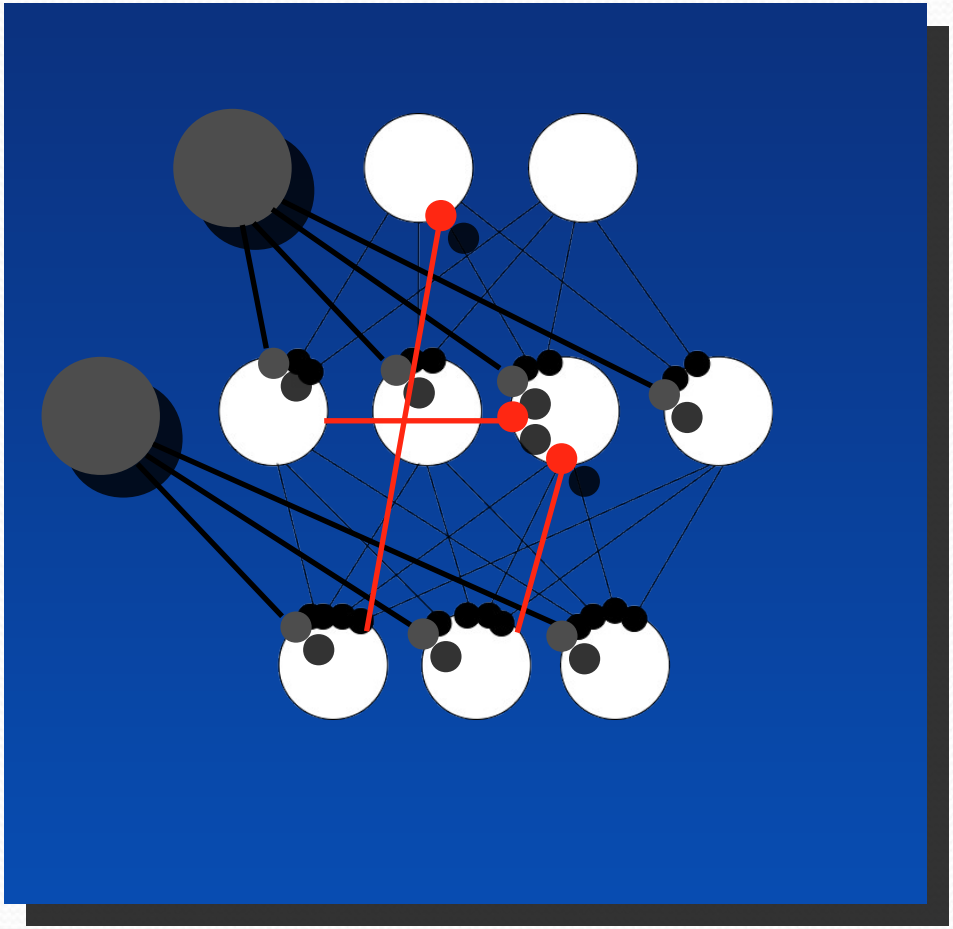
Neural Networks

- **Recurrent networks**
- **Boltzmann networks**
- **Deep belief networks**

Partially based on a tutorial by Marcus Frean

Recurrent Networks

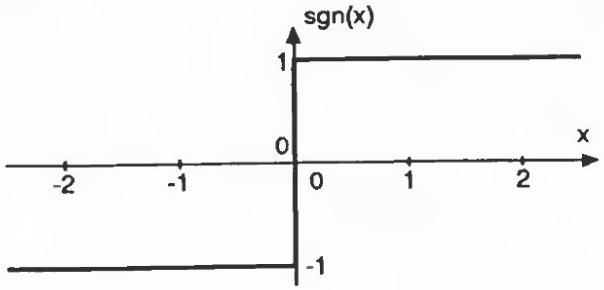
- The output of any neuron can be the input of any other



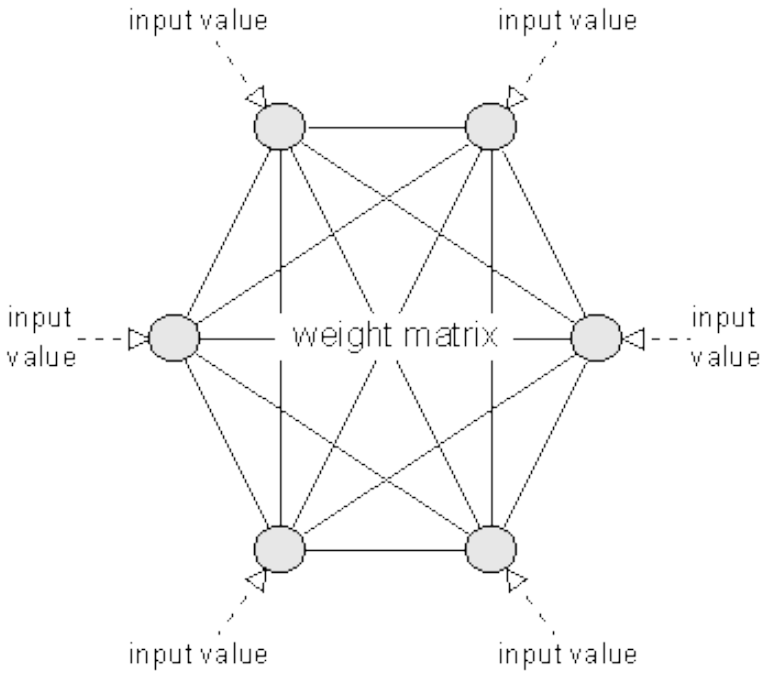
Hopfield Network

- **ALL** nodes are input & output

Activation function:



Input = activation: {-1,1}

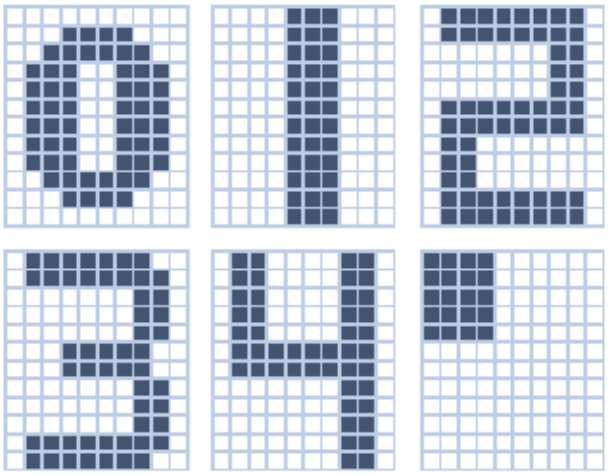


Recurrent Networks: Input Processing

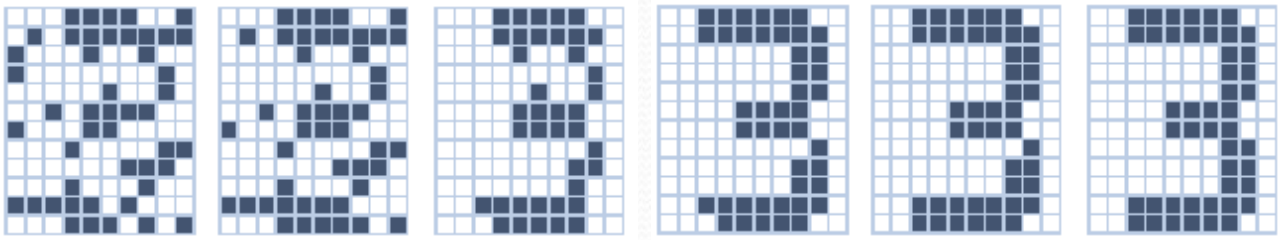
- Given an input $\vec{x}$
- Asynchronously: (Common)
  - **Step 1:** sample an arbitrary unit
  - **Step 2:** update its activation
  - **Step 3:** if activation does not change, stop, otherwise repeat
- Synchronously:
  - **Step 1:** save all current activations (time *t*)
  - **Step 2:** recompute activation for all units a time *t+1* using activations at time *t*
  - **Step 3:** if activation does not change, stop, otherwise repeat

Hopfield Network: Associative Memory

- Patterns "stored" in the network:



- Retrieval task: for given input, find the input that is closest:



Recurrent Networks

- Other choices as compared to Hopfield networks:
  - not all units input
  - not all units output
  - different activation functions
  - different procedures for processing input and reaching a stable point
  - non-binary input and output
  - different learning algorithms

Non-Binary Hopfield Networks

- Assume continuous activation functions, $f(x) = (1 / (1 + e^{-x}))$
- Till now:
  - Asynchronous updates
  - Synchronous updates
- Third option: **Continuous updates**
  - move the activation synchronously in the desired direction: (V_i is current activation of unit i)

$$\frac{dV_i}{dt} = \eta \left(-V_i + f(\sum_j w_{ij} V_j) \right)$$

Non-Binary Hopfield Networks

- Third option: **Continuous updates**
  - move the activation synchronously in the desired direction: (V_i is current activation of unit *i*)

$$\frac{dV_i}{dt} = \eta \left(-V_i + f(\sum_j w_{ij} V_j) \right)$$

  - **equivalent alternative:** maintain the **input** u_i for each unit, move the *input* in the desired direction

$$\frac{du_i}{dt} = \eta \left(-u_i + \sum_j w_{ij} V_j \right)$$

Non-Binary Hopfield Networks

- Do continuous updates lead to a stable state?
- Energy function:

$$H = -\frac{1}{2}\sum_{ij} w_{ij} V_i V_j + \sum_i \int_0^{V_i} f^{-1}(V)dV$$

Does energy decrease in update steps? Prove that:

$$\frac{dH}{dt} \leq 0$$

Non-Binary Hopfield Networks

$$\frac{dH}{dt} = -\frac{d\frac{1}{2}\sum_{ij} w_{ij} V_i V_j}{dt} + \sum_i \frac{d\int_0^{V_i} f^{-1}(V)dV}{dt}$$

- Product rule:

$$\frac{dH}{dt} = -\frac{1}{2}\sum_{ij} w_{ij}\frac{V_i}{dt}V_j - \frac{1}{2}\sum_{ij} w_{ij}\frac{V_j}{dt}V_i + \sum_i f^{-1}(V_i)\frac{dV_i}{dt}$$

- Rewrite: $u_i = f^{-1}(V_i)$ and use w_{ij} symmetric

$$\frac{dH}{dt} = -\sum_i \frac{dV_i}{dt}\left(\sum_j w_{ij} V_j - u_i\right)$$

Non-Binary Hopfield Networks

$$\frac{dH}{dt} = -\sum_i \frac{dV_i}{dt}\left(\sum_j w_{ij}V_j - u_i\right)$$

- Last term equals a unit input update in a time step

$$\frac{dH}{dt} = -\sum_i \frac{1}{\eta}\frac{dV_i}{dt}\frac{du_i}{dt} = -\sum_i \frac{1}{\eta}g'(u_i)\left(\frac{du_i}{dt}\right)^2 \leq 0$$

Only if weights are symmetric!!!

Recurrent Backpropagation

- Any unit can be input/output; define error for each node:

$$E_k = \begin{cases} \xi_k - V_k & \text{if } k \text{ is an output unit} \\ 0 & \text{otherwise} \end{cases}$$

where ξ_k is expected output, V_k current output

(After continuous updates)

- Define error as $E = \frac{1}{2} \sum_k E_k^2$

- Essentially, we determine an update $\Delta w_{pq} = -\eta \frac{\partial E}{\partial w_{pq}}$

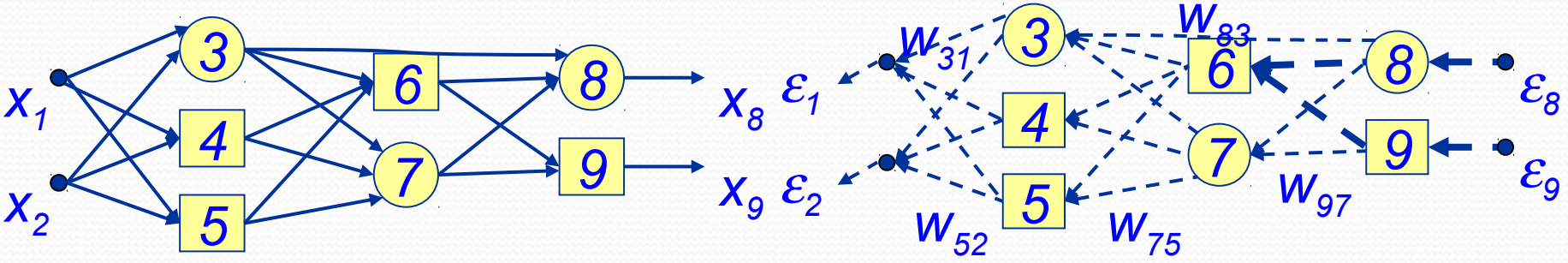
Recurrent Backpropagation

- Remember: "traditional" backpropagation:

$$\Delta_i = E_i g'(u_i) \qquad \text{(Output)}$$

$$\Delta_j = g'(u_j) \sum_i w_{ji} \Delta_i \qquad \text{(Others)}$$

$$w_{pq} \leftarrow w_{pq} + \underbrace{\eta \cdot V_p \cdot \Delta_q}_{\Delta w_{pq}}$$



Recurrent Backpropagation

- New update rules for weights:

$$\Delta_j = g'(u_j) Y_j$$

$$w_{pq} \leftarrow w_{pq} + \underbrace{\eta \cdot V_p \cdot \Delta_q}_{\Delta w_{pq}}$$

and use continuous updates to find Y_i s

$$\frac{\partial Y_i}{\partial dt} = \eta' \left(-Y_i + \sum_p w_{ip} \Delta_p + E_i \right)$$

→ Similar to continuous activation

→ Generalizes ordinary backpropagation

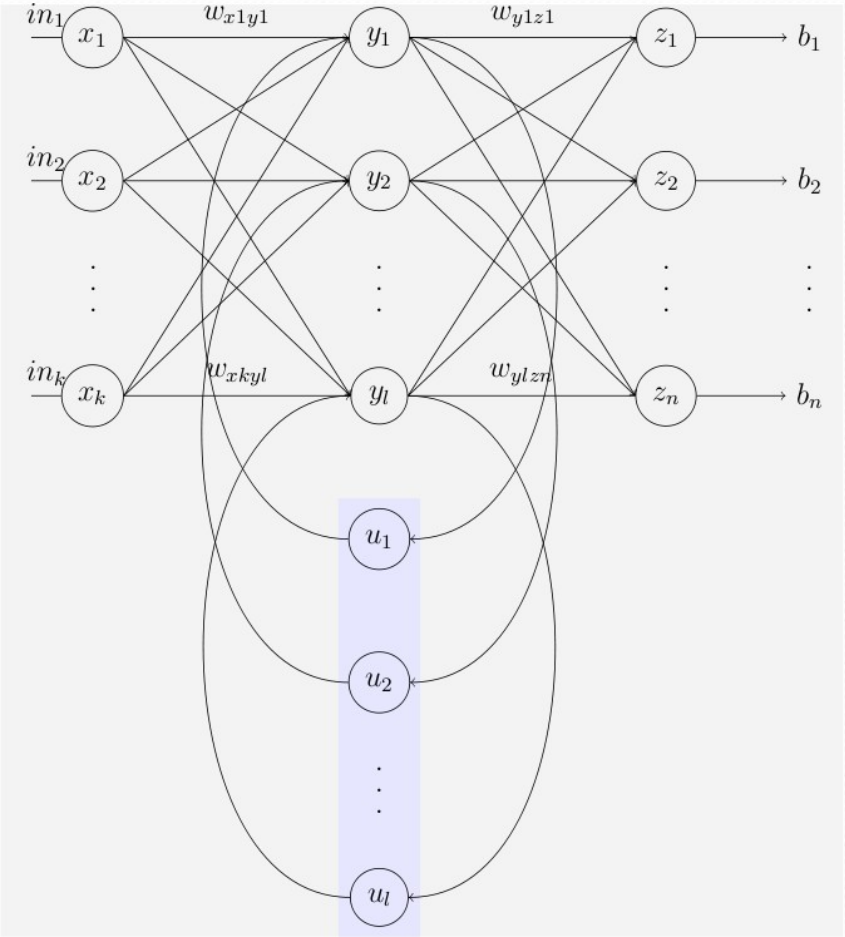
→ **Assuming** stable state exists

Old

$$\Delta_i = E_i g'(u_i)$$

$$\Delta_j = g'(u_j) \sum_i W_{ji} \Delta_i$$

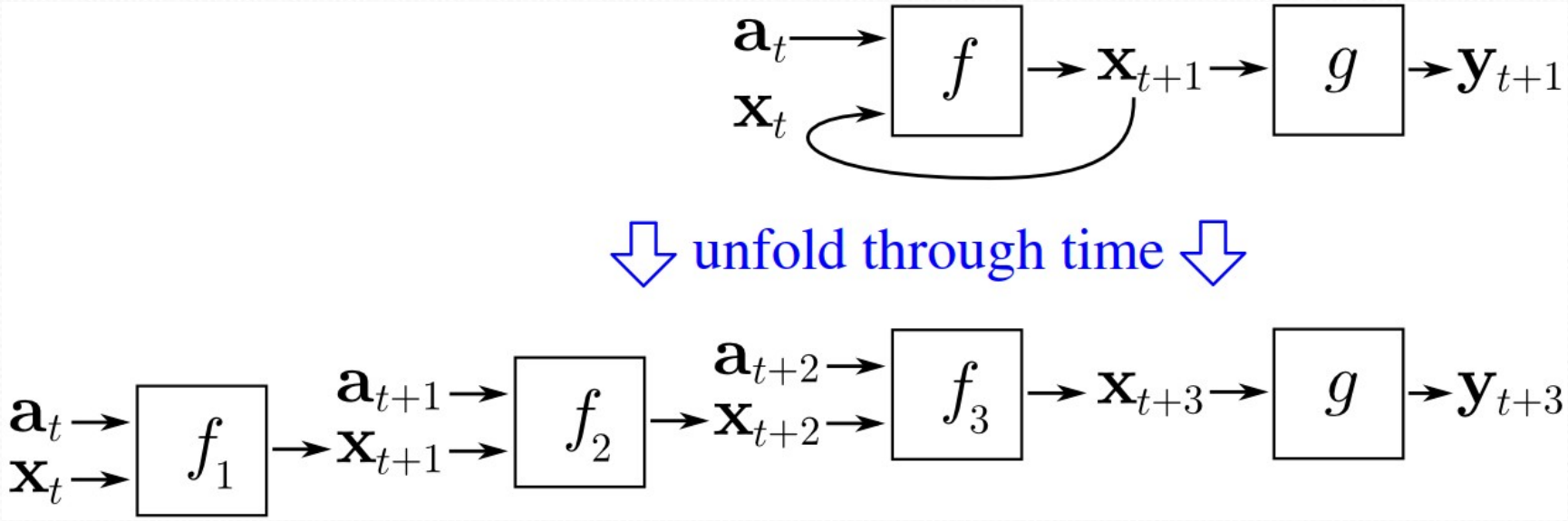
“Simple Recurrent Networks”



- Elman network:
  - one hidden layer
  - the *output* of each hidden node is *copied* to a “context unit”
  - operates on a sequence of patterns
  - context units are used as input when the next pattern is processed
  - hence when sequences of patterns are processed, the NN has a memory

Backpropagation through time

- Step 1: unfold network



- Step 2: perform backpropagation on this network, providing patterns $\mathbf{a}_t$, $\mathbf{a}_{t+1}$, ... in order
- Step 3: average weights to obtain weights for original network

Hopfield Networks: Limitations

- Assume 4 patterns over 3 variables:

-1	1	1	ξ^1
1	-1	1	ξ^2
1	1	-1	ξ^3
-1	-1	-1	ξ^4

- Hebbian rule: every weight in the network is

$$1/N \sum_{\mu} \xi_i^{\mu} \xi_j^{\mu}$$

Value of all weights?

Boltzmann Machines

- Extension of Hopfield networks
  - (symmetric weights, +1, -1 activations)
  - hidden units
  - stochastic: activate with a probability

$$p(S_i = +1) = g(h_i)$$

where $$h_i = \sum_j w_{ij} S_j$$

and $$g(h) = \frac{1}{1 + e^{-2/Th}} \qquad (\beta = 1/T)$$

→ The network can "walk" out of local minima

Boltzmann Machines

- Each state is reachable from every other state
- If we "simulate" the network for a long time, each state is encountered with a probability

$$p(S) = e^{\beta H(S)}/Z$$ (Boltzmann distribution)

where

- *Z* is a constant that ensures that the sum of probabilities over all states is 1
- *H(S)* is the energy of the state:

$$H(S) = -\frac{1}{2}\sum_{ij} w_{ij} S_i S_j$$

Boltzmann Machines

- Assume a subset of units X is visible (the remainder is hidden)

- Learning task: find weights such that the probability of generating training data is high (i.e. training examples have high probability, others low)

  - similar to Hopfield networks: “retrieve training examples” with high probability

Boltzmann Machines

- Probability of a training example:

$$p(\xi^\mu) = \sum_H p(\xi^\mu, H)$$

H ← All possible states for hidden units

- Likelihood (probability) of training data:

$$\log L \equiv \sum_\mu \log p(\xi^\mu)$$

- Perform gradient descent:

$$\Delta w_{ij} = \eta \frac{\partial}{\partial w_{ij}} \log L$$

Boltzmann Machines

- Gradient turns out to be (without proof):

$$\frac{\partial}{\partial w_{ij}} \log L = \langle S_i S_j \rangle_{clamped} - \langle S_i S_j \rangle_{free}$$

Product activation of nodes

here

- $\langle S_i S_j \rangle_{clamped}$ is the expected value of $S_i S_j$ according to the network, **assuming** we always fix the visible nodes to a pattern in the training data
- $\langle S_i S_j \rangle_{free}$ is the expected value of $S_i S_j$ according to the network, **without** fixing the visible nodes

Boltzmann Machines

- Calculating $\langle S_i S_j \rangle_{free}$: "Gibbs sampling":
  - initialize network in random state
  - run a simulation for a long time (let's say *n* epochs, asynchronous updates)
  - count how many times S_i and S_j are in each of the possible states

    S_i S_j

    1 1 *n1*

    -1 1 *n2*

    1 -1 *n3*

    -1 -1 *n4*
  - calculate expected value *(n1 – n2 – n3 + n4) / n*

Boltzmann Machines

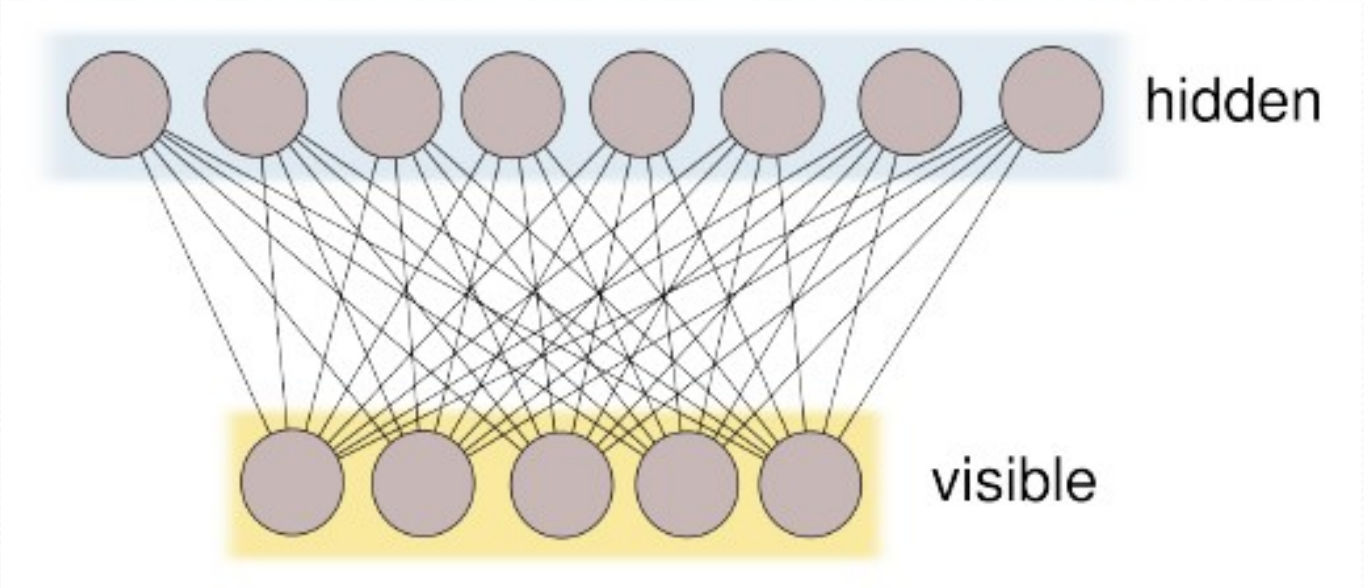
- Calculating $\langle S_i S_j \rangle_{clamped}$: repeated Gibbs sampling
- For each training pattern:
  - fix visible nodes to their value
  - run Gibbs sampling, not allowing visible nodes to change
  - calculate expected value from this run
- Average over all training patterns

Boltzmann Machines

- Disadvantages:
  - training is very slow (long simulations)
  - training is inaccurate (if simulations don't converge quickly enough)
- Not usable in practice
- Usable modification:
  Restricted Boltzmann Machines (RBMs)
  - *restricted structure:* only links between hidden and visible units
  - *different learning algorithm*

Restricted Boltzmann Machines

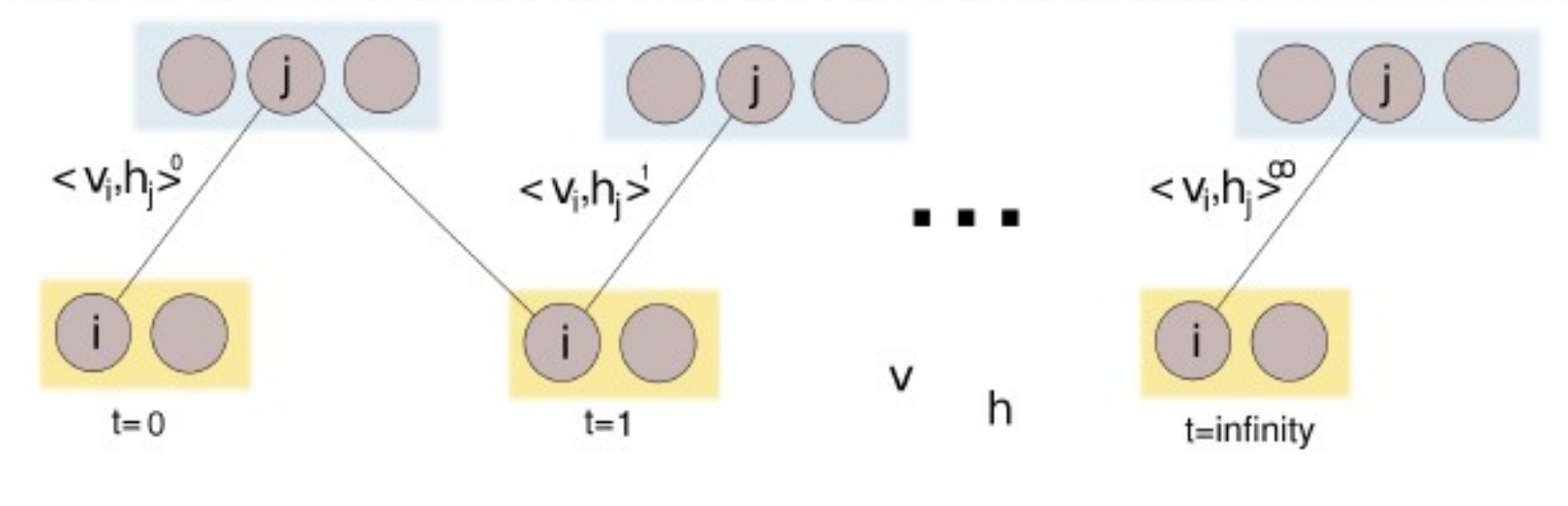
- Structure



Calculating $\langle S_i S_j \rangle_{clamped}$ is easy:
no sampling is needed (given fixed visible layer, we can calculate the probability that a hidden unit is in a given state exactly, and hence the probability that a pair of units is in a certain state)

Restricted Boltzmann Machines

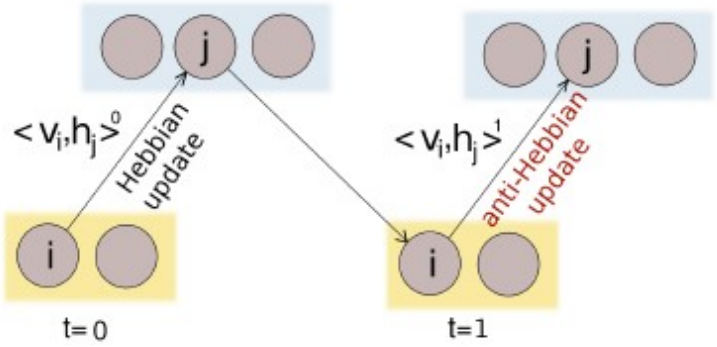
- Optimization 1: “alternating Gibbs sampling”: iterate between hidden and visible layer, update the whole layer



- Optimization 2: start the sampling process from a pattern (without “fixing” or “clamping” this pattern)

Restricted Boltzmann Machines

- Optimization 3: only 2 iterations of sampling



1. start with a training vector on the visible units
2. update all the hidden units in parallel
3. update all the visible units in parallel to get a "reconstruction"
4. update the hidden units again

$$\Delta w_{ij} = \eta \left[\langle v_i\, h_j \rangle^0 - \langle v_i\, h_j \rangle^1 \right]$$

(for each training pattern)

Computed in the same way from the visible state

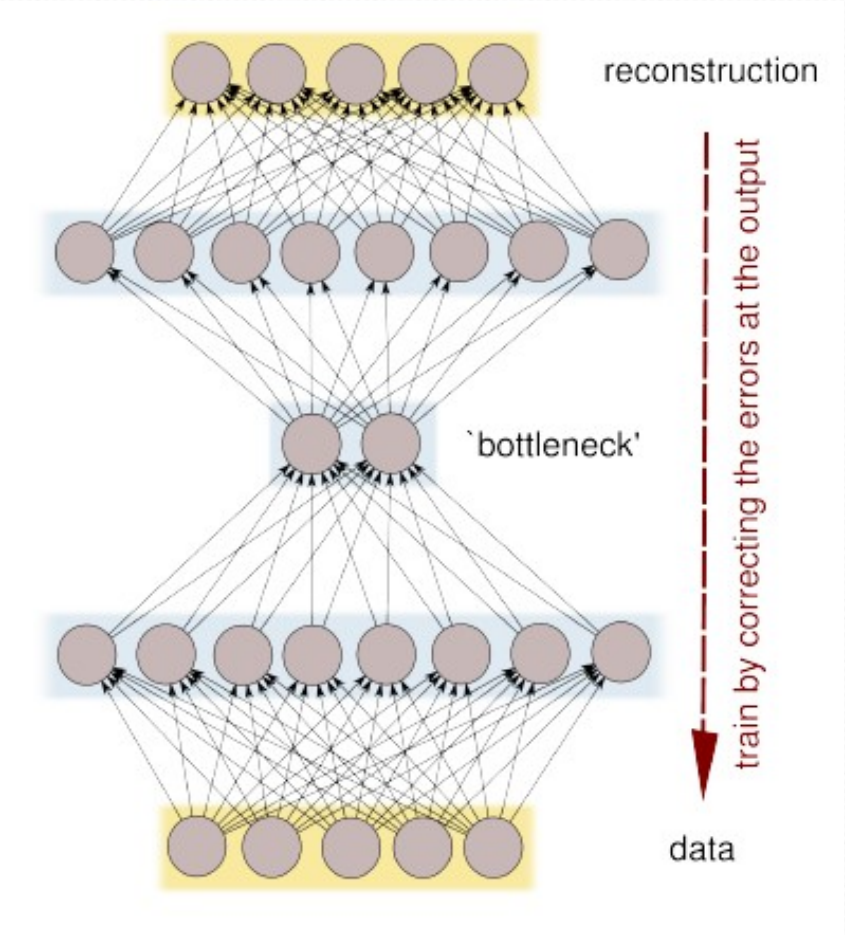
Restricted Boltzmann Machines

- Consequence: Restricted Boltzmann Machines can be learned efficiently, without extensive sampling
  - only two iterations of “alternating Gibbs sampling”
  - use exact calculations to compute the probability that a pair of units in a given state, which allows to calculate the expected value of this pair efficiently

Deep Belief Networks

- "Hot topic" in neural networks
- Key idea: learn a deep (many layers) neural network as an associative memory

Deep Belief Networks



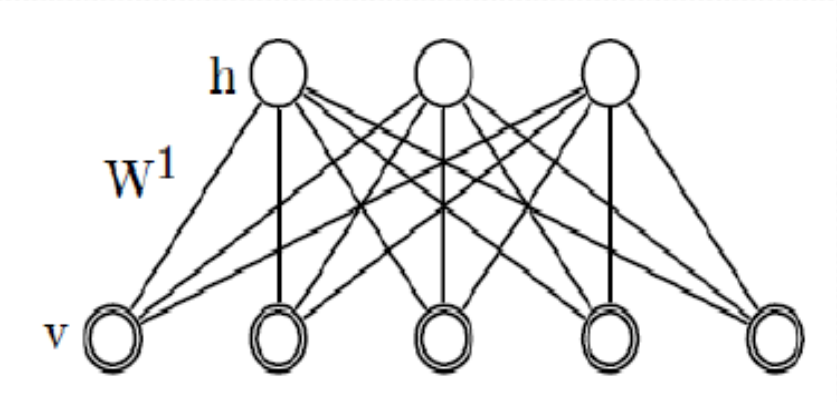
Traditional approach

- Many layers, one layer of which has few units
- Train the network with the same pattern as input and as output
- Hopefully, the networks learns to predict training examples
- Idea: coordinates in “small layer” identify the input concisely! (“autoencoding”)
- Unfortunately, backpropagation doesn't work in practice here

Deep Belief Networks

- Alternative idea: iteratively use Restricted Boltzmann Machines

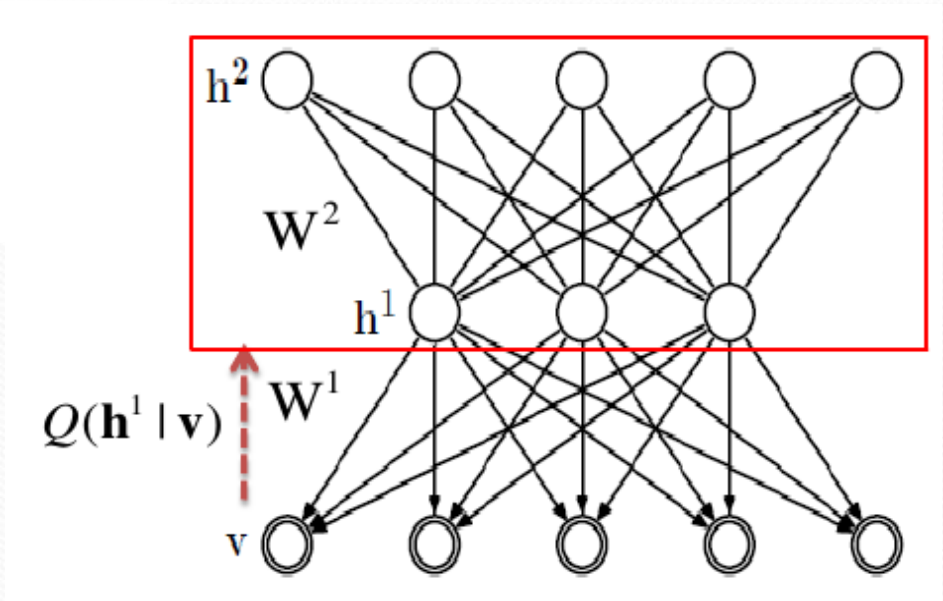
- Step 1: construct an initial RBM:



(Only a sketch of the idea is given!)

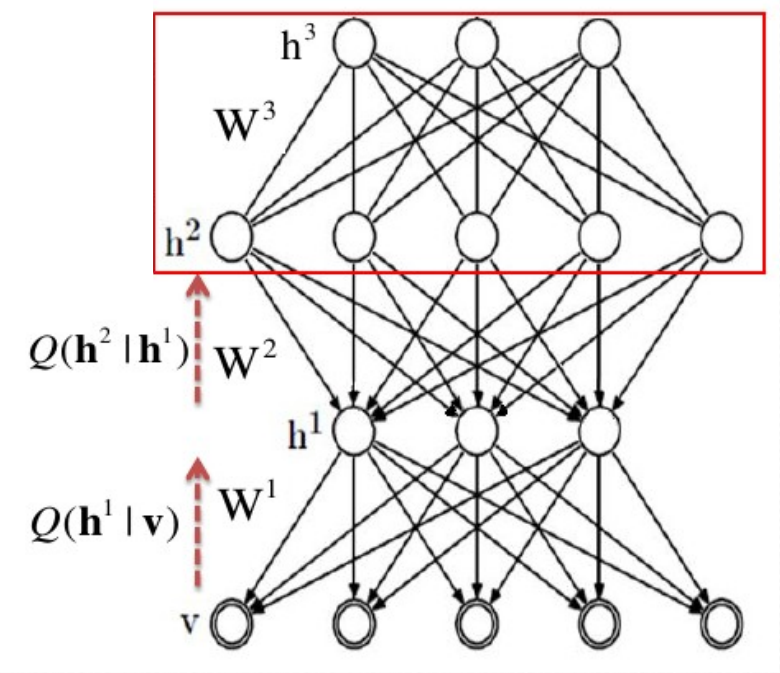
Deep Belief Networks

- Step 2: for each pattern in the training dataset, sample a pattern in the hidden layer → results in a new dataset
- Step 3: build a RBN for this dataset

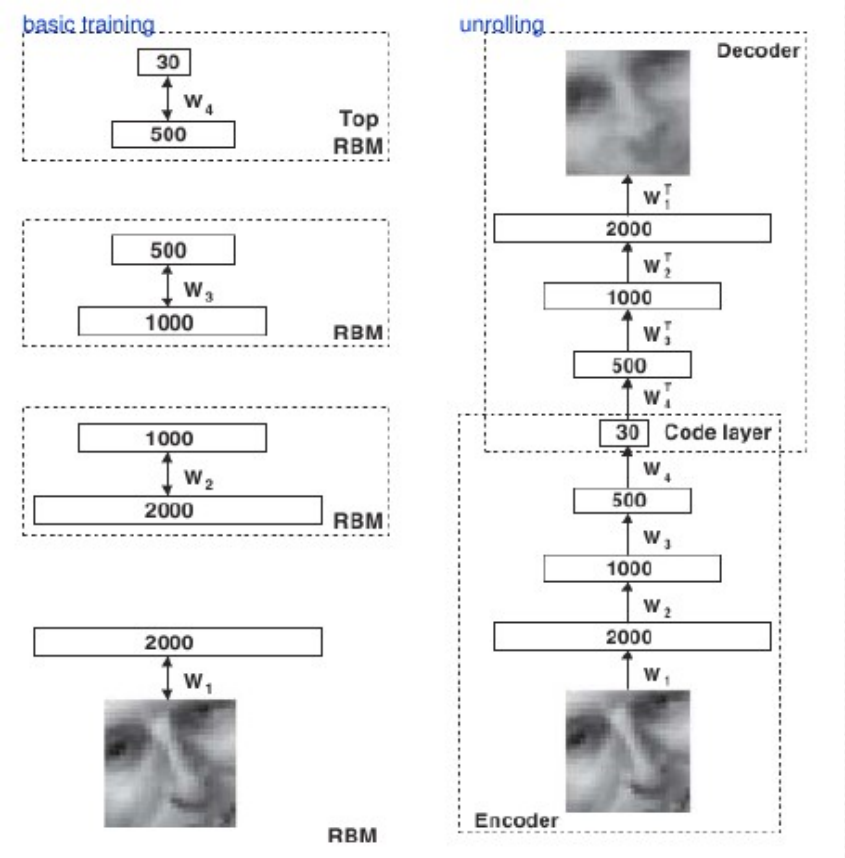


Deep Belief Networks

- Step 4: repeat for as many layers as desired



Deep Belief Networks

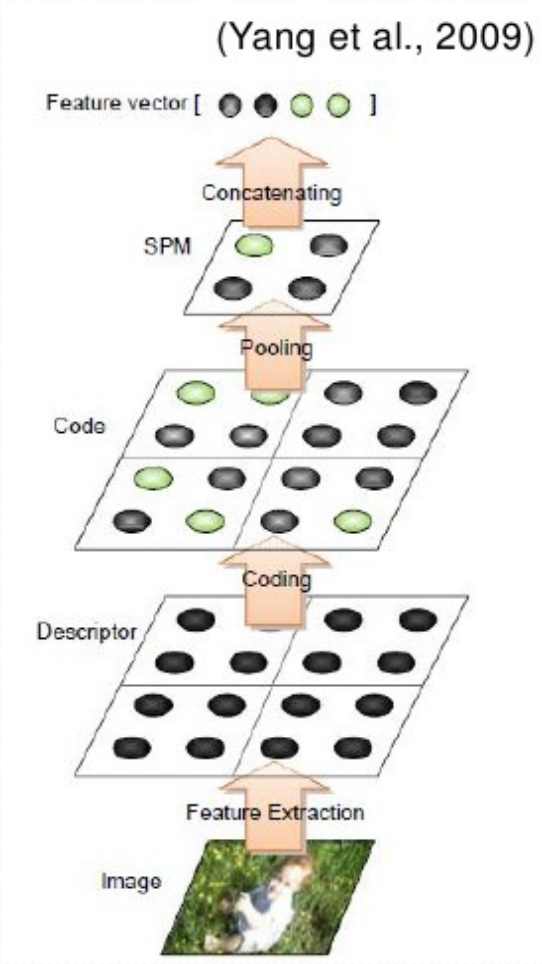


- Treat the stack of RBMs as a *directed* stochastic network
- Starting from a pattern, compute output several times to compute an expected output

  → Treat network as probabilistic model

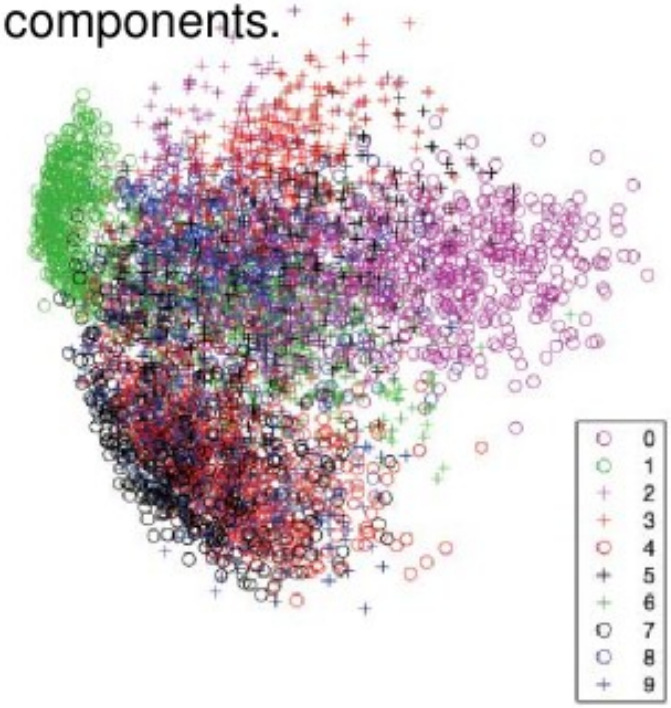
Deep Belief Networks

- Intuition: layers of features

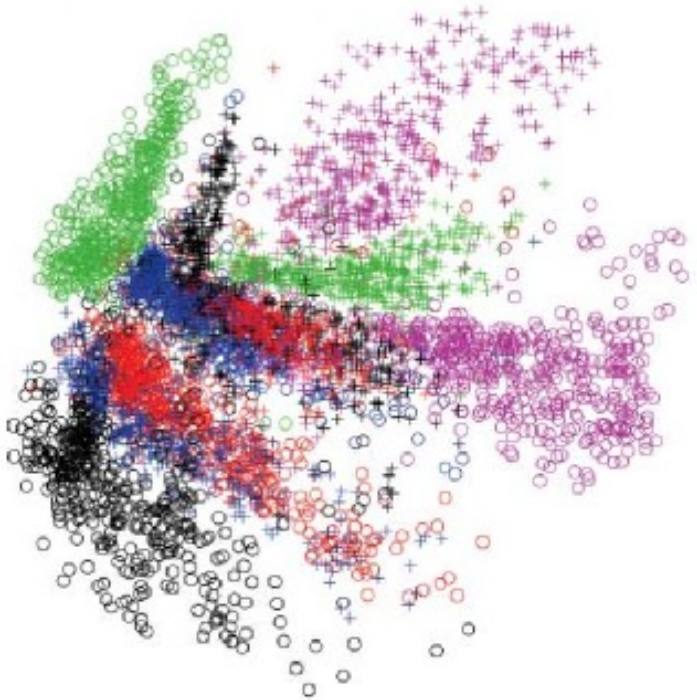


Deep Belief Networks

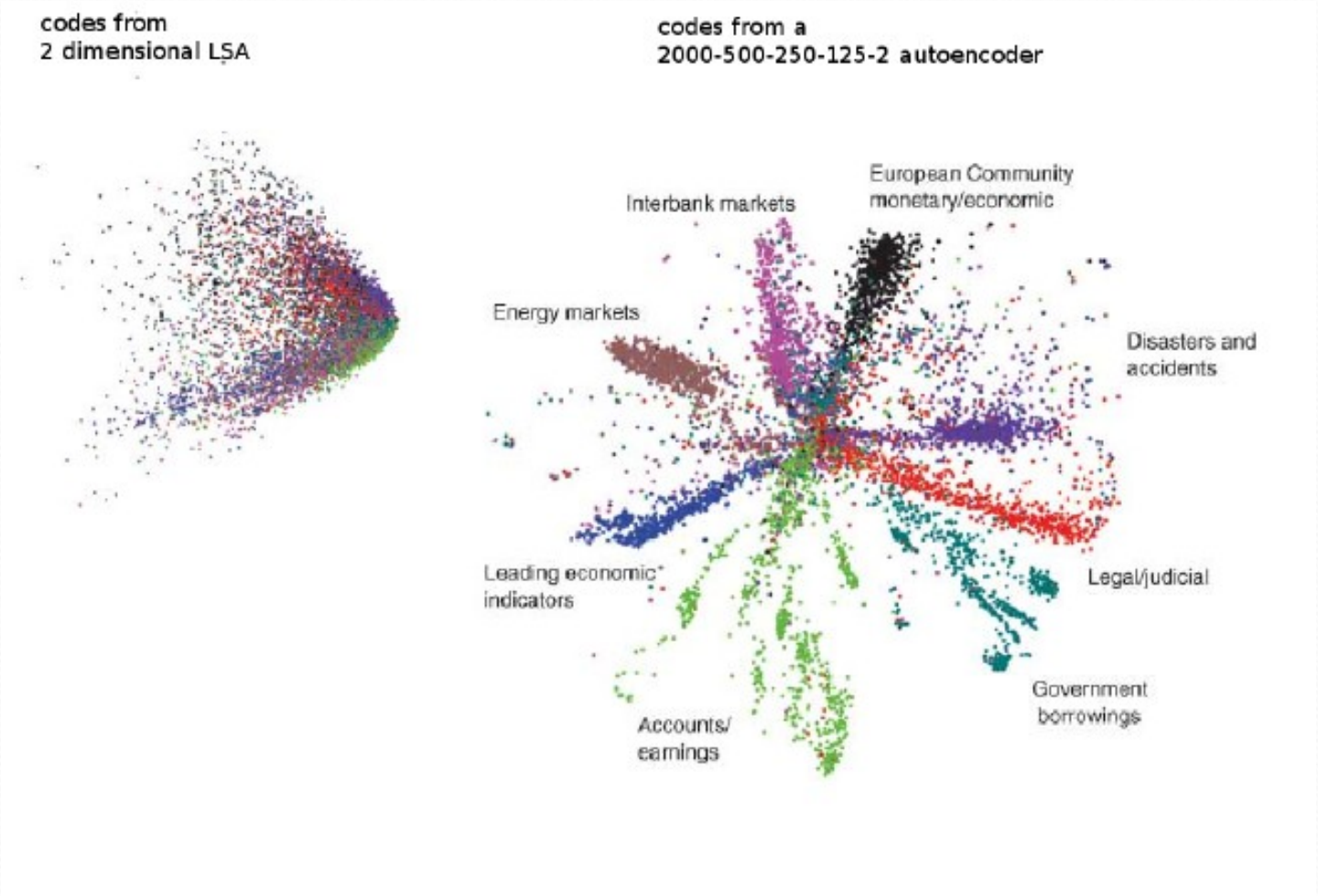
Codes for digits, produced by taking the first 2 principal components.



Codes from a 784-1000-500-250-2 autoencoder.



Deep Belief Networks



Hinton & Salakhutdinov, *Science*, 2007