# Introduction to Machine Learning

## Second Edition

Ethem Alpaydın

The MIT Press
Cambridge, Massachusetts
London, England

For information about special quantity discounts, please email
special_sales@mitpress.mit.edu.

Typeset in 10/13 Lucida Bright by the author using $\LaTeX\,2_\varepsilon$.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Information

Alpaydin, Ethem.
Introduction to machine learning / Ethem Alpaydin. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-01243-0 (hardcover : alk. paper)
1. Machine learning. I. Title
Q325.5.A46 2010
006.3’1—dc22 2009013169
CIP

10 9 8 7 6 5 4 3 2 1

# *Brief Contents*

# *Preface*

Machine learning is programming computers to optimize a performance criterion using example data or past experience. We need learning in cases where we cannot directly write a computer program to solve a given problem, but need example data or experience. One case where learning is necessary is when human expertise does not exist, or when humans are unable to explain their expertise. Consider the recognition of spoken speech—that is, converting the acoustic speech signal to an ASCII text; we can do this task seemingly without any difficulty, but we are unable to explain how we do it. Different people utter the same word differently due to differences in age, gender, or accent. In machine learning, the approach is to collect a large collection of sample utterances from different people and learn to map these to words.

Another case is when the problem to be solved changes in time, or depends on the particular environment. We would like to have general-purpose systems that can adapt to their circumstances, rather than explicitly writing a different program for each special circumstance. Consider routing packets over a computer network. The path maximizing the quality of service from a source to destination changes continuously as the network traffic changes. A learning routing program is able to adapt to the best path by monitoring the network traffic. Another example is an intelligent user interface that can adapt to the biometrics of its user—namely, his or her accent, handwriting, working habits, and so forth.

Already, there are many successful applications of machine learning in various domains: There are commercially available systems for recognizing speech and handwriting. Retail companies analyze their past sales data to learn their customers' behavior to improve customer relationship management. Financial institutions analyze past transactions to predict customers' credit risks. Robots learn to optimize their behavior to complete a task using minimum resources. In bioinformatics, the huge amount of data can only be analyzed and knowledge extracted using computers. These are only some of the applications that we—that is, you and I—will discuss throughout this book. We can only imagine what future applications can be realized using machine learning: Cars that can drive themselves under different road and weather conditions, phones that can translate in real time to and from a foreign language, autonomous robots that can navigate in a new environment, for example, on the surface of another planet. Machine learning is certainly an exciting field to be working in!

The book discusses many methods that have their bases in different fields: statistics, pattern recognition, neural networks, artificial intelligence, signal processing, control, and data mining. In the past, research in these different communities followed different paths with different emphases. In this book, the aim is to incorporate them together to give a unified treatment of the problems and the proposed solutions to them.

This is an introductory textbook, intended for senior undergraduate and graduate-level courses on machine learning, as well as engineers working in the industry who are interested in the application of these methods. The prerequisites are courses on computer programming, probability, calculus, and linear algebra. The aim is to have all learning algorithms sufficiently explained so it will be a small step from the equations given in the book to a computer program. For some cases, pseudocode of algorithms are also included to make this task easier.

The book can be used for a one-semester course by sampling from the chapters, or it can be used for a two-semester course, possibly by discussing extra research papers; in such a case, I hope that the references at the end of each chapter are useful.

The Web page is http://www.cmpe.boun.edu.tr/~ethem/i2ml/ where I will post information related to the book that becomes available after the book goes to press, for example, errata. I welcome your feedback via email to alpaydin@boun.edu.tr.

I very much enjoyed writing this book; I hope you will enjoy reading it.

# *Notes for the Second Edition*

Machine learning has seen important developments since the first edition appeared in 2004. First, application areas have grown rapidly. Internet-related technologies, such as search engines, recommendation systems, spam fiters, and intrusion detection systems are now routinely using machine learning. In the field of bioinformatics and computational biology, methods that learn from data are being used more and more widely. In natural language processing applications—for example, machine translation—we are seeing a faster and faster move from programmed expert systems to methods that learn automatically from very large corpus of example text. In robotics, medical diagnosis, speech and image recognition, biometrics, finance, sometimes under the name pattern recognition, sometimes disguised as data mining, or under one of its many cloaks, we see more and more applications of the machine learning methods we discuss in this textbook.

Second, there have been supporting advances in theory. Especially, the idea of kernel functions and the kernel machines that use them allow a better representation of the problem and the associated convex optimization framework is one step further than multilayer perceptrons with sigmoid hidden units trained using gradient-descent. Bayesian methods through appropriately chosen prior distributions add expert knowledge to what the data tells us. Graphical models allow a representation as a network of interrelated nodes and efficient inference algorithms allow querying the network. It has thus become necessary that these three topics—namely, kernel methods, Bayesian estimation, and graphical models—which were sections in the first edition, be treated in more length, as three new chapters.

Another revelation hugely significant for the field has been in the realization that machine learning experiments need to be designed better. We have gone a long way from using a single test set to methods for cross-validation to paired $t$ tests. That is why, in this second edition, I have rewritten the chapter on statistical tests as one that includes the design and analysis of machine learning experiments. The point is that testing should not be a separate step done after all runs are completed (despite the fact that this new chapter is at the very end of the book); the whole process of experimentation should be designed beforehand, relevant factors defined, proper experimentation procedure decided upon, and then, and only then, the runs should be done and the results analyzed.

It has long been believed, especially by older members of the scientific community, that for machines to be as intelligent as us, that is, for artificial intelligence to be a reality, our current knowledge in general, or computer science in particular, is not sufficient. People largely are of the opinion that we need a new technology, a new type of material, a new type of computational mechanism or a new programming methodology, and that, until then, we can only "simulate" some aspects of human intelligence and only in a limited way but can never fully attain it.

I believe that we will soon prove them wrong. First we saw this in chess, and now we are seeing it in a whole variety of domains. Given enough memory and computation power, we can realize tasks with relatively simple algorithms; the trick here is learning, either learning from example data or learning from trial and error using reinforcement learning. It seems as if using supervised and mostly unsupervised learning algorithms—for example, machine translation—will soon be possible. The same holds for many other domains, for example, unmanned navigation in robotics using reinforcement learning. I believe that this will continue for many domains in artificial intelligence, and the key is learning. We do not need to come up with new algorithms if machines can learn themselves, assuming that we can provide them with enough data (not necessarily supervised) and computing power.

I would like to thank all the instructors and students of the first edition, from all over the world, including the reprint in India and the German translation. I am grateful to those who sent me words of appreciation and errata or who provided feedback in any other way. Please keep those emails coming. My email address is `alpaydin@boun.edu.tr`.

The second edition also provides more support on the Web. The book's

Web site is `http://www.cmpe.boun.edu.tr/~ethem/i2ml`.

I would like to thank my past and present thesis students, Mehmet Gönen, Esma Kılıç, Murat Semerci, M. Aydın Ulaş, and Olcay Taner Yıldız, and also those who have taken CmpE 544, CmpE 545, CmpE 591, and CmpE 58E during these past few years. The best way to test your knowledge of a topic is by teaching it.

It has been a pleasure working with the MIT Press again on this second edition, and I thank Bob Prior, Ada Brunstein, Erin K. Shoudy, Kathleen Caruso, and Marcy Ross for all their help and support.

# *Notations*

| | |
|---|---|
| $x$ | Scalar value |
| $\boldsymbol{x}$ | Vector |
| $\mathbf{X}$ | Matrix |
| $\boldsymbol{x}^T$ | Transpose |
| $\mathbf{X}^{-1}$ | Inverse |
| | |
| $X$ | Random variable |
| $P(X)$ | Probability mass function when $X$ is discrete |
| $p(X)$ | Probability density function when $X$ is continuous |
| $P(X\|Y)$ | Conditional probability of $X$ given $Y$ |
| $E[X]$ | Expected value of the random variable $X$ |
| $\mathrm{Var}(X)$ | Variance of $X$ |
| $\mathrm{Cov}(X, Y)$ | Covariance of $X$ and $Y$ |
| $\mathrm{Corr}(X, Y)$ | Correlation of $X$ and $Y$ |
| | |
| $\mu$ | Mean |
| $\sigma^2$ | Variance |
| $\boldsymbol{\Sigma}$ | Covariance matrix |
| $m$ | Estimator to the mean |
| $s^2$ | Estimator to the variance |
| $\mathbf{S}$ | Estimator to the covariance matrix |

| | |
|---|---|
| $\mathcal{N}(\mu, \sigma^2)$ | Univariate normal distribution with mean $\mu$ and variance $\sigma^2$ |
| $\mathcal{Z}$ | Unit normal distribution: $\mathcal{N}(0, 1)$ |
| $\mathcal{N}_d(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ | $d$-variate normal distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ |
| | |
| $x$ | Input |
| $d$ | Number of inputs (input dimensionality) |
| $y$ | Output |
| $r$ | Required output |
| $K$ | Number of outputs (classes) |
| $N$ | Number of training instances |
| $z$ | Hidden value, intrinsic dimension, latent factor |
| $k$ | Number of hidden dimensions, latent factors |
| $C_i$ | Class $i$ |
| $\mathcal{X}$ | Training sample |
| $\{x^t\}_{t=1}^N$ | Set of $x$ with index $t$ ranging from 1 to $N$ |
| $\{x^t, r^t\}_t$ | Set of ordered pairs of input and desired output with index $t$ |
| | |
| $g(x\|\theta)$ | Function of $x$ defined up to a set of parameters $\theta$ |
| $\arg\max_\theta g(x\|\theta)$ | The argument $\theta$ for which $g$ has its maximum value |
| $\arg\min_\theta g(x\|\theta)$ | The argument $\theta$ for which $g$ has its minimum value |
| $E(\theta\|\mathcal{X})$ | Error function with parameters $\theta$ on the sample $\mathcal{X}$ |
| $l(\theta\|\mathcal{X})$ | Likelihood of parameters $\theta$ on the sample $\mathcal{X}$ |
| $\mathcal{L}(\theta\|\mathcal{X})$ | Log likelihood of parameters $\theta$ on the sample $\mathcal{X}$ |
| | |
| $1(c)$ | 1 if $c$ is true, 0 otherwise |
| $\#\{c\}$ | Number of elements for which $c$ is true |
| $\delta_{ij}$ | Kronecker delta: 1 if $i = j$, 0 otherwise |

# 11 *Multilayer Perceptrons*

*The multilayer perceptron is an artificial neural network structure and is a nonparametric estimator that can be used for classification and regression. We discuss the backpropagation algorithm to train a multilayer perceptron for a variety of applications.*

## 11.1 Introduction

ARTIFICIAL NEURAL network models, one of which is the *perceptron* we discuss in this chapter, take their inspiration from the brain. There are cognitive scientists and neuroscientists whose aim is to understand the functioning of the brain (Posner 1989; Thagard 2005), and toward this aim, build models of the natural neural networks in the brain and make simulation studies.

ARTIFICIAL NEURAL NETWORKS

However, in engineering, our aim is not to understand the brain per se, but to build useful machines. We are interested in *artificial neural networks* because we believe that they may help us build better computer systems. The brain is an information processing device that has some incredible abilities and surpasses current engineering products in many domains—for example, vision, speech recognition, and learning, to name three. These applications have evident economic utility if implemented on machines. If we can understand how the brain performs these functions, we can define solutions to these tasks as formal algorithms and implement them on computers.

NEURONS

The human brain is quite different from a computer. Whereas a computer generally has one processor, the brain is composed of a very large ($10^{11}$) number of processing units, namely, *neurons*, operating in parallel. Though the details are not known, the processing units are believed to be much simpler and slower than a processor in a computer. What also makes the brain different, and is believed to provide its computational power, is the large connectivity. Neurons in the brain have connections, called *synapses*, to around $10^4$ other neurons, all operating in parallel. In a computer, the processor is active and the memory is separate and passive, but it is believed that in the brain, both the processing and memory are distributed together over the network; processing is done by the neurons, and the memory is in the synapses between the neurons.

SYNAPSES

### 11.1.1 Understanding the Brain

According to Marr (1982), understanding an information processing system has three levels, called the *levels of analysis*:

LEVELS OF ANALYSIS

1. *Computational theory* corresponds to the goal of computation and an abstract definition of the task.

2. *Representation and algorithm* is about how the input and the output are represented and about the specification of the algorithm for the transformation from the input to the output.

3. *Hardware implementation* is the actual physical realization of the system.

One example is sorting: The computational theory is to order a given set of elements. The representation may use integers, and the algorithm may be Quicksort. After compilation, the executable code for a particular processor sorting integers represented in binary is one hardware implementation.

The idea is that for the same computational theory, there may be multiple representations and algorithms manipulating symbols in that representation. Similarly, for any given representation and algorithm, there may be multiple hardware implementations. We can use one of various sorting algorithms, and even the same algorithm can be compiled on computers with different processors and lead to different hardware implementations.

To take another example, ‘6’, ‘VI’, and ‘110’ are three different representations of the number six. There is a different algorithm for addition depending on the representation used. Digital computers use binary representation and have circuitry to add in this representation, which is one particular hardware implementation. Numbers are represented differently, and addition corresponds to a different set of instructions on an abacus, which is another hardware implementation. When we add two numbers in our head, we use another representation and an algorithm suitable to that representation, which is implemented by the neurons. But all these different hardware implementations—for example, us, abacus, digital computer—implement the same computational theory, addition.

The classic example is the difference between natural and artificial flying machines. A sparrow flaps its wings; a commercial airplane does not flap its wings but uses jet engines. The sparrow and the airplane are two hardware implementations built for different purposes, satisfying different constraints. But they both implement the same theory, which is aerodynamics.

The brain is one hardware implementation for learning or pattern recognition. If from this particular implementation, we can do reverse engineering and extract the representation and the algorithm used, and if from that in turn, we can get the computational theory, we can then use another representation and algorithm, and in turn a hardware implementation more suited to the means and constraints we have. One hopes our implementation will be cheaper, faster, and more accurate.

Just as the initial attempts to build flying machines looked very much like birds until we discovered aerodynamics, it is also expected that the first attempts to build structures possessing brain's abilities will look like the brain with networks of large numbers of processing units, until we discover the computational theory of intelligence. So it can be said that in understanding the brain, when we are working on artificial neural networks, we are at the representation and algorithm level.

Just as the feathers are irrelevant to flying, in time we may discover that neurons and synapses are irrelevant to intelligence. But until that time there is one other reason why we are interested in understanding the functioning of the brain, and that is related to parallel processing.

### 11.1.2 Neural Networks as a Paradigm for Parallel Processing

Since the 1980s, computer systems with thousands of processors have been commercially available. The software for such parallel architectures, however, has not advanced as quickly as hardware. The reason for this is that almost all our theory of computation up to that point was based on serial, one-processor machines. We are not able to use the parallel machines we have efficiently because we cannot program them efficiently.

PARALLEL PROCESSING

There are mainly two paradigms for *parallel processing*: In Single Instruction Multiple Data (SIMD) machines, all processors execute the same instruction but on different pieces of data. In Multiple Instruction Multiple Data (MIMD) machines, different processors may execute different instructions on different data. SIMD machines are easier to program because there is only one program to write. However, problems rarely have such a regular structure that they can be parallelized over a SIMD machine. MIMD machines are more general, but it is not an easy task to write separate programs for all the individual processors; additional problems are related to synchronization, data transfer between processors, and so forth. SIMD machines are also easier to build, and machines with more processors can be constructed if they are SIMD. In MIMD machines, processors are more complex, and a more complex communication network should be constructed for the processors to exchange data arbitrarily.

Assume now that we can have machines where processors are a little bit more complex than SIMD processors but not as complex as MIMD processors. Assume we have simple processors with a small amount of local memory where some parameters can be stored. Each processor implements a fixed function and executes the same instructions as SIMD processors; but by loading different values into the local memory, they can be doing different things and the whole operation can be distributed over such processors. We will then have what we can call Neural Instruction Multiple Data (NIMD) machines, where each processor corresponds to a neuron, local parameters correspond to its synaptic weights, and the whole structure is a neural network. If the function implemented in each processor is simple and if the local memory is small, then many such processors can be fit on a single chip.

The problem now is to distribute a task over a network of such processors and to determine the local parameter values. This is where learning comes into play: We do not need to program such machines and determine the parameter values ourselves if such machines can learn from examples.

Thus, artificial neural networks are a way to make use of the parallel hardware we can build with current technology and—thanks to learning—they need not be programmed. Therefore, we also save ourselves the effort of programming them.

In this chapter, we discuss such structures and how they are trained.

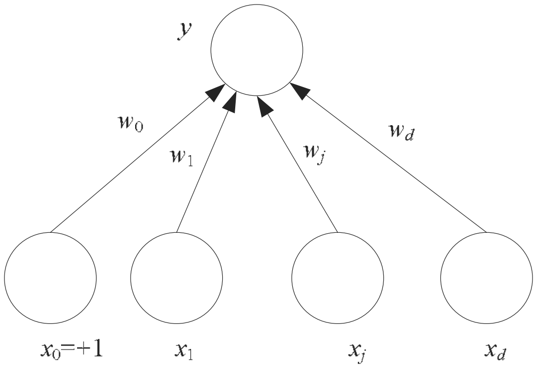

**Figure 11.1** Simple perceptron. $x_j, j = 1, \ldots, d$ are the input units. $x_0$ is the bias unit that always has the value 1. $y$ is the output unit. $w_j$ is the weight of the directed connection from input $x_j$ to the output.

Keep in mind that the operation of an artificial neural network is a mathematical function that can be implemented on a serial computer—as it generally is—and training the network is not much different from statistical techniques that we have discussed in the previous chapters. Thinking of this operation as being carried out on a network of simple processing units is meaningful only if we have the parallel hardware, and only if the network is so large that it cannot be simulated fast enough on a serial computer.

## 11.2 The Perceptron

PERCEPTRON The *perceptron* is the basic processing element. It has inputs that may come from the environment or may be the outputs of other perceptrons. CONNECTION WEIGHT SYNAPTIC WEIGHT Associated with each input, $x_j \in \Re, j = 1, \ldots, d$, is a *connection weight*, or *synaptic weight* $w_j \in \Re$, and the output, $y$, in the simplest case is a weighted sum of the inputs (see figure 11.1):

$$y = \sum_{j=1}^{d} w_j x_j + w_0 \tag{11.1}$$

BIAS UNIT $w_0$ is the intercept value to make the model more general; it is generally modeled as the weight coming from an extra *bias unit*, $x_0$, which is always

+1. We can write the output of the perceptron as a dot product

$$y = \boldsymbol{w}^T \boldsymbol{x} \tag{11.2}$$

where $\boldsymbol{w} = [w_0, w_1, \ldots, w_d]^T$ and $\boldsymbol{x} = [1, x_1, \ldots, x_d]^T$ are *augmented* vectors to include also the bias weight and input.

During testing, with given weights, $\boldsymbol{w}$, for input $\boldsymbol{x}$, we compute the output $y$. To implement a given task, we need to *learn* the weights $\boldsymbol{w}$, the parameters of the system, such that correct outputs are generated given the inputs.

When $d = 1$ and $x$ is fed from the environment through an input unit, we have

$$y = wx + w_0$$

which is the equation of a line with $w$ as the slope and $w_0$ as the intercept. Thus this perceptron with one input and one output can be used to implement a linear fit. With more than one input, the line becomes a (hyper)plane, and the perceptron with more than one input can be used to implement multivariate linear fit. Given a sample, the parameters $w_j$ can be found by regression (see section 5.8).

The perceptron as defined in equation 11.1 defines a hyperplane and as such can be used to divide the input space into two: the half-space where it is positive and the half-space where it is negative (see chapter 10). By using it to implement a linear discriminant function, the perceptron can separate two classes by checking the sign of the output. If we define $s(\cdot)$ as the *threshold function*

THRESHOLD FUNCTION

$$s(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.3}$$

then we can

$$\text{choose} \begin{cases} C_1 & \text{if } s(\boldsymbol{w}^T\boldsymbol{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

Remember that using a linear discriminant assumes that classes are linearly separable. That is to say, it is assumed that a hyperplane $\boldsymbol{w}^T\boldsymbol{x} = 0$ can be found that separates $\boldsymbol{x}^t \in C_1$ and $\boldsymbol{x}^t \in C_2$. If at a later stage we need the posterior probability—for example, to calculate risk—we need to use the sigmoid function at the output as

$$\begin{aligned} o &= \boldsymbol{w}^T\boldsymbol{x} \\ y &= \text{sigmoid}(o) = \frac{1}{1 + \exp[-\boldsymbol{w}^T\boldsymbol{x}]} \end{aligned} \tag{11.4}$$

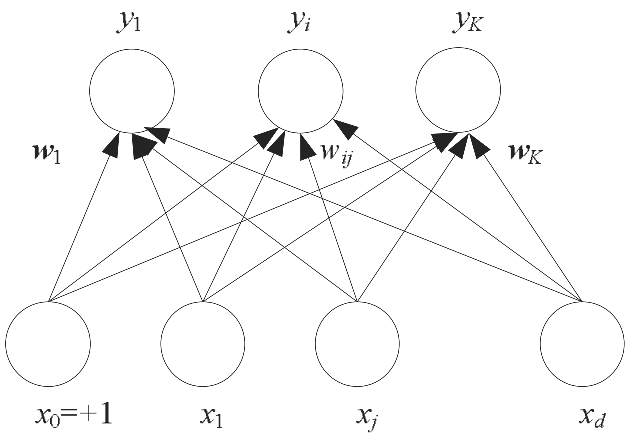

**Figure 11.2** *K* parallel perceptrons. $x_j, j = 0, \dots, d$ are the inputs and $y_i, i = 1, \dots, K$ are the outputs. $w_{ij}$ is the weight of the connection from input $x_j$ to output $y_i$. Each output is a weighted sum of the inputs. When used for *K*-class classification problem, there is a postprocessing to choose the maximum, or softmax if we need the posterior probabilities.

When there are $K > 2$ outputs, there are $K$ perceptrons, each of which has a weight vector $\boldsymbol{w}_i$ (see figure 11.2)

$$y_i = \sum_{j=1}^{d} w_{ij} x_j + w_{i0} = \boldsymbol{w}_i^T \boldsymbol{x}$$

$$\boldsymbol{y} = \mathbf{W}\boldsymbol{x} \tag{11.5}$$

where $w_{ij}$ is the weight from input $x_j$ to output $y_i$. $\mathbf{W}$ is the $K \times (d+1)$ weight matrix of $w_{ij}$ whose rows are the weight vectors of the $K$ perceptrons. When used for classification, during testing, we

$$\text{choose } C_i \text{ if } y_i = \max_k y_k$$

In the case of a neural network, the value of each perceptron is a *local* function of its inputs and its synaptic weights. However, in classification, if we need the posterior probabilities (instead of just the code of the winner class) and use the softmax, we also need the values of the other outputs. So, to implement this as a neural network, we can see this as a two-stage process, where the first stage calculates the weighted sums, and the second stage calculates the softmax values; but we still denote this as a single layer of output units:

$$\begin{aligned} o_i &= \boldsymbol{w}_i^T \boldsymbol{x} \\ y_i &= \frac{\exp o_i}{\sum_k \exp o_k} \end{aligned} \tag{11.6}$$

Remember that by defining auxiliary inputs, the linear model can also be used for polynomial approximation; for example, define $x_3 = x_1^2, x_4 = x_2^2, x_5 = x_1 x_2$ (section 10.2). The same can also be used with perceptrons (Durbin and Rumelhart 1989). In section 11.5, we see multilayer perceptrons where such nonlinear functions are learned from data in a "hidden" layer instead of being assumed a priori.

Any of the methods discussed in chapter 10 on linear discrimination can be used to calculate $\boldsymbol{w}_i, i = 1, \ldots, K$ offline and then plugged into the network. These include parametric approach with a common covariance matrix, logistic discrimination, discrimination by regression, and support vector machines. In some cases, we do not have the whole sample at hand when training starts, and we need to iteratively update parameters as new examples arrive; we discuss this case of *online* learning in section 11.3.

Equation 11.5 defines a linear transformation from a $d$-dimensional space to a $K$-dimensional space and can also be used for dimensionality reduction if $K < d$. One can use any of the methods of chapter 6 to calculate $\mathbf{W}$ offline and then use the perceptrons to implement the transformation, for example, PCA. In such a case, we have a two-layer network where the first layer of perceptrons implements the linear transformation and the second layer implements the linear regression or classification in the new space. We note that because both are linear transformations, they can be combined and written down as a single layer. We will see the more interesting case where the first layer implements *nonlinear* dimensionality reduction in section 11.5.

## 11.3 Training a Perceptron

The perceptron defines a hyperplane, and the neural network perceptron is just a way of *implementing* the hyperplane. Given a data sample, the weight values can be calculated *offline* and then when they are plugged in, the perceptron can be used to calculate the output values.

In training neural networks, we generally use online learning where we are not given the whole sample, but we are given instances one by one and would like the network to update its parameters after each instance, adapting itself slowly in time. Such an approach is interesting for a number of reasons:

1. It saves us the cost of storing the training sample in an external memory and storing the intermediate results during optimization. An approach like support vector machines (chapter 13) may be quite costly with large samples, and in some applications, we may prefer a simpler approach where we do not need to store the whole sample and solve a complex optimization problem on it.

2. The problem may be changing in time, which means that the sample distribution is not fixed, and a training set cannot be chosen a priori. For example, we may be implementing a speech recognition system that adapts itself to its user.

3. There may be physical changes in the system. For example, in a robotic system, the components of the system may wear out, or sensors may degrade.

ONLINE LEARNING

In *online learning*, we do not write the error function over the whole sample but on individual instances. Starting from random initial weights, at each iteration we adjust the parameters a little bit to minimize the error, without forgetting what we have previously learned. If this error function is differentiable, we can use gradient descent.

For example, in regression the error on the single instance pair with index $t$, $(x^t, r^t)$, is

$$E^t(\boldsymbol{w}|\boldsymbol{x}^t, r^t) = \frac{1}{2}(r^t - y^t)^2 = \frac{1}{2}[r^t - (\boldsymbol{w}^T\boldsymbol{x}^t)]^2$$

and for $j = 0, \ldots, d$, the online update is

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \tag{11.7}$$

where $\eta$ is the learning factor, which is gradually decreased in time for convergence. This is known as *stochastic gradient descent*.

STOCHASTIC GRADIENT DESCENT

Similarly, update rules can be derived for classification problems using logistic discrimination where updates are done after each pattern, instead of summing them and doing the update after a complete pass over the training set. With two classes, for the single instance $(\boldsymbol{x}^t, \boldsymbol{r}^t)$ where $r_i^t = 1$ if $\boldsymbol{x}^t \in C_1$ and $r_i^t = 0$ if $\boldsymbol{x}^t \in C_2$, the single output is

$$y^t = \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x}^t)$$

and the cross-entropy is

$$E^t(\boldsymbol{w}|\boldsymbol{x}^t, r^t) = -r^t \log y^t - (1 - r^t)\log(1 - y^t)$$

Using gradient descent, we get the following online update rule for $j = 0, \ldots, d$:

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \tag{11.8}$$

When there are $K > 2$ classes, for the single instance $(\boldsymbol{x}^t, \boldsymbol{r}^t)$ where $r_i^t = 1$ if $\boldsymbol{x}^t \in C_i$ and 0 otherwise, the outputs are

$$y_i^t = \frac{\exp \boldsymbol{w}_i^T \boldsymbol{x}^t}{\sum_k \exp \boldsymbol{w}_k^T \boldsymbol{x}^t}$$

and the cross-entropy is

$$E^t(\{\boldsymbol{w}_i\}_i|\boldsymbol{x}^t, \boldsymbol{r}^t) = -\sum_i r_i^t \log y_i^t$$

Using gradient descent, we get the following online update rule, for $i = 1, \ldots, K,\ j = 0, \ldots, d$:

$$\Delta w_{ij}^t = \eta(r_i^t - y_i^t)x_j^t \tag{11.9}$$

which is the same as the equations we saw in section 10.7 except that we do not sum over all of the instances but update after a single instance. The pseudocode of the algorithm is given in figure 11.3, which is the online version of figure 10.8.

Both equations 11.7 and 11.9 have the form

$$\text{Update} = \text{LearningFactor} \cdot (\text{DesiredOutput} - \text{ActualOutput}) \cdot \text{Input} \tag{11.10}$$

Let us try to get some insight into what this does. First, if the actual output is equal to the desired output, no update is done. When it is done, the magnitude of the update increases as the difference between the desired output and the actual output increases. We also see that if the actual output is less than the desired output, update is positive if the input is positive and negative if the input is negative. This has the effect of increasing the actual output and decreasing the difference. If the actual output is greater than the desired output, update is negative if the input is positive and positive if the input is negative; this decreases the actual output and makes it closer to the desired output.

```
For $i = 1, \ldots, K$
    For $j = 0, \ldots, d$
        $w_{ij} \leftarrow$ rand($-0.01, 0.01$)
Repeat
    For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order
        For $i = 1, \ldots, K$
            $o_i \leftarrow 0$
            For $j = 0, \ldots, d$
                $o_i \leftarrow o_i + w_{ij} x_j^t$
        For $i = 1, \ldots, K$
            $y_i \leftarrow \exp(o_i) / \sum_k \exp(o_k)$
        For $i = 1, \ldots, K$
            For $j = 0, \ldots, d$
                $w_{ij} \leftarrow w_{ij} + \eta(r_i^t - y_i) x_j^t$
Until convergence
```

**Figure 11.3** Perceptron training algorithm implementing stochastic online gradient descent for the case with $K > 2$ classes. This is the online version of the algorithm given in figure 10.8.

When an update is done, its magnitude depends also on the input. If the input is close to 0, its effect on the actual output is small and therefore its weight is also updated by a small amount. The greater an input, the greater the update of its weight.

Finally, the magnitude of the update depends on the learning factor, $\eta$. If it is too large, updates depend too much on recent instances; it is as if the system has a very short memory. If this factor is small, many updates may be needed for convergence. In section 11.8.1, we discuss methods to speed up convergence.

## 11.4 Learning Boolean Functions

In a Boolean function, the inputs are binary and the output is 1 if the corresponding function value is true and 0 otherwise. Therefore, it can be seen as a two-class classification problem. As an example, for learning to AND two inputs, the table of inputs and required outputs is given in table 11.1. An example of a perceptron that implements AND and its

**Table 11.1** Input and output for the AND function.

| $x_1$ | $x_2$ | $r$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

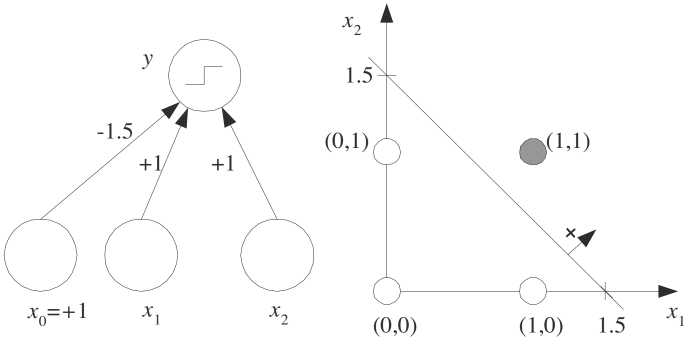

**Figure 11.4** The perceptron that implements AND and its geometric interpretation.

geometric interpretation in two dimensions is given in figure 11.4. The discriminant is

$$y = s(x_1 + x_2 - 1.5)$$

that is, $\boldsymbol{x} = [1, x_1, x_2]^T$ and $\boldsymbol{w} = [-1.5, 1, 1]^T$. Note that $y = s(x_1+x_2-1.5)$ satisfies the four constraints given by the definition of AND function in table 11.1, for example, for $x_1 = 1, x_2 = 0$, $y = s(-0.5) = 0$. Similarly it can be shown that $y = s(x_1 + x_2 - 0.5)$ implements OR.

Though Boolean functions like AND and OR are linearly separable and are solvable using the perceptron, certain functions like XOR are not. The table of inputs and required outputs for XOR is given in table 11.2. As can be seen in figure 11.5, the problem is not linearly separable. This can also be proved by noting that there are no $w_0, w_1$, and $w_2$ values that

**Table 11.2** Input and output for the XOR function.

| $x_1$ | $x_2$ | $r$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

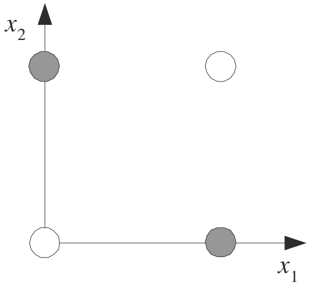

**Figure 11.5** XOR problem is not linearly separable. We cannot draw a line where the empty circles are on one side and the filled circles on the other side.

satisfy the following set of inequalities:

$$\begin{array}{rrrl} & & w_0 & \leq 0 \\ & w_2+ & w_0 & > 0 \\ w_1+ & & w_0 & > 0 \\ w_1+ & w_2+ & w_0 & \leq 0 \end{array}$$

This result should not be very surprising to us since the VC dimension of a line (in two dimensions) is three. With two binary inputs there are four cases, and thus we know that there exist problems with two inputs that are not solvable using a line; XOR is one of them.

## 11.5 Multilayer Perceptrons

A perceptron that has a single layer of weights can only approximate linear functions of the input and cannot solve problems like the XOR, where the discrimininant to be estimated is nonlinear. Similarly, a perceptron cannot be used for nonlinear regression. This limitation does not apply to feedforward networks with intermediate or *hidden layers* between the input and the output layers. If used for classification, such *multilayer perceptrons* (MLP) can implement nonlinear discriminants and, if used for regression, can approximate nonlinear functions of the input.

HIDDEN LAYERS

MULTILAYER PERCEPTRONS

Input $\boldsymbol{x}$ is fed to the input layer (including the bias), the “activation” propagates in the forward direction, and the values of the hidden units $z_h$ are calculated (see figure 11.6). Each hidden unit is a perceptron by itself and applies the nonlinear sigmoid function to its weighted sum:

$$z_h = \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj}x_j + w_{h0}\right)\right]},\ h = 1, \ldots, H \tag{11.11}$$

The output $y_i$ are perceptrons in the second layer taking the hidden units as their inputs

$$y_i = \boldsymbol{v}_i^T \boldsymbol{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0} \tag{11.12}$$

where there is also a bias unit in the hidden layer, which we denote by $z_0$, and $v_{i0}$ are the bias weights. The input layer of $x_j$ is not counted since no computation is done there and when there is a hidden layer, this is a two-layer network.

As usual, in a regression problem, there is no nonlinearity in the output layer in calculating $y$. In a two-class discrimination task, there is one sigmoid output unit and when there are $K > 2$ classes, there are $K$ outputs with softmax as the output nonlinearity.

If the hidden units’ outputs were linear, the hidden layer would be of no use: linear combination of linear combinations is another linear combination. Sigmoid is the continuous, differentiable version of thresholding. We need differentiability because the learning equations we will see are gradient-based. Another sigmoid (S-shaped) nonlinear basis function that can be used is the hyperbolic tangent function, tanh, which ranges from $-1$ to $+1$, instead of 0 to $+1$. In practice, there is no difference between using the sigmoid and the tanh. Still another possibility is the Gaussian, which uses Euclidean distance instead of the dot product for similarity; we discuss such radial basis function networks in chapter 12.

The output is a linear combination of the nonlinear basis function values computed by the hidden units. It can be said that the hidden units make a nonlinear transformation from the $d$-dimensional input space to

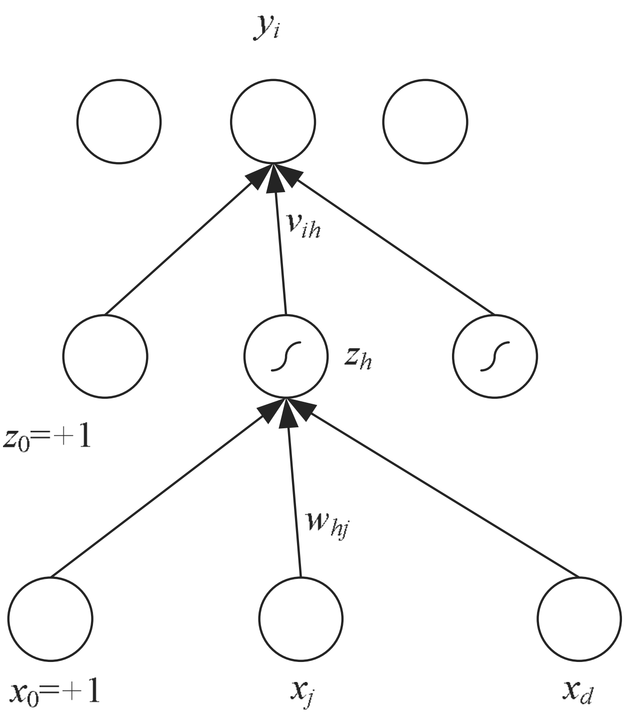

**Figure 11.6** The structure of a multilayer perceptron. $x_j, j = 0, \ldots, d$ are the inputs and $z_h, h = 1, \ldots, H$ are the hidden units where $H$ is the dimensionality of this hidden space. $z_0$ is the bias of the hidden layer. $y_i, i = 1, \ldots, K$ are the output units. $w_{hj}$ are weights in the first layer, and $v_{ih}$ are the weights in the second layer.

the $H$-dimensional space spanned by the hidden units, and, in this space, the second output layer implements a linear function.

One is not limited to having one hidden layer, and more hidden layers with their own incoming weights can be placed after the first hidden layer with sigmoid hidden units, thus calculating nonlinear functions of the first layer of hidden units and implementing more complex functions of the inputs. In practice, people rarely go beyond one hidden layer since analyzing a network with many hidden layers is quite complicated; but sometimes when the hidden layer contains too many hidden units, it may be sensible to go to multiple hidden layers, preferring “long and narrow” networks to “short and fat” networks.

## 11.6 MLP as a Universal Approximator

We can represent any Boolean function as a disjunction of conjunctions, and such a Boolean expression can be implemented by a multilayer perceptron with one hidden layer. Each conjunction is implemented by one hidden unit and the disjunction by the output unit. For example,

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } \sim x_2) \text{ OR } (\sim x_1 \text{ AND } x_2)$$

We have seen previously how to implement AND and OR using perceptrons. So two perceptrons can in parallel implement the two AND, and another perceptron on top can OR them together (see figure 11.7). We see that the first layer maps inputs from the $(x_1, x_2)$ to the $(z_1, z_2)$ space defined by the first-layer perceptrons. Note that both inputs, (0,0) and (1,1), are mapped to (0,0) in the $(z_1, z_2)$ space, allowing linear separability in this second space.

Thus in the binary case, for every input combination where the output is 1, we define a hidden unit that checks for that particular conjunction of the input. The output layer then implements the disjunction. Note that this is just an existence proof, and such networks may not be practical as up to $2^d$ hidden units may be necessary when there are $d$ inputs. Such an architecture implements table lookup and does not generalize.

UNIVERSAL APPROXIMATION

We can extend this to the case where inputs are continuous to show that similarly, any arbitrary function with continuous input and outputs can be approximated with a multilayer perceptron. The proof of *universal approximation* is easy with two hidden layers. For every input case or region, that region can be delimited by hyperplanes on all sides using hidden units on the first hidden layer. A hidden unit in the second layer then ANDs them together to bound the region. We then set the weight of the connection from that hidden unit to the output unit equal to the desired function value. This gives a *piecewise constant approximation* of the function; it corresponds to ignoring all the terms in the Taylor expansion except the constant term. Its accuracy may be increased to the desired value by increasing the number of hidden units and placing a finer grid on the input. Note that no formal bounds are given on the number of hidden units required. This property just reassures us that there is a solution; it does not help us in any other way. It has been proven that an MLP with *one* hidden layer (with an arbitrary number of hidden units) can learn any nonlinear function of the input (Hornik, Stinchcombe, and White 1989).

PIECEWISE CONSTANT APPROXIMATION

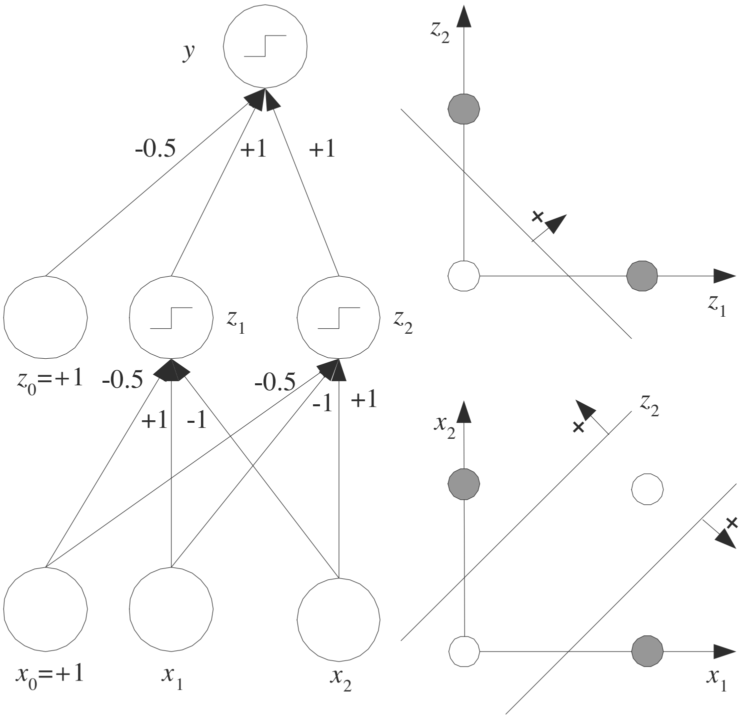

**Figure 11.7** The multilayer perceptron that solves the XOR problem. The hidden units and the output have the threshold activation function with threshold at 0.

## 11.7 Backpropagation Algorithm

Training a multilayer perceptron is the same as training a perceptron; the only difference is that now the output is a nonlinear function of the input thanks to the nonlinear basis function in the hidden units. Considering the hidden units as inputs, the second layer is a perceptron and we already know how to update the parameters, $v_{ij}$, in this case, given the inputs $z_h$. For the first-layer weights, $w_{hj}$, we use the chain rule to calculate the gradient:

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_h}\frac{\partial z_h}{\partial w_{hj}}$$

BACKPROPAGATION

It is as if the error propagates from the output $y$ back to the inputs and hence the name *backpropagation* was coined (Rumelhart, Hinton, and Williams 1986a).

### 11.7.1 Nonlinear Regression

Let us first take the case of nonlinear regression (with a single output) calculated as

$$y^t = \sum_{h=1}^{H} v_h z_h^t + v_0 \tag{11.13}$$

with $z_h$ computed by equation 11.11. The error function over the whole sample in regression is

$$E(\mathbf{W}, \boldsymbol{v}|\mathcal{X}) = \frac{1}{2}\sum_t (r^t - y^t)^2 \tag{11.14}$$

The second layer is a perceptron with hidden units as the inputs, and we use the least-squares rule to update the second-layer weights:

$$\Delta v_h = \eta \sum_t (r^t - y^t) z_h^t \tag{11.15}$$

The first layer are also perceptrons with the hidden units as the output units but in updating the first-layer weights, we cannot use the least-squares rule directly as we do not have a desired output specified for the hidden units. This is where the chain rule comes into play. We write

$$\begin{aligned}
\Delta w_{hj} &= -\eta \frac{\partial E}{\partial w_{hj}} \\
&= -\eta \sum_t \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial z_h^t} \frac{\partial z_h^t}{\partial w_{hj}} \\
&= -\eta \sum_t \underbrace{-(r^t - y^t)}_{\partial E^t/\partial y^t} \underbrace{v_h}_{\partial y^t/\partial z_h^t} \underbrace{z_h^t(1 - z_h^t)x_j^t}_{\partial z_h^t/\partial w_{hj}} \\
&= \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t
\end{aligned} \tag{11.16}$$

The product of the first two terms $(r^t - y^t)v_h$ acts like the error term for hidden unit $h$. This error is *backpropagated* from the error to the hidden unit. $(r^t - y^t)$ is the error in the output, weighted by the "responsibility" of the hidden unit as given by its weight $v_h$. In the third term, $z_h(1 - z_h)$

is the derivative of the sigmoid and $x_j^t$ is the derivative of the weighted sum with respect to the weight $w_{hj}$. Note that the change in the first-layer weight, $\Delta w_{hj}$, makes use of the second-layer weight, $v_h$. Therefore, we should calculate the changes in both layers and update the first-layer weights, making use of the *old* value of the second-layer weights, then update the second-layer weights.

Weights, $w_{hj}, v_h$ are started from small random values initially, for example, in the range $[-0.01, 0.01]$, so as not to saturate the sigmoids. It is also a good idea to normalize the inputs so that they all have 0 mean and unit variance and have the same scale, since we use a single $\eta$ parameter.

BATCH LEARNING

With the learning equations given here, for each pattern, we compute the direction in which each parameter needs be changed and the magnitude of this change. In *batch learning*, we accumulate these changes over all patterns and make the change once after a complete pass over the whole training set is made, as shown in the previous update equations.

EPOCH

It is also possible to have online learning, by updating the weights after each pattern, thereby implementing stochastic gradient descent. A complete pass over all the patterns in the training set is called an *epoch*. The learning factor, $\eta$, should be chosen smaller in this case and patterns should be scanned in a random order. Online learning converges faster because there may be similar patterns in the dataset, and the stochasticity has an effect like adding noise and may help escape local minima.

An example of training a multilayer perceptron for regression is shown in figure 11.8. As training continues, the MLP fit gets closer to the underlying function and error decreases (see figure 11.9). Figure 11.10 shows how the MLP fit is formed as a sum of the outputs of the hidden units.

It is also possible to have multiple output units, in which case a number of regression problems are learned at the same time. We have

$$y_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0} \tag{11.17}$$

and the error is

$$E(\mathbf{W}, \mathbf{V}|\mathcal{X}) = \frac{1}{2} \sum_t \sum_i (r_i^t - y_i^t)^2 \tag{11.18}$$

The batch update rules are then

$$\Delta v_{ih} = \eta \sum_t (r_i^t - y_i^t) z_h^t \tag{11.19}$$

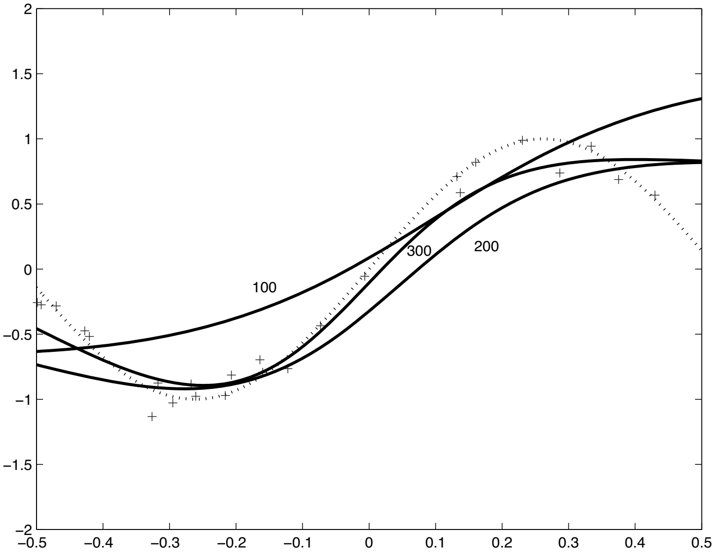

**Figure 11.8** Sample training data shown as '+', where $x^t \sim U(-0.5, 0.5)$, and $y^t = f(x^t) + \mathcal{N}(0, 0.1)$. $f(x) = \sin(6x)$ is shown by a dashed line. The evolution of the fit of an MLP with two hidden units after 100, 200, and 300 epochs is drawn.

$$\Delta w_{hj} \quad = \quad \eta \sum_t \left[ \sum_i (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t \tag{11.20}$$

$\sum_i (r_i^t - y_i^t) v_{ih}$ is the accumulated backpropagated error of hidden unit $h$ from all output units. Pseudocode is given in figure 11.11. Note that in this case, all output units share the same hidden units and thus use the same hidden representation, hence, we are assuming that corresponding to these different outputs, we have related prediction problems. An alternative is to train separate multilayer perceptrons for the separate regression problems, each with its own separate hidden units.

### 11.7.2 Two-Class Discrimination

When there are two classes, one output unit suffices:

$$y^t = \text{sigmoid}\left( \sum_{h=1}^{H} v_h z_h^t + v_0 \right) \tag{11.21}$$

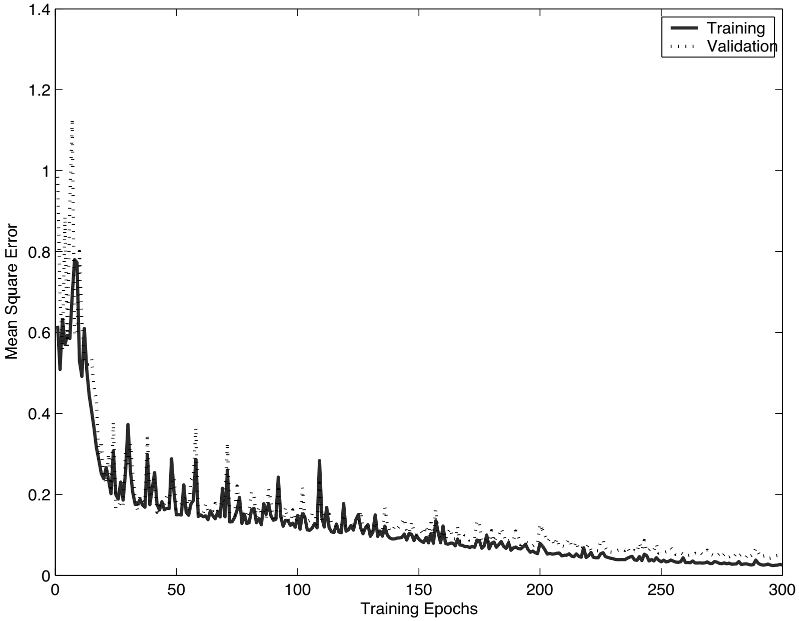

**Figure 11.9** The mean square error on training and validation sets as a function of training epochs.

which approximates $P(C_1|\boldsymbol{x}^t)$ and $\hat{P}(C_2|\boldsymbol{x}^t) \equiv 1 - y^t$. We remember from section 10.7 that the error function in this case is

$$E(\mathbf{W}, \boldsymbol{v}|\mathcal{X}) = -\sum_t r^t \log y^t + (1 - r^t)\log(1 - y^t) \tag{11.22}$$

The update equations implementing gradient descent are

$$\Delta v_h = \eta \sum_t (r^t - y^t) z_h^t \tag{11.23}$$

$$\Delta w_{hj} = \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t \tag{11.24}$$

As in the simple perceptron, the update equations for regression and classification are identical (which does not mean that the values are).

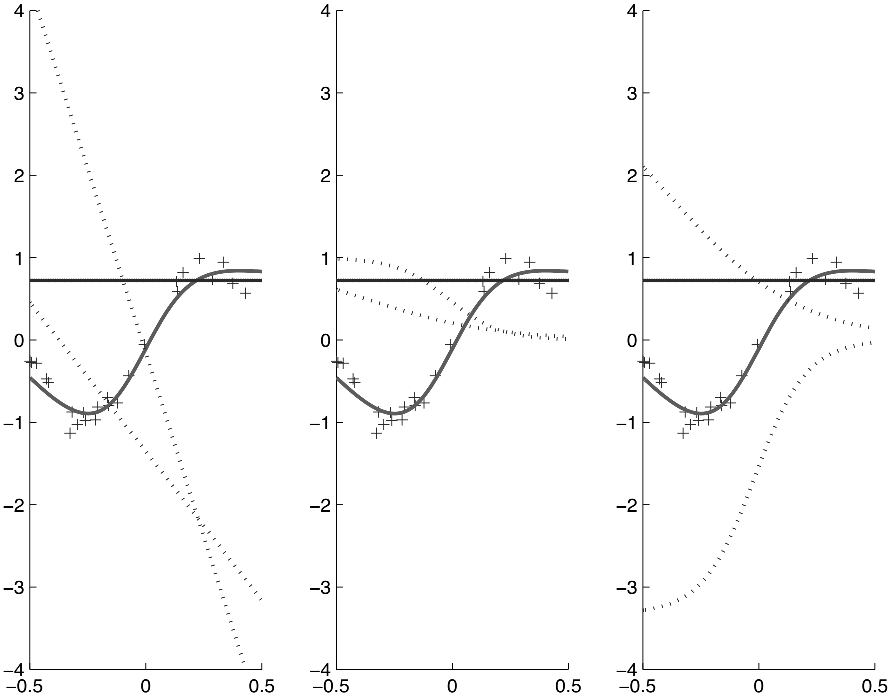

**Figure 11.10** (a) The hyperplanes of the hidden unit weights on the first layer, (b) hidden unit outputs, and (c) hidden unit outputs multiplied by the weights on the second layer. Two sigmoid hidden units slightly displaced, one multiplied by a negative weight, when added, implement a bump. With more hidden units, a better approximation is attained (see figure 11.12).

### 11.7.3 Multiclass Discrimination

In a $(K > 2)$-class classification problem, there are $K$ outputs

$$o_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0} \tag{11.25}$$

and we use softmax to indicate the dependency between classes; namely, they are mutually exclusive and exhaustive:

$$y_i^t = \frac{\exp o_i^t}{\sum_k \exp o_k^t} \tag{11.26}$$

```
Initialize all $v_{ih}$ and $w_{hj}$ to rand$(-0.01, 0.01)$
Repeat
    For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order
        For $h = 1, \ldots, H$
            $z_h \leftarrow \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}^t)$
        For $i = 1, \ldots, K$
            $y_i = \boldsymbol{v}_i^T \boldsymbol{z}$
        For $i = 1, \ldots, K$
            $\Delta \boldsymbol{v}_i = \eta (r_i^t - y_i^t) \boldsymbol{z}$
        For $h = 1, \ldots, H$
            $\Delta \boldsymbol{w}_h = \eta (\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) \boldsymbol{x}^t$
        For $i = 1, \ldots, K$
            $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i + \Delta \boldsymbol{v}_i$
        For $h = 1, \ldots, H$
            $\boldsymbol{w}_h \leftarrow \boldsymbol{w}_h + \Delta \boldsymbol{w}_h$
Until convergence
```

**Figure 11.11** Backpropagation algorithm for training a multilayer perceptron for regression with $K$ outputs. This code can easily be adapted for two-class classification (by setting a single sigmoid output) and to $K > 2$ classification (by using softmax outputs).

where $y_i$ approximates $P(C_i|\boldsymbol{x}^t)$. The error function is

$$E(\mathbf{W}, \mathbf{V}|\mathcal{X}) = -\sum_t \sum_i r_i^t \log y_i^t \tag{11.27}$$

and we get the update equations using gradient descent:

$$\Delta v_{ih} = \eta \sum_t (r_i^t - y_i^t) z_h^t \tag{11.28}$$

$$\Delta w_{hj} = \eta \sum_t \left[ \sum_i (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t \tag{11.29}$$

Richard and Lippmann (1991) have shown that given a network of enough complexity and sufficient training data, a suitably trained multilayer perceptron estimates posterior probabilities.

### 11.7.4 Multiple Hidden Layers

As we saw before, it is possible to have multiple hidden layers each with its own weights and applying the sigmoid function to its weighted sum. For regression, let us say, if we have a multilayer perceptron with two hidden layers, we write

$$\begin{aligned} z_{1h} &= \text{sigmoid}(\boldsymbol{w}_{1h}^T \boldsymbol{x}) = \text{sigmoid}\left(\sum_{j=1}^{d} w_{1hj} x_j + w_{1h0}\right),\ h = 1, \ldots, H_1 \\ z_{2l} &= \text{sigmoid}(\boldsymbol{w}_{2l}^T \boldsymbol{z}_1) = \text{sigmoid}\left(\sum_{h=0}^{H_1} w_{2lh} z_{1h} + w_{2l0}\right),\ l = 1, \ldots, H_2 \\ y &= \boldsymbol{v}^T \boldsymbol{z}_2 = \sum_{l=1}^{H_2} v_l z_{2l} + v_0 \end{aligned}$$

where $\boldsymbol{w}_{1h}$ and $\boldsymbol{w}_{2l}$ are the first- and second-layer weights, $z_{1h}$ and $z_{2h}$ are the units on the first and second hidden layers, and $\boldsymbol{v}$ are the third-layer weights. Training such a network is similar except that to train the first-layer weights, we need to backpropagate one more layer (exercise 5).

## 11.8 Training Procedures

### 11.8.1 Improving Convergence

Gradient descent has various advantages. It is simple. It is local; namely, the change in a weight uses only the values of the presynaptic and postsynaptic units and the error (suitably backpropagated). When online training is used, it does not need to store the training set and can adapt as the task to be learned changes. Because of these reasons, it can be (and is) implemented in hardware. But by itself, gradient descent converges slowly. When learning time is important, one can use more sophisticated optimization methods (Battiti 1992). Bishop (1995) discusses in detail the application of conjugate gradient and second-order methods to the training of multilayer perceptrons. However, there are two frequently used simple techniques that improve the performance of the gradient descent considerably, making gradient-based methods feasible in real applications.

### Momentum

Let us say $w_i$ is any weight in a multilayer perceptron in any layer, including the biases. At each parameter update, successive $\Delta w_i^t$ values may be so different that large oscillations may occur and slow convergence. $t$ is the time index that is the epoch number in batch learning and the iteration number in online learning. The idea is to take a running average by incorporating the previous update in the current change as if there is a *momentum* due to previous updates:

MOMENTUM

$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1} \tag{11.30}$$

$\alpha$ is generally taken between 0.5 and 1.0. This approach is especially useful when online learning is used, where as a result we get an effect of averaging and smooth the trajectory during convergence. The disadvantage is that the past $\Delta w_i^{t-1}$ values should be stored in extra memory.

### Adaptive Learning Rate

In gradient descent, the learning factor $\eta$ determines the magnitude of change to be made in the parameter. It is generally taken between 0.0 and 1.0, mostly less than or equal to 0.2. It can be made adaptive for faster convergence, where it is kept large when learning takes place and is decreased when learning slows down:

$$\Delta\eta = \begin{cases} +a & \text{if } E^{t+\tau} < E^t \\ -b\eta & \text{otherwise} \end{cases} \tag{11.31}$$

Thus we increase $\eta$ by a constant amount if the error on the training set decreases and decrease it geometrically if it increases. Because $E$ may oscillate from one epoch to another, it is a better idea to take the average of the past few epochs as $E^t$.

## 11.8.2 Overtraining

A multilayer perceptron with $d$ inputs, $H$ hidden units, and $K$ outputs has $H(d+1)$ weights in the first layer and $K(H+1)$ weights in the second layer. Both the space and time complexity of an MLP is $\mathcal{O}(H \cdot (K+d))$. When $e$ denotes the number of training epochs, training time complexity is $\mathcal{O}(e \cdot H \cdot (K+d))$.

In an application, $d$ and $K$ are predefined and $H$ is the parameter that we play with to tune the complexity of the model. We know from previous chapters that an overcomplex model memorizes the noise in the training set and does not generalize to the validation set. For example, we have previously seen this phenomenon in the case of polynomial regression where we noticed that in the presence of noise or small samples, increasing the polynomial order leads to worse generalization. Similarly in an MLP, when the number of hidden units is large, the generalization accuracy deteriorates (see figure 11.12), and the bias/variance dilemma also holds for the MLP, as it does for any statistical estimator (Geman, Bienenstock, and Doursat 1992).

A similar behavior happens when training is continued too long: as more training epochs are made, the error on the training set decreases, but the error on the validation set starts to increase beyond a certain point (see figure 11.13). Remember that initially all the weights are close to 0 and thus have little effect. As training continues, the most important weights start moving away from 0 and are utilized. But if training is continued further on to get less and less error on the training set, almost all weights are updated away from 0 and effectively become parameters. Thus as training continues, it is as if new parameters are added to the system, increasing the complexity and leading to poor generalization. Learning should be *stopped early* to alleviate this problem of *overtraining*. The optimal point to stop training, and the optimal number of hidden units, is determined through cross-validation, which involves testing the network's performance on validation data unseen during training.

EARLY STOPPING
OVERTRAINING

Because of the nonlinearity, the error function has many minima and gradient descent converges to the nearest minimum. To be able to assess expected error, the same network is trained a number of times starting from different initial weight values, and the average of the validation error is computed.

### 11.8.3 Structuring the Network

In some applications, we may believe that the input has a local structure. For example, in vision we know that nearby pixels are correlated and there are local features like edges and corners; any object, for example, a handwritten digit, may be defined as a combination of such primitives. Similarly, in speech, locality is in time and inputs close in time can be grouped as speech primitives. By combining these primitives, longer ut-

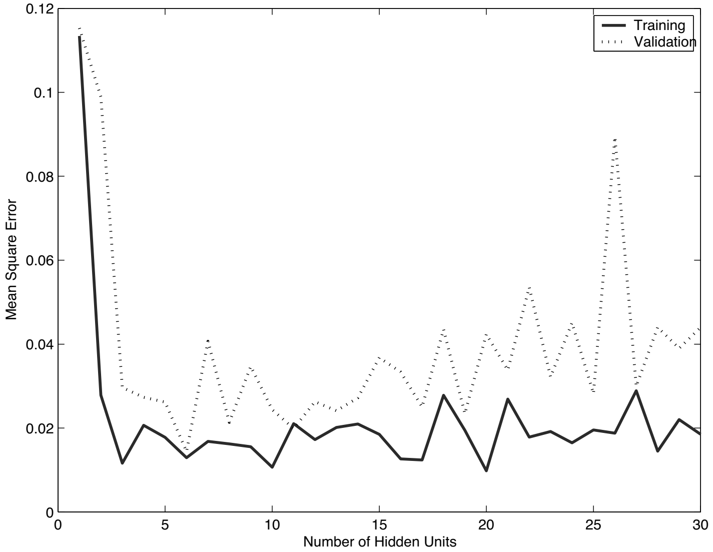

**Figure 11.12** As complexity increases, training error is fixed but the validation error starts to increase and the network starts to overfit.

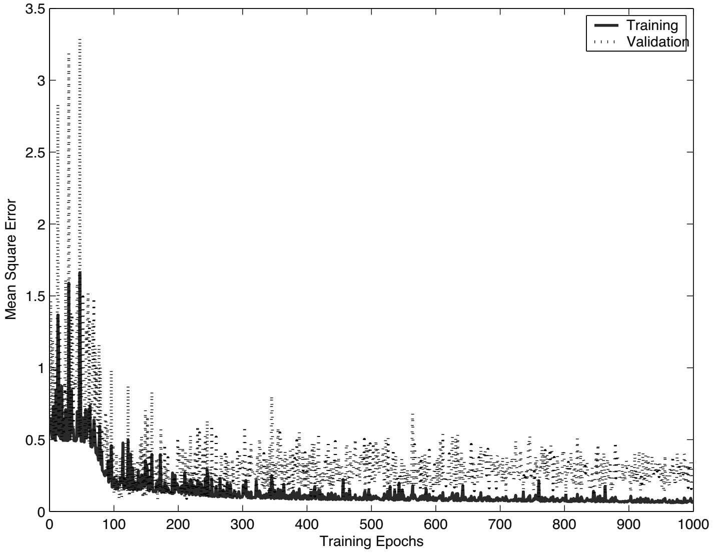

**Figure 11.13** As training continues, the validation error starts to increase and the network starts to overfit.

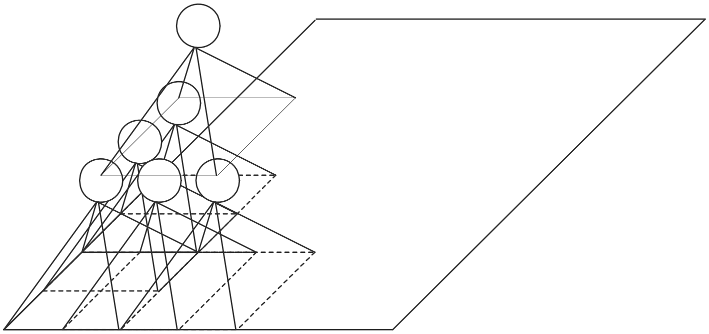

**Figure 11.14** A structured MLP. Each unit is connected to a local group of units below it and checks for a particular feature—for example, edge, corner, and so forth—in vision. Only one hidden unit is shown for each region; typically there are many to check for different local features.

terances, for example, speech phonemes, may be defined. In such a case when designing the MLP, hidden units are not connected to all input units because not all inputs are correlated. Instead, we define hidden units that define a window over the input space and are connected to only a small local subset of the inputs. This decreases the number of connections and therefore the number of free parameters (Le Cun et al. 1989).

We can repeat this in successive layers where each layer is connected to a small number of local units below and checks for a more complicated feature by combining the features below in a larger part of the input space until we get to the output layer (see figure 11.14). For example, the input may be pixels. By looking at pixels, the first hidden layer units may learn to check for edges of various orientations. Then by combining edges, the second hidden layer units can learn to check for combinations of edges—for example, arcs, corners, line ends—and then combining them in upper layers, the units can look for semi-circles, rectangles, or in the case of a face recognition application, eyes, mouth, and so forth. This is the example of a *hierarchical cone* where features get more complex, abstract, and fewer in number as we go up the network until we get to classes.

HIERARCHICAL CONE

In such a case, we can further reduce the number of parameters by *weight sharing*. Taking the example of visual recognition again, we can see that when we look for features like oriented edges, they may be

WEIGHT SHARING

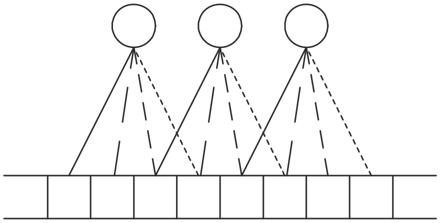

**Figure 11.15** In weight sharing, different units have connections to different inputs but share the same weight value (denoted by line type). Only one set of units is shown; there should be multiple sets of units, each checking for different features.

present in different parts of the input space. So instead of defining independent hidden units learning different features in different parts of the input space, we can have copies of the same hidden units looking at different parts of the input space (see figure 11.15). During learning, we calculate the gradients by taking different inputs, then we average these up and make a single update. This implies a single parameter that defines the weight on multiple connections. Also, because the update on a weight is based on gradients for several inputs, it is as if the training set is effectively multiplied.

### 11.8.4 Hints

The knowledge of local structure allows us to prestructure the multilayer network, and with weight sharing it has fewer parameters. The alternative of an MLP with completely connected layers has no such structure and is more difficult to train. Knowledge of any sort related to the application should be built into the network structure whenever possible. These are called *hints* (Abu-Mostafa 1995) and are the properties of the target function that are known to us independent of the training examples. HINTS

In image recognition, there are invariance hints: the identity of an object does not change when it is rotated, translated, or scaled (see figure 11.16). Hints are auxiliary information that can be used to guide the learning process and are especially useful when the training set is limited. There are different ways in which hints can be used:

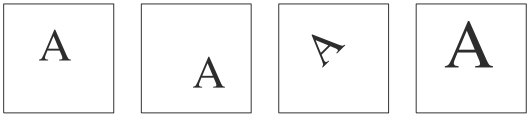

**Figure 11.16** The identity of the object does not change when it is translated, rotated, or scaled. Note that this may not always be true, or may be true up to a point: 'b' and 'q' are rotated versions of each other. These are hints that can be incorporated into the learning process to make learning easier.

VIRTUAL EXAMPLES

1. Hints can be used to create *virtual examples.* For example, knowing that the object is invariant to scale, from a given training example, we can generate multiple copies at different scales and add them to the training set with the same label. This has the advantage that we increase the training set and do not need to modify the learner in any way. The problem may be that too many examples may be needed for the learner to learn the invariance.

2. The invariance may be implemented as a preprocessing stage. For example, optical character readers have a preprocessing stage where the input character image is centered and normalized for size and slant. This is the easiest solution, when it is possible.

3. The hint may be incorporated into the network structure. Local structure and weight sharing, which we saw in section 11.8.3, is one example where we get invariance to small translations and rotations.

4. The hint may also be incorporated by modifying the error function. Let us say we know that $\boldsymbol{x}$ and $\boldsymbol{x}'$ are the same from the application's point of view, where $\boldsymbol{x}'$ may be a "virtual example" of $\boldsymbol{x}$. That is, $f(\boldsymbol{x}) = f(\boldsymbol{x}')$, when $f(\boldsymbol{x})$ is the function we would like to approximate. Let us denote by $g(\boldsymbol{x}|\theta)$, our approximation function, for example, an MLP where $\theta$ are its weights. Then, for all such pairs $(\boldsymbol{x}, \boldsymbol{x}')$, we define the penalty function

   $$E_h = [g(\boldsymbol{x}|\theta) - g(\boldsymbol{x}'|\theta)]^2$$

   and add it as an extra term to the usual error function:

   $$E' = E + \lambda_h \cdot E_h$$

This is a penalty term penalizing the cases where our predictions do not obey the hint, and $\lambda_h$ is the weight of such a penalty (Abu-Mostafa 1995).

Another example is the approximation hint: Let us say that for $x$, we do not know the exact value, $f(x)$, but we know that it is in the interval, $[a_x, b_x]$. Then our added penalty term is

$$E_h = \begin{cases} 0 & \text{if } g(x|\theta) \in [a_x, b_x] \\ (g(x) - a_x)^2 & \text{if } g(x|\theta) < a_x \\ (g(x) - b_x)^2 & \text{if } g(x|\theta) > b_x \end{cases}$$

This is similar to the error function used in support vector regression (section 13.10), which tolerates small approximation errors.

TANGENT PROP

Still another example is the *tangent prop* (Simard et al. 1992) where the transformation against which we are defining the hint—for example, rotation by an angle—is modeled by a function. The usual error function is modified (by adding another term) so as to allow parameters to move along this line of transformation without changing the error.

## 11.9 Tuning the Network Size

STRUCTURAL ADAPTATION

Previously we saw that when the network is too large and has too many free parameters, generalization may not be well. To find the optimal network size, the most common approach is to try many different architectures, train them all on the training set, and choose the one that generalizes best to the validation set. Another approach is to incorporate this *structural adaptation* into the learning algorithm. There are two ways this can be done:

1. In the *destructive* approach, we start with a large network and gradually remove units and/or connections that are not necessary.

2. In the *constructive* approach, we start with a small network and gradually add units and/or connections to improve performance.

WEIGHT DECAY

One destructive method is *weight decay* where the idea is to remove unnecessary connections. Ideally to be able to determine whether a unit or connection is necessary, we need to train once with and once without and check the difference in error on a separate validation set. This is costly since it should be done for all combinations of such units/connections.

Given that a connection is not used if its weight is 0, we give each connection a tendency to decay to 0 so that it disappears unless it is reinforced explicitly to decrease error. For any weight $w_i$ in the network, we use the update rule:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} - \lambda w_i \tag{11.32}$$

This is equivalent to doing gradient descent on the error function with an added penalty term, penalizing networks with many nonzero weights:

$$E' = E + \frac{\lambda}{2} \sum_i w_i^2 \tag{11.33}$$

Simpler networks are better generalizers is a hint that we implement by adding a penalty term. Note that we are not saying that simple networks are always better than large networks; we are saying that if we have two networks that have the same training error, the simpler one—namely, the one with fewer weights—has a higher probability of better generalizing to the validation set.

The effect of the second term in equation 11.32 is like that of a spring that pulls each weight to 0. Starting from a value close to 0, unless the actual error gradient is large and causes an update, due to the second term, the weight will gradually decay to 0. $\lambda$ is the parameter that determines the relative importances of the error on the training set and the complexity due to nonzero parameters and thus determines the speed of decay: With large $\lambda$, weights will be pulled to 0 no matter what the training error is; with small $\lambda$, there is not much penalty for nonzero weights. $\lambda$ is fine-tuned using cross-validation.

DYNAMIC NODE CREATION

Instead of starting from a large network and *pruning* unnecessary connections or units, one can start from a small network and add units and associated connections should the need arise (figure 11.17). In *dynamic node creation* (Ash 1989), an MLP with one hidden layer with one hidden unit is trained and after convergence, if the error is still high, another hidden unit is added. The incoming weights of the newly added unit and its outgoing weight are initialized randomly and trained with the previously existing weights that are not reinitialized and continue from their previous values.

CASCADE CORRELATION

In *cascade correlation* (Fahlman and Lebiere 1990), each added unit

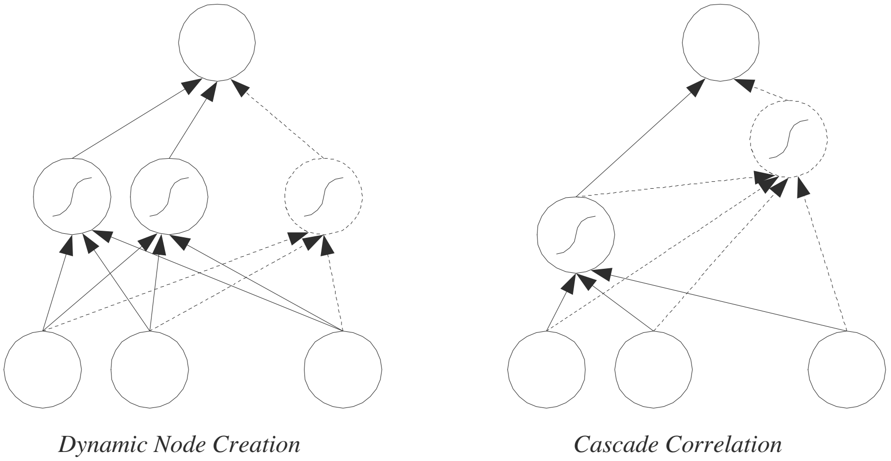

**Figure 11.17** Two examples of constructive algorithms. Dynamic node creation adds a unit to an existing layer. Cascade correlation adds each unit as a new hidden layer connected to all the previous layers. Dashed lines denote the newly added unit/connections. Bias units/weights are omitted for clarity.

is a new hidden unit in another hidden layer. Every hidden layer has only one unit that is connected to all of the hidden units preceding it and the inputs. The previously existing weights are frozen and are not trained; only the incoming and outgoing weights of the newly added unit are trained.

Dynamic node creation adds a new hidden unit to an existing hidden layer and never adds another hidden layer. Cascade correlation always adds a new hidden layer with a single unit. The ideal constructive method should be able to decide when to introduce a new hidden layer and when to add a unit to an existing layer. This is an open research problem.

Incremental algorithms are interesting because they correspond to modifying not only the parameters but also the model structure during learning. We can think of a space defined by the structure of the multilayer perceptron and operators corresponding to adding/removing unit(s) or layer(s) to move in this space (Aran et al. 2009). Incremental algorithms then do a search in this state space where operators are tried (according to some order) and accepted or rejected depending on some goodness measure, for example, some combination of complexity and validation error. Another example would be a setting in polynomial regression where high-order terms are added/removed during training automatically, fitting model complexity to data complexity. As the cost of computation gets lower, such automatic model selection should be a part of the learning process done automatically without any user interference.

## 11.10 Bayesian View of Learning

The Bayesian approach in training neural networks considers the parameters, namely, connection weights, $w_i$, as random variables drawn from a prior distribution $p(w_i)$ and computes the posterior probability given the data

$$p(\boldsymbol{w}|\mathcal{X}) = \frac{p(\mathcal{X}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{X})} \tag{11.34}$$

where $\boldsymbol{w}$ is the vector of all weights of the network. The MAP estimate $\hat{\boldsymbol{w}}$ is the mode of the posterior

$$\hat{\boldsymbol{w}}_{MAP} = \arg\max_{\boldsymbol{w}} \log p(\boldsymbol{w}|\mathcal{X}) \tag{11.35}$$

Taking the log of equation 11.34, we get

$$\log p(\boldsymbol{w}|\mathcal{X}) = \log p(\mathcal{X}|\boldsymbol{w}) + \log p(\boldsymbol{w}) + C$$

The first term on the right is the log likelihood, and the second is the log of the prior. If the weights are independent and the prior is taken as Gaussian, $\mathcal{N}(0, 1/2\lambda)$

$$p(\boldsymbol{w}) = \prod_i p(w_i) \text{ where } p(w_i) = c \cdot \exp\left[-\frac{w_i^2}{2(1/2\lambda)}\right] \tag{11.36}$$

the MAP estimate minimizes the augmented error function

$$E' = E + \lambda\|\boldsymbol{w}\|^2 \tag{11.37}$$

where $E$ is the usual classification or regression error (negative log likelihood). This augmented error is exactly the error function we used in weight decay (equation 11.33). Using a large $\lambda$ assumes small variability in parameters, puts a larger force on them to be close to 0, and takes the prior more into account than the data; if $\lambda$ is small, then the allowed variability of the parameters is larger. This approach of removing unnecessary parameters is known as *ridge regression* in statistics. RIDGE REGRESSION

REGULARIZATION This is another example of *regularization* with a cost function, combining the fit to data and model complexity

$$\text{cost} = \text{data-misfit} + \lambda \cdot \text{complexity} \tag{11.38}$$

The use of Bayesian estimation in training multilayer perceptrons is treated in MacKay 1992a, b. We are going to talk about Bayesian estimation in more detail in chapter 14.

SOFT WEIGHT SHARING

Empirically, it has been seen that after training, most of the weights of a multilayer perceptron are distributed normally around 0, justifying the use of weight decay. But this may not always be the case. Nowlan and Hinton (1992) proposed *soft weight sharing* where weights are drawn from a mixture of Gaussians, allowing them to form multiple clusters, not one. Also, these clusters may be centered anywhere and not necessarily at 0, and have variances that are modifiable. This changes the prior of equation 11.36 to a mixture of $M \geq 2$ Gaussians

$$p(w_i) = \sum_{j=1}^{M} \alpha_j p_j(w_i) \tag{11.39}$$

where $\alpha_j$ are the priors and $p_j(w_i) \sim \mathcal{N}(m_j, s_j^2)$ are the component Gaussians. $M$ is set by the user and $\alpha_j, m_j, s_j$ are learned from the data. Using such a prior and augmenting the error function with its log during training, the weights converge to decrease error and also are grouped automatically to increase the log prior.

## 11.11 Dimensionality Reduction

In a multilayer perceptron, if the number of hidden units is less than the number of inputs, the first layer performs a dimensionality reduction. The form of this reduction and the new space spanned by the hidden units depend on what the MLP is trained for. If the MLP is for classification with output units following the hidden layer, then the new space is defined and the mapping is learned to minimize classification error (see figure 11.18).

We can get an idea of what the MLP is doing by analyzing the weights. We know that the dot product is maximum when the two vectors are identical. So we can think of each hidden unit as defining a template in its incoming weights, and by analyzing these templates, we can extract knowledge from a trained MLP. If the inputs are normalized, weights tell us of their relative importance. Such analysis is not easy but gives us

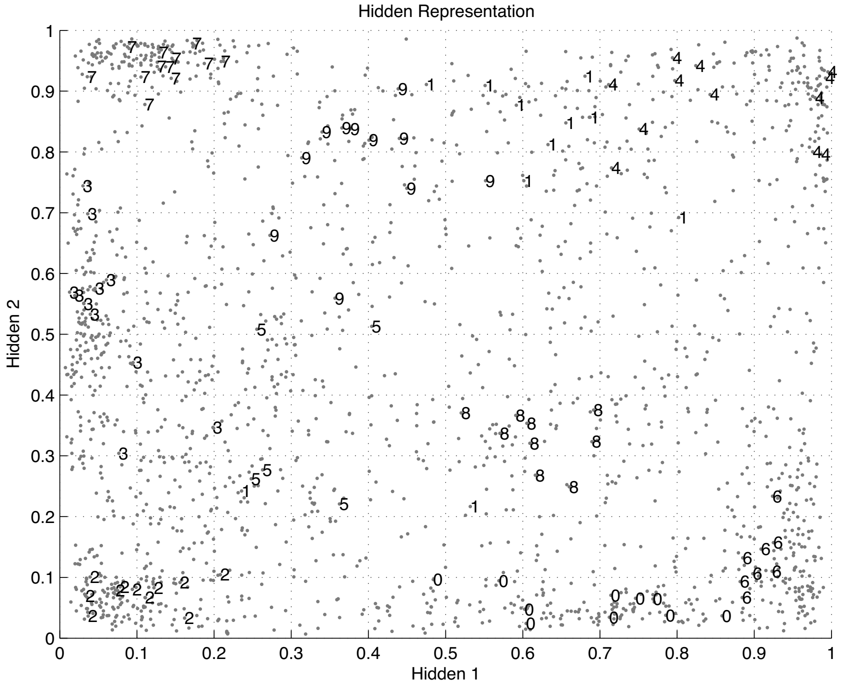

**Figure 11.18** Optdigits data plotted in the space of the two hidden units of an MLP trained for classification. Only the labels of one hundred data points are shown. This MLP with sixty-four inputs, two hidden units, and ten outputs has 80 percent accuracy. Because of the sigmoid, hidden unit values are between 0 and 1 and classes are clustered around the corners. This plot can be compared with the plots in chapter 6, which are drawn using other dimensionality reduction methods on the same dataset.

some insight as to what the MLP is doing and allows us to peek into the black box.

AUTOASSOCIATOR

An interesting architecture is the *autoassociator* (Cottrell, Munro, and Zipser 1987), which is an MLP architecture where there are as many outputs as there are inputs, and the required outputs are defined to be equal to the inputs (see figure 11.19). To be able to reproduce the inputs again at the output layer, the MLP is forced to find the best representation of the inputs in the hidden layer. When the number of hidden units is less than the number of inputs, this implies dimensionality reduction. Once

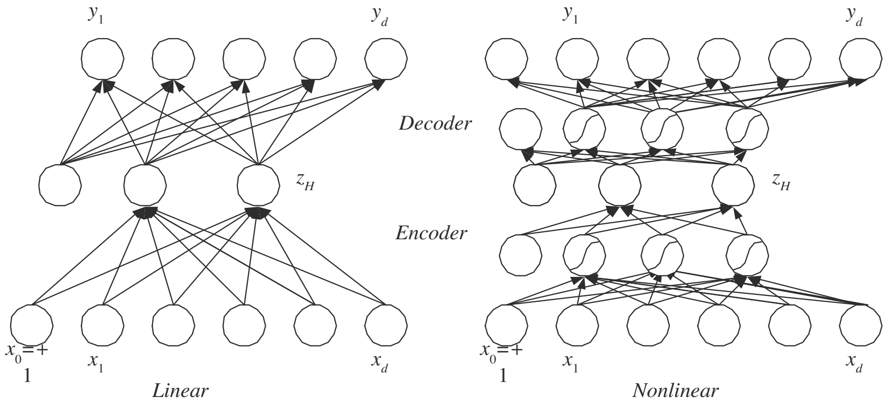

**Figure 11.19** In the autoassociator, there are as many outputs as there are inputs and the desired outputs are the inputs. When the number of hidden units is less than the number of inputs, the MLP is trained to find the best coding of the inputs on the hidden units, performing dimensionality reduction. On the left, the first layer acts as an encoder and the second layer acts as the decoder. On the right, if the encoder and decoder are multilayer perceptrons with sigmoid hidden units, the network performs nonlinear dimensionality reduction.

the training is done, the first layer from the input to the hidden layer acts as an encoder, and the values of the hidden units make up the encoded representation. The second layer from the hidden units to the output units acts as a decoder, reconstructing the original signal from its encoded representation.

It has been shown (Bourlard and Kamp 1988) that an MLP with one hidden layer of units implements principal components analysis (section 6.3), except that the hidden unit weights are not the eigenvectors sorted in importance using the eigenvalues but span the same space as the $H$ principal eigenvectors. If the encoder and decoder are not one layer but multilayer perceptrons with sigmoid nonlinearity in the hidden units, the encoder implements nonlinear dimensionality reduction (Hinton and Salakhutdinov 2006).

Another way to use an MLP for dimensionality reduction is through multidimensional scaling (section 6.5). Mao and Jain (1995) show how an MLP can be used to learn the *Sammon mapping*. Recalling equation 6.29, SAMMON MAPPING

Sammon stress is defined as

$$E(\theta|\mathcal{X}) = \sum_{r,s} \left[ \frac{\|\boldsymbol{g}(\boldsymbol{x}^r|\theta) - \boldsymbol{g}(\boldsymbol{x}^s|\theta)\| - \|\boldsymbol{x}^r - \boldsymbol{x}^s\|}{\|\boldsymbol{x}^r - \boldsymbol{x}^s\|} \right]^2 \tag{11.40}$$

An MLP with $d$ inputs, $H$ hidden units, and $k < d$ output units is used to implement $g(\boldsymbol{x}|\theta)$, mapping the $d$-dimensional input to a $k$-dimensional vector, where $\theta$ corresponds to the weights of the MLP. Given a dataset of $\mathcal{X} = \{\boldsymbol{x}^t\}_t$, we can use gradient descent to minimize the Sammon stress directly to learn the MLP, namely, $g(\boldsymbol{x}|\theta)$, such that the distances between the $k$-dimensional representations are as close as possible to the distances in the original space.

## 11.12 Learning Time

Until now, we have been concerned with cases where the input is fed once, all together. In some applications, the input is temporal where we need to learn a temporal sequence. In others, the output may also change in time. Examples are as follows:

- *Sequence recognition.* This is the assignment of a given sequence to one of several classes. Speech recognition is one example where the input signal sequence is the spoken speech and the output is the code of the word spoken. That is, the input changes in time but the output does not.

- *Sequence reproduction.* Here, after seeing part of a given sequence, the system should predict the rest. Time-series prediction is one example where the input is given but the output changes.

- *Temporal association.* This is the most general case where a particular output sequence is given as output after a specific input sequence. The input and output sequences may be different. Here both the input and the output change in time.

### 11.12.1 Time Delay Neural Networks

The easiest way to recognize a temporal sequence is by converting it to a spatial sequence. Then any method discussed up to this point can be utilized for classification. In a *time delay neural network* (Waibel et al. 1989),

TIME DELAY NEURAL NETWORK

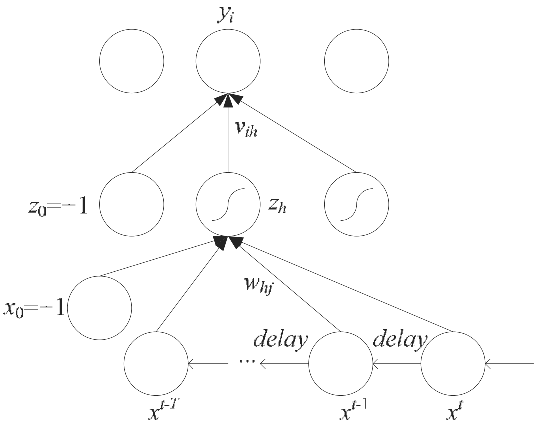

**Figure 11.20** A time delay neural network. Inputs in a time window of length $T$ are delayed in time until we can feed all $T$ inputs as the input vector to the MLP.

previous inputs are delayed in time so as to synchronize with the final input, and all are fed together as input to the system (see figure 11.20). Backpropagation can then be used to train the weights. To extract features local in time, one can have layers of structured connections and weight sharing to get translation invariance in time. The main restriction of this architecture is that the size of the time window we slide over the sequence should be fixed a priori.

### 11.12.2 Recurrent Networks

RECURRENT NETWORK

In a *recurrent network*, additional to the feedforward connections, units have self-connections or connections to units in the previous layers. This recurrency acts as a short-term memory and lets the network remember what happened in the past.

Most frequently, one uses a partially recurrent network where a limited number of recurrent connections are added to a multilayer perceptron (see figure 11.21). This combines the advantage of the nonlinear approximation ability of a multilayer perceptron with the temporal representation ability of the recurrency, and such a network can be used to implement any of the three temporal association tasks. It is also possible to have hidden units in the recurrent backward connections, these being

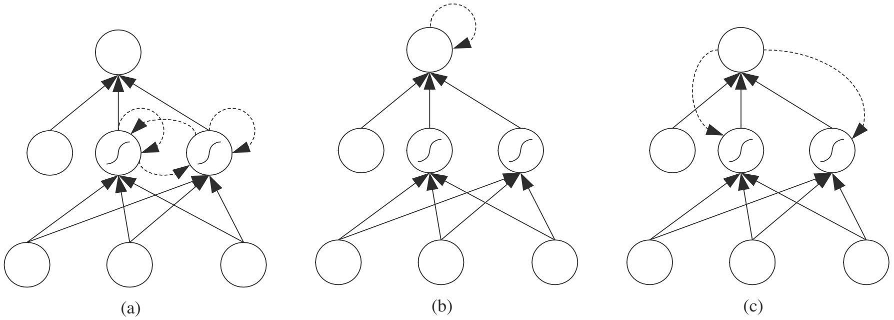

**Figure 11.21** Examples of MLP with partial recurrency. Recurrent connections are shown with dashed lines: (a) self-connections in the hidden layer, (b) self-connections in the output layer, and (c) connections from the output to the hidden layer. Combinations of these are also possible.

known as *context units*. No formal results are known to determine how to choose the best architecture given a particular application.

UNFOLDING IN TIME

If the sequences have a small maximum length, then *unfolding in time* can be used to convert an arbitrary recurrent network to an equivalent feedforward network (see figure 11.22). A separate unit and connection is created for copies at different times. The resulting network can be trained with backpropagation with the additional requirement that all copies of each connection should remain identical. The solution, as in weight sharing, is to sum up the different weight changes in time and change the weight by the average. This is called *backpropagation through time* (Rumelhart, Hinton, and Willams 1986b). The problem with this approach is the memory requirement if the length of the sequence is large. *Real time recurrent learning* (Williams and Zipser 1989) is an algorithm for training recurrent networks without unfolding and has the advantage that it can use sequences of arbitrary length.

BACKPROPAGATION THROUGH TIME

REAL TIME RECURRENT LEARNING

## 11.13 Notes

Research on artificial neural networks is as old as the digital computer. McCulloch and Pitts (1943) proposed the first mathematical model for the artificial neuron. Rosenblatt (1962) proposed the perceptron model and a learning algorithm in 1962. Minsky and Papert (1969) showed the limita-

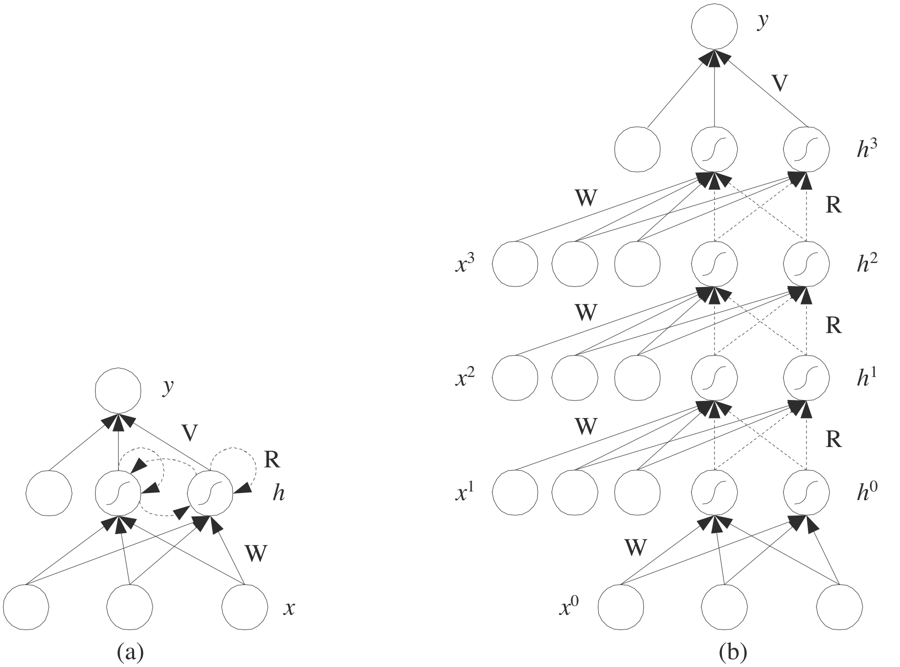

**Figure 11.22** Backpropagation through time: (a) recurrent network, and (b) its equivalent unfolded network that behaves identically in four steps.

tion of single-layer perceptrons, for example, the XOR problem, and since there was no algorithm to train a multilayer perceptron with a hidden layer at that time, the work on artificial neural networks almost stopped except at a few places. The renaissance of neural networks came with the paper by Hopfield (1982). This was followed by the two-volume Parallel Distributed Processing (PDP) book written by the PDP Research Group (Rumelhart, McClelland, and the PDP Research Group 1986). It seems as though backpropagation was invented independently in several places almost at the same time and the limitation of a single-layer perceptron no longer held.

Starting in the mid-1980s, there has been a huge explosion of work on artificial neural network models from various disciplines: physics, statistics, psychology, cognitive science, neuroscience, and lingustics, not to mention computer science, electrical engineering, and adaptive control.

Perhaps the most important contribution of research on artificial neural networks is this synergy that bridged various disciplines, especially statistics and engineering. It is thanks to this that the field of machine learning is now well established.

The field is much more mature now; aims are more modest and better defined. One of the criticisms of backpropagation was that it was not biologically plausible! Though the term “neural network” is still widely used, it is generally understood that neural network models, for example, multilayer perceptrons, are nonparametric estimators and that the best way to analyze them is by using statistical methods.

PROJECTION PURSUIT

For example, a statistical method similar to the multilayer perceptron is *projection pursuit* (Friedman and Stuetzle 1981), which is written as

$$y = \sum_{h=1}^{H} \phi_h(\boldsymbol{w}_h^T \boldsymbol{x})$$

the difference being that each “hidden unit” has its own separate function, $\phi_h(\cdot)$, though in an MLP, all are fixed to be sigmoid. In chapter 12, we will see another neural network structure, named radial basis functions, which uses the Gaussian function at the hidden units.

There are various textbooks on artificial neural networks: Hertz, Krogh, and Palmer 1991, the earliest, is still readable. Bishop 1995 has a pattern recognition emphasis and discusses in detail various optimization algorithms that can be used for training, as well as the Bayesian approach, generalizing weight decay. Ripley 1996 analyzes neural networks from a statistical perspective.

Artificial neural networks, for example, multilayer perceptrons, have various successful applications. In addition to their various successful applications in adaptive control, speech recognition, and vision, two are noteworthy: Tesauro’s TD-Gammon program (Tesauro 1994) uses reinforcement learning (chapter 18) to train a multilayer perceptron and plays backgammon at a master level. Pomerleau’s ALVINN is a neural network that autonomously drives a van up to 20 miles per hour after learning by observing a driver for five minutes (Pomerleau 1991).

## 11.14 Exercises

1. Show the perceptron that calculates NOT of its input.
2. Show the perceptron that calculates NAND of its two inputs.

3. Show the perceptron that calculates the parity of its three inputs.
4. Derive the update equations when the hidden units use tanh, instead of the sigmoid. Use the fact that $\tanh' = (1 - \tanh^2)$.
5. Derive the update equations for an MLP with two hidden layers.
6. Consider a MLP architecture with one hidden layer where there are also direct weights from the inputs directly to the output units. Explain when such a structure would be helpful and how it can be trained.
7. Parity is cyclic shift invariant; for example, “0101” and “1010” have the same parity. Propose a multilayer perceptron to learn the parity function using this hint.
8. In cascade correlation, what are the advantages of freezing the previously existing weights?
9. Derive the update equations for an MLP implementing Sammon mapping that minimizes Sammon stress (equation 11.40).
10. In section 11.6, we discuss how a MLP with two hidden layers can implement piecewise constant approximation. Show that if the weight in the last layer is not a constant but a linear function of the input, we can implement piecewise linear approximation.
11. Derive the update equations for soft weight sharing.
12. In the autoassociator network, how can we decide on the number of hidden units?
13. Incremental learning of the structure of a MLP can be viewed as a state space search. What are the operators? What is the goodness function? What type of search strategies are appropriate? Define these in such a way that dynamic node creation and cascade-correlation are special instantiations.
14. For the MLP given in figure 11.22, derive the update equations for the unfolded network.

## 11.15 References

Abu-Mostafa, Y. 1995. “Hints.” *Neural Computation* 7: 639–671.

Aran, O., O. T. Yıldız, and E. Alpaydın. 2009. “An Incremental Framework Based on Cross-Validation for Estimating the Architecture of a Multilayer Perceptron.” *International Journal of Pattern Recognition and Artificial Intelligence* 23: 159–190.

Ash, T. 1989. “Dynamic Node Creation in Backpropagation Networks.” *Connection Science* 1: 365–375.

Battiti, R. 1992. “First- and Second-Order Methods for Learning: Between Steepest Descent and Newton’s Method.” *Neural Computation* 4: 141–166.

Bishop, C. M. 1995. *Neural Networks for Pattern Recognition.* Oxford: Oxford University Press.

Bourlard, H., and Y. Kamp. 1988. “Auto-Association by Multilayer Perceptrons and Singular Value Decomposition.” *Biological Cybernetics* 59: 291–294.

Cottrell, G. W., P. Munro, and D. Zipser. 1987. “Learning Internal Representations from Gray-Scale Images: An Example of Extensional Programming.” In *Ninth Annual Conference of the Cognitive Science Society*, 462–473. Hillsdale, NJ: Erlbaum.

Durbin, R., and D. E. Rumelhart. 1989. “Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks.” *Neural Computation* 1: 133–142.

Fahlman, S. E., and C. Lebiere. 1990. “The Cascade Correlation Architecture.” In *Advances in Neural Information Processing Systems 2*, ed. D. S. Touretzky, 524–532. San Francisco: Morgan Kaufmann.

Friedman, J. H., and W. Stuetzle. 1981. “Projection Pursuit Regression.” *Journal of the American Statistical Association* 76: 817–823.

Geman, S., E. Bienenstock, and R. Doursat. 1992. “Neural Networks and the Bias/Variance Dilemma.” *Neural Computation* 4: 1–58.

Hertz, J., A. Krogh, and R. G. Palmer. 1991. *Introduction to the Theory of Neural Computation.* Reading, MA: Addison Wesley.

Hinton, G. E., and R. R. Salakhutdinov. 2006. “Reducing the dimensionality of data with neural networks.” *Science* 313: 504–507.

Hopfield, J. J. 1982. “Neural Networks and Physical Systems with Emergent Collective Computational Abilities.” *Proceedings of the National Academy of Sciences USA* 79: 2554–2558.

Hornik, K., M. Stinchcombe, and H. White. 1989. “Multilayer Feedforward Networks Are Universal Approximators.” *Neural Networks* 2: 359–366.

Le Cun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. “Backpropagation Applied to Handwritten Zipcode Recognition.” *Neural Computation* 1: 541–551.

MacKay, D. J. C. 1992a. “Bayesian Interpolation.” *Neural Computation* 4: 415–447.

MacKay, D. J. C. 1992b. “A Practical Bayesian Framework for Backpropagation Networks” *Neural Computation* 4: 448–472.

Mao, J., and A. K. Jain. 1995. “Artificial Neural Networks for Feature Extraction and Multivariate Data Projection.” *IEEE Transactions on Neural Networks* 6: 296–317.

Marr, D. 1982. *Vision.* New York: Freeman.

McCulloch, W. S., and W. Pitts. 1943. “A Logical Calculus of the Ideas Immenent in Nervous Activity.” *Bulletin of Mathematical Biophysics* 5: 115–133.

Minsky, M. L., and S. A. Papert. 1969. *Perceptrons.* Cambridge, MA: MIT Press. (Expanded ed. 1990.)

Nowlan, S. J., and G. E. Hinton. 1992. “Simplifying Neural Networks by Soft Weight Sharing.” *Neural Computation* 4: 473–493.

Pomerleau, D. A. 1991. “Efficient Training of Artificial Neural Networks for Autonomous Navigation.” *Neural Computation* 3: 88–97.

Posner, M. I., ed. 1989. *Foundations of Cognitive Science.* Cambridge, MA: MIT Press.

Richard, M. D., and R. P. Lippmann. 1991. “Neural Network Classifiers Estimate Bayesian *a Posteriori* Probabilities.” *Neural Computation* 3: 461–483.

Ripley, B. D. 1996. *Pattern Recognition and Neural Networks.* Cambridge, UK: Cambridge University Press.

Rosenblatt, F. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* New York: Spartan.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986a. “Learning Representations by Backpropagating Errors.” *Nature* 323: 533–536.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986b. “Learning Internal Representations by Error Propagation.” In *Parallel Distributed Processing*, ed. D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, 318–362. Cambridge, MA: MIT Press.

Rumelhart, D. E., J. L. McClelland, and the PDP Research Group, eds. 1986. *Parallel Distributed Processing.* Cambridge, MA: MIT Press.

Simard, P., B. Victorri, Y, Le Cun, and J. Denker. 1992. “Tangent Prop: A Formalism for Specifying Selected Invariances in an Adaptive Network.” In *Advances in Neural Information Processing Systems 4*, ed. J. E. Moody, S. J. Hanson, and R. P. Lippman, 895–903. San Francisco: Morgan Kaufmann.

Tesauro, G. 1994. “TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play.” *Neural Computation* 6: 215–219.

Thagard, P. 2005. *Mind: Introduction to Cognitive Science.* 2nd ed. Cambridge, MA: MIT Press.

Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. 1989. “Phoneme Recognition Using Time-Delay Neural Networks.” *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37: 328–339.

Williams, R. J., and D. Zipser. 1989. “A Learning Algorithm for Continually Running Fully Recurrent Neural Networks.” *Neural Computation* 1: 270–280.