Temporal Processing

Aims
• to consider the role of time in neural net processing
• to study feedforward neural nets that handle time-varying data
• to study certain types of recurrent network to handle time-varying data

Reference
Other references to journal articles occur as footnotes within these notes.

Keywords
data representation, time-delay, spectrogram, frame, Jordan net, state units/context units, temporal XOR, HU activation clustering, backpropagation through time.

Plan
• data & time in neural networks
• NETtalk
• time-delay neural units
• Time-Delay Neural Networks (TDNNs)
• Jordan networks
• simple recurrent networks (Elman networks)
• capabilities of simple recurrent networks
• tower / NARX networks
• unfolding recurrent networks - backpropagation through time

Representing Data in Neural Networks

<table>
<thead>
<tr>
<th>Binary encoded values</th>
<th>One-of-many choices (as output representation, called &quot;winner-take-all&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>○ ○ ○ ○ ○ = 0</td>
<td>● ○ ○ ○ ○ = 0</td>
</tr>
<tr>
<td>● ○ ○ ○ ○ = 1</td>
<td>○ ● ○ ○ ○ = 1</td>
</tr>
<tr>
<td>● ● ○ ○ ○ = 2</td>
<td>○ ○ ● ○ ○ = 2</td>
</tr>
<tr>
<td>● ● ● ○ ○ = 3</td>
<td>○ ○ ○ ● ○ = 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thermometer codes</th>
<th>Grey-scale values</th>
</tr>
</thead>
<tbody>
<tr>
<td>● ○ ○ ○ ○ = 0</td>
<td>○ = 0</td>
</tr>
<tr>
<td>● ● ○ ○ ○ = 1</td>
<td>○ = 1</td>
</tr>
<tr>
<td>● ● ● ○ ○ = 2</td>
<td>● = 2</td>
</tr>
<tr>
<td>● ● ● ● ○ = 3</td>
<td>● = 3</td>
</tr>
</tbody>
</table>

Representing Data in Neural Networks 2

• You would normally only use grey-scale values if the thing that the neuron represents is itself naturally graded (e.g. size, money, colour)
• You could use more than one grey-scale neuron to provide more “resolution”
• You could use a mix of different types of neuron
  For example, you could use a thermometer code system in which the final non-zero neuron had a grey-scale value to represent the fractional part of the number being represented.
• See also localist vs distributed representations, below.

Representing Data in Neural Networks 3

It only makes sense to speak of choosing representations for inputs or outputs (not HUs).

Input data we can usually control and thus specify its form. Pre-processing may be needed.

With output targets, we can specify the targets, but we may not achieve those targets, for a number of reasons:
1 imperfect learning - e.g. we specify we want an output of 1, but we only get 0.993, say;
2 inconsistent data - we specify a target of 1 in one training pattern, and an output of 0 in another training pattern with the same input values:
  2a the data may be valid, and the inconsistency may result from a stochastic aspect of the task (or some aspect of the task is not modelled by the input data collected);
  2b the data may contain errors - e.g. measurement errors in the equipment that collected the data.
Inconsistent Data

With inconsistent data, whether the data is valid or not, the result will be an output value between the target values:

- For example, if half the training patterns for a particular input pattern specify an output of 1 and the other half specify an output of 0, the learning algorithm will find a compromise output around 0.5.
- In the case of HU (hidden unit) activations, we have no direct control.
- HU values are determined by the input data, the training targets, the learning algorithm, and the random initial weights.
- We certainly don’t get to specify any form of HU representation.
- We can, however, inspect HU patterns and perform, say, cluster analysis on them.
- We’ll see this in the section later on the work of Elman.

Localist vs Distributed Representations

<table>
<thead>
<tr>
<th>localist</th>
<th>distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>cat</td>
</tr>
<tr>
<td>dog</td>
<td>dog</td>
</tr>
<tr>
<td>pig</td>
<td>pig</td>
</tr>
<tr>
<td>cow</td>
<td>cow</td>
</tr>
</tbody>
</table>

- In a distributed representation, the identity of the concept represented is distributed across the representation. There is usually a degree of redundancy in a distributed representation.
- When neurons, or weights, are deleted from a trained net that uses distributed representations, performance degrades gracefully. In contrast, if the "cow" neuron is deleted from a net based on the localist representation above, then all recognition of the concept of a cow is lost.
- Distributed representations may be chosen in some random way, or built on microfeatures. In a microfeature-based representations, the neurons encode features of the concept. In the example above, the first column could signify "mammal" and the fourth (say) could represent "ungulate".

Time in Neural Networks

Examples of situations where time is important include:
- language (speech processing, sequences of words);
- prediction (financial, electricity demand);
- signal processing (radar/sonar, diagnosing malfunctions from sequences of sensor readings);

Vision has a temporal component, too - many approaches to machine vision focus on interpreting a static image, but it is ultimately important to be able to, for example, decide which pixel in an image at time step t corresponds to which pixel in the image from time step $t-1$.

There are implicit constraints in image sequence interpretation - in the right-hand sequence of images, each black pixel moves at most to an adjacent square, so that there is less actual pixel motion than in the left-hand sequence.

Representing Time / Sequence

Spatial Time Representation - NETtalk

In some cases, sequences can be represented by making the whole of the (relevant part of the) sequence the input to the network, as in the NETtalk network, which learns to pronounce words given the letters in them, and has, as inputs, units representing 7 letters (current letter and 3 either side):

<table>
<thead>
<tr>
<th>NETtalk</th>
<th>NETtalk</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

This is sometimes referred to as a spatial representation of time. Spatial representations won’t work if the dependence on time is on information from arbitrarily far in the past, or even from a bounded but large number of steps in the past. It also requires some form of pre-processing to map the sequence into the spatial representation. With NETtalk, it also means that processing cannot begin until several letters beyond the one to be pronounced have been read.

Temporal Processing in NNs

- **NETtalk 2**

  • Consider the pronunciation of “a” in the following:
    - pa  pat  pate  paternal (ə)
    - mo  mod  mode  modern

  - NETtalk architecture

  ![NETtalk architecture diagram]

- **NETtalk Achievements**

  • the more items the network was trained on, the better it was at generalizing and correctly pronouncing new words
  • the performance of the network degraded very slowly as synaptic connections in the network were damaged
  • re-learning after damage to the network was much faster than learning during the original training.

- **Sequential Input as a Time Representation Technique**

  An alternative method of representing time/sequence is to present the networks with successive members of the sequence at successive processing cycles:

  ![Sequential input diagram]

  However, with a feedforward network, the output is a function of the current input only, so this approach to time would not work with standard feedforward nets.

  We’ll investigate how non-standard feedforward nets can handle this type of information.

- **Feedforward Nets**

  • A net is a feedforward net if the the graph consisting of the neurons as nodes and connections as directed edges is a directed acyclic graph.
  • One can then find nodes with in-degree zero (no connections coming into them) = input nodes; and nodes with out-degree zero (no connections leaving them) = output nodes; anything else is termed a hidden node or unit.
  • Nets like this are often drawn as follows:

    ![Feedforward net diagram]

    where each rectangle represents a group of neurons/nodes: the input nodes are on the left, output nodes on the right, and hidden units in the middle.

    • The arrows with ⊗ signify that there are connections, with trainable weights, between all neurons in one layer and all those in the next.
Temporal Processing in NNs

**Fully Recurrent Nets**

Another possibility for connectivity (also called topology) of networks is total interconnection - every neuron has a weighted connected to every other neuron. The connections may or may not be trainable. Such nets are called **fully recurrent**.

![Diagram of fully recurrent connections](image)

Totally connected nets are rather undisciplined. Backpropagation won’t work on them, as the problem of finding the error derivatives is too non-linear. (But see also backpropagation-through-time.)

In between feedforward and fully recurrent networks are many types of **partly recurrent** networks. A network is partly recurrent if there is a cyclic activation flow somewhere in it, that is, if the underlying directed graph (aka digraph) contains a directed cycle.

**Time Delay Neural Networks (TDNNs)**

Lang, Waibel, Hinton, and others devised this type of feedforward net. Their task was to classify sound spectrograms as representing one of the four “words” bee, dee, ee, and vee. A spectrogram is a 2-D representation of a sound (sequence). One dimension is time, the other is sound frequency. For a digital representation, we must divide the frequency spectrum into bands, one of which might represent the average intensity of sound of frequency [240-260] cycles/second. Similarly time is quantised, say into 12 millisecond slices or frames, with intensities averaged over the time slice as well as over the frequency band.

![Spectrogram of a sound](image)

Neural data structure for representing a spectrogram of 12 x 120ms frames sampled at 16 frequencies


**Time-Delay as a Memory Technique**

An NN that is to process temporal data needs some form of memory, either for past input patterns or for results of processing in previous time steps.

To see past input patterns, we can define a neural structure that copies its input to its output with a one-time-step delay. If such a processing unit Δ is presented with a sequence of inputs \( x(n) \), then its output at time step \( n \), \( \Delta x(n) = x(n-1) \):

\[
\Delta x(n) = x(n-1)
\]

[Note: Haykin denotes this delay operator \( \Delta \) as \( z^{-1} \).]

Such delay-line structures occur in the brain and are used in speech processing. A suitable array of delays converts a single input into a sequence of inputs similar to that used by NETTalk.

![Diagram of delay-line structure](image)

**TDNNs**

Speech processing involves telling when a word / phoneme starts. The TDNN addresses this by processing frames in a time-invariant way.

Each column of hidden units is connected to 3 frames’ input units, but the 48 weights coming into each column of HUs are the same for each column (obtained by averaging the changes that raw backprop would prescribe). The same applies to the hidden-to-output weights.

W.H & L’s TDNN outperformed other neural and non-neural techniques, coming in at 90.9% accuracy (as against 94% accuracy for humans tested on the same data). The TDNN discussed considered 3 consecutive frames at a time - obviously this is adjustable: 2-5 frames (at least) have been tried. Similarly for hidden to output connections.
### TDNNs 3

This is a simplified version of the input-to-hidden layer connections in a TDNN. Instead of 16 input units, the input columns contain 2 units. Instead of 8 hidden units, the hidden columns contain 1 unit each. Instead of a 1-to-3 relationship between inputs & hiddens, the diagram shows a 1-to-2 relationship. Instead of 10 hidden unit columns, there are 2 hidden unit columns. The weights of a given colour constitute a weight group.

### Temporal Translation Invariance

The TDNN algorithm achieves *temporal translation invariance* — that is, it doesn’t matter where in the frame the B/D/E/V sound begins – the TDNN will still recognise it. The sonogram can be translated backward or forward in time and the sound will still be classified correctly.

### TDNN People

<table>
<thead>
<tr>
<th>Alex Waibel</th>
<th>Geoffrey Hinton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now at CMU. His research interests centre around the interpretation and integration of speech, language and other human communication signals in the design of human-computer and human-human communication systems. (From his web page at <a href="http://www.csd.cs.cmu.edu/research/faculty_research/waibel.html">http://www.csd.cs.cmu.edu/research/faculty_research/waibel.html</a>)</td>
<td>Hinton is one of the great figures of modern neural network research. One of the three authors of the original Backprop paper (with Rumelhart and Williams), he has made several other major contributions to NN research. Recipient of the first Rumelhart prize, in 2001. See <a href="http://rumelhartprize.org/recipients.htm">http://rumelhartprize.org/recipients.htm</a> You’ll have a chance to see Hinton diagrams when we start using tlearn.</td>
</tr>
</tbody>
</table>

### TDNNs 4

The waveforms shown in column 1 are 144ms slices extracted from around the consonant-vowel transition in the words B, D, E, and V. Column 2 shows the 12 x 16 input spectrograms derived from these waveforms. Column 3 contains the hidden unit activation patterns triggered by these spectrograms in a 3-layer network with replicated hidden units. The 10 copies of 8 hidden units show the presence or absence of 8 features in 10 successive time positions. Column 4 contains output unit activation patterns from a 3-layer network with replicated output units. These patterns show the confidence levels of the network at successive time positions. Observe how the D and B detection events are localised in time. (Adapted from Lang, Waibel & Hinton, Neural Networks 3 (1990) Figure 1.)

### Recurrent Networks

A *recurrent* network is one which is not *feed-forward*. In a recurrent network, therefore, there will be at least one *cyclic path* of activation flow.

Recurrent networks can be used to process *sequences* of data. Sequences include samples of data at successive *time* intervals.

In prediction tasks (like predicting tomorrow’s Telstra share price), part of the input would likely be yesterday’s price. The rest of the input would be other economic information, etc. This could be modelled as follows, introducing a feedback (i.e. recurrent) connection into the network:

How do we *train* such a network?

This section focuses on ways of applying backpropagation to training certain kinds of recurrent networks, and on capabilities of these kinds of recurrent networks.
Jordan Networks

The model shown above feeds back output units to input units. Jordan instead fed back the output units to the hidden units, via a buffer termed the context or state units.


Training Jordan Nets

In training Jordan nets (also called “sequential nets”) the current state units are treated as extra input units. At time zero, the current state units are zeroed, and the input units (which Jordan termed the “plan units”) are set to the pattern representing the first sequence member. In the next time step, the plan units are set to the next sequence member, and the current state units are set to the previous output unit values (plus µ times the previous state unit value, 0 ≤ µ ≤ 1).

Backpropagation then works as usual. Note that in many sequence prediction tasks, perfect prediction is not possible (e.g. predicting share prices), so one does not expect the error to drop all the way to zero during training.

Digression - backprop in tlearn

The tlearn package is driven by 3 data files: .cf, .data, and .teach. The .cf is the configuration file, the .data has the training inputs, and the .teach has the corresponding training outputs. Here are examples for an 8-5-8 encoder (which copies 8 inputs to 8 outputs but "squeezes" the data through a hidden layer of 5 nodes):

858.cf:
NODES:
  nodes = 13
  inputs = 8
  outputs = 8
  output nodes are 6-13

CONNECTIONS:
  groups = 0
  1-13 from 0
  1-5 from i1-i8
  6-13 from 1-5

SPECIAL:
  weight-limit = 1.

Details under "Software" on the class home page.

The .data and .teach files are in this case identical:

858.data:
distributed
  8
  1 0 0 0 0 0 0 0
  0 1 0 0 0 0 0 0
  0 0 1 0 0 0 0 0
  0 0 0 1 0 0 0 0
  0 0 0 0 1 0 0 0
  0 0 0 0 0 1 0 0
  0 0 0 0 0 0 1 0

858.teach:
distributed
  8
  1 0 0 0 0 0 0 0
  0 1 0 0 0 0 0 0
  0 0 1 0 0 0 0 0
  0 0 0 1 0 0 0 0
  0 0 0 0 1 0 0 0
  0 0 0 0 0 1 0 0
  0 0 0 0 0 0 1 0

This error graph shows 5,000 epochs of training (× 8 patterns).
### Definition of Epoch

An **epoch** in neural network learning means *a complete pass through all of the training data.*

In the example above, there are 8 training patterns, so an epoch consists of presenting all of the 8 training patterns and updating the weights (either after each pattern, or at the end of the epoch).

In tlearn, the term “sweep” is also used – a sweep means the presentation of a single training pattern, and any consequent weight updates.

So in this case, an epoch is 8 sweeps.

In general, if there are $n$ training patterns, then an epoch = $n$ sweeps.

### Implementing Jordan Nets with the tlearn Package

Let's train a Jordan net with 4 inputs and 4 outputs to simulate a shift register (shifting both forward and backward).

![Jordan Net Diagram]

### Jordan Network Definition

Here is a network definition for the tlearn simulation package, with configuration file seq.cf:

```plaintext
NODES:
nodes = 12
inputs = 4
outputs = 4
output nodes are 5-8

CONNECTIONS:
groups = 0
1-8 from 0
1-4 from i1-i4
5-8 from 1-4
1-4 from 9-12
9-12 from 5-8 = 1. & 1. fixed one-to-one

SPECIAL:
linear = 9-12
weight-limit = 1.
```

The NODES section tells us that 1-4 are hidden units, 9-12 are context units.

The way tlearn works, it is critical to have any copyback units have numbers higher than the nodes to which they are providing input.

The specification `linear = 9-12` in the SPECIAL section ensures that the values copied back to nodes 9-12 are not "squashed" by passing them through the logistic function.
Training a Jordan Network to be a Shift Register

Training patterns for shift register:

<table>
<thead>
<tr>
<th>seq.data:</th>
<th>seq.teach:</th>
</tr>
</thead>
<tbody>
<tr>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>1 0 0 0 A</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>0 1 0 0 B</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>0 0 1 0 C</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>0 0 0 1 D</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>0 0 1 0 E</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>0 0 1 0 F</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>0 1 0 0 G</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>1 0 0 0 H</td>
</tr>
</tbody>
</table>

The sequence in which these patterns are presented is important.

The Jordan net feeds back its output values to the hidden layer. Thus the input to the hidden layer consists of the previous output, along with a pattern indicating whether to shift left or right.

When you start this net with 0s in the state units and 1 0 1 0 in the plan units, it is supposed to produce, in sequence, the outputs A, B, C, and D. When you start it with 0s in the state units and 0 1 0 1 in the plan units, it is supposed to produce the outputs E, F, G, and H.

Jordan Network Options in tlearn

It is also necessary to set some options in tlearn's Network/Training Options dialog: first click on the "more" button to get the full range of options, then check the "Use reset file" box and the "Train sequentially" button, and type 10000 into the "Training sweeps" box.

The "Use reset file" checkbox (or -X flag in Unix command line version) tells tlearn to look at the contents of seq.reset:

<table>
<thead>
<tr>
<th>seq.reset:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

This specifies 2 resets, before the 1st and 5th training patterns. 'Reset' signifies here that each copyback unit is reset to zero at the appropriate time. This ensures that the network does not try to learn sequential connections where we do not intend it to do so.

When testing that the network has learned, in tlearn, don't forget to specify that resets are to be used during testing!

When trained and tested with 2 instead of 4 input units, the net didn't train correctly on all occasions (training succeeded on 7 of 10 attempts after 125,000 epochs, compared to 9 out of 10 for the original version of the network).

People in Backprop-trained Recurrent Nets

<table>
<thead>
<tr>
<th>Michael Jordan</th>
<th>Jeffrey Elman</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Michael Jordan" /></td>
<td><img src="image2" alt="Jeffrey Elman" /></td>
</tr>
</tbody>
</table>

Now at Berkeley, Jordan has worked across a number of disciplines since he invented Jordan nets in the 1980s – including Bioinformatics, Classification, Control and Reinforcement, Graphical Models, Human Motor Control, Independent Component Analysis, Information Retrieval, Kernel Methods, Mixture Models, Neural Networks, Nonparametric Bayes, Optimization, Spectral Methods, Speech and Language, and Variational Methods.

Elman was the 2007 recipient of the Rumelhart Prize (US $100,000 for a contribution to the theoretical foundations of human cognition). He is a Distinguished Professor of Cognitive Science at UCSD, and co-director of the Kavli Institute for Brain and Mind. [http://crl.ucsd.edu/~elman](http://crl.ucsd.edu/~elman)

Elman’s group produced tlearn, as a software adjunct to the book Rethinking Innateness by Plunkett and Elman.

Simple Recurrent Networks (Elman Nets)

These networks differ from Jordan networks in that the recurrent connection is from the hidden layer to the state vector, not from the output layer.

In this case, the number of state vector units must equal the number of hidden units. As before, state vector units look like input units to the backpropagation algorithm.

At time zero, the current state units are set to a fixed value (0.5 for Elman), and the input units are set to the pattern representing the first sequence member. At subsequent time steps, the input units are set to the next sequence member, and the current state units are set to the previous hidden unit values. Backpropagation works as usual. If several different sequences are being used, the state unit values are reset to the fixed value at the start of each new sequence.

3 J.L. Elman, Finding structure in time, *Cognitive Science* 14 (1990) 179-211. Many of the diagrams in this section are adapted from this paper.
Capabilities of Simple Recurrent Nets

Most of Elman’s tasks were linguistic in nature, but should be taken as indicating types of problems that simple recurrent networks can solve. We’ll look at:

- a temporal version of the XOR problem
- finding the structure in letter sequences
- discovering the notion “word”
- discovering lexical classes from word order
- the importance of starting small

1. A Temporal Version of the XOR Problem

XOR is usually presented as a problem involving 2-bit input vectors (00, 11, 01, 10) yielding 1-bit output vectors (0, 0, 1, 1 respectively).

One way to translate this into a sequence format is to construct a sequence of 1-bit inputs by presenting the 2-bit inputs one bit at a time, followed by the 1-bit output, then continuing with another randomly chosen input/output triple.

A sample input sequence might be:

```
1 0 1 0 0 0 0 1 1 1 1 0 1 0 1 ...
```

Here the first and second bits are XOR-ed to give the third; the fourth and fifth are XOR-ed to give the sixth, etc.

Learning XOR then becomes a prediction task: the network has to learn to predict the next bit in the sequence. Thus the input / target-output performance on the example bitstring would be:

- Input: 1 0 1 0 0 0 0 1 1 1 1 0 1 0 1 ...
- Output: 0 1 0 0 0 1 1 1 1 1 0 1 0 1 ? ...

Of course, the network will only have any chance of doing this at better than chance every third bit.

Elman's Sequential XOR Net

Elman trained a 1-2-1 simple recurrent network to do this task, using a sequence of 3,000 bits of the form described above.

He trained the network for 600 epochs with an unspecified learning rate. (Learning rates used for simple recurrent networks are usually relatively low - 0.05 to 0.15 might be typical. Pattern-by-pattern learning works best, too.)

a) The figure above shows the type of pattern-by-pattern error he found;
b) He averaged his error measure over 1200 trials;
c) Error drops every third value - because these are predictable;
d) Elman reported that one of his two hidden units was sensitive to 1,0 and 0,1 and the other hidden unit was activated when 1,1 or 0,0 showed up.
2. Structure in Letter Sequences

Elman’s next experiment was with a phonological encoding of sound sequences. His input was a sequence of 1000 “speech sounds” chosen at random from the set (ba, dii, guu) with each letter represented as a 6-bit “phonological” vector:

<table>
<thead>
<tr>
<th>consonant</th>
<th>vowel</th>
<th>interrupted</th>
<th>high</th>
<th>back</th>
<th>voiced</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

An example of the sort of letter sequence the system had to process would be diibaguubadiidiiguuu - with each letter represented by a bit vector as above.

**System task**: predict the (bit vector for the) next letter.

**Elman’s architecture**: simple recurrent network with 6 inputs, 6 outputs, 20 hidden units (and so 20 state units). Elman trained the system for 200 epochs (“1000 training patterns”).

### Discussion of Results for badiiguuu Net

- Once the network sees d, b, or g, it knows that it will see 2 i’s, 3 u’s or an a, respectively, so error drops for these, then jumps for the next d/b/g (which is unpredictable).
- Thus the net learns what is possible to learn, in a partly random environment.
- Analysing the output bit-by-bit, the network always gets low error on bit 1 (which indicates whether the letter is a consonant or not). It can tell when a consonant is coming - it just can’t tell which one.
- In the XOR experiment, the amount of context required was fixed (the previous input only was needed). This experiment shows that simple recurrent networks can cope with variable length context.

### 3. Discovery of the Concept of a Word

In speech, words are often run together. How do kids learn the concept of word as they acquire language? Elman’s next experiment bears on this. He generated 200 sentences, of lengths from 5-9 words, from a vocabulary of 15 words and ran them all together:

manyyearsagoaboysandgirlslivedbytheseaheplayedhappily...

This provided him with 4963 letters (1270) words. He then converted them into a 5-bit code and trained a 5-20-5 simple recurrent net on a next-letter prediction task, giving the network 10 presentations of the sequence of 4963 letter codes.

Large increase in error usually coincides with the beginning of a new word (more precisely, with the beginning of a common repeating pattern). This net has discovered the concept of word, or rather the interpreter can learn the concept of word by looking at the error patterns.
4. Discovering Lexical Classes from Word Order

Word order is important in English - it affects meaning.

Order is constrained by syntax, semantic considerations (e.g. stones don’t eat, so you won’t see the word eat following the word stone), subcategorisation (you can have an embedded sentence after I believe, but not after I ate) and discourse considerations (Ice cream, I like vs I like ice cream).

Linguists classify words into groups: nouns, verbs, adjectives, adverbs, determiners, prepositions, ...

They may also subclassify these classes: verbs are subclassified into intransitive verbs like sleep, transitive verbs like eat, bitransitive verbs like give (I sleep, I eat my dinner, I give my mother flowers).

Can a simple recurrent network capture this kind of information from a collection of English sentences?

Elman did an experiment in this area with simplified lexical categories, vocabulary, and grammar. The grammar had some semantic features (the grammar you learn in school normally only covers syntax - it is assumed that you understand the world well enough not to need semantic well-formedness rules!)

Elman’s Lexical Categories

| NOUN-HUMAN  | man, woman |
| NOUN-ANIMATE| book, rock |
| NOUN-AGGRESSIVE | dragon, monster |
| NOUN-FOOD | glass, plate |
| VERB-INTRANSITIVE | cookie, bread |
| VERB-TRANSITIVE | think, sleep |
| VERB-AGPAT | see, chase |
| VERB-PERCEPTUAL | move, break |
| VERB-DESTROY | break, smash |
| VERB-EAT | eat |

Elman’s Sentence Templates

Sentences were all simplified two or three word sentences fitting the following patterns:

| NOUN-HUMAN  | VERB-EAT  | NOUN-FOOD |
| NOUN-HUMAN  | VERB-PERCEPTUAL | NOUN-INANIMATE |
| NOUN-HUMAN  | VERB-DESTROY | NOUN-FRAGILE |
| NOUN-HUMAN  | VERB-INTRANSITIVE | NOUN-HUMAN |
| NOUN-HUMAN  | VERB-TRANSITIVE | NOUN-INANIMATE |
| NOUN-HUMAN  | VERB-AGPAT | NOUN-INANIMATE |
| NOUN-ANIMATE | VERB-EAT  | NOUN-FOOD |
| NOUN-ANIMATE | VERB-TRANSITIVE | NOUN-INANIMATE |
| NOUN-ANIMATE | VERB-AGPAT | NOUN-INANIMATE |
| NOUN-ANIMATE | VERB-AGPAT | NOUN-INANIMATE |
| NOUN-AGGRESSIVE | VERB-DESTROY | NOUN-FRAGILE |
| NOUN-AGGRESSIVE | VERB-EAT  | NOUN-HUMAN |
| NOUN-AGGRESSIVE | VERB-EAT  | NOUN-ANIMATE |
| NOUN-AGGRESSIVE | VERB-EAT  | NOUN-FOOD |

An example of the third last template (NOUN-AGGRESSIVE VERB-EAT NOUN-HUMAN) would be dragon eat woman.
Interpreting the Results of Elman's Experiment

- Error is not useful in interpreting the results in this case (because of the sparse input/output vectors, turning off the output vectors - that is, setting them to the value zero - gave a RMS error of 1.0, as opposed to the figure of 0.88 after 6 epochs of training).

- What the network might be expected to do in this situation is learn, for each context, the probability of each successor word, and activate the output units in proportion to these probabilities. For example, if the previous two words were man eat ... then given the templates used, the probability of bread and cookie would be high, and that of every other word would be low.

- It is possible to compute these likelihoods directly from the original data, so Elman did so. Every word in a sentence was compared against all other sentences that are, up to that point, identical. These constituted the comparison set. The probability of occurrence for all possible successors was then determined from this set. This yields a vector for every word in the training sequence.

- Consider the partial sentence dragon eat ... According to the sentence templates, dragon eat could be followed by a NOUN-HUMAN or a NOUN-ANIMATE or a NOUN-FOOD (last 3 templates). So possibilities include dragon eat man, dragon eat woman, dragon eat cat, dragon eat mouse, dragon eat cookie, and dragon eat bread.

Elman's Interpretation Using HU Activation Clustering

- Elman hypothesized that the internal representations computed from the input sequence by the hidden units held information about the lexical classes and state in processing a sentence.

- He collected the hidden unit activations from a pass through the training set (27,500 x 150-component vectors).

- He then sorted them according to the word which caused the activation (29 collections of many 150-component vectors).

- He then averaged each of the 29 collections, so that he had for each word, a 150-component vector representing the average activation which it produced in the hidden units.

- He then applied cluster analysis to this set of 29 x 150-component vectors, and obtained the following clustering:
What has Elman’s Net Learned?

- Inputs were presented in context, and the hidden units vectors averaged across multiple contexts.
- Arguably, the network has, from an operational point of view, learned the concepts of noun and verb, and of food, fragile object, inanimate object, etc., from the way in which the words are used in the training set.
- It is also possible to do cluster analysis in context.
- To do this, take all the cases where the sentence woman see BOY occurs in the 10,000 training sentences, and average the hidden unit patterns produced for the input BOY. Do the same for man see BOY, woman see GIRL, man see GIRL, dragon eat GIRL, monster eat GIRL, lion eat GIRL, etc. Then perform cluster analysis on these HU patterns in context.
- Elman found that, for example, dragon eat GIRL, monster eat GIRL, and lion eat GIRL were clustered together, but away from woman see GIRL, and man see GIRL. All the BOY patterns were separated from all the GIRL patterns.
- So the network has learned to appreciate context, which is the first step in understanding how discourse considerations affect word order.

5. The Importance of Starting Small

Elman went on to subsequent work on language learning in simple recurrent nets. He noted that the greatest learning in humans occurs at a time when the most dramatic maturational changes are occurring, too.

His early attempts to get SRNs to learn to predict the next word in sentences with embedded clauses, like The girls whom the teacher picked practise every afternoon, failed.

Subsequently, he tried:

- training the SRN just on simple sentences like The girls practised every afternoon and The teacher picked the girls
- then training the SRN on sentences with embedded clauses

This worked. In effect, the SRN learned the simple grammar first, then with the concepts of simple sentences (like nouns and verbs and noun phrases) under its “belt”, it was able to handle the more complicated sentences.

---

Starting Small Can Mean Starting with Limited Memory

Elman noted that this corresponded to children's language production in the sense that they start with simple utterances, which gradually become more complex.

However, the "training data" that children learn from is not simplified in the way that the training data that this SRN learned from is simplified. Adults use modified language around very young children, but by no means restrict themselves to simple sentences.

However, children's working memory capacity and attention span increases as they develop. Elman's next simulation involved training the network using the full language (including complex sentences) from the beginning, but restricting the SRN's memory in five phases as follows:

1. Recurrent feedback eliminated every 3-4 words (randomly) by resetting all context units to 0.5.
2. Recurrent feedback eliminated every 4-5 words.
3. Recurrent feedback eliminated every 5-6 words.
4. Recurrent feedback eliminated every 6-7 words.
5. Recurrent feedback not interfered with at all.

He found that the network could learn the grammar, but that a prolonged phase 1 (12 epochs as opposed to 5) was necessary to achieve this.

In other words, gradually incrementing the "memory" of the SRN made the grammar almost as learnable as had been the case when the grammar had gradually been made more complex.

Summary of Elman’s Conclusions

- Some problems change their nature when expressed as temporal events: his XOR net developed a solution different to what is typically found when feedforward nets learn XOR - typically one HU would activate for “mid-frequency” sequences (10 and 01) and the other for “high and low frequency” sequences (00 and 11).
- Time-varying error signal can be used as a clue to temporal structure: as with the nets that learned to recognize word boundaries.
- Increasing the sequential dependencies in a task does not necessarily result in worse performance; the two word boundary nets both did fine, even though the second one (the manyyearsagoaboyandgirl... net) had a longer sequence to train on, a contentless input representation, and many more patterns to learn (15 words as opposed to 3).
- Representations need not be "flat", atomistic, or unstructured: the cluster analysis in the final task shows that sequential inputs may give rise to internal representations that are hierarchical in nature.
- It is wise to start training with simple data and progress to more complex data.
- In some cases it can make sense to start by limiting memory sequences by re-initialising the context units every few time steps.
- It can help to use carefully chosen data, or noisy data, at the start of training.

Elman’s Thoughts on Learning

Elman noted that the derivative \( \phi(x) \) of the sigmoid function \( \phi(x) = 1/(1 + e^{-x}) \) has its maximum at \( x = 0 \), and for a backprop net that starts with small random values for weights, 0 is the expected value for the total net input, \( net \). As the delta rule’s weight change prescription for output layer nodes is

\[
\Delta w_j = \eta \phi(\text{net}_j) \text{error}_j
\]

weight changes are likely to be large to begin with.

Subsequently, as some of the weights become larger in magnitude, \( net \) is likely to be larger, so \( \phi(\text{net}_j) \) is likely to be smaller.

Thus much of the learning may occur early on, when the net has least data to go on. Subsequently, learning decreases. "The more the system knows (whether right or wrong), the harder it is to learn something new." How can this be fixed?

1. Have better training data. Then the system starts off in the right direction.
2. Have "worse" training data. This can happen if the training data are noisier at the outset than later on. If the training data are corrupted by noise, this slows learning and keeps the network from moving too far from the initial weights until it has seen more of the training data.

Note that this perspective is one that you are more likely to reach if you use very large training sets and few training epochs, as Elman does.

Tower Networks

It was found that simple recurrent networks had problems when trying to pick out temporal patterns with long "wavelengths" (such as recurring themes in music).

Task: predicting the next letter in a word with a range of words (called graphotactic prediction). The network is trained on a large number of English words, considered as separate patterns. With a trained net, next-letter-prediction error on known words decreases as more and more letters are seen, because the set of predicted next letters gets smaller. With valid but novel words, error should decrease in a similar way, as the net should have learned the 'shape' of English words. In mis-spelled words, error should stay high, if the word breaks the English graphotactic rules.


---

Notes © Bill Wilson, 2009


Networks related to tower networks have also been studied by a range of authors (see p. 783 in Haykin for details) and referred to as NARX networks (Non-linear AutoRegressive with eXogenous inputs).
Problems with SRN for Next-letter Prediction

However, initial experiments with such a network failed: error remained high; little improvement occurred with more training.

If one state vector is a good thing, maybe more state vectors are a better thing.

We’ll call these things Elman tower networks. [Similarly, there could be Jordan tower networks.]

The extra state vector(s) keep save pristine copies of the hidden unit activations from further back in the sequence of inputs.

Weight matrices for Feedforward and Simple Recurrent Nets

\[
\begin{bmatrix}
  w_{\text{ih}} & w_{\text{ho}} \\
  w_{\text{hi}} & w_{\text{ho}}
\end{bmatrix}
\]

Feed forward net and its weight matrix

\[
\begin{bmatrix}
  w_{\text{ih}} & w_{\text{hs}} & w_{\text{ho}} \\
  w_{\text{hi}} & w_{\text{hs}} & w_{\text{ho}}
\end{bmatrix}
\]

Simple recurrent net and its weight matrix

How Backprop Works on Elman Tower Networks

How Backprop sees

Why Tower Networks are Logically Unnecessary - 2 states

Elman net to simulate 2-state Elman tower

\[
\begin{bmatrix}
  s_1 & s_2 & h_1 & h_2 & o \\
  0 & 0 & W_{\text{ih}1} & 0 & 0 \\
  s_1 & 0 & W_{\text{hs}1} & 1 & 0 \\
  s_2 & 0 & W_{\text{hs}2} & 0 & 0 \\
  h_1 & 0 & 0 & 0 & W_{\text{ho}} \\
  0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

Weight matrix for 2-state Elman tower. Submatrices in bold would have to be learned.
Why Tower Networks are Logically Unnecessary - $k$ states

Elman net to simulate $k$-state Elman tower

In practice, tower networks perform better than just-plain-Elman networks, as we shall see …

Comparing Performance of SRNs & Elman Tower Nets

There is a problem in comparing the performance of a regular simple recurrent network with that of an Elman tower. When do two networks with different architectures have comparable computational resources?

The computational resources of a network are the nodes, weights, and biases. More nodes means more resources, and the same for weights and biases. Memory resides in the weights and biases. Thus two networks have comparable computational resources if they have the same number of weights and biases. A network with $s$ state vectors, $h$ hidden units, $i$ input and $o$ output units has $w = ih + h + o + sh^2$ weights and biases.

For the graphotactic task, $i = 26$ and $o = 27$. It turns out that there are several ways of constructing networks with $w = 5859$. That is, there are several pairs $(s, h)$ such that $w(s, h) = 5859$:

<table>
<thead>
<tr>
<th>State Vectors</th>
<th>Hidden Units</th>
<th>Weights+Biases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54</td>
<td>5859</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>5859</td>
</tr>
<tr>
<td>6</td>
<td>27</td>
<td>5859</td>
</tr>
</tbody>
</table>

The nets with more state vectors do have more nodes $(i + h + o + sh = 161, 195, 244.)$

Performance of Elman Tower Networks

On the next page is a graph of total sum of squares error (TSS) on the training set against epochs.

Notice that for the networks with more state vectors, the learning curve descends more steeply at first (faster learning) and reaches lower levels (though the “smoke trail” effect is more marked).

The “smoke trail” effect can be eliminated by zeroing the context units between words.
Does the Same Trick Work with Jordan Nets?

A 2-state Jordan Tower

The short answer is yes. Learning performance of Jordan towers is better than that of comparable (1-state) Jordan nets. Jordan nets and towers do not perform as well as comparable Elman nets and towers on the graphotactic prediction task.

If Elman >> Jordan, is there an x such that x >> Elman?

This non-recurrent net resembles the NetTalk net, but there the task was to generate a phoneme for the letter in the middle, while here the task is to predict the next letter given 7 preceding letters. It differs from a TDNN as it does not use weight averaging.

How do InputWindow nets perform on the graphotactic prediction task?

Answer: quite well.

Moral: you don’t always need a fancy solution!
Summary of Tower Networks

- More states are better than fewer states
- InputWindow >> Elman tower >> Jordan tower if the input window captures all relevant time steps
- In both cases, improvements affect both learning speed and final total error

Try It Yourself

Sample .cf, .data, & .teach files for a simple Elman tower under tlearn are shown at:
http://www.cse.unsw.edu.au/~billw/elmantower.html

Unfolding Recurrent Networks

Here is the smallest meaningful fully recurrent network:

With synchronous activation updates, we can model the state of the network at successive time steps by saying that at each time step \( i \), there will be activation values \( u_i \) and \( v_i \) for units \( u \) and \( v \).

Unfolding Recurrent Networks 2

When we’ve done this, the weighted connections actually connect the \( u_i \) and \( v_i \) to \( u_{i+1} \) and \( v_{i+1} \). Diagrammatically, the network can thus be “unfolded” as follows:

The unfolded net is feedforward, and has identical behaviour to the original, over time steps 0 to \( t \). This depends on two facts about the unfolded network:
1) it has the same weight values as the original net; and
2) corresponding weights in different time-periods have identical weights - e.g. the weight from \( u_0 \) to \( v_1 \) has the same value \( (w_{uv}) \) as the weight from \( u_1 \) to \( v_2 \), and so on.
**Backpropagation Through Time**

If we try to train the (unfolded) net using (ordinary) error backpropagation, then the different instances of the $w_{uv}$ weight (for example) will most likely all be changed by different amounts. This can be fixed by averaging the changes: in effect, the changes to $w_{uv}$ that are required by backpropagation for each level, are averaged over all the time steps, and applied to all the $w_{uv}$ weights.

This process can be adapted to train fully or partly recurrent networks, and it is called “backpropagation-through-time.” The problem with it is the memory required - for a $k$-time-step version one requires $k$ times as many activations and weights, error derivatives and so on to be remembered.

---

**Learning to be a Shift Register, Revisited**

We shall try to train a 3-neuron fully recurrent network to simulate a shift (rotate) register. The task is to learn to take an arbitrary pattern of 1s and 0s on the three input units and rotate that pattern two bits to the right. (Or one bit to the left, if you prefer.)

Three copies of the three units are made, with the ‘lowest’ being regarded as input units and the ‘highest’ being regarded as the output units (unit 0 is the bias unit):

---

**Training Patterns for Shift Register**

The `.data` file:

```
distributed
8
0 0 0
0 0 0
0 1 0
0 0 1
0 1 0
0 1 1
1 0 0
1 1 0
1 0 1
1 1 0
1 1 1
```

The `.teach` file:

```
distributed
8
0 0 0
0 0 0
0 1 0
0 0 1
0 1 0
0 1 0
1 0 0
1 1 0
1 0 1
1 1 0
1 1 1
```

---

*Werbos, P.J., Backpropagation through time: What it is and how to do it, IEEE Proceedings 78 (1990) 1550-1560.*
Configuration File for Shift Register

The \texttt{.cf} file looks like this (\texttt{bptt.cf}):

\begin{verbatim}
NODES:
  nodes = 6
  inputs = 3
  outputs = 3
  output nodes are 4-6

CONNECTIONS:
  groups = 12
  1 from i1 = group 1
  4 from 1 = group 10
  2 from i2 = group 2
  5 from 2 = group 11
  3 from i3 = group 3
  6 from 3 = group 12
  1 from i2 = group 4
  4 from 2 = group 4
  2 from i3 = group 5
  5 from 3 = group 5

SPECIAL:
  selected = 1-6
  weight_limit = 0.1
\end{verbatim}

Test Results after 1250 epochs

\begin{verbatim}
Output activities
using bptt-10000.wts / bptt.data
\begin{tabular}{cccc}
\text{(Output)} & \text{(Target)} \\
\hline
0.053 & 0.052 & 0.052 & 0 0 0 \\
0.061 & 1.000 & 0.060 & 0 1 0 \\
1.000 & 0.060 & 0.059 & 1 0 0 \\
0.935 & 0.935 & 0.000 & 1 1 0 \\
0.061 & 0.060 & 1.000 & 0 0 1 \\
0.000 & 0.935 & 0.936 & 0 1 1 \\
0.935 & 0.000 & 0.936 & 1 0 1 \\
0.944 & 0.945 & 0.945 & 1 1 1 \\
\end{tabular}
\end{verbatim}

Error graph during training

(with one particular set of initial weights)

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{error_graph}
\caption{Error graph during training}
\end{figure}

Note the two plateaux in this error curve.

Summary

- Data representation
- Time representation
- TDNNs – Time Delay Neural Networks
- Fully and partly recurrent networks
- Jordan nets
- Elman nets (simple recurrent nets)
- The importance of starting small
- Effects of the logistic function on initial phases of backprop learning
- Tower networks
- Backpropagation through time