Neural Networks

William Cohen

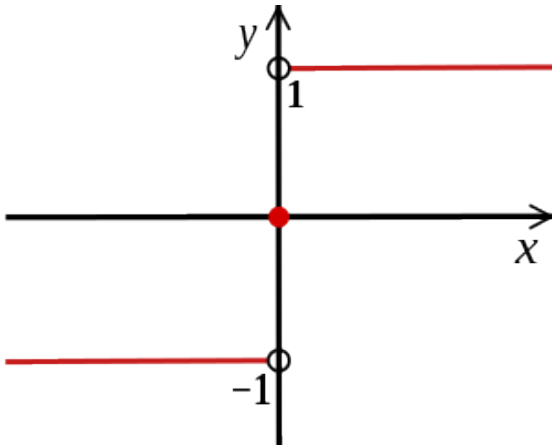
10-601

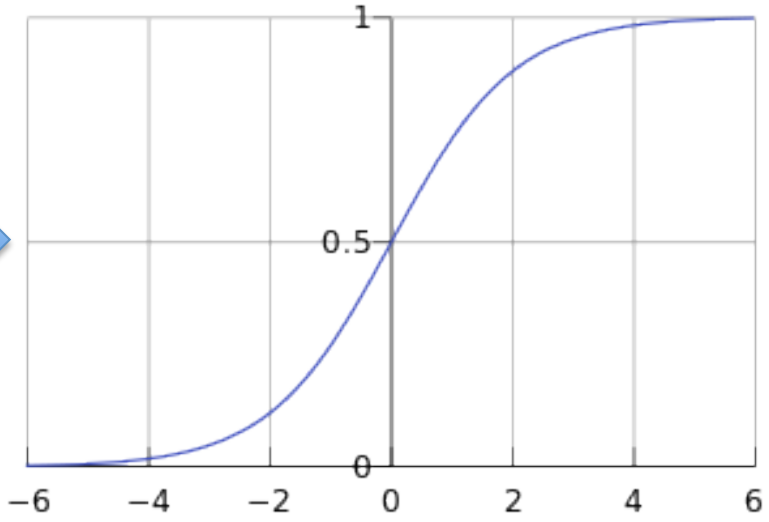
[pilfered from: Ziv; Geoff Hinton; Yoshua Bengio; Yann LeCun; Hongkak Lee - NIPs 2010 tutorial]

WHAT ARE NEURAL NETWORKS?

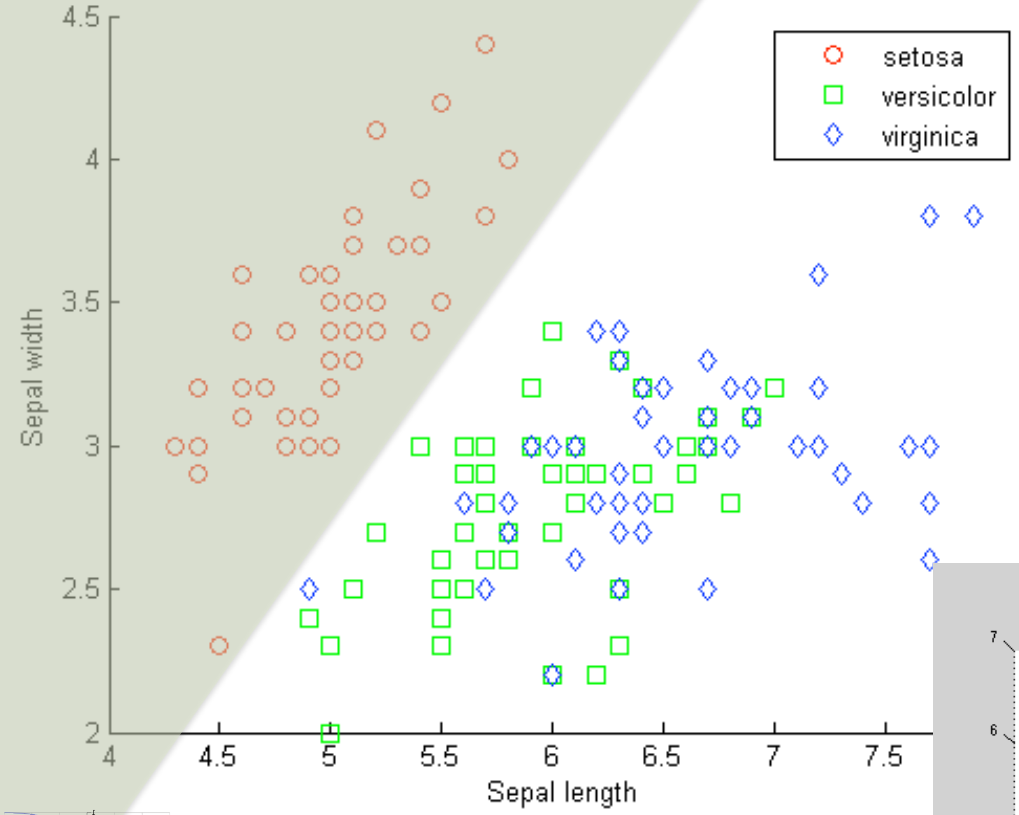
Logistic regression

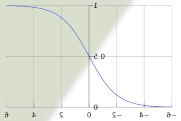
$$P(y_i \mid \mathbf{x}_i, \mathbf{w}) \equiv \begin{cases} \dfrac{1}{1+\exp(-\mathbf{x}_i \cdot \mathbf{w})} & \text{if } y_i = 1 \\ \left(1 - \dfrac{1}{1+\exp(-\mathbf{x}_i \cdot \mathbf{w})}\right) & \text{if } y_i = 0 \end{cases}$$





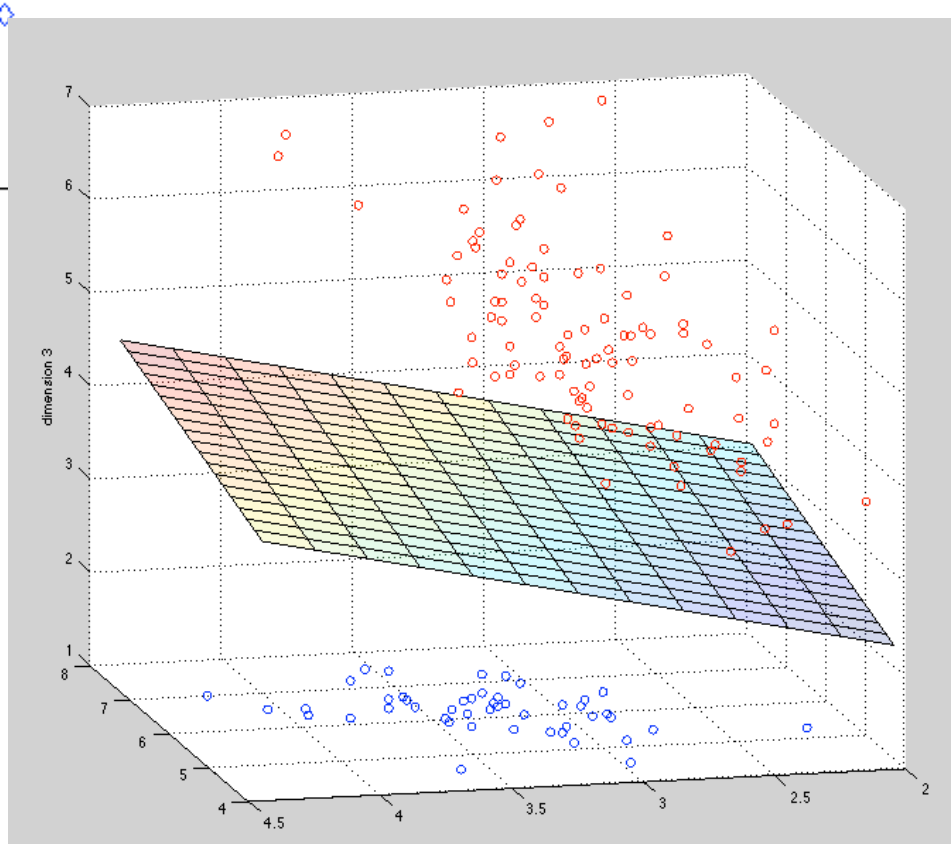
$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$





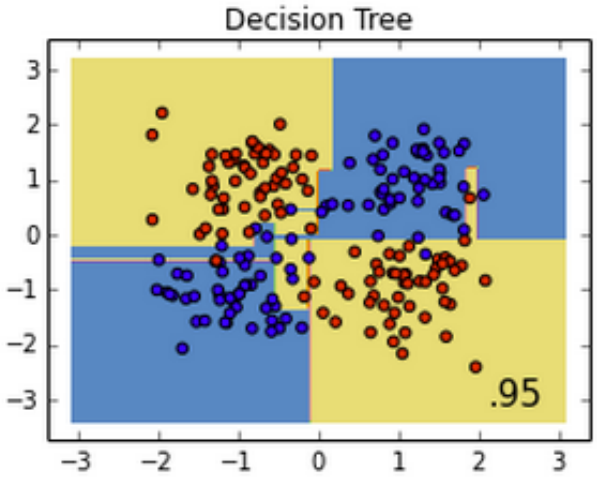
Predict "pos" on (x1,x2) iff
$ax1 + bx2 + c > 0$

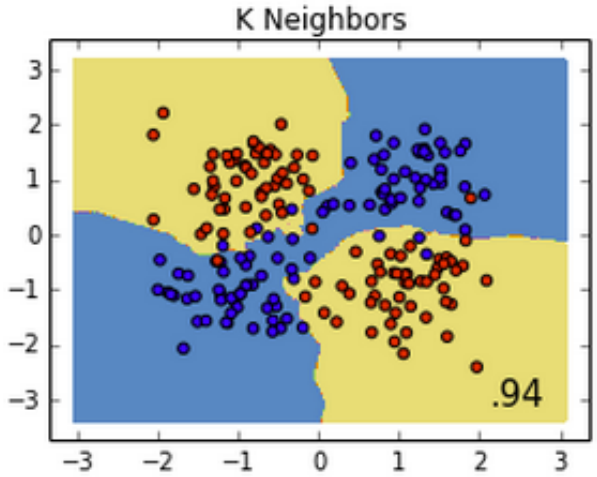
$= S(ax1 + bx2 + c)$

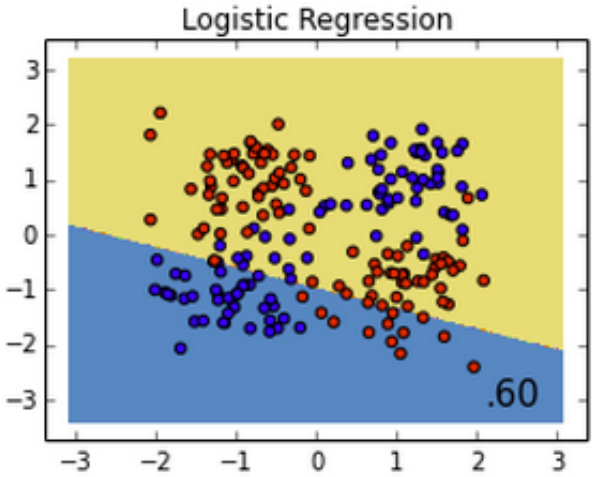


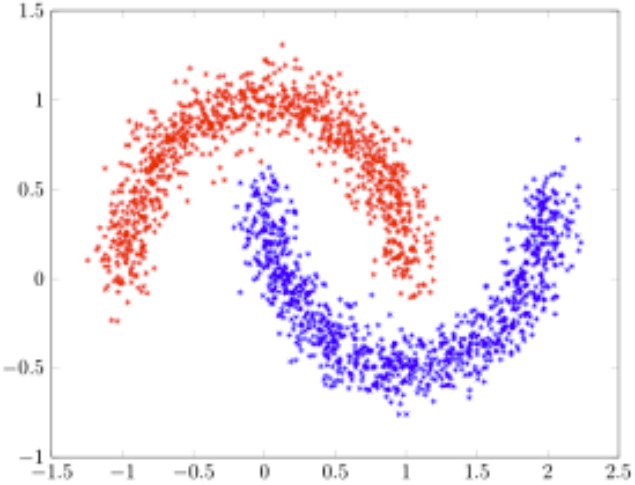
On beyond linear classifiers

- What about data that's not linearly separable?



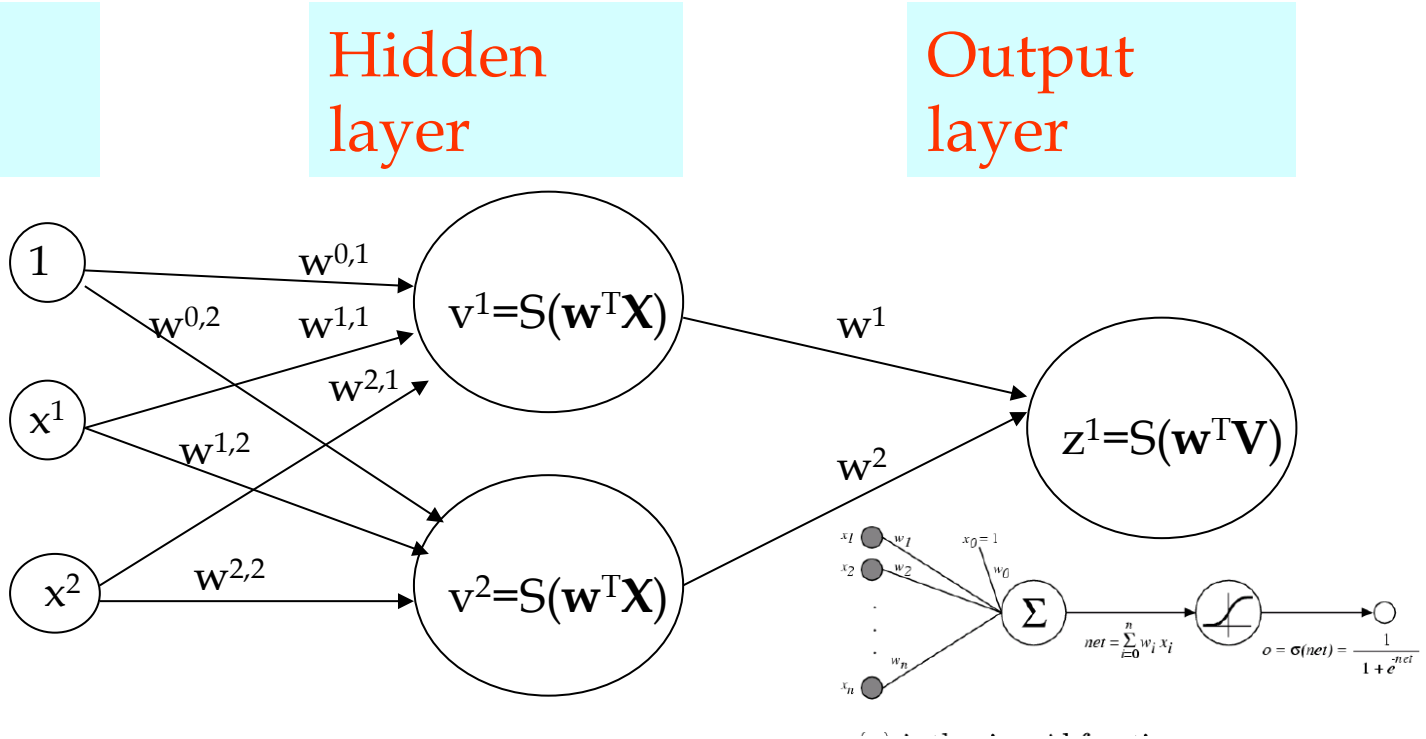






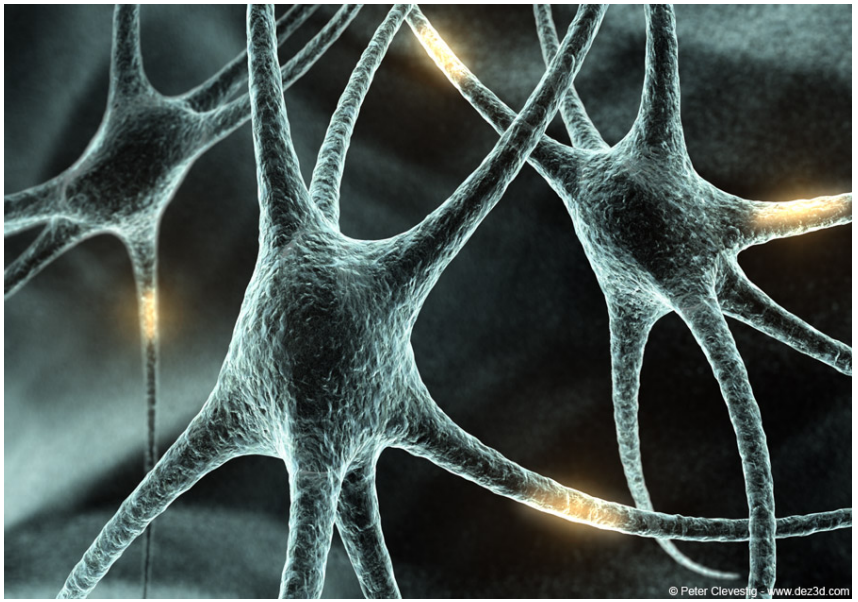
One powerful extension to logistic regression: multilayer networks

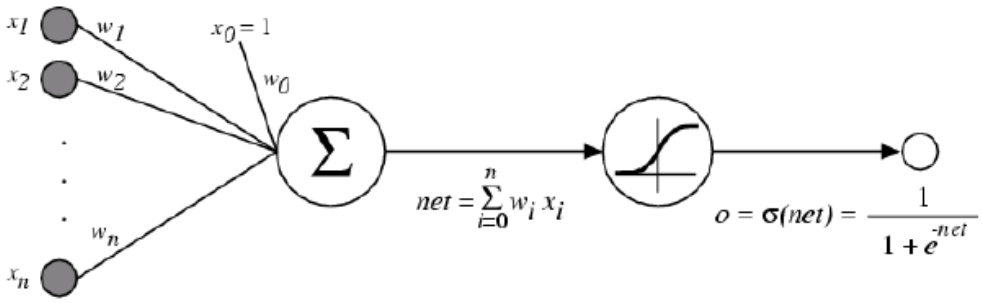
- Classifier is a multilayer *network* of *logistic units*
- Each *unit* takes some inputs and produces one output using a logistic classifier
- Output of one unit can be the input of another



Multilayer Logic Regression: one type of Artificial Neural Network (ANN)

- Neuron is a cell in the brain
- Highly connected to other neurons, and performs computations by combining signals from other neurons
- Outputs of these computations may be transmitted to one or more other neurons





Does AI need ANNs?

- Neurons take ~ 0.001 second to change state
- We can do complex scene recognition tasks in ~ 0.1 sec or about 100 steps of computation
- ANNs are similar:
  - lots of parallelism
  - not many *steps* of computation

BACKPROP: LEARNING FOR NEURAL NETWORKS

Ideas from Linear and Logistic regression

- Define a loss which is squared error
  - But over a network of logistic units
- Minimize loss with gradient descent

$$J_{\mathbf{X},\mathbf{y}}(\mathbf{w}) = \sum_i \left(y^i - \hat{y}^i\right)^2$$

  - But output is network output
- Quick review: the math for linear regression and logistic regression

Gradient Descent for Linear Regression (simplified)

Differentiate the loss function:

predict with: $\hat{y}^i = \sum_j^n w_j x_j$

$$J_{\mathbf{X},\mathbf{y}}(\mathbf{w}) = \frac{1}{2}\sum_i \left(y^i - \hat{y}^i\right)^2 = \frac{1}{2}\sum_i \left(y^i - \sum_j w_j x_j\right)^2$$

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i \left(y^i - \hat{y}^i\right)^2$$

$$= \sum_i \left(y^i - \hat{y}^i\right) \frac{\partial}{\partial w_j}\hat{y}^i$$

$$= \sum_i \left(y^i - \hat{y}^i\right) \frac{\partial}{\partial w_j}\sum_j w_j x_j$$

$$= \sum_i \left(y^i - \hat{y}^i\right) x_j$$

Gradient Descent for Logistic Regression (simplified)

Output of a logistic unit:

$$p \equiv \frac{1}{1 + e^{-\mathbf{x} \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}$$

After some math:

$$\frac{\partial}{\partial w^j} p = p(1 - p) x^j$$

$$p \equiv \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}} = \frac{1}{1+\exp(-\sum_j x^j w^j)}$$

$$1-p = \frac{1+\exp(-\sum_j x^j w^j)}{1+\exp(-\sum_j x^j w^j)} - \frac{1}{1+\exp(-\sum_j x^j w^j)} = \frac{\exp(-\sum_j x^j w^j)}{1+\exp(-\sum_j x^j w^j)}$$

$$\frac{\partial}{\partial w^j} p = \frac{\partial}{\partial w^j}(1+\exp(-\sum_j x^j w^j))^{-1}$$

$$(f^n)' = nf^{n-1}\cdot f'$$

$$(e^f)' = e^f f'$$

$$= (-1)(1+\exp(-\sum_j x^j w^j))^{-2} \frac{\partial}{\partial w^j} \exp(-\sum_j x^j w^j)$$

$$= (-1)(1+\exp(-\sum_j x^j w^j))^{-2} \exp(-\sum_j x^j w^j)(-x^j)$$

p

$$= \frac{1}{1+\exp(-\sum_j x^j w^j)} \frac{\exp(-\sum_j x^j w^j)}{1+\exp(-\sum_j x^j w^j)} x^j$$

$$\frac{\partial}{\partial w^j} p = p(1-p)x^j$$

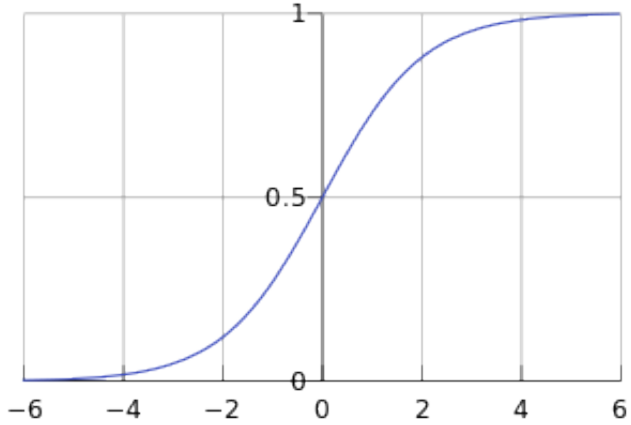
Gradient Descent for Logistic Regression (simplified)

Output of a logistic unit:

$$p \equiv \frac{1}{1 + e^{-\mathbf{x}\cdot\mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}$$

After some math:

$$\frac{\partial}{\partial w^j} p = p(1-p)x^j$$



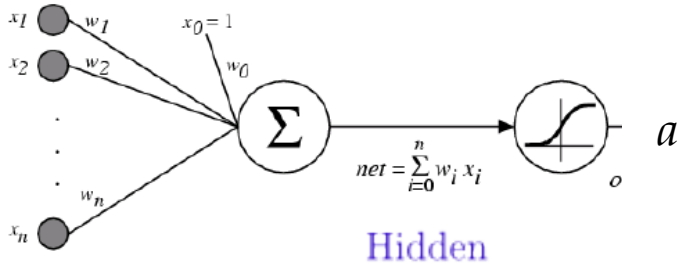
$$S(g) = \frac{1}{1+e^{-g}}$$

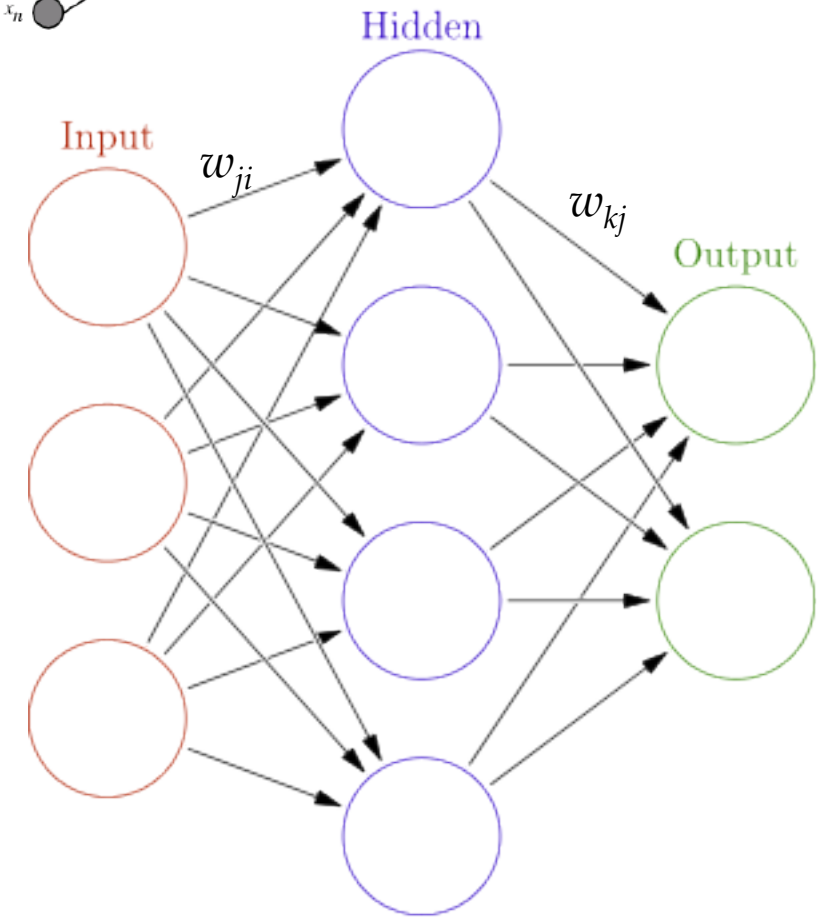
$$\frac{\partial S}{\partial g} = g(1-g)$$

DERIVATION OF BACKPROP

Notation

- *y* becomes a "target" *t*
- *a* is an "activation" and *net* is a weighted sum of inputs – i.e., *a=S(net)*
- *S* is sigmoid function
- w_{kj} is a weight from hidden *j* to output *k*
- w_{ji} is a weight from input *i* to hidden *j*





$$J(\mathbf{w}) = \frac{1}{2}\sum_{k}\left(t_k - a_k\right)^2$$

Derivation: gradient for output-layer weights

$$J(\mathbf{w}) = \frac{1}{2}\sum_{k}\left(t_k - a_k\right)^2$$

$$\frac{\partial}{\partial w_{kj}} J = \frac{\partial J}{\partial a_k}\frac{\partial a_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}}$$

Derivation: output layer

$$\frac{\partial net_k}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}} \sum_{j'} w_{kj'} a_{j'} = a_j$$

$$J(\mathbf{w}) = \frac{1}{2} \sum_k \left(t_k - a_k \right)^2$$

$$\frac{\partial}{\partial w_{kj}} J = \frac{\partial J}{\partial a_k} \frac{\partial a_k}{\partial net_k} \frac{\partial net_k}{\partial w_{kj}}$$

$$\frac{\partial J}{\partial a_k} = \left(t_k - a_k \right)$$

$$\frac{\partial a_k}{\partial net} = a_k \left(1 - a_k \right)$$

From logistic regression result

Derivation: output layer

$$J(\mathbf{w}) = \frac{1}{2}\sum_k \left(t_k - a_k\right)^2$$

$$\frac{\partial}{\partial w_{kj}} J = \left(t_k - a_k\right) a_k\left(1 - a_k\right) a_j$$

$$\delta_k \equiv \left(t_k - a_k\right) a_k\left(1 - a_k\right)$$

error

sensitivity

$$\Rightarrow \frac{\partial}{\partial w_{kj}} J = \delta_k a_j$$

Derivation: gradient for hidden-layer weights

$$J(\mathbf{w}) = \frac{1}{2}\sum_k \left(t_k - a_k\right)^2$$

$$J(\mathbf{w}) = \frac{1}{2}\sum_k \left(t_k - S\left(\sum_j w_{kj} a_j\right)\right)^2$$

$$J(\mathbf{w}) = \frac{1}{2}\sum_k \left(t_k - S\left(\sum_j w_{kj} S\left(\sum_i w_{ji} a_i\right)\right)\right)^2$$

$$\frac{\partial}{\partial w_{ji}} J = \left(\sum_k \frac{\partial J}{\partial a_k}\frac{\partial a_k}{\partial net_k}\frac{\partial net_k}{\partial a_j}\right)\frac{\partial a_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

Derivation: hidden layer

$$\frac{\partial}{\partial w_{ji}} J = \left(\sum_k \frac{\partial J}{\partial a_k} \frac{\partial a_k}{\partial net_k} \frac{\partial net_k}{\partial a_j} \right) \frac{\partial a_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \left(\sum_k \delta_k w_{kj} \right) a_j (1 - a_j) a_i$$

$$\delta_j \equiv \sum_k \left(\delta_k w_{kj} \right) a_j \left(1 - a_j \right)$$

$$\Rightarrow \frac{\partial J}{\partial w_{ji}} = \delta_j a_i$$

Computing the weight update

For nodes k in output layer:

$$\delta_k \equiv \left(t_k - a_k\right)\ a_k\left(1 - a_k\right)$$

For nodes j in hidden layer:

$$\delta_j \equiv \sum_k \left(\delta_k w_{kj}\right)\ a_j\left(1 - a_j\right)$$

For all weights:

$$w_{kj} = w_{kj} - \varepsilon\ \delta_k a_j$$

$$w_{ji} = w_{ji} - \varepsilon\ \delta_j a_i$$

"Propagate errors backward" BACKPROP

Can carry this recursion out further if you have multiple hidden layers

Some extensions to BackProp….

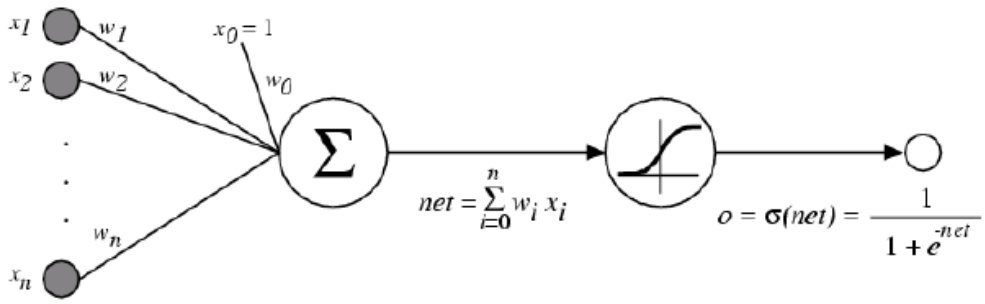
- Different units:
  - Can replace $(1 + e^{-x})^{-1}$ with tanh, …
- Different architectures:
  - Can extend this method to any directed graph
- Weight sharing/parameter “tie”-ing
  - Can have the same weights appear in multiple locations in the graph (more later)

EXPRESSIVENESS OF ANNS

Networks of logistic units can do a lot

- One logistic unit can implement and AND or an OR of a subset of inputs
  - e.g., (x_3 AND x_5 AND … AND x_{19})
- Every boolean function can be expressed as an OR of ANDs
  - e.g., (x_3 AND x_5) OR (x_7 AND x_{19}) OR …
- So one hidden layer can express any BF

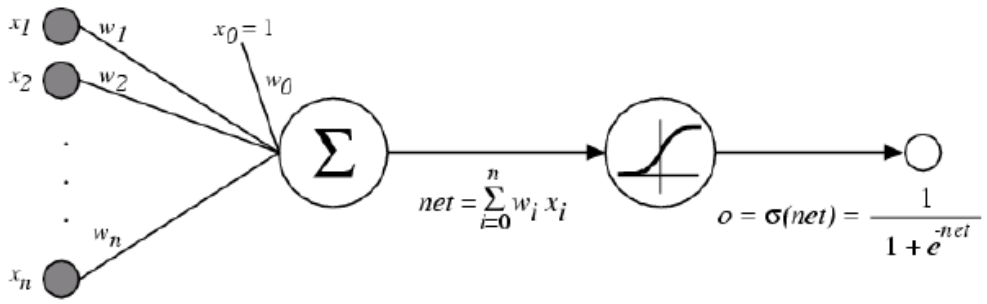
(But it might need lots and lots of hidden units)



Networks of logistic units can do a lot

- An ANN with one hidden layer can also approximate every bounded continuous function.

(But it might need lots and lots of hidden units)



Limitations of Backprop

- Learning is slow: thousands of iterations
- Learning is often ineffective on deep networks
  - more than 1-2 hidden layers (exception: convolutional nets)
- Lots of parameters to tweak
  - structure of the network
  - regularization, learning rate, "momentum" terms, number of iterations/convergence, …
  - common trick is *early stopping*: train till error on a held-out "validation set" stops decreasing
    - similar to L2-regularization
- BP finds a local minima, not a global one
  - starting point matters (multiple restarts)
  - warning: you usually want to randomize the weights when you start
- It only is useful for supervised learning tasks:
  - or, is it?

DEEP NEURAL NETWORKS

Observations and motivations

- Slow is not scary anymore
  - Moore's law, GPUs, …
- Brains seem to be hierarchical
  - "deeper" more abstract features computed from simpler ones
  - 100 steps != 3 steps

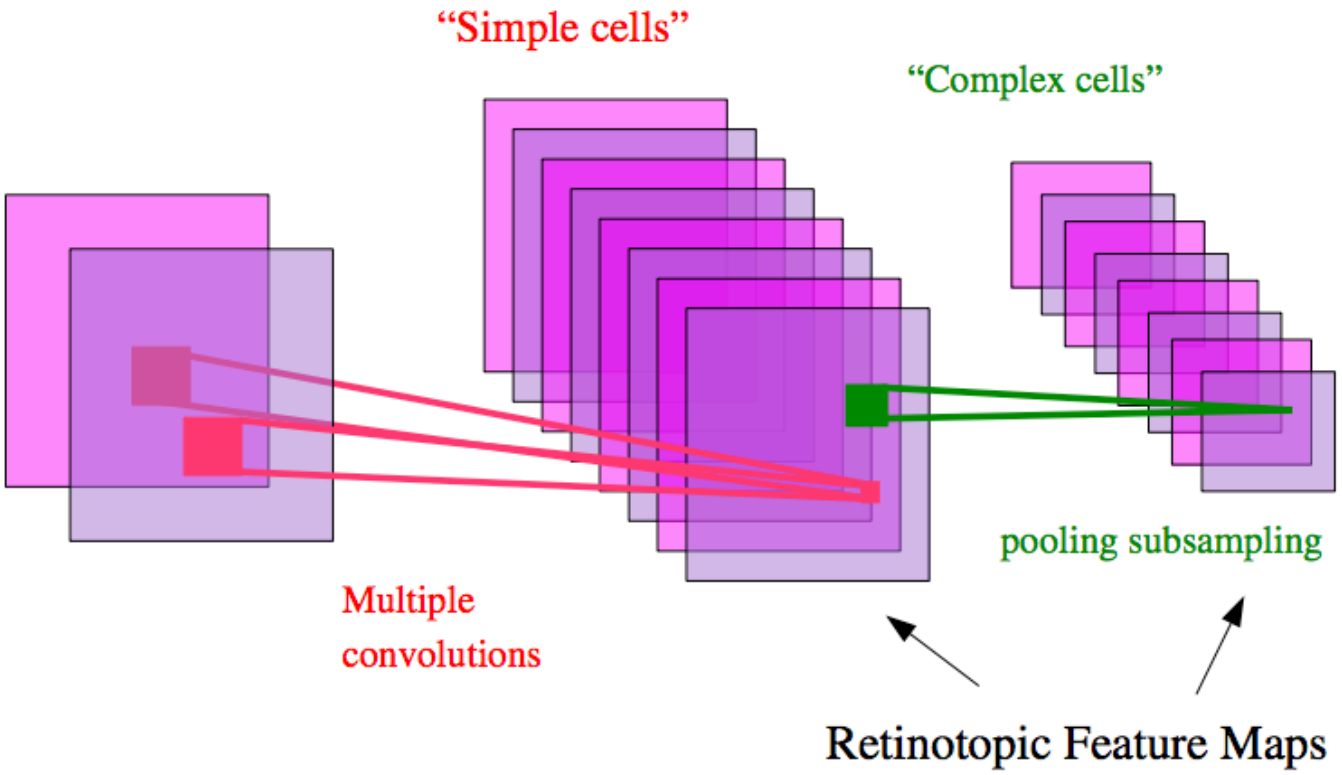
CONVOLUTIONAL NETS

…weight-sharing and convolutional networks

- Convolutional neural networks for vision tasks:
  - These can be trained with Backprop, even with many hidden layers
  - Every unit is a "feature detector" for part of the "retina", with a *receptor field* that defines its inputs and (conceptually) a *location* on the retina
  - Two types of units:
    - detection: learn to recognize some feature (copied to multiple places via weight-sharing)
    - pooling: combine detectors from multiple nearby locations together (e.g., max, sampling, …)

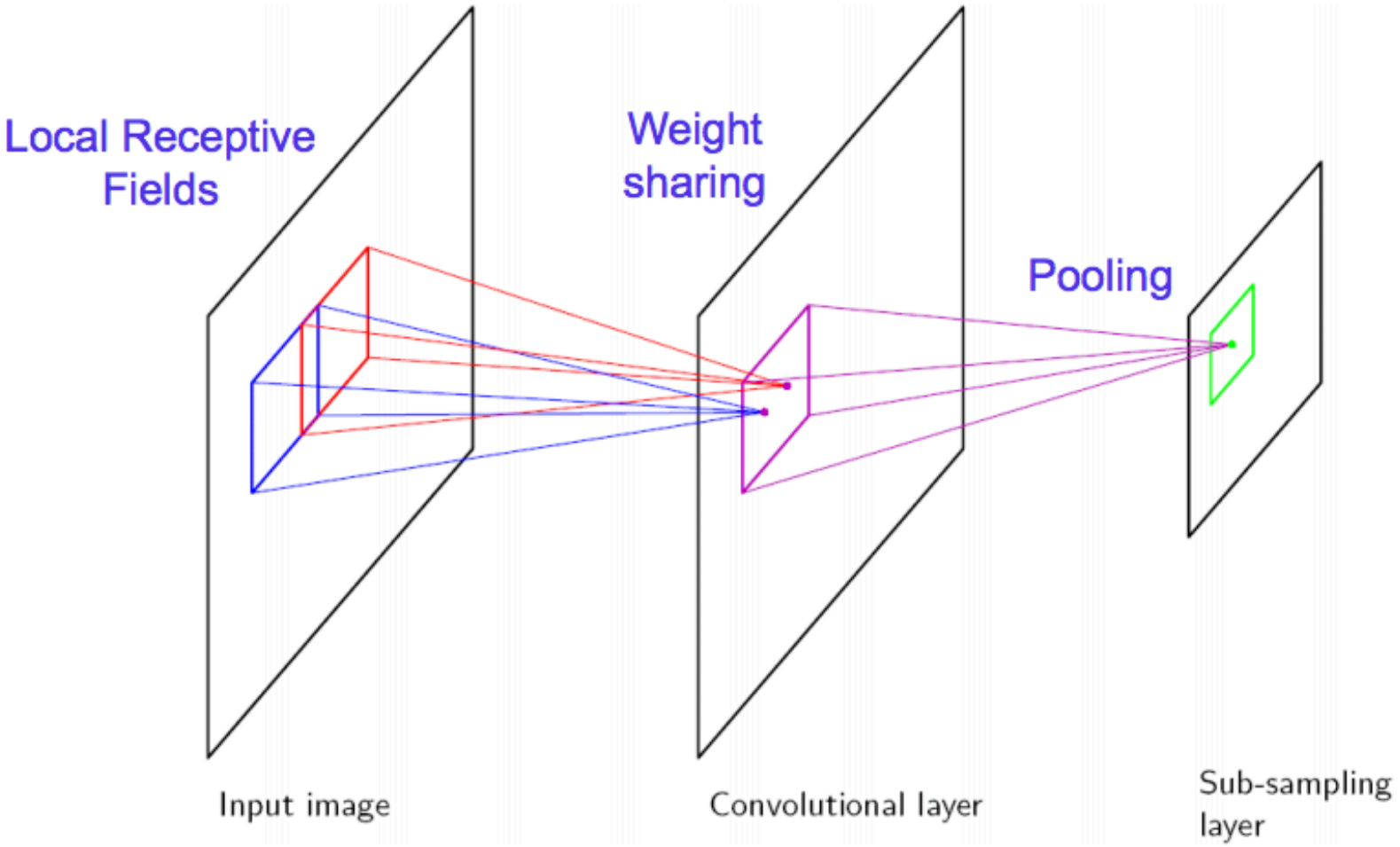
Model of vision in animals

- **[Hubel & Wiesel 1962]:**
  - simple cells detect local features
  - complex cells "pool" the outputs of simple cells within a retinotopic neighborhood.

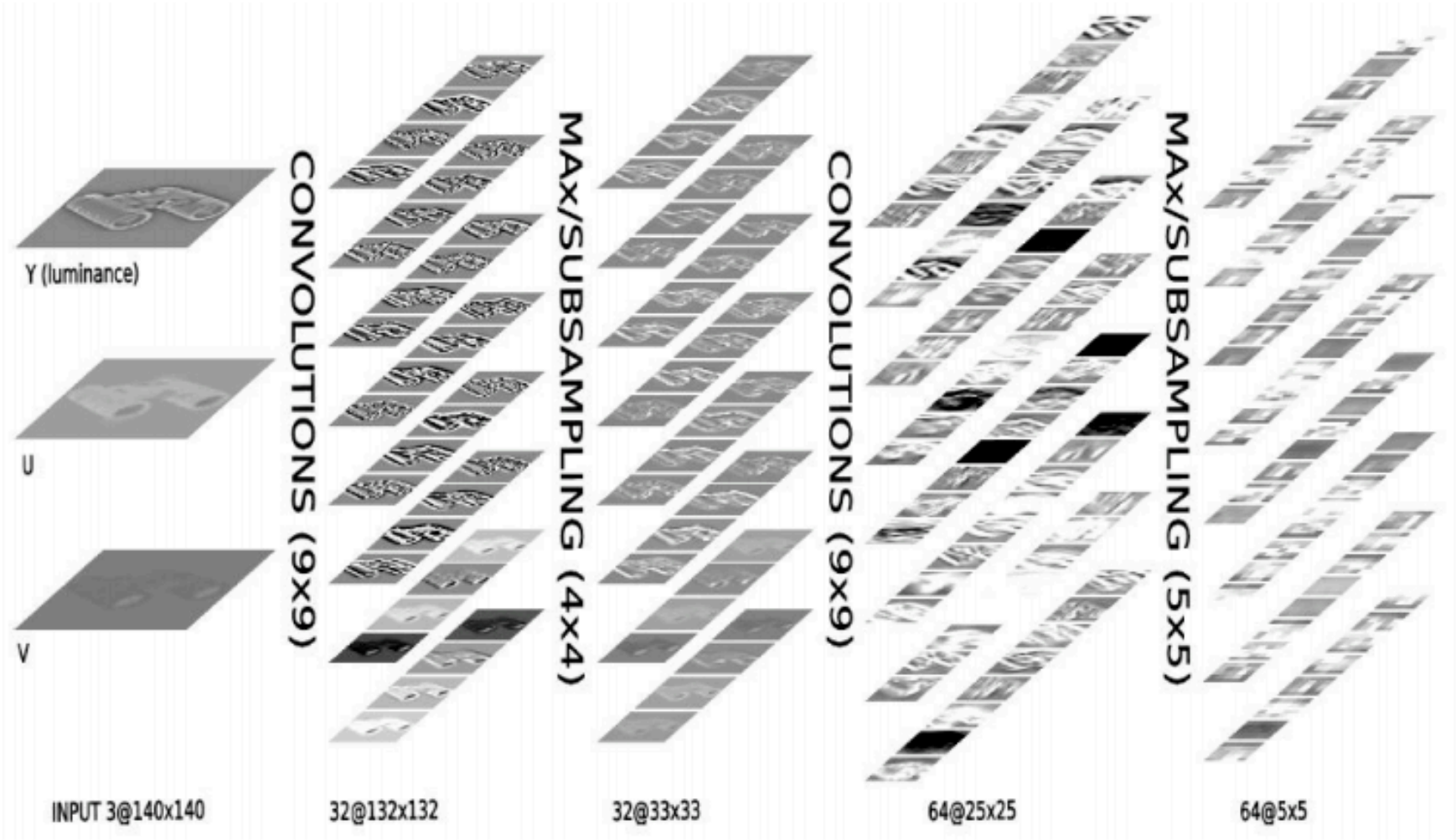


Vision with ANNs

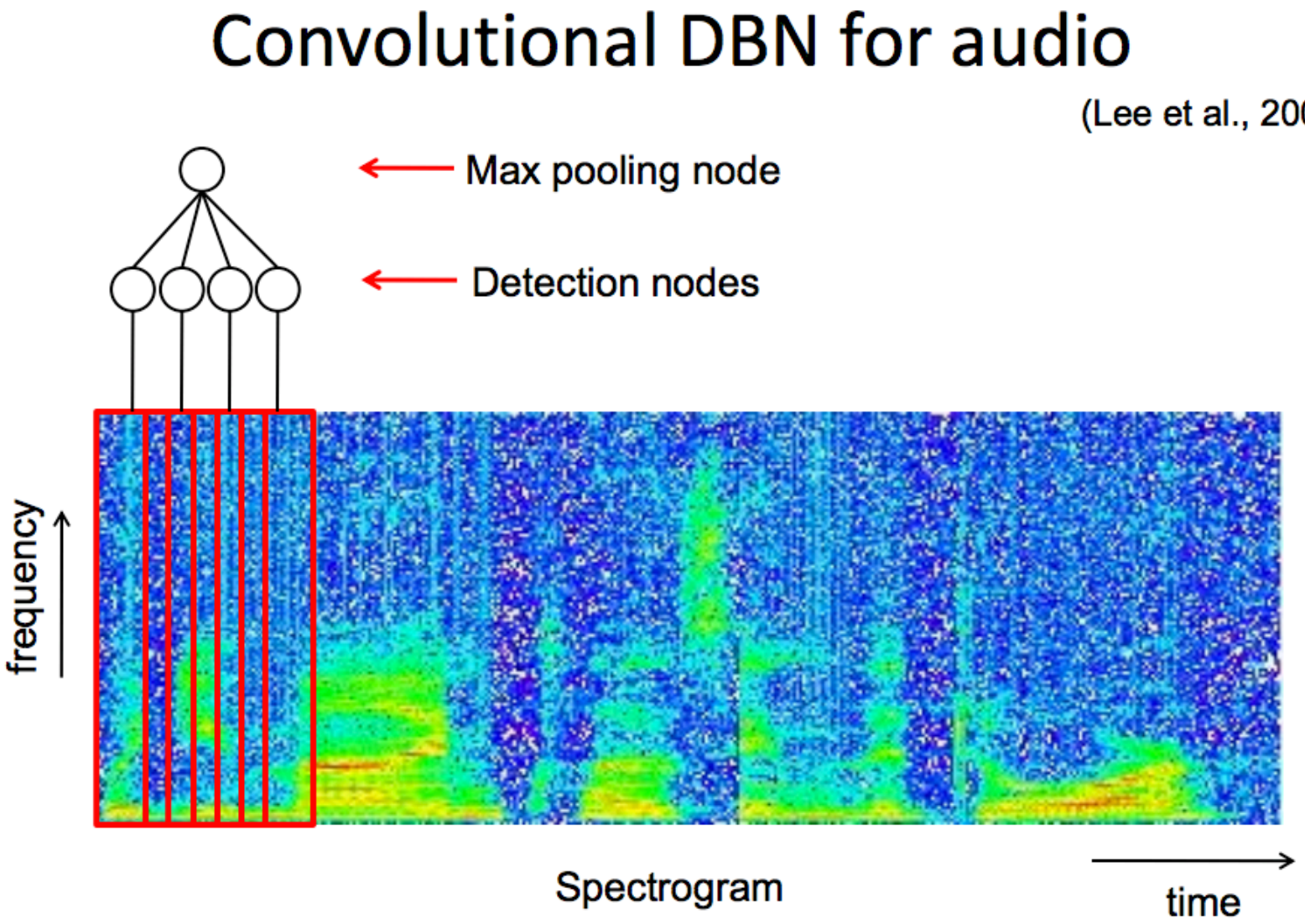
(LeCun et al., 1989)



Vision with ANNs



Similar technique applies to audio



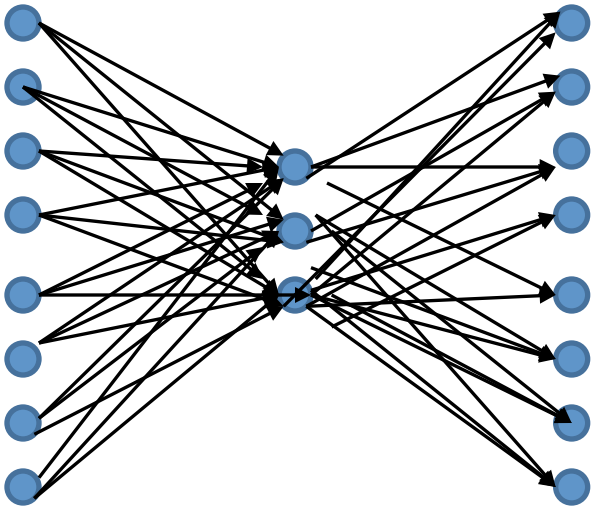
DENSITY MODELLING WITH ANNS

Neural network auto-encoding

- Assume we would like to learn the following (trivial?) output function:

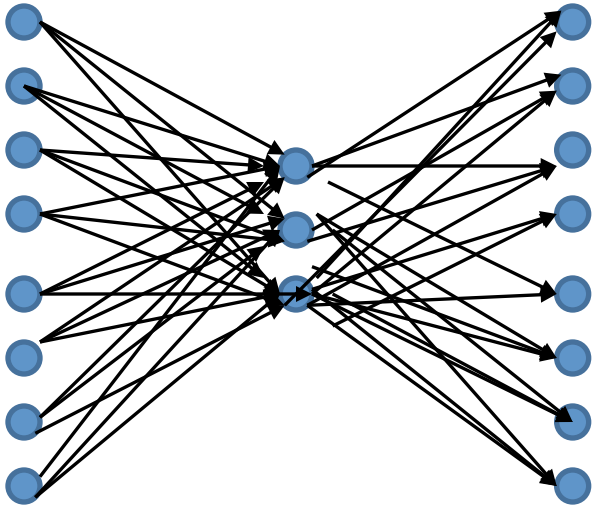
Input	Output
00000001	00000001
00000010	00000010
00000101	00000100
00001000	00001000
00010000	00010000
00100000	00100000
01000000	01000000
10000000	10000000

- Using the following network:



- Can this be done?

Learned parameters

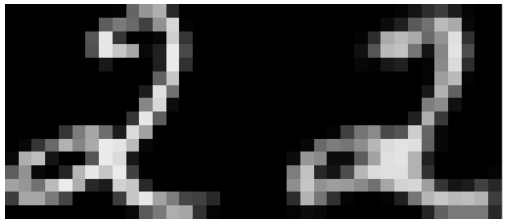


Note that each value is assigned to the edge from the corresponding input

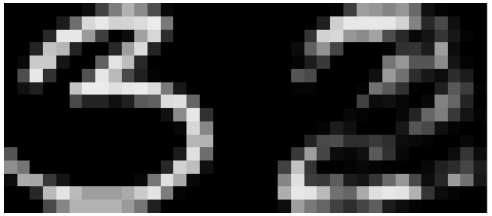
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Hypothetical example
(not actually from an autoencoder)

Data Reconstruction



Data Reconstruction



Neural network autoencoding

- The hidden layer is a *compressed version* of the data
- *Reconstruction error* on a vector *x* is related to P(*x*) on the probability that the auto-encoder was trained with.

- Denoising auto-encoders: trained to reconstruct **x** from a "noisy" version of **x**

Density Modeling with ANNs

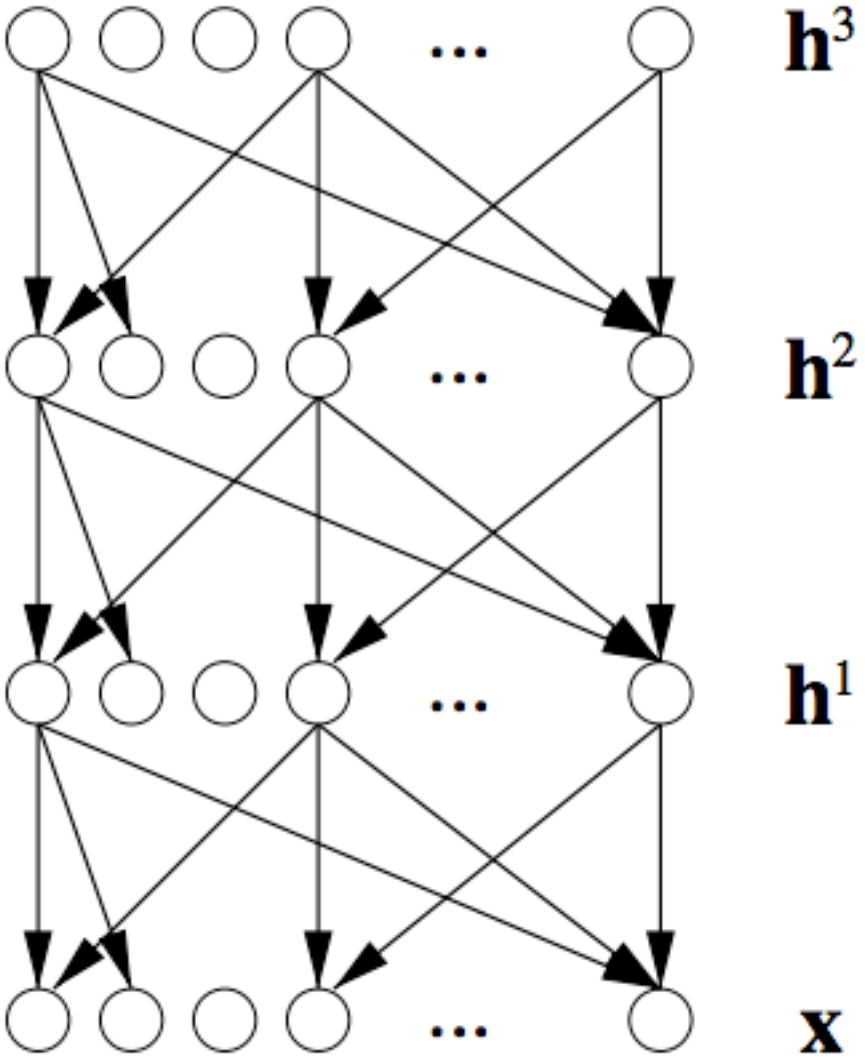
Sigmoid belief nets (Neal, 1992): explicitly model a generative process.

Top layer nodes are generated by a binomial.

Sampling from (generating data with) **a SBN**:

- Sample $\mathbf{h}^k$ from $P(\mathbf{h}^k)$
- Sample from $\mathbf{h}^{k-1}$ from $P(\mathbf{h}^{k-1} \mid \mathbf{h}^k)$
- …
- Sample $\mathbf{x}$ from $P(\mathbf{x} \mid \mathbf{h}^{1})$

What about learning?



Recall gradient for logistic regression

- This can be interpreted as a difference between the expected value of $y|x^j=1$ *in the data* and the expected value of $y|x^j=1$ *as predicted by the model*
- Gradient ascent tries to make those equal

$$\frac{\partial}{\partial w^j} \log P(D|\mathbf{w}) = \frac{1}{n} \sum_i (y_i - p_i) x_i^j =$$

$$= \frac{1}{n} \sum_{i:x_i^j=1} y_i - \frac{1}{n} \sum_{i:x_i^j=1} p_i$$

So: If we can *sample* from the model we can approximate the gradient.

$$= E_{\mathbf{x},y\sim Data}[y \mid \mathbf{x}] - E_{\mathbf{x}\sim Data,\mathbf{y}\sim \hat{P}(y|\mathbf{x})}[y \mid \mathbf{x}]$$

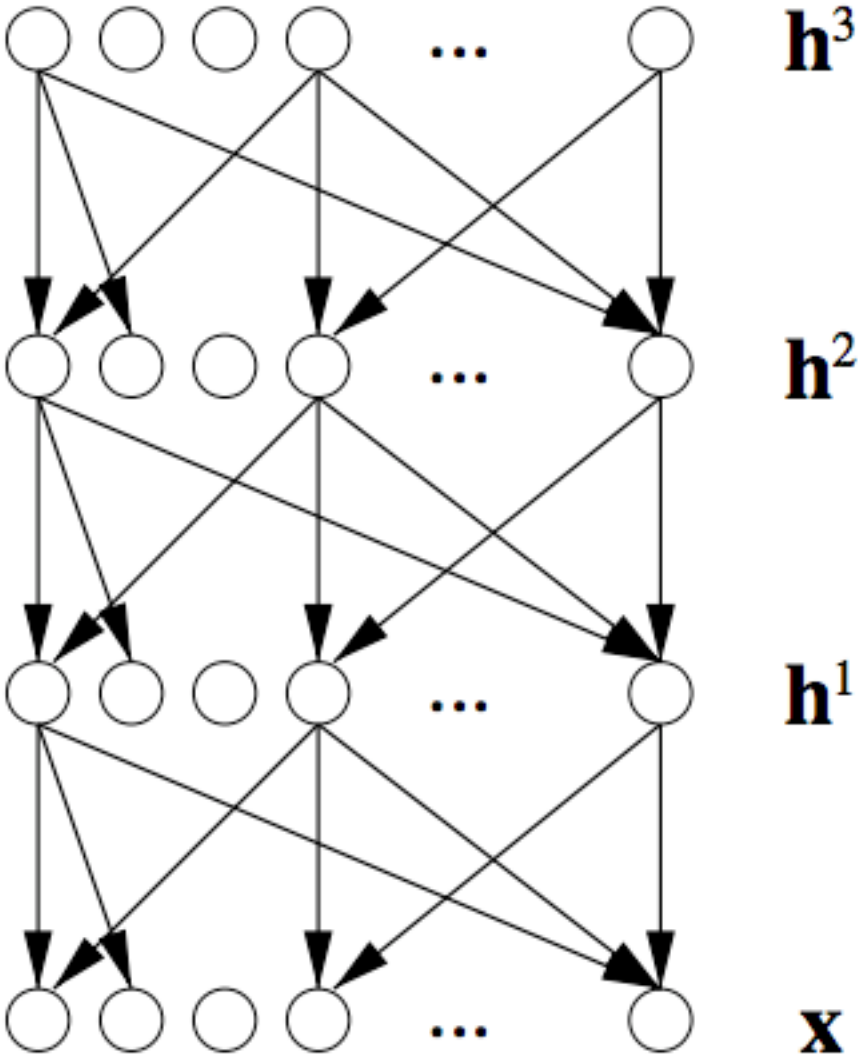
Density Modeling with ANNs

Sigmoid belief nets (Neal, 1992): explicitly model a generative process.

Top layer nodes are generated by a binomial.

Sampling from (generating data with) **a SBN**:

- Sample $\mathbf{h}^k$ from $P(\mathbf{h}^k)$
- Sample from $\mathbf{h}^{k-1}$ from $P(\mathbf{h}^{k-1} \mid \mathbf{h}^k)$
- …
- Sample $\mathbf{x}$ from $P(\mathbf{x} \mid \mathbf{h}^{1})$



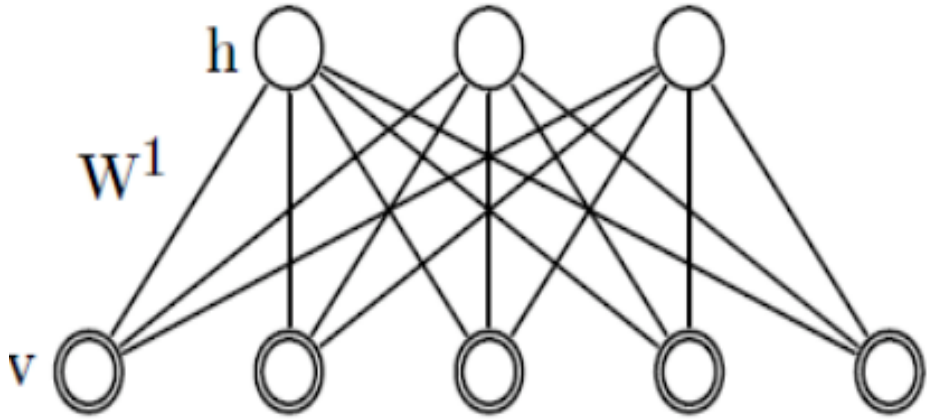
Density Modeling with ANNs

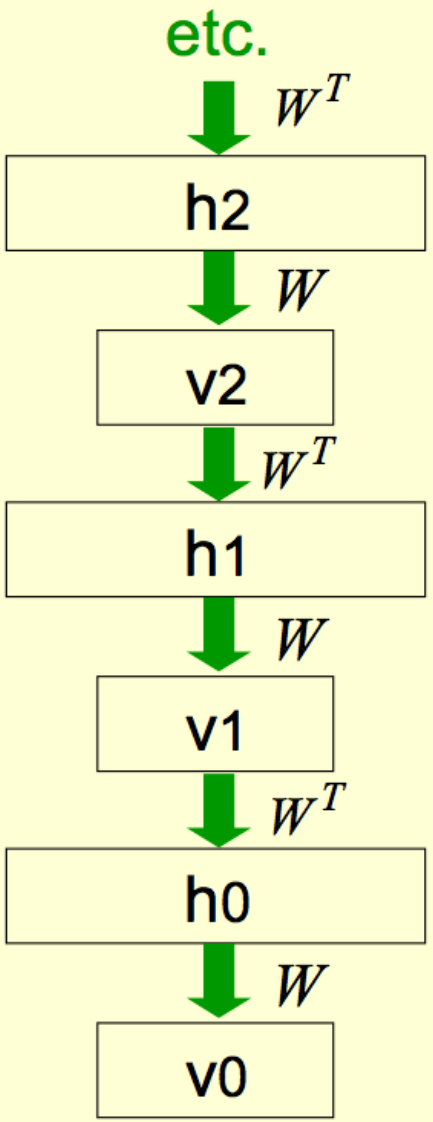
Notation: **x** is now "visible units" **v**

Restricted Boltzmann machine: equivalent to a very deep sigmoid belief network, with *lots* of weights tied.

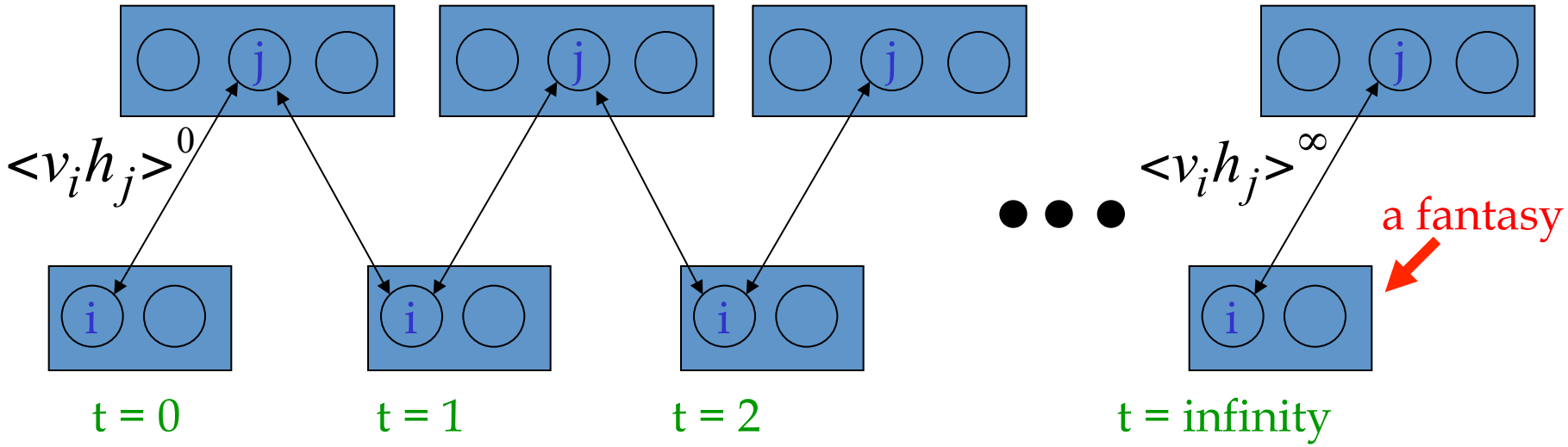
Usually show this as a two-layer network with one hidden layer and symmetric weights.

RBNs can compute Pr(**v**|**h**) or Pr(**h**|**v**) directly…





Computing probabilities with RBM via Gibbs sampling



Start with any value on the visible units.

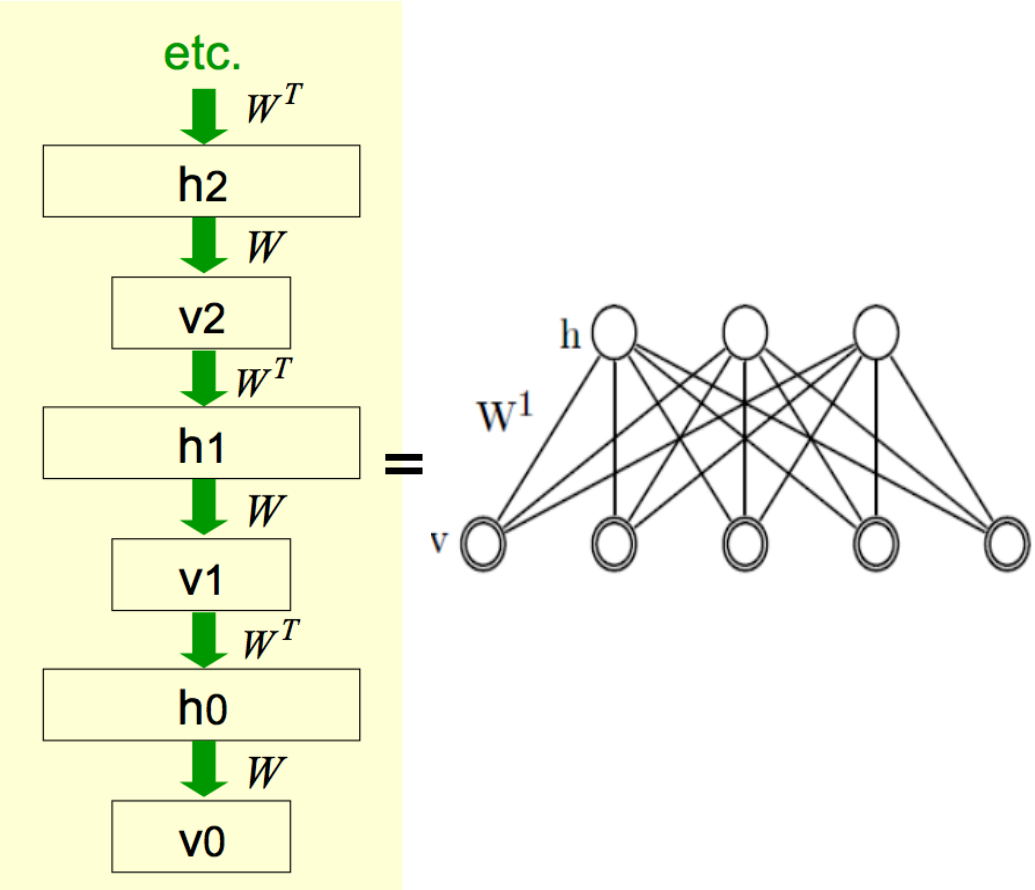
Then alternate between updating the hidden units (in parallel) and the visible units in parallel.

System will converge to a stream of <*v,h*> drawn from Pr*(v,h)*

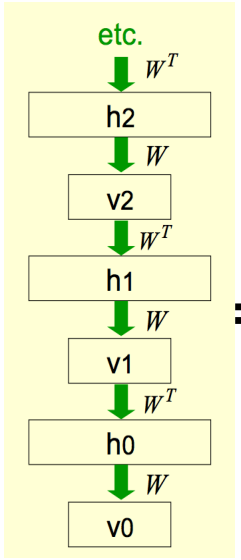
Goal of learning: push system's Pr*(v,h)* toward the observed probabilities…Start with observed **x,** let system "drift away", then train it to stay "closer" to the observed **x.**

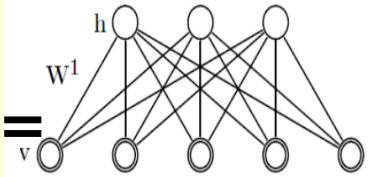
variant - "clamp" some units to a fixed value *c*, and sample from $Pr(v_1, h \mid v_2 = c)$

Training a stack of RBNs



Training a stack of RBNs



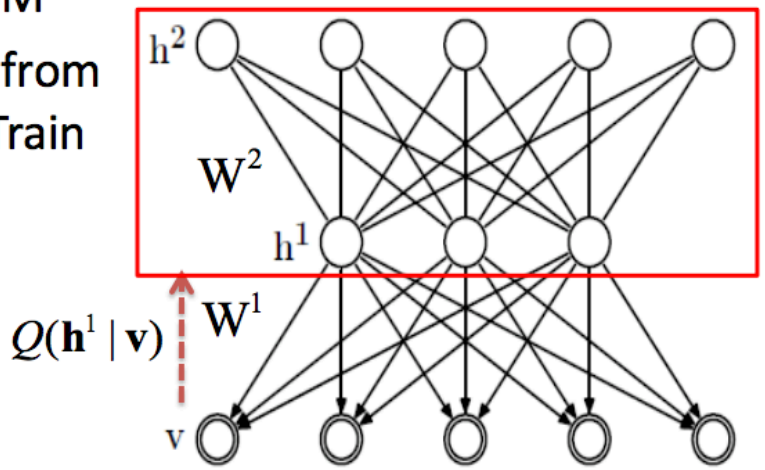


Step 1: train an RBN

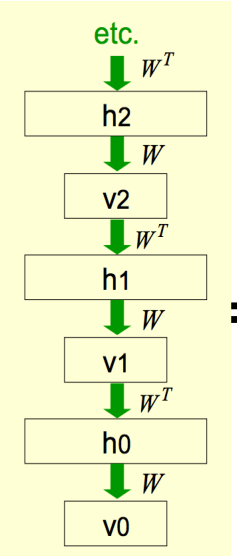
Step 2:

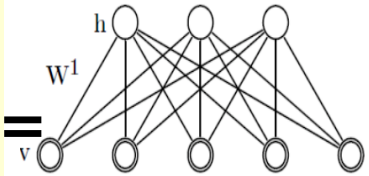
– Stack another hidden layer on top of the RBM to form a new RBM

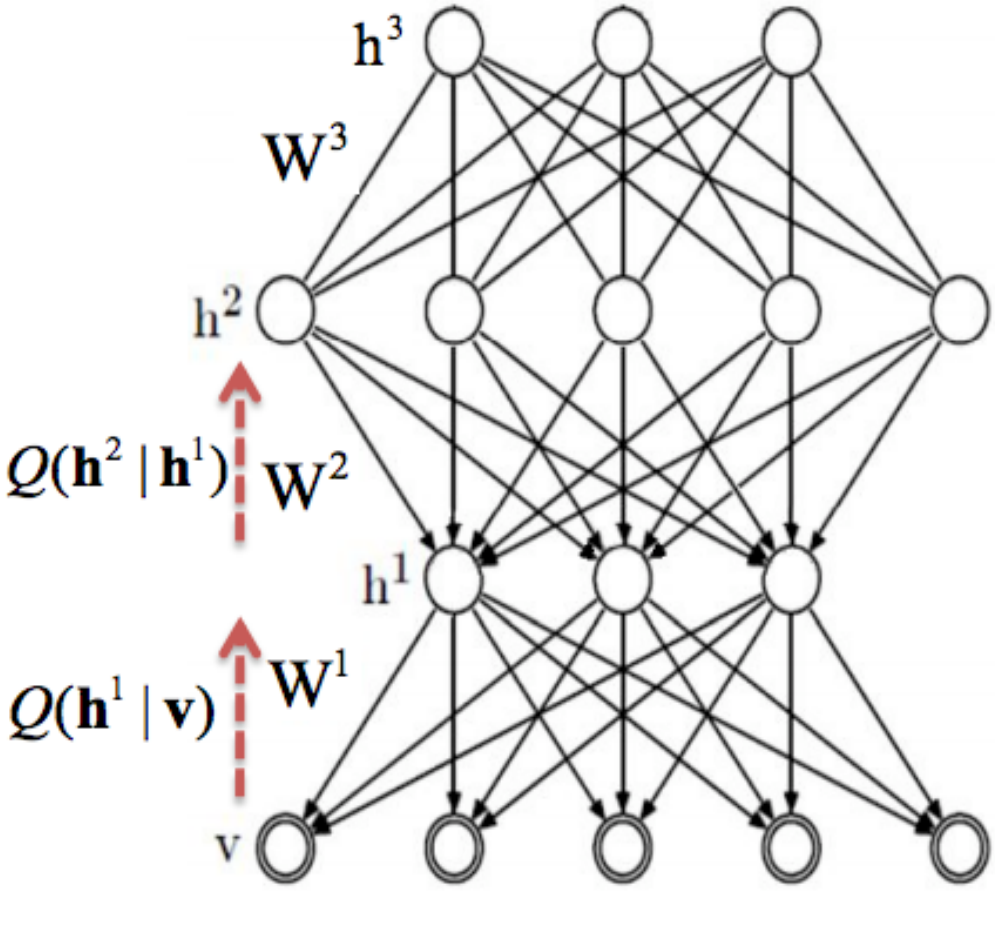
– Fix $\mathbf{W}^1$, sample $\mathbf{h}^1$ from $Q(\mathbf{h}^1 \mid \mathbf{v})$ as input. Train $\mathbf{W}^2$ as RBM.



Training a stack of RBNs a "deep Boltzmann machine"





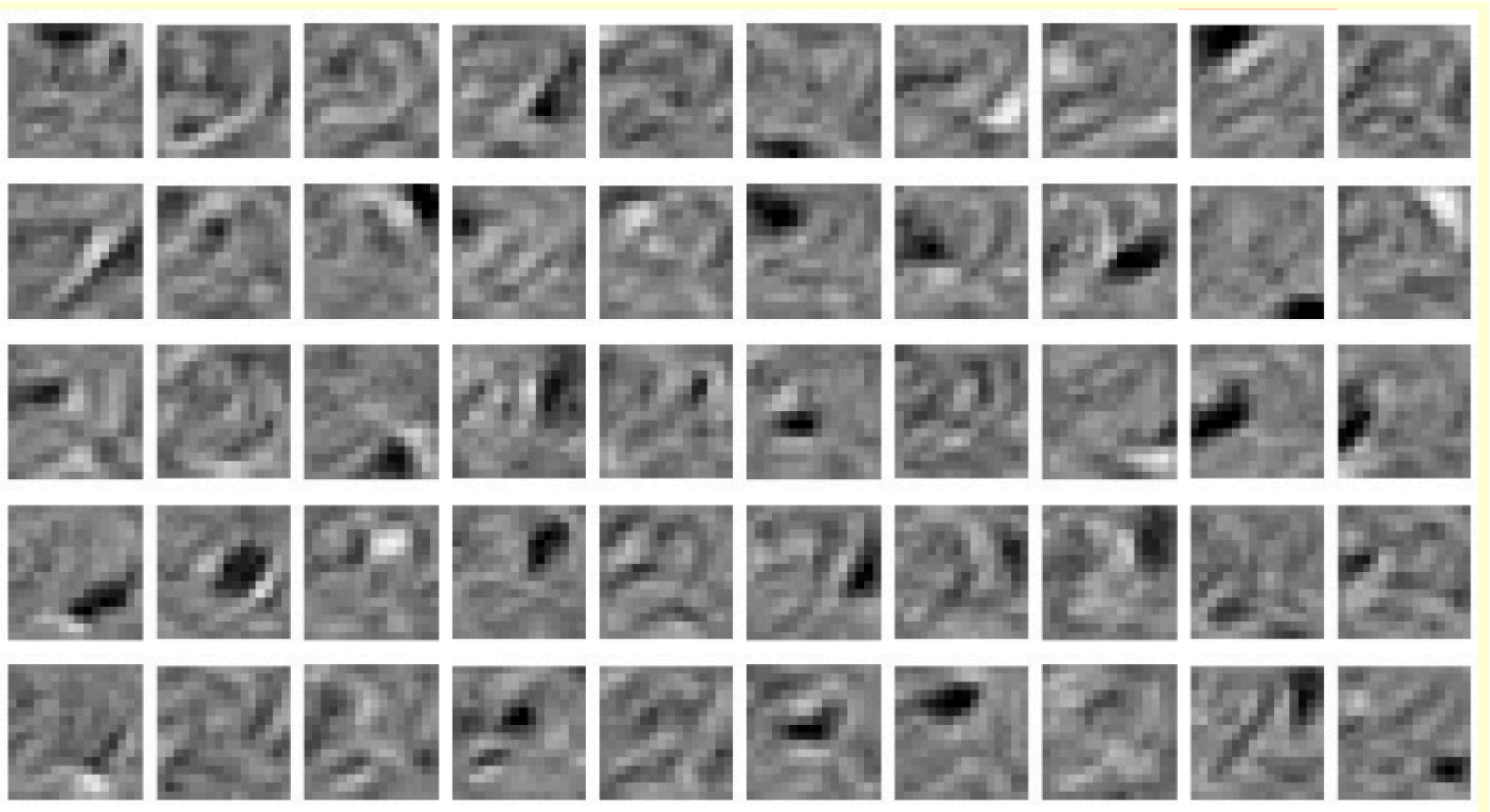


Steps 3, …: repeat as needed.

Then fine-tune with BP on a discriminative task: "clamp" n-1 of the inputs to **x** and use Gibbs to sample from $\Pr(y \mid \mathbf{x})$

DBN examples: learning the digit “2”

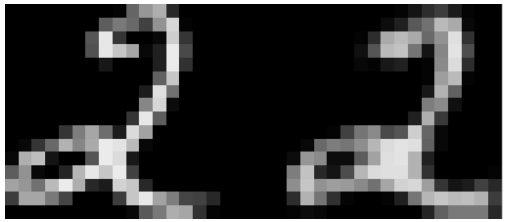
16 x 16 = 256 pixel image; 50 x 256 hidden units



Example: reconstructing digits with a network trained on different copies of "2"

Data

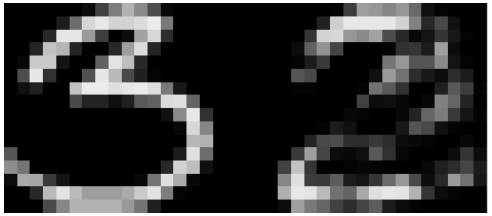
Reconstruction from activated binary features



New test images from the digit class that the model was trained on

Data

Reconstruction from activated binary features

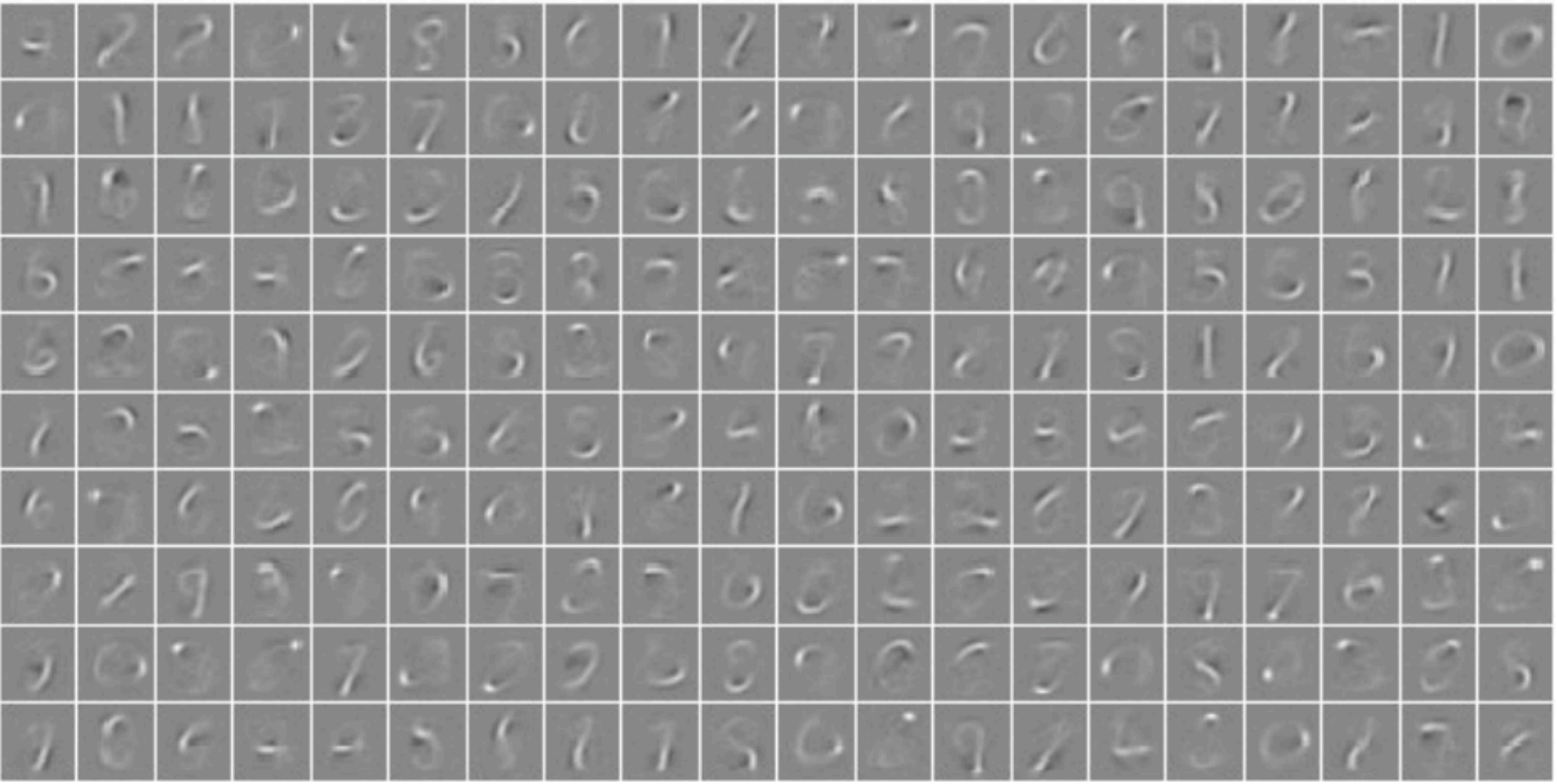


Images from an unfamiliar digit class (the network tries to see every image as a 2)

Samples generated after clamping the label *y*

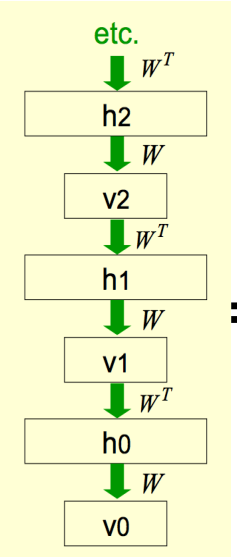
PSD features learned from all digits

[LeCunn]

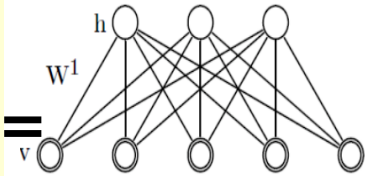


DENSITY MODELLING COMBINED WITH BACKPROP

Training a stack of RBNs a "deep Boltzmann machine"

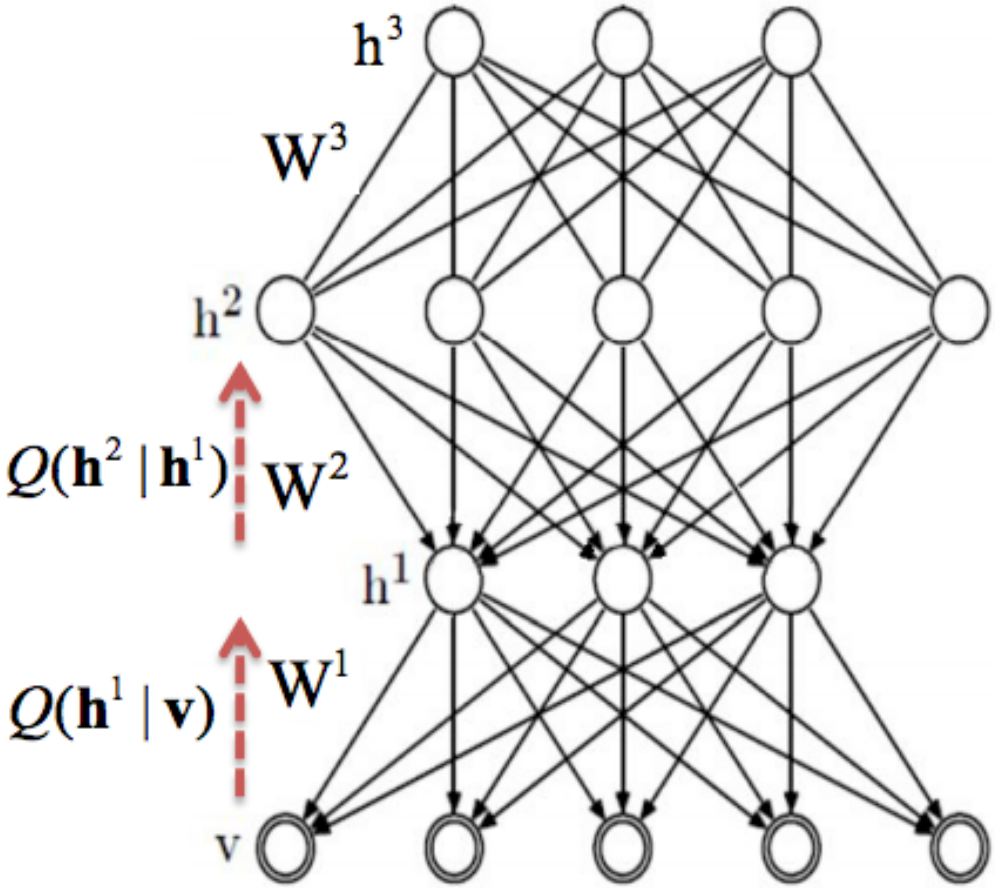


=



Steps 3, …: repeat as needed.

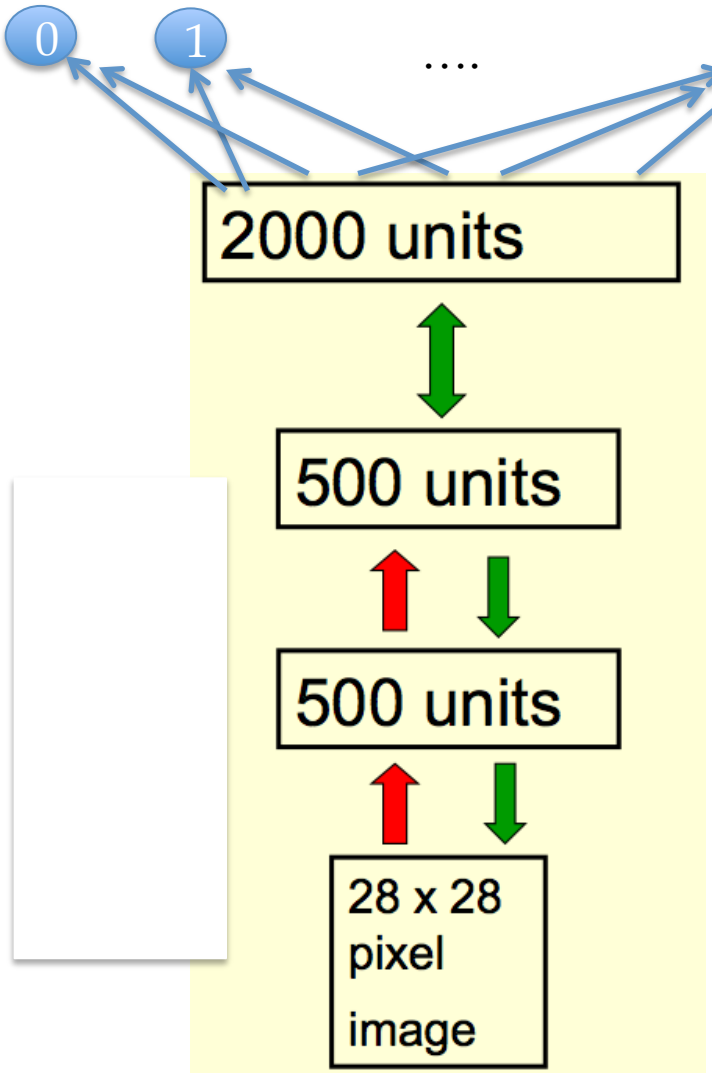
Then **fine-tune with BP on a discriminative task**: …



Deep learning methods

- Greedily learn a deep model of the unlabeled data.
  - Deep Boltzmann machines, denoising autoencoders, PSDs, …
  - Learn a hidden layer from the inputs;
  - Fix this layer and learn a deeper layer from it; …

- Then fine-tune with BP.
  - Another approach for RBNs: “unroll” the stack of RBNs into a multi-layer network, add appropriate output notes, and use it starting point for BP.



- Very carefully trained backprop net with one or two hidden layers (Platt; Hinton) — 1.6%
- SVM (Decoste & Schoelkopf, 2002) — 1.4%
- Generative model of joint density of images and labels (+ generative fine-tuning) — 1.25%
- Generative model of unlabelled digits followed by gentle backpropagation (Hinton & Salakhutdinov, Science 2006) — 1.15%

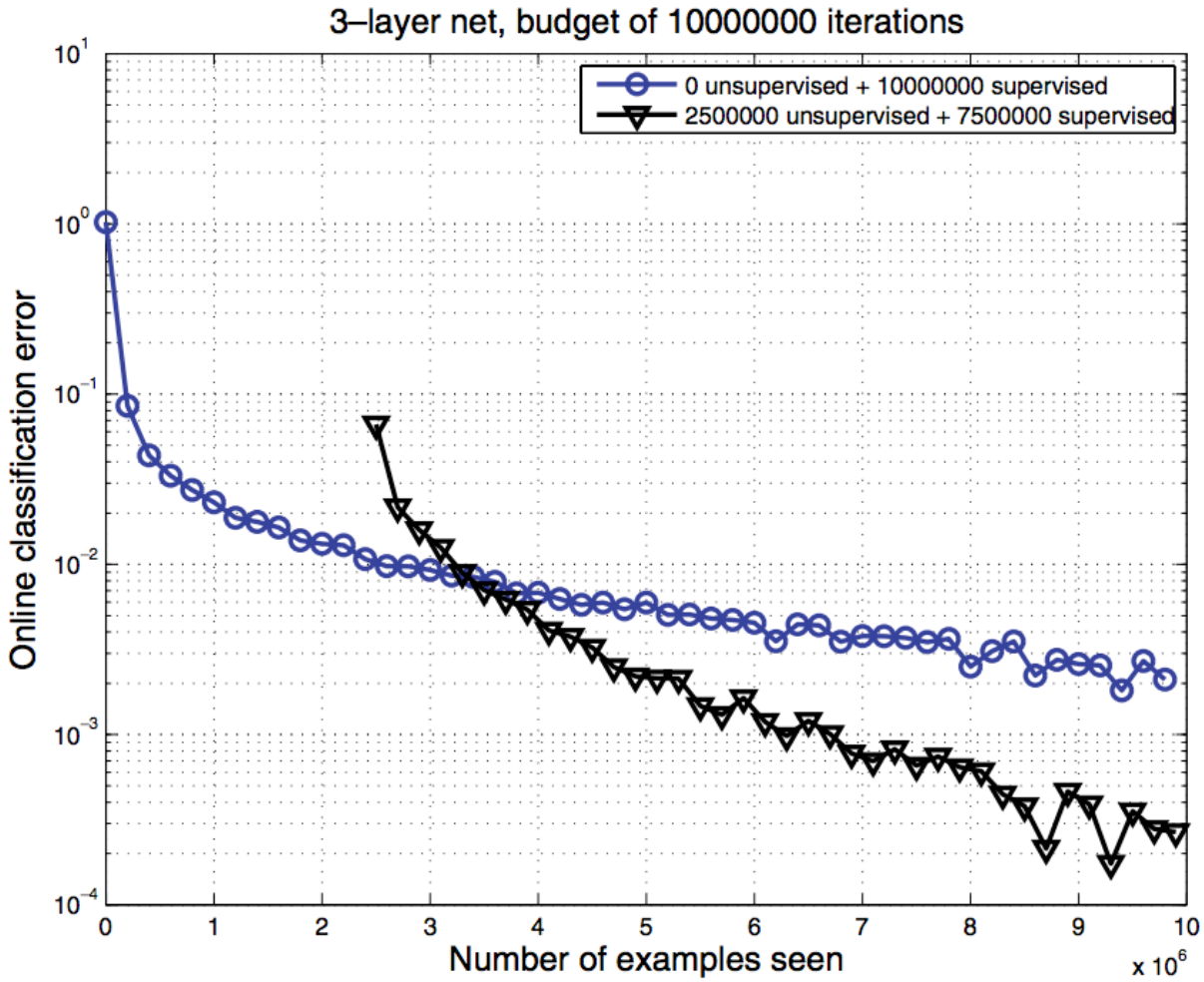


Fig. 4.2 Deep architecture trained online with 10 million examples of digit images, either with pre-training (triangles) or without (circles). The classification error shown (vertical axis, log-scale) is computed online on the next 1000 examples, plotted against the number of examples seen from the beginning. The first 2.5 million examples are used for unsupervised pre-training (of a stack of denoising auto-encoders). The oscillations near the end are because the error rate is too close to 0, making the sampling variations appear large on the log-scale. Whereas with a very large training set regularization effects should dissipate, one can see that without pre-training, training converges to a poorer apparent local minimum: unsupervised pre-training helps to find a better minimum of the online error. Experiments were performed by Dumitru Erhan.

[Hinton & Salakhutdinov, 2006]

Error reduced from 16% to 6% by using the "deep features".

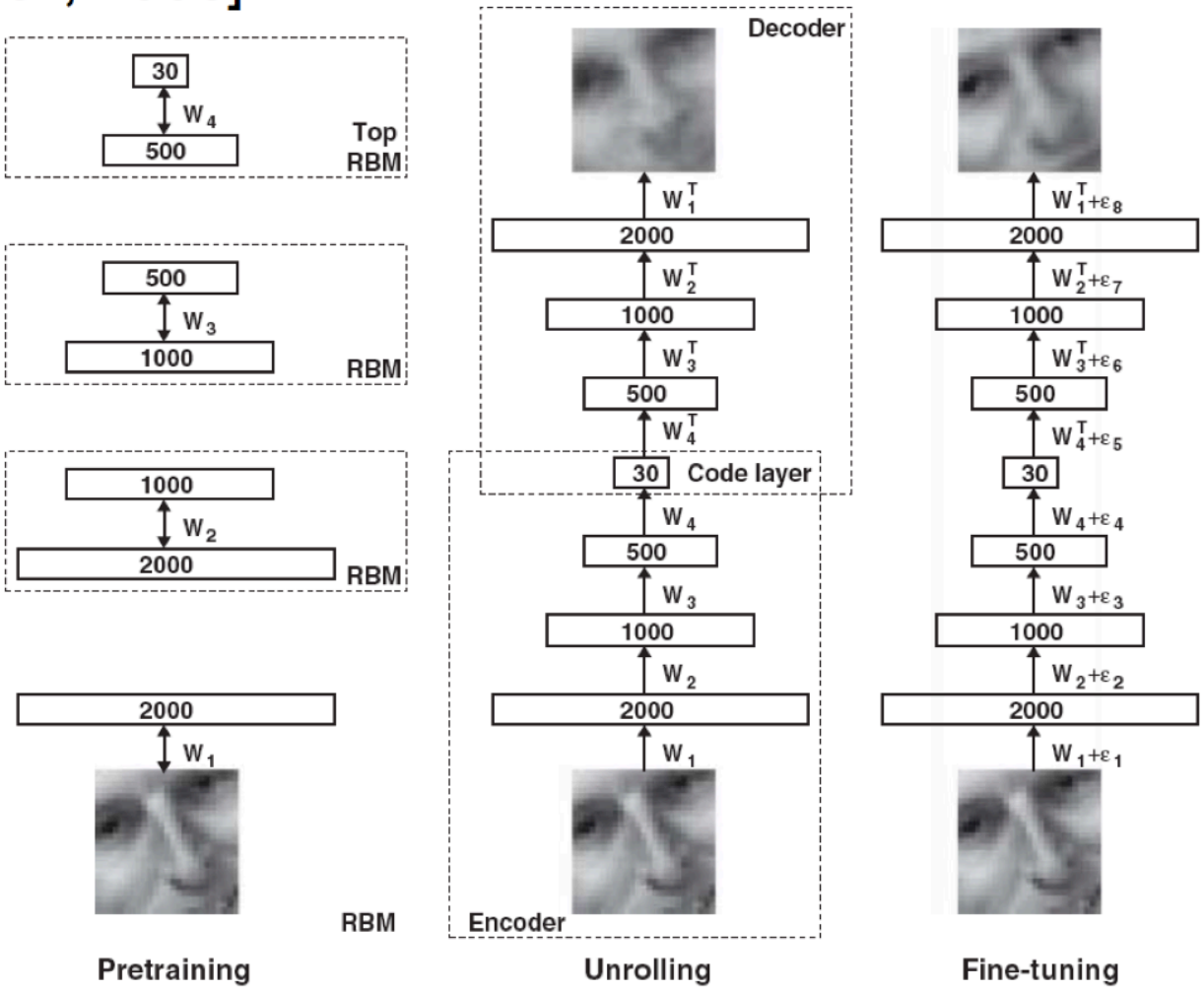
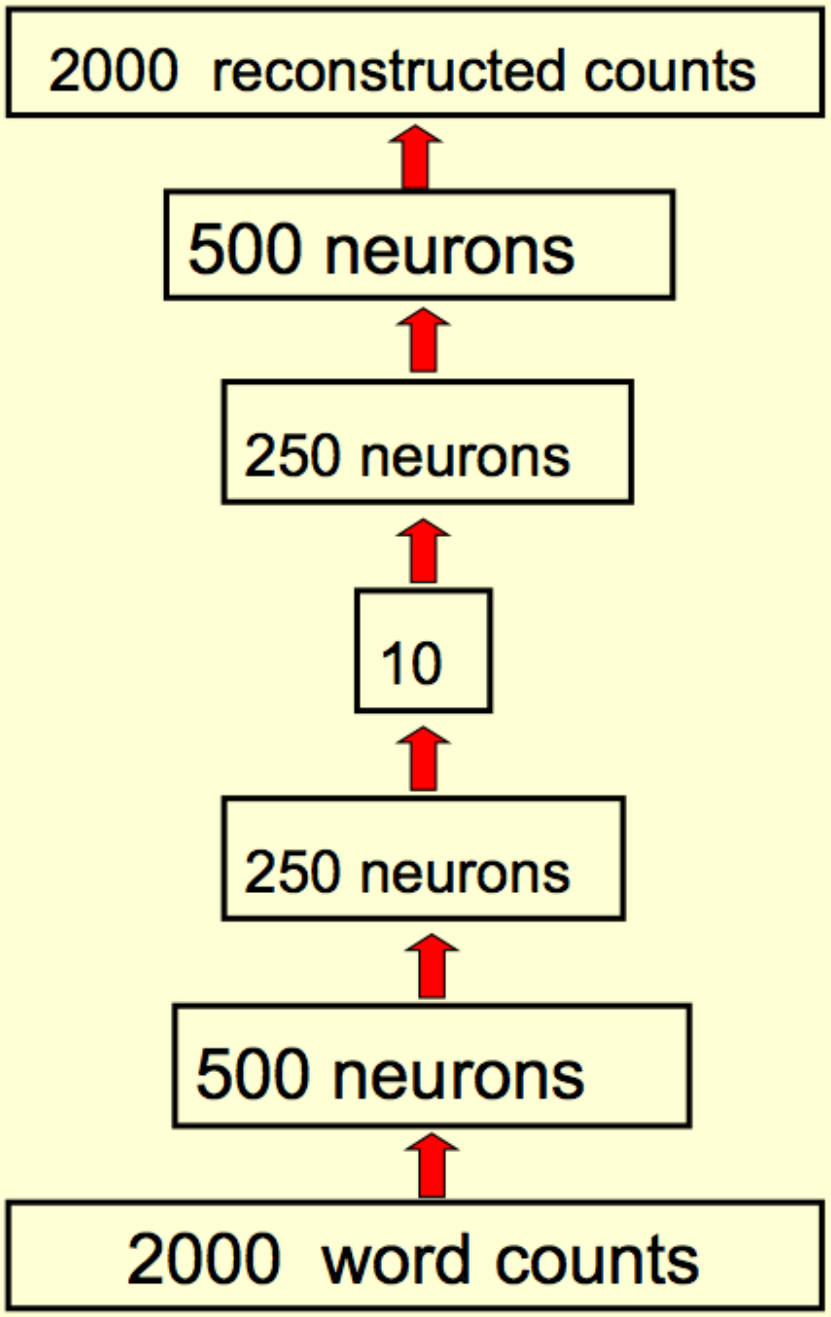


Fig. 1. Pretraining consists of learning a stack of restricted Boltzmann machines (RBMs), each having only one layer of feature detectors. The learned feature activations of one RBM are used as the "data" for training the next RBM in the stack. After the pretraining, the RBMs are "unrolled" to create a deep autoencoder, which is then fine-tuned using backpropagation of error derivatives.



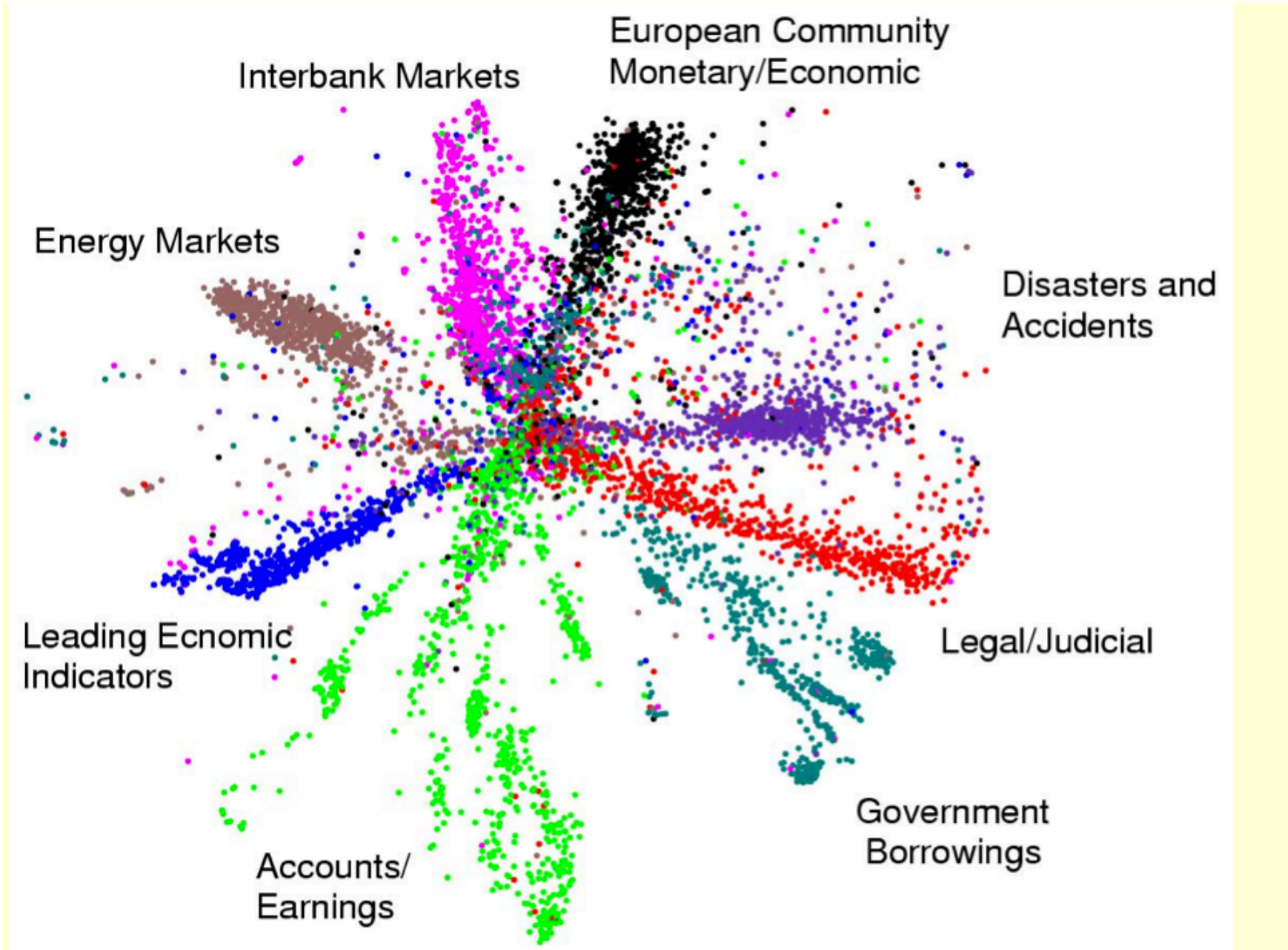
output vector

input vector

Modeling documents using top 2000 words.

- We train the neural network to reproduce its input vector as its output
- This forces it to compress as much information as possible into the 10 numbers in the central bottleneck.
- These 10 numbers are then a good way to compare documents.

First compress all documents to 2 numbers.
Then use different colors for different document categories



Very Large Scale Use of DBN's

[Quoc Le, et al., *ICML*, 2012]

Data: 10 million 200x200 unlabeled images, sampled from YouTube

Training: use 1000 machines (16000 cores) for 1 week

Learned network: 3 multi-stage layers, 1.15 billion parameters

Achieves 15.8% accuracy classifying 1 of 20k categories in ImageNet data

Images that most excite the feature:

Image synthesized to most excite the feature:

What you should know

- What a neural network is and what sorts of things it can compute
- Why it's not linear
- Backprop: loss function, updates, etc.
- Limitations of Backprop without pre-training on deep networks