(\mathbf{D}|\mathbf{X}) = \int p(\mathbf{D}|\mathbf{X},\boldsymbol{\omega})p(\boldsymbol{\omega})d\boldsymbol{\omega}.$$

Our goal is to find $\boldsymbol{\omega}$ with a large posterior, so we can decide to minimize

$$-\ln p(\boldsymbol{\omega}|\mathcal{D}) = -\ln p(\mathbf{D}|\mathbf{X},\boldsymbol{\omega}) - \ln p(\boldsymbol{\omega}) + \ln p(\mathbf{D}|\mathbf{X}) \tag{3.26}$$

The last logarithm can be omitted, since it is independent of $\boldsymbol{\omega}$. With indepedent samples, $p(\mathbf{D}|\mathbf{X},\boldsymbol{\omega}) = \prod_n p(\mathbf{d}_n|\mathbf{x}_n,\boldsymbol{\omega})$, we then get to minimize

$$S(\boldsymbol{\omega}) = -\sum_n \ln p(\mathbf{d}_n|\mathbf{x}_n,\boldsymbol{\omega}) - \ln p(\boldsymbol{\omega}) \tag{3.27}$$

Gaussian noise model

The Gaussian prior is a very common way to represent an expectation of small weights:

$$p(\boldsymbol{\omega}) \propto \exp\left(-\frac{1}{2\sigma_\omega^2}\sum_i \omega_i^2\right).$$

This leads to the L2-regularization term in the error function (see chapter 3.5.2):

$$S(\boldsymbol{\omega}) = \sum_n E_n(\boldsymbol{\omega}) + \frac{1}{2\sigma_\omega^2}\sum_i \omega_i^2. \tag{3.28}$$

When we discussed this earlier, we did not introduce σ_ω^2 as a free parameter, but rather a regularization strength $\alpha = \frac{1}{N\sigma_\omega^2}$.

The weight vector $\boldsymbol{\omega}_{MP}$ that maximizes the posterior $p(\boldsymbol{\omega}|\mathcal{D})$ is found by minimizing $S(\boldsymbol{\omega})$. Due to the prior, it differs from the maximum likelihood weight vector $\boldsymbol{\omega}_{ML}$ that minimizes $\sum_n E_n(\boldsymbol{\omega})$. Figure 3.27 illustrates the difference between maximum likelihood and maximum a posteriori.

3.7.2 Distribution of network outputs

Lets now look at the "complete" Bayesian training of neural networks. Once the posterior $p(\boldsymbol{\omega}|\mathcal{D})$ is found, we can use it to predict the target distribution for any $\mathbf{x}$, even those not present in the dataset $\mathcal{D}$. Using the rules of probability we can write

$$p(d|\mathbf{x},\mathcal{D}) = \int p(d|\mathbf{x},\boldsymbol{\omega})p(\boldsymbol{\omega}|\mathcal{D})d\boldsymbol{\omega} \tag{3.29}$$

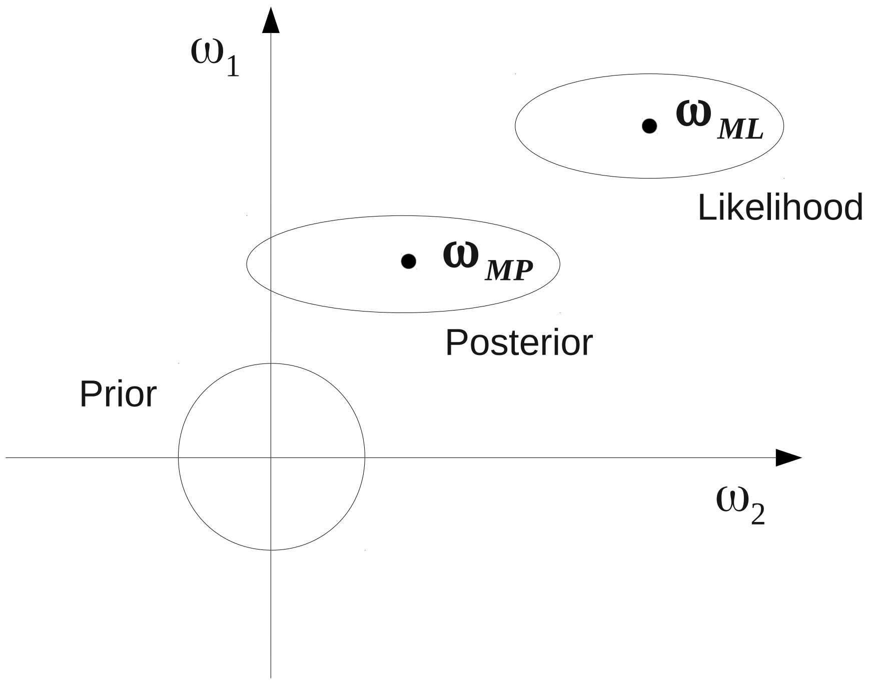


Figure 3.27: Illustration of the difference between maximum likelihood and maximum a posteriori. The posterior distribution is now a combination of both the likelihood and the prior, changing the obtained weight vector from $\boldsymbol{\omega}_{ML}$ to $\boldsymbol{\omega}_{MP}$.

We can see this formula as a weighting of all possible models $p(d|\mathbf{x}, \boldsymbol{\omega})$, where the weighting is taken care of by the posterior weight distribution $p(\boldsymbol{\omega}|\mathcal{D})$. We saw previously that the MLP gives us the conditional average $\langle d|\mathbf{x}, \mathcal{D}\rangle$. The above eqn. 3.29 gives us a distribution of outputs which we could use to e.g. assign confidence intervals to our prediction.

- MLP: Uses $\boldsymbol{\omega}_{MP}$ to make predictions based on expectation values $\langle d|\mathbf{x}, \mathcal{D}\rangle$.
- Bayes MLP: Uses many (infinite) $\boldsymbol{\omega}$ to make predictions based on distributions $p(d|\mathbf{x}, \mathcal{D})$.

How do one do in practice to compute $p(d|\mathbf{x}, \mathcal{D})$ according to eqn. 3.29? The are two alternatives:

1. Monte Carlo approximation: This means that we approximate

$$p(d|\mathbf{x}, \mathcal{D}) \approx \frac{1}{K} \sum_{k=1}^{K} p(d|\mathbf{x}, \boldsymbol{\omega}_k)$$

where $\boldsymbol{\omega}_k$ are drawn from the posterior weight distribution $p(\boldsymbol{\omega}|\mathbf{D})$. Practically this is accomplished using various Monte Carlo methods.

2. Approximation by Gaussians: Here we approximate the posterior weight distribution $p(\boldsymbol{\omega}|\mathcal{D})$ by a Gaussian distribution. This can be done by Taylor expanding the expression for $S(\boldsymbol{\omega})$ (eqn. 3.28) up the second order. Note that when we minimize a function like $E(\boldsymbol{\omega})$ or $S(\boldsymbol{\omega})$, it can first be multiplied by any positive constant, and it is conventient to normalize by the number of samples. Now, when we want the width of a $\boldsymbol{\omega}$ distribution, it is important to use $S(\boldsymbol{\omega}) = -\ln p(\boldsymbol{\omega}|\mathcal{D})$.

3.8 Error minimization issues

Here we discuss how to improve the efficiency of the gradient descent method, and also briefly introduce more advanced minimization methods.

3.8.1 Gradient descent enhancements

The gradient descent method has the following basic updating formula

$$\Delta\omega_i = -\eta \frac{\partial E}{\partial \omega_i}$$

where the learning rate η controls the size of the update. The gradient descent method is easy to implement and fast, but not always the most efficient one. It can suffer from the problem of local minima and regions with small gradients (see figure 3.28). Furthermore, it is not always easy to set the value of the learning rate η. Even though the stochastic gradient descent approach helps avoiding local minima, there are still for improvements. Here are a few of them.

Momentum term (poor man's conjugate gradient)

The momentum term adds a part of the previous update to the current one,

$$\Delta\omega_i(t+1) = -\eta \frac{\partial E}{\partial \omega_i} + \alpha \Delta\omega_i(t)$$

where α is usually > 0.5 and t denotes the iteration index. This will accelerate the minimization when several updates in a row have the same sign and decelerate when they have opposite sign (oscillation). The momentum term will usually speed up the minimization process.

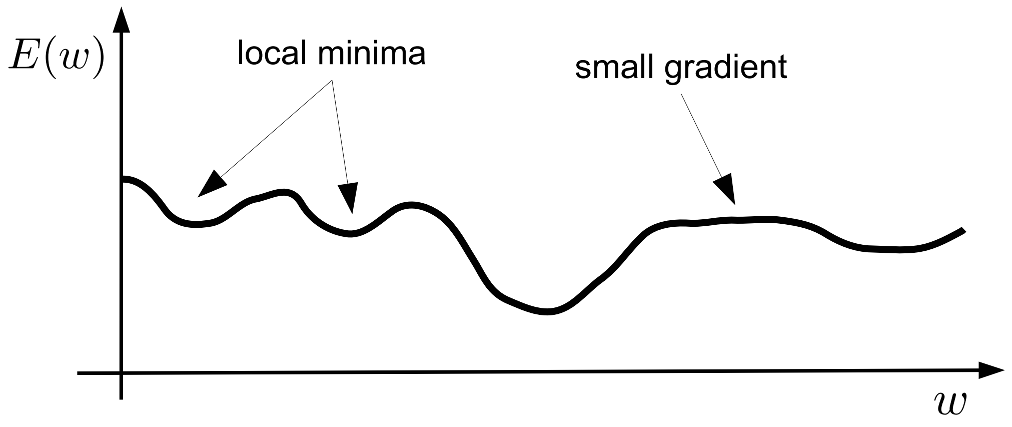


Figure 3.28: Illustration of the problem of local minima and small gradients.

Dynamical learning rate

Instead of having a fixed learning rate throughout the whole minimization procedure one can try to dynamically change its value to speed up the minimization. The following idea (called "bold driver") modifies the learning rate based on the sign of the error changes.

$$\eta(t+1) = \begin{cases} \eta(t) \cdot \gamma & \text{if } E(t+1) \geq E(t) \\ \eta(t) \cdot \left(1 + \frac{1-\gamma}{10}\right) & \text{otherwise} \end{cases} \tag{3.30}$$

The scale factor γ is close to but less than one.

There are better ways of implementing a minimization method using dynamical learning rates. One such method is called Adam, but we will introduce RPROP and RMSPROP before that.

Individual and adaptive learning factors - RPROP

We have seen how to have dynamical learning rates, here we go one step beyond and introduce individual learning rates,

$$\Delta\omega_{ij} = -\eta_{ij}\frac{\partial E}{\partial \omega_{ij}}$$

One such method is called RPROP (Resilient PROPagation). RPROP uses only the sign on the gradient and not its size

$$\Delta\omega_{ij}(t) = -\eta_{ij}(t) sgn\left(\frac{\partial E(t)}{\partial \omega_{ij}}\right)$$

Thus RPROP has individual learning factors who are allowed to vary in time. You update η_{ij} according to the following schedule:

$$\eta_{ij}(t) = \begin{cases} \gamma^{+}\eta_{ij}(t-1) & \text{if} \quad \frac{\partial E(t)}{\partial \omega_{ij}} \cdot \frac{\partial E(t-1)}{\partial \omega_{ij}} > 0 \\ \gamma^{-}\eta_{ij}(t-1) & \text{if} \quad \frac{\partial E(t)}{\partial \omega_{ij}} \cdot \frac{\partial E(t-1)}{\partial \omega_{ij}} < 0 \end{cases} \tag{3.31}$$

where $0 < \gamma^{-} < 1 < \gamma^{+}$. If two successive gradients points in the same direction we will increase the step, otherwise we will decrease it, similar to the momentum term. One negative aspect of RPROP is that one have to tune the new parameters γ^{-} and γ^{+}.

RMSPROP

The new thing with RPROP was to use individual learning rates for each weight and only use the sign of the gradient. A more modern version of RPROP is called RMSPROP (short for Root Mean Square Propagation) that works well for the stochastic gradient descent idea of using mini-batches. RMSPROP only uses one common learning rate, but it keeps a running average of the squared gradient for each weight that is used to normalize the magnitude of the gradient, thereby effectively introducing individual weights. Define this running average $v_i(t)$ as,

$$v_i(t) = \gamma v_i(t-1) + (1-\gamma)\left(\frac{\partial E(t)}{\partial \omega_i}\right)^2$$

where t is an iteration index and ω_i is the i:th weight in the network. The parameter γ is the decay rate parameter. The RMSPROP updates the weights according to,

$$\omega_i(t+1) = \omega_i(t) - \frac{\eta}{\sqrt{v_i(t)}}\frac{\partial E(t)}{\partial \omega_i}$$

where η is the common learning rate. Note that $\partial E(t)/\partial \omega_i$ can be computed using mini-batches, meaning that RMSPROP can be used as a stochastic gradient descent method. A typical value for γ is 0.9. RMSPROP is in itself a popular minimization method.

Adam

If you have experimented with training networks before you may have used the Adam (short for Adaptive moment estimation) "minimizer". Let's now describe this method! In addition to keeping a running average of the square of the past gradients, as RMSPROP does, Adam also keeps a running average of the past gradients. We define,

$$\begin{aligned} m_i(t+1) &= \beta_1 m_i(t) + (1-\beta_1)\frac{\partial E(t)}{\partial \omega_i} \\ v_i(t+1) &= \beta_2 v_i(t) + (1-\beta_2)\left(\frac{\partial E(t)}{\partial \omega_i}\right)^2 \end{aligned}$$

Here β_1 and β_2 are decay rate parameters. In practice $\beta_1 < \beta_2$ since we want the average gradient to be more responsive to fast changes of the gradient compared to the square gradient this is used for normalization. To avoid an initial bias towards small values, since both v_i and m_i are initialized with zero, a correction is applied for them,

$$\hat{m}_i = \frac{m_i(t+1)}{1-\beta_1^t}$$
$$\hat{v}_i = \frac{v_i(t+1)}{1-\beta_2^t}$$

such that $\hat{m}_i$ becomes $m_i(t+1)$ when t is large and similar for $\hat{v}_i$. Finally the weight update of the Adam method is given by,

$$\omega_i(t+1) = \omega_i(t) - \eta\frac{\hat{m}_i}{\sqrt{\hat{v}_i}+\epsilon}$$

where ϵ is use to avoid numerical problems if $\hat{v}_i$ becomes to small. The authors of the Adam method suggested $\beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

3.8.2 Conjugate gradients

Before we start with the conjugate gradient method let's take a look at the method of steepest descent. Steepest descent is a line minimization procedure,

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \lambda\mathbf{p}$$

with $\mathbf{p} = -\nabla_\omega E(\boldsymbol{\omega})$. In every step you decide λ to minimize $E(\boldsymbol{\omega})$ (in the direction of $\mathbf{p}$). One can see that steepest descent must follow a "zigzag" path. One can realize this by the following argument. Since we minimize in each step in the direction of $\mathbf{p}$, then by definition,

$$0 = \frac{\partial}{\partial\lambda}E\left(\boldsymbol{\omega}(t) + \lambda\mathbf{p}(t)\right) = \nabla_{\boldsymbol{\omega}}E\left(t+1\right)\cdot\mathbf{p}(t)$$

This means the new direction $\nabla_{\boldsymbol{\omega}}E\left(t+1\right)$ is perpendicular to the old direction $\mathbf{p}(t)$. Conjugate gradient methods uses the following update equation,

$$\mathbf{p}(t+1) = -\nabla_{\boldsymbol{\omega}}E\left(t+1\right) + \beta\mathbf{p}(t)$$

for some suitable choice of β. There are a few different rule for computing β, here is the so called Polak-Ribiere rule.

$$\beta = \frac{\nabla E(t+1) - \nabla E(t)\cdot\nabla E(t+1)}{(\nabla E(t))^2}$$

Note: Steepest descent and the conjugate gradient methods are all batch algorithms, i.e. they use all training data to compute ∇E.

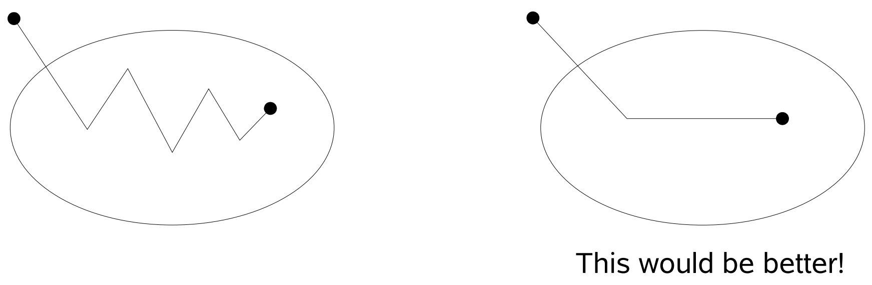


Figure 3.29: The steepest descent “zigzag” path.

3.8.3 Second order methods

Second order methods usually uses a local quadratic approximation to the error function. Assume that we Taylor expand $E(\boldsymbol{\omega})$ around some point $\boldsymbol{\omega}_o$,

$$E(\boldsymbol{\omega}) \approx E(\boldsymbol{\omega}_o) + (\boldsymbol{\omega} - \boldsymbol{\omega}_o)^T \nabla E|_{\boldsymbol{\omega}_o} + \frac{1}{2}(\boldsymbol{\omega} - \boldsymbol{\omega}_o)^T \mathbf{H}(\boldsymbol{\omega} - \boldsymbol{\omega}_o) \tag{3.32}$$

where the elements of the *Hessian* matrix $\mathbf{H}$ is defined as

$$H_{ij} = \left.\frac{\partial^2 E}{\partial \omega_i \partial \omega_j}\right|_{\boldsymbol{\omega}_o}$$

Now suppose that $\boldsymbol{\omega}_o$ is a local minimum of E, then

$$\nabla E|_{\boldsymbol{\omega}_o} = 0$$

This will then reduce eqn. 3.32 to

$$E(\boldsymbol{\omega}) \approx E(\boldsymbol{\omega}_o) + \frac{1}{2}(\boldsymbol{\omega} - \boldsymbol{\omega}_o)^T \mathbf{H}(\boldsymbol{\omega} - \boldsymbol{\omega}_o)$$

Thus a local approximation of the gradient around the minimum can be written

$$\nabla E \approx \mathbf{H}(\boldsymbol{\omega} - \boldsymbol{\omega}_o)$$

In the so called Newton method we solve for $\boldsymbol{\omega}_o$ in this expression to obtain

$$\boldsymbol{\omega}_o = \boldsymbol{\omega} - \mathbf{H}^{-1}\mathbf{g} \tag{3.33}$$

where we have defined $\mathbf{g} = \nabla E|_{\boldsymbol{\omega}}$. Because the local quadratic approximation is not always a good approximation one usually have to use this method in an iterative fashion. However, Newton’s method as presented here is not used without modifications since,

- It can be computationally demanding to compute $\mathbf{H}$.
- $\mathbf{H}$ must be inverted, which may be numerically difficult.
- If $\mathbf{H}$ is not positive definite ($\boldsymbol{\omega}^T\mathbf{H}\boldsymbol{\omega} > 0 \quad \forall\, \boldsymbol{\omega}$) then the Newton step can point away from the local minimum. This can be the case if we are not close enough to the local minimum.

One simple way of getting rid of the above problems is to use a diagonal approximation of the Hessian. This would result in the following update equation for the weights,

$$\Delta\boldsymbol{\omega}_i = -\left(\left|\frac{\partial^2 E}{\partial\boldsymbol{\omega}_i^2}\right| + \lambda\right)^{-1} \cdot \frac{\partial E}{\partial\boldsymbol{\omega}_i}$$

where $\lambda > 0$ but small constant.

Quasi Newton

The quasi Newton method builds up an approximation to the inverse Hessian over a number iterations. It has the following nice properties,

- One generates $\mathbf{G}(t)$ matrices that are better and better approximations of $\mathbf{H}^{-1}$.
- The method only uses gradient information to compute $\mathbf{G}(t)$.
- $\mathbf{G}(t)$ is always positive definite.

The most common method for computing $\mathbf{G}(t)$ is the so called BFGS (Broyden-Fletcher-Goldfarb-Shanno) method, where

$$\mathbf{G}(t+1) = \mathbf{G}(t) + \frac{\mathbf{p}\mathbf{p}^T}{\mathbf{p}^T\mathbf{v}} - \frac{(\mathbf{G}(t)\mathbf{v})\,\mathbf{v}^T\mathbf{G}(t)}{\mathbf{v}^T\mathbf{G}(t)\mathbf{v}} + \left(\mathbf{v}^T\mathbf{G}(t)\mathbf{v}\right)\mathbf{u}^T\mathbf{u} \tag{3.34}$$

where

$$\begin{aligned}
\mathbf{p} &= \boldsymbol{\omega}(t+1) - \boldsymbol{\omega}(t) \\
\mathbf{v} &= \nabla E(t+1) - \nabla E(t) \\
\mathbf{u} &= \frac{\mathbf{p}}{\mathbf{p}^T\mathbf{v}} - \frac{\mathbf{G}(t)\mathbf{v}}{\mathbf{v}^T\mathbf{G}(t)\mathbf{v}}
\end{aligned}$$

The update rule for the quasi Newton method is then,

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \alpha(t)\mathbf{G}(t)\underbrace{\mathbf{g}(t)}_{\nabla E(t)}$$

where $\alpha(t)$ is determined by a line minimization.

Levenberg-Marquardt (brief)

Another efficient minimization method is the Levenberg-Marquardt. The method only works for summed squared error functions.

$$E = \frac{1}{2}\sum_n e_n^2 = \frac{1}{2}\|\mathbf{e}\|^2$$

The new error after the move $\boldsymbol{\omega}_{old} \to \boldsymbol{\omega}_{new}$ is approximately

$$\mathbf{e}(\boldsymbol{\omega}_{new}) = \mathbf{e}(\boldsymbol{\omega}_{old}) + \mathbf{Z}(\boldsymbol{\omega}_{new} - \boldsymbol{\omega}_{old})$$

where

$$Z_{ni} = \frac{\partial e_n}{\partial \omega_i}$$

The new error to minimize is then

$$\tilde{E} = \frac{1}{2}\|\mathbf{e}(\boldsymbol{\omega}_{old}) + \mathbf{Z}(\boldsymbol{\omega}_{new} - \boldsymbol{\omega}_{old})\|^2$$

This can be minimized with respect to $\boldsymbol{\omega}_{new}$ to obtain,

$$\boldsymbol{\omega}_{new} = \boldsymbol{\omega}_{old} - \left(\mathbf{Z}^T\mathbf{Z}\right)^{-1}\mathbf{Z}^T\mathbf{e}(\boldsymbol{\omega}_{old})$$

(This looks very much like linear regression!)

A practical problem with the above formula is that the step size may be too large, because the linear approximation is not good enough. To overcome this we minimize a modified error,

$$\begin{aligned}
\tilde{E} &= \frac{1}{2}\|\mathbf{e}(\boldsymbol{\omega}_{old}) + \mathbf{Z}(\boldsymbol{\omega}_{new} - \boldsymbol{\omega}_{old})\|^2 + \lambda\|\boldsymbol{\omega}_{new} - \boldsymbol{\omega}_{old}\|^2 \\
&\Rightarrow \\
\boldsymbol{\omega}_{new} &= \boldsymbol{\omega}_{old} - \left(\mathbf{Z}^T\mathbf{Z} + \lambda\mathbf{I}\right)^{-1}\mathbf{Z}^T\mathbf{e}(\boldsymbol{\omega}_{old})
\end{aligned}$$

Note: If $\lambda >> 1$ then we are back to the gradient descent method.

General considerations regarding the choice of minimization method

There are many different minimization methods when it comes to training neural networks. The question is which of them to choose. We have the stochastic gradient descent (SGD) as some kind of baseline first order method. We can add a lot of enhancements to SGD, such as momentum, RMSPROP or Adam. On the other hand we also have more advanced first order methods such as conjugate gradient methods or second order methods, such Quasi Newton or Levenberg Marquart. For large datasets and for large networks second order methods may not be computationally feasible to use. But for small dataset and shallow networks they may be very efficient. Since deep learning is a lot about training large networks with large datasets second order methods are not used a lot. However one should always consider using them for smaller problems.

3.9 Misc things

There are other aspects of neural networks that we have not talked about, both practically and more theoretical. Some of the more practical details you may have to consider when you are working with neural networks are listed here.

3.9.1 Dimensionality reduction

This can also been seen as a way of doing preprocessing of input data. Here the goal is to reduce the number of inputs to the network. A very common method is here *PCA = (principal component analysis)*. We will talk about PCA later.

3.9.2 Missing values

A very common problem with real world data is *missing values*. This means that some of the input variables are missing for a part of the data set. Since the network need input values for each input node, one needs to replace the missing value with something new. The procedure of replacing missing values is called *imputation*. There are many different ways of doing missing data imputation. Here are a few alternatives:

- Average value: The average value for the input variable, over the full data set, excluding the missing ones are computed. Missing values are then replaced by this mean value. This procedure is straight forward for continuous-valued inputs. For binary inputs one often replaces the missing value by the most common category. This procedure can however result in biased results if the way missing values occur is correlated with the target value.

- Estimate: This means that the missing values is estimated by another model that is built in order to model this variable. The autoencoder network can be such a model. (We will talk more about autoencoders later.)

- Random imputation: Missing values are replaced by random values of the known variables. With this approach we can actually create many different datasets, each with a specific random imputation of missing values.

3.9.3 Ranking of input variables

Can we in some way rank the input variables in terms of importance for the classification or function approximation problem? This could give us important information about the problem in hand. Below is one example of how to rank variables using a trained neural network.

Compute the "saliency" for each input variable according to

$$S_k = \frac{1}{N} \sum_{n=1}^{N} \left| \frac{\partial y(\mathbf{x}_n)}{\partial x_k} \right|$$

Variables with "small" S_k are less important compared to variables with "large" S_k. Many other approaches to input variable ranking exists. Can you develop one of your own?

3.10 Radial basis function networks

The description of radial basis function (RBF) networks are provided mainly because of historical reasons and to show that alternatives to the MLP architecture exists. It is probably fair to say that RBF networks are not used so much today.

The network models discussed sofar have hidden nodes that are non-linear functions of the scalar product between an input vector and a weight vector.

$$h_j(\mathbf{x}) = \varphi\Big(\sum_k \omega_{jk} x_k(n)\Big) = \varphi\Big(\boldsymbol{\omega}_j^T \mathbf{x}\Big)$$

This is sometimes called a projective basis. We are now going to look at a class of neural networks where the activation of a hidden node is determined by the "distance" between the input vector and the weight vector.

$$h_j(\mathbf{x}) = f(||\mathbf{x} - \boldsymbol{\omega}_j||)$$

3.10.1 The interpolation problem

RBF networks have their origins in techniques for performing exact interpolation of a set of data points in a multi-dimensional space.

Given a set of N different points $\{\mathbf{x}_i \in \mathbb{R}^m | i = 1, 2, \ldots, N\}$ and a corresponding set of N real numbers $\{d_i \in \mathbb{R}^1 | i = 1, 2, \ldots, N\}$, find a function $F : \mathbb{R}^m \to \mathbb{R}^1$ that satisfies the interpolation condition:

$$F(\mathbf{x}_i) = d_i, \ i = 1, 2, \ldots, N \tag{3.35}$$

The *radial-basis-functions* technique has the form

$$F(\mathbf{x}) = \sum_{i=1}^{N} \omega_i \varphi(||\mathbf{x} - \mathbf{x}_i||) \tag{3.36}$$

where $\varphi(\cdot)$ is a (non-linear) function, called *radial basis* function. The known data points $\mathbf{x}_i$ are the centers of the radial-basis functions. Combining eqn. 3.35 and 3.36 gives the following set of equations for the coefficients ω_i,

$$\begin{bmatrix} \varphi_{11} & \varphi_{12} & \cdots & \varphi_{1N} \\ \varphi_{21} & \varphi_{22} & \cdots & \varphi_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ \varphi_{N1} & \varphi_{N2} & \cdots & \varphi_{NN} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix} \tag{3.37}$$

where $\varphi_{ij} = \varphi(||\mathbf{x}_i - \mathbf{x}_j||)$. The question is now if we can solve this system or equivalently find the inverse of the matrix $\Phi = \{\varphi_{ij}\}_{i,j=1}^{N}$. *Micchelli's theorem* states that a large class of radial basis functions can be used here and where the most important ones, considering ANN, are the *Gaussian functions*

$$\varphi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right)$$

Here σ is a parameter and $r \in \mathbb{R}$.

3.10.2 RBF neural networks

We get a RBF neural network by introducing a set of modifications to the above idea:

1. The number (M) of basis functions is usually $<$ then the number of data points (N).
2. The centers of the basis functions need not be data points. Determination of suitable centers is part of the training procedure.
3. $\sigma \to \sigma_j$, meaning that we introduce individual widths for each basis function.
4. Bias parameters are included in the linear sum.

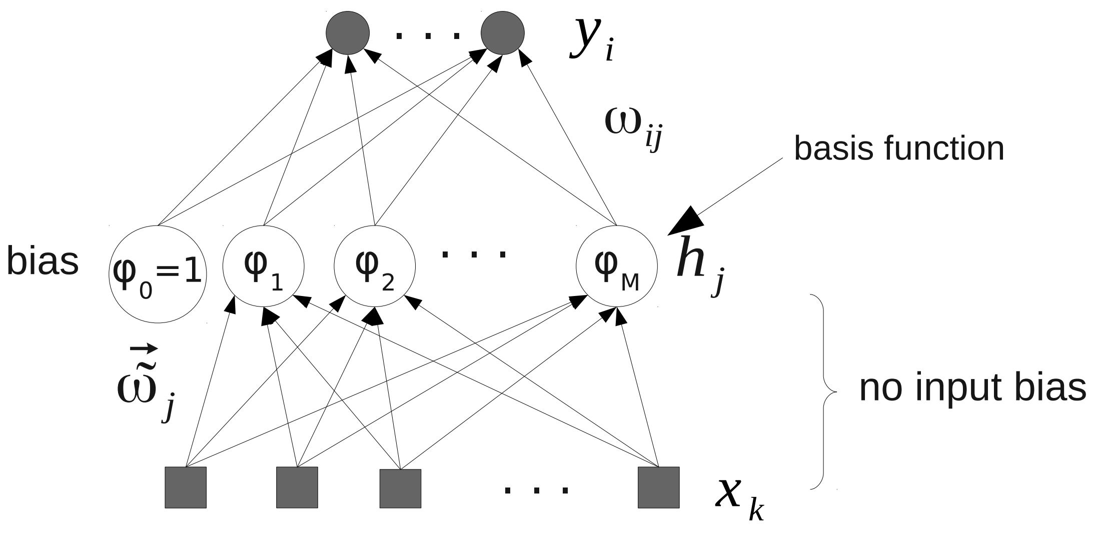


Figure 3.30: The structure of a RBF network (suitable for regression problems).

This leaves us with the following RBF network (see figure 3.30). The output from the RBF network is just a weighted sum of the basis functions,

$$y_i(\mathbf{x}) = \sum_{j=1}^{M} \omega_{ij} h_j(\mathbf{x}) + \omega_{i0}$$

The weights in the input layer (hidden node vectors) are centers for the different basis functions.

$$h_j(\mathbf{x}) = \varphi_j(\mathbf{x}) = \exp\left(-\frac{||\mathbf{x} - \tilde{\boldsymbol{\omega}}_j||^2}{2\sigma_j^2}\right)$$

- In the RBF network the input vector is compared to a “reference” vector. The scaled distance between them is used instead of the scalar product as in MLP networks.
- The output from this RBF network is a linear combination of the basis functions.
- Since the basis functions (hidden nodes) are localized they receive information only from a local part of the input space.

3.10.3 Learning strategies

Because the weights ω_{ij} (hidden-output) and $\tilde{\boldsymbol{\omega}}_j$ (input-hidden) have different meaning the training procedure (is often) different from an MLP.

Input-hidden weights

- Fixed, using random selection of data points. Choose randomly M input data points and put
$$\tilde{\boldsymbol{\omega}}_j = \mathbf{x}(j) \qquad j = 1, ..., M$$
- Fixed, using a clustering method (e.g. K-means clustering). The obtained cluster centers $\boldsymbol{\mu}_j$ are used
$$\tilde{\boldsymbol{\omega}}_j = \boldsymbol{\mu}_j \qquad j = 1, ..., M$$
- Learnable (see below)

The widths of the basis functions must be chosen so that they are not too peeked or too flat. One suggestion is
$$\sigma_j = \frac{1}{M} \max_{j'} \left(||\tilde{\boldsymbol{\omega}}_j - \tilde{\boldsymbol{\omega}}_{j'}|| \right)$$

Hidden-output weights

If we have fixed the input-hidden weights it becomes an easy task to find the ω_{ij} weights since now $\mathbf{y}$ is just a linear combination of the hidden nodes. From the training of the perceptron using a linear output function we had
$$\boldsymbol{\omega} = \mathbf{R}_{xx}^{-1} \mathbf{r}_{xd}$$
We can write the solution for the RBF weights in the same way
$$\boldsymbol{\omega} = \mathbf{R}_{hh}^{-1} \mathbf{R}_{hd}$$
where

- $\mathbf{R}_{hh}$ = correlation matrix for the hidden signals
- $\mathbf{R}_{hd}$ = correlation matrix between output and hidden signal

Another learning strategy is to train all parameters using gradient descent. Use the summed squeare error function
$$E(\boldsymbol{\omega}) = \frac{1}{N} \sum_n \sum_i \left(y_i(n) - d_i(n) \right)^2$$
where
$$y_i(n) = \sum_{j=1}^{M} \omega_{ij} \exp\left(-\frac{||\mathbf{x}(n) - \tilde{\boldsymbol{\omega}}_j||^2}{2\sigma_j^2} \right) + \omega_{io}$$

And then train using the following update equations:

$$\begin{aligned}
\Delta\omega_{ij} &= -\eta_1 \frac{\partial E}{\partial \omega_{ij}} \\
\Delta\tilde{\omega}_{jk} &= -\eta_2 \frac{\partial E}{\partial \tilde{\omega}_{jk}} \\
\Delta\sigma_{j} &= -\eta_3 \frac{\partial E}{\partial \sigma_{j}}
\end{aligned}$$

A drawback with this learning strategy is speed. It usually takes much longer to train this way.

3.10.4 RBF versus MLP

- MLP uses a global representation and RBF a local

$$\begin{aligned}
\varphi\left(\mathbf{x}^T \boldsymbol{\omega}\right) &\rightarrow \text{global} \\
\varphi\left(\frac{||\mathbf{x} - \tilde{\boldsymbol{\omega}}_j||}{\sigma}\right) &\rightarrow \text{local}
\end{aligned}$$

- The MLP can have a complicated structure (many hidden layers), where RBF has a simple 1-hidden layer structure.
- For MLP:s all weights are determined in one learning session, but for the RBF the weights are determined at different stages.
- RBF suffers more from the "curse of dimensionality" than the MLP does.