6. Backpropagation training

6.1 Background

To understand well how a feedforward neural network is built and it functions, we consider its basic first steps. We return to its “history” for a while.

In 1949 D. Hebb proposed a *learning rule* that was later named Hebbian learning. He postulated his rule of reinforcing *active* connections only between neurons. In more modern approach connections are either strengthened and weakened. Since the learning is guided by knowing the target (class), this is known as supervised learning.

The learning paradigm for a single perceptron neuron or node can be summarised as follows.

- Set the weights and the threshold *Θ* of the Heaviside function f_h (Section 2, p. 64) randomly

$$y = f_h\left(\sum_{i=1}^{p} w_i x_i - b\right) \qquad (6.1)$$

where f_h is a step function with *p* variables and bias *b*= *Θ*

$$f_h(x) = 1 \quad \mathrm{x} > 0$$
$$f_h(x) = 0 \quad \mathrm{x} \leq 0 \qquad (6.2)$$

and which produces outputs either 1 or 0.

- Give an input.
- Compute the output by taking the threshold value of the weighted sum of the inputs.
- Alter the weights to reinforce correct decisions (classifications) and weaken incorrect decisions - reduce the error.
- Give the next input and continue the process until stopping criterion is true, no change in weights.

Next we show the most elementary perceptron algorithm for a single neuron, such as in Fig. 6.1.

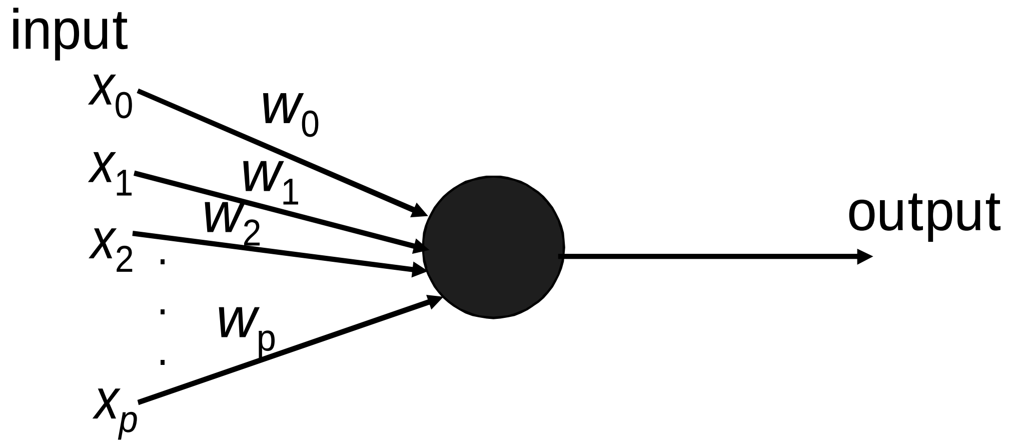


Fig. 6.1 The basic model of one neuron.

Perceptron learning algorithm

1. Initialise weights and threshold

Define $w_i(t)$, i=0,..,p, as a weight from input i at time and Θ as a threshold value of the node. Set bias w_0=- Θ and constant x_0=1. Assign $w_i(0)$ to be small random values, i=1,..,p.

2. Give input and desired target output

Present input x_i, i=1,..,p and desired output (class) $y(t)$.

3. Compute output

$$y = f_h\left(\sum_{i=0}^{p} w_i x_i\right) \qquad (6.3)$$

4. Train weights, $i=1,..,p$

If a correct classification	$w_i(t+1) = w_i(t)$
If output 0, but should be 1 (Class A)	$w_i(t+1) = w_i(t)+x_i(t)$
If output 1, but should be 0 (Class B)	$w_i(t+1) = w_i(t)-x_i(t)$

5. Iterate (from Step 2) until no change is encountered.

This was the most basic perceptron algorithm of all. The next development was the introduction of learning rate $0<\eta<1$, making the network take smaller step towards the solution.

The Step 4 was modified.

4. Train weights, i=1,..,p

If a correct classification	$w_i(t+1) = w_i(t)$
If output 0, but should be 1 (Class A)	$w_i(t+1) = w_i(t)+\eta x_i(t)$
If output 1, but should be 0 (Class B)	$w_i(t+1) = w_i(t)-\eta x_i(t)$

Widroff and Hoff proposed a quite similar algorithm after having realised that it would be best to to change the weights much when the weighted sum is a long way from the desired value, while altering them only slightly when the weighted sum is close to that required to give the correct solution. Weight adjustment is then carried out in proportion to that error.

Using the Widrow-Hoff delta rule, the error term can be expressed as

$$\Delta = y(t) - \widehat{y}(t) \qquad (6.4)$$

where *y*(*t*) is the desired response of the neuron and *y*(*t*)-hat is the actual response, either 0 or 1. Thus, Δ will be either -1 or +1 if the two values are not equal. If they are equal, the weights are unchanged.

Then Step 4 was modified in the following way.

4. Train weights according to Widroff-Hoff delta rule

$$\Delta = y(t) - \widehat{y}(t)$$

$$w_i(t+1) = w_i(t) + \eta \Delta x_i(t)$$

$$y(t) = \begin{cases} +1, \text{if input from Class A} \\ 0, \text{if input from Class B} \end{cases}$$

where 0<η<1 is the learning rate.

Let us look at the weights in the form of vector w so that its direction in the variable space is taken into account. Then (in the 2-dimensional space) the training or changing the weight vector can be shown like in Fig. 6.2. Fig. 6.3 shows the process until a successful classification.

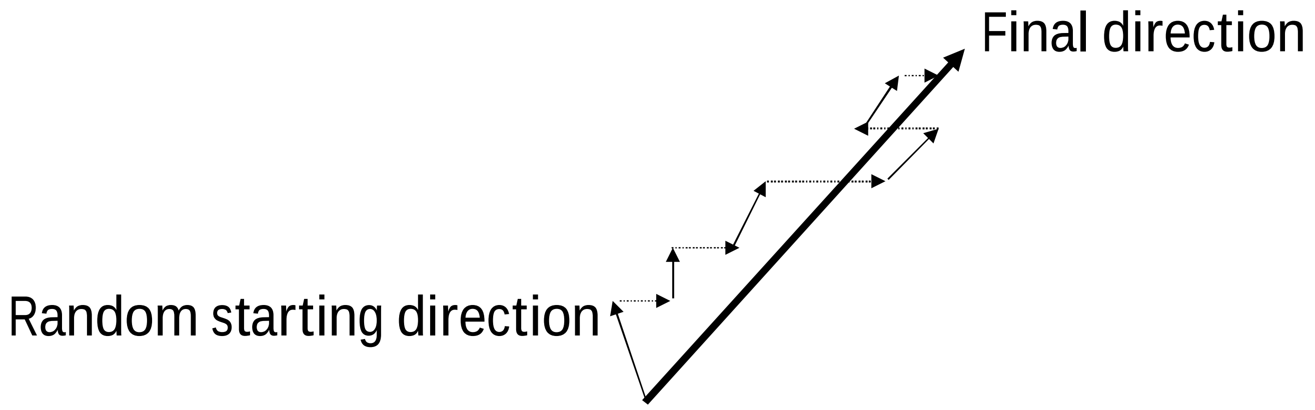


Fig.6.2 Training the weight vector.

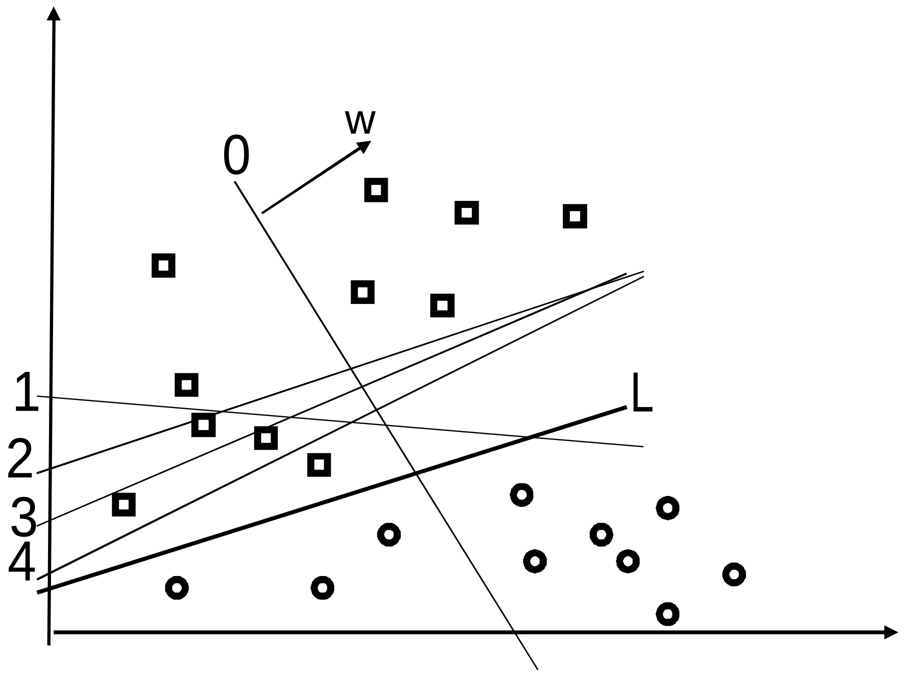


Fig. 6.3 Beginning from a random normal direction of line 0 the weight vector w is gradually trained to define the normal of L that separates two classes.

We saw, intuitively, how the perceptron learning rule produces a solution. It was proved by Rosenblatt who stated that, given it is possible to classify a series of inputs, i.e., two classes can be separated with a line, then a perceptron will find that classification. In other words, he proved that the perceptron weight vector would eventually align itself with the ideal weight vector, and would not oscillate around it for ever.

In 1969 M. Minsky and S. Papert showed that a perceptron including only one (processing) layer is incapable of solving some rather simple nonlinear problems such as exclusive OR (XOR).

In 1986 J.L. McClelland and D.E. Rumelhart introduced their ideas to apply more than one processing layers which boosted rapid development in the area of neural network research.

In addition, noticing the advantage of nonlinear activation function (sigmoid or others based on the use of exponential function e^{-a} where non-negative *a* is activation) at least for one layer was also necessary. This enabled the development of the basic backpropagation learning algorithm and several others. It became possible to perform complicated multiclassification or regression tasks.

Before we will make acquaintance of backpropagation training, we begin from the basic situation presented in Section 2 on p. 65, where two classes of cases were entirely separable shown by a linear classifier. This is, naturally, a simplified situation not present in real data sets. Still, it is a good beginning to better understand the continuation guiding forward.

Let us use two lines defined by extending the presentation of p. 65, (bias $b=w_{l0}$, $x_0=1$) to obtain the below expressions

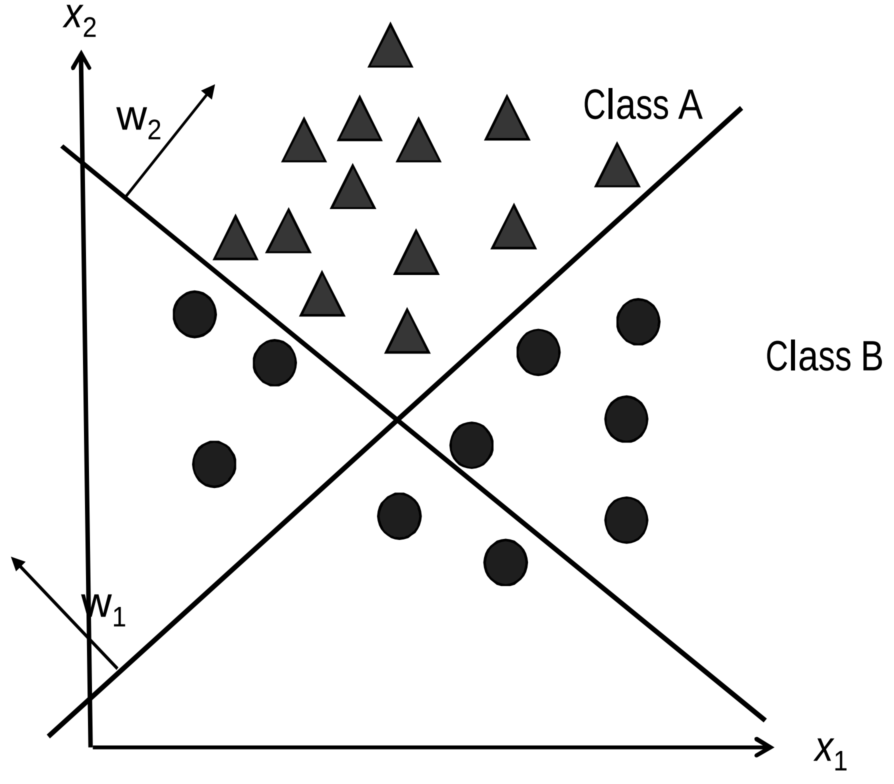
$$\sum_{i=0}^{p} w_{1i}x_i = \mathbf{w}_1 \cdot \mathbf{x} \qquad \sum_{i=0}^{p} w_{2i}x_i = \mathbf{w}_2 \cdot \mathbf{x} \tag{6.5}$$

depicted in Fig. 6.1. Vector x represents any case in the variable space (in Fig. 6.4 there are only two variables, p=2).

Instead of lines, planes are used for three-dimensional variable spaces and generally hyperplanes for higher dimensions, that is, real data sets.

Fig. 6.4 Piecewise linear classification for two non-linearly separable classes that cannot be separated with one line only, i.e., linearly.

Now the intersection of the positive (upper) half spaces defined by the two lines contains all cases of Class A. The other three sections include all cases of Class B. This implies that by adding complexity, more separating lines or mappings generally, more complicated divisions can be made.



6.2 Backpropagation approach

The *backpropagation* (BP) network is a multilayer feedforward network with a different transfer or activation function, such as sigmoid, for a processing node and a more powerful learning algorithm. (However, terms used may vary and, sometimes, the term of multilayer perceptron (MLP) network is used to mean the BP network.)

The cases (instances, items, samples,observations, patterns or examples) of the training set to be used for building a BP network (model) must be input many times in order for weights between nodes to settle into a state for correct classification of input cases.

While the network can recognise cases similar to those learnt so far, they are not able to recognise fully new (different) cases. To recognise these, the network has to be retrained with these as training cases along with previously known. If only new ones are provided for retraining, then old cases may be forgotten. In this way, learning is not incremental over time. This is the major limitation for supervised learning networks. In addition, the backpropagation network is prone to local minima. To overcome this limitation, some attachment or ”add-on” should be employed, for instance, *simulated annealing*, to be considered later.

Backpropagation is a learning method, which is a gradient descent technique with backward error propagation (Fig. 6.5).The training case set for the network has to be presented many times in order to train or change network weights to model or map the chracteristics of training data.

A gradient descent procedure searches for the solution along with the negative of the gradient, steepest descent. The gradient concept is the multidimensional extension for the one-dimensional derivative. For a gradient, symbol ∇ called nabla is used to constitute the partial derivative for each dimension (variable) x_i, i=1,...,p, all of these forming the vector (6.6).

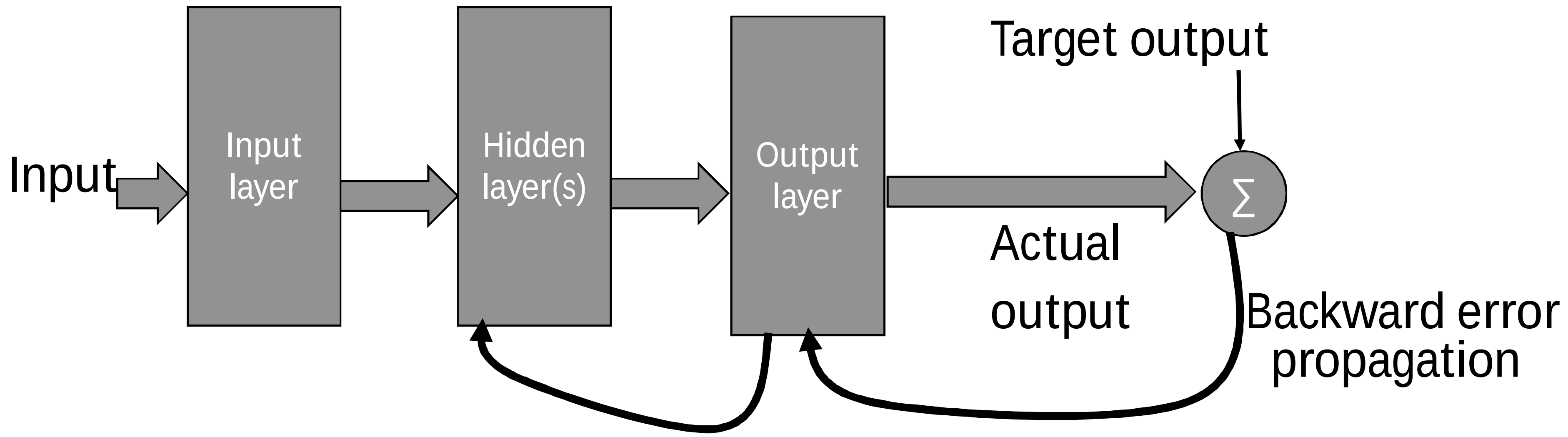


Fig. 6.5 The backpropagation network.

$$\nabla J(\mathbf{x}) = \begin{pmatrix} \dfrac{\partial J(\mathbf{x})}{\partial x_1} \\ \dfrac{\partial J(\mathbf{x})}{\partial x_2} \\ . \\ . \\ . \\ \dfrac{\partial J(\mathbf{x})}{\partial x_p} \end{pmatrix} = \left(\frac{\partial J(\mathbf{x})}{\partial x_1} \frac{\partial J(\mathbf{x})}{\partial x_2} \cdots \frac{\partial J(\mathbf{x})}{\partial x_p} \right)^T \qquad (6.6)$$

Suppose x is the solution vector that minimises the criterion function J(x). Beginning with an arbitrarily chosen vector x_1, the procedure finds the solution vector by iteratively applying the formula

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta_k \nabla J(\mathbf{x}_k) \qquad (6.7)$$

until convergence, where η_k is a positive coefficient that sets the step size and subscript *k* denotes the *k*th iteration and (nabla) ∇ is the gradient operator.

Training is a search for the set of weights that will make the neural network to have lowest error for a training set. The training technique utilise minimisation methods to avoid performing exhaustive (brute-force) search over all weight values. This would be impossible even for small networks because of virtually infinite number of weight combinations.

The sign or direction (up or down) of the gradient shows the following information. For a negative gradient, the weight should be increased to achieve a lower error. For a positive gradient, the weight should be decreased to yield a lower error. For a zero gradient, the weight is not contributing to the error of the network. See Fig. 6.6.

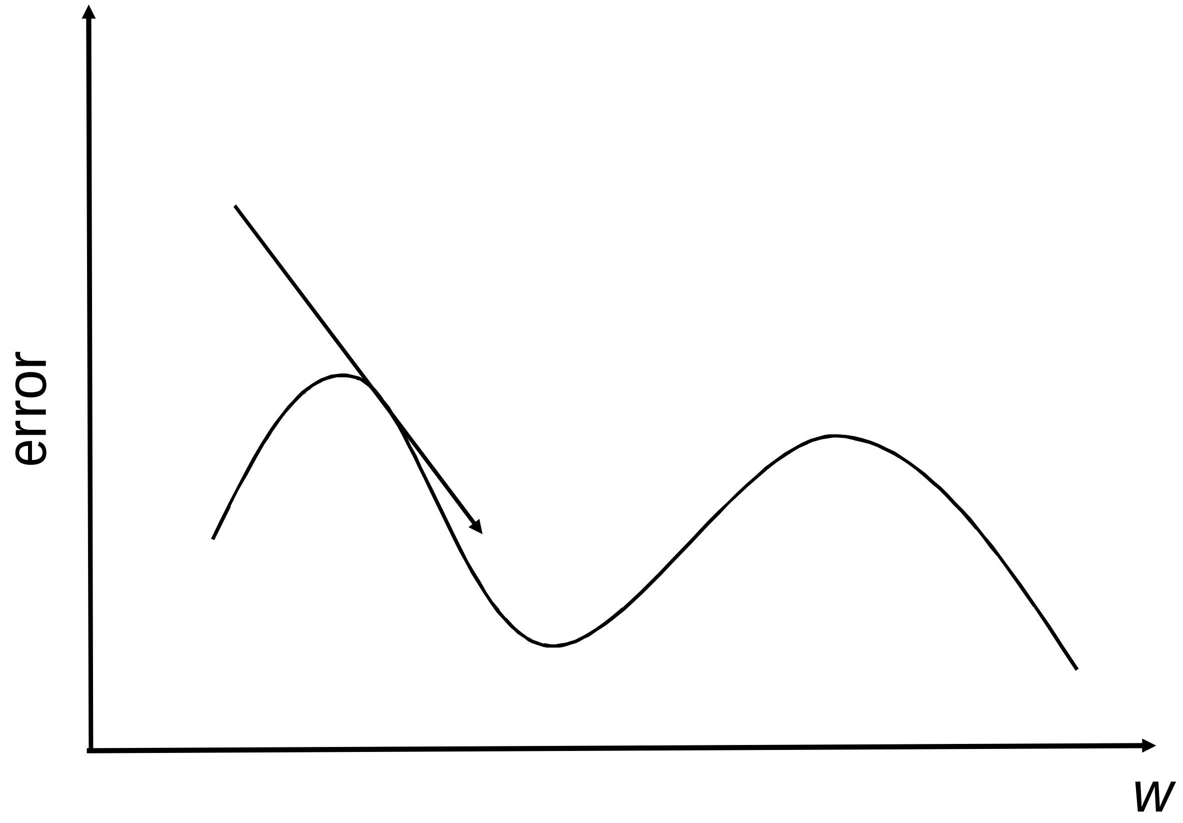


Fig. 6.6 Gradient vector of a single weight.

Each of the weights is considered as a variable in training, because these weight values will change independently as the neural network changes. The partial derivatives of the weights indicate the independent influence of each weight on the error function.

The backpropagation algorithm originating from Rumelhart, Hinton and Williams (1986) is formulated below.

Backprogation algorithm

- Weight initialisation

Set all weights and node thresholds to small random numbers. The node threshold is the negative of the weight from the bias node, in which the activation level is fixed at 1.

- Activation computation

1. The activation level of an input unit is determined by the case presented to the network.

2. The activation level o_j of a hidden and output node is given by

$$o_j = \phi\left(\sum w_{ij} o_i - \theta_j\right) \tag{6.8}$$

where w_{ij} is the weight from input o_i, Θ_j is the node threshold and ϕ is a sigmoid activation function (or some else).

$$\phi(a) = \frac{1}{1+e^{-a}} \tag{6.9}$$

- Weight training

1. Start at the ouput nodes and process backward to the hidden layers with a recursive equation. Adjust weights by

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) \tag{6.10}$$

where $w_{ij}(t)$ is the weight form node *i* to node *j* at time (iteration) *t* and $\Delta\, w_{ij}(t)$ is the weight adjustment.

2. The weight change is computed by

$$\Delta w_{ij}(t) = \eta \delta_j o_i \qquad (6.11)$$

where η is learning rate, $0 < \eta < 1$, for example 0.1 and δ_j the error gradient at node j. Convergence is often faster by adding a momentum term ($t>0$)

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_j o_i + \alpha\left(w_{ij}(t) - w_{ij}(t-1)\right) \qquad (6.12)$$

where $0<\alpha<1$.

3. The error gradient is given for the output nodes by

$$\delta_j = \hat{y}_j (1 - \hat{y}_j)(y_j - \hat{y}_j) \qquad (6.13)$$

where y_j is the desired target output, such as known class, y_j–hat is the actual output activation at output node *j*, and for the hidden nodes by

$$\delta_j = o_j (1 - o_j) \sum_k \delta_k w_{jk} \qquad (6.14)$$

where o_j is actual output activation and δ_k is the error gradient at node *k* to which a connection points from hidden node *j*.

4. Iterate until convergence in terms of the selected error criterion. An iteration includes presenting a case, computing activations and changing weights.

There are various stopping criteria. One is based on error to be minimised. Since not all training cases are normally classified into correct classes, a fixed threshold or cutoff value is used so that the procedure is stopped if the error is below it. However, the criterion does not guarantee generalization to new data, i.e., an appropriately modified new model.

Another criterion is based on the gradient. The procedure is terminated when the gradient is sufficiently small. Note that the gradient will be zero at a minimum by definition.

A third one is based on crossvalidation performance (using a validation set separate from the training and test sets to compute an error value and to stop if this increases). This can be used to monitor generalization performance during learning and to terminate when there is no more improvement.

A simple ”elementary” stopping criterion is to stop after the maximum number of iterations. More than one criterion can also be applied jointly, and, for example, the maximum number of iterations can be the last stopping alternative to be ”fired”.

The name backpropagation comes from the technique in which the error (gradient) of hidden nodes are derived from propagating backward the errors of output nodes calculated by the previous Eq. (6.14) since the target values for the hidden nodes are not given.

The sigmoid activation function is good when it compresses the output value into range [0,1] and hyperbolic tangent into [-1,1]. (On the other hand, however, such a property is not always necessarily desirable.) This accommodates large signals without saturation while allowing the passing of small signals without excessive attenuation. Further, it is a continuous, smooth function so that gradients required for a gradient descent search can be calculated easily in the closed form.

Example

We still consider XOR problem and build a feedforward network as given in Fig. 6.7.

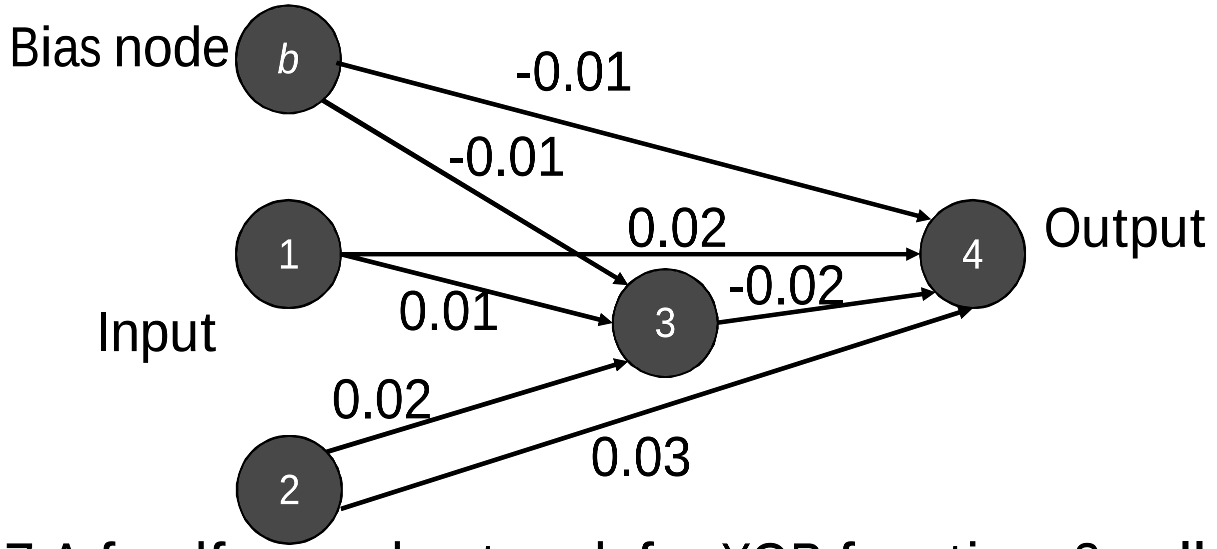
The weights are initialised randomly:

W_{13}=0.01, W_{14}=0.02, W_{23}=0.02, W_{24}=0.03, W_{34}=-0.02, W_{b3}=-0.01, W_{b4}= -0.01

To compute activation, a training case of input vector (1,1) and desired target (”class”) is employed (bias is always 1):

$$o_1 = o_2 = 1$$

$$o_3 = \frac{1}{1+e^{-(1\cdot 0.01+1\cdot 0.02-1\cdot 0.01)}} = 0.505 \quad o_4 = \frac{1}{1+e^{-(0.505\cdot(-0.02)+1\cdot 0.02+1\cdot 0.03-1\cdot 0.01)}} = 0.508$$



Inputs	Output
(1,1)	0
(1,0)	1
(0,1)	1
(0,0)	0

Fig. 6.7 A feedforward network for XOR function. Small random initial weights are attached to the arcs or connections. (Note this network is specific when there are arcs from the input layer to both later layers. Normally in practice there is always layer by layer, only arcs from each layer to its following layer.)

Next the weight training follows assuming the learning rate η=0.3 (actually this is normally at most 0.1):

$$\delta_4 = 0.508(1-0.508)(0-0.508) = -0.127$$
$$\Delta w_{14} = 0.3 \cdot (-0.127) \cdot 1 = -0.038$$
$$\delta_3 = 0.505(1-0.505)(-0.127 \cdot (-0.02)) = 0.0006$$
$$\Delta w_{12} = 0.3 \cdot 0.0006 \cdot 1 = 0.0002$$

The rest of the weight training adjustments are omitted. Note that the threshold which is the negative of the weight from the bias node is adjusted likewise. It takes several iterations like this before the training process stops. The following set of final weights gave the mean squared error of less than 0.01.

w_{13}=5.62, w_{14}=4.98, w_{23}=5.62, w_{24}=4.98, w_{34}=-11.30, w_{b3}=-8.83, w_{b4}= -2.16

Derivation of backpropagation

Next we consider how to derive backpropagation training rule given by

$$\Delta w_{ij} = \eta \delta_j o_i \qquad (6.15)$$

If node *j* is an output node, then its error gradient is computed with

$$\delta_j = \left(y_j - \widehat{y}_j\right)\phi'\left(a_j\right) \qquad (6.16)$$

where

$$a_j = \sum_i w_{ij} o_i \qquad (6.17)$$

Here ϕ is a sigmoid function and

$$\hat{y}_j = \phi(a_j) = \phi\left(\sum_i w_{ij} o_i\right) \tag{6.18}$$

If node *j* is a hidden node, then the error gradient is given by

$$\delta_j = \phi'(a_j) \sum_k \delta_k w_{jk} \tag{6.19}$$

The backpropagation procedure minimises the error criterion

$$E = \frac{1}{2} \sum_j \left(y_j - \hat{y}_j\right)^2 \tag{6.20}$$

Gradient descent yields

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{6.21}$$

According to the chain rule, we obtain

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial w_{ij}} \tag{6.22}$$

In the case when node *j* is an output node,

$$\frac{\partial E}{\partial \hat{y}_j} = -\left(y_j - \hat{y}_j\right) \text{ and } \frac{\partial \hat{y}_j}{\partial w_{ij}} = \phi'\left(a_j\right) o_i \tag{6.23}$$

Thus
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \widehat{y}_j}\frac{\partial \widehat{y}_j}{\partial w_{ij}} \qquad (6.24)$$
$$= -(y_j - \widehat{y}_j)\phi'(a_j)o_i$$
$$= -\delta_j o_i$$

So we achieve
$$\Delta w_{ij} = \eta \delta_j o_i \qquad (6.25)$$

When node *j* is a hidden node, there is no *y*$_j$. The chain rule produces
$$\frac{\partial E}{\partial o_j} = \sum_k \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial o_j} \qquad (6.26)$$

The ouput of node *k* is given by

$$o_k = \phi\left(\sum_j w_{jk} o_j\right) \tag{6.27}$$

Thus, the term $\partial o_k / \partial o_j$ can be transformed by

$$\frac{\partial o_k}{\partial o_j} = \phi'(a_k) w_{jk} \tag{6.28}$$

This results in

$$\begin{aligned}\frac{\partial E}{\partial o_j} &= \sum_k \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial o_j} \\ &= -\sum_k (y_k - o_k)\phi'(a_k) w_{jk} \qquad (\text{here } o_k = \hat{y}_k) \\ &= -\sum_k \delta_k w_{jk}\end{aligned} \tag{6.29}$$

Furthermore

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial w_{ij}} \\ &= -\left(\sum_k \delta_k w_{jk}\right)\phi'(a_j)o_i \\ &= -\delta_j o_i\end{aligned} \tag{6.30}$$

We obtain

$$\Delta w_{ij} = \eta \delta_j o_i \tag{6.31}$$

(This presentation only considered a network with one hidden layer.)

To conclude, perhaps the most essential issue here was to notice the difference in formulas (6.16) and (6.19).

Learning rate can be reduced along with iterations t, $\eta(t)$, so the steps forward become shorter which aids to make fine-tuning looking for a minimum. On the other hand, this may result in problem if the minimum is local such as in Fig.6.8. Learning rate can be attempted to increase back temporarily. Further, momentum, Eq. (6.12) on p. 177, is efficient in order to jump over a ”ridge” in Fig. 6.8 and Fig. 6.9.

Addition of random noise (slight changes in weights trained) can be used to perturb the gradient descent method from the track of the deepest descent, and often this noise is enough to knock the process out of a local minimum.

Local minima can also be passed by using simulated annealing, a method to be considered later.

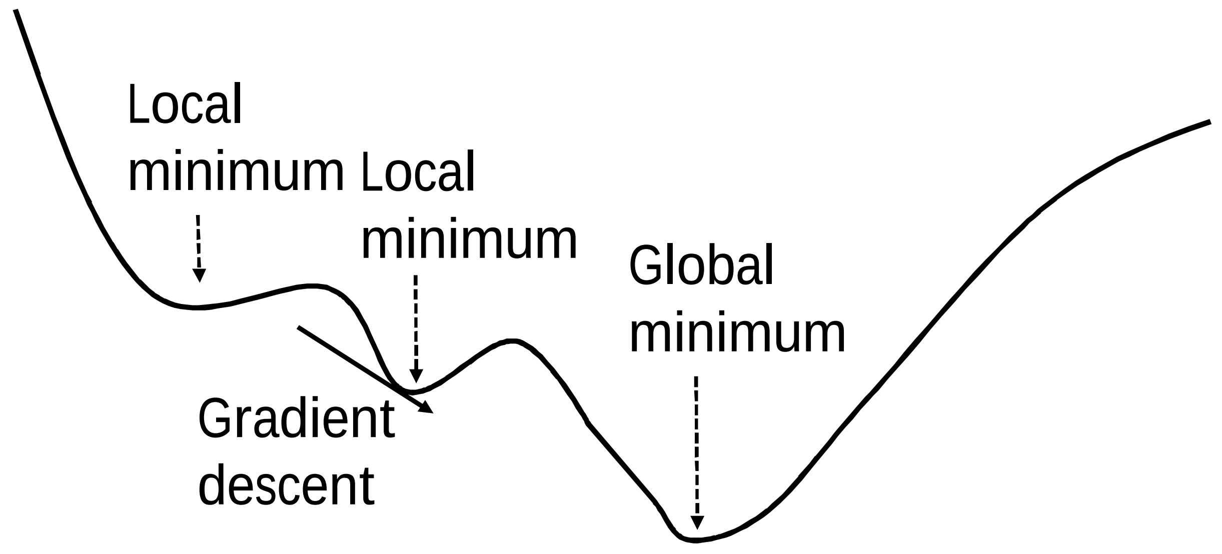


Fig. 6.8 One-dimensional mapping or cross section of an error surface along the gradient.

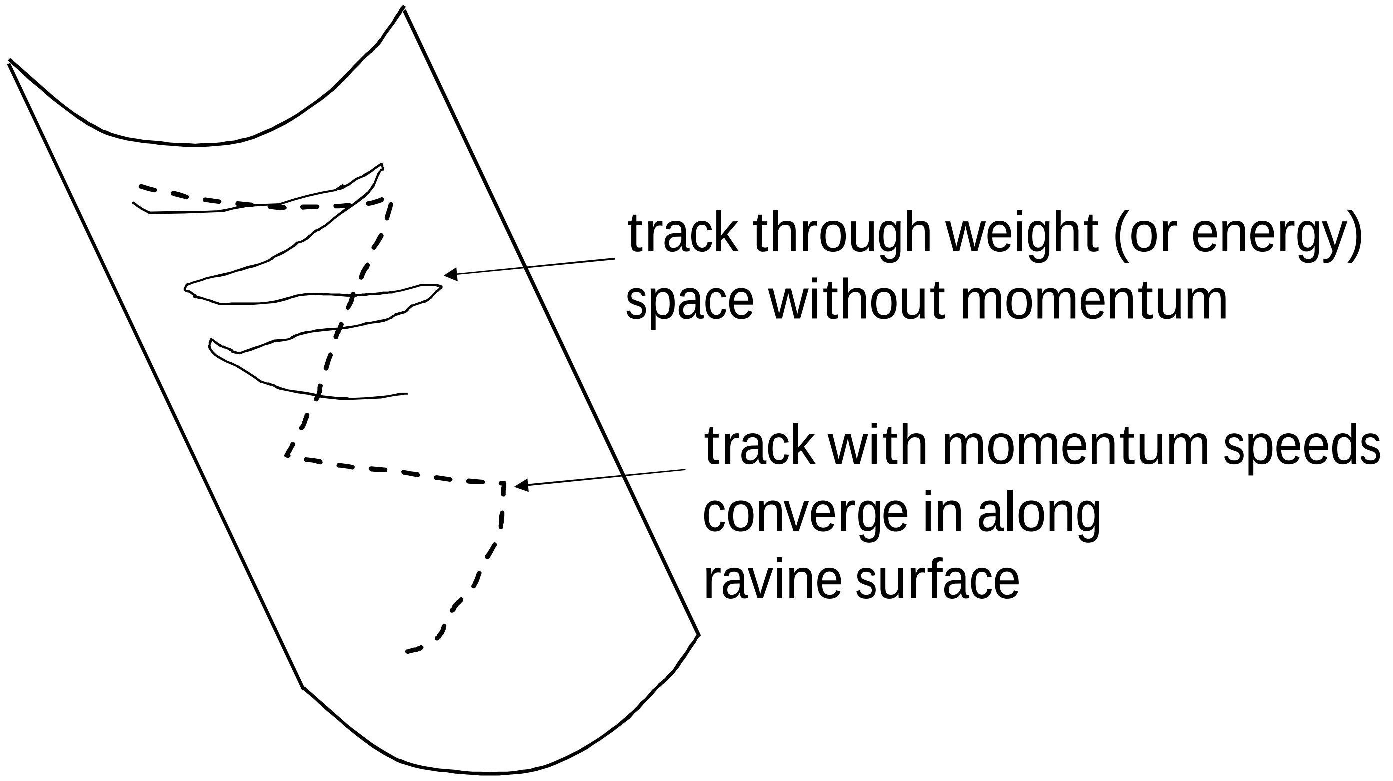


Fig. 6.9 The addition of a momentum term can speed up convergence, particularly along a ravine.

Subject to *complexity of learning*, it has been shown that the problem of finding a set of weights for a network which performs the desired mapping exactly for given training data is NP-complete. That is, we cannot find optimal weights in polynomial time.

Learning algorithms like backpropagation are gradient descent methods which seek only a local minimum. These algorithms usually do not take exponential time (required by NP) to run. The empirically tested learning time with a serial (non-parallel) computer for backpropagation is about $O(N^3)$ where N is the number of weights (Hinton 1989). Just to compare, the time complexity of self-organising map is $O(nm^2)$, in which n is the number of input cases and m that of (output) nodes. However, these repsentations may vary in the literature depending on what has actually analysed, e.g., taking stopping criteria or convergence into account.

6.3 Derivatives of the activation functions

While basic feedforward neural networks use sigmoid, hyperbolic tangent and linear activation functions, deep learning networks employ linear, softmax and rectified linear unit (ReLU) ones

The derivative of the linear function

$$\phi(x) = x \tag{6.32}$$

is simply

$$\phi'(x) = 1 \tag{6.33}$$

Typically, the softmax activation function, along with the linear activation function, is used only on the output layer of the neural network. Non-linearity is often needed in preceding layers to enable the learning of complex data.

The softmax activation function

$$\phi_i = \frac{e^{z_i}}{\sum\limits_{j \in otput\ nodes} e^{z_j}} \tag{6.34}$$

differs from those others in that its value is dependent on the other output nodes, not just on the output node currently being calculated.

The z=(z_1,…,z_C) vector represents the output from all output nodes. The derivative of the softmax activation function is as a partial derivative

$$\frac{\partial \phi_i}{\partial z_i} = \phi_i\left(1-\phi_i\right) \qquad (6.35)$$

when we remember that the derivative of e^x equals the function itself. This was presented as a partial derivative, because it is used for the gradient of output vector z.

The derivative of the sigmoid activation function is the following.

$$\phi'(x) = \phi(x)(1 - \phi(x)) \qquad (3.36)$$

Recognise that the advantage of derivating these functions including e^x is that, in a way, no actual derivation in the closed (symbolic) form is necessary, because we can derive the derivative formula on the basis of function $\phi(x)$ itself. In fact, these are like recurrence formulas.

The derivative of the hyperbolic tangent function is the following.

$$\phi'(x) = 1 - \phi(x)^2 \qquad (3.37)$$

Ultimately, the derivative of the rectified linear unit (ReLU) activation function is straightforwardly.

$$\phi'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \le 0 \end{cases} \qquad (3.38)$$

Strictly thinking, rectified linear unit (ReLU) does not have a derivative at 0. However, because of convention, the gradient of 0 is substituted when *x* is 0.

6.4 Applying backpropagation

Backpropagation is a training method that adjusts the weights of the network with its gradients computed. This is a form of gradient descent since the process descends the gradients to lower values. While adjusting weights, the network should yield more desirable outcomes. The global error of the network computation process should fall during training.

Training can be performed through two approaches called online and batch training. In the former the weights are modified after input every single training case (element). Training progresses to the next training set case and also calculates an update to the network to complete one iteration. Using all cases once is called an epoch. These are normally made many times, sometimes even thousands or tens of thousands.

Batch training also utilises all the training cases. Nonetheless, the weights are not updated for every iteration. Instead, we sum the gradients for each training set case. Once we have summed these, we can update the network weights.

Sometimes, we can set a batch size. For example, if there are 10 000 cases in the training set, the process could update the weights every 1000 cases, thereby causing the neural network weights to be updated ten times during one epoch.

In *stochastic gradient descent* a training case is selected randomly, not as above one by one for the whole training set and this all to be repeated. Choosing random training case often converges to an acceptable weight faster than looping through the entire training set for each epoch.

Momentum and *learning rate* contribute to the success of training. Choosing the suitable momentum and learning rate can impact the effectiveness of training. The learning rate affects, as said, the speed at which a neural network learns. Decreasing it makes the training more meticulous. A higher learning rate might skip past optimal weight settings. A lower one will mostly produce better results, but can greatly increase running time. Lowering the learning rate as the network learns can be an effective technique.

The momentum is a tool to combat local minima. A higher momentum value can push the process past local minima encountered along the track.

At the beginning of the process, the learning rate is at most 0.1. The momentum is typically set to 0.9.

6.5 Other propagation training

The backpropagation algorithm has influenced many training algorithms, for example, the above-mentioned stochastic gradient descent. There are also others, such as two popular and efficient algorithms named resilient propagation and Levenberg-Marquardt algorithm.

Resilient propagation (Reidmiller and Braun 1993) differs from backpropagation in the way of the use of gradients. Although the resilient propagation does contain training parameters, these are not defined by the user, but normally given by default.

The resilient propagation keeps an array of update values for the weights, which determines how much each weight is altered. Now this is much better compared with backpropagation, because the algorithm adjusts individually the upadate value of every weight as training progresses.

The update values are started at the default of 0.1, according to the initial update values argument.

Another approach for an already trained neural network is to save the update values once training stops and use them for the new training. This will allow to resume training without the initial spike in errors normally seen when resuming training.

Other parameters not determined by the user are also used for training.

Levenberg-Marquardt algorithm is an efficient training method for feedforward neural networks and often outperforms the resilient propagation algorithm. Levenberg (1940) introduced its foundation in numerical optimization, and Marquardt (1963) expanded it.

Levenberg-Marquardt algorithm is a hybrid algorithm on the basis of Newton’s method and gradient descent (backpropagation). Levenberg-Marquardt algorithm is an approximation of Newton’s method (Hagan and Menhaj 1994). Although gradient descent is guaranteed to converge to a local minimum (that might also luckily be global), it is slow. Newton’s method is fast, but often fails to converge.

By using a damping factor to interpolate between the two, a hybrid technique is obtained. The following equation presents how Newton's method is employed.

$$\mathbf{W}_{\min} = \mathbf{W}_0 - \mathbf{H}^{-1}\mathbf{g} \tag{6.39}$$

H^{-1} represents the inverse matrix of Hess, g represents the gradients and W the weights. The Hessian matrix contains the second derivatives according to Equation (6.40), where *e* represents neural network output.

$$\mathbf{H}(e) = \begin{pmatrix} \frac{\partial^2 e}{\partial w_1^2} & \frac{\partial^2 e}{\partial w_1 w_2} & \cdots & \frac{\partial^2 e}{\partial w_1 w_p} \\ \frac{\partial^2 e}{\partial w_2 w_1} & \frac{\partial^2 e}{\partial w_2^2} & \cdots & \frac{\partial^2 e}{\partial w_2 w_p} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 e}{\partial w_p w_1} & \frac{\partial^2 e}{\partial w_p w_2} & \cdots & \frac{\partial^2 e}{\partial w_p^2} \end{pmatrix} \tag{6.40}$$

The second derivative is the derivative of the first derivative. Let us recall that the derivative of the function is the slope at any point. The slope indicates that the curve is approaching for a local minimum. The second derivative can also be seen as a slope, and it points in a direction to minimise the first derivative. The goal of Newton’s method as well as Levenberg-Marquardt algorithm is to reduce all of the gradients to 0.

Newton’s method will converge the weights of a neural network to a local minimum, a local maximum or a saddle point (Fig. 6.10). Such convergence is attained by minimising all gradients (first derivatives) to 0. These are 0 at local minima, maxima or saddle points.

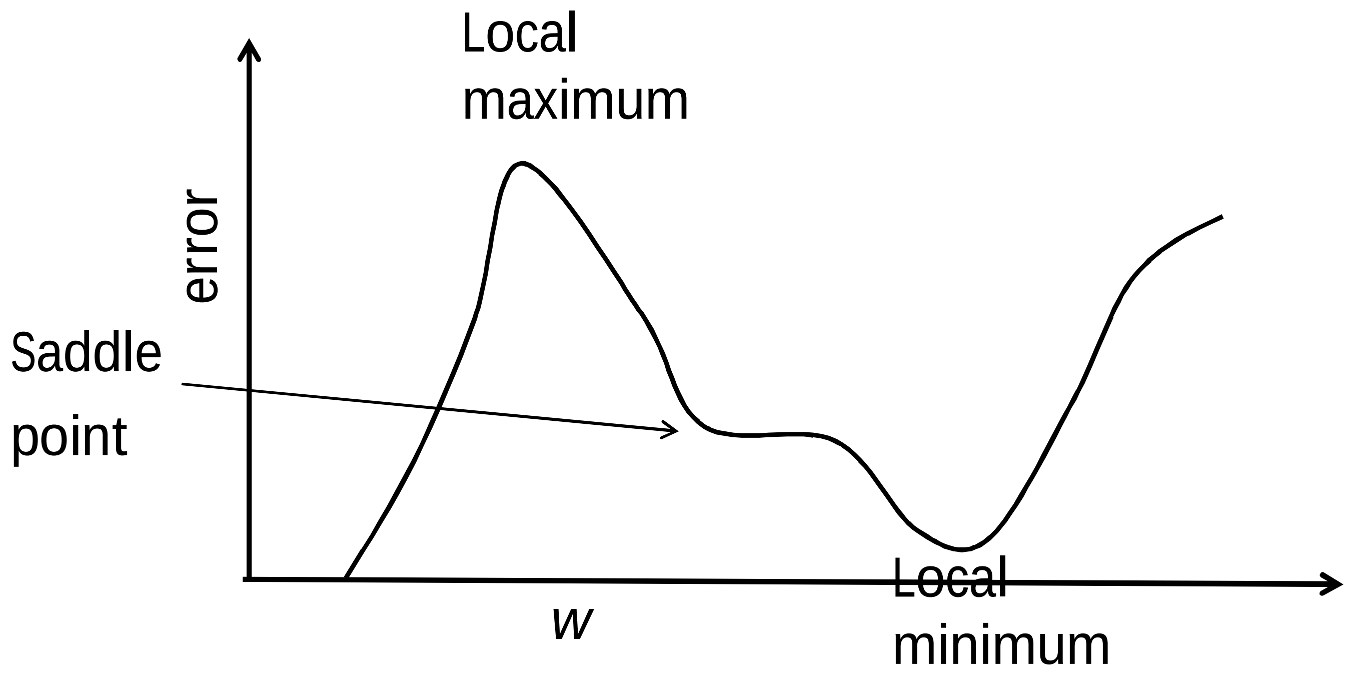


Fig. 6.10 Local minimum, maximum and saddle point.

The algorithm implementation has to leave out local maxima and saddle points. The Hessian matrix is typically approximated.

Levenberg-Marquardt algorithm enhances Newton’s method to the following form

$$\mathbf{W}_{\min} = \mathbf{W}_0 - \left(\mathbf{H} + \lambda \mathbf{I}\right)^{-1} \mathbf{g} \tag{6.41}$$

where an identity matrix I is multiplied with a damping coefficient λ. On the diagonal, the elements of I are equal to 1, and all others are equal to 0.

A compact survey on these two and other algorithms originating from backpropagation is presented in the current article[5].

[5] H. Joutsijoki et al.: Evaluating the performance of artificial neural networks for the classification of freshwater benthic macroinvertebrates, Ecological Informatics, 20, 1-7, 2014.

Example

In the following, some results are shown from the article[6] where both feedforward neural networks and other machine learning methods were used to classify biometric data in order to verify subjects on the basis of their saccade eye movements.

As to feedforward neural networks, the most promising results were gained with Levenberg-Marquardt algorithm, some results of which are shown. In addition, some results given by logistic discriminant analysis, support vector machines and radial basis function networks will be shown.

[6] Y. Zhang and M. Juhola: On biometric verification of a user by means of eye movement data mining, Proceedings of the Second International Conference on Advances in Information Mining and Management (IMMM2012), p. 85-90, Venice, Italy, 2012.

In the verification task, in fact, classification there are two classes to be detected as those of authenticated (e.g. computer) users and those of impostors (such as if attempting to log in illegally). The task is to recognise a subject to be either an authenticated or impostor on the basis of measured eye movement data.

Five recordings were measured form each subject. There were 132 subjects tested so that every subject was one by one in the role of an authenticated user and then in that of an impostor. Saccades were recognised in the saccade signals and feature values computed from them. In the preprocessing, all the feature data were normalised into range [0,1].

The verification results given as classification accuracies are shown in Table 6.1.

Table 6.1 Average accuracies of binary classification in percent.

	Neural network of 8 hidden nodes	Neural network of 10 hidden nodes	Logistic discriminant analysis	Support vector machine with Gaussian kernel	Radial basis function neural network
Authenticated users	80.0	79.6	86.6	92.1	88.5
Impostors	80.2	82.7	77.4	84.8	88.9

6.6 Training and testing neural networks

When a large data set composed of known training cases corresponding to known classes or output values exists, perhaps the simplest manner to divide the whole data set into the training and testing data set is to apply the *hold out* method, in which the data set is divided into two halves of equal size. The partition has always to be made randomly subject to which part a case is selected. Then we may assume that both sets follow the class distribution of the original population of the data. The former set is used for building the model that is tested with the latter.

What is a large enough data set for hold out method essentially depends on the number of weights of the network to be learnt.

The rule of thumb is to use at least ten times the number of the weights to a training set. This comes from an error measure derived statistically (not to be considered here) with some certain assumptions. Thus, sometimes smaller numbers may be enough depending on data and a task to be computed.

Generally, the important principle is that enough cases are selected for a training set to guarantee that a neural network can be trained to model the characteristics of a data source. Therefore, if the data source is scarce, a training set is set to include, e.g, 60%, 80% or even 90% of all data, and the rest is left for its test set.

Crossvalidation method is usually applied to machine learning tasks, particulary in the circumstances of relatively restricted numbers of data cases at disposal.

In *k*-fold crossvalidation, the data set is first divided into *k* subsets of approximately equal size. Always the partition is executed randomly. Then each subset of 10% size is used as test set and the other *k*-1 subsets as a training set. One by one, this is performed for each of 10 subsets. Typically 10-fold crossvalidation is applied according to Fig. 6.11.

If the data set is small, leave-one-out is good where the training set includes all but one case to be used as the only test case of a model. Thus, *n* models and test cases are used when *n* is the number of the whole data set.

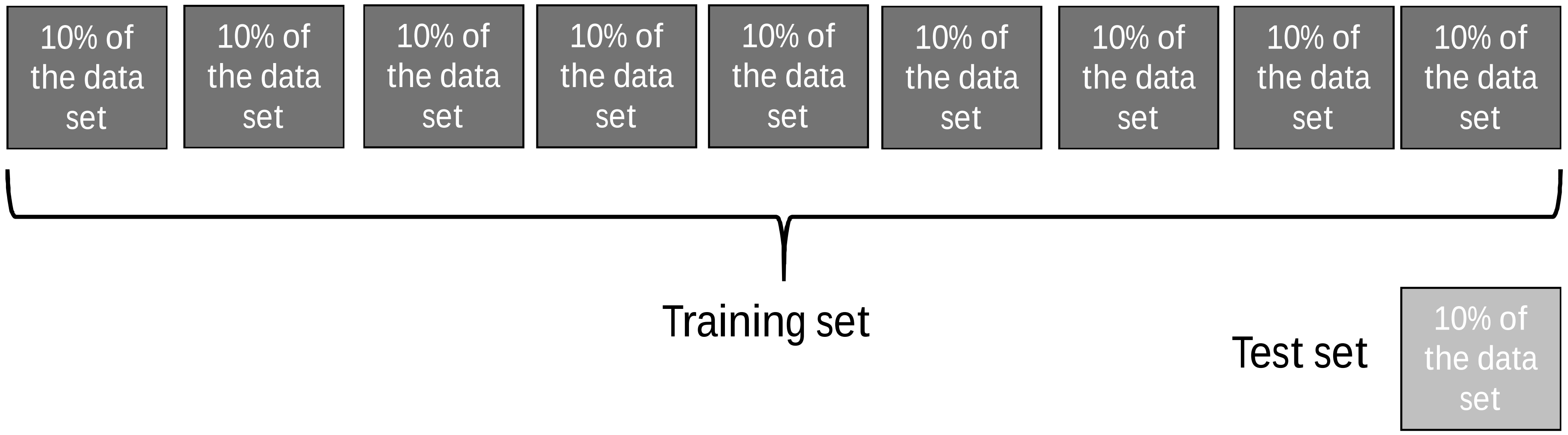


Fig. 6.11 One by one each 10% partition forms a single test set and the rest of the subsets the corresponding training set.

When we apply any machine learning methods, one or just a few test runs are rather a random ”pinpoint experiment” than an actual test setup. Because of ever-existent randomness subject to the selection of training and test sets and, in general, all data available, we have always to run numerous test series and ultimately to compute their mean and standard deviation (and possibly some additional central values) in order to obtain reliable information about the capability of computational models prepared.

Furthermore, neural networks as some other machine learning methods contain random initial values for their weights or some control parameters. Thus, they need several different beginnings for the otherwise same models. See Fig. 6.12 and Fig. 6.13.

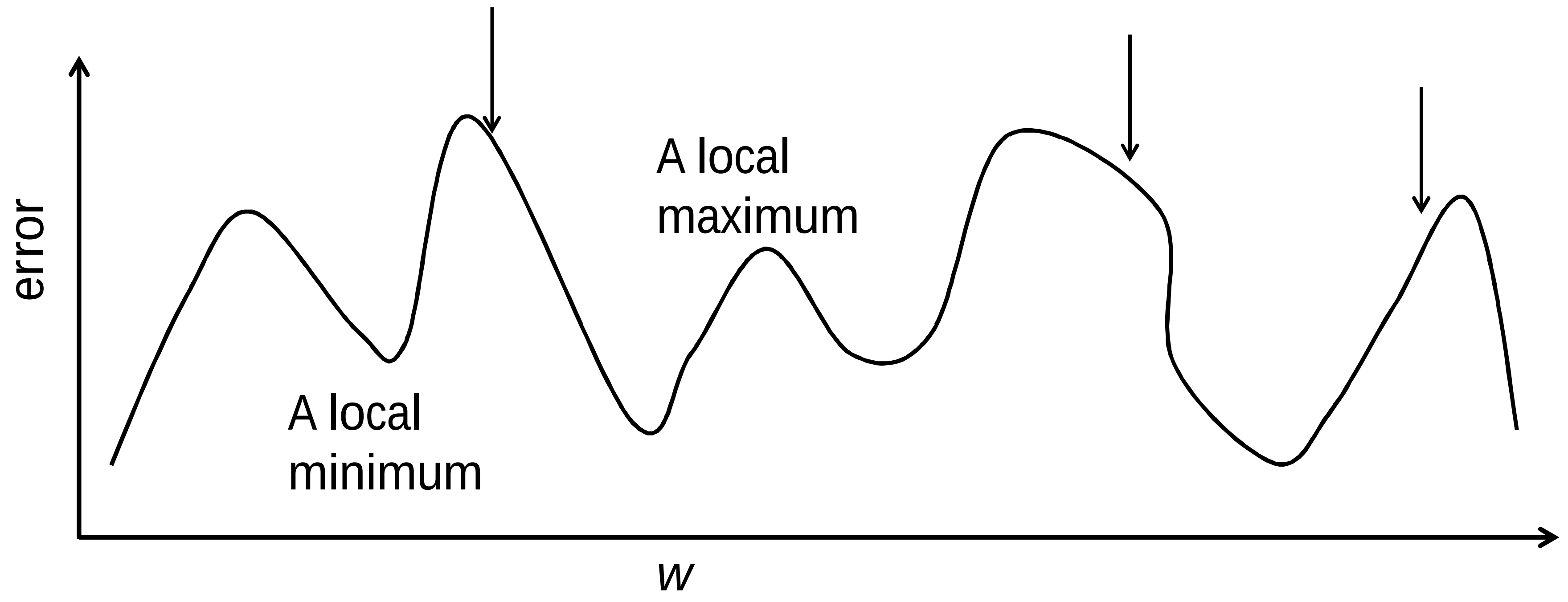


Fig. 6.12 Black, red and green vertical arrows represent different random initial weights appearing in random locations of the weight space. Accordingly, sometimes the training process may result in different local minima representing somewhat different combinations of weight values, i.e., models constructed.

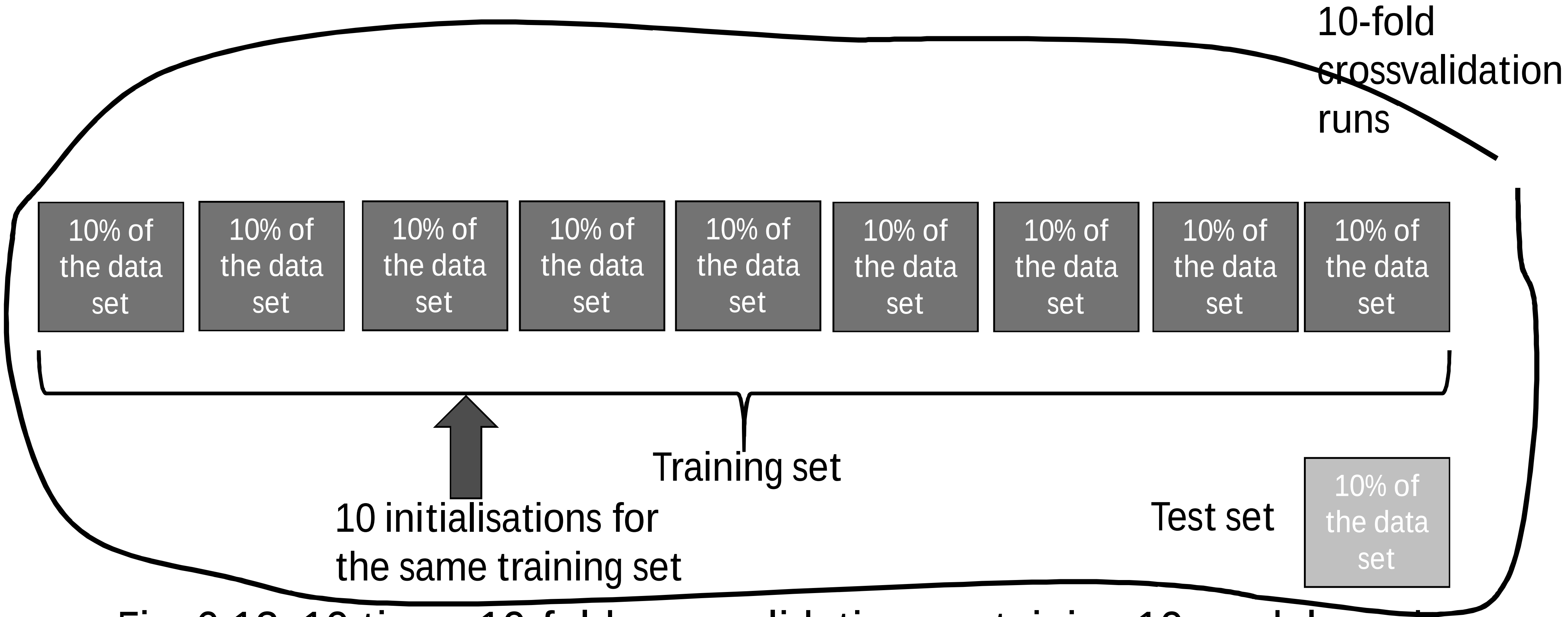


Fig. 6.13 10 times 10-fold crossvalidation containing 10 models each tested 10 times produces 100 test runs altogether.

When feedforward neural networks are constructed, often another separate subset is still separated. Then there are three disjoint subsets formed: training set (8 times 10%), test set (10%) and validation set (10%). Of course, their size may vary, but the total is naturally 100%. A validation set is fully kept outside the training and testing of a model. It is used only to evaluate an error measure, usually mean squared error (MSE) or root mean squared error (RMSE).

In theory, training and validation mainly follow a kind of double curves outlined in Fig. 6.14. Then an optimal stopping is just before the validation curve will begin to increase. See also Fig. 6.15.

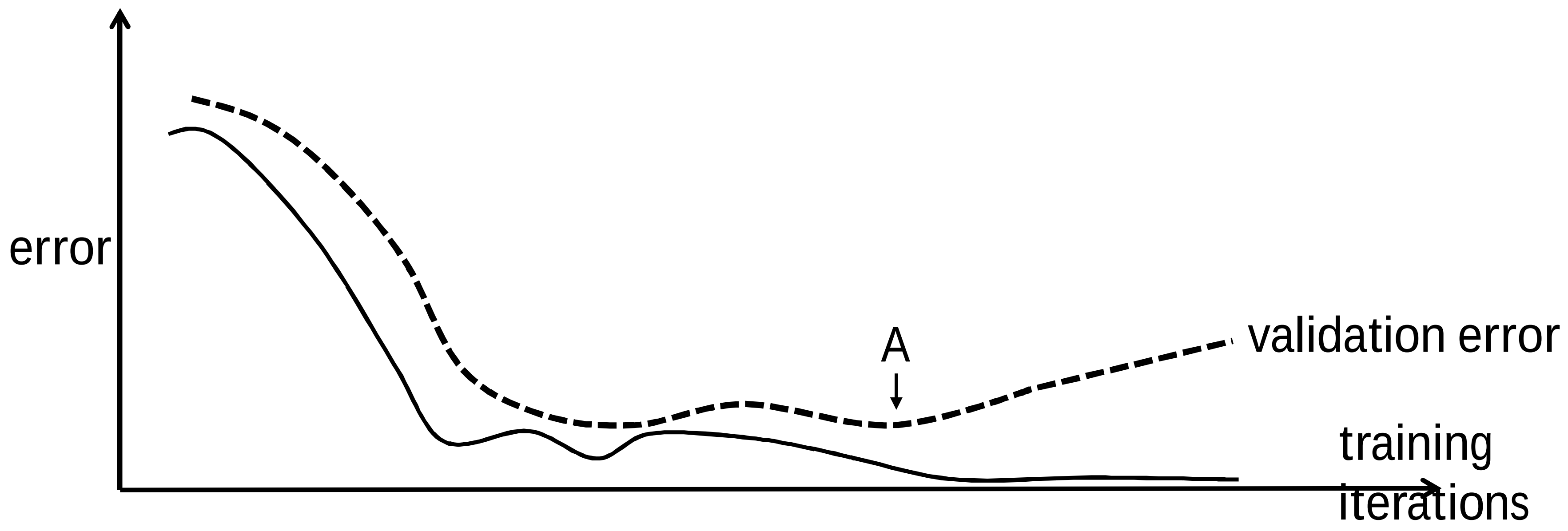


Fig. 6.14 When a single model is trained along with successive iterations with the same training set, first its error (black) curve converges fast and then gets smooth oscillating slightly. The validation set separate from the training and test sets, instead, may begin to increase after some point A (red curve). This is a consequence of overtraining.

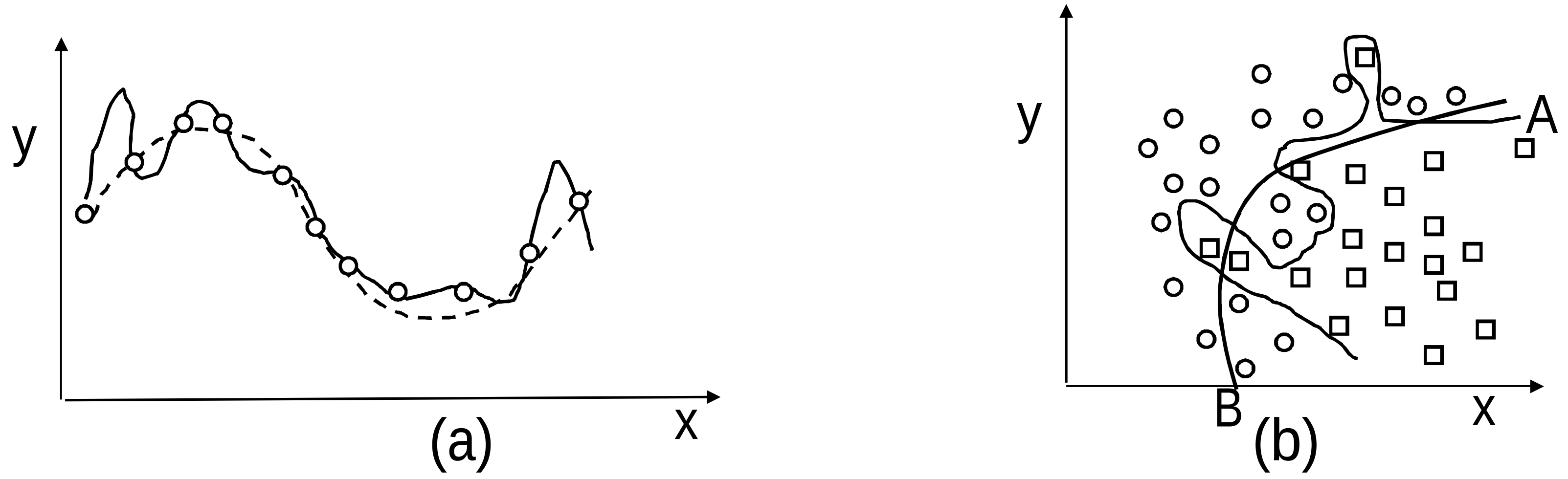


Fig. 6.15 (a) The dots represent overtraining of polynomial approximation when the training data curve has been learnt too precisely. A better choice is a less accurate learning according to the dashed model. (b) Overtraining in the form of boundary A in the classification of two classes after too detailed learning. On the other hand, B is too rough.

Overtraining, overlearning or overfitting is a phenomenon that is not easy to control very strictly. However, it is often useful to stop training early enough when the error (black curve) in Fig. 6.14 has dropped and begun to stabilise more or less.

Fig. 6.15(a) describes too detailed training when too many values of a polynomial function are taken tightly as such from the curve resulting in overlearning. A better approximation is a smoother curve including the main properties of the curve but not getting stuck to ”noise”. Fig. 6.15(b) describes too exact and too poor boundaries between the two classes. Obviously, a mapping between these two approaches subject to their complexity would be better.

Too exact learning in Fig. 6.15 is not good, because if we take a new sample from the data source, when it again follows the distribution of the data source, but is not entirely similar to the previous sample. In other words, the cases of the new sample are not located precisely in the same places compared to the previous sample in the variable space. Therefore, an overtrained model would not cope well with the new sample, but probably a more flexible model is better. This issue concerns *generalization*. A neural network constructed is supposed to be also used for novel data. Moreover, from time to time it is best to update the model with a partially updated training set.