Forced Simple Recurrent Neural Networks and Grammatical Inference

Arun Maskara
New Jersey Institute of Technology
Department of Computer and Information Sciences
University Heights, Newark, NJ 07102
arun@hertz.njit.edu

Andrew Noetzel
The William Paterson College
Department of Computer Science
Wayne, NJ 07470

Abstract

A simple recurrent neural network (SRN) introduced by Elman [1990] can be trained to infer a regular grammar from the positive examples of symbol sequences generated by the grammar. The network is trained, through the back-propagation of error, to predict the next symbol in each sequence, as the symbols are presented successively as inputs to the network. The modes of prediction failure of the SRN architecture are investigated. The SRN's internal encoding of the context (the previous symbols of the sequence) is found to be insufficiently developed when a particular aspect of context is not required for the immediate prediction at some point in the input sequence, but is required later. It is shown that this mode of failure can be avoided by using the auto-associative recurrent network (AARN). The AARN architecture contains additional output units, which are trained to show the current input and the current context.

The effect of the size of the training set for grammatical inference is also considered. The SRN has been shown to be effective when trained on an infinite (very large) set of positive examples [Servan-Schreiber *et al.*, 1991]. When a finite (small) set of positive training data is used, the SRN architectures demonstrate a lack of generalization capability. This problem is solved through a new training algorithm that uses both positive and negative examples of the sequences. Simulation results show that when there is restriction on the number of nodes in the hidden layers, the AARN succeeds in the cases where the SRN fails.

Introduction

The problem of inferring a regular grammar from examples has often been studied. An overview of traditional algorithms can be found in [Angluin and Smith, 1983]. Following the development of the back-propagation algorithm [Rumelhart and McClelland, 1986], various recurrent neural architectures using back-propagation have been shown to have some capability for grammatical inference when trained from examples [Servan-Schreiber *et al.*, 1991, Pollack, 1991, Giles *et al.*, 1990].

The idea of training recurrent neural networks with back-propagation was first introduced by Jordan [1986]. In the *recurrent* neural network, the symbols of a sequence are presented sequentially as network inputs. Also, the output of a higher level layer during one time interval is fed back as an input to a lower level layer at the next interval.

Two training schemes are associated with two different paradigms for grammatical inference. In the *classification* paradigm, it is assumed that the desired output is not known until the end of the sequence. Training this case requires back-propagation of error in time [Rumelhart and McClelland, 1986]. It is hard to train a recurrent neural network using this paradigm since the network makes many decisions before it is influenced by the results of those decisions. The classification paradigm has been studied by Giles *et al.* [1990] and Pollack [1991].

The other paradigm is the *predictive* paradigm. In this case, it is assumed that the desired output at each time interval is known. To use this paradigm in the problem of grammatical inference, the network is trained to predict the next input in any sequence. An on-line training algorithm can be used to back-propagate error at the end of each interval. The simple recurrent network (SRN) introduced by Elman [1990] has been shown to behave as a finite state automata (FSA) when trained in the predictive paradigm [Servan-Schreiber *et al.*, 1991].

The current work is based on the the predictive paradigm. It is shown that when the SRN predicts incorrectly it is because it has failed to encode the aspects of the context (the previous symbols of the sequence) that are necessary to predict the next input. We show that this problem can be solved by modifying the SRN to include features of an auto-associative network. The result is the auto-associative recurrent network

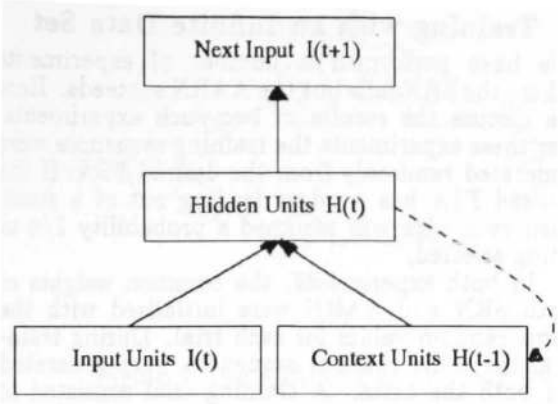

Figure 1: Simple Recurrent Network

(AARN). When trained to predict the next input, the AARN generally performs better than the SRN.

However, when only a finite set of positive examples are used for training, all networks of the SRN type exhibit a limited ability to generalize. Their generalization capability is improved when a finite set of positive and negative examples are used for training.

The Simple Recurrent Network

A diagram of the simple recurrent network (SRN) introduced by Elman [1990] is shown in Figure 1. In this model, the state of the hidden units is copied into the context unit at the end of each interval in the temporal sequence. At time t the context unit holds the state of the hidden units at $t-1$. And at t, the network is trained to predict the input at $t+1$. In order to produce an output corresponding to the next symbol of any sequence, the SRN creates an internal representation, or *encoding*, of the previous symbols of the sequence. The symbols preceding any input symbol will be called the *context* for that input.

The input and output layers use a local representation for the symbols of the sequence: each symbol is represented by a single node. The back-propagation algorithm [Rumelhart and McClelland, 1986] is used to train the network from the positive examples of the sequences. For a particular context, a positive error is back-propagated for the next symbol, and a negative error is back-propagated for all the other symbols. After completion of training, when a symbol sequence is presented at the input, the the network generates output node values that represent likelihoods for the next input symbol.

If a particular context is always followed by one particular symbol, then in that context a positive error is back-propagated only for that symbol, and a negative error for all other symbols. In an ideal case the network should learn to generate a high likelihood (1.0) for that symbol and low likelihood (0.0) for all other symbols. When different instances of a particular context are followed by different symbols, then during training the successor symbols will sometimes back-propagate a positive error and at other times a negative error. The number of times each error is back-propagated will depend on the frequency of the symbol. Therefore the value of likelihood shown by a symbol in a context will depend on the frequency of occurrence of the symbol in that particular context. If in the training data p different symbols follow a particular context with equal frequency, then after training each of those p symbol should ideally generate a likelihood of $1/p$.

The network is said to *accept* a sequence if each symbol in the sequence is predicted; and a symbol is said to be predicted if its associated output unit shows a value of greater than the positive threshold τ on the preceding symbol. The value of τ depends on the number of different symbols which can follow a context. If at most two symbols can follow a context then ideally a threshold of 0.5 should be used. Since the momentum is used in training, in case of two successor symbols a nominal value of 0.3 is used for the threshold τ [Servan-Schreiber *et al.*, 1991]. The sequence is said to be *rejected* if the network is unable to predict any symbol.

The process of encoding context in the hidden layer can be conceptualized in the following way. Early in training, each symbol forms its own distinct code in the hidden layer. Since the output of the hidden layer at $t-1$ is used as the context input at t, the encoded context gradually begins to show a representation of the previous symbol. Then the hidden layer can begin to encode combinations of the previous and current symbols. With this combination as the context, the hidden layer begins to encode the relevant aspects of three consecutive symbols. Eventually, it encodes the relevant aspects of the entire sequence.

The Auto-Associative Recurrent Network

Sometimes the SRN is unable to encode the context required to predict the next input. During training, at time t the network encodes in the hidden layer the information necessary to predict the next input. The context at t is the output of hidden layer at $t-1$. If a particular aspect of any context is not encoded at t, it will not be propagated to times greater than t. The network fails to predict properly if at time t a particular aspect of the context is not encoded (since it is not relevant to predict the input at $t+1$), but that aspect of context is required for prediction at a later time.

An auto-associative network is one in which the output is trained to be the same as (or similar to) the input. Adding the auto-associative feature to the SRN results in auto-associative recurrent network (AARN). The AARN succeeds in the cases where the SRN fails. The AARN has output units that show the current context and the current input, as well as the predicted next input. A diagram of the AARN is shown in Figure 2. At the beginning of interval t, the previous activation pattern in the hidden units (the encoded

context) is copied into the context unit. The network is trained through back-propagation to show the current input and the current context that is active in the context unit, as well as to predict the next input.

The idea of auto-association in recurrent networks was initially used by Pollack [1990], in a model called the recursive auto-associative memory (RAAM). Gharamani and Allen [1991] showed that the AARN performs better than the SRN for the XOR problem. The AARN will encode aspects of the context that are not required to predict the next symbol, but are required at a later time. The encoding of the context is more efficient in the AARN because the hidden layer is forced to represent simultaneously the past (the current context) and the present (the current input).

The formation of internal code in the AARN begins same as in the SRN. Each symbol initially becomes associated with a single internal representation. But in the SRN, two symbols that result in a common prediction could form similar internal representations. And the AARN architecture guarantees that each symbol will have a unique representation, since the hidden layer must learn to represent the current symbol. In the SRN, only the context necessary to predict the next input is gradually encoded. But since the AARN forces the hidden layer to show the previous context, the entire sequence is always gradually encoded.

We have simulated the training of the SRN from examples in which two different contexts must have different hidden layer activations, and yet must make the same prediction. In terms of finite state automata (FSA), this is the same as the requirement that the minimal FSA has two different states that have same set of successor symbols. Let u be the number of states in the minimal FSA from which the training sequences were generated. (We will call the FSA used to generate the training set the 'desired' FSA.) For many examples of FSA, the SRN with $O(\lceil \log_2 u \rceil)$ units in the hidden layer was unable to correctly encode the desired FSA. An AARN with $O(\lceil \log_2 u \rceil)$ units succeeded in the cases where the SRN failed.

Training with an Infinite Data Set

We have performed a number of experiments where the SRN fails but the AARN succeeds. Here we discuss the results of two such experiments. For these experiments the training sequences were generated randomly from the desired FSA. If the desired FSA has n edges leading out of a state, then each edge was assigned a probability $1/n$ of being selected.

In both experiments, the common weights of both SRN and AARN were initialized with the same random values for each trial. During training, the same random sequences were generated for both the cases. A training trial consisted of 60,000 randomly generated sequences. We performed 15 different training runs and results were evaluated at the end of each run. After completion of the training the performance of the network was evaluated with threshold $\tau = 0.3$ (since each state of the FSA has at the most two possible successors).

The performance of the network was evaluated by two types of randomly generated sequences. The first type are those sequences that were randomly generated from the FSA. These are positive examples of the language of the FSA. We will call these *random positive sequences*. The second type are random sequences of valid symbols. Each was started with the symbol S, and then valid symbols other than S were picked randomly until the symbol E was encountered. The special symbols S and E are used to indicate the beginning and the end of each sequence. We will call these *random sequences*. Most of the random sequences are the negative examples of the language of the desired FSA.

For the experiments discussed next, the performance was evaluated on 60,000 sequences: 10,000 are random positive sequences and 50,000 are random sequences. The network was declared a failure if it was unable to correctly classify even one of the 60,000 test sequences.

The first experiment consisted of training the networks to recognize the sequences generated by the FSA shown in Figure 3. In this FSA, state 7 is the final state. For each of the 15 runs the SRN

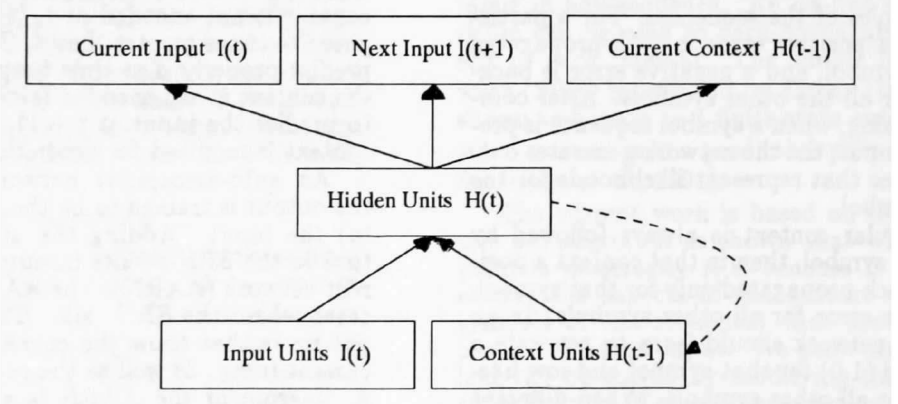

Figure 2: Auto-Associative Recurrent Network

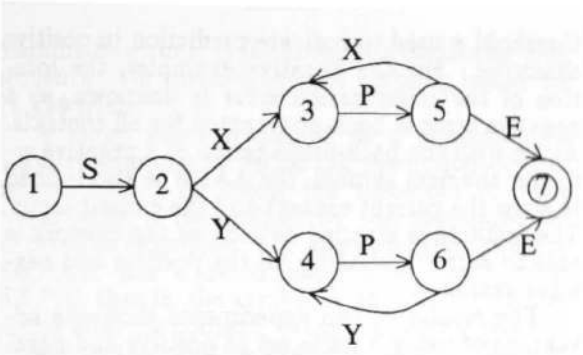

Figure 3: The FSA for Experiment 1

with three units in the hidden layer was never able to correctly classify the entire test set. By examining the hidden layer encodings, we found that at the end of the each training run the SRN had generated similar encodings for the contexts SX and SY.

The AARN with three units in the hidden layer was able to correctly encode the desired FSA in 9 out of 15 runs. After investigating, it was found that in each of the six failure, the context SX and SY had different encoding, the reason of failure seems to be lack of generalization.

The second experiment consisted of training the networks to recognize the sequences generated by the FSA shown in Figure 4. In this FSA, state 10 is the final state. An SRN with four units in the hidden layer was trained in on the language of the FSA in 15 training runs. At the end of each training run, the SRN correctly predicted the next symbols of all the random positive sequences, but failed for few of the random negative sequences. By examining the hidden layer code of the SRN it was found that the network had similar hidden layer representations for the contexts $SPXP$ and $SPPXP$.

The failures occurred because the number of P's preceding X is not required to predict the next symbol. This aspect of the context was therefore not encoded. After the context $SPXP$ and $SPPXP$, the SRN predicted a value above the threshold τ for the symbols X and P. In other words, the SRN learned to recognize the sequences of the regular expression $((P|PP)X)^+$, instead of the expression $((PX)^+|(PPX)^+)$.

The AARN with four units in the hidden layer was able to correctly encode the desired FSA in 10 out of 15 runs. Again the reason of failures in five runs seems to be lack of generalization.

In the related set of experiments we have run some simulations to show the necessity of forcing SRN to show both the current input and the current context. The SRN which was only forced to show the current input succeed for the FSA used shown in Figure 3, where the SRN always failed. The SRN and the SRN with current input failed for the FSA shown in Figure 4, but the SRN with current context succeeded in this case. When a FSA that combines the FSA's shown in Figures 3 and 4 was used, the SRN, the SRN with only current input, and the SRN with only current context were unable to encode that FSA. However, the AARN was able to succeed in that case. More detail of the experiment can be found in [Maskara and Noetzel, 1992].

Training with a Finite Data Set

In the problem of grammatical inference, we cannot assume that the FSA is given to us and the sequences are generated randomly from the FSA. Rather, a small set of training data is provided, and the problem is to find the rules (FSA) which will accept the entire class of language implied by the training data. Depending on the application and the inferring algorithm the training could be carried out by a finite set of positive examples, or a finite set of positive and negative examples.

In the next section we will show that when a finite set of positive data is used for training, the SRN encounters some additional problems. However, we show that these problems can be solved by using positive and negative training examples.

Using Only Positive Examples

After training, the SRN learns to predict the likelihood of the next symbol for each context. If the training sequence is randomly generated from the desired FSA, and each successive symbol is picked up with equal probability, then the SRN learns to generate equal likelihood values for each possible next symbol.

If a finite set of positive data is used for train-

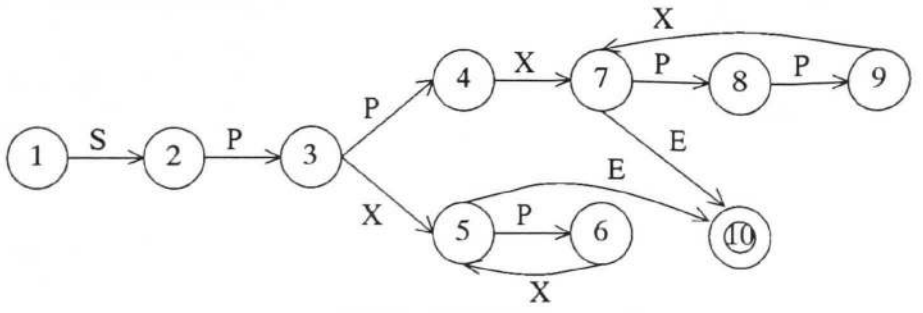

Figure 4: The FSA for Experiment 2

ing, then the value of the symbol predicted by the SRN will depend on the frequency of the occurrence of the symbol in the particular context with in the training set. For example, suppose in a data set with five sequences, the context SX is followed by E in one sequence, and by P in the rest. In this case, after training the SRN should learn to predict E with a value of 0.2, and P with 0.8. This creates a serious problem in determining the value of threshold τ, which is important in deciding whether a sequence is accepted or rejected.

To illustrate this problem, the SRN was trained to learn the regular expression XP^*, from the following six sequences: SXE, $SXPE$, $SXPPE$, $SXPPPE$, $SXPPPPE$, and $SXPPPPPE$. After 5,000 training trials for the context SX, the SRN learned to predict E with a value of 0.19, and P with value 0.82. As the training continued the problem worsened. After 20,000 trails, for the context SX it predicted E with a value 0.15, and X with a value 0.83. The situation did not improve with further training.

The AARN, when trained with the finite set of positive examples, behaved in a similar way. We have found that the problem of dependence on the frequencies of cases in the training set can be avoided by using both positive and negative examples.

Using Positive and Negative Examples

We consider a training algorithm that uses both positive and negative examples. This has two main advantages. The first is that the threshold is not dependent on the frequency of the symbols, so a predetermined value can be used. The second advantage is that the training can be halted as soon as the network is able to correctly classify all the given examples. This will help the network to avoid overgeneralization. If the training is continued after the correct classification of the examples, the network continue to encode the entirety of each sequence in the training set, which will decrease its capability for generalization.

In the training algorithm, the error is back-propagated only if the example is incorrectly classified. A positive example is correctly classified if each context results in the correct prediction of the next symbol. That is, for all contexts, the output for next symbol has a value greater than the positive threshold τ. If a positive sequence is incorrectly classified, a positive error is back-propagated for the instances of the context for which the next symbol prediction failed. If the AARN is used, it is also trained to show the current input and the current context throughout the sequence. This will allow the context to be propagated even when the next symbol is correctly predicted.

A negative example is correctly classified if at some point in the sequence the network does not predict a following symbol. A prediction failure in a negative example is indicated by an output cell value that is less than a threshold (called the negative threshold ρ) that is generally less than the threshold τ used to indicate prediction in positive examples. For the negative examples, the location of the classification error is unknown, so a negative error is back-propagated for all contexts. Along with the back-propagation of a negative error for the next symbol, the AARN is also trained to show the current context and the current input. The training is stopped as soon as the network is able to correctly classify all the positive and negative examples.

The results of two experiments show the advantage of using a finite set of positive and negative examples. During training, a positive threshold of $\tau = 0.7$ and a negative threshold of $\rho = 0.3$ was used. After training, a threshold equal to 0.5 was used to check the validity of the sequences. The number of sequences used to measure the performance will be mentioned in the results.

In the first experiment, the SRN was trained to recognize the sequences generated by the regular expression XP^*. These four positive examples were again used: SXE, $SXPE$, $SXPPE$, $SXPPPPE$. In addition, five negative examples were used: SE, SPE, $SXXE$, $SXPPPXE$, $SXPXE$. The SRN was trained by picking a sequence randomly, and back-propagating the error if the sequence was incorrectly classified. This process was repeated until all the nine sequences were correctly classified. After the completion of the training, the performance was evaluated by using 100 random positive sequences, and 1,000 random sequences.

A network was said to generalize correctly, if it correctly classified all of the randomly generated sequences. A SRN with two hidden units was trained for 15 different runs. For each of the runs, the SRN correctly classified all of the randomly generated sequences. The average number of training trials required were 1,800. For the same experiment the AARN succeeded with an average of 1,427 training trials.

The second experiment was trained the networks to recognize the sequences generated from the automata shown in Figure 3. The following positive and negative examples were used.

Positive Examples: $SXPE$, $SXPXPE$, $SXPXPXPE$, $SYPE$, $SYPYPE$, $SYPYPYPE$.
Negative Examples: SE, SP, SXX, SXY, SXE, $SXPY$, $SXPP$, $SXPXX$, $SXPXY$, $SXPXE$, SYX, SYY, SYE, $SYPX$, $SYPP$, $SYPYX$, $SYPYY$, $SYPYE$.

The SRN with three hidden units was trained for 60,000 random presentation (trials) of the above sequences. After 15 different training runs, the SRN never correctly classified all of the training sequences. An AARN with three hidden units was trained for 15 different runs. After each the AARN correctly classified all the training sequences. To check the performance of the network, 10,000 random positive sequences, and 50,000 random sequences were generated. The AARN was able to correctly classify all the random generated sequences in 11 out of 15 runs. In each of the re-

maining four runs, it failed for one particular long sequence. On the average the training lasted for 12,424 trials.

A *cutoff* point is defined as a position in a negative example at which the prediction of next symbol should be below the threshold ρ. The cutoff point is determined by the associated set of positive examples. For example, the sequence $SXPXX$ has a cutoff point after the context $SXPX$, that is, the symbol X should not be predicted after the context. Since the context $SXPX$ is used in the positive example, each of the next symbol should have a value greater than τ. The only symbol which can have a value less than ρ is the symbol X after context $SXPX$.

Each negative examples in the experiment described above has exactly one cutoff point. But if the examples with more than one cutoff points are used in the training, the generalization will deteriorate. For example, the sequence $SXXPE$ has three cutoff point, the symbol X after the context SX, the symbol P after the context SXX, and the symbol E after the context $SXXP$. During training, the network is required to learn just one of the cutoff point, since the output value below the threshold ρ for any of the next symbol will be taken as correct negative classification. During training, the algorithm can pick any cutoff point, but the maximal generalization will be done in the case when the symbol X after the context SX is picked as the cutoff point. We are still working to develop a training algorithm for an SRN type network which will always find the optimal cutoff point such that the generalization is at its maximum.

Related Work

Pollack [1991] and Giles *et al.* [1990] have developed neural network architectures that learn from positive and negative examples. They used a classification paradigm in which the error is backpropagated at the end of each example. In the classification paradigm, a network does not have a trap state. An error followed by a long sequence of correct examples will be accepted by such a network [Pollack, 1991]. However, the predictive paradigm has a trap state, so that the network stops accepting the input as soon an error is detected.

Conclusions

The AARN architecture can be used in any application where the SRN has been used. We show that there are cases for which the SRN fails to encode the FSA, but the AARN with same number of units in the hidden layer succeeds. The training algorithm used by Servan-Schreiber *et al.* [1991], is suitable for training the network from randomly generated sequences from the FSA. For most applications, the FSA is not known in advance. Hence only a finite set training data may be available. We show that when a finite set of positive examples is used, the network will not perform well if a predetermined threshold value is used for classification. This problem can be solved by using a finite set of positive and negative examples. The results of using both positive and negative examples are encouraging, but the performance of the network deteriorates if the negative examples have more than one cutoff points. Work is in progress to remove this deficiency.

Acknowledgement

The author Arun Maskara is a Ph.D. Candidate at the Polytechnic University, Brooklyn, New York. This work has been done as part of his Ph.D. dissertation.

References

[Angluin and Smith, 1983] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *Computing Survey*, 15(3):237–269, 1983.

[Elman, 1990] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[Ghahramani and Allen, 1991] Zoubin Ghahramani and Robert B. Allen. Temporal processing with connectionist networks. In *Proceedings of the International Joint Conference on Neural Networks*, pages 541–546. Lawrence Erlbaum, 1991.

[Giles *et al.*, 1990] C. Lee Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, and D. Chen. Higher order recurrent networks & grammatical inference. In *Advance in Neural Information Processing Systems 2*, pages 380–387. Morgan Kaufmann, 1990.

[Jordan, 1986] Michael I. Jordan. Attractors dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the 8th Annual Conference of the Cognitive Science Society*, 1986.

[Maskara and Noetzel, 1992] Arun Maskara and Andrew Noetzel. Forced learning in simple recurrent neural networks. In *Proceedings of the Fifth Conference on Neural Networks and Parallel Distributed processing*. Indiana-Purdue University, Fort Wayne, Indiana, 1992.

[Pollack, 1990] Jordan B. Pollack. Recursive distributed representation. *Artificial Intelligence*, 46:77–105, 1990.

[Pollack, 1991] Jordan B. Pollack. The induction of dynamical recognizer. *Machine Learning*, 7(2/3):227–252, 1991.

[Rumelhart and McClelland, 1986] David E. Rumelhart and James L. McClelland, editors. *Parallel Distributed Processing*. MIT press, 1986.

[Servan-Schreiber *et al.*, 1991] David Servan-Schreiber, Axel Cleeremans, and James L. McClelland. Graded state machine: The representation of temporal contingencies in simple recurrent networks. *Machine Learning*, 7(2/3):161–193, 1991.