

## ABSTRACT

Title of dissertation:      **EXTRACTING SYMBOLIC  
REPRESENTATIONS LEARNED BY  
NEURAL NETWORKS**

Thuan Q. Huynh, Doctor of Philosophy, 2012

Dissertation directed by:   **Professor James A. Reggia  
Department of Computer Science**

Understanding what neural networks learn from training data is of great interest in data mining, data analysis, and critical applications, and in evaluating neural network models. Unfortunately, the product of neural network training is typically opaque matrices of floating point numbers that are not obviously understandable. This difficulty has inspired substantial past research on how to extract symbolic, human-readable representations from a trained neural network, but the results obtained so far are very limited (e.g., large rule sets produced). This problem occurs in part due to the distributed hidden layer representation created during learning. Most past symbolic knowledge extraction algorithms have focused on progressively more sophisticated ways to cluster this distributed representation. In contrast, in this dissertation, I take a different approach. I develop ways to alter the error backpropagation neural network training process itself so that it creates a representation of what has been learned in the hidden layer activation space that is more amenable to existing symbolic representation extraction methods.

In this context, this dissertation research makes four main contributions. First,

modifications to the backpropagation learning procedure are derived mathematically, and it is shown that these modifications can be accomplished as local computations. Second, the effectiveness of the modified learning procedure for feedforward networks is established by showing that, on a set of benchmark tasks, it produces rule sets that are substantially simpler than those produced by standard backpropagation learning. Third, this approach is extended to simple recurrent networks, and experimental evaluation shows remarkable reduction in the sizes of the finite state machines extracted from the recurrent networks trained using this approach. Finally, this method is further modified to work on echo state networks, and computational experiments again show significant improvement in finite state machine extraction from these networks. These results clearly establish that principled modification of error backpropagation so that it constructs a better separated hidden layer representation is an effective way to improve contemporary symbolic extraction methods.

EXTRACTING SYMBOLIC  
REPRESENTATIONS LEARNED BY  
NEURAL NETWORKS

by

Thuan Quang Huynh

Dissertation submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
2012

Advisory Committee:

Professor James A. Reggia, Chair/Advisor

Associate Professor Lise C. Getoor, Committee Member

Professor David W. Jacobs, Committee Member

Professor Dana S. Nau, Committee Member

Professor Juan Uriagereka, Dean's Representative

© Copyright by  
Thuan Quang Huynh  
2012

# Table of Contents

List of Tables	iv
List of Figures	v
1 Introduction	1
1.1 Goals and Specific Aims . . . . .	3
1.2 Overview . . . . .	6
2 Background	8
2.1 Multilayer Feedforward Neural Networks . . . . .	10
2.2 Simple Recurrent Neural Networks . . . . .	14
2.3 Echo State Networks . . . . .	20
2.4 Symbolic Representation Extraction . . . . .	23
2.4.1 Rule Extraction from Feedforward Neural Networks . . . . .	23
2.4.2 Finite State Machine Extraction from Recurrent Neural Networks	30
3 Rule Extraction From Feedforward Neural Networks	36
3.1 New Error Term $E_2$ . . . . .	36
3.2 Rule Extraction Algorithm . . . . .	39
3.3 An Illustrative Example . . . . .	44
3.4 Experimental Results . . . . .	46
3.5 Class Label-Aware Separation . . . . .	55
3.5.1 Learning Rules . . . . .	56
3.5.2 Experimental Results . . . . .	57
3.6 Discussion . . . . .	67
4 Finite State Machine Extraction from Simple Recurrent Networks	70
4.1 Generalizing $E_2$ . . . . .	71
4.2 Finite State Machine Extraction . . . . .	73
4.3 Experimental Methods . . . . .	76
4.4 Results . . . . .	81
4.4.1 An Illustrative Example . . . . .	81
4.4.2 Systematic Evaluation . . . . .	86
4.5 Discussion . . . . .	94
5 Finite State Machine Extraction from Echo State Networks	96
5.1 $E_2$ Derivation and Algorithm . . . . .	98
5.1.1 The Derivation . . . . .	98
5.1.2 Algorithm to Decrease $E_2$ in an ESN . . . . .	102
5.2 Experimental Methods . . . . .	105
5.2.1 Data Sets . . . . .	105
5.2.2 Experimental Settings . . . . .	107
5.2.3 $E_2$ Normalization . . . . .	109

5.3	Results . . . . .	111
5.3.1	Regular ESN versus ESN with $E_2$ . . . . .	112
5.3.2	Verification of Separation into Clusters . . . . .	117
5.3.2.1	Principal Component Analysis . . . . .	117
5.3.2.2	Histograms of Distances . . . . .	120
5.3.2.3	Weight Permutation Analysis . . . . .	123
5.3.3	Learning with $E_2$ versus ESN+ . . . . .	124
5.4	Discussion . . . . .	128
6	Conclusion . . . . .	131
6.1	Summary . . . . .	131
6.2	Contributions . . . . .	137
6.3	Limitations and Future Directions . . . . .	139
	Bibliography . . . . .	142

## List of Tables

3.1	Data Sets Used for Evaluation . . . . .	47
3.2	Regular versus Modified Backpropagation (Averaged over 100 Runs) .	50
3.3	Means ( $\sigma$ ) of Accuracy and Number of Rules over 100 Runs . . . . .	53
3.4	P-value of t-tests Comparing $E_1$ and $E_1 + E_2$ . . . . .	54
3.5	Means ( $\sigma$ ) of Accuracy and Number of Rules over 100 Runs . . . . .	58
3.6	P-value of t-tests on Number of Rules . . . . .	60
3.7	P-value of t-tests on Accuracy Rates . . . . .	60
3.8	Means ( $\sigma$ ) of Number of Antecedents over 100 Runs . . . . .	61
3.9	Means ( $\sigma$ ) of Accuracy and Number of Rules over 100 Runs with Rules Resulting in Default Class Removed . . . . .	64
3.10	Average Time for Rule Extraction (RE) and Total Running Time (Seconds) . . . . .	66
4.1	Context Free Grammar #1 . . . . .	78
4.2	Context Free Grammar #2 . . . . .	78
4.3	Properties of Data Sets Used for Evaluation . . . . .	78
4.4	Regular versus Modified Backpropagation (Averaged over 100 Runs) .	85
4.5	Means ( $\sigma$ ) of Number of States and Transitions over 100 Runs . . . .	87
5.1	$E_1$ and $E_2$ of Regular versus Modified ESN (Averaged over 100 Runs)	111
5.2	Means ( $\sigma$ ) of Number of States and Transitions over 100 Runs . . . .	113
5.3	ESN* $_{E_2}$ versus ESN with $E_2$ and ESN (Averaged over 100 Runs) . . .	123
5.4	Means ( $\sigma$ ) of Number of States and Transitions over 100 Runs . . . .	123

## List of Figures

2.1	A typical artificial neuron . . . . .	8
2.2	A fully connected feedforward neural network . . . . .	10
2.3	A feedforward network processes a sentence using a moving window . . . . .	15
2.4	A small simple recurrent neural network . . . . .	16
2.5	An example of how a simple recurrent network is trained . . . . .	17
2.6	A typical echo state network . . . . .	20
2.7	An illustrative example of the VIA method . . . . .	26
2.8	An example of hidden unit activation clustering and discretization . . . . .	27
2.9	An example of NeuroLinear’s hidden unit activation discretization . . . . .	28
2.10	A simple recurrent network’s dynamics viewed as a FSM’s . . . . .	31
2.11	Finite state machine for the <i>badigu</i> language . . . . .	31
3.1	Input patterns in the hidden activation space for the <i>waveform</i> problem . . . . .	45
3.2	Mean squared distance and mean squared error . . . . .	50
3.3	A typical plot of the average network error and average hidden layer activation pattern separation . . . . .	52
3.4	Mean number and accuracy of extracted rules . . . . .	53
3.5	Mean number of extracted rules . . . . .	58
3.6	Mean accuracy of extracted rules . . . . .	59
3.7	A rule set extracted from a neural network trained on the <i>splice</i> data set . . . . .	61
3.8	Mean number of extracted rules with rules resulting in default class removed . . . . .	64
3.9	Mean accuracy of extracted rules with rules resulting in default class removed . . . . .	65
4.1	A small simple recurrent neural network . . . . .	71
4.2	Finite state machine for the <i>badigu</i> language . . . . .	77
4.3	A typical hidden unit activation vectors throughout all time steps for the <i>badigu</i> problem . . . . .	83
4.4	Clustered hidden unit activation vectors throughout all time steps for the <i>badigu</i> problem . . . . .	84
4.5	Mean squared distance and mean squared error . . . . .	86
4.6	Mean number of states and transitions . . . . .	87
4.7	Extracted FSM for data set <i>Tomita #4</i> . . . . .	89
4.8	Extracted FSM for data set <i>CFG #1</i> . . . . .	90
4.9	Extracted FSM for data set <i>CFG #2</i> . . . . .	91
4.10	Hand-design minimal FSM for <i>CFG #2</i> . . . . .	93
5.1	A typical echo state network . . . . .	96
5.2	Two sets of simple made-up hidden unit activation space patterns. . . . .	109
5.3	Mean squared distance and mean squared error . . . . .	112
5.4	Mean number of states and transitions . . . . .	114
5.5	Extracted FSM from ESN for data set <i>CFG #2</i> . . . . .	115



5.6	A typical PCA projection of the hidden activation patterns of a regular ESN . . . . .	118
5.7	A typical PCA projection of the hidden activation patterns of an ESN augmented with $E_2$ . . . . .	118
5.8	A typical histogram of normalized distances between hidden unit activation vectors of a regular ESN . . . . .	120
5.9	A typical histogram of normalized distances between hidden unit activation vectors of an ESN augmented with $E_2$ . . . . .	121
5.10	A typical PCA projection of the hidden activation space of a regular ESN . . . . .	126
5.11	A typical histogram of normalized distances between hidden unit activation vectors in an ESN+ . . . . .	127

## Chapter 1

### Introduction

Error backpropagation is the most widely used supervised learning method for neural networks and has achieved success in a wide range of applications. There are several variants of the algorithm that are used to train both feedforward and recurrent networks. Almost all of them are driven by minimizing the sum of squared error between the network's actual output values and the teaching signals. The hidden units' activation patterns are central to a network's decision at the outputs because they form the internal representation of the input. During error backpropagation training, the learning algorithm is free to create any hidden layer representation as long as it minimizes the error at the output. While there has been much work on training neural networks to achieve lower output error rates, faster speed, and simpler sets of weights, there is currently only a limited understanding of how to directly influence the creation of the hidden layer representation and how this representation affects various performance measures of a network.

Another issue with error backpropagation is that the end result of training is large opaque weight matrices of floating point numbers that are very difficult for a person to understand. This difficulty has inspired substantial research on how to extract a symbolic, human readable representation from trained backpropagation networks. Such representations are useful for knowledge acquisition and data mining

because they allow us to gain insight into the data. Moreover, they are instrumental in verifying neural network solutions for critical applications. An example, in the credit risk evaluation task, a single yes/no decision is not sufficient because one must also provide reasons for denying an application to make sure that one complies with appropriate laws. In spite of a large amount of work addressing this issue [2, 4, 14, 37, 57], the results obtained are still very limited.

There are three main approaches that have been taken in past work on symbolic representation extraction from neural networks: pedagogical, decompositional and eclectic (a hybrid of the first two) [2, 37]. *Pedagogical methods* consider a neural network to be a black box oracle that provides class labels for any input vectors, including the ones that are not in the training set. Notable algorithms include OSRE [19], RE-RX [71], and Minerva [36]. They extract input-output rules without looking at the units and weights. *Decompositional methods* investigate hidden units and weight matrices to produce rules that follow the internal working of the networks [37, 73, 83]. *Eclectic methods* are a hybrid of the other two methods. Decompositional methods are the most popular and successful methods mainly because they can take advantage of more knowledge about the networks. Our focus in the rest of this dissertation will be on decompositional methods.

Many decompositional methods share an important clustering step in which the hidden activation space is divided into regions that will: (1) lead to the same classification at the output (for feedforward network), or (2) represent states of the network dynamics (for recurrent networks). For both types of networks, the result of this step is crucial in determining the quality of the final results. If the cluster

regions are numerous or ambiguous, the symbolic representation will be complex and verbose. On the other hand, having just a few clear-cut regions will lead to simple symbolic representations that are easy to understand. Research in this area has largely focused on creating progressively more powerful clustering methods, better ways to express the rules, and learning networks with fewer weights. In contrast, relatively little work has been done on altering training methods to learn a better hidden representation so that *any* symbolic representation extraction method becomes more effective. Potentially, a better hidden layer representation might allow the extraction of fewer and more compact regions in the hidden activation space, thereby leading to a more concise and easier-to-understand symbolic representation.

## 1.1 Goals and Specific Aims

The central goal of this research is to develop ways to influence neural network training so that it produces better separated hidden activation patterns, and to study how the altered training methods affect a network's accuracy, performance, generalizability, and comprehensibility. More specifically, the intent is to create methods that will allow the extraction of simpler symbolic representations from neural networks while maintaining the networks' performance. My hypothesis is that among the many possible encodings that can appear at the hidden layer during learning, the ones that create a more separable set of activation patterns will be easier to convert into symbolic forms. Further, because the hidden layer activation space is bounded, being limited by the range of the activation functions, as we make

the portion of space occupied by activity patterns more separable, for example, pushing the hidden unit activation vectors away from each other in appropriate ways, they will form clusters of activation vectors with greater separation from each other.

In this context, the following are the specific objectives of this research:

1. Derive modified error backpropagation learning rules that lead to better separated representations of hidden unit activation patterns in feedforward neural networks. The main approach to be taken is to augment the usual error function with new “error” terms that increase when the hidden layer activation vectors are closer together. In the spirit of neural computation, these new error terms should be capable of being computed or approximated locally and efficiently. Gradient descent is applied on the combined error terms to adapt the weights so that the network produces the correct output with a better separated internal representation than would occur with standard backpropagation.
2. Apply the modified error backpropagation method that is derived as above to a battery of data sets to determine the effects such an approach has on extracting symbolic representations from a feedforward network’s learned mapping. Historically, most past work on representing what a neural network has learned symbolically has focused on developing increasingly powerful rule extraction algorithms. In contrast, my goal is not to develop new rule extraction algorithms but to apply existing ones to networks trained *with* and *without* the new error terms, and then compare the final number and accuracy of the rules extracted. Experimental results are used to show that better separated

activity patterns allow the extraction of fewer clusters and intervals of hidden unit activations, thus leading to fewer intermediate rules being extracted by the rule extraction process, and to fewer rules overall.

3. Generalize the methods derived above for feedforward networks having static inputs to simple recurrent networks having temporal sequences as inputs. A simple recurrent network, as that term is used here, has a set of context units that is the network's internal representation of input it has processed so far, thus giving the network a *context* when it processes a new input. The methods used for feedforward networks are now used to make the training process create internal representations that are more separable and can be divided into fewer well-defined clusters. I then test the hypothesis that this helps extract simpler finite state machines (FSMs) from recurrent networks on data sets consisting of temporal sequences generated from both regular and context-free grammars.
4. Extend the above methods to work with echo state networks (ESNs). ESNs are large recurrent neural networks whose connections from input units to hidden units, and connections among hidden units forming a reservoir, are typically initialized randomly and held fixed, while only connections from the reservoirs to the output layer are trained. Building on the results above, I design ways to adapt the weights from the input layer to the hidden layer that create a better separated hidden unit representation. Then I test the hypothesis that such representations also facilitate the extraction of FSMs from ESNs.

This research extends our knowledge of neural networks' internal encoding

and how to influence the learning process to get better encodings for subsequent representations in symbolic forms. Furthermore, it presents ways to extract simpler sets of symbolic rules and simpler FSMs, which allows us to understand how the networks work, and also to understand the data better. In addition, it provides better ways to initialize and train different types of recurrent neural networks, and opens new paths to improve and design new neural network training algorithms.

## 1.2 Overview

The rest of this dissertation is organized as follows. Chapter 2 presents background information about supervised learning in feedforward, simple recurrent, and echo state neural networks. It then discusses existing symbolic representation extraction methods for the above types of networks. Chapter 3 presents three new error terms that encourage error backpropagation to learn a better separated encoding at the hidden layer of feedforward networks while keeping classification accuracy as good as with regular error backpropagation. It also presents a symbolic rule extraction algorithm for feedforward neural networks and shows how, using this altered encoding, significantly simpler sets of symbolic rules can be extracted from the networks without sacrificing accuracy. Chapter 4 presents the generalization of one new error term to simple recurrent networks, an FSM extraction algorithm, and how it takes advantage of the better encoding to produce simpler FSMs. Chapter 5 presents the extension of the above methods to ESNs and FSM extraction from ESNs. It also discusses extensive experiments to verify the separability of ESNs' very

high dimensional activation space and compares the methods developed here with ESN+, a variant of ESN. Chapter 6 concludes the dissertation and discusses future research directions.



## Chapter 2

### Background

This chapter briefly reviews the main artificial neural network architectures, from simple feedforward networks to recurrent networks, that are directly relevant to the proposed research. We will first look at how each type of network is set up and trained to produce correct behaviors, and then at methods for symbolic representation extraction from neural networks.

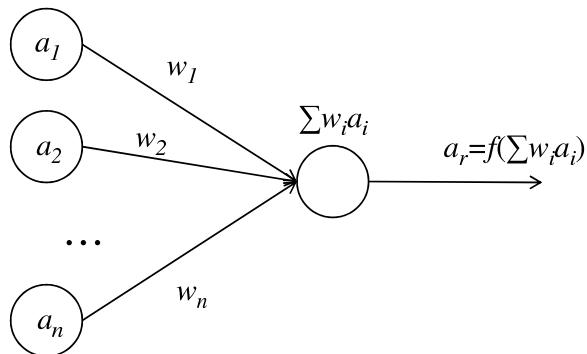


Figure 2.1: A typical artificial neuron  $r$ . The values  $a_1 \dots a_n$  are activations of input neurons,  $w_1 \dots w_n$  are connection weights, and  $a_r$  is the activation level of the output/response unit. The latter neuron sends its activation  $\sigma(\sum w_i a_i)$  to other neurons.

Artificial neural networks are computational models inspired by biological neural networks [54]. A network consists of simple interconnected units, or neurons, where each unit/node receives input from other units and sends its output to others. Figure 2.1 shows an artificial neuron model first described by McCulloch and Pitts in 1943 [54]. It takes a weighted sum of the inputs, passes the sum through an

activation function, usually a step threshold or a sigmoid function, and then sends the output value to other receiving neurons. This paradigm persists today in most artificial neural network models.

When  $f$  is a linear function, a network of such neurons can only do a linear transformation on the input, and thus can only learn a linear function of the input pattern. When  $f$  is a linear threshold function (or other non-linear function), a network with only input and output units can only learn to classify linearly separable inputs. But when we assemble neurons with nonlinear activation functions, such as logistic or hyperbolic tangent functions, so that intermediate neurons are also present, the network can exhibit much more complex behaviors, including approximating any discrete/continuous function, discrete time dynamical systems or Turing machines [7, 28, 77]. In such a network, neurons are typically divided into three classes by their role: input units represent the input given to the network, output units express the network's output, and hidden units do the internal computations.

Artificial neural networks can learn to produce a correct/target behavior by adjusting their connection weights. Error backpropagation is the most popular supervised learning algorithm for training neural networks. A form of error backpropagation was first described by Rosenblatt in 1962 [66], and gained popularity in 1986 with a more modern version produced by the work of Rumelhart, Hinton, Williams [67]. The algorithm computes the derivative of the error function with respect to each weight efficiently by propagating the error signal from the output units to the input.

## 2.1 Multilayer Feedforward Neural Networks

A *multilayer feedforward neural network* is a type of artificial neural network in which the units are organized into ordered layers and there are only connections in the forward direction, so the network is acyclic. While networks with multiple hidden layers are appropriate for some problems, such as hierarchical processing and function compositions, networks with one hidden layer remain the most popular architecture because they are simpler and relatively fast but still can approximate any function with a finite number of hidden units [28]. In this research, we will focus initially on networks having one hidden layer. Figure 2.2 shows a very simple example of a fully connected feedforward neural network.

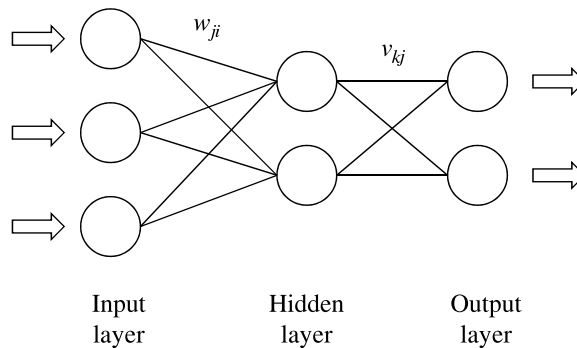


Figure 2.2: A fully connected feedforward neural network illustrating input, hidden and output layers.

The activation of the  $j^{th}$  hidden unit when the  $p^{th}$  input pattern is presented is typically calculated as the logistic function of the weighted sum of inputs:

$$a_{H_j}^p = \sigma\left(\sum_{i \in \text{input}} w_{ji} x_i^p\right)$$

where

- $x_i^p$  is the  $i^{th}$  input unit value of the  $p^{th}$  training instance.

- $w_{ji}$  is the weight from the  $i^{th}$  input unit to the  $j^{th}$  hidden unit, and
- $\sigma()$  is the logistic function  $\sigma(x) = \frac{1}{1+e^{-x}}$ .

The activation of the  $k^{th}$  output is calculated as the logistic function of the weighted sum of hidden unit activations:

$$a_{O_k}^p = \sigma\left(\sum_{j \in \text{hidden}} v_{kj} a_{H_j}^p\right)$$

where  $v_{kj}$  is the weight from the  $j^{th}$  hidden unit to the  $k^{th}$  output unit. Another way to look at what the activation rule does is that it maps the input vector  $x^p$  of the  $p^{th}$  training instance to the hidden activation vector  $a_H^p$ , and maps the hidden activation vector  $a_H^p$  to the output vector  $a_O^p$ .

In supervised training with multilayer feedforward neural networks, *target* correct values of the output units for each set of input values are given. Training amounts to adjusting the weights so that the network's actual output and the target output match within a given tolerance. This is most commonly done by error backpropagation where the *error*, or difference of output and target values, is propagated from the output layer back to the hidden layer.

The usual error function computed over the output units is designated  $E_1$  here, and is given by:

$$E_1 = \frac{1}{2} \sum_{p=1}^N \sum_{k \in \text{output}} (T_k^p - a_{O_k}^p)^2$$

where  $T_k^p$  is the *target* output for the  $k^{th}$  output unit when the  $p^{th}$  input pattern is presented, and  $N$  is the size of the input data set (number of input-output pairs in the training data). At each layer, the error signals are used to compute new

weights. The appropriate weight change rules are derived using gradient descent [55]. For networks using logistic units and the  $E_1$  error function above, the weight changes  $\Delta v_{kj}^p$  and  $\Delta w_{ji}^p$  for  $v_{kj}$  and  $w_{ji}$  when the  $p^{th}$  training example is presented are computed as follows:

$$\begin{aligned}
\delta_{O_k}^p &= (T_k - a_{O_k}^p)(a_{O_k}^p)(1 - a_{O_k}^p) \\
\Delta v_{kj}^p &= \eta \delta_{O_k}^p a_{H_j}^p \\
\delta_{H_k}^p &= \sum_{k \in output} v_{kj} \delta_{O_k}^p (a_{H_j}^p)(1 - a_{H_j}^p) \\
\Delta w_{ji}^p &= \eta \delta_{H_k}^p x_i^p
\end{aligned} \tag{2.1}$$

where  $\eta$  is a small learning rate. This algorithm has been widely successful in training feedforward networks for classification and regression problems [26].

Throughout this dissertation, *RPROP* [64] (resilient backpropagation), an improved, modern version of the backpropagation learning algorithm that trains networks faster by adjusting the weight update based solely on the direction of the gradient instead of also including the magnitude of the derivatives is used. It also requires fewer training parameters than regular backpropagation does. The main idea of RPROP is that if the product  $\frac{\partial E^t}{\partial w_{ji}} \times \frac{\partial E^{t-1}}{\partial w_{ji}} < 0$ , i.e. the gradient  $\frac{\partial E}{\partial w_{ji}}$  at time step  $t$  *changed* sign, the last weight update must have “overshot” the target. Thus, the next weight update for  $w_{ji}$  should be smaller, otherwise, the weight update can be larger to speed up training.

The RPROP algorithm lets each weight  $w_{ji}$  have an associated weight update value  $\Delta_{ji}$  that is initialized to a small value, typically 0.1. Then, at each time step  $t$ , if  $\frac{\partial E^t}{\partial w_{ji}} \times \frac{\partial E^{t-1}}{\partial w_{ji}} < 0$ ,  $\Delta_{ji}$  is decreased, usually by half, because the last weight update

“overshot”. Alternatively,  $\Delta_{ji}$  is increased if  $\frac{\partial E^t}{\partial w_{ji}} \times \frac{\partial E^{t-1}}{\partial w_{ji}} > 0$ . Finally, each weight  $w_{ji}$  is increased or decreased depending on the direction of the gradient  $\frac{\partial E}{\partial w_{ji}}$  and the magnitude of the weight update value  $\Delta_{ji}$ :

$$w_{ji} = w_{ji} - \text{sign}\left(\frac{\partial E^t}{\partial w_{ji}}\right) \times \Delta_{ji}$$

There are two previous types of past work that have modified the error function  $E_1$  to guide error backpropagation learning and are closely related to the work proposed here: *regularization* and the *SIR* (Separation of Internal Representations) model. *Regularization* restricts the hypothesis space by adding more information to a problem in order to prevent overfitting. For neural networks, this is usually done by adding penalty terms to the error function to limit the number of connections, hidden units, or weight sizes. One of the most popular and efficient methods is *weight decay* which uses an additional error term  $E_d = \lambda \sum_j \sum_i w_{ji}^2$  that is added to  $E_1$  to prevent weights from getting too large [46]. Having large weights makes learning unstable and impairs networks’ ability to generalize to novel data. Unimportant weights are usually reduced by this term to near zero, and could subsequently be removed.

The second closely related work, the *SIR* model, was published in two recent papers [47, 48]. In the first of these papers [47], the authors adopt the same philosophy that one should maximally separate the activity patterns in different classes. But the specific approach taken in *SIR* was limited in that it works only on one layer networks (no backpropagation), it does not combine with error minimization ( $E_1$ ), and it was just a preliminary demonstration that one could separate patterns better.

A subsequent recent paper [48] suggests that such an approach could be integrated with error backpropagation in a fashion similar to what is done here. However, this latter work did not actually describe the derived learning method in detail, making it problematic for others to use. It presented a *global* learning approach in which the complete network state is required to compute each weight change. That is both computationally expensive and in violation of the spirit of neural computation where the whole point is to implement learning as local computation. The authors only applied the algorithm to small feedforward problems (Fisher’s Iris data set [3], and an 8-bit encoding problem), and did not examine rule extraction from trained neural networks in any systematic way as is done in this thesis.

## 2.2 Simple Recurrent Neural Networks

Because feedforward neural networks can only learn the relationships between fixed length multivariate input and output patterns, they would need to use a moving window to process sequential data, such as time series. In this approach, the input layer consists of  $k$  groups of input units corresponding to  $k$  time steps that are needed to look back, each group presenting the input signals to be received at a different time step [70]. Figure 2.3 shows an example of a feedforward neural network as it processes a sentence using a moving window in a next-word prediction task. At the first time step, the network receives the first two words of the sentence *the quick brown fox jumps over . . .* and learns to predict the target word *brown* at the output layer. Next, it receives the second and third word (*quick, brown*) and learns

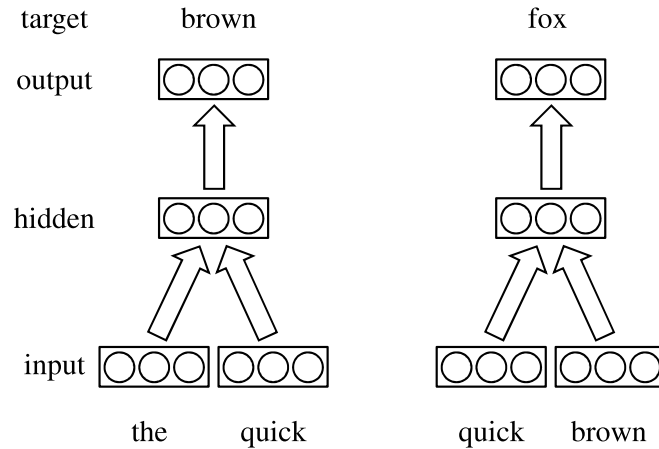


Figure 2.3: A feedforward neural network processes a sentence using a moving window of size 2. Arrows denote the directed connections from the input layer to the hidden layer and from the hidden layer to the output layers. The input layer is drawn as two blocks to signify that two groups of input units receive two input symbols at each time step.

to predict the fourth word *fox*. In this network, the window allows the network to look back one time step in the past beside the current word. This usually results in too many weights being needed because the hidden units have to be connected to each set of inputs for every time step the network looks back. Furthermore, the weights are usually duplicated for processing the same type of input, and this leads to poor generalization. Thus, the network is limited to looking back only a few time steps to keep the number of weights from growing too big.

*Elman recurrent neural network* was introduced by Elman in 1990 [17] to address the problem above. It allows recurrent connections without having fully arbitrary connectivity among the units. Elman network belongs to a class of recurrent networks called *simple recurrent networks* because only the forward connections are trained. There are many variants of simple recurrent networks [43], which may contain many layers and complex connectivity, but Elman network is the most



popular. From here on in this dissertation, we will mean this variant when we refer to simple recurrent networks.

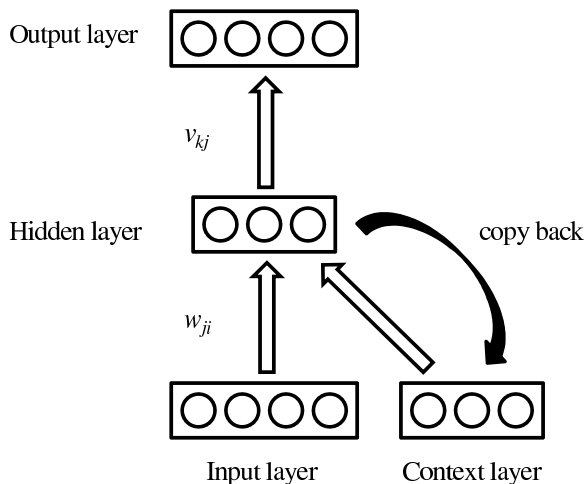


Figure 2.4: A small simple recurrent neural network; the number of nodes in each layer varies and can be quite large.

In these networks, the recurrent connections are fixed, not arbitrary, and restricted to performing a copy operation so that no special backpropagation method is required. Then, a set of units called *context units* are added to the network as illustrated in Figure 2.4. These units' activations are not from the input data but instead copied from either the activations at the output layer or the hidden layer from the last time step. Thus, the context units store a representation of the past time steps inputs and activity of the network. The recurrent connections are from the hidden layer to the context layer, and only copy the activations from the hidden layer in the previous time step to the context layer.

Figure 2.5 shows an example of how a simple recurrent network is trained on a next-word prediction task on the sentence *The quick brown fox jumps ...* through

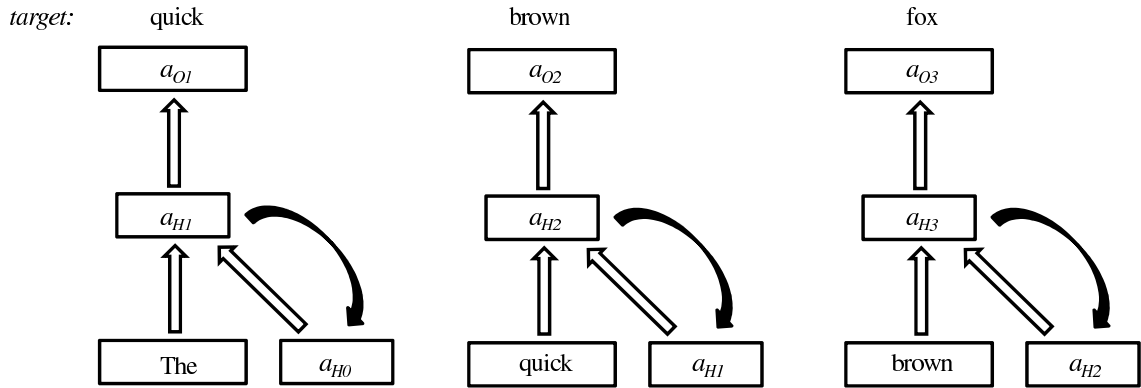


Figure 2.5: An example of how a simple recurrent network is trained on a next-word prediction task through the first three time steps.

the first three time steps. At the beginning, the context unit’s activation vector is usually initialized to  $a_{H_0} = 0$  and the network receives an input activation for the word *the*. The hidden unit activation and the output vector can be calculated as  $a_{H_1}$  and  $a_{O_1}$ , respectively. The difference (error) between the output  $a_{O_1}$  and the representation of the correct target *quick* is then used by error backpropagation to calculate the proper weight changes in a manner similar to that of feedforward networks. As noted above, only the forward weights (in white) are changed, while the recurrent connections from the hidden layer to the context layer (in black) are not. These connections are fixed and only serve to copy the activations. Next, at the second time step,  $a_{H_1}$  is copied from the hidden layer to the context layer. The network now receives the next word *quick* and the context  $a_{H_1}$ . Thus,  $a_{H_1}$  is considered to store the “context” in which the word *the* has been processed. Using the augmented input, the network calculates the hidden activation vector  $a_{H_2}$  and the output  $a_{O_2}$  are calculated. Now,  $a_{H_2}$  stores the context in which two words *the* and *quick* have been processed. Error backpropagation is again used to calculate the

weight changes. The process is repeated until the end of the sentence.

Formally, the activation of the  $j^{th}$  hidden unit when the input pattern at time step  $t$  is presented is calculated as the logistic function of the weighted sum of inputs and context activations:

$$a_{H_j}^t = \sigma\left(\sum_{i \in input} w_{ji}x_i^t + \sum_{i \in context} w_{ji}x_i^t\right)$$

where

- $x^t$  is the augmented vector of the input and the context at time step  $t$ ;
- $w_{ji}$  is the weight from the  $i^{th}$  input/context unit to the  $j^{th}$  hidden unit; and
- $\sigma()$  is the logistic function  $\sigma(x) = \frac{1}{1+e^{-x}}$ .

Similarly, the activation of the  $k^{th}$  output is calculated as the logistic function of the weighted sum of hidden unit activations:

$$a_{O_k}^t = \sigma\left(\sum_{j \in hidden} v_{kj}a_{H_j}^t\right)$$

where  $v_{kj}$  is the weight from the  $j^{th}$  hidden unit to the  $k^{th}$  output unit (see Figure 2.4).

The activation rule maps the augmented input vector  $x^t$  at time step  $t$  consisting of the input data and the context, to a hidden unit activation vector  $a_H^t$ . Then,  $a_H^t$  is mapped to an output vector  $a_O^t$ .

The usual error function computed over the output units is:

$$E_1 = \frac{1}{2} \sum_{t=1}^N \sum_{k \in output} (T_k^t - a_{O_k}^t)^2$$

where  $T_k^t$  is the target output for the  $k^{th}$  output unit at time step  $t$ , and  $N$  is the length of the input sequence.

Training a simple recurrent network proceeds by presenting the training sequence to the network repeatedly and using error backpropagation to calculate the weight changes. Learning only occurs on forward connections because the backward copy connections are fixed. At each time step, the activation from the previous time step is copied back to the context layer, and then the network acts and can be trained exactly like a feedforward network with the input consisting of both the input data and the context values. For networks using logistic units and the  $E_1$  error function above, the weight change  $\Delta v_{kj}^t$  and  $\Delta w_{ji}^t$  for  $v_{kj}$  and  $w_{ji}$  at time step  $t$  are computed as follows:

$$\begin{aligned}
\delta_{O_k}^t &= (T_k - a_{O_k}^t)(a_{O_k}^t)(1 - a_{O_k}^t) \\
\Delta v_{kj}^t &= \eta \delta_{O_k}^t a_{H_j}^t \\
\delta_{H_k}^t &= \sum_{k \in \text{output}} v_{kj} \delta_{O_k}^t (a_{H_j}^t)(1 - a_{H_j}^t) \\
\Delta w_{ji}^t &= \eta \delta_{H_k}^t x_i^t
\end{aligned} \tag{2.2}$$

Simple recurrent networks have been used successfully in learning temporal sequence tasks including context-free languages [6, 65], next word predictions [18], and word forms and pronunciations [76]. In this past work, the authors analyzed the state space and state trajectory to gain an understanding into how the networks worked, predicted, and generalized, and why they did not work in certain cases.

## 2.3 Echo State Networks

One of the biggest problems with using recurrent networks is that there is still no efficient algorithm to train large networks. An *echo state network* (ESN) is a recent approach that addresses this problem in an interesting way: it typically does not train the *input*  $\rightarrow$  *hidden* and the *hidden*  $\rightarrow$  *hidden* (recurrent) connections at all [39]. In an echo state network, typically only the connections from the hidden units to the output units are trained, while the rest are initialized randomly and held fixed. As illustrated in Figure 2.6, the set of hidden unit is called the *reservoir* and usually consists of a large number of hidden units, usually 100 or more. Hidden units receive activation from a sparse random subset of other hidden units, so they respond differently and produce their own sequence of activation values as input signals come into the network. The connections from the hidden units to the output units are seen

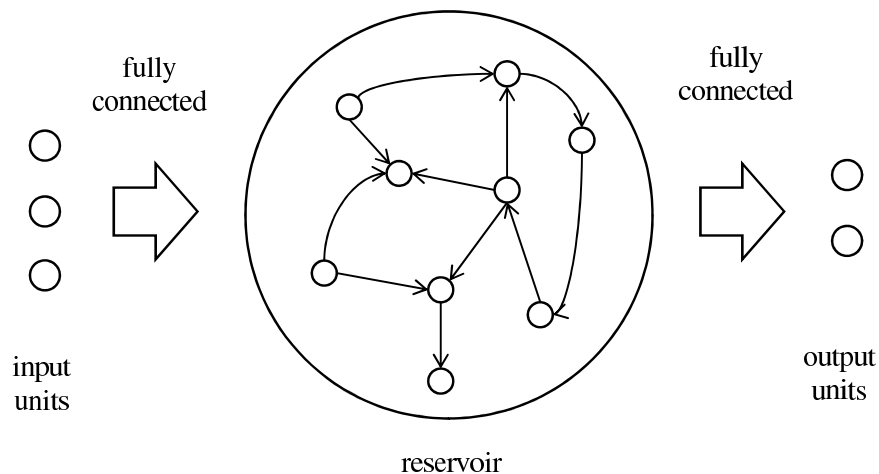


Figure 2.6: A small but otherwise typical echo state network with 2 input units and 2 output units. Connections from input units to the reservoir and connections inside the reservoir are generated randomly and sparsely. The reservoir is fully connected to the output units. These latter connections are trained so that the output comes to match the target training signals.

as capturing the relevant dynamics from these “basis” activation patterns. These hidden-to-output connections are trained in a supervised manner so that the output matches the desired values for every time step, and such training is very fast. These connections are also called “read-out” because they select and aggregate the complex dynamics of the hidden units in the reservoir to produce the output.

Training an echo state network proceeds by presenting the training sequence to the network continuously and recording the activations  $a_{H_j}^t$  of each hidden unit at time step  $t$  in the reservoir. The output activation  $a_{O_k}^t$  is computed as:

$$a_{O_k}^t = \sum_{j \in \text{hidden}} w_{kj}^{\text{out}} a_{H_j}^t$$

with  $w_{kj}^{\text{out}}$  being the weight of the connection from the  $j^{\text{th}}$  hidden unit in the reservoir to the  $k^{\text{th}}$  output unit and  $a_{H_j}^t$  being the activation of the  $j^{\text{th}}$  hidden unit at time step  $t$ .  $a_{H_j}^t$  is computed as:

$$a_{H_j}^t = \sigma \left( \sum_{i \in \text{hidden}} w_{ji}^{\text{res}} a_{H_i}^{t-1} + \sum_{i \in \text{input}} w_{ji}^{\text{in}} x_i^t \right) \quad (2.3)$$

where

- $w_{ji}$  is the weight from the  $j^{\text{th}}$  hidden unit to the  $i^{\text{th}}$  hidden unit;
- $w_{ji}^{\text{in}}$  is the weight from the  $i^{\text{th}}$  input unit to the  $j^{\text{th}}$  hidden unit; and
- $w_{ji}^{\text{res}}$  is the weight from the  $i^{\text{th}}$  hidden unit to the  $j^{\text{th}}$  hidden unit, both in the reservoir; and
- $x^t$  is the input vector at time step  $t$ .

The goal is to learn the weight matrix of  $w_{kj}^{out}$  that minimizes the sum of squared differences between the output  $a_{O_k}^t$  and the desired output  $T_k(t)$ . That can be done very efficiently by linear regression as shown in [39]. Training an echo state network only needs one pass through the data, is thus very fast, and has been shown to be very effective in many problems [41, 42, 52].

In order for the echo state network approach to work, the randomly generated reservoir has to have the “echo state” property which depends on the connection weights *and* the training data. The property can be stated informally as “if the network has been run for a very long time (from minus infinity time), the current network state is uniquely determined by the history of the input and the teacher-forced output (target)” ([40]). The formal mathematical definition can be found in [39]. While necessary and sufficient conditions for the echo state property are not known, there is a heuristic to generate a reservoir that gives this property for most training data [39]: generate a sparse weight matrix with random small weights, then scale it so that its spectral radius is smaller than 1.

It remains an open problem to construct the reservoir, or adapt it so that it has richer, more appropriate dynamics for a particular problem. In fact, it was reported by Prokhorov in [60] that out of 1000 runs in which the author trained an echo state network on the Mackey-Glass [53] problem, a task for which echo state network was shown to have superior performance, only a fraction of the solutions achieved high accuracy, and most diverged quickly after a few hundreds of time step. Currently, it is not possible to know if a reservoir is good or not for a problem before seeing the results of training [60].

## 2.4 Symbolic Representation Extraction

Symbolic representation extraction from neural networks is the process of generating a set of symbolic, human-understandable representations that can be used to determine the output of a neural network without putting the input through the network itself. With this, a person can understand better what was learned from the data and be more confident about the network's output. Consequently, it can be used for knowledge acquisition, data mining, and verifying neural network solutions for use in mission critical applications. Typically, the result of symbolic representation extraction from a feedforward network is a set of symbolic rules, while the result of extraction from a recurrent network is a finite state machine (FSM)[27]. Consequently, the processes are called *rule extraction* and *finite state machine extraction*, respectively. This section will present an overview of both types of extraction with a focus on past work that is most relevant to this research.

### 2.4.1 Rule Extraction from Feedforward Neural Networks

A feedforward neural network training algorithm learns a mapping between the input and the output units in the form of a weight matrix consisting of floating point numbers. The large matrix often involved makes it very difficult for a person to know what the network has learned. As noted earlier, that has inspired a large amount of work addressing this issue, but the results obtained are still very limited [2, 11, 13, 35, 37, 80]. Most of these methods extract propositional logic rules having the form



$$(input_{i_1} = v_1, input_{i_2} = v_2, \dots) \rightarrow class = C_j$$

or the form of *M of N* rules [72, 82]:

$$((M \text{ of } (input_{i_1}, input_{i_2}, \dots, input_{i_N}) \text{ are on}), \dots) \rightarrow class = C_j$$

There is also some work that extracts fuzzy rules [8, 30] or first order logic rules [57]. While *M of N* rules can be more expressive with the same number of rules in some problems, propositional logic rules are more intuitive to read and there has been extensive work on this type of rules. The work presented in this thesis also extracts propositional logic rules from feedforward networks, so it should be directly comparable to much of this past related work.

Three main approaches have been taken in past work on rule extraction from neural networks: pedagogical, decompositional, and eclectic (the latter being a hybrid of the first two) [2]. Pedagogical methods consider a neural network as a black box oracle that provides class labels for both seen and unseen inputs. They extract rules by examining the output of the network without looking at the units and weights. Decompositional methods investigate hidden unit activations and weight matrices to produce rules that follow the internal working of the networks. While pedagogical methods are virtually independent of the network architecture, topology, and training algorithm, and can even be applied to non-neural network techniques, they cannot utilize the available information captured by weights and connections in the networks like decompositional methods do, and they do not directly capture the internal representation of the neural networks. For these reasons, in this research we will be using solely the decompositional approach to rule extraction.

This latter approach first extracts the rules that explain the mapping between

the hidden unit activation patterns and the output, and then extracts the rules governing the input-hidden layer relationship. These intermediate rules are then combined to produce the final input-output rules. One of the earliest work using this approach is the Validity Interval Analysis (VIA) method [78, 79]. VIA extracts rule in the form:

*if input  $\in$  hypercube  $H_i^I$  then class is  $C$ .*

A hypercube is defined using a set of intervals:  $\{[a_1, b_1], [a_2, b_2], \dots, [a_n, b_n]\}$ , where  $n$  is the number of dimensions. A vector  $(x_1, x_2, \dots, x_n)$  is in the hypercube if and only if:  $a_1 \leq x_1 \leq b_1, a_2 \leq x_2 \leq b_2, \dots, a_n \leq x_n \leq b_n$ .

VIA starts by assigning random hypercube  $H_i^H$  in the hidden unit activation space to hidden unit activation vectors. For each  $H_i^H$ , VIA use linear programming to calculate a hypercube  $H_i^I$  in the *input* space such that input vectors inside  $H_i^I$  have their corresponding hidden activation vectors inside the hypercube  $H_i^H$  in the *hidden unit activation space*. In other words, if *input*  $\in$  hypercube  $H_i^I$  then hidden unit activation vector  $\in$  hypercube  $H_i^H$ .

The hypercubes in the input space and hidden activation space are then continuously refined so that they are small and non-overlapping. Then, it is assumed that hidden activation vectors in each hypercube  $H_i^H$  only maps to output vectors belonging to a single class  $C$ . The combined result is that if an input vector belongs to the hypercube  $H_i^I$ , its corresponding hidden activation vector will belong to the hypercube  $H_i^H$ , and then the output class will be  $C$ . Figure 2.7 shows a simplified example of VIA's extracted rules and hypercubes. After the hypercubes are refined,

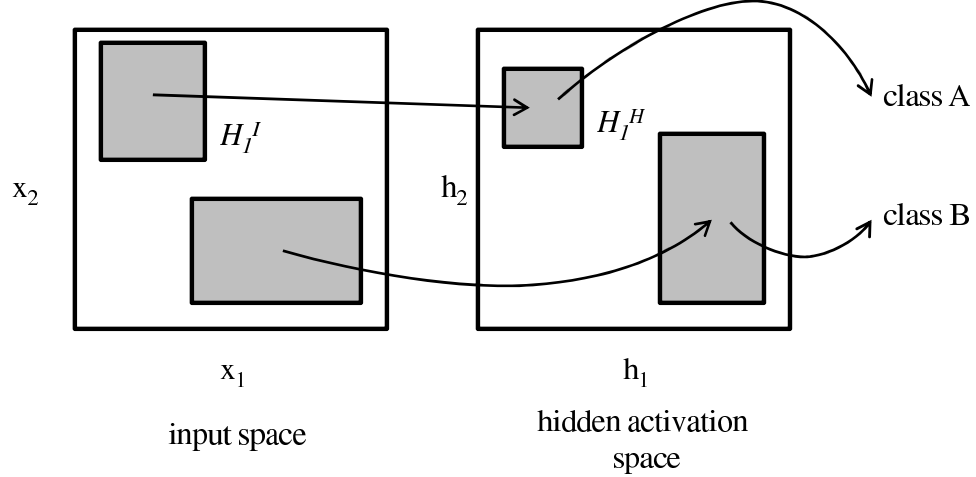


Figure 2.7: A simplified example of the VIA method on a network with two input units, two hidden units, and two output classes. The shaded boxes denote the hypercubes in the input and hidden activation spaces. The arrows represent the intermediate rules expressing how hypercubes in input spaces are mapped to hypercubes in hidden activation spaces, and from hypercubes in hidden activation spaces to the final output class.

the input-hidden rules are extracted. In this example, if an input vector  $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in H_1^I$  is given at the input, the hidden activation vector  $\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \in H_1^H$ . In addition, if a hidden activation vector is in  $H_1^H$ , the activation rule will result in only class A at the output layer. The two rules are then combined to:

$$\text{if input} \in H_1^I \text{ then class is A.}$$

The main limitation of the VIA method is that when the hidden activation patterns are distributed, it requires a large number of hypercubes  $H_i^H, H_i^I$  to divide them, and thus it leads to a large number of rules.

A different approach is taken in NeuroRule in which the individual hidden unit activation patterns are clustered and discretized [50, 74]. This algorithm uses

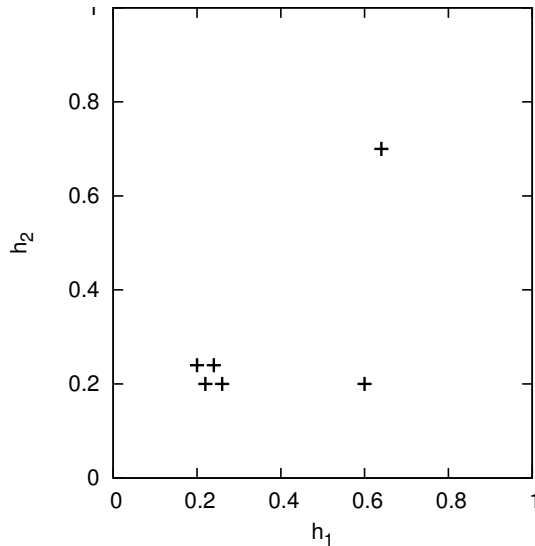


Figure 2.8: An example of hidden unit activation clustering and discretization by NeuroRule using a 2-dimensional hidden unit activation space. Each cross represents one hidden activation vector. See text for details.

a small parameter  $\epsilon$  to determine whether a hidden unit activation value is “close” enough to a cluster, and thus is considered to have the same discretized value as the other activation values in the same cluster. For instance, Figure 2.8 shows a two-dimensional hidden unit activation space with 6 hidden unit activation vectors. The activation values of hidden unit  $h_1$  are: 0.20, 0.22, 0.24, 0.26, 0.60, and 0.64. With  $\epsilon = 0.1$ , the algorithm finds two clusters  $\{0.20, 0.22, 0.24, 0.26\}$  and  $\{0.60, 0.64\}$  for  $h_1$  and assigns two discretized values, 0.25 and 0.62, to them (averages of values in a cluster). It also finds two clusters  $\{0.20, 0.24\}$  and  $\{0.7\}$  for  $h_2$  and assigns two discretized values 0.215 and 0.7 to them. Consequently, the four vectors at the lower left corner all have discretized coordinate (0.25, 0.22) and the other two vectors have discretized coordinate (0.62, 0.22) and (0.62, 0.7). If all four vectors at the lower left corner map to the same class  $C$  at the output, the discretization is a success because the algorithm can extract the rule:

*if  $(h_1, h_2)$  is close to  $(0.25, 0.22)$  then class is  $C$ .*

Then, NeuroRule will extract the rules prescribing the conditions on the input such that the hidden activation vector will be close to  $(0.25, 0.22)$ , and combine that rule with the hidden-output rule above. Otherwise, if all four vectors do not map to the same class, the algorithm reduces  $\epsilon$  and repeats the procedure again. A lower value of  $\epsilon$  results in smaller clusters and hence increases the chance that all hidden activation vectors with the same discretized coordinates map to the same class. However, it also results in having more clusters.

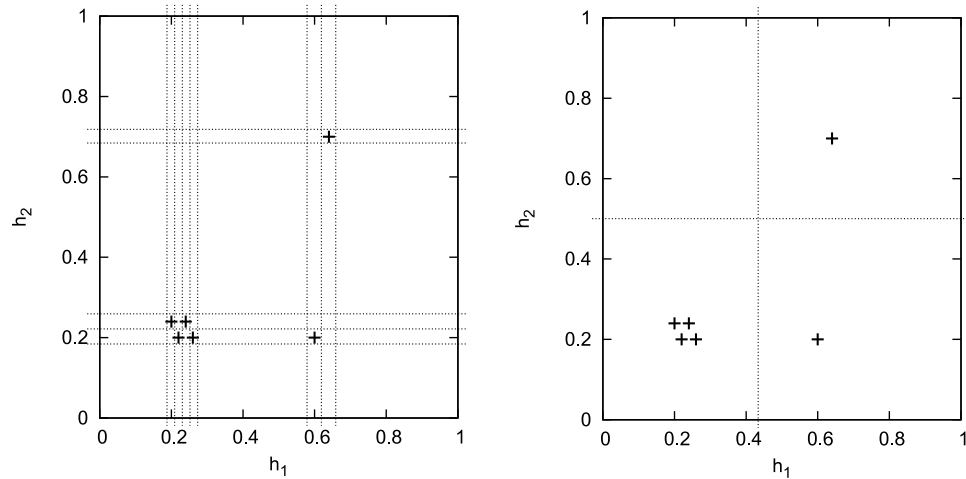


Figure 2.9: An example of NeuroLinear’s hidden unit activation discretization on a 2-dimensional hidden unit activation space. Each cross represents one hidden unit activation vector. The dotted lines indicate where the activation range is divided into intervals. The left figure shows the initial division in which each value is assigned a separate interval. The right figure shows the final result after the intervals have been merged.

A similar approach is taken in NeuroLinear [75] in which the range of each hidden unit’s activation is divided into multiple intervals having different sizes using the Chi2 method [49]. The main idea is that the algorithm starts by assigning each unique hidden unit activation value a very small interval that contains the value,

then repeatedly merges adjacent intervals as long as each grid cell produced by the intervals (as shown in Figure 2.9) maps to exactly one class at the output. This requirement is similar to the requirement in the preceding method in which each hypercube must map to a single class. At the beginning, each unique activation has its own interval so the grid cells are very small; hence each grid cell is trivially mapped to exactly one class at the output. The Chi2 method uses a heuristic to choose the next pair of intervals to be merged given that the resulting grid cells still uniquely map to one class. When no more intervals can be merged, one rule describing the conditions on the input will be extracted for each grid cell having at least one hidden activation vector. Each rule can then be easily made into a rule describing the conditions on the input so that the output is a certain class.

Research in this area has largely focused on learning networks with fewer weights and better ways to express the rules. Most methods use the highly distributed representation produced by standard backpropagation, and a common problem has been the production of fairly large rule sets. Relatively little work has been done on inducing training methods to learn a better hidden layer representation so that *any* rule extraction process becomes more effective. Potentially, a better representation might allow the first stage to extract fewer and more compact regions in hidden activation space, thus leading to a more concise, easier-to-understand set of rules. It is this simple idea that is pursued in this research work.

There are a number of criteria for evaluating rule extraction methods [2]. The two most important ones are classification accuracy and number of rules. First, the rules must at least perform the classification as well as the correspondingly source

network. Second, the number of rules should be kept small so that a human can understand them more easily. Both criteria will be used in evaluating the methods presented in this research.

## 2.4.2 Finite State Machine Extraction from Recurrent Neural Networks

While most past work with feedforward networks has focused on rule extraction, work with recurrent networks has largely focused instead on extracting finite state machines (FSMs) that capture the state transition of a network’s dynamics. Consider the example of a simple recurrent network learning the sequence “*The quick brown fox jumps . . .*” shown earlier in Figure 2.5. Here  $a_{H0}$ ,  $a_{H1}$ ,  $a_{H2}$ , and  $a_{H3}$  are four vectors of context unit activations that give the network a “context” of what it has seen before. Given the context  $a_{H0}$  and the input *The*, the network activation rule deterministically calculates the hidden unit activation vector  $a_{H1}$  and the output vector  $a_{O1}$ . Then,  $a_{H1}$  becomes the context for the next time step. A recurrent neural network is a dynamical system in which the states are the vectors of hidden unit activations and the evolution rule is the activation rule of the neural network.

We can also view this same dynamics as a finite state machine as illustrated in Figure 2.10. When the initial state  $a_{H0}$  receives the input symbol “*the*”, it makes a transition to the state  $a_{H1}$ . In addition,  $a_{H1}$  produces an output symbol “*quick*”. Similarly, at state  $a_{H1}$ , the input symbol “*quick*” makes the transition to state  $a_{H2}$  and produces the output “*brown*”. This is the dynamics of the *Moore machine*, a

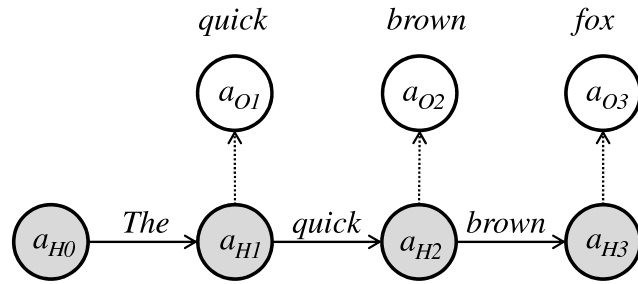


Figure 2.10: A simple recurrent network’s dynamics viewed as a finite state machine. The shaded circles denote the states. The solid arrows denote the transitions while the labels on the arrows denote the input symbol. The dotted arrows denote the output function.

class of FSM. From here on in this dissertation, we will mean this class of FSM when we refer to a FSM.

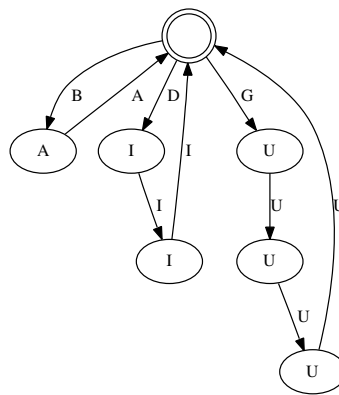


Figure 2.11: Finite state machine for the *badigu* language. Each circle represents one state, and the double circle represents the initial state. Symbols inside circles are the FSM’s outputs (prediction of the next symbol). With this specific grammar, all states are acceptable terminal states. The transitions are shown by the arrows and labeled by their corresponding input symbols.

The example in Figure 2.10 is a simplified case in which only one sequence is learned. In practice, a state  $a_{H0}$  can receive different inputs and can have transitions to multiple states such as in Figure 2.11. This latter figure shows the *Moore machine* for the *badigu* regular language. The regular expression for the language is  $(ba|dii|guuu)^*$ . In other words, it contains strings having only three substrings *ba*,



*dii*, or *guuu*. In comparison with a regular FSM, a Moore machine may not have accept/reject states. The more important difference is that a Moore machine has an output function mapping each state to the output alphabet. Namely, each state has to produce one output symbol. In Figure 2.11, the initial state can receive three different input symbols  $b, d$ , and  $g$ . If the input is  $b$ , the current state is changed to the left most node with the output  $a$ . This is the correct prediction for the next input symbol because an  $a$  must follow a  $b$ . Similarly, if the input is  $d$ , the next state predicts an  $i$ , then another  $i$ .

As a neural network learns a data set consisting of many sequences, there are a very large number of states  $a_{Hi}$ . Fortunately, it is often possible to cluster these states into a small number of clusters so that the transitions between clusters mirror the transitions between their constituent  $a_{Hi}$ 's, and the output symbol of each cluster also mirrors the output symbol of its  $a_{Hi}$ 's. The result is a FSM that closely approximates and expresses the dynamics of the original neural network's temporal data sets. Most symbolic representation extraction from recurrent networks has achieved success using this approach. Since the final result is a FSM, it is often called *finite state machine extraction*. It should be noted that if the data set is generated from an FSM, this method can potentially learn that underlying FSM or an equivalent one. On the contrary, if the underlying language is more complex, such as a context-free grammar, the resulting FSM can only learn an approximate model of the data.

In early work, the hidden unit activation patterns of simple recurrent networks trained on a battery of temporal data generated from known FSMs were studied [10].

It was found that the patterns form clusters in the hidden unit activation space . Moreover, the clusters usually corresponded to the states of the source FSMs. Similar results were also found when simple recurrent networks were trained on context-free grammars [18]. These results support the approach described above and provide the basis for most finite state machine extraction work done subsequently.

The work most relevant to this dissertation is done by Schellhammer et. al. [69]. In this work, FSMs are extracted from simple recurrent networks on a next-word prediction task. The networks were trained on texts derived from a primary school reader. After training, a  $k$ -means algorithm was used to cluster the hidden unit activation patterns. As a result,  $k$  was the number of states of the extracted FSM. There is a trade-off between accuracy and comprehensibility: large values of  $k$  resulted in more accurate but difficult to understand FSMs. On the contrary, smaller  $k$ 's produced simpler FSMs but with less accuracy. Among different values of  $k$  from 6 to 22, 18 produced a relatively simple FSM that is as accurate as a tri-gram model while the tri-gram model is obviously very difficult to understand. A modified approach was taken in [86] in which a large value of  $k$  was used for  $k$ -means, then a symbolic machine reduction algorithm was used on the extracted FSM. In addition, hierarchical clustering was also used as in [1, 68].

Another method used to find clusters is dividing the hidden unit activation space into  $q^n$  equally-sized hypercubes, where  $n$  is the number of hidden units and each unit activation's range is divided into  $q$  equally-sized intervals [24, 58]. While  $q^n$  can be very large, only hypercubes containing activation vectors are assigned one state of the FSM, so the size of the FSM can be kept manageable.

For simple recurrent neural networks, existing approaches for FSM extraction use various methods to partition the state space by different clustering methods and vector quantization with limited success. As with feedforward networks, during learning error backpropagation is free to create any encoding scheme over the hidden units as long as the final error at the output layer is minimized. This again presents a problem for clustering and vector quantization methods because often the hidden layer representations are so complex or distributed that very many clusters are required to partition the space.

A recent survey of rule extraction algorithms for recurrent networks can be found in [37]. Research in this area has largely focused on better clustering and vector quantization methods, but relatively little work has been done on inducing training methods to learn a better hidden layer representation so that *any* finite state machine extraction process become more effective. Potentially, a better representation might allow partitioning the hidden activation space into fewer regions, thus leading to a FSM with fewer states. A few past studies took this approach by forcing hidden layer representations to be binary vectors. For example, in [86] the hidden units' sigmoid activation functions were replaced with threshold functions and a pseudo-gradient learning method was used during training. Although the representation capability of the hidden layer is restricted, experiments showed that the networks can still perform well on some simple data sets. It is unclear whether the same approach can be applied to larger or real-world data sets. Similarly, in [45] training was modified so that a trained network only has weights with values either  $H$  or  $-H$ , and hidden activation vectors have only one component with value 1 while the rest are 0. While

this local hidden layer representation makes clustering very easy because the number of clusters *is* the number of hidden units, the networks trained using this fully local representation method are even more constrained than in [86]. A different approach was used in [12] where an interpolation scheme was adopted to move the hidden activation vectors closer to the center of the clusters. While this avoided the problem of binary vectors, the activation values are no longer computed solely on activation rules, thus compromising the representation capability of the network.

To the best of my knowledge, while echo state networks have achieved much success in recent years, there is only one published work on extracting FSM from ESN [23]. The lack of research in this area is probably caused by the difficulty in extracting information from reservoirs because they have a large number of hidden units. In other words, the hidden unit activation spaces that have to be partitioned have a large number of dimensions. Furthermore, the activation spaces are very distributed because the weights are initialized randomly and not trained. In order to counter the latter problem, the authors in [23] used ESN+ (developed in [5, 22]) which has an equation *assigned* to each weight from the input to the reservoir. It was shown that the same FSM extraction method can extract a much simpler FSM from an ESN+ than from a regular ESN.

## Chapter 3

### Rule Extraction From Feedforward Neural Networks

This chapter begins by introducing a modified error backpropagation learning rule that leads to better separated representations of hidden unit activation patterns in feedforward networks [32, 33]. The main approach to be taken is to augment the usual error function  $E_1$  with a new “error” term named  $E_2$  that increases when the hidden layer activation vectors are closer together. Then, an efficient and local way to compute the gradient of  $E_2$  with respect to each weight is derived. Next, an illustrative example is presented to visualize the working of the  $E_2$  term and its effectiveness.  $E_2$  is then evaluated systematically on five large public artificial and real-world data sets. The results show that  $E_2$  indeed helps to extract simpler rule sets without compromising the standard sum of squared error at the outputs. Finally, two more advanced terms  $E_3$  and  $E_4$  developed from  $E_2$  are presented, along with experimental results that compare them to  $E_2$  and to the popular *C4.5rules* software.

#### 3.1 New Error Term $E_2$

In this chapter we are interested in extracting rules from multilayer feedforward neural networks with one hidden layer as shown in Figure 2.2. The result shown here is adaptable to other kinds of networks and forms a basis of the work in subsequent

chapters. First, a penalty term  $E_2$  that decreases as the hidden unit activation vectors are further apart is introduced:

$$E_2 = -\frac{1}{2} \sum_{p=1}^N \sum_{q=1}^N \sum_{k \in \text{hidden}} (a_{H_k}^p - a_{H_k}^q)^2$$

where  $a_{H_k}^p$  is the activation of the  $k^{\text{th}}$  hidden unit when the  $p^{\text{th}}$  input pattern is presented and  $a_{H_k}^q$  is analogous for the  $q^{\text{th}}$  input pattern. Thus  $E_2$  is the sum over all pairs  $(p, q)$  of the squared Euclidean distance between two hidden activation vectors for the  $p^{\text{th}}$  and  $q^{\text{th}}$  input patterns. The negative sign ensures that when neural network training minimizes the measure  $E_2$ , it will maximize the distances between the hidden layer vectors. When  $p = q$ , only zeroes enter the sum so no special attention is given to that situation.

The new total error function guiding learning is:

$$E = \alpha E_1 + \beta E_2$$

where  $\alpha, \beta > 0, \alpha + \beta = 1$ . Note that the double sum over  $p$  and  $q$  can make  $E_2$  quite large relative to  $E_1$ , so  $\beta$  must be quite small to scale  $E_1$  and  $E_2$  appropriately.

In order to train the network with error backpropagation, we need to compute the components of the gradient of  $E$  given by:

$$\begin{aligned} \frac{\partial E}{\partial w_{ji}} &= \alpha \frac{\partial E_1}{\partial w_{ji}} + \beta \frac{\partial E_2}{\partial w_{ji}} \\ \frac{\partial E}{\partial v_{kj}} &= \alpha \frac{\partial E_1}{\partial v_{kj}} + \beta \frac{\partial E_2}{\partial v_{kj}} \end{aligned}$$

where  $w_{ji}$  is an input-to-hidden weight, and  $v_{kj}$  is a hidden-to-output weight. Of course, the standard terms  $\frac{\partial E_1}{\partial w_{ji}}$  and  $\frac{\partial E_1}{\partial v_{kj}}$  can be computed efficiently as in [26]. We

also have  $\frac{\partial E_2}{\partial v_{kj}} = 0 \forall j, k$  with  $v_{kj}$  being the weight to the  $k^{th}$  output unit from the  $j^{th}$  hidden unit because  $E_2$  does not have any  $v_{kj}$  component.

Now,  $\frac{\partial E_2^p}{\partial w_{ji}}$  can be computed efficiently as follows. Let  $E_2 = \sum_p E_2^p$  where:

$$E_2^p = -\frac{1}{2} \sum_{q=1}^N \sum_{k \in \text{hidden}} (a_{H_k}^p - a_{H_k}^q)^2 \quad (3.1)$$

The gradient of the error  $E_2^p$  with respect to weight  $w_{ji}$  can be calculated as:

$$\frac{\partial E_2^p}{\partial w_{ji}} = \sum_{k \in \text{hidden}} \frac{\partial E_2^p}{\partial a_{H_k}^p} \frac{\partial a_{H_k}^p}{\partial w_{ji}}$$

Because  $\frac{\partial a_{H_k}^p}{\partial w_{ji}} = 0$  with  $k \neq j$ , we have:

$$\begin{aligned} \frac{\partial E_2^p}{\partial w_{ji}} &= \frac{\partial E_2^p}{\partial a_{H_j}^p} \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\ &= -\frac{1}{2} \sum_{q=1}^N \sum_{k \in \text{hidden}} \frac{\partial (a_{H_k}^p - a_{H_k}^q)^2}{\partial a_{H_j}^p} \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\ &= -\sum_{q=1}^N \sum_{k \in \text{hidden}} (a_{H_k}^p - a_{H_k}^q) \left( \frac{\partial a_{H_k}^p}{\partial a_{H_j}^p} - \frac{\partial a_{H_k}^q}{\partial a_{H_j}^p} \right) \frac{\partial a_{H_j}^p}{\partial w_{ji}} \end{aligned}$$

With  $k \neq j$ , we have  $\frac{\partial a_{H_k}^p}{\partial a_{H_j}^p} \frac{\partial a_{H_j}^p}{\partial w_{ji}} = 0$  and  $\frac{\partial a_{H_k}^q}{\partial a_{H_j}^p} \frac{\partial a_{H_j}^p}{\partial w_{ji}} = 0$  because  $a_{H_k}^p$  and  $a_{H_k}^q$  do not have  $w_{ji}$  components. So:

$$\begin{aligned} \frac{\partial E_2^p}{\partial w_{ji}} &= -\sum_{q=1}^N \left[ (a_{H_j}^p - a_{H_j}^q) \left( \frac{\partial a_{H_j}^p}{\partial a_{H_j}^p} - \frac{\partial a_{H_j}^q}{\partial a_{H_j}^p} \right) \right] \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\ &= -\sum_{q=1}^N \left[ (a_{H_j}^p - a_{H_j}^q) \left( 1 - \frac{\partial a_{H_j}^q}{\partial a_{H_j}^p} \right) \right] \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \end{aligned}$$

For  $p \neq q$ , we can assume that  $a_{H_j}^q$  does not change when we process pattern  $p$ .

This leads to  $\frac{\partial a_{H_j}^q}{\partial a_{H_j}^p} = 0$  or  $(1 - \frac{\partial a_{H_j}^q}{\partial a_{H_j}^p}) = 1$ . When  $p = q$ , we have  $(a_{H_j}^p - a_{H_j}^q) = 0$ . So:

$$\begin{aligned}
\frac{\partial E_2^p}{\partial w_{ji}} &= - \sum_{q=1, q \neq p}^N (a_{H_j}^p - a_{H_j}^q) \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\
&= -((N-1)a_{H_j}^p - \sum_{q=1, q \neq p}^N a_{H_j}^q) \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\
&= -(Na_{H_j}^p - \sum_{q=1}^N a_{H_j}^q) \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\
&= -N(a_{H_j}^p - \overline{a_{H_j}}) \frac{\partial a_{H_j}^p}{\partial w_{ji}}
\end{aligned} \tag{3.2}$$

with  $N$  being the number of training patterns (a constant) and  $\overline{a_{H_j}}$  being the average activation of the  $j^{th}$  hidden unit over all input samples. As with the usual backpropagation derivation using the logistic transfer function, we have  $\frac{\partial a_{H_j}^p}{\partial w_{ji}} = a_{H_j}^p(1 - a_{H_j}^p)x_i^p$ . Thus,

$$\frac{\partial E_2^p}{\partial w_{ji}} = -N(a_{H_j}^p - \overline{a_{H_j}})a_{H_j}^p(1 - a_{H_j}^p)x_i^p \tag{3.3}$$

It is remarkable that when computing  $\frac{\partial E_2}{\partial w_{ji}}$  for the  $p^{th}$  input sample, besides looking at the activation of the  $j^{th}$  hidden unit and the  $i^{th}$  input unit as is done with the usual backpropagation training, we only need one more value  $\overline{a_{H_j}}$  which can be computed and stored locally at the  $j^{th}$  hidden unit. This local property is highly desired in neural network training.

### 3.2 Rule Extraction Algorithm

The same rule extraction algorithm is used for both the experimental condition ( $E = E_1 + E_2$ ) and the control condition ( $E = E_1$ , which is basic error



backpropagation). The outline of the rule extraction algorithm in both cases is as follows:

- Step 1: Train the network.
- Step 2: Cluster the individual hidden unit activation values.
- Step 3: Extract rules explaining the output in terms of clustered hidden unit activation values.
- Step 4: Prune unnecessary weights connecting the input layer to the hidden layer.
- Step 5a: If the data consists of continuous attributes: generate rules in the form of linear inequalities on inputs for hidden unit activation cluster values.
- Step 5b: If the data consists of binary attributes: generate decision tree rules for each hidden unit activation value cluster using the program *C4.5rules* [61].

Steps 1 to 4 are similar to past rule extraction methods in [50, 75, 82] but differ in a number of ways: (1) a different error function that puts a strong emphasis on hidden unit activation patterns' separability rather than pruning [34], (2) a different learning algorithm, and (3) *C4.5rules* for extracting the simplified hidden-output mapping is used. Regardless of these differences from past work, the same rule extraction procedure is applied in comparing standard backpropagation ( $E_1$ ) versus the enhanced method ( $E_1 + E_2$ ).

RPROP [64] (resilient backpropagation) is used in step 1. It is an improved backpropagation learning algorithm that trains networks faster by adjusting the

weight update based on the direction of the gradient instead of the magnitude of the derivatives. It also requires few training parameters. The error function is augmented with the popular *weight decay* term  $E_d = \lambda \sum_j \sum_i w_{ji}^2$  to prevent weights from getting too large [46]. Weight decay has been shown to improve the generalization performance of neural networks (regularization). This term is used implicitly in both control and experimental simulations in this work.

The logistic hidden unit activation values are in the range  $(0, 1)$ . After training, the values experienced at each hidden unit can be clustered together into disjoint intervals  $[0, r_1), [r_1, r_2), \dots, [r_n, 1]$  such that we only need to know which interval the hidden activation values are in to determine the class label of training instances. The Chi2 discretization algorithm [49] is used to cluster the activation values. This algorithm first makes one interval for each activation value, sorts the intervals in increasing order, and then uses  $\chi^2$  statistics to determine which pair of adjacent intervals should be merged next. Some pairs of intervals are not allowed to be merged because that would affect the classification accuracy. For example, when there are two training examples  $p$  and  $q$  with different class labels such that  $a_{H_j}^p$  is in the first interval and  $a_{H_j}^q$  is in the second interval, the two intervals cannot be merged as we no longer could determine which class label to assign knowing only the interval that the  $j^{\text{th}}$  hidden unit is in.

Step 3 extracts rules having the form  $(H_{i_1} = l_1, H_{i_2} = l_2, \dots) \rightarrow \text{class} = c_j$  which means that *if the  $i_1^{\text{th}}$  hidden unit's activation value is in interval  $l_1$  and the  $i_2^{\text{th}}$  hidden unit's activation value is in interval  $l_2$  and ... then classify the sample as class  $c_j$ .* Rule extraction is done using *C4.5rules*. This extraction step is also a base

step in other rule extraction algorithms. It is very important to have fewer rules at this point because the number of rules here strongly affects the final number of rules that ultimately specify the input-output relationship.

The novelty of this method is in the use of the new error term that “pushes” the hidden activation vectors away from each other, so that (as seen below) their component values tend to cluster toward the two ends of the interval  $[0, 1]$ . This in turn results in many hidden units having values clustered into only two intervals  $[0, r)$ ,  $(r, 1]$ . Having such simple binary splits is highly desirable for making fewer and simpler rules.

Step 4 prunes the network by removing unnecessary connections from the input units to the hidden units. Pruning reduces the number of weights, thus making rules with continuous inputs simpler. It also helps in extracting simpler rules for discrete inputs. A simple pruning scheme that greedily removes weights in increasing order of their magnitudes and stops when the accuracy in the validation set drops below a specified threshold is used.

Step 5 is different for continuous and discrete attributes. If the inputs consist of continuous attributes, rules that depend upon when the  $j^{th}$  hidden unit activation is in an interval  $[r_1, r_2)$  can be generated directly as follows:

$$\begin{aligned}
 r_1 &\leq a_{H_j} < r_2 \\
 r_1 &\leq \sigma(w_{j1}x_1 + w_{j2}x_2 + \dots + w_{jn}x_n) < r_2 \\
 \sigma^{-1}(r_1) &\leq w_{j1}x_1 + w_{j2}x_2 + \dots + w_{jn}x_n < \sigma^{-1}(r_2)
 \end{aligned} \tag{3.4}$$

Not all  $x_i$  are present in each rule since unnecessary weights were already pruned in step 4. Every hidden-output rule produced in step 3 is a conjunction of which interval each hidden unit value must be in, so the terms in the conjunctions can easily be pruned with the above inequality to produce rules explaining the output classification directly from the input.

For problems with discrete inputs, *C4.5rules* [61] is used to generate one set of rules for each hidden unit's activation. The rules tell the conditions on inputs that would make a hidden unit activation value fall into one interval. For example, a rule for the  $j^{th}$  hidden unit has the form:

$$(x_{i_1} = b_1, x_{i_2} = b_2, \dots) \rightarrow a_{H_j} \in k^{th} \text{ interval}$$

Because the rules in this step are only concerned with which interval a hidden unit activation is in, there are usually very few simple rules. Each term  $H_i = l_i$  in step 3 is then replaced with the input-hidden layer rules. Next, the boolean expressions are simplified, and the duplicates are removed. The final result of these steps is rules explaining the classification directly from the input values.

If, as sometimes occur, both continuous and discrete input attribute exist, the common approach is to discretize all the continuous attributes and then any method that works with discrete attributes can be used. Alternatively, there is also a little work on extracting rules directly from mixed discrete and continuous data sets [62, 71]. In this thesis, we will focus on data sets with either continuous or discrete attributes.

The important distinction between this rule extraction method and *C4.5rules* is that *C4.5rules* generates a *single* decision tree/rule set directly from the data set

while this method generates *two* intermediate rule sets and combines them. The first rule set captures the relationship between the hidden activation intervals and the output. It is usually very simple with very few rules because of the improved hidden activation patterns. The second rule set explains the relationship between the hidden activation intervals and the input data. These rules are also simple because they are concerned with specific hidden activation intervals. It should also be noted that methods other than *C4.5rules* could be used to extract these intermediate rules.

### 3.3 An Illustrative Example

In this section, the hidden unit encodings learned by the neural network for the *waveform* problem [3] are used to illustrate this approach. The *waveform* data set consists of 5000 instances of waves. Each wave is characterized by 21 continuous inputs with noise. The problem is to classify these waves into one of three classes.

First the inputs are standardized using z-values [15]. The 5000 instances are divided randomly into three sets: 4000 for training, 500 for testing, and 500 for validation. A three layer feedforward neural network with 4 hidden units is trained on the data.

After training, the hidden unit activations  $(a_{H_1}^p, a_{H_2}^p, a_{H_3}^p, a_{H_4}^p)$  of the four hidden units for each instance  $p$  can be calculated. This vector is an encoding of the 21-dimension vector input. We are interested in how these 4 dimensional vectors are arranged in the four dimensional space when the new error term  $E_2$  is used and when it is not. Without losing generality, 3 of the 4 dimensions are chosen arbitrarily

in order to visualize the locations of these vectors in the following representative example from one of the runs.

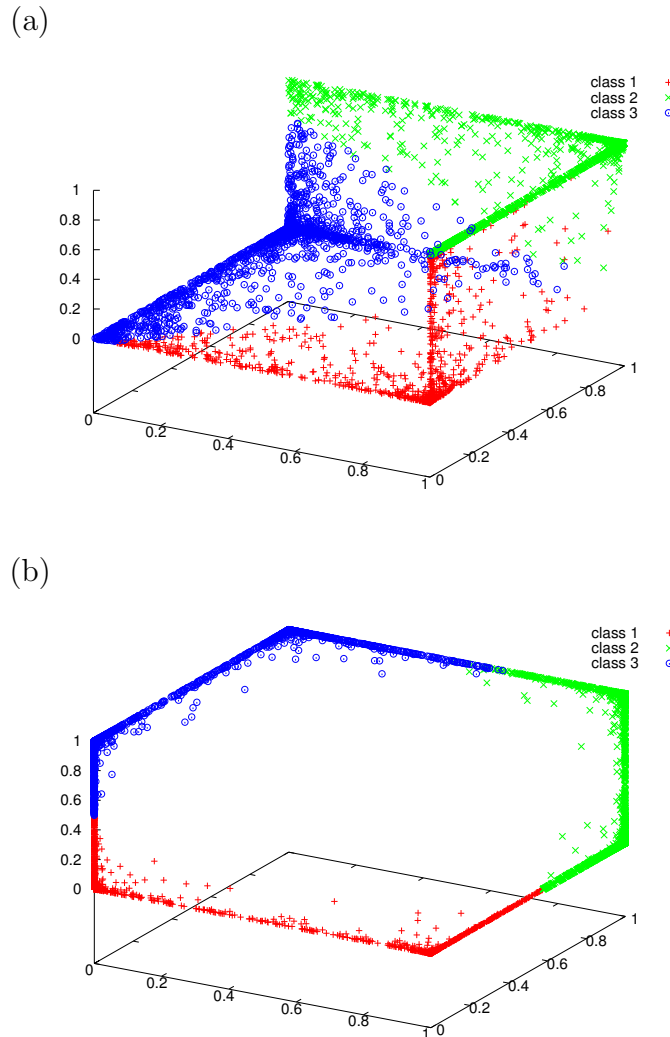


Figure 3.1: Input patterns throughout hidden unit activation space for the *waveform* problem after training with (a) regular backpropagation ( $E = E_1$ ) versus (b) the same error function but augmented to include the new error term ( $E = E_1 + E_2$ ).

Figure 3.1a shows the training data patterns plotted in hidden unit activation space after the network has been trained with the regular sum of squared error function  $E_1$  used in standard backpropagation. It can be seen that the vectors are

clustered into 3 groups corresponding to the 3 classes. While many of them are in the corners or along the edges, quite a number are spread out over the interior instead and close to vectors in other classes. These vectors make it hard to draw planes separating the clusters; in other words, more rules would be expected to be needed to explain the hidden activation-output activation relationship.

What we want to do is to push these vectors further away from each other during learning so that it is easier to separate them. This is done with the help of the new error term  $E_2$  that penalizes having vectors close together. The effect of the training with this new error term is shown in Figure 3.1b. The three clusters are more visible as they move closer to the edges and three corners, and fewer vectors are in the interior. Clearly, the augmented learning procedure (Figure 3.1b) pushes the interior hidden encodings for input patterns in different classes further from each other than with standard backpropagation (Figure 3.1a) in this example.

### 3.4 Experimental Results

The goal of this evaluation is to compare the number of rules extracted from a trained error backpropagation network when  $E_2$  is included in the error function (experimental condition) versus the number when  $E_2$  is not included (control condition of  $E_1$  alone, i.e., standard backpropagation). It is not, however, to show that the extraction method presented in this chapter is superior to existing ones, but to show that training with  $E_2$  produces better separated encoding at the hidden layer, and thus would improve the performance of existing rule extraction methods. The

effectiveness of the rule extraction method is evaluated on five data sets having more than 1000 instances selected arbitrarily from the UCI Machine Learning Repository [3]: the *waveform*, *yeast*, *image-segmentation*, *nursery* and *splice* problems. These are large and difficult data sets with many attributes and classes.

Table 3.1: Data Sets Used for Evaluation

data set	no.attrs	no.class	no.instances	input
waveform	21	3	5000	continuous
yeast	8	10	1484	continuous
imgseg	18	7	2310	continuous
nursery	8	5	1296	discrete
splice	60	4	3190	discrete

Table 3.1 shows the characteristics of the five data sets used in the experiments. Three of these have continuous inputs: the *waveform problem* involves classifying waves into one of three classes based on 21 noisy features, the *yeast problem* is a protein localization site determination problem, and the *image segmentation* problem classifies pixels in images using 17 continuous value features. The other two data sets have discrete/categorical inputs. Data set *nursery* is an application ranking database for admission to nursery schools. Applications are classified into 5 classes indicating how strongly the applicant is recommended. The 8 categorical attributes are encoded into 25 binary input units using nominal encodings: a category with  $m$  unique values is encoded as  $m$  binary input units, with only one bit corresponding to the value being on. A set of 1296 (10%) instances were chosen randomly from 12961 instances in the original data set to shorten the running time. The *splice* problem is to recognize the boundary between exons and introns in a DNA sequence. The 60



attributes, each representing one nucleotide {A,T,G,C}, are encoded into 240 binary input units.

For each data set, the settings for both the experimental and control runs are as follows:

1. Ten-fold cross validation scheme: each data set is split randomly into 10 subsets of approximately equal size. Eight subsets were used for training, one was used for validation and one for measuring the accuracy of the extracted rules. The procedure is repeated 10 times, where each time one different subset was used as the testing set. Each experiment is also run 10 times with different random initial weights. The reported number of rules and accuracy are averages over all 100 runs. Having so many runs ensures that any improvement comes from the method and not just by chance. For the *image-segmentation* data set, which was already divided into training and test sets by the data donor, the original training and test sets are merged into one single set so that ten-fold cross validation can be used as with other data sets.
2. In each run, the experiments with and without the new error terms have the same starting point - i.e., matched initial weights and data set division to make comparison maximally compatible. Paired t-tests are used to evaluate the results.
3. Weight decay rate was set to 0.00001.
4.  $\beta$  was set to 0.00001 for *waveform* and *nursery*, 0.00005 for *yeast*, 0.0003 for the *splice* problem, and 0.00007 for the *image-segmentation* problem. To determine

these values, a few pilot runs were done with each data set where  $\beta$  is initialized to 0 (the control case of using  $E_1$  alone) and slowly increased until the accuracy rate dropped more than 5% compared to the control case. This determined the values of  $\beta$  that were used for the 100 runs reported in the experimental results (and also for  $\alpha$  since  $\alpha = 1 - \beta$ ). The contribution of  $\beta E_2$  is much more significant than it looks. At the end of training,  $E_1$  is of the order of  $10^2$  because it is a sum of over 1000 squared errors from all output units.  $E_2$  is of the order of  $10^6$  because it is the sum over *all pairs* of Euclidean distances. These choices of  $\beta$  make the contribution of  $E_2$  about 5%  $\sim$  70% of  $E$  for the five problems.

5. The number of output units corresponds to the number of classes in the data. When doing a classification, the class whose output unit has the highest activation value is chosen as the class for the instance.
6. Continuous input attribute values were standardized with z-value scores [15].
7. RPROP with weight backtracking was set up to run for a maximum of 400 epochs or until validation error goes up for 10 consecutive epochs.  $\eta^+$  and  $\eta^-$  are set to 1.25 and 0.5, respectively. The network with highest validation accuracy was saved for subsequent rule extraction.

Table 3.2 and Figure 3.2 show the effect of the new error term  $E_2$  on the network's average testing error ( $E_1/N$ ) and the average distances among hidden unit activation vectors ( $-2E_2/N^2$ ) where  $N$  is the number of data instances. A network's

Table 3.2: Regular versus Modified Backpropagation (Averaged over 100 Runs)

data set	$E_1/N$		$-2E_2/N^2$	
	regular	new	regular	new
waveform	0.128	0.149	1.632	1.949 (+19%)
yeast	0.384	0.393	0.706	1.190 (+69%)
imgseg	0.085	0.107	1.979	2.684 (+35%)
nursery	0.075	0.095	1.050	1.125 (+7%)
splice	0.068	0.073	0.971	1.175 (+21%)

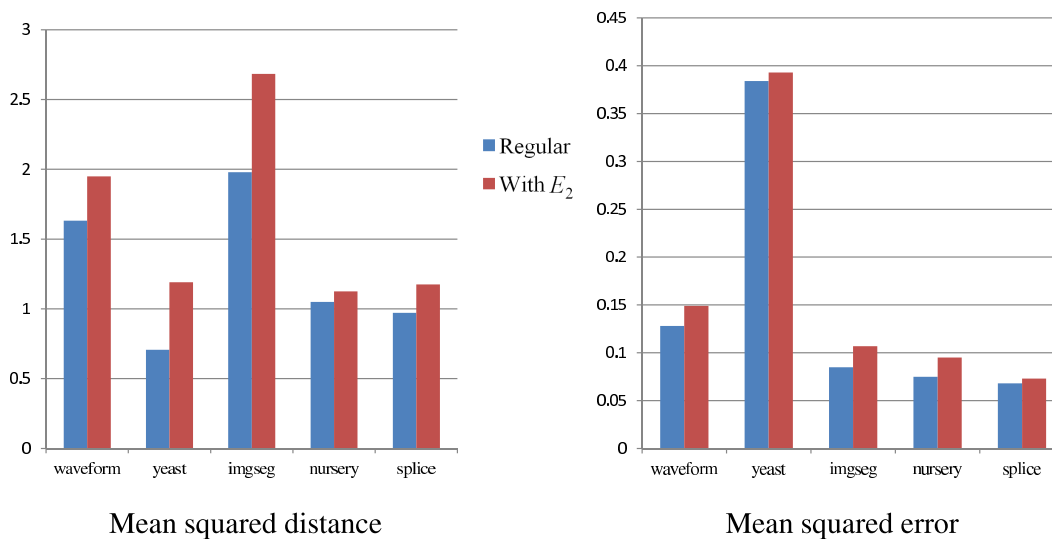


Figure 3.2: Mean squared distance and mean squared error of networks trained by regular and modified backpropagation.

average testing error and average squared distance between hidden activation vectors are shown after training with regular backpropagation versus the new combined error term. Since  $E_2$  is 1/2 of the sum of squared distances between each pair (total  $N^2$  pairs) of activations,  $-2E_2/N^2$  is the average squared distance between each pair. The *regular* columns show results when the networks were trained with “regular” backpropagation using  $E = E_1$ . The *new* columns show the results with  $E = E_1 + E_2$ . These data show that activation pattern distances were increased up to 69% with only a small change in network errors. The small change in  $E_1$  backs the hypothesis that it is possible to make error backpropagation learn a different encoding that satisfies other criteria (smaller  $E_2$ ) while still maintaining the network’s accuracy. Modified backpropagation was able to learn an encoding with increased pattern separation at the hidden layer that had higher total squared distances between the hidden unit activation vectors while still maintaining near minimum error at the output layer. The choice of  $\beta$  has a strong impact on the accuracy and  $E_2$ .

Figure 3.3 shows the values of (a)  $E_1/N$  and (b)  $-2E_2/N^2$  during one training run on the *waveform* data set using error backpropagation with the regular error function  $E_1$  and with the new error term  $E_2$ . The training error was slightly higher when trained with  $E = \alpha E_1 + \beta E_2$ . This is expected because error backpropagation has to minimize both terms in this latter case. But the change is very small and not enough to affect the overall classification accuracy significantly. Figure 3.3(b) shows the average squared distance between hidden unit activation pairs  $-2E_2/N^2$ . Training quickly increases the distance in both cases, but significantly more when

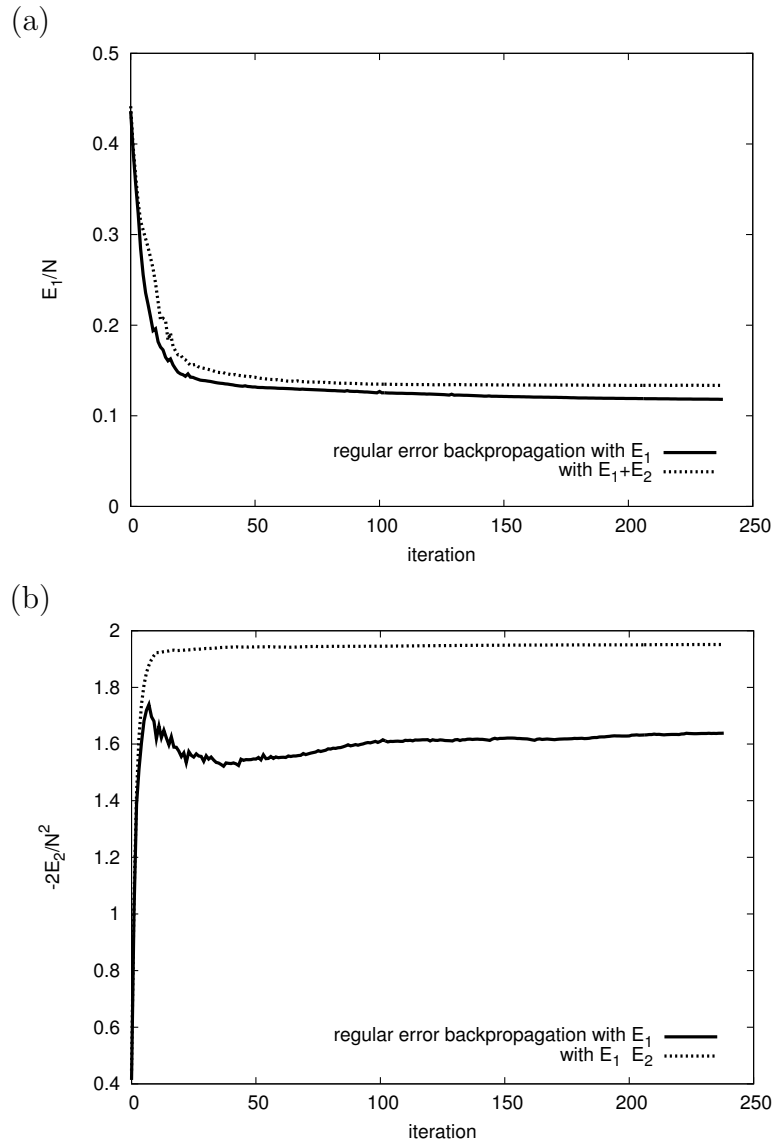


Figure 3.3: A typical plot of (a) average network error  $E_1/N$  and (b) average hidden layer activation pattern separation  $-2E_2/N^2$  during network training.

trained with  $E = \alpha E_1 + \beta E_2$ .

Table 3.3: Means ( $\sigma$ ) of Accuracy and Number of Rules over 100 Runs

Data set	No. of rules			Rule accuracy	
	$E_1$	$E_1 + E_2$	reduced	$E_1$	$E_1 + E_2$
waveform	70.12 (26.89)	14.30 (13.56)	80%	85.08% (1.96)	85.19% (2.04)
yeast	90.17 (23.09)	51.37 (18.23)	43%	51.55% (4.21)	51.40% (4.37)
imgseg	38.34 (9.27)	32.02 (7.37)	16%	91.58% (2.33)	89.29% (3.06)
nursery	192.42 (95.34)	41.85 (43.85)	78%	88.55% (4.01)	89.33% (2.86)
splice	90.21 (84.42)	78.66 (54.49)	13%	90.19% (3.81)	89.85% (4.09)

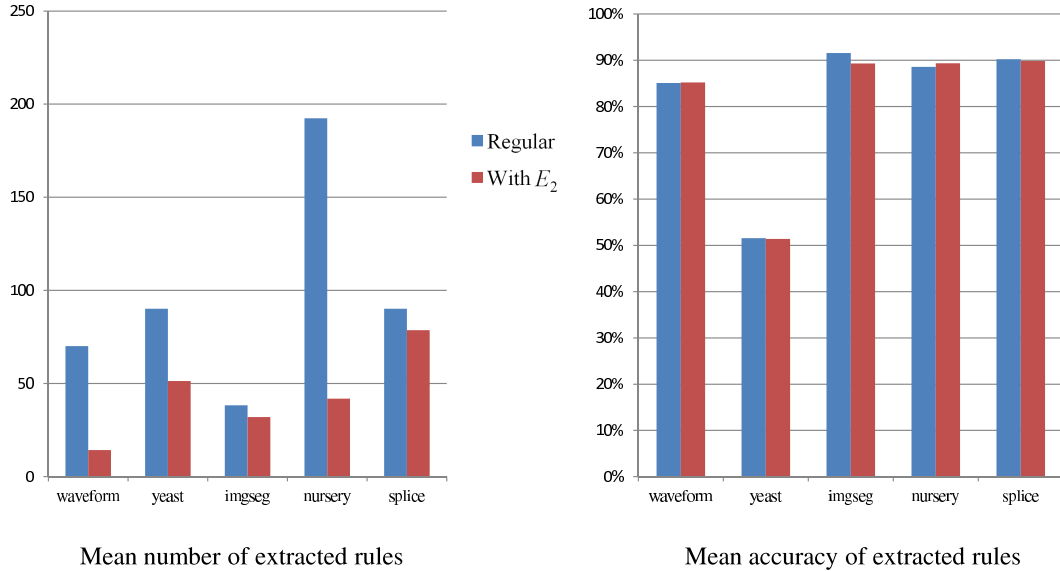


Figure 3.4: Mean number and accuracy of rules extracted from networks trained by regular and modified backpropagation.

Table 3.3 and Figure 3.4 present the experimental results concerning rule extraction. The new error term helped reduce the number of rules significantly, at least 13% for the *splice* problem and up to 80% for the *waveform* problem. The new, smaller sets of rules also have roughly the same classification rates as the ones produced without the new error term (rightmost columns of Table 3.3). Note that

the accuracy rate for the *yeast* problem is quite low, but it is still comparable to the best published results of 54% in [29]. The reason for such a low accuracy rate is that the data set is very difficult with 10 classes unevenly distributed.

Table 3.4: P-value of t-tests Comparing  $E_1$  and  $E_1 + E_2$

data set	avg. no. of rules	accuracy rates
waveform	$5.4 \times 10^{-35}$	0.4818
yeast	$4.7 \times 10^{-25}$	0.7179
imgseg	$1.9 \times 10^{-06}$	$6.4 \times 10^{-9}$
nursery	$4.3 \times 10^{-28}$	0.0456
splice	0.243	0.5574

Paired t-tests are used to determine whether the reduction in numbers of rules and the change in accuracy caused by training with  $E_1$  versus  $E_1 + E_2$  are significant, using a standard significance level 0.05. Bonferroni correction [20] for 10 tests requires significance to be defined as  $p < 0.05/10 = 0.005$ . Statistically significant changes are printed in italics in Table 3.3. Corresponding p-values are shown in Table 3.4. The reduction in numbers of rules is significant in all cases ( $p$  from  $5.4 \times 10^{-35}$  to  $1.9 \times 10^{-6}$ ) except for the *splice* problem ( $p = 0.243$ ). The change in accuracy is *not* significant in all cases ( $p$  from 0.45 to 0.71) except for the *image-segmentation* problem ( $p = 6.4 \times 10^{-9}$ ). The tests confirmed that  $E_2$  reduced the number of rules without degrading accuracy.

More significantly, the best among 100 runs in the experiments were able to extract even smaller rule sets than the averages described above. Rule extraction using  $E_2$  extracted only 5 rules explaining the classification of 5000 *waveform* data

instances with 88% accuracy rate, 19 rules for the *yeast* data set with 57% accuracy, 17 rules for the *image-segmentation* data set with 90% accuracy, 15 rules for the *nursery* data set with 93% accuracy, and 14 rules for the *splice* data set with 94% accuracy. These are better than the best numbers of rules using  $E_1$ : 14, 49, 19, 15, and 19 respectively.

### 3.5 Class Label-Aware Separation

Since  $E_2$  incorporates the sum of distances between all pairs of hidden unit activations, its effect is to push every activation pattern away from the rest. Such an approach ignores class labels, and this omission suggests another more targeted strategy. If one could take into account the class labels of the training data, then just the activation patterns of instances from *different* classes could be pushed apart, while instead the activations of instances from the *same* class could be treated differently, i.e., they could be pushed closer to one another. Potentially such an approach could be even more effective in lowering the number of rules generated. Therefore, two new penalty terms  $E_3$  and  $E_4$  are proposed:  $E_3$  penalizes hidden unit activation vectors *from different classes* having small Euclidean distances, while  $E_4$  penalizes hidden unit activation vectors *from the same class* having big Euclidean distances. More specifically,  $E_3$  and  $E_4$  are given by:

$$E_3 = -\frac{1}{2} \sum_{p=1}^N \sum_{\substack{q=1 \\ \text{class}(q) \neq \text{class}(p)}}^N \sum_{k \in \text{hidden}} (a_{H_k}^p - a_{H_k}^q)^2$$

$$E_4 = \frac{1}{2} \sum_{p=1}^N \sum_{\substack{q=1 \\ \text{class}(q) = \text{class}(p)}}^N \sum_{k \in \text{hidden}} (a_{H_k}^p - a_{H_k}^q)^2$$



It is important to note the negative sign in  $E_3$  and its absence in  $E_4$ . Minimization of  $E_3$  and  $E_4$  *increases* the distances of hidden activation vectors from different classes and *decreases* the distances of activations from the same class.

### 3.5.1 Learning Rules

As with  $E_2$ , we need  $\frac{\partial E_3^p}{\partial w_{ji}}$  and  $\frac{\partial E_4^p}{\partial w_{ji}}$  in order to compute the weight change for gradient descent. Computing  $\frac{\partial E_3^p}{\partial w_{ji}}$  follows the same derivation as with  $\frac{\partial E_2^p}{\partial w_{ji}}$ , until Equation 3.2. The only difference is that the sum is only over  $q$ 's that are in a different class from  $p$ . Let  $C(p)$  be the set of training patterns having the same class as  $p$ . We have:

$$\begin{aligned} \frac{\partial E_3^p}{\partial w_{ji}} &= - \sum_{q \notin C(p)} (a_{H_j}^p - a_{H_j}^q) \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\ &= - \left( \sum_{q \notin C(p)} a_{H_j}^p - \sum_{q \notin C(p)} a_{H_j}^q \right) \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \end{aligned} \quad (3.5)$$

Consider the left factor of Equation 3.5:

$$\begin{aligned} &\left( \sum_{q \notin C(p)} a_{H_j}^p - \sum_{q \notin C(p)} a_{H_j}^q \right) \\ &= (N - N_{C(p)})a_{H_j}^p - \left( \sum_{q=1}^N a_{H_j}^q - \sum_{q \in C(p)} a_{H_j}^q \right) \\ &= Na_{H_j}^p - N_{C(p)}a_{H_j}^p - N\overline{a_{H_j}} + N_{C(p)}\overline{a_{H_j}^{C(p)}} \\ &= N(a_{H_j}^p - \overline{a_{H_j}}) - N_{C(p)}(a_{H_j}^p - \overline{a_{H_j}^{C(p)}}) \\ &= N(a_{H_j}^p - \overline{a_{H_j}}) - N_{C(p)}(a_{H_j}^p - \overline{a_{H_j}^{C(p)}}) \end{aligned} \quad (3.6)$$

with  $N_{C(p)}$  being the number of training patterns in the same class as  $p$  and  $\overline{a_{H_j}^{C(p)}}$  is the average activation of the  $j^{th}$  hidden unit when patterns in  $C(p)$  are presented at

the input layer. Substituting Equation 3.6 to Equation 3.5 gives:

$$\begin{aligned} \frac{\partial E_3^p}{\partial w_{ji}} &= \left( N(a_{H_j}^p - \overline{a_{H_j}}) - N_{C(p)}(a_{H_j}^p - \overline{a_{H_j}^{C(p)}}) \right) \\ &\quad \times a_{H_j}^p (1 - a_{H_j}^p) x_i^p \end{aligned}$$

Similarly, we can compute  $\frac{\partial E_4^p}{\partial w_{ji}}$  as

$$\begin{aligned} \frac{\partial E_4^p}{\partial w_{ji}} &= \sum_{q \in C(p)} (a_{H_j}^p - a_{H_j}^q) \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\ &= \left( \sum_{q \in C(p)} a_{H_j}^p - \sum_{q \in C(p)} a_{H_j}^q \right) \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\ &= \left( N_{C(p)} a_{H_j}^p - N_{C(p)} \overline{a_{H_j}^{C(p)}} \right) \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\ &= \left( N_{C(p)} (a_{H_j}^p - \overline{a_{H_j}^{C(p)}}) \right) \times \frac{\partial a_{H_j}^p}{\partial w_{ji}} \\ &= \left( N_{C(p)} (a_{H_j}^p - \overline{a_{H_j}^{C(p)}}) \right) a_{H_j}^p (1 - a_{H_j}^p) x_i^p \end{aligned}$$

Computing  $\frac{\partial E_3^p}{\partial w_{ji}}$  and  $\frac{\partial E_4^p}{\partial w_{ji}}$  is also local because it requires only local information to be stored at each hidden unit: the average activation and number of examples for each class. However, unlike with  $E_2$ , hidden units must also know the target class of each instance. This can be done by backpropagating the class label from the output layer to the hidden layer together with the error signal.

### 3.5.2 Experimental Results

The purpose of this second set of experiments is to evaluate the effectiveness of the new error terms  $E_3$  and  $E_4$  on the same five large data sets described earlier. The results are compared with regular error backpropagation, error backpropagation with  $E_2$ , and with *C4.5rules*.

Table 3.5: Means ( $\sigma$ ) of Accuracy and Number of Rules over 100 Runs

data set	waveform		yeast		imgseg	
	#rules	accuracy	#rules	accuracy	#rules	accuracy
$E_1$	70.12 (26.89)	85.08% (1.96)	90.17 (23.09)	51.55% (4.21)	38.34 (9.27)	91.58% (2.33)
$E_1 \& E_2$	14.30 (13.56)	85.19% (2.04)	51.37 (18.23)	51.40% (4.37)	32.02 (7.37)	89.29% (3.06)
$E_1 \& E_3 \& E_4$	8.79 (3.77)	85.71% (2.34)	52.52 (17.67)	51.75% (4.43)	26.12 (8.18)	91.86% (2.11)
C4.5rules	77.50 (8.50)	77.30% (1.63)	36.50 (4.32)	59.22% (5.15)	30.00 (1.80)	95.70% (1.00)

data set	nursery		splice	
	#rules	accuracy	#rules	accuracy
$E_1$	192.42 (95.34)	88.55% (4.01)	90.21 (84.42)	90.19% (3.81)
$E_1 \& E_2$	41.85 (43.85)	89.33% (2.86)	78.66 (54.49)	89.85% (4.09)
$E_1 \& E_3 \& E_4$	29.15 (21.08)	89.93% (2.41)	26.92 (14.63)	90.93% (4.19)
C4.5rules	71.49 (6.03)	91.29% (2.74)	39.90 (3.67)	94.33% (1.18)

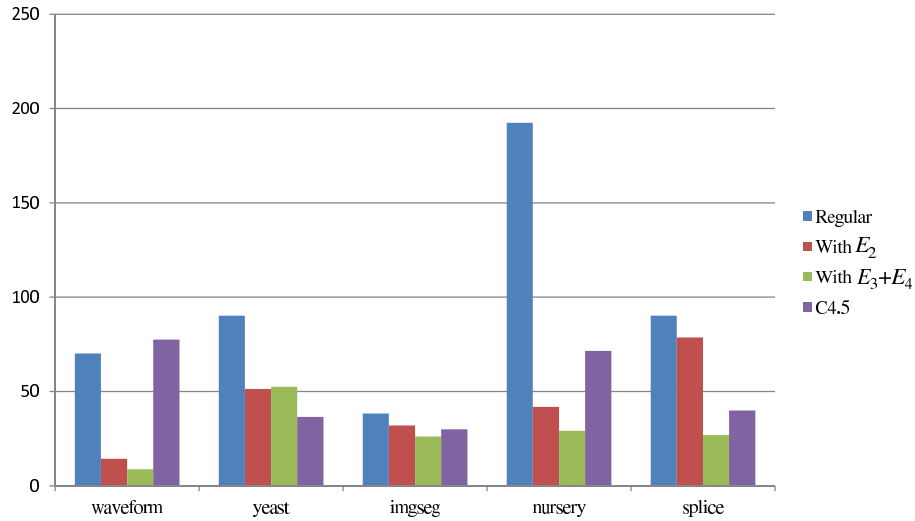


Figure 3.5: Mean number of rules extracted from networks trained by regular backpropagation,  $E_1 + E_2$ ,  $E_1 + E_3 + E_4$ , and mean number of rules learned by *C4.5rules*.

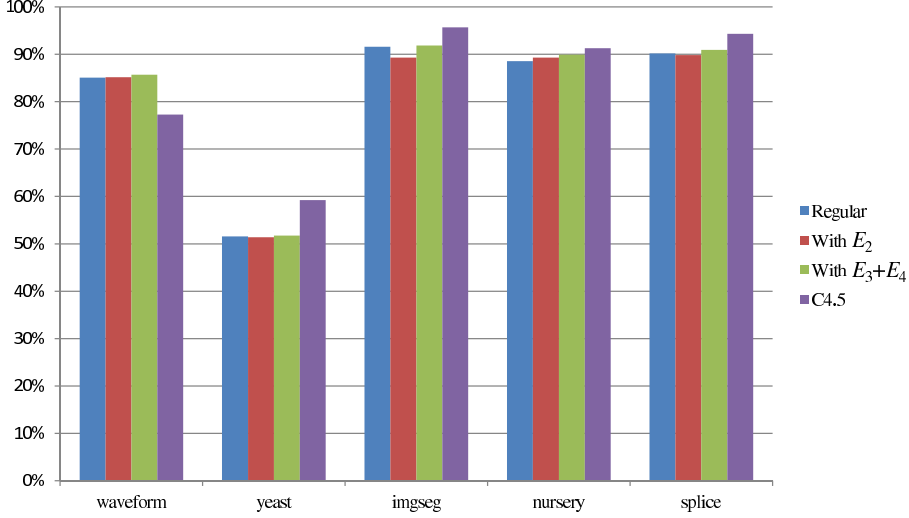


Figure 3.6: Mean accuracy of rules extracted from networks trained by regular backpropagation,  $E_1 + E_2$ ,  $E_1 + E_3 + E_4$ , and mean accuracy of rules learned by *C4.5rules*.

As described earlier in Section 3.4, the results of rule extraction using  $E = \alpha E_1 + \beta E_2$  already show a clear improvement over  $E = E_1$  (summarized in rows  $E_1$  and  $E_1 + E_2$  of Table 3.5). The results with  $E_3$  and  $E_4$  are even better for four out of five data sets (see Table 3.5, Figures 3.5 and 3.6). The numbers of rules for the *waveform* and *splice* data sets are reduced further by 40%, the number of rules for the *nursery* data set is reduced further by 31%, and the number of rules for the *image-segmentation* data set is reduced further by 19%, with no significant change in classification accuracy rates. For the fifth data set *yeast*, the result is the same as with  $E_2$ .

Paired t-tests are used to determine whether the reduction in numbers of rules and the change in accuracy caused by training with  $E_1$  versus  $E_1 + E_2$  and  $E_1 + E_2$  versus  $E_1 + E_3 + E_4$  are significant. Bonferroni correction for 20 tests (5 data sets  $\times$  2 settings  $\times$  2 criteria) requires  $p < 0.05/20 = 0.0025$ . Statistically significant

Table 3.6: P-value of t-tests on Number of Rules

data set	$E_1$ vs $E_1 + E_3 + E_4$	$E_1 + E_2$ vs $E_1 + E_3 + E_4$
waveform	$6.3 \times 10^{-42}$	0.0001
yeast	$2.6 \times 10^{-22}$	0.6524
imgseg	$3.3 \times 10^{-15}$	$5.3 \times 10^{-6}$
nursery	$7.4 \times 10^{-31}$	0.0063
splice	$1.4 \times 10^{-11}$	$1.3 \times 10^{-14}$

Table 3.7: P-value of t-tests on Accuracy Rates

data set	$E_1$ vs $E_1 + E_3 + E_4$	$E_1 + E_2$ vs $E_1 + E_3 + E_4$
waveform	0.0005	0.0007
yeast	0.6262	0.4305
imgseg	0.2994	$4.3 \times 10^{-11}$
nursery	0.0002	0.0103
splice	0.2125	0.0680

changes are printed in italics in Table 3.5. Corresponding p-values are shown in Table 3.6 and 3.7. The tests showed that  $E_1 + E_3 + E_4$  further reduced the numbers of rules significantly compared to training with  $E_1 + E_2$  in three data sets *waveform*, *image-segmentation*, and *splice* with  $p < 0.0001$ . For the other two, the reduction is still significant compared to training with  $E_1$  (regular error backpropagation) with  $p < 1.4 \times 10^{-11}$ . These tests also showed that the changes in accuracy rates are not significant with the exception of *waveform* and *image-segmentation* using  $E_1 + E_3 + E_4$  versus  $E_1 + E_2$ , *nursery* using  $E_1$  versus  $E_1 + E_3 + E_4$ . Interestingly, in these three cases, the accuracy rates actually *increased* with the use of the newer penalty terms. Overall, training with new error terms  $E_2$ ,  $E_3$ , and  $E_4$  showed a significant reduction in number of rules with insignificant change in accuracy over

regular error backpropagation.

Table 3.8: Means ( $\sigma$ ) of Number of Antecedents over 100 Runs

data set	nursery	splice
$E_1$	5.6 (1.1)	6.3 (1.4)
$E_1 \& E_2$	3.4 (1.1)	6.6 (1.1)
$E_1 \& E_3 \& E_4$	3.1 (0.8)	6.1 (1.2)
C4.5rules	3.5 (0.2)	4.5 (0.1)

$(a_{30} = G, a_{31} = T, a_{34} = G) \rightarrow class E$
$(a_{29} \neq T, a_{30} = G, a_{31} = T, a_{32} = A) \rightarrow class E$
$(a_4 \neq A, a_{22} \neq A, a_{27} \neq A, a_{27} \neq G, a_{28} = A, a_{29} = G, a_{31} \neq T) \rightarrow class I$
$(a_4 \neq A, a_{22} \neq A, a_{27} \neq A, a_{27} \neq G, a_{28} = A, a_{29} = G, a_{30} \neq G) \rightarrow class I$
$(a_{17} \neq G, a_{20} \neq A, a_{23} \neq A, a_{27} \neq A, a_{27} \neq G, a_{28} = A, a_{29} = G, a_{30} \neq G) \rightarrow class I$
$(a_{17} \neq G, a_{20} \neq A, a_{23} \neq A, a_{27} \neq A, a_{27} \neq G, a_{28} = A, a_{29} = G, a_{31} \neq T) \rightarrow class I$
$(a_4 \neq A, a_{21} \neq A, a_{22} \neq A, a_{27} \neq A, a_{27} \neq G, a_{28} = A, a_{29} = G, a_{32} \neq A, a_{34} \neq G) \rightarrow class I$
$(a_{17} \neq G, a_{20} \neq A, a_{21} \neq A, a_{23} \neq A, a_{27} \neq A, a_{27} \neq G, a_{28} = A, a_{29} = G, a_{32} \neq A, a_{34} \neq G) \rightarrow class I$
$(a_{29} \neq G, a_{34} \neq G) \rightarrow class N$
$(a_{29} \neq G, a_{33} = G) \rightarrow class N$
$(a_{29} \neq G, a_{30} \neq G) \rightarrow class N$
$(a_{28} \neq A, a_{32} = C) \rightarrow class N$
$(a_{28} \neq A, a_{31} \neq T) \rightarrow class N$
$(a_{28} \neq A, a_{30} \neq G) \rightarrow class N$
$(a_{27} = G, a_{31} \neq T) \rightarrow class N$
$(a_{13} \neq T, a_{27} = A, a_{31} \neq T) \rightarrow class N$
Default rule: <i>class N</i>

Figure 3.7: A rule set extracted from a neural network trained on the *splice* data set. Here  $a_i$  denotes the nucleotide at position  $i$ , where  $a_i$  can be either A, T, G, or C. I (intron), E (exon), and N (neither) are the three classes of the DNA sequences to be predicted.

When extracting rules from data sets having all discrete input attributes such as *nursery* and *splice*, it is important to have small numbers of antecedents per rule to keep the rules easier to understand. Further, it is conceivable that when lowering the number of rules when using  $E_2$  or  $E_3 + E_4$ , one might simultaneously be increasing the number of antecedents per rule, thereby compromising the parsimony gained by

the modified hidden layer presentation. Table 3.8, which shows the average number of antecedents per rule for each method, indicates that this problem did not occur. These averages are either lower or almost the same. It shows that this method was able to reduce the number of rules without making the rules more complex. With the *nursery* data set, the rules extracted using the new error terms actually have fewer antecedents per rule. The average is about the same as with *C4.5rules*, yet there are many fewer rules compare to *C4.5rules* result. Figure 3.7 shows one rule set extracted from the data set *splice* with 17 rules where the average number of antecedents is higher than with *C4.5rules* but still not too overly complex to understand.

As described in Step 5b of the algorithm (Section 3.2), the rules in Figure 3.7 were constructed by combining two sets of *input*  $\rightarrow$  *hidden* rules and *hidden*  $\rightarrow$  *output* rules. First, the Chi2 algorithm (Step 2) divided hidden unit 2's activation range into two intervals  $[0, 0.56)$  and  $[0.56, 1]$ , hidden unit 3's activation range into two intervals  $[0, 0.67)$  and  $[0.67, 1]$ , and did not divide hidden unit 1's activation range. Using the intervals above, Step 3 extracted the following *hidden*  $\rightarrow$  *output* rules:

$h_3 \in [0.67, 1] \rightarrow \text{class } E$ $h_2 \in [0, 0.56) \text{ and } h_3 \in [0, 0.67) \rightarrow \text{class } I$ $h_2 \in [0.56, 1] \rightarrow \text{class } N$ <p>Default rule: class <i>N</i></p>
--

For each of the above rules, a set of *input*  $\rightarrow$  *hidden* rules was extracted. For instance, two conditions  $(a_{30} = G, a_{31} = T, a_{34} = G)$  and  $(a_{29} \neq T, a_{30} = G, a_{31} = T, a_{32} = A)$  were extracted for the condition that  $h_3 \in [0.67, 1]$ : Combined with the

rule  $h_3 \in [0.67, 1] \rightarrow \text{class } E$ , the following two rules were extracted:

$(a_{30} = G, a_{31} = T, a_{34} = G) \rightarrow \text{class } E$ $(a_{29} \neq T, a_{30} = G, a_{31} = T, a_{32} = A) \rightarrow \text{class } E$
--

In the same way, the rules for the second and third *hidden*  $\rightarrow$  *output* rule were extracted. The three sets of rules were then put together into the final set of rules in Figure 3.7.

When extracting rules from data sets having continuous attributes such as *waveform*, *yeast*, and *image-segmentation* the parsimony of rules is measured by the number of terms left in Equation 3.4, which also indicates the number of input-hidden weights after pruning. The average numbers of these weights are increased insignificantly from 49.8 to 54.2 for the *waveform* data set, 17.3 to 17.6 for the *yeast* data set, and dropped slightly from 65.66 to 63.67 for both  $E_2$  and  $E_3 + E_4$  compared to  $E_1$ . It should be noted that because the format of the rules (inequalities) for these two continuous input data sets are different from *C4.5rules*'s, the *number of rules* are not directly comparable. Pruning has a side effect that reduces the *fidelity* of the rule extraction method. Fidelity is a measure of how closely the extracted rules follow the network's behavior. In all experiments, the average accuracies of the rules and the networks differed by no more than 3%.

More significantly, the best among 100 runs in the experiments were able to extract even smaller rule sets than the averages described above. Rule extraction using  $E_3 + E_4$  extracted only 5 rules explaining the classification of 5000 *waveform* data instances with 89% accuracy rate, 23 rules for the *yeast* data set with 52%



Table 3.9: Means ( $\sigma$ ) of Accuracy and Number of Rules over 100 Runs with Rules Resulting in Default Class Removed

data set	waveform		yeast		imgseg	
	#rules	accuracy	#rules	accuracy	#rules	accuracy
$E_1$	44.34 (18.18)	85.08% (1.94)	61.29 (19.04)	51.61% (4.40)	38.34 (9.27)	91.58% (2.33)
$E_1 \& E_2$	9.33 ( 8.24)	85.20% (2.08)	35.86 (12.94)	51.42% (4.41)	32.02 (7.37)	89.29% (3.06)
$E_1 \& E_3 \& E_4$	6.20 ( 2.40)	85.73% (2.32)	37.52 (12.86)	51.86% (4.19)	26.12 (8.18)	91.86% (2.11)

data set	nursery		splice	
	#rules	accuracy	#rules	accuracy
$E_1$	113.04 (69.74)	88.57% (4.04)	28.73 (31.61)	88.55% (4.45)
$E_1 \& E_2$	25.25 (29.78)	89.37% (2.81)	31.19 (31.26)	89.18% (4.32)
$E_1 \& E_3 \& E_4$	18.85 (13.56)	89.94% (2.43)	15.88 ( 6.25)	90.12% (4.57)

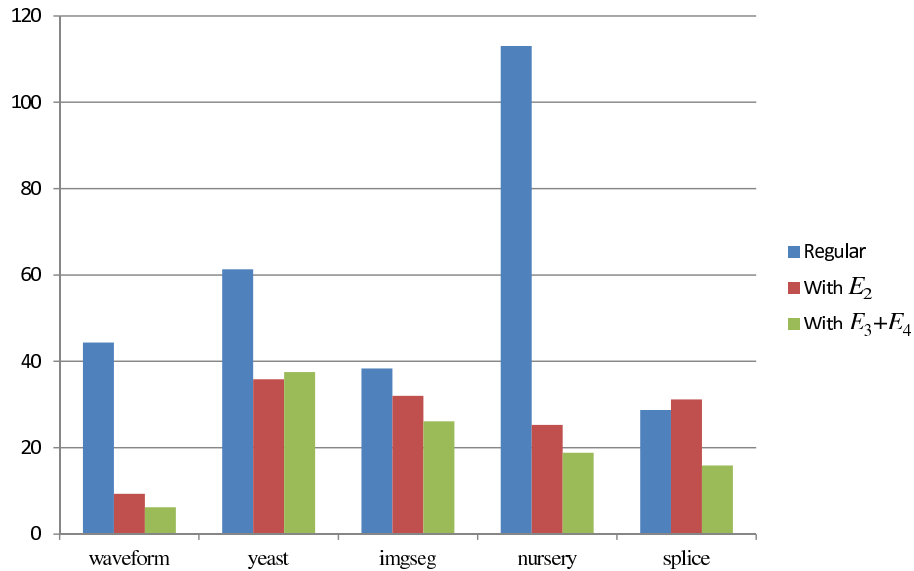


Figure 3.8: Mean number of rules extracted from networks trained by regular backpropagation,  $E_1 + E_2$ , and  $E_1 + E_3 + E_4$  with rules resulting in default class removed.

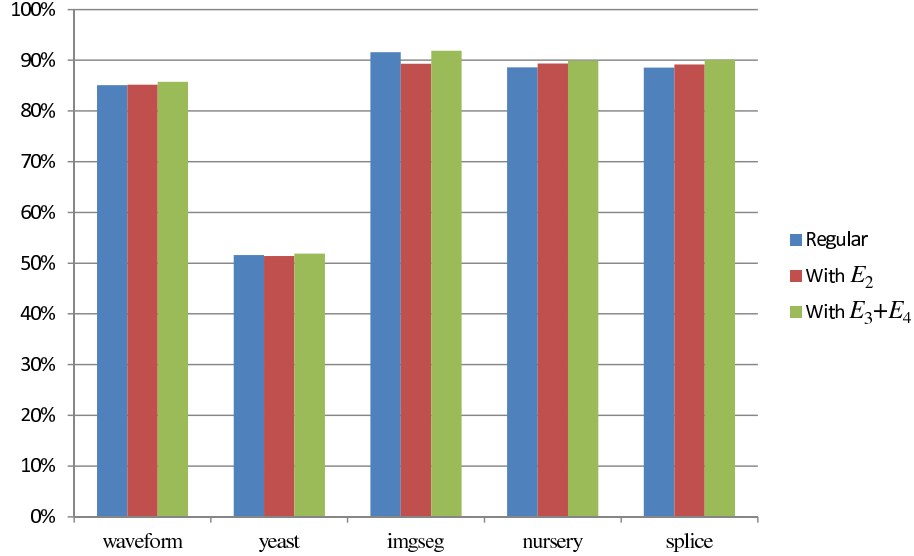


Figure 3.9: Mean accuracy of rules extracted from networks trained by regular backpropagation,  $E_1 + E_2$ , and  $E_1 + E_3 + E_4$  with rules resulting in default class removed.

accuracy, 12 rules for the *image-segmentation* data set with 92.21% accuracy, 13 rules for the *nursery* data set with 93.8% accuracy, and 17 rules for the *splice* data set with 94% accuracy. Rule extraction with new error terms clearly outperformed the popular rule-based system *C4.5rules* in extracting rules for four out of five data sets. For the remaining *yeast* data set, it also helped reduce the number of rules, although not enough to surpass *C4.5rules*.

*C4.5rules*' rules usually overlap, have to be applied in order, and a *default class* is assigned if no rule matches the input data. Some of the rules have the outcome the same as the *default class*. Since this rule extraction method uses *C4.5rules*, the final combined rules also have the same properties. When the rules that have the same outcome as the *default* are removed, one would expect that the accuracy rates to decrease significantly, but they did not. Table 3.9, Figures 3.8 and 3.9 show the results with these rules removed. The average numbers of rules were significantly

lower while there are insignificant changes to the accuracy rates. The reason could be that each intermediate rule set is responsible for an interval of hidden unit activation so its rules are simple and not overlapping. Such rule sets can be simplified by removing rules having the same outcome as the default class. The combination of these simple rules thus does not need these extra rules either.

Table 3.10: Average Time for Rule Extraction (RE) and Total Running Time (Seconds)

data set	waveform		yeast		imgseg		nursery		splice	
	RE	total	RE	total	RE	total	RE	total	RE	total
$E_1$	7.3	12.1	1.1	2.5	3.6	7.5	0.8	1.8	0.8	3.3
$E_1 \& E_2$	4.5	11.3	1.0	2.6	3.6	9.0	0.7	2.2	0.8	7.7
$E_1 \& E_3 \& E_4$	6.9	17.7	1.0	2.9	3.6	11.9	0.7	2.8	0.9	14.5

Table 3.10 shows the average running time in seconds spent on rule extraction and the total running time including training the networks. All experiments were run on an Intel Core2 2.4 GHz system. The new penalty terms increased the running time due to more computation, as would be expected. In particular, the most increase in running time was recorded with the *splice* data set and it was only 4.4 times. In their original forms,  $E_2$ ,  $E_3$ , and  $E_4$  require  $N$  times more computation to compute than  $E_1$  where  $N$  takes values from 1296 to 5000. But the derivation using local computations made it possible to keep the increase in running time surprisingly low even though  $N = 3190$  for the *splice* data set.

### 3.6 Discussion

In this chapter, a method is presented for improving the extraction of symbolic rules from multilayer feedforward neural networks by adding additional terms to the error function is presented. These terms encourage the formation of a more separable internal representation at the hidden layer. Efficient ways to incorporate them into the training process *while retaining local computations* are also derived and implemented. Unlike past rule extraction methods, this method focuses on modifying training so that existing rule extraction methods work more effectively. The three introduced penalty terms  $E_2$ ,  $E_3$ , and  $E_4$  share the same purpose of making the hidden unit activations of different classes more separable. While  $E_2$  is simple and does not rely on class labels,  $E_3$  and  $E_4$  are more complex and employ class labels to increase and decrease the activation distances discretionarily.

Extensive experiments with five large, publicly available data sets showed that this approach helped reduce the number of rules significantly without sacrificing classification accuracy. Rule sets extracted from networks trained with  $E_2$  are smaller than with regular error backpropagation. Even fewer rules can be extracted from networks trained with  $E_3$  and  $E_4$ . These results showed that the rule extraction method also outperformed the popular *C4.5rules* program in four out of five of these data sets. An important future research direction will be to compare these results with those of other rule extraction methods in the literature.

The aim of this work was not to produce a new rule extraction algorithm, but to provide an improved way to train networks that might help most rule

extraction algorithms using the decompositional approach. The rule extraction process being used is fairly standard and similar to those used in many other algorithms. Those algorithms use different approaches to extract the intermediate rules, from exhaustive search to complicated heuristics. Here *C4.5rules* is used as a standard and straightforward algorithm in order to generate the intermediate rules. This should not be confused with using *C4.5rules* to generate rules directly from the whole data set. The same rule extraction algorithm is used for both the control condition (regular backpropagation using  $E_1$  alone) and experimental learning condition ( $E_2$ ,  $E_3$ , and  $E_4$ ) to show the effectiveness of the new terms on rule extraction algorithms using the hidden activation intervals. It is hypothesized that *any* rule extraction method using a similar approach will benefit from the use of these terms.

The surprising result with the default class in Table 3.9 demonstrated another advantage of neural network based rule extraction. The extracted rules often appear to be non-overlapping, so that rules resulting in the default class could be removed. Such rule sets are easier to apply and also easier for a person to use to study properties of data sets. Although it is not clear what caused the rules to be non-overlapping, the answer is likely to be the way neural networks using regular backpropagation divide the input space using hidden unit activation intervals.

Experimental results showed that accuracy rates and  $E_1$  changed very little when networks were trained with the augmented penalty terms. This demonstrates a well-known property of neural networks: that there are many possible encodings at the hidden layer that can provide correct outputs. These encodings are biased

towards more separation of activity patterns with the approach introduced here. Based on these promising results, an important future research direction will be to study other ways to bias the encodings beyond the sum of squared distances. Presumably this approach can also be applied to other error backpropagation learning rules with different error functions such as [9, 51, 85],

As with many approaches using penalty terms, there is a trade off in terms of parameter adjustments: a small value of  $\alpha$  will make the activation patterns very separated and good for rule extraction, but it cannot keep the training error low enough. The opposite holds for small  $\beta$ . The problem is to influence the training enough to produce the desired separation without compromising  $E_1$  or accuracy. Since no single value of  $\beta$  that works best across all data sets is found, an important future direction for research is to study different schemes to adapt  $\alpha$  and  $\beta$  automatically as training progresses. Such work might try to establish properties of a data set that predict reasonable values for  $\beta$ , or investigate how varying  $\beta$  during training as a function of error rate influences the rule acquisition process.

## Chapter 4

### Finite State Machine Extraction from Simple Recurrent Networks

In this chapter, the error term  $E_2$  in the previous chapter is generalized to work with simple recurrent networks (“Elman networks”; see Section 2.2) [31]. The purpose of  $E_2$  is once again to make hidden layer activation patterns more separated from one another. It is not obvious a priori that the method will continue to work unaltered on simple recurrent networks because these networks’ temporal dynamics is completely different from feedforward networks’ static mapping. Moreover, it is necessary to extract finite state machines instead of propositional logic rules from simple recurrent networks. Above all, the hidden layer is now affected not only by the input, but also by the *context* layer, so it is unclear if the term  $E_2$  will still be effective in pushing the activation patterns in the hidden layer apart. Therefore, it is imperative to evaluate the effect of  $E_2$  systematically on a variety of data sets.

This chapter starts by reexamining  $E_2$  and the gradient calculation in the context of simple recurrent networks. Next, a simple algorithm to extract FSMs that takes advantage of the improved representation is introduced. Finally, computational experiments on four data sets generated from regular and context-free grammars are used to evaluate the effect of  $E_2$  on FSM extraction.

## 4.1 Generalizing $E_2$

In this section we are interested in extracting FSMs from simple recurrent networks with one hidden layer as shown in Figure 4.1. The error function, as defined in Section 2.2, is computed over the output units:

$$E_1 = \frac{1}{2} \sum_{t=1}^N \sum_{k \in \text{output}} (T_k^t - a_{O_k}^t)^2$$

where  $T_k^t$  is the target output for the  $k^{\text{th}}$  output unit at time step  $t$ , and  $N$  is the length of the input sequence.

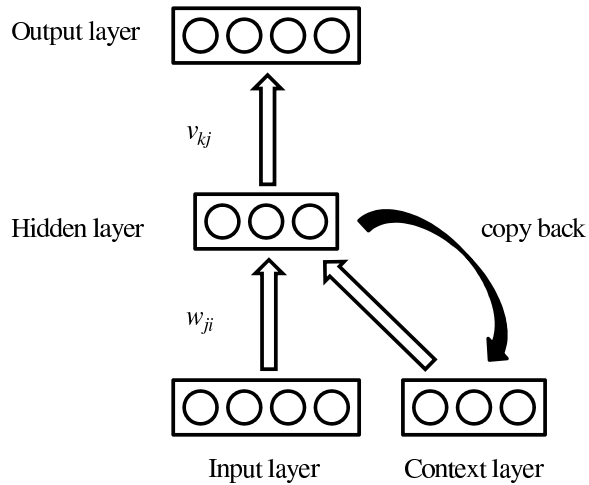


Figure 4.1: A small simple recurrent neural network; the number of nodes in each layer varies and can be quite large.

The penalty term  $E_2$  is generalized to recurrent networks as:

$$E_2 = -\frac{1}{2} \sum_{t=1}^N \sum_{t'=1}^N \sum_{k \in \text{hidden}} (a_{H_k}^t - a_{H_k}^{t'})^2$$

where  $a_{H_k}^t$  is the activation of the  $k^{\text{th}}$  hidden unit at time step  $t$  and  $a_{H_k}^{t'}$  is analogous at time step  $t'$ . Thus  $E_2$  is the sum over all pairs  $(t, t')$  of the squared Euclidean distance between two hidden activation vectors at time step  $t$  and  $t'$ . The negative



sign ensures that when neural network training minimizes the measure  $E_2$ , it will maximize the distances between the hidden layer vectors. When  $t = t'$ , only zeroes enter the sum so no special attention is given to that situation. As a result,  $E_2$  decreases as the hidden unit activation vectors are further apart:

As before, the total error function guiding learning is:

$$E = \alpha E_1 + \beta E_2$$

where  $\alpha, \beta > 0, \alpha + \beta = 1$ . Note that the double sum over  $t$  and  $t'$  can make  $E_2$  quite large relative to  $E_1$ , so  $\beta$  must be quite small to scale  $E_1$  and  $E_2$  appropriately.

In order to train the network with error backpropagation, we need to compute the components of the gradient of  $E$  given by:

$$\begin{aligned} \frac{\partial E}{\partial w_{ji}} &= \alpha \frac{\partial E_1}{\partial w_{ji}} + \beta \frac{\partial E_2}{\partial w_{ji}} \\ \frac{\partial E}{\partial v_{kj}} &= \alpha \frac{\partial E_1}{\partial v_{kj}} + \beta \frac{\partial E_2}{\partial v_{kj}} \end{aligned}$$

where  $w_{ji}$  is an input-to-hidden weight, and  $v_{kj}$  is a hidden-to-output weight. Of course, the standard terms  $\frac{\partial E_1}{\partial w_{ji}}$  and  $\frac{\partial E_1}{\partial v_{kj}}$  can be computed efficiently as in [17]. We also have  $\frac{\partial E_2}{\partial v_{kj}} = 0 \forall j, k$  with  $v_{kj}$  being the weight to the  $k^{th}$  output unit from the  $j^{th}$  hidden unit because  $E_2$  does not have any  $v_{kj}$  component. As shown in Section 3.1,  $\frac{\partial E_2^p}{\partial w_{ji}}$  can be computed efficiently as follows:

$$\frac{\partial E_2^t}{\partial w_{ji}} = -N(a_{H_j}^t - \overline{a_{H_j}})a_{H_j}^t(1 - a_{H_j}^t)x_i^t$$

with  $N$  being the length of the training data sequence, and  $\overline{a_{H_j}}$  being the average activation of the  $j^{th}$  hidden unit over all time steps. The equation is identical to Equation 3.3 for feedforward networks except for the superscript of the term  $a_{H_j}^t$  as

might be expected. Each hidden unit activation  $a_{H_j}^t$  at time step  $t$  plays the role of the activation of the hidden unit for the  $p^{th}$  training example.

Note that the derivation above used a single  $N$ -length sequence just for clarity. When training with multiple sequences with different lengths, it is trivial to extend the method so that  $N$  is the total length of all training sequences and  $\overline{a_{H_j}}$  is the average activation over all training time steps.

As with feedforward networks, when computing  $\frac{\partial E}{\partial w_{ji}}$  for time step  $t$ , besides looking at the activation of the  $j^{th}$  hidden unit and the  $i^{th}$  input unit as is done with the usual backpropagation training, we only need one more value  $\overline{a_{H_j}}$  which can be computed and stored locally at the  $j^{th}$  hidden unit. This local property is highly desirable in neural network training.

## 4.2 Finite State Machine Extraction

The same FSM extraction algorithm is used for both the experimental condition ( $E = E_1 + E_2$ ) and the control condition ( $E = E_1$ , which is basic backpropagation). An outline of the FSM extraction algorithm in both cases is:

- Step 1: Train the network.
- Step 2: Cluster the hidden unit activation vectors.
- Step 3: Construct the FSM with clusters as states.
- Step 4: If there are two hidden unit activation vectors  $v_1$  and  $v_2$  in the same cluster  $C$  such that upon receiving the same input  $x^t$ , the next state vectors  $v'_1$

and  $v'_2$  are in two different clusters, split cluster  $C$  and go to Step 3.

- Step 5: If hidden unit activation vectors in one cluster produce different predictions at the output layer, split the cluster and go to Step 3.

These are typical steps used in previous FSM extraction algorithms [37, 38]. In this study, RPROP [64] (resilient backpropagation) is used again in Step 1. It is an improved backpropagation learning algorithm that trains networks faster by adjusting the weight update based on the direction of the gradient instead of the magnitudes of the derivatives. It also requires fewer training parameters.

In Step 2 the K-means clustering algorithm [44] is used.  $K$  is set to the number of output symbols. This is a lower bound of the number of states because each state can only predict one output symbol. Hence, an FSM has to have at least as many states as output symbols. Step 2 is only executed once so  $K$  only varies by data sets. Note that in subsequent steps, the algorithm K-means is used again but with values of  $k$  determined in different ways as explained below.

Step 3 is done by going over the training data sequence to construct the FSM state transition function and the output function. At time step  $t$ , let  $s^t$  be the current input symbol. Using the input encoding scheme, the augmented input vector  $x^t$  is constructed from the encoding of  $s^t$  and the context units  $a_H^t$  copied from the previous time step. Applying the network activation rule, the hidden unit activation  $a_H^{t+1}$  at the next time step is calculated. Suppose  $a_H^t$  is in cluster  $C_0$  and  $a_H^{t+1}$  is in cluster  $C_1$ , the transition  $C_0 \xrightarrow{s^t} C_1$  will be added to the FSM state transition function. Intuitively,  $a_H^t$  is the “state” of the network after processing the input

patterns up until time  $t$ . Also, the input  $s^t$  changes the network's state to  $a_H^{t+1}$ . From a FSM perspective,  $s^t$  is the symbol that causes the transition  $a_H^t \rightarrow a_H^{t+1}$ . Besides, the clusters  $C_0, C_1$  represent the activation patterns that are close to  $a_H^t$  and  $a_H^{t+1}$ , respectively. That is, activation patterns, or the network's "state", in  $C_0$  are expected to make the same transition under the same input  $s^t$  to the activation patterns in  $C_1$ .

Steps 4 and 5 split existing clusters into smaller ones. That is, the hierarchy resulting from clustering is not kept and the new clusters are used as new states in the FSM. For example, let  $N$  be the number of clusters/states and we need to split cluster  $C_i$  into 3 clusters. The K-means algorithm is again used to cluster the set of vectors belonging to cluster  $C$  into 3 clusters  $C'_0, C'_1$ , and  $C'_2$ . After this step, the FSM will have  $N + 2$  states, with  $C_i$  being replaced by  $C'_0$ , and two new clusters  $C'_1$  and  $C'_2$ .

With any given activation vector  $a_H^t$ , the network activation rule lets us calculate the output vector  $a_O^t$ , which can be decoded to a symbol. Therefore, for each cluster  $C$ , the activation rule is used to calculate the output vectors from all hidden unit activation vectors  $a_H^t$  belonging to it. Subsequently, the output symbols can be decoded from the output vectors. Because the activations are in the same clusters, they are close together; hence the output vectors produced by them are similar and should be decoded to a single symbol. If more than one symbol is produced by  $a_H^t$  in the same cluster  $C$ , this cluster will be split in Step 5.

As other finite state machine extraction work, we are only concerned with the next-word/symbol prediction task. In the same way, the input symbol is encoded

at the input layer using the one-hot encoding scheme (explained in the following section) so that all the attributes are binary. Hence, unlike in the previous chapter, no distinction in the algorithm needs to be made between continuous and discrete value attributes.

### 4.3 Experimental Methods

The goal of this evaluation is to compare the FSM extracted from a trained simple recurrent network when  $E_2$  is included in the error function (experimental condition) versus the number when  $E_2$  is not included (control condition of  $E_1$  alone, i.e., standard backpropagation). The effectiveness of the FSM extraction method is evaluated on two data sets generated by regular languages and two data sets generated from context-free languages consisting of 50 to 1500 sequences. The first two data sets have been used extensively in the past in evaluating FSM extraction methods from regular languages [17, 37, 38, 48, 25]. The second two data sets have been used repeatedly in evaluating recurrent neural networks performance and FSM extraction methods from context-free languages [21, 22, 23, 84].

The first data set is generated from the regular language *badigu*, first used in [17] to evaluate the learning capability of simple recurrent networks. The regular expression defining it is  $(ba|dii|guuu)^*$ . The alphabet consists of 6 symbols:  $\{b, a, d, i, g, u\}$ . Figure 4.2 shows an FSM for this language. In the language, every  $b$  is followed by exactly one  $a$ , every  $d$  is followed by exactly two  $i$ 's and every  $g$  is followed by exactly three  $u$ 's. The task for the neural network is to predict the

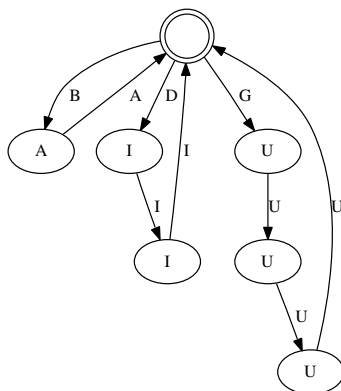


Figure 4.2: Finite state machine for the *badigu* language. Each circle represents one state, and the double circle represents the initial state. Symbols inside circles are the FSM’s outputs (prediction of the next symbol). With this specific grammar, all states are acceptable terminal states. The transitions are shown by the arrows and labeled by their corresponding input symbols.

next symbol given the preceding ones. Obviously, every *a*, *i*, and *u* can be predicted perfectly, but *b*, *d*, and *g* are random and cannot be predicted. This data set consists of 50 sequences of strings in the language with length from 5 to 50. The network is given symbols one by one and has to predict the next symbol.

The second data set is generated from the *Tomita #4* language [81]. The language is defined as any string over the alphabet  $\{0, 1\}$  that does not have “000” as a substring. The task for the neural network is to tell whether the symbols presented up until each time step constitute a string in the language. This task is of interest because the correct output is completely determined at each time step, while about 30% of the symbols in the *badigu* task cannot be predicted (the *b*, *d*, and *g* symbols are random).

The third and fourth data sets are generated from the context-free grammars (CFG) in Tables 4.1 and 4.2. *CFG #1* was used previously in [21] to evaluate recurrent network performance on CFGs. It is used here to evaluate the ability

Table 4.1: Context Free Grammar #1

S	→	Simple   Right   Center
Simple	→	N V N <i>[end]</i>
Right	→	N V N <i>who</i> V N <i>[end]</i>
Center	→	N <i>who</i> N V V N <i>[end]</i>
N	→	$n_1 n_2 n_3 n_4$
V	→	$v_1 v_2 v_3 v_4$

Table 4.2: Context Free Grammar #2

S	→	NP <sub>subj</sub>   NP <sub>obj</sub> <i>[end]</i>
NP <sub>subj</sub>	→	N <sub>subj</sub> (70%)   N <sub>subj</sub> SRC (6%)   N <sub>subj</sub> ORC (9%)   N <sub>subj</sub> PP <sub>subj</sub> (15%)
NP <sub>obj</sub>	→	N <sub>obj</sub> (70%)   N <sub>obj</sub> (6%)   N <sub>obj</sub> (9%)   N <sub>obj</sub> PP <sub>obj</sub> (15%)
SRC	→	<i>that</i> V NP <sub>obj</sub>
ORC	→	<i>that</i> N <sub>subj</sub> V
PP <sub>subj</sub>	→	<i>from</i> NP <sub>subj</sub>   <i>with</i> NP <sub>subj</sub>
PP <sub>obj</sub>	→	<i>from</i> NP <sub>obj</sub>   <i>with</i> NP <sub>obj</sub>
N <sub>subj</sub>	→	N <sub>female</sub>   N <sub>male</sub>
N <sub>obj</sub>	→	N <sub>animal</sub>
N <sub>female</sub>	→	women   girls   sisters
N <sub>male</sub>	→	men   boys   brothers
N <sub>animal</sub>	→	bats   giraffes   elephants   dogs   cats   mice
V	→	chase   see   swing   love   avoid   follow   hate   hit   eat   like

Table 4.3: Properties of Data Sets Used for Evaluation

data set	No.symbols	No.seq.	Avg.length	Network
badigu	6	50	25	6-2-6
Tomita #4	2	50	30	2-2-2
CFG #1	10	1000	4	10-4-10
CFG #2	26	1500	5	26-3-26

of the algorithm on a relatively simple data set where the states and transitions can be easily checked. *CFG #2* is a large and difficult CFG used in [22, 23] to evaluate recurrent networks and FSM extraction from them. While *CFG #1* has only 10 terminal symbols and simple production rules, *CFG #2* is considerably more complex. It has 26 terminal symbols and many more production rules with deeply nested structures. However, all nouns are plural so subject-verb agreement rules are not necessary. These two CFGs are chosen so that the extraction algorithm can be evaluated on both simple and complex grammars. Table 4.3 summarizes the sizes and characteristics of the four data sets. It also shows the size of the network layers used in the experiments (e.g. 6-2-6 means 6 input, 2 hidden, and 6 output units). Exploratory simulations with a larger number of hidden units produced no qualitatively different results.

For all data sets, the settings for the experimental and control runs are identical except for which error function is used, and are similar to those used with feedforward networks in the preceding chapter, with some modifications:

1. Ten-fold cross validation scheme: Each data set is split randomly into 10 subsets of approximately equal size. Eight subsets were used for training, one for validation, and one for measuring the accuracy of the networks. The procedure is repeated 10 times, each time using a different subset as the testing set. Each experiment is also run 10 times with different random initial weights. The reported number of rules and accuracy are averages over all 100 runs. Having so many runs ensures that any improvement comes from the method



and not just by chance.

2. In each run, the experiments with and without the new error terms have the same starting point, i.e., matched initial weights and data set division, to make comparison maximally compatible.
3. RPROP with weight backtracking was set up to run for a maximum of 1000 epochs. The network with highest validation accuracy was saved for subsequent FSM extraction.
4. A local one-hot encoding scheme is used to encode the input symbol. For instance, let  $n$  be the size of the input symbol set. The  $i^{th}$  symbol is encoded as  $\vec{a}_I = (a_{I_1}, a_{I_2}, \dots, a_{I_n})$  with  $a_{I_k} = 0.9$  for  $k = i$  and  $a_{I_k} = 0.1$  for  $k \neq i$ . For *CFG #2*,  $a_{I_k}$  is encoded exactly as in [23] to make the results comparable. Thus,  $a_{I_k}$  is encoded using 0.99 for  $k = i$  and 0.01 for  $k \neq i$  instead.

Fixing the values of  $\alpha$  and  $\beta$  improperly during training either causes  $E_2$  to be not effective (when  $\beta$  is too small), or  $E_1$  to be not effective with a large error  $E_1$  at the end ( $\beta$  is too large). At the beginning,  $E_1$  is very large, hence  $\alpha$  has to be small so that the gradient from  $E_1$  does not drown out the gradient from  $E_2$ . Conversely, near the end of training,  $E_1$  is small, and thus  $\alpha$  needs to be bigger to keep the network error low. While using  $\alpha$  and  $\beta$  is good for describing and calculating  $\partial E_2 / \partial w_{ji}$ , a more practical way to control the contributions of  $E_1$  and  $E_2$  is used. First, note that  $\partial E / \partial w_{ji} = \partial E_1 / \partial w_{ji} + \partial E_2 / \partial w_{ji}$  where  $w_{ji}$  is some individual weight. Let  $d_1$  and  $d_2$  be the mean of the magnitude of  $\partial E_1 / \partial w_{ji}$  and  $\partial E_2 / \partial w_{ji}$

respectively. It follows that  $d_1$  and  $d_2$  are two positive scalars that approximate the contributions of  $E_1$  and  $E_2$  in the gradient  $\partial E/\partial w_{ji}$ . Hence, the ratio  $d_2/d_1$  changes during training: it is very small at the beginning when  $E_1$  is large and becomes large near the end when  $E_1$  is small. In order to keep  $E_1$  and  $E_2$ 's contributions relatively stable, this ratio is fixed to  $d_2/d_1 = \gamma$  during the training process. This can easily be accomplished by scaling  $\partial E_2/\partial w$  in each training epoch. Conveniently, RPROP only uses the sign of  $\partial E/w_{ji}$  instead of using *both* the sign *and* the magnitude as other error backpropagation methods. As a result, the scaled  $\partial E_1/\partial w_{ji} + \partial E_2/\partial w_{ji}$  can be readily used by RPROP without further normalization. The value of  $\gamma$  is set to 0.9 for *badigu* and *CFG #1*, 0.5 for *Tomita #4* and 1 for *CFG #2* based on a few pilot simulations.

The experiments in this section are implemented in C++ and run single threaded on a Core i7 3.4 GHz CPU. The average time for each run ranges from 0.5 seconds (for *Tomita #4*) to 25 seconds (for *CFG #2*) depending on the data set and experiment settings.

## 4.4 Results

### 4.4.1 An Illustrative Example

Here the hidden unit encodings learned by a two-hidden unit neural network for the *badigu* problem is used to illustrate the approach. After training, the hidden unit activation vector  $(a_{H_1}, a_{H_2})$  of the two hidden units for each time step is calculated. This vector is an encoding of the current “state” of the network after seeing all

previously presented symbols. Each state alone is sufficient to calculate the output vector, which is the prediction of the next symbol. We are interested in how these vectors are arranged in the hidden activation space when the new error term  $E_2$  is used versus when it is not. During FSM extraction, these vectors are divided into clusters, and each cluster is represented as one state of the FSM. Consequently, it is desirable that there are as few as possible states in the FSM so that a human can understand the FSM easily.

Figure 4.3a shows the hidden unit activation space after the network has been trained with the regular sum of squared error function  $E_1$  used in standard backpropagation. Note how the clusters spread through the interior regions of the space in Figure 4.3a, as would be expected from the results in Chapter 3. While the clusters are not circular, circles are added to the plot only to make it easy to see which activation vectors belong to which clusters. In addition, the centers of the circles are the means of the vectors belonging to the clusters, while the radius is the maximum distance to the vector furthest from the center. As a result, some circles overlap, but this does not mean that the clusters do. Furthermore, some clusters contain vectors that spread out more so the circles are bigger. In Figure 4.3a, there are 12 circles representing the 12 clusters/states the FSM extraction algorithm found using standard backpropagation. Some of the states are quite close together as if they could be merged into one. Nonetheless, none of them could, including the closest ones, because either (1) given the same input symbol to the two states, the FSM would move to different states, or (2) the hidden activation vectors belonging

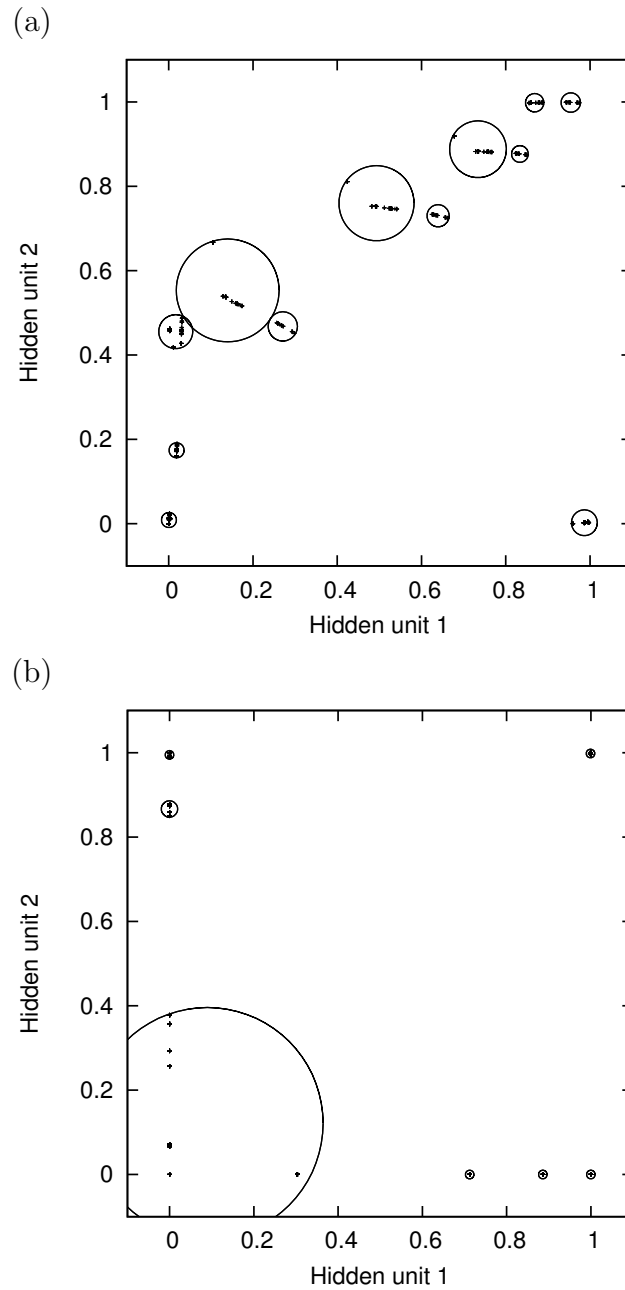


Figure 4.3: A typical hidden unit activation vectors throughout all time steps for the *badigu* problem after training with (a) regular backpropagation ( $E = E_1$ ) versus (b) the same error function but augmented to include the new error term ( $E = E_1 + E_2$ ). Note that the clusters are not circular, circles are only added for illustrative purposes to delineate the clusters.

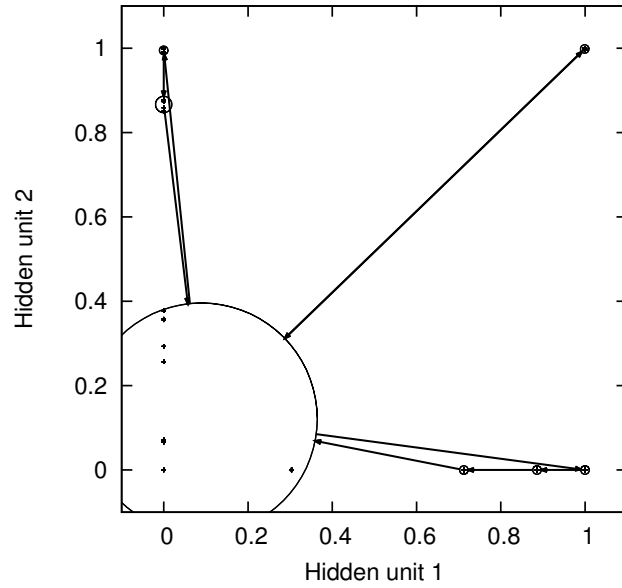


Figure 4.4: Hidden unit activation vectors throughout all time steps for the *badigu* problem after training with the augmented error function  $E_1 + E_2$ . The 7 clusters delineated by the 7 circles represent the 7 states of the extracted FSM. The arrows connecting the circles denotes the transitions in the FSM.

to the two states produce different symbols at the output. These situations are used as the preconditions for Step 4 and Step 5 of the FSM extraction algorithm described in Section 4.2.

Figure 4.3b shows the hidden unit activation space after the network with initial weights identical to those used in the previous example has been trained with the help of  $E_2$ . The new error term  $E_2$  pushed the activations apart while  $E_1$  keeps the activations that should be together close to one another. Clearly, the combined effect is that these activation vectors are automatically grouped into fewer very tight and separated clusters. The number of clusters has been reduced from 12 to 7 and most of the vectors are almost identical so they are on top of each other in the plots and were represented by just a few crosses. Further, unlike when  $E_1$  is used alone, clusters are now preferentially located along boundaries (contrast Figure 4.3a with

Figure 4.3b).

It is important to note the difference between this approach and the common previous approaches that force hidden activation vectors to be binary values [45, 86]. While most activations have 0/1 components, many of them have other components not equal to 0/1. Hence, only two hidden units are necessary to encode the solution. The binary vector approaches would need more hidden units to encode 7 states. Even so, the encoding scheme is very restrictive. The approach presented in this work allows more efficient and flexible encodings at the hidden layers.

In Figure 4.4, arrows showing the state transition graph corresponding to Figure 4.3b are added. The extracted FSM is isomorphic to the source FSM shown in Figure 4.2. The big cluster near  $(0, 0)$  represents the initial state, which is the only state from which the one cannot predict the next symbol. But the dynamics of neural networks require that the network has to attempt to predict some output the best it can. Hence, there are multiple subclusters inside this cluster that actually predict either  $b$ ,  $d$ , or  $g$ , although with low accuracy. The path through the three clusters on the bottom right, the clusters on the top left, and the single cluster on the top right represent the paths that predicts  $guuu$ ,  $dii$ , and  $ba$  respectively.

Table 4.4: Regular versus Modified Backpropagation (Averaged over 100 Runs)

data set	$E_1/N$		$-2E_2/N^2$	
	regular	new	regular	new
badigu	0.185	0.197	0.613	0.729 (+19%)
Tomita #4	0.003	0.001	0.576	0.650 (+13%)
CFG #1	0.413	0.420	0.605	1.061 (+75%)
CFG #2	0.752	0.766	0.823	1.000 (+22%)

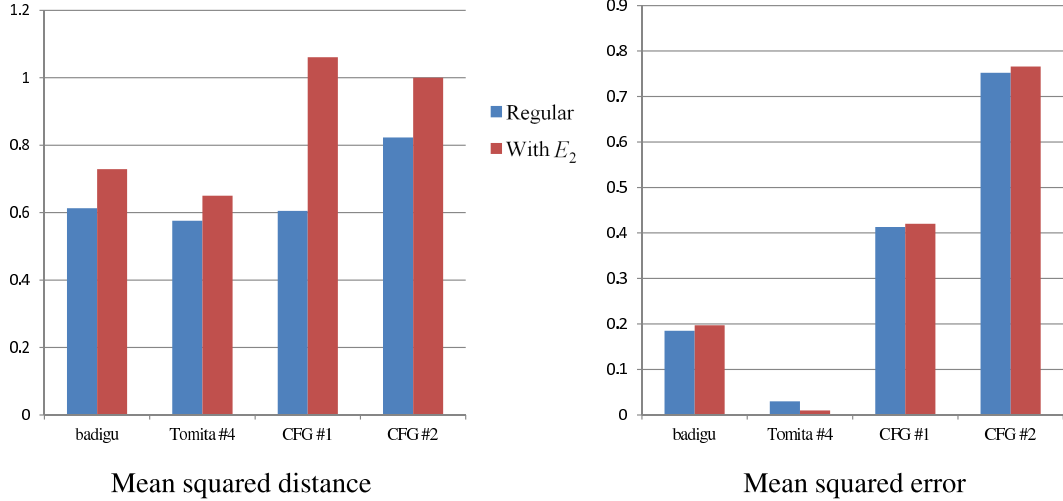


Figure 4.5: Mean squared distance and mean squared error of networks trained by regular and modified backpropagation.

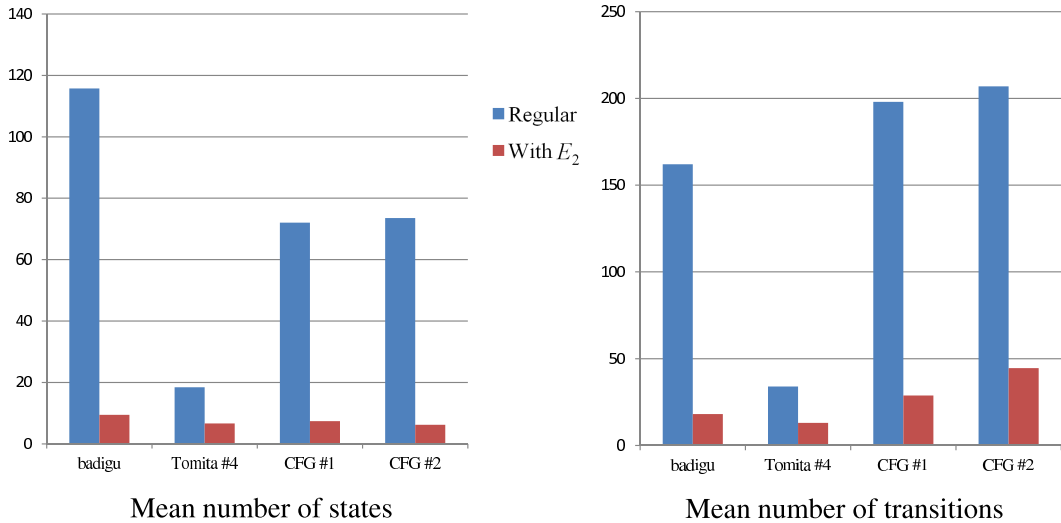
#### 4.4.2 Systematic Evaluation

Table 4.4 and Figure 4.5 show the effect of the new error term  $E_2$  on the network’s average testing error ( $E_1/N$ ) and the average squared distances among hidden unit activation vectors ( $-2E_2/N^2$ ) where  $N$  is the total length of all training sequences. A network’s average testing error and average squared distance between hidden activation vectors are shown after training with regular backpropagation (control condition) versus the new combined error term (experimental condition). Since  $E_2$  is 1/2 of the negated sum of squared distances between each pair (total  $N^2$  pairs) of activations,  $-2E_2/N^2$  is the average squared distance between each pair. The *regular* columns show results when the networks were trained with “regular” backpropagation using  $E = E_1$ . The *new* columns show results with  $E = E_1 + E_2$ . These data show that activation pattern distances were increased up to 75% with only a small change in network errors. The small change in  $E_1$  again supports the

hypothesis that it is possible to make error backpropagation learn a different encoding that satisfies other criteria (smaller  $E_2$ ) while still maintaining near minimum error at the output layer.

Table 4.5: Means ( $\sigma$ ) of Number of States and Transitions over 100 Runs

Data set	No. of states		No. of transitions	
	regular	new	regular	new
badigu	115.76 (91.9)	9.44 (6.3)	162.09 (119.2)	18.15 (9.2)
Tomita #4	18.45 (24.3)	6.63 (4.4)	34.00 (42.1)	13.05 (8.8)
CFG #1	72.05 (71.8)	7.36 (4.8)	198.00 (167.4)	28.80 (15.8)
CFG #2	73.53 (69.3)	6.20 (2.6)	206.89 (156.6)	44.60 (11.8)



$E_2$

Figure 4.6: Mean number of states and transitions of finite state machines extracted from networks trained by regular and modified backpropagation.

Table 4.5 and Figure 4.6 present the experimental results concerning FSM extraction. The new error term helped to reduce the number of states and the standard deviations significantly. With regular error backpropagation ( $E_1$ ), the



number of states of the extracted FSM is sometimes very large because the hidden activation space is very distributed, and hidden activation vectors encoding different outputs are close together. The latter problem makes it very difficult to separate these vectors into distinct clusters. As a result, the control algorithm has to partition the hidden activation space into large numbers of regions. In other words, the FSM has to have a lot of states. This problem has been reported previously by others in which the number of states of FSMs extracted for the same problem varied from 6 to 190 [23]. Remarkably, using data generated using the same method, the FSM extraction algorithm presented in this work produced FSMs with an *average* of 6.2 states, very close to 6.

The large standard deviations in Table 4.5 for the regular/control runs raise the question of whether the reduction in average number of states is statistically significant. For this purpose, paired *t*-tests are used using a standard significance level of 0.05. Bonferroni correction [20] for 4 tests requires the significance to be defined as  $p < 0.05/4 = 0.0125$ . The corresponding p-values for the four data sets are all very low and well below 0.0125. The largest p-value is only  $5.8 \times 10^{-6}$  for *Tomita #4*, and the smallest p-value is  $2.2 \times 10^{-16}$  for *badigu*. These tests confirmed that the improvement from  $E_2$  is statistically significant. Large standard deviations in the control experiments are consistent with past experience of FSM extraction from recurrent networks [23, 38]. Table 4.5 (right two columns) shows that the numbers of transitions and the standard deviations are also markedly reduced by using  $E_2$ .

For *badigu* and *Tomita #4* data sets, the data was generated from simple human-

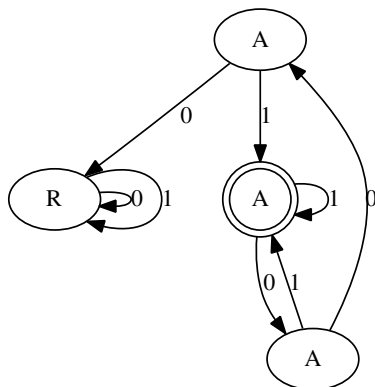


Figure 4.7: Extracted FSM for data set *Tomita #4*. The circles and arrows represent the states and transitions of the FSM respectively. The state with the double circle is the initial state. At each state, the network outputs a symbol *A*(ccept) or *R*(eject) to indicate whether the input so far belongs to or does not, respectively, the *Tomita #4* language. Those symbols are shown on the FSM’s node labels. The labels on the arrows denote the input symbols for the transitions.

designed FSMs. Moreover, they have been studied under FSM extraction before in [17, 37]. These earlier studies found that the simplest FSMs, the ones with fewest states and transitions, are the original source FSMs. The FSM extraction algorithm presented in this work also found these FSMs in the best runs. The extracted FSMs are shown in Figure 4.4 and Figure 4.7. To the best of my knowledge, while previous work has been able to extract the simplest FSMs in the reported best runs, none has reported extensive experiments with multiple runs using different data partitioning and initial weights as is done here. This FSM extraction algorithm performs *consistently* well on 100 runs with 10-fold cross validation and different initial weights. The average number of states for the two *badigu* and *Tomita #4* problems (9.44 and 6.63) are very close to the number of states of the original source FSMs (7 and 4).

Even though data set *CFG #1* has been used previously to study the perfor-

mance of recurrent networks [21, 22], I am not aware of any work that evaluated FSM extraction on it. The data set is interesting because a huge number of sentences could be generated from very simple production rules. Consequently, this is difficult

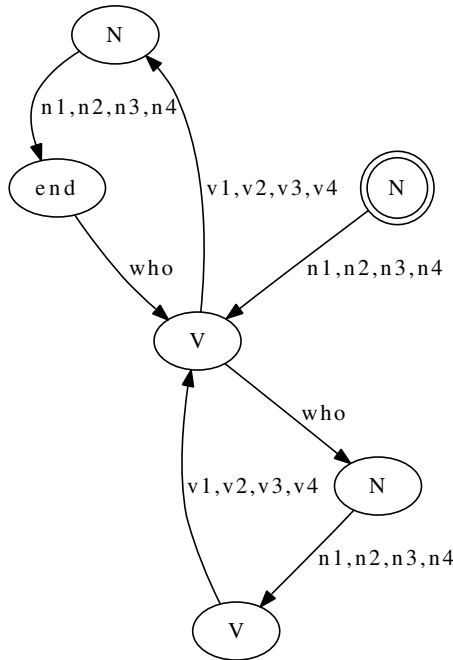


Figure 4.8: Extracted FSM for data set *CFG #1*. The circles and arrows represent the states and transitions of the FSM respectively. The state with the double circle is the initial state. The symbol on each node label indicates the prediction of the network/FSM about the next word type at the corresponding state. The labels on the arrows denote the input symbols for the transitions.

for a simple recurrent neural network to learn because the hidden layer has to encode a lot of different network states. On the other hand, the extracted rules should be simple and easy to check for correctness because there are only three main types of sentences: *N-V-N-end*, *N-V-N-who-V-N-end*, and *N-who-N-V-V-N-end*. Figure 4.8 shows one extracted FSM from *CFG #1* with only 6 states. Moreover, all three aforementioned sentence types can be traced as three paths on this FSM. It appears that this is the FSM with fewest states that can mimic the dynamics of *CFG #1*.

Experimental results in Table 4.5 show that  $E_2$  helped reduced the *average* number of FSM states from 72.05 to 7.36, which is close to 6.

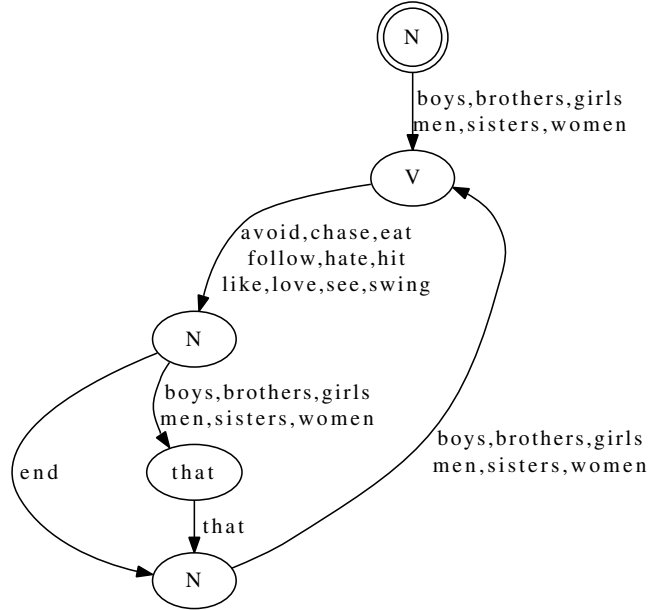


Figure 4.9: Extracted FSM for data set *CFG #2*. Same notation as in Figure 4.8.

Figure 4.9 shows an extracted FSM for *CFG #2*. The methods in creating the data sets used for training and extracting FSMs in [23] are followed exactly in order to make the results comparable. That is, the network is trained on the full grammar in Table 4.2, while the extraction in this figure is done on a *reduced* test set consisting of sentences in the form  $N-V-N-that-N-V-end$ . In addition, instead of labeling the FSM's nodes with the predictions of the exact words, the nodes are labeled with the word types  $\{N, V, end, that\}$ . For instance, a node is labeled with  $N$  if the corresponding state only predicts nouns from the grammar in Table 4.2. Note that if a state predicted both  $N$  and  $V$ , it would have been split by Step 5 of the extraction algorithm.

Figure 4.9 shows the extracted FSM from one run. The FSM is very comparable to the 6-state FSM found in [23] while having only 5 states. It gives us a very clear picture of what was learned from the data. The network learned that the initial word of a sentence was a noun  $N$  as shown in the initial state. It is always followed by a  $V$ , a  $N$ , and a *that*. Then it goes back to another  $N-V$ . Here the network predicts that the sentence will continue with another noun, while a sentence in the test set should end here. This is because there are different ways a sentence continues at this point in the *training* set: it can either continue with another  $N$  or it can end. From the FSM, we know that the network learned that a  $N$  is more likely to appear next. But the FSM also accepts an *end* at that state and begins a new sentence by predicting a  $N$  right after that. In summary, the FSM successfully modeled the dynamics in the test set. Since the design of the test set (also the data set used for FSM extraction) in [23] requires that the set consists of only one sentence form  $N-V-N-that-N-V-end$ , the extracted FSM is much simpler than if it were extracted from the full training set. The average results showed that  $E_2$  consistently helped reduce the average number of states from 73.53 to 6.2.

While Figure 4.9 shows 6 transitions, Table 4.5 shows that the average number of transitions for *CFG #2* is 44.6, which is much higher than 6. The reason is that in Figure 4.9, similar transitions are grouped together and shown as one arrow (but with very long labels), hence there are actually many transitions in the figure.

Out of 100 runs, 27 produced the best FSMs with only 5 states. While these FSMs are better than results reported in earlier studies (6 states) [23], they still

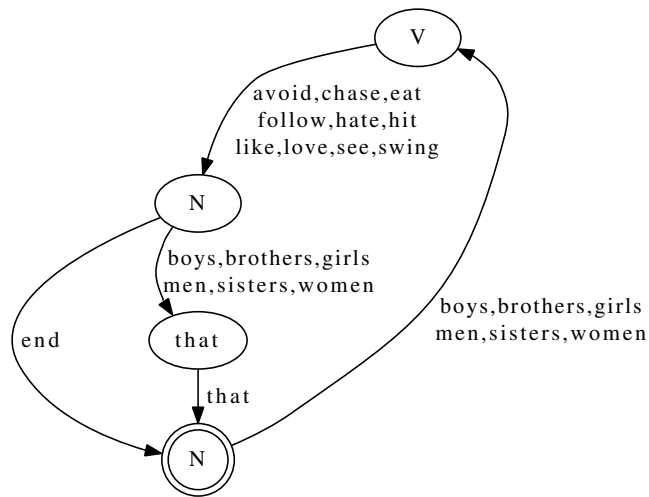


Figure 4.10: Hand-design minimal FSM for *CFG #2*. Same notation as in Figure 4.8.

have one state more than the smallest FSM a human can design by hand as shown in Figure 4.10. The only difference between the extracted FSM in Figure 4.9 and the optimal FSM in Figure 4.10 is the initial state. The extracted FSM uses the initial state as the beginning of a sentence and does not reuse it for the beginning of subsequent sentences. While it is possible for a human to recognize that the two states at the top and bottom of Figure 4.9 can be merged into one to make a simpler net, Figure 4.9 allows us to learn the way a simple recurrent network organizes its hidden layer representation, namely, using two different regions of the space for the beginnings of sentences. This limitation is likely due to the fact that a simple recurrent network's starting state is a fixed, unlearnable initial hidden activation vector, while a learned state at the beginning of a sentence must satisfy three conditions: (1) be the result of the network activation when the *end* input is given to the network while the network state is at the end of a sentence, (2) prepare the network to predict a new sentence, and (3) predict a noun. The network must

have been unable to reuse the given fixed initial hidden activation vector for the beginnings of sentences.

## 4.5 Discussion

In this chapter, a method to improve the extraction of finite state machines from simple recurrent backpropagation networks after training by adding an additional term  $E_2$  to the error function is presented. This new term encourages the formation of a more separable internal representation at the hidden layer, and is readily incorporated into the training process *while still retaining local computations*. This method is superior to similar approaches in the past in that it does not force the hidden activation vectors to be binary, thus allowing more flexible encoding at the hidden layer and making the method applicable to many complex problems.

Extensive experiments with four data sets, two from regular languages and two from context-free languages, showed that this approach consistently and substantially reduces the number of states and number of transitions of the extracted FSMs without sacrificing accuracy. While previous studies have been able to extract optimal FSMs from trained networks, they only showed the results of a few chosen runs, many of which apparently produced large, complex FSMs. To the best of my knowledge, this work is the first to demonstrate the *consistent* extraction of small, understandable FSMs across numerous runs. This is significant because it shows that this method performs well across different initial weights and data divisions, and shows that the improved results are not just the result of the *best* run.

As with many approaches using penalty terms, there is a trade-off in terms of parameter adjustments. Too large a value of  $\gamma$ , the term to control the contribution of  $E_1$  and  $E_2$ , will make the activation patterns very separated and good for FSM extraction, but it cannot keep the training error low enough. The opposite holds for too small a value of  $\gamma$ . The problem is to influence the training enough to produce the desired separation without compromising  $E_1$  for accuracy. Unfortunately, no single  $\gamma$  value has been found to work well across all data sets. Hence, it remains a parameter that has to be determined experimentally. How to optimize  $\gamma$  or make it adaptive is an important issue for future study.



## Chapter 5

### Finite State Machine Extraction from Echo State Networks

In this chapter, the term  $E_2$  and the learning rule used in the preceding chapter are modified to work on echo state networks (ESNs). Recall that ESNs consist of a hidden layer, the reservoir, whose internal nodes are recurrently connected (see Figure 5.1). These recurrent intra-reservoir connections are sparse, randomly generated, and have random fixed weights that do not change during learning. There is typically no direct connection from the input layer to the output layer. Only the weights from the reservoir to the output layer are trained, and this can be done very efficiently by linear regression. ESNs have been used successfully in many applications with temporal data, such as control problems and predicting the next items of sequences

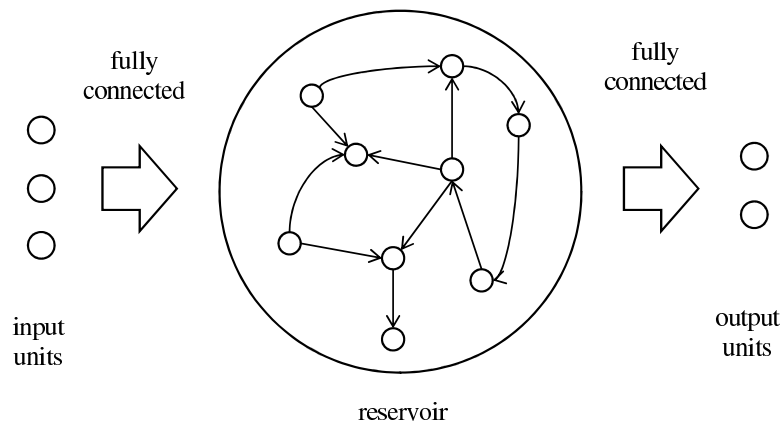


Figure 5.1: A small but otherwise typical echo state network (ESN) with 3 input units and 2 output units. Connections from input units to the reservoir and connections inside the reservoir are generated randomly and sparsely. The reservoir is fully connected to the output units only. Only these latter connections are trained so that the output comes to match the target training signals.

[42, 52].

Modifications are needed to the methods developed in previous chapters because ESNs are markedly different from simple recurrent networks in several aspects. First, recall that with ESNs, the regular sum of squared error  $E_1$  is usually *not* used for training the weights from the input layer to the hidden layer and the weights between hidden units (Section 2.3), while  $E_1$  is used for training all weights in simple recurrent networks. Thus, the term  $E_2$  can only be used by itself without the help of  $E_1$ . Second, there are large numbers of hidden units in an ESN, usually 100 or higher, compared to only a few in simple recurrent networks. This raises several issues, such as: (1) how effective  $E_2$  can be when pushing the very high-dimensional hidden activation pattern vectors away from each other; (2) whether the more separated activation patterns still cluster into groups that represent the “states” for FSMs; and (3) whether a K-means algorithm can still extract these clusters effectively. Furthermore, the “echo state” dynamics of ESNs is different from simple recurrent networks’ dynamics. For these reasons, this chapter begins with a reexamination of  $E_2$  and the derivation of a learning rule to increase  $E_2$  in ESN. A novel method for applying the learning rule to ESNs is then presented. The tests and results needed to verify the effectiveness of  $E_2$  in increasing the separability of the hidden unit activation patterns are also discussed. Finally, the computational experiment results of ESNs on the same data sets used in Section 4.3 are presented and analyzed. These experiments show the effectiveness of  $E_2$  by comparing the FSM extraction result with regular ESNs, ESNs trained with  $E_2$ , simple recurrent networks, and a new architecture *ESN+* developed by others [5, 22].

## 5.1 $E_2$ Derivation and Algorithm

### 5.1.1 The Derivation

In this section, we are concerned with ESNs as described in Section 2.3. Figure 5.1 show a small but otherwise typical echo state networks with 3 input units, 2 output units, fully connected *input*  $\rightarrow$  *hidden* and *hidden*  $\rightarrow$  *output* connections, and sparse inter-reservoir *hidden*  $\rightarrow$  *hidden* connections. There is no direct connection from the input layer to the output layer. While some variants of ESNs have optional direct connections from the input layer to the output layer, the addition of these connections does not affect the derivation below, and the methods described here will work the same. In the same way as with recurrent networks, ESNs are trained with temporal sequences: at each time step, an input pattern and a corresponding target vector (the desired correct output) are presented. If the training data has multiple input (and corresponding target) sequences, the sequences are presented successively.

We now consider how learning in an ESN can be modified so as to produce more highly separated, but hopefully still clustered, sets of activity patterns over an ESN's reservoir. First,  $E_2$  is defined similarly to  $E_2$  in previous chapters:

$$E_2 = -\frac{1}{2} \sum_{p=1}^N \sum_{q=1}^N \sum_{k \in \text{hidden}} (a_{H_k}^p - a_{H_k}^q)^2$$

where  $a_{H_k}^p$  is the activation of the  $k^{\text{th}}$  hidden unit at time step  $p$ ,  $a_{H_k}^q$  is analogous for time step  $q$ , and  $N$  is the total number of time steps over all input sequences. For instance, if the training data has two input sequences with length  $N_1$  and  $N_2$ , then

$N = N_1 + N_2$ , and  $p, q \in \{1 \dots N\}$ . Hence, there are  $N$  hidden unit activation vectors corresponding to  $N$  time steps, and  $E_2$  is the sum over all  $(N^2/2)$  squared Euclidean distances between these  $N$  hidden unit activation vectors. Let  $E_2 = \sum_p E_2^p$  where:

$$E_2^p = -\frac{1}{2} \sum_{q=1}^N \sum_{k \in \text{hidden}} (a_{H_k}^p - a_{H_k}^q)^2$$

Unlike with feedforward and simple recurrent networks,  $a_{H_k}$  of ESNs are calculated differently as in Equation 2.3:

$$a_{H_j}^t = \sigma(\text{in}_{H_j}^t) = \sigma \left( \sum_{k \in \text{hidden}} w_{jk}^{\text{res}} a_{H_k}^{t-1} + \sum_{k \in \text{input}} w_{jk}^{\text{in}} x_k^t \right) \quad (5.1)$$

where  $w_{jk}^{\text{res}}$  and  $w_{jk}^{\text{in}}$  are the intra-reservoir weights between hidden units in the reservoir and the weights from the input layer to the reservoir, respectively, and  $x_k^t$  is the value of the  $k^{\text{th}}$  input unit at time step  $t$ . We need to calculate  $\frac{\partial E_2^p}{\partial w_{ji}^{\text{res}}}$  for  $i \in \text{hidden}$  and  $\frac{\partial E_2^p}{\partial w_{ji}^{\text{in}}}$  for  $i \in \text{input}$ . This derivation follows, and is similarly to that in Section 3.1 until Equation 3.2. Now, for  $i \in \text{input}$ , we have:

$$\begin{aligned} \frac{\partial a_{H_j}^p}{\partial w_{ji}^{\text{in}}} &= \frac{\partial(\sigma(\text{in}_{H_j}^p))}{\partial w_{ji}^{\text{in}}} \\ &= \frac{d\sigma(\text{in}_{H_j}^p)}{d\text{in}_{H_j}^p} \frac{\partial \text{in}_{H_j}^p}{\partial w_{ji}^{\text{in}}} \\ &= a_{H_j}^p (1 - a_{H_j}^p) \left( \sum_{k \in \text{hidden}} \frac{\partial w_{jk}^{\text{res}} a_{H_k}^{p-1}}{\partial w_{ji}^{\text{in}}} + \sum_{k \in \text{input}} \frac{\partial w_{jk}^{\text{in}} x_k^p}{\partial w_{ji}^{\text{in}}} \right) \end{aligned}$$

We have  $\sum_{k \in \text{input}} \frac{\partial w_{jk}^{\text{in}} x_k^p}{\partial w_{ji}^{\text{in}}} = x_i^p$  because  $\frac{\partial w_{jk}^{\text{in}}}{\partial w_{ji}^{\text{in}}} = 0$  with  $i \neq k$ . Note that  $x_i^p$  is a constant because it is the input at time step  $p$ . It is typical not to propagate the error beyond one time step [40], so we can assume  $a_{H_k}^{p-1}$  is a constant here. Consequently,  $\sum_{k \in \text{hidden}} \frac{\partial w_{jk}^{\text{res}} a_{H_k}^{p-1}}{\partial w_{ji}^{\text{in}}} = 0$  since  $w_{ji}^{\text{res}}$  is always treated as a constant when doing a partial

derivative with respect to  $w_{ji}^{in}$ . Thus, for weights between input and hidden nodes:

$$\frac{\partial a_{H_j}^p}{\partial w_{ji}^{in}} = a_{H_j}^p (1 - a_{H_j}^p) x_i^p$$

Combined with Equation 3.2, we have:

$$\frac{\partial E_2^p}{\partial w_{ji}^{in}} = -N(a_{H_j}^p - \overline{a_{H_j}}) a_{H_j}^p (1 - a_{H_j}^p) x_i^p \quad (5.2)$$

This result is identical to Equation 3.1 for simple recurrent networks in the preceding chapter. Turning to  $\frac{\partial E_2^p}{\partial w_{ji}^{res}}$  for  $i \in hidden$ , we have:

$$\frac{\partial a_{H_j}^p}{\partial w_{ji}^{res}} = \frac{\partial(\sigma(in_{H_j}^p))}{\partial w_{ji}^{res}} \quad (5.3)$$

$$\begin{aligned} &= \frac{d\sigma(in_{H_j}^p)}{din_{H_j}^p} \frac{\partial in_{H_j}^p}{\partial w_{ji}^{res}} \\ &= a_{H_j}^p (1 - a_{H_j}^p) \left( \sum_{k \in hidden} \frac{\partial w_{jk}^{res} a_{H_k}^{p-1}}{\partial w_{ji}^{res}} + \sum_{k \in input} \frac{\partial w_{jk}^{in} x_i^p}{\partial w_{ji}^{res}} \right) \end{aligned} \quad (5.4)$$

We have  $\sum_{k \in hidden} \frac{\partial w_{jk}^{res} a_{H_k}^{p-1}}{\partial w_{ji}^{res}} = a_{H_i}^{p-1}$  because  $\frac{\partial w_{jk}^{res}}{\partial w_{ji}^{res}} = 1$  with  $i = k$  and  $\frac{\partial w_{jk}^{res}}{\partial w_{ji}^{res}} = 0$  otherwise. Also,  $\sum_{k \in input} \frac{\partial w_{jk}^{in} x_i^p}{\partial w_{ji}^{res}} = 0$  because the numerator does not have any  $w_{ji}^{res}$  component. Thus,

$$\frac{\partial a_{H_j}^p}{\partial w_{ji}^{res}} = a_{H_j}^p (1 - a_{H_j}^p) a_{H_i}^{p-1}$$

Combined with Equation 3.2, we have:

$$\frac{\partial E_2^p}{\partial w_{ji}^{res}} = -N(a_{H_j}^p - \overline{a_{H_j}}) a_{H_j}^p (1 - a_{H_j}^p) a_{H_i}^{p-1} \quad (5.5)$$

Equation 5.5 is slightly different from its counterpart in feedforward networks in that the last term on the right side is  $a_{H_i}^{p-1}$ , the activation of the hidden unit at the

preceding time step; whereas the last term is the input attribute  $x_i^p$  for feedforward networks. As in previous chapters, we only need  $\overline{a_{H_j}}$ , which can be computed and stored locally at the  $j^{\text{th}}$  hidden unit, in addition to the activations of the units and its inputs in order to compute  $\frac{\partial E_2}{\partial w_{ji}}$  and  $\frac{\partial E_2}{\partial w_{ji}^{\text{in}}}$ . This local property is highly desired in neural network training.

It is important to note that for ESNs, the regular error function  $E_1$  is only used to train the weights from the reservoir to the output layer, while  $w_{ji}^{\text{in}}$  and  $w_{ji}^{\text{res}}$  are initialized randomly but are *not* trained with  $E_1$ . This allows ESNs to be trained efficiently by linear regression because only one layer of weights has to be trained, i.e., there is no propagation of error signals. Thus,  $\frac{\partial E_2}{\partial w_{ji}^{\text{res}}}$  and  $\frac{\partial E_2}{\partial w_{ji}^{\text{in}}}$  are the only gradients used for training  $w_{ji}^{\text{in}}$  and  $w_{ji}^{\text{res}}$ . In addition, Equations 5.2 and 5.5 show that the calculation of the gradients does not involve looking at the *target* signal  $T^t$ . Therefore, the process of increasing  $E_2$  of an ESN is effectively unsupervised learning. I use “unsupervised” here because  $E_1$ , which measures performance error, is not used.

Not training with  $E_1$  presents an important problem with this method:  $w_{ji}^{\text{res}}$  and  $w_{ji}^{\text{in}}$  can potentially grow very large, thus making the network unstable and not amenable to generalizing well. Indeed, having large weights has been one of the main sources of poor generalization, and there has been a lot of work (e.g., regularization, resulting in weight decay) to tackle the problem [46, 63]. Exploratory experiments confirmed that  $w_{ji}^{\text{res}}$  and  $w_{ji}^{\text{in}}$  grew large very quickly. Therefore, weight decay as described in Section 3.2 (paragraph 3) is used to keep the magnitude of the weights

relatively small based on an additional error term that is to be minimized:

$$E_d = \frac{\lambda}{2} \sum_{j \in \text{hidden}} \sum_{i \in \text{input}} (w_{ji}^{\text{in}})^2 + \frac{\varphi}{2} \sum_{j \in \text{hidden}} \sum_{i \in \text{hidden}} (w_{ji}^{\text{res}})^2$$

where  $\lambda$  and  $\varphi$  are coefficients determining the amount of decay. It can be easily calculated that  $\frac{\partial E_d}{\partial w_{ji}^{\text{in}}} = \lambda w_{ji}^{\text{in}}$  and  $\frac{\partial E_d}{\partial w_{ji}^{\text{res}}} = \varphi w_{ji}^{\text{res}}$ . The combined gradient of  $E$  with respect to  $w_{ji}^{\text{in}}$  and  $w_{ji}^{\text{res}}$  are:

$$\begin{aligned} \frac{\partial E_2^p}{\partial w_{ji}^{\text{in}}} &= -N(a_{H_j}^p - \overline{a_{H_j}}) a_{H_j}^p (1 - a_{H_j}^p) x_i^p + \lambda w_{ji}^{\text{in}} \\ \frac{\partial E_2^p}{\partial w_{ji}^{\text{res}}} &= -N(a_{H_j}^p - \overline{a_{H_j}}) a_{H_j}^p (1 - a_{H_j}^p) a_{H_i}^{p-1} + \varphi w_{ji}^{\text{res}} \end{aligned}$$

Besides the sigmoid function, *tanh* is another widely used activation function for ESNs. The derivation above also works with the *tanh* activation function, with only a small change from  $a_{H_j}^p(1 - a_{H_j}^p)$  to  $(1 - (a_{H_j}^p)^2)$  because  $d(\tanh(x))/dx = (1 - \tanh^2(x))$ . So, if the hidden units in the reservoir use the *tanh* activation functions:

$$\begin{aligned} \frac{\partial E_2^p}{\partial w_{ji}^{\text{in}}} &= -N(a_{H_j}^p - \overline{a_{H_j}}) (1 - (a_{H_j}^p)^2) x_i^p \\ \frac{\partial E_2^p}{\partial w_{ji}^{\text{res}}} &= -N(a_{H_j}^p - \overline{a_{H_j}}) (1 - (a_{H_j}^p)^2) a_{H_i}^{p-1} \end{aligned}$$

However, in the remainder of this chapter, only the sigmoid activation is used to be consistent and comparable with previous chapters and relevant work by others.

### 5.1.2 Algorithm to Decrease $E_2$ in an ESN

We use gradient descent to decrease  $E_2$  (i.e., to increase the separation of reservoir patterns) by making the weight changes  $\Delta w_{ji}^{\text{in}} = -\eta \frac{\partial E_2}{\partial w_{ji}^{\text{in}}}$  and  $\Delta w_{ji}^{\text{res}} = -\eta \frac{\partial E_2}{\partial w_{ji}^{\text{res}}}$ . The algorithm can be summarized as follows:

Initialize  $w^{in}$  and  $w^{res}$  randomly

Scale  $w^{res}$  so that  $\rho(w^{res})$  is equal to a specified spectral radius

**for**  $iteration = 1$  to  $num\_iteration$  **do**

calculate  $\overline{a_{H_j}}$  for  $j \in reservoir$

**for**  $j \in reservoir, i \in input, t$  time steps **do**

$$\Delta w_{ji}^{in} = \eta N(a_{H_j}^t - \overline{a_{H_j}}) a_{H_j}^t (1 - a_{H_j}^t) x_i^t - \lambda w_{ji}^{in}$$

$$w_{ji}^{in} = w_{ji}^{in} + \Delta w_{ji}^{in}$$

$$\Delta w_{ji}^{res} = \eta N(a_{H_j}^t - \overline{a_{H_j}}) a_{H_j}^t (1 - a_{H_j}^t) a_{H_i}^{t-1} - \varphi w_{ji}^{res} \quad *$$

$$w_{ji}^{res} = w_{ji}^{res} + \Delta w_{ji}^{res} \quad *$$

**end for**

**end for**

Pilot experiments with both data sets *CFG #1* and *CFG #2* found that changes made to  $w_{ji}^{res}$  by the algorithm only caused negligible quantitative change in  $E_2$  and finite state machine extraction. A possible explanation is that the spectral radius scaling procedure has made the changes ineffective. In particular, recall that in order for ESNs to possess the “echo” state property, the spectral radius  $\rho(W)$  (the largest eigenvalue) of the reservoir weight matrix has to be smaller than 1, and the closer that the spectral radius is to 1 the longer in time that the reservoir dynamics “remembers”. Besides, we have  $\rho(kW) = k\rho(W)$  where  $k$  is a scalar constant. Therefore, this property is usually enforced by scaling the weights between units in the reservoirs so that the spectral radius is always equal to a chosen constant, commonly in  $[0.8, 0.95]$ . In regular ESNs, this procedure is only used for initialization



because the intra-reservoir weights are not changed. Here, the intra-reservoir weights are changed by this algorithm, so the procedure has to be applied either after each iteration, or after all iterations have been complete in order to maintain the “echo” state property of the networks. Neither of these methods was effective for  $E_2$  and FSM extraction. Hence, the final algorithm is modified so that it does not change  $w_{ji}^{res}$ , i.e. the two lines marked with a (\*) are removed. This also fits with the traditional spirit of ESNs of not changing  $w_{ji}^{res}$  at all during learning.

The algorithm starts by initializing the weights from the input layer to the reservoir and the weights between units inside the reservoir according to typical ESN initialization methods. Then, for each iteration, the average activations of units in the reservoir  $\overline{a_{H_j}}$  are computed and used to calculate the weight change  $\Delta w_{ji}^{in}$  for weight  $w_{ji}^{in}$  of the connection from the  $i^{th}$  input unit to the  $j^{th}$  hidden unit in the reservoir using Equation 5.2. The weight change  $\Delta w_{ji}^{in}$  is then augmented with the weight decay. In this weight change,  $\eta$  is the learning rate and  $\lambda$  is the weight decay coefficient. As with other gradient descent methods, a large learning rate  $\eta$  makes learning fast but unstable while a small learning rate makes learning occur too slowly. Experimental results show that setting  $\eta = 0.01$ ,  $\lambda \in \{0.2, 0.4\}$  (data set dependent), and  $num\_iteration = 60$  generally gives a good balance between learning speed and quality of the final result. The parameter  $num\_iteration$  was determined using pilot runs by observing the performance of the network and FSM extraction when successive iterations were applied. If  $num\_iteration$  is too small, the algorithm does not decrease  $E_2$  enough, and thus extracts large FSMs. Setting  $num\_iteration$  larger than 60 does not improve the results further, and in some cases,

reduces the accuracy rates at the outputs. This behavior is largely similar to the tradeoff of  $\alpha$  and  $\beta$  for feedforward networks and  $\gamma$  for simple recurrent networks.

## 5.2 Experimental Methods

This section describes the experimental methods used to evaluate the effectiveness of training ESNs with  $E_2$ . First, it is pointed out that the two data sets *CFG #1* and *CFG #2* already used in the previous chapter can be reused, and that allows one to compare the results here with simple recurrent network results. Next, the experimental setting is presented. Lastly, a method to normalize  $E_2$  for better quantitative evaluation is discussed and explained.

The main experiment compares FSM extraction from regular ESNs (control condition) versus from ESNs trained with  $E_2$  (experimental condition). The hypothesis is that it is possible to make ESNs use a different encoding at the reservoir that facilitates the extraction of simpler FSMs, while maintaining the performance in terms of sum of squared error at the output layer.

### 5.2.1 Data Sets

The same FSM extraction procedure as that used in Section 4.2 can be used to extract FSMs from ESNs because ESNs and simple recurrent networks have similar dynamics. Namely, the ESNs' hidden activation vectors also represent the internal reservoir' "state" of the ESN, and both types of networks process temporal sequences. For these reasons, it would be logical to use the same data sets already used to

evaluate  $E_2$  on simple recurrent networks to evaluate  $E_2$  on ESNs. However, the data sets *badigu* and *Tomita #4* cannot be used with ESNs for several reasons. First, I discovered that the extraction algorithm is already able to consistently extract FSMs with only 7 states, the minimal number of states as discussed in Section 4.3, from regular ESNs (control condition, 30-unit reservoir, sum of squared error is similar to that reported in the same section). Thus,  $E_2$  would not be able to improve any further using the methods derived above. This is a surprising result: how can randomly initialized networks have a perfect internal representation space? The reason for this lies in one important property of ESNs: the reservoir state (the hidden unit activation vector) of an ESN is completely determined by a *long enough* sequence of input, and is *not* dependent on the initial state. For instance, the states of an ESN after receiving two sequences:

- $d_1, d_2, \dots, d_k, s_1, s_2, \dots, s_m$
- $d'_1, d'_2, \dots, d'_k, s_1, s_2, \dots, s_m$

are identical if  $m$  is large enough. Therefore, the number of ESN states is limited to the number of these long-enough  $s_1, \dots, s_m$  sequences. For the *badigu* data set, it is likely that the simplicity of the language allows the reservoir states to be completely determined by relatively short sequences. In addition, it is obvious that there are only a limited number of short sequences because strings in this language are comprised of only three subsequences: *ba*, *dii*, and *guuu*. Hence, there are only a limited number of reservoir states. In brief, the number of states of an ESN with the *badigu* data set is small by its own nature.

The *Tomita #4* data set is unsuitable for ESN learning because the correctness of the predictions required by this data set rely on the complete input strings (i.e., whether the string has “000” in it or not) whereas ESNs *have to* ignore the beginnings which may or may not contain the string “000”. Thus, ESNs fail to process this data set correctly. It is however possible to modify the sequence generation process so that no string “000” occurs in the ignored prefixes of the sequences so that a network is always in A(ccept) states after it has processed the beginnings. In that case, the reasoning in the preceding paragraph for the data set *badigu* also holds true for this data set, hence the number of states of an ESN with this data set would also be small by its own nature.

For these reasons, only *CFG #1* and *CFG #2* are used to evaluate FSM extraction from ESNs trained with and without  $E_2$ . As these two data sets are much more difficult than *badigu* and *Tomita #4*, the omission of *badigu* and *Tomita #4* does not significantly weaken the evaluation.

## 5.2.2 Experimental Settings

As presented in Section 2.3, the ESN training algorithm is a linear regression problem that does not utilize a validation set. Hence, 10-fold cross validation is not used as in the previous two chapters. However, it is necessary to run the experiments multiple times under different condition to ensure that any difference in the control condition results and the experimental condition results are not due to chance. For that reason, cross validation is replaced by using 10 different sets of training and

separate test data created randomly with different random seeds. Each set of data was given to 10 different randomly generated ESNs (i.e., random initial weights). Furthermore, to ensure the validity of the comparisons, identical starting points (initial random weights) and data are given to the corresponding control and the experimental runs.

For all data sets, the settings for the experimental and control runs are identical except for the particular learning algorithm to be tested in each experiment:

- Reservoir connectivity is 15%: Each unit in the reservoir connects randomly to approximately 15% other units.
- 50 and 100 units are used in the reservoir for *CFG #1* and *CFG #2*, respectively.
- $\lambda$  is set to 0.2 and 0.4 for *CFG #1* and *CFG #2*, respectively.
- $\eta$  is set to 0.01.
- *num\_iteration* is set to 60.
- Spectral radius is scaled to 0.95.
- 20 time steps at the beginning of the training sequences are used to stabilize the network before training commences.

These are typical settings employed by previous work [39, 52].

The experiments are implemented in C++ and run single threaded on a Core i7 3.4 GHz CPU. The average time for each run ranges from 1 to 2 minutes depending on the data set and experiment settings. PCA analysis (Section 5.3.2.1) is done

in Octave [16], an open-source numerical computation software environment very similar to Matlab.

### 5.2.3 $E_2$ Normalization

In the following experimental section, we need a quantitative measure to compare the separability of the hidden unit activation patterns. Unlike in the preceding chapters with feedforward and recurrent networks where  $E_2$  was used both to drive the algorithm and to evaluate the separability of the hidden unit activation patterns,  $E_2$  alone cannot be used to quantify the separability of hidden unit activation patterns in ESNs because it is possible to decrease  $E_2$  *without* making the patterns more amenable to clustering. As an example, consider Figure 5.2 that shows two 2-dimensional made-up hidden unit activation space patterns with 16 hidden activation vectors each. The activations are limited to  $[0,1]$  because of the

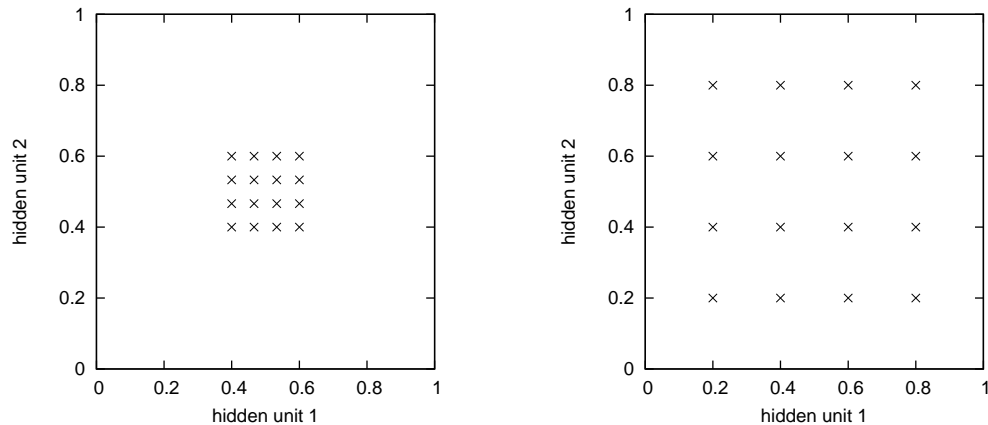


Figure 5.2: Two sets of simple made-up hidden unit activation space patterns with two hidden units. Each cross represents one hidden unit activation pattern. While the value of  $E_2$  for the patterns on the right is much smaller (more negative) than the value for the space on the left, it is equally difficult to cluster the two spaces because all patterns are equally distributed.

sigmoid activation function. Obviously, the one on the right has a smaller  $E_2$  (i.e., a larger sum of squared distance, since  $E_2$  is defined as the negation of the sum) because the vectors are much further from each other. However, from a clustering perspective, the one on the right is as difficult to cluster as the one on the left because it also has 16 evenly distributed points. In other words, the one on the right is not *qualitatively* better than the one on the left in terms of how distributed or separated the activations are. As a result, if a method only scales the space so that the activations are closer to the boundaries, it would reduce  $E_2$  without making the space any better. For that reason,  $E_2$  alone is not a good measure for how separated an activation space is. In this section, I will use a measure called *normalized  $E_2$* , designated  $E'_2$ , to quantitatively assess how amenable to clustering a set of activity patterns are in the hidden activation space. In order to compute  $E'_2$ , the hidden activation vectors' components are first linearly scaled so that each component's maximum and minimum value is 1 and 0, respectively. Next,  $E'_2$  is calculated as the negation of the sum of squared distances between these scaled vectors. Thus,  $E'_2$  is invariant to scaling, i.e. if an algorithm only scales the hidden activation patterns evenly similar to what is shown in Figure 5.2, it would make no change to  $E'_2$ .

Being a gradient descent-based method, the algorithm that reduces  $E_2$  presented in this work changes the weights in any way that reduces  $E_2$ ; hence, it would: (1) make the activation vectors further away from each other; and (2) scale the activation vectors as discussed above. Hence, the quantitative reduction in  $E_2$  reflects the result of *both* (1) and (2), while we are only interested in how much the activation vectors are further apart (1). In contrast,  $E'_2$  allows us to assess quantitatively how effective

the algorithm is in accomplishing (1) alone.

Note that this was not a problem with feedforward networks (Chapter 3) and simple recurrent networks (Chapter 4) because  $E_1$  was used at the same time  $E_2$  was used to keep the activation patterns together. Namely, if a certain method only scales to make the activation patterns closer to the boundaries, the sum of squared error  $E_1$  would grow very large. Therefore, it was not necessary to use  $E_2'$  in Chapters 3 and 4.

### 5.3 Results

This section first presents the main experiment comparing FSM extraction from regular ESNs (control condition) versus from equivalent ESNs trained with  $E_2$  (experimental condition). Next, a battery of tests and analyses are presented to verify that the decrease in  $E_2$  actually results in improving the separability into clusters of the activity patterns in the hidden activation space. These results are also compared and contrasted with results of extracting FSMs from a variant of ESN called ESN+ [22].

Table 5.1:  $E_1$  and  $E_2$  of Regular versus Modified ESN (Averaged over 100 Runs)

Data set	$E_1/N$		$-2E_2'/N^2$	
	regular	new	regular	new
CFG #1	0.417	0.410	9.132	12.383 (+36%)
CFG #2	0.904	0.835	14.478	22.165 (+53%)



### 5.3.1 Regular ESN versus ESN with $E_2$

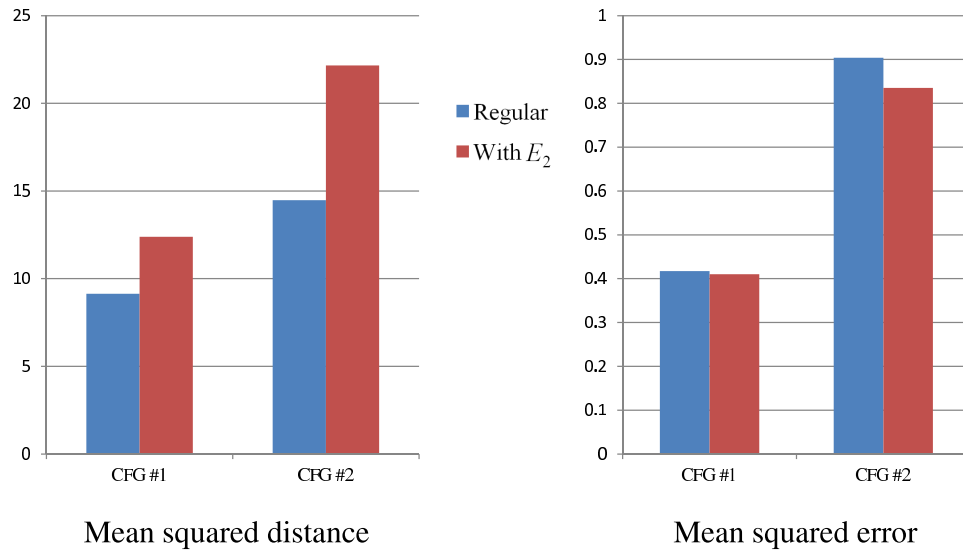


Figure 5.3: Mean squared distance and mean squared error of regular ESNs and ESNs trained with  $E_2$ .

Table 5.1 and Figure 5.3 show the effects of the new error term  $E_2$  on the network’s average testing error ( $E_1/N$ ) and the average squared distances among hidden unit activation vectors ( $-2E_2/N^2$ ) where  $N$  is the total number of time steps of all training sequences. A network’s average testing error and average squared distance between hidden activation vectors are shown after training regular ESNs (control condition) versus after training ESNs improved by  $E_2$  (experimental or “new” condition). Both sets of experiments use the same learning rate  $\eta = 0.01$  and are run for 60 epochs. Since  $E_2$  is 1/2 of the negated sum of squared distances between each pair (total  $N^2$  pairs) of activations,  $-2E_2/N^2$  is the average squared distance between each pair. The *regular* columns show results with regular ESNs. The *new* columns show results with ESNs augmented with  $E_2$ . These data show that activation pattern distances were increased up to 53% with only a small change in

network errors for *CFG #1*, and a notable 9% *reduction* in error for *CFG #2* from 0.904 to 0.835. Paired t-tests found that all changes in Table 5.1 are statistically significant (p-value  $< 2.2 \times 10^{-16}$ ). The small change in  $E_1$  for *CFG #1* and the reduction of  $E_1$  for *CFG #2* support the hypothesis that it is possible to make error backpropagation learn a different encoding that satisfies other criteria (smaller  $E_2$ ) while still maintaining near minimum error at the output layer.

The improvement in  $E_1$  for *CFG #2* is completely unexpected. It is the first experiment in which modifying  $E_2$  has caused any statistically significant changes in  $E_1$  (paired t-test gives  $p < 2.2 \times 10^{-16}$ ). To understand this, we compare the mean  $E_1/N$  for regular ESNs (0.904) with mean  $E_1/N$  for simple recurrent networks in Table 4.4 (0.752). It is clear that regular ESNs perform worse than simple recurrent networks in terms of sum of squared errors. The poor performance of ESNs on this data set has been observed previously in [23]. In that work, the authors compared three architectures ESN, ESN+, and Markov models on the same data that is used here. Their results showed that ESN+ performs “at least as well as Markov models” ([23]), while regular ESN performed much worse than both. ESN+ is a variation of ESN proposed in [5, 23] specifically for this data set. In the next section, I will present and discuss more results comparing ESN+ with ESN improved with  $E_2$ .

Table 5.2: Means ( $\sigma$ ) of Number of States and Transitions over 100 Runs

Data set	No. of states		No. of transitions	
	regular	new	regular	new
CFG #1	82.42 (10.3)	21.40 (3.5)	293.95 (28.4)	66.00 (12.2)
CFG #2	192.03 (51.2)	6.74 (4.8)	380.00 (63.3)	42.64 (16.3)

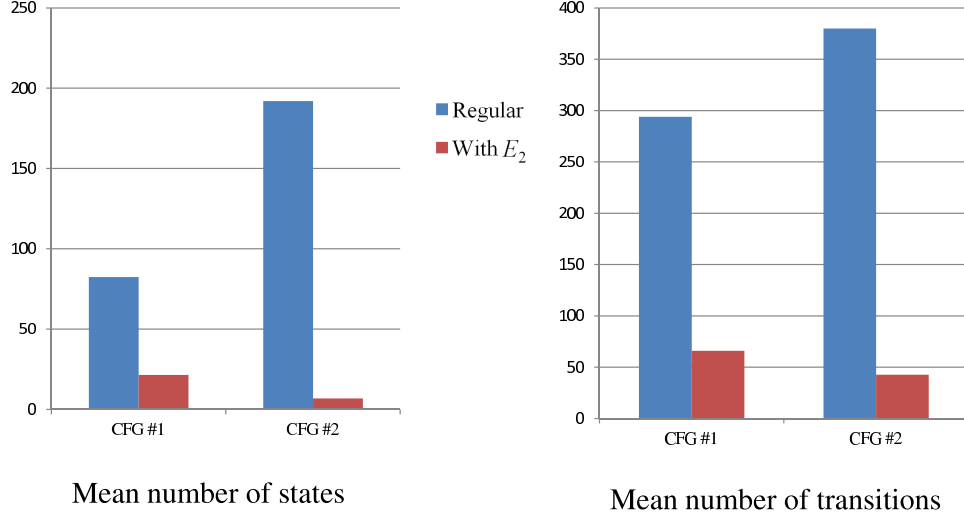


Figure 5.4: Mean number of states and transitions of finite state machines extracted from regular ESNs and ESNs trained with  $E_2$ .

Table 5.2 and Figure 5.4 present the experimental results concerning FSM extraction. The new error term helped to reduce the numbers of states and standard deviations significantly for both data sets. The control experiments extracted very large FSMs with hundreds of states to explain what the ESNs have learned. Similar large numbers of states were previously reported in [23] when the authors extracted FSMs from regular ESNs trained on *CFG #2*.

The large standard deviations in Table 5.2 for the regular/control runs raise the question of whether the reduction in average number of states is statistically significant. For this purpose, paired  $t$ -tests with a standard significance level of 0.05 are used. Bonferroni correction [20] for 2 tests requires the significance to be defined as  $p < 0.05/2 = 0.025$ . We found that the corresponding p-values for both pair of experiments are below  $2.2 \times 10^{-16}$ . These tests confirmed that the improvement from  $E_2$  is statistically significant. Table 5.2 (right two columns) shows that the numbers of transitions and the standard deviations are also markedly reduced by using  $E_2$ .

Even though the mean number of states for *CFG #1* has been reduced significantly, the mean is still relatively high compared to the average of 7.36 achieved with simple recurrent networks, and the best run produced an FSM with 18 states. While this is a significant improvement to the hundreds of states in regular ESN and indicates that the activation space is a lot more separated, the final FSM is still too complicated for a human to read and understand what the network has learned.

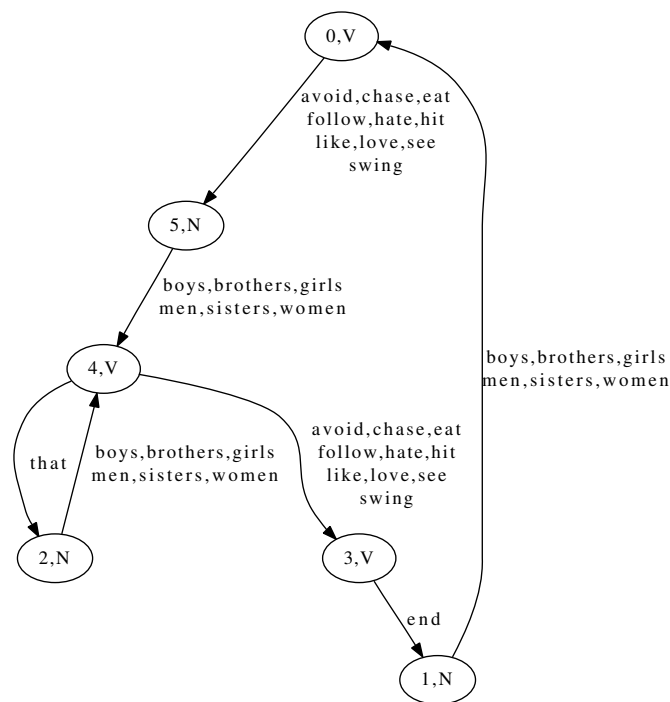


Figure 5.5: Extracted FSM from ESN for data set *CFG #2*. Same notation as in Figure 4.8 except that node labels also contain a state number.

For *CFG #2*, in order to compare results with [23] and with the simple recurrent networks of Chapter 4, the same method in creating the data sets is used for training and extracting FSMs as was used for *CFG #2* in Chapter 4. That is, the network is trained on the full grammar in Table 4.2, while the extraction in this figure is done on a *reduced* test set consisting of sentences in the form *N-V-N-that-N-V-end*. Figure

5.5 shows an extracted FSM for *CFG #2* with 6 states. It is the FSM with the smallest number of states the method was able to extract. Unlike the FSM extracted from simple recurrent network in Figure 4.9, extracted FSMs from ESNs do not have initial states. The reason is that the beginning of each training sequence must be used to stabilize the network and then be discarded, so the first recorded state is usually some state in the middle of a sentence. Nevertheless, it is clear that state #1 encodes the beginning of each sentence because it is reached by an *end* transition. As we follow the FSM from this state #1, we see that it makes the correct predictions at states #1, #0, and #5, but makes a mistake of predicting a V at state #4. This is wrong because the subsequence *N-V-N-V* does not appear in the training data. If we continue to follow the sentence through state #2 and back to #4, the predictions are correct. At state #3, the FSM predicts a V while we expect an *end*. However, this is not a misprediction because the subsequence *N-V-V* appears very often in the data. Compared to the FSM extracted from a simple recurrent network in Figure 4.9, this FSM makes more mistakes. This is not unexpected because of the high sum of squared error  $E_1$  as reported in Table 5.1. Yet the extraction result has allowed us to see exactly where the network made the mistakes, in terms of how it organizes its internal representations to make the prediction.

In [23], the authors showed that the same extraction method produced an FSM with the same number of states (6) from ESN+, but the FSM does not make the mistake pointed out in the preceding paragraph. It was also reported that a very large FSM with 190 states was extracted from a regular ESN. Unlike the work presented here, [23] did not use multiple runs, so it is not possible to compare the

average results.

### 5.3.2 Verification of Separation into Clusters

With simple recurrent networks as in Chapter 4, the hidden layer contains a small number of hidden units, usually 3 or smaller. This allows us to visualize the hidden activation space in 2 or 3 dimensional graphs, such as the examples in Section 3.3 and Section 4.4.1. In contrast, the number of hidden units in the reservoirs of ESNs is usually very large. Here, principal component analysis, histograms of distances, and a test with weight redistribution are used to inspect the new hidden activation space patterns created with the help of  $E_2$ , and to verify that the method indeed makes the reservoir activation space patterns more separable.

#### 5.3.2.1 Principal Component Analysis

Principal component analysis (PCA) [15, 59] is one of the most widely used tools for dimensionality reduction. PCA finds a linear transformation of the original data so that the first dimension is in the direction of maximum variance, and subsequent dimensions have the next highest variances. By taking just the first few PCA dimensions, we have a projection of the data onto a space with fewer dimensions while preserving much of the variance in the data. This allows us to visualize the high-dimensional hidden activation space and the qualitative effect of  $E_2$  on separability of the hidden activation space patterns. Furthermore, one can calculate how much variance is preserved in each new projected dimension, and thus

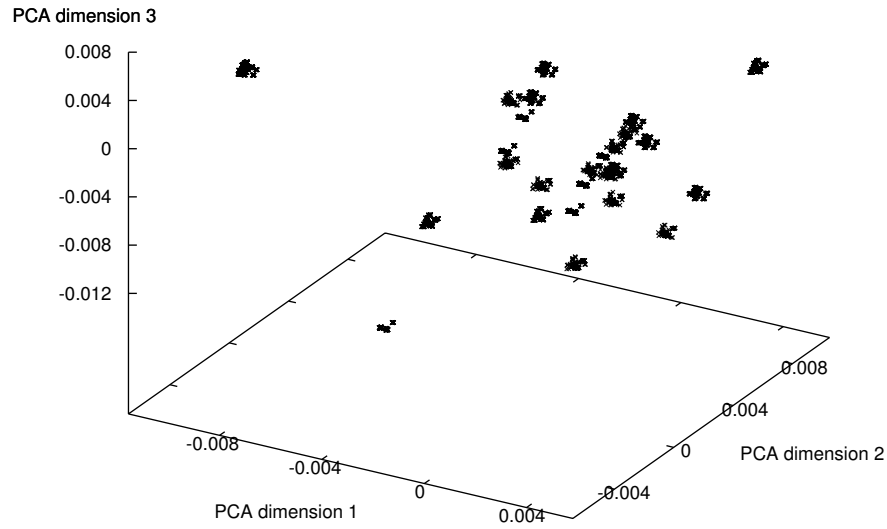


Figure 5.6: A typical projection of the hidden activation patterns of a regular ESN having 100 hidden units trained on the *CFG #2* data set onto a 3 dimensional space using PCA. The projected activation vectors appear to form many loose clusters.

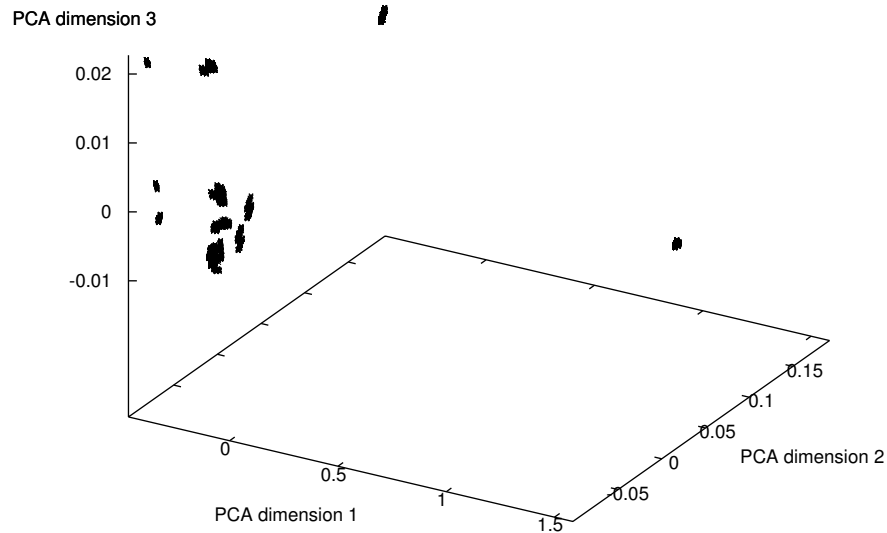


Figure 5.7: A typical projection of the hidden activation patterns of an ESN having 100 hidden units augmented with  $E_2$  trained on the *CFG #2* data set onto a 3 dimensional space using PCA. The projected activation vectors appear to form a small number of easily distinguished clusters.

can determine how close the transformed low dimensional space approximates the data.

Figure 5.6 shows the projection of a hidden activation space of a regular ESN having 100 hidden units trained on the *CFG #2* data set onto a 3 dimensional space. While most of the vectors tend to cluster together, there are a lot of clusters and some of the clusters are not separated clearly. Moreover, only 30% of the variance is retained by this projection. Thus, the underlying data is even more complex and much more difficult to cluster.

Figure 5.7 shows the projection of a hidden activation space of an ESN with identical initial weights but augmented with  $E_2$  on the same data set. There are many fewer clusters and all of the clusters are cleanly separated. Furthermore, 99.9% of the variance is retained using just the first 3 dimensions compared to only 30% in Figure 5.6 for a regular ESN. This indicates that this reduced dimensional space reflects the original (non-PCA) activation space very accurately.

It is interesting to note that the distances between the projected points in Figure 5.7 are much longer than the distances in Figure 5.6 by looking at the scale of the axes. Recall that PCA projection preserves the total variance in the data, which is equivalent to the sum of squared distance to the multidimensional mean vector. Thus, the longer distances between points in Figure 5.7 indicate that the hidden unit activation vectors in the original space are also further apart. In summary, the original (non-PCA) activation space patterns are very separable. This indicates that training with  $E_2$  has indeed made the vectors in the hidden activation space better separated and easier for clustering.



### 5.3.2.2 Histograms of Distances

While it is not possible to visually look at the 100-dimensional hidden unit activation space, we can indirectly inspect the distances between the individual vectors. If a set of patterns are very distributed, most of the distances will be longer and there would be much variation in the distances. If a space consists of a small number of tight clusters, there would be two different sets of distances: (1) the distances between the vectors inside the same cluster should be relatively small; (2) the distances between the vectors in different clusters should be relatively large. In addition, if  $a_1, a_2$  are in cluster  $A$  and  $b_1, b_2$  are in cluster  $B$ , the distances  $D(a_1, b_1)$  and  $D(a_2, b_2)$  of two pairs of vectors  $(a_1, b_1)$  and  $(a_2, b_2)$  should be very similar to the

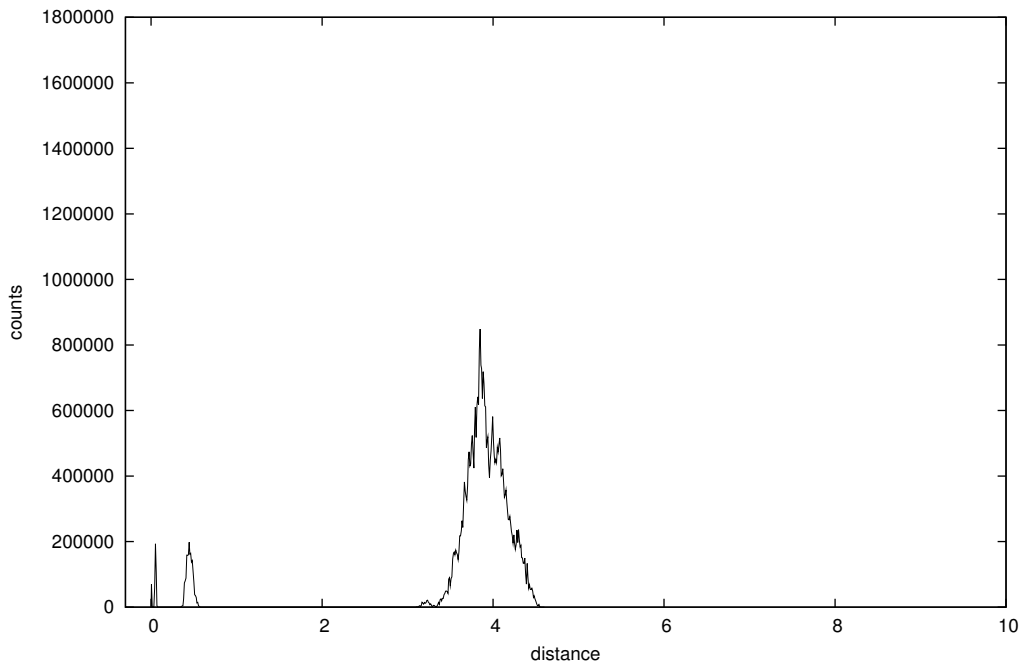


Figure 5.8: A typical histogram of normalized distances between hidden unit activation vectors of a regular ESN trained on the data set *CFG #2*. The histogram exhibits a mostly Gaussian-like distribution indicating that the distances between hidden unit activation vectors are largely random.

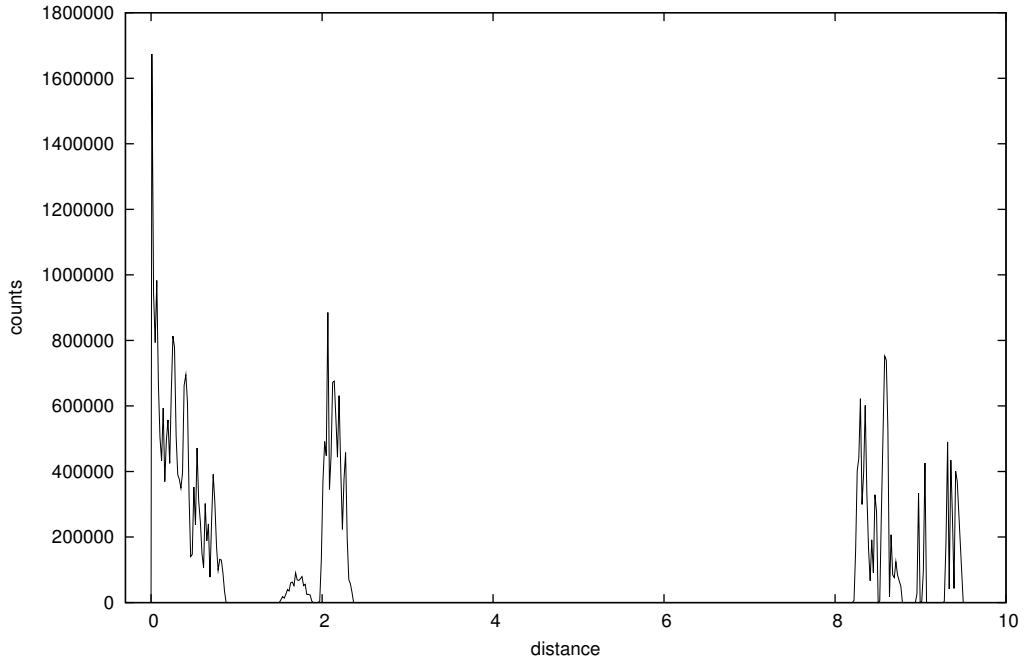


Figure 5.9: A typical histogram of normalized distances between hidden unit activation vectors of an ESN augmented with  $E_2$  trained on the same data set as in Figure 5.8. Compared with Figure 5.8, there are many more sharp spikes. This is more consistent with the presence of tight clusters (see text).

distance between their two clusters  $A, B$ . If there are  $k$  clusters, then there should be roughly  $k(k - 1)$  different values for distances among the  $k$  clusters. Hence, the distances between pairs of vectors would be closer to one of these  $k(k - 1)$  values rather than be distributed in a continuous range.

Figure 5.8 and 5.9 show the histograms of normalized distances between *all* pairs of hidden activation vectors for a regular ESN and for an ESN augmented with  $E_2$  on *CFG #2*. First, the training sequence is run through each ESN and all 8878 hidden activation vectors are recorded. Then, the vectors are normalized as described in Section 5.2.3. Next, all 40 million distances among all pairs are calculated and put into 500 histogram bins.

In Figure 5.8 for a regular ESN, the inter-vector distances vary over a wide

range from 3.1 to 4.5 and exhibit a shape similar to that of a Gaussian distribution. This is expected from a very distributed activation space representation. We can see that there are two small spikes near 0 and 0.8. It was found that 48% of the distances in these two spikes come from distances between the states of the network *after* receiving the special symbol *end* indicating the end of a sentence, and the rest are mostly caused by activation vectors being close to each other by chance. The reason for the states of the network after receiving the special symbol *end* being close together is that there are a large number of input symbols *end* (1,500 for 1,500 training sentences, out of 8878 input symbols), and upon receiving these symbols at the input, the network’s reservoir states (the hidden activation vectors) move to a generally small region, so the distances between them are under 0.5. It is important to recall that the hidden activation vectors in ESNs are not trained by  $E_1$ , so the states of the reservoir become similar only because they receive the same input (*end*), not because the network “knows” it is the end of a sentence. Thus, hypothetically, if there were a lot of symbols  $x$  other than *end* in the training sentences, the distances between the hidden activation vectors after receiving the symbol  $x$  would similarly create a spike. Moreover, because since there is no “force” ( $E_1$ , as with simple recurrent networks) to pull the similar states close together, this region is not very small. Consequently, the distances between these states have a Gaussian-like distribution rather than a sharp spike at 0.

In Figure 5.9 for an ESN augmented with  $E_2$ , the largest histogram bin is close to 0. This matches the expectation that activation vectors are in tight clusters, thus producing lots of pairs with short distances. The rest of the distribution also

contains notably more sharp spikes than in Figure 5.8 as we anticipated above.

### 5.3.2.3 Weight Permutation Analysis

It was observed that training ESNs with  $E_2$  changed the range of the *input*  $\rightarrow$  *hidden* weights (i.e., the minimum and maximum weights). Thus, it is needed to make sure that the improvement in FSM extraction presented in previous sections is not only because of the change in the range of the weights, but also because of the specific arrangement of them created by training with  $E_2$ . For each ESN already trained with  $E_2$ , let a corresponding  $\text{ESN}_{E_2}^*$  be the same trained ESN but with its *input*  $\rightarrow$  *hidden* weights randomly permuted (no further training).  $\text{ESN}_{E_2}^*$  is then compared with regular ESN and ESN trained with  $E_2$  on FSM extraction.

Table 5.3:  $\text{ESN}_{E_2}^*$  versus ESN with  $E_2$  and ESN (Averaged over 100 Runs)

Data set	$E_1/N$			$-2E_2'/N^2$		
	ESN	$\text{ESN}_{E_2}^*$	$\text{ESN}_{E_2}$	ESN	$\text{ESN}_{E_2}^*$	$\text{ESN}_{E_2}$
CFG #1	0.417	0.416	0.410	9.132	7.065	12.383
CFG #2	0.904	0.867	0.835	14.478	5.967	22.468

Table 5.4: Means ( $\sigma$ ) of Number of States and Transitions over 100 Runs

Data set	No. of states			No. of transitions		
	ESN	$\text{ESN}_{E_2}^*$	$\text{ESN}_{E_2}$	ESN	$\text{ESN}_{E_2}^*$	$\text{ESN}_{E_2}$
CFG #1	82.42 (10.43)	78.49 (8.1)	21.40 (3.5)	293.95 (28.4)	284.56 (25.2)	66.00 (12.2)
CFG #2	192.03 (51.2)	121.40 (12.0)	6.74 (4.8)	380.00 (63.3)	283.94 (22.1)	42.64 (16.3)

Tables 5.3 and 5.4 show that  $\text{ESN}_{E_2}^*$  performs very similarly to regular ESN on *CFG #1*. For *CFG #2*,  $\text{ESN}_{E_2}^*$  shows a small improvement over regular ESN,

but is still much worse than ESN trained with  $E_2$ . These results confirm that the improvement is indeed specifically from training with  $E_2$  and not just due to weight magnitude changes.

### 5.3.3 Learning with $E_2$ versus ESN+

ESN+ is a method for calculating each individual  $w_{ji}^{in}$  directly from data. It was developed in [5, 22] to improve the prediction accuracy of ESN on *CFG #2* and was used again in [23] to extract FSMs. In [5], the authors found that simple word co-occurrence statistics contain a lot of useful semantic information. This motivated the work in [22] to assign the weights  $w_{ji}^{in}$  entirely based on word co-occurrence statistics. In particular, consider an ESN with  $N_I$  input units and  $N_H$  hidden units, with  $N_I < N_H$ , and where the input layer uses one-hot encoding so there are also  $N_I$  input symbols. Each connection from the  $i^{th}$  input unit to the  $j^{th}$  hidden unit where  $i \in \{1..N_I\}$  and  $j \in \{1..N_I\}$  is then set to:

$$w_{ji}^{in} = N \times \frac{N(i, j) + N(j, i)}{N(i)N(j)} \quad (5.6)$$

where  $N$  is the total length of the training sentence,  $N(i, j)$  is the number of times the  $i^{th}$  symbol is *followed* by the  $j^{th}$  symbol, and  $N(i)$  is the number of times the  $i^{th}$  symbol appears in the training data. Note that the equation only applies for  $j \in \{1..N_I\}$ . For  $j \in \{(N_I + 1)..N_H\}$ ,  $w_{ji}^{in}$  is set to 0 for  $i \in \{1..N_I\}$ . In other words, an input symbol will stimulate the hidden unit corresponding to the symbols that are more likely to follow or precede it. For instance, in *CFG #2*, the word *boys*, often being the first word of a sentence, usually follows *end* (the special marker for the

end of a sentence), and precedes verbs such as *like*, *chase*, and *see*. So the weights coming from the input unit corresponding to *boys* to the hidden units corresponding to *end*, *like*, *chase* and *see* are set with a relatively higher value than the rest of the weights from *boys*. As the task of the network is to predict the next word, this alone already gives the network a lot of information about how to do the task. If the next word depended on *only* the preceding word, the network could do perfect predictions using just this information. However, the task requires more than just the preceding word, so the network also needs the dynamics from the intra-reservoir connections. This ESN+ method was found to improve predictive performance of ESNs significantly for *CFG #2*, and helped the FSM extraction in [23] to reduce the size of the extracted FSMs significantly.

I implemented ESN+ and ran it 100 times with the same data sets and initial condition as described in Section 5.2.1. The average mean squared error for 100 runs is 0.813, less than the average mean squared error 0.835 of ESNs trained with  $E_2$ . The average normalized  $-2E'_2/N^2$  is 27.1, 22% larger than the 22.2 value for of ESNs trained with  $E_2$ . While ESN+ outperforms ESN trained with  $E_2$  in both  $E_1$  and  $E_2$ , it suffers from an important drawback: the method requires *global* computations. In particular, the quantities  $N(i, j)$  in Equation 5.6 have to be computed separately outside of the network, and this involves looking at data structures in a global manner. In contrast, training with  $E_2$  is done only on the network while each node only stores limited local data ( $\overline{a_{H_j}}$ ). In the spirit of neural computation, it is very important that a method is *local* so that it remains biologically plausible and scalable to larger architectures.

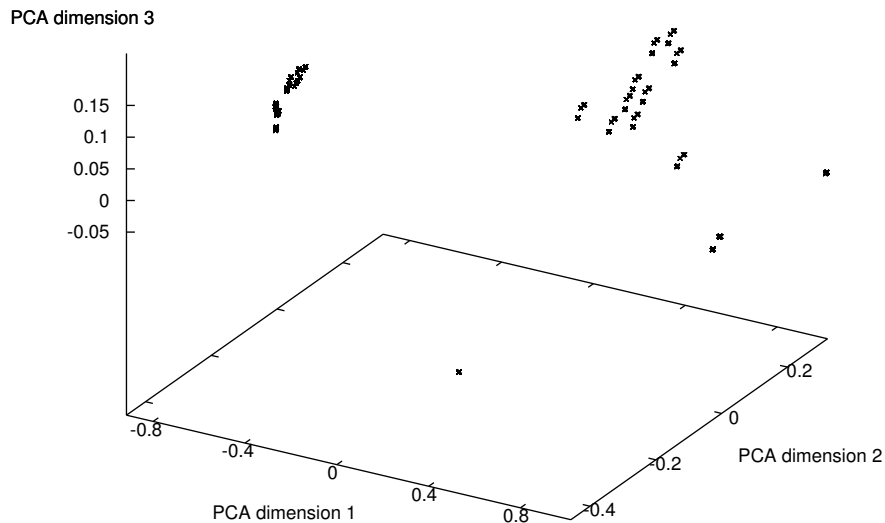


Figure 5.10: A typical projection of the hidden activation space of an ESN+ having 100 hidden units trained on the *CFG #2* data set onto a 3 dimensional space using PCA. The projected activation vectors appear to cluster together more than in Figure 5.6 for a corresponding regular ESN, but there are still a large number of clusters, and they again are not cleanly separated.

Despite having a large  $-E'_2$ , FSM extraction from ESN+ trained on the *CFG #2* data set produces FSMs with 131.8 states on average, significantly larger than the average 6.74 from ESNs trained with  $E_2$  (paired t-test gives  $p < 2.2 \times 10^{-16}$ ). Figure 5.10 shows a typical projection of the hidden activation space of an ESN+ onto a 3 dimensional space. Note that 98% of the variance is retained in this projection. The patterns in this space are more separated, and activations are clustered together more than with a regular ESN (see Figure 5.6). However, there are still a large number of clusters and they are not as clearly separated as in Figure 5.7 for the corresponding ESN trained with  $E_2$  on the same data. This partly explains why FSMs extraction via ESN+ did not work very well.

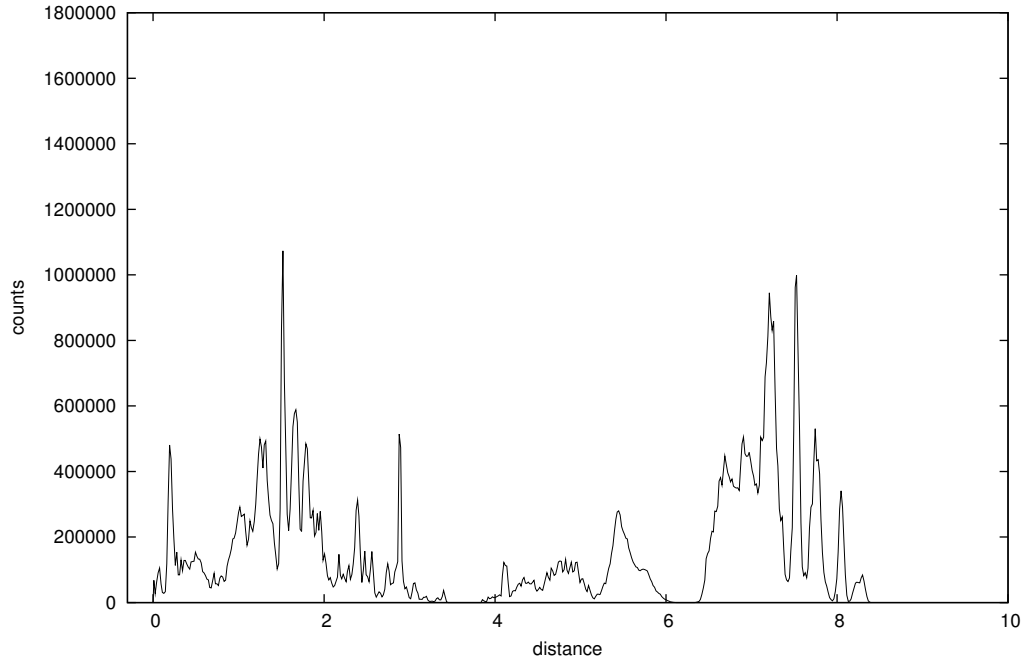


Figure 5.11: A typical histogram of normalized distances between hidden unit activation vectors in an ESN+ trained on the same data set as in Figure 5.8. There are now spikes indicating the presence of clusters, but the spikes are not as sharp as in Figure 5.9.

Figure 5.11 shows the histogram of the same ESN+ with identical data sets used in Figures 5.8 and 5.9 above, i.e., for *CFG #2*. First, this figure does not show the Gaussian distribution pattern as in Figure 5.8 of a regular ESN, indicating that the distances are more diverse. Second, there are many more spikes in the data, but they do not appear to be as sharp as in Figure 5.9 (from an ESN trained with  $E_2$ ). The sharpness of the spikes directly corresponds to the tightness of the clusters. That is, if two clusters are small, the distances between hidden unit activation vectors belonging to the two clusters are very similar, and this would create a sharp spike in the histogram. Furthermore, this would create many short distances, which are absent in the figure. Thus, it appears that the clusters are not as tight as the clusters created by training ESNs with  $E_2$ . Having loose clusters make



clustering very difficult because it is harder to distinguish one cluster from another.

## 5.4 Discussion

In this chapter, a method is presented for improving the extraction of finite state machines from echo state networks by adapting the weights from the input layer to the reservoir using an error term  $E_2$ . This term encourages the formation of a more easily separable internal representation at the reservoir, and is readily incorporated into the training process *while retaining local computations* and maintaining performance accuracy. Extensive experiments with two context-free language data sets showed that this method can reliably extract smaller FSMs from ESNs augmented with  $E_2$  than with regular ESNs. To the best of my knowledge, this is only the second work that has extracted symbolic representations from ESNs, and is the first work that uses a distance-based clustering approach.

The large size of ESN reservoirs poses significant challenges to analyzing the hidden unit activation space and to verifying that the  $E_2$  term indeed helps create a better separated representation. Tests with principal component analysis, histograms of distances, and weight distributions confirm that training ESNs with  $E_2$  results in hidden activation patterns that are grouped into a small number of tight clusters that are more amenable to clustering and hence better FSM extraction.

The surprising result with the increase in accuracy for the data set *CFG #2* shows another advantage of training ESNs with  $E_2$ . A possible explanation for the increase is that the improved encoding at the reservoir creates a richer reservoir

dynamics, and that this allows networks to learn better predictions at the output layer. However, the accuracy level is still below what can be achieved with ESN+. A promising research direction is to study the weights and hidden unit activation patterns created by training ESNs with  $E_2$  and ESN+, or a combination of both, in order to train ESNs that are both superior in accuracy and more cooperative with FSM extraction.

Although the method presented here can extract relatively small, understandable FSMs from ESNs augmented with  $E_2$ , these FSMs are still somewhat larger than those extracted from simple recurrent networks. With many more units in the reservoir, ESNs arguably have more flexibility than simple recurrent networks in forming hidden layer encodings that are both good for low output error and FSM extraction. Moreover, this allows ESNs to work on larger problems because the reservoir can encode more information and have more complex dynamics. Hence, it would potentially be very valuable to improve this method so that one can extract symbolic representations from larger data sets.

As shown in the PCA analysis, the hidden activation space of ESNs augmented with  $E_2$  can be approximated very closely using low dimensional PCA projections. This conceivably can lead to a new approach that extracts symbolic representations using these reduced-dimension spaces instead of the high dimensional hidden activation space. This could open up new research in extracting symbolic representations from ESN trained on regression and control problems in which both the input and output are continuous values [42, 52]. At present, although ESNs are very successful with these problems, almost no past work has been done in extracting symbolic

representations from them partly because of the difficulties involved in working with large reservoirs.

## Chapter 6

### Conclusion

This chapter concludes this dissertation by summarizing the work and highlighting the contributions to symbolic representation extraction from a variety of neural networks architectures. It also discusses the limitations and suggests possible future work that would build on the reported results.

#### 6.1 Summary

While neural networks have achieved success in a wide range of applications, it remains a difficult problem to understand what a network has learned during training because the result of the learning process is generally large opaque matrices of floating point numbers. Therefore, much work has been done to extract symbolic, human-readable representations from learned networks. Past research in this area has largely focused on building progressively more powerful methods to extract and express the symbolic representations. Most of these methods contain a common task: clustering the activity patterns in the hidden unit activation space. The activity patterns are the network's internal representation of the input. During training, the network is free to create any internal representation that is useful, as long as the error at the output is minimized. The activity patterns produced are usually very distributed, thus making the clustering task very difficult. Moreover, the results of

clustering are often complicated, with many clusters and with cluster boundaries at times being difficult to identify.

This dissertation focuses on improving the separability of hidden layer activity patterns so that *any* symbolic extraction method working with the more separated activity patterns can do better. The central hypothesis in this thesis is that it is possible to alter training methods to learn a better internal representation that is amenable to the clustering task with only negligible change to the error at the output. To support this hypothesis, methods were developed to augment the popular error backpropagation algorithm so that it creates better internal representations on feedforward, simple recurrent, and echo state networks.

In this context, the error term  $E_2$  was first introduced in Chapter 3 to help increase the distance between hidden activation vectors with feedforward neural networks. It was hypothesized that the combined effect of the original error function  $E_1$ , which keeps activation vectors that should be together close to one another, and the new error term  $E_2$  that pushes activation vectors away from each other, is that activation vectors will form tight clusters that are further away from each other. As a result, it would be much easier to cluster the hidden activation patterns. An efficient way to calculate the gradient of  $E_2$  with respect to network's weights was derived. This was a crucial step in incorporating the term into gradient descent using standard error backpropagation. Furthermore, it is remarkable that the derived calculation of the gradient only requires values that can be computed and stored locally at each hidden unit. This local property is highly desired in neural network training.

The effectiveness of training with both  $E_1$  and  $E_2$  was illustrated in an example of two neural networks trained with and without  $E_2$  on the data set *waveform*. The final hidden activation spaces were shown and clearly demonstrated that  $E_2$  helped to make the activation vectors cluster into three separate groups, while in contrast the patterns in the hidden activation space of the network trained with regular error backpropagation were very distributed.

A simple algorithm was derived that extracts symbolic rules from feedforward neural networks, taking advantage of the improved hidden layer representation when  $E_2$  is used (Chapter 3). The algorithm was evaluated on five large public data sets having more than 1000 instances from the UCI Machine Learning Repository. These data sets are difficult, and have many attributes and classes. It was found that the algorithm performed well on all five data sets relative to standard backpropagation learning. Careful considerations were taken to ensure the validity in the experiments. First, the settings for the control (regular error backpropagation with  $E_1$  alone) and the experimental ( $E_1$  with  $E_2$ ) runs were identical except for the error function being used. Second, 10-fold cross validation were used throughout all experiments. Furthermore, each experiment was repeated 10 times with different random initial weights. Thus, the reported results are averaged over all 100 runs. Having so many runs ensured that any improvement came from the method and not just by chance. The results clearly demonstrated that  $E_2$  consistently helps to increase the average sum of squared distances between hidden activation vectors, thus creating a sparser activation space. More importantly, it also helped reduce the average number of rules extracted from trained networks significantly. Moreover, the best runs with  $E_2$  also

produced simpler rule sets than the best runs with regular error backpropagation.

Building upon the success of  $E_2$ , I introduced two new *class-aware* terms  $E_3$  and  $E_4$  that *push* the activation patterns of instances from *different* classes apart and *pull* the activation patterns of instances from the *same* classes together, respectively. Local learning rules were again derived to compute the gradient of  $E_3$  and  $E_4$  with respect to the weights. Extensive experiments on the same five data sets were then conducted to compare networks trained with  $E_1$ ,  $E_1 + E_2$ ,  $E_1 + E_3 + E_4$ , and with the popular *C4.5rules* program. The results showed that  $E_3$  and  $E_4$  help to extract even simpler rule sets than with  $E_2$  alone, and that the neural network based rule extraction method outperforms *C4.5rules*.

Throughout all of these experiments, the accuracy rates and  $E_1$  value changed very little when networks are trained with the augmented terms, relative to when they are trained with the unaugmented  $E_1$  alone. This supports the hypothesis that there are many possible encodings at the hidden layer that can provide correct outputs. The additional  $E_2$ ,  $E_3$ , and/or  $E_4$  terms bias these encodings toward more separated ones that facilitate rule extraction. Another advantage of neural network based rule extraction was found during the analysis of the *default class* output. Extracted rules appeared to be non-overlapping, and did not require a default class output as in *C4.5rules*. Such rule sets are easier to apply and also easier for a person to understand.

A limitation of feedforward neural networks is that they are not suitable for processing temporal sequences, an important class of data. Instead, simple recurrent networks with feedback connections and context units can learn and represent such

data more naturally. In addition, the extracted representations must also be able to express the temporal dynamics in data. Hence, finite state machines rather than symbolic rules are usually extracted from simple recurrent networks. In Chapter 4,  $E_2$  was generalized to work on simple recurrent networks. The derivation of the gradient for  $E_2$  with respect to the weights was reexamined and modified accordingly but still retains the local property as with feedforward networks. Then, a simple example with the data set *badigu* was used to verify the effectiveness of training with  $E_2$  versus training with regular error backpropagation. Two hidden layer activation spaces of networks trained with and without  $E_2$  from identical initial conditions showed that training with  $E_2$  indeed pushed the activation vectors apart, and the activation vectors grouped into few very tight and separated clusters. Moreover, the number of clusters was also reduced significantly.

A simple algorithm was introduced to extract finite state machines (FSMs) from simple recurrent networks. This algorithm takes advantage of the better separated hidden representation. The algorithm was used to evaluate FSM extraction from networks trained with and without  $E_2$  on four different data sets generated from both regular grammars and context-free grammars. Again, multiple runs with 10-fold cross validation and identical control/experimental set-up were used to ensure the validity of the experiments. The results clearly demonstrated that training with  $E_2$  consistently reduces the average sizes (number of states and transitions) of the extracted FSMs on all data sets. These averages are also very close to the simplest hand-design FSMs.

Finally,  $E_2$ -modified backpropagation was adapted to work on echo state



networks (ESNs), a variant of recurrent neural networks with a large number of hidden units (the reservoir) and special ways of training its weights. ESN posed unique challenges to adapting the error term  $E_2$ , to incorporating  $E_2$  into the training procedure, and to analyzing the effectiveness of  $E_2$ . The derivation for the gradient of  $E_2$  with respect to the weights was reexamined and modified to work with ESN while retaining solely local computations. Then, a new unsupervised weight modification method was devised to improve the hidden activation space using  $E_2$ .

The large numbers of hidden units in ESNs makes it very difficult to visualize the corresponding high dimensional hidden activation space with two or three dimensional plots. Hence, dimensionality reduction using PCA was used to analyze the hidden activation space of an ESN having 100 hidden units on a large context-free grammar data set. Two reduced-dimension hidden activation spaces, one from regular ESN and the other from ESN adapted with  $E_2$ , showed clear improvement in separability of the hidden activation space patterns with the help of  $E_2$ . In addition, another test for the impact of  $E_2$  was done by investigating the histograms of Euclidean distances between all pairs of hidden activation vectors. The histogram of distances from regular ESN showed a pattern of distances produced from sparsely distributed activation vectors, while the histogram from ESN adapted with  $E_2$  showed a pattern of distances produced from fewer and tighter clusters. In brief, these careful tests support the hypothesis that  $E_2$  improves hidden activation spaces for better clustering.

Finally, a series of experiments on two complex context-free grammars showed that training with  $E_2$  aided the FSM extraction process in producing machines with

many fewer states and numbers of transitions. A surprising result is that  $E_2$  even helped improve the prediction accuracy of ESNs on the data set *CFG #2*. As far as accuracy is concerned, ESN+ [22] is a variant of ESN that calculates directly the input-hidden layer weights in order to improve the network’s prediction accuracy. Experiments were done to compare and contrast the results of ESN trained with  $E_2$  and ESN+. The results showed that ESN+ has better prediction accuracy than ESN trained with  $E_2$ . However, ESN+ suffers an important draw back: it requires global computations while using  $E_2$  still only requires local computations. Turning to finite state machine extraction, the hidden activation space of ESN+ was analyzed and its patterns were found to be only slightly more separated than regular ESN. Thus, ESN+ does not support clustering-based FSM extraction in the same way that using  $E_2$  does.

## 6.2 Contributions

Throughout this dissertation, I have argued that it should be possible to make neural networks learn a better separation of hidden activation patterns that is more amenable to symbolic representation extraction, while still maintaining accuracy of the outputs. Along the way of building the arguments for this, the following contributions have been made:

- I proposed three new error terms  $E_2$ ,  $E_3$ , and  $E_4$  for feedforward neural networks that helped to create better separated hidden activation patterns. While  $E_2$  is class-unaware and pushes all hidden activation vectors away from each other,

$E_3$  and  $E_4$  selectively push the hidden activation vectors encoding different classes away from each other and pull the hidden activation vectors encoding the same class closer together. The combined effect of the new error terms and the regular sum of squared error is that similar hidden activation vectors automatically cluster together, making it easier to extract symbolic rules from trained networks. Using gradient descent, I derived efficient and local learning rules to incorporate these terms into error backpropagation-based network training algorithms.

- I applied the modified error backpropagation methods derived above to five large real-world data sets and found that they consistently helped improve a rule extraction algorithm, reducing the size of the rule set while maintaining classification performance. To rule out the effect of randomness and be more confident in the results, the experiments were repeated multiple times with different initial weights, and 10-fold cross validation was used. Accordingly, results were averaged over 100 runs in which each pair of control and experimental runs started from identical initial conditions. This methodical testing procedure was used throughout the dissertation and distinguishes this work from many past studies of rule/FSM extraction.
- I generalized  $E_2$  and the learning rule to simple recurrent backpropagation networks. Trained with the regular sum of squared error  $E_1$  augmented by  $E_2$ , these recurrent networks also created tight clusters in the hidden unit activation space. This allowed clustering, one of the crucial steps in extracting

FSMs from recurrent networks, to be done much more efficiently and accurately. The effectiveness of  $E_2$  on FSM extraction was demonstrated empirically on four data sets generated from regular and context-free grammars. Again, the accuracy of the networks was kept virtually unchanged despite the addition of the new terms. Also, the experiments were repeated multiple times and the averages were reported instead of the best runs.

- I also adapted  $E_2$  to echo state networks, which are very large recurrent networks with special training procedures. ESN poses unique difficulties in this case because of the high dimensional state spaces. An efficient unsupervised method was derived to increase the distances between hidden activation vectors in ESN. The method was demonstrated to be effective in encouraging hidden activation vectors to form clusters in reservoir activation spaces through two careful experiments in which PCA and histograms of distances were applied. Empirical experiments with two large data sets generated from context-free grammars showed that the better separated activation vectors in the reservoir helped to extract simpler FSMs from the networks.

### 6.3 Limitations and Future Directions

As in many approaches using penalty terms, there is a trade-off: a too big contribution of  $E_2$  relative to  $E_1$  will make the activation patterns very separated and very good for clustering, but if the contribution of  $E_2$  is too small, the improvement in the hidden activation space representation will be minimal. The problem is to

influence the training enough to produce the desired separation without compromising  $E_1$  or accuracy. In this work, the coefficients for  $E_1$  and  $E_2$  had to be chosen empirically. As such, a method that automatically determines the contribution of  $E_2$ , or adapts it during training, could be invaluable in making the methods introduced here more applicable and easy to use.

One limitation of the proposed error terms is that they all depend on the Euclidean distances among the activation vectors. The Euclidean metric is simple, easy to work with, and has been used by most past rule extraction algorithms. But it may not be the best metric in measuring the similarity of representations because it grows quickly when vectors are further apart in very high dimensional spaces such as hidden unit activation spaces. Hence, an important future research direction will be to study other ways to bias the encodings beyond sum of squared Euclidean distances. Potentially, other distance metrics may be able to improve the separability of hidden activation space with even less effect on  $E_1$ .

In each 100-run experiment, there were usually a few runs that resulted in very large rule sets or FSMs with a large number of states. Such runs are responsible for most of the variance in the sizes of the extracted rule sets and the FSMs. A study into what exactly caused these runs to be immune to  $E_2$  would conceivably yield insights into how to improve  $E_2$  further.

Finally, another major future research direction would be to study how to apply this approach to other recent neural network architectures, such as evolved designs of recurrent networks [43], or generalized LSTM [56]. Since  $E_2$  has now been applied successfully to a substantial a variety of neural network architectures,

including feedforward, simple recurrent, and ESN architectures, it is hopeful that the approach will also be applicable and effective in other types of networks.

## Bibliography

- [1] R. Alquézar Mancho, A. Sanfeliu Cortés, M. Saínz, et al. Experimental assessment of connectionist regular inference from positive and negative examples. *VII Simposium Nacional de Reconocimiento de Formas y Analisis de Imagenes*, 1:49–54, 1997.
- [2] R. Andrews, A.B. Tickle, and J. Diederich. A review of techniques for extracting rules from trained artificial neural networks. *Clinical Applications of Artificial Neural Networks*, pages 256–297, 2001. Cambridge University Press.
- [3] A. Asuncion and D.J. Newman. UCI Machine Learning Repository, 2007.
- [4] S. Bader, S. Holldobler, and V. Mayer-Eichberger. Extracting propositional rules from feedforward neural networks - A new decompositional approach. *Proceedings of the 3rd International Workshop on Neural-Symbolic Learning and Reasoning*, 2007.
- [5] J.A. Bullinaria and J.P. Levy. Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior Research Methods*, 39(3):510–526, 2007.
- [6] B. Cartling. On the implicit acquisition of a context-free grammar by a simple recurrent neural network. *Neurocomputing*, 71(7-9):1527–1537, 2008.
- [7] M. Casey. The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, 8(6):1135–1178, 1996.
- [8] J.L. Castro, C.J. Mantas, and J.M. Benitez. Interpretation of artificial neural networks by means of fuzzy rules. *IEEE Transactions on Neural Networks*, 13(1):101–116, 2002.
- [9] S. Cho and J.A. Reggia. Learning competition and cooperation. *Neural Computation*, 5(2):242–259, 1993.
- [10] A. Cleeremans, D. Servan-Schreiber, and J.L. McClelland. Finite state automata and simple recurrent networks. *Neural computation*, 1(3):372–381, 1989.
- [11] A. Darbari. Rule extraction from trained ANN: A survey. Technical report, Institute of Artificial Intelligence, Department of Computer Science, TU Dresden, Germany, 2000.
- [12] S. Das and M. Mozer. Dynamic on-line clustering and state extraction: an approach to symbolic learning. *Neural Networks*, 11(1):53–64, 1998.

- [13] J. Diederich, A. Tickle, and S. Geva. Quo vadis? reliable and practical rule extraction from neural networks. *Advances in Machine Learning I*, pages 479–490, 2010.
- [14] W. Duch, R. Adamczak, and K. Grabczewski. A new methodology of extraction, optimization and application of crisp and fuzzy logical rules. *IEEE Transactions on Neural Networks*, 12(2):277–306, 2001.
- [15] R.O. Duda, P.E. Hart, and D.G. Stock. *Pattern Classification (2nd edition)*. John Wiley and Sons, 2001.
- [16] John W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002.
- [17] J.L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.
- [18] J.L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7(2):195–225, 1991.
- [19] T.A. Etchells and P.J.G. Lisboa. Orthogonal search-based rule extraction (OSRE) for trained neural networks: a practical and efficient approach. *IEEE Transactions on Neural Networks*, 17(2):374–384, 2006.
- [20] J.J. Foster, E. Barkus, and C. Yavorsky. *Understanding and Using Advanced Statistics*. Sage Publications Ltd, 2006.
- [21] S.L. Frank. Learn more by training less: systematicity in sentence processing by recurrent networks. *Connection Science*, 18(3):287–302, 2006.
- [22] S.L. Frank and M. Cernansky. Generalization and Systematicity in Echo State Networks. In *Proceedings of the 30th Annual Conference of the Cognitive Science Society*, 2008.
- [23] S.L. Frank and H. Jacobsson. Sentence-processing in echo state networks: a qualitative analysis by finite state machine extraction. *Connection Science*, 22(2):135–155, 2010.
- [24] C.L. Giles and W.O. Christian. Extraction, insertion and refinement of symbolic rules in dynamically driven recurrent neural networks. *Connection Science*, 5(3-4):307–337, 1993.
- [25] M. Gori, M. Maggini, E. Martinelli, and G. Soda. Inductive inference from noisy examples using the hybrid finite state filter. *IEEE Transactions on Neural Networks*, 9(3):571–575, 1998.
- [26] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1994.
- [27] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison Wesley Publishing Company, 1979.



- [28] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.
- [29] P. Horton and K. Nakai. A probabilistic classification system for predicting the cellular localization sites of proteins. *Intelligent Systems in Molecular Biology*, 4:109–115, 1996.
- [30] S.H. Huang and H. Xing. Extract intelligible and concise fuzzy rules from neural networks. *Fuzzy Sets and Systems*, 132(2):233–244, 2002.
- [31] T.Q. Huynh and J.A. Reggia. Symbolic representation of recurrent neural network dynamics. *IEEE Transactions on Neural Networks (submitted)*.
- [32] T.Q. Huynh and J.A. Reggia. Improving rule extraction from neural networks by modifying hidden layer representations. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1316–1321, 2009.
- [33] T.Q. Huynh and J.A. Reggia. Guiding hidden layer representations for improved rule extraction from neural networks. *IEEE Transactions on Neural Networks*, 22(2):264–275, 2011.
- [34] T.Q. Huynh and R. Setiono. Effective neural network pruning using cross-validation. In *Proceedings of the International Joint Conference on Neural Networks*, 2005.
- [35] J. Huysmans, B. Baesens, and J. Vanthienen. Using rule extraction to improve the comprehensibility of predictive models. Research 0612. *K.U.Leuven KBI*, 2006.
- [36] J. Huysmans, R. Setiono, B. Baesens, and J. Vanthienen. Minerva: sequential covering for rule extraction. *IEEE Transactions on Systems, Man, and Cybernetics. Part B, Cybernetics*, 38(2):299–309, 2008.
- [37] H. Jacobsson. Rule Extraction from Recurrent Neural Networks: A Taxonomy and Review. *Neural Computation*, 17(6):1223–1263, 2005.
- [38] H. Jacobsson. *Rule extraction from recurrent neural networks*. PhD thesis, University of Sheffield, 2006.
- [39] H. Jaeger. The “echo state” approach to analysing and training recurrent neural networks. Technical Report GMD 148, German National Research Center for Information Technology, 2001.
- [40] H. Jaeger. Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach. Technical Report GMD 159, German National Research Center for Information Technology, 2002.
- [41] H. Jaeger. Discovering multiscale dynamical features with hierarchical Echo State Networks. Technical Report Number 10, Jacobs University, 2007.

- [42] H. Jaeger, W. Maass, and J. Principe. Special issue on echo state networks and liquid state machines. *Neural Networks*, 20(3):287–289, 2007.
- [43] J.Y. Jung and J.A. Reggia. Evolutionary design of neural network architectures using a descriptive encoding language. *IEEE Transactions on Evolutionary Computation*, 10(6):676–688, 2006.
- [44] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, and A.Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.
- [45] S.C. Kremer, R.P. Neco, and M.L. Forcada. Constrained second-order recurrent networks for finite-state automata induction. In *Proceedings of the 8th International Conference on Artificial Neural Networks ICANN*, volume 2, pages 529–534. London: Springer, 1998.
- [46] A. Krogh and J.A. Hertz. A simple weight decay can improve generalization. *Advances in Neural Information Processing Systems*, 4:950–957, 1992.
- [47] C.Y. Liou, H.T. Chen, and J.C. Huang. Separation of internal representations of the hidden layer. In *Proceedings of the International Computer Symposium, Workshop on Artificial Intelligence*, pages 6–8, 2000.
- [48] C.Y. Liou and W.C. Cheng. Resolving hidden representations. *Proceedings of the International Conference on Neural Information Processing*, pages 254–263, 2008.
- [49] H. Liu and R. Setiono. Chi2: Feature selection and discretization of numeric attributes. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence*, pages 388–391, 1995.
- [50] H. Lu, R. Setiono, and H. Liu. Neurorule: A connectionist approach to data mining. *Proceedings of the International Conference on Very Large Data Bases*, pages 478–489, 1995.
- [51] O. Ludwig and U. Nunes. Novel maximum-margin training algorithms for supervised neural networks. *IEEE Transactions on Neural Networks*, 21(6):972–984, 2010.
- [52] M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.
- [53] M.C. Mackey and L. Glass. *Science*, (197):287, 1977.
- [54] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943.

- [55] K. Mehrotra, C.K. Mohan, and S. Ranka. *Elements of artificial neural networks*. MIT Press, 1996.
- [56] D. Monner and J.A. Reggia. A generalized LSTM-like training algorithm for second-order recurrent neural networks. *Neural Networks*, 2011.
- [57] R. Nayak. Generating rules with predicates, terms and variables from the pruned neural networks. *Neural Networks*, 22(4):405–414, 2009.
- [58] C.W. Omlin and C.L. Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural networks*, 9(1):41–52, 1996.
- [59] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.
- [60] D. Prokhorov. Echo state networks: appeal and challenges. *Proceedings of the International Joint Conference on Neural Networks*, 3:1463–1466, 2005.
- [61] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
- [62] J.R. Rabuñal, J. Dorado, A. Pazos, J. Pereira, and D. Rivero. A new approach to the extraction of ANN rules and to their generalization capacity through GP. *Neural computation*, 16(7):1483–1523, 2004.
- [63] R. Reed. Pruning algorithms - a survey. *Neural Networks, IEEE Transactions on*, 4(5):740–747, 1993.
- [64] M. Riedmiller and H. Braun. RPROP-A fast adaptive learning algorithm. *Proceedings of the International Symposium on Computer and Information Science*, 1992.
- [65] P. Rodriguez, J. Wiles, and J.L. Elman. A recurrent neural network that learns to count. *Connection Science*, 11(1):5–40, 1999.
- [66] F. Rosenblatt. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Spartan Books, 1962.
- [67] D.E. Rumelhart and J.L. McClelland. *Parallel distributed processing: explorations in the microstructure of cognition, vol. 2: psychological and biological models*. MIT Press Cambridge, 1986.
- [68] A. Sanfeliu and R. Alquezar. Active grammatical inference: a new learning methodology. *Proceedings of the IAPR Int. Workshop on Structural and Syntactic Pattern Recognition*, October 1994.
- [69] I. Schellhammer, J. Diederich, M. Towsey, and C. Brugman. Knowledge extraction and recurrent neural networks: An analysis of an Elman network trained on a natural language learning task. *Proceedings of the Joint Conference on New Methods in Language Processing and Computational Natural Language Learning*, pages 73–78, 1998.

- [70] T.J. Sejnowski and C.R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1(1):145–168, 1987.
- [71] R. Setiono, B. Baesens, and C. Mues. Recursive neural network rule extraction for data with mixed attributes. *IEEE Transactions on Neural Networks*, 19(2):299–307, 2008.
- [72] R. Setiono and W.K. Leow. FERNN: An algorithm for fast extraction of rules from neural networks. *Applied Intelligence*, 12(1):15–25, 2000.
- [73] R. Setiono, W.K. Leow, and J.M. Zurada. Extraction of rules from artificial neural networks for nonlinear regression. *IEEE Transactions on Neural Networks*, 13(3):564–577, 2002.
- [74] R. Setiono and H. Liu. Symbolic representation of neural networks. *Computer*, 29(3):71–77, 1996.
- [75] R. Setiono and H. Liu. NeuroLinear: from neural networks to oblique decision rules. *Neurocomputing*, 17(1):1–24, 1997.
- [76] D.E. Sibley, C.T. Kello, D.C. Plaut, and J.L. Elman. Large-scale modeling of wordform learning and representation. *Cognitive Science: A Multidisciplinary Journal*, 32(4):741–754, 2008.
- [77] H.T. Siegelmann and E.D. Sontag. On the computational power of neural nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 440–449, 1992.
- [78] S. Thrun. Extracting provably correct rules from artificial neural networks. technical report IAI-TR-93-5. *University of Bonn, Institut fur Informatik III, D-53117 Bonn*, 1993.
- [79] S. Thrun. Extracting rules from artificial neural networks with distributed representations. *Advances in Neural Information Processing Systems*, pages 505–512, 1995.
- [80] A.B Tickle, R. Andrews, M. Golea, and J. Diederich. The truth will come to light: directions and challenges in extracting the knowledge embedded within trained artificial neural networks. *IEEE Transactions on Neural Networks*, 9(6):1057–1068, 1998.
- [81] M. Tomita. Dynamic construction of finite-state automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Cognitive Science Conference*, pages 105–108, 1982.
- [82] G. Towell and J. Shavlik. The extraction of refined rules from knowledge based neural networks. *Machine Learning*, 13(1):71–101, 1993.

- [83] H. Tsukimoto. Extracting rules from trained neural networks. *IEEE Transactions on Neural Networks*, 11(2):377–389, 2000.
- [84] F. Van der Velde, Gwendid T. van der Voort van der Kleij, and M. de Kamps. Lack of combinatorial productivity in language processing with simple recurrent networks. *Connection Science*, 16(1):21–46, 2004.
- [85] Z.B. Xu, R. Zhang, and W.F. Jing. When does online BP training converge? *IEEE Transactions on Neural Networks*, 20(10):1529–1539, 2009.
- [86] Z. Zeng, R.M. Goodman, and P. Smyth. Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, 5(6):976–990, 1993.