J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*
Copyright © 1999 John Wiley & Sons, Inc.

NEURAL NETS, RECURRENT

Recurrent neural nets (*RNNs*) are the most general class of neural nets (see Neural Net Architecture); they are nonlinear dynamic systems (see Nonlinear Systems) in which processing units, or *neurons*, are connected in such a way that the graph of connections contains cycles: that is, signals may flow in such a way that outputs from a processing unit may *feed back* as inputs to that processing unit, in general after having been processed by other units. Unlike in feedforward neural networks (see Feedforward Neural Nets), the presence of feedback makes time a relevant magnitude; as will be seen later, this explains the use of RNNs in temporal processing applications. In the most general class of RNNs, some units may receive inputs from outside the network; in addition, some of the units may be designated as outputs of the RNN. Units not designated as outputs are usually called *hidden* or *state* units.

There are three important ways in which RNNs may be classified:

- As to the nature of the output of processing units, RNNs may be classified as *continuous-state*, that is, when the outputs of processing units may take any value within an interval of real numbers, and *discrete-state*, when outputs take values on a finite set (usually binary).
- As to the treatment of time, RNN may be classified in two main classes: *discrete-time recurrent neural networks* (*DTRNNs*) and *continuous-time recurrent neural networks* (*CTRNNs*). DTRNNs use processing units whose outputs change instantaneously in response to any change in their inputs; therefore, it is convenient for outputs to be updated in discrete time steps, and a synchronizing device such as a clock is implied in the design of a DTRNN. CTRNNs use processing units whose output varies continuously in time in response to the instantaneous values of inputs; therefore, no clock is needed.
- As to the way in which they are used, RNNs may be classified in two broad groups: *temporal-processing RNNs* and *relaxation RNNs*. In temporal-processing RNNs we are interested in (a) the final output of the DTRNN after processing a time-varying input pattern, or (b) the time-varying output pattern produced by the RNN, starting in a particular initial state, by feeding either a time-varying input pattern, a constant input, or no input at all. Relaxation RNNs are started in a particular state and allowed to evolve in time until they reach a stationary state and outputs are read (although some networks may reach a periodic motion or *limit cycle* or show chaotic behavior instead of settling to a fixed point). The reader is advised that some authors (even some journals) use the denomination *recurrent neural networks* to refer only to temporal-processing RNNs.

The main emphasis of this article will be on discrete-time RNNs for sequence or temporal processing but we will also briefly cover relaxation DTRNNs and CTRNNs, the latter both for temporal processing and of the relaxation type.

DISCRETE-TIME RECURRENT NEURAL NETS FOR SEQUENCE PROCESSING

Sequence Processing. The word *sequence* (from Latin *sequentia*, i.e., "the ones following") is used to refer to a series of data, each one taken from a certain set of possible values U, so that each one of them is assigned an index (usually consecutive integers) indicating the order in which the data are generated or measured. Since the index usually refers to time, some researchers like to call sequences *time series*, as in "time series prediction" (1). In the field of signal processing, this would usually be called a discrete-time sampled signal; researchers in this field would identify the subject of this discussion as that of *discrete-time signal processing* (2).

In most of the following, we will consider, for convenience, that U is a vector space in the broadest possible sense. Examples of sequences are:

- Words on an alphabet (where U is the alphabet of possible letters and the integer labels 1, 2, ... are used to refer to the first, second, ... letter of the word)
- Acoustic vectors obtained every T milliseconds after suitable preprocessing of a speech signal (here U is a vector space, and the indices refer to sampling times)

What can be done with sequences? Without having the intention of being exhaustive and formal, one may classify sequence processing operations in the following broad classes (classification inspired by 3, p. 177):

- Sequence Classification, Sequence Recognition In this kind of processing, a whole sequence $u = u[1]u[2]\ldots u[L_u]$ is read, and a single value, label, or pattern (not a sequence) y, taken from a suitable set Y, is computed from it. For example, a sequence of acoustic vectors such as the one mentioned above may be assigned a label that describes the word that was pronounced, or a vector of probabilities for each possible word. Or a word on a given alphabet may be recognized as belonging to a certain language. For convenience, Y will also be considered to be some kind of vector space.
- Sequence Transduction or Translation, Signal Filtering In this kind of processing, a sequence $u = u[1]u[2]\ldots\ u[L_u]$ is transformed into another sequence $y = y[1]y[2]\ \ldots\ y[L_y]$ of data taken from a set Y. In principle, the lengths of the input L_u and the output L_y may be different. Processing may occur in different modes. Some sequence processors read the whole input sequence u and then generate the sequence y. Another mode is *sequential* processing, in which the output sequence is produced incrementally while processing the input sequence. Sequential processing has the interesting property that, if the result of processing of a given sequence u_1 is a sequence y_1, then the result of processing a sequence that starts with u_1 is always a sequence that starts with y_1 (this is sometimes called the *prefix property*). A special case of sequential processing is *synchronous* processing: the processor reads and writes one datum at a time, and therefore, both sequences grow at the same rate during processing. For example, Mealy and Moore machines, two classes of finite-state machines, are sequential, finite-memory, synchronous processors that read and write symbol strings. Examples of transductions and filtering include machine translation of sentences and filtering of a discrete-time sampled signal. Note that sequence classification applied to each prefix $u[1]$, $u[1]u[2]$, ... of a sequence $u[1]u[2]u[3]\ \ldots$ is equivalent to synchronous sequence transduction.
- Sequence Continuation or Prediction In this case, the sequence processor reads a sequence $u[1]u[2]\ \ldots\ u[t]$ and produces as an output a possible continuation of the sequence $\hat{u}[t+1]\hat{u}[t+2]\ldots$. This is usually called *time series prediction* and has interesting applications in meteorology and finance, where the ability to predict the future behavior of a system is a primary goal. Another interesting application of sequence prediction is predictive coding and compression. If the prediction is good enough, the difference between the predicted continuation of the signal and its actual continuation may be transmitted using a channel with a lower bandwidth or a lower bit rate. This is extensively used in *speech coding* (4)—for example, in

digital cellular phone systems. Sequence continuation is not very different from sequence transduction: the key difference is that in the former, one cannot assume the presence of causality.

- Sequence Generation In this mode, the process generates an output string $y[1]y[2]\ldots$ from a single input u or no input at all. For example, a text-to-speech system may generate the audio signal for each syllable in its dictionary.

State-Based Sequence Processors. Sequence processors may be built around a *state*; state-based sequence processors maintain and update at each time t a state $x[t]$, which stores the information about the input sequence they have seen so far ($u[1], \ldots, u[t]$), which is necessary to compute the current output $y[t]$ or future outputs. State is computed *recursively*: the state at time t, $x[t]$, is computed from the state at time $t-1$, $x[t-1]$, and the current input $u[t]$ using a suitable *next-state* function:

$$x[t] = f(x[t-1], u[t]) \tag{1}$$

The output is then computed using an *output* function, usually from the current state [as in Moore machines in automata theory (5); see also the next subsection]:

$$y[t] = h(x[t]) \tag{2}$$

but sometimes from the previous state and the current input [as in Mealy machines in automata theory (5); see also the next subsection]:

$$y[t] = h(x[t-1], u[t]) \tag{3}$$

Such a state-based sequence processor is therefore defined by the set of available states, its initial state $x[0]$, and the next-state (f) and output (h) functions (the nature of the inputs and outputs is defined by the task itself). The state of a state-based sequence processor may in general be *hidden*; that is, the current state may not in general be inferrable by studying a finite-length window of past inputs, a finite-length window of past outputs, or both, but sometimes it is. In any of the last three cases, state is said to be *observable* (6).

DISCRETE-TIME RECURRENT NEURAL NETS AS NEURAL STATE MACHINES

Neural nets may be used and trained as state-based adaptive sequence processors. The most general architecture is a DTRNN, that is, a neural net in which the output of some units is fed back as an input to some others. In DTRNNs, processing occurs in discrete steps, as if the net were driven by an external clock, and each of the neurons is assumed to compute its output instantaneously; hence the name.

DTRNNs may therefore be applied to any of the four broad classes of sequence-processing tasks mentioned before: in sequence classification, the output of the DTRNN is examined only at the end of the sequence; in synchronous sequence transduction tasks, the DTRNN produces a temporal sequence of outputs corresponding to the sequence of inputs it is processing; in sequence continuation or prediction tasks, the output of the DTRNN after having seen an input sequence may be interpreted as a continuation of it; finally, in sequence generation tasks, a constant or no input may be applied in each cycle to generate a sequence of outputs.

In a DTRNN with n_X hidden units and n_Y output units receiving n_U input signals, we will denote by $x_i[t]$ (respectively $y_j[t]$) the state of hidden (respectively output) unit $i = 1, \ldots n_X$ (respectively $j = 1, \ldots n_Y$) at time t. The kth external input signal at time t will be called $u_k[t]$. Inputs, hidden states, and outputs may be expressed as vectors $\boldsymbol{u}[t]$, $\boldsymbol{x}[t]$, and $\boldsymbol{y}[t]$ respectively. The discrete-time evolution of the hidden state of the network may be

expressed, in general terms, as in Eqs. (1) to (3), with functions $\boldsymbol{f}$ and $\boldsymbol{h}$ realized as single-layer or multilayer feedforward neural networks (see Feedforward Neural Nets; unlike in Eqs. (1) to (3), bold lettering is used here to emphasize the vectorial nature of states, inputs, outputs, and next-state and output functions).

It is therefore natural to see DTRNNs, (7, Chap. 15; 3 Chap. 7; 8) as *neural state machines* (*NSMs*), and to define them in a way that is parallel to the definitions of Mealy and Moore machines used in formal language theory (5). This parallelism is inspired by the relationship established by Pollack (9) between deterministic finite automata (*DFAs*) and a class of second-order DTRNNs, under the name of *dynamical recognizers*.

A *neural state machine* N is a sextuple

$$N = (X,\ U,\ Y,\ \boldsymbol{f}, \boldsymbol{h}, \boldsymbol{x}_0) \tag{4}$$

in which

- $[S_0,S_1]_X^n$ is the state space of the NSM, with S_0 and S_1 the endpoints of defining the range of values for the state of each unit, and n_X the number of state units.
- $U = R_n^U$ is the set of possible input vectors, with n_U the number of input lines.
- $[S_0,S_1]_Y^n$ is the set of outputs of the NSM, with n_Y the number of output units.
- $\boldsymbol{f} : X \times U \to X$ is the *next-state function*, a feedforward neural network that computes a new state $\boldsymbol{x}[t]$ from the previous state $\boldsymbol{x}[t-1]$ and the input just read, $\boldsymbol{u}[t]$:

$$\mathrm{x}[t] = \boldsymbol{f}(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) \tag{5}$$

- $\boldsymbol{h}$ is the *output function*, which in the case of a Mealy NSM is $\boldsymbol{h} : X \times U \to Y$, that is, a feedforward neural network that computes a new output $\boldsymbol{y}[t]$ from the previous state $\boldsymbol{x}[t-1]$ and the input just read, $\boldsymbol{u}[t]$:

$$\boldsymbol{y}[t] = \boldsymbol{h}(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) \tag{6}$$

  and in the case of a Moore NSM is $\boldsymbol{h} : X \to Y$, a feedforward neural network that computes a new output $\boldsymbol{y}[t]$ from the newly reached state $\boldsymbol{x}[t]$:

$$\boldsymbol{y}[t] = \boldsymbol{h}(\boldsymbol{x}[t]) \tag{7}$$

- $\boldsymbol{x}_0$ is the initial state of the NSM, that is, the value that will be used for $\boldsymbol{x}[0]$.

Most classical DTRNN architectures may be directly defined using the NSM scheme; the following sections show some examples (in all of them, weights and biases are assumed to be real numbers). The generic block diagrams of neural Mealy and neural Moore machines are given in Fig. 1 and 2 respectively.

Neural Mealy Machines. Omlin and Giles (10) have used a second-order recurrent neural network [similar to the one used by other authors (11, 9)], which may be formulated as a Mealy NSM described by a next-state function whose ith coordinate ($i = 1, \ldots, n_X$) is

$$f_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = g\left(\sum_{j=1}^{n_X}\sum_{k=1}^{n_U} W_{ijk}^{xxu} x_j[t-1]u_k[t] + W_i^x\right) \tag{8}$$

where g: $R \to [S_0, S_1]$ (usually $S_0 = 0$ or -1 and $S_1 = 1$) is the activation function [also called *transfer function*, *gain function*, and *squashing function* (3, p. 4)] of the neurons, and an output function whose ith coordinate

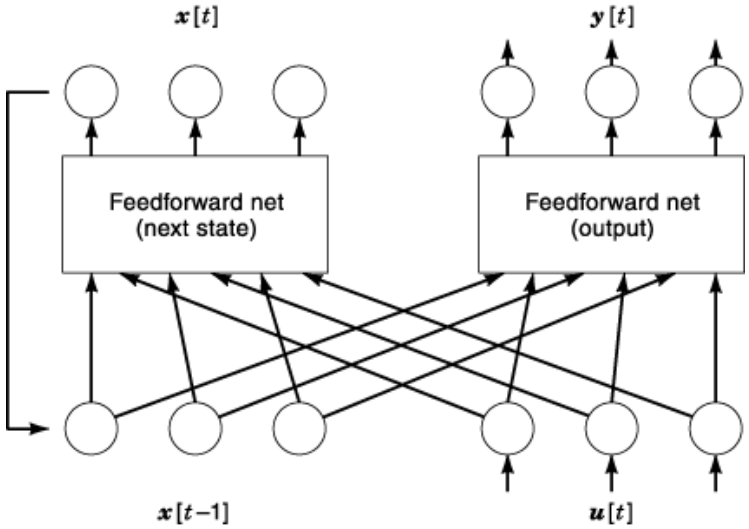

Fig. 1. Block diagram of a neural Mealy machine.

$(i = 1, \ldots, n_Y)$ is

$$h_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = g\left(\sum_{j=1}^{n_X}\sum_{k=1}^{n_V} W_{ijk}^{yxxu} x_j[t-1]u_k[t] + W_i^y\right) \tag{9}$$

Throughout this article, a homogeneous notation will be used for weights. Superscripts indicate the computation in which the weight is involved: the *xxu* in $W_{ijk}{}^{xxu}$ indicates that the weight is used to compute a state (x) from a state and an input (xu); the y in $W^y{}_i$ (a bias) indicates that it is used to compute an output. Subscripts designate, as usual, the particular units involved and run parallel to superscripts.

Activation functions $g(x)$ are usually required to be real-valued, monotonically growing, continuous (very often also differentiable), and bounded; they are usually nonlinear. Two commonly used examples of differentiable activation functions are the logistic function $g_{\mathrm{L}}(x) = 1/(1 + e^{-x})$, which is bounded by 0 and 1, and the hyperbolic tangent $g_{\mathrm{T}}(x) = \tanh x = (1 - e^{-2x})/(1 + e^{-2x})$, which is bounded by -1 and 1. Activation functions are usually required to be differentiable because this allows the use of training algorithms based on gradients. There are also a number of architectures that do not use sigmoidlike activation functions but instead use *radial basis functions* (7, Chap. 5; 3, p. 248), which are not monotonic but instead are Gaussianlike functions that reach their maximum value for a given value of their input. DTRNN architectures using radial basis functions have been used by various authors (see, e.g., 12, 13).

Another Mealy NSM is Robinson and Fallside's *recurrent error propagation network* (14), a first-order DTRNN that has a next-state function whose ith coordinate $(i = 1, \ldots, n_X)$ is given by

$$f_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = g\left(\sum_{j=1}^{n_X} W_{ij}^{xx} x_j[t-1] + \sum_{j=1}^{n_V} W_{ij}^{xu} u_j[t] + W_i^x\right) \tag{10}$$

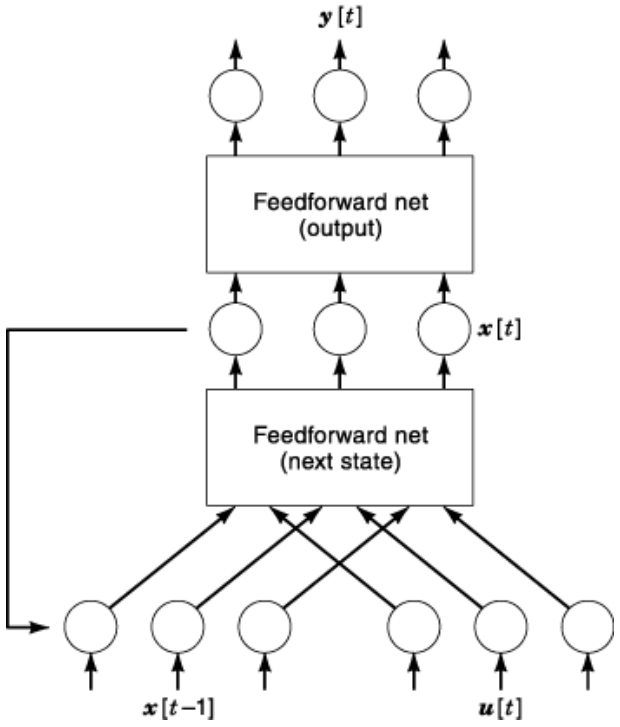

Fig. 2. Block diagram of a neural Moore machine.

and an output function $\boldsymbol{h}(\mathbf{x}[t-1], \boldsymbol{u}[t])$ whose ith component ($i = 1, \ldots, n_Y$) is given by

$$h_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = g\left(\sum_{j=1}^{n_X} W_{ij}^{yx} x_j[t-1] + \sum_{j=1}^{n_V} W_{ij}^{yu} u_j[t] + W_i^y\right) \qquad (11)$$

Jordan nets (15) may also be formulated as Mealy NSMs. Both the next-state and the output function use an auxiliary function $\boldsymbol{z}(\boldsymbol{x}[t-1], \boldsymbol{u}[t])$ whose ith coordinate is

$$z_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = g\left(\sum_{j=1}^{n_X} W_{ij}^{zx} x_j[t-1] + \sum_{j=1}^{n_V} W_{ij}^{zu} u_j[t] + W_i^z\right) \qquad (12)$$

with $i = 1, \ldots, n_Z$. The ith coordinate of the next-state function is

$$f_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = \alpha x_i[t-1] + g\left(\sum_{j=1}^{n_Z} W_{ij}^{xz} z_j(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) + W_i^x\right) \qquad (13)$$

(with $\alpha \in [0,1]$ a constant), and the ith coordinate of the output function is

$$h_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = g\left(\sum_{j=1}^{n_Z} W_{ij}^{xz} z_j(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) + W_i^x\right) \tag{14}$$

are computed.

Neural Moore Machines. Elman's *simple recurrent net* (16), a widely used Moore NSM, is described by a next-state function identical to the next-state function of Robinson and Fallside's net, Eq. (10), and an output function $\boldsymbol{h}(\boldsymbol{x}[t])$ whose ith component ($i = 1, \ldots, n_Y$) is given by

$$h_i(\boldsymbol{x}[t]) = g\left(\sum_{j=1}^{n_X} W_{ij}^{yx} x_j[t] + W_i^y\right) \tag{15}$$

However, an even simpler DTRNN is the one used by Williams and Zipser (12), which has the same next-state function but an output function that is simply a projection of the state vector $y_i[t] = x_i[t]$ for $i = 1, \ldots, n_Y$ with $n_Y \leq n_X$. This architecture is also used in the encoder part of Pollack's RAAM (18) when encoding sequences.

The second-order counterpart of Elman's (16) simple recurrent net has been used by Blair and Pollack (19) and Carrasco et al. (20). In that case, the ith coordinate of the next-state function is identical to Eq. (8), and the output function is identical to Eq. (15).

Second-order DTRNNs such as the one used by Giles et al. (11) and Pollack (9) may be formulated as a Moore NSM in which the output vector is simply a projection of the state vector $f_i(\boldsymbol{x}[t]) = x_i[t]$ for $i = 1, \ldots, n_Y$ with $n_Y \leq n_X$, and may then be viewed as the second-order counterpart of 17. The classification of these second-order nets as Mealy or Moore NSMs depends on the actual configuration of feedback weights used by the authors. For example, Giles et al. (11) use one of the units of the state vector $\boldsymbol{x}[t]$ as an output unit; this makes their net a neural Moore machine in which $y[t] = x_1[t]$ (this unit is part of the state vector, because its value is also fed back to form $\mathbf{x}[t-1]$ for the next cycle).

Architectures without Hidden State. There are a number of discrete-time neural net architectures that do not have a hidden state (their state is observable because it is simply a combination of past inputs and past outputs) but may still be classified as recurrent. One such example is the NARX or Narendra–Parthasarathy net (21), which may be formulated in state-space form by defining a state that is simply a window of the last n_I inputs and a window of the last n_O outputs. Accordingly, the next-state function simply incorporates a new input (discarding the oldest one) and a freshly computed output (discarding the oldest one) in the windows and shifts each one of them one position. The $n_X = n_I n_U + n_O n_Y$ components of the state vector are distributed as follows:

- The first $n_I n_U$ components are allocated to the window of the last n_I inputs: $u_i[t-k]$ ($k = 0, \ldots, n_I - 1$) is stored in $X_{i+kn_U}[t]$.
- The $n_O n_Y$ components from $n_I n_U + 1$ to n_X are allocated to the window of the last n_O outputs: $y_i[t-k]$ ($k = 1, \ldots, n_O$) is stored in $x_{n_I n_U + i + (k-1)n_Y}[t]$.

The next-state function $\boldsymbol{f}$ performs, therefore, the following operations:

- Incorporating the new input $\boldsymbol{u}[t]$ and shifting past inputs:

$$f_i(\mathrm{x}[t-1], \boldsymbol{u}[t]) = \begin{cases} u_i[t], 1 \le i \le n_U, \\ x_{i-n_U}[t-1], n_U < i \le n_U n_I \end{cases} \tag{16}$$

- Shifting past outputs:

$$f_i(\mathrm{x}[t-1], \boldsymbol{u}[t]) = x_{i-n_Y}[t-1], \qquad n_U n_I + n_Y < i \le n_X \tag{17}$$

- Computing new state components using an intermediate hidden layer of n_Z units:

$$f_{i+n_U n_I}(\mathrm{x}[t-1], \boldsymbol{u}[t]) = \left(\sum_{j=1}^{n_Z} W_{ij}^{xz} z_j[t] + W_i^x \right), \qquad 1 \le i \le n_Y \tag{18}$$

with

$$z_i[t] = g\left(\sum_{j=1}^{n_x} W_{ij}^{zx} x_i[t-1] + \sum_{j=1}^{n_U} W_{ij}^{zu} u_j[t] + W_i^z \right), \qquad 1 \le i \le n_Z \tag{19}$$

The output function is then simply

$$h_i(\boldsymbol{x}[t]) = x_{i+n_U n_I}[t] \tag{20}$$

with $1 \le i \le n_Y$. Note that the output is computed by a two-layer feedforward neural net. The operation of a NARX net N may then be summarized as follows (see Fig. 3):

$$\boldsymbol{y}[t] = N(\boldsymbol{u}[t], \boldsymbol{u}[t-1], \ldots, \boldsymbol{u}[t-n_I], \boldsymbol{y}[t-1], \boldsymbol{y}[t-2], \ldots, \boldsymbol{y}[t-n_O]) \tag{21}$$

Its operation is therefore a nonlinear variation of that of an autoregressive moving-average (*ARMA*) model or that of an infinite-time impulse response *IIR* filter.

When the state of the discrete-time neural net is simply a window of the last inputs, we have a net usually called a *time-delay neural net* (*TDNN*), but also NetTalk, after a successful application (22) to text-to-speech conversion. In state-space formulation, the state is simply the window of the last n_I inputs, and the next-state function simply incorporates a new input in the window and shifts it one position in time:

$$f_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = \begin{cases} x_{i-n_U}[t-1], \ n_U < i \le n_U n_I \\ u_i[t], \ 1 \le i \le n_U \end{cases} \tag{22}$$

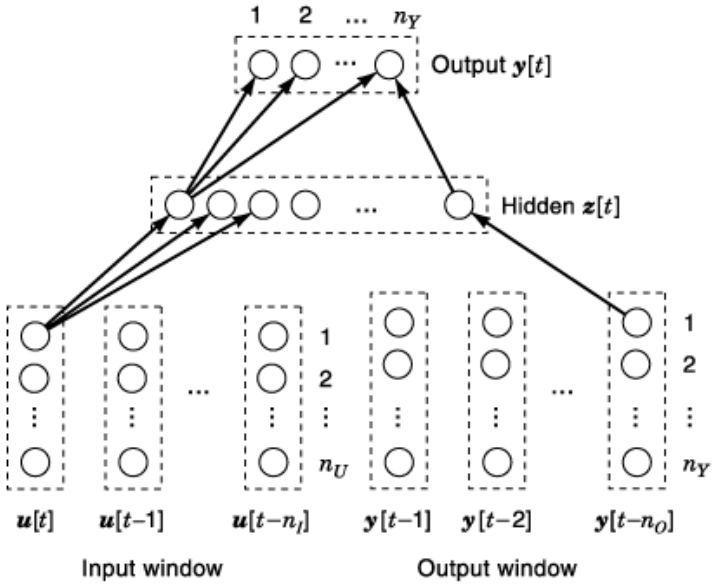

Fig. 3. Block diagram of a NARX network (the network is fully connected, but for clarity not all arrows have been drawn).

with $n_X = n_U n_I$; and the output is usually computed by a two-layer perceptron:

$$h_i(\boldsymbol{x}[t-1], \boldsymbol{u}[t]) = g\left(\sum_{j=1}^{n_Z} W_{ij}^{yz} z_j[t] + W_i^y\right), \qquad 1 \le i \le n_Y \tag{23}$$

with

$$z_i[t] = g\left(\sum_{j=1}^{n_x} W_{ij}^{zx} x_j[t-1] + \sum_{j=1}^{n_u} W_{ij}^{zu} u_j[t] + W_i^z\right), \qquad 1 \le i \le n_Z \tag{24}$$

The operation of a TDNN N may then be summarized as follows (see Fig. 4):

$$\boldsymbol{y}[t] = N(\boldsymbol{u}[t], \boldsymbol{u}[t-1], \ldots, \boldsymbol{u}[t-n_I]) \tag{25}$$

Their operation is therefore a nonlinear variant of that of a moving-average MA model or that of a finite-time impulse response *FIR* filter.

The weights connecting the window of inputs to the hidden layer may be organized in blocks sharing weight values, so that the components of the hidden layer retain some of the temporal ordering in the input window. TDNNs have been used for tasks such as phonetic transcription (22), protein secondary structure prediction (23), and phoneme recognition (24, 25).

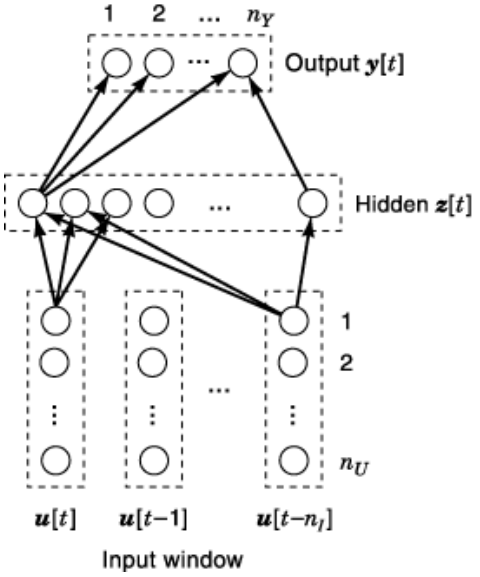

Fig. 4. Block diagram of a TDNN (the network is fully connected, but for clarity not all arrows have been drawn).

APPLICATION OF DTRNNs TO SEQUENCE PROCESSING

DTRNNs have been applied to a wide variety of sequence-processing tasks; here is a survey of some of them:

Channel Equalization. In digital communications, when a series of symbols is transmitted, the effect of the channel (see Multipath Channels) may yield a signal whose decoding may be impossible without resorting to a compensation or reversal of these effects at the receiver side. This sequence transduction task (which converts the garbled sequence received into something as similar as possible to the transmitted signal) is usually known as *equalization*. A number of researchers have studied DTRNNs for channel equalization purposes (26, 12, 27).

Speech Recognition. Speech recognition (see Speech Recognition and also Speech Processing) may be formulated either as a sequence transduction task (for example, continuous speech recognition systems aim at obtaining a sequence of phonemes from a sequence of acoustic vectors derived from a digitized speech sample) or as a sequence recognition task (for example, as in isolated-word recognition, which assigns a word in a vocabulary to a sequence of acoustic vectors). DTRNNs have been extensively used in speech recognition tasks (14, 28, 29).

Speech Coding. Speech coding (see Speech Coding) aims at obtaining a compressed representation of a speech signal so that it may be sent at the lowest possible bit rate. A family of speech coders are based on the concept of *predictive coding*: if the speech signal at time t may be predicted using the values of the signal at earlier times, then the transmitter may simply send the prediction error instead of the actual value of the signal, and the receiver may use a similar predictor to reconstruct the signal; in particular, a DTRNN may be used as a predictor. The transmission of the prediction error may be arranged in such a way that the number of bits necessary is much smaller than that needed to send the actual signal with the same reception quality (4). For instance, in 30 DTRNN predictors are used for speech coding.

System Identification and Control. DTRNNs may be trained to be models of time-dependent processes such as a stirred-tank continuous chemical reactor: this is usually referred to as *system identification*. Control goes a step further: a DTRNN may be trained to drive a real system (a "plant") so that the properties of its output follow a desired temporal pattern. DTRNNs have been extensively used both in system identification (see, e.g., 31, 32) and control (see, e.g., 21, 33, 34, 35).

Time Series Prediction. The prediction of the next item in a sequence may be of interest in many applications besides speech coding. For example, short-term electrical load forecasting is important for controlling electrical power generation and distribution. Time series prediction is a classical sequence prediction application of the DTRNN. See, for example, 36, 37).

Natural Language Processing. The processing of sentences written in any natural (human) language (see Natural Language Understanding) may itself be seen as a sequence-processing task, and has been also approached with DTRNNs. Examples include discovering grammatical and semantic classes of words when predicting the next word in a sentence (16) and training a DTRNN to judge the grammaticality of natural language sentences (38).

Grammatical Inference. In recent years, there has been a lot of interest in the use of DTRNNs to learn formal grammars and language recognizers, with an emphasis on the induction of simple finite-state language recognizers (39, 9, 11, 40, 41) or finite-state transducers (42) from input–output strings. Parallel work has studied the computational power of DTRNNs in connection with finite-state computation (43, 44, 45, 10, 46) or Turing machines (47).

LEARNING IN DTRNNs

Learning Algorithms for DTRNNs. When we want to train a DTRNN as a sequence processor, the usual procedure is to choose the architecture and parameters of the architecture: the number of input neurons (n_U) and the number of output neurons (n_Y) will usually be determined by the nature of the input sequence itself and by the nature of the processing we want to perform; the number of state neurons (n_X) will have to be determined through experimentation or used as a computational bias restricting the computational power of the DTRNN when we have *a priori* knowledge about the computational requirements of the task. It is also possible to modify the architecture as training proceeds (see e.g. 48), as will be mentioned later. Then we train the DTRNN on examples of processed sequences; training a DTRNN as a discrete-time sequence processor involves adjusting its learnable parameters. In a DTRNN these are the weights, biases, and initial states ($\boldsymbol{x}_0$) (learning the initial state is not very common in the DTRNN literature (49)—surprisingly, because it seems rather straightforward to do so). To train the network we usually need an *error* measure, which describes how far the actual outputs are from their desired targets; the learnable parameters are modified to minimize the error measure. It is very convenient if the error is a differentiable function of the learnable parameters (this is usually the case with sigmoidlike activation functions, as we have discussed in the subsection "Neural Mealy Machines").

A number of different problems may occur when training a DTRNN—and, in general, any neural network—by error minimization. These problems are reviewed in the next subsection.

Learning algorithms (also called *training algorithms*) for DTRNNs may be classified according to diverse criteria. All learning algorithms [except trivial algorithms such as a random search (50)] implement a heuristic to search the many-dimensional space of learnable parameters for minima of the error function chosen; the nature of this heuristic may be used to classify them. Some of the divisions that will be described in the following may also apply to nonrecurrent neural nets.

A major division occurs between *gradient-based* algorithms, which compute the gradient of the error function with respect to the learnable parameters at the current search point and use this vector to define

the next point in the search sequence, and *non-gradient-based* algorithms, which use other (usually local) information to decide the next point. Obviously, gradient-based algorithms require that the error function be differentiable, whereas most non-gradient-based algorithms may dispense with this requirement. In the following, this will be used as the main division.

Another division relates to the schedule used to decide the next set of learnable parameters. *Batch* algorithms compute the total error function for all of the patterns in the current learning set and update the learnable parameters only after a complete evaluation of the total error function has been performed. *Pattern* algorithms compute the contribution of a single pattern to the error function and update the learnable parameters after computing this contribution. This formulation of the division may be applied to most neural net learning algorithms; however, in the case of DTRNNs used as sequence processors, targets may be available not only for a whole sequence (as, for instance, in a classification task) but also for parts of a sequence (as in a synchronous translation task in which the targets are known after each item of the sequence). In the second case, a third learning mode, *online learning*, is possible: the contribution of each partial target to the error function may be used to update some of the learnable parameters even before the complete sequence has been processed. Online learning is the only possible choice when the learning set consists of a single sequence without a defined endpoint or when patterns can only be presented once (for a detailed discussion of gradient-based learning algorithms for DTRNNs and their modes of application, the reader is referred to 51, which gives an excellent survey with an emphasis on continuously running DTRNNs).

A third division has already been mentioned. Most learning algorithms for DTRNNs do not change the architecture during the learning process. However, there are some algorithms that modify the architecture of the DTRNN while training it [for example, Fahlman's *recurrent cascade correlation* (48) adds neurons to the network during training].

Gradient-Based Algorithms. The two most common gradient-based algorithms for DTRNNs are *back-propagation through time* (*BPTT*) and *real-time recurrent learning* (*RTRL*). Most other gradient-based algorithms may be classified as using an intermediate or hybrid strategy combining the desirable features of these two canonical algorithms.

The simplest kind of gradient-based algorithm—used also for feedforward neural net—is a gradient-descent learning algorithm, which updates each learnable parameter p of the network according to the rule

$$p_{\text{new}} = p_{\text{old}} - \alpha_p \frac{\partial E}{\partial p} \tag{26}$$

where α_p is a positive magnitude (not necessarily a constant) called the *learning rate* for the parameter p, and E is either the total error for the whole learning set (as in batch learning) or the error for the pattern just presented (as in pattern learning). Most gradient-based algorithms are improvements on this simple scheme (for details see e.g. 7), pp. 220, 233ff.; 3, pp. 103ff. 123ff. 157); all of them require the calculation of derivatives of error with respect to all of the learnable parameters. The derivatives for a DTRNN may be computed (or approximated) in different ways, which lead to a variety of methods.

Backpropagation through Time. BPTT may be considered as the earliest training algorithm for DTRNNs. The most commonly used reference for BPTT is 52, although earlier descriptions of BPTT may be found (see, e.g., 53). The central idea of BPTT is the *unfolding* of the discrete-time recurrent neural network into a multilayer feedforward neural network (FFNN) each time a sequence is processed. The FFNN has a layer for each "time step" in the sequence; each layer has n_X units, that is, as many as there are state units in the original networks. It is as if we were using time to index layers in the FFNN. Next state is implemented by connecting state units in layer $t-1$ and inputs in time t to state units in layer t. Output units (which are also repeated in each "time step" where targets are available) are connected to state units (and input units when the DTRNN is a Mealy NSM) as in the DTRNN itself.

The resulting FFNN is trained using the standard backpropagation (*BP*) algorithm, but with one restriction: since layers have been obtained by replicating the DTRNN over and over, weights in all layers should be the same. To achieve this, BPTT updates all equivalent weights using the sum of the gradients obtained for weights in equivalent layers, which may be shown to be the exact gradient of the error function for the DTRNN.

In BPTT, weights can only be updated after a complete forward step and a complete backward step, just as in regular BP. When processing finite sequences, weights are usually updated after a complete presentation of the sequence.

The time complexity of BPTT is one of its most attractive features: for a first-order DTRNN in which the number of states is larger than the number of inputs ($n_X > n_U$), the temporal cost of the backward step used to compute the derivatives grows as ${n_X}^2$, that is, the same as the cost of the forward step used to process the sequence and compute the outputs. The main drawback of BPTT is its space complexity, proportional to the length of the sequence, which comes from the need to replicate the DTRNN for each step of the sequence. This also makes it a bit trickier to program than RTRL.

For more details on BPTT the reader is referred to 7 (p. 751) and 3 (p. 182).

Real-Time Recurrent Learning. RTRL has been independently derived by many authors; the most commonly cited reference for it is 17 [for more details see also 3 (p. 184) and 7 (p. 756)]. This algorithm computes the derivatives of outputs and states with respect to all weights as the network processes the string, that is, during the forward step. No unfolding is performed or necessary. For instance, if the network has a simple next-state dynamics such as the one described in Eq. (10), derivatives may be computed together with the next state. The derivative of states with respect to, say, state–state weights at time t, will be computed from the states and derivatives at time $t-1$ and the input at time t as follows:

$$\frac{\partial x_i[t]}{\partial W_{kl}^{xx}} = g'(\Xi_i[t]) \left(\delta_{ik} x_l[t] + \sum_{j=1}^{n_X} W_{ij}^{xx} \frac{\partial x_j[t-1]}{\partial W_{kl}^{xx}} \right) \tag{27}$$

with $g'(\,)$ the derivative of the activation function, δ_{ik} Kronecker's delta (1 if $i = k$ and 0 otherwise) and

$$\Xi_i[t] = \sum_{j=1}^{n_X} W_{ij}^{xx} x_j[t-1] + \sum_{j=1}^{n_U} W_{ij}^{xu} u_j[t] + W_i^x \tag{28}$$

the net input to state unit i. The derivatives of states with respect to weights at $t = 0$ are initialized to zero. Derivatives with respect to the components of the initial state $\boldsymbol{x}[0]$ may also be easily computed (49, 54, 19) by initializing them accordingly (that is, $\partial x_i[0]/\partial x_j[0] = \delta_{ij}$).

Since derivatives of outputs are easily defined in terms of state derivatives for all architectures, the parameters of the DTRNN may be updated after every time step in which output targets are defined, even after having processed only part of a sequence. This is one of the main advantages of RTRL in applications where online learning is necessary; the other one is the ease with which it may be derived and programmed for a new architecture. However, its time complexity is much higher than that of BPTT; for first-order DTRNNs such as the above, with more state units than input lines ($n_X > n_U$), the dominant term in the time complexity is ${n_X}^4$. A detailed derivation of RTRL for a second-order DTRNN architecture may be found in 11.

The reader should be aware that the name RTRL is applied to two different concepts: RTRL (17) may be viewed solely as a method to compute the derivatives or as a method to compute derivatives *and* update weights (in each cycle). One may use RTRL to compute derivatives and update the weights after processing a complete sample made up of a number of sequences (batch update), after processing each sequence (pattern update), and after processing each item in each sequence. In these last two cases, the derivatives approach the true gradient as the learning rate approaches zero. For batch and pattern weight updates, RTRL and BPTT

are equivalent, since they compute the same derivatives. Hybrid or compromise algorithms combining the best features of RTRL and BPTT have also been proposed (see, e.g., 55).

Other Derivative-Based Methods. It is also possible to train a DTRNN using the *extended Kalman filter* (*EKF*), a nonlinear extension of Kalman filters (see Kalman Filters), of which RTRL may be shown to be a special case (56); the EKF has been successfully used in many applications, such as neurocontrol (33). The EKF is also related to recursive least squares (*RLS*) algorithms.

Non-gradient Methods. Gradient-based algorithms are the most used of all learning algorithms for DTRNNs. But there are also some interesting non-gradient-based algorithms. Of those, two batch learning algorithms are worth mentioning:

- Alopex (57) biases random weight updates according to the observed correlation between previous updates of each learnable parameter and the change in the total error for the learning sample. It does not need any knowledge about the net's particular structure; that is, it treats the net as a black box, and, indeed, it may be used to optimize parameters of systems other than neural nets; this makes it specially attractive when it comes to test a new architecture for which derivatives have not been derived yet.
- Cauwenberghs's (58) algorithm uses a related learning rule: the change effected by a random perturbation π of the weight vector $\boldsymbol{W}$ on the total error $E(\boldsymbol{W})$ is computed, and weights are updated in the direction of the perturbation so that the new weight vector is $\boldsymbol{W} - \mu[E(\boldsymbol{W} + \pi) - E(\mathbf{W})]\ \pi$, where μ acts as a learning rate. This algorithm performs gradient descent on average when the components of the weight perturbation vector are mutually uncorrelated with uniform autovariance, with error decreasing in each epoch for small enough π and μ, and with a slowdown with respect to gradient descent proportional to the square root of the number of parameters.

Architecture-Coupled Methods. A number of training algorithms for DTRNNs are coupled to a particular architecture: for example, BPS (59) is a special algorithm used to train local feedback nets, that is, DTRNNs in which the value of a state unit $x_i[t]$ is computed by using only its previous value $x_i[t-1]$ and not the rest of the state values $x_j[t-1], j \neq i$ (in particular, BPS is neither a special case of BPTT nor of RTRL; it is local in both space and time). But sometimes not only are learning algorithms specialized on a particular architecture, but also they modify the architecture during learning. One such algorithm is Fahlman's *recurrent cascade correlation* (48), which is described below.

Recurrent Cascade Correlation. Fahlman (48) has recently proposed a training algorithm that establishes a mechanism to grow a DTRNN during training by adding hidden state units, which are trained separately so that their output does not affect the operation of the DTRNN. Training starts with an architecture without hidden state units,

$$y_i[t] = g\left(\sum_{j=1}^{n_U} W_{ij}^{yu} u_j[t] + W_i^y\right), \qquad i = 1, \ldots, n_Y \tag{29}$$

and a pool of n_C candidate hidden units with local feedback, which are connected to the inputs, are trained to follow the residual error of the network:

$$x_i[t] = g\left(\sum_{j=1}^{n_U} W_{ij}^{xu} u_j[t] + W_{ii}^{xx} + x_i[t-1] + W_i^x\right) \tag{30}$$

with $i = 1, \ldots, n_C$. Training adds the best candidate unit to the network in a process called *tenure*. If there are already k tenured hidden units, the state of candidate i is

$$x_i[t] = g\left(\sum_{j=1}^{n_U} W_{ij}^{xu} u_j[t] + W_{ii}^{xx} + x_i[t-1]\sum_{j=1}^{k} W_{ij}^{xx'} x_j[t] + W_i^x\right) \qquad (31)$$

Tenure adds the best of the candidates to the network as a hidden unit labeled $k + 1$ (where k is the number of existing hidden units), its incoming weights are frozen, and connections are established with the output units and subsequently trained. Therefore, hidden units form a lower triangular structure in which each of the units receives feedback only from itself and the output is computed from the input and each of the hidden units:

$$y_i[t] = g\left(\sum_{j=1}^{n_U} W_{ij}^{yu} u_j[t] + \sum_{j=1}^{k} W_{ij}^{ux} x_j[t] + W_i^y\right), \qquad i = 1, \ldots, n_Y \qquad (32)$$

Learning Problems. When one comes to train a DTRNN to perform a certain sequence processing task, the first thing that should be checked is whether the DTRNN architecture chosen can actually represent or approximate the task that we want to learn. However, this is seldom possible, either because of our incomplete knowledge of the computational nature of the sequence-processing task itself, or because of our lack of knowledge about the tasks that a given DTRNN architecture can actually perform. In most of the following, we will assume that the DTRNN architecture (including the representation used for inputs, the interpretation assigned to outputs, and the number of neurons in each layer) has already been chosen and that further learning may only occur through adjustment of weights, biases, and similar parameters. We will review some of the problems that may occur during the adjustment of these parameters. Of these, some may appear regardless of the kind of learning algorithm used, and others may be related to gradient-based algorithms.

Multiple Minima. The error function for a given sample is usually a function of a fairly large number of parameters. For example, small DTRNN, say, an Elman net (see the subsection "Neural Moore Machines) with two inputs, two output units, and three state units has 21 weights, 5 biases, and, in case we decide to adjust them, 3 initial state values. Assume we have already found a minimum in the error surface. Due to the structure of connections, choosing any of the six possible permutations of the three state neurons would yield exactly the same value for the error function. But, in addition to this, it is very likely that the 26-dimensional space of weights and biases is plagued with local minima, some of which may actually not correspond to the computational task we want to learn. Since it is not feasible for any learning algorithm to sample the whole 26-dimensional space, the possibility that it finds a suboptimal minimum of the error function is very large. This problem is especially important with local-search algorithms such as gradient descent: if the algorithm slowly modifies the learnable parameters to go downhill on the error surface, it may end up trapped in any local minimum. The problem of multiple minima is not even a specific problem of DTRNN; it affects almost all neural net architectures. For a study of local minima in DTRNNs and, in particular, for conditions under which local minima may be avoided in DTRNNs, see 60.

Long-Term Dependences. The problem of long-term dependences, when training a DTRNN to perform tasks in which a late output depends on a very early input that has to be remembered, is more specific to DTRNNs, because it is a sequence-processing problem; one of the most exhaustive studies of this problem may be found in 61. The problem may be formulated as follows: when the sequence-processing task is such that the output after reading a long sequence depends on details of the early items of the sequence, it may occur that learning algorithms are unable to acknowledge this dependence due to the fact that the actual output of the

DTRNN at the current time is very insensitive to small variations in the early input, or, what is equivalent, to the small variations in the weights involved in the early processing of the event (even if the change in the early input is large); this is known as the problem of vanishing gradients (see also 7, p. 773). Small variations in weights are the modus operandi of most learning algorithms, in particular, but not exclusively, of gradient-descent algorithms. Bengio et al. (61) prove that the vanishing of gradients is especially severe when we want the DTRNN to robustly store information about a very early effect.

RELAXATION DISCRETE-TIME RECURRENT NEURAL NETS

When DTRNNs are used in such a way that we are only interested in the output(s) they produce after letting them evolve for a sufficiently large number of input steps, we may assume either that we have no inputs or that we have a constant input that may be modeled as a bias to the corresponding units. We are interested in the final output of the network after either (a) having placed it in a particular initial state or (b) having placed it in a standard initial state and having set the inputs (or biases) to particular values. Formulations (a) and (b) are equivalent. Most of the DTRNN architectures defined in the previous section may be adapted for this kind of processing. Following formulation (a), and using a definition parallel to the one given in the subsection "Discrete-time Recurrent Neural Nets as Neural State Machines," a *relaxation neural state machine* (*RNSM*) N is a quadruple

$$N = (X,\ Y\ \boldsymbol{f}, \boldsymbol{h}) \tag{33}$$

in which all of the elements have the same definition as in Eq. (4). The way in which a RNSM is used is, however, different. The RNSM computes a function $F : X \to Y$ as follows: after setting the initial state $\boldsymbol{x}[0]$ to the desired input vector $\boldsymbol{x}$, it is allowed to perform state transitions until it reaches the stationary state; in a a continuous-state RNSM, the output $\boldsymbol{y} = F(\boldsymbol{x})$ is

$$\lim_{t \to \infty, \boldsymbol{x}_0 = \boldsymbol{x}} y[t]$$

In practice, the network is allowed to evolve either for a fixed number of time steps or until two successive output vectors differ less than a predetermined tolerance. It may be said that the network computes the function F by successive approximations. Of course, it may be possible that, instead of settling to a finite state, the network starts to repeat values in a cyclic fashion or shows chaotic behavior.

There are a number of architectures that may be classified as RNSM, the most representative being perhaps Hopfield networks. Hopfield networks are among the most widely used RNSM architectures. The original Hopfield net (62) is a discrete-state RNSM in which the function $\boldsymbol{h}$ is the identity (and therefore $Y = X$) and the function $\boldsymbol{f}$ is defined as follows: a random unit i is chosen in the range $i = 1, \ldots, n_X$; then, the ith component of $\boldsymbol{f}$ has the form

$$f_i(\boldsymbol{x}[t-1]) = g_H\left(\sum_{j=1}^{n_X} W_{ij}^{xx} x_j[t-1]\right) \tag{34}$$

and the rest of the components are

$$f_j(\boldsymbol{x}[t-1]) = x_j[t-1] \tag{35}$$

where $g_H(x)$ is 1 if $x \geq 0$ and -1 otherwise, and the weights have the property that ${W_{ij}}^{xx} = {W_{ji}}^{xx}$ and $W_{ii} = 0$ for all $i = 1, \ldots, n_X$ (they are *symmetric*; nonsymmetric weights may yield a network that does not settle to a fixed point but instead oscillates in what is called a *limit cycle*). This is the *asynchronous* update mode; the *synchronous* variant or *Little model* (63) uses Eq. () for all of the components. A typical application of Hopfield nets is the recall of a "clean" binary pattern $\boldsymbol{x}_p$ stored in advance, starting from a "noisy" version of it; for example, to recover the transmitted bits from a noisy signal in digital cellular communications (64). Storing patterns in Hopfield nets is easy: one may store up to approximately $n_X/\log n_X$ binary patterns (3, p. 19) by using a version of Hebb's rule (7, p. 55):

$$W_{ij}^{xx} = \sum_{p=1}^{n_\mathrm{P}} d_i^{(p)} d_j^{(p)} \tag{36}$$

where n_p is the number of patterns and $\boldsymbol{d}^{(p)}$ is the pth pattern. The continuous-state version of Hopfield networks (65) has the hyperbolic tangent as an activation function.

CONTINUOUS-TIME RECURRENT NEURAL NETS

CTRNNs may also be either used for temporal processing or allowed to obtain a solution by relaxing. The following subsections briefly review these two approaches.

Continuous-Time Recurrent Neural Nets for Temporal Processing. In CTRNNs, the time variation in the state of each one of the units at time t is a function of the instantaneous state of one or several units at the same time: the activation of the whole net is a continuous function of time. CTRNNs may be used to process continuous-time signals (*CTSs*) much as DTRNNs may be used to process sequences (discrete-time signals): a CTRNN may be used to recognize or classify a CTS, to transform a CTS into another CTS, or to generate a CTS.

If we use a notation parallel to the one used for DTRNNs, the time evolution of state unit i may be expressed as

$$\frac{dx_i}{dt} = F_i(\boldsymbol{x}, \boldsymbol{u}) \tag{37}$$

Similar equations will apply to the output vector $\boldsymbol{y}$. One of the most usual forms for the previous equation is, in a single-layer, fully connected CTRNN,

$$\tau_i^x \frac{dx_i}{dt} = -x_i + g\left(\sum_{j=1}^{n_X} W_{ij}^{xx} x_j\left(t - \delta_{ij}^{xx}\right) + \sum_{j=1}^{n_U} W_{ij}^{xu} u_j\left(t - \delta_{ij}^{xu}\right) + W_i^x\right) \tag{38}$$

where ${\tau^x}_i$ is the time constant of state unit i, ${W^{xx}}_{ij}$ is the matrix of weights connecting state units to state units, ${W^{xu}}_{ij}$ is the matrix of weights connecting inputs to state units, ${W^x}_i$ is the bias of state unit i, ${\delta^{xx}}_{ij}$ is the delay matrix for state-unit–state-unit connections, and ${\delta^{xu}}_{ij}$ is the delay matrix for input–state-unit connections. The

corresponding equation for the output units, defined in analogous terms, is

$$\tau_i^y \frac{dy_i}{dt} = -y_i + g\left(\sum_{j=1}^{n_X} W_{ij}^{yx} x_j\left(t - \delta_{ij}^{yx}\right) + \sum_{j=1}^{n_U} W_{ij}^{yu} u_j\left(t - \delta_{ij}^{yu}\right) + W_i^y\right) \qquad (39)$$

Such a network may be trained, for example, to describe a particular continuous-time trajectory (66).

There exist continuous-time counterparts of DTRNN-training algorithms similar to BPTT and RTRL (for a review, see 67). *Batch* training algorithms try to minimize the time integral of the error

$$E = \int_0^T dt \sum_{i=1}^{n_Y} [y_i(t) - d_i(t)]^2 \qquad (40)$$

whereas *online* algorithms rather try to minimize the instantaneous error $\Sigma^{ny}{}_{i=1}\,[y_i(t) - d_i(t)]^2$ at each time.

Differentiation of Eqs. (37) and (39) with respect to any weight w [also with respect to any time constant τ (68)] yields a system of differential equations, which may be numerically integrated forward in time [as the dynamics of the CTRNN, Eqs. (37) and (39), is simulated] to obtain the instantaneous values of the derivatives of states with respect to each weight ($\partial x_i/\partial w$, $\partial y_i/\partial w$). These values may be either (a) used for the numerical integration of the derivative of the total error with respect to the weight $\partial E/\partial w$ (for batch updating of derivatives) or (b) used online to compute the instantaneous time derivative of each weight dw/dt, which may then be numerically integrated in a forward fashion.

Relaxation Continuous-Time Recurrent Neural nets. As with DTRNNs (see the preceding section), the state units of a CTRNN may also be initialized with a certain pattern, and the CTRNN allowed to evolve until it reaches a fixed point; then the states of output units are read, without paying much attention to the actual temporal evolution that occurred (the reader is reminded that it may be possible for a CTRNN to settle to a non-fixed-point behavior such as a limit cycle or even chaos). A CTRNN used in this way may be used to compute functions that transform vectors into vectors. Pineda (69) and Almeida (70) independently found an efficient gradient-descent training algorithm for these networks. This algorithm (see also 3, p. 172) builds another CTRNN (the error-propagation network) that has the same topology and settles to the derivative of the error function.

FURTHER READING

The interested reader may find excellent chapters on recurrent neural nets in textbooks (such as Chap. 15 in 7 or Chap. 7 in 3) as well as complete monographs devoted to the subject (71, 72).

BIBLIOGRAPHY

1. A. S. Weigend N. A. Gershenfeld (eds.) *Time Series Prediction: Forecasting the Future and Understanding the Past*, Proc. NATO Advanced Research Workshop on Comparative Time Series Analysis, Santa Fe, NM, May 14–17, 1992, Reading, MA: Addison-Wesley, 1993.
2. A. V. Oppenheim R. W. Schafer *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989.
3. J. Hertz A. Krogh R. G. Palmer *Introduction to the Theory of Neural Computation*, Redwood City, CA: Addison-Wesley, 1991.
4. R. Sluijter *et al.* State of the art and trends in speech coding, *Philips J. Res.*, **49** (4): 455–488, 1995.

5. J. E. Hopcroft J. D. Ullman *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison–Wesley, 1979.
6. Z. Kohavi *Switching and Finite Automata Theory*, 2nd ed., New York: McGraw-Hill, 1978.
7. S. Haykin *Neural Networks—A Comprehensive Foundation*, 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 1998.
8. A. C. Tsoi A. Back Discrete time recurrent neural network architectures: A unifying review, *Neurocomputing*, **15**: 183–223, 1997.
9. J. Pollack The induction of dynamical recognizers, *Mach. Learn.*, **7** (2/3): 227–252, 1991.
10. C. W. Omlin C. L. Giles Constructing deterministic finite-state automata in recurrent neural networks, *J. Assoc. Comput. Mach.*, **43** (6): 937–972, 1996
11. C. L. Giles *et al.* Learning and extracted finite state automata with second-order recurrent neural networks, *Neural Comput.*, **4** (3): 393–405, 1992.
12. J. Cid-Sueiro A. Artes-Rodriguez A. R. Figueiras-Vidal Recurrent radial basis function networks for optimal symbol-by-symbol equalization, *Signal Process.*, **40**: 53–63, 1994.
13. P. Frasconi *et al.* Representation of finite-state automata in recurrent radial basis function networks, *Mach. Learn.*, **23**: 5–32, 1996.
14. T. Robinson F. Fallside A recurrent error propagation network speech recognition system, *Comput. Speech Lang.*, **5**: 259–274, 1991.
15. M. Jordan Serial order: A parallel distributed processing approach, Technical Report 8604, Institute for Cognitive Science, Univ. of California at San Diego, La Jolla, CA, 1986.
16. J. L. Elman Finding structure in time, *Cogn. Sci.*, **14**: 179–211, 1990
17. R. J. Williams D. Zipser A learning algorithm for continually running fully recurrent neural networks, *Neural Comput.*, **1** (2): 270–280, 1989.
18. J. B. Pollack Recursive distributed representations, *Artif. Intell.*, **46**: 77–105, 1990.
19. A. Blair J. B. Pollack Analysis of dynamical recognizers, *Neural Comput.*, **9** (5): 1127–1142, 1997.
20. R. C. Carrasco M. L. Forcada L. Santamaría Inferring stochastic regular grammars with recurrent neural networks, in L. Miclet and C. de la Higuera, (eds.), *Grammatical Inference: Learning Syntax from Sentences*, Proc. Third Int. Colloq. on Grammatical Inference, Montpellier, France, September 1996, 25–27, Berlin: Springer-Verlag, 1996, pp. 274–281.
21. K. S. Narendra K. Parthasarathy Identification and control of dynamical systems using neural networks, *IEEE Trans. Neural Netw.*, **1**: 4–27, 1990.
22. T. Sejnowski C. Rosenberg Parallel Networks that Learn to Pronounce English text, *Complex Syst.*, **1**: 145–168, 1987.
23. N. Qian T. J. Sejnowski Predicting the secondary structure of globular proteins using neural network models, *J. Mol. Biol.*, **202**: 865–884, 1988.
24. A. Waibel *et al.* Phoneme recognition using time–delay neural networks, *IEEE Trans. Acoust. Speech Signal Process.*, **37** (3): 328–339, 1989.
25. K. J. Lang A. H. Waibel G. E. Hinton A time-delay neural network architecture for isolated word recognition, *Neural Netw.*, **3**: 23–44, 1990.
26. G. Kechriotis E. Zervas E. S. Manolakos Using recurrent neural networks for adaptive communication channel equalization, *IEEE Trans. Neural Netw.*, **5**: 267–278, 1994.
27. R. Parisi *et al.* Fast adaptive digital equalization by recurrent neural networks, *IEEE Trans. Signal Process*, **45**: 2731–2739, 1997.
28. R. L. Watrous B. Ladendorf G. Kuhn Complete gradient optimization of a recurrent network applied to /b/, /d/, /g/ discrimination, *J. Acoust. Soc. Am.*, **87**: 1301–1309, 1990.
29. G. Kuhn R. L. Watrous B. Ladendorf Connected recognition with a recurrent network, *Speech Commun.*, **9**: 41–48, 1990.
30. S. Haykin L. Li Nonlinear adaptive prediction of nonstationary signals, *IEEE Trans. Signal Process.*, **43**: 526–535, 1995.
31. T. Adali *et al.* Modeling nuclear reactor core dynamics with recurrent neural networks, *Neurocomputing*, **15** (3–4): 363–381, 1997.
32. Y. Cheng T. W. Karjala D. M. Himmelblau Identification of nonlinear dynamic process with unknown and variable dead time using an internal recurrent neural network, *Ind. Eng. Chem. Res.*, **34**: 1735–1742, 1995.
33. G. V. Puskorius L. A. Feldkamp Neurocontrol of nonlinear dynamical systems with Kalman filter-trained recurrent networks, *IEEE Trans. Neural Netw.*, **5**: 279–297, 1994.

34. T. Chovan T. Catfolis K. Meert Neural network architecture for process control based on the RTRL algorithm, *AIChE J.*, **42** (2): 493–502, 1996.
35. J. Wang G. Wu Recurrent neural networks for synthesizing linear control systems via pole placement, *Int. J. Syst. Sci.*, **26** (12): 2369–2382, 1995.
36. J. T. Connor R. D. Martin Recurrent neural networks and robust time series prediction, *IEEE Trans. Neural Netw.*, **5**: 240–254, 1994.
37. A. Aussem F. Murtagh M. Sarazin Dynamical recurrent neural networks—towards environmental time series prediction, *Int. J. Neural Syst.*, **6**: 145–170, 1995.
38. S. Lawrence C. L. Giles S. Fong Can recurrent neural networks learn natural language grammars? *Proc. ICNN'96*, 1996, pp. 1853–1853.
39. A. Cleeremans D. Servan-Screiber J. L. McClelland Finite state automata and simple recurrent networks, *Neural Comput.*, **1** (3): 372–381, 1989.
40. P. Manolios R. Fanelli First order recurrent neural networks and deterministic finite state automata, *Neural Comput.*, **6** (6): 1154–1172, 1994.
41. M. Gori *et al.* Inductive inference from noisy examples using the hybrid finite state filter, *IEEE Trans. Neural Netw.*, **9**: 571–575, 1998.
42. P. Tiňo J. Sajda Learning and extracting initial Mealy automata with a modular neural network model, *Neural Comput.*, **7** (4): 822–844, 1995.
43. N. Alon A. K. Dewdney T. J. Ott Efficient simulation of finite automata by neural nets, *Assoc. Comput. Mach.*, **38** (2): 495–514, 1991.
44. B. G. Horne D. R. Hush Bounds on the complexity of recurrent neural network implementations of finite state machines, *Neural Netw.*, **9** (2): 243–252, 1996.
45. S. C. Kremer On the computational power of Elman-style recurrent networks, *IEEE Trans. Neural Netw.*, **6**: 1000–1004, 1995.
46. R. C. Carrasco *et al.* Stable encoding of finite-state machines in discrete-time recurrent neural nets with sigmoid units, *Neural Comput.*, **12**: 2129–2174, 2000.
47. H. T. Siegelmann E. D. Sontag Turing computability with neural nets, *Appl. Math. Lett.*, **4** (6): 77–80, 1991.
48. S. E. Fahlman The recurrent cascade-correlation architecture, in R. P. Lippmann, J. E. Moody, D. S. Touretzky, (eds.), *Advances in Neural Information Processing Systems 3*, Denver: Morgan Kaufmann, 1991, pp. 190–196.
49. M. L. Forcada R. C. Carrasco Learning the initial state of a second-order recurrent neural network during regular-language inference, *Neural Comput.*, **7** (5): 923–930, 1995.
50. J. Schmidhuber S. Hochreiter Guessing can outperform many long time lag algorithms, Technical Note IDSIA-19-96, IDSIA, 1996.
51. R. J. Williams D. Zipser Gradient-based learning algorithms for recurrent networks and their computational complexity, in Y. Chauvin and D. E. Rumelhart, (eds.), *Back-propagation: Theory, Architectures and Applications*, Hillsdale, NJ: Lawrence Erlbaum 1995, Chap. 13, pp. 433–486.
52. D. Rumelhart G. Hinton R. Williams Learning internal representations by error propagation, in *Parallel Distributed Processing*, Cambridge, MA: MIT Press, 1986, Chap. 8.
53. P. J. Werbos Beyond regression: New tools for prediction and analysis in the behavioral sciences, Doctoral Dissertation, Applied Mathematics, Harvard Univ., 1974.
54. A. B. Bulsari H. Saxén A recurrent network for modeling noisy temporal sequences, *Neurocomput.*, **7** (1): 29–40, 1995.
55. J. H. Schmidhuber A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks, *Neural Comput.* **4** (2): 243–248, 1992.
56. R. J. Williams Training recurrent networks using the extended Kalman filter, *Proc. 1992 Int. Joint Conf. Neural Netw.*, 1992, Vol. 4, pp. 241–246.
57. K. P. Unnikrishnan K. P. Venugopal Alopex: A correlation-based learning algorithm for feedforward and recurrent neural networks, *Neural Comput.*, **6** (3): 469–490, 1994.
58. G. Cauwenberghs A fast-stochastic error-descent algorithm for supervised learning and optimization, in *Advances in Neural Information Processing Systems 5*, San Mateo, CA: Morgan Kaufmann, 1993, pp. 244–251.
59. M. Gori Y. Bengio R. De Mori BPS: A learning algorithm for capturing the dynamical nature of speech, *Proc. IEEE-IJCNN89*, Washington, 1989.

60. M. Bianchini M. Gori M. Maggini On the problem of local minima in recurrent neural networks, *IEEE Trans. Neural Netw.*, **5**: 167–177, 1994.
61. Y. Bengio P. Simard P. Frasconi Learning long-term dependencies with gradient descent is difficult, *IEEE Trans. Neural Netw.*, **5**: 157–166, 1994.
62. J. J. Hopfield Neural networks and physical systems with emergent computational abilities, *Proc. Nat. Acad. Sci. U.S.A.*, **79**: 2554, 1982.
63. W. Little The existence of persistent states in the brain, *Math. Biosci.*, **19**: 101–120, 1974.
64. G. I. Kechriotis E. S. Manolakos Hopfield neural network implementation of the optimal CDMA multiuser detector, *IEEE Trans. Neural Netw.*, **7** (1): 131–141, 1996.
65. J. Hopfield Neurons with graded responses have collective computational properties like those of two-state neurons, *Proc. Nat. Acad. Sci. U.S.A.*, **81**: 3088–3092, 1984.
66. B. Pearlmutter Learning state space trajectories in recurrent neural networks, *Neural Comput.*, **1** (2): 263–269, 1989.
67. B. A. Pearlmutter Gradient calculations for dynamic recurrent neural networks: A survey, *IEEE Trans. Neural Netw.*, **6**: 1212–1228, 1995.
68. S. Day M. Davenport Continuous-time temporal back-progagation with adaptive time delays, *IEEE Trans. Neural Netw.*, **4**: 348–354, 1993.
69. F. J. Pineda Generalization of back-propagation to recurrent neural networks, *Phys. Rev. Lett.*, **59** (19): 2229–2232, 1987.
70. L. Almeida Backpropagation in perceptrons with feedback, in R. Eckmiller and C. von der Malsburg (eds.), *Neural Computers*, Berlin: Springer-Verlag, 1988, pp. 199–208.
71. L. Medsker L. Jain *Recurrent Neural Networks: Design and Applications*, Boca Raton, FL CRC Press, 2000.
72. J. Kolen S. Kremer, (eds.) *A Field Guide to Dynamical Recurrent Networks*, IEEE Press, New York: Wiley, 2001.

MIKEL L. FORCADA
Universitat d'Alacant
MARCO GORI
Università di Siena