# Modeling sequences

Gary Cottrell

With some slides borrowed from Geoff Hinton

See also:
"The unreasonable effectiveness of recurrent networks"
(Andrej Karpathy blog):
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

# The issue

- Time is obviously important for everything.
- How do we represent time in connectionist networks?
- Two approaches:
  - Map time into space
  - Map time into the state of the network

# Mapping Time into Space

- John likes a banana in buffers:

  | John | likes | a | banana |

  X1 X2 X3 X4

  What about longer sentences?

- Banana in NETTalk:

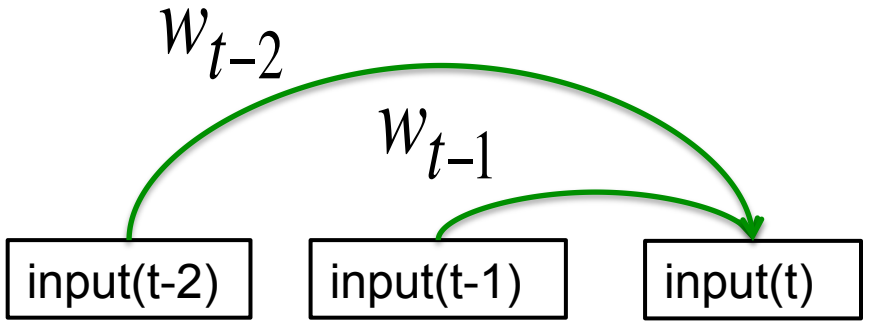
  _/1 B/2 A/3 N/4 A/5 N/6 A/7 (pronouncing “N”)

  B/1 A/2 N/3 A/4 N/5 A/6 _/7 (pronouncing “A”)

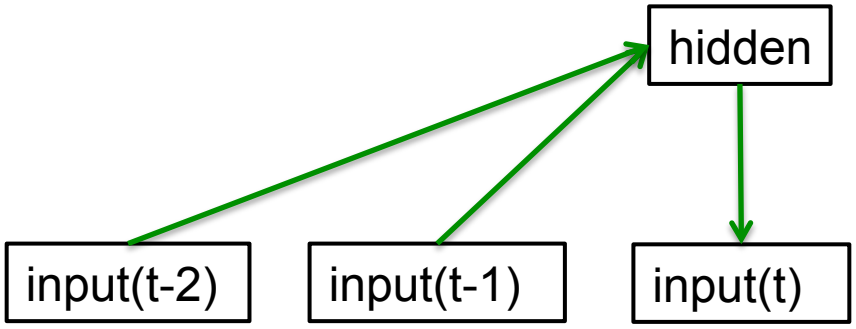
  A/1 N/2 A/3 N/4 A/5 _/6 _/7 (pronouncing the 2nd “N”)

# Mapping Time into Space

- Autoregressive models
  Predict the next term in a sequence from a fixed number of previous terms using “delay taps”.




- Feed-forward neural nets
  These generalize autoregressive models by using one or more layers of non-linear hidden units



# Mapping Time into Space

- Works for simple problems
- Doesn’t work so well for arbitrary length items - e.g., sentences.

# Mapping Time into Space

- Works for simple problems
- ~~Doesn't work so well for arbitrary length items - e.g., sentences.~~
- Learning doesn't transfer between locations in the input: there is no inherent similarity between A/1 and A/2 (but could use shared weights).

# Mapping Time into Space

- Works for simple problems
- ~~Doesn't work so well for arbitrary length items - e.g., sentences.~~
- ~~Learning doesn't transfer between locations in the input: there is no inherent similarity between A/1 and A/2~~ (but could use shared weights).
- In fact, recent work (transformer networks) does exactly this:
  - Uses identical networks over every location in the input (up to 256 locations!)
  - The networks then interact via attention mechanisms: we talk about these later...

# Clicker Question

Autoregressive models try to predict what happens next, based on what happened in a few preceding time steps (2 steps, in the diagram). Why don't we include connections from *every* preceding time step?

A) Because for every preceding time step we want to use, we need to learn some weights, which means for long sequences, we would be learning very many weights.

B) There's basically no relevant information in previous time steps. That is, what happens next has very little to do with what happened a few time steps ago.

# Clicker Question

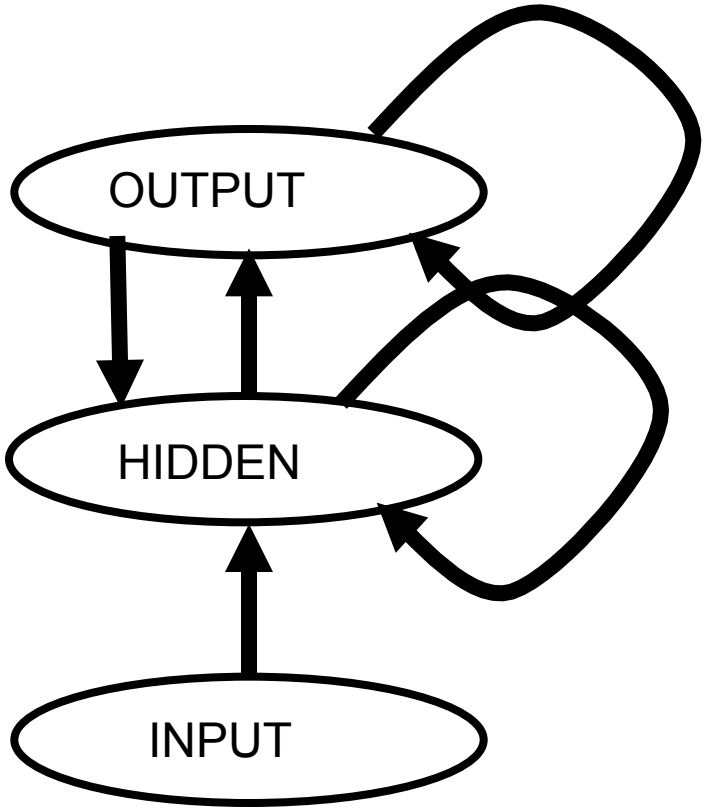
Autoregressive models try to predict what happens next, based on what happened in a few preceding time steps (2 steps, in the diagram). Why don't we include connections from *every* preceding time step?

**A) Because for every preceding time step we want to use, we need to learn some weights, which means for long sequences, we would be learning very many weights.**

B) There's basically no relevant information in previous time steps. That is, what happens next has very little to do with what happened a few time steps ago.

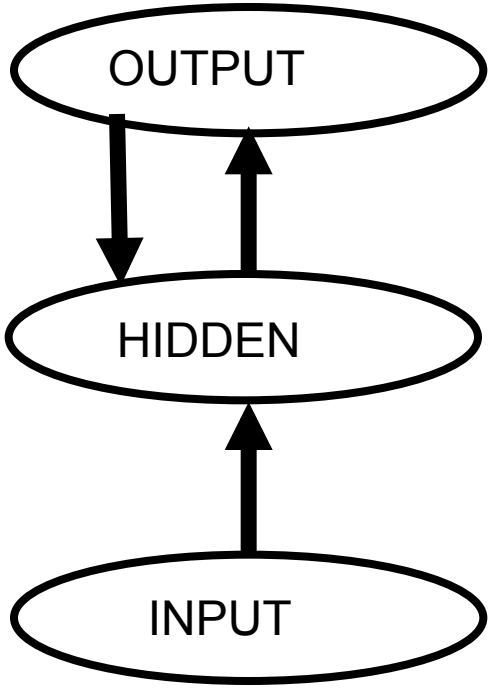
# Mapping Time into State

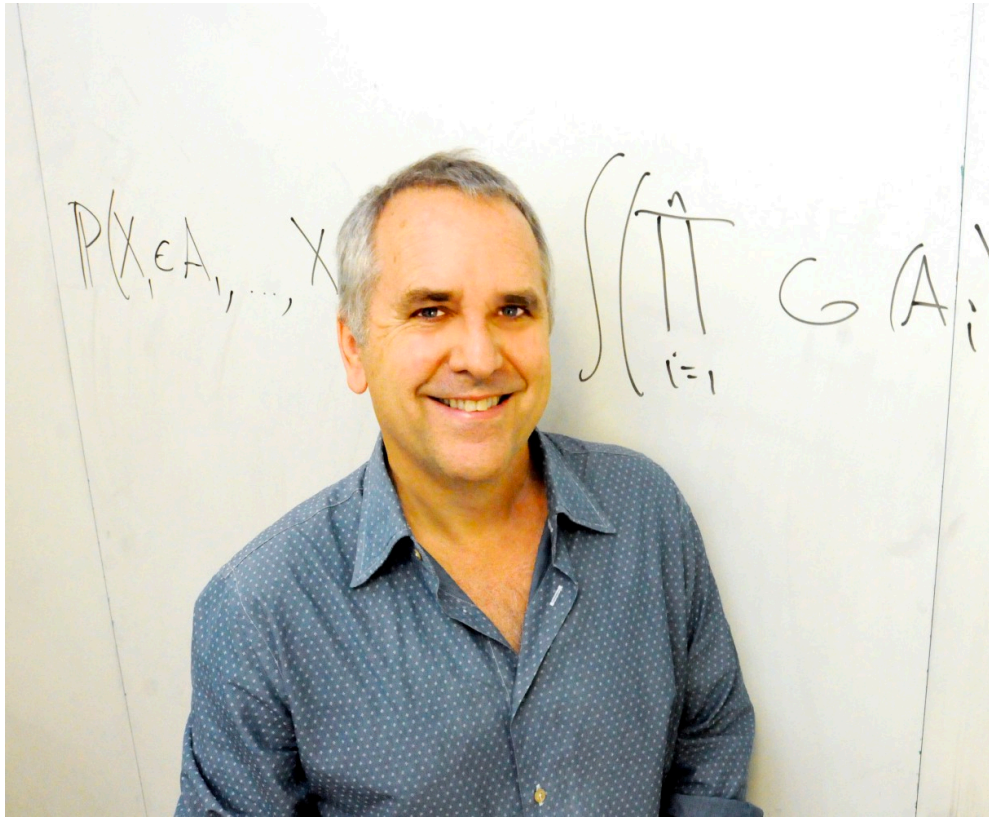
- Use some *memory* so that new things are processed based upon what has come before.



# Mapping Time into State

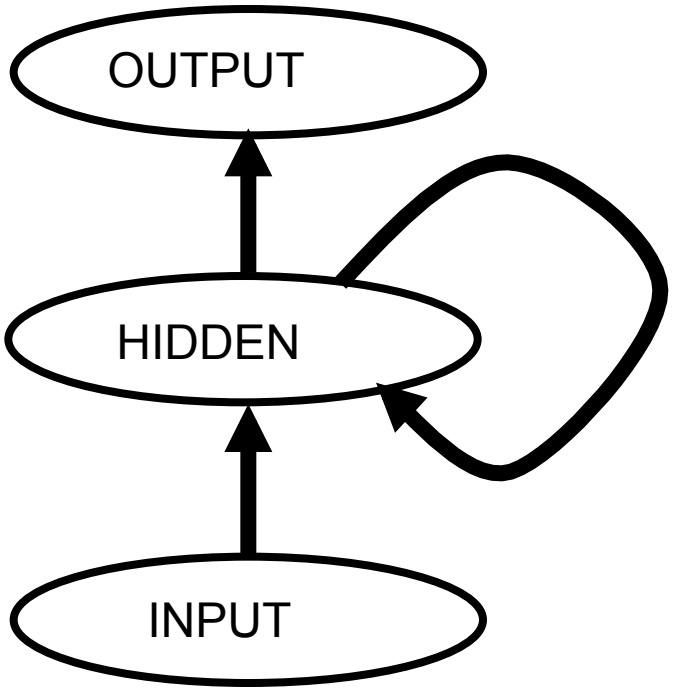
- Historically, two variants were commonly used:
- Jordan networks:



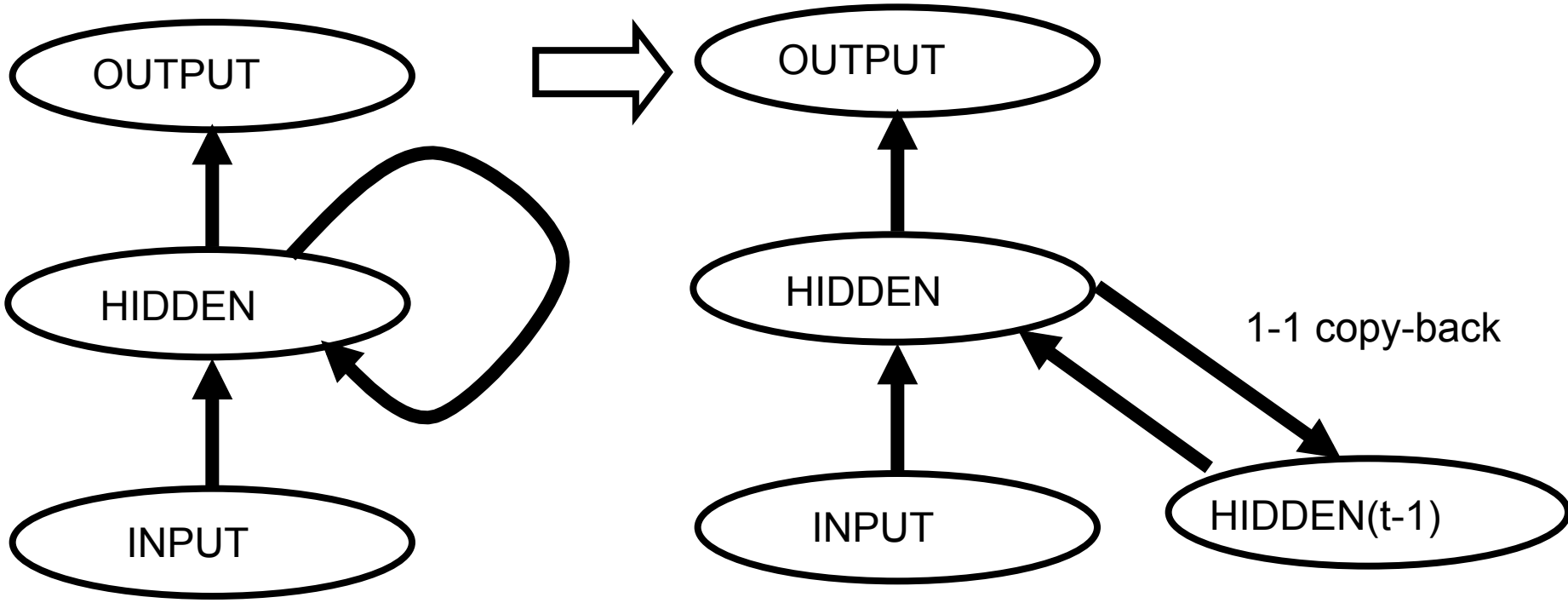
# Mapping Time into State

- Simple recurrent networks, or Elman nets:



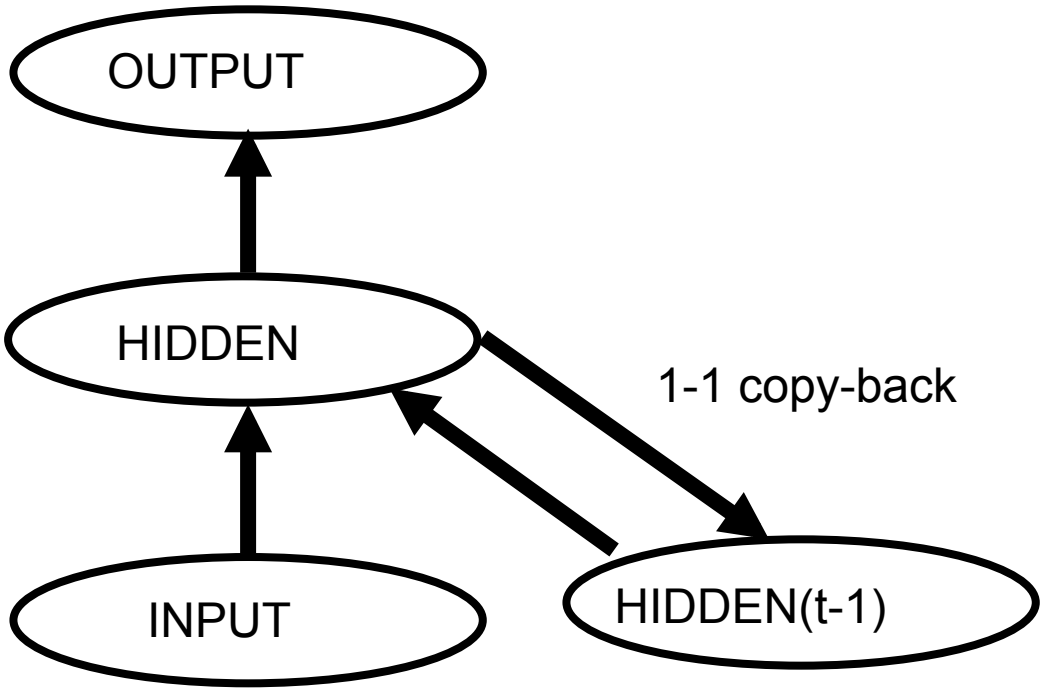

# Mapping Time into State

These were used with approximations to the true error gradient, as follows:



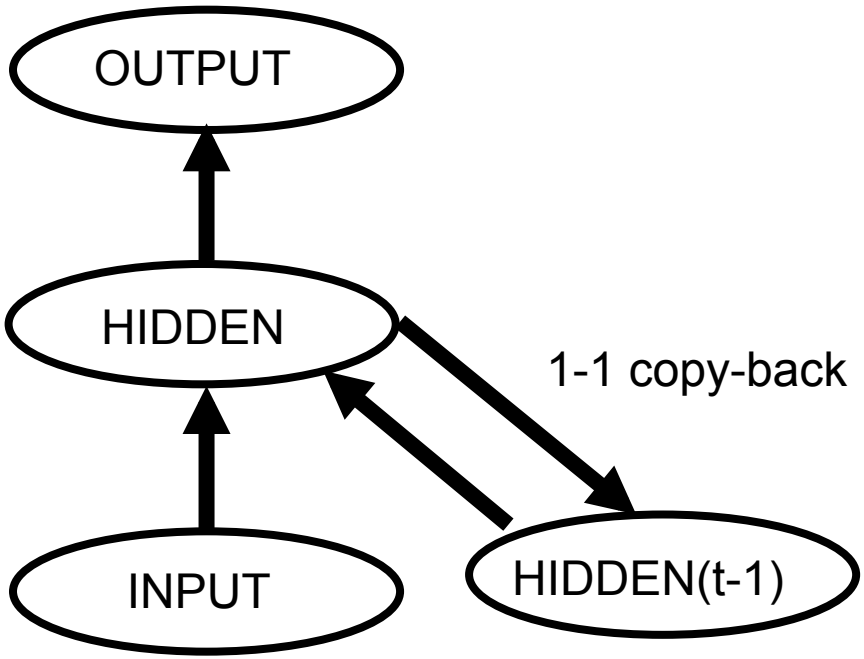
# Mapping Time into State

- This is called one-step backprop through time (BPTT(1))



# Mapping Time into State

- These networks can be used for many things, but typically, SRNs are used for *prediction*.
- This is theoretically simple, because the teacher is the input (one step later)
- A *temporal* autoencoder



# Types of problems

- Prediction: Predict the next word, “Predict” the next pixel
- Sequence generation: produce a word or a sentence, caption an image
- Sequence recognition: recognize a sentence, recognize an action
- Sequence transformation: speech->text, English->French
- Learning a “program”: sequential adder net, Neural Turing Machine
- Oscillations: walk, talk, chew, fly

# Getting targets when modeling sequences

- When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain.
  - –*E. g.* turn a sequence of sound pressures into a sequence of word identities.
- Elman's approach: *prediction,* is very useful when there is no separate target sequence.
  - –The target output sequence is the input sequence with an advance of 1 step.
  - –This approach can be generalized to trying to predict one pixel in an image from the other pixels, or one patch of an image from the rest of the image.
  - –For temporal sequences there is a natural order for the predictions
  - –Less clear for the pixel idea...
- Like autoencoding, predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning.
  - –It uses methods designed for supervised learning, but it doesn't require a separate teaching signal.

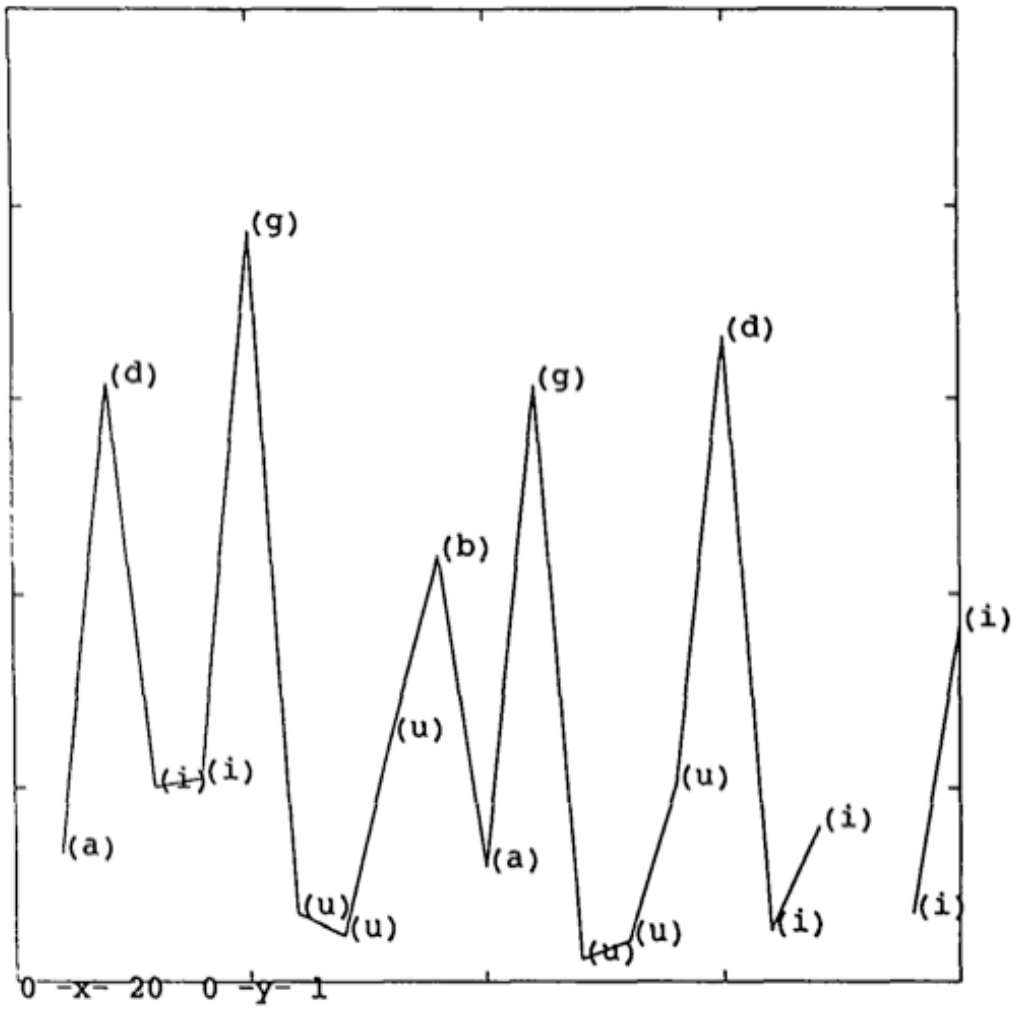
# Finding Structure in Time (Elman, 1990)

- Used predicting the next element exclusively

- First simulation: simulate “word learning”
- Given a very simple regular grammar:
  - A->ba
  - B->dii
  - C->guuu
- Randomly generate a sequence of these elements and ask the network to predict the next letter.

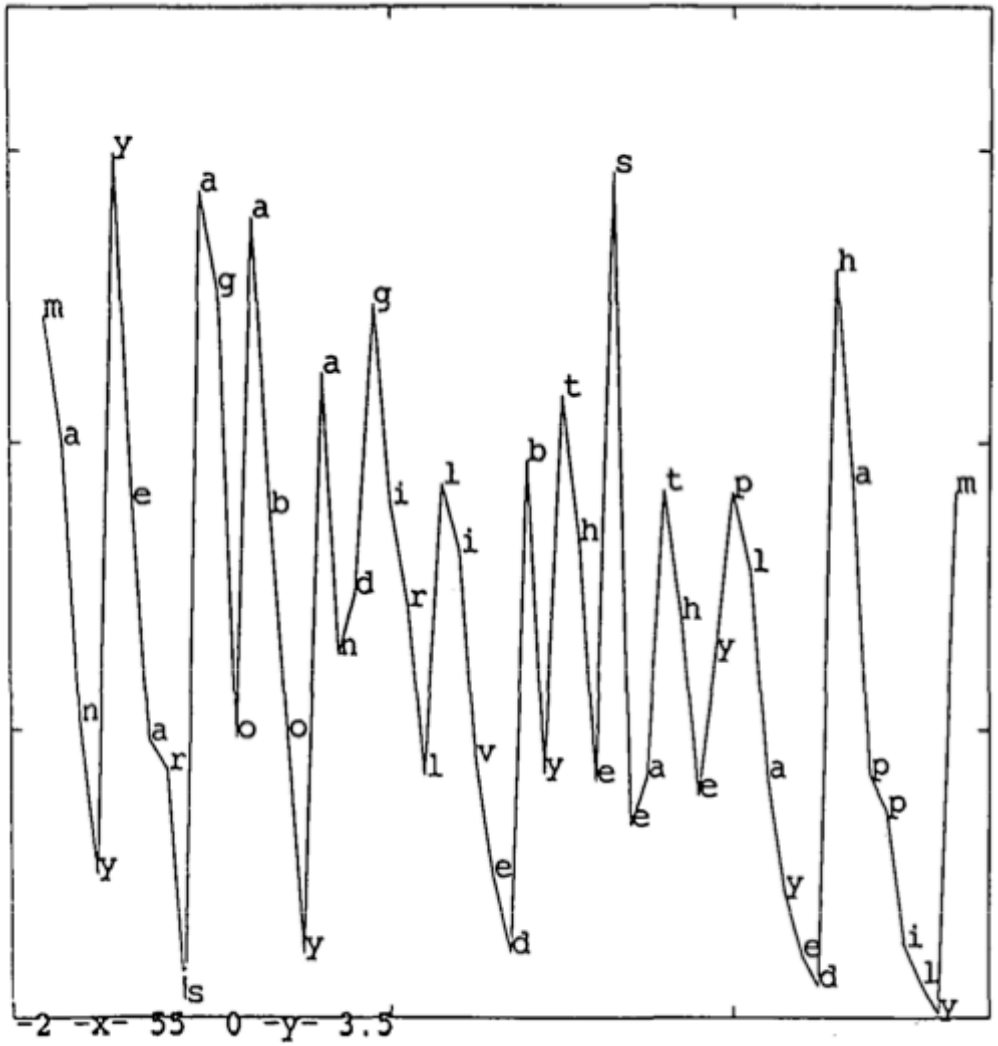
# Finding Structure in Time

- Results:



# Finding Structure in Time

- Results on a task with 15 words
- The network has discovered word boundaries…



# Finding Structure in Time: "sentences"

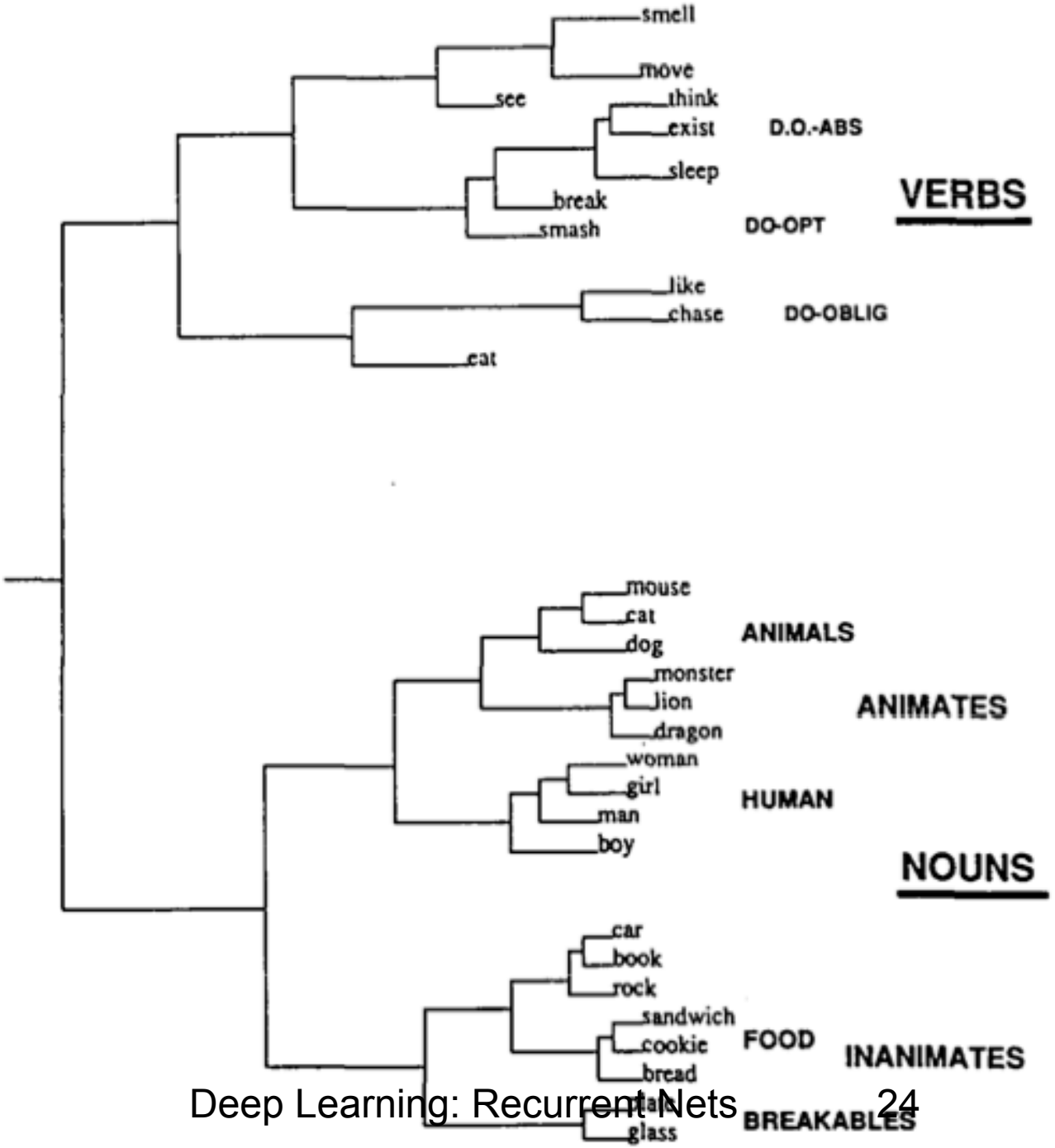
- Trained the network on a bunch of 2 and 3 word "sentences" involving 29 words:
  - Boy eat sandwich
  - Monster eat boy
  - Girl exists
  - Boy smash plate
  - Monster ate cookie
  - Boy chase girl
- Crucially, these words were represented in a localist fashion: there was *no* similarity structure that the network could use.

# Finding Structure in Time: “sentences”

- Obviously, the network cannot predict the next word - but it can predict what *type* of word may follow:
- So, “boy eat” must be followed by sandwich cookie, or bread: these three units would be equally excited.
- Monster eat would include people and animals as well.

# Finding Structure in Time: “sentences”

- Analysis:
  - Collect the 150 hidden unit representation of every word in every sentence
  - Average them together
  - Perform hierarchical cluster analysis
  - This will show how the network has learned to group items together.


smell
move
see
think
exist
sleep
break
smash
D.O.-ABS
DO-OPT
VERBS
like
chase
cat
DO-OBLIG
mouse
cat
dog
monster
lion
dragon
ANIMALS
ANIMATES
woman
girl
man
boy
HUMAN
NOUNS
car
book
rock
sandwich
cookie
bread
FOOD
INANIMATES
glass
BREAKABLES

# This network demonstrated:

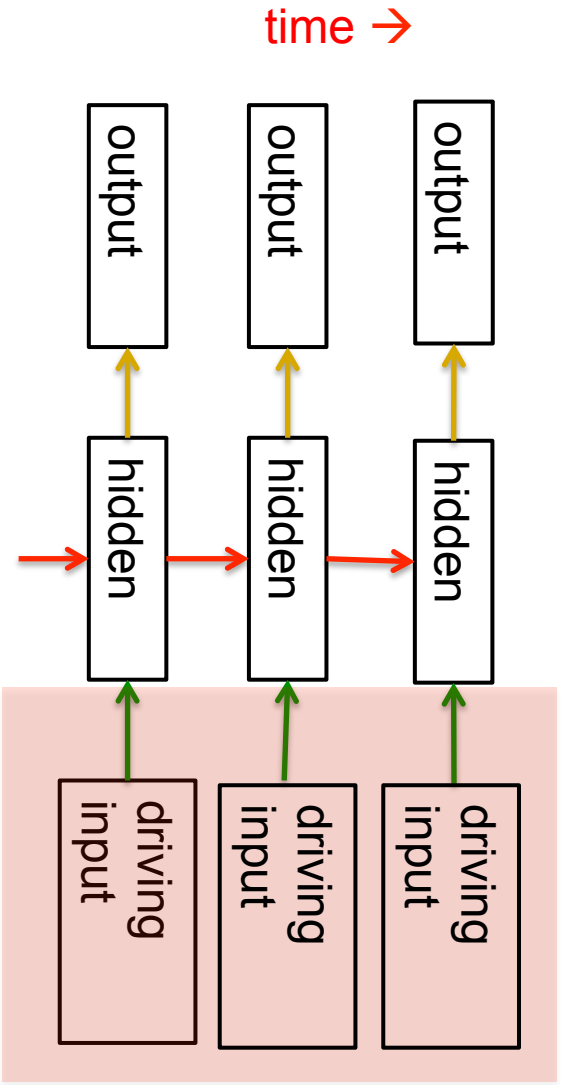
- Sensitivity to temporal context
- Ability to learn which words should be similar based on their predictive properties
- Learned semantics by “listening to the radio”
- ***BUT***, this model is “computationally naïve,” in that it uses a truncated gradient. This made sense at the time, when computer power was much less.

# Beyond memoryless models

- If we give our generative model some hidden state, and if we give this hidden state its own internal dynamics, we get a much more interesting kind of model.
  - It can store information in its hidden state for a long time.
  - If the dynamics is noisy and the way it generates outputs from its hidden state is noisy, we can never know its exact hidden state.
  - The best we can do is to infer a probability distribution over the space of hidden state vectors.
- This inference is only tractable for two types of hidden state model.
  - The next three slides are mainly intended for people who already know about these two types of hidden state model. They show how RNNs differ.
  - Do not worry if you cannot follow the details. (This course does not cover HMMs or Linear Dynamical Systems)

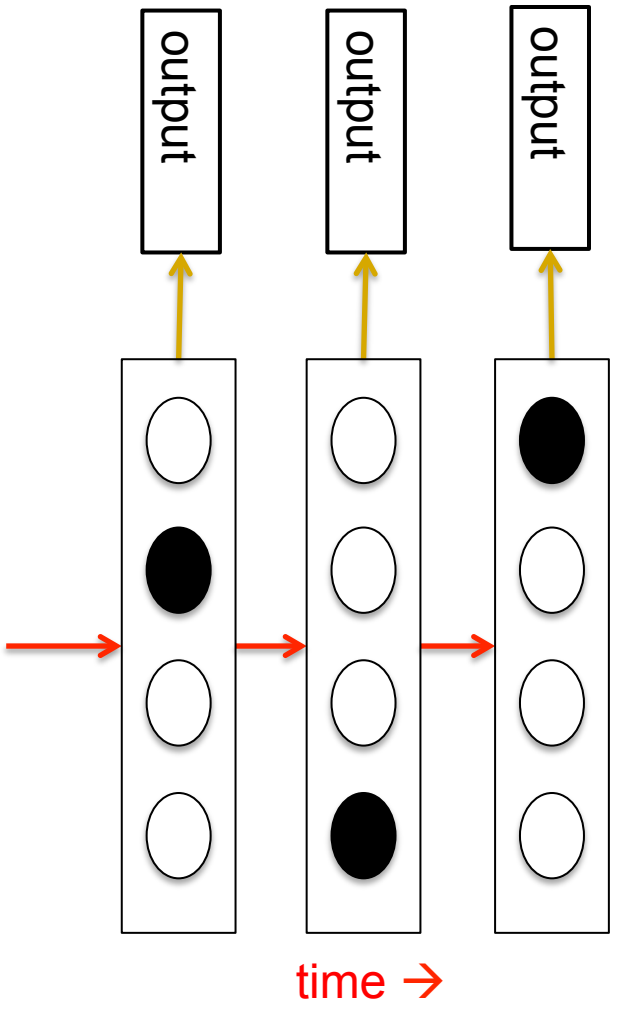
# Linear Dynamical Systems (engineers love them!)

- These are generative models. They have a real-valued hidden state that cannot be observed directly.
  - The hidden state has linear dynamics with Gaussian noise and produces the observations using a linear model with Gaussian noise.
  - There may also be driving inputs.
- To predict the next output (so that we can shoot down the missile) we need to infer the hidden state.
  - A linearly transformed Gaussian is a Gaussian. So the distribution over the hidden state given the data so far is Gaussian. It can be computed using "Kalman filtering".



# Hidden Markov Models (computer scientists love them!)

- Hidden Markov Models have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic.
  - We cannot be sure which state produced a given output. So the state is "hidden".
  - It is easy to represent a probability distribution across N states with N numbers.
- To predict the next output we need to infer the probability distribution over hidden states.
  - HMMs have efficient algorithms for inference and learning.

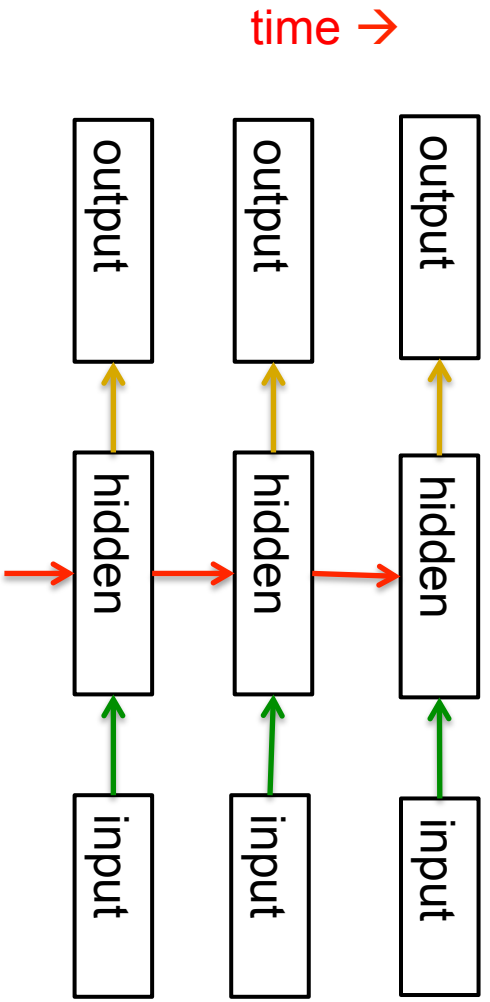


# A fundamental limitation of HMMs

- Consider what happens when a hidden Markov model generates data.
  - At each time step it must select one of its hidden states. So with N hidden states it can only remember log(N) bits about what it generated so far.
- Consider the information that the first half of an utterance contains about the second half:
  - The syntax needs to fit (e.g. number and tense agreement).
  - The semantics needs to fit. The intonation needs to fit.
  - The accent, rate, volume, and vocal tract characteristics must all fit.
- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half. 2^100 is big!

# Recurrent neural networks

- RNNs are very powerful, because they combine two properties:
  1. Distributed hidden state that allows them to store a lot of information about the past efficiently.
  2. Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.
  - RNNs are Turing Complete (Siegelmann and Sontag, 1995)



# Do generative models need to be stochastic?

- Linear dynamical systems and HMMs are **stochastic models.**
  - But the posterior probability distribution over their hidden states given the observed data so far is a deterministic function of the data.
- Recurrent neural networks are **deterministic**.
  - So think of the hidden state of an RNN as the equivalent of the deterministic probability distribution over hidden states in a linear dynamical system or a HMM

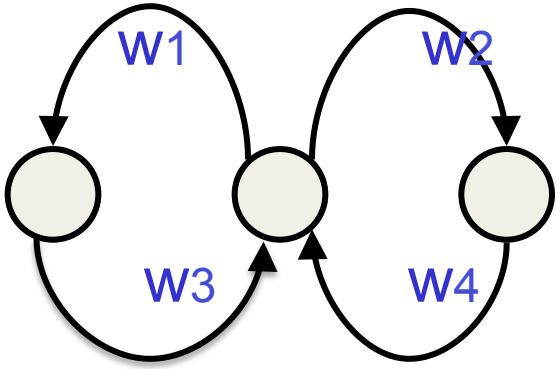
# Recurrent neural networks

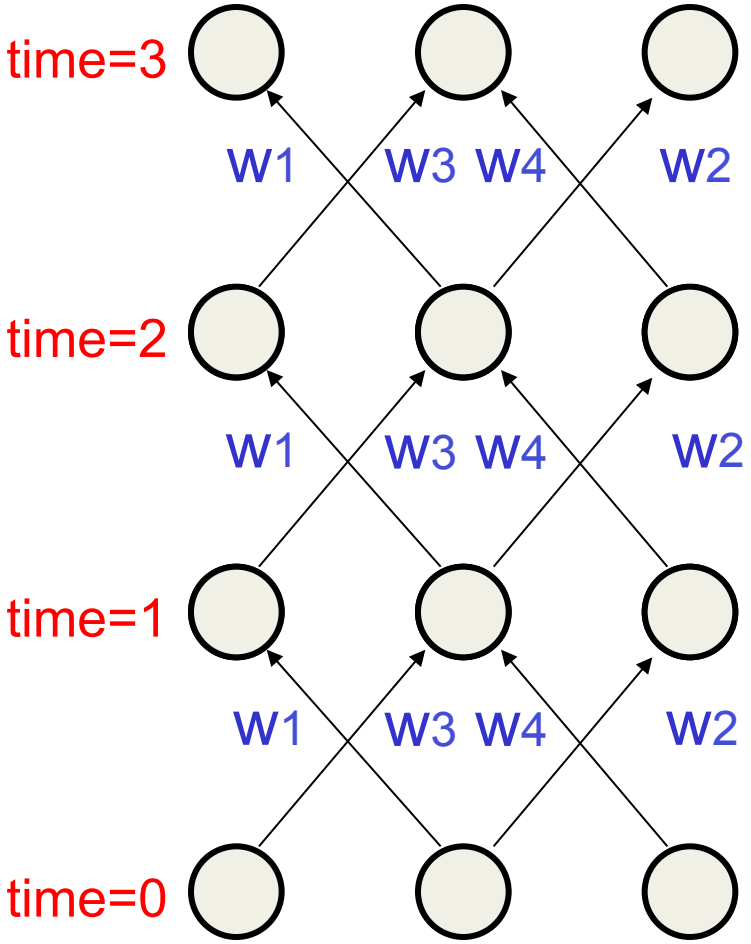
- What kinds of behavior can RNNs exhibit?
  - They can oscillate. Good for motor control
  - They can settle to point attractors. Good for retrieving memories
  - They can behave systematically: Good for learning transformations
  - They can behave chaotically. Bad for information processing?
  - RNNs could potentially learn to implement lots of small programs that each capture a nugget of knowledge and run in parallel, interacting to produce very complicated effects.

# Recurrent neural networks

- But the computational power of RNNs makes them very hard to train.
  - –For many years we could not exploit the computational power of RNNs despite some heroic efforts (e.g. Tony Robinson's speech recognizer).
  - –This was why Elman and others (e.g., Mike Jordan) used truncated gradients.

# Unrolling recurrent networks
## "A recurrent network is just a special case of a feedforward network" – Dave Rumelhart




Assume that there is a time delay of 1 in using each connection.

The recurrent net is just a layered net that keeps reusing the **same weights**.

# Backpropagation with weight constraints

- It is easy to modify the backprop algorithm to **incorporate linear constraints** between the weights.

1. Start with weights equal
2. Compute the gradients as usual
3. Calculate the total gradient as a sum or average and then modify the gradients so that they satisfy the constraints.

$$To\ \ constrain: \quad w_1 = w_2$$

$$we\ \ need: \quad \Delta w_1 = \Delta w_2$$

$$compute: \quad \frac{\partial E}{\partial w_1} \quad and \quad \frac{\partial E}{\partial w_2}$$

$$use \quad \frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2} \quad for\ w_1\ and\ w_2$$

– So if the weights started off satisfying the constraints, they will continue to satisfy them.

# Backpropagation Through Time (BPTT)

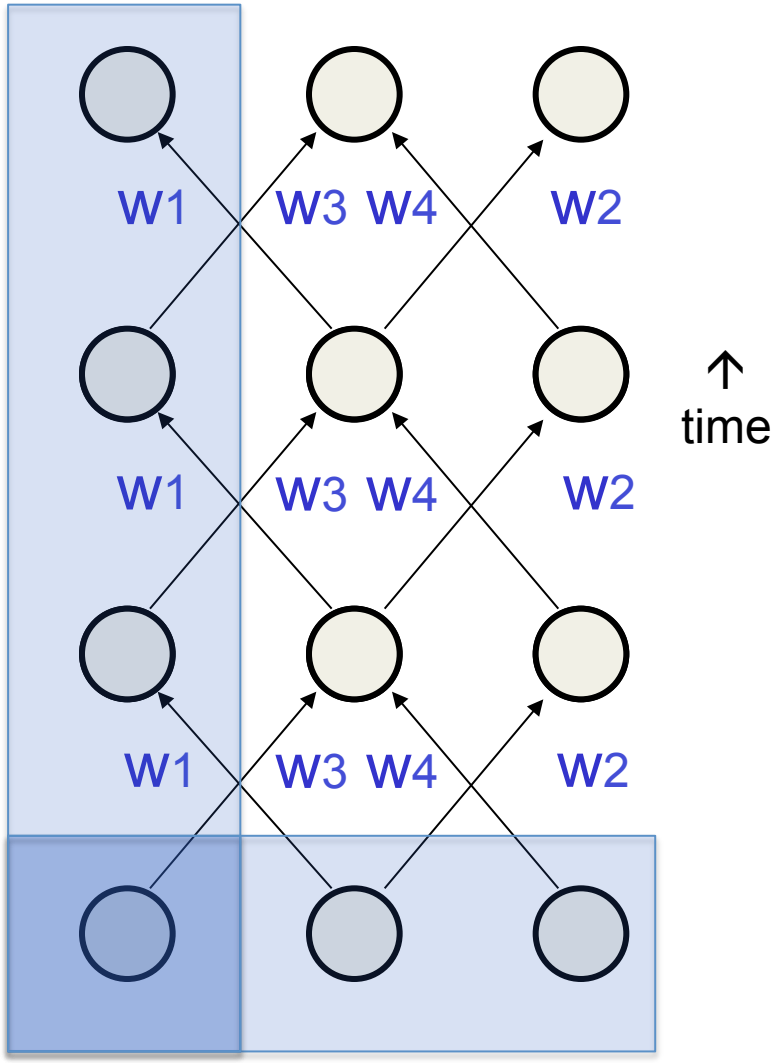
- We can think of the recurrent net as a layered, feed-forward net with shared weights and then train the feed-forward net with weight constraints.
- We can also think of this training algorithm in the **time domain**:
  1. The forward pass builds up a stack of the activities of all the units at each time step.
  2. The backward pass peels activities off the stack to compute the error derivatives at each time step.
  3. After the backward pass we add together the derivatives at all the different times for each weight.
  4. Now change all the copies of the weight by that cumulative amount.

# Initialization of Recurrent Networks

- We need to specify the **initial activity state** of all the hidden and output units.
- We could just fix these initial states to have some default value like 0.5.
- But it is better to treat the initial states as **learned parameters**.
- We learn them in the same way as we learn the weights.
  - Start off with an initial random guess for the initial states.
  - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state.
  - Adjust the initial states by following the negative gradient.

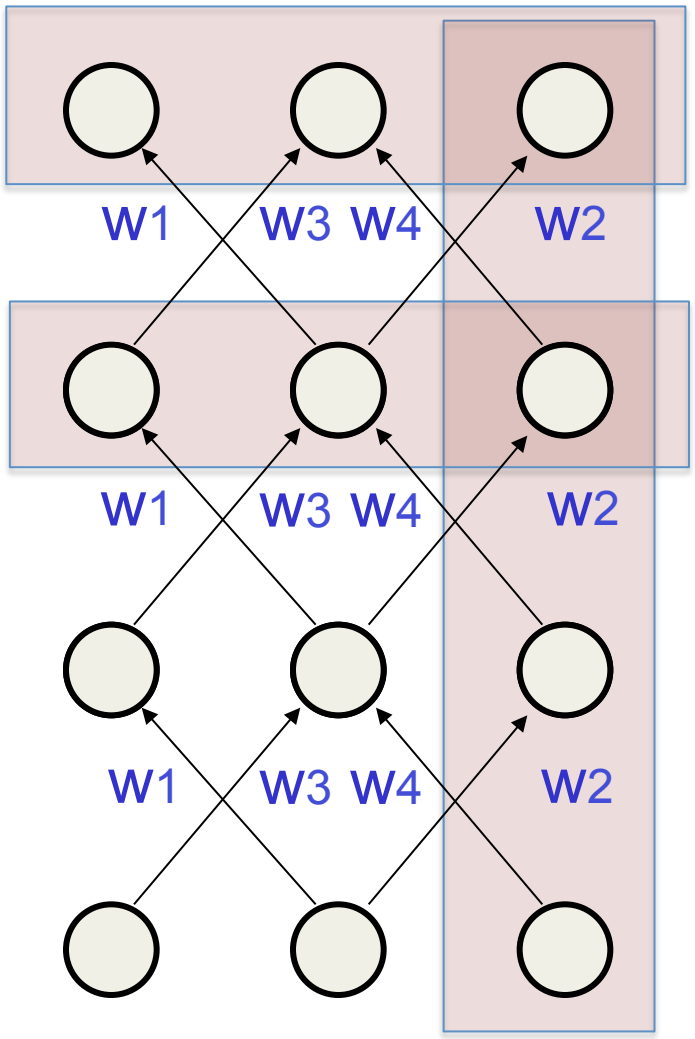
# Providing input to recurrent networks

- We can specify inputs in several ways:
  - Specify the initial states of all the units.
  - Specify the initial states of a subset of the units.
  - Specify the states of the same subset of the units at every time step.
    - This is the natural way to model most sequential data:
    - We specify each value as the next element in the sequence.



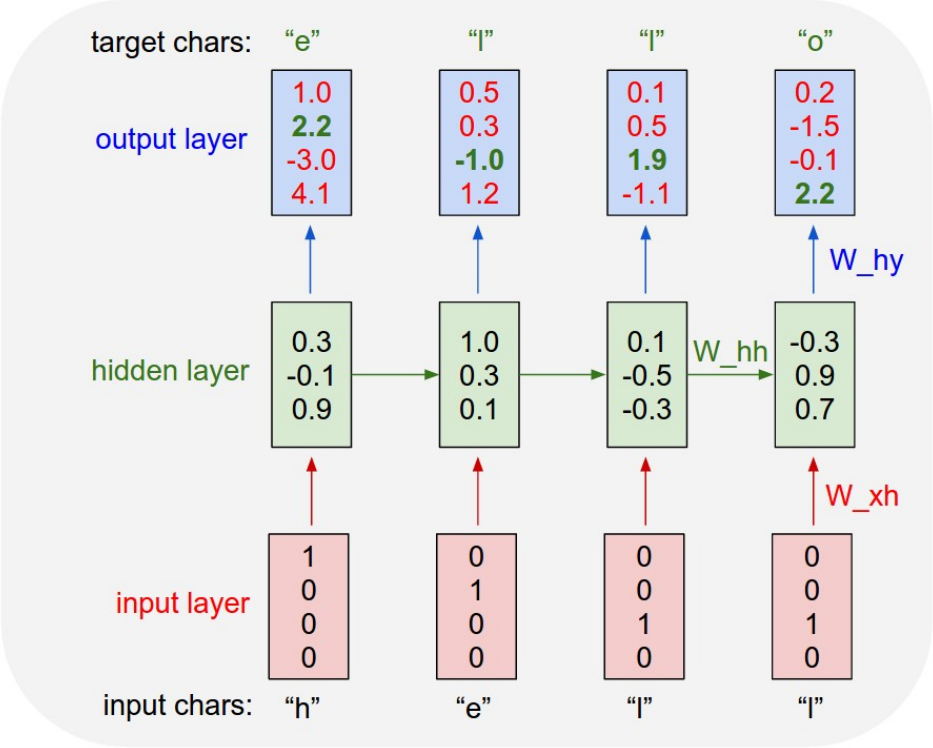
# Teaching signals for recurrent networks

- We can specify targets in several ways:
  - Specify desired final activities of all the units
  - Specify desired activities of all units for the last few steps
    - Good for learning attractors
    - It is easy to add in extra error derivatives as we backpropagate.
  - Specify the desired activity of a subset of the units.
    - The other units are input or hidden units.



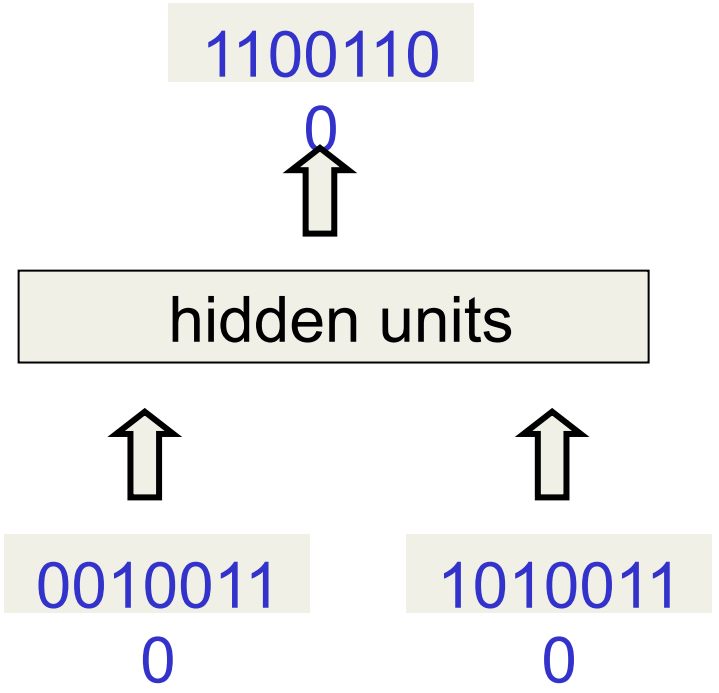
# Recurrent Network for Language Generation

- Commonly a *character-level* language model
- Start with a huge corpus of text and then ask it to model the probability distribution of the next character given the prior sequence of characters
- Simple example shown where our dictionary is only {‘h’, ‘e’, ‘l’, ‘o’}
- Note we have shared weights



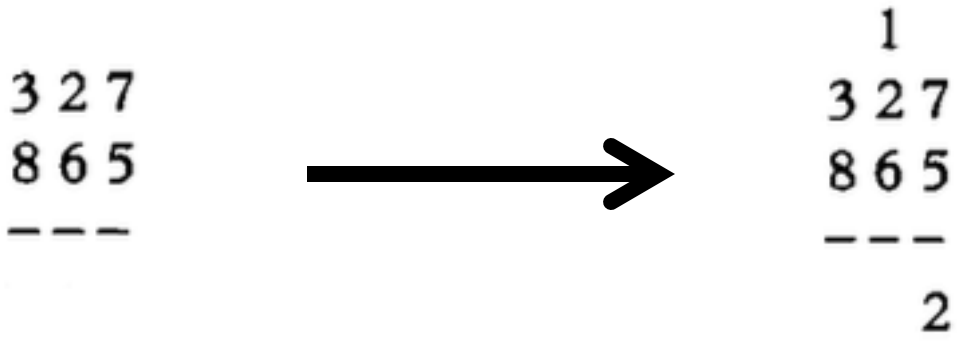
# A good toy problem for a recurrent network

- We can train a feedforward net to do binary addition, but there are obvious regularities that it cannot capture efficiently.
  - We must decide in advance the **maximum number of digits in each number.**
  - The **processing** applied to the beginning of a long number **does not generalize** to the end of the long number because it uses different weights.
- As a result, feedforward nets do not generalize well on the binary addition task.



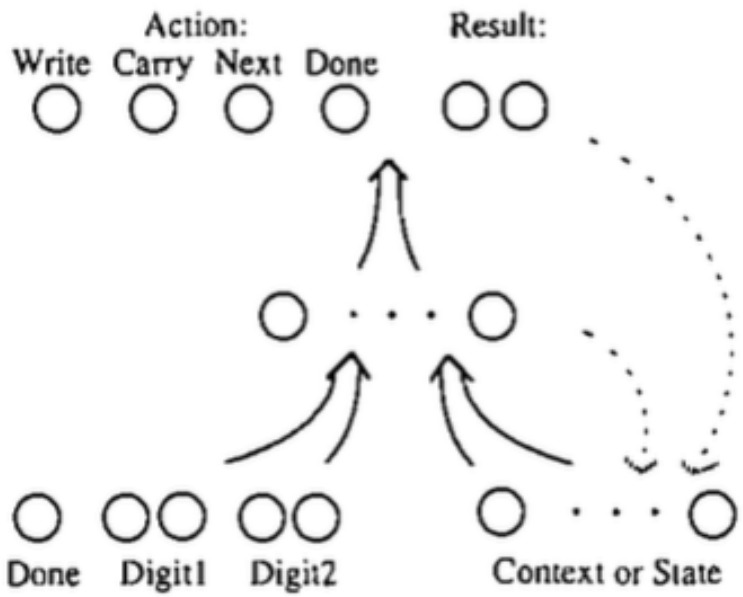
# Recurrent network for addition

- Instead, we could give a recurrent network a column at a time (Tsung & Cottrell, 1993)



# Recurrent network for addition

- Here, we are explicitly teaching a neural net to implement a program:

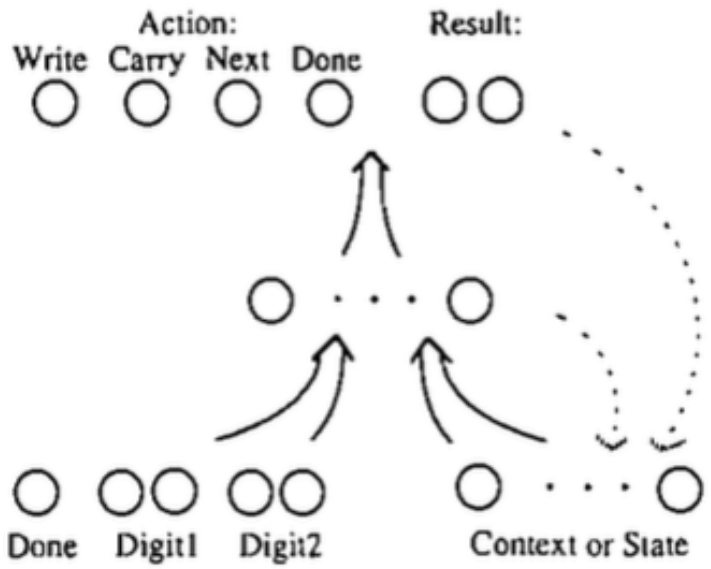



PROGRAM 1

```
while not done do
begin
  output(WRITE, low_order_digit);
  if sum > radix then
    output(CARRY, ???);
  output(NEXT, ???);
end
if carry_on_previous_input then
      output(WRITE, '01');
output(DONE, ???);
```

# Recurrent network for addition

Example of the training sequence:



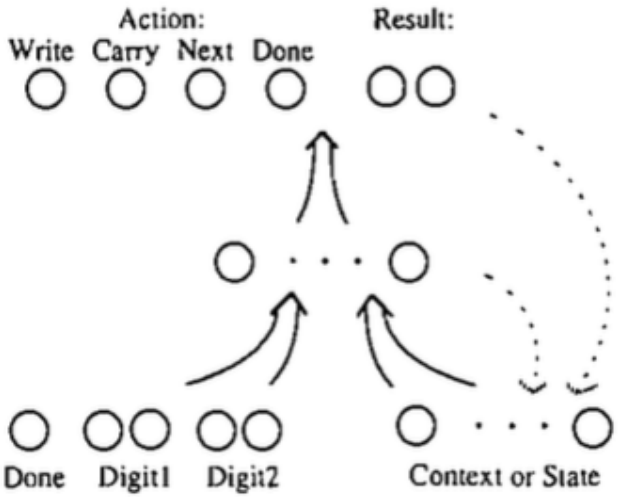

| Input | | | Output | |
|---|---|---|---|---|
| | | | Action | Result |
| 3 | 2 | ⑦ | 1. Write | 8 |
| 4 | 8 | ① | 2. Next | — |
| 3 | ② | 7 | 3. Write | 0 |
| 4 | ⑧ | 1 | 4. Carry | — |
| | | | 5. Next | — |
| ③ | 2 | 7 | 6. Write | 8 |
| ④ | 8 | 1 | 7. Next | — |
| 3 | 2 | 7 | 8. Done | — |
| 4 | 8 | 1 | | |

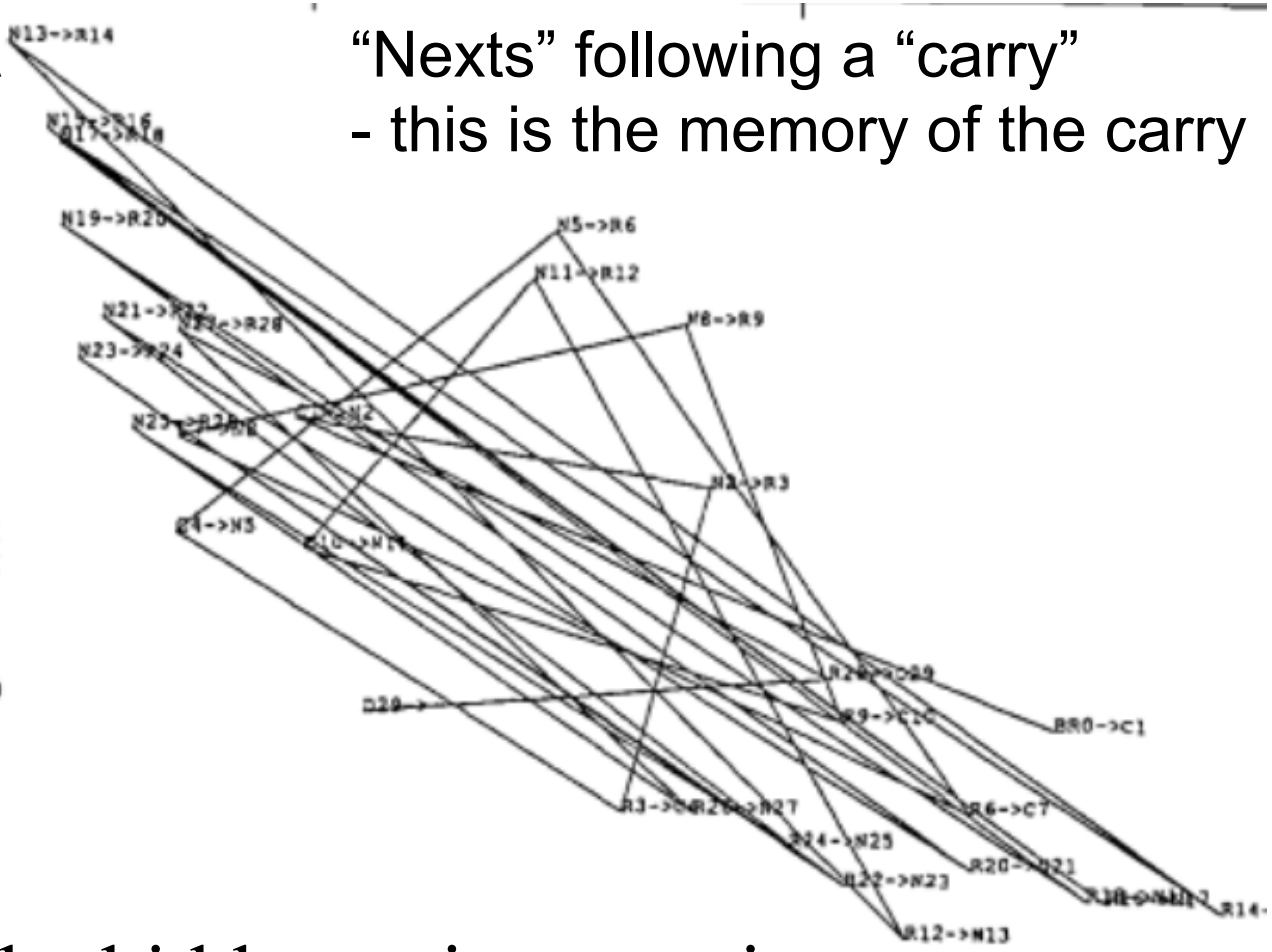
This network generalized to longer examples than it was trained on

# Recurrent network for addition: The internal state space

"Nexts" following a "write result"

"Nexts" following a "carry" - this is the memory of the carry



PCA of the hidden units over time

# Combined subset training

- We also developed a kind of curriculum training:
  - Train on a subset of the training set until it was "reasonably good"
  - Double the training set size
  - Repeat
  - The jump in the error when doubling the training set size got smaller and smaller...

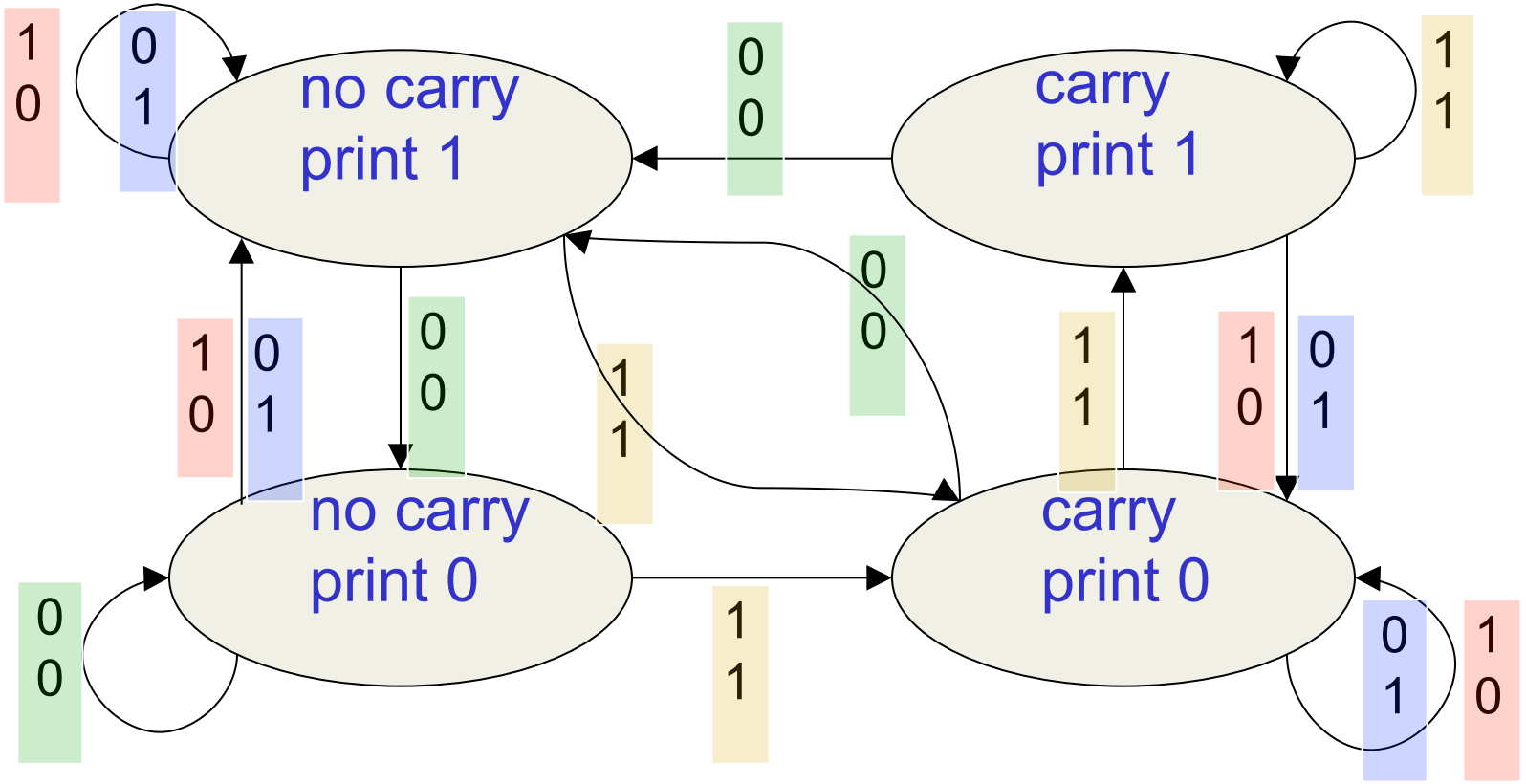
# Recurrent network for addition

- We also showed a difference between Elman & Jordan nets by just changing the order of two lines of the program:
  - Instead of saying there's a carry and then asking for the next input...
  - Ask for the next input, and *then* say there's a carry.
  - Jordan networks couldn't do this: They can't remember anything that isn't represented in their output.

- Elman nets could learn this, but it took 5,000 more epochs to remember *one* bit!

- This suggests that *learning to remember* is a bad idea...or at least, without some sort of memory structure

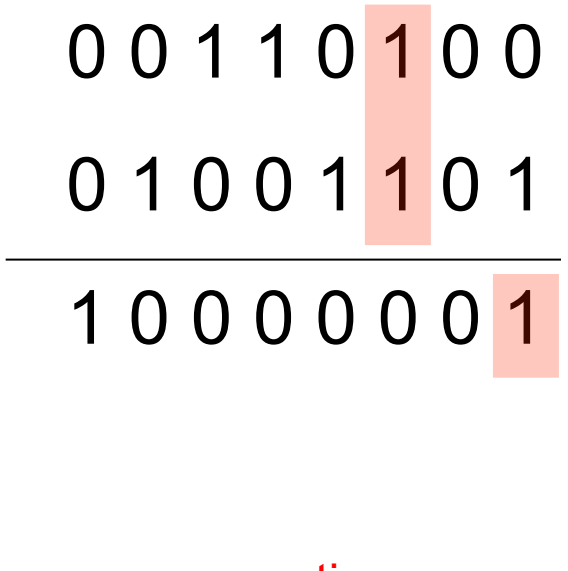
# The algorithm for binary addition




This is a **finite state automaton**. It decides what transition to make by looking at the next column. It prints after making the transition. It moves from right to left over the two input numbers.

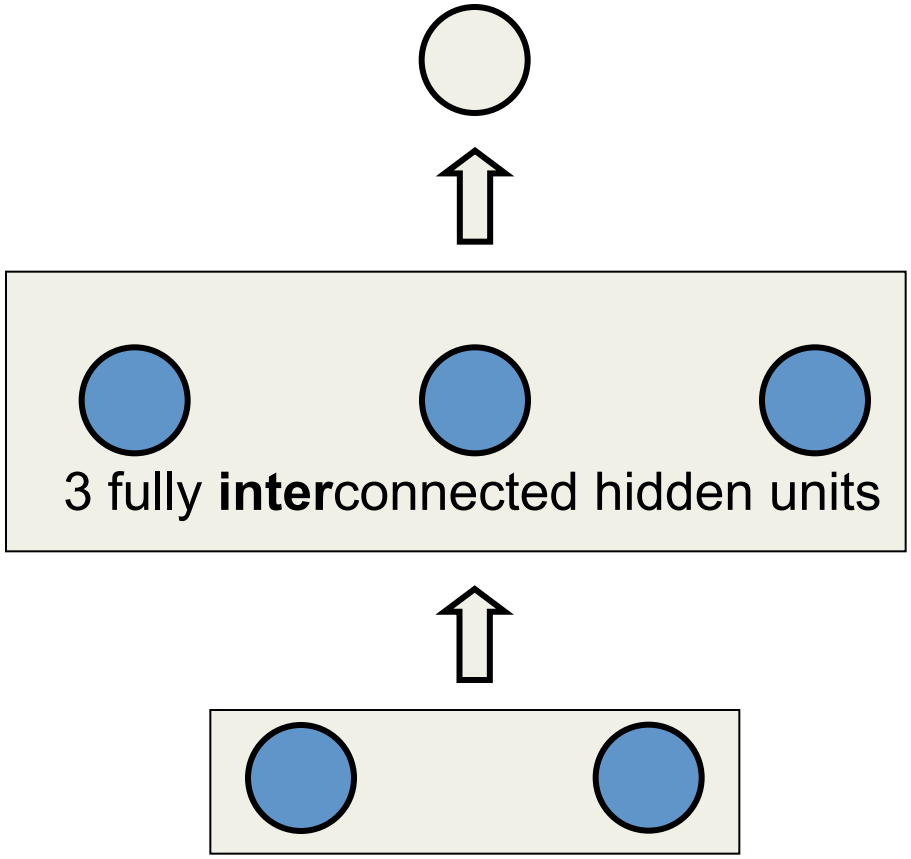
# A recurrent net for binary addition

- The network has two input units and one output unit.
- It is given two input digits at each time step.
- The desired output at each time step is the output for the column that was provided as input two time steps ago.
  - It takes one time step to update the hidden units based on the two input digits.
  - It takes another time step for the hidden units to cause the output.



# The connectivity of the network

- The 3 hidden units are fully interconnected in both directions.
  - This allows a hidden activity pattern at one time step to vote for the hidden activity pattern at the next time step.
- The input units have feedforward connections that allow then to vote for the next hidden activity pattern.



# What the RNN Learns

- It learns four distinct patterns of activity for the 3 hidden units. These patterns correspond to the nodes in the finite state automaton.
  - Do not confuse units in a neural network with nodes in a finite state automaton.
  - **Nodes are like activity vectors.**
  - The automaton is restricted to be in exactly one state at each time. The hidden units are restricted to have exactly one activity vector of activity at each time.
- A recurrent network can emulate a finite state automaton, but it is **exponentially more powerful**. With N hidden neurons it has 2^N possible binary activity vectors (but only N^2 weights)
  - This is important when the input stream has two separate things going on at once.
  - A finite state automaton needs to square its number of states.
  - An RNN needs to double its number of units.

# Clicker Question!

1. A recurrent network is a special case of a feedforward network where

A. The “forward” part corresponds to time
B. The “backwards” part corresponds to error being passed into the future
C. The connections between units are replicated for each time step
D. A&C
E. B&C

# Clicker Question!

1. A recurrent network is a special case of a feedforward network where

A. The “forward” part corresponds to time

B. The “backwards” part corresponds to error being passed into the future

C. The connections between units are replicated for each time step

**D. A&C**

E. B&C

# Clicker Question!

2. Time is represented in a *recurrent network* by

A. Space – there are multiple banks of inputs for each time step

B. State – there are different activation states as the input is processed over time

C. A variable *t* that corresponds to time

D. The weights

# Clicker Question!

2. Time is represented in a *recurrent network* by

A. Space – there are multiple banks of inputs for each time step

**B. State – there are different activation states as the input is processed over time**

C. A variable *t* that corresponds to time

D. The weights

# Clicker Question!

3. Training a recurrent network uses

A. Back propagation from the future to the past
B. Back propagation from the past to the future
C. Back propagation needs to be adjusted to work in this context.
D. Back propagation is just the same, but weight changes need to be averaged over time
E. A&D

# Clicker Question!

3. Training a recurrent network uses

A. Back propagation from the future to the past
B. Back propagation from the past to the future
C. Back propagation needs to be adjusted to work in this context.
D. Back propagation is just the same, but weight changes need to be averaged over time
**E. A&D**

# Clicker Question

4. Since a recurrent net can be viewed as a feedforward net unrolled over time, what issues with gradient propagation could arise if we unroll it for say, 100 states?

A. There are no issues with gradient propagation even if we unroll it for 500 states
B. Gradients may explode as we perform BPTT into the earlier layers
C. Gradients may vanish as we perform BPTT into the earlier layers
D. Both (b) and (c)

# Clicker Question

4. Since a recurrent net can be viewed as a feedforward net unrolled over time, what issues with gradient propagation could arise if we unroll it for say, 100 states?

A. There are no issues with gradient propagation even if we unroll it for 500 states

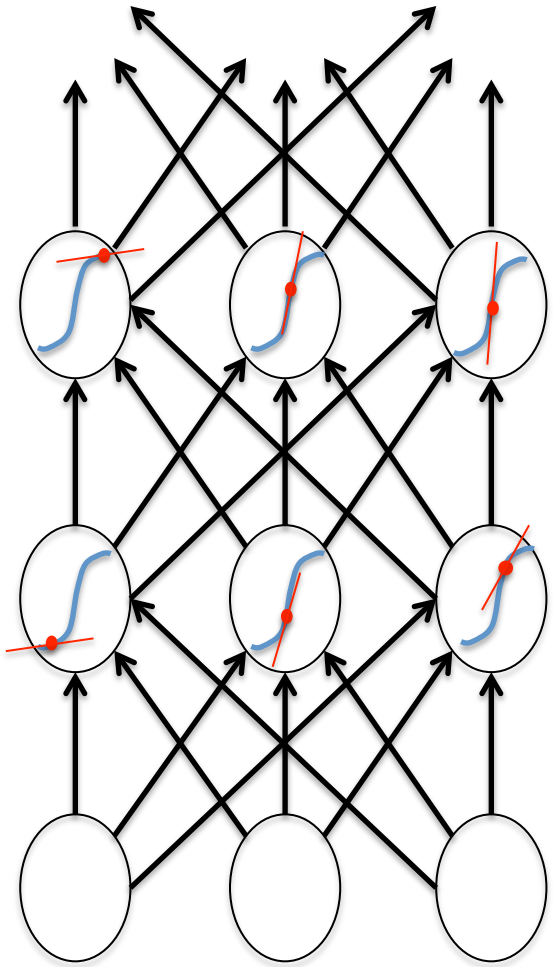
B. Gradients may explode as we perform BPTT into the earlier layers

C. Gradients may vanish as we perform BPTT into the earlier layers

**D. Both (b) and (c)**

# The backward pass is linear

- There is a big difference between the forward and backward passes.
- In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding.
- The backward pass, is completely linear. If you double the error derivatives at the final layer, all the error derivatives will double.
  - The forward pass determines the slope (used to multiply the deltas) of the linear function used for backpropagating through each neuron.



# The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  - If the weights are small, the gradients shrink exponentially.
  - If the weights are big the gradients grow exponentially.
- Shallow feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.

# The problem of exploding or vanishing gradients

- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
  - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.
  - So RNNs have difficulty dealing with long-range dependencies.

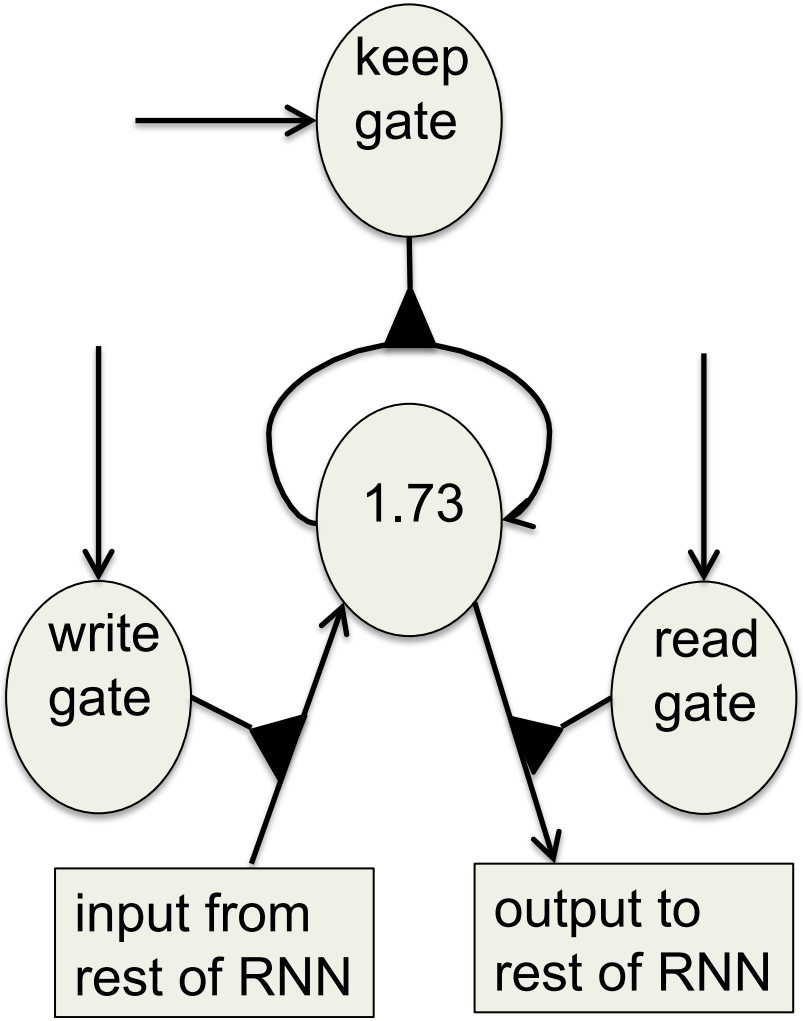
# Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.
- Information gets into the cell whenever its “write” gate is on.
- The information stays in the cell so long as its “keep” gate is on.
- Information can be read from the cell by turning on its “read” gate.

# Implementing a memory cell in a neural network

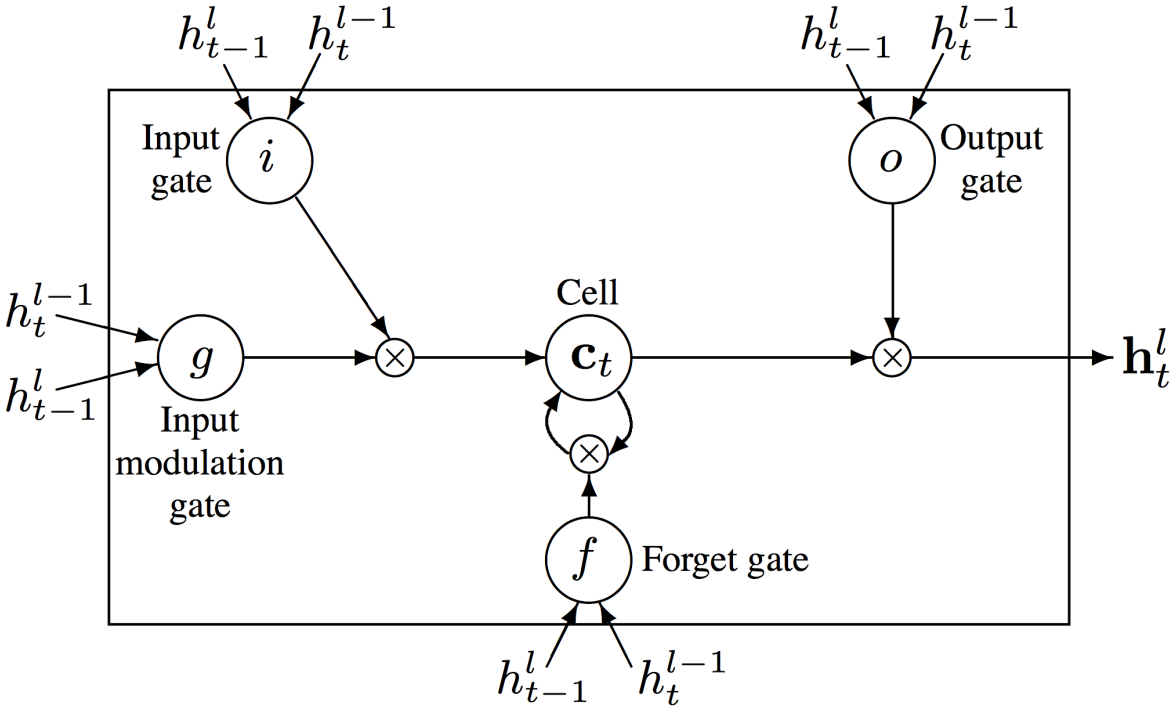
To preserve information for a long time in the activities of an RNN, we use a circuit that implements an analog memory cell.

- A linear unit that has a self-link with a weight of 1 will maintain its state.
- Information is stored in the cell by activating its write gate.
- Information is retrieved by activating the read gate.
- We can backpropagate through this circuit because logistic units have nice derivatives.



# LSTM in Gory Detail

(note: all of the names have been changed to protect the innocent)

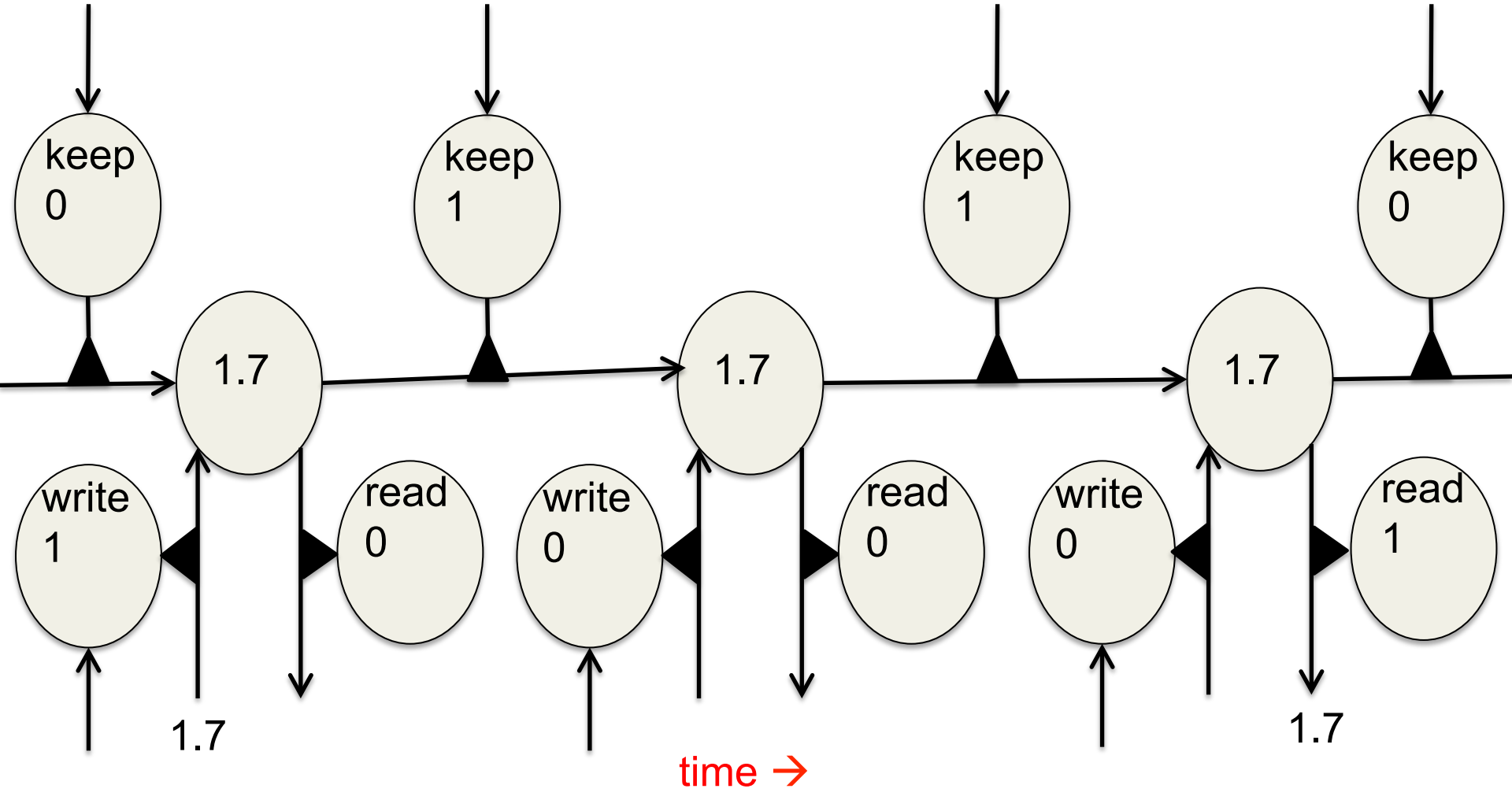



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} \mathbf{D}(h_t^{l-1}) \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

# Backpropagation through a memory cell



# Reading cursive handwriting

- This is a natural task for an RNN.
- The input is a sequence of (x,y,p) coordinates of the tip of the pen, where p indicates whether the pen is up or down.
- The output is a sequence of characters.
- Graves & Schmidhuber (2009) showed that RNNs with LSTM are currently the best systems for reading cursive writing.
  - They used a sequence of small images as input rather than pen coordinates.

# A demonstration of online handwriting recognition by an RNN with Long Short Term Memory (from Alex Graves)

- The movie that follows shows several different things:
- Row 1: This shows when the characters are recognized.
  - It never revises its output, so difficult decisions are delayed.
- Row 2: This shows the states of a subset of the memory cells.
  - Notice how they get reset when it recognizes a character.
- Row 3: This shows the writing. The net sees the x and y coordinates.
  - Optical input actually works better than pen coordinates.
- Row 4: This shows the gradient backpropagated all the way to the x and y inputs from the currently most active character.
  - This lets you see which bits of the data are influencing the decision.

# YouTube

https://www.youtube.com/watch?v=mLxsbWAYIpw

# Summary

- To model sequential data, we can
  - Use input “buffering” to represent the sequence – mapping time into space, like NETTalk
  - Use *recurrence* in the network, mapping time into the state of the network
- Recurrence can be implemented by *unrolling the network in time* to turn a recurrent net into a feedforward one.
  - Early versions (Jordan and Elman) use simple architectures and unrolled one time step
  - We can envision the internal state space using PCA

# Summary

- Networks like this can be used in multiple ways:
  - To recognize a sequence
  - To produce a sequence
  - To transform a sequence into another domain
  - To control a device – say, a robot
- LSTM units are a gadget that allows the network to “latch” a memory, and hold onto it relatively indefinitely
- LSTM units have revolutionized recurrent networks, allowing them to learn all sorts of interesting tasks