

Artificial Ne

27. Artificial Neural Network Models

Peter Tino, Lubica Benuskova, Alessandro Sperduti

We outline the main models and developments in the broad field of artificial neural networks (ANN). A brief introduction to biological neurons motivates the initial formal neuron model – the perceptron. We then study how such formal neurons can be generalized and connected in network structures. Starting with the biologically motivated layered structure of ANN (feed-forward ANN), the networks are then generalized to include feedback loops (recurrent ANN) and even more abstract generalized forms of feedback connections (recursive neuronal networks) enabling processing of structured data, such as sequences, trees, and graphs. We also introduce ANN models capable of forming topographic lower-dimensional maps of data (self-organizing maps). For each ANN type we out-

27.1	Biological Neurons.....	455
27.2	Perceptron	456
27.3	Multilayered Feed-Forward ANN Models	458
27.4	Recurrent ANN Models.....	460
27.5	Radial Basis Function ANN Models	464
27.6	Self-Organizing Maps.....	465
27.7	Recursive Neural Networks	467
27.8	Conclusion.....	469
	References.....	470

line the basic principles of training the corresponding ANN models on an appropriate data collection.

The human brain is arguably one of the most exciting products of evolution on Earth. It is also the most powerful information processing tool so far. Learning based on examples and parallel signal processing lead to emergent macro-scale behavior of neural networks in the brain, which cannot be easily linked to the behavior of individual micro-scale components (neurons). In this chapter, we will introduce *artificial neural network* (ANN) models motivated by the brain that can learn in the presence of a *teacher*. During the course of learning the teacher specifies what the right responses to input examples should be. In addition, we will also mention ANNs that can learn without a teacher, based on principles of self-organization.

To set the context, we will begin by introducing basic neurobiology. We will then describe the *perceptron* model, which, even though rather old and simple, is an

important building block of more complex *feed-forward ANN* models. Such models can be used to approximate complex non-linear functions or to learn a variety of association tasks. The feed-forward models are capable of processing patterns without temporal association. In the presence of temporal dependencies, e.g., when learning to predict future elements of a time series (with certain prediction horizon), the feed-forward ANN needs to be extended with a memory mechanism to account for temporal structure in the data. This will naturally lead us to *recurrent neural network* models (RNN), which besides feed-forward connections also contain feedback loops to preserve, in the form of the information processing state, information about the past. RNN can be further extended to *recursive ANNs* (RecNN), which can process structured data such as trees and acyclic graphs.

27.1 Biological Neurons

It is estimated that there are about 10^{12} neural cells (*neurons*) in the human brain. Two-thirds of the neurons

form a 4–6 mm thick cortex that is assumed to be the center of cognitive processes. Within each neuron com-

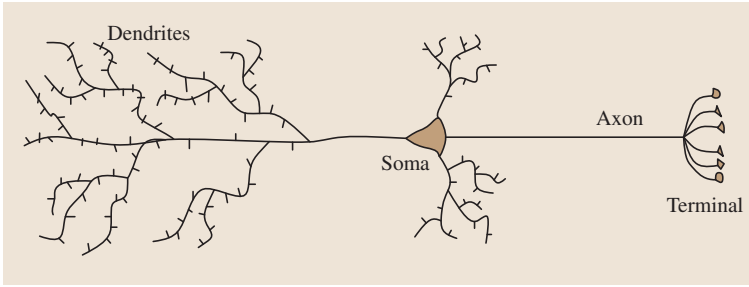


Fig. 27.1 Schematic illustration of the basic information processing structure of the biological neuron

plex biological processes take place, ensuring that it can process signals from other neurons, as well as send its own signals to them. The signals are of electro-chemical nature. In a simplified way, signals between the neurons can be represented by real numbers quantifying the *intensity* of the incoming or outgoing signals. The point of signal transmission from one neuron to the other is called the *synapse*. Within synapse the incoming signal can be reinforced or damped. This is represented by the *weight* of the synapse. A single neuron can have up to 10^3 – 10^5 such *points of entry* (synapses). The input to the neuron is organized along *dendrites* and the *soma* (Fig. 27.1). Thousands of dendrites form a rich tree-like structure on which most synapses reside.

Signals from other neurons can be either excitatory (positive) or inhibitory (negative), relayed via excitatory or inhibitory synapses. When the sum of the positive and negative contributions (signals) from other neurons, weighted by the synaptic weights, becomes greater than a certain excitation threshold, the neuron will generate an electric spike that will be transmitted over the output channel called the *axon*. At the end of

axon, there are thousands of output branches whose terminals form synapses on other neurons in the network. Typically, as a result of input excitation, the neuron can generate a series of spikes of some average frequency – about $1 - 10^2$ Hz. The frequency is proportional to the overall stimulation of the neuron.

The first principle of information coding and representation in the brain is *redundancy*. It means that each piece of information is processed by a redundant set of neurons, so that in the case of partial brain damage the information is not lost completely. As a result, and crucially – in contrast to conventional computer architectures, gradually increasing damage to the computing substrate (neurons plus their interconnection structure) will only result in gradually decreasing processing capabilities (*graceful degradation*). Furthermore, it is important what set of neurons participate in coding a particular piece of information (*distributed representation*). Each neuron can participate in coding of many pieces of information, in conjunction with other neurons. The information is thus associated with patterns of distributed activity on sets of neurons.

27.2 Perceptron

The perceptron is a simple neuron model that takes input signals (patterns) coded as (real) *input* vectors $\bar{x} = (x_1, x_2, \dots, x_{n+1})$ through the associated (real) vector of synaptic *weights* $\bar{w} = (w_1, w_2, \dots, w_{n+1})$. The output o is determined by

$$o = f(\text{net}) = f(\bar{w} \cdot \bar{x}) = f\left(\sum_{j=1}^{n+1} w_j x_j\right) = f\left(\sum_{j=1}^n w_j x_j - \theta\right), \quad (27.1)$$

where *net* denotes the weighted sum of inputs, (i. e., dot product of weight and input vectors), and f is the *activation function*. By convention, if there are n inputs to

the perceptron, the input $(n+1)$ will be fixed to -1 and the associated weight to $w_{n+1} = \theta$, which is the value of the *excitation threshold*.

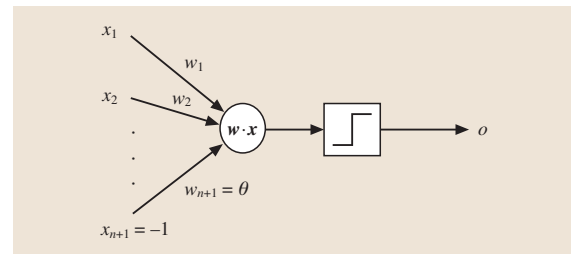


Fig. 27.2 Schematic illustration of the perceptron model

In 1958 Rosenblatt [27.1] introduced a discrete perceptron model with a bipolar activation function (Fig. 27.2)

$$f(\text{net}) = \text{sign}(\text{net}) = \begin{cases} +1 & \text{if net} \geq 0 \Leftrightarrow \sum_{j=1}^n w_j x_j \geq \theta \\ -1 & \text{if net} < 0 \Leftrightarrow \sum_{j=1}^n w_j x_j < \theta. \end{cases} \quad (27.2)$$

The *boundary* equation

$$\sum_{j=1}^n w_j x_j - \theta = 0, \quad (27.3)$$

parameterizes a hyperplane in n -dimensional space with normal vector \bar{w} .

The perceptron can classify input patterns into two classes, if the classes can indeed be separated by an $(n-1)$ -dimensional hyperplane (27.3). In other words, the perceptron can deal with *linearly-separable problems* only, such as logical functions AND or OR. XOR, on the other hand, is not linearly separable (Fig. 27.3). Rosenblatt showed that there is a simple training rule that will find the separating hyperplane, provided that the patterns are linearly separable.

As we shall see, a general rule for training many ANN models (not only the perceptron) can be formulated as follows: the weight vector \bar{w} is changed proportionally to the product of the input vector and a *learning signal* s . The learning signal s is a function of \bar{w} , \bar{x} , and possibly a teacher feedback d

$$s = s(\bar{w}, \bar{x}, d) \quad \text{or} \quad s = s(\bar{w}, \bar{x}). \quad (27.4)$$

In the former case, we talk about *supervised learning* (with direct guidance from a teacher); the latter case is known as *unsupervised learning*. The update of the j -th weight can be written as

$$w_j(t+1) = w_j(t) + \Delta w_j(t) = w_j(t) + \alpha s(t) x_j(t). \quad (27.5)$$

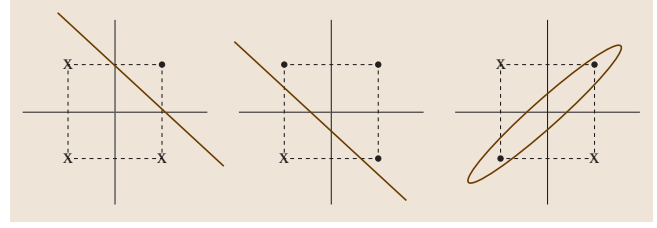


Fig. 27.3 Linearly separable and non-separable problems

The positive constant $0 < \alpha \leq 1$ is called the learning rate.

In the case of the perceptron, the learning signal is the disproportion (difference) between the desired (target) and the actual (produced by the model) response, $s = d - o = \delta$. The update rule is known as the δ (delta) rule

$$\Delta w_j = \alpha (d - o) x_j. \quad (27.6)$$

The same rule can, of course, be used to update the activation threshold $w_{n+1} = \theta$.

Consider a training set

$$A_{\text{train}} = \{(\bar{x}^1, d^1)(\bar{x}^2, d^2) \dots (\bar{x}^p, d^p) \dots (\bar{x}^P, d^P)\}$$

consisting of P (input,target) couples. The perceptron training algorithm can be formally written as:

- **Step 1:** Set $\alpha \in (0, 1)$. Initialize the weights randomly from $(-1, 1)$. Set the counters to $k = 1, p = 1$ (k indexes sweep through A_{train} , p indexes individual training patterns).
- **Step 2:** Consider input \bar{x}^p , calculate the output $o = \text{sign}(\sum_{j=1}^{n+1} w_j x_j^p)$.
- **Step 3:** Weight update: $w_j \leftarrow w_j + \alpha (d^p - o^p) x_j^p$, for $j = 1, \dots, n+1$.
- **Step 4:** If $p < P$, set $p \leftarrow p + 1$, go to step 2. Otherwise go to step 5.
- **Step 5:** Fix the weights and calculate the cumulative error E on A_{train} .
- **Step 6:** If $E = 0$, finish training. Otherwise, set $p = 1, k = k + 1$ and go to step 2. A new training epoch starts.

27.3 Multilayered Feed-Forward ANN Models

A breakthrough in our ability to construct and train more complex multilayered ANNs came in 1986, when Rumelhart et al. [27.2] introduced the error back-propagation method. It is based on making the transfer functions differentiable (hence the error functional to be minimized is differentiable as well) and finding a local minimum of the error functional by the gradient-based steepest descent method.

We will show derivation of the back-propagation algorithm for two-layer feed-forward ANN as demonstrated, e.g., in [27.3]. Of course, the same principles can be applied to a feed-forward ANN architecture with any (finite) number of layers. In feed-forward ANNs neurons are organized in layers. There are no connections among neurons within the same layer; connections only exist between successive layers. Each neuron from layer l has connections to each neuron in layer $l + 1$.

As has already been mentioned, the *activation functions* need to be differentiable and are usually of the sigmoid *shape*. The most common activation functions are

- *Unipolar sigmoid*:

$$f(\text{net}) = \frac{1}{1 + \exp(-\lambda \text{net})} \quad (27.7)$$

- *Bipolar sigmoid* (hyperbolic tangent):

$$f(\text{net}) = \frac{2}{1 + \exp(-\lambda \text{net})} - 1. \quad (27.8)$$

The constant $\lambda > 0$ determines steepness of the sigmoid curve and it is commonly set to 1. In the limit $\lambda \rightarrow \infty$ the bipolar sigmoid tends to the sign function (used in the perceptron) and the unipolar sigmoid tends to the step function.

Consider the single-layer ANN in Fig. 27.4. The output and input vectors are $\bar{y} = (y_1, \dots, y_j, \dots, y_J)$ and $\bar{o} = (o_1, \dots, o_k, \dots, o_K)$, respectively, where $o_k = f(\text{net}_k)$ and

$$\text{net}_k = \sum_{j=1}^J w_{kj} y_j. \quad (27.9)$$

Set $y_J = -1$ and $w_{kJ} = \theta_k$, a threshold for $k = 1, \dots, K$ output neurons. The desired output is $\bar{d} = (d_1, \dots, d_k, \dots, d_K)$.

After training, we would like, for all training patterns $p = 1, \dots, P$ from A_{train} , the model output to

closely resemble the desired values (target). The training problem is transformed to an optimization one by defining the error function

$$E_p = \frac{1}{2} \sum_{k=1}^K (d_{pk} - o_{pk})^2, \quad (27.10)$$

where p is the training point index. E_p is the sum of squares of errors on the output neurons. During learning we seek to find the weight setting that minimizes E_p . This will be done using the gradient-based steepest descent on E_p ,

$$\Delta w_{kj} = -\alpha \frac{\partial E_p}{\partial w_{kj}} = -\alpha \frac{\partial E_p}{\partial (\text{net}_k)} \frac{\partial (\text{net}_k)}{\partial w_{kj}} = \alpha \delta_{ok} y_j, \quad (27.11)$$

where α is a positive learning rate. Note that $-\partial E_p / \partial (\text{net}_k) = \delta_{ok}$, which is the generalized training signal on the k -th output neuron. The partial derivative $\partial (\text{net}_k) / \partial w_{kj}$ is equal to y_j (27.9). Furthermore,

$$\delta_{ok} = -\frac{\partial E_p}{\partial (\text{net}_k)} = -\frac{\partial E_p}{\partial o_k} \frac{\partial o_k}{\partial (\text{net}_k)} = (d_{pk} - o_{pk}) f'_k, \quad (27.12)$$

where f'_k denotes the derivative of the activation function with respect to net_k . For the unipolar sigmoid (27.7), we have $f'_k = o_k(1 - o_k)$. For the bipolar sigmoid (27.8), $f'_k = (1/2)(1 - o_k^2)$. The rule for updating the j -th weight of the k -th output neuron reads as

$$\Delta w_{kj} = \alpha (d_{pk} - o_{pk}) f'_k y_j, \quad (27.13)$$

where $(d_{pk} - o_{pk}) f'_k = \delta_{ok}$ is *generalized error signal* flowing back through all connections ending in the k -th output neuron. Note that if we put $f'_k = 1$, we would obtain the perceptron learning rule (27.6).

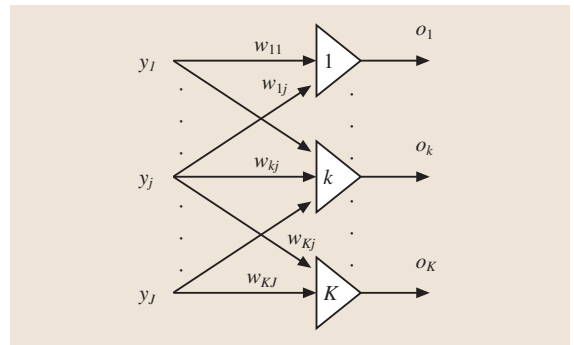


Fig. 27.4 A single-layer ANN

We will now extend the network with another layer, called the hidden layer (Fig. 27.5).

Input to the network is identical with the input vector $\bar{x} = (x_1, \dots, x_i, \dots, x_I)$ for the hidden layer. The output neurons process as inputs the outputs $\bar{y} = (y_1, \dots, y_j, \dots, y_J)$, $y_j = f(\text{net}_j)$ from the hidden layer. Hence,

$$\text{net}_j = \sum_{i=1}^I v_{ji} x_i. \quad (27.14)$$

As before, the last (in this case the I -th) input is fixed to -1 . Recall that the same holds for the output of the J -th hidden neuron. Activation thresholds for hidden neurons are $v_{jI} = \theta_j$, for $j = 1, \dots, J$.

Equations (27.11)–(27.13) describe modification of weights from the hidden to the output layer. We will now show how to modify weights from the input to the hidden layer. We would still like to minimize E_p (27.10) through the steepest descent.

The hidden weight v_{ji} will be modified as follows

$$\Delta v_{ji} = -\alpha \frac{\partial E_p}{\partial v_{ji}} = -\alpha \frac{\partial E_p}{\partial (\text{net}_j)} \frac{\partial (\text{net}_j)}{\partial v_{ji}} = \alpha \delta_{yj} x_i. \quad (27.15)$$

Again, $-\partial E_p / \partial (\text{net}_j) = \delta_{yj}$ is the generalized training signal on the j -th hidden neuron that should flow on the input weights. As before, $\partial (\text{net}_j) / \partial v_{ji} = x_i$ (27.14). Furthermore,

$$\delta_{yj} = -\frac{\partial E_p}{\partial (\text{net}_j)} = -\frac{\partial E_p}{\partial y_j} \frac{\partial y_j}{\partial (\text{net}_j)} = -\frac{\partial E_p}{\partial y_j} f'_j, \quad (27.16)$$

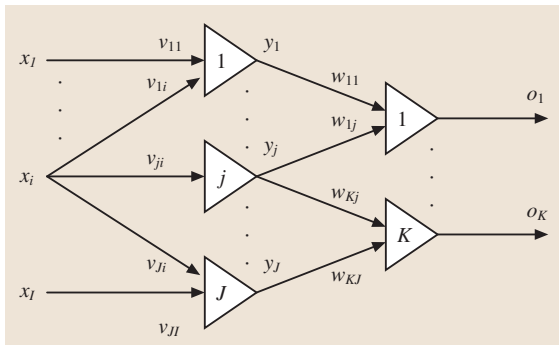


Fig. 27.5 A two-layer feed-forward ANN

where f'_j is the derivative of the activation function in the hidden layer with respect to net_j ,

$$\begin{aligned} \frac{\partial E_p}{\partial y_j} &= -\sum_{k=1}^K (d_{pk} - o_{pk}) \frac{\partial f(\text{net}_k)}{\partial y_j} \\ &= -\sum_{k=1}^K (d_{pk} - o_{pk}) \frac{\partial f(\text{net}_k)}{\partial (\text{net}_k)} \frac{\partial (\text{net}_k)}{\partial y_j}. \end{aligned} \quad (27.17)$$

Since f'_k is the derivative of the output neuron sigmoid with respect to net_k and $\partial (\text{net}_k) / \partial y_j = w_{kj}$ (27.9), we have

$$\frac{\partial E_p}{\partial y_j} = -\sum_{k=1}^K (d_{pk} - o_{pk}) f'_k w_{kj} = -\sum_{k=1}^K \delta_{ok} w_{kj}. \quad (27.18)$$

Plugging this to (27.16) we obtain

$$\delta_{yj} = \left(\sum_{k=1}^K \delta_{ok} w_{kj} \right) f'_j. \quad (27.19)$$

Finally, the weights from the input to the hidden layer are modified as follows

$$\Delta v_{ji} = \alpha \left(\sum_{k=1}^K \delta_{ok} w_{kj} \right) f'_j x_i. \quad (27.20)$$

Consider now the general case of m hidden layers. For the n -th hidden layer we have

$$\Delta v_{ji}^n = \alpha \delta_{yj}^n x_i^{n-1}, \quad (27.21)$$

where

$$\delta_{yj}^n = \left(\sum_{k=1}^K \delta_{ok}^{n+1} w_{kj}^{n+1} \right) (f_j^n)', \quad (27.22)$$

and $(f_j^n)'$ is the derivative of the activation function of the n -layer with respect to net_j^n .

Often, the learning speed can be improved by using the so-called momentum term

$$\begin{aligned} \Delta w_{kj}(t) &\leftarrow \Delta w_{kj}(t) + \mu \Delta w_{kj}(t-1), \\ \Delta v_{ji}(t) &\leftarrow \Delta v_{ji}(t) + \mu \Delta v_{ji}(t-1), \end{aligned} \quad (27.23)$$

where $\mu \in (0, 1)$ is the momentum rate.

Consider a training set

$$A_{\text{train}} = \{(\bar{x}^1, \bar{d}^1)(\bar{x}^2, \bar{d}^2) \dots (\bar{x}^p, \bar{d}^p) \dots (\bar{x}^P, \bar{d}^P)\}.$$

The back-propagation algorithm for training feed-forward ANNs can be summarized as follows:

- **Step 1:** Set $\alpha \in (0, 1)$. Randomly initialize weights to *small* values, e.g., in the interval $(-0.5, 0.5)$. Counters and the error are initialized as follows: $k = 1, p = 1, E = 0$. E denotes the accumulated error across training patterns

$$E = \sum_{p=1}^P E_p, \quad (27.24)$$

where E_p is given in (27.10). Set a tolerance threshold ε for the error. The threshold will be used to stop the training process.

- **Step 2:** Apply input \vec{x}^p and compute the corresponding \vec{y}^p and \vec{o}^p .
- **Step 3:** For every output neuron, calculate δ_{ok} (27.12), for hidden neuron determine δ_{yj} (27.19).
- **Step 4:** Modify the weights $w_{kj} \leftarrow w_{kj} + \alpha \delta_{ok} y_j$ and $v_{ji} \leftarrow v_{ji} + \alpha \delta_{yj} x_i$.
- **Step 5:** If $p < P$, set $p = p + 1$ and go to step 2. Otherwise go to step 6.
- **Step 6:** Fixing the weights, calculate E . If $E < \varepsilon$, stop training, otherwise permute elements of A_{train} , set $E = 0, p = 1, k = k + 1$, and go to step 2.

Consider a feed-forward ANN with fixed weights and single output unit. It can be considered a real-valued function G on I -dimensional vectorial inputs,

$$G(\vec{x}) = f \left(\sum_{j=1}^J w_{jf} \left(\sum_{i=1}^I v_{ji} x_i \right) \right).$$

There has been a series of results showing that such a parameterized function class is *sufficiently rich* in the space of *reasonable* functions (see, e.g., [27.4]). For example, for any smooth function F over a compact domain and a precision threshold ε , for sufficiently large number J of hidden units there is a weight setting so that G is not further away from F than ε (in L-2 norm).

When training a feed-forward ANN a key decision must be made about how complex the model should be. In other words, how many hidden units J one should use. If J is too small, the model will be too rigid (high

bias) and it will not be able to sufficiently adapt to the data. However, under different samples from the same data generating process, the resulting trained models will vary relatively little (low variance). On the other hand, if J is too high, the model will be too complex, modeling even such irrelevant features of the data such as output noise. The particular data will be interpolated exactly (low bias), but the variability of fitted models under different training samples from the same process will be immense. It is, therefore, important to set J to an *optimal* value, reflecting the complexity of the data generating process. This is usually achieved by splitting the data into three disjoint sets – *training*, *validation*, and *test sets*. Models with different numbers of hidden units are trained on the training set, their performance is then checked on a held-out validation set. The *optimal* number of hidden units is selected based on the (smallest) validation error. Finally, the test set is used for independent comparison of selected models from different model classes.

If the data set is not large enough, one can perform such a *model selection* using *k-fold cross-validation*. The data for model construction (this data would be considered training and validation sets in the scenario above) is split into k disjoint folds. One fold is selected as the validation fold, the other $k - 1$ will be used for training. This is repeated k times, yielding k estimates of the validation error. The validation error is then calculated as the mean of those k estimates.

We have described data-based methods for model selection. Other alternatives are available. For example, by turning an ANN into a probabilistic model (e.g., by including an appropriate output noise model), under some prior assumptions on weights (e.g., a-priori small weights are preferred), one can perform Bayesian model selection (through *model evidence*) [27.5].

There are several seminal books on feed-forward ANNs with well-documented theoretical foundations and practical applications, e.g., [27.3, 6, 7]. We refer the interested reader to those books as good starting points as the breadth of theory and applications of feed-forward ANNs is truly immense.

27.4 Recurrent ANN Models

Consider a situation where the associations in the training set we would like to learn are of the following (abstract) form: $a \rightarrow \alpha, b \rightarrow \beta, b \rightarrow \alpha, b \rightarrow \gamma, c \rightarrow \alpha, c \rightarrow \gamma, d \rightarrow \alpha$, etc., where the Latin and Greek letters stand for input and output vectors, respectively. It is

clear that now for one input item there can be different output associations, depending on the *temporal context* in which the training items are presented. In other words, *the model output is determined not only by the input, but also by the history of presented items so far*.

Obviously, the feed-forward ANN model described in the previous section cannot be used in such cases and the model must be further extended so that the temporal context is properly represented.

The architecturally simplest solution is provided by the so-called *time delay neural network* (TDNN) (Fig. 27.6). The input *window into the past* has a finite length D . If the output is an estimate of the next item of the input time series, such a network realizes a non-linear autoregressive model of order D .

If we are lucky, even such a simple solution can be sufficient to capture the temporal structure hidden in the data. An advantage of the TDNN architecture is that some training methods developed for feed-forward networks can be readily used. A disadvantage of TDNN networks is that fixing a finite order D may not be adequate for modeling the temporal structure of the data generating source. TDNN enables the feed-forward ANN to see, besides the current input at time t , the other inputs from the past up to time $t - D$. Of course, during the training, it is now imperative to preserve the order of training items in the training set. TDNN has been successfully applied in many fields where spatial-temporal structures are naturally present, such as robotics, speech recognition, etc. [27.8, 9].

In order to extend the ANN architecture so that the *variable* (even unbounded) length of input window can be flexibly considered, we need a different way of capturing the temporal context. This is achieved through the so-called *state space formulation*. In this case, we will need to change our outlook on training. The new architectures of this type are known as *recurrent neural networks* (RNN).

As in feed-forward ANNs, there are connections between the successive layers. In addition, and in contrast to feed-forward ANNs, connections between neurons of the same layer are allowed, but subject to a *time de-*

lay. It also may be possible to have connections from a higher-level layer to a lower layer, again subject to a time delay. In many cases it is, however, more convenient to introduce an additional fictional *context layer* that contains delayed activations of neurons from the selected layer(s) and represent the resulting RNN architecture as a feed-forward architecture with some fixed one-to-one delayed connections. As an example, consider the so-called *simple recurrent network* (SRN) of Elman [27.10] shown in Fig. 27.7. The output of SRN at time t is given by

$$\begin{aligned} o_k^{(t)} &= f \left(\sum_{j=1}^J m_{kj} y_j^{(t)} \right), \\ y_j^{(t)} &= f \left(\sum_{i=1}^J w_{ji} y_i^{(t-1)} + \sum_{i=1}^I v_{ji} x_i^{(t)} \right). \end{aligned} \quad (27.25)$$

The hidden layer constitutes the state of the input-driven dynamical system whose role it is to represent the relevant (with respect to the output) information

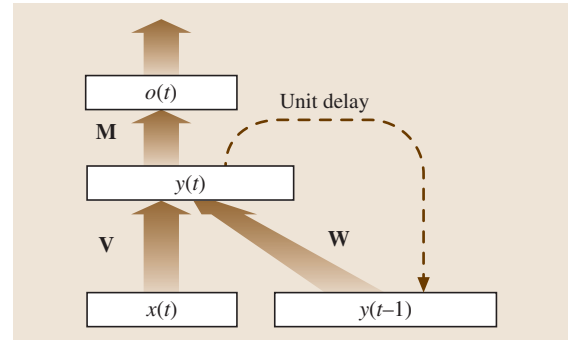


Fig. 27.7 Schematic depiction of the SRN architecture

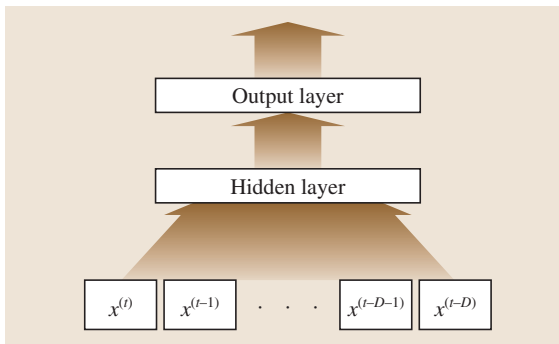


Fig. 27.6 TDNN of order D

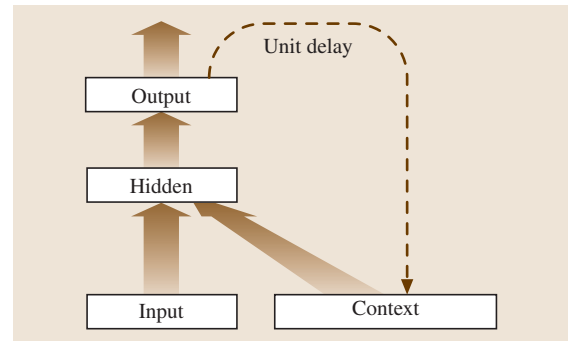


Fig. 27.8 Schematic depiction of the Jordan's RNN architecture

about the input history seen so far. The state (as in generic state space model) is updated recursively.

Many variations on such architectures with time-delayed feedback loops exist. For example, *Jordan* [27.11] suggested to feed back the outputs as the relevant temporal context, or *Bengio et al.* [27.12] mixed the temporal context representations of SRN and the Jordan network into a single architecture. Schematic representations of these architectures are shown in Figs. 27.8 and 27.9.

Training in such architectures is more complex than training of feed-forward ANNs. The principal problem is that changes in weights propagate in time and this needs to be explicitly represented in the update rules. We will briefly mention two approaches to training RNNs, namely *back-propagation through time* (BPTT) [27.13] and *real-time recurrent learning* (RTRL) [27.14]. We will demonstrate BPTT on a clas-

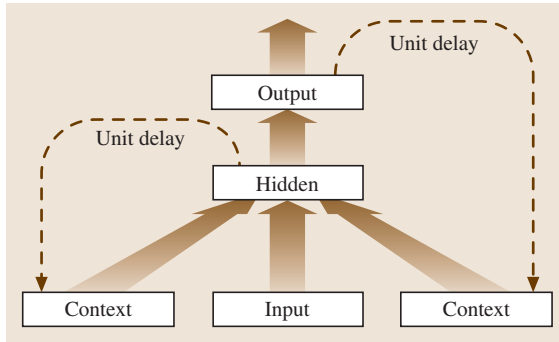


Fig. 27.9 Schematic depiction of the Bengio's RNN architecture

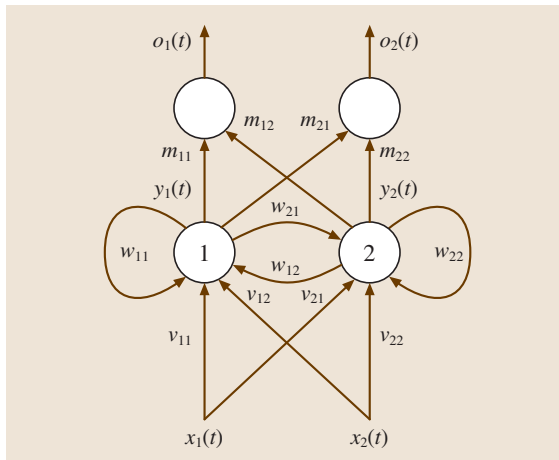


Fig. 27.10 A two-neuron SRN

sification task, where the label of the input sequence is known only after T time steps (i.e., after T input items have been processed). The RNN is unfolded in time to form a feed-forward network with T hidden layers. Figure 27.10 shows a simple two-neuron RNN and Fig. 27.11 represents its unfolded form for $T = 2$ time steps.

The first input comes at time $t = 1$ and the last at $t = T$. Activities of context units are initialized at the beginning of each sequence to some fixed numbers. The unfolded network is then trained as a feed-forward network with T hidden layers. At the end of the sequence, the model output is determined as

$$o_k^{(T)} = f \left(\sum_{j=1}^J m_{kj}^{(T)} y_j^{(T)} \right),$$

$$y_j^{(t)} = f \left(\sum_{i=1}^J w_{ji}^{(t)} y_i^{(t-1)} + \sum_{i=1}^I v_{ji}^{(t)} x_i^{(t)} \right). \quad (27.26)$$

Having the model output enables us to compute the error

$$E(T) = \frac{1}{2} \sum_{k=1}^K \left(d_k^{(T)} - o_k^{(T)} \right)^2. \quad (27.27)$$

The hidden-to-output weights are modified according to

$$\Delta m_{kj}^{(T)} = -\alpha \frac{\partial E(T)}{\partial m_{kj}} = \alpha \delta_k^{(T)} y_j^{(T)}, \quad (27.28)$$

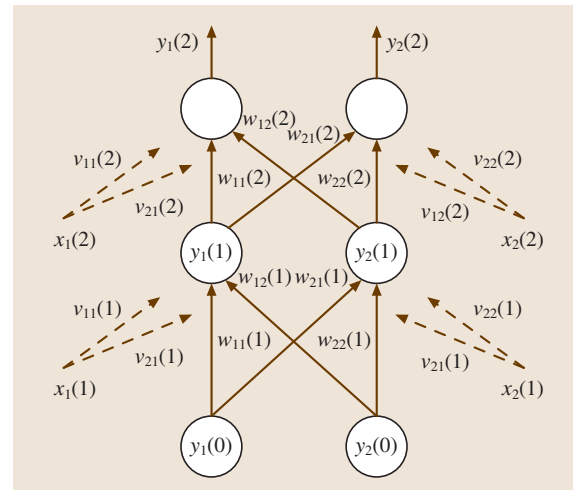


Fig. 27.11 Two-neuron SRN unfolded in time for $T = 2$

where

$$\delta_k^{(T)} = \left(d_k^{(T)} - o_k^{(T)}\right) f' \left(\text{net}_k^{(T)}\right). \quad (27.29)$$

The other weight updates are calculated as follows

$$\Delta w_{hj}^{(T)} = -\alpha \frac{\partial E(T)}{\partial w_{hj}} = \alpha \delta_h^{(T)} y_j^{(T-1)}; \quad (27.30)$$

$$\delta_h^{(T)} = \left(\sum_{k=1}^K \delta_k^{(T)} m_{kh}^{(T)} \right) f' \left(\text{net}_h^{(T)} \right)$$

$$\Delta v_{ji}^{(T)} = -\alpha \frac{\partial E(T)}{\partial v_{ji}} = \alpha \delta_j^{(T)} x_i^{(T)}; \quad (27.31)$$

$$\delta_j^{(T)} = \left(\sum_{k=1}^K \delta_k^{(T)} m_{kj}^{(T)} \right) f' \left(\text{net}_j^{(T)} \right)$$

$$\Delta w_{hj}^{(T-1)} = \alpha \delta_h^{(T-1)} y_j^{(T-2)}; \quad (27.32)$$

$$\delta_h^{(T-1)} = \left(\sum_{j=1}^J \delta_j^{(T-1)} w_{jh}^{(T-1)} \right) f' \left(\text{net}_h^{(T-1)} \right)$$

$$\Delta v_{ji}^{(T-1)} = \alpha \delta_j^{(T-1)} x_i^{(T-1)}; \quad (27.33)$$

$$\delta_j^{(T-1)} = \left(\sum_{h=1}^J \delta_h^{(T-1)} w_{hj}^{(T-1)} \right) f' \left(\text{net}_j^{(T-1)} \right),$$

etc. The final weight updates are the averages of the T partial weight update suggestions calculated on the unfolded network

$$\Delta w_{hj} = \frac{\sum_{t=1}^T \Delta w_{hj}^{(t)}}{T} \quad \text{and} \quad \Delta v_{ji} = \frac{\sum_{t=1}^T \Delta v_{ji}^{(t)}}{T}. \quad (27.34)$$

For every new training sequence (of possibly different length T) the network is unfolded to the desired length and the weight update process is repeated. In some cases (e.g., continual prediction on time series), it is necessary to set the maximum unfolding length L that will be used in every update step. Of course, in such cases we can lose vital information from the past. This problem is eliminated in the RTRL methodology.

Consider again the SRN architecture in Fig. 27.6. In RTRL the weights are updated *on-line*, i. e., at every

time step t

$$\Delta w_{kj}^{(t)} = -\alpha \frac{\partial E^{(t)}}{\partial w_{kj}^{(t)}}, \quad (27.35)$$

$$\Delta v_{ji}^{(t)} = -\alpha \frac{\partial E^{(t)}}{\partial v_{ji}^{(t)}},$$

$$\Delta m_{jl}^{(t)} = -\alpha \frac{\partial E^{(t)}}{\partial m_{jl}^{(t)}}.$$

The updates of hidden-to-output weights are straightforward

$$\Delta m_{kj}^{(t)} = \alpha \delta_k^{(t)} y_j^{(t)} = \alpha \left(d_k^{(t)} - o_k^{(t)} \right) f'_k \left(\text{net}_k^{(t)} \right) y_j^{(t)}. \quad (27.36)$$

For the other weights we have

$$\Delta v_{ji}^{(t)} = \alpha \sum_{k=1}^K \left(\delta_k^{(t)} \sum_{h=1}^J w_{kh} \frac{\partial y_h^{(t)}}{\partial v_{ji}^{(t)}} \right), \quad (27.37)$$

$$\Delta w_{ji}^{(t)} = \alpha \sum_{k=1}^K \left(\delta_k^{(t)} \sum_{h=1}^J w_{kh} \frac{\partial y_h^{(t)}}{\partial w_{ji}^{(t)}} \right),$$

where

$$\frac{\partial y_h^{(t)}}{\partial v_{ji}^{(t)}} = f' \left(\text{net}_h^{(t)} \right) \left(x_i^{(t)} \delta_{jh}^{\text{Kron.}} + \sum_{l=1}^J w_{hl} \frac{\partial y_l^{(t-1)}}{\partial v_{ji}^{(t)}} \right) \quad (27.38)$$

$$\frac{\partial y_h^{(t)}}{\partial w_{ji}^{(t)}} = f' \left(\text{net}_h^{(t)} \right) \left(x_i^{(t)} \delta_{jh}^{\text{Kron.}} + \sum_{l=1}^J w_{hl} \frac{\partial y_l^{(t-1)}}{\partial w_{ji}^{(t)}} \right), \quad (27.39)$$

and $\delta_{jh}^{\text{Kron.}}$ is the Kronecker delta ($\delta_{jh}^{\text{Kron.}} = 1$, if $j = h$; $\delta_{jh}^{\text{Kron.}} = 0$ otherwise). The partial derivatives required for the weight updates can be recursively updated using (27.37)–(27.39). To initialize training, the partial derivatives at $t = 0$ are usually set to 0.

There is a well-known problem associated with gradient-based parameter fitting in recurrent networks (and, in fact, in any parameterized state space models of similar form) [27.15]. In order to *latch* an important piece of past information for future use, the state-transition dynamics (27.25) should have an attractive set.

However, in the neighborhood of such an attractive set, the derivatives of the dynamic state-transition map

vanish. Vanishingly small derivatives cannot be reliably propagated back through time in order to form a useful latching set. This is known as the *information latching problem*. Several suggestions for dealing with information latching problem have been made, e.g., [27.16]. The most prominent include *long short term memory* (LSTM) RNN [27.17] and *reservoir computation* models [27.18].

LSTM models operate with a specially designed formal neuron model that contains so-called *gate* units. The gates determine when the input is *significant* (in terms of the task given) to be remembered, whether the neuron should continue to remember the value, and when the value should be output. The LSTM architecture is especially suitable for situations where there are long time intervals of unknown size between *important* events. LSTM models have been shown to provide superior results over traditional RNNs in a variety of applications (e.g., [27.19, 20]).

Reservoir computation models try to avoid the information latching problem by fixing the state-transition part of the RNN. Only linear readout from the state activations (hidden recurrent layer) producing the output is fit to the data. The state space with the as-

sociated dynamic state transition structure is called the *reservoir*. The main idea is that the reservoir should be sufficiently complex so as to capture a large number of potentially useful features of the input stream that can be then exploited by the simple readout.

The reservoir computing models differ in how the fixed reservoir is constructed and what form the readout takes. For example, *echo state networks* (ESN) [27.21] have fixed RNN dynamics (27.25), but with a linear hidden-to-output layer map. *Liquid state machines* (LSM) [27.22] also have (mostly) linear readout, but the reservoirs are realized through the dynamics of a set of coupled spiking neuron models. *Fractal prediction machines* (FPM) [27.23] are reservoir RNN models for processing discrete sequences. The reservoir dynamics is driven by an affine iterative function system and the readout is constructed as a collection of multinomial distributions. Reservoir models have been successfully applied in many practical applications with competitive results, e.g., [27.21, 24, 25].

Several books that are solely dedicated to RNNs have appeared, e.g., [27.26–28] and they contain a much deeper elaboration on theory and practice of RNNs than we were able to provide here.

27.5 Radial Basis Function ANN Models

In this section we will introduce another implementation of the idea of feed-forward ANN. The activations of hidden neurons are again determined by the *closeness* of inputs $\bar{x} = (x_1, x_2, \dots, x_n)$ to weights $\bar{c} = (c_1, c_2, \dots, c_n)$. Whereas in the feed-forward ANN in Sect. 27.3, the closeness is determined by the dot-product of \bar{x} and \bar{c} , followed by the sigmoid activation function, in *radial basis function* (RBF) networks the closeness is determined by the squared Euclidean distance of \bar{x} and \bar{c} , transferred through the inverse exponential. The output of the j -th hidden unit with input weight vector \bar{c}_j is given by

$$\varphi_j(\bar{x}) = \exp\left(-\frac{\|\bar{x} - \bar{c}_j\|^2}{\sigma_j^2}\right), \quad (27.40)$$

where σ_j is the *activation strength* parameter of the j -th hidden unit and determines the width of the spherical (un-normalized) Gaussian. The output neurons are usually linear (for regression tasks)

$$o_k(\bar{x}) = \sum_{j=1}^J w_{kj} \varphi_j(\bar{x}). \quad (27.41)$$

The RBF network in this form can be simply viewed as a form of kernel regression. The J functions φ_j form a set of J linearly independent basis functions (e.g., if all the centers \bar{c}_j are different) whose span (the set of all their linear combinations) forms a linear subspace of functions that are realizable by the given RBF architecture (with given centers \bar{c}_j and kernel widths σ_j).

For the training of RBF networks, it is important that the basis functions $\varphi_j(\bar{x})$ cover the structure of the inputs space *faithfully*. Given a set of training inputs \bar{x}^p from $A_{\text{train}} = \{(\bar{x}^1, \bar{d}^1)(\bar{x}^2, \bar{d}^2) \dots (\bar{x}^p, \bar{d}^p) \dots (\bar{x}^P, \bar{d}^P)\}$, many RBF-ANN training algorithms determine the centers \bar{c}_j and widths σ_j based on the inputs $\{\bar{x}^1, \bar{x}^2, \dots, \bar{x}^P\}$ only. One can employ different clustering algorithms, e.g., k -means [27.29], which attempts to position the centers among the training inputs so that the overall sum of (Euclidean) distances between the centers and the inputs they represent (i.e., the inputs falling in their respective Voronoi compartments – the set of inputs for which the current center is the closest among all the centers) is minimized:

- *Step 1:* Set J , the number of hidden units. The optimum value of J can be obtained through a model selection method, e.g., cross-validation.
- *Step 2:* Randomly select J training inputs that will form the initial positions of the J centers \bar{c}_j .
- *Step 3:* At time step t :
 - a) Pick a training input $\bar{x}(t)$ and find the center $\bar{c}(t)$ closest to it.
 - b) Shift the center $\bar{c}(t)$ towards $\bar{x}(t)$

$$\bar{c}(t) \leftarrow \bar{c}(t) + \rho(t)(\bar{x}(t) - \bar{c}(t)), \quad \text{where } 0 \leq \rho(t) \leq 1. \quad (27.42)$$

The learning rate $\rho(t)$ usually decreases in time towards zero. The training is stopped once the centers settle in their positions and move only slightly (some norm of weight updates is below a certain threshold). Since k -means is guaranteed to find only locally optimal solutions, it is worth re-initializing the centers and re-running the algorithm several times, keeping the solution with the lowest quantization error.

Once the centers are in their positions, it is easy to determine the RBF widths, and once this is done, the

output weights can be solved using methods of linear regression.

Of course, it is more optimal to position the centers with respect to *both* the inputs and target outputs in the training set. This can be formulated, e.g., as a gradient descent optimization. Furthermore, covering of the input space with spherical Gaussian kernels may not be optimal, and algorithms have been developed for learning of general covariance structures. A comprehensive review of RBF networks can be found, e.g., in [27.30].

Recently, it was shown that if enough hidden units are used, their centers can be set randomly at very little cost, and determination of the only remaining free parameters – output weights – can be done cheaply and in a closed form through linear regression. Such architectures, known as *extreme learning machines* [27.31] have shown surprisingly high performance levels. The idea of extreme learning machines can be considered as being analogous to the idea of reservoir computation, but in the *static* setting. Of course, extreme learning machines can be built using other implementations of feed-forward ANNs, such as the sigmoid networks of Sect. 27.3.

27.6 Self-Organizing Maps

In this section we will introduce ANN models that learn without any signal from a teacher, i.e., learning is based solely on training inputs – there are no output targets. The ANN architecture designed to operate in this setting was introduced by *Kohonen* under the name *self-organizing map* (SOM) [27.32]. This model is motivated by organization of neuron sensitivities in the brain cortex.

In Fig. 27.12a we show schematic illustration of one of the principal organizations of biological neural networks. In the bottom layer (grid) there are receptors representing the inputs. Every element of the inputs (each receptor) has forward connections to all neurons in the upper layer representing the cortex. The neurons are organized spatially on a grid. Outputs of the neurons represent activation of the SOM network. The neurons, besides receiving connections from the input receptors, have a lateral interconnection structure among themselves, with connections that can be excitatory, or inhibitory. In Fig. 27.12b we show a formal SOM architecture – neurons spatially organized on a grid receive inputs (elements of input vectors) through connections with synaptic weights.

A particular feature of the SOM is that it can map the training set on the neuron grid in a manner that preserves the training set's topology – two input patterns *close* in the input space will activate neurons most that are close on the SOM grid. Such *topological mapping* of inputs (feature mapping) has been observed in biological neural networks [27.32] (e.g., visual maps, orientation maps of visual contrasts, or auditory maps, frequency maps of acoustic stimuli).

Teuvo Kohonen presented one possible realization of the Hebb rule that is used to train SOM. Input

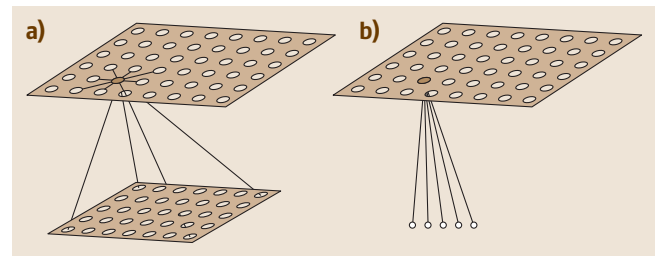


Fig. 27.12a,b Schematic representation of the SOM ANN architectures

weights of the neurons are initialized as small random numbers. Consider a training set of inputs, $A_{\text{train}} = \{\bar{x}_p\}_{p=1}^P$ and linear neurons

$$o_i = \sum_{j=1}^m w_{ij} x_j = \bar{w}_i \bar{x}, \quad (27.43)$$

where m is the input dimension and $i = 1, \dots, n$. Training inputs are presented in random order. At each training step, we find the (winner) neuron with the weight vector *most similar* to the current input \bar{x} . The measure of similarity can be based on the dot product, i. e., the index of the winner neuron is $i^* = \arg \max (\bar{w}_i^T \bar{x})$, or the (Euclidean) distance $i^* = \arg \min_i \|\bar{x} - \bar{w}_i\|$. After identifying the winner the learning continues by adapting the winner's weights *along with the weights all its neighbors on the neuron grid*. This will ensure that nearby neurons on the grid will eventually represent similar inputs in the input space. This is moderated by a *neighborhood function* $h(i^*, i)$ that, given a winner neuron index i^* , quantifies how many other neurons on the grid should be adapted

$$\bar{w}_i(t+1) = \bar{w}_i(t) + \alpha(t) \cdot h(i^*, i) \cdot (\bar{x}(t) - \bar{w}_i(t)). \quad (27.44)$$

The learning rate $\alpha(t) \in (0, 1)$ decays in time as $1/t$, or $\exp(-kt)$, where k is a positive time scale constant. This ensures convergence of the training process. The simplest form of the neighborhood function operates with rectangular neighborhoods,

$$h(i^*, i) = \begin{cases} 1, & \text{if } d_M(i^*, i) \leq \lambda(t) \\ 0, & \text{otherwise,} \end{cases} \quad (27.45)$$

where $d_M(i^*, i)$ represents the $2\lambda(t)$ (*Manhattan*) distance between neurons i^* and i on the map grid. The neighborhood size $2\lambda(t)$ should decrease in time, e.g., through an exponential decay as or $\exp(-qt)$, with time scale $q > 0$. Another often used neighborhood function is the Gaussian kernel

$$h(i^*, i) = \exp\left(-\frac{d_E^2(i^*, i)}{\lambda^2(t)}\right), \quad (27.46)$$

where $d_E(i^*, i)$ is the Euclidean distance between i^* and i on the grid, i. e., $d_E(i^*, i) = \|\bar{r}_{i^*} - \bar{r}_i\|$, where \bar{r}_i is the co-ordinate vector of the i -th neuron on the grid SOM.

Training of SOM networks can be summarized as follows:

- *Step 1:* Set α_0 , λ_0 and t_{\max} (maximum number of iterations). Randomly (e.g., with uniform distribution) generate the synaptic weights (e.g., from $(-0.5, 0.5)$). Initialize the counters: $t = 0$, $p = 1$; t indexes time steps (iterations) and p is the input pattern index.
- *Step 2:* Take input \bar{x}^p and find the corresponding winner neuron.
- *Step 3:* Update the weights of the winner and its topological neighbors on the grid (as determined by the neighborhood function). Increment t .
- *Step 4:* Update α and λ .
- *Step 5:* If $p < P$, set $p \leftarrow p + 1$, go to step 2 (we can also use randomized selection), otherwise go to step 6.
- *Step 6:* If $t = t_{\max}$, finish the training process. Otherwise set $p = 1$ and go to step 2. A new training epoch begins.

The SOM network can be used as a tool for non-linear data visualization (grid dimensions 1, 2, or 3). In general, SOM implements constrained vector quantization, where the codebook vectors (vector quantization centers) cannot move freely in the data space during adaptation, but are constrained to lie on a lower dimensional manifold Ψ in the data space. The dimensionality of Ψ is equal to the dimensionality of the neural grid. The neural grid can be viewed as a discretized version of the local co-ordinate system \mathcal{Y} (e.g., computer screen) and the weight vectors in the data space (connected by the neighborhood structure on the neuron grid) as its image in the data space. In this interpretation, the neuron positions on the grid represent co-ordinate functions (in the sense of differential geometry) mapping elements of the manifold Ψ to the coordinate system \mathcal{Y} . Hence, the SOM algorithm can also be viewed as one particular implementation of manifold learning.

There have been numerous successful applications of SOM in a wide variety of applications, e.g., in image processing, computer vision, robotics, bioinformatics, process analysis, and telecommunications. A good survey of SOM applications can be found, e.g., in [27.33]. SOMs have also been extended to temporal domains, mostly by the introduction of additional feedback connections, e.g., [27.34–37]. Such models can be used for topographic mapping or constrained clustering of time series data.

27.7 Recursive Neural Networks

In many application domains, data are naturally organized in structured form, where each data item is composed of several components related to each other in a non-trivial way, and the specific nature of the task to be performed is strictly related not only to the information stored at each component, but also to the structure connecting the components. Examples of structured data are parse trees obtained by parsing sentences in natural language, and the molecular graph describing a chemical compound.

Recursive neural networks (RecNN) [27.38, 39] are neural network models that are able to directly process structured data, such as trees and graphs. For the sake of presentation, here we focus on positional trees. Positional trees are trees for which each child has an associated index, its position, with respect to the siblings. Let us understand how RecNN is able to process a tree by analogy with what happens when unfolding a RNN when processing a sequence, which can be understood as a special case of tree where each node v possesses a single child.

In Fig. 27.13 (top) we show the unfolding in time of a sequence when considering a graphical model (*re-*

cursive network) representing, for a generic node v , the functional dependencies among the input information x_v , the state variable (hidden node) y_v , and the output variable o_v . The operator q^{-1} represents the shift operator in time (unit time delay), i. e., $q^{-1}y_t = y_{t-1}$, which applied to node v in our framework returns the child of node v . At the bottom of Fig. 27.13 we have reported the unfolding of a binary tree, where the recursive network uses a generalization of the shift operator, which given an index i and a variable associated to a vertex v returns the variable associated to the i -th child of v , i. e., $q_i^{-1}y_v = y_{\text{ch}_i[v]}$. So, while in RNN the network is *unfolded in time*, in RecNN the network is *unfolded on the structure*. The result of unfolding, in both cases, is the *encoding network*. The encoding network for the sequence specifies how the components implementing the different parts of the recurrent network (e.g., each node of the recurrent network could be instantiated by a layer of neurons or by a full feed-forward neural network with hidden units) need to be interconnected. In the case of the tree, the encoding network has the same semantics: this time, however, a set of parameters (weights) for each child should be considered, leading to a net-

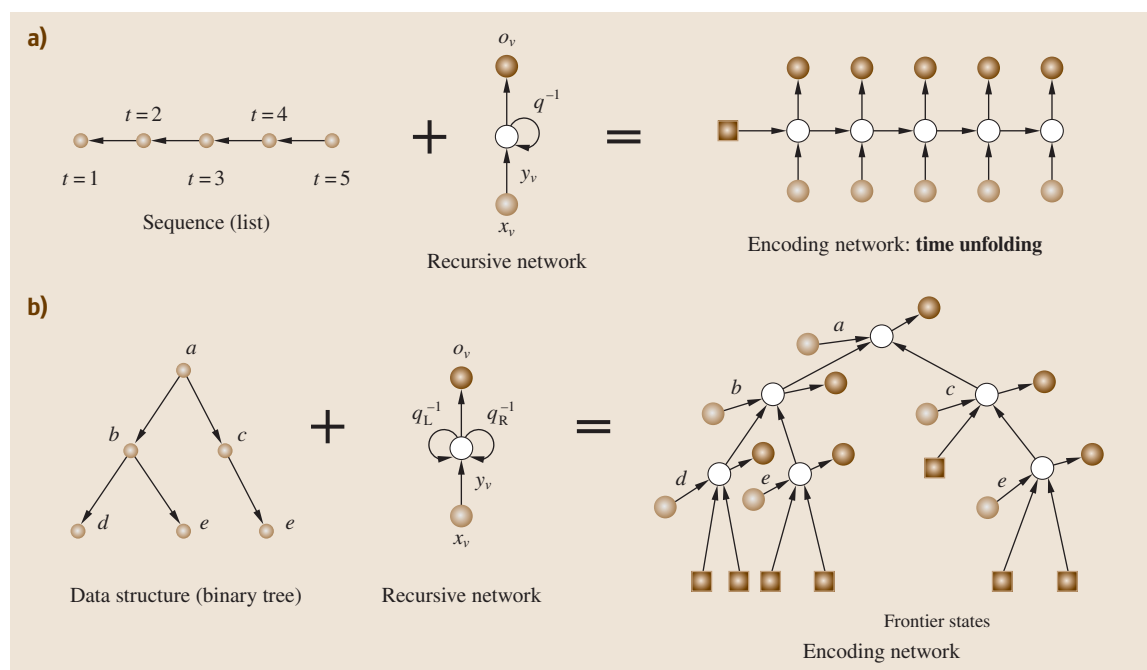


Fig. 27.13a,b Generation of the encoding network (a) for a sequence and (b) a tree. Initial states are represented by squared nodes

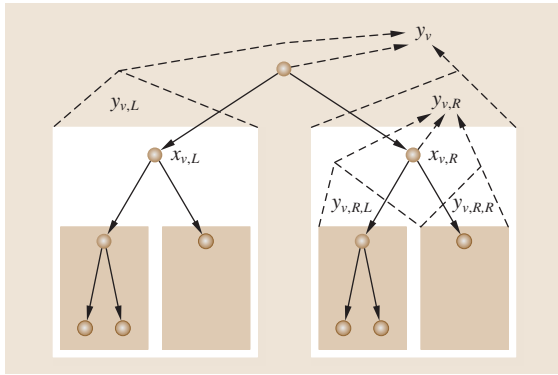


Fig. 27.14 The causality style of computation induced by the use of recursive networks is made explicit by using nested boxes to represent the recursive dependencies of the hidden variable associated to the root of the tree ◀

work that, given a node v , can be described by the equations

$$o_k^{(v)} = f \left(\sum_{j=1}^J m_{kj} y_j^{(v)} \right),$$

$$y_j^{(v)} = f \left(\sum_{s=1}^d \sum_{i=1}^J w_{ji}^s y_i^{(\text{ch}_s[v])} + \sum_{i=1}^I v_{ji} x_i^{(v)} \right),$$

where d is the maximum number of children an input node can have, and weights w_{ji}^s are indexed on the s -th child. Note that it is not difficult to generalize *all the learning algorithms* devised for RNN to these extended equations.

It should be remarked that recursive networks clearly introduce a causal style of computation, i.e., the computation of the hidden and output variables for a vertex v only depends on the information attached to v and the hidden variables of the children of v . This dependence is satisfied recursively by all v 's descendants and is clearly shown in Fig. 27.14. In the figure, nested boxes are used to make explicit the recursive dependencies among hidden variables that contribute to the determination of the hidden variable y_v associated to the root of the tree.

Although an encoding network can be generated for a directed acyclic graph (DAG), this style of computation limits the discriminative ability of RecNN to the class of trees. In fact, the hidden state is not able to encode information about the *parents* of nodes. The introduction of *contextual processing*, however, allows us to discriminate, with some specific exceptions, among DAGs [27.40]. Recently, *Micheli* [27.41] also showed how contextual processing can be used to extend RecNN to the treatment of *cyclic* graphs.

The same idea described above for supervised neural networks can be adapted to unsupervised models, where the output value of a neuron typically represents the similarity of the weight vector associated to the neuron with the input vector. Specifically, in [27.37] SOMs were extended to the processing of structured data (SOM-SD). Moreover, a general framework for self-organized processing of structured data

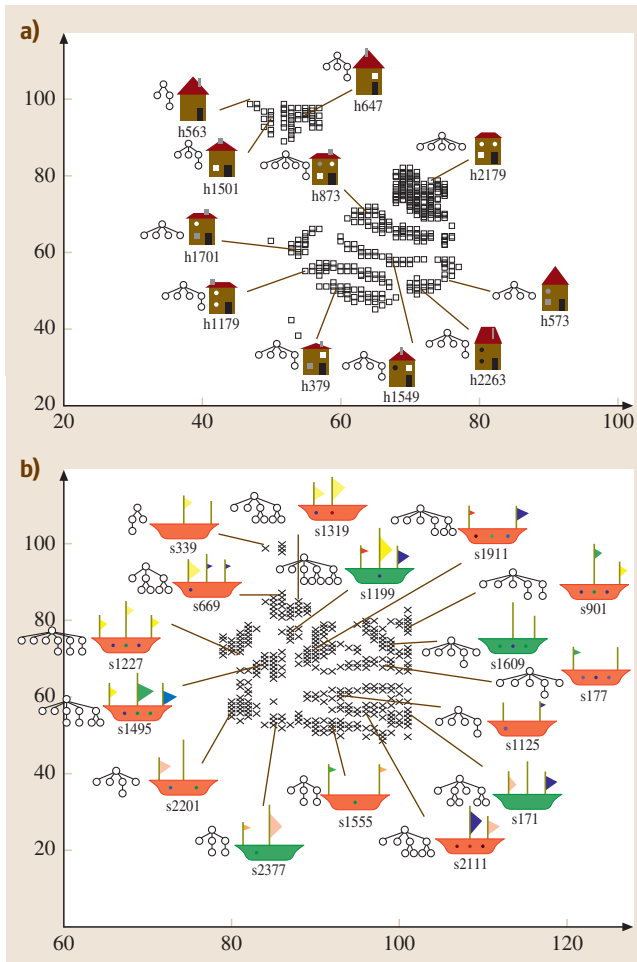


Fig. 27.15a,b Schematic illustration of a principal organization of biological self-organizing neural network (a) and its formal counterpart SOM ANN architecture (b)

was proposed in [27.42]. The key concepts introduced are:

- i) The explicit definition of a representation space R equipped with a similarity measure $d_R(\cdot, \cdot)$ to evaluate the similarity between two hidden states.
- ii) The introduction of a general representation function, denoted $rep(\cdot)$, which transforms the activation of the map for a given input into an hidden state representation.

In these models, each node v of the input structure is represented by a tuple $[\bar{x}_v, \bar{r}_{v_1}, \dots, \bar{r}_{v_d}]$, where \bar{x}_v is a real-valued vectorial encoding of the information attached to vertex v , and \bar{r}_{v_i} are real-valued

vectorial representations of hidden states returned by the $rep(\cdot)$ function when processing the activation of the map for the i -th neighbor of v . Each neuron n_j in the map is associated to a weight vector $[\bar{w}_j, \bar{c}_j^1, \dots, \bar{c}_j^d]$. The computation of the winner neuron is based on the joint contribution of the similarity measures $d_x(\cdot, \cdot)$ for the input information, and $d_R(\cdot, \cdot)$ for the hidden states, i.e., the internal representations. Some parts of a SOM-SD map trained on DAGs representing visual patterns are shown in Fig. 27.15. Even in this case the style of computation is causal, ruling out the treatment of undirected and/or cyclic graphs. In order to cope with general graphs, recently a new model, named GraphSOM [27.43], was proposed.

27.8 Conclusion

The field of artificial neural networks (ANN) has grown enormously in the past 60 years. There are many journals and international conferences specifically devoted to neural computation and neural network related models and learning machines. The field has gone a long way from its beginning in the form of simple threshold units existing in isolation (e.g., the perceptron, Sect. 27.2) or connected in circuits. Since then we have learnt how to generalize such networks as parameterized differentiable models of various sorts that can be fit to data (*trained*), usually by transforming the learning task into an optimization one.

ANN models have found numerous successful practical applications in many diverse areas of science and engineering, such as astronomy, biology, finance, geology, etc. In fact, even though basic feed-forward ANN architectures were introduced long time ago, they continue to surprise us with successful applications, most recently in the form of *deep networks* [27.44]. For example, a form of deep ANN recently achieved the best performance on a well-known benchmark problem – the recognition of handwritten digits [27.45]. This is quite remarkable, since such a simple ANN architecture trained in a purely data driven fashion was able to outperform the current state-of-art techniques, formulated in more sophisti-

cated frameworks and possibly incorporating domain knowledge.

ANN models have been formulated to operate in supervised (e.g., feed-forward ANN, Sect. 27.3; RBF networks, Sect. 27.5), unsupervised (e.g., SOM models, Sect. 27.6), semi-supervised, and reinforcement learning scenarios and have been generalized to process inputs that are much more general than simple vector data of fixed dimensionality (e.g., the recurrent and recursive networks discussed in Sects. 27.4 and 27.7). Of course, we were not able to cover all important developments in the field of ANNs. We can only hope that we have sufficiently motivated the interested reader with the variety of modeling possibilities based on the idea of interconnected networks of formal neurons, so that he/she will further consult some of the many (much more comprehensive) monographs on the topic, e.g., [27.3, 6, 7].

We believe that ANN models will continue to play an important role in modern computational intelligence. Especially the inclusion of ANN-like models in the field of probabilistic modeling can provide techniques that incorporate both explanatory model-based and data-driven approaches, while preserving a much fuller modeling capability through operating with full distributions, instead of simple point estimates.

References

- 27.1 F. Rosenblatt: The perceptron, a probabilistic model for information storage and organization in the brain, *Psychol. Rev.* **62**, 386–408 (1958)
- 27.2 D.E. Rumelhart, G.E. Hinton, R.J. Williams: Learning internal representations by error propagation. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1 Foundations*, ed. by D.E. Rumelhart, J.L. McClelland (MIT Press/Bradford Books, Cambridge 1986) pp. 318–363
- 27.3 J. Zurada: *Introduction to Artificial Neural Systems* (West Publ., St. Paul 1992)
- 27.4 K. Hornik, M. Stinchcombe, H. White: Multilayer feedforward networks are universal approximators, *Neural Netw.* **2**, 359–366 (1989)
- 27.5 D.J.C. MacKay: Bayesian interpolation, *Neural Comput.* **4**(3), 415–447 (1992)
- 27.6 S. Haykin: *Neural Networks and Learning Machines* (Prentice Hall, Upper Saddle River 2009)
- 27.7 C. Bishop: *Neural Networks for Pattern Recognition* (Oxford Univ. Press, Oxford 1995)
- 27.8 T. Sejnowski, C. Rosenberg: Parallel networks that learn to pronounce English text, *Complex Syst.* **1**, 145–168 (1987)
- 27.9 A. Weibel: Modular construction of time-delay neural networks for speech recognition, *Neural Comput.* **1**, 39–46 (1989)
- 27.10 J.L. Elman: Finding structure in time, *Cogn. Sci.* **14**, 179–211 (1990)
- 27.11 M.I. Jordan: Serial order: A parallel distributed processing approach. In: *Advances in Connectionist Theory*, ed. by J.L. Elman, D.E. Rumelhart (Erlbaum, Hillsdale 1989)
- 27.12 Y. Bengio, R. Cardin, R. DeMori: Speaker independent speech recognition with neural networks and speech knowledge. In: *Advances in Neural Information Processing Systems II*, ed. by D.S. Touretzky (Morgan Kaufmann, San Mateo 1990) pp. 218–225
- 27.13 P.J. Werbos: Generalization of backpropagation with application to a recurrent gas market model, *Neural Netw.* **1**(4), 339–356 (1988)
- 27.14 R.J. Williams, D. Zipser: A learning algorithm for continually running fully recurrent neural networks, *Neural Comput.* **1**(2), 270–280 (1989)
- 27.15 Y. Bengio, P. Simard, P. Frasconi: Learning long-term dependencies with gradient descent is difficult, *IEEE Trans. Neural Netw.* **5**(2), 157–166 (1994)
- 27.16 T. Lin, B.G. Horne, P. Tino, C.L. Giles: Learning long-term dependencies with NARX recurrent neural networks, *IEEE Trans. Neural Netw.* **7**(6), 1329–1338 (1996)
- 27.17 S. Hochreiter, J. Schmidhuber: Long short-term memory, *Neural Comput.* **9**(8), 1735–1780 (1997)
- 27.18 M. Lukosevicius, H. Jaeger: *Overview of Reservoir Recipes*, Technical Report, Vol. 11 (School of Engineering and Science, Jacobs University, Bremen 2007)
- 27.19 A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, J. Schmidhuber: A novel connectionist system for improved unconstrained handwriting recognition, *IEEE Trans. Pattern Anal. Mach. Intell.* **31**, 5 (2009)
- 27.20 S. Hochreiter, M. Heusel, K. Obermayer: Fast model-based protein homology detection without alignment, *Bioinformatics* **23**(14), 1728–1736 (2007)
- 27.21 H. Jaeger, H. Hass: Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless telecommunication, *Science* **304**, 78–80 (2004)
- 27.22 W. Maass, T. Natschlager, H. Markram: Real-time computing without stable states: A new framework for neural computation based on perturbations, *Neural Comput.* **14**(11), 2531–2560 (2002)
- 27.23 P. Tino, G. Dorffner: Predicting the future of discrete sequences from fractal representations of the past, *Mach. Learn.* **45**(2), 187–218 (2001)
- 27.24 M.H. Tong, A. Bicket, E. Christiansen, G. Cottrell: Learning grammatical structure with echo state network, *Neural Netw.* **20**, 424–432 (2007)
- 27.25 K. Ishii, T. van der Zant, V. Becanovic, P. Ploger: Identification of motion with echo state network, *Proc. OCEANS 2004 MTS/IEEE-TECHNO-OCEAN Conf.*, Vol. 3 (2004) pp. 1205–1210
- 27.26 L. Medsker, L.C. Jain: *Recurrent Neural Networks: Design and Applications* (CRC, Boca Raton 1999)
- 27.27 J. Kolen, S.C. Kremer: *A Field Guide to Dynamical Recurrent Networks* (IEEE, New York 2001)
- 27.28 D. Mandic, J. Chambers: *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability* (Wiley, New York 2001)
- 27.29 J.B. MacQueen: Some models for classification and analysis of multivariate observations, *Proc. 5th Berkeley Symp. Math. Stat. Probab.* (Univ. California Press, Oakland 1967) pp. 281–297
- 27.30 M.D. Buhmann: *Radial Basis Functions: Theory and Implementations* (Cambridge Univ. Press, Cambridge 2003)
- 27.31 G.-B. Huang, Q.-Y. Zhu, C.-K. Siew: Extreme learning machine: theory and applications, *Neurocomputing* **70**, 489–501 (2006)
- 27.32 T. Kohonen: *Self-Organizing Maps*, Springer Series in Information Sciences, Vol. 30 (Springer, Berlin, Heidelberg 2001)
- 27.33 T. Kohonen, E. Oja, O. Simula, A. Visa, J. Kangas: Engineering applications of the self-organizing map, *Proc. IEEE* **84**(10), 1358–1384 (1996)
- 27.34 T. Koskela, M. Varsta, J. Heikkonen, K. Kaski: Recurrent SOM with local linear models in time series prediction, *6th Eur. Symp. Artif. Neural Netw. (D-facto Publications, 1998)* pp. 167–172
- 27.35 T. Voegtlin: Recursive self-organizing maps, *Neural Netw.* **15**(8/9), 979–992 (2002)
- 27.36 M. Strickert, B. Hammer: Merge som for temporal data, *Neurocomputing* **64**, 39–72 (2005)

- 27.37 M. Hagenbuchner, A. Sperduti, A. Tsoi: Self-organizing map for adaptive processing of structured data, *IEEE Trans. Neural Netw.* **14**(3), 491–505 (2003)
- 27.38 A. Sperduti, A. Starita: Supervised neural networks for the classification of structures, *IEEE Trans. Neural Netw.* **8**(3), 714–735 (1997)
- 27.39 P. Frasconi, M. Gori, A. Sperduti: A general framework for adaptive processing of data structures, *IEEE Trans. Neural Netw.* **9**(5), 768–786 (1998)
- 27.40 B. Hammer, A. Micheli, A. Sperduti: Universal approximation capability of cascade correlation for structures, *Neural Comput.* **17**(5), 1109–1159 (2005)
- 27.41 A. Micheli: Neural network for graphs: A contextual constructive approach, *IEEE Trans. Neural Netw.* **20**(3), 498–511 (2009)
- 27.42 B. Hammer, A. Micheli, A. Sperduti, M. Strickert: A general framework for unsupervised processing of structured data, *Neurocomputing* **57**, 3–35 (2004)
- 27.43 M. Hagenbuchner, A. Sperduti, A.-C. Tsoi: Graph self-organizing maps for cyclic and unbounded graphs, *Neurocomputing* **72**(7–9), 1419–1430 (2009)
- 27.44 Y. Bengio, Y. LeCun: Greedy Layer-Wise Training of Deep Network. In: *Advances in Neural Information Processing Systems 19*, ed. by B. Schölkopf, J. Platt, T. Hofmann (MIT Press, Cambridge 2006) pp. 153–160
- 27.45 D.C. Ciresan, U. Meier, L.M. Gambardella, J. Schmidhuber: Deep big simple neural nets for handwritten digit recognition, *Neural Comput.* **22**(12), 3207–3220 (2010)



<http://www.springer.com/978-3-662-43504-5>

Springer Handbook of Computational Intelligence

Kacprzyk, J.; Pedrycz, W. (Eds.)

2015, LV, 1634 p. 534 illus., 65 illus. in color.,

Hardcover

ISBN: 978-3-662-43504-5