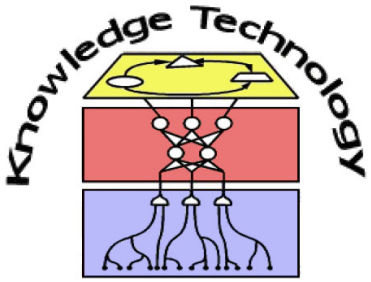
Knowledge Processing with Neural Networks

Lecture 3: Learning in Multilayer Networks



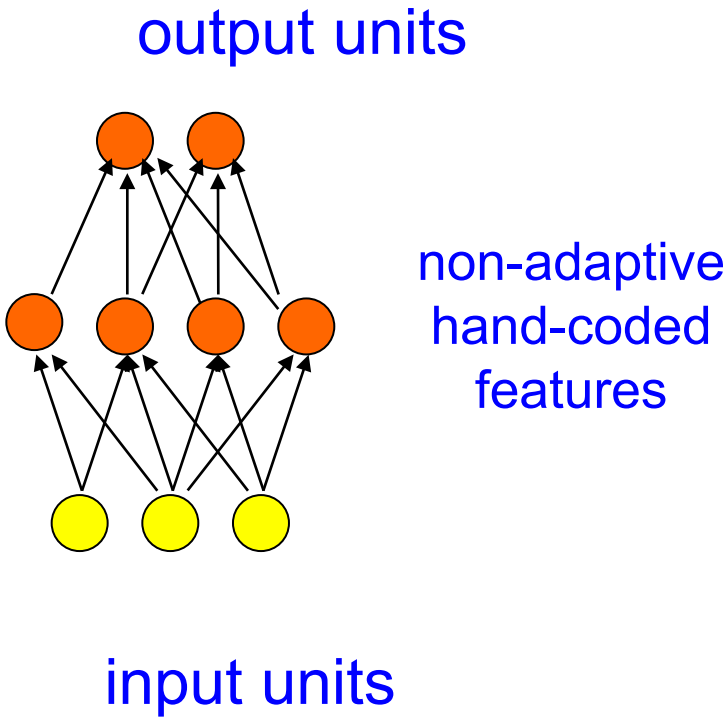
http://www.informatik.uni-hamburg.de/WTM/

Preprocessing the input vectors

- Instead of trying to predict the answer directly from the raw inputs we could start by ***extracting a layer of*** "***features***".
  - Sensible if we already know that certain combinations of input values would be useful (e.g. edges or corners in an image).
- Instead of learning the features we could design them by hand.
  - The ***hand-coded features*** are equivalent to a layer of non-linear neurons that do not need to be learned.
  - So far as the learning algorithm is concerned, the activities of the hand-coded features are the input vector.

The connectivity of a perceptron

- The **output units** are binary threshold neurons and are each learned independently.
- Only the top layer of weights is learned.
- The input is recoded using hand-picked features that do not adapt.



Is preprocessing forbidden or effective?

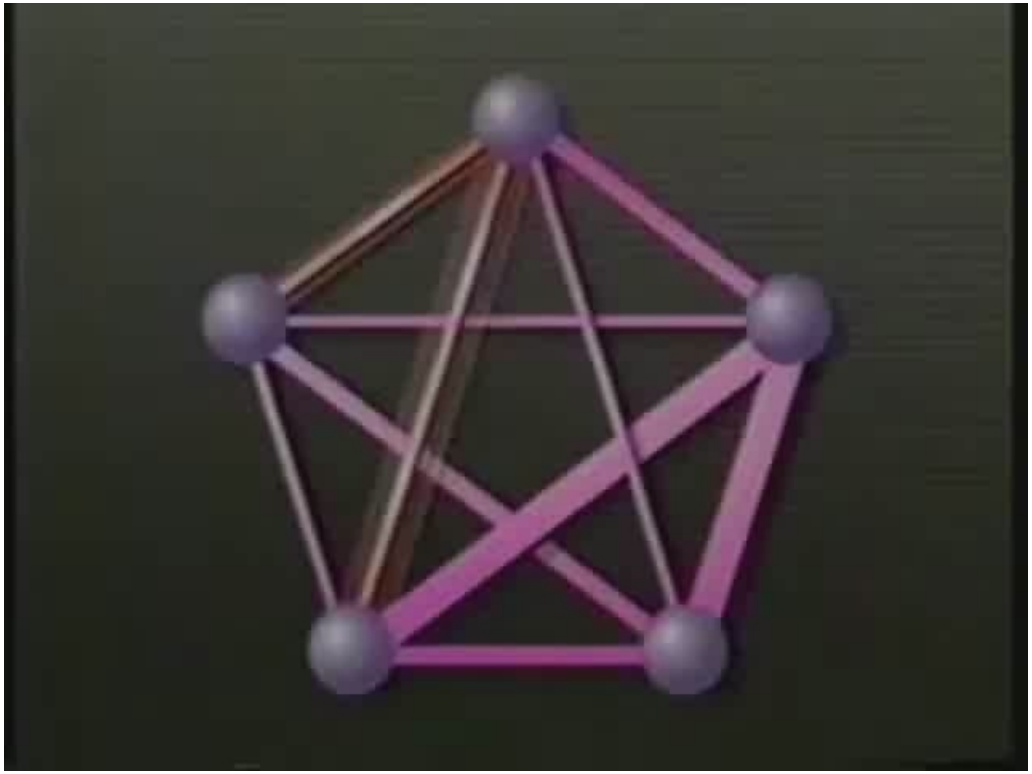
- Some task is supported by the preprocessing. It seems like it should be forbidden if the aim is to show how powerful learning is?
- Its certainly ok if we learn the preprocessing.
  - This makes learning much more difficult and much more interesting…
- Its ok if we use a very big set of non-linear features that is task-independent.
  - Support Vector Machines make it possible to use a huge number of features without requiring much computation or data.
- Nevertheless it can be effective to introduce domain knowledge into the network

What can perceptrons do?

- They can only solve tasks if the hand-coded features convert the original task into a linearly separable one. How difficult is this?
- The N-bit parity task:
  - Requires N features of the form: Are at least m bits on?
  - Each feature must look at all the components of the input.
- The 2-D connectedness task
  - requires an exponential number of features!

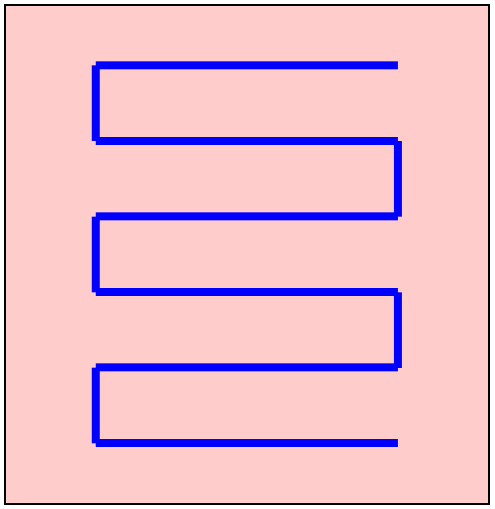
The 7-bit parity task
1011010 → 0
0111000 → 1
1010111 → 1

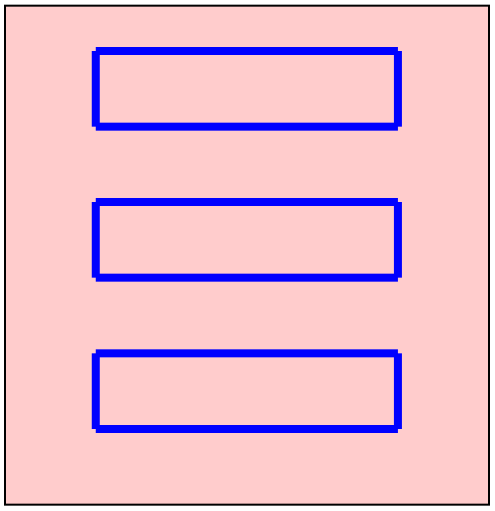
Perceptrons in 50-60’



Why connectedness is hard to compute

- Even for simple line drawings, there are exponentially many cases.
- Removing one segment can break connectedness
  - But this depends on the precise arrangement of the other pieces.
  - Unlike parity, there are no simple summaries of the other pieces that tell us what will happen.
- Connectedness is easy to compute with a serial algorithm.
  - Start anywhere in the ink
  - Propagate a marker
  - See if all the ink gets marked.





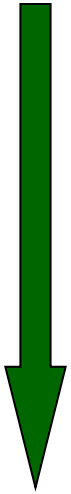
Learning with hidden units

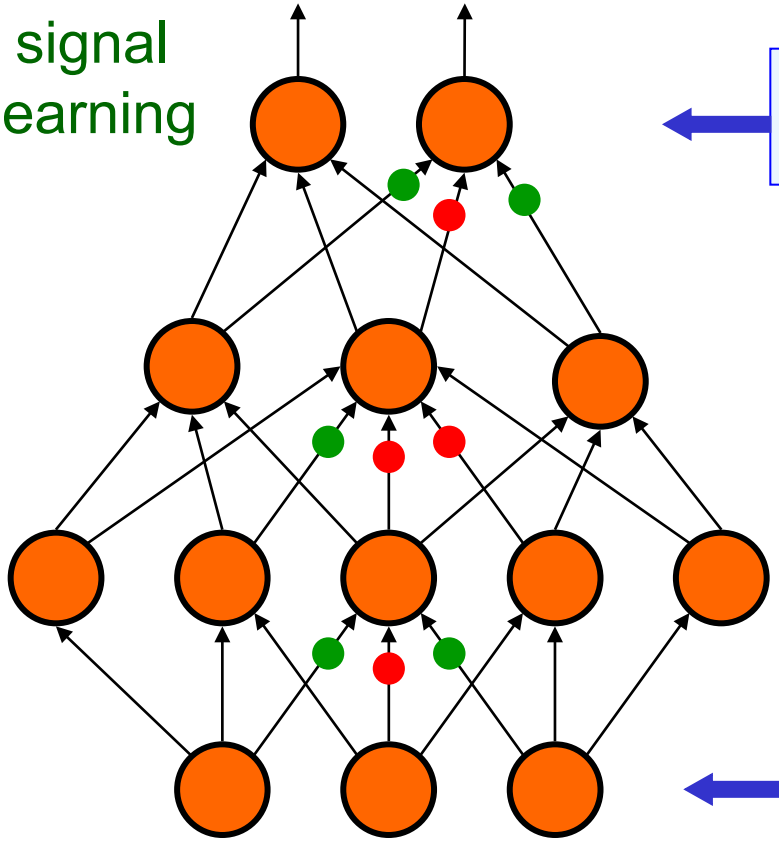
- Networks ***without hidden*** units are very ***limited*** in the input-output mappings they can model.
  - More layers of linear units do not help. Its still linear.
  - Fixed output non-linearities are not enough
- We need ***multiple*** layers of ***adaptive non-linear*** hidden units. This gives us a universal approximator.
- But how can we train such nets?
  - We need an efficient way of adapting all the weights, not just the last layer. This is hard. Learning the weights going into hidden units is equivalent to learning features.
  - Nobody is telling us directly what hidden units should do. (That's why they are called hidden units).

Learning by back-propagating error derivatives

Compare outputs with correct answer to get error signal

Back-propagate error signal to get derivatives for learning





outputs

hidden layers

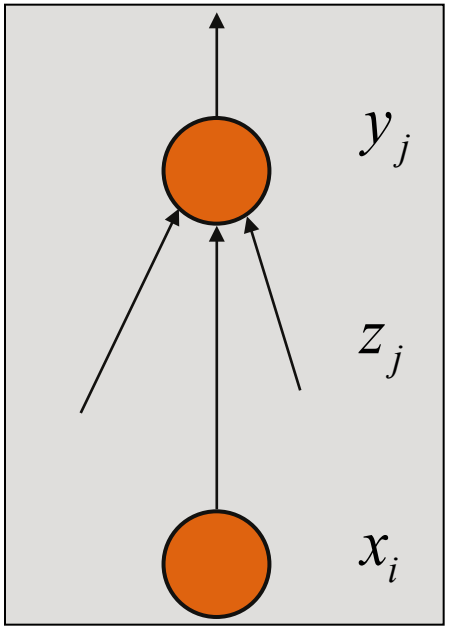
input vector

The idea behind backpropagation

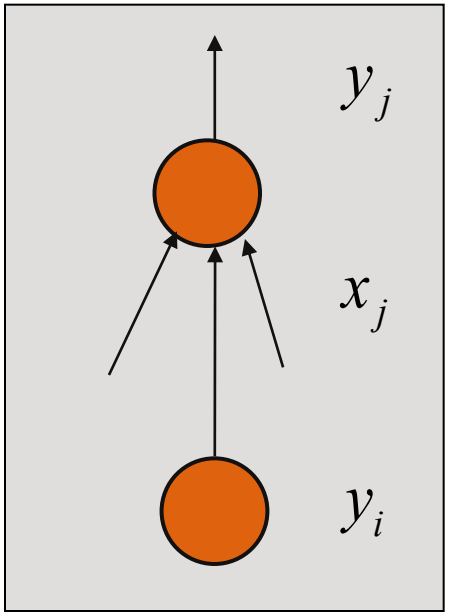
- We don't know what the hidden units ought to do, but we can compute how the error changes as we change a hidden activity: direction and magnitude of error change.
- Instead of using desired activities to train the hidden units, use ***error derivatives w.r.t. hidden activities***.
- Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.
- We can compute error derivatives for ***all*** the hidden units efficiently.
- Once we have the error derivatives for the hidden activities, it is easy to get the error derivatives for the weights going into a hidden unit.

A change of notation

- For simple networks we use the notation
  - ***x*** for activities of input units
  - ***y*** for activities of output units
  - ***z*** for the summed input to an output unit



- For networks with multiple hidden layers:
  - ***y*** is used for the output of a unit in any layer
  - ***x*** is the summed input to a unit in any layer
  - The ***index*** indicates which layer a unit is in.



Non-linear neurons with smooth derivatives

- For backpropagation, we need neurons that have well-behaved derivatives.
  - Typical: the logistic function
  - The output is a smooth function of the summed input

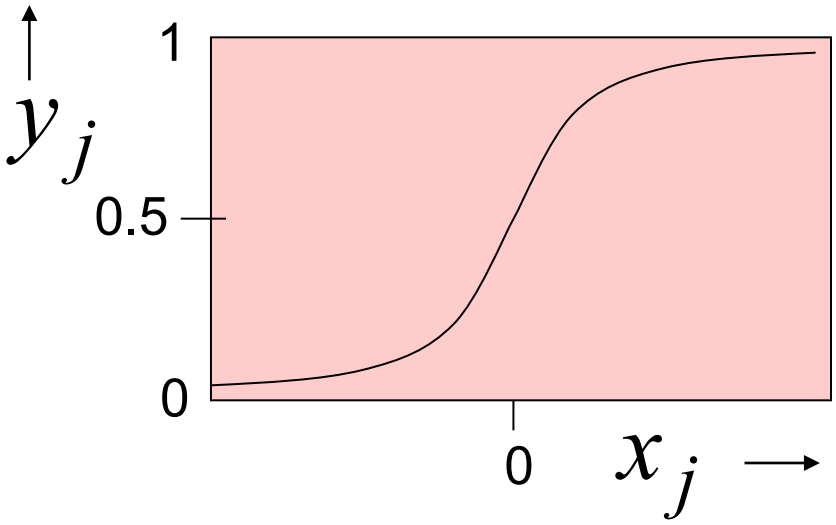
$$x_j = b_j + \sum_i y_i w_{ij}$$

$$y_j = \frac{1}{1 + e^{-x_j}}$$

$$\frac{\partial x_j}{\partial w_{ij}} = y_i \qquad \frac{\partial x_j}{\partial y_i} = w_{ij}$$

$$\frac{dy_j}{dx_j} = y_j (1 - y_j)$$

Its odd to express it in terms of y.



Sketch of the backpropagation algorithm on a single training case

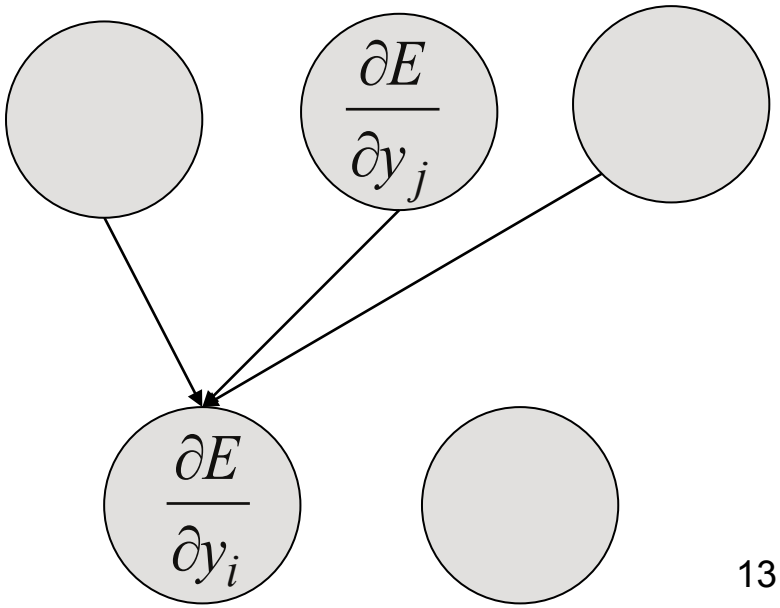
- First convert the discrepancy between each output and its target value into an error derivative.

$$E = \frac{1}{2}\sum_j (y_j - d_j)^2$$

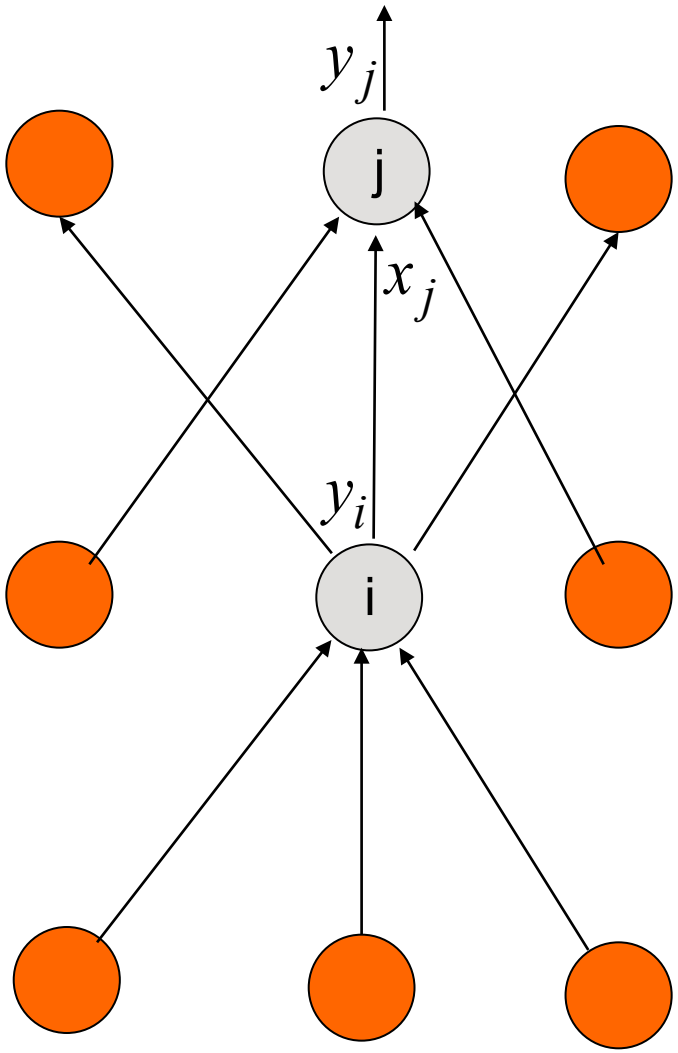
$$\frac{\partial E}{\partial y_j} = y_j - d_j$$

- Then compute error derivatives in each hidden layer from error derivatives in the layer above.

- Then use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.



The derivatives



$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j}\frac{dy_j}{dx_j} = \frac{\partial E}{\partial y_j} y_j (1 - y_j)$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} = \frac{\partial E}{\partial x_j} y_i$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j}\frac{dx_j}{dy_i} = \sum_j \frac{\partial E}{\partial x_j} w_{ij}$$

Ways to use weight derivatives

- How often to update
  - after each training case?
  - after a full sweep through the training data?
  - after a “mini-batch” of training cases?

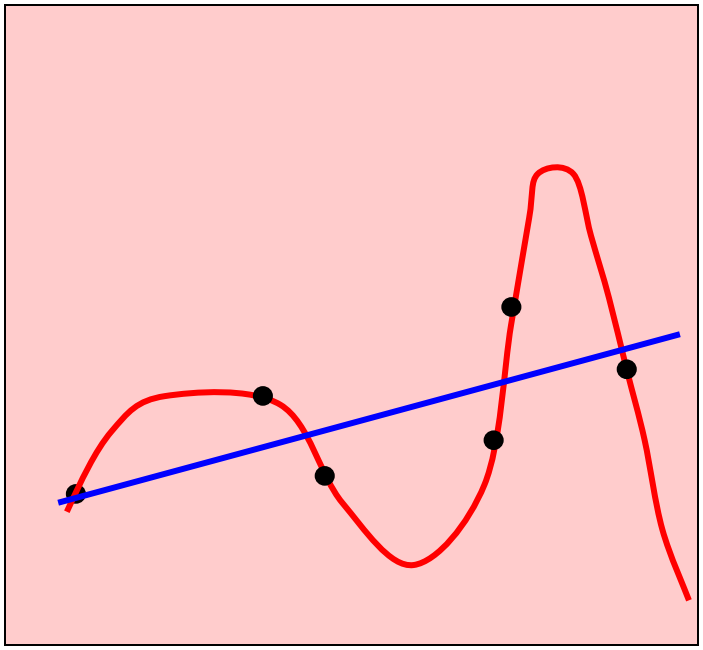
- How much to update
  - Use a fixed learning rate?
  - Adapt the learning rate?

  - Add momentum?
  - Don’t use steepest descent?

Overfitting

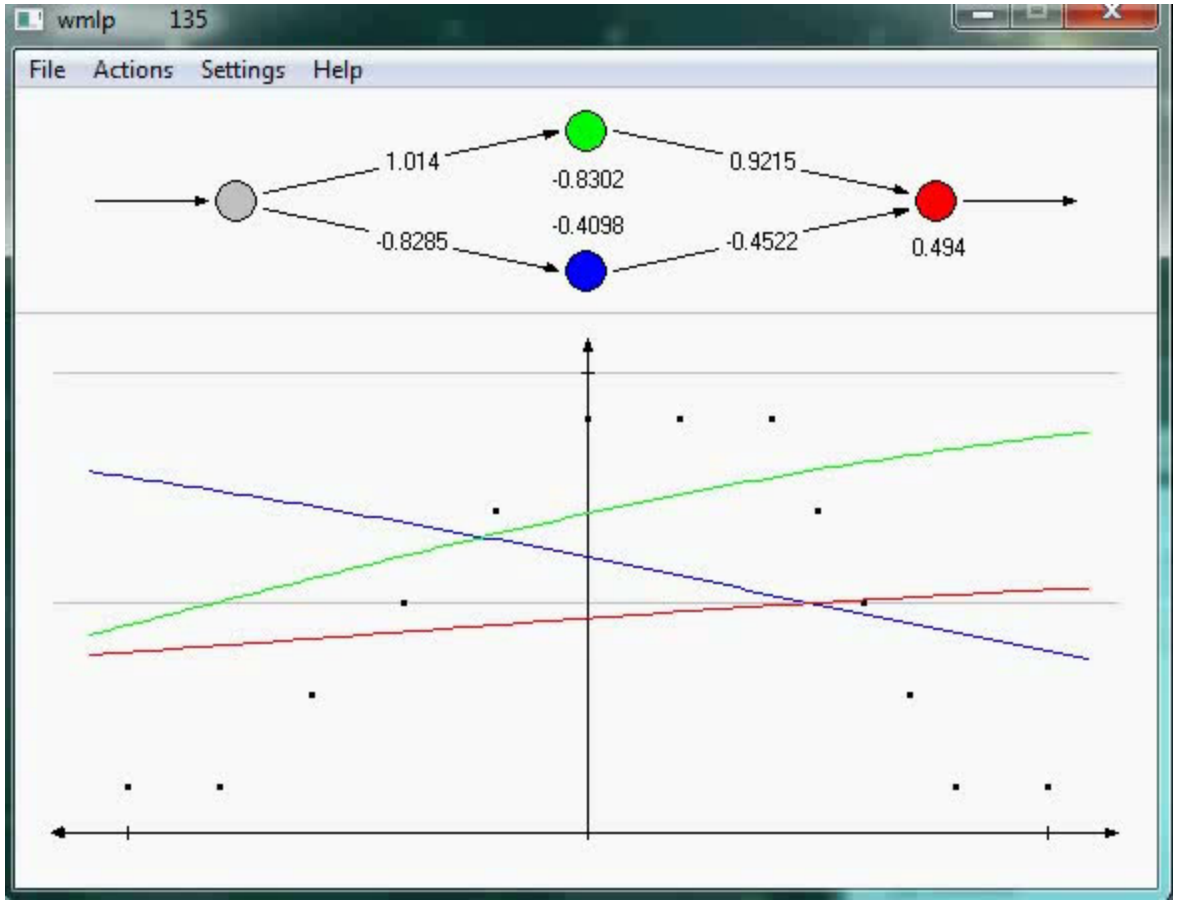
- The training data contains information about the regularities in the mapping from input to output. But it also contains noise
  - The target values may be unreliable.
  - There is ***sampling error***. There will be accidental regularities just because of the particular training cases that were chosen.
- The model cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity.
  - If the model is very flexible it can model the sampling error really well. ***This is a disaster***.

A simple example of overfitting



- Which model do you believe?
  - The complicated model ***fits*** the ***data better***.
  - But it is ***not economical***
- A model is convincing when it fits a lot of data surprisingly well.
  - It is not surprising that a complicated model can fit a small amount of data.

An example of successful training of real-valued functions

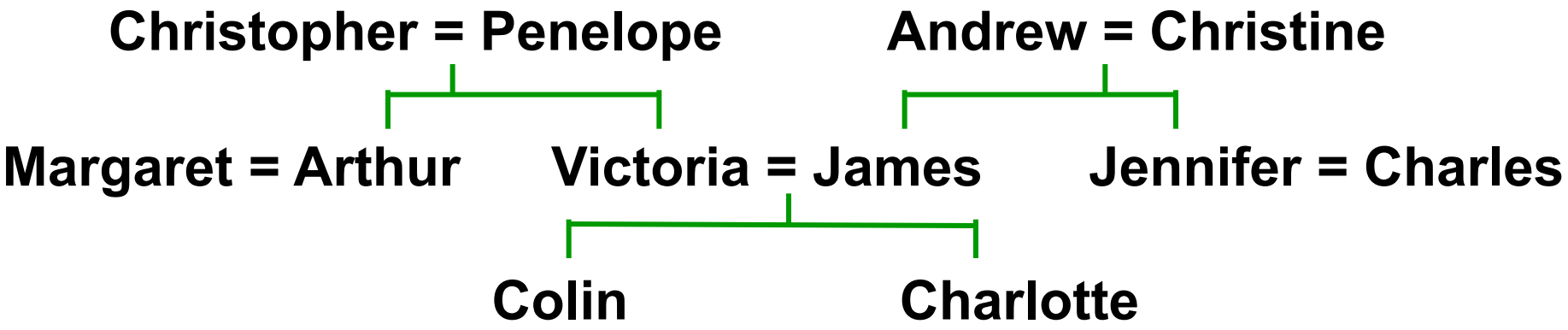


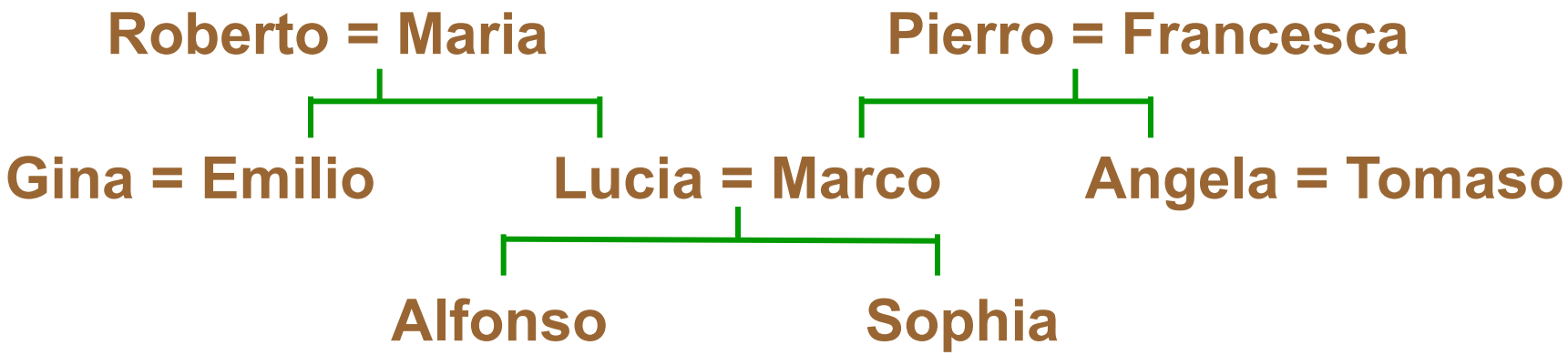
http://www.borgelt.net/

Some Success Stories

- Back-propagation has been used for a large number of practical applications.
  - Recognizing hand-written characters
  - Predicting the future price of stocks
  - Detecting credit card fraud
  - Recognize speech (wreck a nice beach)
  - Predicting the next word in a sentence from the previous words
    - This is essential for good speech recognition.
  - Understanding the effects of brain damage

An example of relational information





Another way to express the same information

- Make a set of propositions using the 12 relationships:
  - son, daughter, nephew, niece
  - father, mother, uncle, aunt
  - brother, sister, husband, wife
- (colin has-father james)
- (colin has-mother victoria)
- (james has-wife victoria) this follows from the two above
- (charlotte has-brother colin)
- (victoria has-brother arthur)
- (charlotte has-uncle arthur) this follows from the above

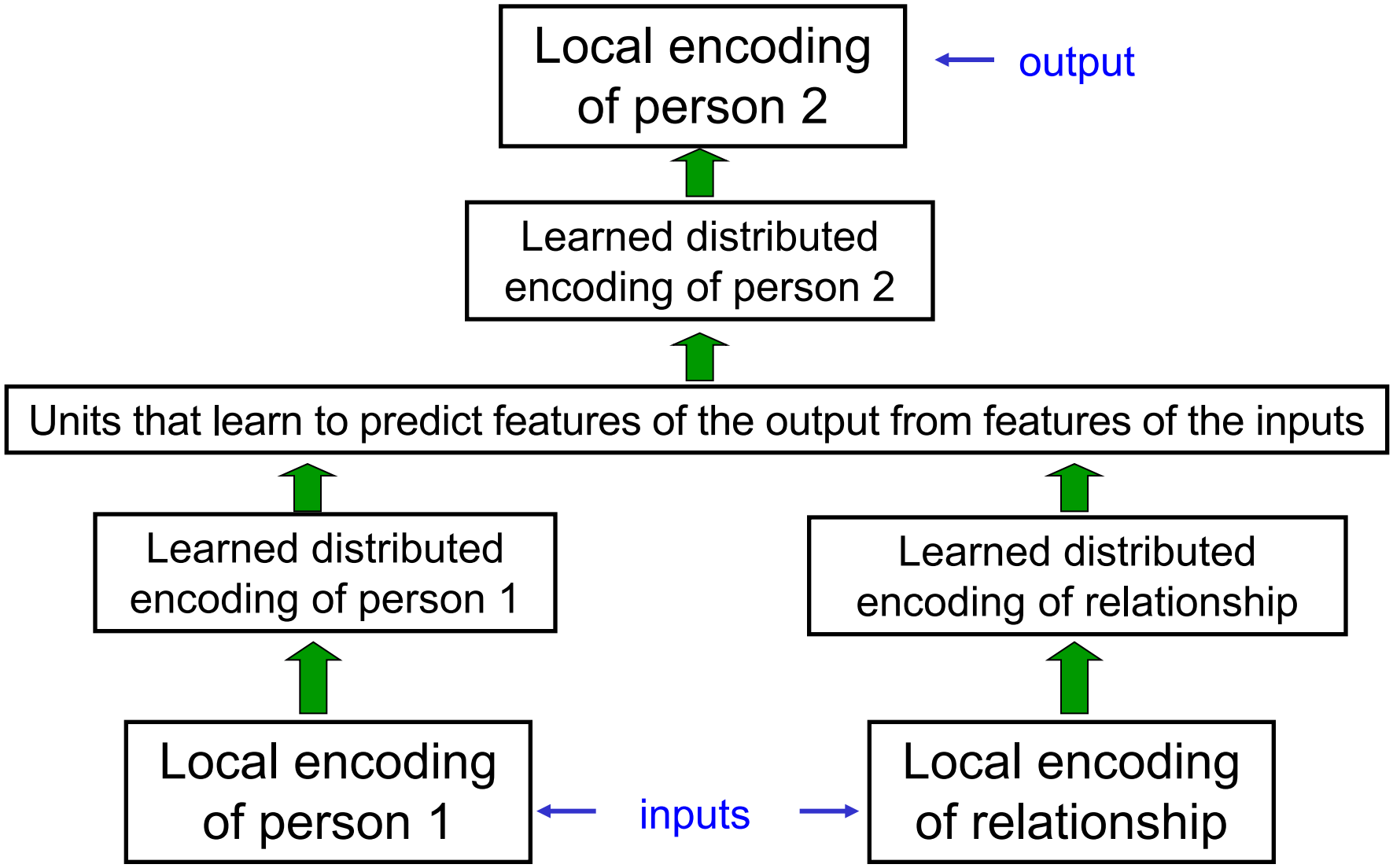
A relational learning task

- Given a large set of triples that come from some family trees, figure out the regularities.
  - The obvious way to express the regularities is as symbolic rules

```
(x has-mother y) & (y has-husband z) => (x has-father z)
```

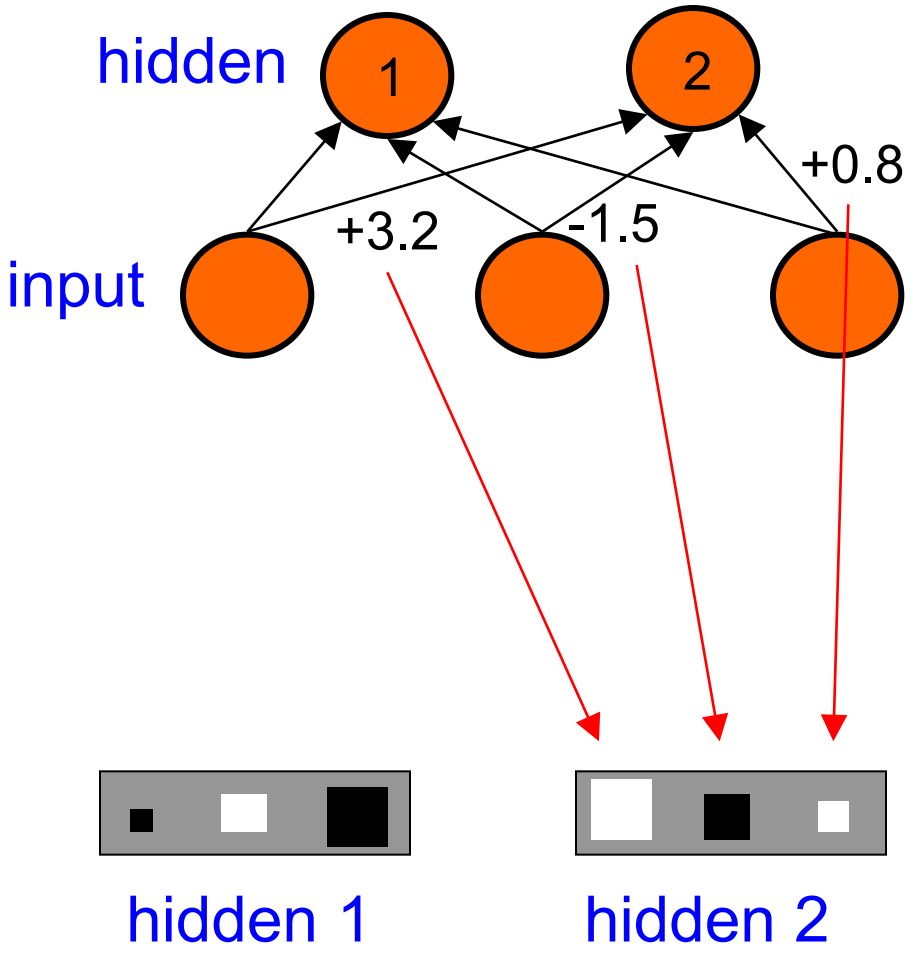
- Finding the symbolic rules involves a difficult search through a very large discrete space of possibilities.
- Can a neural network capture the same knowledge by searching through a continuous space of weights?

The structure of the neural net

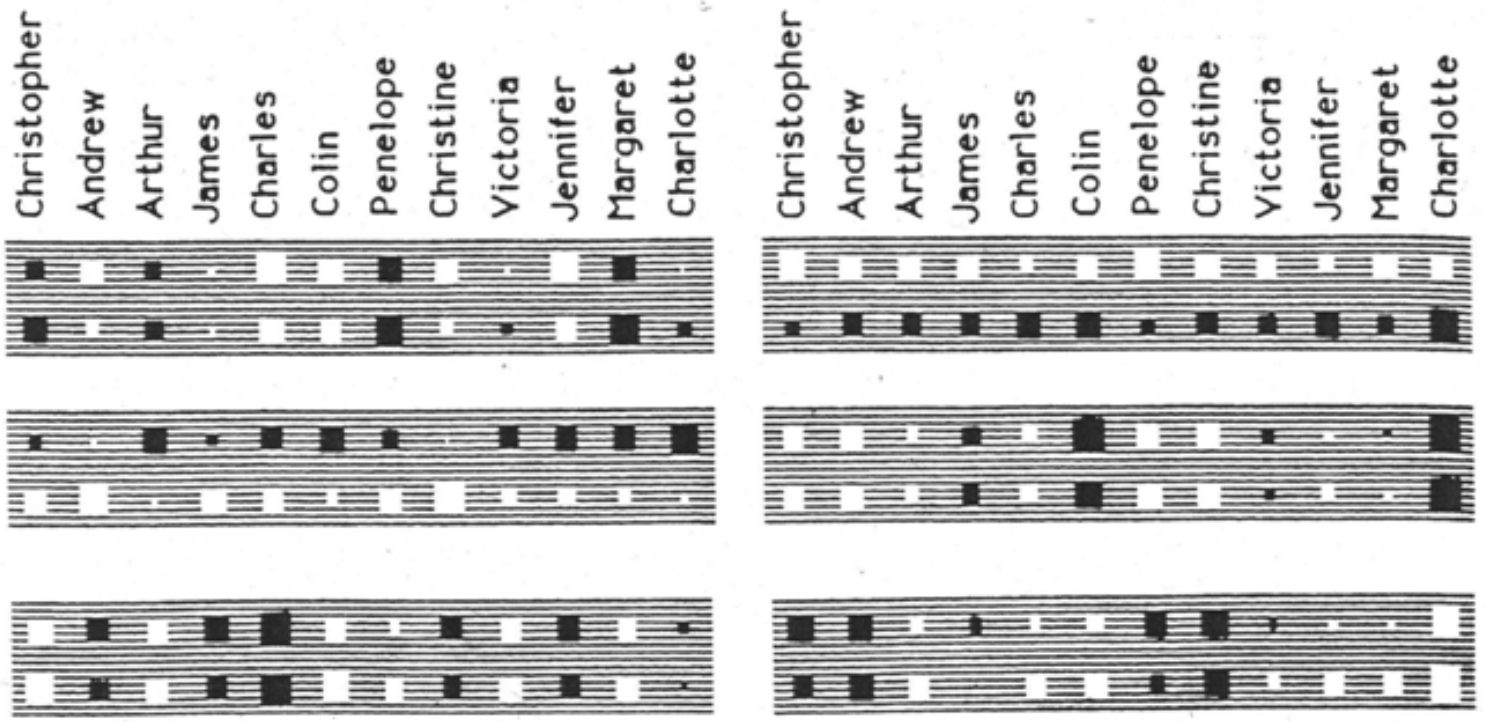


How to show the weights of hidden units

- The obvious method is to show numerical weights on the connections:
  - Try showing 25,000 weights this way!
- Its better to show the weights as black or white blobs in the locations of the neurons that they come from
  - Better use of pixels
  - Easier to see patterns



The features it learned for person 1



Christopher = Penelope **Andrew = Christine**

Margaret = Arthur **Victoria = James** **Jennifer = Charles**

Colin **Charlotte**

What the network learns

- The six hidden units in the bottleneck connected to the input representation of person 1 learn to represent features of people that are useful for predicting the answer.
  - Nationality, generation, branch of the family tree.
- These features are only useful if the other bottlenecks use similar representations and the central layer learns how features predict other features.
  - For example:

    Input person is of generation 3

    and

    relationship requires answer to be one generation up

    implies

    Output person is of generation 2

Another way to see that it works

- Train the network on all but some of the triples that can be made using the 12 relationships
  - It needs to sweep through the training set many times adjusting the weights slightly each time.

- Then test it on the held-out cases.

Why this is interesting

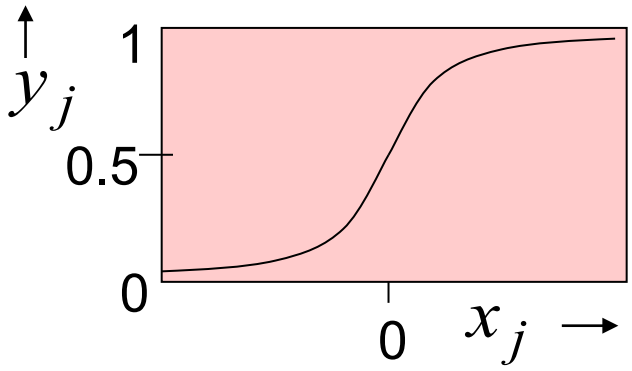
- There has been a big debate in cognitive science between two rival theories of what it means to know a concept:

  The ***feature theory***: A concept is a set of ***semantic features***.
  - This is good for explaining similarities between concepts
  - Its convenient: a concept is a vector of feature activities.

  The ***structuralist theory***: The meaning of a concept lies in its ***relationships to other concepts***.
  - So conceptual knowledge is best expressed as a relational graph.
- These theories need not be rivals. A neural net can use semantic features to implement the relational graph.
  - This means that no explicit inference is required to arrive at the intuitively obvious consequences of the facts that have been explicitly learned. The neural net contains the answer!

Problems with squared error

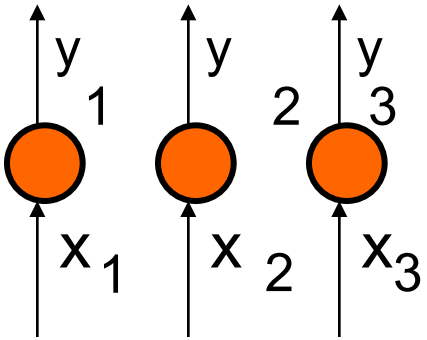
- The squared error measure has some drawbacks
  - If the desired output is 1 and the actual output is 0.00000001 there is almost no gradient for a logistic unit to fix up the error.
  - If we are trying to assign probabilities to class labels, we know that the outputs should sum to 1, but we are depriving the network of this knowledge.



- Is there a different cost function that is more appropriate and works better?
  - Force the outputs to represent a probability distribution across discrete alternatives.

Softmax

The output units use a non-local non-linearity:



output units

$$y_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

$$\frac{\partial y_i}{\partial x_i} = y_i \left(1 - y_i\right)$$

A basic problem in speech recognition

- We cannot identify phonemes perfectly in noisy speech
  - The acoustic input is often ambiguous: there are several different words that fit the acoustic signal equally well.

- People use their understanding of the meaning of the utterance to hear the right word.
  - We do this unconsciously
  - We are very good at it

- This means speech recognizers have to know which words are likely to come next and which are not.
  - Can this be done without full understanding?

The standard "trigram" method

- Take a huge amount of text and count the frequencies of all triples of words. Then use these frequencies to make bets on the next word in a b ?

$$\frac{p(w_3 = c \mid w_2 = b, w_1 = a)}{p(w_3 = d \mid w_2 = b, w_1 = a)} = \frac{count(abc)}{count(abd)}$$

- Until very recently this was state-of-the-art.
  - We cannot use a bigger context because there are too many quadgrams
  - We have to "back-off" to digrams when the count for a trigram is zero.
    - The probability is not zero just because we didn't see one.

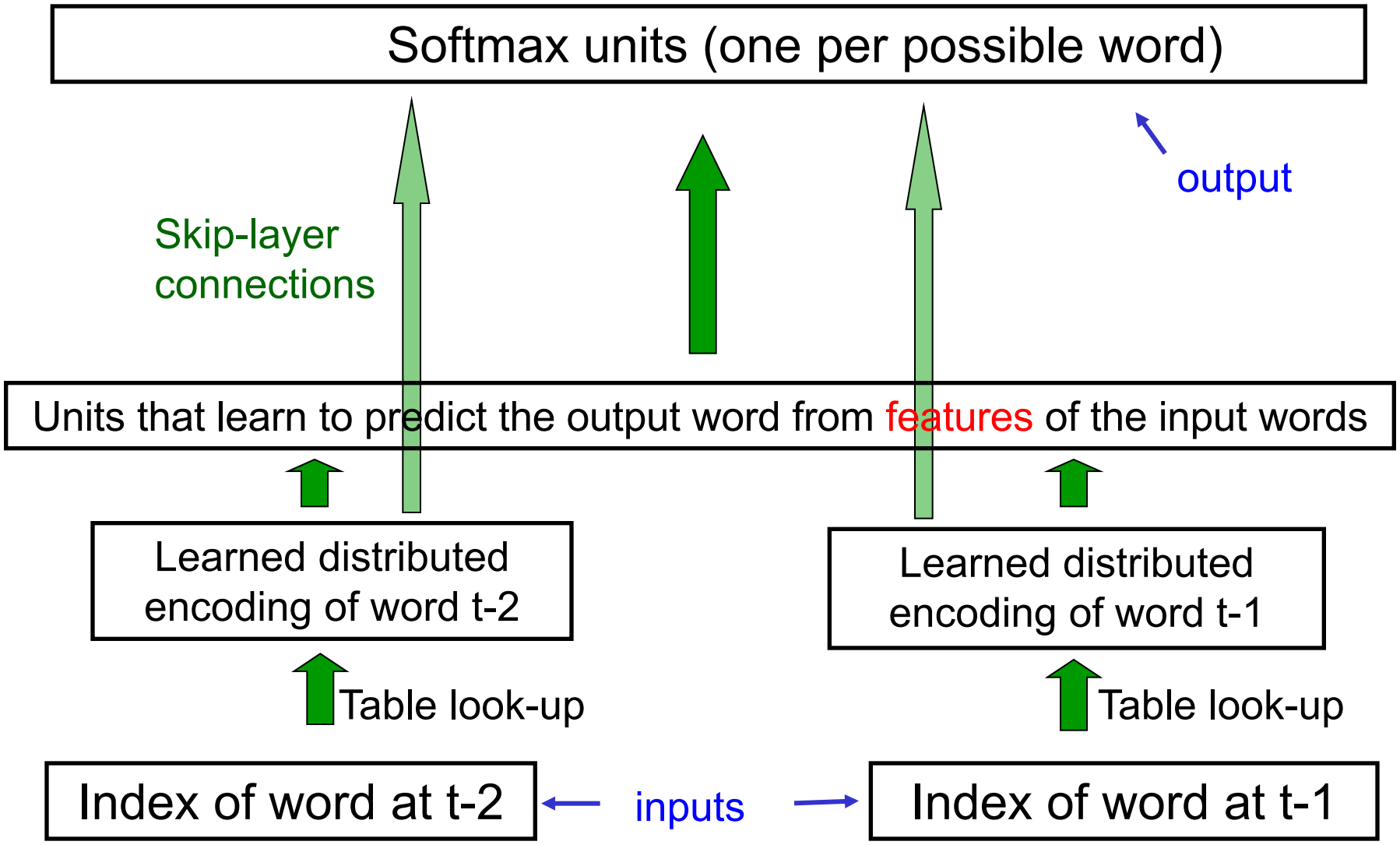
Why the trigram model is inefficient

- Suppose we have seen the sentence

  “the cat got squashed in the garden on friday”
- This should help us predict words in the sentence

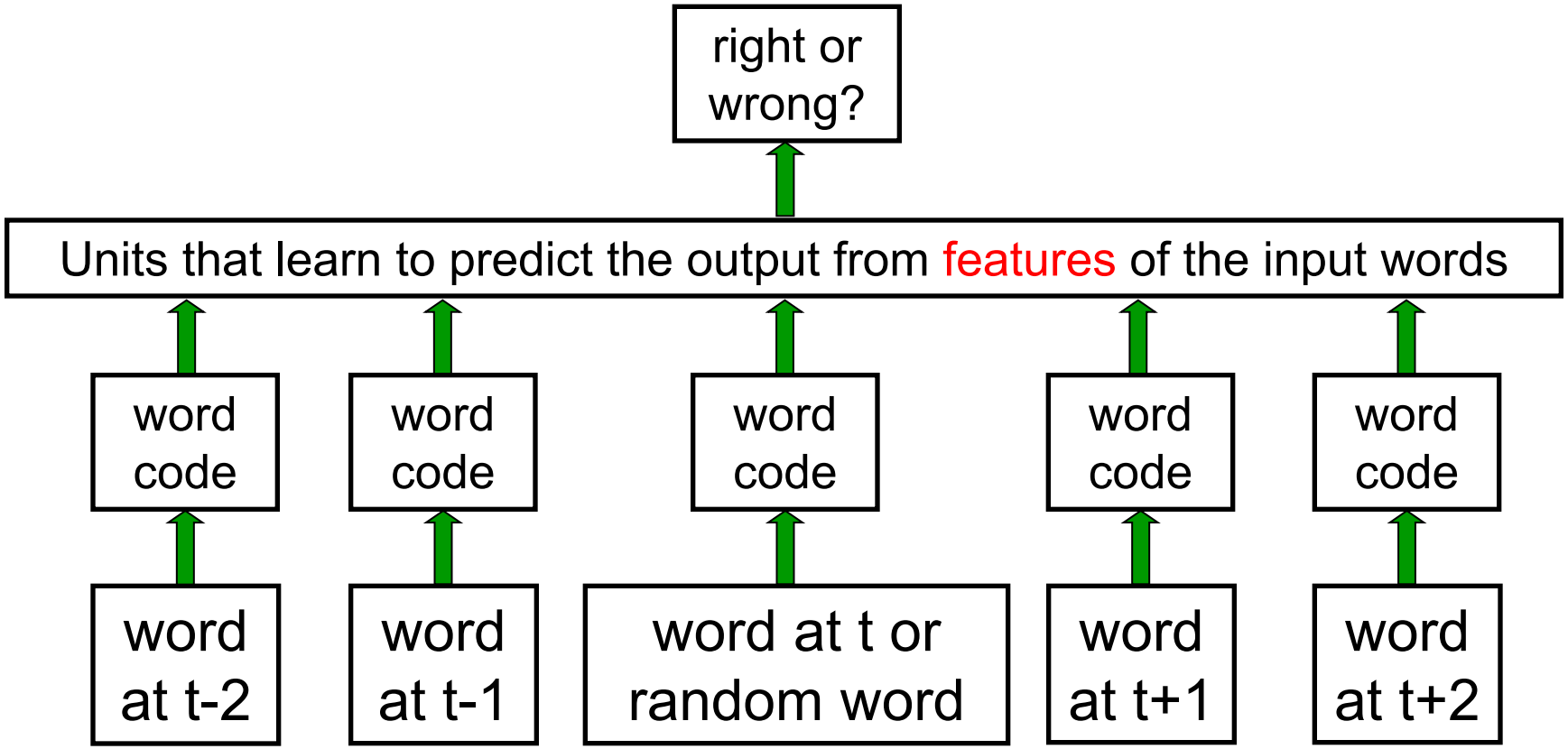
  “the dog got flattened in the yard on monday”
- A trigram model does not understand the similarities between
  - cat/dog squashed/flattened garden/yard friday/monday
- To overcome this limitation, we need to use the features of previous words to predict the features of the next word.
  - Using a feature representation and a learned model of how past features predict future ones, we can use many more words from the past history.

Bengio's neural net for predicting the next word



The Collobert and Weston net

- Learn to judge if a word fits the 5 word context on either side of it. Train on ~600 million words.



Summary

- Multilayer Perceptrons are very effective in various ***classification*** tasks.

- MLPs can be trained with ***gradient descent*** methods, e.g. the backpropagation algorithm
  - Suitable and ***derivable transfer functions*** are necessary.
  - Reasonable stopping criteria help to avoid ***overfitting*.**
  - Divergence during training can be avoided using ***momentum***.

- MLPs also can be used to ***predict features*** of future events. Extension: Recurrent connections...
  - ... details will be given next week.

Perceptrons today: Real time automatic facial expression recognition

http://www.cim.mcgill.ca/