TECHNICAL REPORT NO. 336

Rap-Master Network
Exploring temporal pattern recognition with recurrent networks

by

Gary McGraw, Robert Montante & David Chalmers

September 1991

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

Rap-Master Network

Exploring temporal pattern recognition with recurrent networks

Gary McGraw[1] Robert Montante[2]
David Chalmers[3]

Center for Research on Concepts and Cognition
Department of Computer Science
Indiana University, Bloomington, Indiana 47405

April 24, 1990

[1]Supported by the Department of Computer Science in 1989 and by the Center for Research on Concepts and Cognition (CRCC) in 1990.
[2]Supported by the Department of Computer Science.
[3]Supported by CRCC.

Abstract

Processing continuous temporal signals has proven to be a very challenging problem for connectionist networks. Our research involves training two different types of recurrent networks to recognize the beat in a musical input signal. The networks are fed one time frame of processed input on each cycle, and are run continuously. Input signals consist of simple melodies created on an electronic keyboard and Fourier-transformed into intensity–versus–frequency signals.

In our initial approach to the problem, we ran experiments using fully recurrent networks consisting of four input units, and either three, five, or seven recurrent units (one of which was an output unit). The system was trained using the real-time recurrent learning algorithm devised by Williams and Zipser [1988]. Results show that although a fully recurrent network can be trained to recognize the beat of many specific melodies with given tempos, generalizing to the task of recognizing a static beat in music is a very difficult problem given such an architecture and learning rule.

We also ran experiments using a seventeen node simple recurrent network (see [Elman 1989]). The network was trained to perform the same task using a standard version of the back-propagation learning algorithm. In general, although the Elman-type architecture converged on an answer more slowly than did the fully recurrent architecture, it had a greater ability to generalize to the task of recognizing a beat in music. We consider some factors that might contribute to the different behavior observed.

1 Introduction

Although connectionism has provided important results in many domains such as pattern recognition, perceptual learning and pattern association, the ability to code and manipulate temporal patterns has not been fully developed. Most of the best known results in connectionism make use of feed-forward architectures (FFA) (e.g. [Sejnowski and Rosenberg 1986], [Rumelhart and McClelland 1987]), but the few attempts to capture temporal pattern with FFA's have been largely unsuccessful (see [Elman 1988]). This research is an attempt to study the behavior of another class of models – those with some type of recurrent architecture – and determine both their limits and abilities to process certain kinds of temporal patterns.

Perceptual and sensory information exists in both space and time. For the most part, connectionist models have tended to stress spatial aspects of problems over temporal aspects – sometimes even going so far as to transform the temporal dimension of a problem into spatial form. In general, connectionism has been criticized for its primitive representation of time. Only recently have researchers begun experimenting with networks that handle sequential inputs and run continuously in time [Elman and Zipser, 1988], [Gallant and King, 1988], [Port and Anderson, 1989], [Smith and Zipser, 1989], [Elman 1989]. We agree with Port and Anderson that networks designed to handle sequential input should be further constrained to receive only one frame of input during each cycle. If a network is to be able to recognize patterns that extend over multiple input frames, time must somehow be implicitly manifest in the internal representations that the network develops.

Our use of recurrent architectures allowed us to investigate one way in which short-term memory (reverberating activation) can participate in the capture of temporal patterns. Since music is comprised of patterns of sound waves in time and space, it provides an ideal domain in which to study temporal events. The recurrence found in our models allows activity to persist over time by reverberating among connected units. Hence a 'short-term' memory of recent input frames can be synthesized with current input in order to discover patterns that occur in time. In a fully connected network, such as the first model we consider, activation in the network is completely dynamic, changing over time with both input and feedback. The activation dynamics of the simple recurrent network, the second model we consider, are somewhat more stable since there are fewer recurrent connections between

nodes, and the connections that do exist are carefully chosen.

In contrast, feed-forward architectures have only 'long-term' memories in which information is coded by connection strengths. Activation in a feed-forward model lasts at most only as many time steps as the model has layers. This means that FFA's lack the short-term memory that is crucial to temporal processing. Since activation can only feed forward, past events can affect processing in two minor ways: 1) before they have fed through the net and 2) during initial training when connection strengths between nodes are being adjusted. Our models make use of both long-term and short-term memories in order to decipher complex patterns. Since recurrent connections allow activation to reverberate in the network, they, in some sense, allow for the creation of internal models of external stimuli. Such models of the environment are a crucial part of any cognitive process. For discussions of the advantages that recurrent architectures offer we refer the reader to [Kaplan, Weaver, and French, 1990], [McGraw, 1989], and [Port 1990].

Our first model makes use of full recurrence (i.e. every node is connected to every other node). It is important to note that temporal pattern processing requires only minimal recurrence. For instance, our second model has recurrent connections only between the hidden layer and some 'context nodes' which subsequently feed back into the model. Other researchers have made use of similar architectures to study temporal pattern processing, see [Jordan 1986], [Elman 1988], [Port and Anderson 1989], and [Elman 1989]. Our research serves to test the limits and abilities of fully recurrent architectures as described by Williams and Zipser [1988] and compare the behavior of such an architecture to that of a simpler recurrent architecture. The problem that we chose to study is a general one that could be examined with any temporal pattern processing model. By applying two different architectures to the same problem we can perhaps shed some light on the execution and utility of both architectures.

2 Learning Algorithms and Architectures

The algorithm used in the fully recurrent model was devised by Williams and Zipser [1988] as a way of generalizing the widely-used back-propagation learning rule to highly interconnected recurrent networks. Some revision of the original learning algorithm was appropriate, as the back-propagation rule

only applies to feed-forward or simple recurrent networks. Back-propagation works by performing a gradient descent through the space of possible weights, in order to minimize the error found on one or more output units relative to some training signal. The beauty of the back-propagation algorithm is that the computation for the gradient descent, which in other circumstances would be a complex calculation of derivatives, can in fact be achieved by following a simple local procedure of propagating errors back through the network. This leads to no theoretical gain, but to a great saving in computational efficiency. For more information on the back-propagation algorithm see [Rumelhart and McClelland 1988].

Unfortunately, this powerful algorithm applies only to feed-forward or simple recurrent networks. Rumelhart, Hinton, and Williams [1986] suggest that less sparsely connected recurrent networks can be approximated by "unfolding" cycles into linear sequences of arbitrary length, but this solution is inelegant and necessarily has limited memory. Only truly recurrent networks can accomplish real temporal processing. Models with simple recurrence, like the network we explored in our second set of experiments, can be trained using the standard back-propagation learning algorithm. Fully recurrent networks cannot.

The algorithm derived by Williams and Zipser shares with back-propagation the property of being a gradient descent through the space of weights, guided by error relative to a training signal. Unfortunately, the recurrent network algorithm shares none of the computational efficiency or elegance of back-propagation.[1] The real-time recurrent learning algorithm is computationally intensive. Computation time on each cycle is of the order of the fourth power of the number of units. This alone, however, should not affect the theoretical ability of the network to learn. So far, little investigation into the learning power of this algorithm has been done. Whether it shares the surprising ability of back-propagation to often converge to an optimal solution remains to be seen. For more information about the real-time recurrent learning algorithm see Appendix A.

For the first set of experiments we trained a fully recurrent network composed of up to six nodes using the real-time recurrent learning algorithm. A fully recurrent network consists of a configuration of processing units ('neu-

[1] A slightly improved variation of the algorithm has been developed since we completed our experiments. It is discussed in [Williams and Zipser 1989].

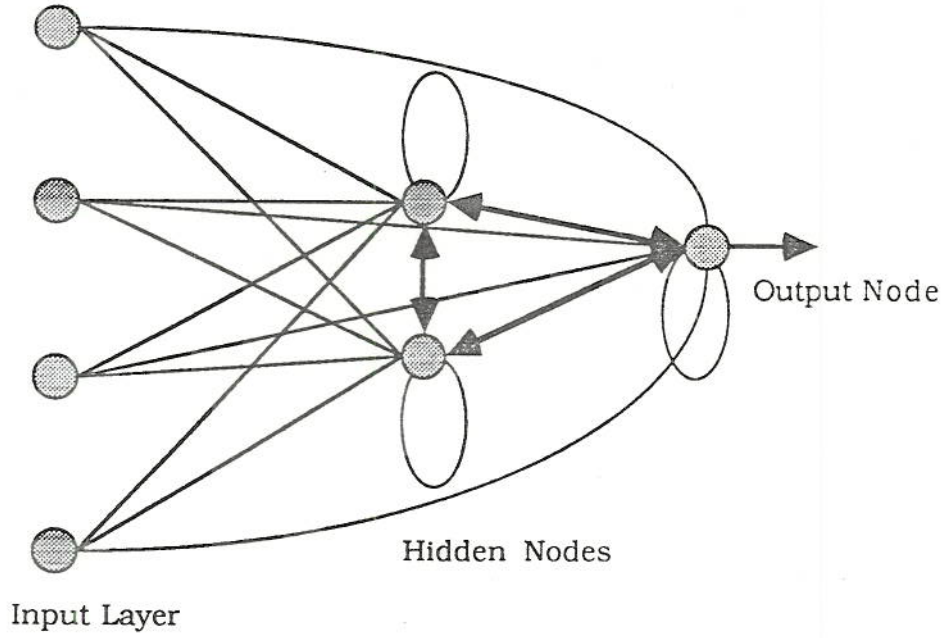


Figure 1: A fully connected network with three recurrent nodes.

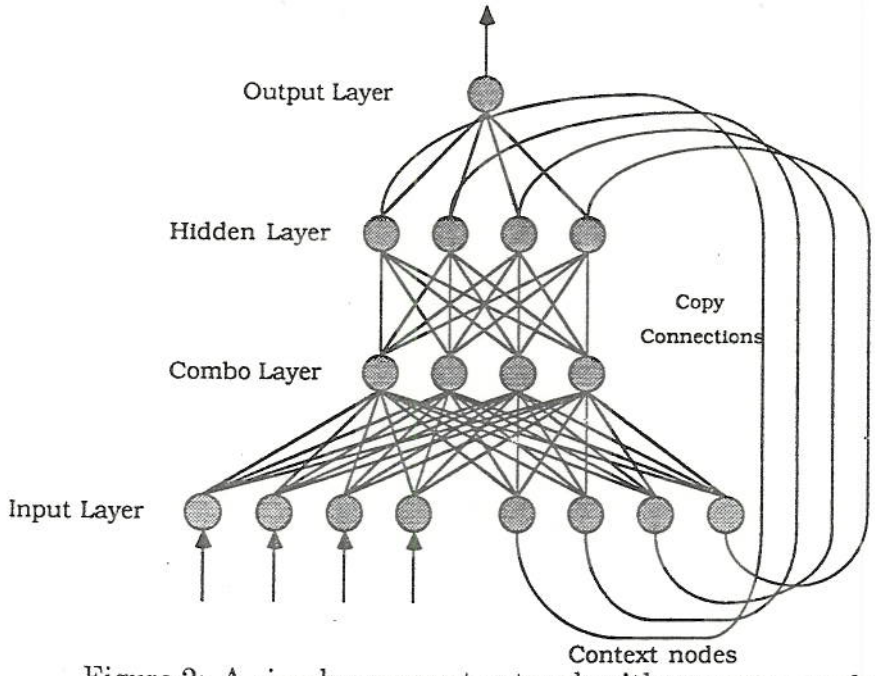


Figure 2: A simple recurrent network with seventeen nodes.

rons' or 'nodes') connected by a system of connections ('synapses'), as in Figure 1. At each time cycle, input values enter the system from some predetermined number of input nodes. Recurrent nodes are connected to every other recurrent node including themselves. Output nodes are a subset of those recurrent nodes whose behavior is scanned by the learning rule or the investigator.

For the second set of experiments we trained a simple recurrent network composed of seventeen nodes using back-propagation. This type of network has several different classes of nodes: input, combination, hidden, context, and output. Figure 2 shows the connectivity of such a network. Note that activation feeds forward through the net unless otherwise indicated. All layers except the context layer are fully connected to the next layer up. Learning occurs on all of the forward connections. The context nodes are connected to the hidden nodes with "copy connections". At each time step, the current activations of the hidden nodes are copied into the context nodes. No learning occurs on the copy connections. Since recurrence in this type of network is limited to the copy connections, and no learning occurs on these connections, we are able to use the back-propagation algorithm for network training.

3 Beat Finding Experiments

The two different architectures and learning algorithms were applied to a problem of beat determination. We recorded several versions of two simple children's melodies on an electronic keyboard. The periodicity of each performance was determined by a metronome, with tempo error limited to that which is acceptable to a musician. (In the following, the word "performance" refers to a particular instance of a melody recorded at a given tempo.) The network was trained to beat along with certain performances that were specified within a given data set. Several experiments were undertaken with varied results.

3.1 Input data

The data generation procedure was rather complicated. Not only were musical data needed, but a synchronized training signal had to be generated

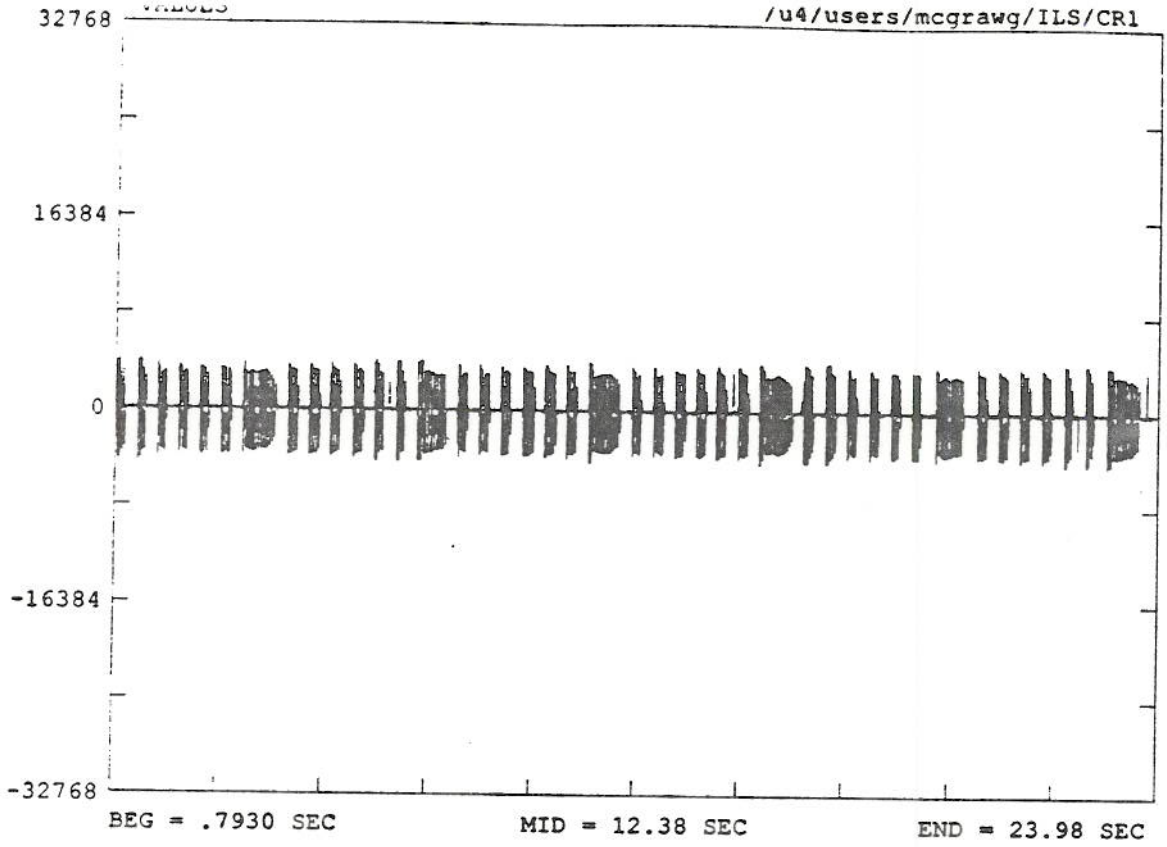


Figure 3: "Twinkle Twinkle Little Star" raw data.

simultaneously. The music was produced using a Casio Model C-140 keyboard. Two familiar children's melodies were recorded using the 'celesta' voice. The tunes were performed several times at different tempos. Tempos were generated by a metronome and ranged from 108 beats per minute to 152 beats per minute. Each performance was recorded on the right channel of a stereo tape. At the same time, the metronome beat was recorded on the left channel. To prevent bleedthrough, the metronome was placed in a soundproof room during recording, and the Casio music was wired directly into the recorder input. The average length of each performance was 25 seconds.

The analog recording of the music was digitized at 16 kHz with a 16–bit analog-to-digital converter, creating a 1-megabyte file for each track of a 25-second recording. A signal-analysis software package called ILS was used to process the recordings into a usable form. Unfortunately, each track had to be digitized independently, so exact synchronization was lost at this point. Synchronization was restored graphically by displaying the data files using the ILS program and selecting a visually distinctive point at which to begin further processing. A typical data file of "Twinkle Twinkle Little Star" is graphically represented in Figure 3.

We determined that a music sampling rate of ten inputs per second was

appropriate for our purposes. The musical signals were divided into 100 millisecond samples. Since each of these samples would look like a 100 millisecond square wave with discontinuous edges, a "Hamming window" was applied.[2] The ILS package's Fast Fourier Transform (FFT) routine was then applied to the raw intensity–versus–time recordings in each sample. The FFT generated intensity–versus–frequency data in the form of spectral intensities from below 50 Hz to 8 kHz (the Nyquist frequency). After processing, we had a set of intensity–versus–frequency inputs at 0.1 second intervals to use as input data for the network.

The FFT routine has a resolution of 1024 values in its frequency range. Rather than supply the network an input vector of 1024 separate input values, we compressed the data into four inputs by dividing the frequency values into four bins and averaging the values in each bin. The four frequency ranges used were: 50–250 Hz, 250–500 Hz, 500–1000 Hz and 1000–4000 Hz. A real number corresponding to the average intensity of the signal within each range was calculated. Scaled versions of these four numbers were used as input to the network.

The training signal was Fourier-transformed in the same manner, but was compressed into one bin using a frequency range of 50–4000 Hz. This produced a signal with low values for much of the time, and a peak value (increase in volume or intensity) on the beat of the metronome. The signal was further processed into a specified wave form of the same period as the metronome beat. Since some of the tempos were not evenly divisible by 0.1 second intervals, the training signal was at times non-uniform. We experimented with several different forms for the training signal. Our first intuition was to represent the training signal with a saw-toothed wave. We also tried modifying the training signal by compressing the range of values with a sigmoidal function. The final version was effectively a binary signal with a value of 1 in each peak interval (or in the second interval of any double peaks). Most of the results discussed below were obtained with the binary training signal, which became the default training signal type.

Input files for the network consist of rows of five values. Each row contains a 0.1 second sample of music which is used as input for one cycle of network operation. The first four values in a row are the inputs from the

[2] A Hamming window is a function which smoothly rolls the edges of a square wave to produce a continuous signal.

four frequency bins (i.e. the average amplitude of the signal within each of the four different frequency ranges), and the fifth value is the desired output (the training signal).

3.2 Implementation of both models

Both network architectures and learning rules were implemented and run on Sun 3 and Sun 4 workstations under the SunOS 4.0 operating system. The fully recurrent model, with the Williams and Zipser learning rule, was implemented in Pascal. This model was verified by training on the XOR function. With three recurrent nodes, the network typically converged within 1000 cycles.

The fully recurrent network contained five input nodes, one for each of the four input frequency ranges and one node to supply a constant bias. Initial experiments were conducted with three, five, and seven recurrent nodes. Seven nodes were used in the final version; this resulted in a twelve-node network, small enough to be operated in reasonable time on a workstation, and large enough to evince learning behavior. One of the recurrent nodes was designated as the output node.

For the experiments with the Williams and Zipser learning algorithm we chose a learning rate of 1.0. All training runs included teacher forcing, and generally lasted for up to 100,000 cycles, recycling through the input file as necessary. Some experiments were terminated early when strong convergence appeared. A few experiments were run beyond 100,000 cycles to verify long-term behavior.

The simple recurrent network, with the backpropagation learning algorithm, was implemented in C on the same Sun workstations. The model was verified both by training and testing on a very simple temporal data set and by removing the copy connections and testing on a simple pattern association task. The simple recurrent network contained seventeen nodes and had the same connectivity as the network in Figure 2. Once again there was only one output node.

For all the experiments with the simple recurrent network, both the learning rate and the bias rate were set to 0.25. The momentum was set to 0.9. All training runs lasted for at least 100,000 cycles, recycling through the input file as in the first set of experiments. A few experiments were run beyond 100,000 cycles to verify long-term behavior. To test the trained network,

learning was turned off and the test data was run through the network using the set of trained connections and beginning with the activation state from the last cycle of the training run.

3.3 Training and testing the first model

Experiments with the fully recurrent network consisted of training runs followed by testing runs. During training runs, the connection weights were adjusted based on the differences between desired output and actual output. For testing runs, the network used weight matrices determined during training runs.

The first experiment used a performance of "Twinkle" with a tempo of 63 beats per minute (bpm) and a saw-toothed representation of the beat for a training signal. The training signal had no beat on the 'off-beats,' which in this case occurred in synchrony with many notes having an identical sound and intensity to those notes on which a beat appeared. The desired output was a saw-toothed pulse once every ten cycles of network operation. The network failed to converge within 100,000 cycles. We hypothesize that the nonconvergence is due to confusion between the beats and the offbeats combined with the very slow tempo.

The next experiment used a performance of "Twinkle" at 120 bpm, with beats on the off-beats. This time, the network converged successfully, to a state with average error around 7%. A graphical representation of the convergence is shown in Figure 4a. (The error values shown in the graph are the mean error values for every 100 cycles.) The convergence shown is that of a network with five recurrent nodes.

Testing the network on its training data yielded good results. However, testing the network on random input produced output almost identical to that produced by the "Twinkle Twinkle Little Star" input. That is, it produced output corresponding to a beat of 120 bpm, although there was no beat to be found. Evidently the network had merely learned to produce the desired output, with no sensitivity to its input. The desired output in this experiment was extremely regular, precisely one beat every five cycles; such behavior is very easy to learn. Indeed, inspection of the connection-weights showed that the weights from the input nodes to the other nodes were all quite low, indicating an effective disregard of the input.

To force the network to pay attention to its inputs, we added a period

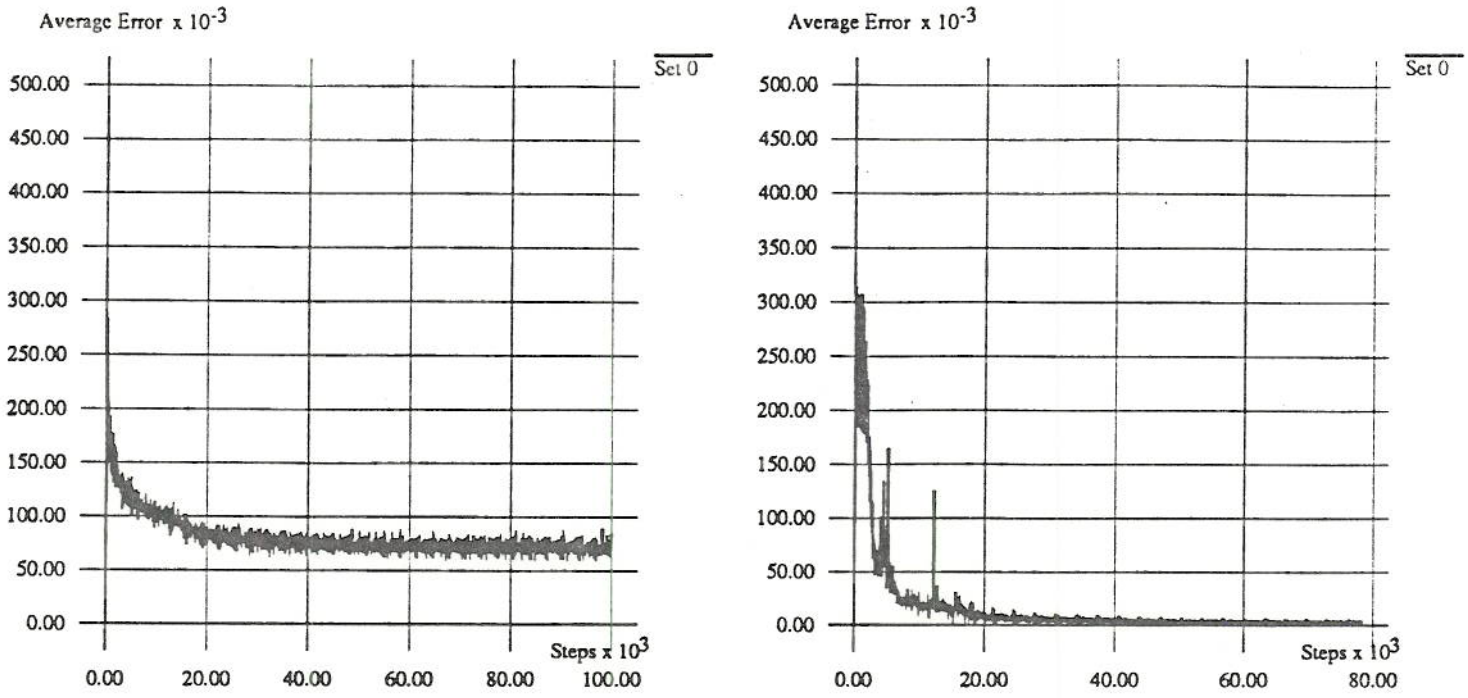


Figure 4:
a) Convergence on "TTLS", 120 bpm, saw-tooth beat.
b) Convergence on "TTLS", 120 bpm, with silence, binary beat.

of 'silence' to the input file. This consisted of a interval of 50 cycles in which all inputs were zero; the training signal during the silent period had a constant value of zero (no beat). We also adopted a simpler desired output for the performance, consisting of a binary training signal with a '1' on the beat and '0's on all other cycles. This form of training signal was used for all subsequent experiments (including those with the simple recurrent architecture).

Variations of the network were trained using the new version of "Twinkle" at 120 bpm as an input file. Models with three, five, and seven recurrent nodes were successfully trained. In each run the network converged quite well. The average error for the seven-node run is shown in Figure 4b. On random input data, the network still produced a 120 bpm output signal. None of the models were responding to the tempo of the performance. Rather, one of the recurrent nodes in each network acted as a 'silence detector.' When it received a zero or near-zero input from all four input nodes, it sent a strong inhibitory signal to the output node, effectively telling it not to beat on during the current cycle. When any of the inputs were sufficiently large, this node had little effect and the network simply emitted a rhythmic output,

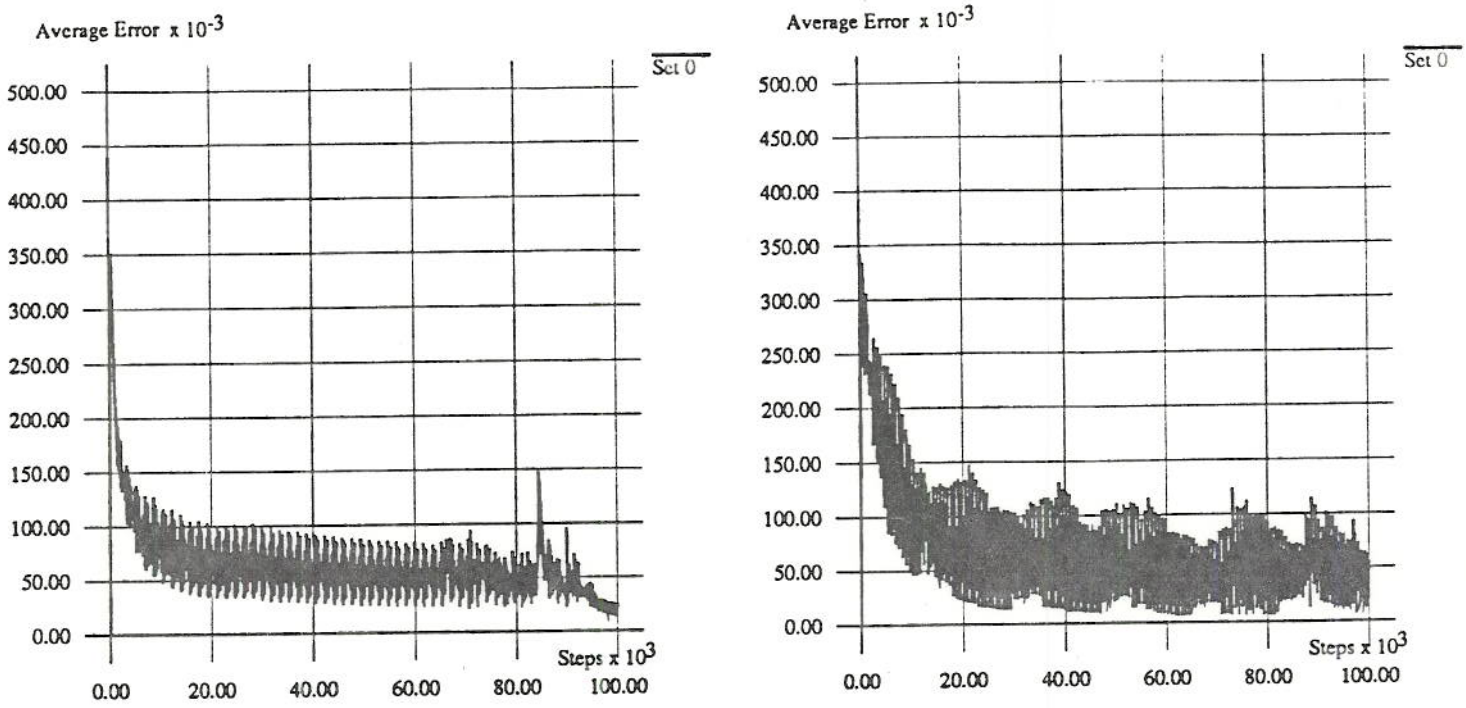


Figure 5:
a) Convergence on "TTLS", 138 bpm, with silence.
b) Convergence on "TTLS", 120/138/152 bpm, with silence.

much as before.

Various delays were introduced between the input and the output signals – the output beat signal, for example, can be trained to fall two cycles after the appropriate input beat. The performance of the network was almost identical with different delays. This again suggested that the network was matching a superficial pattern in the input signal and replying with a simple mimicking of the output signal.

A five-node network was also trained on a performance of "Twinkle" at 138 bpm, mixed with silence. This beat is more irregular relative to the network's sampling rate – sometimes there is a period of five cycles between beats, and sometimes a period of four. The network nevertheless converged adequately, as shown in Figure 5a. An interesting phenomenon can be noted. Convergence remained almost static between cycles 30,000 and 80,000, but around cycle 80,000 there is an unexpected change. There is a brief period of high error, followed rapidly by a convergence to an error rate of less than half the previous value. This suggests that the network had been caught in a local minimum in the space of weights, but had somehow found its way out of this local minimum and into a better configuration.

Multi-tempo Training

Our goal for the network was not merely to identify a tempo, but to generalize to tempos other than those upon which it was trained. To address this goal and the "silence-detector node" problem, we made input files consisting of "Twinkle" at different tempos, and trained the network on these combinations. We combined a 120 bpm performance and a 138 bpm performance, separated by silent periods, into a single input file. This concatenation forced the network to discriminate between the two different sets of inputs. When the network was initialized with weights from the 138 bpm experiment and retrained on the new input file, it managed to converge to within an average 4% error. We repeated the experiment with another input file, consisting of the same two tempos in a different order, and again the network managed to beat correctly on the appropriate performances.

The trained network was then tested by providing it with one of the individual performances as input, without any training signal; it produced the correct output beat. More than mere memorization was going on; the network had at least learned to discriminate between the two performances of "Twinkle". However, when the trained network was tested on a new performance, "Frere Jacques" at 120 bpm, it failed to produce the desired beat. Instead, the output had an irregular period which seemed closer to a tempo of 138 bpm than to 120 bpm. For brief periods it could approximate the correct output, but there was no evidence that this was in response to the input.

Further tests with various inputs showed that the network usually produced one of the two beats (120 bpm or 138 bpm) on all inputs, with little flexibility or continuous gradation between the two. Effectively it was wired to produce only those tempos. The tempo of the output bore little relation to the input except in the case of the two performances upon which it was trained. We concluded that the network was not responding to any detailed temporal pattern. Rather, it used certain superficial details of the input patterns to identify which performance was being played, and then triggered a 'memorized' beat.

A seven-node network was trained on "Twinkle" performances at 120 bpm, 138 bpm and 152 bpm. With three tempos to learn, the network again achieved a reasonable convergence around 4% (see Figure 5b). However, it still could not generalize, producing an output of either 120 bpm, 138 bpm,

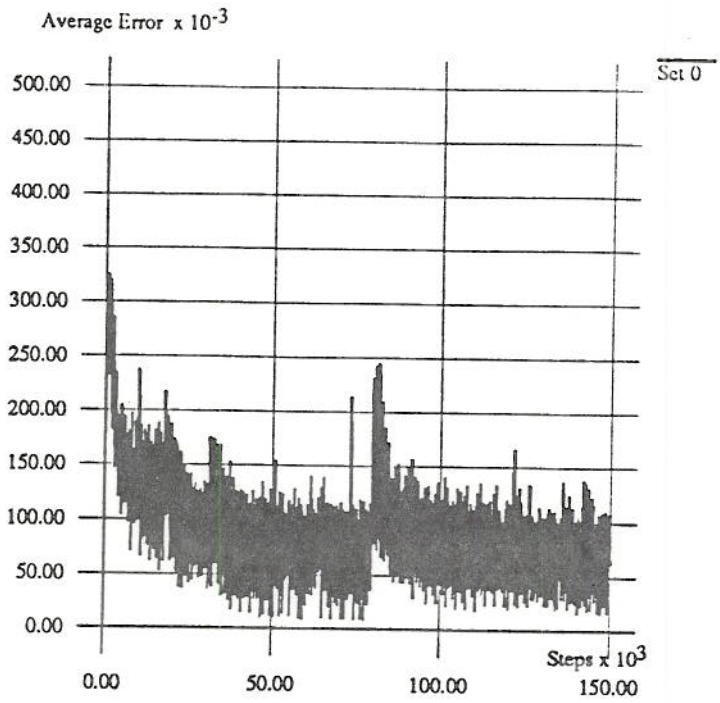


Figure 6: Convergence of 7-node network trained on "TTLS" Range 2.

or 152 bpm for any non-zero inputs.

In an attempt to force the network to generalize, we introduced a data set that encompassed more than two or three tempos. We trained networks on three ranges of tempos, all of which were missing a central tempo that could be used for testing purposes. A seven-node network was subsequently trained on three combinations of "Twinkle" performances, each covering a range of tempos. Range 1 consisted of performances at 116 bpm, 120 bpm, 126 bpm, 138 bpm, 144 bpm, and 152 bpm, with intervening silence periods. Range 2 included only the 120 bpm, 126 bpm, 138 bpm, and 144 bpm tempos (and the silence periods); Range 3 added a 112 bpm tempo to the six in Range 1. In all three cases the network converged successfully in less than 100,000 cycles, although with more distinct tempos present the average output errors were higher — around 8% to 15%. Results for Range 2, the most restricted and the most successful run, are shown in Figure 6.

For each range, the trained network was then tested on a 132 bpm performance of "Twinkle". Note that no 132 bpm performance was present in any of the ranges. Correct results for this in some sense required only limited generalization, to the extent of interpolating an output based on a relatively small deviation from the training inputs. As a control the networks were also tested on the 120 bpm and 144 bpm performances in order to illustrate

effectiveness on individual portions of the training inputs. Test results on the interpolated 132 bpm performance were substantially poorer than the training results.

Upon close analysis it was determined that the network was not generalizing to the beat production task. The network did perform better on the non-experienced tempo of "Twinkle" than it did on random input, but it did not learn the task of beat production based on tempo of some given input data. Instead, it seems that some type of "weighted averaging" of the tempos in the data set was being performed such that it would attempt to beat along with any performance of "Twinkle" at some average tempo. Instead of beating at the tempo of the performance, it would beat at a slightly adjusted average tempo. The network minimized error production in the global picture by producing a set of weights that would generate a "pretty good" beat given a production within the range. It is not surprising that this tendency to produce an almost acceptible beat was carried over to the 132 bpm tempo, which falls just in the middle of each of the ranges. Mean error figures for a network trained on Range 2 were as follows: 120 bpm near 27%, 132 bpm near 37%, 144 bpm near 24%, random input near 50%.

3.4 Training and testing the second model

A simple recurrent architecture was also applied to the beat finding experiment. In the following, average error is a measure of the absolute difference between the network's output and desired output averaged over a given number of time cycles. Because of the complexity of the beat detection task, the average error criterion can sometimes be misleading. We have studied the behavior of each model carefully so as not to be misled by sometimes confusing average error rates. Note that the percentages given as average error of convergence are very rough and are based on low resolution graphical output.

Data for the simple recurrent network included a binary training signal and a beat that fell on off-beats. A series of models were trained on performances of "Twinkle" with different tempos. A period of silence was added to the end of a performance so that the network would pay attention to its inputs. The network successfully converged on performances at 116 bpm, 120 bpm, 132 bpm, 138 bpm, 144 bpm and 152 bpm. It did not converge on either the 112 bpm or the 126 bpm performances. Convergence with the new architecture was not as good as convergence with the fully recurrent net-

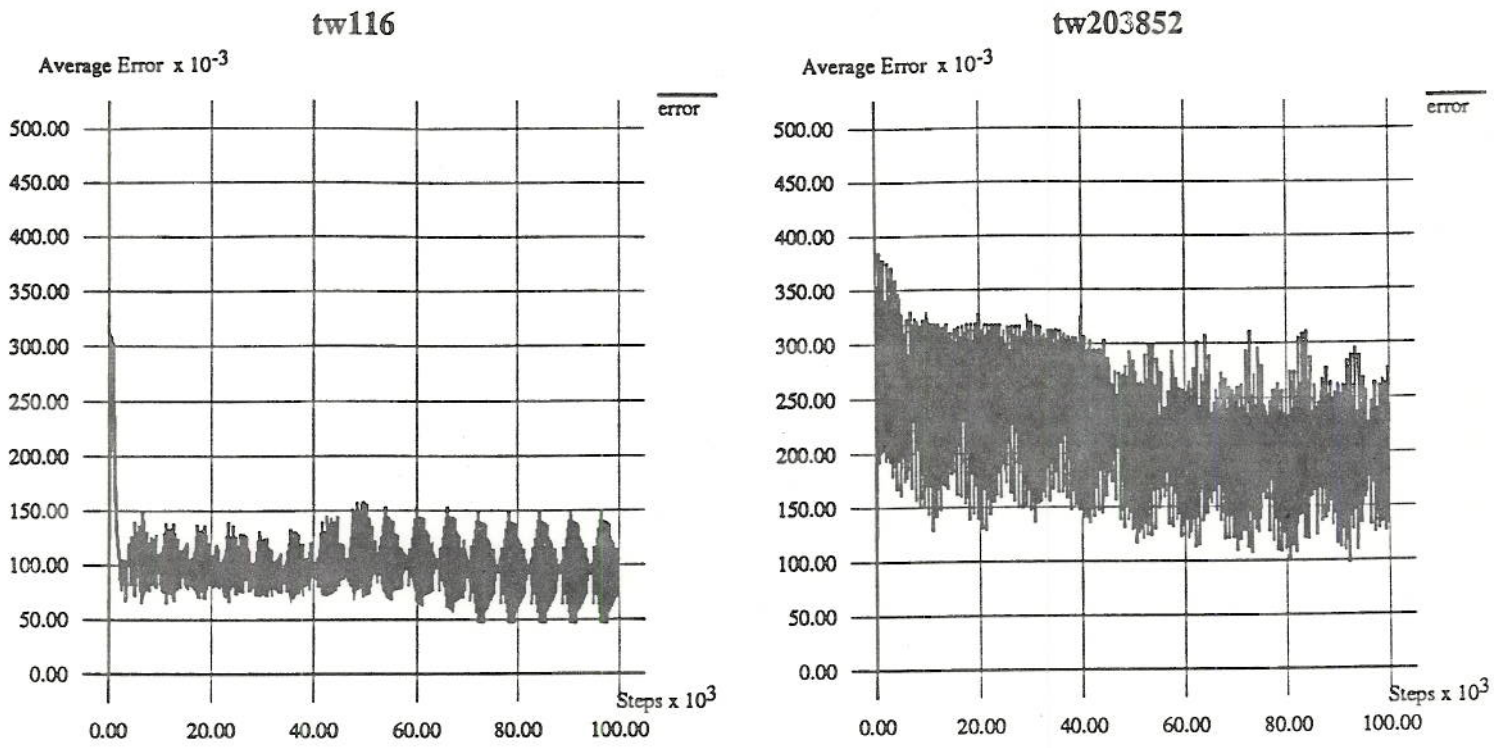


Figure 7:
a) Convergence on "TTLS", 116 bpm, with a simple recurrent network.
b) Convergence on "TTLS", 120/138/152 bpm.

works. Average error for convergence on a given performance test was 14.6%. The best average error, of 9%, was obtained at 116 bpm while the worst, 21%, was obtained at 120 bpm. Analysis of network performance shows that the largest error occurs for the most part during beats. Interestingly most of the errors of this type occur not because a beat is not registering but because the network is not beating strongly enough. Figure 7a shows convergence at 116 bpm.

We are not sure why it is that the network converges more easily on some data sets than on others. Our first intuition was that it was a function of the ratio of beats over time steps. For instance given a tempo of 126 bpm we have a ratio that simplifies to 21/100 whereas a tempo of B120 bpm simplifies to 1/5 (remember, there are 10 time slices per second). This conjecture turned out to be wrong however. In fact, the tempo with the least reducible ratio, 116 bpm, had the best overall average error. It would seem that a tempo of 120 bpm would be the easiest for the network to learn, but clearly it was not.

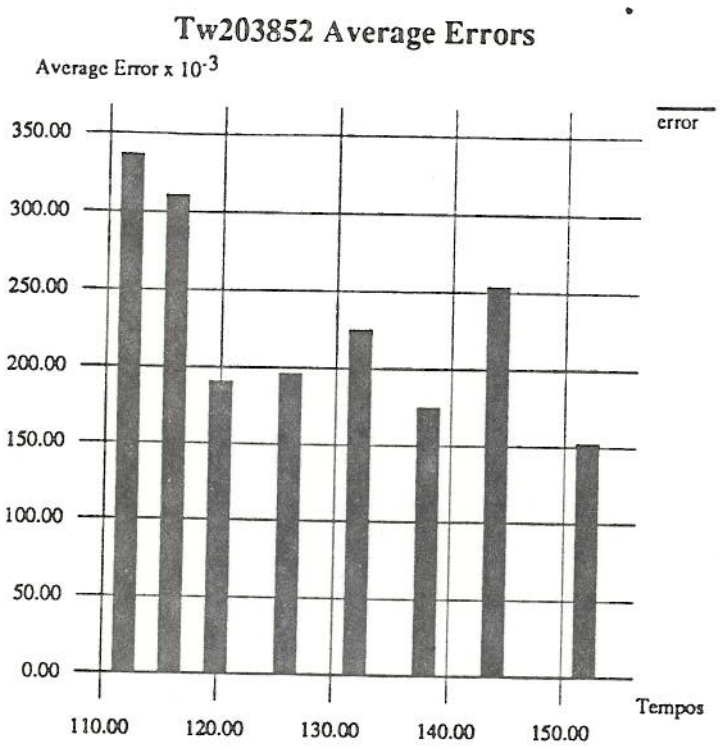
The behavior of the network trained at 116 bpm was tested on a performance at 152 bpm. Surprisingly, the network beat along with the 152 version of "Twinkle" with an average error (over one performance) of only 16%. The

data show that the network beats with the 152 bpm tempo, missing an occassional few beats but beating strongly on the beats that it detects. As a control we tested the network with random data (presented with a 116 bpm beat signal). Unlike the fully recurrent network which happily beat along at its previously trained tempo, the simple recurrent network beat erratically and achieved an average error of 34%. The simple architecture is doing something more subtle than learning to produce only desired output with no sensitivity to its input. Two further tests were done with varying results. The network was tested on a performance of "Twinkle" at 120 bpm. Average error for the run was 30%, and performance was quite bad. The network was also tested on "Frere Jacques" at 120 bpm. The average error for the second test was 22%. The fact that the network behaved differently on different inputs was promising. Such behavior indicates that the network was not learning merely to beat at a constant tempo given any input as did the fully recurrent networks that were trained on one tempo.

Multi-tempo Training

Encouraged by the behavior of the simple recurrent architecture on single tempos, we investigated multiple tempos as well. We began this set of experiments with a data set consisting of three performances of "Twinkle" at 120 bpm, 138 bpm and 152 bpm, seperated by silent periods. Beginning with a set of random weights, a network was trained on this data set for 100,000 cycles. It converged to a rough average error of 20% (see Figure 7b).

The trained network was then tested on single tempo performances from its set of input tempos. The average errors for performances at 120 bpm, 138 bpm, and 152 bpm were 19%, 17%, and 15% respectively. Inspection of the data showed that the network was indeed beating along with the music, and was beating more strongly with the some tempos than with others. For example, the network beats along with a 120 bpm performance with a beat intensity around 0.50 while the same network beats along with a 144 bpm performance at around 0.17. The next step was to see whether the network could beat at tempos other than those in its training set. Testing on all the available performances of "Twinkle" provided an interesting result. The network performed reasonably well at some tempos, with errors ranging from a low of 19.6% at 126 bpm to an acceptable high of 25% at 144 bpm. Results are shown as a bar graph in Figure 8a. The network was also tested on "Frere Jacques" at 120 bpm. During this test, the network realized an average error



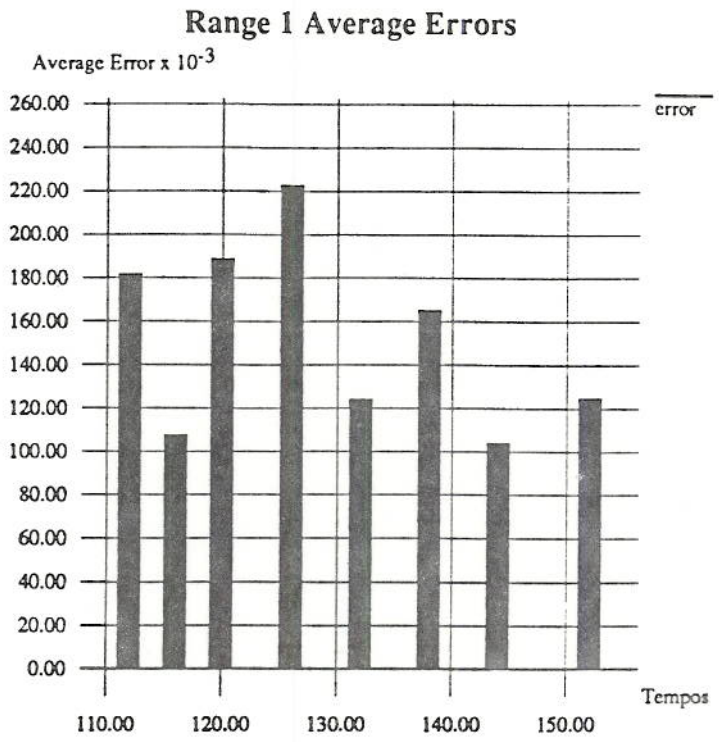
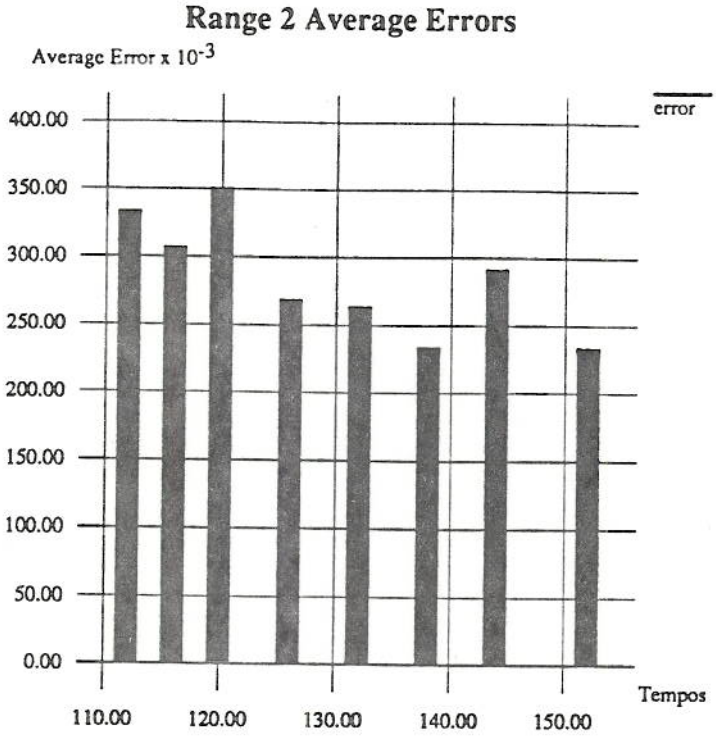


Figure 8:
a) Average error of "TTLS" performances on the 120/138/152 bpm network.
b) Average error of "TTLS" performances on the Range 1 network.

of 30%. Network performance on the new piece was nowhere near as good as it was on "Twinkle" at the same tempo. Apparently the network had developed some sort of beat detector that was dependent on the melody itself. It was able to learn a limited version of the general task of beat detection. Task generalization was limited to multiple tempos within one melody.

Next, the simple recurrent network was trained using the three ranges developed for use with the fully recurrent network as data sets (see page 13). Convergence was once again not as good as it was with the fully recurrent network. Average errors at the end of training were roughly: 20% on Range 1, 23% on Range 2, and 20% on Range 3. Each of the range-trained models was tested on all the available performances of "Twinkle". Results of these tests are shown in Figures 8b and 9. Results of the test runs show that the network trained on Range 1 has developed the best set of weights. Average overall error for all performances on Range 1 was 15%. The network also did very well on the 132 bpm performance which it had not seen before. Testing resulted in an error of 12% on the untrained tempo. This result was notably better than several of the tempos on which the network had been trained.



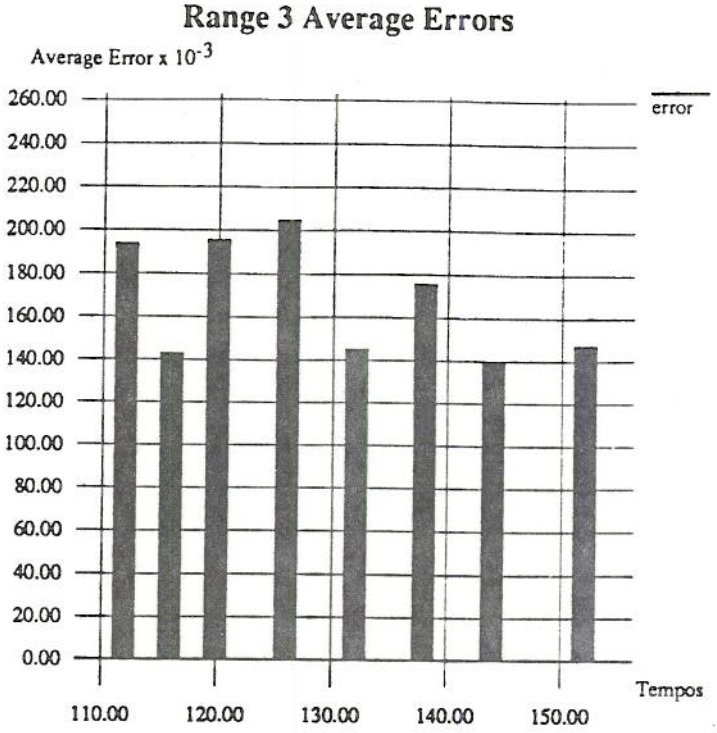


Figure 9:
a) Average error of "TTLS" performances on the Range 2 network.
b) Average error of "TTLS" performances on the Range 3 network.

In fact, all the networks that were trained on ranges missing the 132 bpm tempo performed fairly well when tested at this tempo (see Figures 8b and 9). Average overall error for Range 3 was 17%, while average overall error for Range 2 was 28%. Note that the network trained on Range 2 was tested on all possible performances just as Range 1 and Range 3 networks were even though Range 2 consisted of only 4 tempos. In this set of experiments, the ranges with the most data allowed for the best performance. This behavior is exactly the opposite of the behavior exhibited by the fully recurrent model.

The network trained on Range 1 was also tested on "Frere Jacques" at 120 bpm. Testing resulted in an average error of 21.5%. Note that this performance is slightly better than network performance on "Twinkle" at 126 bpm. Close analysis of the data shows that even with this error rate, performance is not very good. The network was beating fairly strongly very close to the beat but was making some drastic mistakes from time to time.

4 Discussion and Conclusions

4.1 Some explanation of the results.

In most of its learning trials, the fully recurrent network converged to within a very small mean error. This type of network learned only to beat at one or two static tempos or not beat at all. Even when trained on range data sets, the fully recurrent network exhibited the same type of behavior – beating during range tests with an average tempo.

The simple recurrent network exhibited slightly poorer convergence overall than did the fully recurrent network. At the same time, this type of network had a greater ability to generalize. There seems to be some relation between excellent convergence and poor ability to generalize within the beat production task. The appearance of this dichotomy in our experiments may or may not be related to architecture. The myriad differences between the learning rules makes further analysis along these lines very difficult, if not impossible.

Simple data sets

Exact reproduction of the given beat in small data sets is easy for the fully recurrent network. During preliminary training and testing, the fully recurrent network learned to produce the beat in a very obvious and non-subtle manner. It would have been nice if the network had learned to produce a beat by extracting the appropriate temporal pattern from the input data. Unfortunately, complex sensitivity to the internal structure of the input data was never required for acceptable output. Instead, a much easier alternative was available – simple reproduction of the appropriate beat, learned directly from the training signal, with no attention given to the other data.

In the experiments with non-complex data sets, the regularity of the training signal was exploited by the fully recurrent network. After all, a training signal of one beat every five cycles is very easy for such a network to learn to simulate, independent of any properties of the input data. The fact that our input data also happened to have a similar temporal pattern was probably ignored by the network. Exploitation of such a pattern was unnecessary, when it had such a simple alternative.

Neither the addition of silence within a data set nor the inclusion of up to seven distinct performances forced the fully recurrent network to pay more

than minimal attention to its input. The network always followed the path of least resistance – in one case inventing a "silence detector" and in the other paying only enough attention to the input pattern to detect which of the performances was being played so that the appropriate cycle of wired-in beat-production could be chosen.

Such an alternative is apparently not quite as easy for the simple recurrent network to develop. This type of network, having only a few recurrent connections, has a harder time oscillating at a given frequency. Input data may play a larger role in the overall processing of such a network.

The simple recurrent network exhibited somewhat different behavior given an uncomplicated data set. Although complete generalization of task was by no means accomplished, a simple recurrent network after having been trained on an elementary data set could beat along with a different production of the same melody with surprising accuracy. Apparently the simpler architecture allows the network to pay more attention to its input stream. We hypothesize that activation state plays second fiddle to connection strength in the simple recurrent network.

It seems that learning in a simple recurrent network is closely coupled with changes in connection weights, whereas learning in a fully recurrent network involves a more dynamic interaction of weights and node activation. This is probably due to the combination layer of nodes in the simple recurrent network. The use of such a layer makes the architecture of the simple recurrent network inherently less dynamic than the fully recurrent architecture because of the addition of exclusively feed-forward connections.

More complex data sets

As many other researchers in the field of machine learning have discovered, a model will usually solve a problem in very easy and unforseen ways when not given enough constraints. The fully recurrent model again provides evidence of the "do what I mean, not what I say" syndrome. If we endeavor to study higher cognitive activities, then we must realize the tendency of connectionist models to use the most simple and minimal cognitive effort possible in solving the tasks we give them.

In this light, let us examine the behavior of the fully recurrent networks on the range tests. Convergence was much better on a given performance during the training phase than it was during the testing phase. Before each testing phase we erased the short term memory by zeroing activation in the recurrent

nodes. This significantly disabled the network's ability to process even those performances upon which it was trained. It is interesting to note that during testing phases on fully recurrent networks trained with non-complex data sets, no such degradation in performance was observed.

It seems that two different types of behavior have developed in the fully recurrent models. In the earlier networks (with non-complex data sets) the model seems to have been able to solve each given problem by ignoring reverberating activation in favor of developing a wired-in tempo generation model in the weight space. This hypothesis is supported by the observation of "silence detectors", as well as the earlier networks' ability to beat only with periods upon which they had been trained. Being unable to develop a purely hard-wired tempo generation model for the more complex data sets, the later models seem to rely much more heavily on activation state. Such reliance on activation state could explain why results of testing were not as good as with previous models. Apparently, the weights tend to generalize to the task of producing an average periodicity which allows activation to play a more important role in beat production. In other words, the behavior of a wired-in *average* tempo generation model is effectively controlled by the current activation state. But, we must also keep in mind that convergence was not as good with the later models. We suppose that the interleaving of "short term" and "long term" memories causes performance to degrade while at the same time allowing a richer domain for processing. Also note that test runs of random data through the range networks lend support to the hypothesis of an "averaging" beat-maker, since performance on random data was far different than it was during early tests and at the same time very close to the other test performances.

Performance of the simple recurrent networks serves to support this hypothesis. Because of their architecture, these networks probably rely more heavily on change in connection strength than they do on activation state. A poorer convergence may lead to a more general model in weight space – one that can perform on variations of the training set as well as or sometimes better than it does on performances from the training set itself. The fact that the simple recurrent models generalized to the task of beat production on a given melody supports the hypothesis that in the interplay between activation and connection strength, connection strength is more important (given a task like beat production).

Activation in the simple recurrent models is far less dynamic than it is

in the fully recurrent models. This is a direct effect of differences in architecture. When nodes are massively interconnected, a highly dynamic system of activation propagation exists. Input activation is a drop in the dynamic activation bucket. Waves of activation state can easily drown out the input signal. A simpler network architecture causes input activation to play a more important role in internal processing. Our version of the simple recurrent network, which makes use of combination nodes, lends no more weight to the activation state than it does to the input data. It is the combination nodes that make the difference.

4.2 Reasons why generalization is hard.

Although the simple recurrent network exhibited some signs of generalization given productions of the same melody, none of the networks ever really generalized to the task of beat production. We feel that there are significant factors that make generalization hard.

(1) *The music-beat relationship is too fuzzy.*

Perhaps if there were a clear-cut, easily isolated relationship between the musical input data and the required output beat, the network might learn to produce the output by exploiting this relationship. But this is usually not the case. The presence of beat in music is a subtle feature which even some humans have difficulty detecting. In our experiments we attempted to make the presence of the beat an easily-detectable pattern, but there are still some subtleties. While a beat is usually associated with an increase in intensity of the sound signal, this is not always the case. Sometimes the beat falls at a moment when the music is relatively quiet.

A further problem lies with the discrete nature of the model. In our networks, one cycle represents exactly 0.1 second. Frequently, a beat does not fall precisely within one of these time slices, but spans two. When the beat happened to fall between two cycles, we used the convention mentioned earlier to determine upon which cycle the beat should be represented. The somewhat arbitrary nature of this convention makes the association more difficult for the network to learn.

(2) *Extended units of input data make training difficult.*

The temporal aspects of the beat problem when handled serially force the network to process several cycles before a beat can be detected. The same

is true of humans as well, who must hear at least some minimum amount of music before a beat can be determined. At least 20 cycles seem to be required to make the concept of beat meaningful to the network. To present one input cycle from "Twinkle Twinkle Little Star" followed by one input cycle from "Frere Jacques" is meaningless, as it ignores the all-important temporal structure. We used at least 200 cycles to represent a given piece, in order to allow the network enough time to pick up the beat adequately.

The temporal extension of natural input units makes it much more difficult to adequately cover the input space. To cover 1000 different pieces would take on the order of 200,000 cycles – each piece having been performed only once. We can see that in general, the time required to cover the same number of points from the input space as with a simple non-temporal feed-forward pattern association model is orders of magnitude longer. Long, complex data units make training very difficult.

Further, this problem does not seem to be restricted to the current project. By definition, temporal patterns will always be extended in time, thus necessitating extended units of input. This leads to three problems. The first has already been alluded to: training time will necessarily be longer than in the feed-forward case. The second problem is that the extended nature of the input units means that the association between input pattern and output is far less direct than in the feed-forward case. The immediate association of input on one cycle with output on the same cycle is much of what makes the back-propagation algorithm so powerful when it is used with feed-forward models; a direct association is much easier to discover. The more indirect association required for temporal patterns may mean that it will be difficult for gradient descent to find an optimal solution. The third problem with extended input units is that a simple tracking of recent data becomes much more likely. The greater the number of cycles that have elapsed, the more likely the network is to 'forget' its training on previous inputs. If a particular unit of input, once processed, is not returned to for another 100,000 cycles, then there is a large chance that any lesson which was once learned will have been forgotten. Without the ability to jump quickly over the input space to reinforce old lessons (an ability which feed-forward networks have), training becomes significantly more difficult.

These problems are not specific to one particular algorithm or architecture. Such considerations suggest that whatever the representational power of recurrent networks, their training is likely to be a difficult problem.

Conclusions

The problem of training a network to recognize beats in musical input is not settled by this work. A more sophisticated approach may be needed to exploit the full power of recurrent networks to represent temporal structure. Although the problem was not solved to our satisfaction, we hope that we have contributed to the knowledge of the behavior of both fully recurrent networks trained with the real-time recurrent learning algorithm and simple recurrent networks trained with back propagation. We used two different architectures to explore this problem. One was a maximally general form; the other was more structured. The structured architecture was more effective when applied to this particular temporal pattern processing task.

References

Elman, L. (1989). "Structured Representation and Connectionist Models". In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society.*

Elman, J. (1988). "Finding Structure in Time". Technical Report 8801, Center for Research in Language, University of California at San Diego.

Elman, J. and Zipser, D. (1988). "Learning the Hidden Structure of Speech". Journal of the Acoustical Society of America, 83:1615-26.

Gallant, S. I. and King, D. J. (1988). "Experiments with Sequential Associative Memories". In *Proceedings of the Tenth Annual Conference of the Cognitive Science Society.*

Jordan, M. (1986) "Serial order". Technical Report 8604, Institute for Cognitive Science, University of California at San Diego.

Kaplan, S., Weaver, M., and French, R. M. (1990). "Active Symbols and Internal Models: Towards a Cognitive Connectionism". AI and Society, January 1990.

McGraw, G. (1989). "Temporal Pattern and Connectionism". Unpublished manuscript, Department of Computer Science and Center for Research on Concepts and Cognition, Indiana University.

Port, R. (1990). "Representation and Recognition of Temporal Patterns." To appear in Connection Science.

Port, R. and Anderson, S. (1989). "Recognition of Melody Fragments in Continuously Performed Music". In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society.*

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). "Learning Internal Representation by Error Propagation". In McClelland and Rumelhart (eds.), *Parallel Distributed Processing, Volume 1*, 1986 MIT Press.

Rumelhart, D. and McClelland, J. (1987). "On learning the past tenses of English verbs". In McClelland and Rumelhart (eds.), *Parallel Distributed Processing, Volume 2*, 1987 MIT Press.

Rumelhart, D. and McClelland, J. (1988). "Training Hidden Units : The Generalized Delta Rule." Chapter 5 of *Explorations in Parallel Distributed Processing*, 1988 MIT Press.

Sejnowski, T. and Rosenberg, C. (1986). "NETtalk: A parallel network that learns to read aloud." Technical Report 86/01, Baltimore, Johns Hopkins University, Department of Electrical Engineering and Computer Science.

Smith, A. and Zipser, D. (1989). "Encoding Sequential Structure: Experience with the real-time recurrent learning algorithm". In *Proceedings of the International Joint Conference on Neural Networks (IJCNN), Volume 1, Summer 1989.* IEEE TAB Neural Network Committee.

Williams, R. and Zipser, D. (1988). "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks". Technical Report 8805, Institute for Cognitive Studies, University of California at San Diego.

Williams, R. and Zipser, D. (1989). "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks." Neural Computation, 1989, volume 1, number 2.

A Real-Time Recurrent Learning

A fully recurrent network consists of a configuration of processing units ('neurons') connected by a system of connections ('synapses'), as in Figure 1. At each time cycle t, m input values enter the system from m external input lines. We can regard the input signals as output from m *input units* $X_1,\ldots,X_m$. Simultaneously, there are n *recurrent units* internal to the system, $X_{m+1},\ldots,X_{m+n}$, which are firing. All units (or *nodes*) fire with values in a continuous range $(-\infty,\infty)$, although in practice our input units always fired with positive values between 0 and 5. The output from unit k at time t is denoted by $z_k(t)$ (where input units have $1 \leq k \leq m$, and recurrent units have $m+1 \leq k \leq n$).

In a recurrent network, a recurrent unit receives input from all units: the m input units, and all n recurrent units, including itself. Signals are fired along $m(m+n)$ connection lines, each of which has a weight assigned to it. The weight of the connection from unit X_j to unit X_i is denoted by w_{ij} (where $1 \leq j \leq m+n$, $m+1 \leq i \leq m+n$).

At every time cycle, the output value from each unit is propagated down connection lines to every recurrent unit. Each of these units receives an input $s_k(t)$ (for $m+1 \leq k \leq m+n$). This input is calculated by

$$s_k(t) = \sum_{l=1}^{m+n} w_{kl} z_l(t) , \qquad k = m+1,\ldots,m+n .$$

The input value at time t determines the output at time $t+1$. Instead of using a step function as a threshold, we use a continuous sigmoid function, which produces output in a continuous range from 0 to 1. A varying threshold (of sorts) for each unit is built in by designating one of the input units as a *bias unit*, with output value always 1. The weight of the connection from the bias unit to a given recurrent unit then determines the threshold. The output at the next time step is determined by

$$z_k(t+1) = f(s_k(t)) , \qquad k = m+1,\ldots,m+n$$

where f is the sigmoid function

$$f(x) = 1/(1+e^{-x}) .$$

This procedure is run continually, with new input entering the system at every cycle. This process, unlike back-propagation, has the property that there is only one time dimension representing both environmental time and processing time. In back-propagation, no new external input enters the system until a complete round of processing is finished. Such processing may require propagation through several layers of the network. In the recurrent system, however, each cycle of new input corresponds to one cycle of processing time, allowing a natural 'real-time' processing of data.

One or more of the units is designated an output unit. In our model, we had just one output node, so here we will choose this node to be X_{m+n}. Output from this node represents output from the system. In our case this output represents the beating of the system, in the range $[0, 1]$. At every cycle, the output is compared to a *training signal* $d(t)$, which represents the expected value of the output for an optimally- performing system. The difference between the output and the training signal represents the *error* $e(t)$, where

$$e(t) = d(t) - z_{m+n}(t) .$$

The *network error* J at time t is

$$J(t) = e(t)^2/2 .$$

The error $J(t)$ is the quantity we want to minimize. To minimize the error we perform gradient descent on the space of weights. Weights are changed to a new, slightly altered set of values, which would have led to a smaller error on the last cycle. To carry out the gradient descent, we must calculate the derivatives of $J(t)$ and use the equation

$$\Delta w_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial w_{ij}}$$

to change the values of the weights, thus enabling the network to learn. (Here α is some fixed positive learning rate. We usually chose $\alpha = 1$.)

The calculation of the derivatives is straightforward, but it is a tedious derivation. The final equations to compute the required updating of the network are shown below. We use a set of auxiliary variables p_{ij}^k in the procedure, to represent $\partial z_k(t)/\partial w_{ij}$ (where $i, k = m+1, \ldots, m+n; j =$

$1, \ldots, m+n$).

$$p_{ij}^{k}(t+1) = f'(s_k(t))[\delta_{ik}\ z_j(t) + \sum_{l=m+1}^{m+n} w_{kl}\ p_{ij}^{l}(t)]$$

with initial conditions

$$p_{ij}^{k}(t_0) = 0\ .$$

Here δ_{ik} is the Kronecker delta

$$\delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

and f' is the derivative of the sigmoid function.

Using these results, we can compute the required change in each weight by the assignment

$$\Delta w_{ij}(t) = \alpha\ e(t)\ p_{ij}^{m+n}(t)\ .$$

At every time cycle, after the computation of output from every node, this updating procedure is run and weights are adjusted accordingly.

During training, the network is *teacher-forced.* Recall that the output unit is a recurrent unit in its own right, and it feeds its value back into all other recurrent units. To teacher- force the network means that instead of feeding the network- calculated (and possibly incorrect) value back into the network, we feed the training signal in its place. This necessitates some minor changes to the learning procedure, but we will not go into them here. Williams and Zipser found that teacher-forcing was essential if the network was to learn to oscillate.