Chapter 1

Introduction

N. Sundararajan
P. Saratchandran
School of Electrical and Electronic Engineering,
Nanyang Technological University,
Singapore
Jim Torresen
Department of Computer and Information Science,
Norwegian University of Science and Technology,
Norway

Artificial Neural Network (ANN) is a discipline that draws its inspiration from the incredible problem-solving abilities of nature's computing engine, the human brain, and endeavors to translate these abilities into computing machines which can then be used to tackle difficult problems in science and engineering. The field of ANN itself is very broad, attracting researchers from such diverse disciplines as neuroscience, engineering, physics, computer science, and biology. The eclectic nature of this field has given rise to several paradigms each purporting to emulate some functional capabilities of the brain. From an application standpoint all these paradigms have their strengths and weaknesses. Each perform very well in solving certain problems but are not appropriate for others.

However, all ANN paradigms involve a learning phase in which the neural network is trained with a set of examples of a problem. The trained network is then used in a real environment to solve instances of the problem not contained in the examples. This is known as the recall phase. The learning phase usually takes a large amount of computing time for all but simple toy problems. For practical problems where the training data is large, training times of the order of days and weeks are not uncommon on serial machines [1–3]. This has been the main stumbling block for ANN's use in real-world applications and has also greatly impeded its wider acceptability.

The problem of large training time can be overcome either by devising faster learning algorithms or by implementing the existing algorithms on parallel computing architectures. Improving the learning algorithm per se is an active area of research [4, 5], but this book focuses on the latter approach of parallel implementation. The good thing about this approach is that an improved fast learning algorithm can be further speeded up by parallel implementation.

1.1 Parallel Processing for Simulating ANNs

An ANN consists of an enormous number of massively interconnected nonlinear computational elements (neurons). Each neuron receives inputs from other neurons, performs a weighted summation, applies an activation function to the weighted sum, and outputs its results to other neurons in the network. Simulation of an ANN comprises simulation of the

learning phase and the recall phase. Parallel processing of neural network simulations has attracted much interest during the past years. The learning and recall of neural networks can be represented mathematically as linear algebra functions that operate on vectors and matrices [6]. Thus, standard parallelization schemes can be exploited for both. However, the focus of parallel neural simulations has been more on the learning phase, which is the most computation-intensive part of neuroprocessing.

Parallel architectures for simulating neural networks can be subdivided into general purpose parallel computers and neurocomputers. Neurocomputers are designed as boards and systems for high-speed ANN simulations [7]. Neurocomputers can be classified as general purpose or special purpose [8]. A general-purpose neurocomputer is programmable and is capable of supporting a large range of neural network models, whereas a special-purpose neurocomputer implements one neural model in dedicated hardware. The latter benefits from higher *speed* than the former. Most neurocomputers are based on processing elements computing in parallel. A survey by Solheim [9] lists about 80 different digital neural hardware projects. However, only 20 of these proposed architectures have been implemented. These systems are designed and built by either research institutes or commercial companies.

1.1.1 Performance Metrics

Two metrics are commonly used for the speed of neural network simulations. Performance during training is measured in connections updates per second (CUPS). This accounts for the number of weights updated per second. For the recall phase performance, connections per second (CPS) are used, which describe the number of weight multiplications in the forward pass per second.

According to Crowl [10], presentation of parallel performance can be easily and unintentionally distorted. To avoid this, he suggests presenting the elapsed time as opposed to the speedup where possible. The use of speedup to define performance is limited by the lack of consensus for speedup definition. Crowl is of the opinion that many machines sacrifice sequential performance for parallel scalability, which gives rise to overestimated speedup. Linear speed (that is, solutions per time unit) is visually similar to speedup and may be used instead. CPS and CUPS measure linear speed by connections computed or updated per second.

The CUPS measure is sensitive to several factors.[1] When the number of neurons is large, the computation grains become large, which in most cases improve performance compared to that obtained from a small number of neurons. However, if the network is too large to be stored in main memory, the training slows down.

1.1.2 General Aspects of Parallel Processing

A parallel computer usually consists of a number of processing elements (PEs). Each processing element consists of a processor and memory.[2] The memory can either be on the processing chip or on separate chip(s). Recently, neural circuits have been produced containing several PEs on a single chip. Because the processing elements may have to

[1] For example, in the case of multilayer networks with backpropagation, inclusion of the momentum term increases the speed of convergence. But, there are more computations per iteration, leading to reduced CUPS performance. Also, the output layer has less backward error computation than the hidden layer. Thus, if more hidden layers are added to a network, the CUPS performance will be reduced.

[2] The name *processing element* was originally used for simple elements in SIMD computers, but today it is also used for the more complex elements in MIMD computers.

exchange data with their neighbors, a communication module may be required for each PE.

A number of topologies exist for interconnecting PEs [11]. The most common are shown in Figure 1.1: broadcast bus, ring, array, 2-D mesh, 2-D toroidal mesh (2D-torus), 3-D mesh, and hypercube. 1-D systems have a much lower optimal processor count than 2-D and 3-D systems [7, 12]. This means that much finer grained parallel processing can be realized by using a multidimensional topology.

Designers of parallel programs should be aware of Amdahl's law, which states in essence that the improvement of overall system performance attributed to the speeding up of one part of the system is limited by the fraction of the job that is not speeded up [13].

Parallel computers can be classified according to Flynn's classification, based on the number of simultaneous instruction and data streams [14]:

SISD (single instruction stream single data stream): A sequential computer with a single CPU.

SIMD (single instruction stream multiple data streams): A single program controls multiple execution units.

MISD (multiple instruction streams single data stream): Systolic arrays with pipelined execution.

MIMD (multiple instruction streams multiple data streams): Computers with more than one processor and the ability to execute more than one program simultaneously. Computers in this category are also called multiprocessors or multicomputers depending on shared or distributed memory, respectively.

A special execution mode of MIMD has been defined:

SPMD (single programs operating on multiple data streams): The same program is downloaded onto all the processing elements. The processors are usually performing the same operations but on different parts of the data [15].

Several different methods are used for interprocessor communication. The two major switching methods for communication [14] are the following :

Circuit Switching. A physical path is established between the source and the destination before the message is sent. SIMD machines frequently use circuit switching.

Packet Switching. A message is split into fixed or flexible-sized packets. Each packet is routed through the interconnection network independent of other packets. As such, packets may take different routes through the network. MIMD machines are usually based on packet switching.

Two major concerns in parallel *implementations* are the following:

Load Balancing. To minimize idle time, it is necessary to keep the processors active. Each processor should be given an equivalent computation load.

Communication. To maximize the time processors perform computation, communication should be minimized. Moreover, the communication should be distributed as evenly as possible over all the communication links [16].

As the number of processors increases, these factors become more dominating. One of the purposes of the work presented in this book is to show how fixed mappings of neural

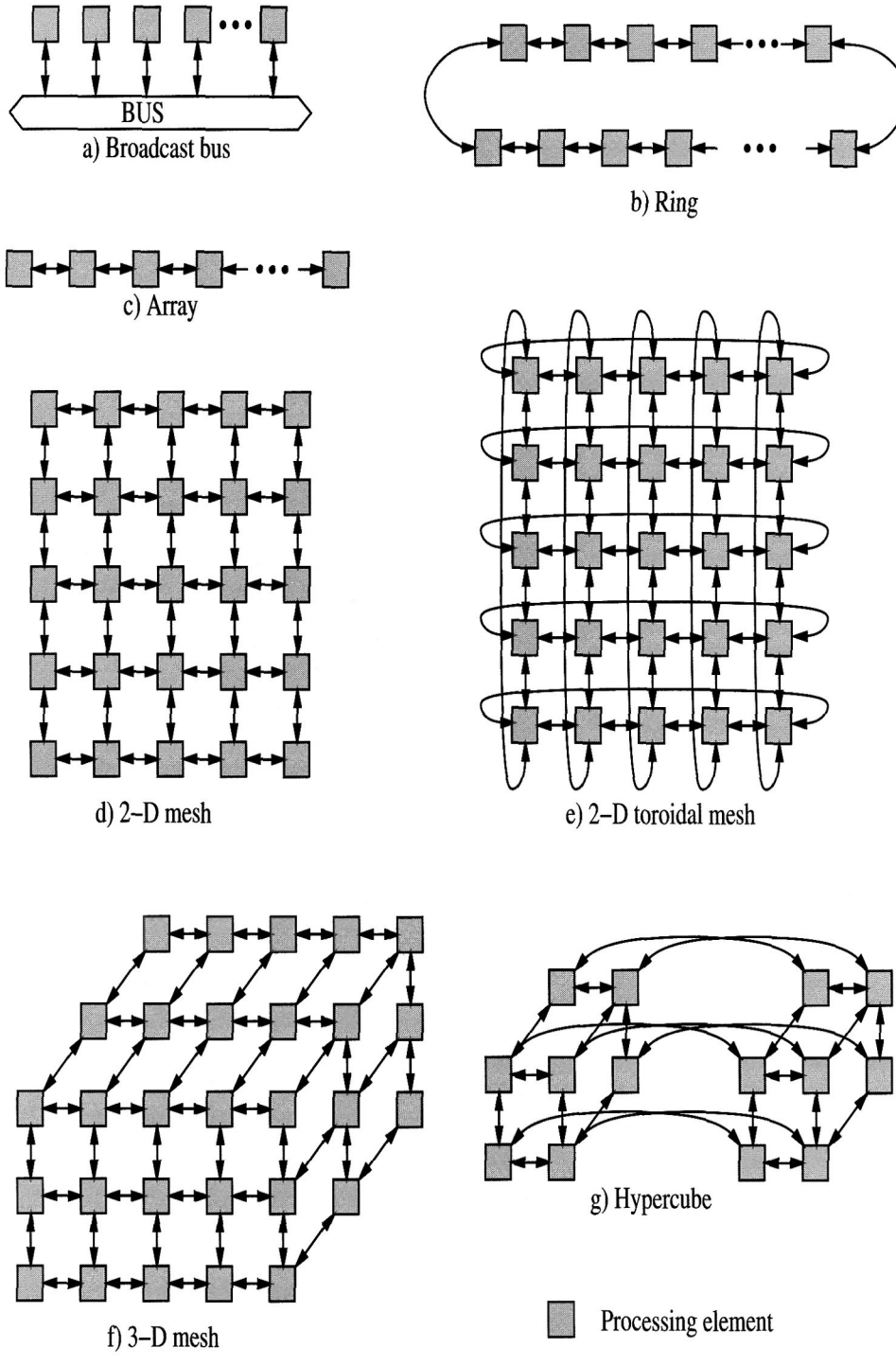

FIGURE 1.1. Processor topologies for simulating ANNs (from [11]).

networks to large parallel systems can be weakened by these problems. Another purpose is to propose solutions to minimize these problems.

The computation grain size is an important factor in load balancing [17]. Grain size determines the basic program segment chosen for parallel processing [18]. *Fine*-grained parallelism means that the computation is spread over a number of small tasks; for *coarse*-grained parallelism, however, the tasks are substantially larger. Some computers are suited for coarse-grained computation (such as message-passing MIMD computers), whereas others are designed for fine-grained computation (massively parallel SIMD computers, for example).

Complexity modeling is theoretically a way to specify the upper bound on a program's running time. It is specified by the "big-oh" notation. For an input size n, $\mathcal{O}(n^2)$ means that there are positive constants c and n_0, such that for n equal to or greater than n_0, the running time for a program is $T(n) \leq cn^2$.

1.2 Classification of ANN Models

Research on artificial neural networks can be traced back to the 1940s, when McCulloch and Pitts published their pioneering paper [19] describing the properties of a simple binary threshold type of artificial neuron that has both excitatory and inhibitory inputs. When connected as a network, these neurons could compute any logical (Boolean) function. This, together with Donald Hebb's discovery that learning in biological neurons occurs through synaptic growth, triggered the development of several artificial neural computing models in the 1950s. The most notable among the early models was the perceptron [20]. During the early 1960s, researchers presented convincing demonstrations using the perceptron model [21]. But in the late 1960s, Minsky and Papert uncovered severe restrictions in the learning capabilities of the perceptron model [22]. Very little research emerged during the next 15 years, although stalwart researchers like Stephen Grossberg, Teuvo Kohonen, James Anderson, Bernerd Widrow, David Willshaw, Amari, and a few others continued to develop various ANN models and learning algorithms. The major thrust of their work was in the area of associative content-addressable memory in which sufficiently similar inputs become associated with one another. Finally, in 1986, Rumelhart and colleagues introduced the learning algorithm for the multilayer perceptron network, called backpropagation (BP) [23]. This initiated a remarkable increase in neural network research resulting in an explosion of network architectures and learning algorithms.

Because the number of learning algorithms and architectures in the ANN literature is large, diverse, and growing, it is impossible to come up with a single classification scheme that can capture all the essential traits of all these paradigms. A possible classification scheme based on the type of learning (supervised/unsupervised/hybrid), architecture of the network (feed-forward/recurrent), and the connectivity employed for the neurons (such as multilayer, competitive, adaptive resonance theory) is shown in Figure 1.2. The learning algorithms presented in Figure 1.2 have wide popularity but by no means are the only ones to be found in the ANN literature.

1.3 ANN Models Covered in This Book

The ANN models for which parallel implementation is discussed in this book are enclosed in shaded boxes in Figure 1.2. A brief description of each of these models follows.

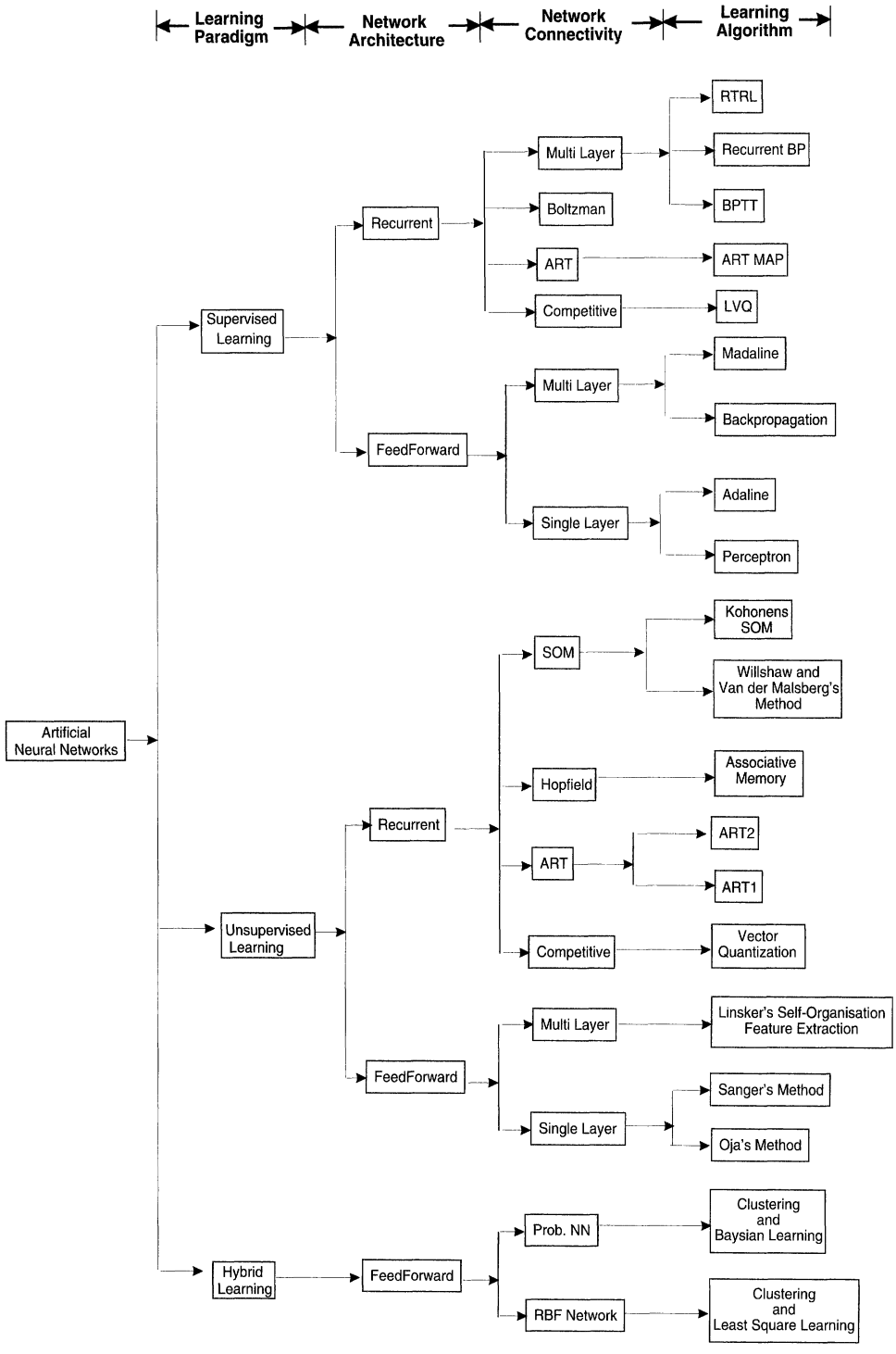

FIGURE 1.2. A Taxonomy of ANN Models.

1.3.1 Multilayer Feed-Forward Networks with BP Learning

A two-layer[3] feed-forward network is shown in Figure 1.3. The network is called fully connected, because there are all-to-all connections between two adjacent neuron layers. The number of neurons (also called units) in each layer is N_i, N_h, and N_o for the input, hidden, and output neuron layers, respectively. The network can be extended to any number of layers; however, because most applications use two-weight layers, the description here has been restricted to two-layer networks. The BP learning phase for a pattern consists of a forward phase followed by a backward phase. The main steps are as follows:

1. Initialize the weights to small random values.
2. Select a training vector pair (input and the corresponding output) from the training set and present the input vector to the inputs of the network.
3. Calculate the actual outputs - this is the *forward phase.*
4. According to the difference between actual and desired outputs (error), adjust the weights W_o and W_h to reduce the difference - this is the *backward phase.*
5. Repeat from step 2 for all training vectors.
6. Repeat from step 2 until the error is acceptably small.

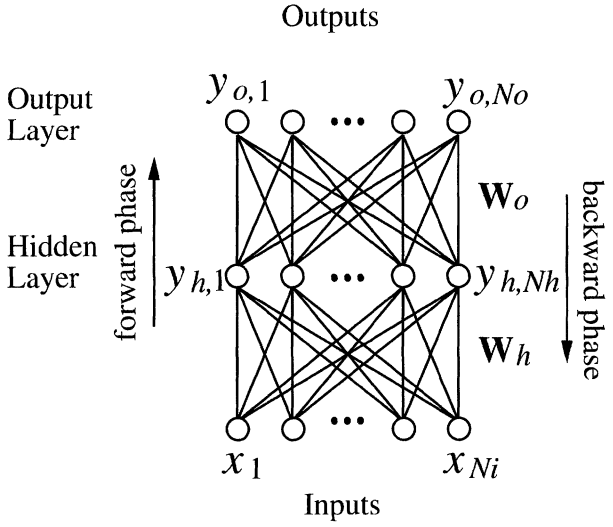

FIGURE 1.3. A two-weight layer feed-forward neural network.

The weight updating scheme used is called *learning by pattern (lbp)* or on-line weight update, and it updates the weights after *each* training pattern has been presented. Experiments have shown this update method, which is stochastic gradient descent [24], converges faster than the total gradient descent, which updates the weights after all training patterns have been presented. However, only the latter method, called *learning by epoch (lbe)*, has

[3] In this section, the word *layer* refers to the number of layers of weights in the network. This is equal to the number of layers of neurons, when excluding the input neuron layer.

been proved to converge.[4] An intermediate method is called *learning by block (lbb)* and it updates the weights after a certain number of patterns have been presented.

1.3.1.1 A Detailed Description of the BP Learning Algorithm. In the forward phase the hidden layer weight matrix W_h is multiplied by the input vector $X = (x_1, x_2, \ldots, x_{N_i})^T$, to calculate the hidden layer output

$$y_{h,j} = f(\sum_{i=1}^{N_i} w_{h,ji} x_i - \theta) \tag{1.1}$$

where $w_{h,ji}$ is the weight connecting input unit i to unit j in the hidden neuron layer.[5] The θ is an offset termed bias incorporated into the training algorithm by a weight connected to +1 for each neuron [21]. This bias-weight is trained like an ordinary weight.

The function f is a nonlinear activation function. Normally the S-shaped sigmoid function

$$f(\alpha) = \frac{1}{1 + e^{-\alpha}} \tag{1.2}$$

is used. It compresses the output value to lie in $< 0, 1 >$, as shown in Figure 1.4. Moreover, the function is differentiable, which is a demand of the training algorithm.

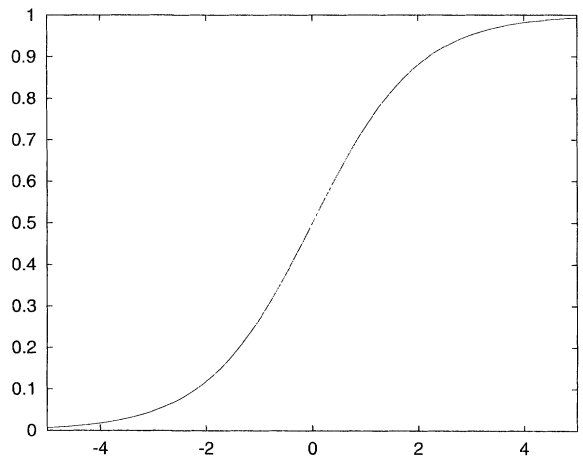

FIGURE 1.4. The sigmoid function $f(\alpha) = \frac{1}{1 + e^{-\alpha}}$.

The output from the hidden layer, $y_{h,j}$, is used to calculate the output of the network, $y_{o,k}$

$$y_{o,k} = f(\sum_{j=1}^{N_h} w_{o,kj} y_{h,j} - \theta). \tag{1.3}$$

[4]However, the number of training set presentations required for convergence has no known theoretical bound.

[5]To distinguish the different neuron layers, the indices i, j, and k are used for indexing the input, hidden, and output neuron layers, respectively.

The error measure E_p for a training pattern p is given by

$$E_p = \frac{1}{2}\sum_{k=1}^{N_o}(d_{p,k} - y_{p,o,k})^2. \tag{1.4}$$

The overall error measure for a training set of P patterns is

$$E = \sum_{p=1}^{P} E_p. \tag{1.5}$$

In the following expressions, the pattern index p has been omitted on all variables to improve clarity. In the backward phase the target, d, and output, y_o, are compared and the difference (error) is used to adapt the weights to reduce the error.

The error used to update the weights can be shown [23] to be

$$\delta_{o,k} = y_{o,k}(1 - y_{o,k})(d_k - y_{o,k}). \tag{1.6}$$

Similar to computing the output delta error, the hidden delta error value for neuron j is

$$\delta_{h,j} = y_{h,j}(1 - y_{h,j})\sum_{k=1}^{N_o}\delta_{o,k}w_{o,kj}. \tag{1.7}$$

The error is not explicitly given and is computed based on the impact of the fan-in of the output delta errors. To perform steepest descent in the weight space, the weight changes become

$$\Delta w_{o,kj} = \eta\delta_{o,k}y_{h,j} \tag{1.8}$$

$$\Delta w_{h,ji} = \eta\delta_{h,j}x_i \tag{1.9}$$

where η is the learning rate coefficient.

If learning by pattern is applied, the output layer weights are changed to $w'_{o,kj}$

$$w'_{o,kj} = w_{o,kj} + \eta\delta_{o,k}y_{h,j}. \tag{1.10}$$

The hidden layer weights are updated accordingly

$$w'_{h,ji} = w_{h,ji} + \eta\delta_{h,j}x_i. \tag{1.11}$$

The training continues for each vector in the training set until the error for the entire set becomes acceptably small.

Instead of updating the weights after each training pattern presentation, they can be updated less frequently by using learning by block. For updates after μ patterns have been presented, Equations 1.10 and 1.11 become 1.12 and 1.13:

$$w'_{o,kj} = w_{o,kj} + \eta\sum_{p=p'+1}^{p'+\mu}\delta_{p,o,k}y_{p,h,j} \tag{1.12}$$

$$w'_{h,ji} = w_{h,ji} + \eta\sum_{p=p'+1}^{p'+\mu}\delta_{p,h,j}x_{p,i}. \tag{1.13}$$

The total number of training patterns is P and $\mu \leq P$. Learning by block is well suited to parallelization, but as shown by Paugam–Moisy [25], the convergence rate declines as μ gets larger. Therefore, an appropriate value for μ needs to be chosen. When $\mu = P$, the scheme is called learning by epoch.

The error measure in Equation 1.5 is dependent on N_o and P. Thus, large numbers for N_o and P result in a large value for the overall error E. An alternative measure for the overall error is the root mean square error (RMSE) given by

$$E_{RMSE} = \sqrt{\frac{1}{PN_o} \sum_{p=1}^{P} \sum_{k=1}^{N_o} (d_{p,k} - y_{p,o,k})^2}. \tag{1.14}$$

As stated, no proven convergence for the backpropagation algorithm exists,[6] and therefore, the word *convergence* is defined as reaching a predetermined stopping criteria [25].

1.3.1.2 Momentum. To obtain true gradient descent requires infinitesimal small changes of the weights. This is obtained by selecting a small value for the learning rate. However, we want to choose as large a learning rate as possible without leading to oscillation,[7] because experiments show that this offers the most rapid learning [23]. To increase the learning rate and avoid oscillation, a momentum can be included. Rumelhart and colleagues proposed to add a fraction, equal to α, of the previous weight update value to the current weight change [23]

$$\Delta w_{o,kj}(p+1) = \eta \delta_{o,k}(p+1) y_{h,j}(p+1) + \alpha \Delta w_{o,kj}(p) \tag{1.15}$$

where p is the training pattern index. The weights are then updated

$$w_{o,kj}(p+1)' = w_{o,kj}(p) + \Delta w_{o,kj}(p+1). \tag{1.16}$$

Similarly, Sejnowski and Rosenberg [26] proposed a smoothing term, α

$$\Delta w_{o,kj}(p+1) = \alpha \Delta w_{o,kj}(p) + (1-\alpha) \delta_{o,k}(p+1) y_{h,j}(p+1). \tag{1.17}$$

The smoothing makes it less necessary to scale the learning rate according the weight update interval

$$w_{o,kj}(p+1)' = w_{o,kj}(p) + \eta \Delta w_{o,kj}(p+1). \tag{1.18}$$

The equations for updating of the hidden weights can be similarly derived. The term α is normally set to around 0.9.

1.3.1.3 The Effect of the Weight Update Interval. It is shown for one neural application by Paugam–Moisy that less frequent weight updates during training reduces the convergence rate. That is, the decrease in error per training iteration is smaller. The network classified patterns into three classes. To avoid unstable behavior during training, the learning rate had to be reduced when the weight update interval was increased. The reason for this can be explained by network paralysis [21]. Adding weight changes for many training vectors together may result in large weight change values. This may lead to large weight

[6] An exception is learning by epoch, but still there is no bound in the number of training iterations required.

[7] The error is not constantly decreasing but is oscillating between large and small error values without reaching convergence.

values if the learning rate is not reduced. Large weights can lead to large output values to be input to the nonlinear function. The derivative of the function, which is used for computing the delta error, approaches zero for large values. This results in a very small change in the weights, and the training can come to a virtual standstill.

A special case of learning by pattern is *delayed weight update* in which the weights are updated for pattern p *after* the forward pass for the next pattern, $-p + 1$, has been computed. This method may be used for weight updating when the computation of each layer is distributed over different processors. The delta weight change values are small compared to the weights and thus the convergence should be very close to the convergence of ordinary pattern weight updates.

For some neural application experiments learning by pattern updating results in a stagnation of the error that does not occur for learning by epoch updating. The quickpropagation (quickprop) algorithm, proposed by Fahlman [27], is based on learning by epoch. That is, all training vectors are applied before the new weights are computed. Thus, training set parallelism can be used in parallel implementation of quickprop without leading to any reduction in the convergence rate.

The redundancy in large training sets slows down the convergence of learning by epoch-based algorithms according to Møller [28]. Accumulating redundant gradient weight vectors implies redundant computation. This is not a problem if the weights are updated after each training vector. In fact, redundancy has a positive effect in the beginning of the training. However, simulations show that a conjugate gradient training algorithm— based on epoch learning— which is more efficient in the end of training even though there is redundancy in the training set. Thus, a learning by block approach is proposed, where the block size varies throughout training. Because the redundancy is dependent on the problem, the block size has to be selected by estimation. For each iteration the block size that leads to a confident decrease in the total error of the training set is determined. Simulation using Nettalk [26] shows that the training starts with a very small block size and ends by updating weights only two or three times per epoch.

1.3.1.4 Learning Performance. The number of floating point operations used for weight updating for learning by pattern differs from the number for learning by block/epoch. In this section, expressions will be derived to show the difference in computation between the different weight update strategies [29]. The expressions are based on a serial execution of the training program. On a parallel computer, additional time is used for communication. The Nettalk application with number of neurons $N_i = 203$, $N_h = 120$, $N_o = 26$, and $P = 5,438$ training patterns is used here to illustrate the difference. The Nettalk training uses the Sejnowski momentum (Equations 1.17 and 1.18) for weight update. The number of floating point operations used for *lbp* weight updating is then 6 per weight, leading to a total number for one training iteration

$$F_{lbp} = 6P(N_i + N_o)N_h \tag{1.19}$$

$$= 6 \times 5,438(203 + 26)120 = 896,617,440. \tag{1.20}$$

For *lbb* the number of floating point operations for weight accumulation and updating—2 and 5, respectively— is found from Equations 1.12, 1.13, 1.17, and 1.18. This leads to a total floating point operation count of

$$F_{lbb} = 2P(N_i + N_o)N_h + 5\left\lceil \frac{P}{\mu} \right\rceil (N_i + N_o)N_h \tag{1.21}$$

$$= (2P + 5\left\lceil \frac{P}{\mu} \right\rceil)(N_i + N_o)N_h \tag{1.22}$$

$$= (2 \cdot 5,438 + 5\left\lceil \frac{5438}{\mu} \right\rceil)(203 + 26)120 \tag{1.23}$$

$$= 298,872,480 + \frac{1}{\mu}747,181,200. \tag{1.24}$$

The first expression in Equation 1.21 represents the weight accumulation and the latter represents weight updating. The number of operations for weight accumulation is less than for weight update. Thus, if the weights are infrequently updated we get $F_{lbp} > F_{lbb}$. Figure 1.5 shows the number of floating point operations for weight accumulation and update for different weight update intervals, μ.

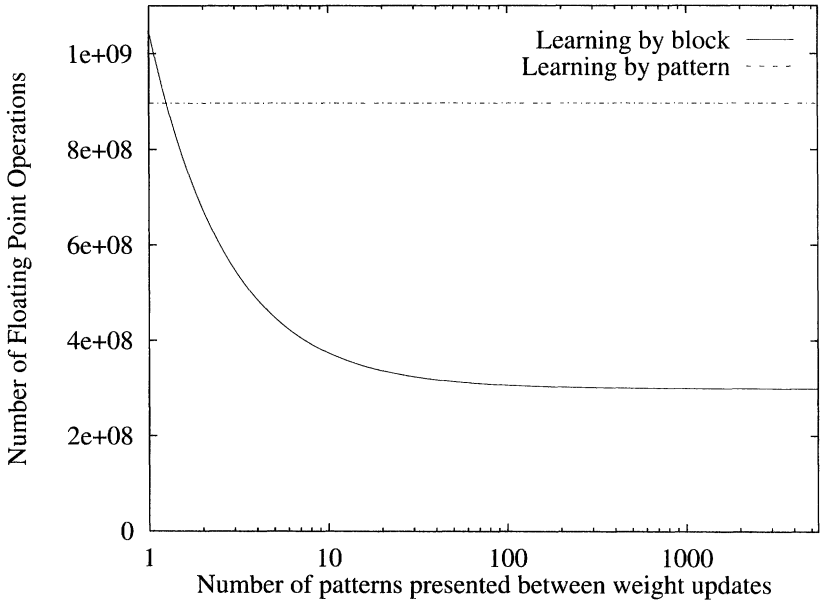

FIGURE 1.5. The number of floating point operations for weight accumulation and update for block updates, F_{lbb}, and for pattern updates, F_{lbp}.

The update method learning by epoch uses a smaller number of operations than learning by pattern if the block size is larger or equals two, that is, $\mu \geq 2$.

Where learning by block/epoch is used, CUPS is given by the number of weight-change values computed per second. That is, the number of weights *updated* is excluded when the CUPS value is calculated. However, the time for weight update is not excluded. Thus, only the number of weights *accumulated* per epoch is used for computing the CUPS value, which is like omitting the second expression in Equation 1.21.

To measure how well a feed-forward neural network has learned, a separate test set should be used in addition to the training set. The available vectors should be partitioned into disjoint sets: learning set and test set. It is preferable that several different test sets are included. The test set should include a representative selection of patterns; for example, for pattern classification, the test set should contain vectors from all classes. The network is trained by the training set and then tested using the test set. The test set or a separate acceptance set is used as an acceptance set for the network.

Variants of the BP Algorithm. During recent years, several variants of BP learning have been proposed, such as quick propagation and conjugate gradient method. A comparison of learning algorithms for feed-forward networks by Nesvik [30] showed that some of the new algorithms were less sensitive to the selection of the learning parameters.

1.3.2 Hopfield Network

The Hopfield network is a single layer recurrent network that embodies the idea of storing information as the stable states of a dynamically evolving network configuration. Using an energy function in terms of the connection weights and outputs of the neurons, Hopfield showed [31, 32] how such networks can be used to solve specific problems in associative memory and combinatorial optimization. Two versions of the Hopfield network exist: discrete and continuous valued. The discrete Hopfield network is used as associative memory, whereas the continuous Hopfield network is used for combinatorial optimization.

1.3.2.1 Discrete Hopfield Network. In this network the neurons can only have two states as their output. The *on* state is represented by +1 and the *off* state is represented by minus 1. A network with N neurons is shown in Figure 1.6. The state of the network is given by the vector $\mathbf{Y} = [y_1, \ldots, y_i, \ldots, y_N]^T$, where y_i denotes the output of neuron i, which can only be ± 1. The output of neuron i is given by

$$y_i = sgn(\sum_{j=1}^{N} w_{ij} y_j - \theta_i) \tag{1.25}$$

where sgn is the signum function, w_{ij} is the weight connecting neuron i and j, w_{ii} is zero, and θ_i is a fixed threshold value. If the argument of the signum function is zero, then the output of neuron i remains unchanged. As an associative network, the operation of Hopfield network has two phases: the storage phase and the retrieval phase.

1.3.2.2 Storage Phase. In this phase the network is trained to memorize the patterns it has to recall later. These patterns are known as prototype patterns or fundamental memories of the network. Training the network is a one-shot operation as the connection weights are directly computed from the following equation:

$$w_{ij} = \frac{1}{N} \sum_{p=1}^{m} \eta_i^p \eta_j^p \tag{1.26}$$

where η_i^p and η_j^p are the i^{th} and j^{th} elements of the prototype pattern η^p and m is the number of prototype patterns the network has to memorize. Once computed, the weights are kept fixed. Note from Equation 1.26 that weights w_{ij} and w_{ji} are the same. The weight matrix $[W]$ for the network is thus symmetric.

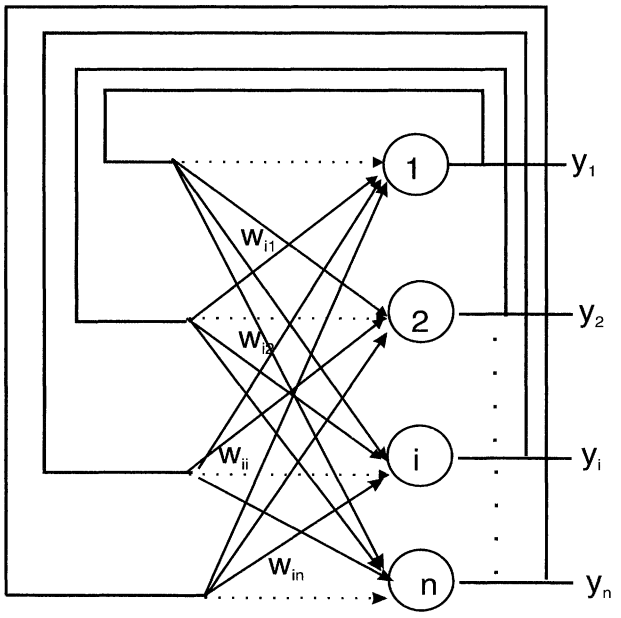

FIGURE 1.6. A Hopfield Network of N neurons.

1.3.2.3 Retrieval Phase. In this phase the network is presented with a pattern that differs from the prototype patterns. Usually it is an incomplete or noisy version of one of the prototype patterns. The state of the network then evolves according to the following difference equation:

$$\mathbf{Y}(k+1) = sgn(\mathbf{W}\,\mathbf{Y}(k) - \Theta) \tag{1.27}$$

where $\mathbf{Y}$ is the output vector and $\mathbf{W}$ is the symmetric weight matrix. The vector $\Theta = [\theta_1, \ldots, \theta_i, \ldots, \theta_N]^T$ is the threshold vector and k is the iteration index. Starting with the input pattern vector, the outputs are updated recursively according to Equation 1.27 until the network settles to a stable state. The updating of the outputs could be synchronous or asynchronous. In synchronous updating, the outputs for all neurons— that is, $y_i(k+1);\ i = 1, \ldots, N$— are computed simultaneously at each iteration index. This is also called the parallel mode of operation. In asynchronous updating, $y_i(k+1);\ i = 1, \ldots, N$ are computed sequentially in some random order. In an iteration, if neurons are selected randomly one by one and their outputs are computed sequentially, then it is known as simple asynchronous updating. If groups of neurons are selected randomly and updated synchronously within the group and sequentially between groups, then it is called general asynchronous dynamics. Note that in simple asynchronous dynamics the neuron updating is always sequential and random whereas in general asynchronous dynamics neuron updating is parallel within a group and serial between groups. However, all neurons must be updated once in each iteration. If the neurons are updated according to the simple asynchronous dynamics, then the Hopfield network will always converge to a stable state. If the neurons are updated synchronously then the network will always converge to a stable state or a limit cycle of length 2 or less [33].

1.3.2.4 Energy Function. Hopfield used an energy function to prove convergence of the network when simple asynchronous updating is used for the network outputs. The energy

function $E(k)$, assuming the threshold vector (Θ) to be zero, is given by

$$E(k) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}y_i(k)y_j(k). \tag{1.28}$$

As the network evolves according to the dynamics of Equation 1.27, the energy $E(k)$ can only decrease or stay unchanged at each update. This is because the change $\Delta E(k)$ due to a change in the output y_i can only be zero or negative [33]. Eventually the network will converge to a (local) minimum energy state because E is bounded from below. The local minimum points in the energy landscape correspond to the prototype patterns stored in the storage phase. The maximum number of patterns a Hopfield network of N neurons can store is approximately equal to $0.15N$ [31].

1.3.2.5 Continuous Hopfield Network. The continuous version of the Hopfield network is obtained from the discrete version by replacing the signum function by a sigmoid function in equation 1.25. The neuron outputs are then continuous in the range 0 to 1 and the outputs of all neurons are updated simultaneously and continuously. Because the outputs are continuous valued and they change continuously, the dynamics of the network can be described by a set of simultaneous nonlinear differential equations as [32]

$$\tau_i\frac{dy_i}{dt} = -y_i + f(\sum_{j=1}^{N} w_{ij}y_j) \quad i = 1,\ldots,N \tag{1.29}$$

where τ_i is a time constant and $f(.)$ is the sigmoid function. Because the weight matrix is symmetric, the solution $y_i(t)$ to the previous equations will always converge to a fixed point as guaranteed by the Cohen-Grossberg theorem [34].

The behavior of the network can then be shown to minimize an energy function as in the case of discrete Hopfield network. Hopfield (1984) used the energy function

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}y_i\, y_j + \sum_{i=i}^{n}\int_0^{y_i} f^{-1}(y)dy$$

and showed that $\frac{dE}{dt} \leq 0$ for all t. It is this property of the Hopfield network that makes it suitable for solving problems in optimization. If a combinatorial optimization problem can be expressed as minimizing an energy function, then a Hopfield network can be used to find the optimal (or suboptimal) solution for that problem. The real difficulty is in mapping the objective function of the optimization problem subject to various constraints into a single energy function. Once the energy function is found, then the weights of the network can be found by relating the terms of the energy function to those of the general form. Hopfield and Tank [35] showed how a continuous Hopfield network can be used to solve the classic Traveling Salesman Problem for 10 cities.

1.3.3 Multilayer Recurrent Networks

Single-layer recurrent networks, such as the Hopfield network, with symmetric connection weights, are guaranteed to converge to stable states and, hence, are not suitable for learning temporal sequences of patterns. If the connection weight matrix in a recurrent network is made assymetric, then the resulting network can converge to stable states or exhibit limit cycles or chaos. Such networks can be used for generating, recognizing, and storing temporal sequences of patterns with applications in speech recognition, time series prediction,

identification and control of nonlinear dynamic systems, and so on. [36]. The dynamics of assymetric recurrent networks have been studied widely and many training algorithms, which are modified forms of backpropagation, have been proposed [37–42]. The real-time recurrent learning algorithm by Williams and Zipser [41] is perhaps the most popular among these because of its ability to train the network on-line as the input patterns are presented sequentially in time.

1.3.3.1 Real-Time Recurrent Learning. In real-time recurrent learning (RTRL) the weights can be incremented on-line or at the end of the whole input sequence. Because on-line updating is possible, the RTRL algorithm can deal with input sequences of arbitrary length and does not require memory proportional to the length of input sequence. The cost function to be minimized is given by $E_{tot}(t_0, t_f)$

$$E_{tot}(t_0, t_f) = \frac{1}{2} \sum_{t=t_0}^{t=t_f} \sum_{k} [E_k(t)]^2 \tag{1.30}$$

where $t = t_0, \ldots, t_f$ is the time domain of interest. $E_k(t)$ is the error for neuron k at time t and is the difference between the desired output $d_k(t)$ and the actual output $y_k(t)$ of the neuron k at time t. If no desired output is specified for neuron k at time t, then $E_k(t)$ will be set to zero.

$$E_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } d_k(t) \text{ is defined,} \\ 0 & \text{otherwise.} \end{cases}$$

Gradient descent is used to adjust the weights of the network. The gradient of $E_{tot}(t_0, t_f)$ separates in time

$$\frac{\partial}{\partial w} E_{tot}(t_0, t_f) = \frac{1}{2} \frac{\partial}{\partial w} \sum_{k} [E_k(t_0)]^2 + \ldots + \frac{1}{2} \frac{\partial}{\partial w} \sum_{k} [E_k(t_f)]^2.$$

Therefore, the total weight increment for any weight becomes the sum of the weight increments calculated at times $t_0, t_1, \ldots, t_f$. The weights can be updated at the end of the sequence at $t = t_f$, but a better procedure is to keep updating them at each time step t_0, t_1 and so on. The resulting algorithm is called the real-time recurrent learning because the weights are updated in real time as the input sequence is presented. The method avoids the need for large storage requirement of long input sequences and works especially well if the learning rate is small.

1.3.4 Adaptive Resonance Theory (ART) Networks

One of the interesting properties of the human brain is its ability to learn new facts continually without forgetting previously learned facts. It is thus flexible (plastic) enough to learn new facts yet rigid (stable) enough to keep irrelevant facts from washing away the old facts. An artificial neural network trying to exhibit this property should have a high degree of stability with respect to what has been learned, yet it should be adaptive enough to learn new facts on a continuous basis. However, it is not easy for an ANN to have both a high degree of stability and plasticity as these two characteristics conflict. Grossberg calls this the *stability-plasticity dilemma*, and the ART networks (ART1, ART2, ARTMAP) were developed by Carpenter and Grossberg [43–46] in an attempt to solve this stability-plasticity problem. The simplest form of ART network is the ART1 network, which is designed to work for binary (0/1) inputs only. Extension of ART1 to handle both binary and continuous

valued inputs is achieved in ART2. Both ART1 and ART2 are unsupervised competitive learning networks. ARTMAP is a modification of ART2 and is a supervised competitive learning network.

The ART1 network consists of an input layer and an output layer that are fully connected to each other. An input pattern vector X is applied and the output neuron that has the largest net input is picked as the winner; that is, if the winning neuron is r, then

$$\frac{W_r \cdot X}{\beta + \Sigma_j w_{jr}} > \frac{W_i \cdot X}{\beta + \Sigma_j w_{ji}} \text{ for all } i \tag{1.31}$$

where W_r and W_i are the weight vectors of the output neurons r and i, respectively, and the symbol "$\cdot$" means scalar product. β is a small positive constant and w_{jr} and w_{ji} are the jth component of W_r and W_i respectively. Note that dividing by $\Sigma_j w_{jr}$ and $\Sigma_j w_{ji}$ normalizes the weight vectors W_r and W_i and β is included merely to break the ties in selecting the winner.

The weight vector W_r of the winning neuron is then tested for its similarity with the input pattern X. The similarity measure (s_r) for the weight vector W_r is computed as the ratio of the number of 1s overlapping in X and W_r to the total number of 1s in X. If $s_r \geq \rho$, where ρ is a threshold called the "vigilance" parameter, then W_r is considered to be sufficiently similar to X. If W_r is sufficiently similar to X, then resonance is said to occur between the input and output and the resonating output neuron becomes a "committed" neuron. The weight vector of the resonating neuron r is then moved closer to X by changing to zeros the 1s in W_r for which the corresponding components in X are zeros.

If W_r is not sufficiently similar to X (that is, $s_r < \rho$), then the winning neuron r is "disabled" or withdrawn from the competition. A new winner is then found using Equation 1.31 from among the output neurons that are not disabled and is tested for resonance with the input vector X. If the input vector does not resonate with any of the output neurons, then an "uncommitted" output neuron is picked (even though it did not resonate) and its weight vector made equal to the input vector to provide resonance with the input. If none of the output neurons resonate with the input vector and no uncommitted output neurons are left, then the input pattern is rejected implying that the network has reached its capacity. The ART1 network runs entirely autonomously without needing any external control or sequencing signals and the architecture is entirely parallel.

1.3.5 Self-Organizing Map (SOM) Networks

The SOM network [47] is a simplified model of the feature-to-local region mapping done in the human brain. The basic idea is that inputs that are close to each other according to some metric in the input space should be mapped to output neurons that are spatially close together. This is known as topology preservation and is an important aspect of feature mapping in the brain. The architecture of SOM network consists of a two-dimensional array of neurons with each neuron connected to all input nodes. Further, all the neurons also have lateral connections to each other. The strength of lateral connections follow a "Mexican-Hat" function [48]. By involving a neighborhood function to achieve the effect of the Mexican-Hat lateral interactions, Kohonen devised a SOM network in which no lateral connections exist. Figure 1.7 shows the basic architecture of Kohonen's SOM network. There are n input units and the weight connecting input units to the output neuron i is denoted by the vector W_i. When an input pattern X is applied, each output neuron computes the Euclidean distance between the input vector and the weight vector of that neuron. Competitive learning rule is then used and the neuron whose weight vector is closest to the

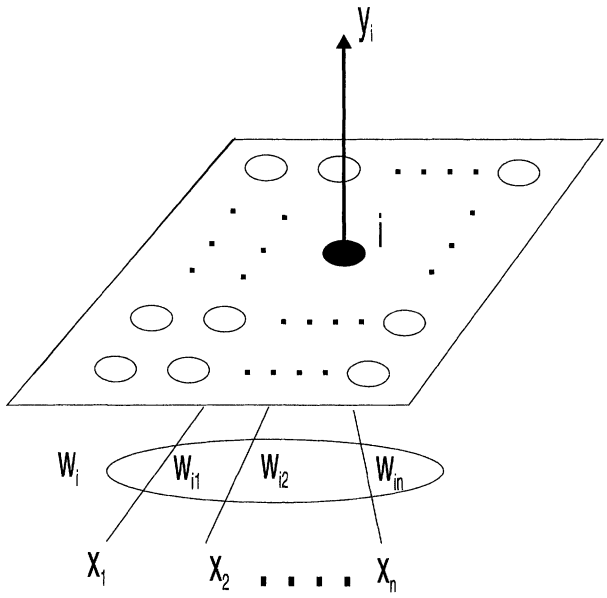

FIGURE 1.7. Kohonen's SOM network.

input vector is chosen as the winner; that is, if the winning neuron is r, then

$$||X - W_r|| = \min_i ||X - W_i|| \text{ for all } i. \tag{1.32}$$

The network learns by changing appreciably the weights of the winning neuron and its neighbors by dragging their weight vectors toward the input pattern X, whereas those far away from the winning neurons experience little change to their weights. This is how the topology preservation is achieved. The weight update equation is given by

$$W_i^{new} = W_i^{old} + \alpha\, N(i,r)\, (X - W_i) \text{ for all } i. \tag{1.33}$$

The function $N(i,r)$ is called the neighborhood function, and its value is 1 for $i = r$ and falls off as the distance between the neurons r and i increases. α is the learning rate. The range of $N(i,r)$ and the value of α are reduced gradually as learning progresses. A common choice for $N(i,r)$ is

$$N(i,r) = \exp\left(-||d_i - d_r||^2/(2\sigma^2)\right) \tag{1.34}$$

where d_i and d_r are vectors indicating the positions of neurons i, r, and σ is a width parameter that controls the range of $N(i,r)$ and is gradually decreased as learning proceeds.

1.3.6 Processor Topologies and Hardware Platforms

This book deals with parallel mapping of the ANN models described in Section 1.3 on various hardware platforms and processor topologies. Details of the hardware platforms, topologies, and the mapping scheme employed are described in the appropriate chapters. Table 1.1 presents an overview.

ANN Model	Processor Topology	Hardware Details
Multilayer Feed-Forward Network BP Learning	Two-Dimensional Torus	Fujitsu AP1000 (MIMD) (Chapter 7)
	Ring	DSPs—MUSIC Machine (MIMD) (Chapter 10)
	Ring	Transputers (Heterogeneous MIMD) (Chapters 3 and 4)
	Two-Dimensional Lattice	DREAM Machine (SIMD) (Chapter 9)
	Vector Microprocessor	Spert–II Machine (Chapter 11)
Hopfield Network	Toroidal Lattice and Planar Lattice	Transputers (MIMD) (Chapter 8)
	Two-Dimensional Lattice	DREAM Machine (SIMD) (Chapter 9)
Recurrent BP Network	Ring	Transputers (MIMD) (Chapter 5)
Adaptive Resonance Theory Network	Ring	Transputers (MIMD) (Chapter 6)
Self-Organizing Map (SOM)	Vector Microprocessor	Spert–II Machine (Chapter 11)

TABLE 1.1. ANN Models and their parallel implementations.

References

[1] H. Drucker and Y. Le Cun, "Improving generalization performance using double backpropagation," *IEEE Transactions on Neural Networks*, vol. 3, no. 6, pp. 991–997, 1992.

[2] Y. Le Cun, et. al., "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[3] A. Waibel, "Consonant recognition by modular construction of large phonetic time-delay neural networks," *Advances in Neural Information Processing Systems*, pp. 215–223, 1989.

[4] C. J. Sheng Ma and J. Farmer, "An efficient EM-based training algorithm for feedforward neural networks," *Neural Networks*, vol. 10, no. 2, pp. 243–256, 1997.

[5] P. B. S. Osowski and M. Stodolski, "Fast second order learning algorithms for feedforward neural networks and its applications," *Neural Networks*, vol. 9, no. 9, pp. 1583–1596, 1996.

[6] R. W. Means, "High speed parallel hardware performance issues," in *IEEE International Conference on Neural Networks (ICNN'94)* (S. K. Rogers, ed.), pp. 10–16, June 28–July 2 1994.

[7] L. E. Atlas and Y. Suzuki, "Digital systems for artificial neural networks," *IEEE Circuits and Devices Magazine*, pp. 20–24, Nov. 1989.

[8] P. Treleven, *Neurocomputers*. Research Note 89/8, Department of Computer Science, University College London, January 1989.

[9] J. G. Solheim, *The RENNS approach to neural computing*. PhD thesis, Norwegian Institute of Technology, In preparation.

[10] L. A. Crowl, "How to measure, present and compare parallel performance," *IEEE Parallel & Distributed Technology*, vol. 2, no. 1, pp. 9–25, 1994.

[11] L. Utne, *Design of a reconfigurable neurocomputer Performance analysis by implementation of recurrent associative memories*. PhD thesis, Norwegian Institute of Technology, 1995. ISBN 82-7119-793-2.

[12] P. Ienne, "Quantitative comparison of architectures for digital neuro-computers," in *Proc. of IEEE Int. Conference on Neural Networks*, pp. 1987–1990, 1993.

[13] K. Asanovic, J. Beck, J. Feldman, N. Morgan, and J.Wawrzynek, "Designing a connectionist network supercomputer," *International Journal of Neural Systems*, vol. 4, pp. 317–326, December 1993. ISSN: 0129-0657.

[14] K. Hwang and F. A. Briggs, *Computer architecture and parallel processing*. McGraw-Hill Book Company, 5th ed., 1989.

[15] N. Morgan, et. al., "The ring array processor (RAP): A multiprocessing peripheral for connectionist applications," *Journal of Parallel and Distributed Computing*, vol. 14, 1992. Special Issue on Neural Networks.

[16] M. E. Azema-Barac, *A generic strategy for mapping neural network models on transputer-based machines*, pp. 244–249. IOS Press, 1992. In: Transputing in Numerical and neural network applications, G.L. Reijns and J. Luo, Eds.

[17] D. B. Davidson, "A parallel processing tutorial," *IEEE Antennas and Propagation Society Magazine*, pp. 6–19, April 1990.

[18] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., 1993.

[19] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas imminent in nervous activity," in *Neurocomputing: Foundations and research* (Anderson and Rosenberg, eds.), ch. 2, pp. 18–28, The MIT Press, 1988. Reprint of the "Bulletin of Mathematical Biophysics", 1943.

[20] R. F., *Principles of Neurodynamics*. Spartan Books, New York, 1962.

[21] P. D. Wasserman, *Neural Computing – Theory and Practice*. Van Nostrand Reinhold, 1989.

[22] M. Minsky and S. Papert, *Perceptrons*. The MIT Press, 1969.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representation by error propagation," in *Parallel Distributed Processing*, vol. 1, pp. 318–362, The MIT Press, 1986.

[24] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*. Englewood Cliffs, N.J.: Prentice-Hall, 1985.

[25] H. Paugam-Moisy, "Parallel neural computing based on neural network duplicating," in *Parallel algorithms for digital image processing, computer vision and neural networks* (I. Pitas, ed.), ch. 10, pp. 305–340, John Wiley & Sons, 1993.

[26] T. J. Sejnowski and C. R. Rosenberg, "Parallel networks that learn to pronounce English text," *Complex Systems*, vol. 1, pp. 145–168, 1987.

[27] S. Fahlman, "Faster-learning variations on back-propagation," in *Proc. of the 1988 Connectionist Models Summer School*, Carnegie-Mellom University, 1988.

[28] M. Møller, "Supervised learning on large redundant training sets," *Int. Journal of Neural Systems*, vol. 4, no. 1, pp. 15–25, 1993. World Scientific Publishing Company.

[29] J. Torresen, *Parallelization of Backpropagation Training for Feed-Forward Neural Networks*. PhD thesis, Norwegian University of Science and Technology, 1996. ISBN 82-7119-906-4.

[30] G. O. Nesvik, *An empirical study of selected learning algorithms for feed-forward neural networks*. PhD thesis, Norwegian Institute of Technology, 1993. ISBN 82-7119-548-4.

[31] J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," in *Proceedings of National Academy of Science*, vol. 79, (USA), pp. 2554–2558, 1982.

[32] J. Hopfield, "Neurons with graded response have collective computations properties like those of two-state neurons," in *Proceedings of National Academy of Science*, vol. 81, (USA), pp. 3088–3092, 1984.

[33] S. Haykin, *Neural Networks: A Comprehensive Foundation*. New York: MacMillan College Publishing Co, 1994.

[34] M. Cohen and S. Grossberg, "Absolute stability of global pattern information and parallel memory storage by competitive neural networks," *IEEE transaction of Systems, Man and Cybernetics*, vol. 13, pp. 815–826, 1983.

[35] J. J. Hopfield and D. W. Tank, "Neural computations of decisions in optimzation problems," *Biological Cybernetics*, vol. 52, pp. 141–152, 1985.

[36] D. Patterson, *Artificial Neural Networks, Theory and Aplications*. Prentice Hall, 1995.

[37] F. Pineda, "Generalization of backpropagation to recurrent neural networks," *Phys. rev. Letters*, vol. 59, pp. 2229–2232, 1987.

[38] L. Almedia, "Backpropagation in non-feedforward networks," in *Neural Computing Architecture* (I. Aleksander, ed.), pp. 74–91, North Oxford Academic, London, 1989.

[39] P. Werbos, "Generalization of backpropagtion with application to a recurrent gas market model," *Neural Networks*, vol. 1, pp. 339–356, 1988.

[40] A. Robison and F. Fallside, "Static and dynamic error propagation networks with applications to speech coding," in *Neural Information Processing Systems* (D. Z. Anderson, ed.), pp. 632–641, American Institute of Physics, New York, 1988.

[41] R. J. Williams and D. Zipser, "A learning algorithm for continually learning fully recurrent neural networks," *Neural Computations*, vol. 1, pp. 270–280, 1989.

[42] B. Pearlmutter, "Learning state space trajectories in recurrent neural networks," *Neural Computations*, vol. 1, pp. 263–269, 1989.

[43] G. Carpenter and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision, Graphics and Image Processing*, vol. 37, pp. 54–115, 1987.

[44] G. Carpenter and S. Grossberg, "ART2: Self-organizing of stable category recognition codes for analog input patterns," *Applied Optics*, vol. 26, pp. 4919–4930, 1987.

[45] G. Carpenter and S. Grossberg, "The ART of adaptive pattern recognition by self-organizing neural networks," *IEEE Computer*, vol. 21, pp. 77–88, March 1988.

[46] G. Carpenter and S. Grossberg, *Pattern Recognition by Self-organizing Neural Networks*. MIT Press, Cambridge, Mass, 1991.

[47] T. Kohonen, *Self-Organization and Associative Memory*. Springer-Verlag, Berlin, 3rd ed., 1989.

[48] A. K. J.Hertz and R. Palmer, *Introduction to the Theory of Neural Computation*. Addison Wesley, Reading, MA, 1991.