A guide to recurrent neural networks and backpropagation

Mikael Bodén*
mikael.boden@ide.hh.se
School of Information Science, Computer and Electrical Engineering
Halmstad University.

November 13, 2001

Abstract

This paper provides guidance to some of the concepts surrounding recurrent neural networks. Contrary to feedforward networks, recurrent networks can be sensitive, and be adapted to past inputs. Backpropagation learning is described for feedforward networks, adapted to suit our (probabilistic) modeling needs, and extended to cover recurrent networks. The aim of this brief paper is to set the scene for applying and understanding recurrent neural networks.

1 Introduction

It is well known that conventional feedforward neural networks can be used to approximate *any* spatially finite function given a (potentially very large) set of hidden nodes. That is, for functions which have a *fixed* input space there is always a way of encoding these functions as neural networks. For a two-layered network, the mapping consists of two steps,

$$y(t) = G(F(x(t))). \tag{1}$$

We can use automatic learning techniques such as backpropagation to find the weights of the network (G and F) if sufficient samples from the function is available.

Recurrent neural networks are fundamentally different from feedforward architectures in the sense that they not only operate on an input space but also on an internal *state* space – a trace of what already has been processed by the network. This is equivalent to an Iterated Function System (IFS; see (Barnsley, 1993) for a general introduction to IFSs; (Kolen, 1994) for a neural network perspective) or a Dynamical System (DS; see e.g. (Devaney, 1989) for a general introduction to dynamical systems; (Tino et al., 1998; Casey, 1996) for neural network perspectives). The state space enables the representation (and learning) of temporally/sequentially extended dependencies over unspecified (and potentially infinite) intervals according to

$$y(t) = G(s(t)) \tag{2}$$

$$s(t) = F(s(t-1), x(t)). \tag{3}$$

*This document was mainly written while the author was at the Department of Computer Science, University of Skövde.

To limit the scope of this paper and simplify mathematical matters we will assume that the network operates in discrete time steps (it is perfectly possible to use continuous time instead). It turns out that if we further assume that weights are at least rational and continuous output functions are used, networks are capable of representing *any* Turing Machine (again assuming that any number of hidden nodes are available). This is important since we then know that all that can be computed, can be processed[1] equally well with a discrete time recurrent neural network. It has even been suggested that if real weights are used (the neural network is completely analog) we get super-Turing Machine capabilities (Siegelmann, 1999).

2 Some basic definitions

To simplify notation we will restrict equations to include two-layered networks, i.e. networks with two layers of nodes excluding the input layer (leaving us with one 'hidden' or 'state' layer, and one 'output' layer). Each layer will have its own index variable: k for output nodes, j (and h) for hidden, and i for input nodes. In a feed forward network, the input vector, $\mathbf{x}$, is propagated through a weight layer, $\mathbf{V}$,

$$y_j(t) = f(net_j(t)) \tag{4}$$

$$net_j(t) = \sum_i^n x_i(t) v_{ji} + \theta_j \tag{5}$$

where n is the number of inputs, θ_j is a bias, and f is an output function (of any differentiable type). A network is shown in Figure 1.

In a simple recurrent network, the input vector is similarly propagated through a weight layer, but also combined with the previous state activation through an additional *recurrent* weight layer, $\mathbf{U}$,

$$y_j(t) = f(net_j(t)) \tag{6}$$

$$net_j(t) = \sum_i^n x_i(t) v_{ji} + \sum_h^m y_h(t-1) u_{jh}) + \theta_j \tag{7}$$

where m is the number of 'state' nodes.

The output of the network is in both cases determined by the state and a set of output weights, $\mathbf{W}$,

$$y_k(t) = g(net_k(t)) \tag{8}$$

$$net_k(t) = \sum_j^m y_j(t) w_{kj} + \theta_k \tag{9}$$

where g is an output function (possibly the same as f).

[1] I am intentionally avoiding the term 'computed'.

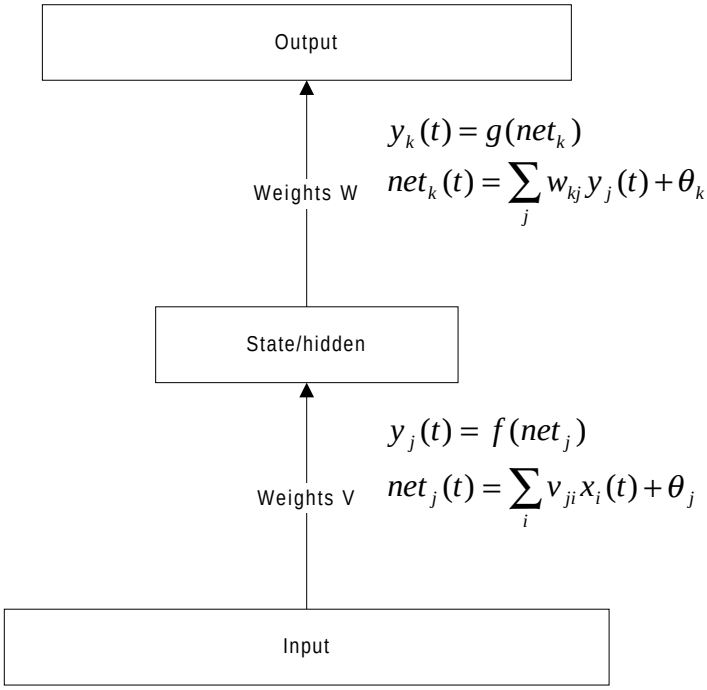

Figure 1: A feedforward network.

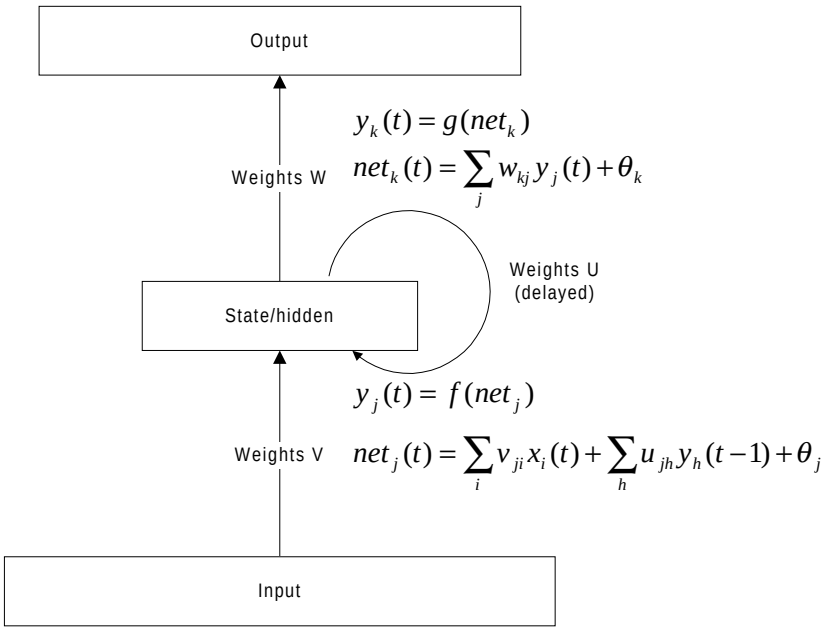

Figure 2: A simple recurrent network.

3 The principle of backpropagation

Any network structure can be trained with backpropagation when desired output patterns exist and each function that has been used to calculate the actual output patterns is differentiable. As with conventional gradient descent (or ascent), backpropagation works by, for each modifiable weight, calculating the gradient of a cost (or error) function with respect to the weight and then adjusting it accordingly.

The most frequently used cost function is the summed squared error (SSE). Each pattern or presentation (from the training set), p, adds to the cost, over all output units, k.

$$C = \frac{1}{2}\sum_{p}^{n}\sum_{k}^{m}(d_{pk} - y_{pk})^2 \tag{10}$$

where d is the desired output, n is the total number of available training samples and m is the total number of output nodes.

According to gradient descent, each weight change in the network should be proportional to the negative gradient of the cost with respect to the specific weight we are interested in modifying.

$$\Delta w = -\eta\frac{\partial C}{\partial w} \tag{11}$$

where η is a learning rate.

The weight change is best understood (using the chain rule) by distinguishing between an error component, $\delta = -\partial C/\partial net$, and $\partial net/\partial w$. Thus, the error for output nodes is

$$\delta_{pk} = -\frac{\partial C}{\partial y_{pk}}\frac{\partial y_{pk}}{\partial net_{pk}} = (d_{pk} - y_{pk})g'(y_{pk}) \tag{12}$$

and for hidden nodes

$$\delta_{pj} = -(\sum_{k}^{m}\frac{\partial C}{\partial y_{pk}}\frac{\partial y_{pk}}{\partial net_{pk}}\frac{\partial net_{pk}}{y_{pj}})\frac{\partial y_{pj}}{\partial net_{pj}} = \sum_{k}^{m}\delta_{pk}w_{kj}f'(y_{pj}). \tag{13}$$

For a first-order polynomial, $\partial net/\partial w$ equals the input activation. The weight change is then simply

$$\Delta w_{kj} = \eta\sum_{p}^{n}\delta_{pk}y_{pj} \tag{14}$$

for output weights, and

$$\Delta v_{ji} = \eta\sum_{p}^{n}\delta_{pj}x_{pi} \tag{15}$$

for input weights. Adding a time subscript, the recurrent weights can be modified according to

$$\Delta u_{jh} = \eta\sum_{p}^{n}\delta_{pj}(t)y_{ph}(t-1). \tag{16}$$

A common choice of output function is the logistic function

$$g(net) = \frac{1}{1+e^{-net}}. \tag{17}$$

The derivative of the logistic function can be written as

$$g'(y) = y(1-y). \tag{18}$$

For obvious reasons most cost functions are 0 when each target equals the actual output of the network. There are, however, more appropriate cost functions than SSE for guiding weight changes during training (Rumelhart et al., 1995). The common assumptions of the ones listed below are that the relationship between the actual and desired output is probabilistic (the network is still deterministic) and has a known distribution of error. This, in turn, puts the interpretation of the output activation of the network on a sound theoretical footing.

If the output of the network is the mean of a *Gaussian* distribution (given by the training set) we can instead minimize

$$C = -\sum_p^n \sum_k^m \frac{(y_{pk} - d_{pk})^2}{2\sigma^2} \tag{19}$$

where σ is assumed to be fixed. This cost function is indeed very similar to SSE.

With a Gaussian distribution (outputs are not explicitly bounded), a natural choice of output function of the output nodes is

$$g(net) = net. \tag{20}$$

The weight change then simply becomes

$$\Delta w_{kj} = \eta \sum_p^n (d_{pk} - y_{pk}) y_{pj}. \tag{21}$$

If a *binomial* distribution is assumed (each output value is a probability that the desired output is 1 or 0, e.g. feature detection), an appropriate cost function is the so-called cross entropy,

$$C = \sum_p^n \sum_k^m d_{pk} \ln y_{pk} + (1 - d_{pk}) \ln(1 - y_{pk}). \tag{22}$$

If outputs are distributed over the range 0 to 1 (as here), the logistic output function is useful (see Equation 17). Again the output weight change is

$$\Delta w_{kj} = \eta \sum_p^n (d_{pk} - y_{pk}) y_{pj}. \tag{23}$$

If the problem is that of "1-of-n" classification, a *multinomial* distribution is appropriate. A suitable cost function is

$$C = \sum_p^n \sum_k^m d_{pk} \ln \frac{e^{net_k}}{\sum_q e^{net_q}} \tag{24}$$

where q is yet another index of all output nodes. If the right output function is selected, the so-called softmax function,

$$g(net_k) = \frac{e^{net_k}}{\sum_q e^{net_q}}, \tag{25}$$

the now familiar update rule follows automatically,

$$\Delta w_{kj} = \eta \sum_p^n (d_{pk} - y_{pk}) y_{pj}. \tag{26}$$

As shown in (Rumelhart et al., 1995) this result occurs whenever we choose a probability function from the exponential family of probability distributions.

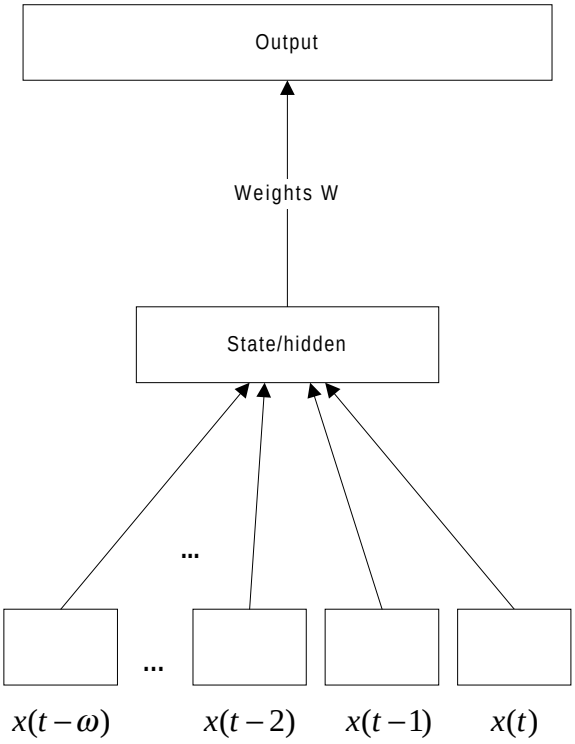

Figure 3: A "tapped delay line" feedforward network.

4 Tapped delay line memory

The perhaps easiest way to incorporate temporal or sequential information into a training situation is to make the temporal domain spatial and use a feedforward architecture. Information available back in time is inserted by widening the input space according to a fixed and pre-determined "window" size, $\mathbf{X} = \mathbf{x}(\mathbf{t}), \mathbf{x}(\mathbf{t}-\mathbf{1}), \mathbf{x}(\mathbf{t}-\mathbf{2}), ..., \mathbf{x}(\mathbf{t}-\omega)$ (see Figure 3). This is often called a tapped delay line since inputs are put in a delayed buffer and discretely shifted as time passes.

It is also possible to manually extend this approach by selecting certain intervals "back in time" over which one uses an average or other pre-processed features as inputs which may reflect the signal decay.

The classical example of this approach is the NETtalk system (Sejnowski and Rosenberg, 1987) which learns from example to pronounce English words displayed in text at the input. The network accepts seven letters at a time of which only the middle one is pronounced.

Disadvantages include that the user has to select the maximum number of time steps which is useful to the network. Moreover, the use of independent weights for processing the same components but in different time steps, harms generalization. In addition, the large number of weights requires a larger set of examples to avoid over-specialization.

5 Simple recurrent network

A strict feedforward architecture does not maintain a short-term memory. Any memory effects are due to the way past inputs are re-presented to the network (as for the tapped delay line).

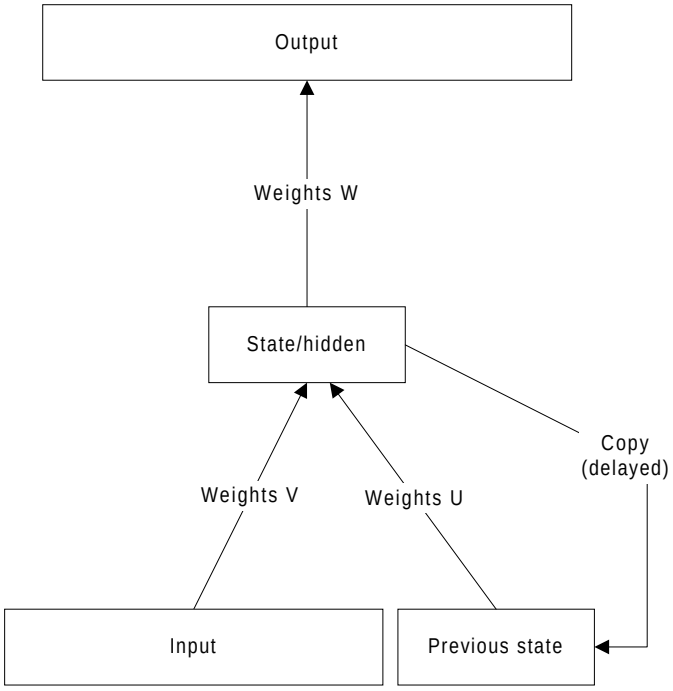

Figure 4: A simple recurrent network.

A simple recurrent network (SRN; (Elman, 1990)) has activation feedback which embodies short-term memory. A state layer is updated not only with the external input of the network but also with activation from the previous forward propagation. The feedback is modified by a set of weights as to enable automatic adaptation through learning (e.g. backpropagation).

5.1 Learning in SRNs: Backpropagation through time

In the original experiments presented by Jeff Elman (Elman, 1990) so-called truncated backpropagation was used. This basically means that $y_j(t-1)$ was simply regarded as an additional input. Any error at the state layer, $\delta_j(t)$, was used to modify weights from this additional input slot (see Figure 4).

Errors can be backpropagated even further. This is called backpropagation through time (BPTT; (Rumelhart et al., 1986)) and is a simple extension of what we have seen so far. The basic principle of BPTT is that of "unfolding." All recurrent weights can be duplicated spatially for an arbitrary number of time steps, here referred to as τ. Consequently, each node which sends activation (either directly or indirectly) along a recurrent connection has (at least) τ number of copies as well (see Figure 5).

In accordance with Equation 13, errors are thus backpropagated according to

$$\delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(y_{pj}(t-1)) \tag{27}$$

where h is the index for the activation receiving node and j for the sending node (one time step back). This allows us to calculate the error as assessed at time t, for node outputs (at the state or input layer) calculated on the basis of an arbitrary number of previous presentations.

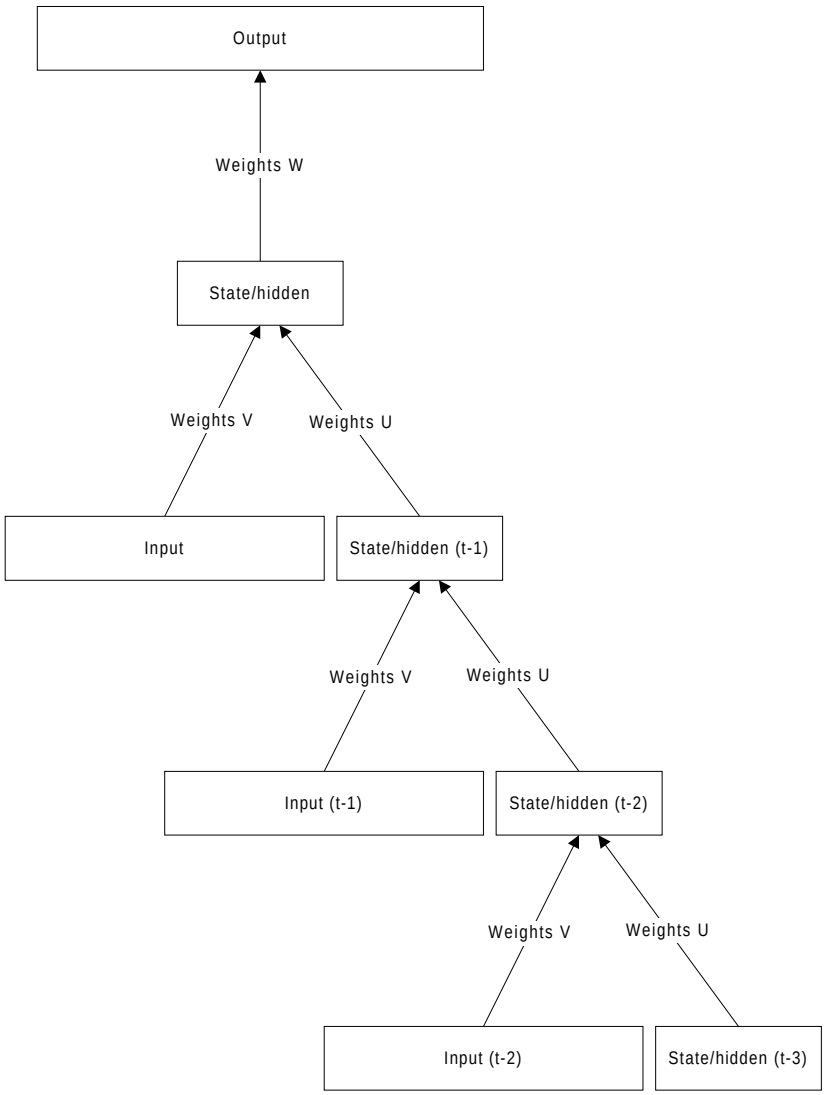

Figure 5: The effect of unfolding a network for BPTT ($\tau = 3$).

It is important to note, however, that after error deltas have been calculated, weights are folded back adding up to one big change for each weight. Obviously there is a greater memory requirement (both past errors and activations need to be stored away), the larger τ we choose.

In practice, a large τ is quite useless due to a "vanishing gradient effect" (see e.g. (Bengio et al., 1994)). For each layer the error is backpropagated through the error gets smaller and smaller until it diminishes completely. Some have also pointed out that the instability caused by possibly ambiguous deltas (e.g. (Pollack, 1991)) may disrupt convergence. An opposing result has been put forward for certain learning tasks (Bodén et al., 1999).

6 Discussion

There are many variations of the architectures and learning rules that have been discussed (e.g. so-called Jordan networks (Jordan, 1986), and fully recurrent networks, Real-time recurrent learning (Williams and Zipser, 1989) etc). Recurrent networks share, however, the property of being able to internally use and create states reflecting temporal (or even structural) dependencies. For simpler tasks (e.g. learning grammars generated by small finite-state machines) the organization of the state space straightforwardly reflects the component parts of the training data (e.g. (Elman, 1990; Cleeremans et al., 1989)). The state space is, in most cases, real-valued. This means that subtleties beyond the component parts, e.g. statistical regularities may influence the organization of the state space (e.g. (Elman, 1993; Rohde and Plaut, 1999)). For more difficult tasks (e.g. where a longer trace of memory is needed, and context-dependence is apparent) the highly non-linear, continous space offers novel kinds of dynamics (e.g. (Rodriguez et al., 1999; Bodén and Wiles, 2000)). These are intriguing research topics but beyond the scope of this introductory paper. Analyses of learned internal representations and processes/dynamics are crucial for our understanding of what and how these networks process. Methods of analysis include hierarchical cluster analysis (HCA), and eigenvalue and eigenvector characterizations (of which Principal Components Analysis is one).

References

Barnsley, M. (1993). *Fractals Everywhere*. Academic Press, Boston, 2nd edition.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bodén, M. and Wiles, J. (2000). Context-free and context-sensitive dynamics in recurrent neural networks. *Connection Science*, 12(3).

Bodén, M., Wiles, J., Tonkes, B., and Blair, A. (1999). Learning to predict a context-free language: Analysis of dynamics in recurrent hidden units. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 359–364, Edinburgh. IEE.

Casey, M. (1996). The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, 8(6):1135–1178.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381.

Devaney, R. L. (1989). *An Introduction to Chaotic Dynamical Systems*. Addison-Wesley.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition*, 48:71–99.

Giles, C. L., Miller, C. B., Chen, D., Chen, H. H., Sun, G. Z., and Lee, Y. C. (1992). Learning and extracted finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405.

Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Conference of the Cognitice Science Society.*

Kolen, J. F. (1994). Fool's gold: Extracting finite state machines from recurrent network dynamics. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems*, volume 6, pages 501–508. Morgan Kaufmann Publishers, Inc.

Pollack, J. B. (1991). The induction of dynamical recognizers. *Machine Learning*, 7:227.

Rodriguez, P., Wiles, J., and Elman, J. L. (1999). A recurrent neural network that learns to count. *Connection Science*, 11(1):5–40.

Rohde, D. L. T. and Plaut, D. C. (1999). Language acquisition in the absence of explicit negative evidence: How important is starting small? *Cognition*, 72:67–109.

Rumelhart, D. E., Durbin, R., Golden, R., and Chauvin, Y. (1995). Backpropagation: The basic theory. In Chauvin, Y. and Rumelhart, D. E., editors, *Backpropagation: Theory, architectures, and applications*, pages 1–34. Lawrence Erlbaum, Hillsdale, New Jersey.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by back-propagating errors. *Nature*, 323:533–536.

Sejnowski, T. and Rosenberg, C. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168.

Siegelmann, H. T. (1999). *Neural Networks and Analog Computation: Beyond the Turing Limit.* Birkhäuser.

Tino, P., Horne, B. G., Giles, C. L., and Collingwood, P. C. (1998). Finite state machines and recurrent neural networks – automata and dynamical systems approaches. In Dayhoff, J. and Omidvar, O., editors, *Neural Networks and Pattern Recognition*, pages 171–220. Academic Press.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280.