Chapter 2
Recurrent Neural Networks

Sajid A. Marhon, Christopher J.F. Cameron, and Stefan C. Kremer

1 Introduction

This chapter presents an introduction to recurrent neural networks for readers familiar with artificial neural networks in general, and multi-layer perceptrons trained with gradient descent algorithms (back-propagation) in particular. A recurrent neural network (RNN) is an artificial neural network with internal loops. These internal loops induce recursive dynamics in the networks and thus introduce delayed activation dependencies across the processing elements (PEs) in the network.

While most neural networks use distributed representations, where information is encoded across the activation values of multiple PEs, in recurrent networks a second kind of distributed representation is possible. In RNNs, it is also possible to represent information in the time varying activations of one or more PEs. Since information can be encoded spatially, across different PEs, and also temporally, these networks are sometimes also called Spatio-Temporal Networks [29].

In a RNN, because time is continuous, the PE activations form a dynamical system which can be described by a system of integral equations; and PE activations can be determined by integrating the contributions of earlier activations over a temporal kernel. If the activation of PE j at time t is denoted by $y_j(t)$, the temporal kernel by $k(\cdot)$; and, f is a transformation function (typically a sigmoidal non-linearity), then

$$y_j(t) = f\left(\sum_i \int_{t'=0}^{t} k_{ji}(t'-t)\cdot y_i(t')\partial t'\right). \tag{1}$$

This formulation defines a system of integral equations in which the variables $y_j(t)$, for different values of j, depend temporally on each other. If we place no restrictions on the indices, i and j, in the kernels, k, then these temporal dependencies can

Sajid A. Marhon · Christopher J.F. Cameron · Stefan C. Kremer
The School of Computer Science at the University of Guelph,
Guelph, Ontario
e-mail: {smarhon,ccameron,skremer}@uoguelph.ca

M. Bianchini et al. (Eds.): *Handbook on Neural Information Processing*, ISRL 49, pp. 29–65.
DOI: 10.1007/978-3-642-36657-4_2 © Springer-Verlag Berlin Heidelberg 2013

contain loops. If we do place dependencies on the indices then we can restrict the graphical topologies of the dependencies to have no loops. In this case, the only temporal behaviour is that encoded directly in the kernel, k, that filters the inputs to the network. In both cases, but particularly in the latter, the kernel, k, is defined by a design choice to model a specific type of temporal dependency.

For the purposes of simulating these systems on digital computers, it is convenient to describe the evolution of the activation values in the models at fixed, regular time intervals. The frequency of the simulation required to maintain fidelity with the original model is governed by Nyquist's Theorem [45]. In this case, the dynamics of the model can be expressed without the necessity of an explicit integration over a temporal kernel. In fact, many models are described only in terms of activations that are defined on the activations at a previous time-step. If w_{ji} is used to represent the impact of the previous time-step's value of PE i on PE j, then

$$y_j(t) = f\left(p_j(t)\right), \tag{2}$$

where,

$$p_j(t) = \sum_i w_{ji} \cdot y_i(t-1). \tag{3}$$

There is a number of operational paradigms in which RNNs can be applied:

- **Vector to Vector mapping** — Conventional multi-layer perceptrons (MLPs) map vectors of inputs into vectors of outputs (possibly using some intermediate hidden layer activations in the process). MLPs can be viewed as special cases of RNNs in which there either are no recurrent connections, or all recurrent connections have weights of zero. Additionally, winner take all, Hopfield networks and other similar networks which receive a static input and are left to converge before an output is withdrawn, fall into this category.
- **Sequence to Vector mapping** — RNNs are capable of processing a sequence of input activation vectors over time, and finally rendering an output vector as a result. If the output vector is then post-processed, to map it to one of a finite number of categories, these networks can be used to classify input sequences. In a degenerate case there may be a single time-varying input, resulting in a device that maps a single input signal to a category; but in general, a number of input values that vary in time can be used.
- **Vector to Sequence mapping** — Less common are RNNs used to generate an output sequence in response to a single input pattern. These are generative models for sequences, in which a dynamical system is allowed to evolve under a constant input signal.
- **Sequence to Sequence mapping** — The final case is the one where both input and output are vector sequences that vary over time. This is the case of sequence transduction and is the most general of the 4 cases. In general, this approach assumes a synchronization between the sequences in the sense that there is a one-to-one mapping between activations values at all points in time among the input and output sequences (for an asynchronous exception see [11]).

The most obvious application of these networks is, of course, to problems where input signals arrive over time, or outputs are required to be generated over time. But, this is not the only way in which these systems are used. Sometimes it is desirable to convert a problem that is not temporal in nature into one that is. A good example of such a problem is one in which input sequences of varying lengths must be processed.

If input sequences vary in length, the use of MLPs may impose an unnatural encoding of the input. One can either: 1) use a network with an input vector large enough to accommodate the longest sequence and pad shorter inputs with zeros, or 2) compress the input sequence into a smaller, fixed-length vector. The first option is impractical for long sequences or when the maximum sequence length is unbounded, and generally leads to non-parsimonious solutions. The second option is very effective when prior knowledge about the problem exists to formulate a compressed representation that does not lose any information that is germaine to the problem at hand. Some approaches to reducing input sequences into vectors use a temporal window which only reveals some parts (a finite vector) of the input sequence to the network, or to use a signal processing technique and feature reduction to compress the information. But, in many cases, such apriori knowledge is not available. Clearly, reducing the amount of information available to the network can be disastrous if information relevant to the intended solution is lost in the process.

Another scenario is one in which input sequences are very long. A long input sequence necessitates a large number of inputs which in turn implies a large number of parameters, corresponding to the connection weights from these inputs to subsequent PEs. If a problem is very simple, then having a large number of parameters typically leads to overfitting. Moreover, if such a network is trained using sequences with some given maximum length, and then evaluated on longer sequences, the network will not only be incapable of correctly generalizing reasonable outputs for these longer patterns, it will not even be able to process a pattern that is longer than its maximum input size at all. By contrast a RNN with only a single input and a moderate number of other PEs could very well solve the problem. The reduced number of parameters in the latter system not only reduces the chance of overfitting, but also simplifies the training process.

Thus, RNNs represent a useful and sometimes essential alternative to conventional networks that every neural network practitioner should be aware of.

This chapter is organized as follows. We begin with a section on "Architecture" in which we present a very general RNN architecture which subsumes all other RNN and MLP models. Next we explore some topologies and some very specific models. We then present a section on "Memory" in which we describe different ways in which RNNs incorporate information about the past. Again we begin with a general model and then present specific examples. The third section of the chapter is about "Learning". In it, we describe the methods for updating the parameters of RNNs to improve performance on training datasets (and hopefully unseen test-data). We also describe an important limitation of all gradient based approaches to adapting the parameters of RNNs. In the section titled "Modeling", we compare these systems to more traditional computational models. We focus on a comparison between

RNNs and finite state automata, but also mention other computational models. The "Applications" section describes some example applications to give the reader some insight into how these networks can be applied to real-world problems. Finally, the "Conclusion" presents a summary, and some remarks on the future of the field.

2 Architecture

In this section, we discuss RNN architectures. As mentioned before, RNNs are a generalization of MLPs, so it is appropriate to begin our discussion with this more familiar architecture. A typical MLP architecture is shown in Figure 1. In this figure, the mid-sized, white circles represent the input units of the network, solid arrows are connections between input and/or PEs, and large circles are PEs. The network is organized into a number of node layers; each layer is fully connected with its preceding and subsequent layers (except of course the original input and terminal output layers). Note that there are no recurrent connections in this network (e.g. amongst PEs in the same layer, or from PEs in subsequent layers to PEs in previous layers). The activation value of each PE y_j in the first PE-layer of this network can be computed as

$$y_j = f(p_j), \tag{4}$$

where $f(p_j)$ is generally a non-linear squashing function, such as the sigmoid function:

$$f(p_j) = \frac{1}{1+e^{-p_j}}, \tag{5}$$

$$p_j = \sum_i w_{ji} \cdot x_i + b_j, \tag{6}$$

where w_{ji} represents the weight of the connection from input i to PE j, p_j is the weighted sum of the inputs to PE j, b_j is a bias term associated with PE j, and x_i is the i-th component of the input vector. For subsequent layers, the formula for the weighted summation (Equation 6) is changed to:

$$p_j = \sum_i w_{ji} \cdot y_i + b_j, \tag{7}$$

where, this time p_j is the weighted sum of the activation values of the PEs in the previous layer.

In a MLP network, this formulation assumes that the weights w_{ji} for any units j and i not in immediately subsequent layers are zero. We can relax this restriction somewhat and still maintain a feedforward neural network (FNN) by requiring only that the weights w_{ji} for $j \leq i$ be zero. This results in a cascade [10] architecture in which every input and every lower indexed unit can connect to every higher index unit. This generalization also allows us to identify an extra input $i = 0$ whose value is

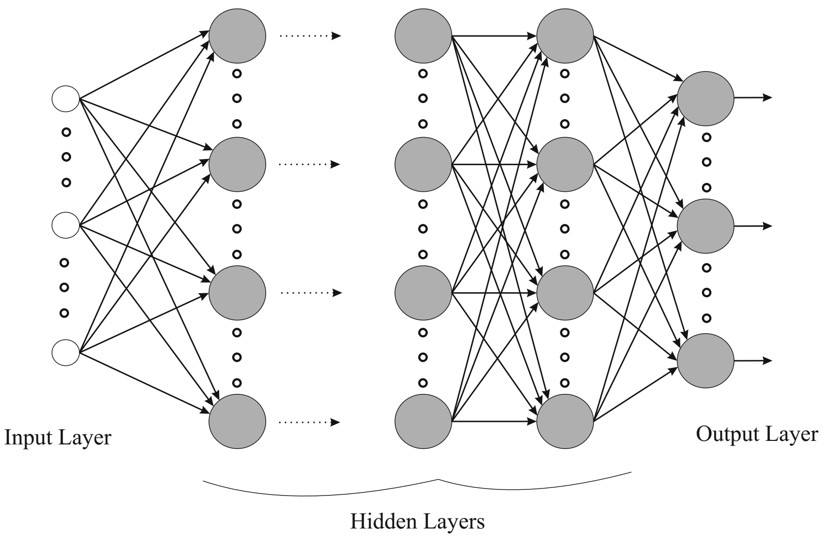


Fig. 1 A multi-layer perceptron neural network

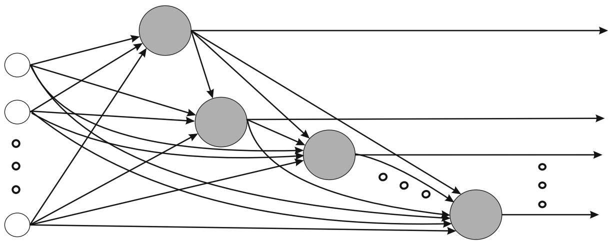


Fig. 2 A cascade architecture of FNN

permanently fixed at $y_0 = 1$ to act as a biasing influence via the connections $w_{j0} = b_j$ to all other PEs j. An example of a cascade architecture is shown in Figure 2.

We now go one step further by removing all restrictions on weights to form a fully connected recurrent network (FCRN) in which every PE is connected to every other PE (including itself). At this stage, it becomes necessary to introduce a temporal index to our notation in order to disambiguate the activation values and unsatisfiable equalities. So, we can now formulate the weighted summations of the PEs as

$$p_j(t) = \sum_h w_{jh} \cdot y_h(t-1) + \sum_i w_{ji} \cdot x_i(t), \tag{8}$$

where the index h is used to sum over the PEs, and the index i sums over the inputs. We assume that the weight values do not change over time (at least not at the same

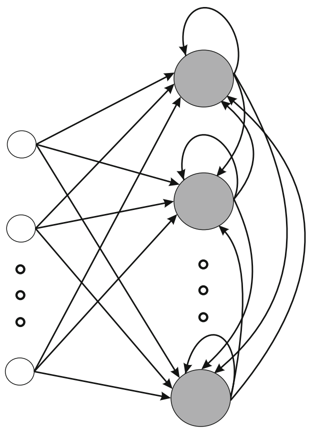


Fig. 3 A fully-connected recurrent network

scale as the activation values). A FCRN is shown in Figure 3. Note that these figures become very difficult to draw as the number of PEs are increased, since there are not topological restrictions to organize elements into layers. However, it is possible to produce a simplified illustration that matches Equation 8 as shown in Figure 4. Note that in this figure time is being represented spatially by the appearance of an additional, virtual layer of PEs that are in fact the same PEs as already shown, but in a previous time-step. This virtual layer has been called a "context" layer [8]. We add a series of additional connections labeled z^{-1} from the PEs to the context units in order to indicate the temporally delayed copy of activation values implied in the system. Figure 5 shows the RNN architecture in Figure 4 as a block diagram. In this figure and all the following figures that present block diagrams in this chapter, the thick arrows indicate full connectivity (many-to-many) between the units in the linked sets. However, the thin bold arrows indicate the unit-to-unit connectivity (one-to-one).

There are a few important things to note about the FCRN architecture presented. First, it is the most general. The MLP and FNN architectures can be implemented within the FCRN paradigm by restricting some of the weights to zero values as indicated in the first two paragraphs of this section. Second, any discrete time RNN can be represented as a FNN. This includes the specific layered recurrent networks discussed in the following sub-sections. Third, the networks introduce an additional parameter set whose values need to be determined. Namely, the initial values of the PE activations $y_j(0)$. Finally, the illustration of activation value calculation in Figure 4 can be solved recursively all the way to the base-case at $t' = 0$, as shown in Figure 6.

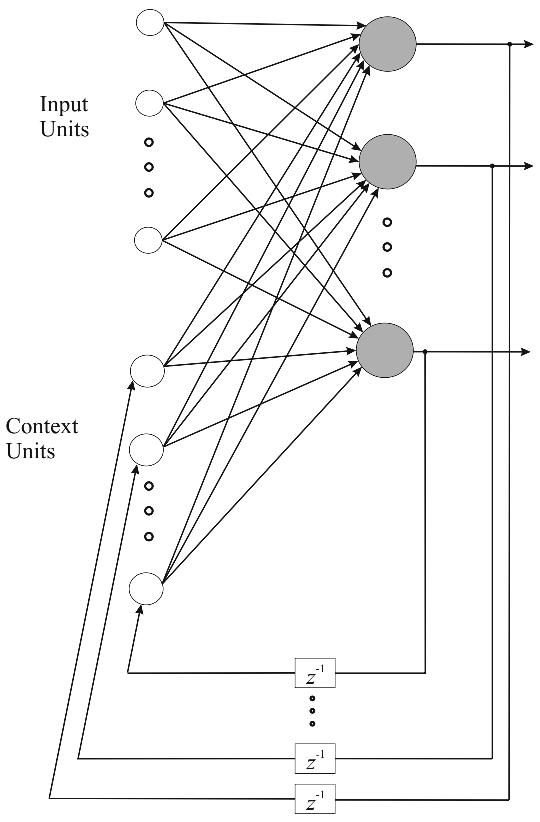


Fig. 4 A RNN including a context layer

2.1 Connectionist Network Topologies

While the general FCRN described in the previous subsection is often used, many other RNNs are structured in layers. A RNN includes an input layer, output layer and typically one or more hidden layers. Each layer consists of a set of PEs. The feedback connections, which are specific to RNNs, can exist within or between any of the network layers. Typically, the inputs to the PE, in a RNN, are from other PEs in a preceding layer and delayed feedback from the PE itself or from other PEs in the same layer or in a successive layer. The sum of the inputs is presented as an activation to a nonlinear function to produce the activation value of the PE.

In RNNs, the topology of the feedforward connections is similar to MLPs. However, the topology of feedback connections, which is limited to RNNs, can be classified into locally recurrent, non-local recurrent and globally recurrent connections. In locally recurrent connections, a feedback connection originates from the output of a PE and feeds back the PE itself. In non-local recurrent connections, a feedback connection links the output of a PE to the input of another PE in the same layer.

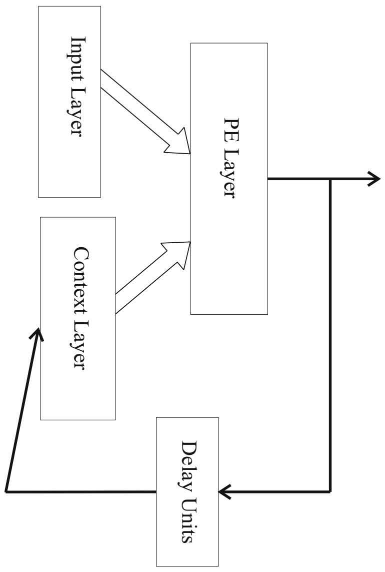


Fig. 5 The block diagram of the RNN architecture in Figure 4

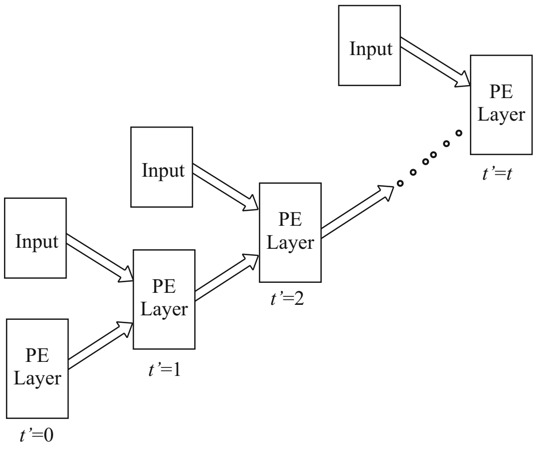


Fig. 6 The unrolled architecture

In globally recurrent connections, the feedback connection is between two PEs in different layers. If we extend this terminology to feedforward connections, all MLPs are considered as global feedforward networks. The non-local recurrent connection class is a special case of the globally recurrent connection class. Based on the feedback topologies, the architecture of RNNs can take different forms as follows:

2.1.1 Locally Recurrent Globally Feedforward (LRGF) Networks

In this class of recurrent networks, recurrent connections can occur in a hidden layer or the output layer. All feedback connections are within the intra PE level. There are no feedback connections among different PEs [51]. When the feedback connection is in the first PE layer of the network, the activation value of PE j is computed as follows:

$$y_j(t) = f\Big(w_{jj} \cdot y_j(t-1) + \sum_i w_{ji} \cdot x_i(t)\Big) \tag{9}$$

where w_{jj} is the intensity factor at the local feedback connection of PE j, the index i sums over the inputs, and $f(\cdot)$ is a nonlinear function, usually a sigmoid function as denoted in Equation 5. For subsequent layers, Equation 9 is changed to:

$$y_j(t) = f\Big(w_{jj} \cdot y_j(t-1) + \sum_i w_{ji} \cdot y_i(t)\Big) \tag{10}$$

where $y_i(t)$ are the activation values of the PEs in the preceding PE layer.

There are three different models of LRGF networks depending on the localization of the feedback.

Local Activation Feedback — In this model, the feedback can be a delayed version of the activation of the PE. The local activation feedback model was studied by [12]. This model can be described by the following equations:

$$y_j(t) = f\big(p_j(t)\big), \tag{11}$$

$$p_j(t) = \sum_{t'=1}^{m} w_{jj}^{t'} \cdot p_j(t-t') + \sum_i w_{ji} \cdot x_i(t), \tag{12}$$

where $p_j(t)$ is the activation at the time step t, t' is a summation index over the number of delays in the system, the index i sums over the system inputs, and $w_{jj}^{t'}$ is the weight of activation feedback of $p_j(t-t')$. Figure 7 illustrates the architecture of this model.

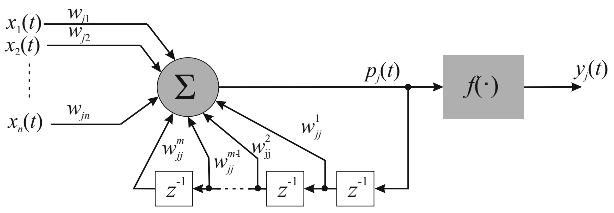


Fig. 7 A PE with local activation feedback

Local Output Feedback — In this model, the feedback is a delayed version of the activation output of the PE. This model was introduced by Gori *et al.* [19]. This feedback model can be illustrated as in Figure 8a, and its mathematical formulation can be given as follows:

$$y_j(t) = f\Big(w_{jj} \cdot y_j(t-1) + \sum_i w_{ji} \cdot x_i(t)\Big). \tag{13}$$

Their model can be generalized by taking the feedback after a series of delay units and the feedback is fed back to the input of the PE as illustrated in Figure 8b. The mathematical formulation can be given as follows:

$$y_j(t) = f\Big(\sum_{t'=1}^{m} w_{jj}^{t'} \cdot y_j(t-t') + \sum_i w_{ji} \cdot x_i(t)\Big), \tag{14}$$

where $w_{jj}^{t'}$ is the intensity factor of the output feedback at time delay $z^{-t'}$, and the index i sums over the input units. From Equation 14, it can be noticed that the output of the PE is filtered by a finite impulse response (FIR) filter.

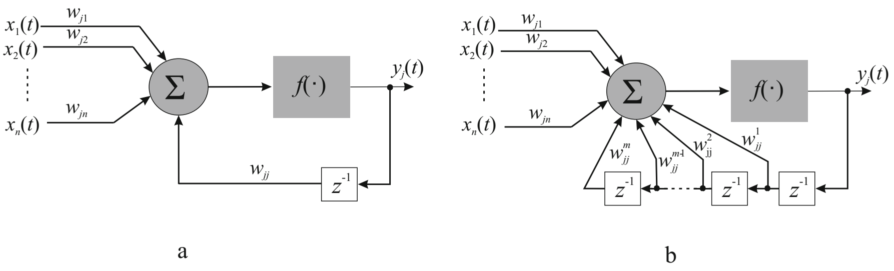


Fig. 8 A PE with local output feedback. a) With one delay unit. b) With a series of delay units.

Local Synapse Feedback — In this model, each synapse may include a feedback structure, and all feedback synapses are summed to produce the activation of the PE. Local activation feedback model is a special case of the local synapse feedback model since each synapse represents an individual local activation feedback structure. The local synapse feedback model represents FIR filter or infinite impulse response (IIR) filter [3]. A network of this model is called a FIR MLP or IIR MLP when the network incorporates FIR synapses or IIR synapses respectively since the globally feedforward nature of this class of networks makes it identical to MLP networks [3, 32]. Complex structures can be designed to incorporate combination of both FIR synapses and IIR synapses [3].

In this model, a linear transfer function with poles and zeros is introduced with each synapse instead of a constant synapse weight. Figure 9 illustrates a PE architecture of this model. The mathematical description of the PE can be formulated as follows:

$$y_j(t) = f\Big(\sum_i G_i(z^{-1}) \cdot x_i(t)\Big), \tag{15}$$

$$G_i(z^{-1}) = \frac{\sum_{l=0}^{q} b_l z^{-l}}{\sum_{l=0}^{r} a_l z^{-l}}, \tag{16}$$

where $G_i(z^{-1})$ is a linear transfer function, and b_l $(l = 0,1,2,\cdots,q)$ and a_l $(l = 0,1,2,\cdots,r)$ are its zeros' and poles' coefficients respectively.

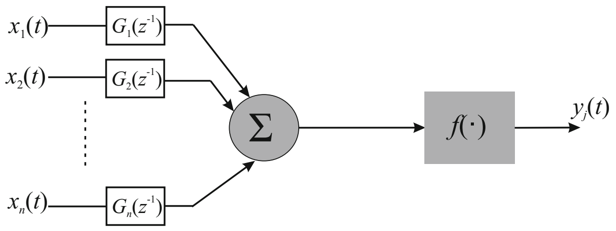


Fig. 9 A PE with local synapse feedback

2.1.2 Non-local Recurrent Globally Feedforward (NLRGF) Networks

In this class of RNNs, the feedback connections to a particular PE are allowed to originate from the PE itself (like LRGF networks) and from other PEs in the same layer. Some researchers classify this type of feedback connections (non-local) as global feedback connections [37]. Based on the description of NLRGF networks, Elman [8] and Williams-Zipser [54] architectures are considered examples of this class of networks [8, 26, 54]. Non-local feedback connections can appear in the hidden or output layers. The mathematical description of PE j that has non-local connections in the first PE layer can be given as follows:

$$y_j(t) = f\Big(\sum_h w_j^h \cdot y_h(t-1) + \sum_i w_{ji} \cdot x_i(t)\Big), \tag{17}$$

where w_j^h is the weight of the feedback connections including the non-local connections from PE h to PE j $(j \neq h)$ and the local connection from the PE itself $(j = h)$, the index h sums over the PEs, and the index i sums over the inputs. For subsequent layers, Equation 17 is changed to:

$$y_j(t) = f\Big(\sum_h w_j^h \cdot y_h(t-1) + \sum_i w_{ji} \cdot y_i(t)\Big), \tag{18}$$

where the index i of the summation sums over the PEs in the preceding PE layer.

2.1.3 Globally Recurrent Globally Feedforward (GFGR) Networks

In this class of recurrent networks, the feedback connections are in the inter layer level. The feedback connections originate from PEs in a certain layer and feed back PEs in a preceding layer. It can be from the output layer to a hidden layer, or it can be from a hidden layer to a preceding hidden layer. Like the MLP, feedforward connections are global connections from a particular layer to the successive layer. The difference between this class and the non-local recurrent network class is that in the latter the feedback connections are in the intra layer level, while in the former the feedback connections are in the inter layer level. When the first PE layer in the network receives global recurrent connections from a successive PE layer, the activation value of PE j in the layer that receives this feedback can be computed as follows:

$$y_j(t) = f\Big(\sum_h w_j^h \cdot y_h(t-1) + \sum_i w_{ji} \cdot x_i(t)\Big), \tag{19}$$

where $y_h(t)$ is the output of PE h in a successive layer which the feedback connections originate from, and the index i sums over the inputs of PE j. For subsequent layers Equation 19 is changed to:

$$y_j(t) = f\Big(\sum_h w_j^h \cdot y_h(t-1) + \sum_i w_{ji} \cdot y_i(t)\Big). \tag{20}$$

Jordan's second architecture is one of the models of this class of recurrent networks [25]. In this architecture, the feedback connections from the output layer are fed back to the hidden layer through a context layer. A unit in the context layer serves as an intermediate state in the model. Figure 10 illustrates the block diagram of the Jordan's second architecture.

There are other classes of RNN topologies which incorporate two of the previously mentioned topologies. For example, the locally recurrent globally recurrent (LRGR) class represents models that include globally recurrent connections as well

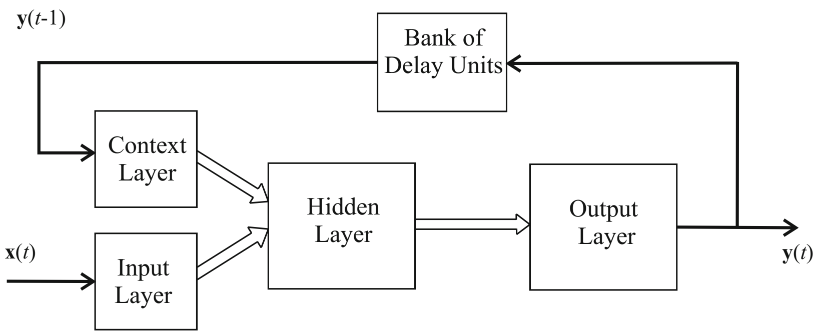


Fig. 10 A block diagram of the Jordan's 2nd architecture

as locally recurrent connections. Another class of RNNs includes networks incorporating non-local recurrent connections and globally recurrent connections at the same time [41].

2.2 *Specific Architectures*

In this section, we present some common RNN architectures which have been proposed in the literature. These architectures can be related to the classes mentioned in Subsection 2.1.

2.2.1 Time Delay Neural Networks (TDNN)

This architecture was proposed by Sejnowski and Rosenberg [44] and applied by Waibel et al. to phoneme recognition [52], and it is a variation of the MLP. It incorporates delay units at the inputs of the PEs. Each input to the PE is delayed with a series of time delays $z^{-i}, \quad (i = 0, 1, 2, \cdots, N)$. The delayed signals are multiplied by weight factors. The sum of the weighted signals is presented to a nonlinear function which is usually a sigmoid function [52]. The mathematical description of the TDNN PE can be formulated as in Equation 21. The architecture of a basic TDNN PE is illustrated in Figure 11.

$$y_j(t) = f\Big(\sum_{l=1}^{M}\sum_{i=0}^{N} w_{jli} \cdot x_l(t-i)\Big) \tag{21}$$

The TDNN network can relate and compare the current input to the history of that input. In the mentioned phoneme recognition application, this architecture was shown to have the ability to learn the dynamic structure of the acoustic signal. Moreover, it has the property of translation invariance which means the features learned by this model are not affected by time shifts.

2.2.2 Williams-Zipser Recurrent Networks

This model has been introduced in this section as the most general form of RNNs. The architecture was proposed by Williams and Zipser [54]. It is called a real-time recurrent network since it was proposed for real-time applications. The network consists of a single layer of PEs. Each PE receives feedback from all other PEs as well as from itself via time delay units. Thus, a fully-connected network is obtained. In addition, the PEs receive external inputs. Each recurrent connection has a unique, adjustable weight to control the intensity of the delayed signal. The diagram of this model is illustrated in Figure 12. The activation of a PE in this network is same as that in Equation 19. This model can be classified to incorporate both local and non-local recurrent connections.

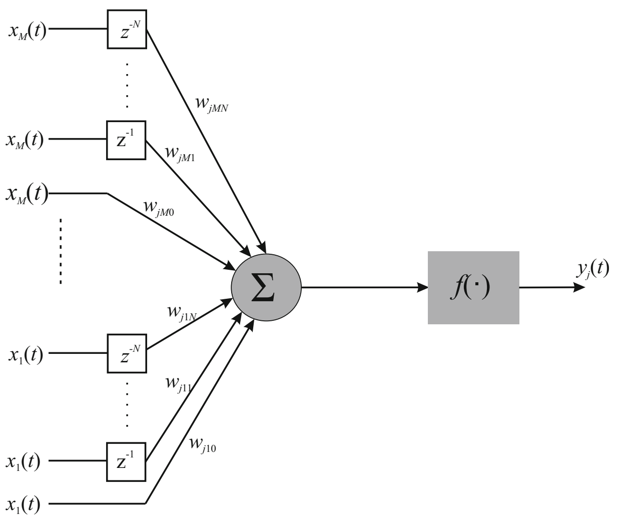


Fig. 11 A typical TDNN PE architecture. The PE has M input signals. Each input signal is delayed by $z^{-0}, \cdots, z^{-N}$. The delayed version of the input as well as the current input (without delay) are multiplied by weights. The weighted sum of all the input signals is computed and presented to a nonlinear function $f(\cdot)$.

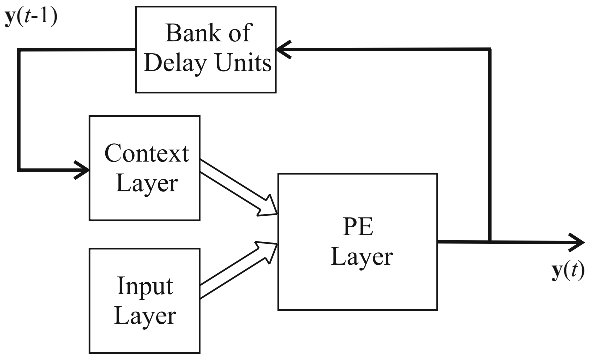


Fig. 12 The architecture of the Williams-Zipser model. Every PE gets feedback from other PEs as well as from itself.

2.2.3 Partially-Connected Recurrent Networks

Unlike the FCRNs, the partially-connected recurrent networks (PCRNs) are based on the MLP. The most obvious example of this architecture is the Elman's [8] and Jordan's [25] models. The Elman model consists of three layers which are the input layer, hidden layer and output layer in addition to a context layer. The context layer receives feedback from the hidden layer, so its units memorize the output of the hidden layer. At a particular time step, the output of the PEs in the hidden layer depends on the current input and the output of the hidden PEs in the previous time step. The mathematical formulation of the output of a hidden PE in this model is given in Equation 17 considering that $y_j(t)$ is the output of hidden PE j. This model can be classified as a non-local recurrent model since each hidden PE receives feedback from itself and from other PEs in the hidden layer. The block scheme of the Elman's model is illustrated in Figure 13.

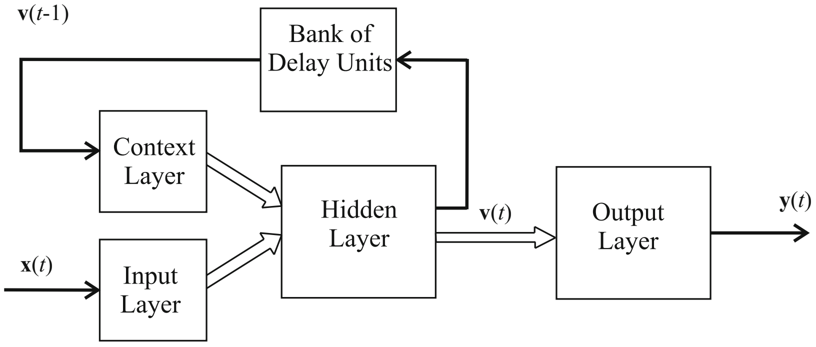


Fig. 13 The architecture of the Elman's model. $\mathbf{y}(t)$ is the network output vector. $\mathbf{x}(t)$ is the input vector. $\mathbf{v}(t)$ is the output vector of the hidden PEs (states).

In contrast to the Elman's model, the Jordan's 2nd model incorporates feedback connections from the output layer to feed back the PEs in the hidden layer via memory unit delays. A context layer receives feedback from the output layer and feeds it to the PEs in the hidden layer. This model is classified as a globally recurrent network since the feedback connections are global between the output and the hidden layers [25]. Figure 10 illustrates the block diagram of the Jordan's 2nd architecture. The mathematical formulation of a hidden PE in the Jordan's 2nd architecture is similar to what has been given in Equation 19.

2.2.4 State-Space Recurrent Networks

This model can include one or more hidden layers. The hidden PEs in a specific layer determine the states of the model. The output of this hidden layer is fed back to that layer via a bank of delay units. The feedback topology of this model can be classified as a non-local recurrent connection class. The number of hidden PEs (states) that feed back the layer can be variant and determine the order of the model [57]. Figure 14 describes the block diagram of the state-space model. The mathematical description of the model is given by the following two equations:

$$\mathbf{v}(t) = \mathbf{f}\big(\mathbf{v}(t-1), \quad \mathbf{x}(t-1)\big), \tag{22}$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{v}(t), \tag{23}$$

where $\mathbf{f}(\cdot)$ is a vector of nonlinear functions, $\mathbf{C}$ is the matrix of the weights between the hidden and output layers, $\mathbf{v}(t)$ is the vector of the hidden PE activations, $\mathbf{x}(t)$ is a vector of source inputs, and $\mathbf{y}(t)$ is the vector of output PE activations.

By comparing the diagrams in Figures 13 and 14, it can be noticed that the architecture of the state-space model is similar to the Elman's (PCRN) architecture except that the Elman's model uses a nonlinear function in the output layer, and there are no delay units in the output of the network. One of the most important features of the state-space model is that it can approximate many nonlinear dynamic functions [57]. There are two other advantages of the state-space model. First, the number of states (the model order) can be selected independently by the user. The other advantage of the model is that the states are accessible from the outside environment which makes the measurement of the states possible at specific time instances [37].

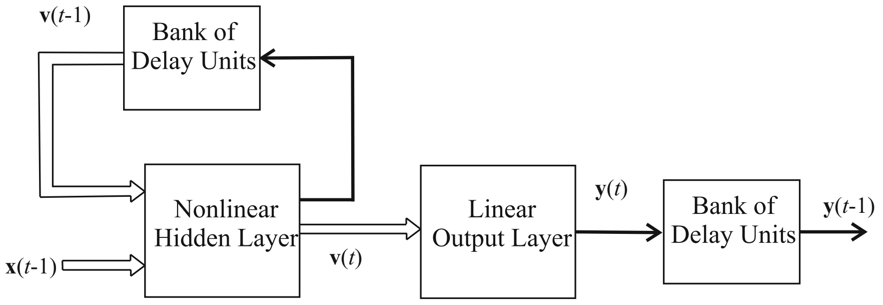


Fig. 14 The block diagram of the state-space model

2.2.5 Second-Order Recurrent Networks

This model was proposed by Giles *et al.* [16]. It incorporates a single layer of PEs. It was developed to learn grammars. The PEs in this model are referred to as second-order PEs since the activation of the next state is computed as the multiplication of the previous state with the input signal. The output of each PE is fed back via a time delay unit and multiplied by each input signal. If the network has N feedback states and M input signals, $N \times M$ multipliers are used to multiply every single feedback state by every single input signal [16]. Thus, the activation value $y_j(t)$ of PE j can be computed as follows:

$$y_j(t) = f\Big(\sum_i \sum_l w_{jil} y_i(t-1) x_l(t-1)\Big), \tag{24}$$

where the weight w_{jil} is applied to the multiplication of the activation value $y_i(t-1)$ and the input $x_l(t-1)$. Figure 15 shows the diagram of a second-order recurrent network.

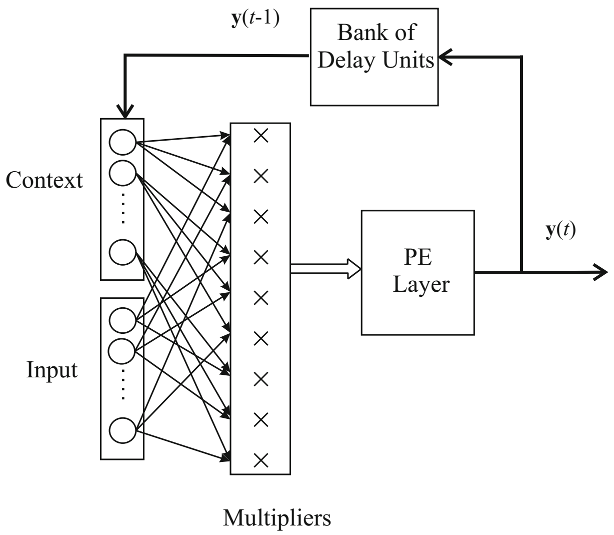


Fig. 15 The block diagram of the second-order recurrent network model

2.2.6 Nonlinear Autoregressive Model with Exogenous Inputs (NARX) Recurrent Networks

In this class of neural networks, the memory elements are incorporated in the input and output layers. The topology of NARX networks is similar to that of the finite memory machines, and this has made them good representative of finite state machines. The model can include input, hidden and output layers. The input to the network is fed via a series of delay units. The output is also fed back to the hidden layer via delay units [6]. The model has been successful in time series and control applications. The architecture of a NARX network with three hidden units is shown in Figure 16. The mathematical description of the model can be given as follows:

$$y(t) = f\Big(\sum_{i=1}^{N} a_i y(t-i) + \sum_{i=1}^{M} b_i x(t-i)\Big), \tag{25}$$

where $x(t)$ is the source input; $y(t)$ is the output of the network; N and M are constants; and a_i and b_i are constants.

In this section, we reviewed the possible topologies of the RNN architectures and the common proposed architectures. The RNN architectures mentioned above have been proposed to tackle different applications. Some models have been proposed for grammatical inference and other models have been proposed for identification and control of dynamic systems. In addition, there is a coordination between the network architecture and the learning algorithm used for training.

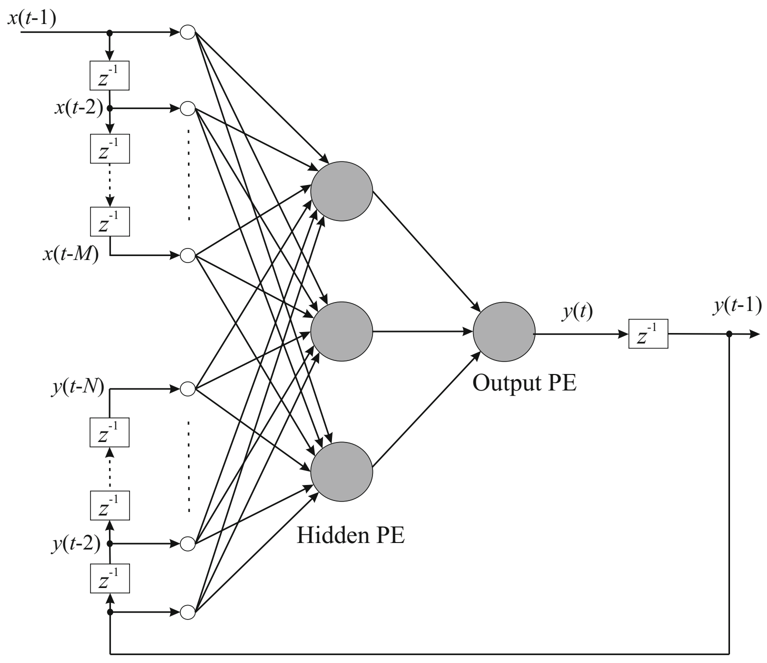


Fig. 16 The architecture of a NARX network

3 Memory

This section of the chapter contains a more descriptive understanding of the importance of the memory for RNNs, how the memory works, and different types of memory.

3.1 Delayed Activations as Memory

Every recurrent network in which activation values from the past are used to compute future activation values, incorporates an implicit memory. This implicit memory can stretch back in time over the entire past history of processing. Consider Figure 5 which depicts a network with recurrent connections within the *PE Layer*. In this illustration, the block labeled *Context Layer* represents a *virtual* set of elements that contain a copy of the *PE Layer*'s elements at the previous time step. Figure 6, shows the same network over a number of time intervals. In this figure, the *PE Layer*'s elements can be seen to depend on the entire history of inputs all the way back to $t' = 0$.

3.2 Short-Term Memory and Generic Predictor

In this subsection, neural networks without global feedback paths, but intra PE level, will be considered. These RNNs consist of two subsystems: a short-term memory and a generic predictor. Short-term memory will be assumed to be of a linear, time invariant, and causal nature. While a generic predictor is considered to be a feed-forward neural network predictor, this predictor consists of nonlinear elements (i.e. a sigmoid function) with associated weights and zero, or more, hidden layers. In the case of this discussion, the generic predictor consists of constant parameters (i.e. weights), nonlinear elements, and it is time invariant.

This structure consisting of the short-term memory and the generic predictor will be considered and referenced as the time invariant nonlinear short-term memory architecture (TINSTMA); the TINSTMA is alternatively known as a memory kernel [34]. The simple structure of the memory kernel is shown in Figure 17.

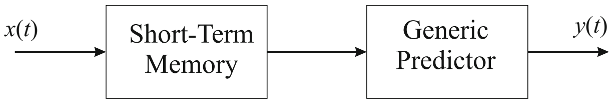


Fig. 17 Example of a memory kernel, the class of RNNs being observed in this section

When developing a TINSTMA, there are 3 issues that must be taken into consideration: architecture, training, and representation. The architecture refers to the internal structural form of a memory kernel, which involves the PE consideration of the number of layers and PEs within the network, the connections formed between these PEs/layers, and the activation function associated to these PEs. Training (as discussed in the next section of the chapter) involves taking into consideration how the kernel (in this case, the internal parameters will be weights) will adapt to the introduction of a set of input patterns to match the targets associated with the input patterns. Representation refers to retention of an input pattern within the short-term memory (i.e. how should it be stored?); nature and quantity of information to be retained is domain dependent. These three issues are views on the same problem, and thus related. The desired representation for a TINSTMA may or may not alter the architecture of a kernel and, consequently, has the possibility of affecting the design of network training. Alternatively a given training design may influence the structure and possible representation for a TINSTMA. In the following subsection, possible types of kernels will be discussed [29].

3.3 Types of Memory Kernels

Memory kernels can typically be considered one of two forms: each modular component consists of the same structural form, but with different parameters (i.e. weights) OR each modular component consists of the same structural form with identical weights.

3.3.1 Modular Components with Different Parameters

In this subsection, cases where each modular component in a memory kernel has the same structural form, but allows different parameters in each node, will be discussed.

Tapped Delay Lines. Considered to be the simplest of memory kernels, a tapped delay line (TDL) is a delay line (a series of nodes) with at least one tap. A delay-line tap extracts a signal output from somewhere within the delay line, optionally scales it, and usually sums it with other taps (if existing) to form an output signal (shown in Figure 18). A tap may be either interpolating or non-interpolating. A non-interpolating tap extracts the signal at some fixed integer delay relative to the input. Tapped delay lines efficiently simulate multiple echoes from the same source signal. Thus, a tap implements a shorter delay line within a larger one. As a result, they are extensively used in the field of artificial reverberation [49].

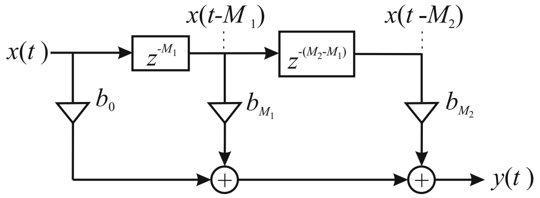


Fig. 18 Example of a TDL network. a TDL consists of an internal tap located at M_1, a total delay length of M_2 samples. Each node may be considered a layer of a multilayer neural network. The output signal of the TDL is a linear combination of the input signal $x(t)$, the delay-line output $x(t-M_2)$, and the tap signal $x(t-M_1)$.

Therefore, the filter output corresponding to Figure 18:

$$y(t) = b_0 x(t) + b_{M_1} x(t-M_1) + b_{M_2} x(t-M_2) \tag{26}$$

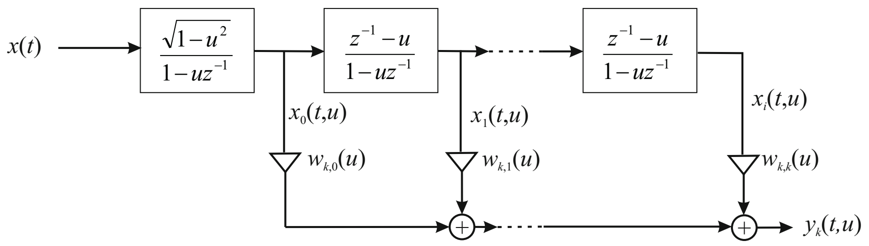
Laguerre Filter. Another popular memory kernel applies a filter based on Laguerre polynomials [55], formulated based on Figure 19 as follows:

$$L_i(z^{-1},u) = \sqrt{1-u^2}\frac{(z^{-1}-u)^i}{(1-uz^{-1})^{i+1}}, \quad i \geq 0, \tag{27}$$

$$y_k(t,u) = \sum_{i=0}^{k} w_{k,i}(u) x_i(t,u), \tag{28}$$

where

$$x_i(t,u) = L_i(z^{-1},u)x(t). \tag{29}$$



$x(t)$
$\frac{\sqrt{1-u^2}}{1-uz^{-1}}$
$\frac{z^{-1}-u}{1-uz^{-1}}$
$\frac{z^{-1}-u}{1-uz^{-1}}$
$x_0(t,u)$
$x_1(t,u)$
$x_i(t,u)$
$w_{k,0}(u)$
$w_{k,1}(u)$
$w_{k,k}(u)$
$y_k(t,u)$

Fig. 19 Example of a Laguerre filter of size k. This filter is stable only if $|u| < 1$. When $u = 0$ the filter degenerates into the familiar transversal filter [48].

FIR/IIR Filters. FIR filters are considered to be nonrecursive since they do not provide feedback ($a_i = 0$, for $i > 0$), compared to IIR or 'recursive' filters which provide feedback ($a_i \neq 0$, for $i > 0$). An example of the format of the filter is shown in the following equations based on Figure 20:

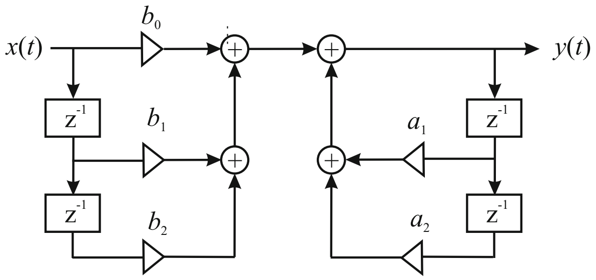


Fig. 20 Example of a second-order IIR filter

$$y(t) = b_0x(t) + b_1x(t-1) + \cdots + b_Mx(t-M) - a_1y(t-1) - \cdots - a_Ny(t-N) \quad (30)$$

$$y(t) = \sum_{i=0}^{M} b_i x(t-i) - \sum_{j=1}^{N} a_j y(t-j) \quad (31)$$

Gamma Filter. The Gamma filter is a special class of IIR filters where the recursion is kept locally, proving effective in identification of systems with long impulse responses [30]. The structure of the Gamma filter is similar to a TDL (refer above). The format of the Gamma filter is shown in the following equations [39]:

$$y(t) = \sum_{k=0}^{K} w_k x_k(t) \quad (32)$$

$$x_k(t) = G(z^{-1})x_{k-1(t)}, \quad k = 1, ..., K, \quad (33)$$

where $y(t)$ is the output signal, $x(t)$ is an input signal, and $G(z^{-1})$ is the generalized delay operator (tap-to-tap transfer function, either recursive or non-recursive).

Moving Average Filter. A Moving Average filter (MAF) is a simplified FIR, alternatively called the exponential filter; and, as the name suggests, it works by averaging a number of points from the input signal to produce each point in the output signal. This filter is considered the moving average, since at any moment, a moving window is calculated using M values of the data sequence [50]. Since the MAF places equal emphasis on all input values, two areas of concern for the MAF is the need for n measurements to be made before a reliable output can be computed and for this computation to occur there needs to be storage of n values [50]. The simplified equation of the MAF is:

$$y(t) = \frac{1}{M} \sum_{j=0}^{M-1} x(t-j), \tag{34}$$

where $y(t)$ is the output signal, $x(t)$ is the input signal, and M is the number of points used in the moving average.

3.3.2 Modular Components with Identical Parameters

In this subsection, cases where each modular component in a memory kernel has the same structural form and node parameter(s) will be discussed. With the possibility of each node's weight being the same, parameter estimation is kept to minimum and each node may be fabricated (i.e. in gate array technology). Note, TDLs may be considered within this section as well.

Modular Recurrent Form. A modular recurrent form is a series of independent nodes organized by some intermediary. Each node provides inputs to the intermediary, which the intermediary processes to produce an output for the network as a whole. The intermediary only accepts nodes outputs, no response (i.e. signal) may be reported back to the node. Nodes do not typically interact with each other.

4 Learning

The most desirable aspect of artificial neural networks is their learning capability. When provided input and output values, a function approximation between these values can be approximated by the network. In this section, an understanding of how a RNN can learn through various learning methods to determine this relationship will be provided.

4.1 Recurrent Back-Propagation: Learning with Fixed Points

Fixed point learning algorithms assume that the network will converge to a stable fixed point. This type of learning is useful for computation tasks such as constraint satisfaction and associative memory tasks. Such problems are provided to the network through an initial input signal or through a continuous external signal, and the solution is provided as the state of the network when a fixed point has been reached.

A problem with fixed point learning is whether or not a fixed point will be reached, since RNNs do not always reach a fixed point [29]. There are several ways to guarantee that a fixed point will be reached with certain special cases:

Weight Symmetry. Linear conditions on weights such as zero-diagonal symmetry ($w_{ij} = w_{ji}, w_{ii} = 0$) guarantee that the Lyapunov function (Equation 35) will decrease until a fixed point has been reached [7]. If weights are considered to be Bayesian constraints, as in Boltzmann Machines, the weight symmetry condition will arise [21].

$$L = -\sum_{j,i} w_{ji} y_i y_j + \sum_i \big(y_i \log(y_i) + (1 - y_i) \log(1 - y_z) \big) \tag{35}$$

Setting Weight Boundaries. If $\sum_{ji} w_{j,i}^2 < \max_x \{f'(x)\}$ where $\max_x \{f'(x)\}$ is the maximal value of $f'(x)$ for any x [2], and $f'(\cdot)$ is the derivative of $f(\cdot)$, convergence to a fixed point will occur. In practice it has been shown that much weaker bounds on weights have an effect [40].

Asymptotic Fixed point Behavior. Some studies have shown that applying the fixed point learning to a network causes the network to exhibit asymptotic fixed point behavior [1, 13]. There is no theoretical explanation of this behavior as of yet, nor replication on larger networks.

Even with the guarantee of a network reaching a fixed point, the fixed point learning algorithm can still have problems reaching that state. As the learning algorithm moves the fixed point location by changing the weights, there is the possibility of the error jumping suddenly due to discontinuity. This occurs no matter how gradually weights are manipulated as the underlying mechanisms are a dynamical system subject to bifurcations, and even chaos [29].

4.1.1 Traditional Back-Propagation Algorithm

The traditional back-propagation algorithm [42, 53] involves the computation of an error gradient with respect to network weights by means of a three-phase process: (1) activation forward propagation, (2) error gradient backward propagation, and (3) weight update. The concept is based on the premise that by making changes in weights proportional to the negation of the error gradient, and the error will be reduced until a locally minimal error can be found. The same premise can be applied to recurrent networks. However, the process of back-propagation becomes complex as soon as recurrent connections are introduced. Also, as we will discover at the end of this section, some complications arise when applying the gradient descent method in recurrent networks.

4.2 Back-Propagation through Time: Learning with Non-fixed Points

The traditional back-propagation algorithm cannot directly be applied to RNNs since the error backpropagation pass assumes that the connections between PEs induce a cycle-free ordering. The solution to the back-propagation through time (BPTT) application is to "unroll" the RNN in time. This "unrolling" involves the stacking of identical copies of the RNN (displayed in Figure 6) and redirecting connections within the network to obtain connections between subsequent copies. The end result of this process provides a feedforward network, which is able to have the backpropagation algorithm applied by back-propagating the error gradient through previous time-steps to $t' = 0$. Note that in Figure 6 the arrows from each of the "Input" rectangles to each of the "PE Layer" rectangles represent different "incarnations" of the exact same set of weights during different time-steps. Similarly, the arrows from each of the "PE Layer" rectangles to their subsequent "PE Layer" rectangles also represent the exact same set of weights in different temporal incarnations. This implies that all the changes to be made to all of the incarnations of a particular weights can be cumulatively applied to that weight. This implies that the traditional equation for updating the weights in a network,

$$\Delta w_{ji} = -\eta a_i \delta_j = -\eta a_i \frac{\partial E}{\partial p_j}, \tag{36}$$

is replaced by a temporal accumulation:

$$\Delta w_{ji} = -\eta \sum_{t'} a_i(t') \delta_j(t'+1) = -\eta \sum_{t'} a_i(t') \frac{\partial E}{\partial p_j(t')}. \tag{37}$$

With the "unrolled" network, a forward propagation begins from the initial copy propagation through the stack updating each connection. For each copy or time t: the input $(u(t))$ is read in, the internal state $(y(t))$ is computed from the input and the previous internal state $(y(t-1))$, and then output $(y(t))$ is calculated. The error (E) to be minimized is:

$$E = \sum_{t=1,\dots,T} \|d(t) - y(t)\|^2 = \sum_{t=1,\dots,T} E(t), \tag{38}$$

where T represents the total length of time, $d(t)$ is the target output vector, and $y(t)$ is the output vector.

The problem with the slow convergence for traditional backpropagation is carried on in BPTT. The computation complexity of an epoch for the network is $O(T\ N^2)$, N is equal to the number of internal/hidden PEs. As with traditional backpropagation, there tends to be the need for multiple epochs (on the scale of 1000s) for a convergence to be reached. It does require manipulation with the network (and processing time) before a desired result can be achieved. This hindrance of the BPTT algorithm tends to lead to smaller networks being used with this design (3-20 PEs), where larger networks tend to go for many hours before convergence.

4.2.1 Real-Time Recurrent Learning

Real-time recurrent learning (RTRL) [54] allows for computation of the partial error gradients at every time step within the BPTT algorithm in a forward-propagated fashion eliminating the need for a temporal back-propagation step (thus, making it very useful for online learning). Rather than computing and recording only the partial derivative of each net value with respect to the total error,

$$\delta_j = \frac{\partial E}{\partial p_j}, \tag{39}$$

it is noted that the total gradient error in a recurrent network is given by:

$$\frac{\partial E}{\partial w_{ji}} = \sum_k (d_k(t) - y_k(t)) \frac{\partial y_k(t)}{\partial w_{ji}}, \tag{40}$$

and that

$$\frac{\partial y_k(t+1)}{\partial w_{ji}} = f'(p_k(t)) \cdot \left(\sum_l w_{lk} \frac{\partial y_l(t)}{\partial w_{ji}} + y_j(t) \right). \tag{41}$$

Note that in this equation, the partial derivatives

$$\frac{\partial y_k(t+1)}{\partial w_{ji}} \tag{42}$$

depend on the previous (not future) values of the same partial derivatives

$$\frac{\partial y_l(t)}{\partial w_{ji}}. \tag{43}$$

This implies that by storing the values

$$\frac{\partial y_k(t+1)}{\partial w_{ji}} \tag{44}$$

at each time-step, the next steps values can be computed until at the final time-step, where a complete error derivative can be computed.

Since w_{kl} has been assumed to be constant, the learning rate or η must be kept small. RTRL has a computational cost of $O(N+L)^4$ for each update step in time ($(N+L)$ dimensional system solved each time), which means that networks employing RTRL must be kept small.

4.2.2 Schmidhuber's Algorithm

Schmidhuber [43] has developed a hybrid algorithm which applies the two previous approaches in a clever, alternating fashion and is able to manage the superior time performance of BPTT while not requiring unbounded memory as sequence length increases.

4.2.3 Time Constants and Time Delays

One advantage of temporal continuous networks is the addition of parameters to control the temporal behavior of the network in ways known to be associated with natural tasks. Time constants represent an example of these additional parameters for networks as shown in [23, 33, 34, 35]. For time delays, consider that a network's connections take a finite period of time between PEs, such that:

$$y_j(t) = \sum_i w_{ji} y_i(t - \tau_{ji}), \tag{45}$$

where τ_{ji} represents the time delay from PE i to PE j.

Using a modification of RTRL, parameters like τ can be learned by a gradient descent approach.

4.3 Long-Term Dependencies

RNNs provide an explicit modeling of time and memory, allowing, in principle, the modelling of any type of open nonlinear dynamical systems. The dynamical system is described by the RNN with a set of real numbers represented by a point in an appropriate state space. There is often an understanding of negativity towards RNNs due to their inability to identify and learn long-term dependencies over time; authors have stated no more than ten steps [4]. Long-term dependencies, in this case, can be defined as a desired output at time T which is dependent on inputs from times t less than T. This difficulty with RNN understanding of long term dependencies has been noted by Mozer, where that RNNs were able to learn short-term musical structure with gradient learning based methods, but had difficulty with a global behavior [34]. Therefore the goal of a network should be to robustly latch information (i.e. a network should be able to store previous inputs with the presence of noise for a long period of time).

As a system robustly latches information, the fraction of the gradient due to information t-steps in the past approaches zero as t becomes large. This is known as the 'problem of vanishing gradients'. Bengio *et al.* [4] and Hochreiter [22] have both noted that the problem of vanishing gradients is the reason why a network trained by gradient-descent methods is unable to distinguish a relationship between target outputs and inputs that occur at a much earlier time. This problem has been termed 'long-term dependencies'.

One way to counteract long-term dependencies is TDL, such as BPTT with a NARX network. BPTT occurs through the process of unrolling the network in time and then back propagating the error through the 'unrolled network' (Figure 6). From this, the output delays will occur as jump ahead connections in the 'unrolled network'. These jump ahead connections provide a shorter path for propagation of the gradient information, and this decreases the sensitivity of a network to long-term dependencies.

Another solution is presented in [23].

5 Modeling

RNNs, which have feedback in their architectures, have the potential to represent and learn discrete state processes. The existence of feedback makes it possible to apply RNN models to solve problems in control, speech processing, time series prediction, natural language processing, and so on [15]. In addition, apriori knowledge can be encoded into these networks which enhances the performance of the applied network models. This section presents how RNNs can represent and model theoretical models such as finite state automata (FSA) and Turing machines (TM). The understanding of the theoretical aspect of these networks in relation to formal models such as FSA is important to select the most appropriate model to solve a given problem. In this section, we will review RNNs for representing FSA and Turing machines.

5.1 Finite State Automata

FSA have a finite number of input and output symbols, and a finite number of internal states. Large portion of discrete processes can be modeled by deterministic finite-state automata (DFA). The mathematical formulation of the FSA M is a 6-tuple and can be defined by:

$$M = \{Q, q_0, \Sigma, \Delta, \delta, \varphi\} \tag{46}$$

where Q is a set of finite number of state symbols: $\{q_1, q_2, \cdots, q_n\}$, n is the number of states, $q_0 \in Q$ is the initial state, Σ is the set of input symbols: $\{u_1, u_2, \cdots, u_m\}$, m is the number of input symbols, Δ is the set of output symbols: $\{o_1, o_2, \cdots, o_r\}$, r is the number of output symbols, δ: $Q \times \Sigma \rightarrow Q$ is the state transition function, and φ is the output function. The class of FSA is basically divided into Mealy and Moore models. Both of the models are denoted by the 6-tuple formulation defined in Equation 46. However, the only difference between the two models is the formulation of the output function φ. In the case of the Mealy model, the output function is φ:$Q \times \Sigma \rightarrow \Delta$, while, in the Moore model, it is: φ:$Q \rightarrow \Delta$ [17, 24]. In general, the FSA M is described by the following dynamic model:

$$M : \begin{cases} q(t+1) = \delta(q(t), u(t)), \quad q(0) = q_0 \\ o(t) = \varphi(q(t), u(t)) \end{cases} \tag{47}$$

Many RNN models have been proven to model FSA. The existence of such equivalence between RNNs and FSA has given the confidence to apply RNN models in solving problems with dynamical systems. Omlin and Giles [36] proposed an algorithm to construct DFA in second-order networks. The architecture of the network is same as the architecture illustrated in Figure 15. The outputs of the PEs in the output layer are the states of the DFA. The network accepts a temporal sequence of inputs and evolves with the dynamic determined by the activation function of the network defined by Equation 24, so the state PEs are computed according to that equation. One of the PEs o_0 is a special PE. This PE represents the output of the network after a string of input is presented to the network. The possible value of this state PE is accept/reject. Therefore, if the modeled DFA has n states and m input symbols, the construction of the network includes $n+1$ state PEs and m inputs. The proposed algorithm is composed of two parts. The first part is to adjust the weights to determine the DFA state transitions. The second part is to adjust the output of the PE for each DFA state.

Kuroe [31] introduced a recurrent network called a hybrid RNN to represent the FSA. In this model, the state symbols q_i $(i = 1, 2, \cdots, n)$, input symbols u_i $(i = 1, 2, \cdots, m)$ and the output symbols o_i $(i = 1, 2, \cdots, r)$ are encoded as binary numbers. $q(t)$, $u(t)$ and $o(t)$ will be expressed as follows:

$$\begin{aligned} q(t) &= (s_1(t), s_2(t), \cdots, s_A(t)) \qquad (s_i(t) \in \{0,1\}) \\ u(t) &= (x_1(t), x_2(t), \cdots, x_B(t)) \qquad (x_i(t) \in \{0,1\}) \\ o(t) &= (y_1(t), y_2(t), \cdots, y_D(t)) \qquad (y_i(t) \in \{0,1\}) \end{aligned} \tag{48}$$

where A, B and D are integer numbers and their values are selected in a way such that $n \leq 2^A$, $m \leq 2^B$ and $r \leq 2^D$ respectively. In this way, there is no need to increase the number of state PEs in the network according to the 1 against 1 basis as the number of states in the FSA increases. The architecture of the hybrid network consists of two types of PEs, which are static and dynamic PEs. The difference between them is that the dynamic PE feedback originates from its output, while the static PE does not [31]. Figure 21 illustrates the diagram of an arbitrary hybrid neural network representing a FSA. The dynamic PEs are represented by darkly-shaded circles, while the static PEs are represented by lightly-shaded circles.

Won *et al.* [55] proposed a new recurrent neural architecture to identify discrete-time dynamical systems (DTDS). The model is composed of two MLPs of five layers. The first MLP is composed of layers 0,1 and 2. The second MPL is composed of layers 3 and 4. The first set of layers $\{0, 1, 2\}$ approximates the states of the FSA, while the second set of layers $\{3, 4\}$ approximates the output of the FSA. Figure 22 shows the architecture of the network proposed by [55].

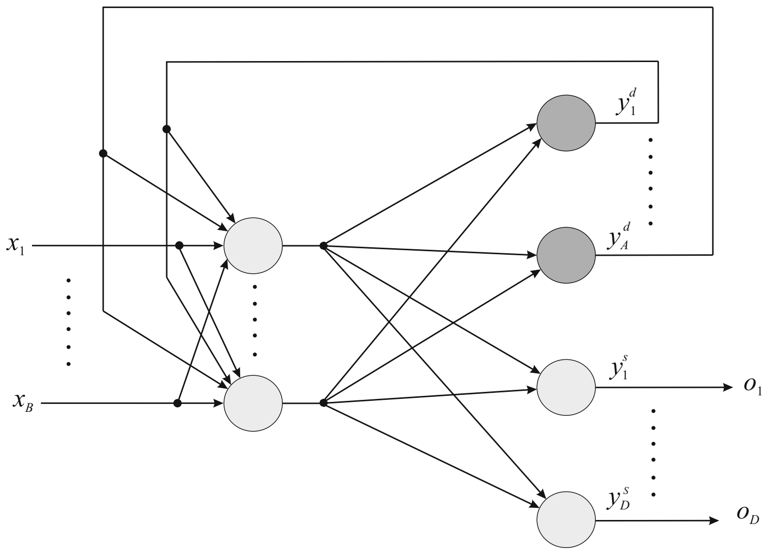


Fig. 21 A hybrid neural network representing a FSA

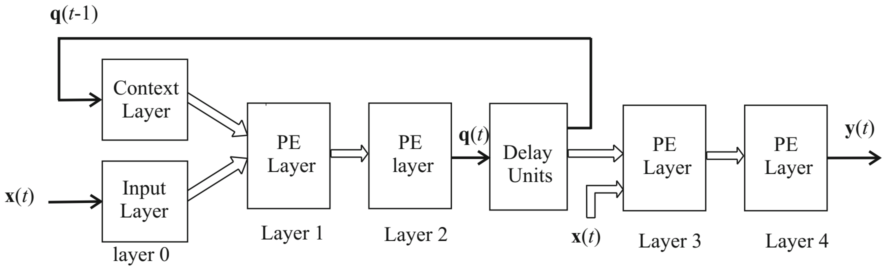


Fig. 22 A RNN composed of two MLPs to identify and extract FSA

5.2 Beyond Finite State Automata

The ability of RNNs to model FSA should inspire great confidence in these systems as FSA represent the computational limits of digital computers with finite memory resources. I.e. anything that a digital computer with finite memory can compute, can also be computed by a FSA, and by a RNN.

Sometimes it is interesting to study even greater levels of computational power by asking questions about what a device could do without memory limitations. This has led to the study, within theoretical computer science of devices such as pushdown automata and Turing machines. It is interesting to note that RNNs measure up to these devices as well. The reader is referred to [28, 38, 47].

6 Applications

The application of RNNs has proved itself in various fields. The spectrum of RNN applications is so wide, and it touched various aspects. Various architectures and learning algorithms have been developed to be applied in solving problems in various fields. The spectrum of application is ranging from natural language processing, financial forecasting, plant modeling, robot control, and dynamic system identification and control. In this section, we review two case studies. One of these cases will be on grammatical inference and the other will be on control.

6.1 Natural Language Processing

In the last years, there has been a lot of efforts and progress in developing RNN architectures for natural language processing. Harigopal and Chen [20] proposed a network to recognize strings which are much longer that the ones which the network was trained on. They used a second-order recurrent network model for the problem of grammatical inference. Zheng *et al.* [59] proposed a discrete recurrent network for learning deterministic context-free grammar. Elman [9] addressed three challenges of natural language processing. One challenge is the nature of the linguistic representations. Second, is the representation of the complex structural relationships. The other challenge is the ability of a fixed resource system to accommodate the open-ended nature of a language.

Grammatical inference is the problem of extracting the grammar from the strings of a language. There exists a FSA that generates and recognizes that grammar. In order to give the reader a clear idea about the application of RNNs in grammatical inference, we will review a method proposed by Chen *et al.* [5] to design a RNN for grammatical inference. Chen *et al.* [5] proposed an adaptive RNN to learn a regular grammar and extract the underlying grammatical rules. They called their model as adaptive discrete recurrent neural network finite state automata (ADNNFSA). The model is based on two recurrent network models, which are the neural network finite state automata (NNFSA) proposed by Giles *et al.* [18] and the discrete neural network finite state automata (DNNFSA) proposed by Zeng *et al.* [58].

Figure 23 shows the network architecture for both NNFSA and DNNFSA which was also used in the ADNNFSA model. The network consists of two layers. The first layer consists of N units (context units) that receive feedback from the next layer (state layer) and M input units that receive input signals. The outputs of this layer is connected to the inputs of the second layer via second-order weight connections. The second layer consists of N PEs. The state PEs are denoted by the vector $\mathbf{s}(t-1)$ and the input units are denoted by the vector $\mathbf{x}(t-1)$. In the second layer, $\mathbf{s}(t)$ is the current-state output vector and $\mathbf{h}(t)$ is the current-state activation vector. The activation of the second layer can be computed as follows:

$$h_j(t) = f\Big(\sum_i \sum_n w_{jin} s_i(t-1) x_n(t-1)\Big), \tag{49}$$

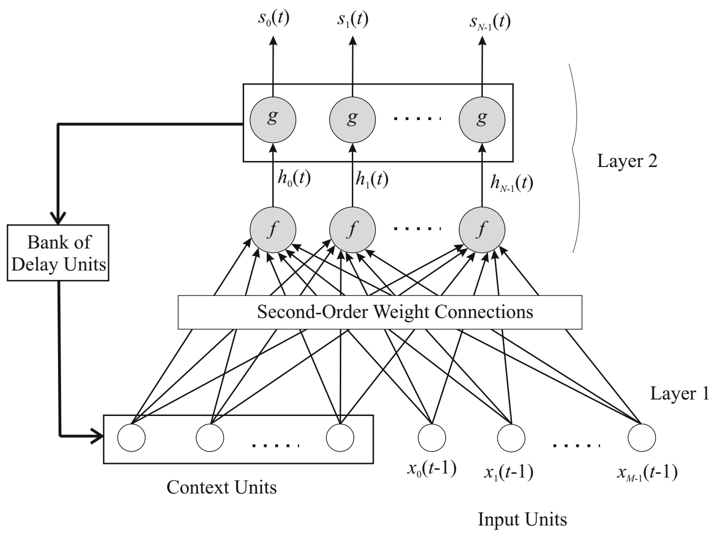


Fig. 23 The general architecture for NNFSA, DNNFSA, and ADNNFSA models

$$s_j(t) = g\big(h_j(t)\big). \tag{50}$$

In the implementation of the NNFSA model, $f(\cdot)$ is a sigmoid function and $g(\cdot)$ is an identity function, while in the implementation of DNNFSA, $g(\cdot)$ is a discrete hard-limiter as follows:

$$g(a) = \begin{cases} 0.8 & \text{if } a > 0.5 \\ 0.2 & \text{if } a < 0.5 \end{cases} \tag{51}$$

The NNFSA model applies the true-gradient descent real-time recurrent learning (RTRL) algorithm, which had a good performance. In the NNFSA model, Giles *et al.* [18] used the analog nature of the network, which does not match with the discrete behavior of a FSA. Therefore, Zeng *et al.* [58], in DNNFSA, discretized the analog NNFSA by using the function in Equation 51. Therefore, all the states are discretized, and the RTRL algorithm is no longer applicable. The pseudo-gradient algorithm was used, and it hinders training because it is an approximation of the true gradient. Therefore, Chen *et al.* [5] used analog internal states at the beginning of the training, and as the training progresses, the model changes gradually to the discrete mode of the internal states. Thus, the current-state activation output $h_j(t)$ is computed same as in Equation 49, and current-state output $s_j(t)$ is computed as follows:

$$s_j(t) = \begin{cases} h_j(t) & \text{if } j \in \text{analog mode} \\ g\big(h_j(t)\big) & \text{if} j \in \text{discrete mode} \end{cases} \tag{52}$$

To decide whether the mode of a state PE has to be switched to the discrete mode, a quantization threshold parameter β is used. If the output of the sate PE j, $s_j(t) < \beta$ or $s_j(t) > 1.0 - \beta$ for all the training strings, the mode of this state PE is switched to the discrete phase. This recurrent network model adapts the training from the initial analog phase, which has a good training performance, to the discrete phase, which fits properly with the nature of the FSA, through the progress of the training for automatic rule extraction.

6.2 Identification and Control of Dynamical Systems

In dynamic systems, different time-variant variables interact to produce outputs. To control such systems, a dynamic model is required to tackle the unpredictable variations in such systems. Adaptive control systems can be used to control dynamic systems since these control systems use a control scheme that have the feature to modify its behavior in response to the variations in dynamic systems. However, most of the adaptive control techniques require the availability of an explicit dynamic structure of the system, and this is impossible for most nonlinear systems which have poorly known dynamics. In addition, these conventional adaptive control techniques lack the ability of learning. This means that such adaptive control systems cannot use the knowledge available from the past and apply it in similar situations in the present or future. Therefore, a class of intelligent control has been applied which is based on neural modeling and learning [27, 46].

Neural networks can deal with nonlinearity, perform parallel computing and process noisy data. These features have made neural networks good tools for controlling nonlinear and time-variant systems. Since dynamic systems involve model states at different time steps, it has been important for the neural network model to memorize the previous states of the system and deal with the feedback signals. This is impossible with the conventional feedforward neural networks. To solve the problem, it has been important to make neural networks have a dynamic behavior by incorporating delay units and feedback links. In other words, RNNs with time delay units and feedback connections can tackle the dynamic behavior of nonlinear time-variant systems. RNNs have proved successfulness in the identification and control of systems which are dynamic in nature. We will review a model that was proposed by Ge *et al.* [14] for the identification and control of nonlinear systems.

Yan and Zhang [56] presented two main characteristics to measure the dynamic memory performance of neural networks. They called them the "depth" and "resolution ratio". Depth refers to how far information can be memorized by the model, while resolution ratio refers to the amount of information of the input to the model that can be retained. Time-delay recurrent neural networks (TDRNN) can retain much information about the input and memorize information of a short time period. Thus, it has a good resolution ratio and poor depth. On the other hand, most recurrent networks such as Elman networks have a good depth and poor resolution ratio [14]. Ge *et al.* [14] proposed a model that has a memory of better depth and

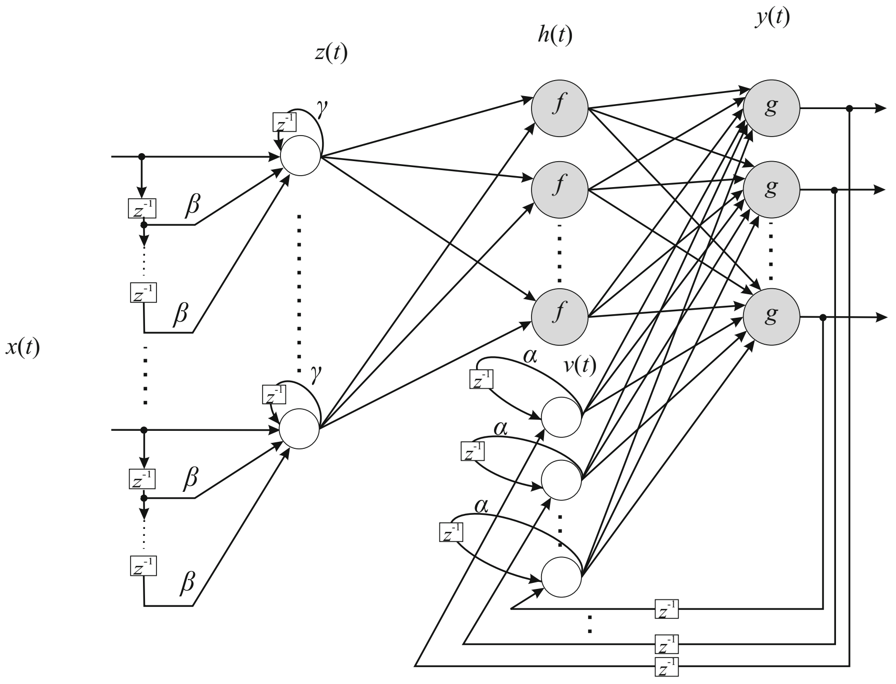


Fig. 24 The architecture of the TDRNN for identification and control of dynamic systems

resolution ratio to identify and control dynamic systems. Figure 24 shows the architecture of the TDRNN proposed by Ge *et al.* [14].

The model incorporates memory units in the input layer with local feedback gain γ $(0 \leq \gamma \leq 1)$ to enhance the resolution ratio. The architecture includes input layer, hidden layer, output layer, and context layer. In this model, the input and context units are different from the traditional recurrent networks since, in this model, the units in the input and context layers are PEs with linear transfer functions. Therefore, in this instance only, we will call them input PEs and context PEs for consistency. The context PEs memorize the activations of the output PEs; in addition, there are local feedback links with constant coefficient α in the context PEs. The output of the context PE can be given as follows:

$$v_j(t) = \alpha v_j(t-1) + y_j(t-1), \quad j = 1,2,\cdots,N \tag{53}$$

where $v_j(t)$ and $y_j(t)$ are the outputs of the j^{th} context PE and the j^{th} output PE respectively, and N is the number of the context PEs and the output PEs.

The mathematical description of the output PEs, hidden PEs, and input PEs are respectively described by the following three equations:

$$y_j(t) = g\Big(\sum_{i=1}^{M} w_{ji}^2 h_i(t) + \sum_{i=1}^{N} w_{ji}^3 v_i(t)\Big), \tag{54}$$

$$h_j(t) = f\Big(\sum_{i=1}^{K} w_{ji}^1 z_i(t)\Big), \tag{55}$$

$$z_j(t) = x_j(t) + \beta \sum_{i=1}^{r} x_j(t-i) + \gamma z_j(t-1), \tag{56}$$

where w^2 is the weight between the hidden and the output PEs, w^3 is the weight between the context and the output PEs, $h_j(t)$ is the output of the j^{th} hidden PE, M is the number of the hidden PEs, w^1 is the weight between the input and hidden PEs, $z_j(t)$ is the output of the j^{th} input PE, K is the number of the input PEs, $x_j(t)$ is the j^{th} external input, r is the number of the unit time delays, and $0 \leq \alpha, \beta, \gamma \leq 1$ and $\beta + \gamma = 1$. The activation function $f(\cdot)$ is a sigmoid function, and the function $g(\cdot)$ is a linear function.

The network was learned by a dynamic recurrent backpropagation learning algorithm which was developed based on the gradient descent method. The model shows good effectiveness in the identification and control of nonlinear systems.

7 Conclusion

In this chapter, we presented an introduction to RNNs. We began by describing the basic paradigm of this extension of MLPs, and the need for networks that can process sequences of varying lengths. Then we classified the architectures used and identified the prevailing models and topologies. Subsequently we tacked the implementation of memory in these systems by describing different approaches for maintaining state in such devices. Then we described the prevailing learning methods and an important limitation that all gradient based approaches to learning in RNNs face. In the next section, we described the relationship between the ability of RNNs to process symbolic input sequences and more familiar computational models that can handle the same types of data. This provided confidence in the model as a general purpose computational tool. The final section presented two sample applications to elucidate the applicability of these networks on real-world problems.

We hope that this chapter has motivated the use of RNNs and whetted the reader's appetite to explore this rich paradigm further. A good place to begin a deeper exploration is [29], which presents the most important results in the field in a collection of mutually consistent papers by the primary researchers in these areas.

References

1. Allen, R.B., Alspector, J.: Learning of stable states in stochastic asymmetric networks. Technical Report TM-ARH-015240, Bell Communications Research, Morristown, NJ (1989)
2. Atiya, A.F.: Learning on a general network. In: Neural Information Processing Systems, New York, pp. 22–30 (1988)

3. Back, A.D., Tsoi, A.C.: FIR and IIR synapses, a new neural network architecture for time series modeling. Neural Computation 3, 375–385 (1991)
4. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gadient descent is difficult. IEEE Transactions on Neural Networks 5, 157–166 (1994)
5. Chen, L., Chua, H., Tan, P.: Grammatical inference using an adaptive recurrent neural network. Neural Processing Letters 8, 211–219 (1998)
6. Chen, S., Billings, S., Grant, P.: Nonlinear system identification using neural networks. International Journal of Control 51(6), 1191–1214 (1990)
7. Cohen, M.A., Grossberg, S.: Stability of global pattern formation and parallel memory storage by competitive neural networks. IEEE Transactions on Systems, Man and Cybernetics 13, 815–826 (1983)
8. Elman, J.L.: Finding structure in time. Cognitive Science 14, 179–211 (1990)
9. Elman, J.L.: Distributed representations, simple recurrent networks and grammatical structure. Machine Learning 7, 195–225 (1991)
10. Fahlman, S.E., Lebiere, C.: The cascade-correlation learning architecture. Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (February 1990)
11. Forcada, M.L., Ñeco, R.P.: Recursive Hetero-Associative Memories for Translation. In: Mira, J., Moreno-Díaz, R., Cabestany, J. (eds.) IWANN 1997. LNCS, vol. 1240, pp. 453–462. Springer, Heidelberg (1997)
12. Frasconi, P., Gori, M., Soda, G.: Local feedback multilayered networks. Neural Computation 4, 120–130 (1992)
13. Galland, C.C., Hinton, G.E.: Deterministic Boltzman learning in networks with asymmetric connectivity. Technical Report CRG-TR-89-6, University of Toronto Department of Computer Science (1989)
14. Ge, H., Du, W., Qian, F., Liang, Y.: Identification and control of nonlinear systems by a time-delay recurrent neural network. Neurocomputing 72, 2857–2864 (2009)
15. Giles, C., Kuhn, G., Williams, R.: Dynamic recurrent neural networks: theory and applications. IEEE Trans. Neural Netw. 5(2), 153–156 (1994)
16. Giles, C.L., Chen, D., Miller, C.B., Chen, H.H., Sun, G.Z., Lee, Y.C.: Second-order recurrent neural networks for grammatical inference. In: 1991 IEEE INNS International Joint Conference on Neural Networks, Seattle, Piscataway, NJ, vol. 2, pp. 271–281. IEEE Press (1991)
17. Giles, C.L., Horne, B.G., Lin, T.: Learning a class of large finite state machines with a recurrent neural network. Neural Networks 8, 1359–1365 (1995)
18. Giles, C.L., Miller, C.B., Chen, D., Chen, H.H., Sun, G.Z., Lee, Y.C.: Learning and extracting finite state automata with second-order recurrent neural networks. Neural Computation 4, 395–405 (1992)
19. Gori, M., Bengio, Y., Mori, R.D.: Bps: A learning algorithm for capturing the dynamic nature of speech. In: International Joint Conference on Neural Networks, vol. II, pp. 417–423 (1989)
20. Harigopal, U., Chen, H.C.: Grammatical inference using higher order recurrent neural networks. In: Proceedings of the Twenty-Fifth Southeastern Symposium on System Theory, SSST 1993, pp. 338–342 (1993)
21. Hinton, T.J., abd Sejnowski, G.E.: Optimal perceptual inference. In: Proceedines of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 448–453. IEEE Computer Society (1983)
22. Hochreiter, S.: Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis (1991)

23. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation 9, 1735–1780 (1997)
24. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
25. Jordan, M.I.: Supervised learning and systems with excess degrees of freedom. Technical Report COINS Technical Report 88–27, Massachusetts Institute of Technology (1988)
26. Karakasoglu, A., Sudharsanan, S., Sundareshan, M.K.: Identification and decentralized adaptive control using dynamic neural networks with application to robotic manipulators. IEEE Trans. Neural Networks 4, 919–930 (1993)
27. Karray, F.O., Silva, C.: Soft Computing and Intelligent Systems Design. Addison Wesley (2004)
28. Kilian, J., Siegelmann, H.T.: On the power of sigmoid neural networks. In: Proceedings of the Sixth ACM Workshop on Computational Learning Theory, pp. 137–143. ACM Press (1993)
29. Kolen, J.F., Kremer, S.C. (eds.): A Field Guide to Dynamical Recurrent Networks. Wiley-IEEE Press (2001)
30. Kuo, J., Celebi, S.: Adaptation of memory depth in the gamma filter. In: Acoustics, Speech and Signal Processing IEEE Conference, pp. 1–4 (1994)
31. Kuroe, Y.: Representation and Identification of Finite State Automata by Recurrent Neural Networks. In: Pal, N.R., Kasabov, N., Mudi, R.K., Pal, S., Parui, S.K. (eds.) ICONIP 2004. LNCS, vol. 3316, pp. 261–268. Springer, Heidelberg (2004)
32. Lippmann, R.P.: An introduction to computing with neural nets. IEEE ASSP Magazine 4, 4–22 (1987)
33. Mozer, M.: A focused background algorithm for temporal pattern recognition. Complex Systems 3 (1989)
34. Mozer, M.C.: Induction of multiscale temporal structure. In: Advances in Neural Information Processing Systems 4, pp. 275–282. Morgan Kaufmann (1992)
35. Nguyen, M., Cottrell, G.: A technique for adapting to speech rate. In: Kamm, C., Kuhn, G., Yoon, B., Chellapa, R., Kung, S. (eds.) Neural Networks for Signal Processing 3. IEEE Press (1993)
36. Omlin, C.W., Giles, C.L.: Constructing deterministic finite-state automata in recurrent neural networks. Journal of the ACM 43(6), 937–972 (1996)
37. Patan, K.: Locally Recurrent Neural Networks. In: Patan, K. (ed.) Artificial. Neural Net. for the Model. & Fault Diagnosis. LNCIS, vol. 377, pp. 29–63. Springer, Heidelberg (2008)
38. Pollack, J.B.: On Connectionist Models of Natural Language Processing. PhD thesis, Computer Science Department of the University of Illinois at Urbana-Champaign, Urbana, Illinois, Available as TR MCCS-87-100, Computing Research Laboratory, New Mexico State University, Las Cruces, NM (1987)
39. Principe, J.C., de Vries, B., de Oliveira, P.G.: The gamma filter - a new class of adaptive IIR filter with restricted feedback. IEEE Transactions on Signal Processing 41, 649–656 (1993)
40. Renals, S., Rohwer, R.: A study of network dynamics. Journal of Statistical Physics 58, 825–848 (1990)
41. Robinson, A.J.: Dynamic Error Propagation Networks. Ph.d., Cambridge University Engineering Department (1989)
42. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: Parallel Distributed Processing. MIT Press, Cambridge (1986)
43. Schmidhuber, J.H.: A fixed size storage $o(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. Neural Computation 4(2), 243–248 (1992)

44. Sejnowski, T.J., Rosenberg, C.R.: Parallel networks that learn to pronounce english text. Complex Syst. I, 145–168 (1987)
45. Shannon, C.E.: Communication in the presence of noise. Proc. Institute of Radio Engineers 37(1), 10–21 (1949); reprinted as classic paper in: Proc. IEEE 86(2) (February 1998)
46. Shearer, J.L., Murphy, A.T., Richardson, H.H.: Introduction to System Dynamics. Addison-Wesley, Reading (1971)
47. Siegelmann, H.T., Sontag, E.D.: Turing computability with neural nets. Applied Mathematics Letters 4(6), 77–80 (1991)
48. Silva, T.O.: Laguerre filters - an introduction. Revista do Detua 1(3) (1995)
49. Smith, J.O.: Delay lines. Physical Audio Signal Processing (2010), `http://ccrma.stanford.edu/ jos/pasp/ Tapped_Delay_Line_TDL.htm` (cited November 28, 2010)
50. Smith, S.W.: The scientist and engineer's guide to digital signal processing. California Technical Publishing (2006), `http://www.dspguide.com/ch15.htm` (cited November 29, 2010)
51. Tsoi, A.C., Back, A.D.: Locally recurrent globally feedforward networks: A critical review of architectures. IEEE Transactions on Neural Networks 5, 229–239 (1994)
52. Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., Lang, L.: Phonemic recognition using time delay neural networks. IEEE Trans. Acoustic Speech and Signal Processing 37(3), 328–339 (1989)
53. Werbos, P.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences. Phd thesis, Harvard University (1974)
54. Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Computation 1, 270–289 (1989)
55. Won, S.H., Song, I., Lee, S.Y., Park, C.H.: Identification of finite state automata with a class of recurrent neural networks. IEEE Transactions on Neural Networks 21(9), 1408–1421 (2010)
56. Yan, P.F., Zhang, C.S.: Artificial Neural Network and Simulated Evolutionary Computation. Thinghua University Press, Beijing (2000)
57. Zamarreno, J.M., Vega, P.: State space neural network. Properties and application. Neural Networks 11, 1099–1112 (1998)
58. Zeng, Z., Goodman, R.M., Smyth, P.: Learning finite state machines with self-clustering recurrent networks. Neural Computation 5(6), 977–990 (1993)
59. Zeng, Z., Goodman, R.M., Smyth, P.: Discrete recurrent neural networks for grammatical inference. IEEE Transactions on Neural Networks 5(2), 320–330 (1994)