Neural Networks

Lecture 7
Learning in recurrent networks

Recurrent networks

- A feed-forward neural net just computes a fixed sequence of non-linear learned transformations to convert an input patter into an output pattern.
- If the connectivity has directed cycles, the network can do much more:
  - It can oscillate. This is useful for generating cycles needed for e.g. walking.
  - It can settle to point attractors. These are a good way to represent the meanings of words
  - It can behave chaotically. This is computationally interesting and may be useful in adversarial situations.
    - But its a bad idea for most information processing.

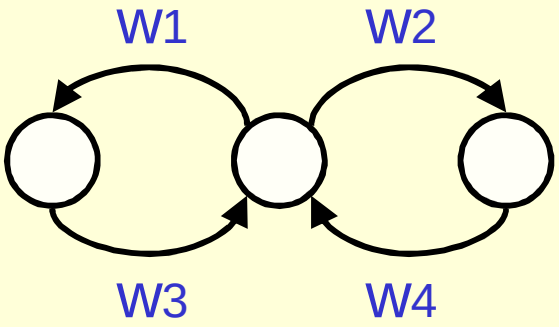
More uses of recurrent networks

- They can remember things for a long time.
  - The network has internal state. It can decide to ignore the input for a while if it wants to.
  - But it is very hard to train a recurrent net to store information that is not used until a long time later (more on this later).
- They can model sequential data in a much more natural way than by using a fixed number of previous inputs to predict the next input (as we did with the model for predicting the next word).
  - The hidden state of a recurrent net can carry along information about a potentially unbounded number of previous inputs.
    - Engineers call it an “infinite impulse response model”.

An advantage of modeling sequential data

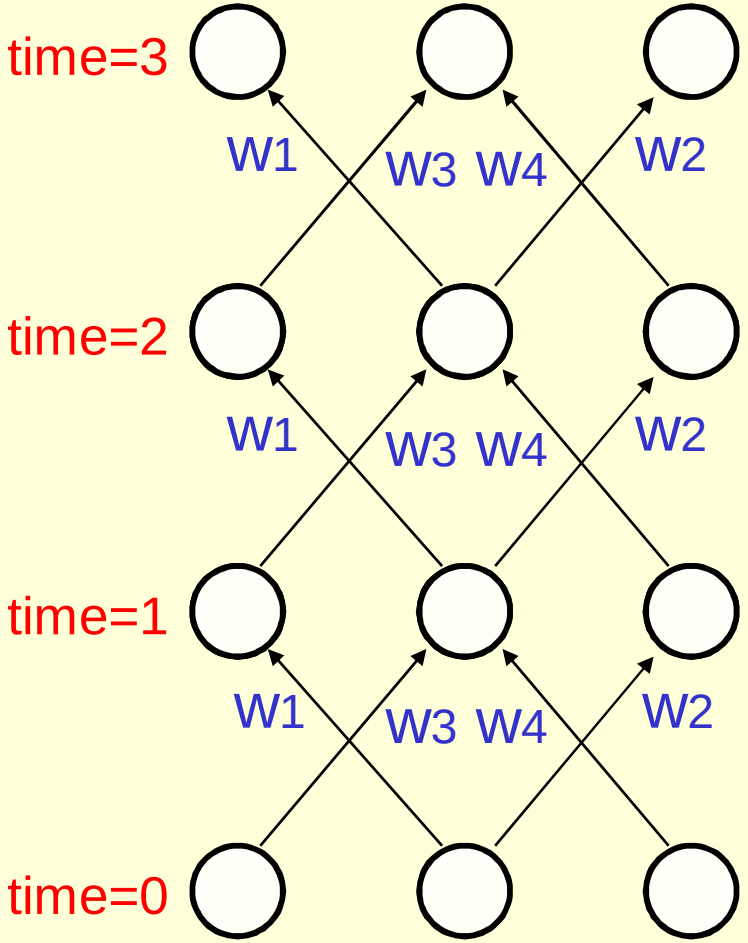
- We can get a teaching signal by trying to predict the next term in a series.
  - This seems much more natural than trying to predict one pixel in an image from the other pixels.

The equivalence between layered, feedforward nets and recurrent nets



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.
- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
  - So if the weights started off satisfying the constraints, they will continue to satisfy them.

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

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

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

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

Backpropagation through time

- We could convert the recurrent net into a layered, feed-forward net and then train the feed-forward net with weight constraints.
  - This is clumsy. It is better to move the algorithm to the recurrent domain rather than moving the network to the feed-forward domain.

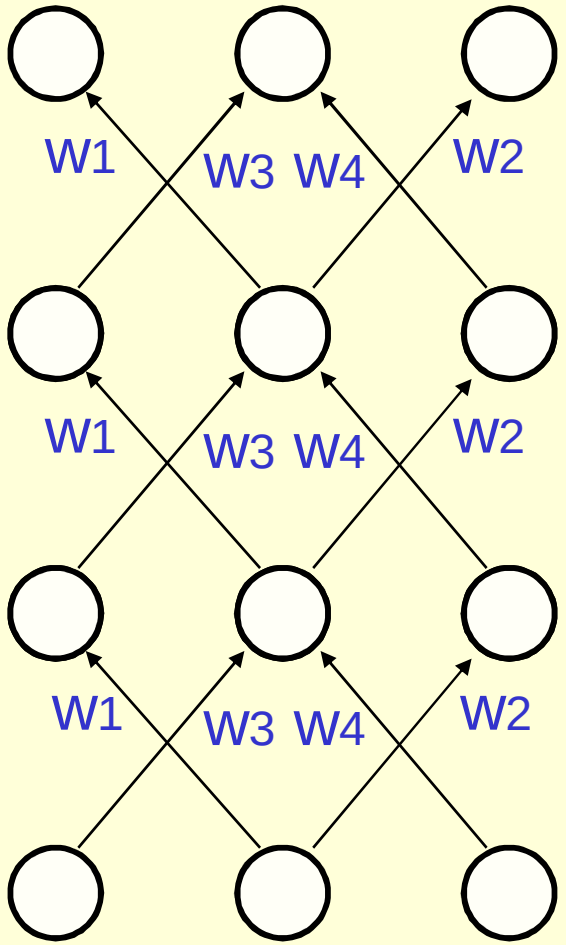
- So the forward pass builds up a stack of the activities of all the units at each time step. The backward pass peels activities off the stack and computes the error derivatives at each time step.
  - After the backward pass we add together the derivatives at all the different times for each weight.

An irritating extra issue

- 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.

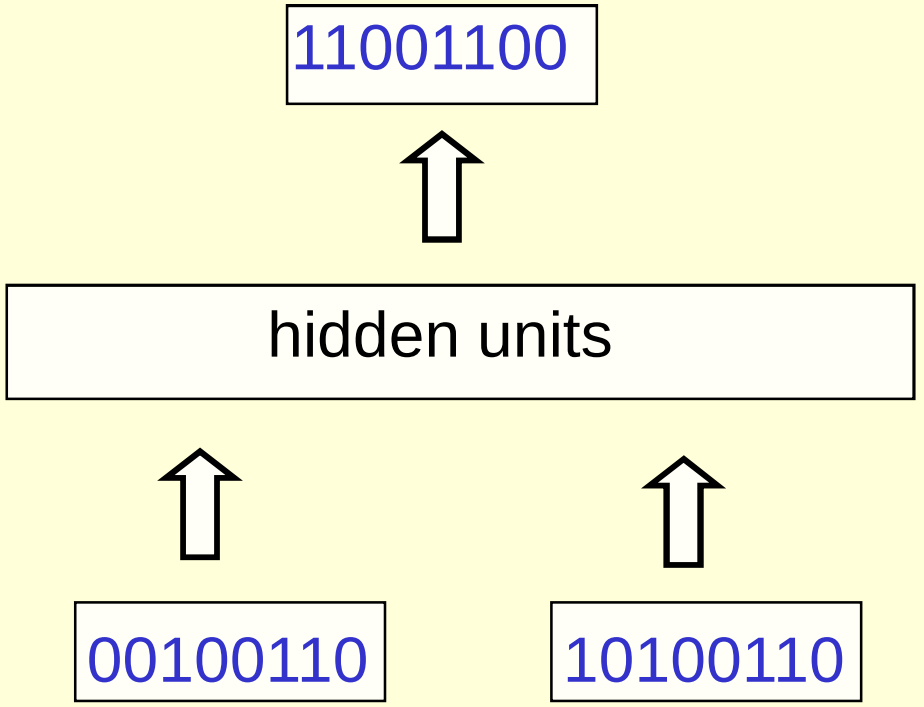
Teaching signals for recurrent networks

- 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 “hidden”

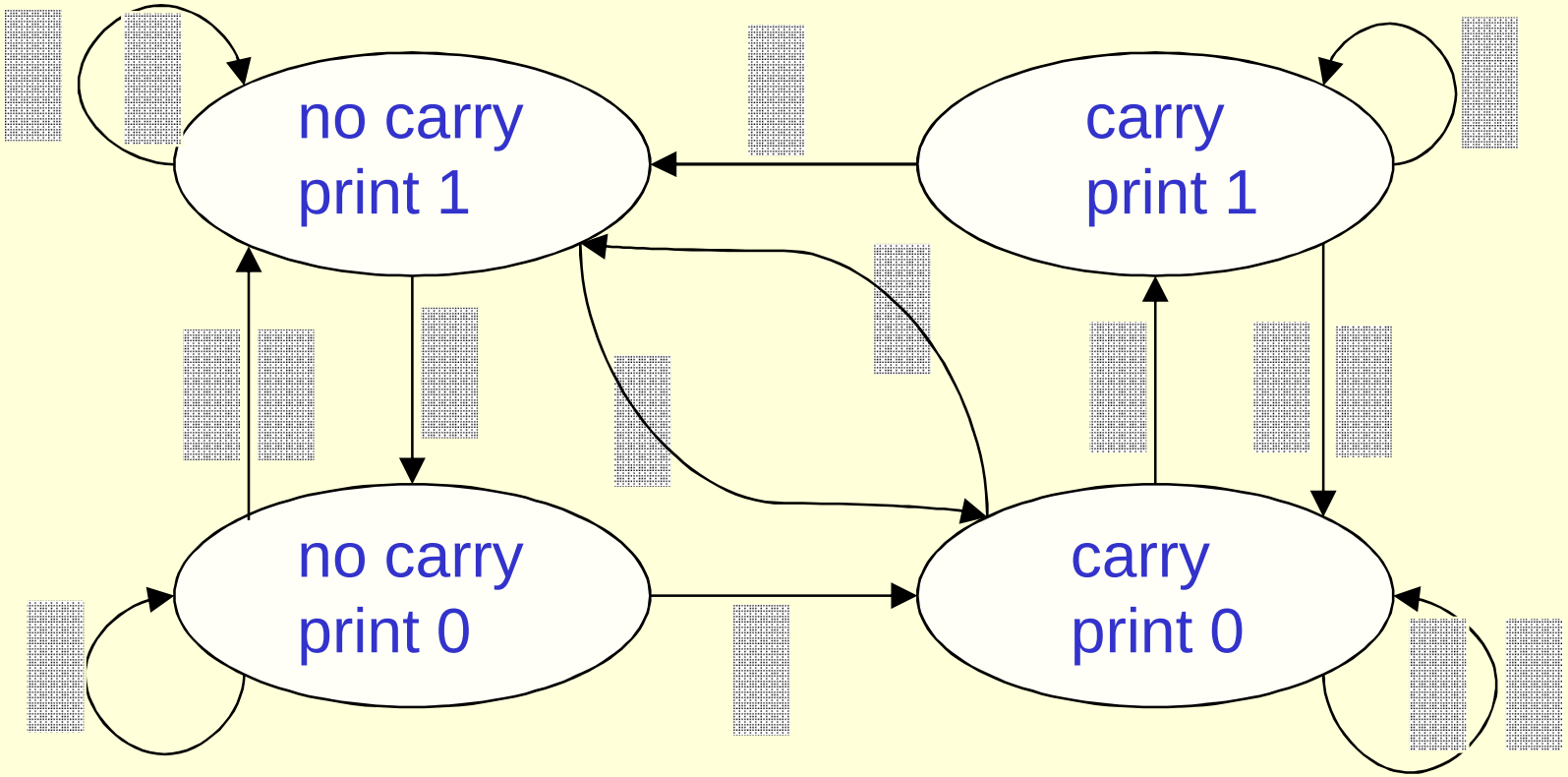


A good problem for a recurrent network

- We can train a feedforward net to do binary addition, but there are obvious regularities that it cannot capture:
  - 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.



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

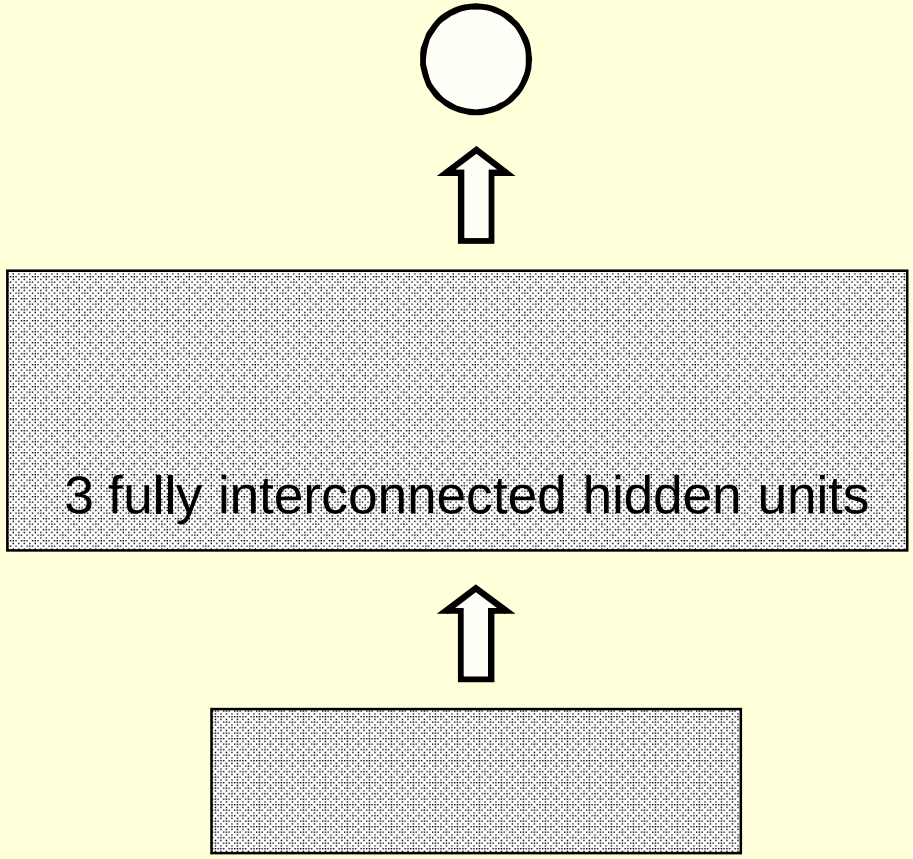
- The network has two input units and one output unit.
- The network is given two input digits at each time step.
- The desired output at each time step is the output for the column that was provided as input two time steps ago.
  - It takes one time step to update the hidden units based on the two input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

- 0 0 1 1 1 0 0
- 0 1 0 0 1 0 1
- 1 0 0 0 0 0 1

← time

The connectivity of the network

- The 3 hidden units have all possible interconnections in all 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.



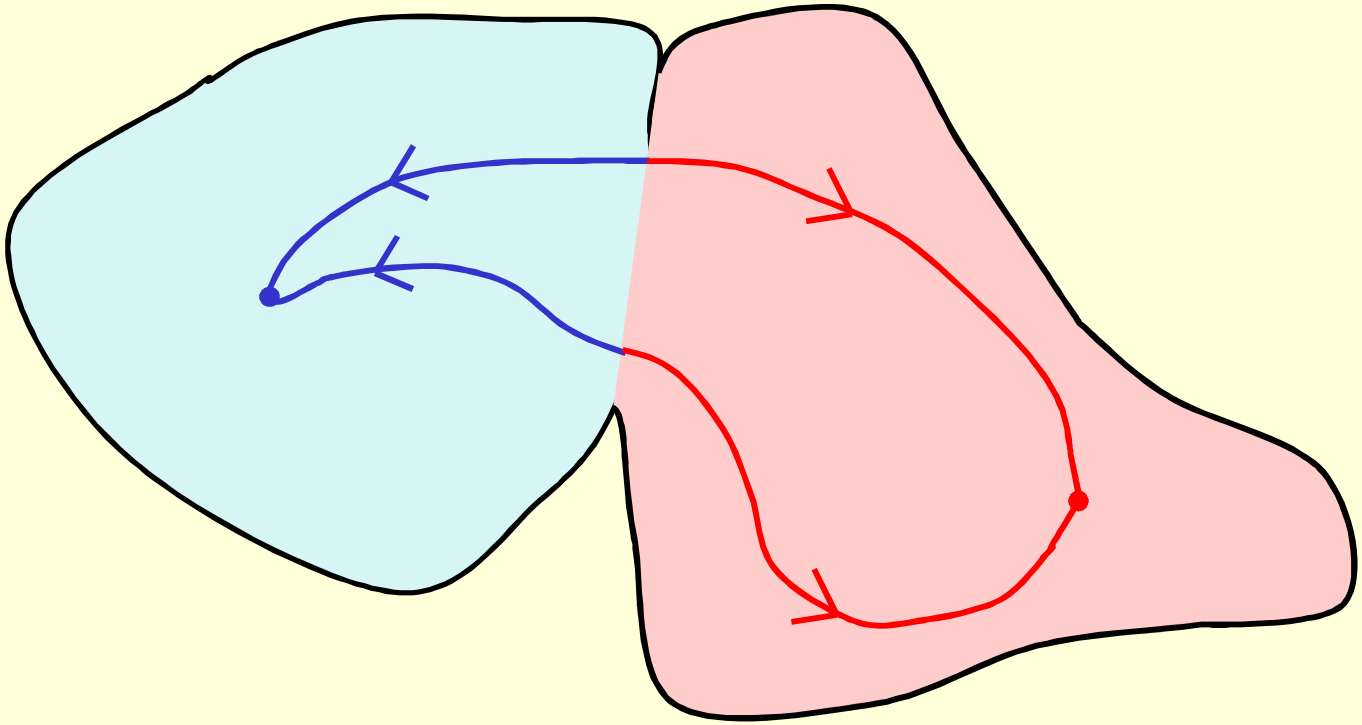
What the network learns

- It learns four distinct pattern 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 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 in the hidden units.
  - This is important when the input stream has several separate things going on at once. A finite state automaton cannot cope with this properly.

A huge disappointment

- Recurrent neural networks are an extremely powerful class of model.
- If we could train them effectively, we could do wonderful things with them.
- Unfortunately, it is very difficult to learn long-term dependencies in a recurrent net.
  - The net is stable during the forward pass because, at each time step, activities are bounded between 0 and 1 by using the sigmoid.
  - But the backward pass is linear. It uses the slope of the sigmoid as a gain factor, but the backpropagated signal can blow up exponentially if the weights get big.

Why the back-propagated gradient blows up



- If we start a trajectory within an attractor, small changes in where we start make no difference to where we end up.
- But if we start almost exactly on the boundary, tiny changes can make a huge difference.

Saved by some heavy math

- Some directions have big gradients but big curvatures.
  - So if you change the weights in those directions, the initial progress is good, but you very quickly start making things worse again.
- Other directions can have tiny gradients but even tinier curvatures.
  - Changing the weights in those directions makes slow progress that lasts for many weight changes.
- There is a very strong correlation between the gradient and the curvature.
  - It's the gradient/curvature ratio that matters.
  - A very clever optimizer can find directions with small gradients but good ratios.