# Deep belief nets

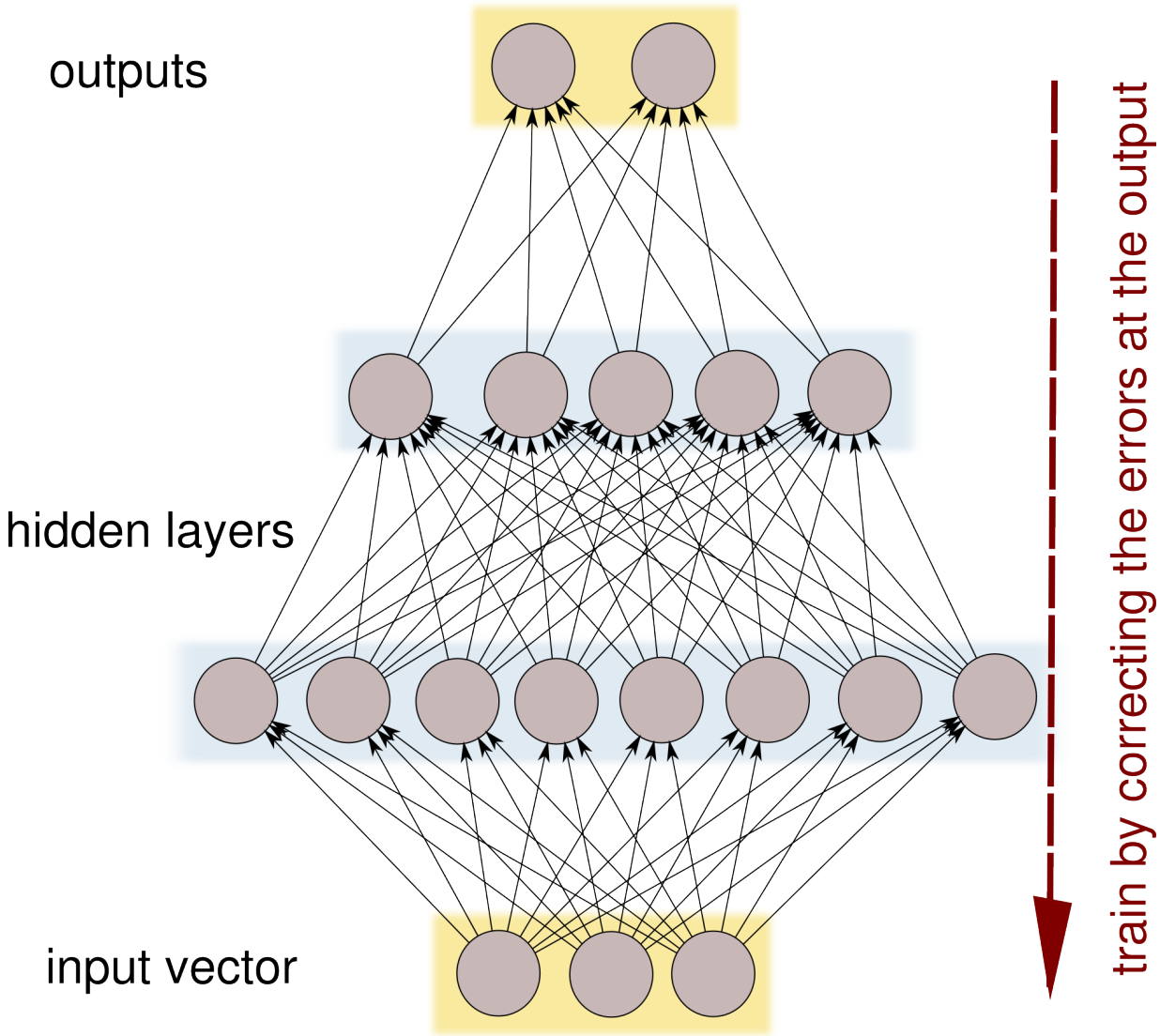
Marcus Frean
Victoria University of Wellington
Wellington, New Zealand
marcus.frean@vuw.ac.nz

# outline of this tutorial

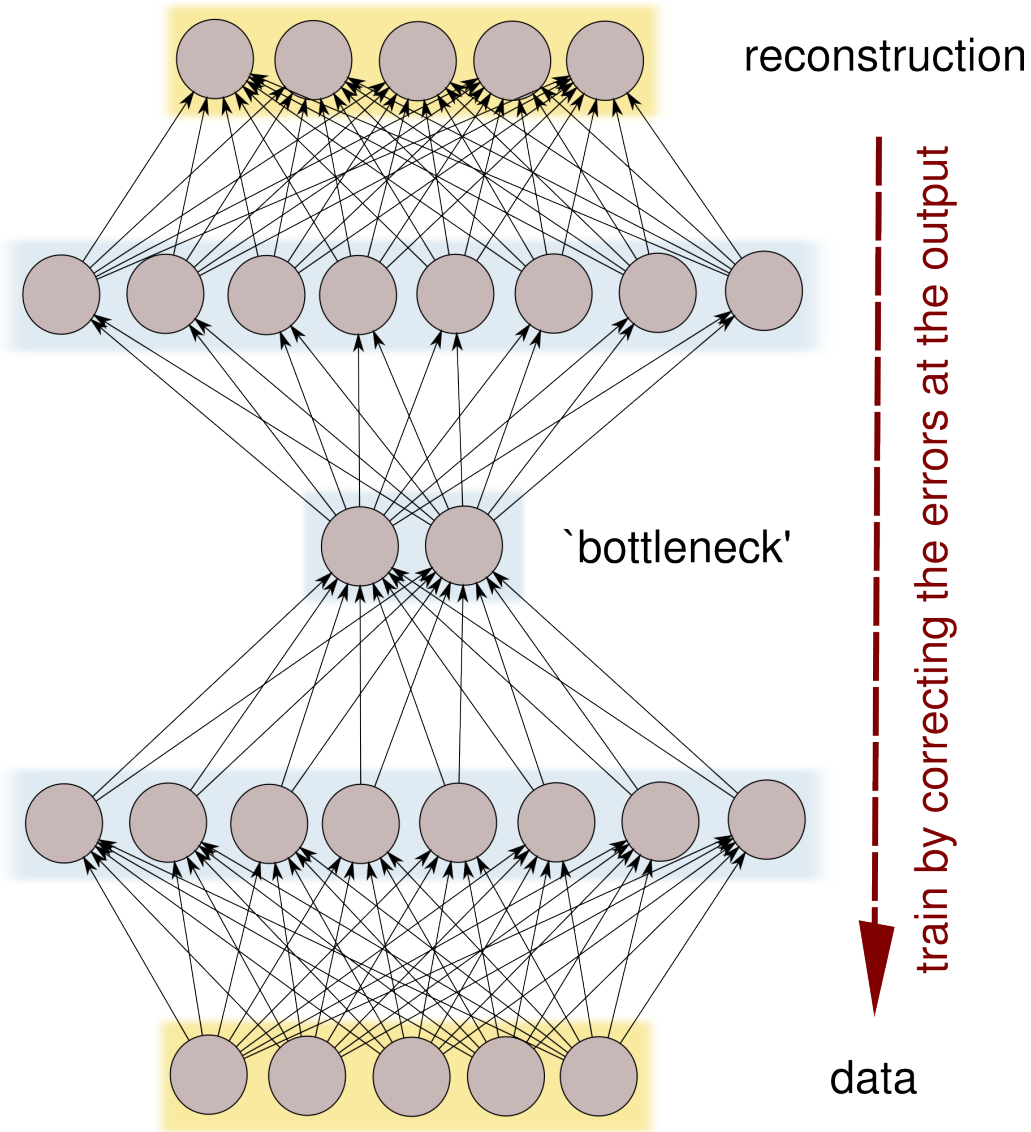
1. motivations
   - deep autoencoders
   - deep belief nets
2. sigmoid belief nets
   - why are they hard to train?
   - could layer-by-layer training work?
3. Boltzmann machines
   - why are they hard to train?
   - the restricted Boltzmann machine (RBM)
4. deep belief nets
   - how to do it
   - why it works
   - fine-tuning the result
   - 2 applications: a classifier and an autoencoder

comment: I've included some detailed maths in these slides for later reference...

# back-propagation networks

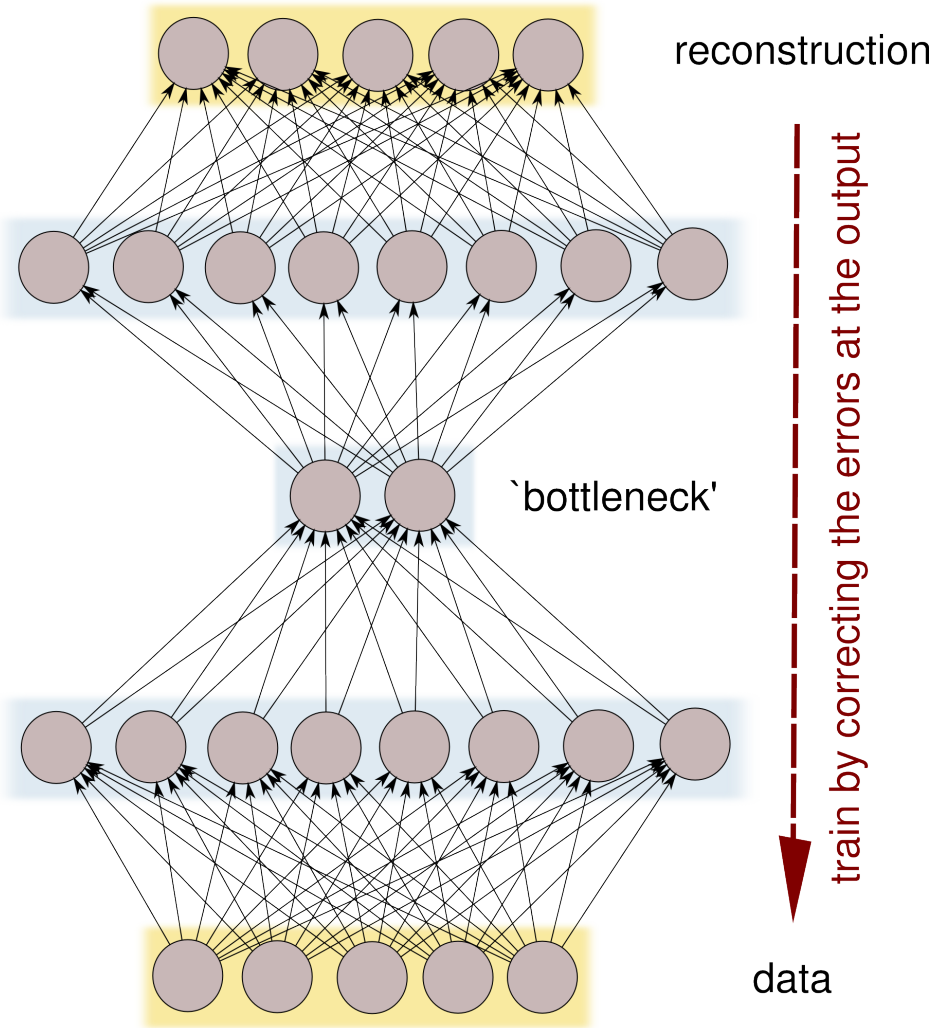


# auto-encoder nets



# why haven't deep auto-encoders worked?

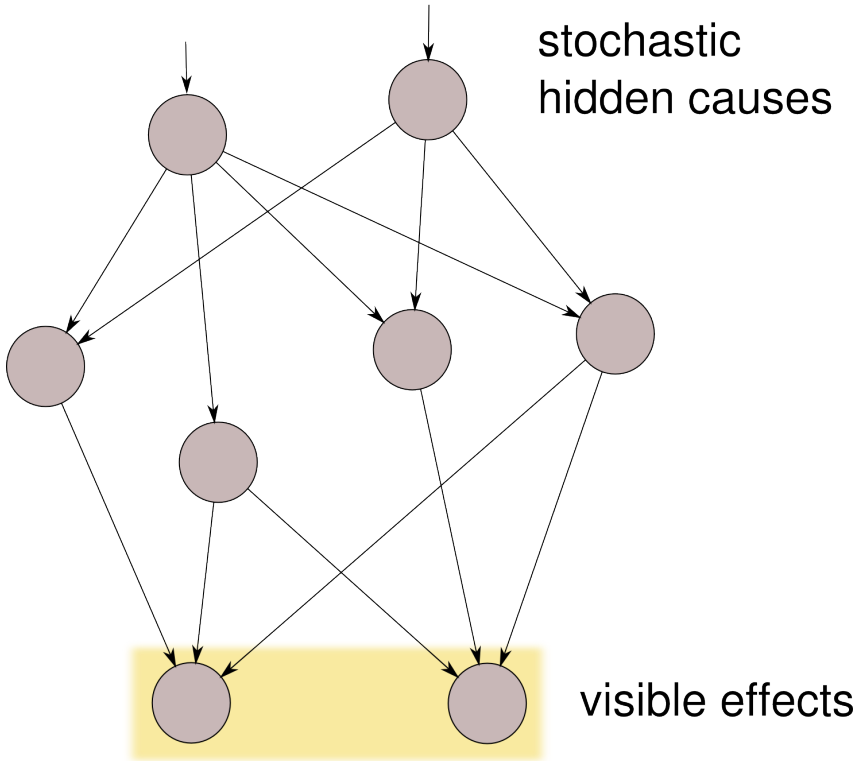
- all the hidden units interact
- the gradient gets tiny as you move away from the "output" layer



# belief nets

- A belief net is a directed acyclic graph composed of stochastic variables
- We get to observe some of the variables and would like to solve two problems:

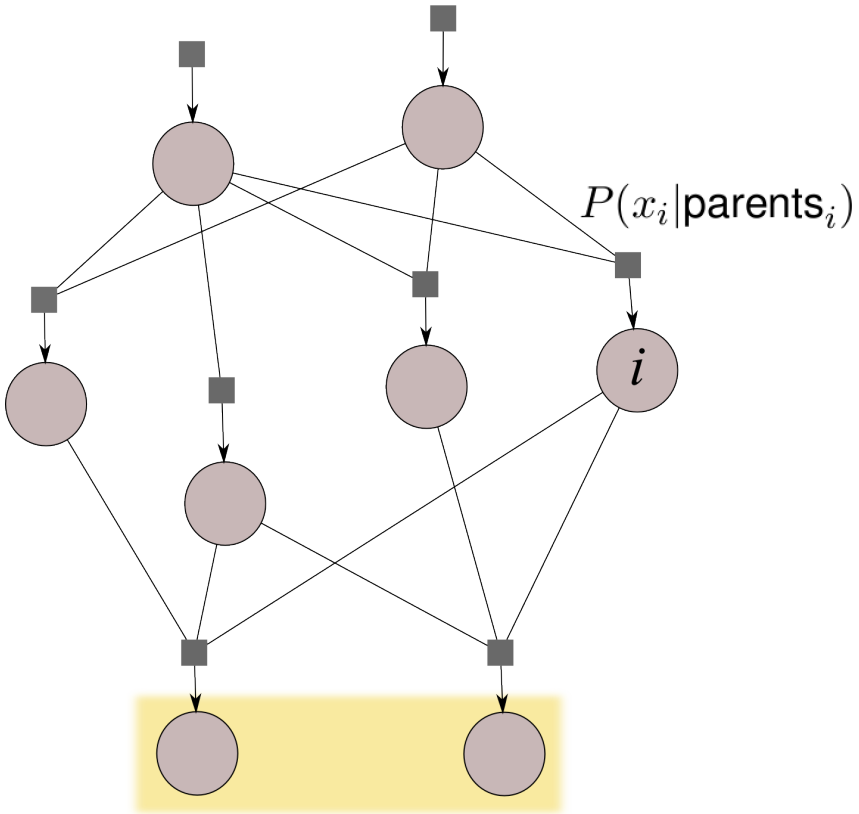
1. The inference problem: Infer the states of unobserved variables
2. The learning problem: Adjust the interactions between variables to make the network more likely to generate the observed data.



# factor graph of a belief network

In any graphical model, the joint probability is simplified to be a product of "factors" ■:

$$P(\mathbf{x}) = \prod_i ■_i$$



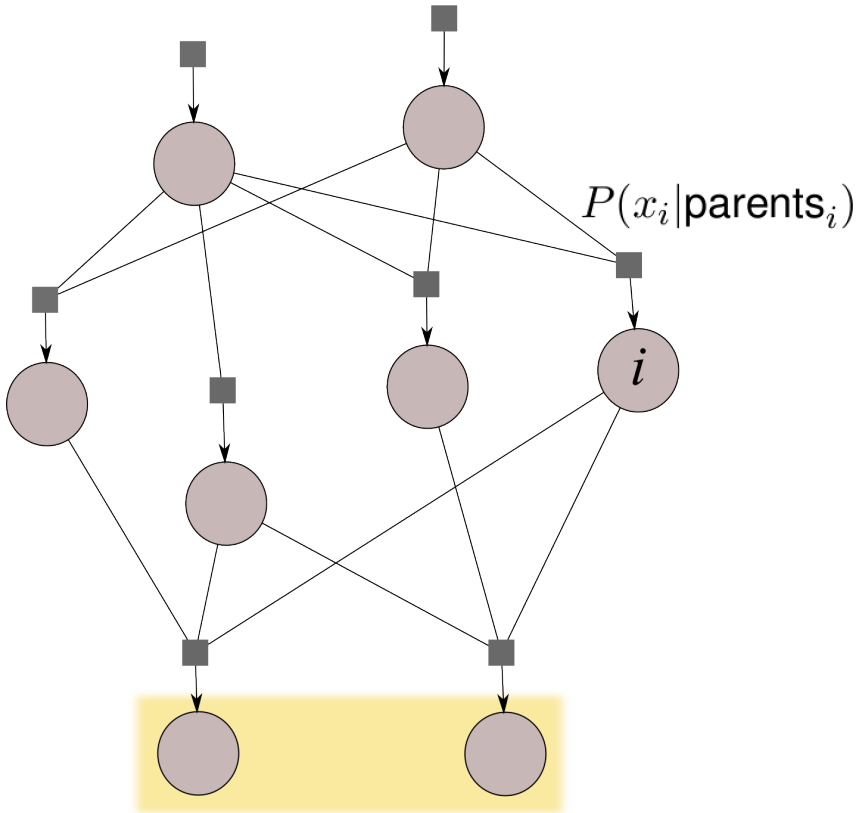

In a belief net (directed graphical model), those factors are conditional probabilities associated with each node:

$$■_i = P(x_i|\text{parents}_i)$$

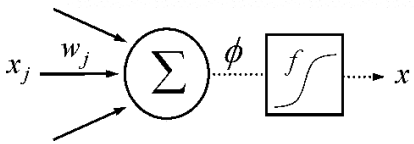
where $x_i$ is the "child" node in the graph.

# parameterized belief networks

- the factors in large belief nets are still too big/powerful to learn with finite data.
- we can parameterize the factors: *e.g.* the sigmoid function



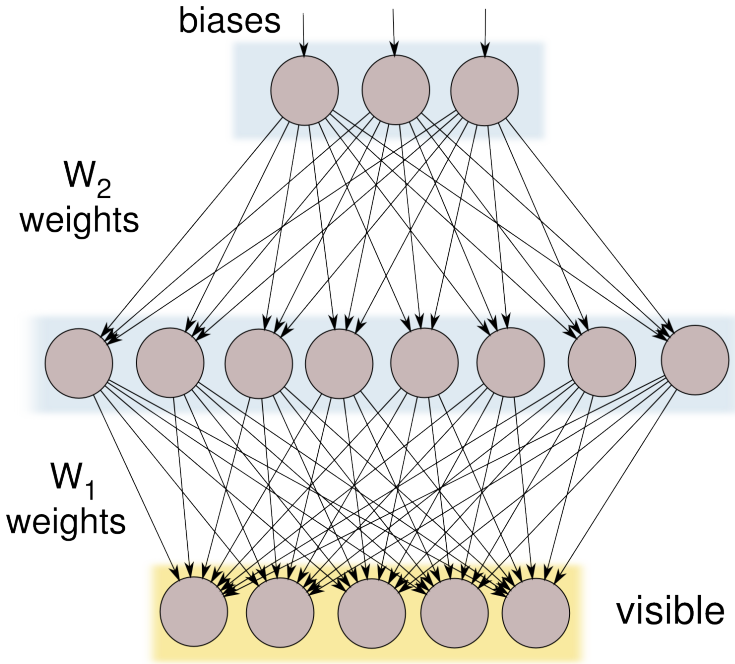

$$\blacksquare_i = P(x_i|\text{parents}_i)$$
$$= \frac{1}{1+e^{-\phi}}, \quad \text{for } x_i = 1$$




$\phi = \sum_j w_{ij} x_j$. This cuts down power of the "factor", exponentially.

# what would a really interesting generative model for (say) images look like?

- stochastic
- lots of units
- several layers
- easy to sample from



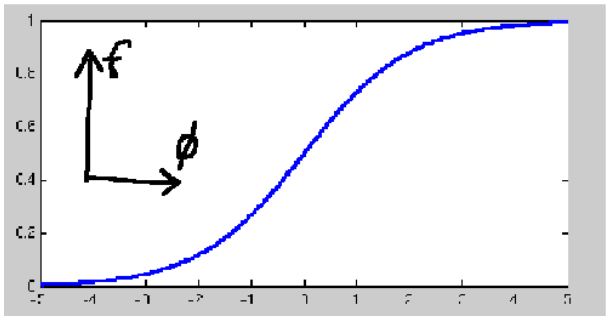

**sigmoid belief net**

an interesting generative model

# stochastic neurons

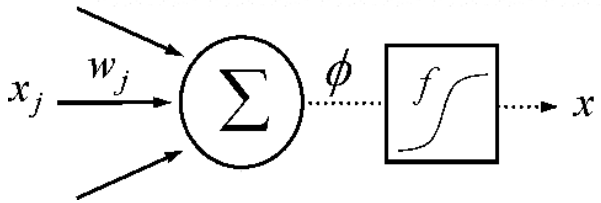
input to the $i^{\text{th}}$ neuron:

$$\phi_i = \sum_j w_{ji} x_j$$




probability of generating a 1:

$$p_i = \frac{1}{1 + \exp(-\phi_i)}$$




learning rule for making $\mathbf{x}$ more likely:

$$\Delta w_{ji} \propto (x_i - p_i) x_j$$

**for a network of these:**

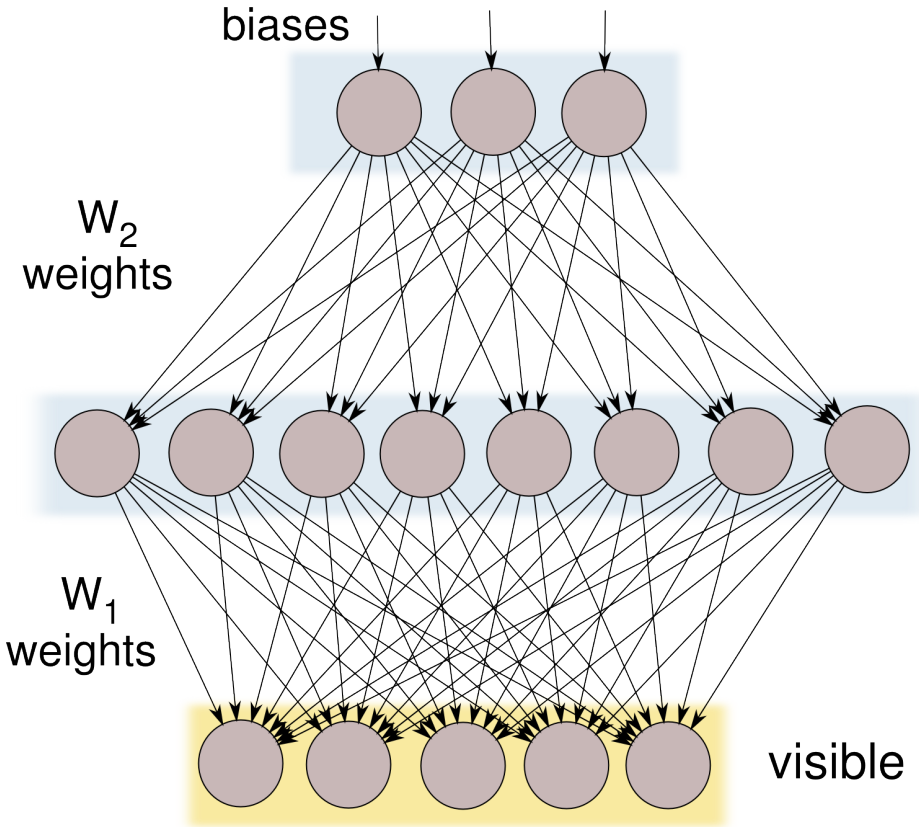
it is easy to make particular patterns more likely

# sampling from the joint

It is easy to sample from the prior $p(\mathbf{x}) = p(\mathbf{h}, \mathbf{v})$ of a sigmoidal belief net:

$$p(\mathbf{x}) = p(x_1, x_2 \dots x_n)$$
$$= \prod_i p(x_i | \text{parents}_i)$$

so we sample from each layer in turn, ending at the visible units.




Seems like an attractive generative model. How about learning?

# a general view of learning in graphical models

Notation: $\mathbf{x}$ for any node, $\mathbf{v}$ for visible, $\mathbf{h}$ for hidden.

- data set $\mathcal{D} = \{\mathbf{v}_n\}, n = 1 \ldots N$
- suppose $\mathbf{v}$ arise from a process involving a number of latent or "hidden" variables $\mathbf{h}$ that are stochastic, mediated by some unknown parameters $\mathbf{w}$
- For any given parameter settings $\mathbf{w}$, there is a joint distribution over $\mathbf{x} = (\mathbf{v}, \mathbf{h})$, namely $P(\mathbf{x} \mid \mathbf{w})$

  Absolutely *everything* that follows is going to be conditioned on $\mathbf{w}$ though, so we'll usually just drop the "$\mid \mathbf{w}$" from the r.h.s.

# normalised *vs* unnormalised probabilities

Denote probabilities by $P$, or by $P^\star$ if they are not yet normalised. For example,

$$P(\mathbf{v},\mathbf{h}) = \frac{P^\star(\mathbf{v},\mathbf{h})}{Z} \qquad \text{with} \quad Z = \sum_{\mathbf{v}}\sum_{\mathbf{h}} P^\star(\mathbf{v},\mathbf{h}) \tag{1}$$

The sum in $Z$ is over all *configurations* (all possible vectors $\mathbf{x}$), so in general it's likely to be intractable.

- In *some* cases of interest it is easy to ensure $P$ is normalized (*e.g.* directed graphical models / belief nets).
- But in many cases it's easy to specify a plausible $P^\star$ yet hard to find $Z$ and set $P$ (*e.g.* undirected graphical models)

# sum and product rules

Notice that by the **sum rule**,

$$P(\mathbf{v}) = \sum_{\mathbf{h}} P(\mathbf{v}, \mathbf{h}) \tag{2}$$

which is called the marginal probability of $\mathbf{v}$. The sum is over all configurations $\mathbf{h}$, so potentially it could be hard to compute in the same way that $Z$ is.

Another useful relation is given by the **product rule**:

$$P(\mathbf{v} \mid \mathbf{h}) = \frac{P(\mathbf{v}, \mathbf{h})}{P(\mathbf{v})} \tag{3}$$

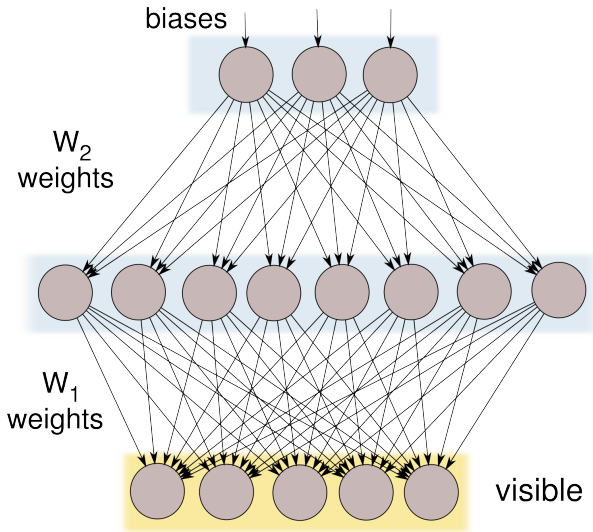
# remember..

The result we're going to get here is COMPLETELY GENERAL - it applies to *any* generative model having hidden variables.

But a particular example always helps:

*In a sigmoidal belief net...*

$$\frac{\partial}{\partial w_{ij}} \log P^{\star}(\mathbf{x}) = \underbrace{(x_i - p_i)x_j}_{\text{the "delta rule"}}$$




where $p_i$ is the probability of a 1, and is given by the sigmoid function.

# log likelihood of a dataset of $\mathbf{v}$

$$\begin{aligned}
\log L &= \log P(\mathcal{D}) \\
&= \sum_{\mathbf{v}\in\mathcal{D}} \log P(\mathbf{v}) \\
&= \sum_{\mathbf{v}\in\mathcal{D}} \log \left(P^\star(\mathbf{v})/Z\right) && \leftarrow \text{in terms of } P^\star \\
&= \sum_{\mathbf{v}\in\mathcal{D}} \left(\log P^\star(\mathbf{v}) \;-\; \log Z\right) \\
&\propto \underbrace{\frac{1}{N}\sum_{\mathbf{v}\in\mathcal{D}} \log P^\star(\mathbf{v})}_{\text{av. log likelihood per pattern}} \;-\; \log Z
\end{aligned}$$

**The trick** for finding the gradient of this: notice that

1. $\nabla_w \log P = (\nabla_w P)/P$ and conversely,
2. $\nabla_w P = P \nabla_w \log P$.

Each term uses this trick once, in each direction...

# gradient of the first term (average of $\log P^\star$)

$$\frac{\partial}{\partial w}\log P^\star(\mathbf{v}) = \frac{1}{P^\star(\mathbf{v})}\ \frac{\partial}{\partial w}P^\star(\mathbf{v}) \quad \leftarrow \text{via trick 1}$$

$$= \frac{1}{P^\star(\mathbf{v})}\ \frac{\partial}{\partial w}\sum_{\mathbf{h}} P^\star(\mathbf{v},\mathbf{h}) \quad \leftarrow \text{sum rule}$$

$$= \sum_{\mathbf{h}} \frac{1}{P^\star(\mathbf{v})}\ \frac{\partial}{\partial w}P^\star(\mathbf{v},\mathbf{h}) \quad \leftarrow \text{reordering}$$

$$= \sum_{\mathbf{h}} \frac{P^\star(\mathbf{v},\mathbf{h})}{P^\star(\mathbf{v})}\ \frac{\partial}{\partial w}\log P^\star(\mathbf{v},\mathbf{h}) \quad \leftarrow \text{via trick 2}$$

$$= \underbrace{\sum_{\mathbf{h}} P^\star(\mathbf{h}\mid\mathbf{v})}_{\text{av. over posterior!}}\ \frac{\partial}{\partial w}\log P^\star(\mathbf{x}) \quad \leftarrow \text{product rule}$$

# gradient of the second term ($\log Z$)

The second term is all about the normalisation factor, $Z$.
(NB. gradient will be *automatically* be zero in any belief net!)

$$\frac{\partial}{\partial w}\log Z = \frac{1}{Z}\frac{\partial}{\partial w}\sum_{\mathbf{v}}\sum_{\mathbf{h}} P^\star(\mathbf{v},\mathbf{h}) \qquad \leftarrow \text{trick 1}$$

$$= \frac{1}{Z}\sum_{\mathbf{v}}\sum_{\mathbf{h}} \frac{\partial}{\partial w} P^\star(\mathbf{v},\mathbf{h})$$

$$= \frac{1}{Z}\sum_{\mathbf{v}}\sum_{\mathbf{h}} P^\star(\mathbf{v},\mathbf{h})\frac{\partial}{\partial w}\log P^\star(\mathbf{v},\mathbf{h}) \qquad \leftarrow \text{trick 2}$$

$$= \underbrace{\sum_{\mathbf{v}}\sum_{\mathbf{h}} P(\mathbf{v},\mathbf{h})}_{\text{average over joint!}} \frac{\partial}{\partial w}\log P^\star(\mathbf{v},\mathbf{h}) \qquad \leftarrow \text{via eqtn 1}$$

# gradient as a whole

$$\frac{\partial}{\partial w} \log L \quad \propto$$

$$\underbrace{\frac{1}{N} \sum_{\mathbf{v} \in \mathcal{D}}}_{\text{data}} \underbrace{\sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v})}_{\text{av. over posterior}} \frac{\partial}{\partial w} \log P^\star(\mathbf{x}) \quad - \quad \underbrace{\sum_{\mathbf{v},\mathbf{h}} P(\mathbf{v}, \mathbf{h})}_{\text{av. over joint}} \frac{\partial}{\partial w} \log P^\star(\mathbf{x})$$

Both terms involve averaging over $\frac{\partial}{\partial w} \log P^\star(\mathbf{x})$.

Another way to write it:

$$\left\langle \frac{\partial}{\partial w} \log P^\star(\mathbf{x}) \right\rangle_{\mathbf{v} \in \mathcal{D},\ \mathbf{h} \sim P(\mathbf{h}|\mathbf{v})} \quad - \quad \left\langle \frac{\partial}{\partial w} \log P^\star(\mathbf{x}) \right\rangle_{\mathbf{x} \sim P(\mathbf{x})}$$

| clamped / wake phase | unclamped / sleep / free phase |
|---|---|
| ↑↑↑ conditioned hypotheses | ↓↓↓ random fantasies |

# example: sigmoid belief nets

For a belief net the joint is automatically normalised: $Z$ is a constant 1

- 2nd term is zero!
- for the weight $w_{ij}$ from $j$ into $i$, the gradient $\frac{\partial \log L}{\partial w_{ij}} = (x_i - p_i)x_j$
- stochastic gradient ascent:

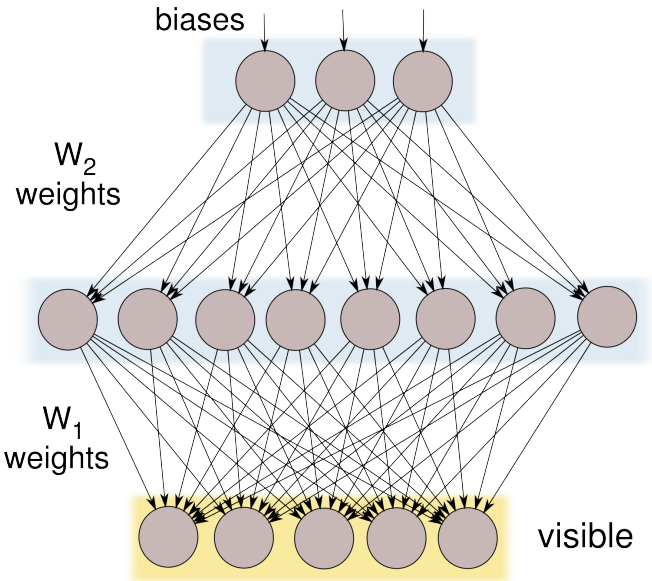
$$\Delta w_{ij} \propto \underbrace{(x_i - p_i)x_j}_{\text{the "delta rule"}}$$

So this is a stochastic version of the EM algorithm, that you may have heard of. We iterate the following two steps:

**E step:** get samples from the posterior

**M step:** apply the learning rule that makes them more likely

# but...



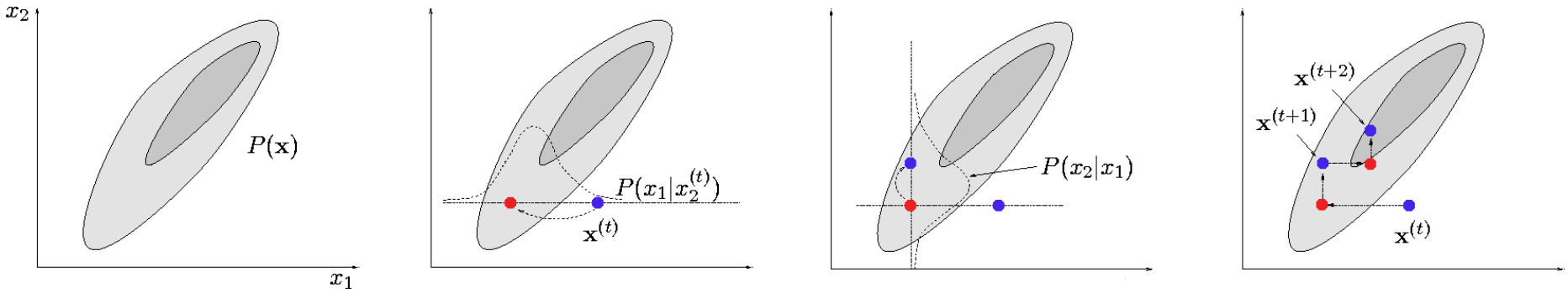

we need to apply this using samples from the *posterior*, not the *prior*

- how to draw such samples?
- filter some "ancestral" samples? *no!*
- Gibbs sampling? *maybe...*

# Gibbs sampling

To draw samples from $p(\mathbf{x}) = p(x_1, x_2, \ldots, x_n)$:

1. choose $i$ at random
2. choose $x_i$ from $p(x_i|\mathbf{x}_{\setminus i})$

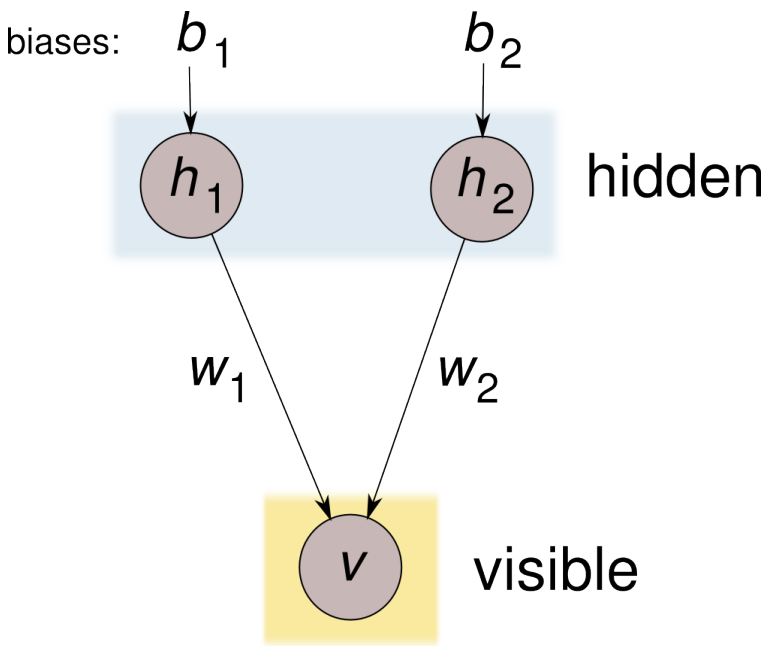



This results in a Markov Chain.

Running the chain for long enough $\longrightarrow$ samples from $p(\mathbf{x})$.

# Gibbs sampling in sigmoid belief nets

So what does $p(x_i|\mathbf{x}_{\setminus i})$ look like, in a sigmoid belief net?




- two hidden "causes"
- visible node is observed...
- Gibbs sampling for $h_1, h_2$:

$$p(h_1 = 1|v = 1)$$

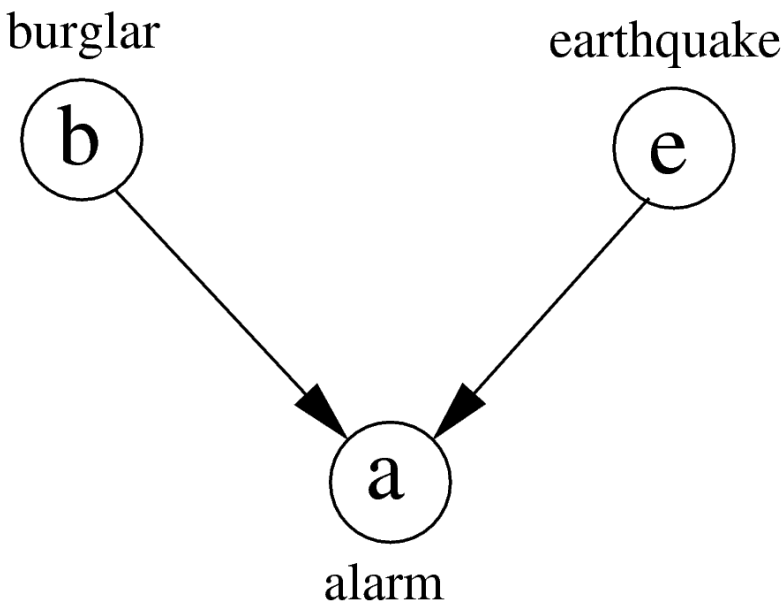
$$= \left[1 + \frac{(1 - f(b_1))\, f(w_2)}{f(b_1)\, f(w_1 + w_2)}\right]^{-1}$$

$$= \text{yuck!}$$

**Gibbs sampler in sigmoid belief net:**

- slow, ugly
- reason is 'explaining away'

# explaining away




Hidden states are

- independent in the *prior*
- dependent in the *posterior*

That dependence means sampling from one hidden unit has to cause a change in how all other hidden units update their states.

But we are interested in nets with *lots* of hidden units.

**an inconvenient truth:**

there's no quick way to draw a sample from $p(\text{hidden}|\text{visible})$

# SUMMARY: sigmoidal belief nets are...

easy to sample from as a generative model, but hard to learn

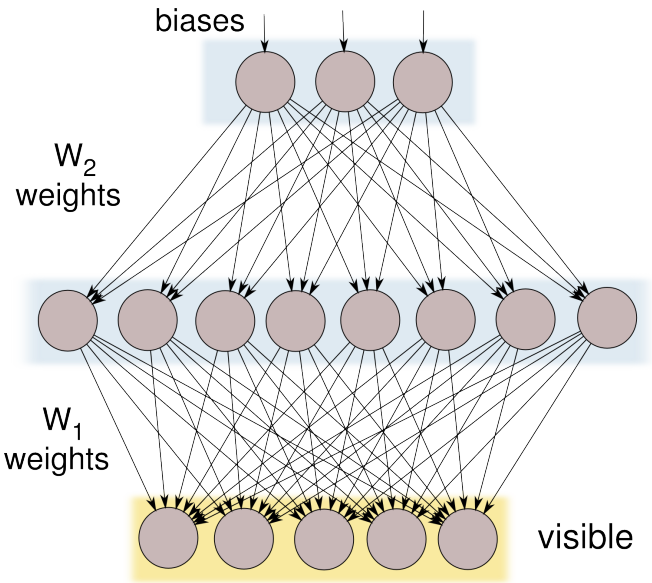
1. sampling from the posterior over hidden states is slow, due to explaining away. This also makes them hard to use for recognition.

2. 'deep' layers learn nothing until the 'shallow' ones have settled, but shallow layers have to learn while being driven by the deep layers (chicken and egg...)

*Let's ignore the quibbles about slowness for a moment, and consider this idea: one way around the second difficulty might be to train the first layer as a simple BN first, and then the second layer, and so on.*

# building a deep BN layer-by-layer

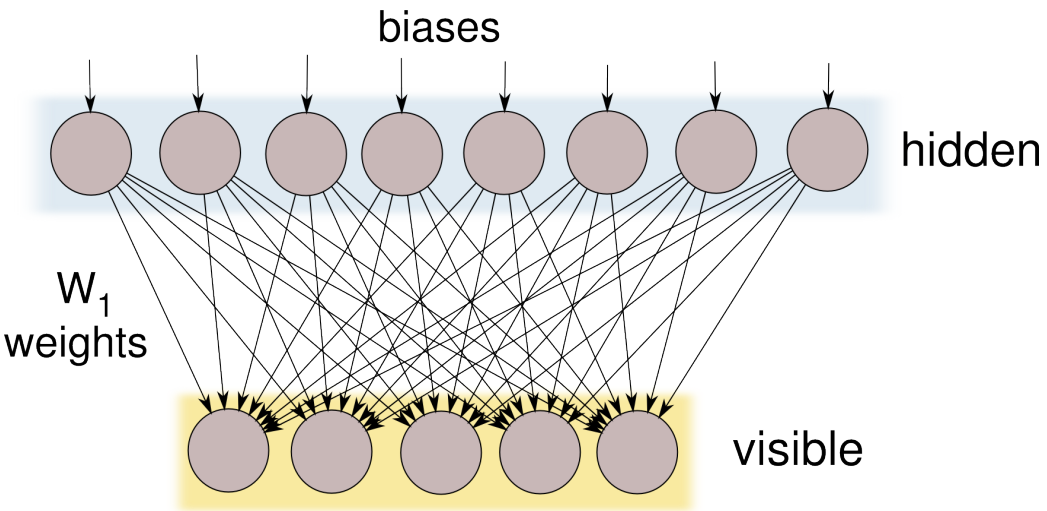
Here's a way to train a multilayer sigmoid belief net:




1. start with a single layer only. The hidden units are driven by bias inputs only. Train to maximize the likelihood of generating the training data.
2. freeze the weights in that layer, and replace the hidden units' bias inputs by a second layer of weights.
3. train the second layer of weights to maximize the likelihood.
4. and so on...

Question: what should the training set be for the second layer?

# aggregate posterior




Averaging over the training set, we have an aggregate posterior distribution:

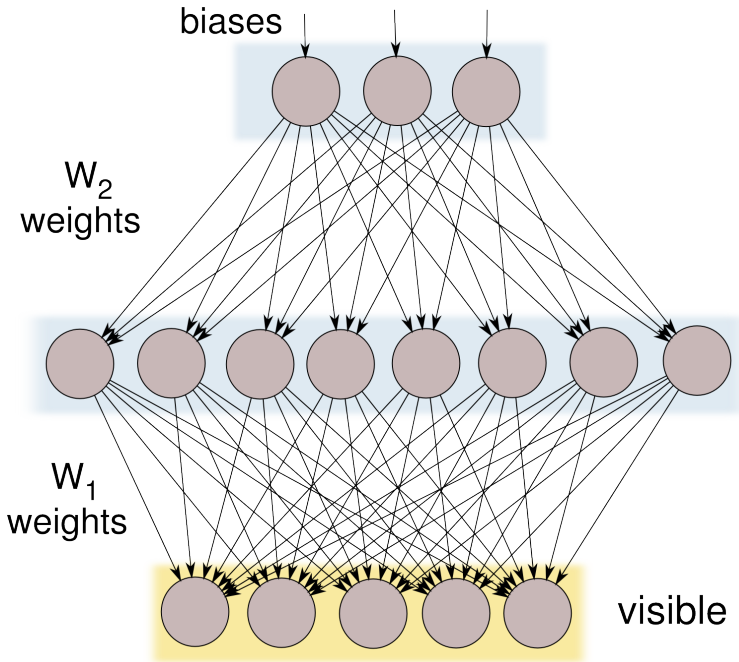
$$p_{\text{agg}}(\mathbf{h}) = \sum_{\mathbf{v} \in \mathcal{D}} p(\mathbf{h}|\mathbf{v})$$

- So the $W_1$ weights end up at values that maximize the likelihood of the data, given $\mathbf{h}$ sampled from the aggregate posterior distribution.
- The hidden bias weights end up at values that approximate this distribution, but their approximation to it is factorial.
- The $W_1$ weights get rewarded for finding values that make the aggregate posterior more factorial.

# training the next layer

Q: what is the *best* thing that the second layer could learn to do?

A: accurately generate the *aggregate posterior* distribution over the layer 1 hidden units. It is the distribution that makes the training data *most likely, given* $W_1$




- Easy! For each visible pattern, we just collect one sample (or more) from $p(\mathbf{h}|\mathbf{v})$.

This gives us a greedy, layer-wise procedure for training deep belief nets.

# a comment about factorial distributions

This greedy procedure doesn't work at all well. Subsequent layers add very little to what the first layer achieves, in modelling the data set. Why not?

Consider some patterns $\mathbf{v}$ on a set of visible nodes.

If these came from a world where the components of each vector are *independent*, then

$$p(\mathbf{v}) = p(v_1, v_2 \dots v_n) = \prod_i p(v_i)$$

and we say $p(\mathbf{v})$ is *factorial*.

**If $p(\mathbf{v})$ is factorial, there is no point in having hidden units**

... a model that included hidden units could do no better than a model with just bias inputs to the visibles.

# why does greedy training fail for deep belief nets?

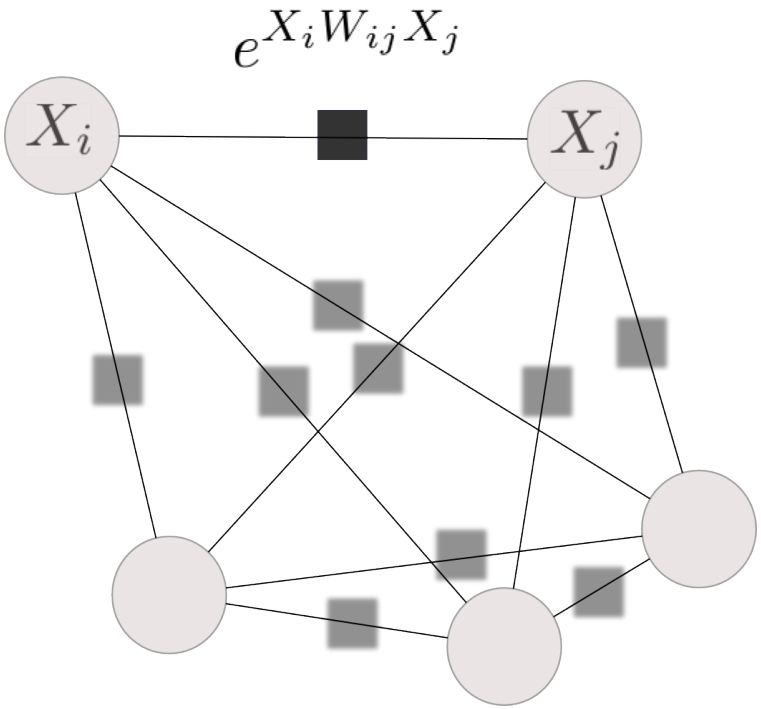
Notice:

- The posterior $p(\mathbf{h}|\mathbf{v})$ is *not* factorial
- The prior over $\mathbf{h}$ *is* factorial
- Learning will reward weights $W_1$ that make the *aggregate* posterior as factorial as possible
- But a factorial *aggregrate* posterior is the very last thing we want for a greedy learning procedure, because it leaves nothing left for $W_2$ to do!

prior distribution is *factorial*
$\longrightarrow$ $W_1$ tries to learn features that are *independent in the prior*

but this is premature: we want to add another layer *precisely because we DON'T BELIEVE THIS!*

# NEW IDEA: an undirected graphical model




| $X_i$ | $X_j$ | factor |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | $\exp(W_{ij})$ |

*i.e.* the factor between $i$ and $j$ is

$$\exp(X_i W_{ij} X_j)$$

So the parameter $W$ is increasing the probability of $X_i$ and $X_j$ being 1 at the same time. A positive value with result in them becoming correlated.

---

A graphical model with this kind of factor is called a **Boltzmann machine**.

# Boltzmann machines

The joint is a product of the factors, so:

$$P^\star(\mathbf{x}) = \prod_{i=1}^{N} \prod_{j=1}^{i} \exp(x_i \, w_{ij} \, x_j) \qquad \text{provided } w_{ii} = 0$$

$$\therefore \quad \log P^\star(\mathbf{x}) = \sum_{i=1}^{N} \sum_{j=1}^{i} x_i \, w_{ij} \, x_j$$

and so the gradient of that is going to be just

$$\frac{\partial}{\partial w_{ij}} \log P^\star(\mathbf{x}) = \quad x_i \, x_j$$

which is about as simple as you imagine.

# the learning rule for Boltzmann machines

$$\frac{\partial}{\partial w_{ij}} \log P^\star(\mathbf{x}) = \quad x_i \, x_j$$

and so the gradient is

$$\Delta w_{ij} \quad \propto \quad \left\langle x_i \, x_j \right\rangle_{\mathbf{v}\in\mathcal{D},\ \mathbf{h}\sim P(\mathbf{h}|\mathbf{v})} \quad - \quad \left\langle x_i \, x_j \right\rangle_{\mathbf{x}\sim P(\mathbf{x})}$$

clamped phase — free phase

Hebbian learning — anti-Hebbian

---

But to do it, we will need to produce samples from those two distributions.

# Gibbs sampling for Boltzmann machines

How does Gibbs sampling work for this case? It's handy to write the state as split up into $x_k$ and the rest, which is $\mathbf{x}_{\setminus k}$. We choose a new state for (say) $x_k$ from the distribution $P(x_k|\mathbf{x}_{\setminus k})$. What's that? Well, we know

$$\log P^\star(\mathbf{x}) = \sum_{i=1}^{N}\sum_{j=1}^{i} x_i\, w_{ij}\, x_j = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} x_i\, w_{ij}\, x_j$$

and $x_k$ appears once in each sum, so:

$$\log P^\star(x_k, \mathbf{x}_{\setminus k}) = \frac{1}{2}\sum_{j=1}^{N} x_k w_{kj} x_j + \frac{1}{2}\sum_{i=1}^{N} x_i w_{ik} x_k$$

$$+ \text{ other terms that don't involve } k$$

Notice that this is zero in the case that $x_k = 0$, so we have that

$$\log P^\star(x_k = 1, \mathbf{x}_{\setminus k}) = \frac{1}{2}\sum_{j=1}^{N} w_{kj} x_j + \frac{1}{2}\sum_{i=1}^{N} x_i w_{ik}$$

Provided $w_{ij} = w_{ji}$, this is

$$\log P^\star(x_k = 1, \mathbf{x}_{\setminus k}) = \sum_i w_{ki} x_i$$

from which it's straightforward to normalise and arrive at

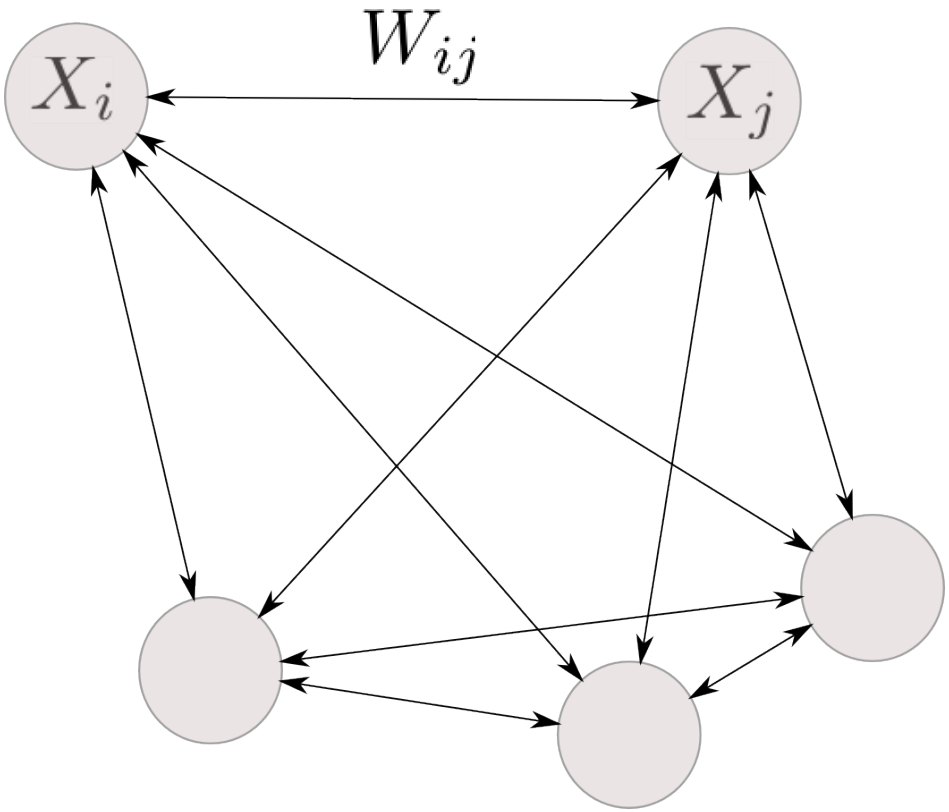
$$P(x_k = 1 \mid \mathbf{x}_{\setminus k}) = \frac{1}{1 + e^{-\sum_i w_{ki} x_i}}$$

So Gibbs sampling in a Boltzmann machine amounts to using the sigmoid function of the weighted inputs - it's a "neuron"!

---

*ie.* "vanilla" neuron updates can be used to get samples from $P(\mathbf{h} \mid \mathbf{v})$ in the clamped phase, as well as samples from $P(\mathbf{h}, \mathbf{v})$ in the free phase.

# The Boltzmann machine is a recurrent neural network




- symmetric weights
- no self-connections
- we've ignored biases, but they're good too

# learning in Boltzmann machines

The learning rule turned out to be:

$$\Delta w_{ij} \;=\; \eta \;\left[\langle x_i\, x_j\rangle_{P_{\mathbf{x}}^{\text{data}}} \;-\; \langle x_i\, x_j\rangle_{P_{\mathbf{x}}}\right]$$

## learning in Boltzmann machines

- **clamped phase**: "clamp" each training pattern to the units, *do Gibbs sampling on the hidden units*, and accumulate the Hebbian changes.
- **unclamped phase**: do Gibbs sampling on *all units*, and accumulate anti-Hebbian changes.

- could think of free phase as a single markov chain run without "clamping" and take lots of samples, or
- could do it one on the end of each clamped phase.

# so much for the euphoria

This sounds great in theory.

Practice is a little different.

**Having to resort to MCMC for both phases is disastrous: we're trying to climb a hill by using the difference between two noisy estimates!**

Nonetheless this is the only option available in fully-connected Boltzmann machines, which **completely suck** as a result.

# summary

| | **sigmoid belief net** | **Boltzmann machine** |
|---|---|---|
| factors | sigmoid($\phi$) | $\exp(x_i w_{ij} x_j)$ |
| $P^\star$ | already normalised, so no second term. | needs normalization, so $Z$ matters... |
| $P(\mathbf{h}\|\mathbf{v})$, clamped phase | hard to draw samples from | hard to draw samples from |
| $P(\mathbf{v},\mathbf{h})$, free phase | easy to draw samples from (*and* no need to!) | hard to draw samples from |
| Gibbs | ugly | sigmoid($\phi$) |

| | | |
|---|---|---|
| verdict: | pretty hopeless | completely useless |

# the problem

The gradient estimate is the difference between two noisy estimates, each of which requires sampling from a long MCMC chain, (after waiting for it to reach equilibrium).

This learning algorithm is beautiful but glacially slow in practice.

Despite their intuitive appeal as generative models, and their tempting similarities with biological neural nets, Boltzmann machines seemed doomed to the scrap heap, until recently.

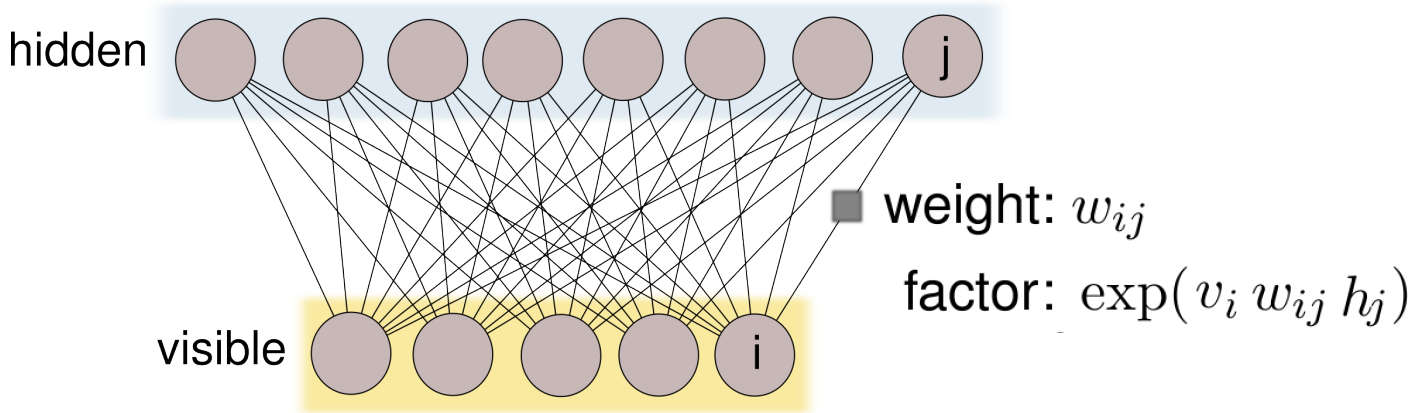
We're going to do two tricks to make Boltzmann machines practical devices:

1. restrict the connectivity.
2. use a new learning algorithm to train the weights.

and *then* we're going to show how to use them to solve the towers problem of sigmoid belief nets...

# trick # 1: restrict the connections

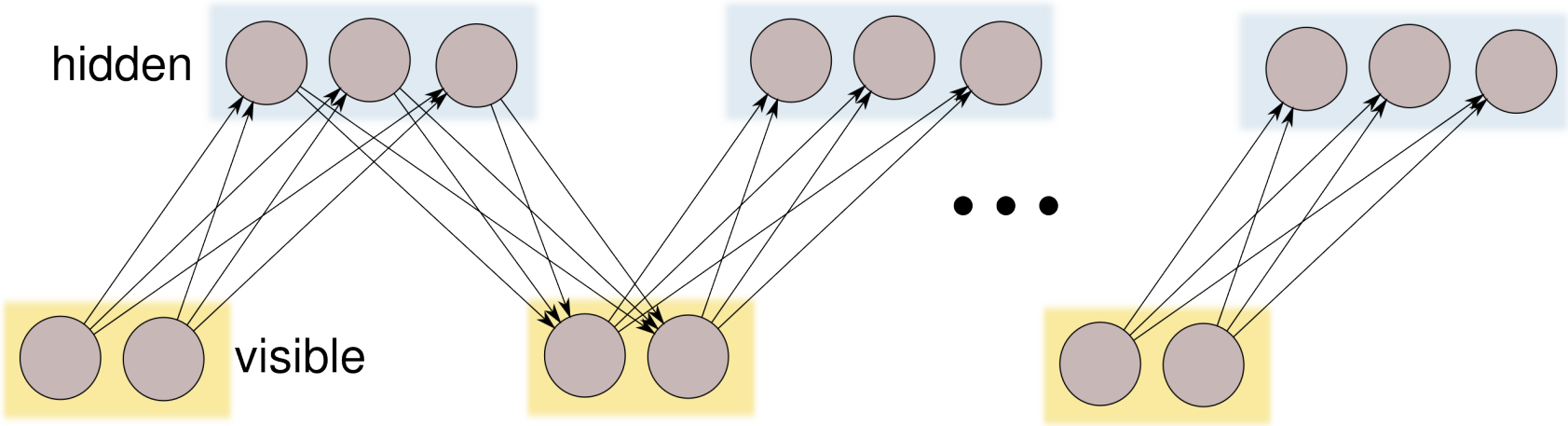
- Assume visible units are one layer, and hidden units are another.
- Throw out all the connections within each layer.




- $h_j \perp\!\!\!\perp h_k \mid \mathbf{v}$
- the posterior $P(\mathbf{h} \mid \mathbf{v})$ factors
  *c.f.* in a belief net, the *prior* $P(\mathbf{h})$ factors
- no explaining away

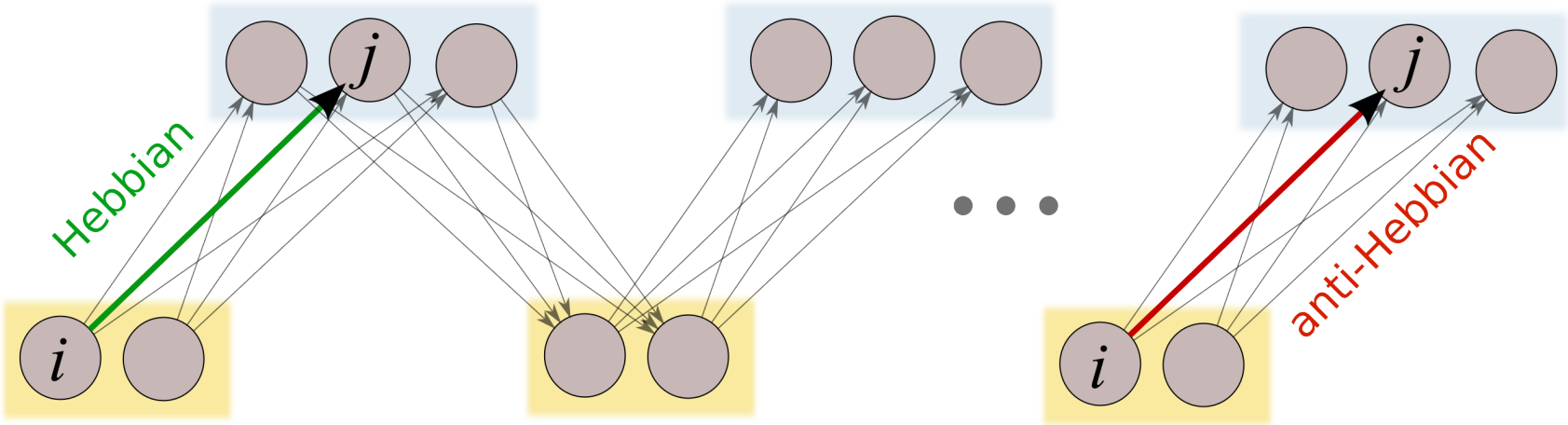
# Alternating Gibbs sampling

Since none of the units within a layer are interconnected, we can do Gibbs sampling by updating the whole layer at a time.




(with time running from left $\longrightarrow$ right)

# learning in an RBM




Repeat for all data:

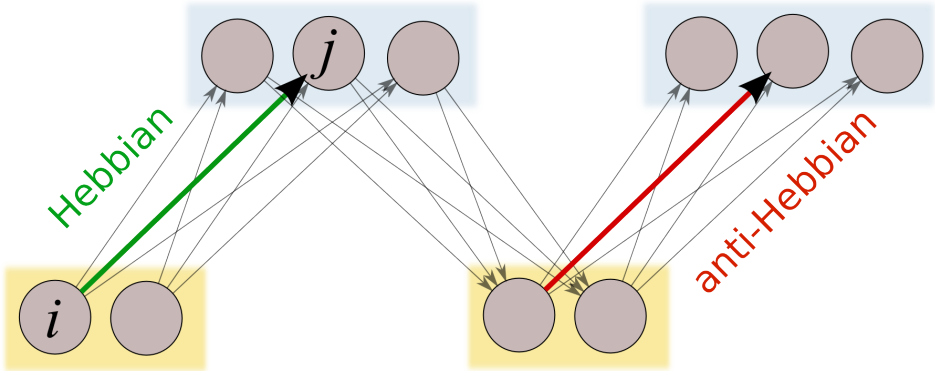
1. start with a training vector on the visible units
2. then alternate between updating all the hidden units in parallel and updating all the visible units in parallel

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

**restricted connectivity is trick #1:**

it saves waiting for equilibrium in the clamped phase.

# trick # 2: curtail the Markov chain during learning




Repeat for all data:

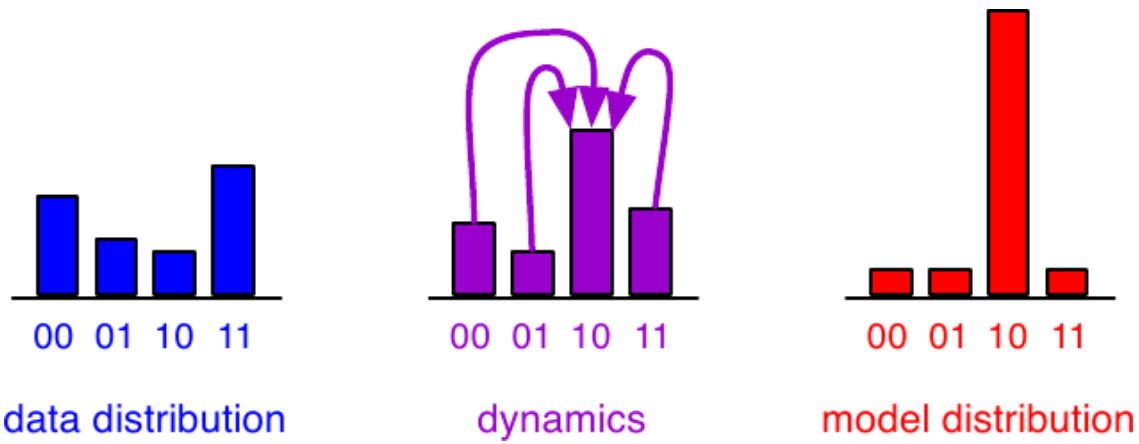
1. start with a training vector on the visible units
2. update all the hidden units in parallel
3. update all the visible units in parallel to get a "reconstruction"
4. update the hidden units again

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

This is not following the correct gradient, but works well in practice. Geoff Hinton calls it learning by "contrastive divergence".

# why does this work?

Starting at a data point, the Markov chain wanders off...
The Bolzmann machine learning rule takes some of the probability mass from where the chain wanders to, and puts it back on the data point:




(fig from “Minimum Probability Flow Learning” Sohl-Dickstein, Battaglino & DeWeese, 2009)

Contrastive Divergence just doesn’t wait for it to wander far: the *direction* of wandering is apparent after a few steps, and that’s good enough.

**trick #2: contrastive divergence**

this saves waiting for equilibrium in the unclamped phase.

# RBM summary

Two tricks were used to make Boltzmann machines practical devices:

**restrict the wiring**

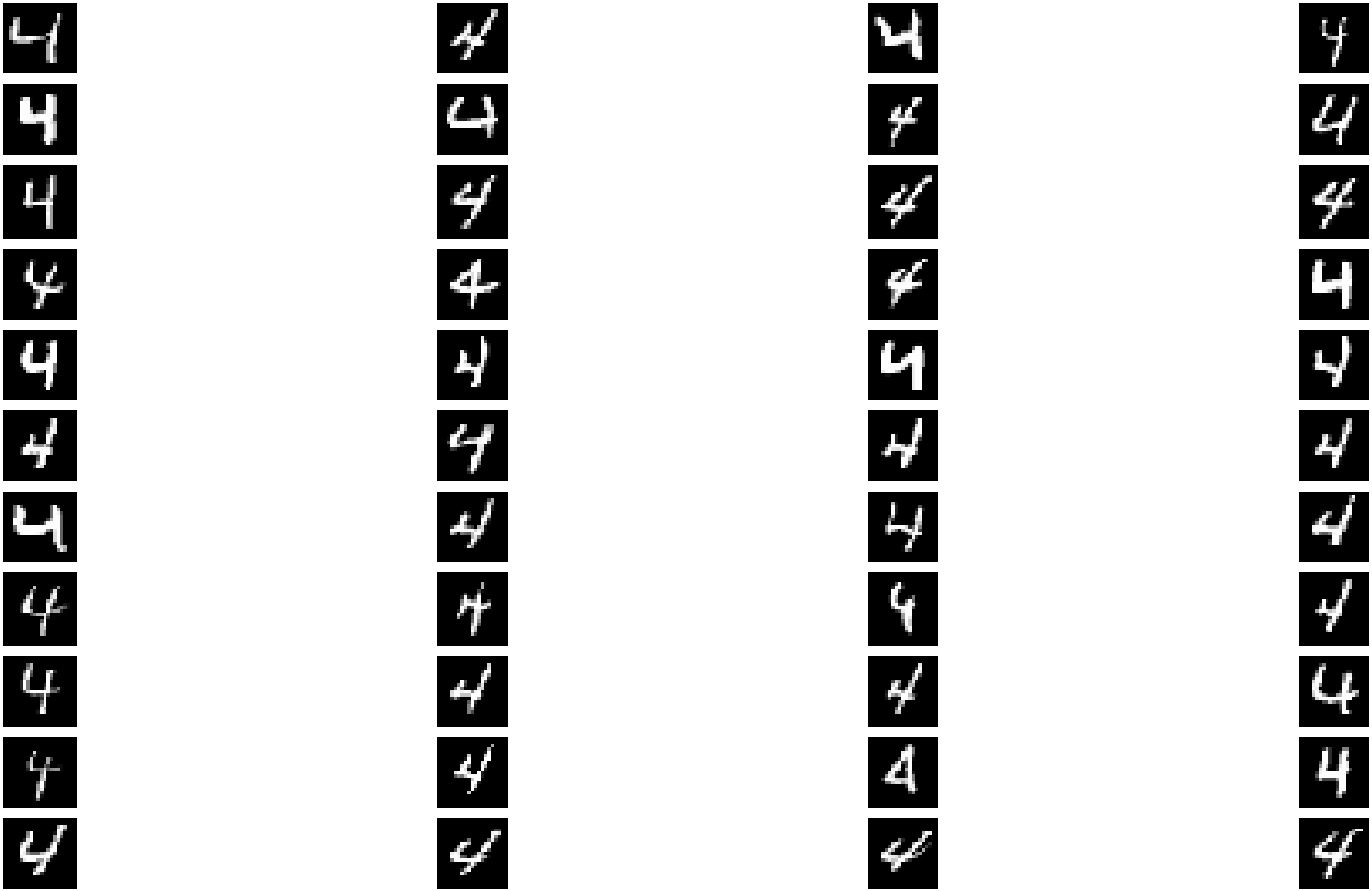
This saves waiting for equilibrium in the clamped phase.

**truncate the Markov chain**

This saves waiting for equilibrium in the unclamped phase.

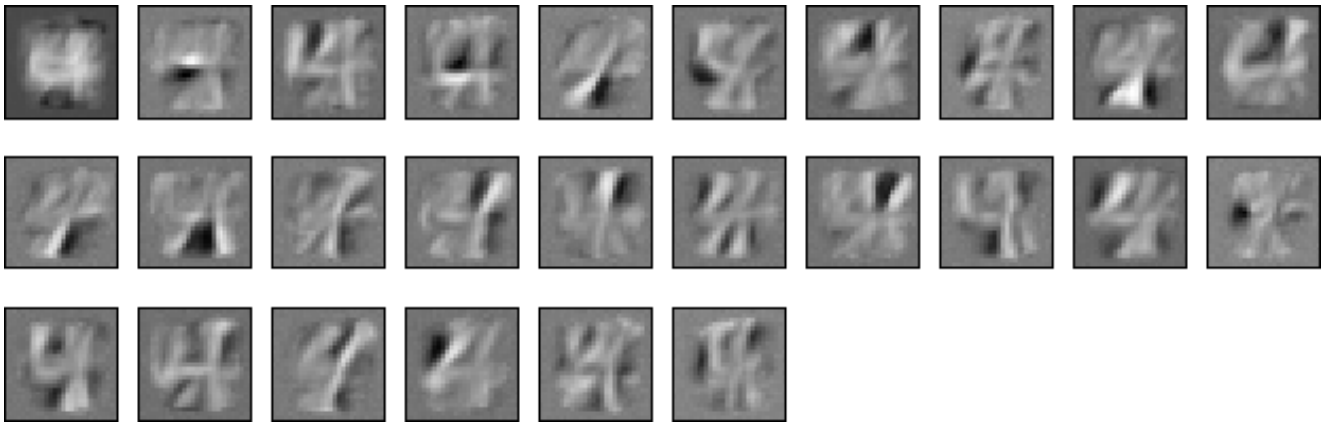
(contrastive divergence)

# e.g. training data for a single class



# e.g. RBM trained on a single class

Weights after training on a single class:

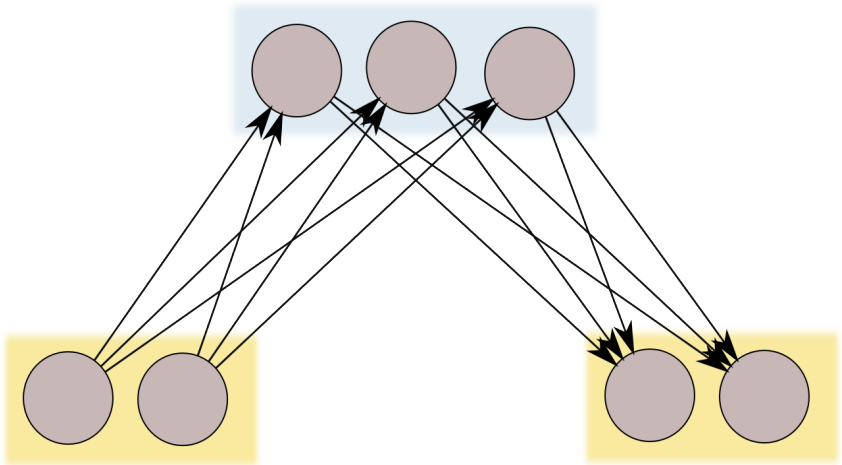


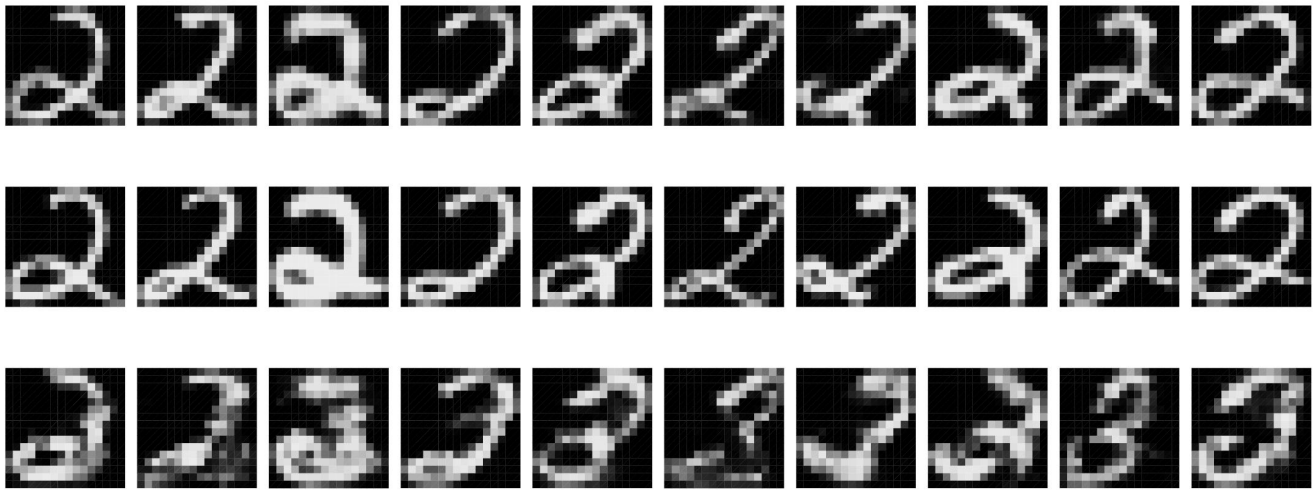
Notice how each neuron has identified a different feature.

---

Samples (long run of alternating Gibbs sampling):

# some reconstructions



from a model trained with “2” examples

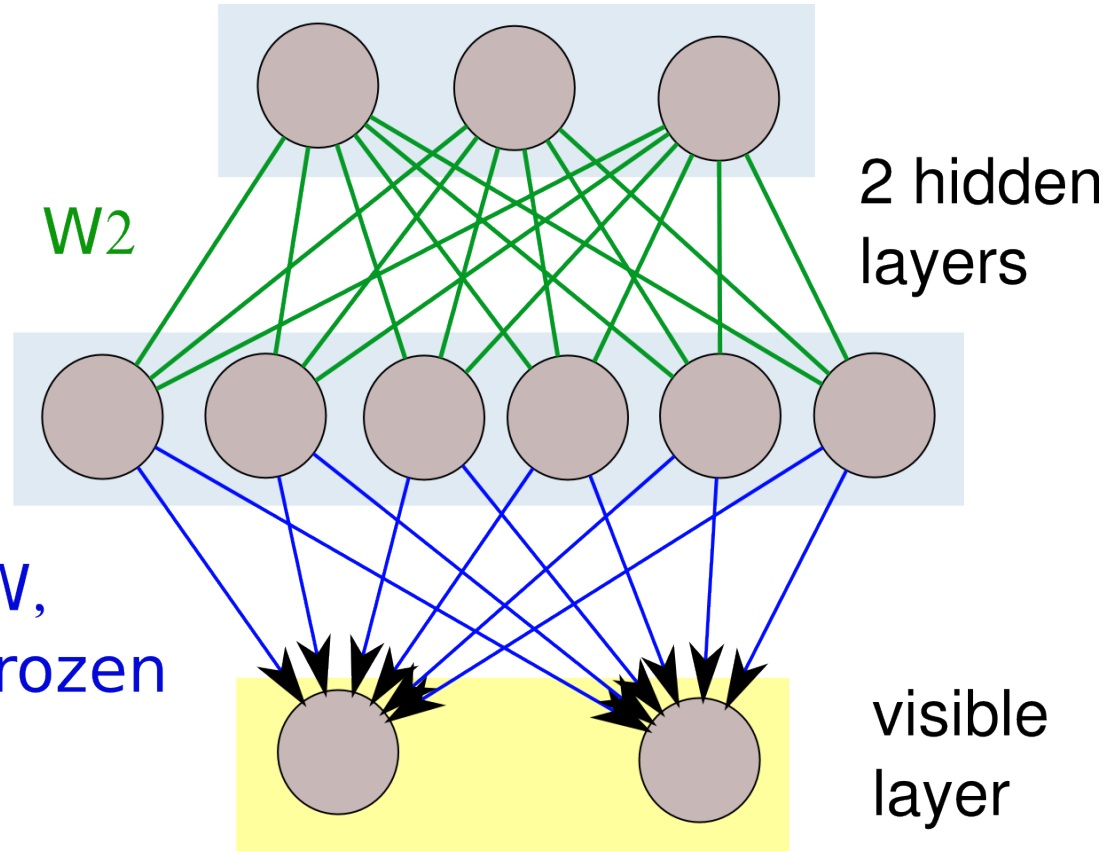
STARTING IMAGE

from a model trained with “3” examples

# greedy, layer-wise learning?

**an idea:**

take samples from the aggregate posterior and use them to train another layer (and then another... and so on).

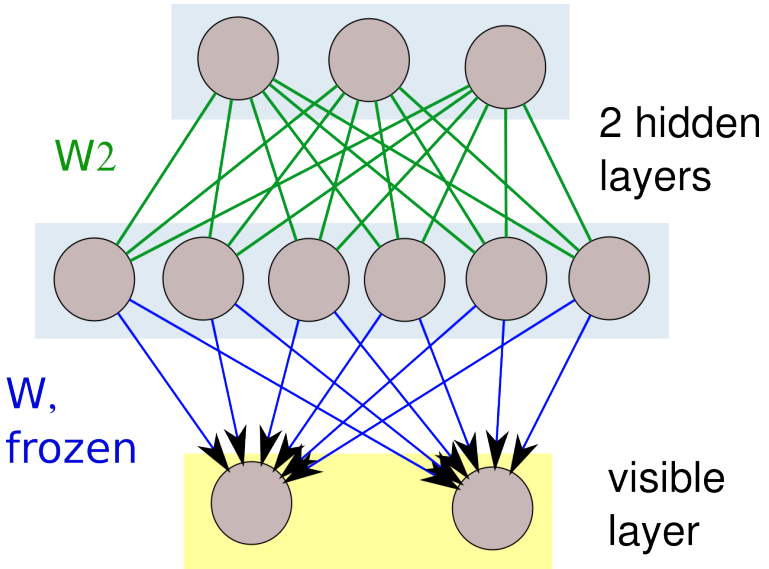



To generate data, we could

1. run alternating Gibbs sampling for a while on the top layer, and then
2. do a top-down pass to the visible layer.

# greedy, layer-wise learning?

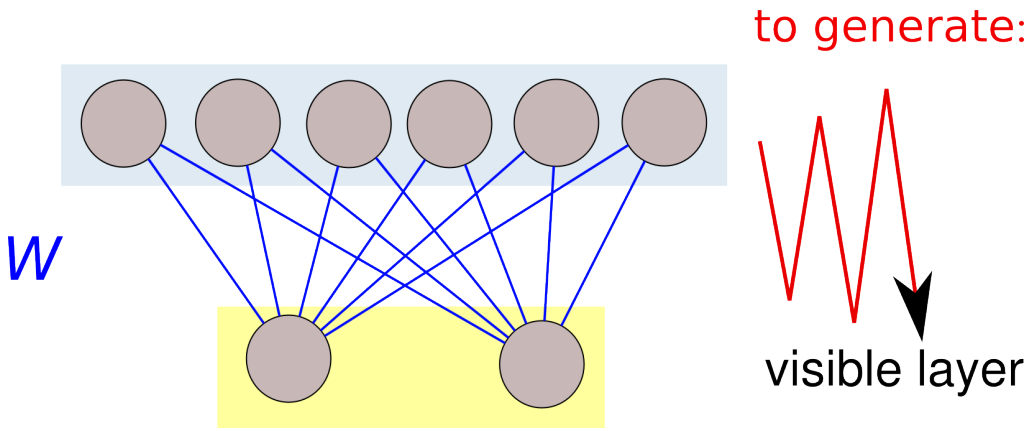
But why would we expect this to do any better than the more obvious procedure of using single-layer sigmoid belief nets in the same way?



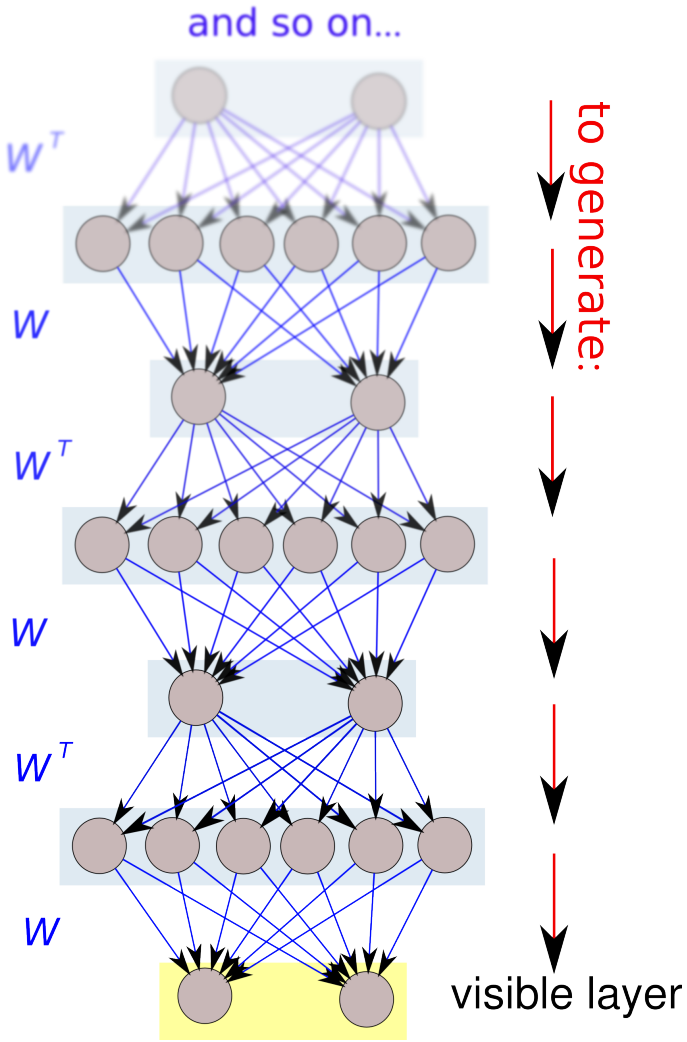

Two ideas that help understanding:

1. RBMs are *already* deep BNs...
2. RBMs *don't* try to force the aggregate posterior to be factorial...

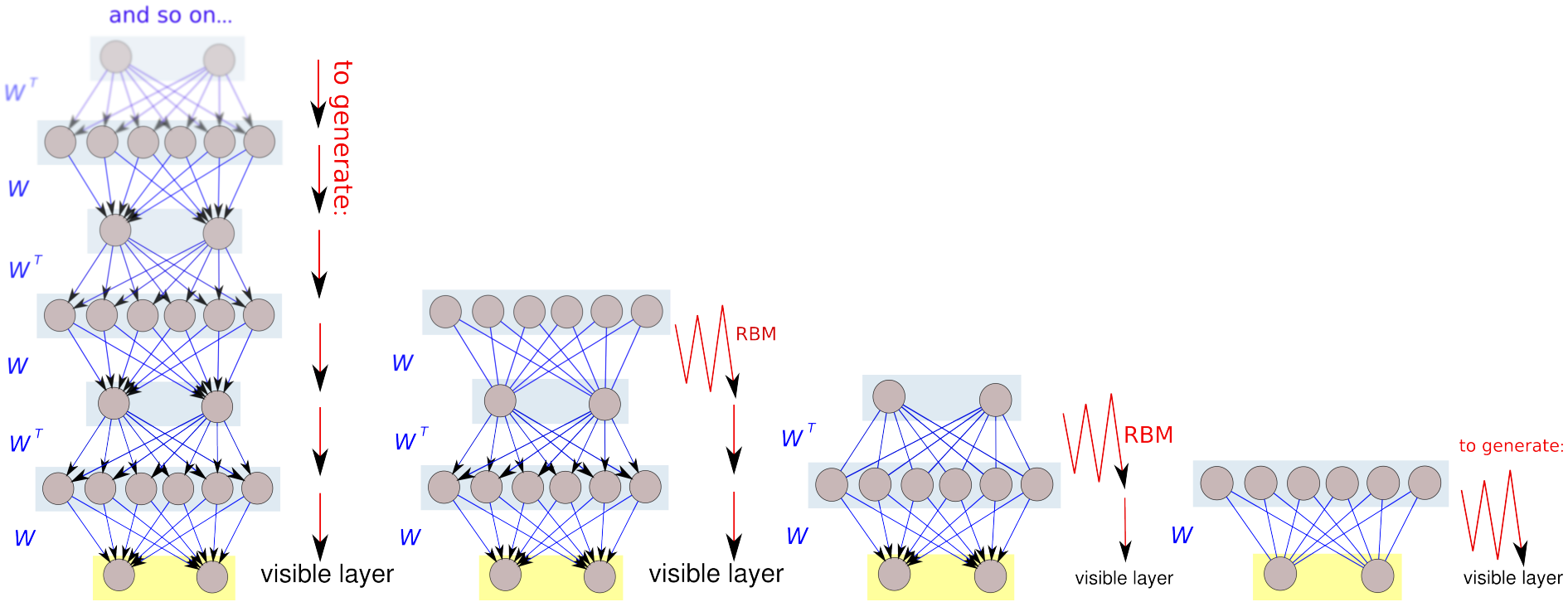
# 1: RBMs are infinitely deep belief nets




sampling from this is the same as sampling from the network on the right.



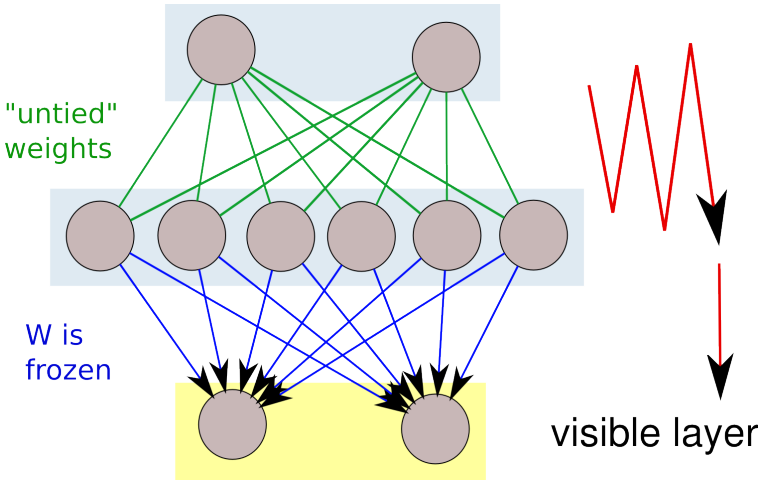
# in fact, all of these are the same animal...

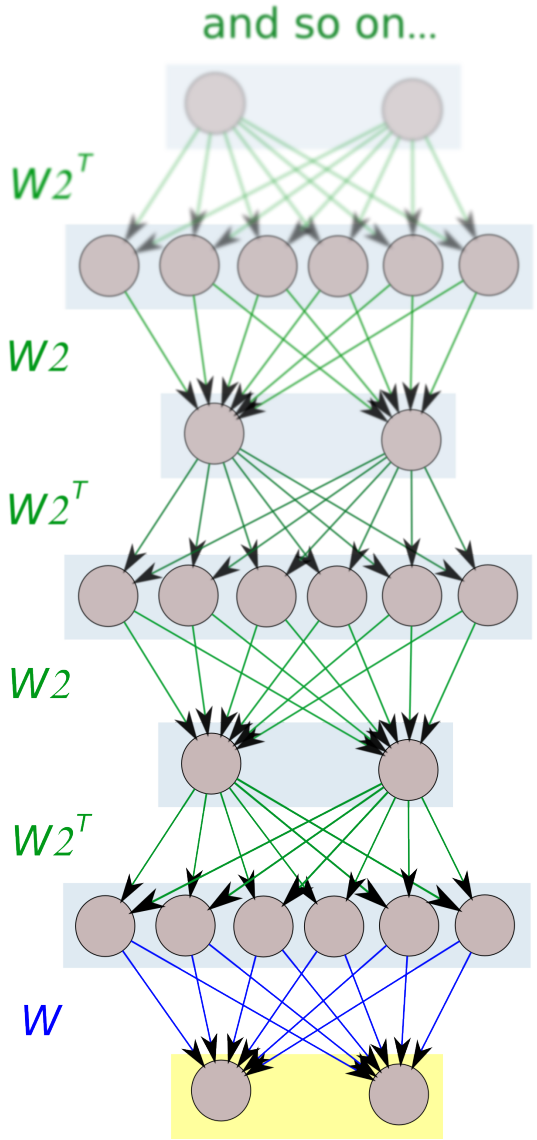



- So when we train an RBM, we're really training an $\infty^{ly}$ deep sigmoid belief net!
- It's just that the weights of all layers are tied.

# un-tie the weights from layer 2 to $\infty$

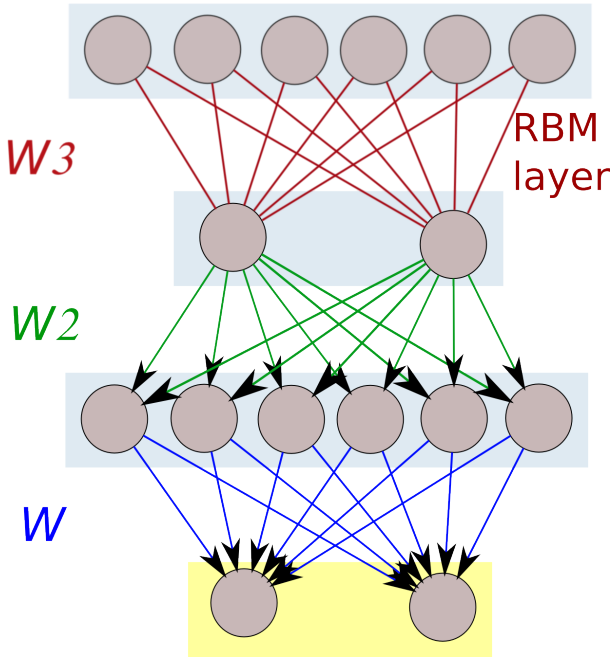
If we freeze the first RBM, and then train another RBM atop it, we are untying the weights of layers 2+ in the $\infty$ net (which remain tied together).

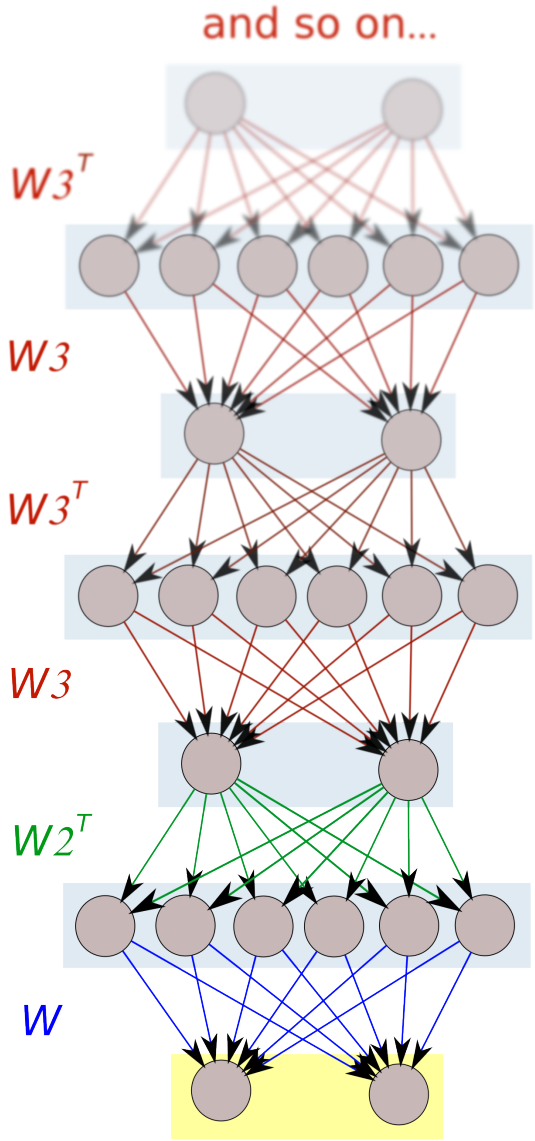


# un-tie the weights from layer 3 to $\infty$

and ditto for the 3rd layer...



# greedy, layer-wise learning?

But why would we expect this to do any better than the more obvious procedure of using single-layer sigmoid belief nets in the same way?

Two ideas that help understanding:

1. RBMs are *already* deep BNs...
2. RBMs *don't* try to force the aggregate posterior to be factorial...

# 2: RBMs *don't* make the aggregate posterior ∼factorial

## in a 1-layer sigmoid belief net:

- The prior over hiddens is factorial, but the posterior isn't.
- Tries to make the aggregate posterior factorial.
- ...leaves little for subsequent layers to do.

## in a restricted Boltzmann machine:

- The posterior over hiddens is factorial, but the prior isn't.
- DOESN'T try to force the aggregate posterior to be factorial.
- ...leaves more for the next layer to do.

# aside: natural distribution?

What kinds of distributions are RBMs well-suited for?

A single RBM and appropriate weights can generate any desired distribution over the visible units, if we give it enough hidden units ($\approx 2^{\#\text{ visible}}$).

The probability of the joint state in an RBM is

$$P(\mathbf{v}, \mathbf{h}|\mathbf{W}) \propto \exp(\mathbf{h}^{\mathrm{T}}\mathbf{W}\mathbf{v})$$

and so

$$P(\mathbf{v}|\mathbf{W}) \propto \sum_{\mathbf{h}} \exp(\mathbf{h}^{\mathrm{T}}\mathbf{W}\mathbf{v})$$

# aside: natural distribution?

Carefully tracking terms (!), in log space this leads to:

$$\log P(\mathbf{v}|\mathbf{W}) = \underbrace{\mathbf{w}_{0\star}}_{\text{biases}} \cdot \mathbf{v} + \sum_j \log(1 + e^{\mathbf{w}_{j\star}\cdot\mathbf{v}}) + \text{constant}$$

where $\mathbf{w}_{j\star}$ is the vector of weights between the $j^{\text{th}}$ hidden unit and the visible units.

- a linear trend across the input space
- a sum of functions of form $f = \log(1 + e^{\phi})$. This is zero for $\phi < 0$ and the identity function for $\phi > 0$, with a smooth transition.

**Each hidden unit represents a "feature" characterised by the direction of its weight vector $\mathbf{w}_{j\star}$**

The hidden unit makes states that are aligned with this vector more likely.

# aside: natural distribution?

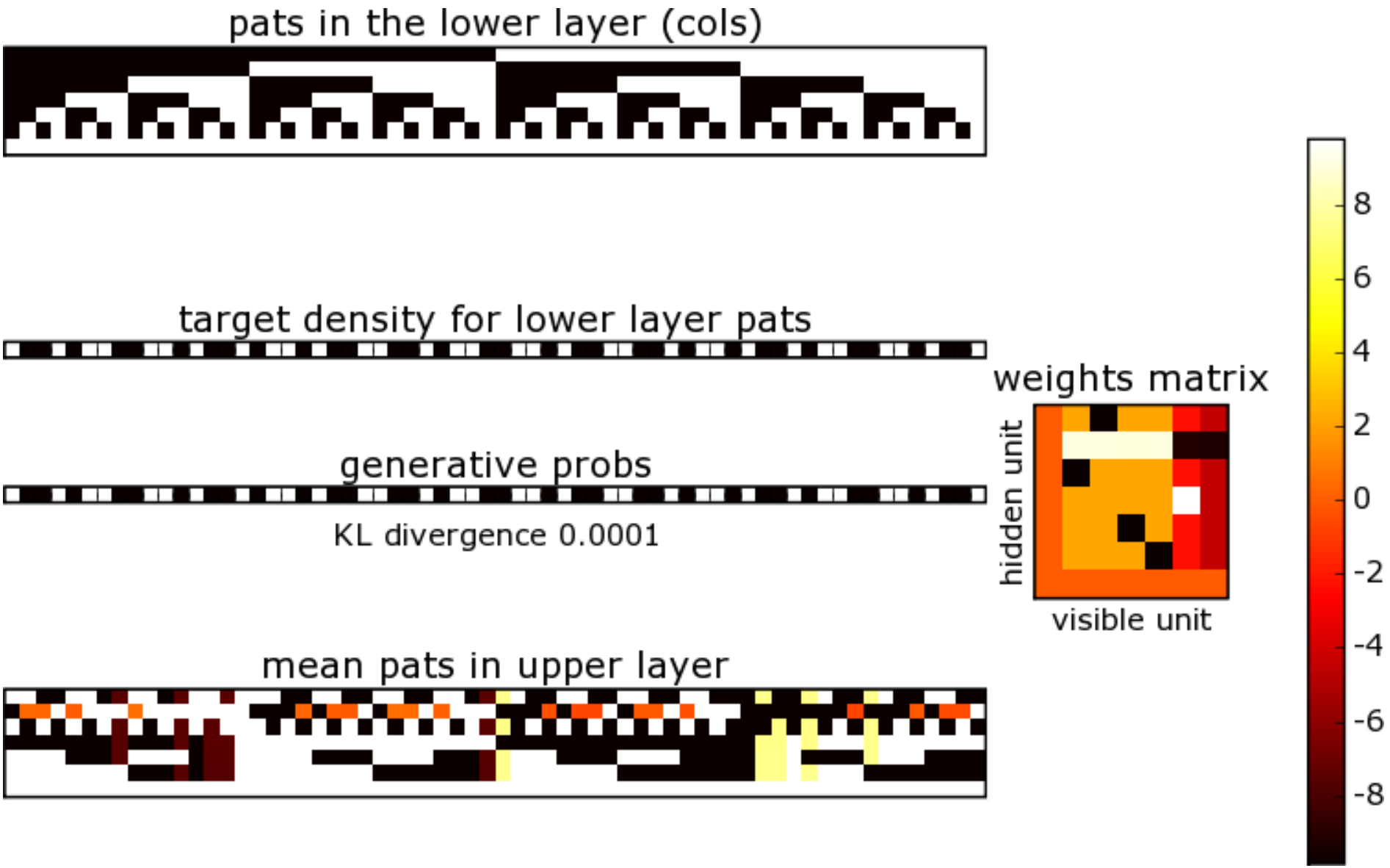
RBMs seem predisposed to capture distributions consisting of conjunctions of high probability features.

They should find it difficult to capture distributions that have 'probability holes' in them, since they can only add thresholded ramps together. To make a 'hole', they essentially have to add probability mass everywhere else.

But this needs to be taken with a grain of salt. David MacKay and I tried training RBMs (using the *exact* gradient) to learn "parity" distribution problems: exponentially many probability holes.

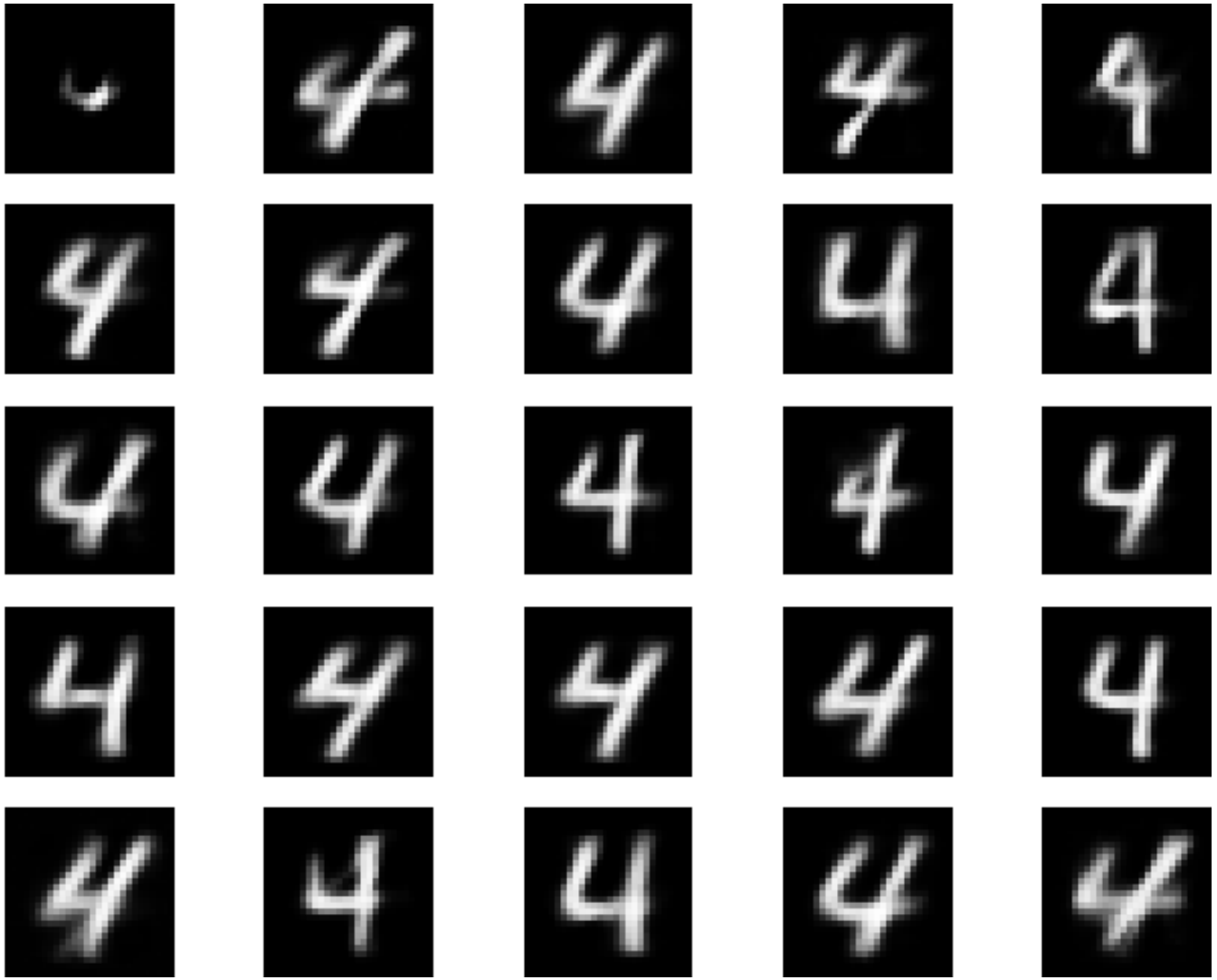
Incredibly, an RBM with 6 hidden units can learn the parity distribution on 6 visible units perfectly! It does this by arranging a set of of 6 ramps in just the right way to get high probability mass on all 32 of the desired patterns, while staying low for the 32 undesirable ones.

# an RBM with 6 hidden units doing 6 bit parity

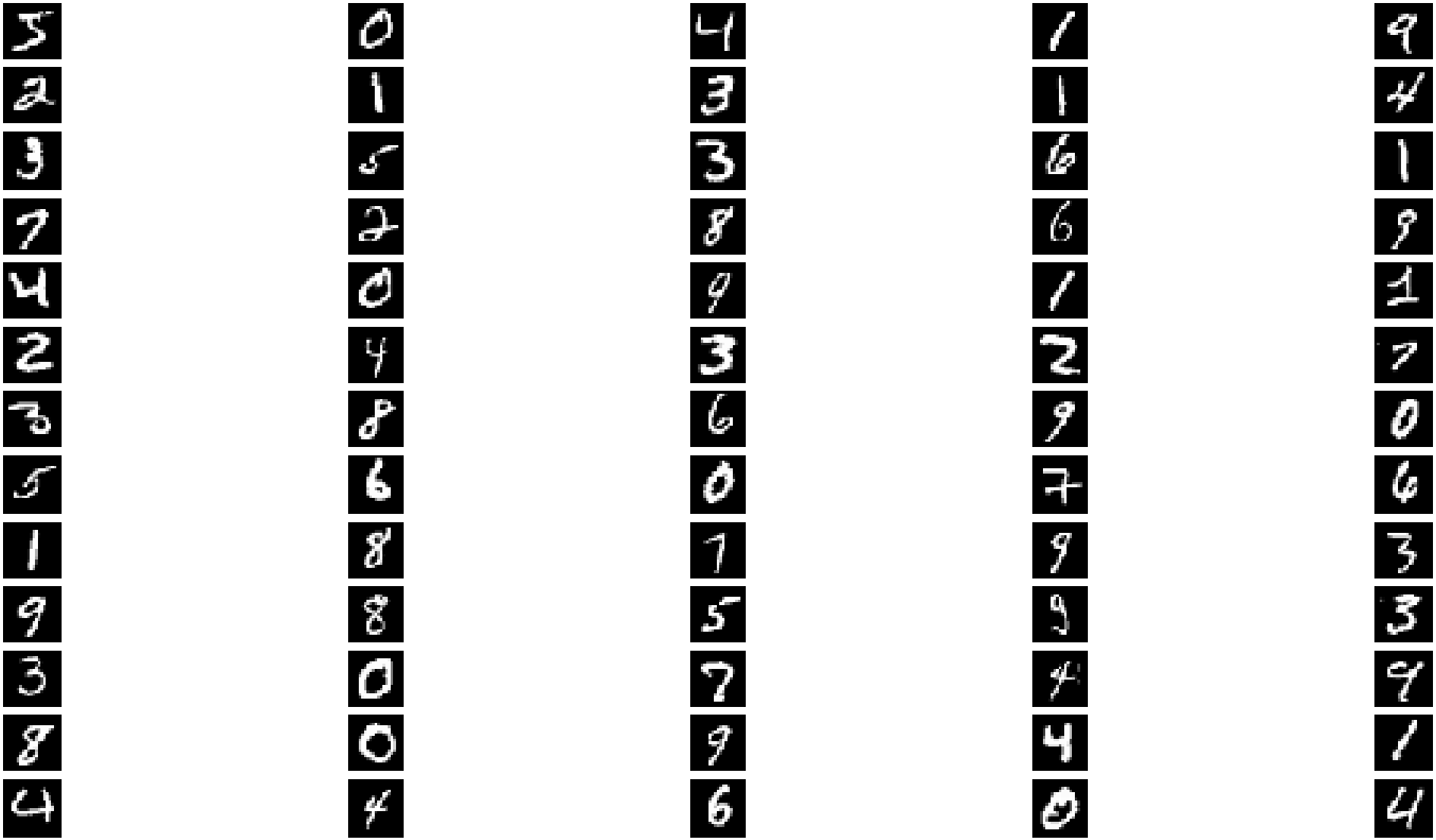


# samples from a 2 layer network

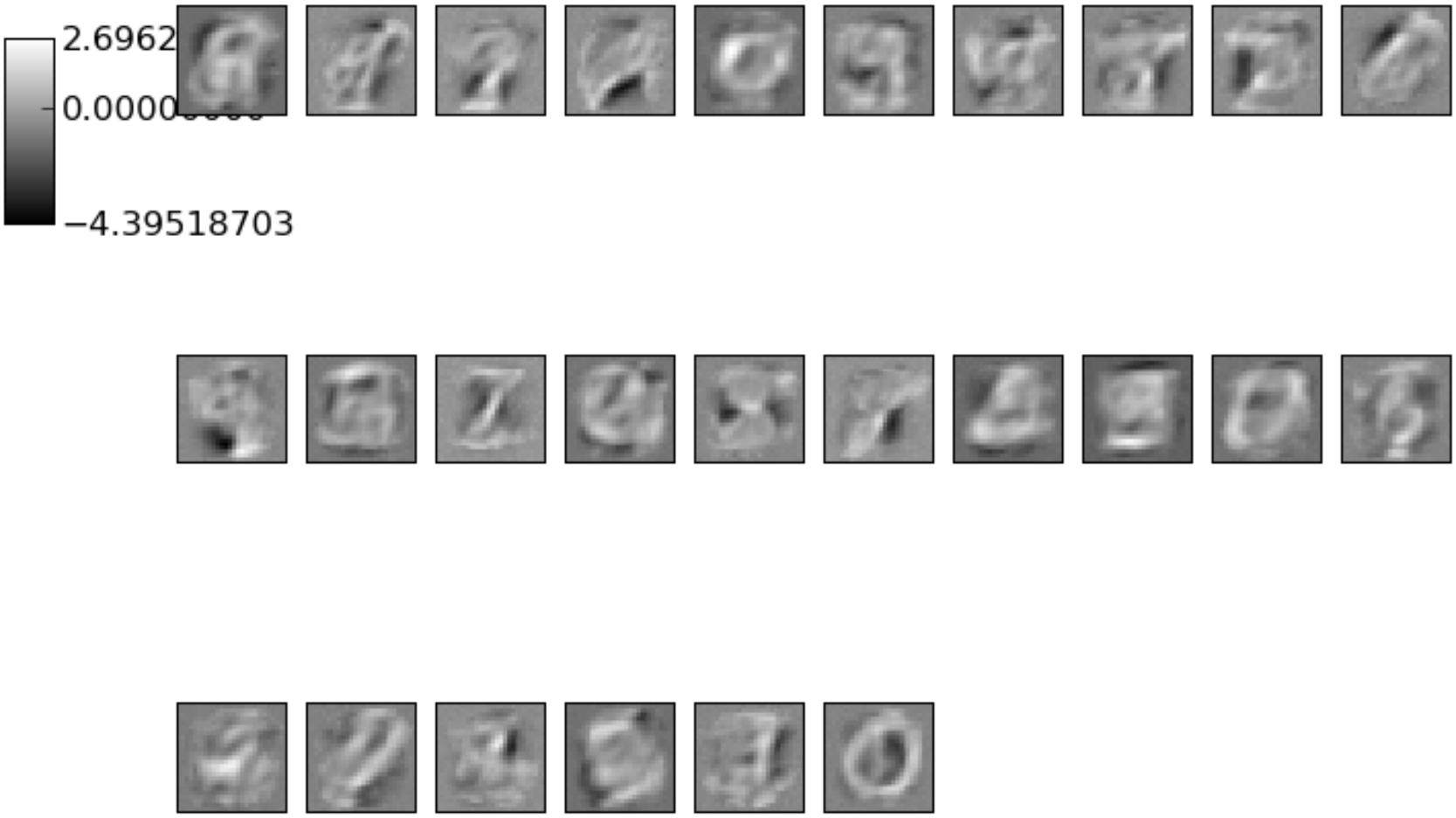
reminder:
samples
from RBM
all looked
like this



# training data for multiple classes



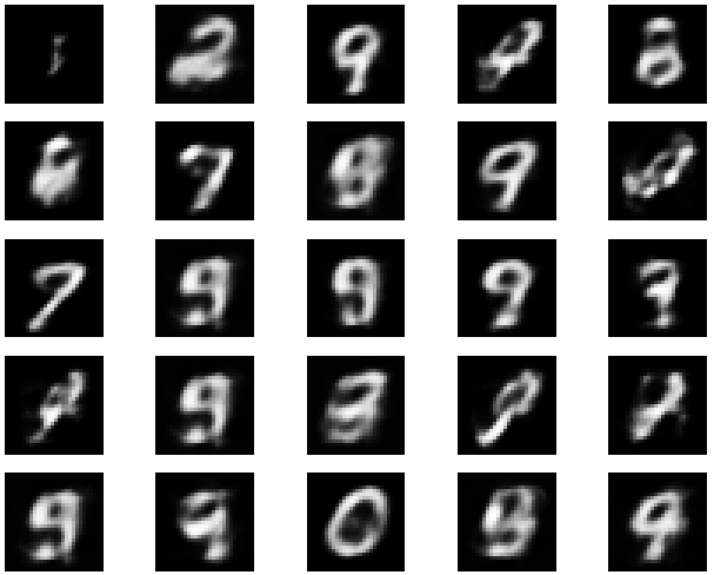
# weights after training on multiple classes



# samples from RBM trained on multiple classes

1st layer RBM alone

with 2nd layer RBM



# fine-tuning with the wake-sleep algorithm

So far, the up and down weights have been symmetric, as required by the Boltzmann machine learning algorithm. And we didn't change the lower levels after "freezing" them.

- wake: do a bottom-up pass, starting with a pattern from the training set. Use the delta rule to make this more likely *under the generative model*.
- sleep: do a top-down pass, starting from an equilibrium sample from the top RBM. Use the delta rule to make this more likely *under the recognition model*.

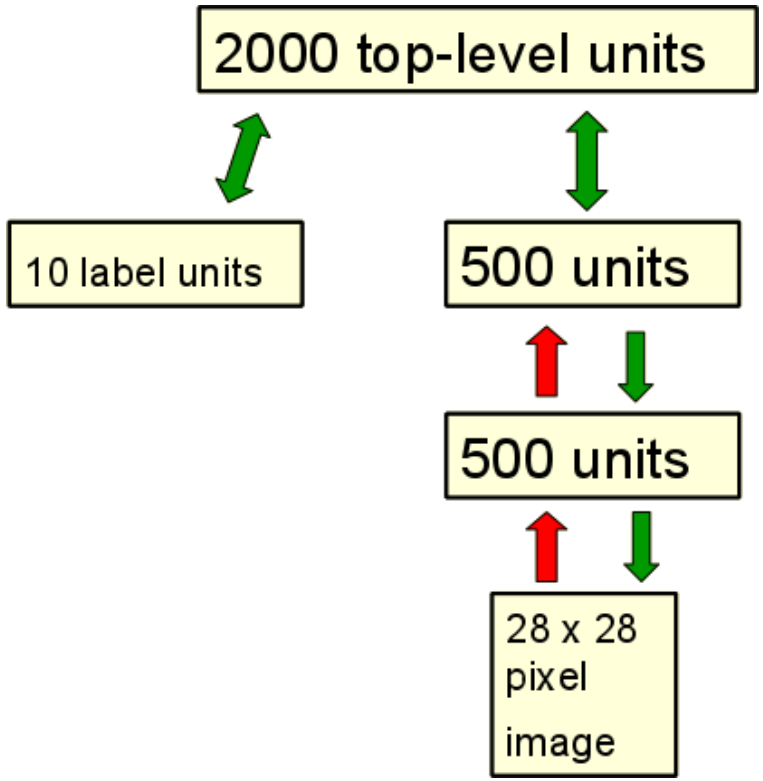
*[CD version: start top RBM at the sample from the wake phase, and don't wait for equilibrium before doing the top-down pass]*.

**wake-sleep learning algorithm**

unties the recognition weights from the generative ones

# application # 1: classification

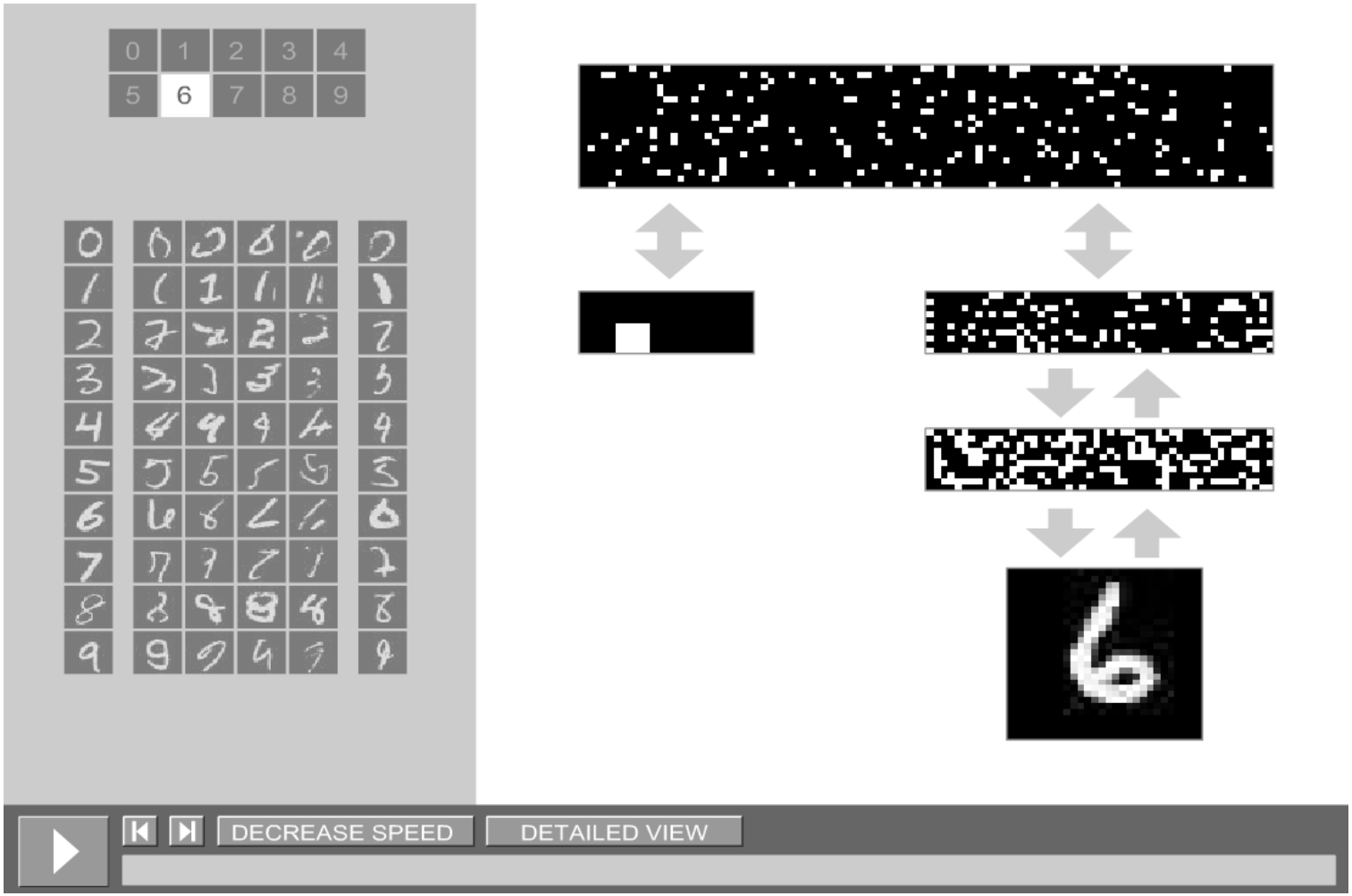
We can make a dedicated digit-model by including a “1-of-10” softmax unit in the RBM at the top:




The soft-max unit could be considered another “visible” unit, if we have labelled data available. Otherwise treat it as another hidden unit.

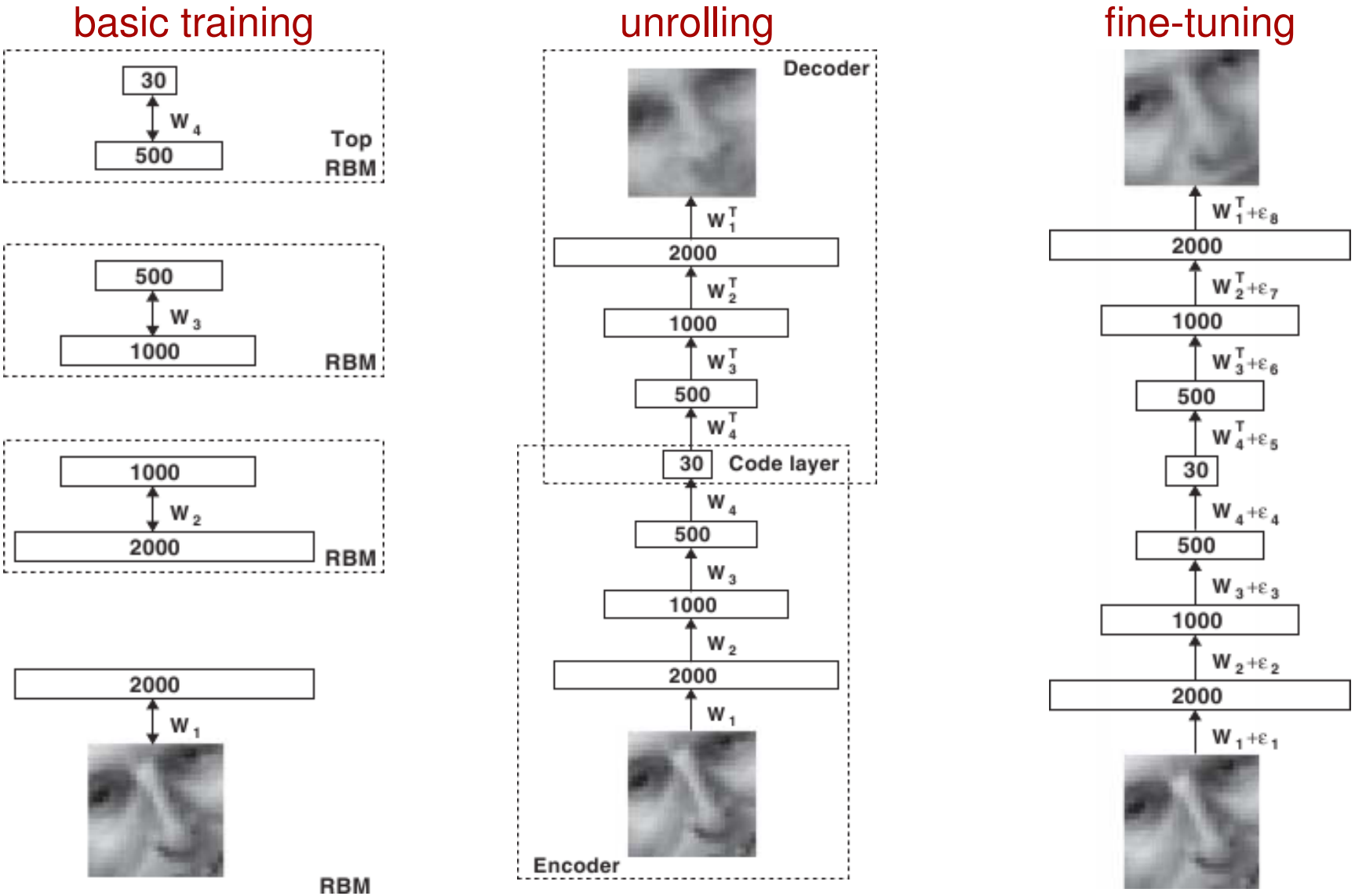
Samples generated by running the top-level RBM with one label clamped. There are 100 iterations of alternating Gibbs sampling between samples.

# demo




http://www.cs.toronto.edu/~hinton/adi/index.htm

# application # 2: autoencoding

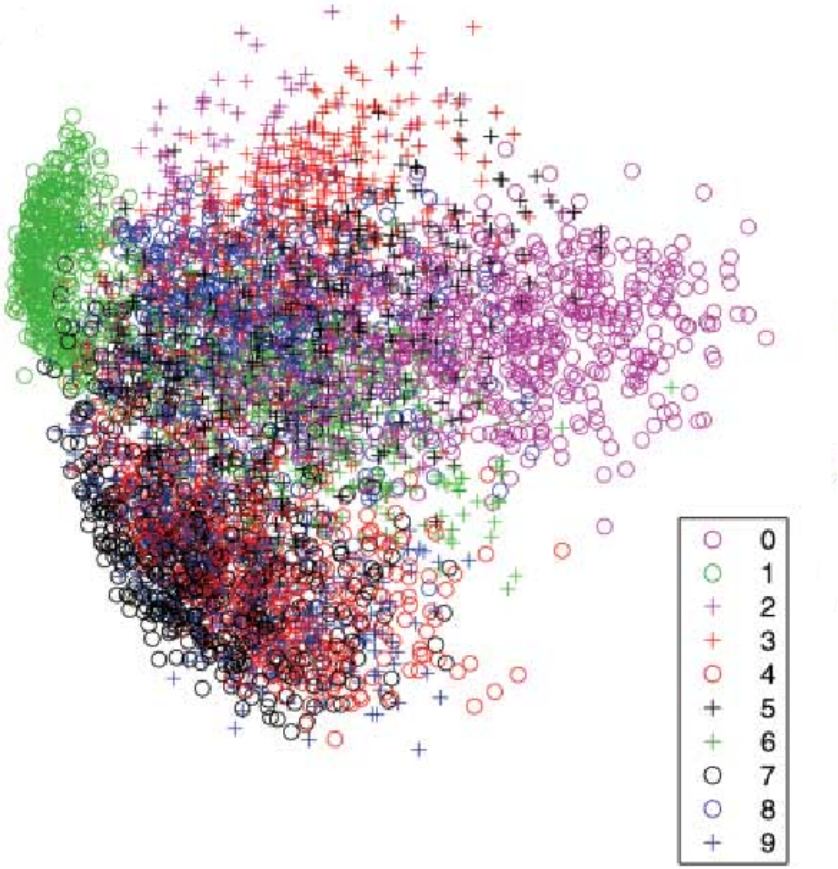



In this case the fine-tuning is done with back-prop!

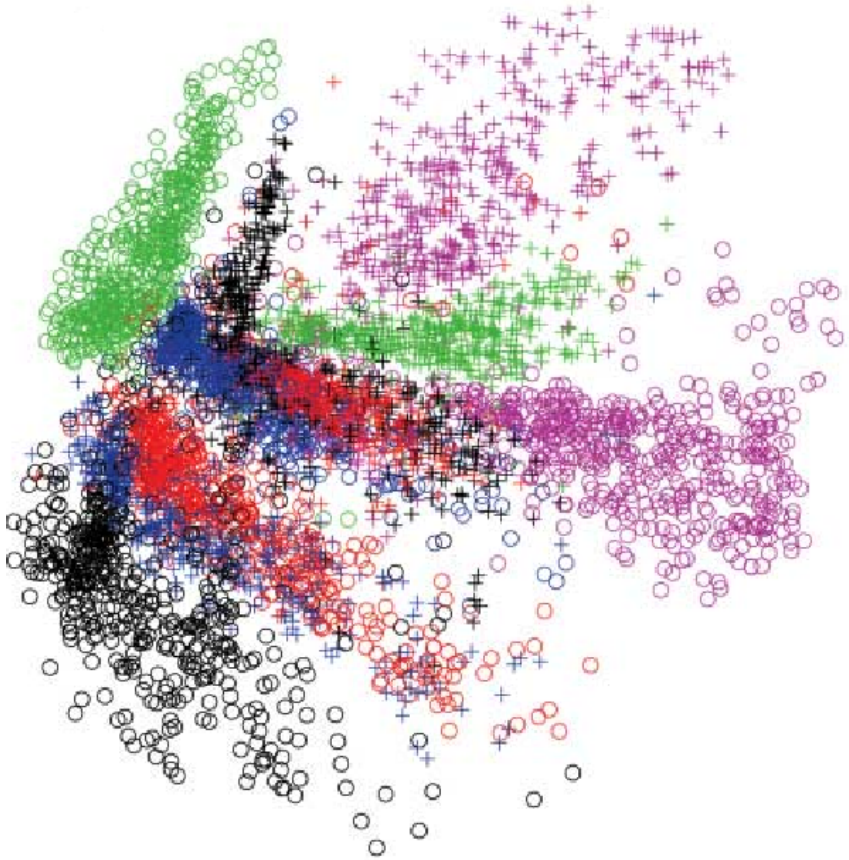
# example: autoencoding digits


Notice: again, the fine tuning (by back-prop this time) achieves its effect by untying the generative and recognition weights.

# example: autoencoding digits



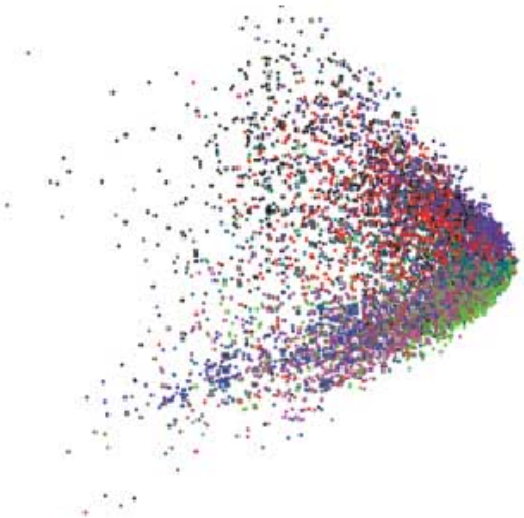

Codes for digits, produced by taking the first 2 principal components.



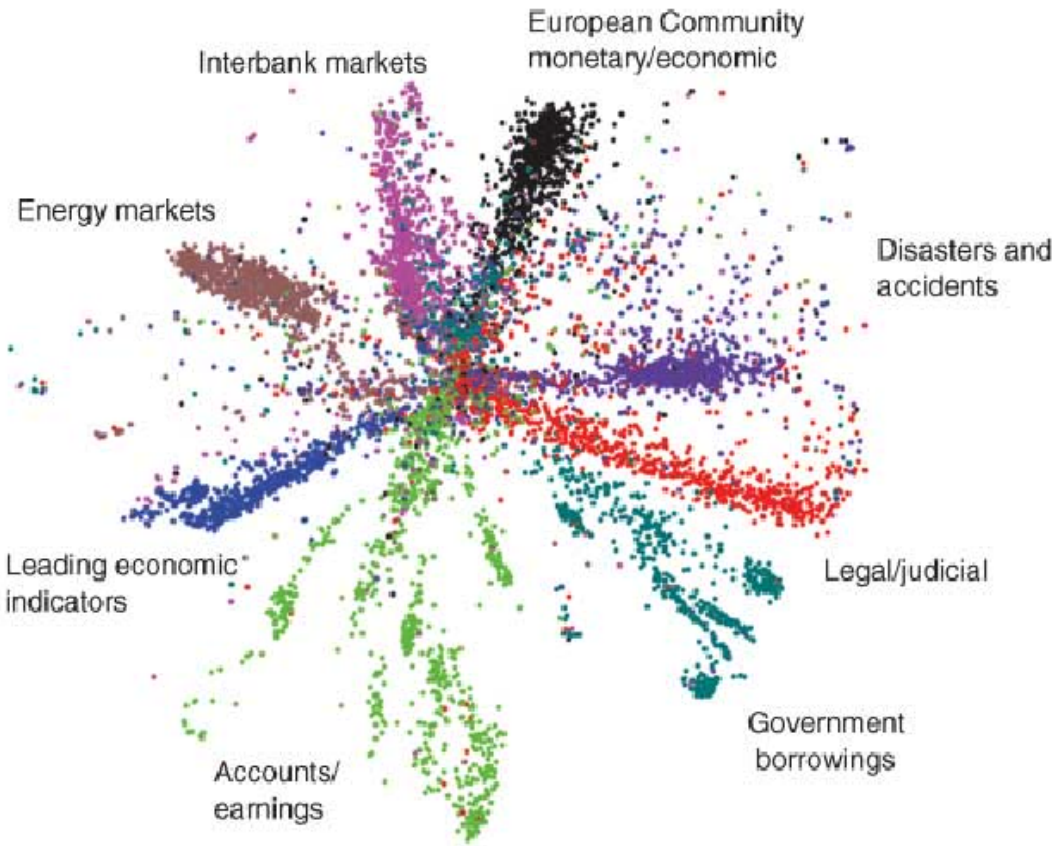
Codes from a 784-1000-500-250-2 autoencoder.

# example: autoencoding documents

codes from
2 dimensional LSA



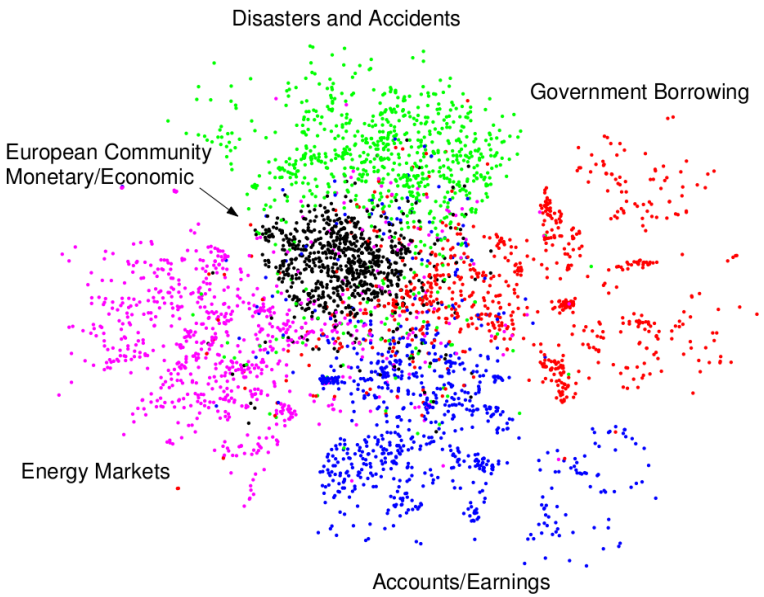
codes from a
2000-500-250-125-2 autoencoder




Hinton & Salakhutdinov, *Science*, 2007

# semantic hashing

- Suppose the input is (say) word counts from documents
- Suppose the bottleneck is (say) 32 binary neurons
- Surprised? *cf.* 20 questions...
- That's a 32-bit hashcode of the document
- It's a "semantic" hash: similar documents will have similar codes
- We can navigate to documents that are "one question away" (without knowing what the question is..!)



# summary

1. motivations
2. sigmoid belief nets
   - why are they hard to train?
   - could layer-by-layer training work?
3. Boltzmann machines
   - why are they hard to train?
   - the restricted Boltzmann machine (RBM)
4. deep belief nets
   - how to do it
   - why it works
   - fine-tuning the result *[Key: untying the weights]*
   - 2 applications: a classifier and an autoencoder