Neural Networks

9

Neural networks are networks of nerve cells in the brains of humans and animals. The human brain has about 100 billion nerve cells. We humans owe our intelligence and our ability to learn various motor and intellectual capabilities to the brain's complex relays and adaptivity. For many centuries biologists, psychologists, and doctors have tried to understand how the brain functions. Around 1900 came the revolutionary realization that these tiny physical building blocks of the brain, the nerve cells and their connections, are responsible for awareness, associations, thoughts, consciousness, and the ability to learn.

The first big step toward neural networks in AI was made 1943 by McCulloch and Pitts in an article entitled "A logical calculus of the ideas immanent in nervous activity" [AR88]. They were the first to present a mathematical model of the neuron as the basic switching element of the brain. This article laid the foundation for the construction of artificial neural networks and thus for this very important branch of AI.

We could consider the field of modeling and simulation of neural networks to be the bionics branch within AI.[1] Nearly all areas of AI attempt to recreate cognitive processes, such as in logic or in probabilistic reasoning. However, the tools used for modeling—namely mathematics, programming languages, and digital computers—have very little in common with the human brain. With artificial neural networks, the approach is different. Starting from knowledge about the function of natural neural networks, we attempt to model, simulate, and even reconstruct them in hardware. Every researcher in this area faces the fascinating and exciting challenge of comparing results with the performance of humans.

In this chapter we will attempt to outline the historical progression by defining a model of the neuron and its interconnectivity, starting from the most important biological insights. Then we will present several important and fundamental models: the Hopfield model, two simple associative memory models, and the—exceedingly important in practice—backpropagation algorithm.

[1] Bionics is concerned with unlocking the "discoveries of living nature" and its innovative conversion into technology [Wik13].

W. Ertel, *Introduction to Artificial Intelligence*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-58487-4_9

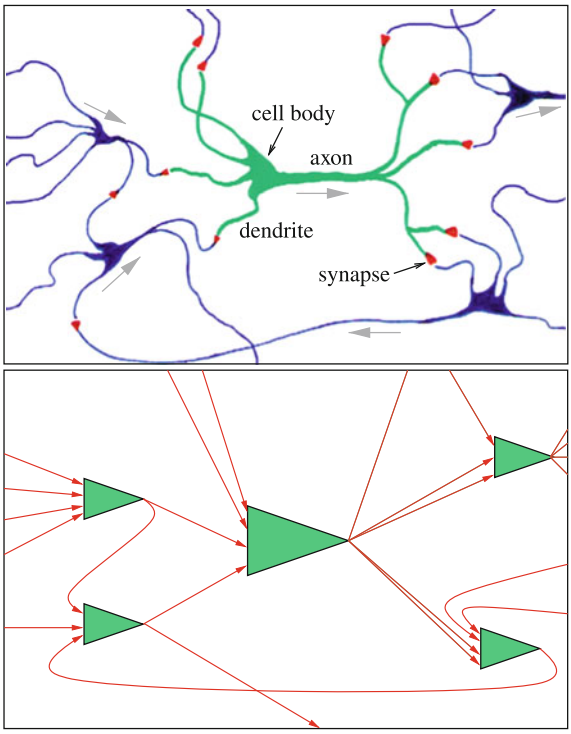

Fig. 9.1 Two stages of the modeling of a neural network. *Above* a biological model and *below* a formal model with neurons and directed connections between them

9.1 From Biology to Simulation

Each of the roughly 100 billion neurons in a human brain has, as shown in a simplified representation in Fig. 9.1, the following structure and function. Besides the cell body, the neuron has an axon, which can make local connections to other neurons over the dendrites. The axon can, however, grow up to a meter long in the form of a nerve fiber through the body.

The cell body of the neuron can store small electrical charges, similarly to a capacitor or battery. This storage is loaded by incoming electrical impulses from other neurons. The more electric impulse comes in, the higher the voltage. If the voltage exceeds a certain threshold, the neuron will fire. This means that it unloads its store, in that it sends a spike over the axon and the synapses. The electrical current divides and reaches many other neurons over the synapses, in which the same process takes place.

Now the question of the structure of the neural network arises. Each of the roughly 10^{11} neurons in the brain is connected to roughly 1000 to 10 000 other neurons, which yields a total of over 10^{14} connections. If we further consider that this gigantic number of extremely thin connections is made up of soft, three-dimensional tissue and that experiments on human brains are not easy to carry out, then it becomes clear why we do not have a detailed circuit diagram of the

brain. Presumably we will never be capable of completely understanding the circuit diagram of our brain, based solely on its immense size.

From today's perspective, it is no longer worth even trying to make a complete circuit diagram of the brain, because the structure of the brain is adaptive. It changes itself on the fly and adapts according to the individual's activities and environmental influences. The central role here is played by the synapses, which create the connection between neurons. At the connection point between two neurons, it is as if two cables meet. However, the two leads are not perfectly conductively connective, rather there is a small gap, which the electrons cannot directly jump over. This gap is filled with chemical substances, so-called neurotransmitters. These can be ionized by an applied voltage and then transport a charge over the gap. The conductivity of this gap depends on many parameters, for example the concentration and the chemical composition of the neurotransmitter. It is enlightening that the function of the brain reacts very sensitively to changes of this synaptic connection, for example through the influence of alcohol or other drugs.

How does learning work in such a neural network? The surprising thing here is that it is not the actual active units, namely the neurons, which are adaptive, rather it is the connections between them, that is, the synapses. Specifically, this can change their conductivity. We know that a synapse is made stronger by however much more electrical current it must carry. Stronger here means that the synapse has a higher conductivity. Synapses which are used often obtain an increasingly higher weight. For synapses which are used infrequently or are not active at all, the conductivity continues to decrease. This can even lead to them dying off.

All neurons in the brain work asynchronously and in parallel, but, compared to a computer, at very low speed. The time for a neural impulse takes about a millisecond, exactly the same as the time for the ions to be transported over the synaptic gap. The clock frequency of the neuron then is under one kilohertz and is thus lower than that of modern computers by a factor of 10^6. This disadvantage, however, is more than compensated for in many complex cognitive tasks, such as image recognition, by the very high degree of parallel processing in the network of nerve cells.

The connection to the outside world comes about through sensor neurons, for example on the retina in the eyes, or through nerve cells with very long axons which reach from the brain to the muscles and thus can carry out actions such as the movement of a leg.

However, it is still unclear how the principles discussed make intelligent behavior possible. Just like many researchers in neuroscience, we will attempt to explain using simulations of a simple mathematical model how cognitive tasks, for example pattern recognition, become possible.

9.1.1 The Mathematical Model

First we replace the continuous time axis with a discrete time scale. The neuron i carries out the following calculation in a time step. The "loading" of the activation

potential is accomplished simply by summation of the weighted output values $x_1, \ldots, x_n$ of all incoming connections over the formula

$$\sum_{j=1}^{n} w_{ij} x_j .$$

This weighted sum is calculated by most neural models. Then an *activation function* f is applied to it and the result

$$x_i = f\left(\sum_{j=1}^{n} w_{ij} x_j \right)$$

is passed on to the neighboring neurons as output over the synaptic weights. In Fig. 9.2 this kind of modeled neuron is shown. For the activation function there are a number of possibilities. The simplest is the identity: $f(x) = x$. The neuron thus calculates only the weighted sum of the input values and passes this on. However, this frequently leads to convergence problems with the neural dynamics because the function $f(x) = x$ is unbounded and the function values can grow beyond all limits over time.

Very well restricted, in contrast, is the threshold function (Heaviside step function)

$$H_\Theta(x) = \begin{cases} 0 & \text{if } x < \Theta, \\ 1 & \text{else.} \end{cases}$$

The whole neuron then computes its output as

$$x_i = \begin{cases} 0 & \text{if } \sum_{j=1}^{n} w_{ij} x_j < \Theta, \\ 1 & \text{else.} \end{cases}$$

This formula is identical to (8.1) on page 187, in other words, to a perceptron with the threshold Θ (Fig. 9.3 on page 249). The input neurons $1, \ldots, n$ here have only the function of variables which pass on their externally set values $x_1, \ldots, x_n$ unchanged.

The step function is quite sensible for binary neurons because the activation of a neuron can only take on the values zero or one anyway. In contrast, for continuous

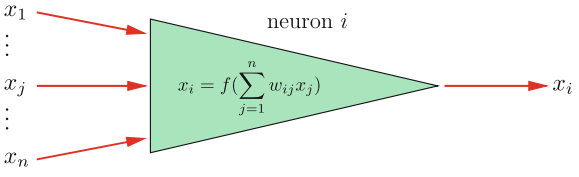

Fig. 9.2 The structure of a formal neuron, which applies the activation function f to the weighted sum of all inputs

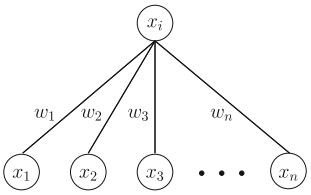

Fig. 9.3 The neuron with a step function works like a perceptron with a threshold

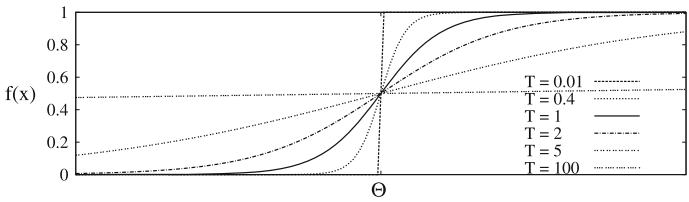

Fig. 9.4 The sigmoid function for various values of the parameter T. We can see that in the limit $T \to 0$ the step function results

neurons with activations between 0 and 1, the step function creates a discontinuity. However, this can be smoothed out by a *sigmoid function*, such as

$$f(x) = \frac{1}{1 + e^{-\frac{x-\Theta}{T}}}$$

with the graph in Fig. 9.4. Near the critical area around the threshold Θ, this function behaves close to linearly and it has an asymptotic limit. The smoothing can be varied by the parameter T.

Modeling learning is central to the theory of neural networks. As previously mentioned, one possibility of learning consists of strengthening a synapse according to how many electrical impulses it must transmit. This principle was postulated by D. Hebb in 1949 and is known as the *Hebb rule*:

If there is a connection w_{ij} between neuron j and neuron i and repeated signals are sent from neuron j to neuron i, which results in both neurons being simultaneously active, then the weight w_{ij} is reinforced. A possible formula for the weight change Δw_{ij} is

$$\Delta w_{ij} = \eta x_i x_j$$

with the constant η (learning rate), which determines the size of the individual learning steps.

There are many modifications of this rule, which then result in different types of networks or learning algorithms. In the following sections, we will become familiar with a few of these.

9.2 Hopfield Networks

Looking at the Hebb rule, we see that for neurons with values between zero and one, the weights can only grow with time. It is not possible for a neuron to weaken or even die according to this rule. This can be modeled, for example, by a decay constant which weakens an unused weight by a constant factor per time step, such as 0.99.

This problem is solved quite differently by the model presented by Hopfield in 1982 [Hop82]. It uses binary neurons, but with the two values −1 for inactive and 1 for active. Using the Hebb rule we obtain a positive contribution to the weight whenever two neurons are simultaneously active. If, however, only one of the two neurons is active, Δw_{ij} is negative.

Hopfield networks, which are a beautiful and visualizable example of *auto-associative memory*, are based on this idea. Patterns can be stored in auto-associative memory. To call up a saved pattern, it is sufficient to provide a similar pattern. The store then finds the most similar saved pattern. A classic application of this is handwriting recognition.

In the learning phase of a Hopfield network, N binary coded patterns, saved in the vectors $\boldsymbol{q}^1, \ldots, \boldsymbol{q}^N$, are supposed to be learned. Each component $q_i^j \in \{-1, 1\}$ of such a vector $\boldsymbol{q}^j$ represents a pixel of a pattern. For vectors consisting of n pixels, a neural network with n neurons is used, one for each pixel position. The neurons are fully connected with the restriction that the weight matrix is symmetric and all diagonal elements w_{ij} are zero. That is, there is no connection between a neuron and itself.

The fully connected network includes complex feedback loops, so-called *recurrences*, in the network (Fig. 9.5).

N patterns can be learned by simply calculating all weights with the formula

$$w_{ij} = \frac{1}{N} \sum_{k=1}^{N} q_i^k q_j^k. \tag{9.1}$$

This formula points out an interesting relationship to the Hebb rule. Each pattern in which the pixels i and j have the same value makes a positive contribution to the weight

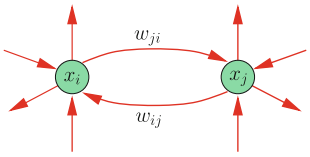

Fig. 9.5 Recurrent connections between two neurons in a Hopfield network

w_{ij}. Each other pattern makes a negative contribution. Since each pixel corresponds to a neuron, here the weights between neurons which simultaneously have the same value are being reinforced. Please note this small difference to the Hebb rule.

Once all the patterns have been stored, the network can be used for pattern recognition. We give the network a new pattern $\boldsymbol{x}$ and update the activations of all neurons in an asynchronous process according to the rule

$$x_i = \begin{cases} -1 & \text{if } \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} x_j < 0, \\ 1 & \text{else} \end{cases} \tag{9.2}$$

until the network becomes stable, that is, until no more activations change. As a program schema this reads as follows:

HOPFIELDASSOCIATOR($\boldsymbol{q}$)
Initialize all neurons: $\boldsymbol{x} = \boldsymbol{q}$
Repeat
  i = Random(1, n)
  Update neuron i according to (9.2)
Until $\boldsymbol{x}$ converges
Return ($\boldsymbol{x}$)

9.2.1 Application to a Pattern Recognition Example

We apply the described algorithm to a simple pattern recognition example. It should recognize digits in a 10 × 10 pixel field. The Hopfield network thus has 100 neurons with a total of

$$\frac{100 \cdot 99}{2} = 4950$$

weights. First the patterns of the digits 1, 2, 3, 4 in Fig. 9.6 above on page 252 are trained. That is, the weights are calculated by (9.1) on page 250. Then we put in the pattern with added noise and let the Hopfield dynamics run until convergence. In rows 2 to 4 in the figure, five snapshots of the network's development are shown during recognition. At 10% noise all four learned patterns are very reliably recognized. Above about 20% noise the algorithm frequently converges to other learned patterns or even to patterns which were not learned. Several such pattern are shown in Fig. 9.6 on page 252 below.

Now we save the digits 0 to 9 (Fig. 9.7 on page 253 top) in the same network and test the network again with patterns that have a random amount of about 10% inverted pixels. In the figure we clearly see that the Hopfield iteration often does not converge to the most similar learned state even for only 10% noise. Evidently the

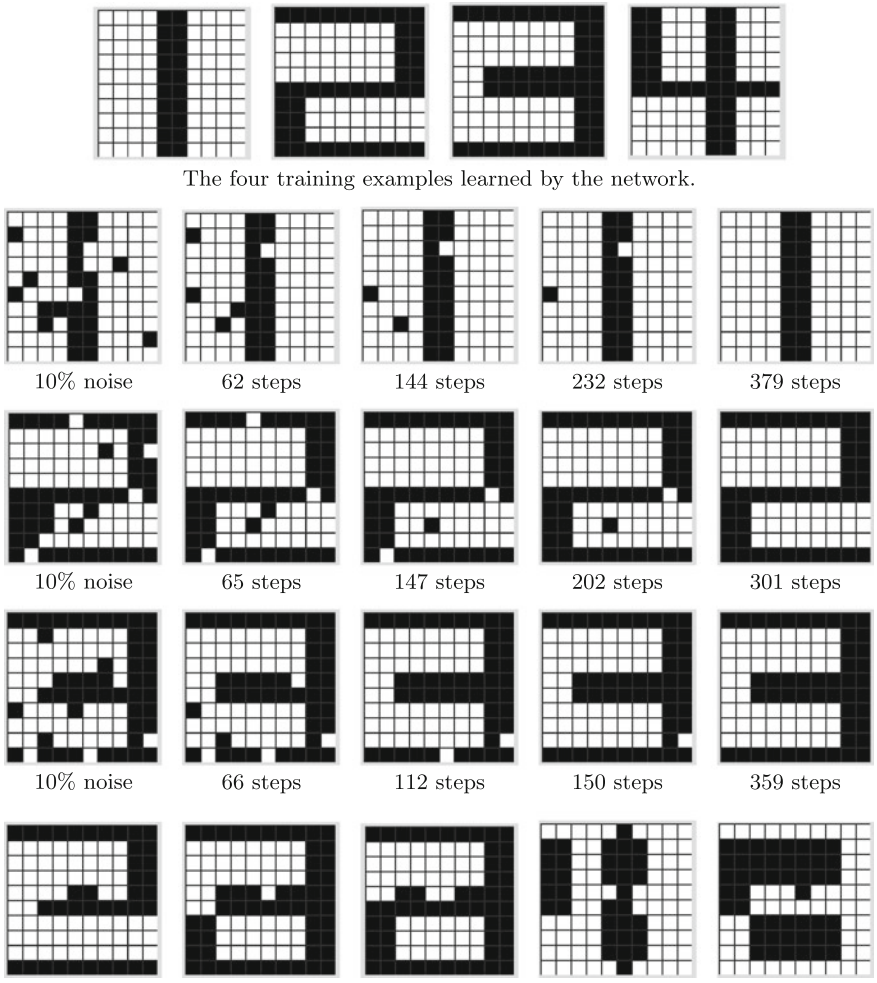

Fig. 9.6 Dynamics of a Hopfield network. In *rows 2*, *3* and *4* we can easily see how the network converges and the learned pattern is recognized after about 300 to 400 iterations. In the *last row* several stable states are shown which are reached by the network when the input pattern deviates too much from all learned patterns

network can securely save and recognize four patterns, but for ten patterns its memory capacity is exceeded. To understand this better, we will take a quick look into the theory of this network.

9.2.2 Analysis

In 1982, John Hopfield showed in [Hop82] that this model is formally equivalent to a physical model of magnetism. Small elementary magnets, so-called spins,

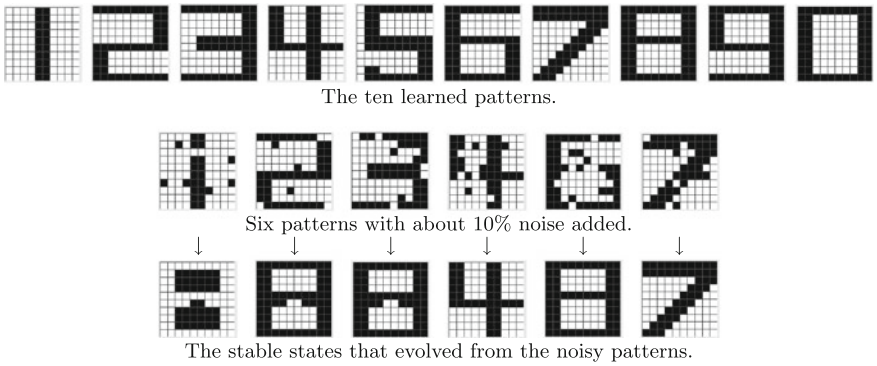

Fig. 9.7 For ten learned states the network shows chaotic behavior. Even with little noise the network converges to the wrong patterns or to artifacts

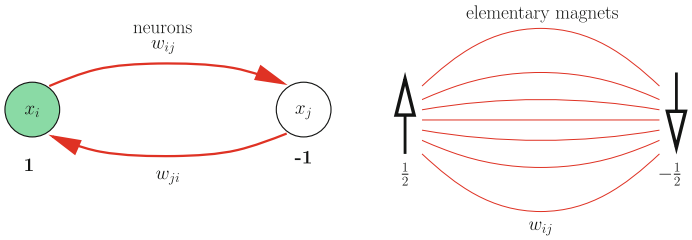

Fig. 9.8 Comparison between the neural and physical interpretation of the Hopfield model

mutually influence each other over their magnetic fields (see Fig. 9.8). If we observe two such spins i and j, they interact over a constant w_{ij} and the total energy of the system is then

$$E = -\frac{1}{2}\sum_{i,j} w_{ij}x_i x_j.$$

By the way, $w_{ii} = 0$ in physics too, because particles have no self-interaction. Because physical interactions are symmetric, $w_{ij} = w_{ji}$.

A physical system in equilibrium takes on a (stable) state of minimal energy and thus minimizes $E(\boldsymbol{x}, \boldsymbol{y})$. If such a system is brought into an arbitrary state, then it moves toward a state of minimal energy. The Hopfield dynamics defined in (9.2) on page 251 correspond exactly to this principle because it updates the state in each iteration such that, of the two states -1 and 1, the one with smaller total energy is taken on. The contribution of the neuron i to total energy is

$$-\frac{1}{2}x_i \sum_{j\neq i}^{n} w_{ij}x_j.$$

If now

$$\sum_{j\neq i}^{n} w_{ij}x_j < 0,$$

then $x_i = -1$ results in a negative contribution to the total energy, and $x_i = 1$ results in a positive contribution. For $x_i = -1$, the network takes on a state of lower energy than it does for $x_i = 1$. Analogously, we can assert that in the case of

$$\sum_{j\neq i}^{n} w_{ij}x_j \geq 0,$$

it must be true that $x_i = 1$.

If each individual iteration of the neural dynamics results in a reduction of the energy function, then the total energy of the system decreases monotonically with time. Because there are only finitely many states, the network moves in time to a state of minimal energy. Now we have the exciting question: what do these minima of the energy function mean?

As we saw in the pattern recognition experiment, in the case of few learned patterns the system converges to one of the learned patterns. The learned patterns represent minima of the energy function in the state space. If however too many patterns are learned, then the system converges to minima which do not correspond to learned patterns. Here we have a transition from an ordered dynamics into a chaotic one.

Hopfield and other physicists have investigated exactly this process and have shown that there is in fact a phase transition at a critical number of learned patterns. If the number of learned patterns exceeds this value, then the system changes from the ordered phase into the chaotic.

In magnetic physics there is such a transition from the ferromagnetic mode, in which all elementary magnets try to orient themselves parallel, to a so-called spin glass, in which the spins interact chaotically. A more visualizable example of such a physical phase transition is the melting of an ice crystal. The crystal is in a high state of order because the H_2O molecules are strictly ordered. In liquid water, by contrast, the structure of the molecules is dissolved and their positions are more random.

In a neural network there is then a phase transition from ordered learning and recognition of patterns to chaotic learning in the case of too many patterns, which can no longer be recognized for certain. Here we definitely see parallels to effects which we occasionally experienced ourselves.

We can understand this phase transition [RMS92] if we bring all neurons into a pattern state, for example $\boldsymbol{q}^1$, and insert the learned weights from (9.1) on page 250 into the term $\sum_{j=1,j\neq i}^{n} w_{ij}q_j$, which is relevant for updating neuron i. This results in

$$\sum_{\substack{j=1\\ j\neq i}}^{n} w_{ij} q_j^1 = \frac{1}{n}\sum_{\substack{j=1\\ j\neq i}}^{n}\sum_{k=1}^{N} q_i^k q_j^k q_j^1 = \frac{1}{n}\sum_{\substack{j=1\\ j\neq i}}^{n}\left(q_j^1 \left(q_j^1\right)^2 + \sum_{k=2}^{N} q_i^k q_j^k q_j^1\right)$$
$$= q_i^1 + \frac{1}{n}\sum_{\substack{j=1\\ j\neq i}}^{n}\sum_{k=2}^{N} q_i^k q_j^k q_j^1 .$$

Here we see the ith component of the input pattern plus a sum with $(n-1)(N-1)$ terms. If these summands are all statistically independent, then we can describe the sum by a normally distributed random variable with standard deviation

$$\frac{1}{n}\sqrt{(n-1)(N-1)} \approx \sqrt{\frac{N-1}{n}}.$$

Statistical independence can be achieved, for example, with uncorrelated random patterns. The sum then generates noise which is not disruptive as long as $N \ll n$, which means that the number of learned patterns stays much smaller than the number of neurons. If, however, $N \approx n$, then the influence of the noise becomes as large as the pattern and the network reacts chaotically. A more exact calculation of the phase transition gives $N = 0.146\, n$ as the critical point. Applied to our example, this means that for 100 neurons, up to 14 patterns can be saved. Since the patterns in the example however are strongly correlated, the critical value is much lower, evidently between 0.04 and 0.1. Even a value of 0.146 is much lower than the storage capacity of a traditional memory list (Exercise 9.3 on page 286).

Hopfield networks in the presented form only work well when patterns with roughly 50% 1-bits are learned. If the bits are very asymmetrically distributed, then the neurons must be equipped with a threshold [Roj96]. In physics this is analogous to the application of an outer magnetic field, which also brings about an asymmetry of the spin 1/2 and the spin −1/2 states.

9.2.3 Summary and Outlook

Through its biological plausibility, the well understood mathematical model, and above all through the impressive simulations in pattern recognition, the Hopfield model contributed to a wave of excitement about neural networks and to the rise of neuroinformatics as an important branch of AI.[2] Subsequently many further network models were developed. On one hand, networks without back-couplings were investigated because their dynamics is significantly easier to understand than recurrent Hopfield networks. On the other hand, attempts were made to improve the storage capacity of the networks, which we will go into in the next section.

[2]Even the author was taken up by this wave, which carried him from physics into AI in 1987.

A special problem of many neural models was already evident in the Hopfield model. Even if there is a guarantee of convergence, it is not certain whether the network will converge to a learned state or get stuck at a local minimum. The Boltzmann machine, with continuous activation values and a probabilistic update rule for its network dynamics, was developed as an attempt to solve this problem. Using a "temperature" parameter, we can vary the amount of random state changes and thus attempt to escape local minima, with the goal of finding a stable global minimum. This algorithm is called "simulated annealing". Annealing is a process of heat treating metals with the goal of making the metal stronger and more "stable".

The Hopfield model carries out a search for a minimum of the energy function in the space of activation values. It thereby finds the pattern saved in the weights, and which is thus represented in the energy function. The Hopfield dynamics can also be applied to other energy functions, as long as the weight matrix is symmetric and the diagonal elements are zero. This was successfully demonstrated by Hopfield and Tank on the traveling salesman problem [HT85, Zel94]. The task here is, given n cities and their distance matrix, to find the shortest round trip that visits each city exactly once.

9.3 Neural Associative Memory

A traditional list memory can in the simplest case be a text file in which strings of digits are saved line by line. If the file is sorted by line, then the search for an element can be done very quickly in logarithmic time, even for very large files.

List memory can also be used to create mappings, however. For example, a telephone book is a mapping from the set of all entered names to the set of all telephone numbers. This mapping is implemented as a simple table, typically saved in a database.

Access control to a building using facial recognition is a similar task. Here we could also use a database in which a photo of every person is saved together with the person's name and possibly other data. The camera at the entrance then takes a picture of the person and searches the database for an identical photo. If the photo is found, then the person is identified and gets access to the building. However, a building with such a control system would not get many visitors because the probability that the current photo matches the saved photo exactly is very small.

In this case it is not enough to just save the photo in a table. Rather, what we want is *associative memory*, which is capable of not only assigning the right name to the photo, but also to any of a potentially infinite set of "similar" photos. A function for finding similarity should be generated from a finite set of training data, namely the saved photos labeled with the names. A simple approach for this is the nearest neighbor method introduced in Sect. 8.3. During learning, all of the photos are simply saved.

To apply this function, the photo most similar to the current one must be found in the database. For a database with many high-resolution photos, this process, depending on the distance metric used, can require very long computation times and thus cannot be implemented in this simple form. Therefore, instead of such a lazy algorithm, we will prefer one which transfers the data into a function which then creates a very fast association when it is applied.

Finding a suitable distance metric presents a further problem. We would like a person to be recognized even if the person's face appears in another place on the photo (translation), or if it is smaller, larger, or even rotated. The viewing angle and lighting might also vary.

This is where neural networks show their strengths. Without requiring the developer to think about a suitable similarity metric, they still deliver good results. We will introduce two of the simplest associative memory models and begin with a model by Teuvo Kohonen, one of the pioneers in this area.

The Hopfield model presented in the previous chapter would be too difficult to use for two reasons. First, it is only an *auto-associative memory*, that is, an approximately identical mapping which maps similar objects to the learned original. Second, the complex recurrent dynamics is often difficult to manage in practice. Therefore we will now look at simple two-layer feedforward networks.

9.3.1 Correlation Matrix Memory

In [Koh72] Kohonen introduced an associative memory model based on elementary linear algebra. This maps query vectors $\boldsymbol{x} \in \mathbb{R}^n$ to result vectors $\boldsymbol{y} \in \mathbb{R}^m$. We are looking for a matrix $\boldsymbol{W}$ which correctly maps out of a set of training data

$$T = \{(\boldsymbol{q}^1, \boldsymbol{t}^1), \ldots, (\boldsymbol{q}^N, \boldsymbol{t}^N)\}$$

with N query-response pairs all query vectors to their responses.[3] That is, for $p = 1, \ldots, N$ it must be the case that

$$\boldsymbol{t}^p = \boldsymbol{W} \cdot \boldsymbol{q}^p, \tag{9.3}$$

or

$$t_i^p = \sum_{j=1}^{n} w_{ij} q_j^p. \tag{9.4}$$

[3]For a clear differentiation between training data and other values of a neuron, in the following discussion we will refer to the query vector as $\boldsymbol{q}$ and the desired response as $\boldsymbol{t}$ (target).

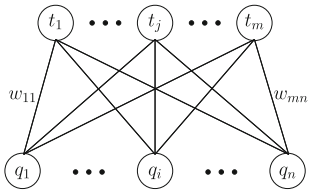

Fig. 9.9 Representation of the Kohonen associative memory as a two-layer neural network

To calculate the matrix elements w_{ij}, the rule

$$w_{ij} = \sum_{p=1}^{N} q_j^p t_i^p \tag{9.5}$$

is used. These two linear equations can be simply understood as a neural network if we define, as in Fig. 9.9, a two-layer network with $\boldsymbol{q}$ as the input layer and $\boldsymbol{t}$ as the output layer. The neurons of the output layer have a linear activation function, and (9.5) is used as the learning rule, which corresponds exactly to the Hebb rule.

Before we show that the network recognizes the training data, we need the following definition:

Definition 9.1 Two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are called *orthonormal* if

$$\boldsymbol{x}^T \cdot \boldsymbol{y} = \begin{cases} 1 & \text{if } \boldsymbol{x} = \boldsymbol{y}, \\ 0 & \text{else.} \end{cases}$$

Thus

Theorem 9.1 *If all N query vectors $\boldsymbol{q}^p$ in the training data are orthonormal, then every vector $\boldsymbol{q}^p$ is mapped to the target vector $\boldsymbol{t}^p$ by multiplication with the matrix $\boldsymbol{W}$ from* (9.5).

Proof: We substitute (9.5) into (9.4) on page 257 and obtain

$$(\boldsymbol{W} \cdot \boldsymbol{q}^p)_i = \sum_{j=1}^{n} w_{ij} q_j^p = \sum_{j=1}^{n} \sum_{r=1}^{N} q_j^r t_i^r q_j^p = \sum_{j=1}^{n} \left(q_j^p q_j^p t_i^p + \sum_{\substack{r=1 \\ r \neq p}}^{N} q_j^r q_j^p t_i^r \right)$$

$$= t_i^p \underbrace{\sum_{j=1}^{n} q_j^p q_j^p}_{=1} + \sum_{\substack{r=1 \\ r \neq p}}^{N} t_i^r \underbrace{\sum_{j=1}^{n} q_j^r q_j^p}_{=0} = t_i^p$$

□

Thus, if the query vectors are orthonormal, all patterns will be correctly mapped to the respective targets. However, orthonormality is too strong a restriction. In Sect. 9.4 we will present an approach that overcomes this limitation.

Since linear mappings are continuous and injective, we know that the mapping from query vectors to target vectors preserves similarity. Similar queries are thus mapped to similar targets due to the continuity. At the same time we know, however, that different queries are mapped to different targets. If the network was trained to map faces to names and if the name Henry is assigned to a face, then we are sure that for the input of a similar face, an output similar to "Henry" will be produced, but "Henry" itself is guaranteed not to be calculated. If the output can be interpreted as a string, then, for example, it could be "Genry" or "Gfnry". To arrive at the most similar learned case, Kohonen uses a binary coding for the output neuron. The calculated result of a query is rounded if its value is not zero or one. Even then we have no guarantee that we will hit the target vector. Alternatively, we could add a subsequent mapping of the calculated answer to the learned target vector with the smallest distance.

9.3.2 The Binary Hebb Rule

In the context of associative memory, the so-called binary Hebb rule was suggested. It requires that the pattern is binary-encoded. This means that for all patterns $\boldsymbol{q}^p \in \{0, 1\}^n$ and $\boldsymbol{t}^p \in \{0, 1\}^m$. Furthermore, the summation from (9.5) on page 258 is replaced by a simple logical OR and we obtain the binary Hebb rule

$$w_{ij} = \bigvee_{p=1}^{N} q_j^p t_i^p. \tag{9.10}$$

The weight matrix is thus also binary, and a matrix element w_{ij} is equal to one if and only if at least one of the entries $q_j^1 t_i^1, \ldots, q_j^N t_i^N$ is not zero. All other matrix elements are zero. We are tempted to believe that a lot of information is lost here during learning because, when a matrix element takes on the value 1 once, it cannot be changed by additional patterns. Figure 9.10 on page 260 shows how the matrix is filled with ones for an example with $n = 10$, $m = 6$ after learning three pairs.

To retrieve the saved patterns we simply multiply a query vector $\boldsymbol{q}$ by the matrix and look at the result $\boldsymbol{Wq}$. We test this on the example and get Fig. 9.11 on page 260.

We see that in the target vector on the right side there is the value 3 in the place where the learned target vector had a one. The correct results would be obtained by setting a threshold value of 3. In the general case we choose the number of ones in the query vector as the threshold. Each output neuron thus works like a perceptron, albeit with a variable threshold.

As long as the weight matrix is sparse, this algorithm performs well. However, if many different patterns are saved, the matrix becomes more and more dense. In the

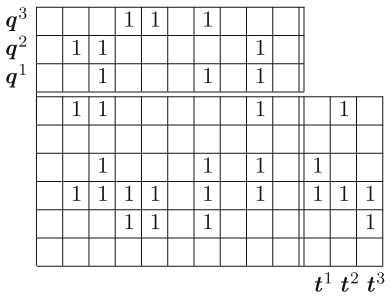

Fig. 9.10 The matrix $\boldsymbol{W}$ after saving three pairs $(\boldsymbol{q}^1, \boldsymbol{t}^1)$, $(\boldsymbol{q}^2, \boldsymbol{t}^2)$, $(\boldsymbol{q}^3, \boldsymbol{t}^3)$. Empty fields correspond to the value 0

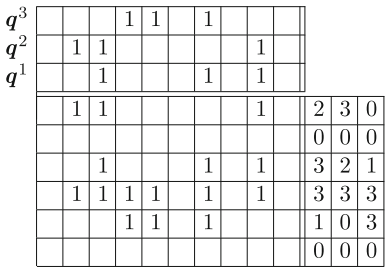

Fig. 9.11 Calculation of the product $\boldsymbol{W}\boldsymbol{q}^1$, $\boldsymbol{W}\boldsymbol{q}^2$, $\boldsymbol{W}\boldsymbol{q}^3$

extreme case it contains only ones. Then, after setting the threshold, all answers would consist of just ones and would no longer contain any information.

This case rarely occurs as long as the number of bits saved in the matrix does not become too large. The matrix has a size of $m\ n$ elements. A pair to be saved has $m + n$ bits. One can show [Pal80] that the number of memorizable patterns N_{max} is determined by the following condition:

$$\begin{aligned} \alpha &= \frac{\text{number of storable bits}}{\text{number of binary matrix elements}} \\ &= \frac{(m+n)N_{max}}{mn} \leq \ln 2 \approx 0.69. \end{aligned} \tag{9.11}$$

For a list memory we have $\alpha = 1$. Associative memory with the binary Hebb rule has a maximum memory efficiency of $\alpha = 0.69$ compared to $\alpha = 0.72$ for Kohonen associative memory and $\alpha = 0.292$ for Hopfield networks [Pal80, Pal91]. The memory capacity of the binary Hebb rule is thus surprisingly high in comparison to the Kohonen model with continuous neurons.

It is obvious that such memory becomes "full" less quickly when the query and target vectors are sparsely populated with ones. This is not the only reason why the encoding of input and output for associative memories—as also for other neural networks—is very important for good performance. We will now demonstrate this on an application of this memory with suitably chosen encodings of input and output.

9.3.3 A Spelling Correction Program

As an application of the described associative memory with the binary Hebb rule, we choose a program that corrects erroneous inputs and maps them to saved words from a dictionary. Clearly an auto-associative memory would be needed here. However, because we encode the query and target vectors differently, this is not the case. For the query vectors $\boldsymbol{q}$ we choose a pair encoding. For an alphabet with 26 characters there are $26 \cdot 26 = 676$ ordered pairs of letters. With 676 bits, the query vector has one bit for each of the possible pairs

$$\text{aa}, \text{ab}, \ldots, \text{az}, \text{ba}, \ldots, \text{bz}, \ldots, \text{za}, \ldots, \text{zz}.$$

If a pair of letters occurs in the word, then a one will be entered in the appropriate place. For the word "hans", for instance, the slots for "ha", "an", and "ns" are filled with ones. For the target vector $\boldsymbol{t}$, 26 bits are reserved for each position in the word up to a maximum length (for example ten characters). For the ith letter in the alphabet in position j in the word then the bit number $(j-1) \cdot 26 + i$ is set. For the word "hans", bits 8, 27, 66, and 97 are set. For a maximum of 10 letters per word, the target vector thus has a length of 260 bits.

The weight matrix $\boldsymbol{W}$ thus has a size of $676 \cdot 260$ bits $= 199420$ bits, which by (9.11) on page 260 can store at most

$$N_{max} \leq 0.69 \frac{m\,n}{m+n} = 0.69 \frac{676 \cdot 260}{676+260} \approx 130$$

words. With 72 first names, we save about half that many and test the system. The stored names and the output of the program for several example inputs are given in Fig. 9.12 on page 262. The threshold is always initialized to the number of bits in the encoded query. Here this is the number of letter pairs, thus the word length minus one. Then it is stepwise reduced to two. We could further automate the choice of the threshold by comparing with the dictionary for each attempted threshold and output the word found when the comparison succeeds.

The reaction to the ambiguous inputs "andr" and "johanne" is interesting. In both cases, the network creates a mix of two saved words that fit. We see here an important strength of neural networks. They are capable of making associations to similar objects without an explicit similarity metric. However, similarly to heuristic search and human decision making, there is no guarantee for a "correct" solution.

Since the training data must be available in the form of input-output pairs for all neural models which have been introduced so far, we are dealing with supervised learning, which is also the case for the networks introduced in the following sections.

Stored words:

agathe, agnes, alexander, andreas, andree, anna, annemarie, astrid, august, bernhard, bjorn, cathrin, christian, christoph, corinna, corrado, dieter, elisabeth, elvira, erdmut, ernst, evelyn, fabrizio, frank, franz, geoffrey, georg, gerhard, hannelore, harry, herbert, ingilt, irmgard, jan, johannes, johnny, juergen, karin, klaus, ludwig, luise, manfred, maria, mark, markus, marleen, martin, matthias, norbert, otto, patricia, peter, phillip, quit, reinhold, renate, robert, robin, sabine, sebastian, stefan, stephan, sylvie, ulrich, ulrike, ute, uwe, werner, wolfgang, xavier

Associations of the program:

```
input pattern: harry

threshold: 4, answer: harry
threshold: 3, answer: harry
threshold: 2, answer: horryrrde

------------------------------
input pattern: ute

threshold: 2, answer: ute

------------------------------
input pattern: gerhar

threshold: 5, answer: gerhard
threshold: 4, answer: gerrarn
threshold: 3, answer: jerrhrrd
threshold: 2, answer: jurtyrrde

------------------------------
input pattern: egrhard

threshold: 6, answer:
threshold: 5, answer:
threshold: 4, answer: gerhard
threshold: 3, answer: gernhrrd
threshold: 2, answer: irrryrrde

------------------------------
input pattern: andr

threshold: 3, answer: andrees
threshold: 2, answer: anexenser
```

```
input pattern: andrees

threshold: 6, answer: a
threshold: 5, answer: andree
threshold: 4, answer: andrees
threshold: 3, answer: mnnrens
threshold: 2, answer: morxsnssr

------------------------------
input pattern: johanne

threshold: 6, answer: johnnnes
threshold: 5, answer: johnnnes
threshold: 4, answer: jornnnrse
threshold: 3, answer: sorrnyrse
threshold: 2, answer: wtrrsyrse

------------------------------
input pattern: johnnnes

threshold: 6, answer: joh
threshold: 5, answer: johnnnes
threshold: 4, answer: johnnyes
threshold: 3, answer: jonnnyes
threshold: 2, answer: jornsyrse

------------------------------
input pattern: johnnyes

threshold: 7, answer:
threshold: 6, answer: joh
threshold: 5, answer: johnny
threshold: 4, answer: johnnyes
threshold: 3, answer: johnnyes
threshold: 2, answer: jonnnyes
```

Fig. 9.12 Application of the spelling correction program to various learned or erroneous inputs. The correct inputs are found with the maximum (that is, the first attempted) threshold. For erroneous inputs the threshold must be lowered for a correct association

9.4 Linear Networks with Minimal Errors

The Hebb rule used in the neural models presented so far works with associations between neighboring neurons. In associative memory, this is exploited in order to learn a mapping from query vectors to targets. This works very well in many cases, especially when the query vectors are linearly independent. If this condition is not fulfilled, for example when too much training data is available, the question arises: how do we find the optimal weight matrix? Optimal means that it minimizes the average error.

We humans are capable of learning from mistakes. The Hebb rule does not offer this possibility. The backpropagation algorithm, described in the following, uses an elegant solution known from function approximation to change the weights such that the error on the training data is minimized.

Let N pairs of training vectors

$$T = \{(\boldsymbol{q}^1, \boldsymbol{t}^1), \ldots, (\boldsymbol{q}^N, \boldsymbol{t}^N)\}$$

be given with $\boldsymbol{q}^p \in [0, 1]^n \boldsymbol{t}^p \in [0, 1]^m$. We are looking for a function $f: [0, 1]^n \rightarrow [0, 1]^m$ which minimizes the squared error

$$\sum_{p=1}^{N} (f(\boldsymbol{q}^p) - \boldsymbol{t}^p)^2$$

on the data. Let us first assume that the data contains no contradictions. That is, there is no query vector in the training data which should be mapped to two different targets. In this case it is not difficult to find a function that minimizes the squared error. In fact, there exist infinitely many functions which make the error zero. We define the function

$$f(\boldsymbol{q}) = 0, \quad \text{if } \boldsymbol{q} \notin \{\boldsymbol{q}^1, \ldots, \boldsymbol{q}^N\}$$

and

$$f(\boldsymbol{q}^p) = \boldsymbol{t}^p \quad \forall p \in \{1, \ldots, N\}.$$

This is a function which even makes the error on the training data zero. What more could we want? Why are we not happy with this function?

The answer is: because we want to build an intelligent system! Intelligent means, among other things, that the learned function can generalize well from the training data to new, unknown data from the same representative data set. In other words it means: we do not want overfitting of the data by memorization. What is it then that we really want?

We want a function that is smooth and "evens out" the space between the points. Continuity and the ability to take multiple derivatives would be sensible requirements. Because even with these conditions there are still infinitely many functions which make the error zero, we must restrict this class of functions even further.

9.4.1 Least Squares Method

The simplest choice is a linear mapping. We begin with a two-layer network (Fig. 9.13) in which the single neuron y of the second layer calculates its activation using

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right)$$

with $f(x) = x$. The fact that we are only looking at output neurons here does not pose a real restriction because a two-layer network with two or more output neurons can always be separated into independent networks with identical input neurons for each of the original output neurons. The weights of the subnetworks are all independent. Using a sigmoid function instead of the linear activation does not give any advantage here because the sigmoid function is strictly monotonically increasing and does not change the order relation between various output values.

Desired is a vector $\boldsymbol{w}$ which minimizes the squared error

$$E(\boldsymbol{w}) = \sum_{p=1}^{N} (\boldsymbol{w}\,\boldsymbol{q}^p - t^p)^2 = \sum_{p=1}^{N} \left(\sum_{i=1}^{n} w_i {q_i}^p - t^p\right)^2.$$

As a necessary condition for a minimum of this error function all partial derivatives must be zero. Thus we require that for $j = 1, \ldots, n$:

$$\frac{\partial E}{\partial w_j} = 2 \sum_{p=1}^{N} \left(\sum_{i=1}^{n} w_i q_i^p - t^p\right) q_j^p = 0.$$

Multiplying this out yields

$$\sum_{p=1}^{N} \left(\sum_{i=1}^{n} w_i q_i^p q_j^p - t^p q_j^p\right) = 0,$$

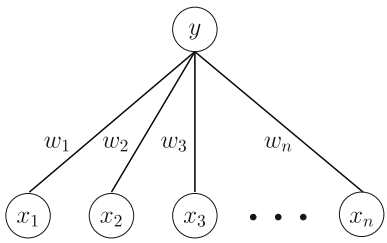

Fig. 9.13 A two-layer network with an output neuron

and exchanging the sums results in the linear system of equations

$$\sum_{i=1}^{n} w_i \sum_{p=1}^{N} q_i^p q_j^p = \sum_{p=1}^{N} t^p q_j^p,$$

which with

$$A_{ij} = \sum_{p=1}^{N} q_i^p q_j^p \quad \textit{and} \quad b_j = \sum_{p=1}^{N} t^p q_j^p \tag{9.12}$$

can be written as the matrix equation

$$\boldsymbol{A}\boldsymbol{w} = \boldsymbol{b}. \tag{9.13}$$

These so-called normal equations always have at least one solution and, when $\boldsymbol{A}$ is invertible, exactly one. Furthermore, the matrix A is positive-definite, which has the implication that the discovered solution in the unique case is a global minimum. This algorithm is known as least squares method.

The calculation time for setting up the matrix A grows with $\Theta(N \cdot n^2)$ and the time for solving the system of equations as $O(n^3)$. This method can be extended very simply to incorporate multiple output neurons because, as already mentioned, for two-layer feedforward networks the output neurons are independent of each other.

9.4.2 Application to the Appendicitis Data

As an application we now determine a linear score for the appendicitis diagnosis. From the LEXMED project data, which is familiar from Sect. 8.4.5, we use the least squares method to determine a linear mapping from symptoms to the continuous class variables *AppScore* with values in the interval [0, 1] and obtain the linear combination

$$\begin{aligned} \mathit{AppScore} = {} & 0.00085\,\mathit{Age} - 0.125\,\mathit{Sex} + 0.025\,\mathit{P1Q} + 0.035\,\mathit{P2Q} - 0.021\,\mathit{P3Q} \\ & - 0.025\,\mathit{P4Q} + 0.12\,\mathit{TensLoc} + 0.031\,\mathit{TensGlo} + 0.13\,\mathit{Losl} \\ & + 0.081\,\mathit{Conv} + 0.0034\,\mathit{RectS} + 0.0027\,\mathit{TAxi} + 0.0031\,\mathit{TRec} \\ & + 0.000021\,\mathit{Leuko} - 0.11\,\mathit{Diab} - 1.83. \end{aligned}$$

This function returns continuous variables for *AppScore*, although the actual binary class variable *App* only takes on the values 0 and 1. Thus we have to decide on a threshold value, as with the perceptron. The classification error of the score as a function of the threshold is listed in Fig. 9.14 on page 266 for the training data and the test data. We clearly see that both curves are nearly the same and have their minimum at $\Theta = 0.5$. In the small difference of the two curves we see that overfitting is not a problem for this method because the model generalizes from the test data very well.

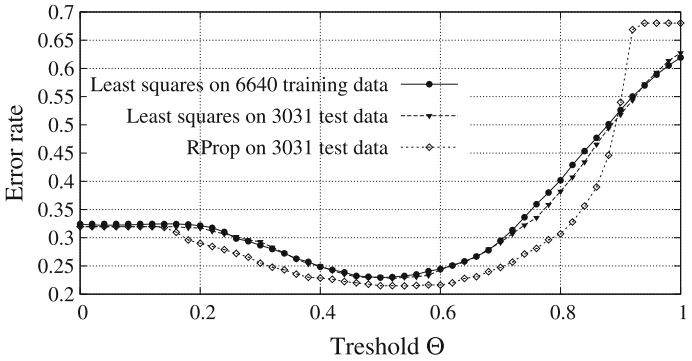

Fig. 9.14 Least squares error for training and test data

Also in the figure is the result for the nonlinear, three-layer RProp network (Sect. 9.5) with a somewhat lower error for threshold values between 0.2 and 0.9. For practical application of the derived score and the correct determination of the threshold Θ it is important to not only look at the error, but also to differentiate by type of error (namely false positive and false negative), as is done in the LEXMED application in Fig. 7.10 on page 156. In the ROC curve shown there, the score calculated here is also shown. We see that the simple linear model is clearly inferior to the LEXMED system. Evidently, linear approximations are not powerful enough for many complex applications.

9.4.3 The Delta Rule

Least squares is, like the perceptron and decision tree learning, a so-called *batch learning algorithm*, as opposed to *incremental learning*. In batch learning, all training data must be learned in one run. If new training data is added, it cannot simply be learned in addition to what is already there. The whole learning process must be repeated with the enlarged set. This problem is solved by incremental learning algorithms, which can adapt the learned model to each additional new example. In the algorithms we will look at in the following discussion, we will additively update the weights for each new training example by the rule

$$w_j = w_j + \Delta w_j.$$

To derive an incremental variant of the least squares method, we reconsider the above calculated n partial derivatives of the error function

$$\frac{\partial E}{\partial w_j} = 2 \sum_{p=1}^{N} \left(\sum_{i=1}^{n} w_i q_i^p - t^p \right) q_j^p.$$

The gradient

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right)$$

as a vector of all partial derivatives of the error function points in the direction of the strongest rise of the error function in the n-dimensional space of the weights. While searching for a minimum, we will therefore follow the direction of the negative gradient. As a formula for changing the weights we obtain

$$\Delta w_j = -\frac{\eta}{2} \frac{\partial E}{\partial w_j} = -\eta \sum_{p=1}^{N} \left(\sum_{i=1}^{n} w_i q_i^p - t^p \right) q_j^p ,$$

where the *learning rate* η is a freely selectable positive constant. A larger η speeds up convergence but at the same time raises the risk of oscillation around minima or flat valleys. Therefore, the optimal choice of η is not a simple task (see Fig. 9.15). A large η, for example $\eta = 1$, is often used to start with, and then slowly shrunk.

By replacing the activation

$$y^p = \sum_{i=1}^{n} w_i q_i^p$$

of the output neuron for applied training example $\boldsymbol{q}^p$, the formula is simplified and we obtain the *delta rule*

$$\Delta w_j = \eta \sum_{p=1}^{N} (t^p - y^p) q_j^p .$$

Thus for every training example the difference between the target $\boldsymbol{t}^p$ and the actual output $\boldsymbol{y}^p$ of the network is calculated for the given input $\boldsymbol{q}^p$. After summing over all patterns, the weights are then changed proportionally to the sum. This algorithm is shown in Fig. 9.16 on page 268.

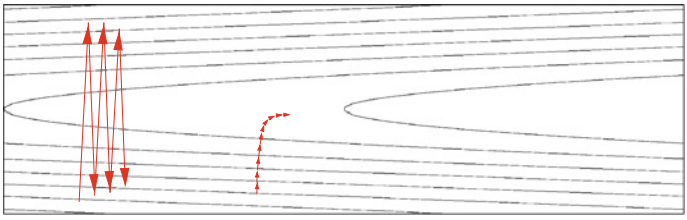

Fig. 9.15 Gradient descent for a large η (*left*) and a very small η (*right*) into a valley descending flatly to the *right*. For a large η there are oscillations around the valley. For a η which is too small, in contrast, convergence in the flat valley happens very slowly

DELTALEARNING(*TrainingExamples*, η)
Initialize all weights w_j randomly
Repeat
  $\Delta \boldsymbol{w} = \mathbf{0}$
  For all $(\boldsymbol{q}^p, t^p) \in$ *TrainingExamples*
    Calculate network output $y^p = \boldsymbol{w}^p \boldsymbol{q}^p$
    $\Delta \boldsymbol{w} = \Delta \boldsymbol{w} + \eta(t^p - y^p)\boldsymbol{q}^p$
  $\boldsymbol{w} = \boldsymbol{w} + \Delta \boldsymbol{w}$
Until $\boldsymbol{w}$ converges

Fig. 9.16 Learning a two-layer linear network with the delta rule. Notice that the weight changes always occur after all of the training data are applied

DELTALEARNINGINCREMENTAL(*TrainingExamples*, η)
Initialize all weights w_j randomly
Repeat
  For all $(\boldsymbol{q}^p, t^p) \in$ *TrainingExamples*
    Calculate network output $y^p = \boldsymbol{w}^p \boldsymbol{q}^p$
    $\boldsymbol{w} = \boldsymbol{w} + \eta(t^p - y^p)\boldsymbol{q}^p$
Until $\boldsymbol{w}$ converges

Fig. 9.17 Incremental variant of the delta rule

We see that the algorithm is still not really incremental because the weight changes only occur after all training examples have been applied once. We can correct this deficiency by directly changing the weights (incremental gradient descent) after every training example (Fig. 9.17), which, strictly speaking, is no longer a correct implementation of the delta rule.

9.4.4 Comparison to the Perceptron

The learning rule for perceptrons introduced in Sect. 8.2.1, the least squares method, and the delta rule can be used to generate linear functions from data. For perceptrons however, in contrast to the other methods, a classifier for linearly separable classes is learned through the threshold decision. The other two methods, however, generate a linear approximation to the data. As shown in Sect. 9.4.2, a classifier can be generated from the linear mapping, if desired, by application of a threshold function.

The perceptron and the delta rule are iterative algorithms for which the time until convergence depends heavily on the data. In the case of linearly separable data, an upper limit on the number of iteration steps can be found for the perceptron. For the delta rule, in contrast, there is only a guarantee of asymptotic convergence without limit [HKP91].

For least squares, learning consists of setting up and solving a linear system of equations for the weight vector. There is thus a hard limit on the computation time. Because of this, the least squares method is always preferable when incremental learning is not needed.

9.5 The Backpropagation Algorithm

With the backpropagation algorithm, we now introduce the most-used neural model. The reason for its widespread use its universal versatility for arbitrary approximation tasks. The algorithm originates directly from the incremental delta rule. In contrast to the delta rule, it applies a nonlinear sigmoid function on the weighted sum of the inputs as its activation function. Furthermore, a backpropagation network can have more than two layers of neurons. The algorithm became known through the article [RHR86] in the legendary PDP collection [RM86].

In Fig. 9.18 on page 270 a typical backpropagation network with an input layer, a hidden layer, and an output layer is shown. Since the current output value x_j^p of the output layer neuron is compared with the target output value t_j^p, these are drawn parallel to each other. Other than the input neurons, all neurons calculate their current value x_j by the rule

$$x_j = f\left(\sum_{i=1}^{n} w_{ji} x_i\right) \tag{9.14}$$

where n is the number of neurons in the previous layer. We use the sigmoid function

$$f(x) = \frac{1}{1+e^{-x}}.$$

Analogous to the incremental delta rule, the weights are changed proportional to the negative gradient of the quadratic error function summed over the output neurons

$$E_p(\boldsymbol{w}) = \frac{1}{2} \sum_{k \in output} (t_k^p - x_k^p)^2$$

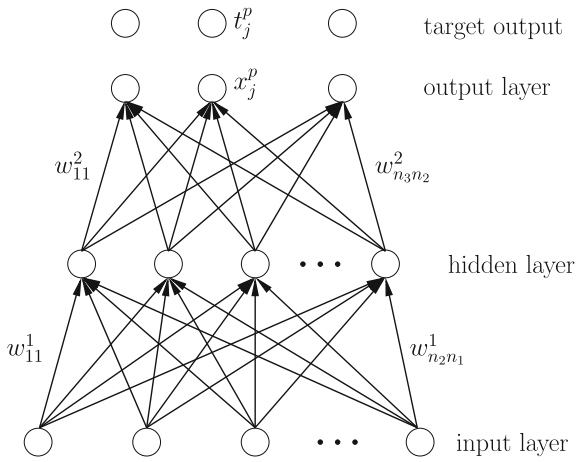

Fig. 9.18 A three-layer backpropagation network with n_1 neurons in the first, n_2 neurons in the second, and n_3 neurons in the third layer

for the training pattern p:

$$\Delta_p w_{ji} = -\eta \frac{\partial E_p}{\partial w_{ji}}.$$

For deriving the learning rule, the above expression is substituted for E_p. Within the expression, x_k is replaced by (9.14) on page 269. Within the equation, the outputs x_i of the neurons of the next, deeper layer occur recursively, etc. By multiple applications of the chain rule (see [RHR86] or [Zel94]) we obtain the backpropagation learning rule

$$\Delta_p w_{ji} = \eta \delta_j^p x_i^p,$$

with

$$\delta_j^p = \begin{cases} x_j^p(1 - x_j^p)(t_j^p - x_j^p) & \text{if } j \text{ is an output neuron,} \\ x_j^p(1 - x_j^p) \sum_k \delta_k^p w_{kj} & \text{if } j \text{ is a hidden neuron,} \end{cases}$$

which is also denoted the generalized delta rule. For all neurons, the formula for changing weight w_{ji} from neuron i to neuron j (see Fig. 9.19 on page 271) contains, like the Hebb rule, a term $\eta x_i^p x_j^p$. The new factor $(1 - x_j^p)$ creates the symmetry, which is missing from the Hebb rule, between the activations 0 and 1 of neuron j. For the output neurons, the factor $(t_j^p - x_j^p)$ takes care of a weight change proportional to the error. For the hidden neurons, the value δ_j^p of neuron j is calculated recursively from all changes δ_k^p of the neurons of the next higher level.

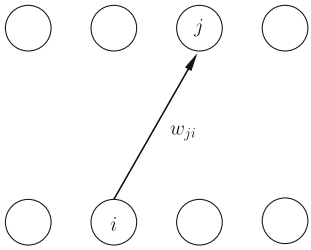

Fig. 9.19 Designation of the neurons and weights for the application of the backpropagation rule

The entire execution of the learning process is shown in Fig. 9.20. After calculating the output of the network (forward propagation) for a training example, the approximation error is calculated. This is then used during backward propagation to alter the weights backward from layer to layer. The whole process is then applied to all training examples and repeated until the weights no longer change or a time limit is reached.

If we build a network with at least one hidden layer, nonlinear mappings can be learned. Without hidden layers, the output neurons are no more powerful than a linear neuron, despite the sigmoid function. The reason for this is that the sigmoid function is strictly monotonic. The same is true for multi-layer networks which only use a linear function as an activation function, for example the identity function. This is because chained executions of linear mappings is linear in aggregate.

Just like with the perceptron, the class of the functions which can be represented are also enlarged if we use a variable sigmoid function

BACKPROPAGATION(*TrainingExamples*, η)
Initialize all weights w_j to random values
Repeat
  For all $(\boldsymbol{q}^p, \boldsymbol{t}^p) \in$ *TrainingExamples*
    1. **Apply the query vector** $\boldsymbol{q}^p$ to the input layer
    2. **Forward propagation**:
      For all layers from the first hidden layer upward
        For each neuron of the layer
          Calculate activation $x_j = f(\sum_{i=1}^{n} w_{ji} x_i)$
    3. **Calculation of the square error** $E_p(\boldsymbol{w})$
    4. **Backward propagation**:
      For all levels of weights from the last downward
        For each weight w_{ji}
          $w_{ji} = w_{ji} + \eta \delta_j^p x_i^p$
Until $\boldsymbol{w}$ converges or time limit is reached

Fig. 9.20 The backpropagation algorithm

$$f(x) = \frac{1}{1 + e^{-(x-\Theta)}}.$$

with threshold Θ. This is implemented analogously to the way shown in Sect. 8.2, in which a neuron whose activation always has the value one and which is connected to neurons in the next highest level is inserted into the input layer and into each hidden layer. The weights of these connections are learned normally and represent the threshold Θ of the successor neurons.

9.5.1 NETtalk: A Network Learns to Speak

Sejnowski and Rosenberg demonstrated very impressively in 1986 what backpropagation is capable of performing [SR86]. They built a system that is able to understandably read English text aloud from a text file. The architecture of the network shown in Fig. 9.21 consists of an input layer with $7 \times 29 = 203$ neurons in which the current letter and three previous letters, as well as three subsequent, letters are encoded. For each of these seven letters, 29 neurons are reserved for the characters "a . . . z␣, .". The input is mapped onto the 26 output neurons over 80 hidden neurons, each of which stands for a specific phoneme. For example, the "a" in "father" is pronounced deep, accented, and central. The network was trained with 1,000 words, which were applied randomly one after another letter by letter. For each letter, the target output was manually given for its intonation. To translate the intonation attributes into actual sounds, part of the speech synthesis system

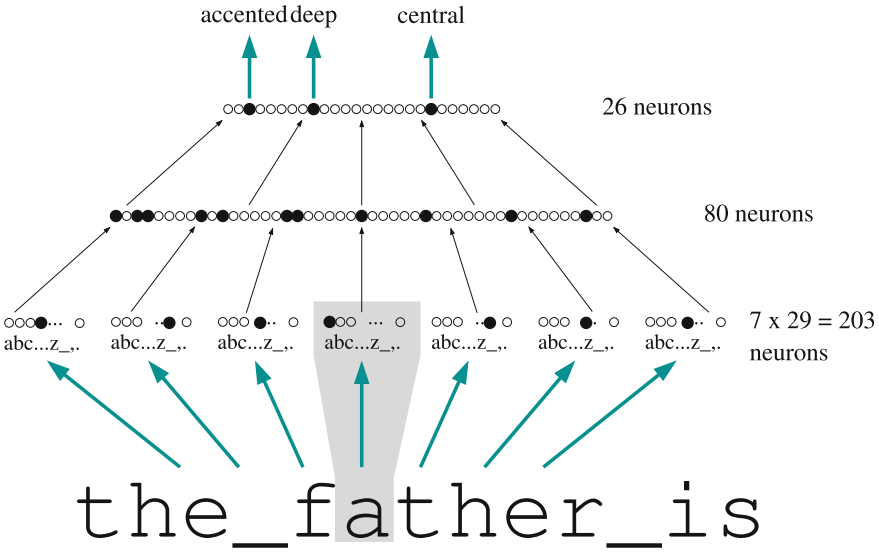

Fig. 9.21 The NETtalk network maps a text to its pronunciation attributes

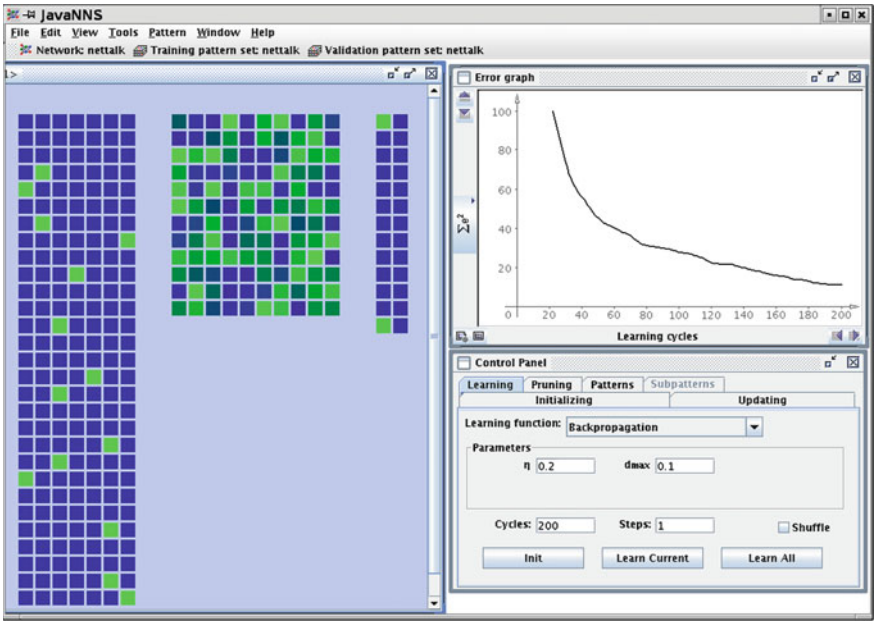

Fig. 9.22 The NETtalk network in JNNS. In the *left window* from *left* to *right* we see the $7 \cdot 29$ input neurons, 80 hidden, and 26 output neurons. Due to their large number, the weights are omitted from the network. *Top right* is a learning curve showing the development of the squared error over time

DECtalk was used. Through complete interconnectivity, the network contains a total of $203 \times 80 + 80 \times 26 = 18320$ weights.

The system was trained using a simulator on a VAX 780 with about 50 cycles over all words. Thus, at about 5 characters per word on average, about $5 \cdot 50 \cdot 1000 = 250\,000$ iterations of the backpropagation algorithm were needed. At a rate of about one character per second, this means roughly 69 hours of computation time. The developers observed many properties of the system which are quite similar to human learning. At first the system can only speak unclearly or use simple words. With time it continued to improve and finally reached 95% correctness of pronounced letters.

For visual experiments with neural networks, the Java Neural Network Simulator JNNS is recommended [Zel94]. The NETtalk network, loaded and trained with JNNS, is shown in Fig. 9.22.

9.5.2 Learning of Heuristics for Theorem Provers

In Chap. 6, algorithms for heuristic search, such as the A*-algorithm and the IDA*-algorithm were discussed. To reach a significant reduction of the search space, a good heuristic is needed for the implementation of these algorithms. In

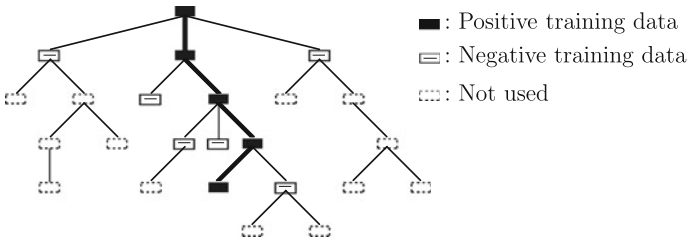

Fig. 9.23 Search tree with a successful path whose nodes are evaluated as positive, and the negatively rated unsuccessful branches

Sect. 4.1 the problem of the exploding search space during theorem provers' search for a proof was demonstrated. This problem is caused by the large number of possible inference steps in each step.

We now attempt to build heuristic proof controlling modules which evaluate the various alternatives for the next step and then choose the alternative with the best rating. In the case of resolution, the rating of the available clauses could be done by a function which, based on certain attributes of clauses such as the number of literals, the complexity of the terms, etc., calculates a value for each pair of resolvable clauses. In [ESS89, SE90] such a heuristic was learned for the theorem prover SETHEO using backpropagation. While learning the heuristic, the attributes of the current clause were saved as training data for the positive class for every proof found at every branch during the search. At all branches which did not lead to a proof, the attributes of the current clause were saved as training data for the negative class. Such a tree, with a successful path and the corresponding node ratings, is sketched in Fig. 9.23.

A backpropagation network was trained on this data and then used to evaluate clauses in the prover. For each clause, 16 numeric attributes were calculated, normalized, and encoded in an input neuron. The network was trained with 25 hidden neurons and one output neuron for the class (positive/negative).

It was shown that the number of attempted inferences can be reduced by many orders of magnitude for difficult problems, which ultimately reduces the computation time from hours to seconds. Thereby it became possible to prove theorems which, without heuristics, were out of reach.

9.5.3 Problems and Improvements

Backpropagation is now 25 years old and has proved itself in various applications, for example in pattern recognition and in robotics. However, its application is sometimes problematic. Especially when the network has many thousands of weights and there is a lot of training data to learn, two problems come up:

The network often converges to local minima of the error function. Furthermore, backpropagation often converges very slowly. This means that many iterations over

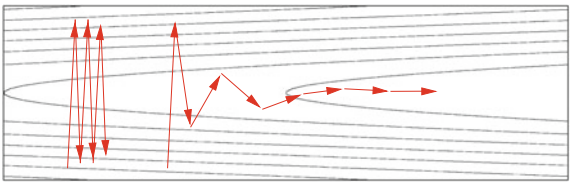

Fig. 9.24 Abrupt direction changes are smoothed out by the use of the momentum term. An iteration without the momentum term (*left*) in comparison to iteration with the momentum term (*right*)

all training patterns are needed. Many improvements have been suggested to alleviate these problems. As mentioned in Sect. 9.4.3, oscillations can be avoided by slowly reducing the learning rate η as shown in Fig. 9.15 on page 267.

Another method for reducing oscillations is the use of a momentum term while updating the weights, which ensures that the direction of gradient descent does not change too dramatically from one step to the next. Here for the current weight change $\Delta_p w_{ji}\ (t)$ at time t another part of the change $\Delta_p w_{ji}\ (t-1)$ from the previous step is added. The learning rule then changes to

$$\Delta_p w_{ji}(t) = \eta \delta_j^p x_i^p + \gamma \Delta_p w_{ji}(t-1)$$

with a parameter γ between zero and one. This is depicted in a two-dimensional example in Fig. 9.24.

Another idea is to minimize the linear error function instead of the square error function, which reduces the problem of slow convergence into flat valleys.

Gradient descent in backpropagation is ultimately based on a linear approximation of the error function. Quickprop, an algorithm which uses a quadratic approximation to the error function and thus achieves faster convergence, was created by Scott Fahlmann.

Through smart unification of the improvements mentioned and other heuristic tricks, Martin Riedmiller achieved a further optimization with the RProp algorithm [RB93]. RProp has replaced backpropagation and is the new state of the art feedforward neural network approximation algorithm. In Sect. 8.9 we applied RProp to the classification of the appendicitis data and achieved an error which is approximately the same size as that of a learned decision tree.

9.6 Support Vector Machines

Feedforward neural networks with only one layer of weights are linear. Linearity leads to simple networks and fast learning with guaranteed convergence. Furthermore, the danger of overfitting is small for linear models. For many applications, however, the linear models are not strong enough, for example because the relevant classes are not linearly separable. Here multi-layered networks such as backpropagation come into use, with the consequence that local minima, convergence problems, and overfitting can occur.

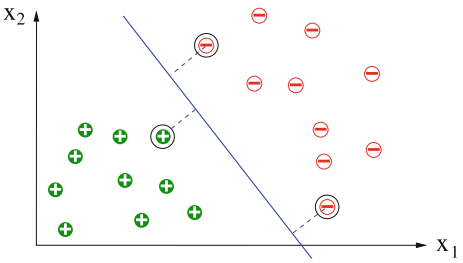

Fig. 9.25 Two classes with the maximally dividing line. The *circled points* are the support vectors

A promising approach, which brings together the advantages of linear and nonlinear models, follows the theory of support vector machines (SVM), which we will roughly outline using a two class problem.[4]

In the case of two linearly separable classes, it is easy to find a dividing hyper plane, for example with the perceptron learning rule. However, there are usually infinitely many such planes, as in the two-dimensional example in Fig. 9.25. We are looking for a plane which has the largest minimum distance to both classes. This plane is usually uniquely defined by a few points in the border area. These points, the so-called *support vectors*, all have the same distance to the dividing line. To find the support vectors, there is an efficient optimizing algorithm. It is interesting that the optimal dividing hyperplane is determined by a few parameters, namely by the support vectors. Thus the danger of overfitting is small.

Support vector machines apply this algorithm to non linearly separable problems in a two-step process: In the first step, a nonlinear transformation is applied to the data, with the property that the transformed data is linearly separable. In the second step the support vectors are then determined in the transformed space.

The first step is highly interesting, but not quite simple. In fact, it is always possible to make the classes linearly separable by transforming the vector space, as long as the data contains no contradictions.[5] Such a separation can be reached for example by introducing a new $(n + 1)$th dimension and the definition

$$x_{n+1} = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \text{class } 1, \\ 0 & \text{if } \boldsymbol{x} \in \text{class } 0. \end{cases}$$

However, this formula does not help much because it is not applicable to new points of an unknown class which are to be classified. We thus need a general transformation which is as independent as possible from the current data. It can be shown that there are such generic transformations even for arbitrarily shaped class division boundaries in the original vector space. In the transformed space, the data are then linearly separable. However, the number of dimensions of the new vector space grows exponentially with the number of dimensions of the original vector

[4]Support vector machines are not neural networks. Due to their historical development and mathematical relationship to linear networks, however, they are discussed here.

[5]A data point is contradictory if it belongs to both classes.

space. However, the large number of new dimensions is not so problematic because, when using support vectors, the dividing plane, as mentioned above, is determined by only a few parameters.

The central nonlinear transformation of the vector space is called the *kernel*, because of which support vector machines are also known as kernel methods. The original SVM theory developed for classification tasks has been extended and can now be used on regression problems also.

The mathematics used here is very interesting, but too extensive for an initial introduction. To delve deeper into this promising young branch of machine learning, we refer the reader to [SS02, Alp04] and [Bur98].

9.7 Deep Learning

In Chaps. 8 and 9 we saw that there are many good learning algorithms today which are capable of learning non-trivial, sometimes complex classifications or approximations for all sorts of applications, such as diagnosis and prognosis based on sensor inputs. We have also seen that, up to now, generation of features has not been successful. Instead it is the job of a data scientist to find a sensible small set of features, which can then be used as input for the learning algorithm.

Why do we not simply use all available sensor data, i.e. a direct image of the world, as input? For example, for object recognition in a photo we could use all ten million pixels as an input vector, which has a length of 30 million (in the case of RGB or HSV images, which each have three color values). What is the problem with this approach? The answer is known as the *"curse of the dimensionality"*. This means, among other things, that training time grows very fast, often exponentially, with the dimension of the input data. To keep computation times within bounds, we must therefore first reduce the input data to short feature vectors. As described above, this mapping is usually created manually. However, for many applications, such as object classification in images it is difficult if not impossible to manually find the formula for features. One old method for automatic reduction of dimensions is principal component analysis (PCA) [Ert15], which determines the directions of highest variance (i.e. the principal component) in the vector space of the training data and projects the data into the subspace of the principal components using a linear transformation. Because of the missing nonlearity of the compression mapping PCA is not as powerful as the new methods described below.

Since about 1995 work has been done on deep learning, a highly promising class of algorithms to solve this problem, and there are now impressive successes to report. Deep learning includes methods such as convolutional neural networks (CNNs), or deep belief networks and variations thereof. The architectures of the multi-layered neural networks with up to twenty or more layers are in part very complex and cannot be explained in detail here. [LBH15] is a good review article, and a very detailed introduction can be found in [GBC16], as well as in `deeplearning.stanford.edu`. Now let us try to understand the most important principles.

Pattern recognition is simple in low-dimensional spaces or in case of classification when the classes are linearly separable. For classes that are not linearly separable in high dimensional spaces, however, problems arise because here learning poses a nonlinear optimization problem. In principle there are solutions using gradient descent algorithms such as backpropagation. However, convergence problems and unacceptably high computation times arise for the classical algorithms, especially when networks with many hidden layers are used. Thus other methods have been sought.

9.7.1 Nature as Example

All successful approaches in deep learning to date work with many layers of neurons. The network is split into two parts, similar to the feedforward network shown schematically in Fig. 9.26. After a preprocessing layer there are several layers which are pre-trained by unsupervised learning (UL). Each layer in this UL network represents features of the input pattern. The lower the layer, the simpler the features. In object recognition in photos, the features of the lower layers typically represent edges or lines in different orientations.[6] Complex features such as the

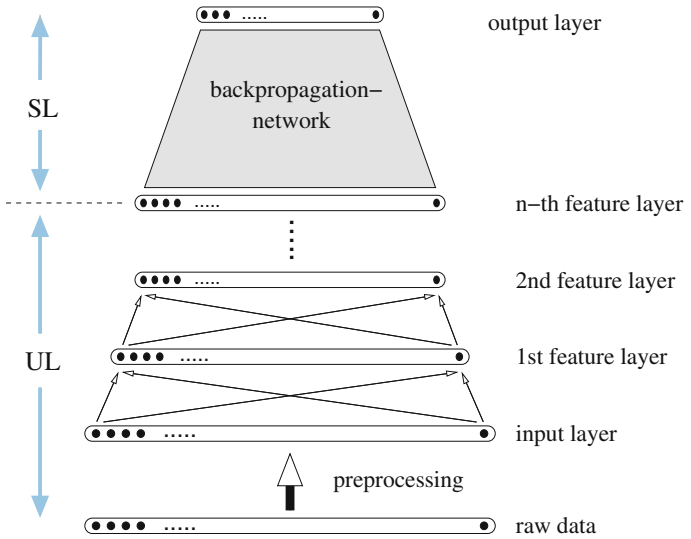

Fig. 9.26 Simplified architecture of a deep learning network. It is composed of a preprocessor, several layers (in this case two) for unsupervised feature detection and then one classical neural network, which, in this example, is trained by backpropagation

[6]Visual representations of such edge features, including explanation, can be found at `deeplearning.stanford.edu`.

presence of a face can form on higher layers. This architecture shows certain similarities to the structure of the brains of humans and animals. Starting from the sense organs, for example the eyes, the brain is built up in many layers, and the higher the layer, the more abstract the information to be found there. However, there is still far too little known about how neural networks work in nature that leads to a significant benefit in deep learning [GBC16].

Attached to the UL network is a classical supervised learning (SL) network, which can be trained with backpropagation or RProp. Thus the learning process works as follows:

1. Unsupervised training of all feature layer weights.
2. Supervised training of the SL network with gradient descent.

The unsupervised learning of the weights to the feature layers remains to be explained. By learning these weights, the features will be extracted. Feature extraction should have the property that input data is mapped into a lower dimensional space, if possible without (too much) loss of information. One could therefore see feature extraction as a form of compression. We now outline one of the many possible algorithms for feature learning.

9.7.2 Stacked Denoising Autoencoder

The feature layer's weights are determined by a fundamentally simple and obvious algorithm called stacked denoising autoencoder [VLL+10], as shown in Fig. 9.27.

To train the first hidden layer, an autoencoder is trained with a supervised learning method, for example RProp. Its job is to learn the identity mapping of all input vectors $\boldsymbol{x}$ onto itself. The feature layer acts as the hidden layer. To make sure that features form in this layer, it is important to avoid overfitting. The classical approach would be to use cross-validation over the number of hidden neurons in the feature layer. That would lead to a small number of hidden neurons and thus few features. It has been shown that such a compressed encoding of the features is not optimal. So-called sparse coding leads to better results.

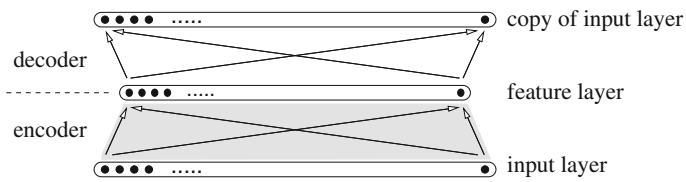

Fig. 9.27 Autoencoder for learning the weights of a feature layer. A classical backpropagation network can be used here to learn the identity mapping from the current input layer onto itself. Features form in the hidden layer during learning

In order to find many different features via such a sparse coding, *denoising*, rather than cross-validation, will be used. During each learning cycle, the values of some of the input neurons are randomly changed, for example with Gaussian noise or by setting the activation to 0 or 1. The distance

$$||\boldsymbol{y} - \boldsymbol{x}||$$

of the calculated output vector $\boldsymbol{y}$ from the original input vector $\boldsymbol{x}$ is used as the error function for the weight change. Thus the encoder is trained to become robust against noise. During learning, a variable parameter weights the noisified neurons more strongly than the others, so as to optimize the desired effect of the noise on learning.

After the autoencoder is trained, the actually unneeded decoder layer, i.e. the second layer of weights, is removed. The first layer is frozen and used to calculate the features in the UL network in Fig. 9.26 on page 278. Then, with this fixed first layer of weights, the second feature layer is trained with the autoencoder algorithm, frozen, and so on until the last feature layer. Thus the unsupervised part of the learning is finished.

Now the SL portion of the network is trained with a classical supervised learning algorithm. For each training example, the output of of the UL network, i.e., the last feature layer is used as input, and a particular data label is the target output. During backpropagation, the weights of the feature layers can be trained again for the purpose of fine-tuning, or they can be left unchanged.

We have not yet mentioned the preprocessing step, which differs depending on the application. Often all of the input variables are normalized. For photos, the pixel image is often transformed into a lower dimensional space, similar to PCA, using a process called whitening, with the goal of making the generated features less redundant. Here one could also directly use PCA.

9.7.3 Other Methods

In addition to the stacked denoising autoencoder, the previously mentioned convolutional neural networks play an important role. To reduce complexity and save time, the feature layers are not fully connected, rather each feature neuron retains input from only a few neurons in the layer below. The layers also alternate between convolution and pooling layers. In each convolution layer, a trainable linear filter is used on the input neurons, and in the pooling layer, an average, maximum or a more complex function is calculated from the input neurons.

Also quite popular are deep belief networks, which use restricted Boltzmann machines for learning [HOT06].

Discovery of features with UL networks can be completely replaced by clustering, in which, for every discovered cluster, a binary feature determines whether a point belongs to that particular cluster. One can also use kernel PCA, a nonlinear generalization of PCA, to learn features. As mentioned earlier, the fully connected SL network can also be replaced by other learning algorithms, for instance by a support vector machine.

9.7.4 Systems and Implementations

Even the best current implementations of deep learning systems are very computationally intensive. The cause of the long computation times is the size of the input layer and the high number of layers in the network. This effect is amplified further if, in the case of a large input layer, the training data are elements of a high-dimensional vector space. To represent the trained classes well, a great many data vectors are needed, which makes the computation time even longer.

This means that a training run can take from minutes up to even days. In addition, the system parameters of the complex networks have to be configured, which, in turn, has a big impact on the quality of the results. As we know from Chap. 8, optimal metaparameters for a learning algorithm can be found with cross-validation, that is by trying out all combinations of values. Due to the complexity of deep learning algorithms, there are many parameters, and the set of parameter combinations grows exponentially with the number of parameters. Therefore, the naive application of cross-validation is not practical. Instead, algorithms are used which look for a point in the space of the system's metaparameters that minimizes errors on the validation data [BBBK11]. Examples of such metaparameters are the number of layers in the UL network as well as the SL network, the number of neurons in the individual layers, the learning rates, and the degree of interconnectedness for CNN networks. The algorithms for optimization of metaparameters use, for example, random search or gradient descent in the space of metaparameters [MDA15].

Thus, for high-dimensional data sets, simple PCs are overwhelmed. Today one works with multiprocessor machines and the learning algorithms are run highly parallel on modern graphics cards, which ultimately leads to training times of hours to weeks for training with hyper parameter optimization.

Thirty-eight different freely available software systems are listed at http://deeplearning.net/software_links.[7] *Theano*, with special support for graphics cards, and the systems *Pylearn2* und *Keras*, which are based on Theano, are especially interesting. Each of these are programmed in the Python programming language. *Tensorflow* from Google Deep Brain also uses Python. The only additional requirement for a successful project is a suitably fast machine, which can be rented from cloud service providers as needed.

9.7.5 Applications of Deep Learning

Especially in object classification in photos, deep learning has shown impressive progress. In [Ben16] there is a collection of results for various data sets. For example, in the recognition of hand-written digits, the classification accuracy of 99.8% has been achieved on the MNIST data set presented in [LBBH98] (Fig. 9.28 on page 282 links). On the SVHN data set (Fig. 9.28 on page 282 center) generated

[7] Accessed April, 2016.

Fig. 9.28 Examples of MNIST data (left), SVHN data (center) and an example photo for the image description system

as part of the Google Street View Project with photos of home address numbers [NWC+11], the accuracy is at about 98.3%. It is even possible to describe photos in complete sentences [VTBE15]. For the photo shown in the right of Fig. 9.28, this system produces the sentence *"A person riding a motorcycle on a dirt road."* It uses a convolutional neural network for object recognition and a recurrent neural network for text generation.

In addition to many other applications, there are impressive variants of deep learning for creatively generating works of art.

9.8 Creativity

With deep learning, programs have emerged that can, for example, compose jazz melodies [BBSK10]. A recurrent neural network is used in [Kar15] to generate texts, syntactically correct XML code, even LaTeX source with mathematical notation and short computer programs. Hence recurrent neural networks can learn grammar. A program trained on Shakespeare's body of work can, from scratch, generate new texts such as:

```
KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.
```

Wikipedia describes creativity as a phenomenon in which something new and somehow valuable is formed. It is easy to appreciate this definition in light of the above text. Let us imagine a program that randomly generates text letter by letter. The result might begin something like

```
NitDjgMQsQfI 6zz1B:6xZkgp1xe.EqyD7z(C
```

While the text is new, it is not very valuable and therefore not a creative artifact. On the other hand, a program that queries and outputs text from a database does not

count as creative either because it produces nothing new. A creative program, for example, should write a novel that is both interesting and new at the same time. One could easily check the text's novely, for example by using software for testing the originality of student thesis papers. Interestingness is more difficult. It would be good if a program could generate new texts that a person would enjoy based on the ratings which that person has given to previously read literature.

Somewhat more formally, we could define an adaptive creative program as one that takes some objects from a given distribution of training data, then approximates the distribution and randomly creates a new object from this distribution.

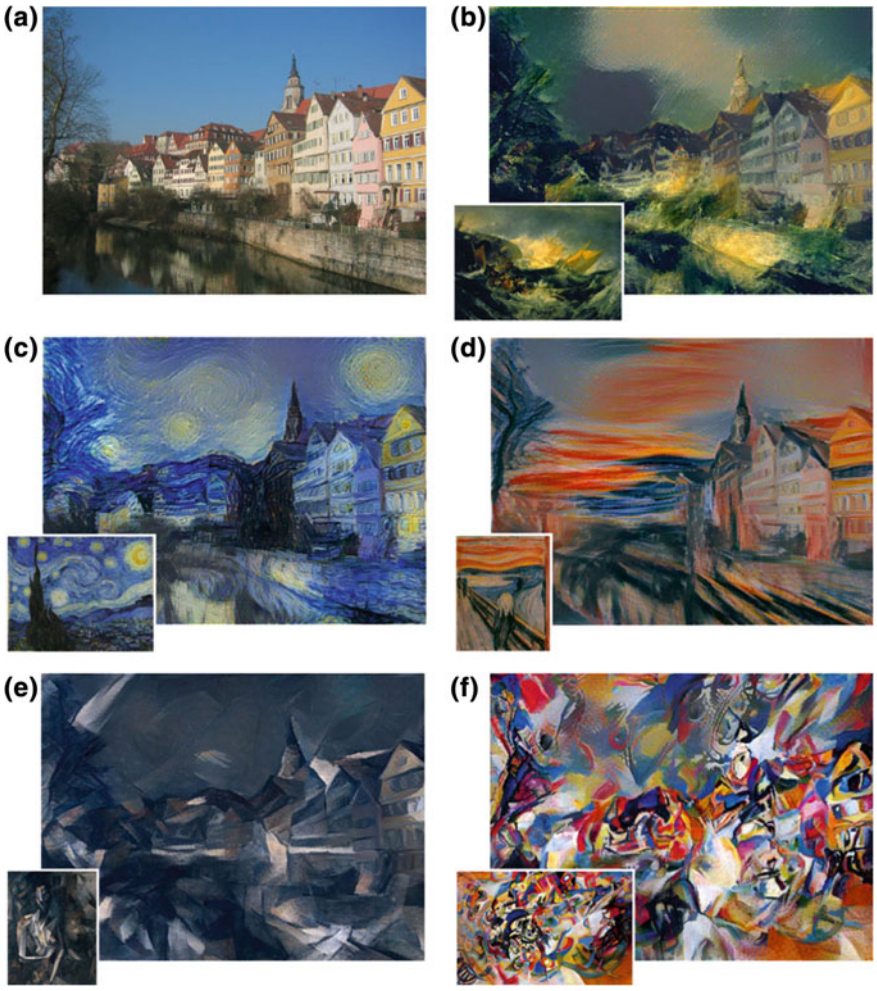

Fig. 9.29 Artistic pictures of the view of Tübingen in the top left generated with the network from [GEB15]. For each pair of images, the famous painting whose style was used to transform the photo can be seen in the bottom left (*Source* [GEB15])

One very nice example of such a creative effort is the CNN in [GEB15], which takes two input images and generates a new one that assumes the style of the first image and the content of the second. The results of this program are shown impressively in Fig. 9.29 on page 283. The photo in the top left is transformed into the respective styles of each painting by the old masters.

The program uses the VGG network [SZ15], which is a convolutional network that was pre-trained on the imagenet large scale visual recognition challenge with hundreds of object categories and millions of images [RDS+15]. From this pre-trained object classification network only the feature layers, but not the fully connected classification layers, are used here. Rather, the authors use only sixteen convolutional layers and five pooling layers.

To gain a better understanding of the procedure we first solve two simpler problems. The photograph is applied and propagated forward through the network, generating the features of the photo image. Now a pixel image with white noise is applied and propagated forward through the network, generating the features of the white noise image. Using gradient descent search, the pixels of the white noise image are now varied until the distance between the features of the image features to those of the photograph is minimized. Ultimately this reproduces the photo's *content*.

Then the photograph is replaced by the old work of art which is applied to another network with the same structure and, similarly to the photo, the features are computed. Again a pixel image with white noise is applied to the network. Using gradient descent search, it is changed until the distance between the correlations of the generated features and those of the painting are minimized. Ultimately this reproduces the painting's *style*.

Now, to transfer the syle of the painting onto the photo, both methods are combined. We apply gradient descent search on a pixel image with white noise but now we minimize a linear combination of both of the above distance functions. Thus an image in the style of the painting with the content of the photo is created.

9.9 Applications of Neural Networks

In addition to the applications given as examples thus far, there are countless applications for neural networks in all areas of industry, especially for deep learning. Pattern recognition in all of its forms is a very important area, whether analysis of photos to recognize people or faces, recognition of fish swarms in sonar readings, recognition and classification of military vehicles in radar scans, or any number of other applications. Neural networks can also be trained to recognize spoken language and hand written text.

Neural networks are not only used for recognizing objects and scenes. They can be trained to control self driving cars or robots based on sensor data, as well as for heuristically controlling search in backgammon and chess computers. An interesting use of networks for reinforcement learning is described in Sect. 10.8

For quite some time, neural networks, in addition to statistical methods, have been used successfully to forecast stock prices and to judge the creditworthiness of bank customers. Speed trading of international financial transactions would be impossible without the help of smart and fast neural networks that autonomously decide about buying or selling.

Other machine learning algorithms can as well be used for many of these applications. Due to the great commercial success of data mining, decision tree learning and support vector machines, there are neural algorithms for many applications as well as others that are not biologically motivated at all. The field of neural networks is a subarea of machine learning.

9.10 Summary and Outlook

With the perceptron, the delta rule, backpropagation, and, built upon them, deep learning, we have introduced the most important class of feedforward networks and shown their relationship to scores and naive Bayes, and also to the method of least squares.

Nearest to their biological role models are the fascinating Hopfield networks. However, due to their complex dynamics, they are difficult to handle in practice.

In all neural models, stored information is distributed over many weights. Therefore, the death of a few neurons has little noticeable effect on the function of a brain. The network is robust against small disturbances due to the distributed storage of the data. Related to this is the ability to recognize noisy patterns, which has been shown in several examples.

The distributed representation of knowledge has the disadvantage, however, that it is difficult for knowledge engineers to localize knowledge. It is practically impossible to analyze and understand the many weights in a trained fully connected neural network. In contrast, it is relatively easy to understand the learned knowledge in a decision tree, and even to represent it as a logical formula. Predicate logic, which allows formalization of relationships, is especially expressive and elegant. For example, a predicate `grandmother(katrin, klaus)` is easy to understand. This type of relationship can also be learned with neural networks. However, it is not possible to locate the "grandmother neuron" in the network. Thus, even today there are problems when connecting neural networks with symbol processing systems. A very important step in this direction is the automatic feature generation of deep learning networks.

Of the great number of interesting neural models, many have not been discussed. For example, self-organizing maps, introduced by Kohonen, are very interesting. A self-organizing map performs biologically inspired mapping from a sensory neural layer onto a second layer of neurons with the property that this mapping is adaptive and preserves similarities.

One problem with the networks presented here is incremental learning. If, for example, a fully trained backpropagation network is trained further with new patterns, many and eventually all of the old patterns will begin to be quickly forgotten.

To solve this problem, Carpenter and Grossberg developed adaptive resonance theory (ART), which has lead to a set of neural models.

As further reading, we recommend the textbooks [Bis05, Zel94, RMS92, Roj96, GBC16]. The collected volumes [AR88, APR90] are recommended for those interested in studying the history of this exciting area from the older, original works.

9.11 Exercises

9.11.1 From Biology to Simulation

Exercise 9.1 Show the point symmetry of the sigmoid function to $(\theta, \frac{1}{2})$.

9.11.2 Hopfield Networks

Exercise 9.2 Use the Hopfield network applet at http://www.cbu.edu/~pong/ai/hopfield/hopfieldapplet.html and test the memory capacity for correlated and uncorrelated patterns (with randomly set bits) with a 10×10 grid size. Use a pattern with 10% noise for testing.

Exercise 9.3 Compare the theoretical limit $N = 0.146\, n$ for the maximum number of patterns which can be stored, given in Sect. 9.2.2, with the capacity of classical binary storage of the same size.

9.11.3 Linear Networks with Minimal Errors

Exercise 9.4

(a) Write a program for learning a linear mapping with the least mean square method. This is quite simple: one must only set up the normal equations by (9.12) on page 265 and then solve the system of equations.

(a) Apply this program to the appendicitis data on this book's website and determine a linear score. Give the error as a function of the threshold, as in Fig. 9.14 on page 266.

(c) Now determine the ROC curve for this score.

9.11.4 Backpropagation

Exercise 9.5 Using a backpropagation program (for example in JNNS or KNIME), create a network with eight input neurons, eight output neurons, and three hidden neurons. Then train the network with eight training data pairs $\boldsymbol{q}^p$, $\boldsymbol{q}^p$ with the

property that, in the pth vector $\boldsymbol{q}^p$, the pth bit is one and all other bits are zero. The network thus has the simple task of learning an identical mapping. Such a network is called an 8-3-8 encoder. After the learning process, observe the encoding of the three hidden neurons for the input of all eight learned input vectors. What do you notice? Now repeat the experiment, reduce the number of hidden neurons to two and then one.

Exercise 9.6 In order to show that backpropagation networks can divide non linearly separable sets, train the XOR function. The training data for this is: ((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 0).

(a) Create a network with two neurons each in the input layer and hidden layer, and one neuron in the output layer, and train this network.
(b) Now delete the hidden neurons and connect the input layer directly to the output neurons. What do you observe?

Exercise 9.7

(a) Show that any multi-layer backpropagation network with a linear activation function is equally powerful as a two-layer one. For this it is enough to show that successive executions of linear mappings is a linear mapping.
(b) Show that a two-layer backpropagation network with any strictly monotonic activation function is not more powerful for classification tasks than one without an activation function or with a linear activation function.

9.11.5 Support Vector Machines

❄ **Exercise 9.8** Two non linearly separable two-dimensional sets of training data M_+ and M_- are given. All points in M_+ are within the unit circle $x_1^2 + x_2^2 = 1$ and all points in M_- are outside. Give a coordinate transform $\boldsymbol{f} : \mathbb{R}^2 \to \mathbb{R}^2$ which makes the data linearly separable. Give the equation of the dividing line and sketch the two spaces and the data points.