

NOTES ON ELMAN NETWORKS.

(This is just a collection of bits and pieces, most of which I wrote over a period in pursuing a more or less continuous train of thought to entertain and stimulate the neural network group. The entertainment seemed to work; I've no particular evidence that anyone but me was significantly stimulated. The originals are significantly changed in the light of further thought, but I haven't tried very hard to unify the treatment throughout. The dates are approximate at best, but at least define the order of presentation. The final bit-and-piece has not previously seen the light of day in printed form, though I've talked about it.)

BIT-AND-PIECE 1 :**12-iv-1993****INFORMATION FLOW THROUGH NEURAL NETWORKS.**

Neural networks can generally be thought of as clumps of (one or more) neurons joined together by links.

Hopfield : clumps are single neurons, links are all possible one-to-other links.

Multilayer perceptron : clumps are layers, links are all forward links between layers.

Movement of information through the network depends on the links : we can talk about the channel width of the link (typically in bits), and about how data are represented in the channel.

What's the channel width ? For an n -neuron-to- m -neuron link, we can work out a value for each end. Taking a simple case, at the n end, there are n values each of which can take - say - s distinguishable states. So the number of signals is s^n , which gives us a channel input width of $n \log_2 s$ bits. Similarly, the channel output width at the m end is something like $m \log_2 t$ bits. The overall channel width is at most the smaller of these two values. The number and arrangement of the links between individual neurons will only affect the channel width if some information is lost somewhere - which is quite likely to be the case because of the network's dependence on threshold logic.

For binary neurons (the common case, whether or not we built it that way, as in practice sigmoid neurons "prefer" to work at the extremes of their sigmoid characteristics), s is 2, so we get an n -bit channel, as we'd expect.

Or do we ?

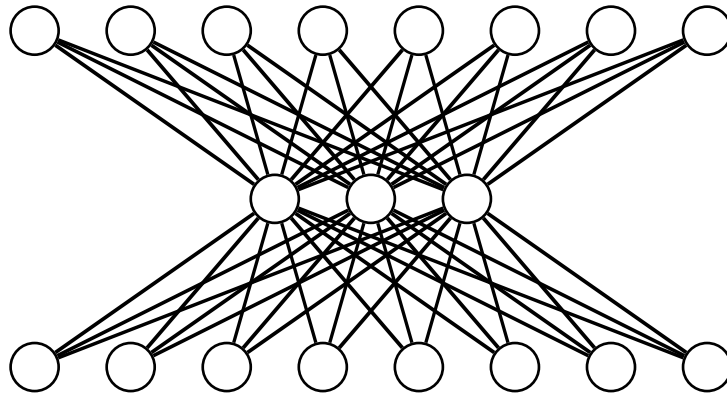
NOT JUST ANY n BITS.

The overall flow of information depends not only on the channel widths of the communications paths; it can be restricted by the neurons. A channel coming from only one binary neuron can transmit only one bit - but it may not be possible to get the bit you want. The classic example is the exclusive-or network. The neuron is presented with two inputs, and is intended to give just one bit of output which indicates whether the input combination belongs to a certain set or not. You can do it for many sets - but not for the exclusive-or function. (Or, equivalently, the equality function. Why does everyone always use exclusive-or ?)

Generally : you have to be able to compute the required output function by threshold logic - or, of course, use a different sort of neuron. Radial basis function neurons will compute exclusive-or.

IDENTITY NETWORKS.

Identity networks are three-layer feedforward networks which are trained to map certain input patterns onto identical output patterns. Consider, for example, the 8-3-8 network :

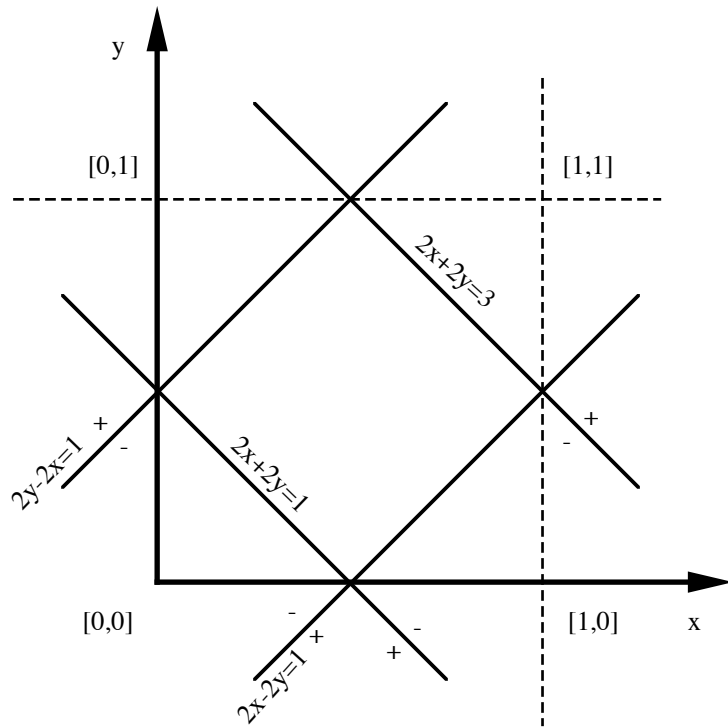


You train it to reproduce the eight one-out-of-eight input patterns (just one input on) as identical output patterns. Three bits should in principle be sufficient to represent the eight distinct values. It works, after a fashion. This example is taken from a published study¹.

Input (and output) patterns	Hidden unit patterns		
10000000	0.5	0	0
01000000	0	1	0
00100000	1	1	0
00010000	1	1	1
00001000	0	1	1
00000100	0.5	0	1
00000010	1	0	0.5
00000001	0	0	0.5

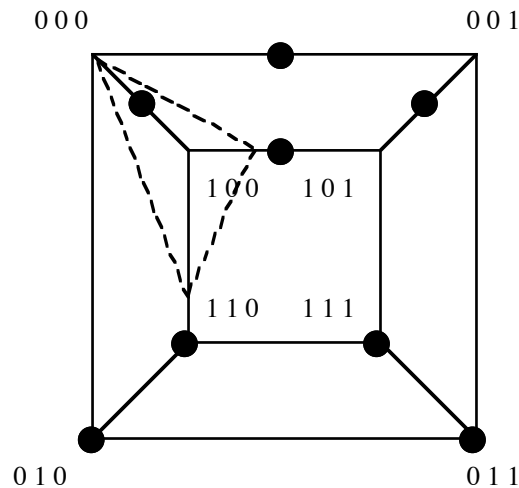
Why the funny values ? Why no [0 0 0] ?

It's easier to see what happens with the simpler but analogous 4-2-4 network. The "expected" representation in the hidden layer is binary, with one combination for each bit (doesn't matter which) in the input and output layers. Consider how the output bits have to respond to the hidden layer. Each has to turn on for just one combination of hidden-layer bits, so the coefficients of the four output neurons must identify thresholds placed as shown in this picture :



The special feature of the threshold isolating the $[0\ 0]$ corner ($2x + 2y = 1$) is that the $[0\ 0]$ corner is on the *positive* side.

Will that ever happen? If the threshold starts off pointing the wrong way - which it will if you always choose small positive weights - then perhaps it might tend to stay that way, though I don't really see why it should. Certainly, though, the states in the three-neuron network can be isolated by planes which keep the $[0\ 0\ 0]$ corner on the negative side. The distribution of the eight states is most interesting; their encodings in the hidden unit activities are plotted on the unit cube in this diagram:



In effect, the obvious encoding using the eight corners of the cube has been altered by rotating the four points in the $[?0?]$ plane to take them further away from the other four points which remain at the corners. There is no simple analogue of this operation in the simpler 4-2-4 network.

The triangle shows how the $[0.5\ 0\ 0]$ state can be isolated by a plane which still keeps the $[0\ 0\ 0]$ corner on the negative side. I have no idea whether that actually happens. An equally plausible reason for the staggered distribution of the points in the $[?0?]$ plane could be that it increases the distances between the eight points.

How often do you want zero input to give positive output? - maybe hardly ever? - but that's not the point. Note that it wouldn't happen with perceptrons either, because the perceptron convergence algorithm only adjusts the coefficients at synapses with positive inputs, arguing that a zero input

contributes nothing to the output, so there's no evidence to change the coefficient. It might happen with a Hebbian method, provided that you used +1 and -1 as the two binary values.

IS CODING IMPORTANT TOO ?

Returning now to the 4-2-4 network and the comforting simplicity of neurons which can only be on or off, supposing that we manage to map the one-out-of-four codes onto the four binary patterns which two neurons can represent, can we also do it for any arbitrary set of four four-bit codes ? If we assume threshold logic then, no, we can't.

That is because there are two problems which must be solved : first the input patterns must be translated into configurations of the hidden-layer neurons, then the hidden configurations must be used to reconstruct the original patterns in the output layer.

It is always possible to find a hidden-layer coding for which the first step is possible. For the four-bit case, each hidden neuron must be on for exactly half of the codes. That means that, in the four-dimensional space of the inputs, the planes defined by the coefficients of each hidden neuron must separate the points representing the input codes in two different ways. That can always be done (even in two dimensions) unless the four points are colinear, which is impossible ex hypothesi. Notice that it doesn't matter which planes are chosen, provided that they produce two different groupings.

The second step is another matter. Consider these four (not exactly randomly chosen) input patterns for a 4-2-4 network :

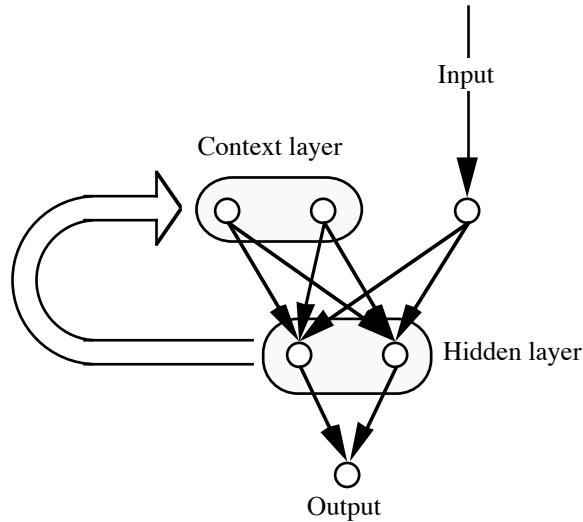
pattern a	1	1	0	0
pattern b	1	0	1	0
pattern c	0	1	1	1
pattern d	0	0	0	1

There are three possible partitions ({ a b | c d }, { a c | b d }, { a d | b c }), all of which are easily coded, and from which we must choose two which will be represented by the hidden neurons. But whichever pair we choose, it is impossible to find coefficients to generate the required behaviour in one of the first three output neurons. That's because each of the bits in the first three columns is always the exclusive-or of the other two, and the last column simply repeats the information of the first. So it's a trick - but we only need one counterexample to demonstrate that the required behaviour isn't generally possible.

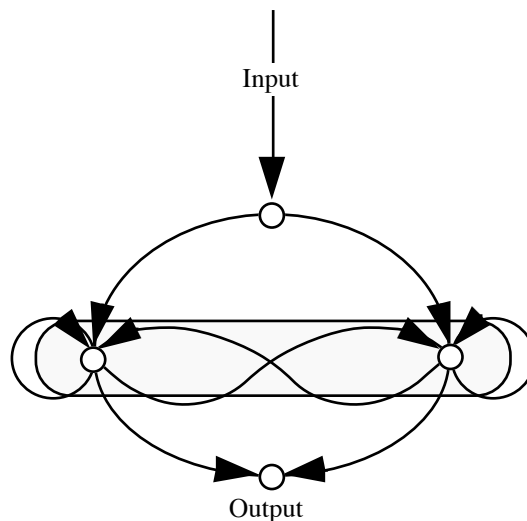
If we assume only positive weights then there are yet more constraints. Suppose that the input to a 4-2-4 network were presented as "thermometer" code - [0 0 0 1], [0 0 1 1], [0 1 1 1], [1 1 1 1]. Now, as each set of "on" neurons includes the set for the "smaller" inputs, there is no set of coefficients which will solve the first of the coding problems, because the assumption of positive weights guarantees that a neuron switched on for two inputs will necessarily be switched on for any set of three or more inputs which includes those two. Of course, it is possible to find suitable coefficients if negative coefficients can be used.

REMARKS ON ELMAN'S XOR NETWORK.

Elman's networks² are characterised by the presence of a *context layer*. This collection of neurons is used as a buffer to hold the state of one of the network layers which can then be used as input to that layer on the next cycle of computation.



After the computation, the contents of the hidden layer are copied without change to the context layer before the next computation cycle begins - in effect, then, the context layer is simply a device to implement this obvious fully connected feedback network in a computationally convenient (and easier to draw) manner :



The intention is to incorporate information about the previous state in the input to the hidden layer, so that its response will not be determined solely by the current input signal but to some extent by the history of the input stream.

Elman achieves some success with his network, but leaves a number of questions unanswered. He presents four examples, addressing different levels of complexity in his general problem of sentence analysis; I shall discuss only the simplest of these here, and the next simplest in bit-and-piece 4 below.

THE EXCLUSIVE-OR NETWORK.

The **XOR** network is intended to learn the **exclusive-or** function given training examples expressed serially - so the single-bit input is presented with a bit stream composed of many instances of the "words" 000, 011, 101, and 110 presented in random order without any separating symbol. The task of the

network is to predict the next input bit. Once trained, it sometimes gets the right answer, and does rather better every three cycles - but Elman's statement (page 11) that "the network predicts successive inputs to be the XOR of the previous two" is patently false, or the error would be zero every three cycles. (Because of his way of presenting results, it is hard to work out from his graph just what the probability of error is.) The real question is therefore : why are his results so bad ?

HOW IT MUST WORK.

Consider how the network must represent its state. (I shall assume binary neurons throughout this discussion.) If the output neuron is to compute a result which depends on the two preceding input bits, then the state of the hidden layer must reflect the two preceding bits. There are four hidden-layer states, which I shall write as $[\lambda \rho]$, with λ and ρ representing the activities of the left and right neurons, respectively; similarly, there are four input sequences, which I shall write as $\{\alpha, \beta\}$, with α and β representing the former and latter input bits. The representation is therefore possible, unless there are constraints which restrict the possible associations of sequences with hidden-layer states.

A CONSTRAINT.

There is at least one such constraint : it must be possible to compute the desired output value from the representation of the input sequence. The obvious encoding therefore doesn't work :

Input sequence	{ 0, 0 }	{ 0, 1 }	{ 1, 0 }	{ 1, 1 }
Hidden layer	[0 0]	[0 1]	[1 0]	[1 1]
Output	0	1	1	0

When we try to determine coefficients for the links to the output neuron, we end up with the standard **exclusive-or** problem, which we can't solve. Not all is lost, though, for we are not required to select this obvious encoding scheme. Indeed, the relationship between the input sequence and the hidden-layer representation is arbitrary, and in an experimental implementation would be chosen by the network itself. Here is a scheme which satisfies this constraint :

Input sequence	{ 0, 0 }	{ 0, 1 }	{ 1, 0 }	{ 1, 1 }
Hidden layer	[0 0]	[0 1]	[1 1]	[1 0]
Output	0	1	1	0

Now the output computation is trivially easy, because the desired output is just the right-hand bit in the hidden layer !

This is not an accident, though I'm not sure just how significant it is. The pairs in the hidden layer must all be different, so all the possible two-bit patterns must be used. The sequence of values for the left-hand bit in the hidden layer, reading across the table from left to right, must therefore conform to one of these patterns :

- $a \quad a \quad b \quad b$ (pattern 1)
- $a \quad b \quad a \quad b$ (pattern 2)
- $a \quad b \quad b \quad a$ (pattern 3)

where a and b stand for 0 and 1, but in either order. Exactly the same argument applies to the sequence of right-hand bits too; and the two sequences must necessarily be different to ensure that there are four different hidden-layer representations. The "obvious" solution combined patterns 1 and 2, but it is clear that with that choice, whatever the values chosen for a and b in each case, we end up with a linearly inseparable problem. The only alternative is to combine pattern 3 with one of the others, as in the second example (pattern 1 for the first bit, pattern 3 for the second) - but pattern 3 is just the **exclusive-or** solution.

ANOTHER CONSTRAINT.

We have therefore found a solution which satisfies the **exclusive-or** constraint, but there is a further constraint : it must be possible to calculate the values of the hidden layer from the input bit and its own previous state in the context layer. Generally, if the network is in the state corresponding to $\{ \alpha, \beta \}$ and the next input bit is γ , then the next state must be that corresponding to input $\{ \beta, \gamma \}$. Consider the development of a network using the second encoding shown above.

Input sequence	$\{ 0, 0 \}$		$\{ 0, 1 \}$		$\{ 1, 0 \}$		$\{ 1, 1 \}$	
Hidden layer	[0 0]		[0 1]		[1 1]		[1 0]	
Output	0		1		1		0	
Next bit	0	1	0	1	0	1	0	1
Input sequence	$\{ 0, 0 \}$	$\{ 0, 1 \}$	$\{ 1, 0 \}$	$\{ 1, 1 \}$	$\{ 0, 0 \}$	$\{ 0, 1 \}$	$\{ 1, 0 \}$	$\{ 1, 1 \}$
Hidden layer	[0 0]	[0 1]	[1 1]	[1 0]	[0 0]	[0 1]	[1 1]	[1 0]

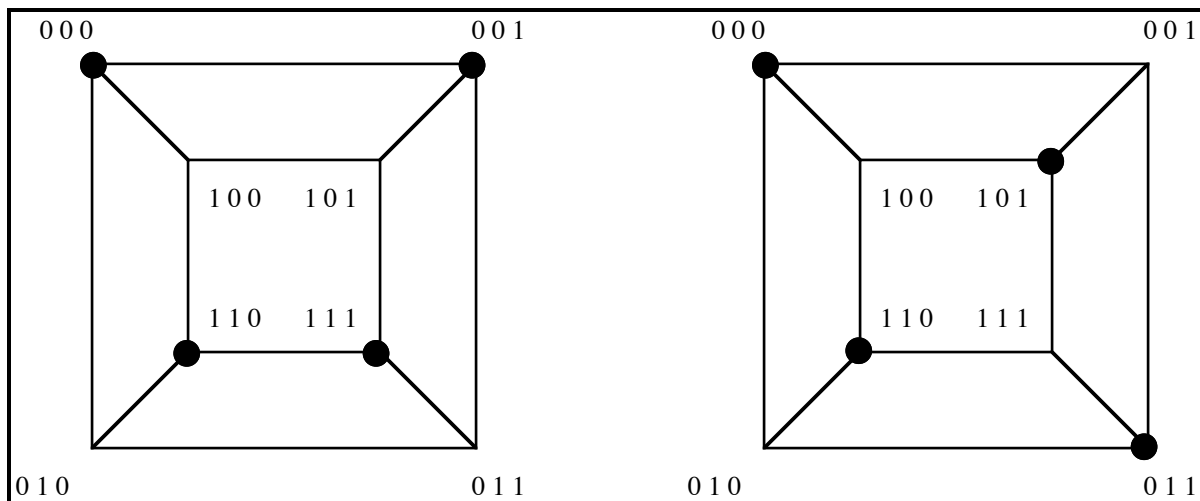
The question is now : can we calculate the final hidden-layer values from the previous hidden-layer value (now in the context layer) and the new input bit ? For example, from the old hidden-layer value [0 1] and the input bit 1 it must be possible to compute the new hidden-layer value [1 0].

The complete set of calculations which we want to compute is set out here. The upper rows show the old hidden-layer values and the new input, while the lower rows show the new hidden-layer values.

0 0 0	0 0 1	0 1 0	0 1 1	1 1 0	1 1 1	1 0 0	1 0 1
[0 0]	[0 1]	[1 1]	[1 0]	[0 0]	[0 1]	[1 1]	[1 0]

Each bit of the hidden-layer values is calculated independently, so we must be able to devise two separate schemes for the computation, one for each bit. Consider the left-hand bit. This must be 0 for the combinations (0 0 0), (0 0 1), (1 1 0), and (1 1 1), and 1 for the remainder. Similarly, the right-hand bit must be 0 for the combinations (0 0 0), (0 1 1), (1 1 0), and (1 0 1), and 1 for the remainder. It is not immediately obvious whether or not these functions can be achieved, but an alternative presentation makes the answer clear.

We appeal to the principle of linear separability. For three binary inputs, the space is the corners of a cube. In the diagrams below, the corners at which the neurons must compute 0 for the left-hand and right-hand bits of the hidden layer are shown as large black dots. It is clearly *impossible* to draw a separating plane for the either bit - so the computation is impossible.

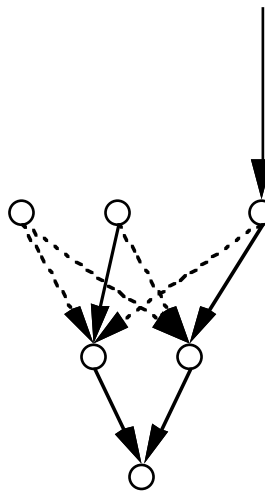


I have tried this routine with several different hidden-layer encoding schemes; they all fail somewhere. I conjecture that it's always impossible, but haven't been able to prove it in any general way.

WHAT (IF ANYTHING) IS SPECIAL ABOUT *EXCLUSIVE-OR* ?

Elman isn't really interested in computing **exclusive-or** - that's just a gimmick. What he really wants to do is to separate words in sentences, and he chose **exclusive-or** as a way of getting a small set of distinct patterns with which he could experiment. It's statistically nicer than (say) the **and** function, because you get equal numbers of 0 and 1 bits. It may be, then, that the difficulties we find with **exclusive-or** are not primarily concerned with the exclusive-orness of the function but are more general. What happens if we try to make an analogous **and** network ?

It turns out to be trivially simple. Following the "obvious" path we began with, everything works out. The final network looks like this, where solid and dotted lines correspond to weights of 1 and 0 respectively :



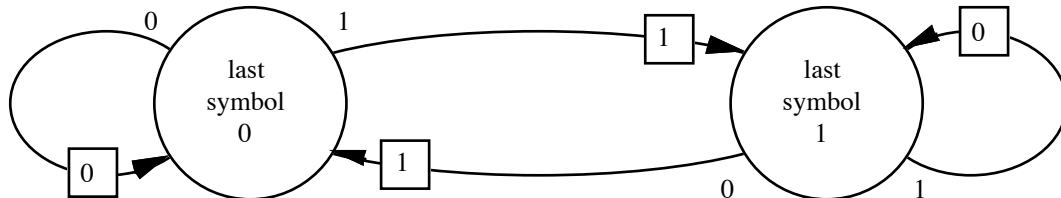
In effect, the context layer simply implements a shift register ! (- which is interesting in itself, because that's the other obvious way to incorporate history into a neural network, and it implements the first stage of the "obvious" exclusive-or encoding suggested earlier).

This sort of architecture will work whenever it is possible to calculate the output directly from a list of successive input values. I conclude that it is indeed the non-linearly-separable nature of the **exclusive-or** function which gives rise to the difficulties.

ELMAN AND FINITE-STATE MACHINES.

DESCRIPTION.

An alternative view of the network (again considered as a binary system) is as a finite-state machine. If I completely ignore the internal structure of the network, I can describe it very simply as a transducer :



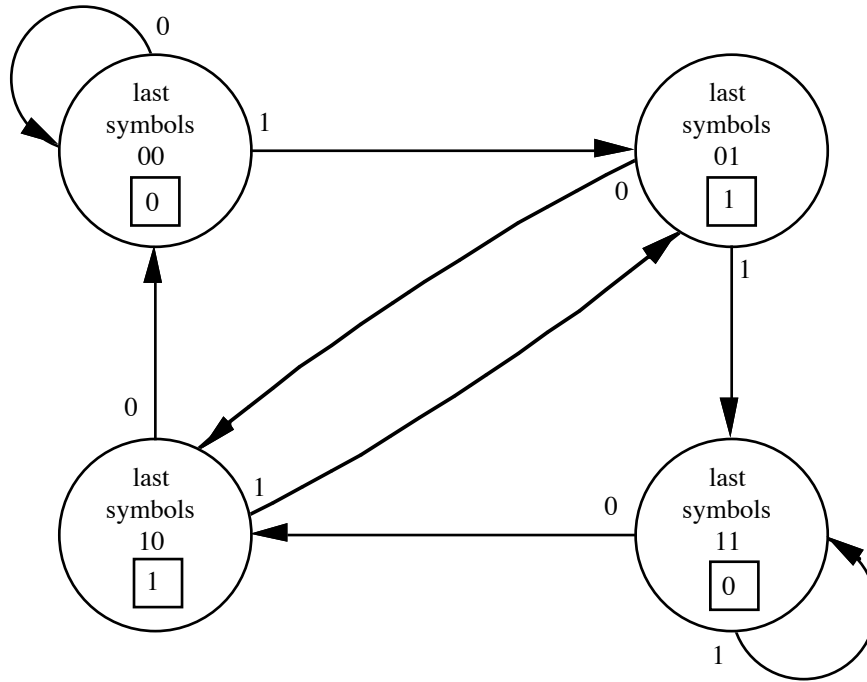
The two states correspond to the previous input signal received, the labels near the origins of the transitions are new input signals, and the numbers in boxes lying on the transitions are the outputs to be generated in response to these inputs.

(I think that this is a Mealy or Moore machine, but I don't remember which. Its output is associated with the transition rather than the state. The machine which I shall describe next is likewise a Moore or Mealy machine, in which the output is associated with the state.)

This machine describes the function of the network, but for more insight I have to analyse it further. The behaviour of the network is determined by the two neurons in the hidden layer and the single neuron in the output layer; how are these connected with the state of the finite-state machine ?

It is legitimate to identify the state with the pattern of activities within the hidden layer; the context layer is simply a device to store the previous state in order to simplify the computational machinery. Indeed, the diagram of the network I gave in my previous bit-and-piece is an excellent representation of the mechanism of the finite-state machine : the top layer (input and context layer, which means input and previous state) determines the behaviour of the hidden layer (the current state), which in turn determines the output (output).

If I regard the state as defined by the activation pattern of the hidden layer, then it is clear that the output is determined by the state, not directly by the transition. Distinguishing the states as represented by the hidden layer, which must correspond to the four possible sequences of two input symbols (because otherwise there isn't enough information do the calculations), I expand the previous machine into this four-state system :



Now the outputs correspond to states rather than transitions, so I have shown them within the nodes.

MACHINERY.

Now these two machines are perfectly ordinary and straightforward finite-state transducers. Unfortunately, though, they don't really model the behaviour of the network, and that's because they take no account of the machinery available to do the calculations. I contended in the preceding bit-and-piece that with the conventional neural machinery it is impossible to construct a network which behaves according to the specification. I shall try to write that down more formally.

The machine's current input, state, and output are represented by vectors **I**, **S**, and **O**.

- I** = { i_1, i_2, \dots, i_f }, where f is the number of neurons in the input (first) layer.
- S** = { s_1, s_2, \dots, s_h }, where h is the number of neurons in the hidden layer.
- O** = { o_1, o_2, \dots, o_u }, where u is the number of neurons in the output (ultimate) layer.

The vectors are related by the two equations

$$\mathbf{S} = \mathbf{F}(\mathbf{S}^*, \mathbf{I})$$

(where a superscript * denotes the *previous* value, and the square brackets denote concatenation of the two vectors), and

$$\mathbf{O} = \mathbf{G}(\mathbf{S}).$$

The functions **F** and **G** are composed of individual functions for each of the elements of the computed vector :

$$\mathbf{F}(\mathbf{S}^*, \mathbf{I}) = \{ F_k(\mathbf{S}^*, \mathbf{I}), 1 \leq k \leq h \};$$

$$\mathbf{G}(\mathbf{S}) = \{ G_k(\mathbf{S}), 1 \leq k \leq u \}$$

The functions F_k and G_k are the critical elements. These describe the functions implemented by the neurons, and are all of the same type. Each is evaluated as

$$N_k(\mathbf{V}) = M(\mathbf{V} \cdot \mathbf{W}_k),$$

where **V** is the input vector, M is some monotonically increasing bounded function, and \mathbf{W}_k is a vector of "weights" characteristic of the neuron concerned, one of which may be a "threshold" value. F_k and G_k are

all of this type, though they differ in their arities and in their weight vectors. In my descriptions of binary networks I have assumed that the M functions are step (Heaviside) functions; more generally, continuous sigmoid functions can be used, and the output of the network – though not necessarily of the individual neurons – is deemed to be 1 or 0 provided that it lies above or below some previously defined limits.

MORE RUDE THINGS ABOUT ELMAN.

In the preceding bit-and-piece I demonstrated, to my satisfaction if to no one else's, that the network proposed by Elman² to compute the exclusive-or function of serial bits could not work. Now I'm going to exhibit a similar network, based on the finite-state machine I discussed, which *will* work, extend the same principle to Elman's next more complicated example, and speculate on general principles behind these ideas.

Note to Dr Elman :

Dear Jeffrey :

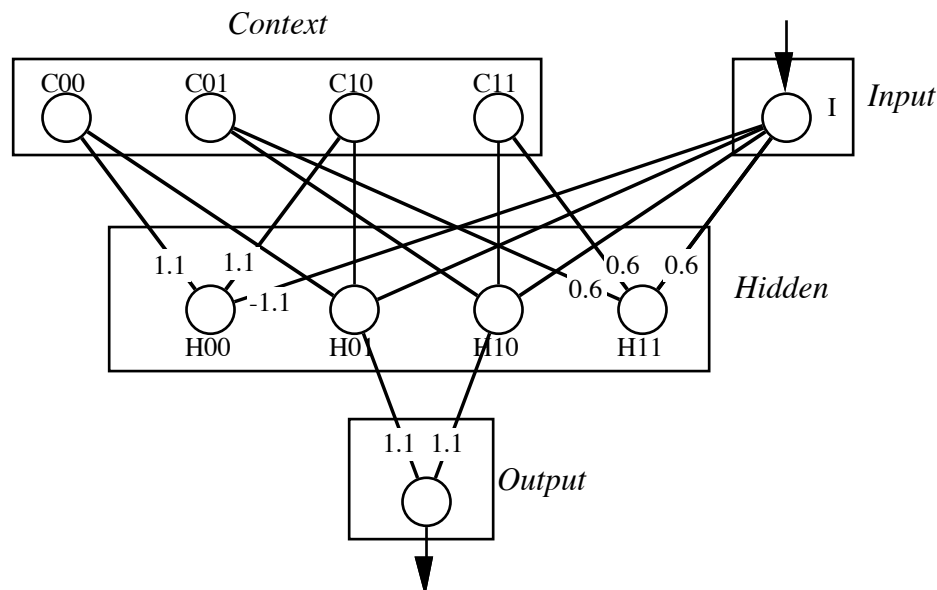
I'm not getting at you personally. I like your networks; if their basic design had been less elegant, I wouldn't have got anywhere at all with them. I'm just curious about what they do, how they do it, and how they can be encouraged to do it better.

THE EXCLUSIVE-OR NETWORK.

My earlier demonstration showed that an Elman network with two hidden neurons (and therefore two context neurons) could not implement the required **exclusive-or** function – although it could implement the functions like **and** and **or** for which the different output values occupy regions in the input space which are linearly separable. Whether an Elman network with a wider hidden layer could compute **exclusive-or** remained as an open question.

(It is important, here and throughout, to bear in mind just what these networks are supposed to be doing. They are intended to operate as transducers, not merely to simulate a finite-state acceptor. A network as originally proposed in Elman's paper, but without an output layer, can simulate the four-state finite state machine I showed in my earlier paper; it simply configures itself as a shift register. What Elman's network cannot do is simulate the finite-state machine and at the same time produce the output prediction. The machines I describe here will always do both.)

The network below, with a four-neuron hidden layer, will compute **exclusive-or**. (Whether or not it can be done with a three-neuron hidden layer remains open.) All neurons are supposed to be binary, with outputs of 0 or 1 computed by step functions with threshold values of 1. Lines with zero coefficients are not shown. Coefficients at H10 and H01 are the same as those at H00 and H11 respectively.

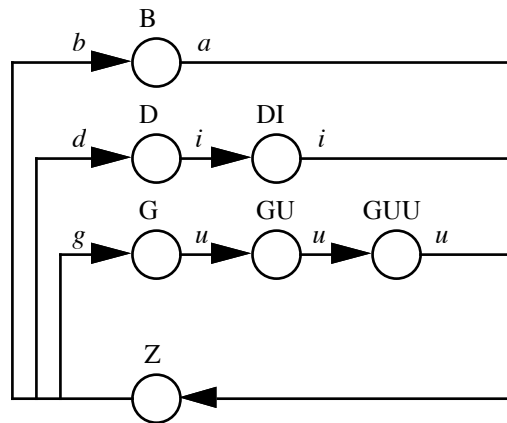


The network is based directly on the four-state finite-state machine which I devised earlier. In developing this network, I argued that, as the states of that machine were so chosen that each corresponded to a single output prediction, if an Elman network could be persuaded to represent these states as individual

hidden neurons it would be easy to compute the output. The next question was obviously whether or not the correct state transitions could be managed; in fact, it turned out to be easy, because at each transition only one of the context neurons is active, and this signal is readily combined with the single input signal to identify the correct successor state.

THE "WORDS" NETWORK.

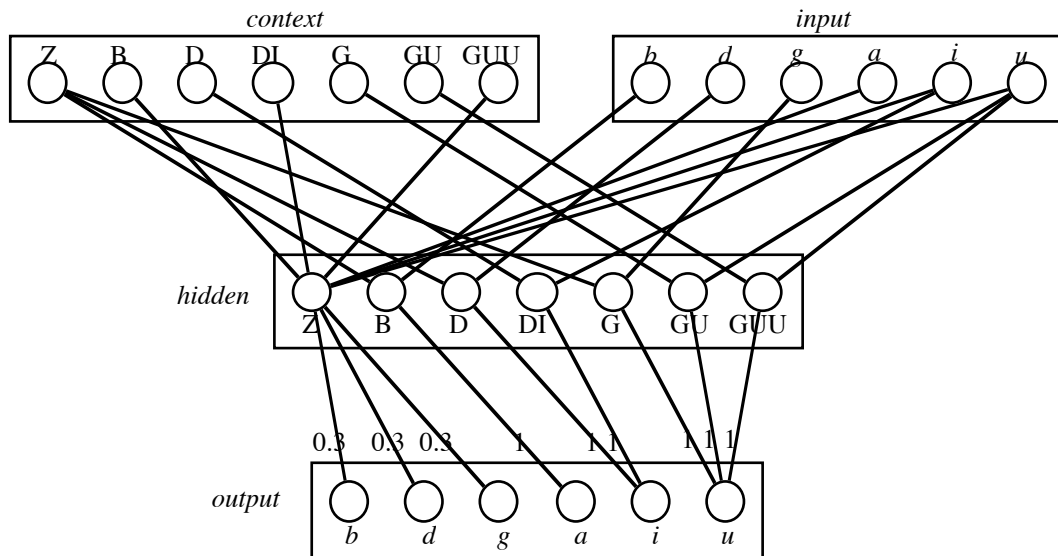
This is actually where the story started. Kim was experimenting with training networks to recognise signal sequences generated by finite-state machines, and conversation turned to Elman's second network. With this, Elman attempted to predict the next character in a sequence of artificial words of different lengths, constructed from the "words" *ba*, *dii*, and *guuu* concatenated in random order. (Elman calls them *letter sequences*, which is more accurate, but less euphonious, and harder to type.) The desired behaviour can be fully represented by this simple finite-state machine with seven states :



(The nodes of the network represent states after receiving the characters, not the character sequences themselves, though the two notions are connected. The Z state in particular represents end-of-word.)

In his implementation, Elman had achieved partial recognition using a network with 20 hidden neurons, which seemed a lot to encode seven states.

The network below will generate as perfect a prediction as possible using just seven hidden neurons. Hidden neurons are as before; all coefficients leading to hidden neurons are 0.6. Output neurons are linear units ($output = \sum_i coefficient_i * input_i$), with coefficients as shown and threshold 1.



The analogy with the exclusive-or network is close. Each hidden neuron represents one machine state, as shown on the diagram, and the performance depends on the input to the hidden layer being always from just one of the input neurons and just one of the context neurons.

In the interests of strict honesty, I record that I have cheated in this network. My model is not the same as Elman's : he represents the inputs and outputs in terms of the phonetic characteristics of the sounds of the various letters, not as a one-of-many code. On the other hand, he cheats too, for he doesn't explain how to segment the components of ii or uuu.

Correctly formed input will give perfect performance. Some input errors are detected, and result in the network's becoming inactive. For example, input *a* in state G results in no hidden neuron receiving sufficient input to turn on its output. On the other hand, the convergence of three transitions onto state Z leads to ambiguity, and input *a* in state DI causes an incorrect transition to Z. This can easily be mended, if required, by splitting Z into three states, one for each possible route into Z – say, aZ, iZ, and uZ. The output connections from each of these states, both in the hidden layer and in the context layer, are identical with those of the current Z state; the only difference is in the inputs to the hidden layer, when aZ receives input only from B and *a*, and so on. Once again, the performance is guaranteed because only one context neuron and only one input neuron is active at any time.

It is even easy to make the network restart from its error state. There are four input signals which lead to unique successor states : *b*, *d*, *g*, and *a*. Consider *b* as an example. If *b* is received, then if the context layer represents machine state Z the next state must be B; that transition is implemented by the network as shown above. If any other state is represented in the context layer, the the input *b* is an error, and should cause the network to enter its dead error state. Now, though, if the network is already dead, it may be reasonable to respond to *b* by restarting at state B. If this is accepted, the condition for a transition to B is :

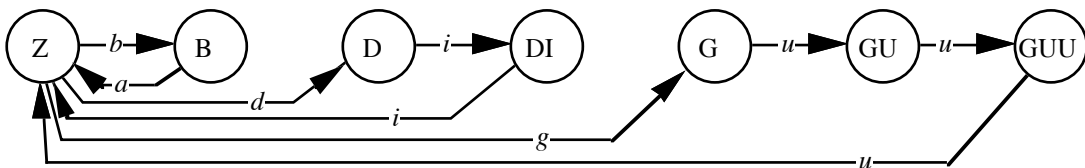
$$\{ b \ \& \ Z \ \& \ \neg(\text{all the other states}) \} \vee \{ b \ \& \ \neg Z \ \& \ \neg(\text{all the other states}) \} \\ = \{ b \ \& \ \neg(\text{all the other states}) \}$$

This function is implemented by setting the input coefficients for the B neuron in the hidden layer to 1.1 for the input from *b*, -1 for all the context neurons except Z, and 0 for all other input neurons and Z.

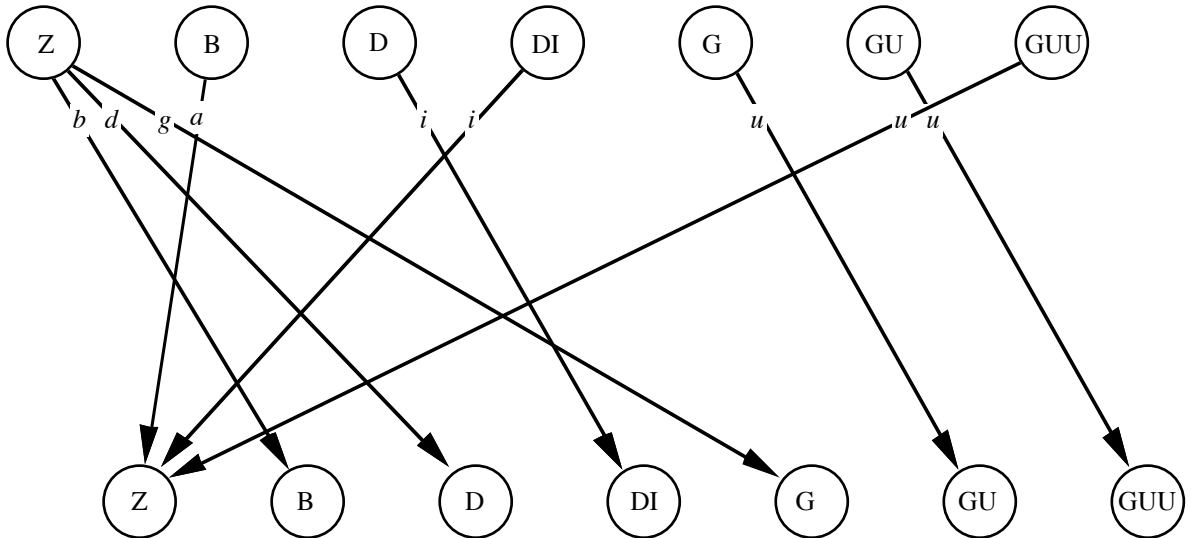
OBVIOUSER AND OBVIOUSER.

The longer you think about this, the more trivial it seems. Here's an informal demonstration, not amounting to a formal proof but pretty convincing nevertheless, that *any* finite-state machine can be represented as an Elman network. I'll present a construction, illustrated by Elman's "words" network.

1. Draw the network as a straight line.



2. Duplicate the nodes, giving an old set (with arrows) and a new set.
3. Transfer the tails of the arrows from the old set to the new set.
4. Separate the sets, pulling the new set upwards. (And a bit to the left, if you like.)



5. Draw a set of input nodes to the right of the new layer; there should be one node for each label which appears on the transitions.
6. Duplicate every arrow in the diagram.
7. Take one arrow of each pair labelled x and move its tail from the new set node to the input node labelled x .
8. Delete the labels in the interests of tidiness.

Voila. The links between hidden and output layers are left as an exercise for the reader.

ANALYSIS.

Why does it work ? – and why is it so easy ? I think the principal reason is the requirement that only one state is active at any time. Because of that, it is possible to construct connections which accept the required input while ignoring the rest, safe in the knowledge that if one input of a set is "on", then the rest are "off".

The coefficients for the hidden-to-output links are trivially easy to set. Only one of the hidden neurons will be active, so that may be connected directly to the required output, with coefficients set to activate the output neuron at a level appropriate to the confidence of the prediction. There is never any difficulty if an output neuron should be activated by more than one hidden neuron. (I remark that – so long as there is no ambiguity in the prediction – it is just as easy to generate an arbitrarily coded output signal. ASCII output is straightforward if you want it.)

The input to the hidden layer is a little more complicated, but not much. Once again we can rely on simplifying factors : only one neuron in the context layer will be active at any time, and, for the "words", only one input neuron will be active. The same is trivially true for the exclusive-or network, as there is only one input neuron, but in this case both the active and inactive states of the input neuron are significant. These two cases are similar, but not identical, and I shall treat them separately. Each has a "special case", in which prediction is still perfect given correct input but some cases of incorrect input are not identified; I shall explain how both special cases can easily be transformed into correct acceptors as well as correct transducers.

To begin with the exclusive-or network, it is clear that if a state is always entered by a transition caused by the same input symbol, then the link can be constructed; if the transition is cause by active input, then the hidden-layer neuron representing the new state receives signals from any set of precursor states and the input neuron, all with coefficient 0.6. As we know that only one of the context neurons can be active, the threshold can only be exceeded if a precursor state receives an active input. Similarly, if the transition to a new state is always caused by inactive input, the same connections are made, but with coefficients of 1.1 on the links from the context layer and -1.1 on the link from the input neuron. There remains the awkward case, with some transitions to a new state caused by active input and some caused by inactive input. I defer consideration of this special case for a paragraph.

The argument for the "words" network is similar, but in this case signals are always positive, so coefficients of 0.6 are always appropriate. There are two easy cases, one with all transitions to a new state caused by the same input, and the other with all transitions to a new state coming from the same old state. The special case this time has transitions to a new state from different old states caused by different inputs.

The two special cases can be handled in the same way, and we have seen an example of the case and its treatment in the transitions to the Z state in the "words" network. It is always possible to use the method described to resolve the problem : split the difficult new state into a set of substates each always entered by the same input signal, though perhaps from more than one different preceding states. Because the outputs from the separate substates can always be combined to provide an output representing the original state, if one is required, nothing has been lost; and because there is still always only a single state active in the hidden and context layers, the preceding arguments still guarantee that appropriate connections can be made. It is interesting that this stratagem of splitting a state into substates each entered in response to the same input signal is just that which I used to convert the two-state exclusive-or network into a four-state equivalent in bit-and-piece 2, though I didn't realise that until I'd worked out both separately.

WHY THE ONE-OF-MANY ENCODING WORKS.

I have demonstrated that an Elman network can be constructed to simulate any finite-state machine, provided that the input is represented as a one-of-many code. It is interesting to notice how the combination of the neurons' linear separability property and the restrictions on their possible inputs combine to give the required behaviour. Consider, for example, the inputs to the H11 hidden neuron in the exclusive-or network. These are from the context C11 and C01 neurons and the input I neuron, and each has weight 0.6. For the threshold value 1, a network with these coefficients will accept the function $(C11 \& I) \vee (C01 \& I) \vee (C01 \& C11) \vee (C01 \& C11 \& I)$, in effect slicing a corner tetrahedron from the cube representing the eight possible states. This is far more than is required – indeed, it is wrong. Nevertheless, it implements the correct behaviour because of the constraint that only one of C01 and C11 can be active in any network state, so that both the combinations with C01 & C11 can never occur.

A similar argument applies to the error recovery method. In order to behave as described, a neuron must respond *either* to a specific input neuron and one context neuron, *or* to the same input neuron and no context neurons. Quite generally, though, $(\text{anything} \& X) \vee (\text{anything} \& \neg X)$ is just one edge of the representative hypercube, so the problem is linearly separable.

IS IT USEFUL ?

The analysis (should it be sound) leads to two conclusions : first, that any finite state transducer can be represented by an Elman network; and, second, that there is a simple algorithm for constructing such a network. It will be useful if either of these is useful.

It is at least interesting that Elman's efforts are shown to be well founded, which was certainly not clear to me when I started this analysis. It is also interesting that an upper bound can be put on the size of the network needed for the task – indeed for either of the two tasks, accepting valid strings and rejecting invalid strings, in both cases predicting the next symbol. (It is, of course, trivially true that if we don't care about identifying any errors or making correct predictions, we can build an accepting network with precisely one neuron.)

The algorithm for building the network will be useful if anyone wants to build one. It beats the experience of those who have tried to train networks for similar purposes^{2,4}, when very long training periods appear to be the norm. But would anyone want to build such a network ? Unless there are applications in some area such as error-correcting data communications, data compression and expansion, or cryptography where symbol sequences must be translated into other symbol sequences according to very specific rules, we don't usually have the information we need to construct the machine. Adalines³ have been used for such purposes. Even if we had the information, though, why would we want a neural network implementation ? It's very easy to build a symbolic programme which implements a finite-state machine compactly and efficiently. It could be that in circumstances in which a very high speed would be

an advantage an implementation of a neural network in "real" hardware could be very fast, but that's a bit far-fetched.

A second consideration is that real systems, certainly those which approximate languages, have very many states, and the requirement for at least one neuron in each of the hidden and context layers for each of the machine states could be unduly demanding. Of course, that is not to say that a more compact network cannot be devised, and the work of Servan-Schreiber, Cleeremans, and McClelland⁴ (hereinafter SCM) shows that it is possible to do so. The one-of-many encoding stipulated for the states is very extravagant, and coarse coding can clearly lead to significant economies. (I discuss the work of SCM briefly below.)

SCM's work is more realistic, in that it begins with examples of the strings to be recognised. For that reason, it is also less well defined, as there are in general many finite-state machines which can accept or translate a given set of examples. Perhaps some of the difficulty encountered in trying to learn finite-state machines from examples comes from the multiplicity of potential targets; if there are many machines which could do the job, it is likely to be computationally hard to select one of them. Perhaps, with the knowledge that solutions must exist with the specific patterns of coefficients which I have investigated, it might be possible to develop methods which will seek for a solution in a more directed way.

NOW SOME RUDE THINGS ABOUT SERVAN-SCHREIBER AND HIS MATES.

SCM⁴ attempt to train Elman networks to predict sequences generated by finite-state machines. Here I present some observations on their paper, with particular note of topics which are related to my discussion above.

THEIR EXAMPLE.

They describe in some detail their experiments on a single grammar. This is a finite-state grammar on seven symbols (including explicit beginning and ending symbols), which can be represented by a finite-state machine of eight states (including explicit Begin and End states). I can write down the coefficients of an Elman network which accepts valid strings in the language and (probabilistically, assuming equal probabilities where there's a choice) predicts the successor characters.

FROM CONTEXT (word or digit) OR INPUT (letter) NEURON –	TO HIDDEN NEURON –							
	Begin	0	1	2	3	4	5	End
Begin	0	0.6	0	0	0	0	0	0
0	0	0	0.6t	0.6p	0	0	0	0
1	0	0	0.6s	0	0.6x	0	0	0
2	0	0	0	0.6t	0	0.6	0	0
3	0	0	0	0.6x	0	0	0.6s	0
4	0	0	0	0	0.6p	0	0.6v	0
5	0	0	0	0	0	0	0	0.6
End	0	0	0	0	0	0	0	0
<i>B</i>	0	0.6	0	0	0	0	0	0
<i>T</i>	0	0	0.6t	0.6t	0	0	0	0
<i>S</i>	0	0	0.6s	0	0	0	0.6s	0
<i>X</i>	0	0	0	0.6x	0.6x	0	0	0
<i>V</i>	0	0	0	0	0	0.6	0.6v	0
<i>P</i>	0	0	0	0.6p	0.6p	0	0	0
<i>E</i>	0	0	0	0	0	0	0	0.6

(The character suffixes appended to some of the coefficient values identify connections which correspond to different substates, as described briefly in the text which follows.)

FROM HIDDEN NEURON –	TO OUTPUT NEURON –						
	<i>B</i>	<i>T</i>	<i>S</i>	<i>X</i>	<i>V</i>	<i>P</i>	<i>E</i>
Begin	1	0	0	0	0	0	0
0	0	0.5	0	0	0	0.5	0
1	0	0	0.5	0.5	0	0	0
2	0	0.5	0	0	0.5	0	0
3	0	0	0.5	0.5	0	0	0
4	0	0	0	0	0.5	0.5	0
5	0	0	0	0	0	0	1
End	0	0	0	0	0	0	0

This network will also accept a few invalid sequences. As I described when discussing Elman's "words" network, this behaviour can be eliminated by splitting certain states according to the symbols which cause the transitions to the states, as indicated in the first table. Thus, state 1 must be split into t1 and s1, with input coefficients as labelled in the table. Similarly, states 2, 3, and 5 must be split into three, two,

and two substates respectively. This adds five states to the network. It is intriguing that this splitting is precisely that noted by SCM (top of page 173), and described by them as "not relevant to the prediction task". That's true – but it's relevant to the error detection task, which they don't notice.

That calculation didn't take me very long. It certainly sounds faster than "60,000 training trials", and gives a network which performs perfectly. As against that, though, my network is significantly bigger than theirs; they use only three hidden neurons, and find excellent performance. It seems that their network has developed much the same set of states, but with much more efficient coarse coding. It would be exciting to find a way to calculate a coarse-coded equivalent to my fine-coded networks, but I can't see how this can be done. The example of the exclusive-or network clearly shows that there are limits to coarse coding, though as my discussion is limited to binary neurons my conclusions aren't binding on their networks. An alternative might be to develop my networks in the direction of multiple-state neurons, though again it isn't clear how to do so; SCM use the example of four-state neurons in estimating the capacity of a network (page 179).

Remarks in the paper suggest that SCM are unclear about the relationship between finite-state machine and grammar – so they mention "the finite state automaton defined by the grammar" (page 169), implying that only one automaton will do. In fact, the number of states which can participate in the machine is limited only by the number of strings in the grammar (and not even by that if you really want to stretch it, but the additional nodes have little meaning), so in these examples there is no limit. More significantly, a state can always be split according to the sequence of symbols by which it has been reached, as I have done in resolving the error detection performance, and by splitting according to longer and longer preceding strings one can construct larger and larger machines. Whether or not these additional states are significant depends on the behaviour of the system; I have found no reason to extend my decomposition beyond the immediately preceding character, and only split in certain cases. SCM observe just this sort of splitting behaviour in their experiments (page 169), but refer to the additional states as "redundant", on the grounds that "If the network behaved *exactly* like a finite state automaton, the exact same patterns would be used during processing of the other strings ...". They are wrong; their network *is* behaving exactly like a finite state automaton – but it's not the automaton they were expecting. (They even make the curious point that "preventing the development of redundant representations may also produce adverse effects" (page 173). So why are they redundant ?)

In effect, the network begins by modelling a simple machine which encodes only the possible transitions from letter to letter without taking note of context (page 180), then refines this model to improve its predictions by specialising its nodes (which SCM oddly call "generalization" (page 173)). This is brought out very clearly by the variance results discussed on pages 182 – 183.

Their "problem [3]" (page 184) is reminiscent of Elman's "words" problem, and could be handled as I suggested earlier. I would devise a network with neurons Z (starting and finishing), P, PS, PSS, and PSSS, and corresponding nodes with T instead of P. I would achieve perfect performance with nine hidden neurons; SCM train a network with fifteen, and get good performance after 10,000 epochs of learning. They also mention experiments relating hidden-layer size to length of repeating string, but unfortunately don't report the results. If they resemble those of figure 19 (page 188), drawn for a related but more complicated experiment, then they are rather what I'd expect.

In summary, my work is related to that of SCM, but complementary. They have worked on learning to recognise samples of a grammar, I have started from the idea of simulating a finite-state machine. Their results clearly show that satisfactory transducers can be built much more economically than my simple construction suggests; the open question is whether it can be done without the arduous training which they need.

And the answer is "perhaps". Stay tuned.

COARSE THINGS ABOUT ELMAN.

No, not really. But the previous bit-and-piece ended with a cliffhanger about coarse coding, and this is the sequel.

It turns out to be possible to add a sort of coarse coding to my computed networks, simply by extending the technique I use to build them – specifically, the idea that a state is represented by a set of neurons with exactly one neuron active.

Unfortunately, my early euphoria was misplaced. My idea works, sometimes (I think), but it isn't quite as good as it sounds, because there is a cost. It turns out that there are some rather fierce constraints on how the network is built (and it's not obvious that it can always be built, but it sometimes can). The finite-state machine will work perfectly provided that it receives perfect input, but is very fragile to incorrect input, and it is not necessarily simple to generate the required output. Nevertheless, it's a clever trick, and perhaps we can patch it up to solve the problems. (The output is easy enough with a few more neurons, and the fragility can be solved with higher-level neurons in the hidden layer, but these stratagems might be considered cheating.)

My first line of attack in trying to extend this to coarse coding was to worry about possibilities of representing states using sets of neurons with exactly two neurons active. It is certainly possible, but you have to be rather careful in choosing neuron pairs to avoid untoward interactions in the input to the hidden layer. I didn't get far enough to work out just how good it would be.


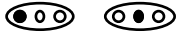
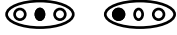

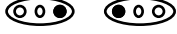

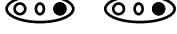
The second line was obvious once I'd thought of it : rather than increase the number of neurons, increase the number of sets. Take, for example, a network with six hidden neurons, which, using my method as previously reported, would represent six states. Now split the six into two subsets of three, and represent each state by two active neurons, one in each of the subsets. There are nine combinations, which is good, and all the previous theory will work, with slight changes to the coefficients, provided that once again we require that in all coded states exactly one neuron is active *in each of the subsets*.

How many subsets should we use ? It makes a difference : using three subsets each containing two neurons, the number of states coded is eight. Generally, for the maximum number of states, we wish to maximise $(N / S)^S$, where N is the number of neurons and S is the number of subsets. Interestingly, this is just the calculation which leads to the optimum number of states in elements used to store data, and the answer is that the size of the subsets should be e . To implement this result is practically tricky, but it is generally true that groups of three will be slightly better than groups of two.

EXAMPLE.

Here is a set of coefficients (the diagram is a nightmare) for Elman's "words" network done with six neurons in the hidden and context layers. I label the neurons $\{ \alpha_n, \beta_n, 1 \leq n \leq 3 \}$. I have encoded the seven-state machine; while the new machine has nine available states, there is little point in encoding the nine-state machine which I suggested as a guard against incorrect input because - as I shall shortly demonstrate - the new sort of machine has enough similar problems of its own to swamp those I corrected.

Encoding :

State	α	β	
Z	1	1	
B	1	2	
D	2	1	
DI	2	2	
G	3	1	
GU	3	2	
GUU	3	3	

(There is an obvious system to the allocation of representations. This is not essential for the working of the finite-state acceptor, but makes it possible to generate the required output easily. I shall comment further on this later.)

Coefficients :

FROM CONTEXT OR INPUT NEURON –	TO HIDDEN NEURON –					
	α_1	α_2	α_3	β_1	β_2	β_3
α_1	0.4	0.4	0.4	0.4	0.4	
α_2	0.4	0.4		0.4	0.4	
α_3	0.4		0.4	0.4	0.4	0.4
β_1		0.4	0.4	0.4	0.4	
β_2	0.4		0.4	0.4		0.4
β_3	0.4			0.4		
b	0.4				0.4	
d		0.4		0.4		
g			0.4	0.4		
a	0.4			0.4		
i	0.4	0.4		0.4	0.4	
u	0.4		0.4	0.4	0.4	0.4

FROM HIDDEN NEURON –	TO OUTPUT NEURON-					
	b	d	g	a	i	u
$\alpha 1$	0.3	0.3	0.3	0.5		
$\alpha 2$	-0.3	-0.3	-0.3	-1	1	
$\alpha 3$				-1		1
$\beta 1$				-1		
$\beta 2$				0.5		
$\beta 3$				-1		

COMMENTS ON THE EXAMPLE.

I think that it's right, but there is no guarantee. The major problem is fixing the connections leading into the hidden layer, and it arises for much the same reason as the difficulty with Z in the "words" network (in bit-and-piece 4), but I think in a more virulent form. The output is also harder to manage than before, though I've managed to cook a set of coefficients which I think will work. I'll discuss the parts separately.

The hidden layer.

Consider the inputs to an arbitrary hidden-layer neuron, called \mathbf{a} , and assumed without loss of generality (because the order of subsets is immaterial) to be in the first subset of neurons. Corresponding to any transition after which the neuron \mathbf{a} is active, there will be exactly one input to \mathbf{a} from each context-layer subset and exactly one from the input layer. I can identify these by listing the indices of the active neurons in each of the earlier layers – so, taking a network with two subsets as an example, the notation $(\mathbf{p}, \mathbf{q}; \mathbf{i})$ will denote the p th neuron in the first subset, the q th neuron in the second subset, and the i th input neuron. (There is little if any difference between the input and the subset neurons in the network machinery, but I've distinguished the input because it separates the two factors, input and state, which determine the behaviour of the finite-state machine.)

If $(\mathbf{p}, \mathbf{q}; \mathbf{i})$ is an input which should activate the neuron \mathbf{a} , then the subset state (\mathbf{p}, \mathbf{q}) encodes a network state (say, S) from which input \mathbf{i} will cause a transition to at least one other network state (T) which is encoded as $(\mathbf{a}, ?)$. I shall call the transition $S \rightarrow T$. To represent this transition in the network, the coefficients in the relevant connections will be set to (in this example) 0.4, so that all three inputs active will activate the neuron \mathbf{a} , but only two won't.

Suppose that $(\mathbf{r}, \mathbf{s}; \mathbf{i})$ is another valid input to neuron \mathbf{a} (perhaps $U \rightarrow V$; U and X are certainly distinct, as they have different context layer representations; V is $(\mathbf{a}, ?)$ ex hypothesi, and may or may not be identical to T , depending on the state of the second subset neurons). Further coefficients are set to 0.4 to represent this transition.

But now consider $(\mathbf{p}, \mathbf{s}; \mathbf{i})$. Given the description above, \mathbf{a} will be activated. That isn't necessarily catastrophic, but it might be. Suppose the transition represented is $W \rightarrow X$; provided that X is represented as $(\mathbf{a}, ?)$, all is well – but if X is $(\mathbf{b}, ?)$, we are lost.

This is quite a complicated constraint, and I don't quite know what to do with it. Elman's "words" network is rather simple, and I think I've managed to avoid complications, but I'm giving no guarantee. For larger networks with more transitions, and particularly as the number of subsets increases, I foresee hairy problems.

The output layer.

The difficulties here (in the example, though not in general) are of an entirely lower order of magnitude, which is just as well. The basic problem is that output neurons now receive input from two or more hidden-layer neurons, both (all) of which must be on to activate the output. It is therefore now important to worry about switching off the neuron when it receives only part of its correct input, so a lot of negative

coefficients appear, and funny values are needed to get activities reflecting the intermediate transition probabilities.

It would be even worse if the same output could be activated by machine states represented by arbitrary combinations of hidden-layer neurons, but I've fixed that in this example. By representing all states predicting i as $(\mathbf{2}, ?)$ and all states predicting u as $(\mathbf{3}, ?)$ I've simplified the problem a lot – but that won't always be possible. I suppose ?

ELMAN MEETS HOPFIELD.

Elman networks are Hopfield networks.

Well, no, they're not, but they're something of the sort. (Once again, they are not as close as I first claimed, but at least I didn't publicise this claim quite so broadly.) I almost got there in my "Remarks on Elman's XOR network" (bit-and-piece 2 herein), but didn't quite connect things up for quite some time. The conclusion depends on the observation that Elman's context layer is no more than a computational device for ensuring synchronised updating when running the network; it is merely a temporary buffer for the hidden layer's previous outputs while the new values are being calculated. As there is no restriction on the connections which can be made between the context layer and the hidden layer - indeed, in principle all connections are present and we merely adjust their strengths through the coefficients - that amounts to an implementation of a fully connected network. A Hopfield network is a special case of a fully connected network, with the very significant constraints that the strengths of the connections must be symmetrical, and that there is no link from a neuron to itself. I shall try to be more precise.

Elman's hidden layer	Hopfield
Each output linked to all neurons.	Each output linked to all neurons except itself.
Sigmoid output functions (but mine are thresholds).	Threshold output functions (but later sigmoids).
Coefficients unrestricted.	Coefficients symmetrical.
Synchronous evaluation of all neurons' output.	Serial evaluation, with new outputs effective immediately.
Input changed every time step.	Network runs until settled in a static environment.
Back-propagation training (but I can calculate the coefficients).	Coefficients can be calculated (or use Hebbian training).

Allowing for elements of simplification and overconfidence, the correspondence is close, except in the matter of symmetry. I have never seen this relationship mentioned anywhere in the literature - I'm sure I would have remembered it if I had.

ENLIGHTENMENT.

With this observation, some of my experiences with the Elman networks can be seen in a new light.

States : One would expect something like special states in any system with feedback loops, but a fair bit is known about the Hopfield network's states, and it will be interesting to see how far these ideas apply to Elman networks. Just how well it will fit depends on just how close the networks' architectures really are. I don't know how much work has been done on networks in which the symmetry requirement is relaxed; Kemp Ashby⁵ did some experiments with networks like Hopfield's but without the symmetry requirement and found that they could be trained to store and reproduce patterns, and with capacities much greater than those of the symmetrical Hopfield networks, but he didn't undertake a systematic study.

Stable states : The Elman networks would run under more Hopfieldian conditions if we kept the input steady until some sort of stable state was attained. What would happen ? The behaviour would be determined by the transition diagram of the finite-state machine; either the network would end up in a stable state (which might be switched off), or it would end up visiting states in a regular cycle. That's just like the Hopfield behaviour. Notice, though, that under normal Elman conditions this convergence is never allowed to happen, as only the first step of each sequence is accepted, after which the conditions are changed by introduction of a new input symbol.

Calculated coefficients : That the coefficients of my finite-state-machine networks can be calculated a priori given the required behaviour is quite unusual, but it's a property shared with Hopfield networks. Perhaps more intriguing is the a posteriori observation that the coefficients are set according to a Hebbian process, though one adapted to cope with the asymmetry of the connections. Instead of reinforcing coefficients of synapses connecting neurons which are simultaneously active, I have reinforced those connecting neurons active now with those required to be active in the next cycle. (And that's interesting too, because it's an idea I explored briefly during John Jensen's project days⁶ : "... We worry about the correlation between THIS cycle's input and NEXT cycle's output ...". Maybe it really works !)

POSSIBILITIES.

As well as throwing some (perhaps dim and flickering) light on past experience, the (perhaps dim and flickering) link with Hopfield networks suggests some new directions.

Hebbian learning : My approach lets me work out coefficients for rather simple representations, relying on "one-of-many" representations for input symbol and internal state. This is restrictive, both for the internal state, as I've already mentioned ad nauseam, and for the input encoding. In fact, in Elman's experiments on his "words" he used more complex input representations which corresponded to the phonetic composition of the "words" (and which accounts for the odd choice of letters). Perhaps Hebbian learning would be worth investigating as a way of coping with these more complicated representations - not necessarily by direct application, but by exploring how the Hebbian method works.

Real Hopfield networks : If we're so close, why not do the job properly ? Would real Hopfield networks do anything useful for us ? Perhaps they would, but only if we used them properly. The particularly valuable property of a Hopfield network is its stable states, so an implementation which gave the network time to settle down would be more appropriate than the Elman pattern in which only the first step is used. The obvious application for the stable states is in coping with different input speeds. If the network settled quickly enough to reach its stable state within at the most a small number of milliseconds, then differences in speech rate could be accommodated. There would have to be ways of allowing for significant differences between long and short sounds, but it's an interesting prospect.

REFERENCES.

- 1 : D.E. Rumelhart, G.E. Hinton, R.J. Williams : "Learning internal representations by error propagation", in *Parallel Distributed Processing* (D.E. Rumelhart, J.L. McClelland (eds), MIT Press, 1986), page 337.
- 2 : J.R. Elman : "Finding structure in time", *Cognitive Science* **14**, 179 (1990).
- 3 : B. Widrow, R. Winter : "Neural nets for adaptive filtering and adaptive pattern recognition", *IEEE Computer* **21#3**, 25 (March, 1988).
- 4 : D. Servan-Schreiber, A. Cleeremans, J.L. McClelland : "Graded state machines : the representation of temporal contingencies in simple recurrent networks", *Machine Learning* **7**, 161 (1993).
- 5 : K.W. Ashby : *Pattern recognition using neural networks*, M.Sc. Thesis, Auckland University Computer Science Department, 1991.
- 6 : G.A. Creak : *Coefficients for stable network behaviour*, handwritten scribble, 5-iv-1991.