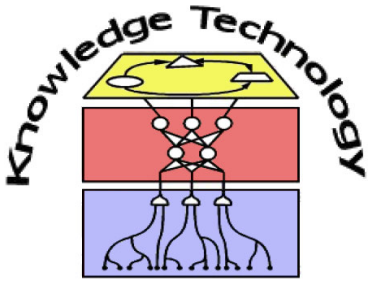
Knowledge Processing with Neural Networks

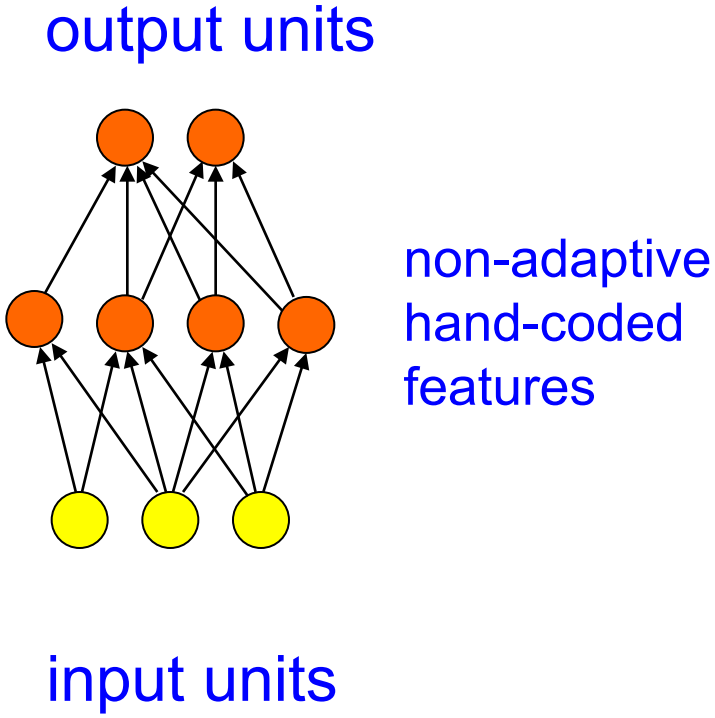
Lecture 4: Recurrent Networks & Distributed and Localist Representations



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

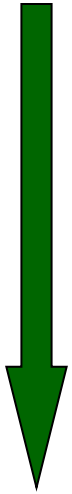
Revision: The connectivity of a perceptron

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

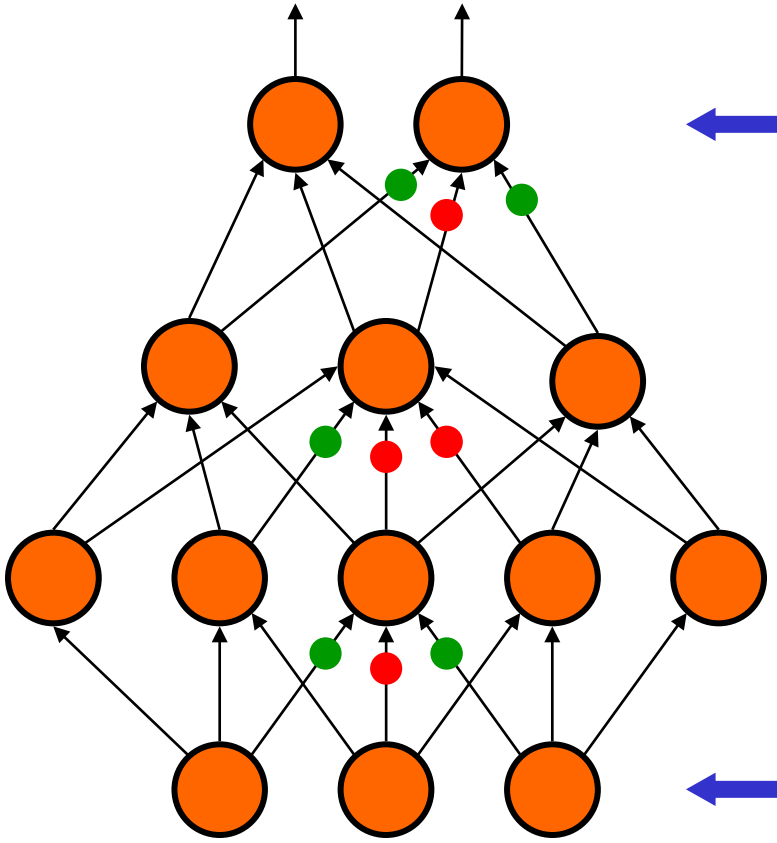


Learning by back-propagating error derivatives

Back-propagate error signal to get derivatives for learning



Compare outputs with correct answer to get error signal



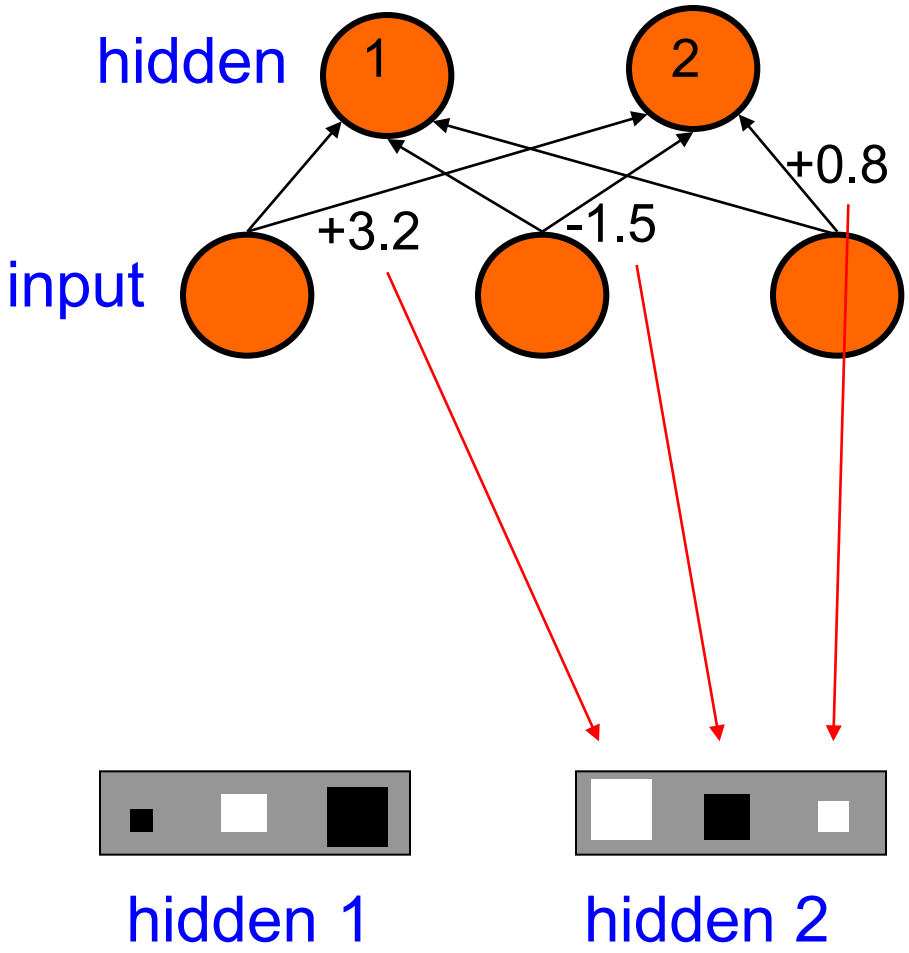
outputs

hidden layers

input vector

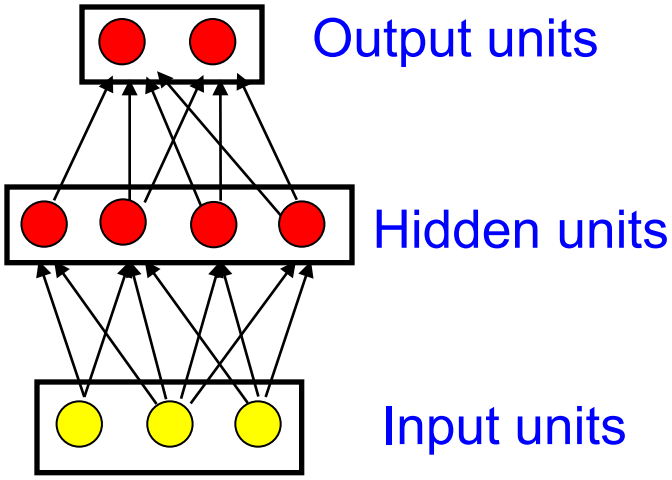
How to show the weights of hidden units

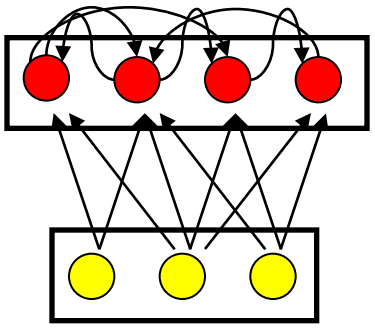
- The obvious method is to show numerical weights on the connections:
  - Try showing 25,000 weights this way!
- Its better to show the weights as black or white blobs in the locations of the neurons that they come from
  - Better use of pixels
  - Easier to see patterns



Introduction to recurrent neural networks: Types of connectivity

- Feedforward networks
  - These compute a series of transformations.
  - Typically, the first layer is the input and the last layer is the output.
- Recurrent networks
  - These have directed cycles in their connection graph. They can have complicated dynamics.
  - More biologically realistic.





Recurrent networks

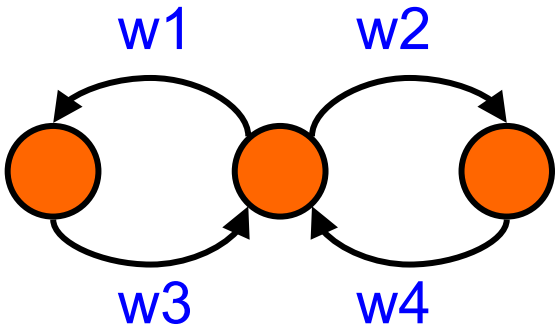
- A ***feed-forward neural*** net just computes a fixed sequence of non-linear learned transformations to convert an input pattern into an output pattern.
- ***If the connectivity has directed cycles***, the network can do much more:
  - It can ***oscillate***. This is useful for generating cycles needed 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. (-> weather)
    - But its a bad idea for most information processing.

More uses of recurrent networks

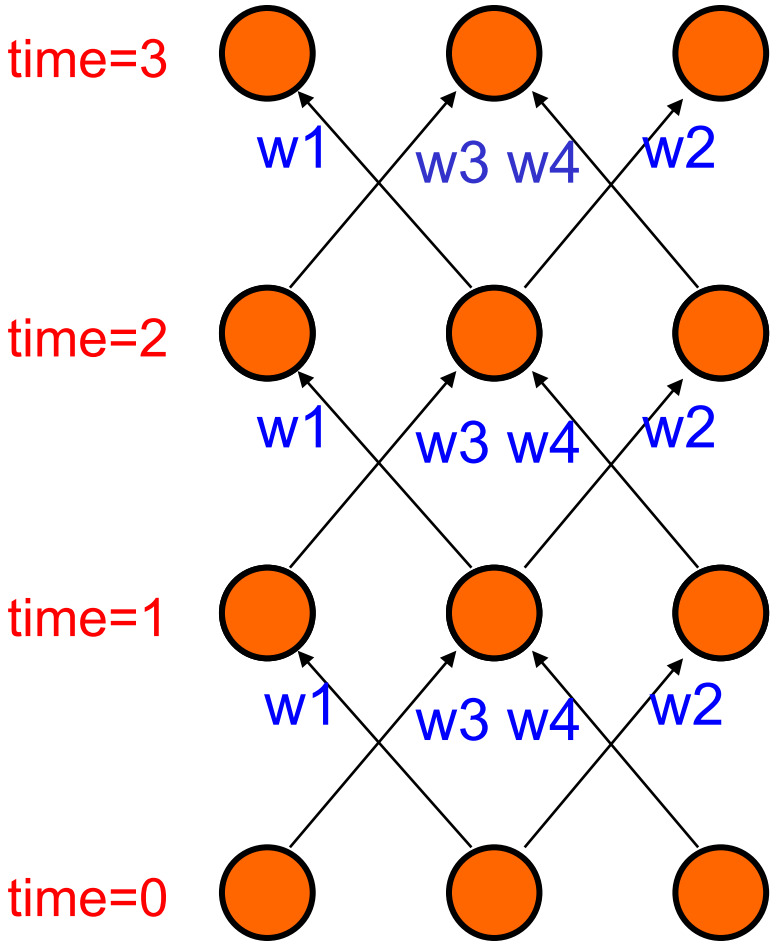
- They can ***remember things***.
  - The network has ***internal state***. It can decide to ignore the input for a while if it wants to.
  - But it is very hard to train a recurrent net for long term dependencies; better for ***local context***

- They can ***model sequential data*** in a much more natural way than by using a fixed number of previous inputs
  - The hidden state of a recurrent net can carry along information about a ***potentially unbounded number of previous inputs***

The relationship between layered, feedforward nets and recurrent nets



- Assume that there is a time delay of 1 in using each connection.
- The recurrent net as 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.

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

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

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

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

Initialization

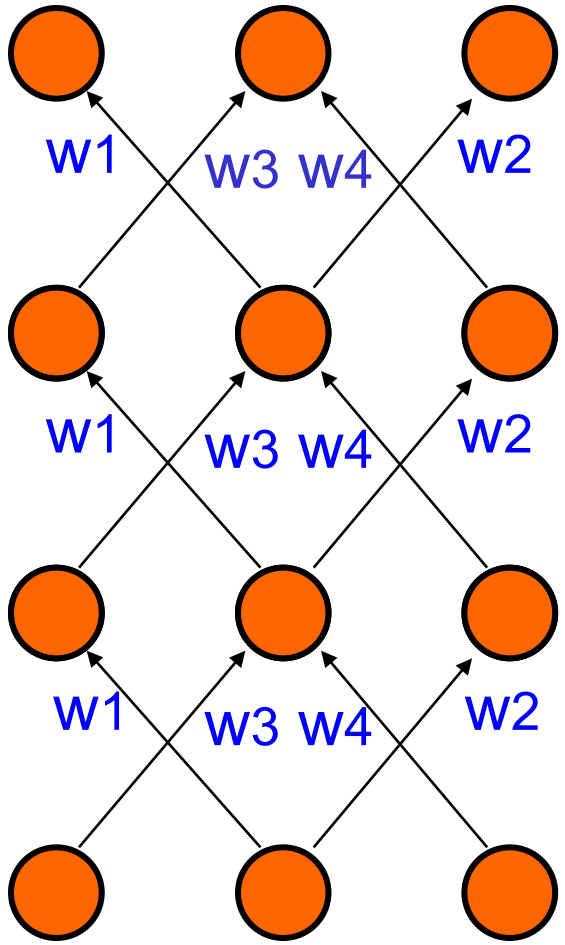
- We need to specify the ***initial activity state*** of all the hidden and output units
  - We could just fix these initial states to have some ***default value*** like 0.5
  - ***better to treat the initial states as learning parameters***

- 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

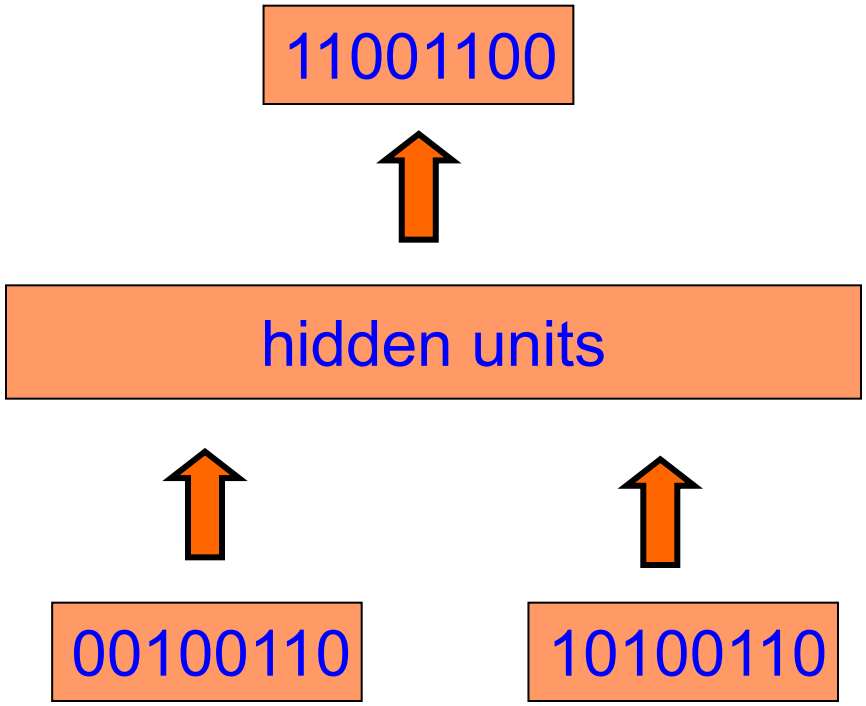
We can specify targets in several ways:

- 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”

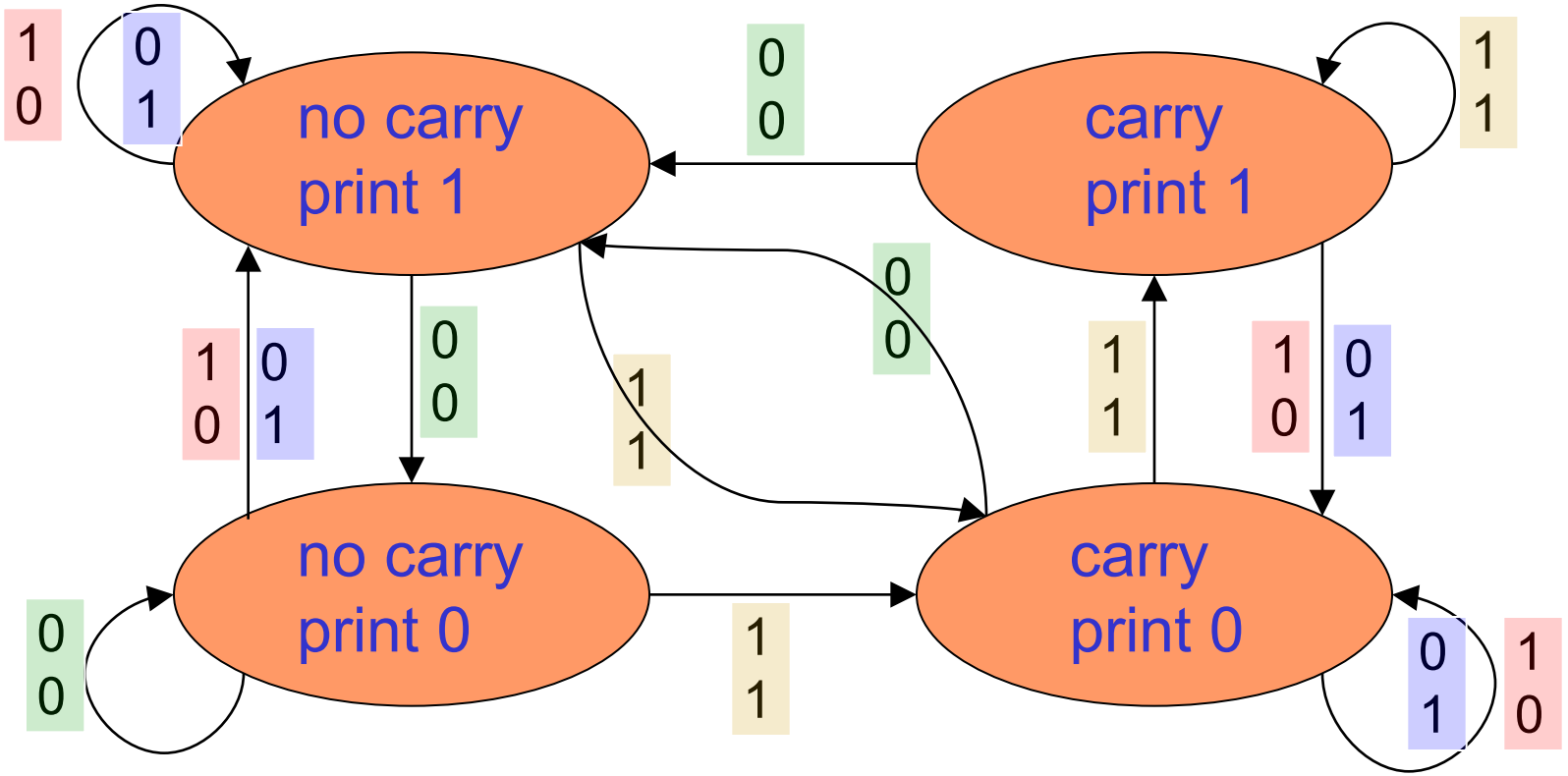


A good problem for a recurrent network

- 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



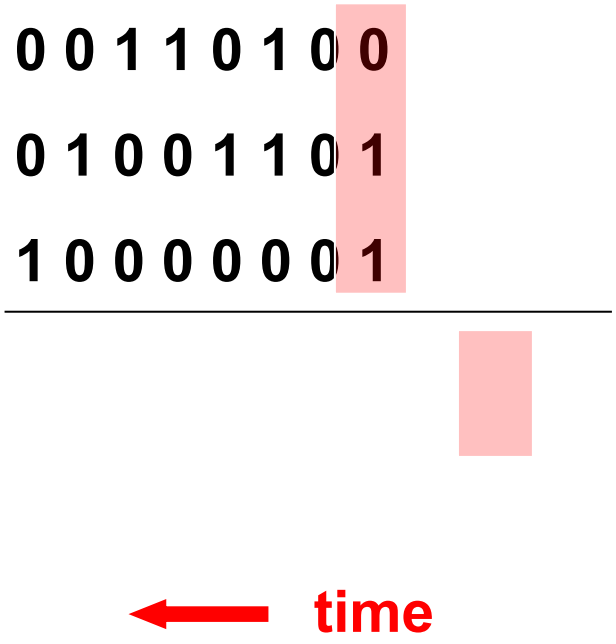
The algorithm for binary addition



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.

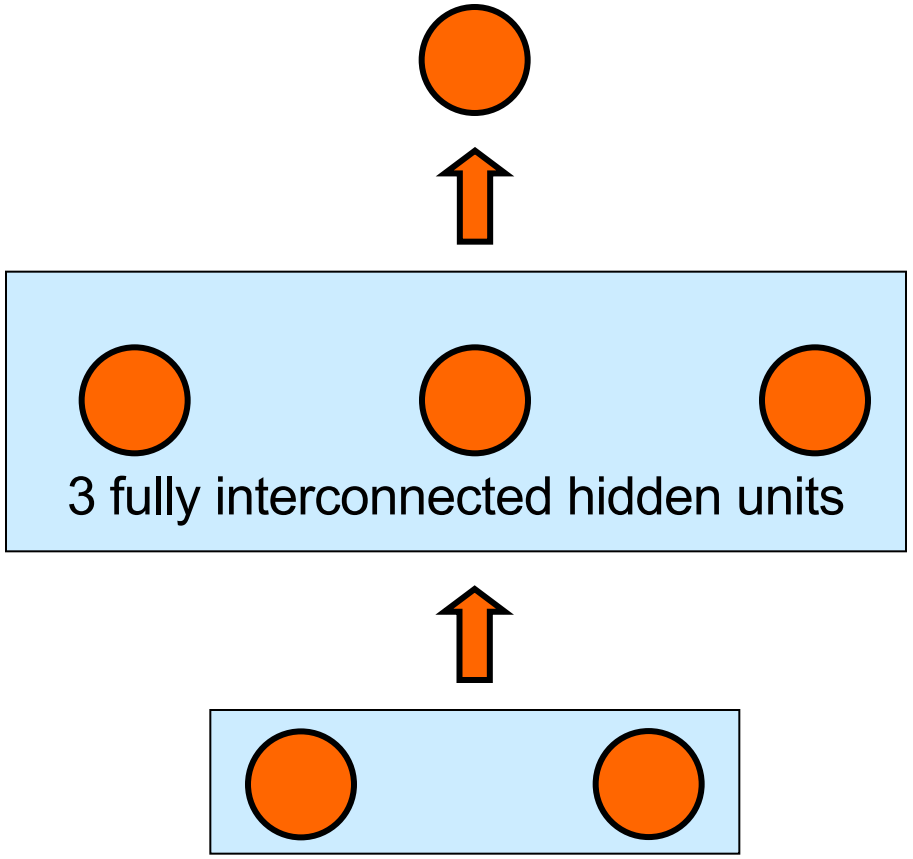
A recurrent net for binary addition

- The network has two input units and one output unit 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 produce the output.



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.



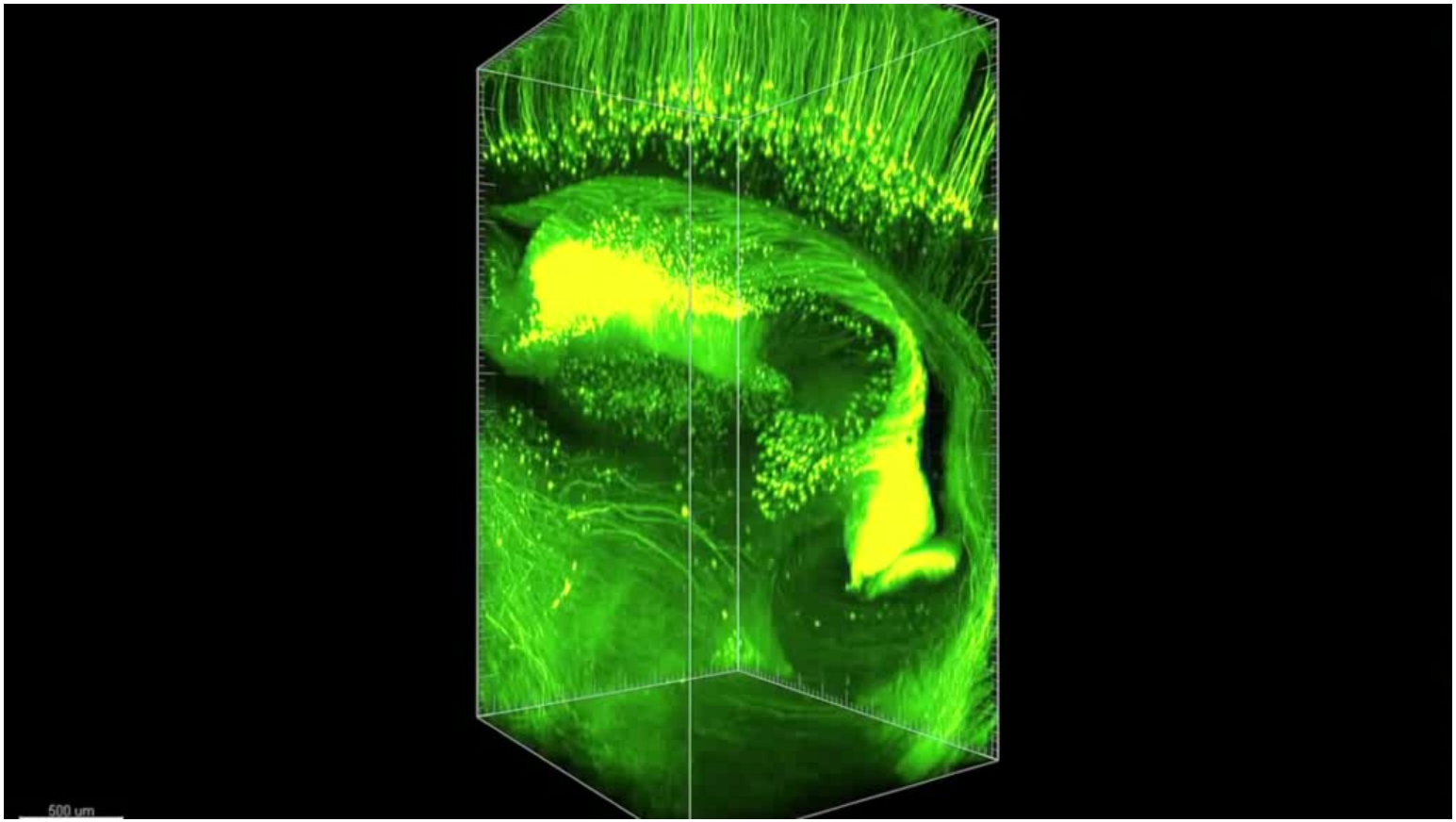
What the network learns

- It learns ***four distinct patterns of activity*** for the 3 hidden units. These patterns correspond to the nodes in the finite state automaton.
  - 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.

Representations of neurons in the brain

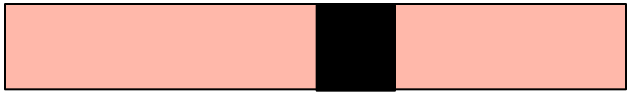
Neurons in an intact mouse hippocampus visualized using CLARITY and fluorescent labelling



Shen, H. See-through brains clarify connections. Nature, vol. 496, pp. 151, Macmillan Publishers Limited, 11 April 2013. Video online

Localist representations

- The simplest way to represent things with neural networks is to dedicate ***one neuron to each concept/feature***.
  - Easy to understand.
  - Easy to code by hand
    - Often used to represent inputs to a net
  - Easy to learn
  - Easy to associate with other representations or responses.



- But localist models are inefficient whenever the data has ***componential*** structure.

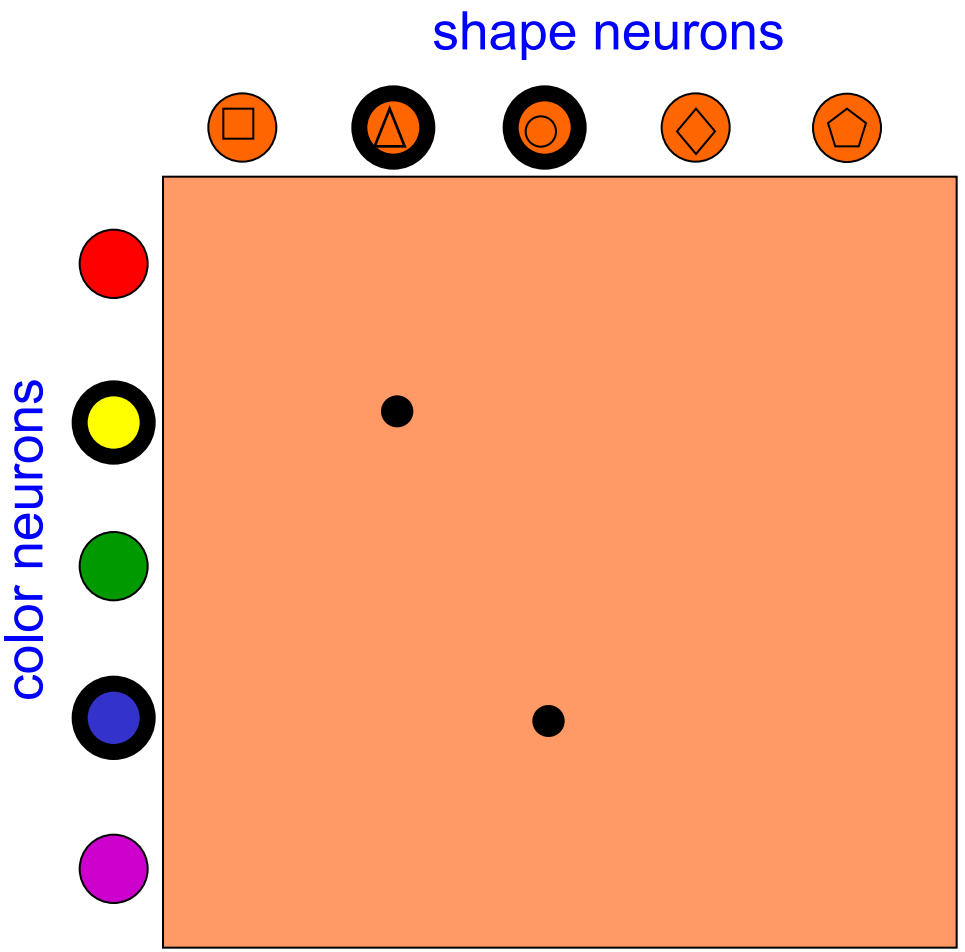
Examples of componential structure

- Consider a ***visual scene***
  - It contains many different objects
  - Each object has many properties like shape, color, size, motion.
  - Objects have spatial relationships to each other.

- Big, yellow, Volkswagen
  - Do we have a neuron for this combination?
    - Is the BYV neuron set aside in advance?
    - Is it created on the fly?
    - How is it related to the neurons for big and yellow and Volkswagen?

Using simultaneity to bind things together

- Represent conjunctions by activating all the constituents at the same time.
  - This does not require connections between the constituents.
  - But what if we want to represent yellow triangle and blue circle at the same time?
- ***Binding problem***: yellow circle? Blue triangle?

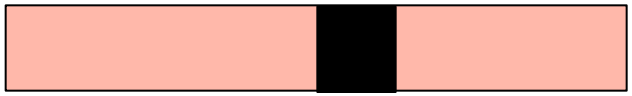


Using space to bind things together

- Conventional computers can bind things together by putting them into neighboring memory locations.
  - This works nicely in vision. Surfaces are generally opaque, so we only get to see one thing at each location in the visual field.
    - If we use topographic maps for different properties, we can assume that properties at the same location belong to the same thing.

The definition of “distributed representation”

- If each neuron represents one concept, this must be a ***localist representation***.



- “Distributed representation” means a ***many-to-many*** relationship between two types of representation (such as concepts and neurons).
  - Each concept is represented by many neurons .
  - Each neuron participates in the representation of many concepts.

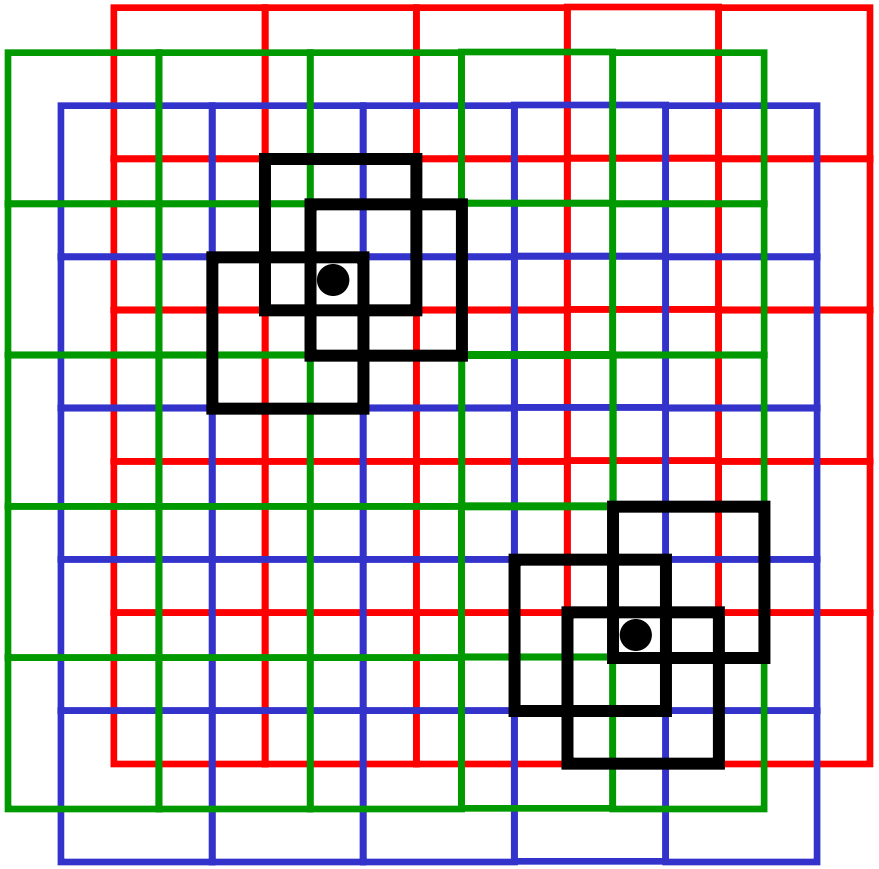
Coarse coding

- Using one neuron per entity is inefficient.
  - An efficient code would have each neuron active half the time.
    - This might be inefficient for other purposes (like associating responses with representations).

- Can we get accurate representations by using lots of inaccurate neurons?
  - If we can it would be very ***robust*** against hardware failure.

Coarse coding

- Use three ***overlapping arrays*** of large cells to get an array of fine cells
  - If a point falls in a fine cell, code it by activating 3 coarse cells.
- This is ***more efficient*** than using a neuron for each fine cell.
  - It loses by needing 3 arrays
  - It wins by a factor of 3x3 per array
  - Overall it wins by a factor of 3



How efficient is coarse coding?

- The efficiency depends on the dimensionality
  - In one dimension coarse coding does not help.
  - In 2-D the saving in neurons is proportional to the ratio of the fine radius to the coarse radius.
  - In k dimensions, by increasing the radius by a factor of r we can keep the same accuracy as with fine fields and get a saving of:

$$saving = \frac{\#\ fine\ neurons}{\#\ coarse\ neurons} = r^{k-1}$$

Applying backpropagation to shape recognition

- People are very good at recognizing shapes
  - It's intrinsically difficult and computers are bad at it. Why?

- Some reasons why it is difficult:
  - ***Segmentation***: Real scenes are cluttered.
  - ***Invariances***: We are very good at ignoring all sorts of variations that do not affect the shape.
  - ***Deformations***: Natural shape classes allow variations (faces, letters, chairs).
  - A huge amount of computation is required.

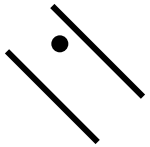
The invariance problem

- Our perceptual systems are very good at dealing with ***invariances***
  - ***translation, rotation, scaling***
  - ***deformation, contrast, lighting***

- We are so good at this that its hard to appreciate how difficult it is.
  - Its one of the main difficulties in making computers perceive.
  - We still do not have generally accepted solutions.

The invariant feature approach

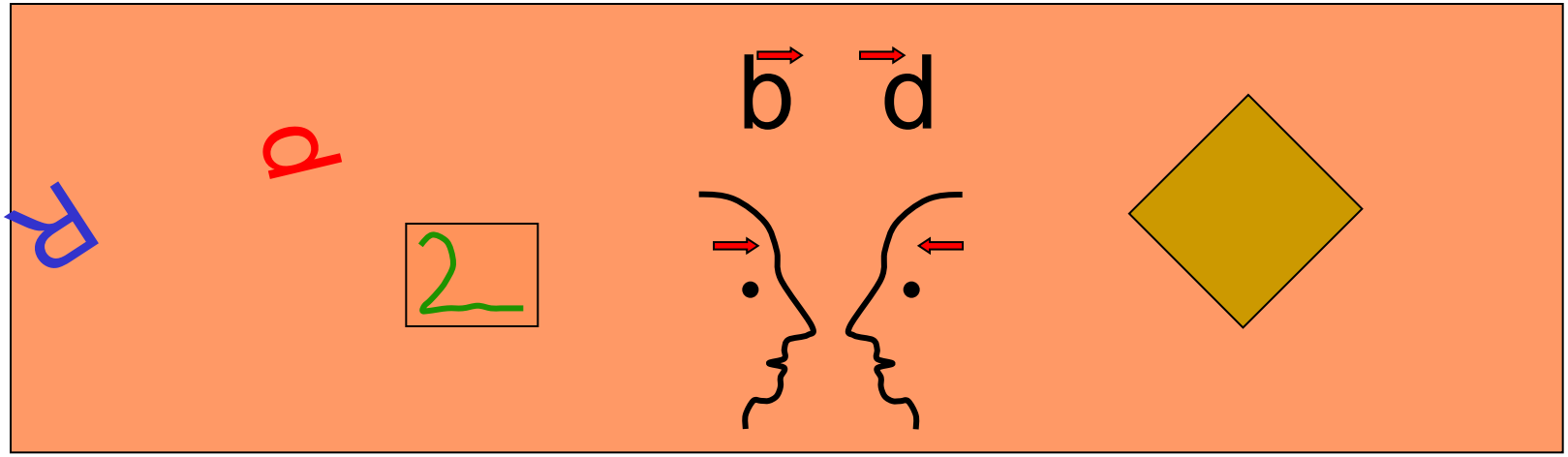
- Extract a large, redundant set of features that are invariant under transformations
  - e.g. “pair of parallel lines with a dot between them.



- With enough of these features, there is only one way to assemble them into an object.
  - we don’t need to represent the relationships between features directly because they are captured by other features.
- We must avoid forming features from parts of different objects!

The normalization approach

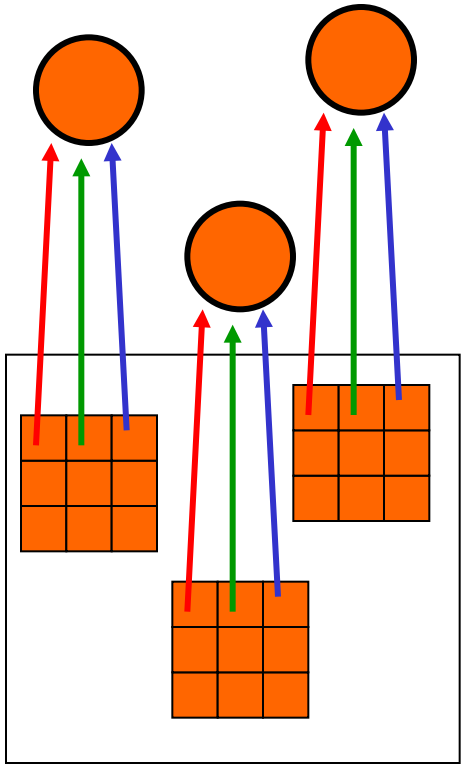
- Do preprocessing to ***normalize*** the data
  - e. g. put a box around an object and represent the locations of its pieces relative to this box.
  - Eliminates as many ***degrees of freedom*** as the box has.
    - translation, rotation, scale, elongation
  - But its not always easy to choose the box



The replicated feature approach

- Use many different copies of the same feature detector.
  - The copies all have slightly different positions.
  - Could also replicate across scale and orientation.
    - Tricky and expensive
  - Replication reduces the number of free parameters to be learned.
- Use several different feature types, each with its own replicated pool of detectors.
  - Allows each patch of image to be represented in several ways.

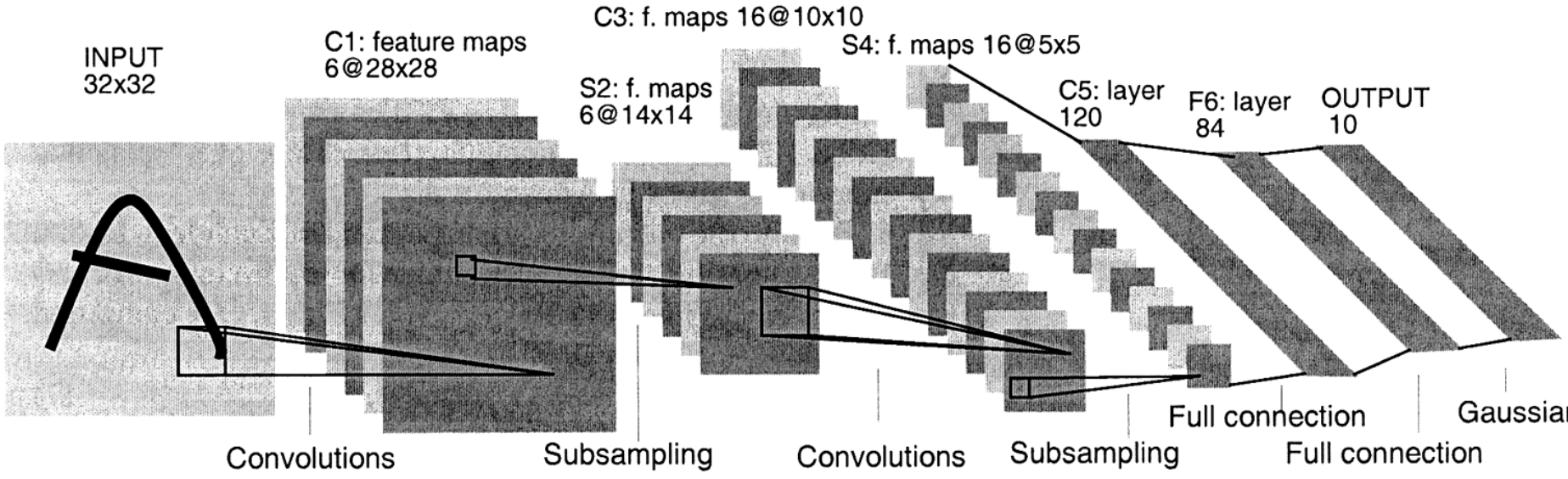
The red connections all have the same weight.



Le Net: domain knowledge into a network

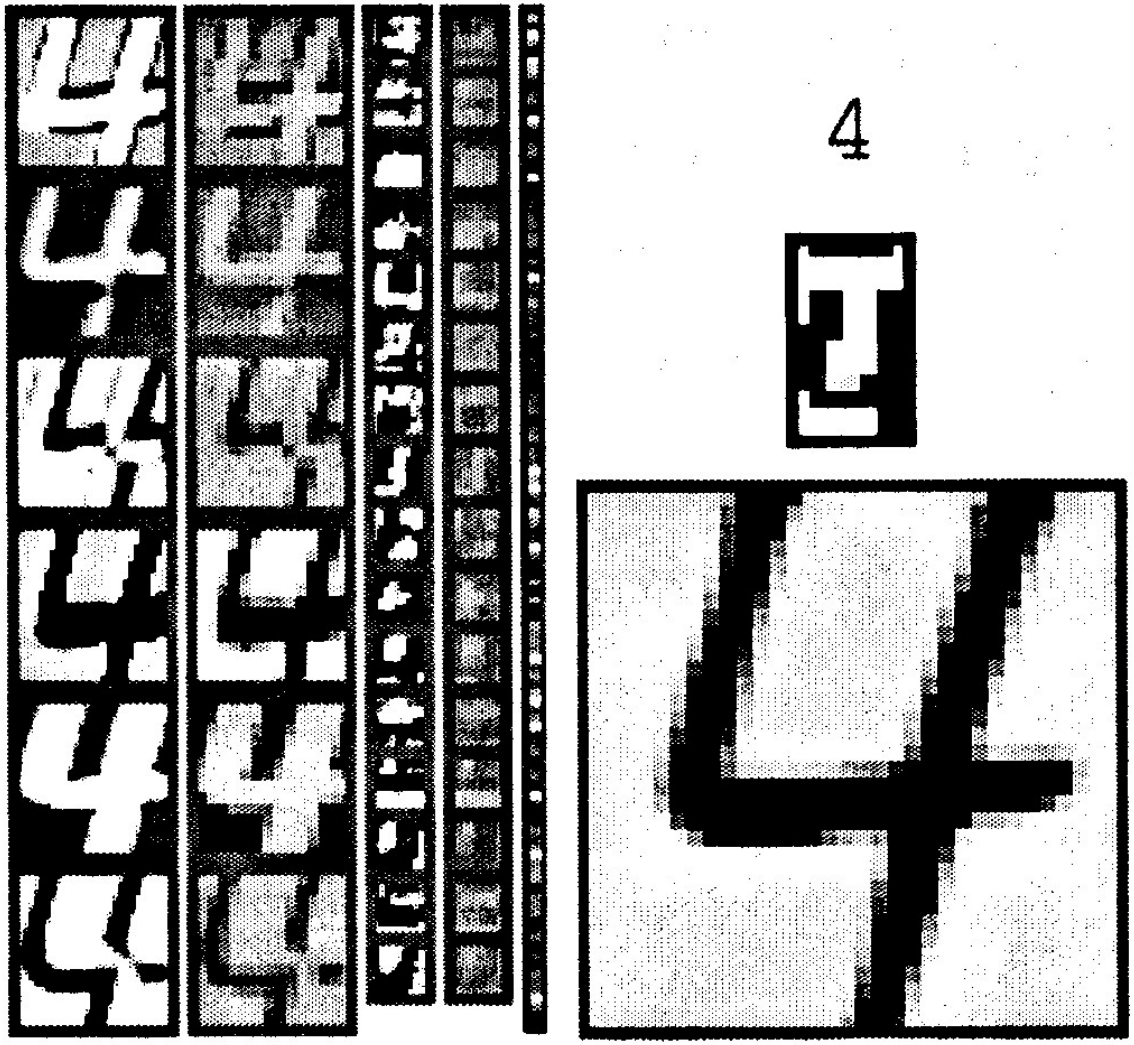
- Yann LeCun and others developed a really good recognizer for handwritten digits by using backpropagation in a feedforward net with:
  - Many hidden layers.
  - Many pools of replicated units in each layer.
  - Averaging of the outputs of nearby replicated units.
  - A wide net that can cope with several characters at once even if they overlap.

The architecture of LeNet5

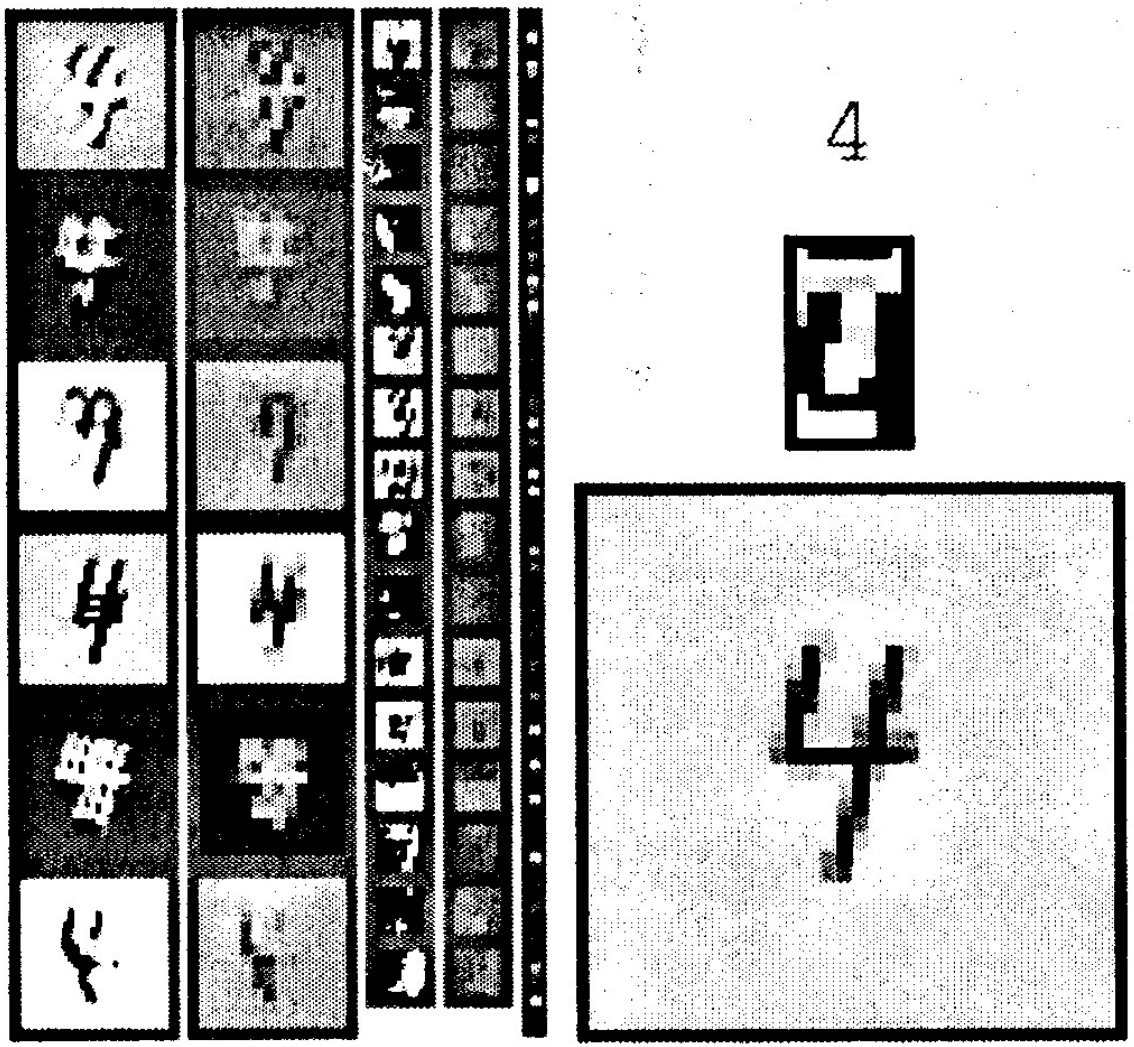


- ***Convolutions***: e.g. at each input location six different types of features are extracted by six units in identical locations in the six feature maps; All units in one feature map at 1st hid. layer share the same 5x5 weights
- ***Subsampling***: performs a local averaging reducing the resolution of the feature map and the sensitivity of the output to shifts and distortions, e.g. by taking the average of a 2x2 area

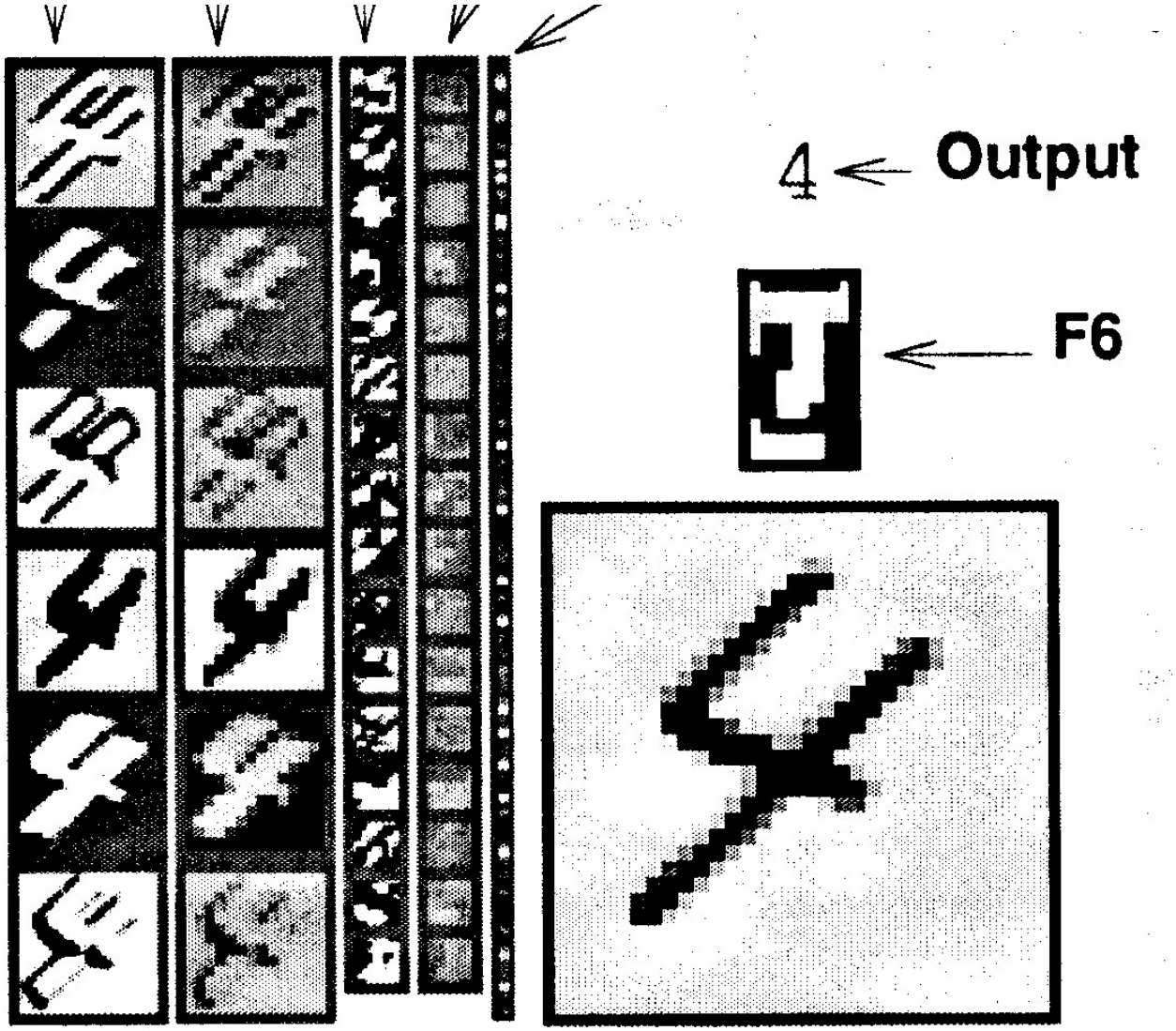
The architecture of LeNet5 (cont.)



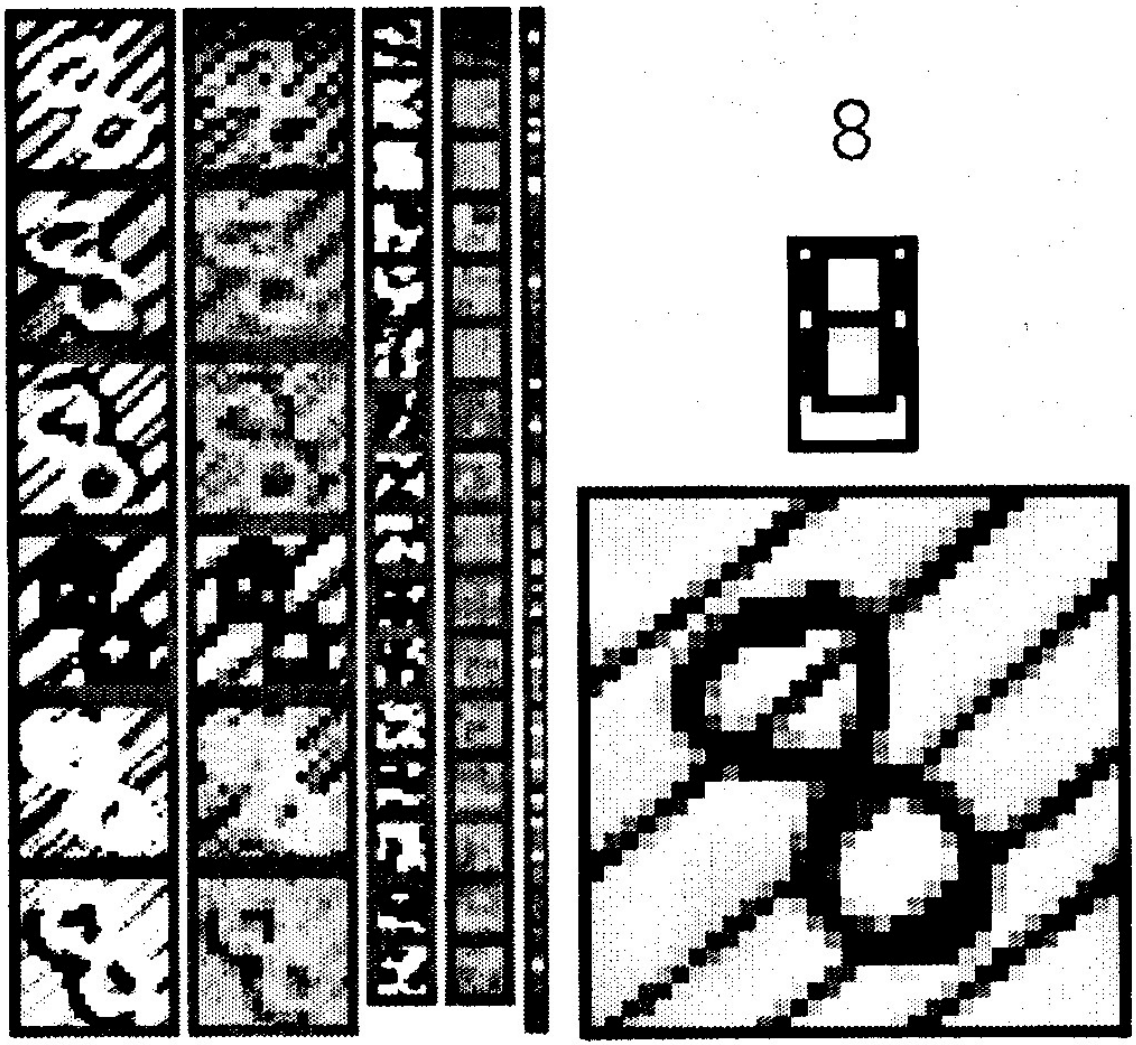
The architecture of LeNet5 (cont.)



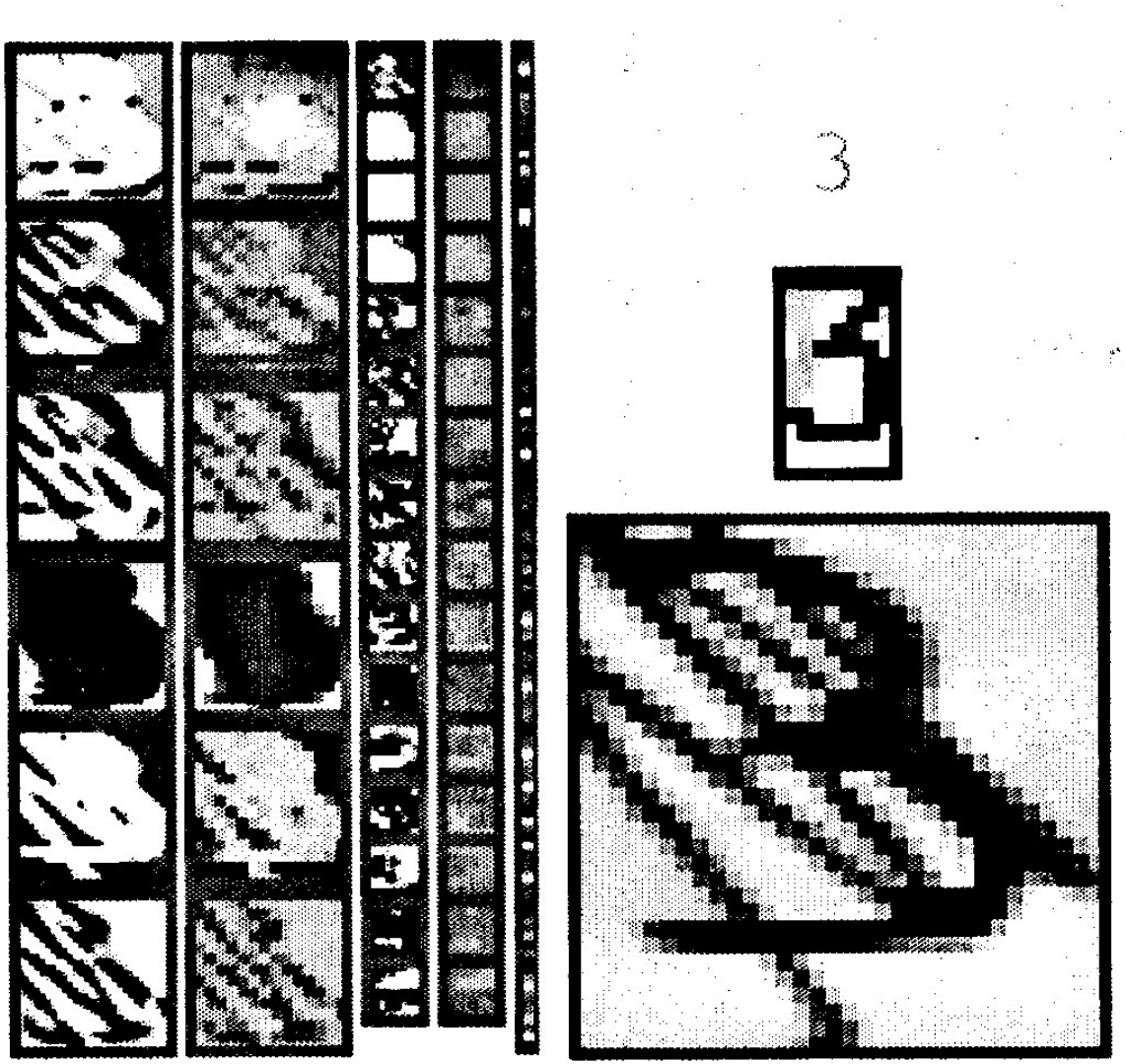
The architecture of LeNet5 (cont.)



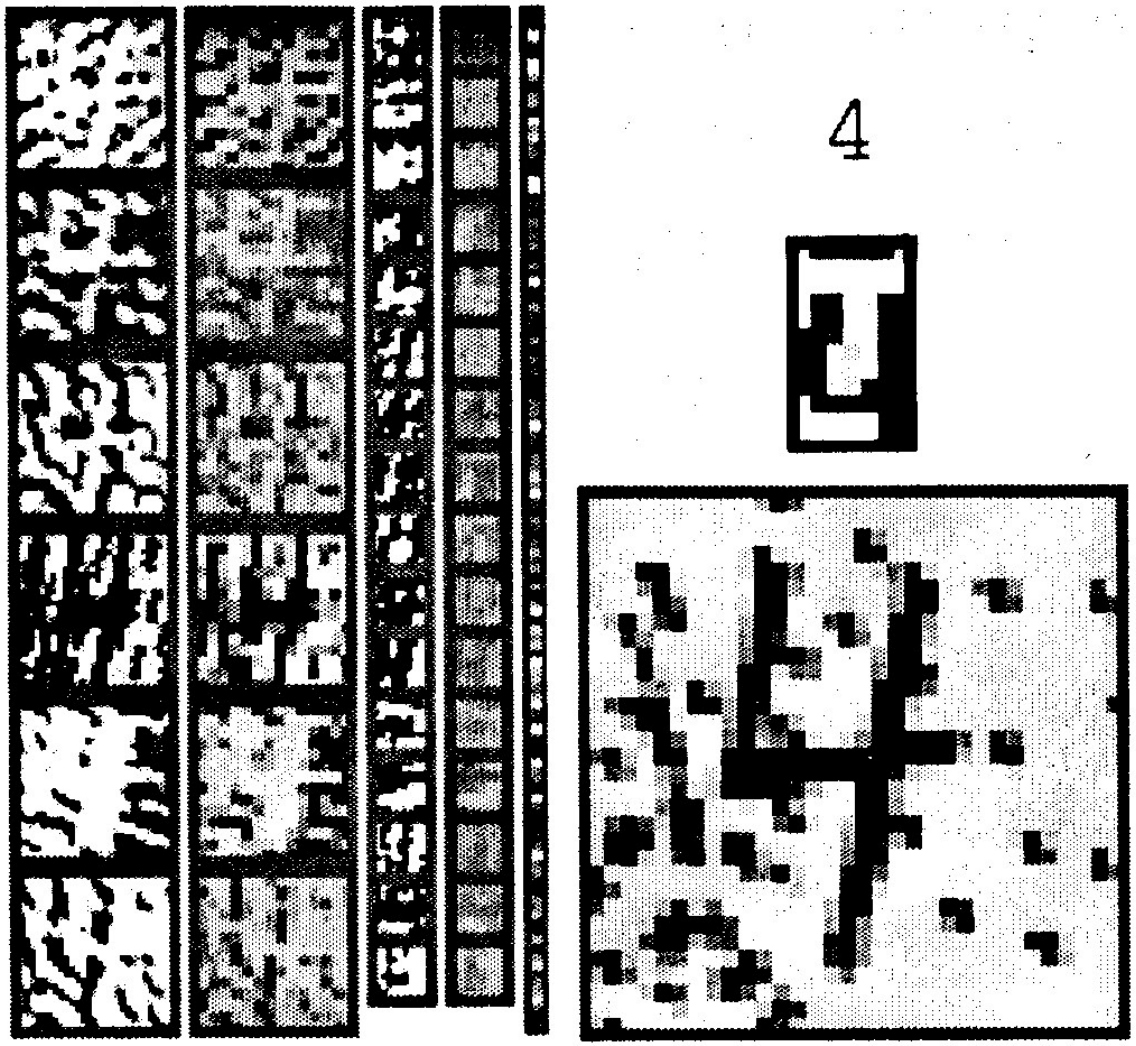
The architecture of LeNet5 (cont.)



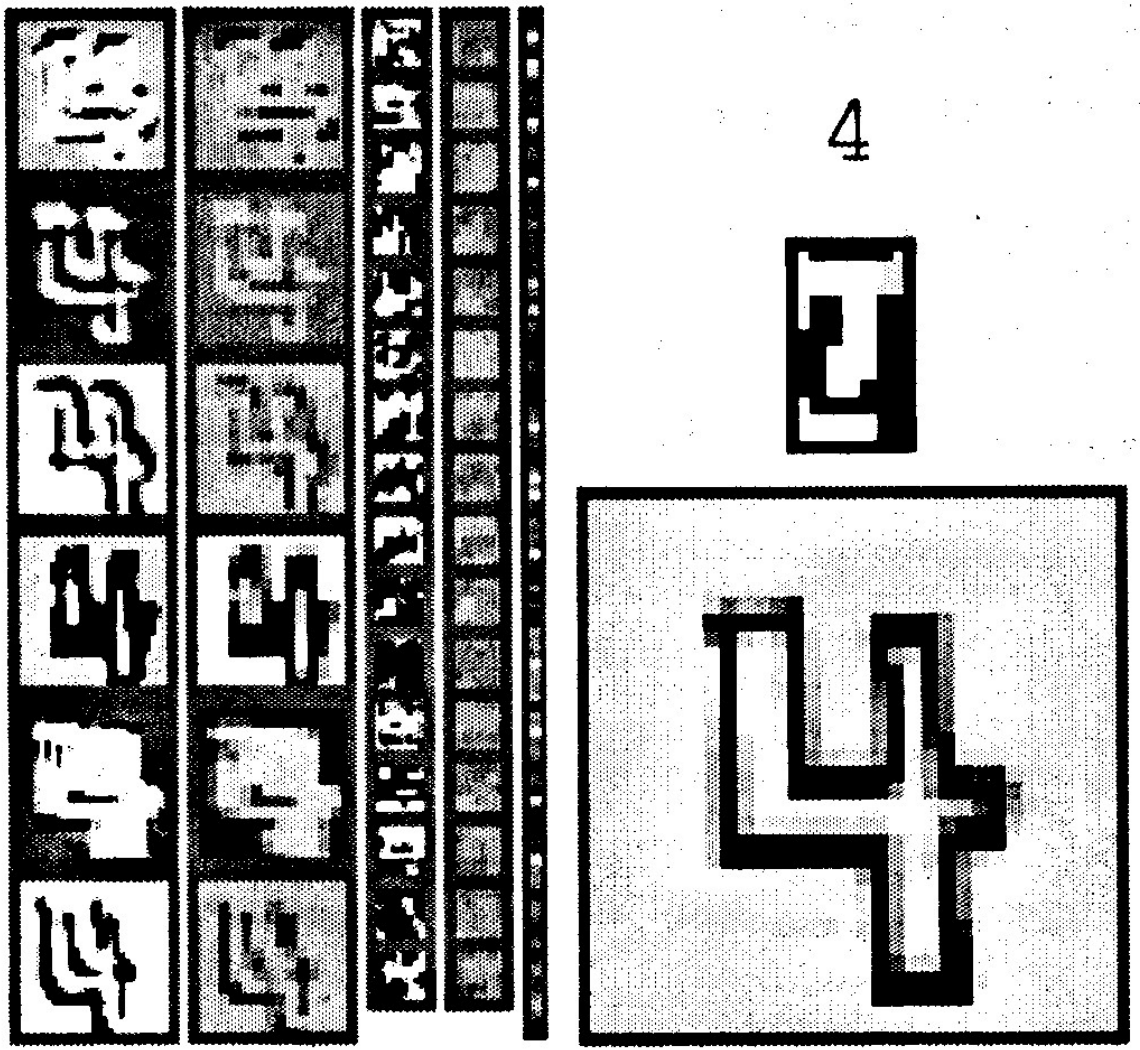
The architecture of LeNet5 (cont.)



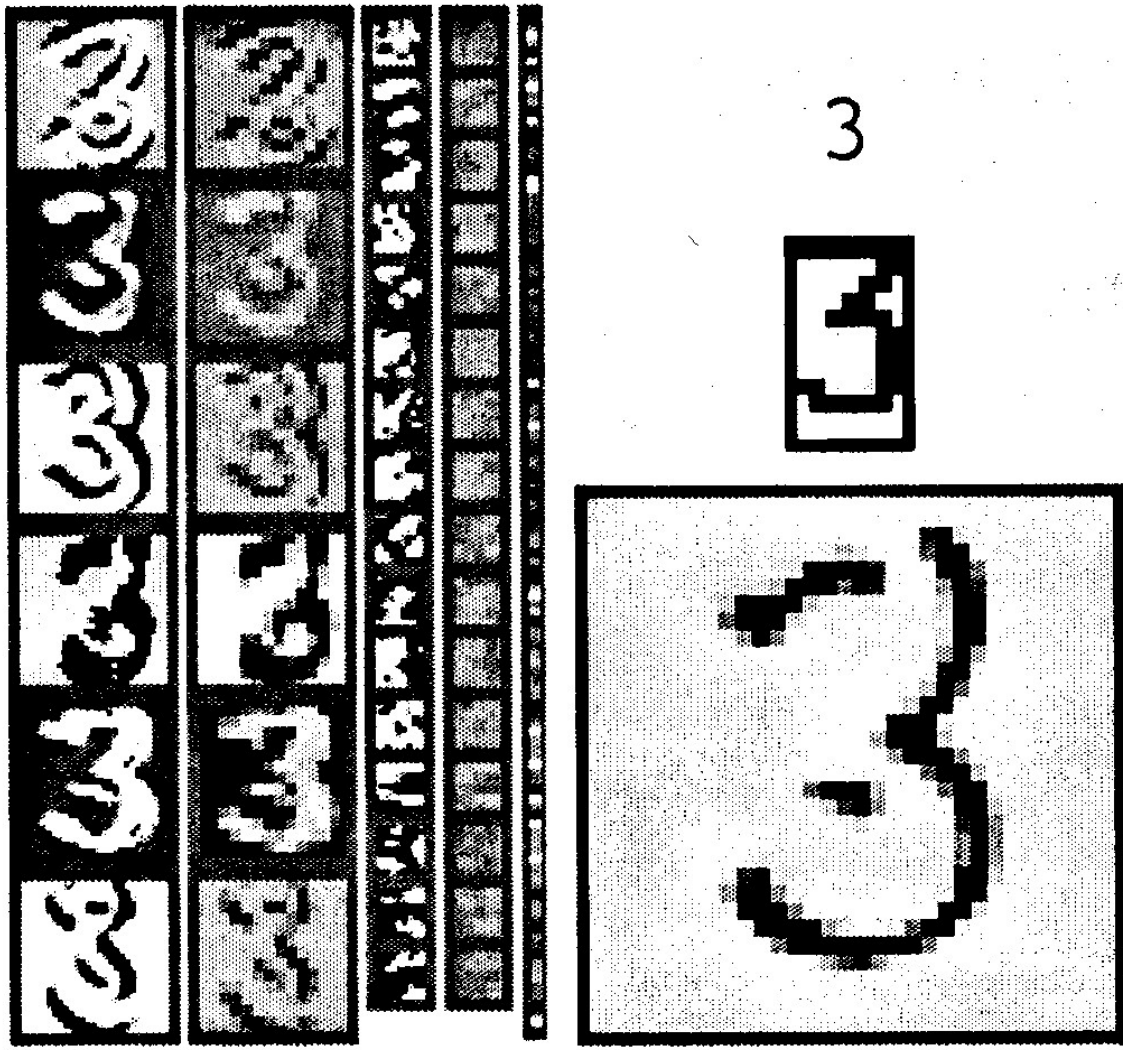
The architecture of LeNet5 (cont.)



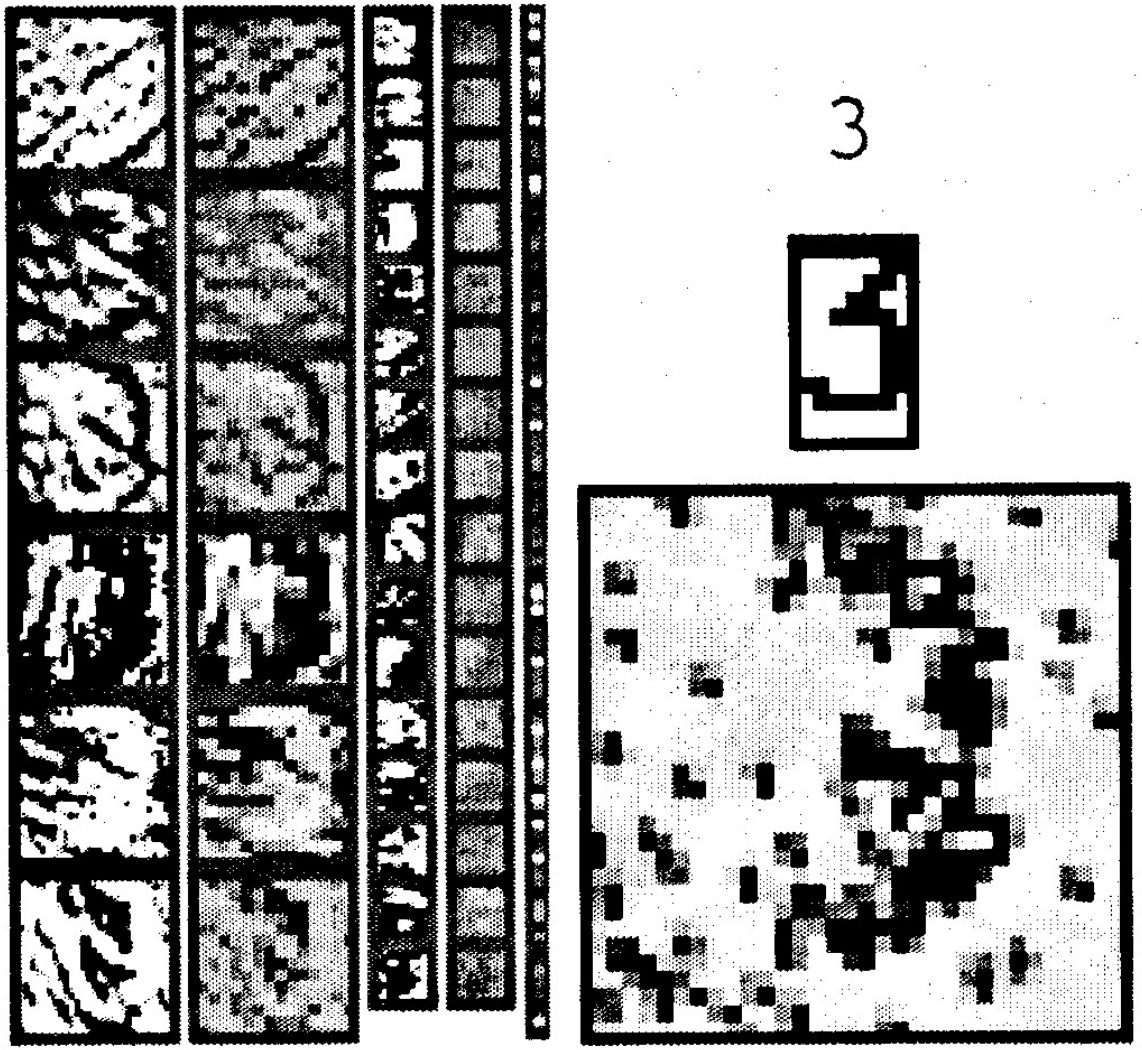
The architecture of LeNet5 (cont.)



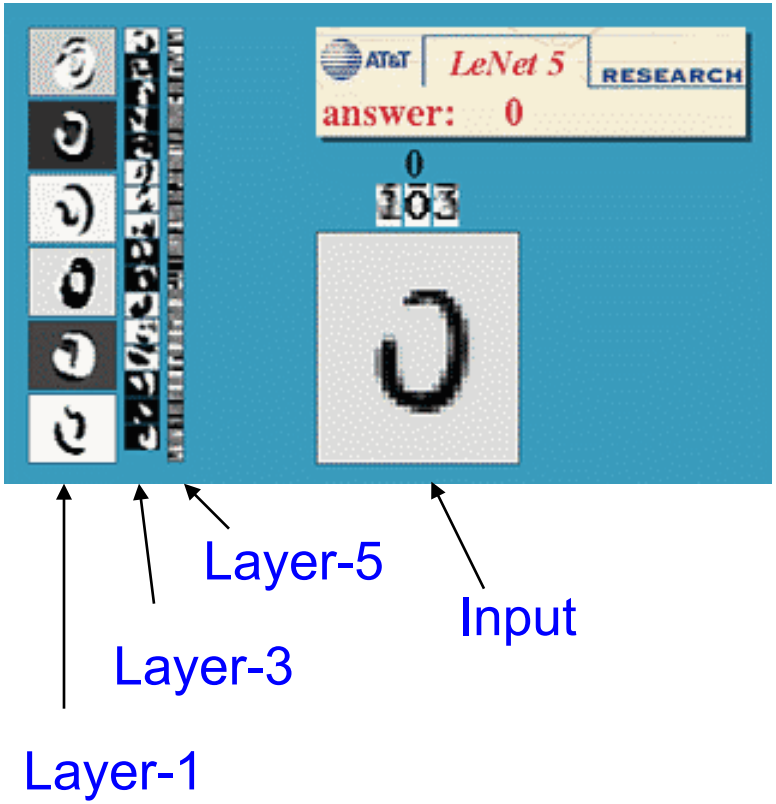
The architecture of LeNet5 (cont.)



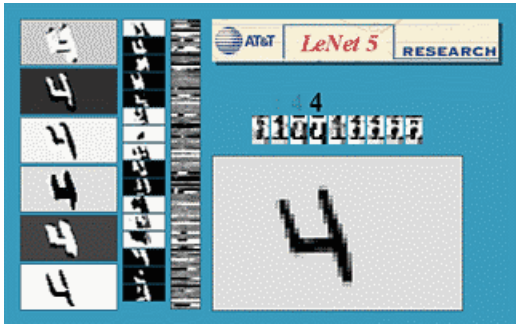
The architecture of LeNet5 (cont.)



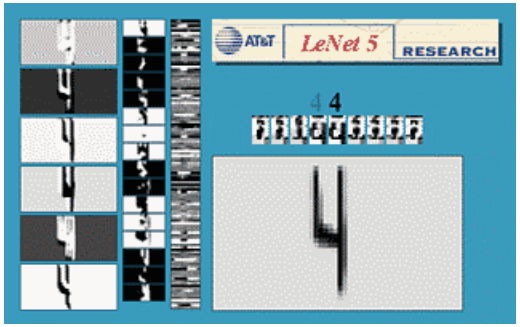
LeNet 5 in action



Rotation Invariance:



Squeezing Invariance:



Scale Invariance:

Demos of LENET at
http://yann.lecun.com/exdb/lenet/index.html

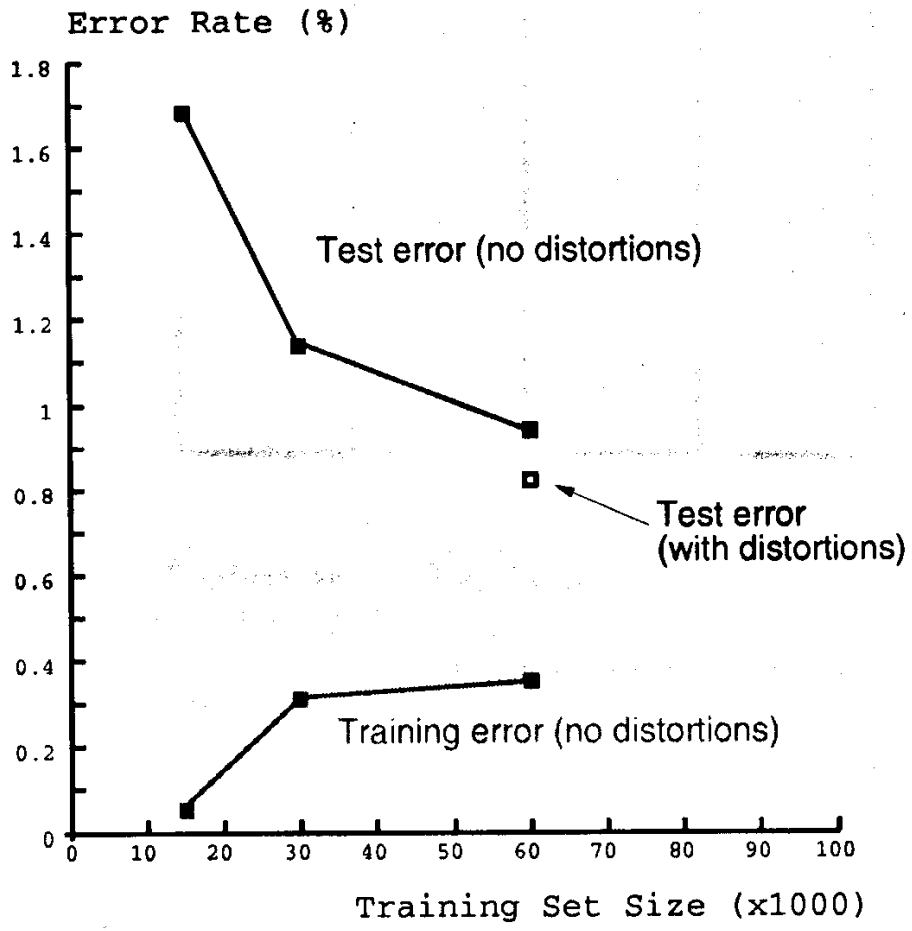
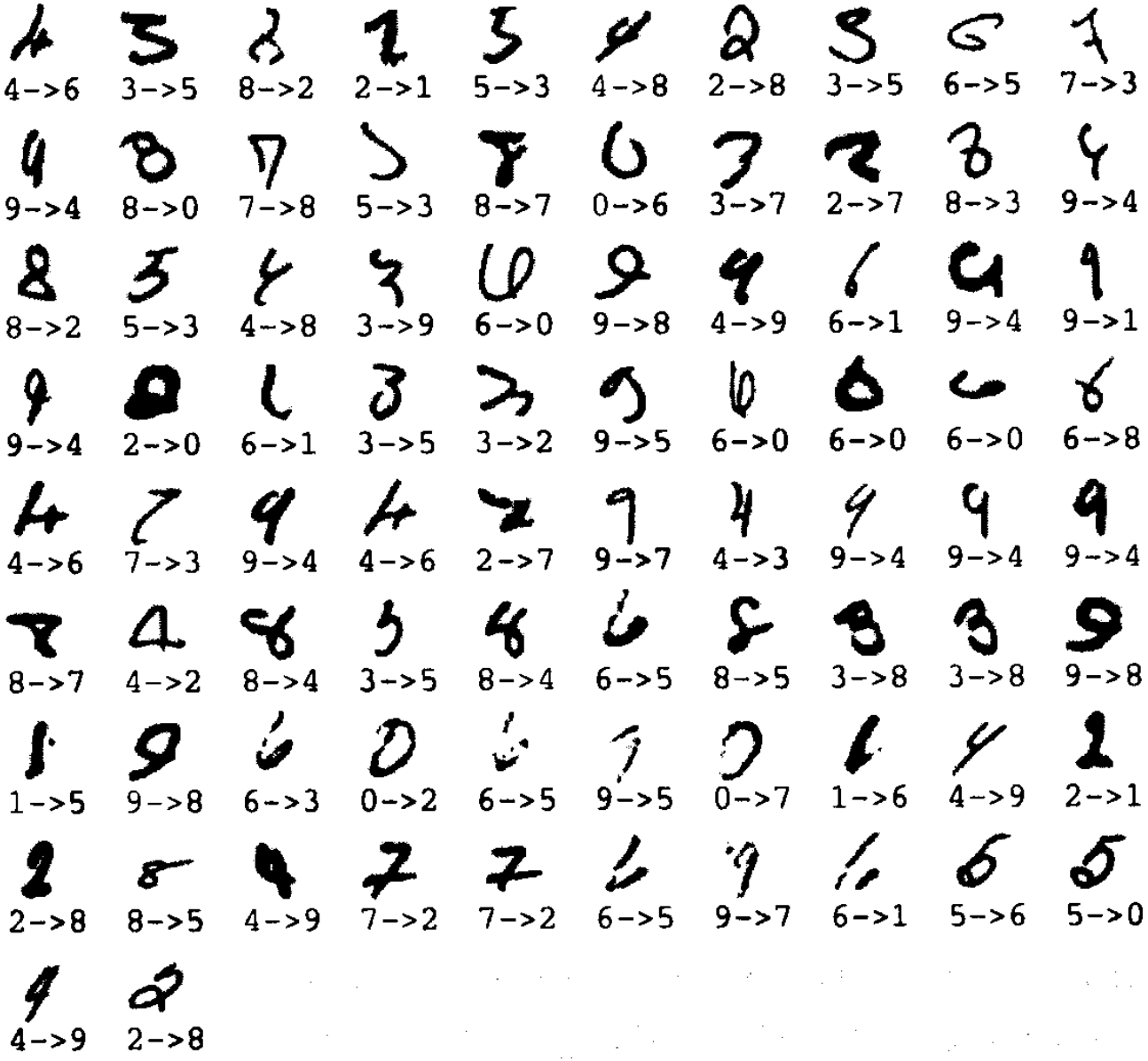


Fig. 6. Training and test errors of LeNet-5 achieved using training sets of various sizes. This graph suggests that a larger training set could improve the performance of LeNet-5. The hollow square show the test error when more training patterns are artificially generated using random distortions. The test patterns are not distorted.

The 82 errors made by LeNet5



Notice that most of the errors are cases that people find quite easy.

The human error rate is probably 20 to 30 errors

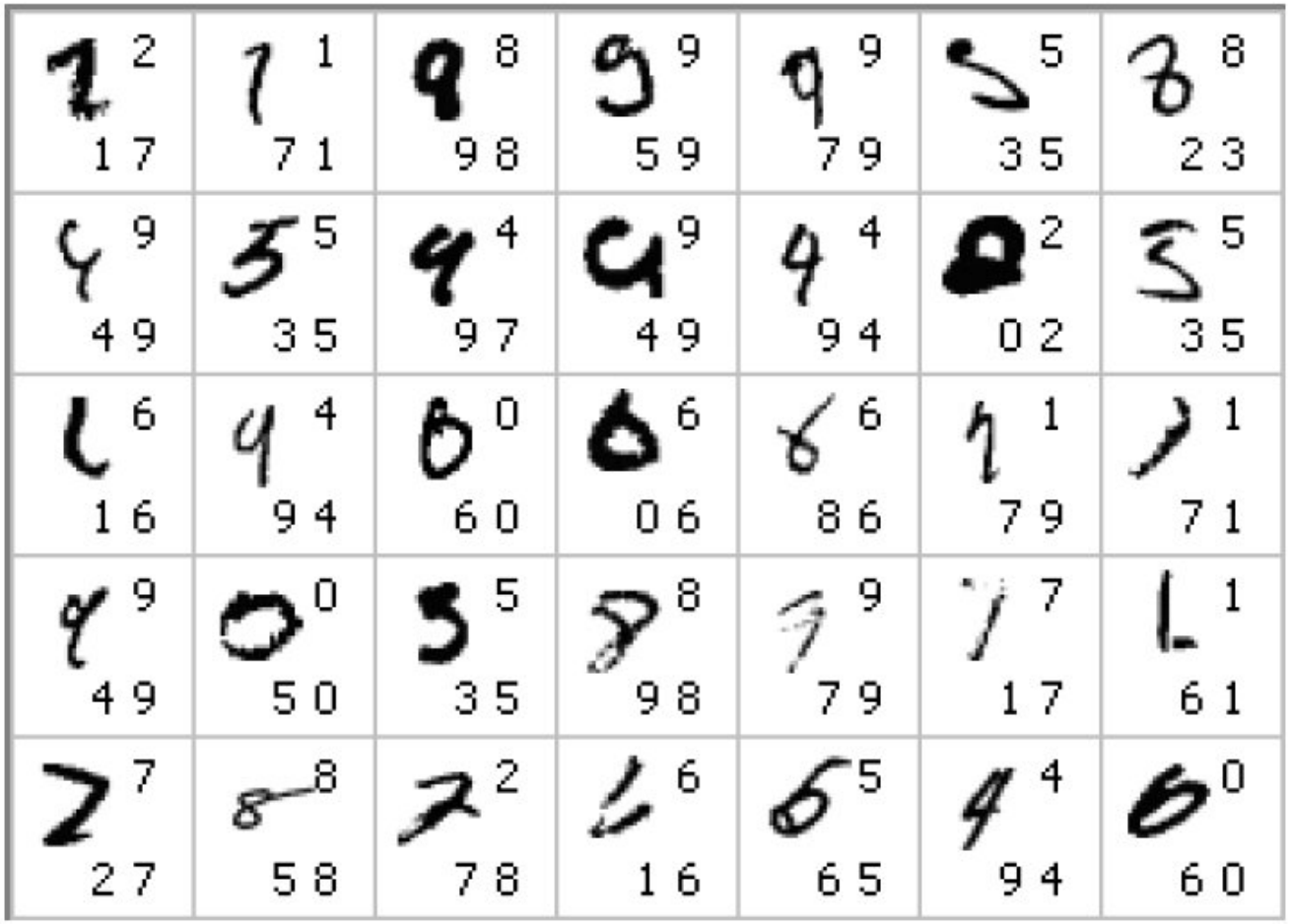
A brute force approach

- LeNet uses knowledge about the invariances to ***design***:
  - the network architecture
  - or the weight constraints
  - or the types of feature
- But its much simpler to incorporate knowledge of invariances by just creating extra training data:
  - For each training image, produce new training data by applying all of the transformations we want to be insensitive to (Le Net can benefit from this too).
  - Then train a large, dumb net on a fast computer.
  - This works surprisingly well if the transformations are not too big

Making simple backpropagation work really well for recognizing digits (Ciresan et. al. 2010)

- Using the standard viewing transformations plus local deformation fields to get LOTS of data.
- Use a large number of hidden layers with a large number of units per layer and no weight constraints.
- Use the appropriate error measure for multi-class categorization:
  - Softmax outputs with cross-entropy error.
- Train by using a big GPU board for a long time (at 5x10^9 weight updates per second)
  - This gets down to only 35 errors which is the record and is close to human performance.

The errors made by the big dumb net trained with lots of fancy transformations of the data

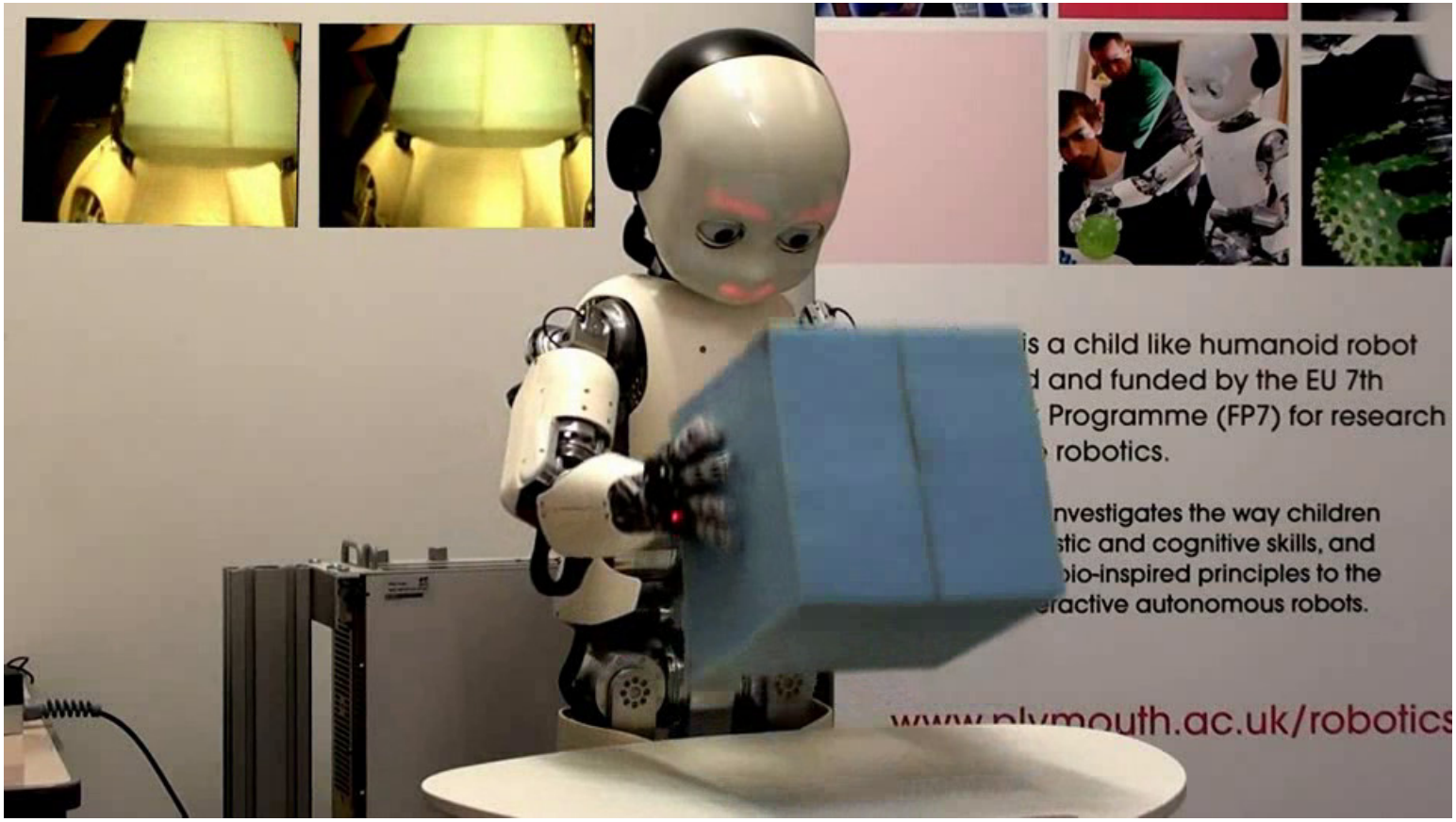


The top printed digit is the right answer. The bottom two printed digits are the network's best two guesses.

Priors and Prejudice

- We can put our prior knowledge about the task into the network by using weight-sharing, or carefully designing the connectivity, or carefully choosing the right types of unit.
  - But this prejudices the network in favor of a particular way of solving the problem. But may be efficient.
- Alternatively, we can use our prior knowledge to create a whole lot more training data.
  - This may require a lot of work and it is much less efficient in terms of the time required for learning.
  - But it does not prejudice the network in favor of a particular way of getting the right answer.

Recurrent neural network for movements on an ICub humanoid robot



http://www.italkproject.com