2019

"Deep Learning Lecture"

# Lecture 7: Generative Model (1)

Wei Wang

Center for Research on Intelligent Perception and Computing (CRIPAC)

National Laboratory of Pattern Recognition (NLPR)

Institute of Automation, Chinese Academy of Science (CASIA)

# Outline

1. Course Review
2. Linear Factor Model
3. Autoencoder
4. DBN and RBM

# Review: Regularization Strategies

- Parameter Norm Penalties
- Dataset Augmentation
- Noise Robustness
- Early Stopping
- Parameter Tying and Parameter Sharing
- Multitask Learning
- Bagging and Other Ensemble Methods
- Dropout
- Adversarial Training

# Review: Optimization

- Things we have looked at last week
  - Stochastic Gradient Descent
  - Momentum Method and the Nesterov Variant
  - Adaptive Learning Methods (AdaGrad, RMSProp, Adam)
  - Batch Normalization
  - Initialization Heuristics

# Outline

1. Course Review
2. Linear Factor Model
3. Autoencoder
4. DBN and RBM

# Linear Factor Model

- We want to build a probabilistic model of the input $\tilde{P}(\mathbf{x})$
- Like before, we are interested in latent factors **h** that explain **x**
- We then care about the marginal:

$$\tilde{P}(\mathbf{x}) = \mathbb{E}_{\mathbf{h}} \tilde{P}(\mathbf{x}|\mathbf{h})$$

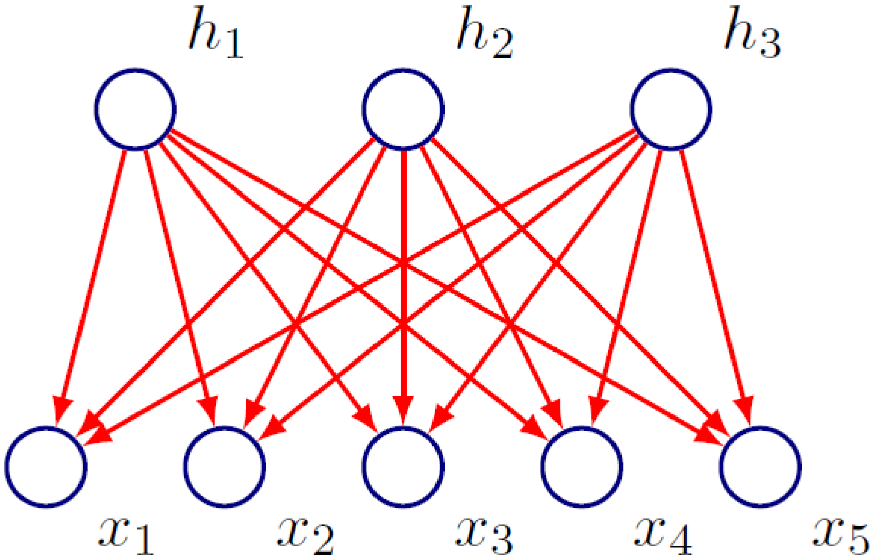
- **h** is a representation of the data

# Linear Factor Model

- The latent factor **h** is an encoding of the data
- Simplest decoding model: Get **x** after a linear transformation of **h** with some noise
- Formally: Suppose we sample the latent factors from a distribution $\mathbf{h} \sim P(\mathbf{h})$
- Then: $\mathbf{x} = W\mathbf{h} + \mathbf{b} + \varepsilon$

# Linear Factor Model

- P(**h**) is a factorial distribution




$$\mathbf{x} = W\mathbf{h} + \mathbf{b} + \epsilon$$

- How do learn in such a model?
- Let's look at a simple example

# Probabilistic PCA

- Suppose underlying latent factor has a Gaussian distribution

$$\mathbf{h} \sim \mathcal{N}(\mathbf{h}; 0, I)$$

- For the noise model: Assume $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$
- Then:

$$P(\mathbf{x}|\mathbf{h}) = \mathcal{N}(\mathbf{x}|W\mathbf{h} + \mathbf{b}, \sigma^2 I)$$

- We care about the marginal P(x) (predictive distribution):

$$P(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mathbf{b}, WW^T + \sigma^2 I)$$

# Probabilistic PCA

$$P(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mathbf{b}, WW^T + \sigma^2 I)$$

- How do we learn the parameters? (EM, ML Estimation)
- Let's look at the ML Estimation:
  - Let $C = WW^T + \sigma^2 I$
  - We want to maximize $\ell(\theta; X) = \sum_i \log P(\mathbf{x}_i|\theta)$

# Probabilistic PCA: ML Estimation

$$
\begin{aligned}
\ell(\theta; X) &= \sum_i \log P(\mathbf{x}_i|\theta) \\
&= -\frac{N}{2}\log|C| - \frac{1}{2}\sum_i (\mathbf{x}_i - \mathbf{b})C^{-1}(\mathbf{x}_i - \mathbf{b})^T \\
&= -\frac{N}{2}\log|C| - \frac{1}{2}Tr[(C^{-1}\sum_i |\mathbf{x}_i - \mathbf{b})(\mathbf{x}_i - \mathbf{b})^T] \\
&= \frac{N}{2}\log|C| - \frac{1}{2}Tr[(C^{-1}S]
\end{aligned}
$$

- Now fit the parameters $\theta$= W, b, $\sigma$ to maximize log-likelihood
- Can also use EM

# Factor Analysis

- Fix the latent factor prior to be the unit Gaussian as before:

$$\mathbf{h} \sim \mathcal{N}(\mathbf{h}; 0, I)$$

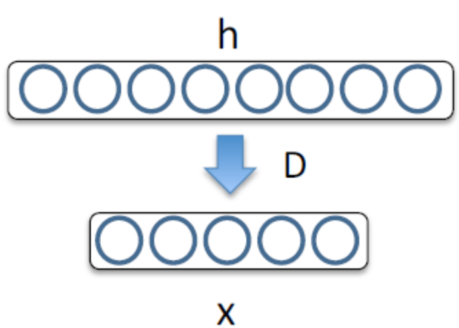
- Noise is sampled from a Gaussian with a diagonal covariance:

$$\Psi = diag([\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2])$$

- Still consider linear relationship between inputs and observed variables: Marginal $P(\mathbf{x}) \sim \mathcal{N}(\mathbf{x}; b, WW^T + \Psi)$

# Sparse Coding

- Sparse coding (Olshausen & Field, 1996). Originally developed to explain early visual processing in the brain (edge detection).
- For each input $\mathbf{x}^{(t)}$ find a latent representation $\mathbf{h}^{(t)}$ such that:
  - it is sparse: the vector $\mathbf{h}^{(t)}$ has many zeros
  - we can reconstruct the original input $\mathbf{x}^{(t)}$



# Sparse Coding

- For each $\mathbf{x}^{(t)}$ find a latent representation $\mathbf{h}^{(t)}$ such that:
  - it is sparse: the vector $\mathbf{h}^{(t)}$ has many zeros
  - we can reconstruct the original input $\mathbf{x}^{(t)}$

- In other words:

Reconstruction: $\widehat{\mathbf{x}}^{(t)}$

Sparsity vs. reconstruction control

$$\min_{\mathbf{D}} \frac{1}{T} \sum_{t=1}^{T} \min_{\mathbf{h}^{(t)}} \underbrace{\frac{1}{2}||\mathbf{x}^{(t)} - \mathbf{D}\,\mathbf{h}^{(t)}||_2^2}_{\text{Reconstruction error}} + \underbrace{\lambda||\mathbf{h}^{(t)}||_1}_{\text{Sparsity penalty}}$$

# Sparse Coding

- For each $\mathbf{x}^{(t)}$ find a latent representation $\mathbf{h}^{(t)}$ such that:
  - it is sparse: the vector $\mathbf{h}^{(t)}$ has many zeros
  - we can good reconstruct the original input $\mathbf{x}^{(t)}$

- In other words:

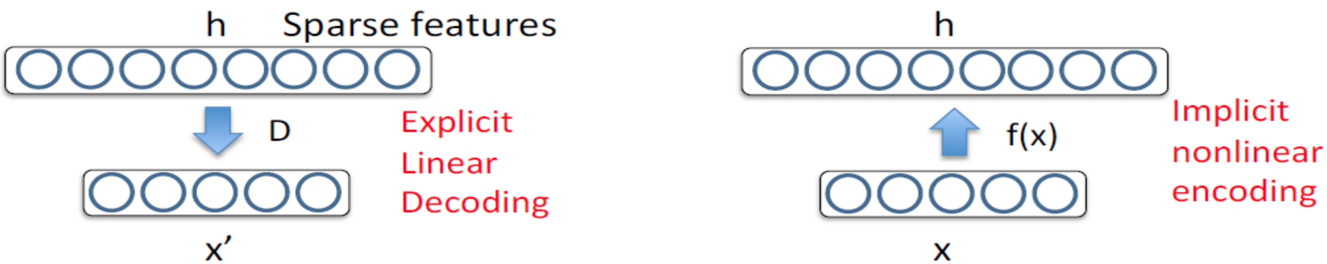
$$\min_{\mathbf{D}} \frac{1}{T} \sum_{t=1}^{T} \min_{\mathbf{h}^{(t)}} \frac{1}{2} ||\mathbf{x}^{(t)} - \mathbf{D}\,\mathbf{h}^{(t)}||_2^2 + \lambda ||\mathbf{h}^{(t)}||_1$$

  - we also constrain the columns of D to be of norm 1
  - otherwise, D could grow big while h becomes small to satisfy the L1 constraint

# Interpreting Sparse Coding

Interpreting Sparse Coding

$$\min_{\mathbf{D}} \frac{1}{T} \sum_{t=1}^{T} \min_{\mathbf{h}^{(t)}} \frac{1}{2} ||\mathbf{x}^{(t)} - \mathbf{D}\ \mathbf{h}^{(t)}||_2^2 + \lambda ||\mathbf{h}^{(t)}||_1$$

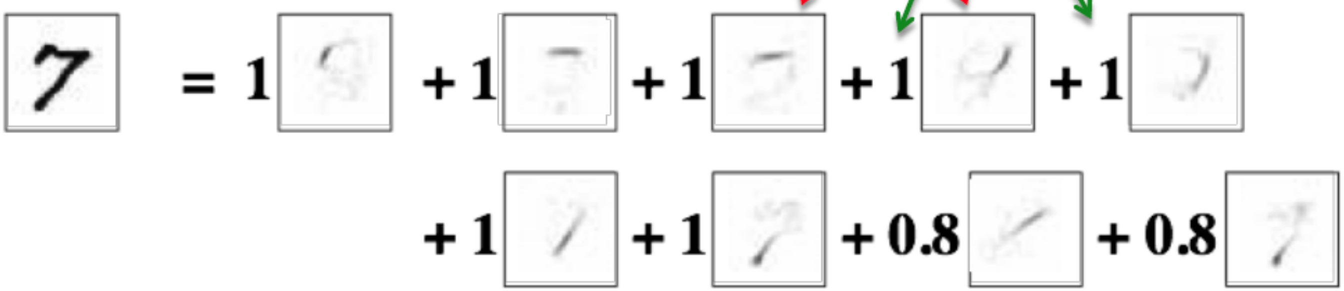



- Sparse, over-complete representation h.
- Encoding h = f(x) is implicit and nonlinear function of x.
- Reconstruction (or decoding) x' = Dh is linear and explicit.

# Sparse Coding

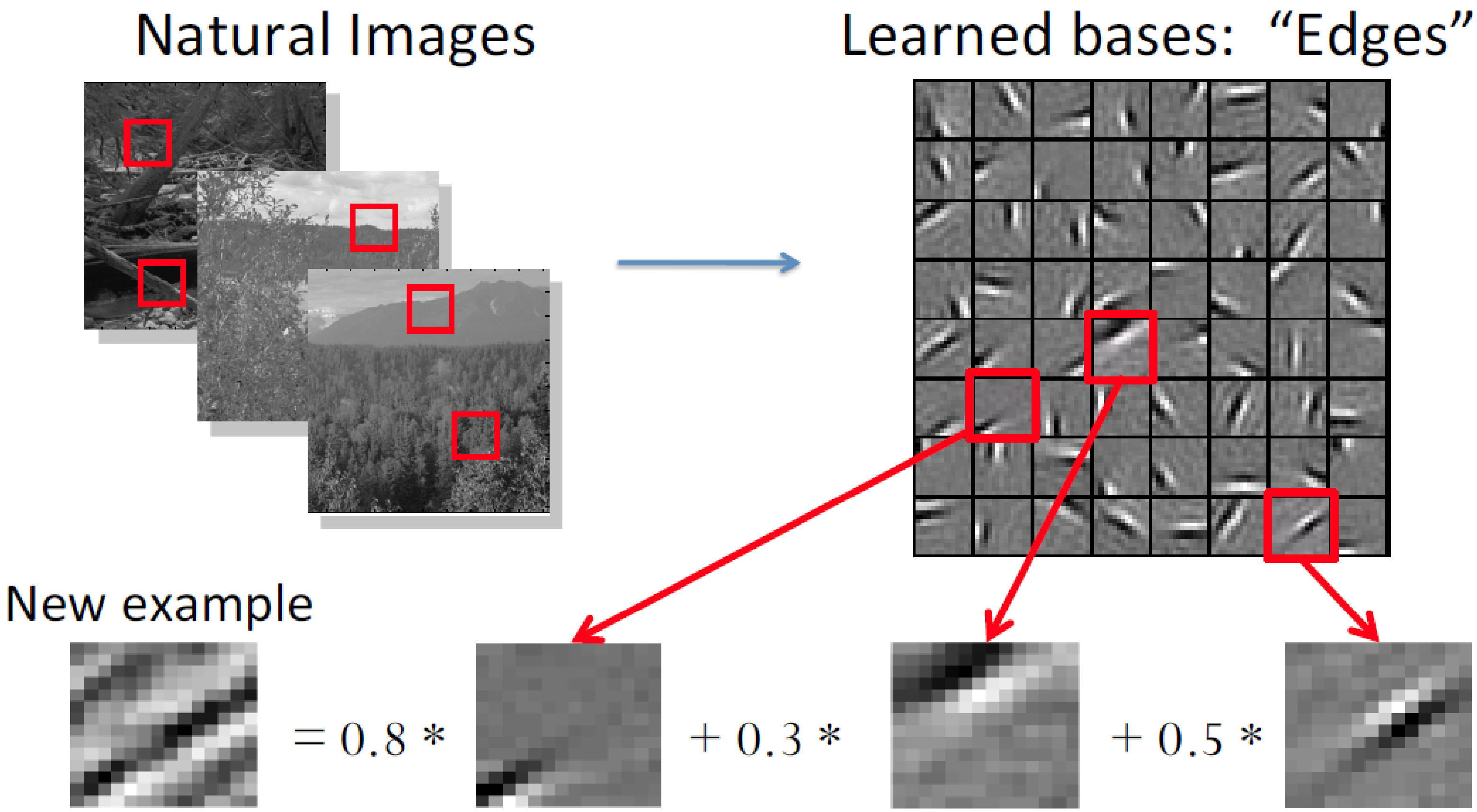
- We can also write:

$$\widehat{\mathbf{x}}^{(t)} = \mathbf{D}\,\mathbf{h}(\mathbf{x}^{(t)}) = \sum_{\substack{k \text{ s.t.} \\ h(\mathbf{x}^{(t)})_k \neq 0}} \mathbf{D}_{\cdot,k}\, h(\mathbf{x}^{(t)})_k$$




  - D is often referred to as Dictionary
  - In certain applications, we know what dictionary matrix to use
  - In many cases, we have to learn it

# Sparse Coding




$$x = 0.8 * \phi_{36} + 0.3 * \phi_{42} + 0.5 * \phi_{65}$$

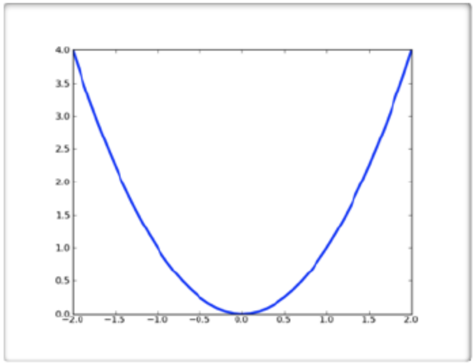
[0, 0, … **0.8**, …, **0.3**, …, **0.5**, …] = coefficients (feature representation)

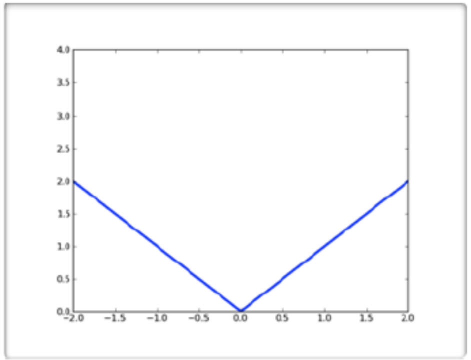
# Inference

- Given dictionary D , how do we compute $\mathbf{h}(\mathbf{x}^{(t)})$
  - We need to optimize:

$$l(\mathbf{x}^{(t)}) = \frac{1}{2}||\mathbf{x}^{(t)} - \mathbf{D}\,\mathbf{h}^{(t)}||_2^2 + \lambda||\mathbf{h}^{(t)}||_1$$

  - This is Lasso.



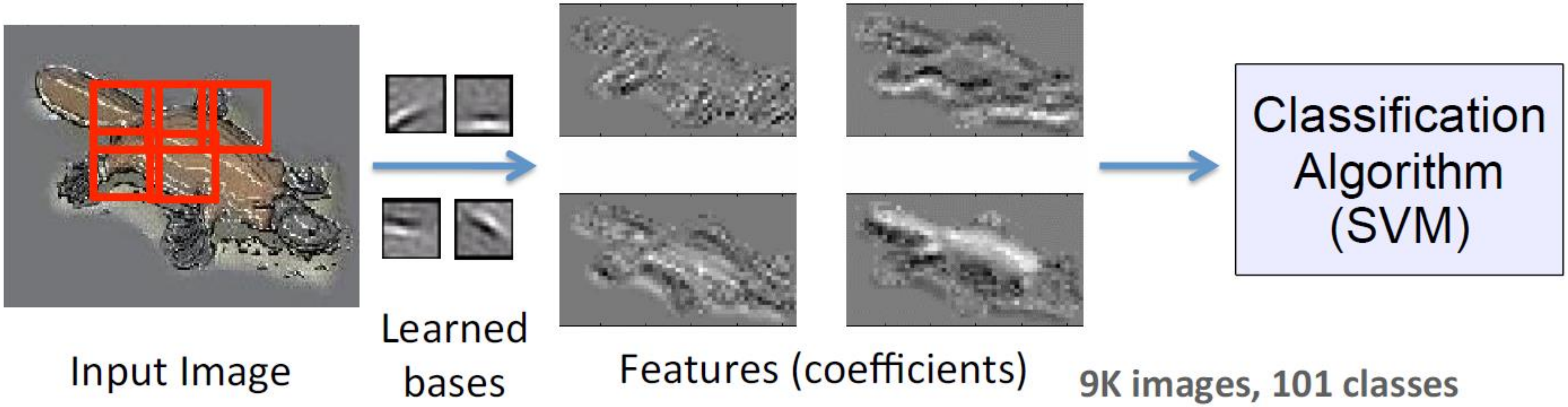


  - We could use a gradient descent method:

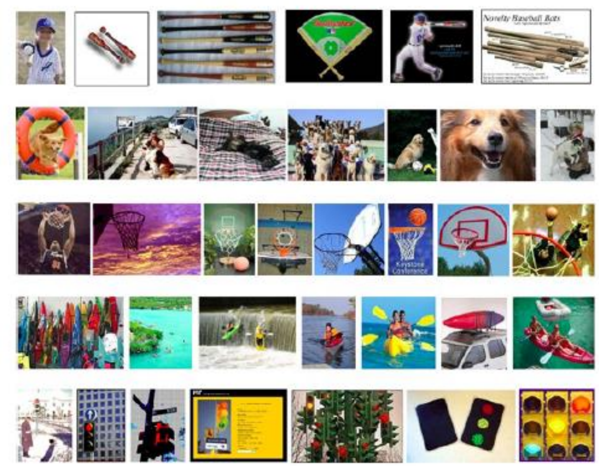
$$\nabla_{\mathbf{h}^{(t)}} l(\mathbf{x}^{(t)}) = \mathbf{D}^\top(\mathbf{D}\,\mathbf{h}^{(t)} - \mathbf{x}^{(t)}) + \lambda\,\mathrm{sign}(\mathbf{h}^{(t)})$$

# Image Classification

- Evaluated on Caltech101 object category dataset




9K images, 101 classes



| Algorithm | Accuracy |
|---|---|
| Baseline (Fei-Fei et al., 2004) | 16% |
| PCA | 37% |
| **Sparse Coding** | **47%** |

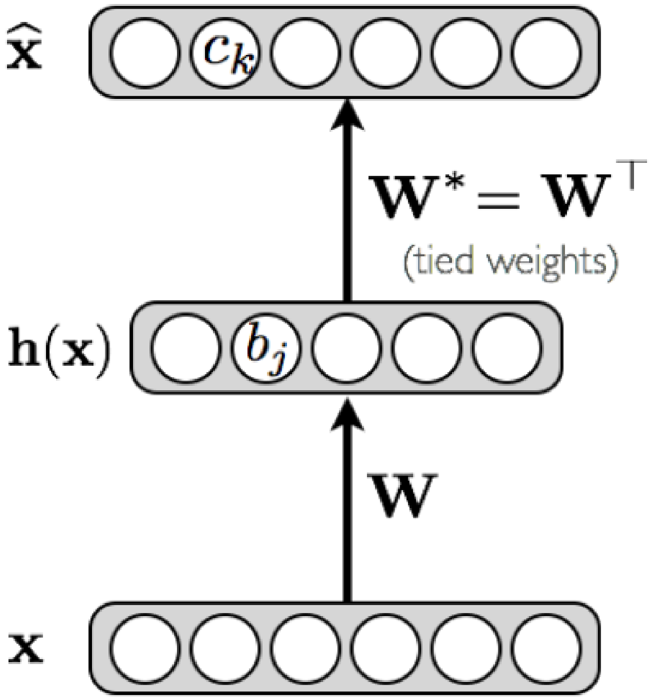
Lee et al., NIPS 2006

# Outline

1. Course Review
2. Linear Factor Model
3. Autoencoder
4. DBN and RBM

# Autoencoder

- Feed-forward neural network trained to reproduce its input at the output layer



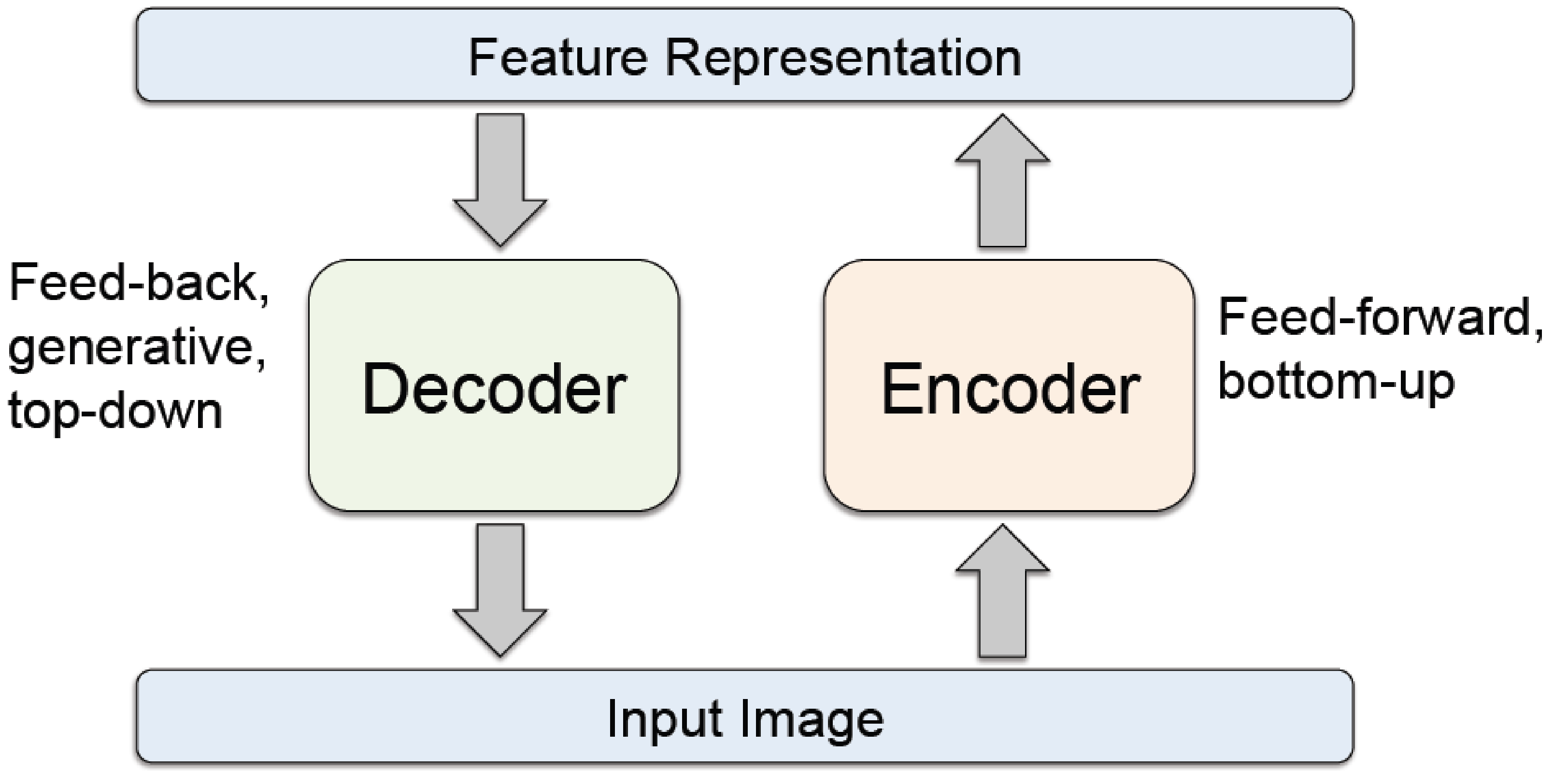

Decoder

$$\begin{aligned}\widehat{\mathbf{x}} &= o(\widehat{\mathbf{a}}(\mathbf{x})) \\ &= \underbrace{\mathrm{sigm}}_{\text{For binary units}}(\mathbf{c} + \mathbf{W}^*\mathbf{h}(\mathbf{x}))\end{aligned}$$

Encoder

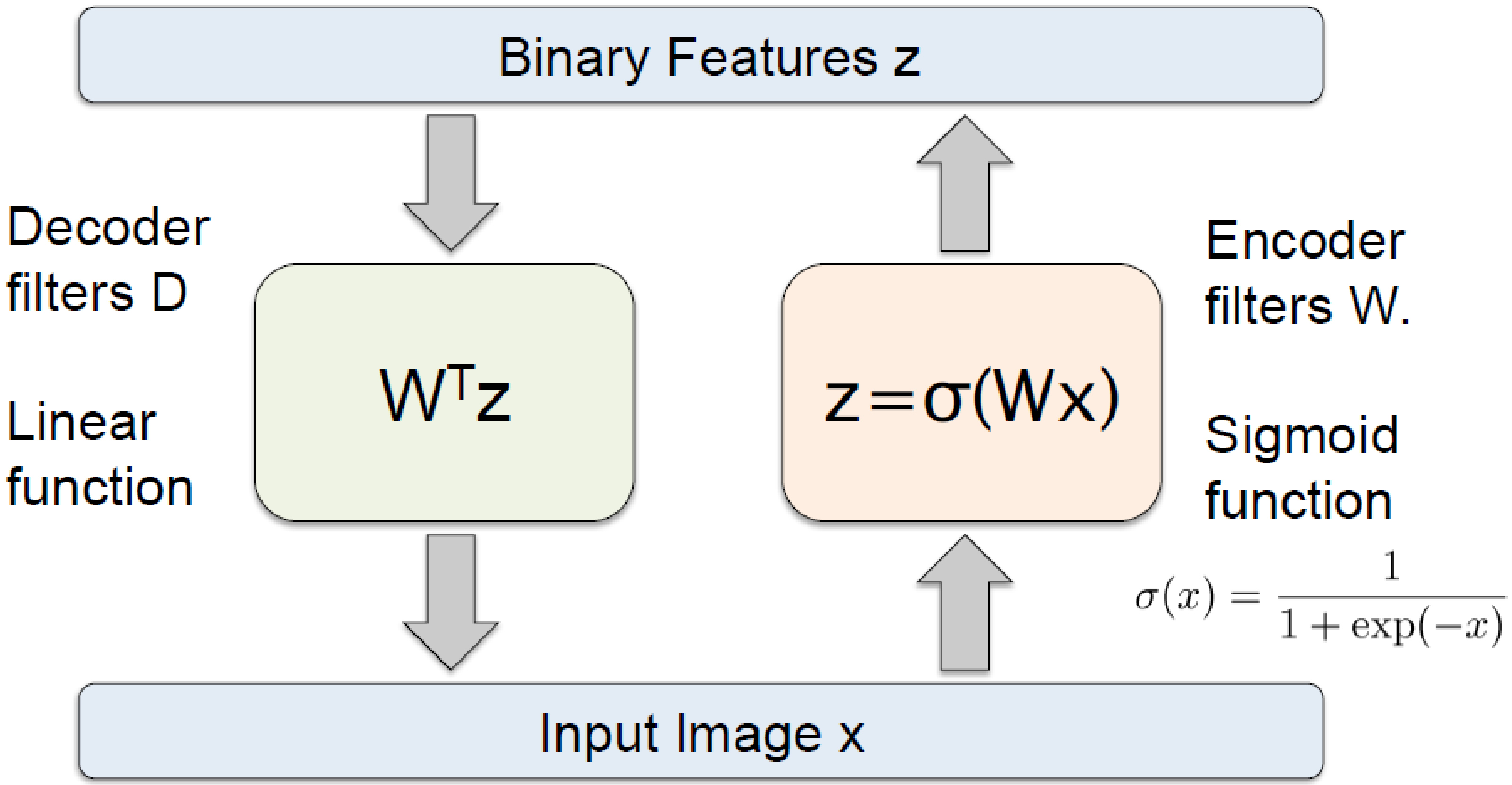
$$\begin{aligned}\mathbf{h}(\mathbf{x}) &= g(\mathbf{a}(\mathbf{x})) \\ &= \mathrm{sigm}(\mathbf{b} + \mathbf{W}\mathbf{x})\end{aligned}$$

# Autoencoder

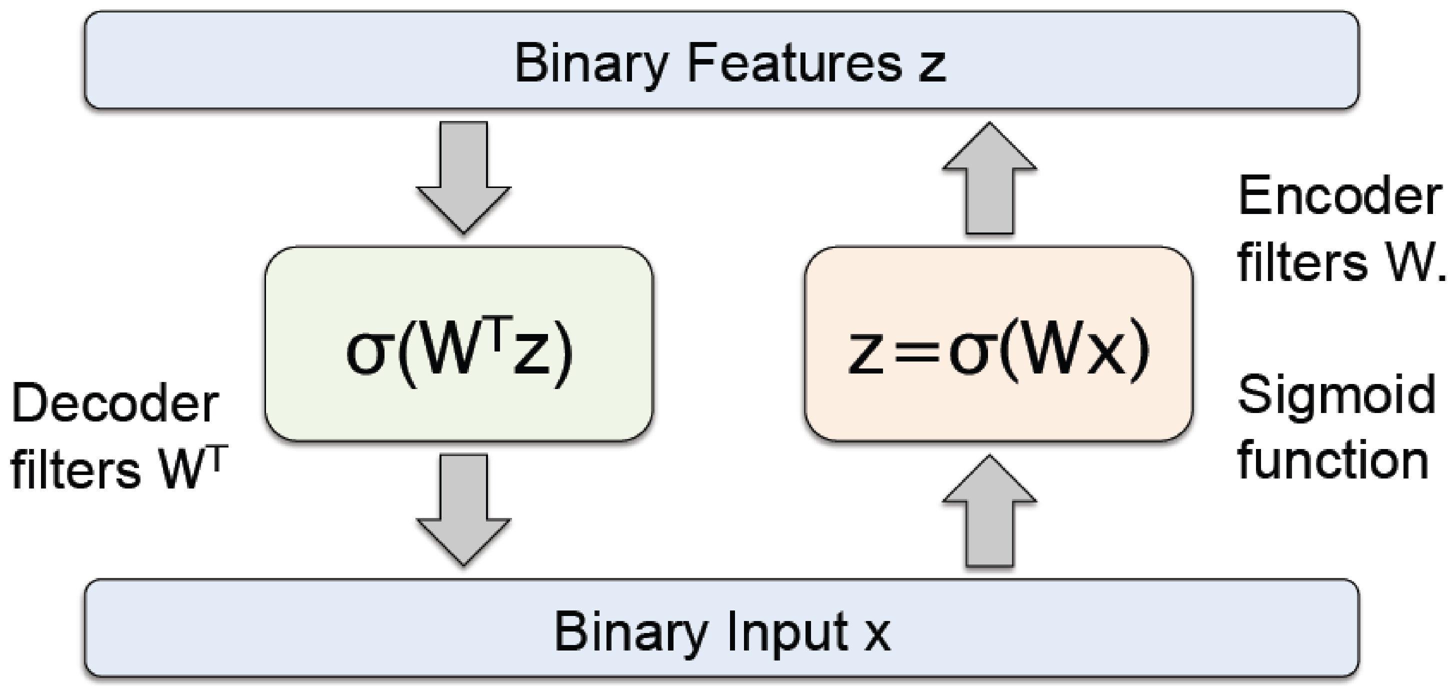



- Details of what goes insider the encoder and decoder matter!
- Need constraints to avoid learning an identity.

# Autoencoder



# Another Autoencoder Model




- Need additional constraints to avoid learning an identity.
- Relates to Restricted Boltzmann Machines
- Encoder and Decoder filters can be different.

# Loss Function

- Loss function for binary inputs

$$l(f(\mathbf{x})) = -\sum_k \left(x_k \log(\widehat{x}_k) + (1 - x_k) \log(1 - \widehat{x}_k)\right)$$

  - Cross-entropy error function (reconstruction loss) $f(\mathbf{x}) \equiv \widehat{\mathbf{x}}$

- Loss function for real-valued inputs

$$l(f(\mathbf{x})) = \frac{1}{2} \sum_k (\widehat{x}_k - x_k)^2$$

  - sum of squared differences (reconstruction loss)
  - we use a linear activation function at the output

# Loss Function

- For both cases, the gradient $\nabla_{\widehat{\mathbf{a}}(\mathbf{x}^{(t)})} l(f(\mathbf{x}^{(t)}))$ has a very simple form:

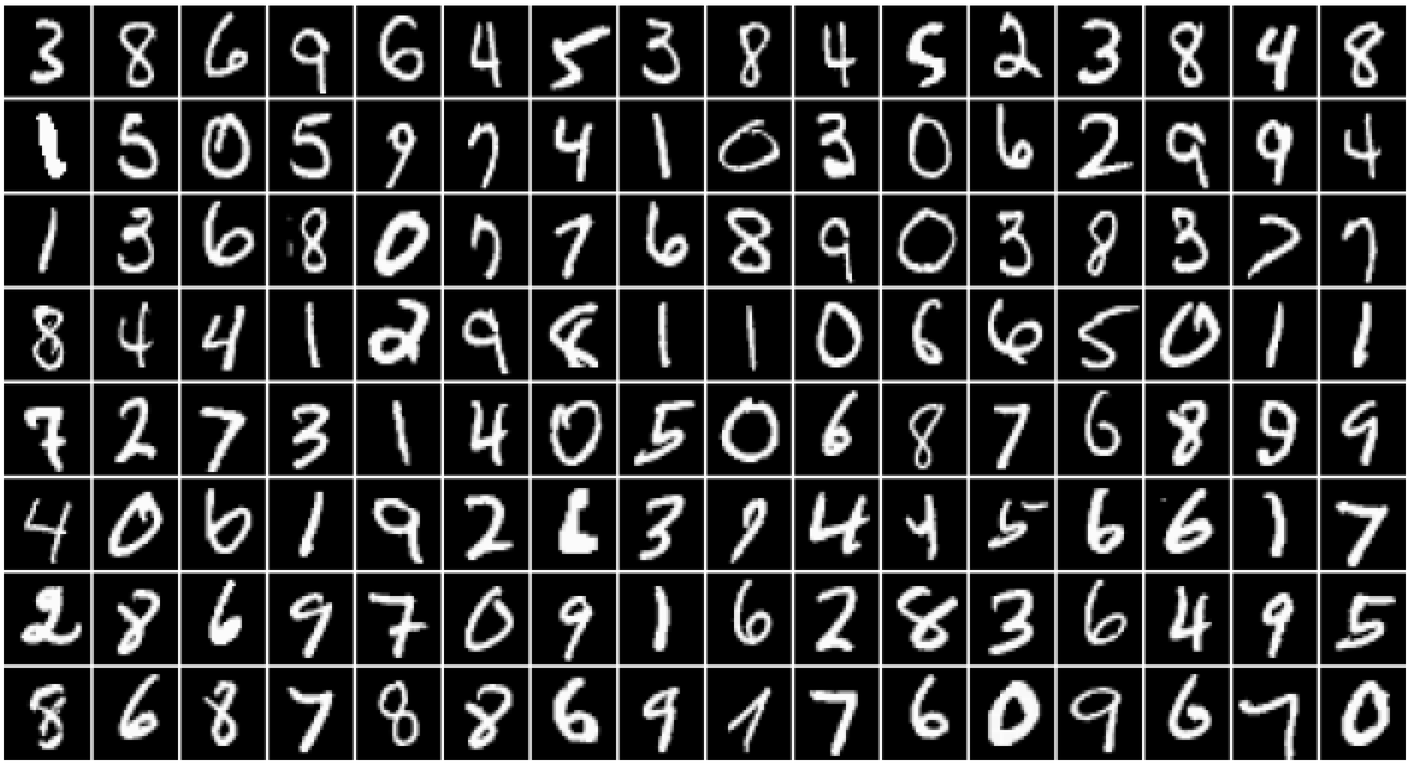
$$\nabla_{\widehat{\mathbf{a}}(\mathbf{x}^{(t)})} l(f(\mathbf{x}^{(t)})) = \widehat{\mathbf{x}}^{(t)} - \mathbf{x}^{(t)} \qquad f(\mathbf{x}) \equiv \widehat{\mathbf{x}}$$

- Parameter gradients are obtained by backpropagating the gradient $\nabla_{\widehat{\mathbf{a}}(\mathbf{x}^{(t)})} l(f(\mathbf{x}^{(t)}))$ like in a regular network

  - important: when using tied weights ($\mathbf{W}^* = \mathbf{W}^\top$), $\nabla_{\mathbf{W}} l(f(\mathbf{x}^{(t)}))$ is the sum of two gradients
  - this is because $\mathbf{W}$ is present in the encoder and in the decoder

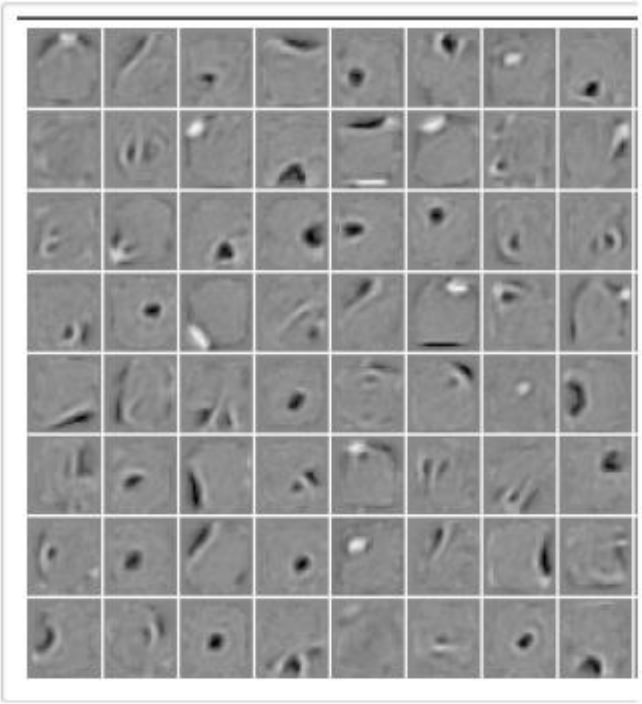
# Example: MNIST

- MNIST dataset:

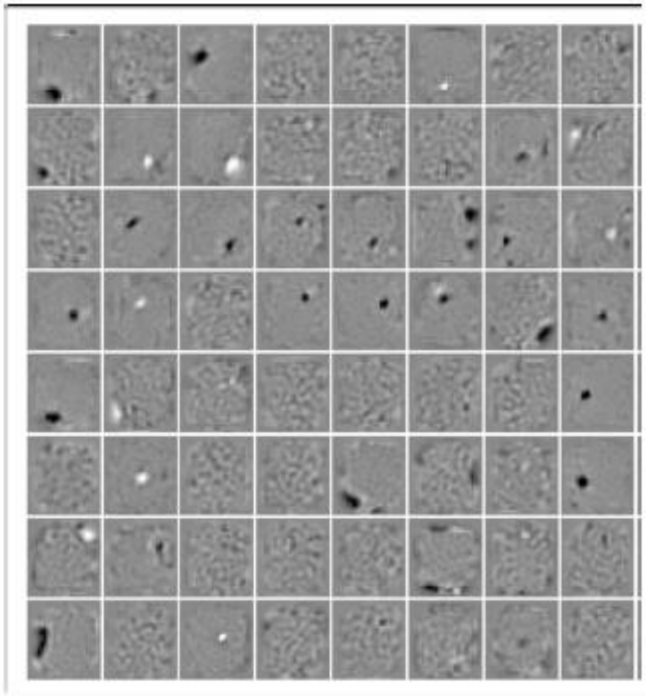


# Learned Features

- MNIST dataset:

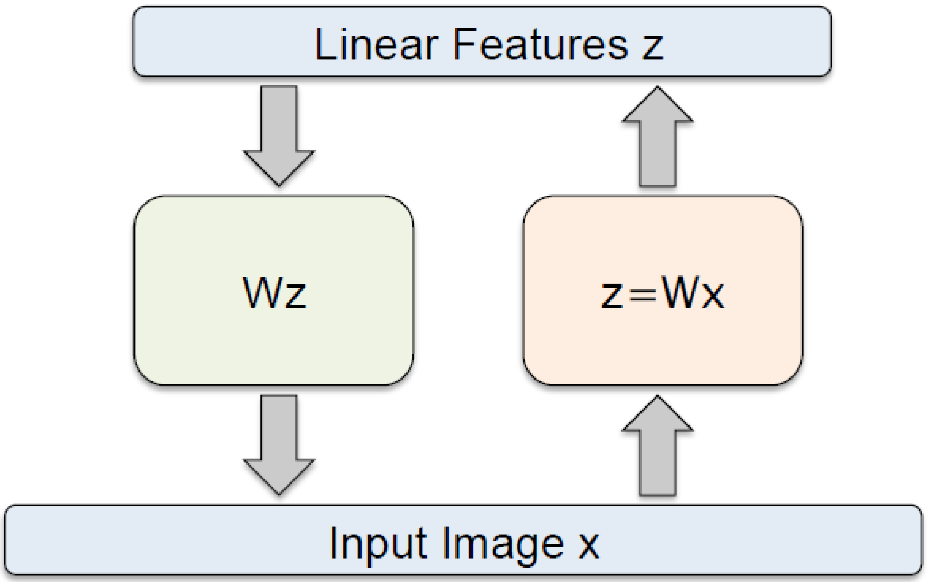


RBM



Autoenncoder

# Optimality of the Linear Autoencoder

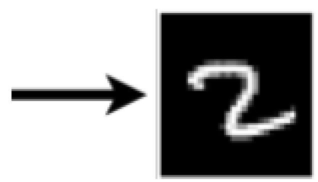


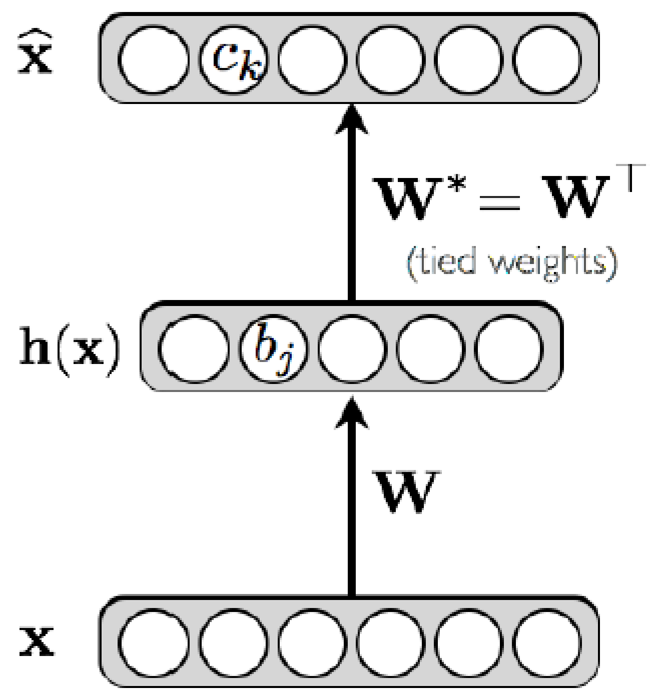

- If the hidden and output layers are linear, it will learn hidden units that are a linear function of the data and minimize the squared error.
- The K hidden units will span the same space as the first k principal components. The weight vectors may not be orthogonal.

- With nonlinear hidden units, we have a nonlinear generalization of PCA.

# Undercomplete Representation

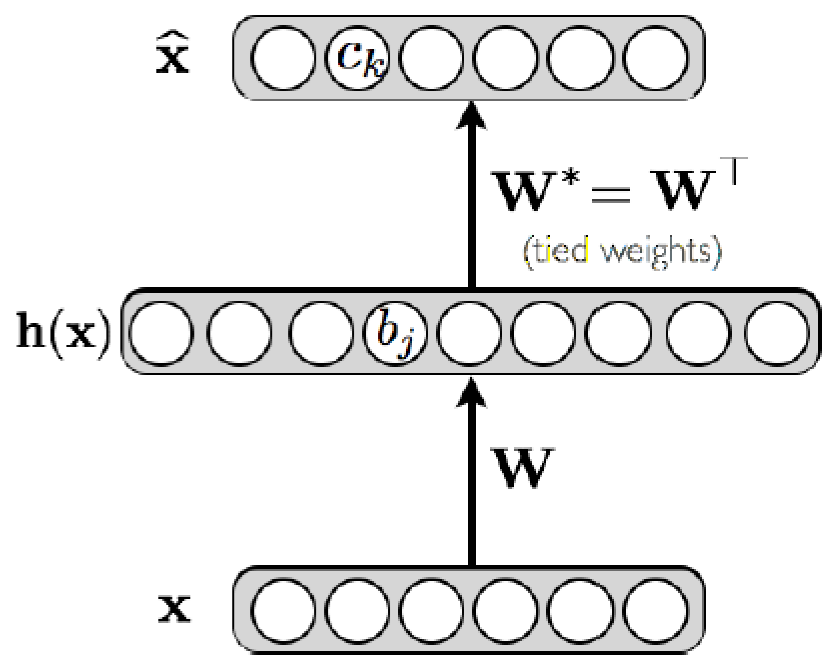
- Hidden layer is undercomplete if smaller than the input layer (bottleneck layer, e.g. dimensionality reduction):
  - hidden layer "compresses" the input
  - will compress well only for the training distribution
- Hidden units will be
  - good features for the training distribution
  - will not be robust to other types of input





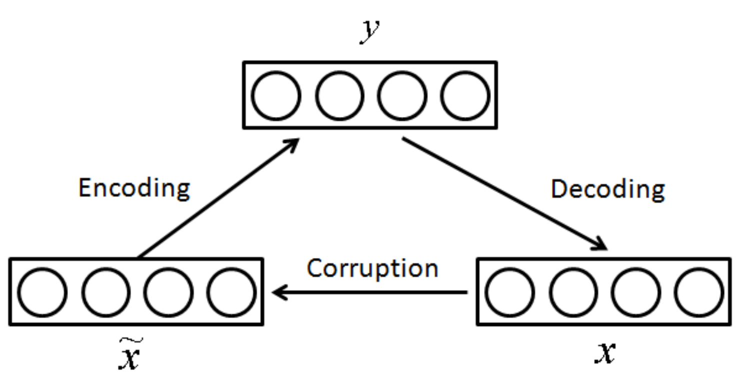
# Overcomplete Representation

- Hidden layer is overcomplete if greater than the input layer
  - no compression in hidden layer
  - each hidden unit could copy a different input component
- No guarantee that the hidden units will extract meaningful structure



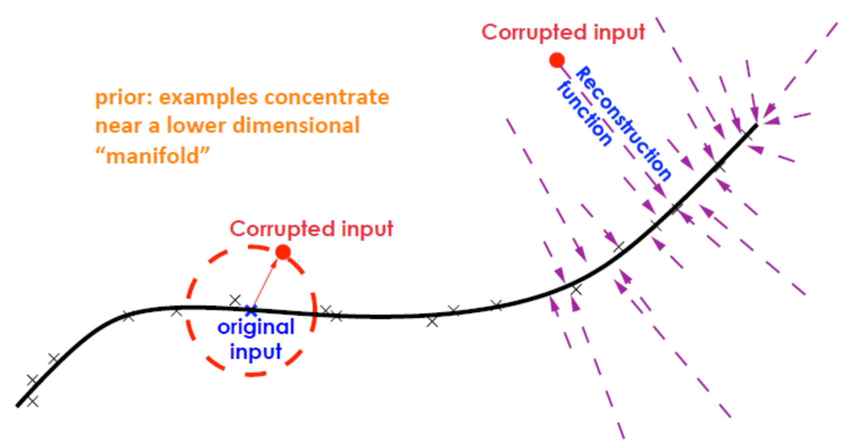
# Denoising Autoencoder

- In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity function, train the autoencoder to reconstruct the input from a corrupted version of it
  - Encode the input (preserve the information about the input)
  - Undo the effect of a corruption process stochastically applied to the input of the auto-encoder
- To convert the autoencoder to a denoising autoencoder, all we need to do is to add a stochastic corruption step operating on the input
  - Randomly sets some of the inputs (as many as half of them) to zero. Hence the denoising auto-encoder is trying to predict the corrupted (i.e. missing) values from the uncorrupted (i.e., non-missing) values, for randomly selected subsets of missing patterns.
  - The input can be corrupted in other ways



# Denoising Autoencoder

- The learner must capture the structure of the input distribution in order to optimally undo the effect of the corruption process, with the reconstruction essentially being a nearby but higher density point than the corrupted input
- The denoising autoencoder is learning a reconstruction function that corresponds to a vector field pointing towards high-density regions (the manifold where examples concentrate)
- Denosing autoencoder basically learns in $r(\tilde{\mathbf{x}}) - \tilde{\mathbf{x}}$ a vector pointing in the direction $\frac{\partial \log P(\tilde{\mathbf{x}})}{\partial \tilde{\mathbf{x}}}$

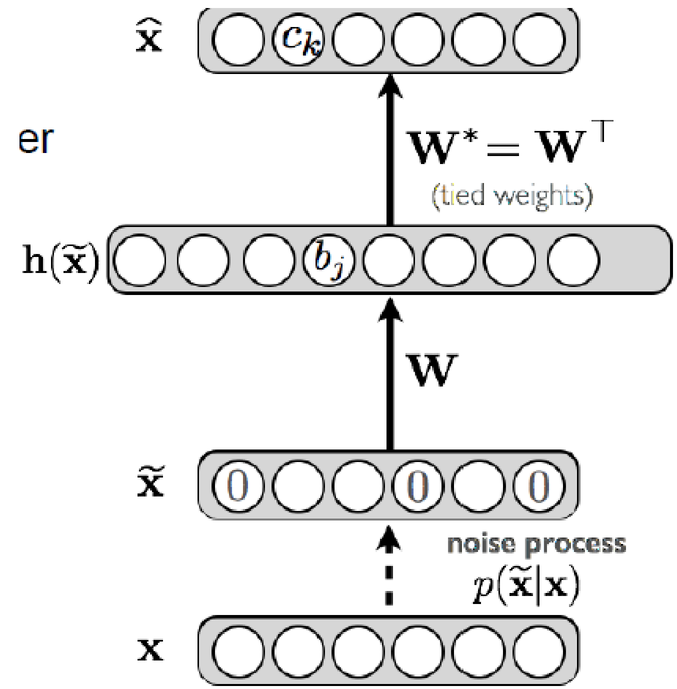


# Denoising Autoencoder

- Idea: representation should be robust to introduction of noise:
    - random assignment of subset of inputs to 0, with probability
    - Similar to dropouts on the input layer
    - Gaussian additive noise

- Reconstruction $\widehat{\mathbf{x}}$ computed from the corrupted input $\widetilde{\mathbf{x}}$

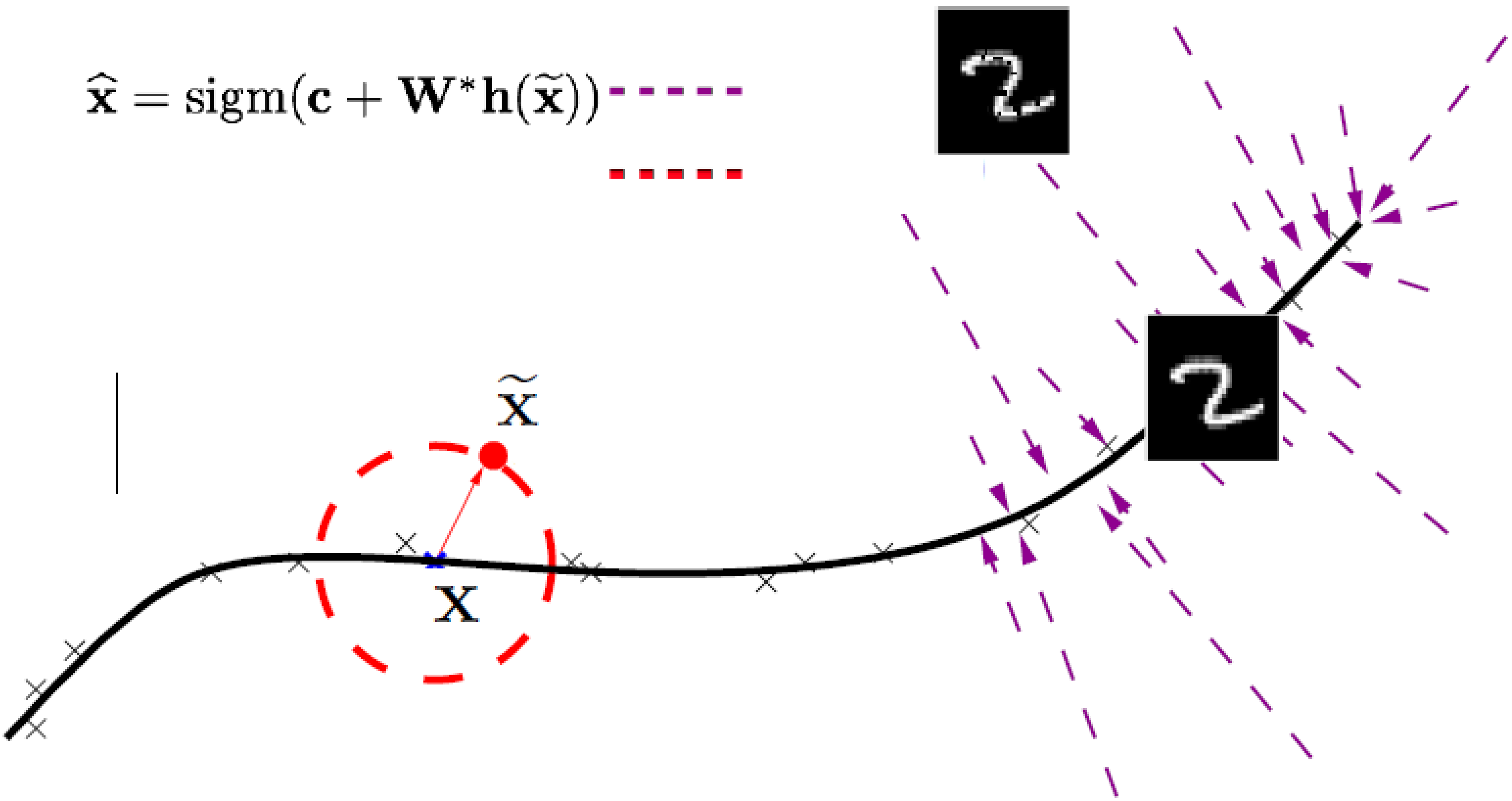
- Loss function compares $\widehat{\mathbf{x}}$ reconstruction with the noiseless input $\mathbf{x}$





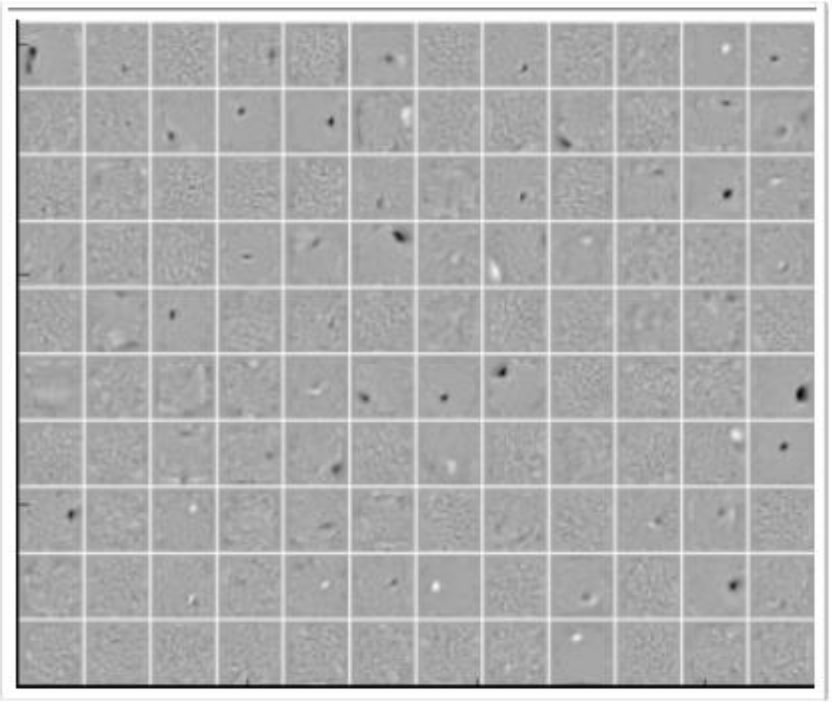

(Vincent et.al., ICML 2008)

# Denoising Autoencoder

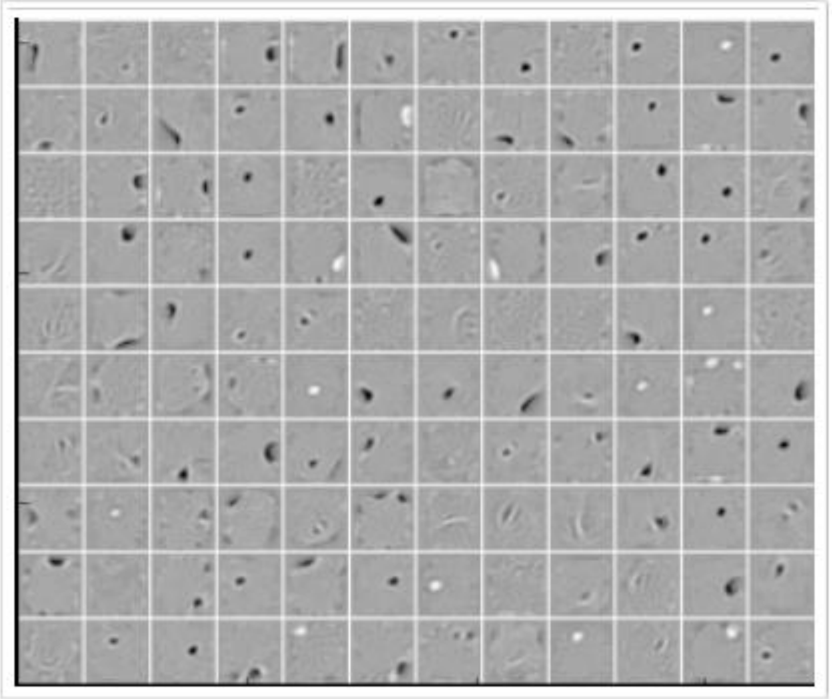


# Learned Filters

Non-corrupted

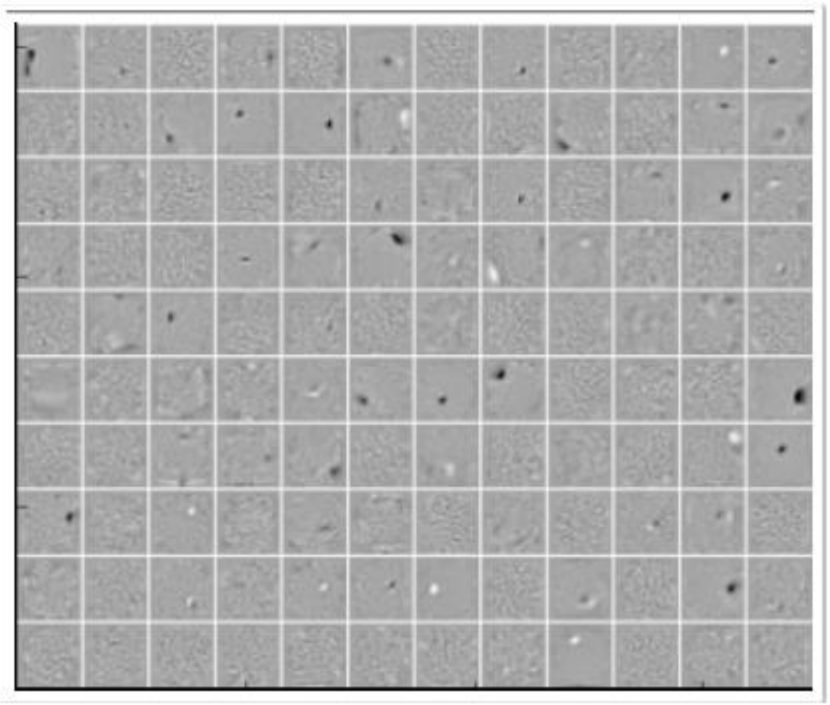


25% corrupted input

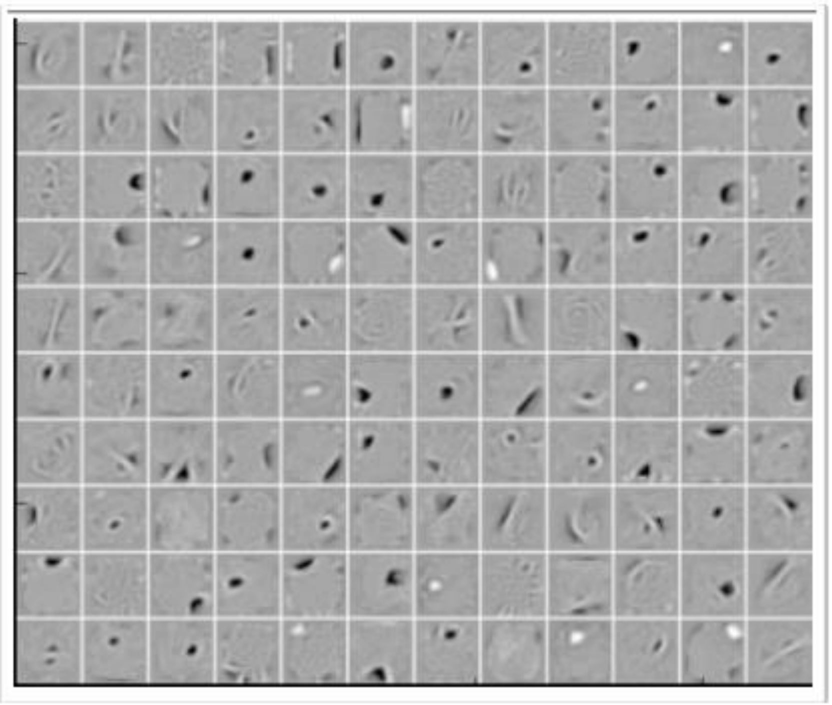


# Learned Filters

Non-corrupted

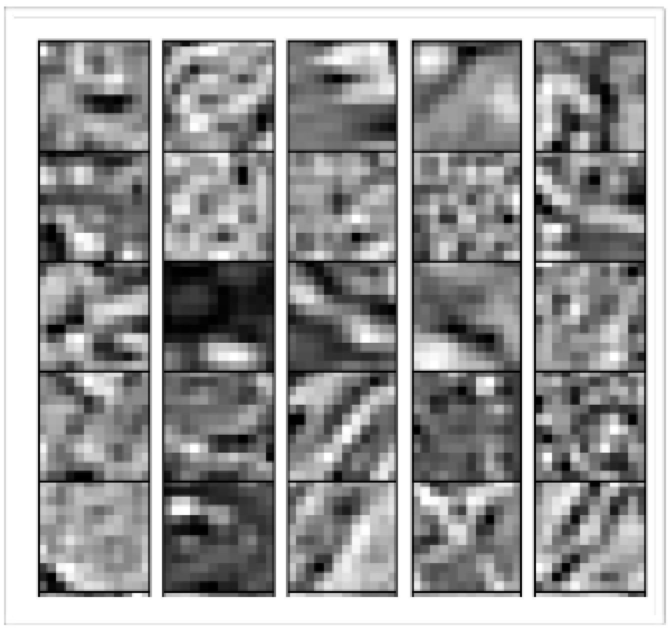


50% corrupted input

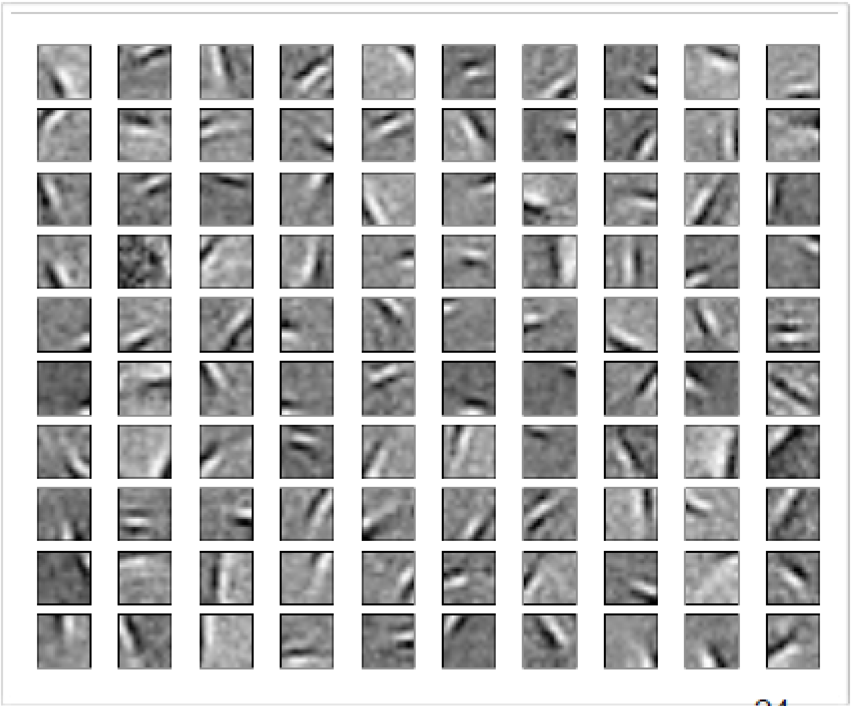


# Squared Error Loss

- Training on natural image patches, with squared loss
  - PCA may not the best solution



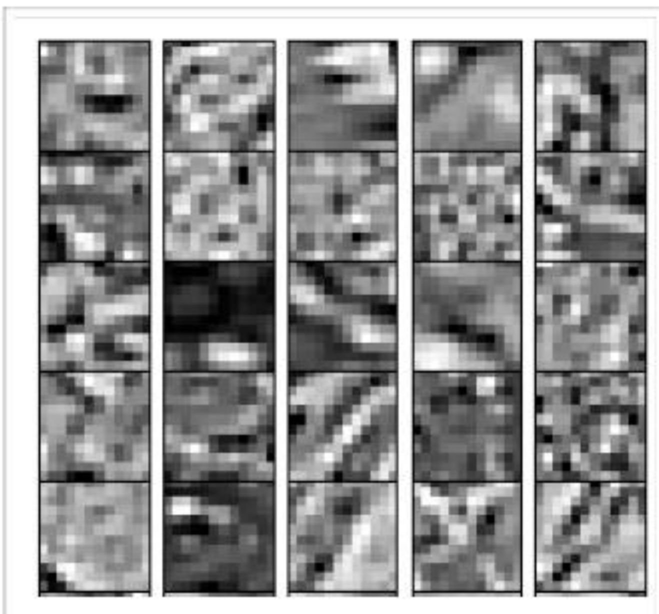
Data



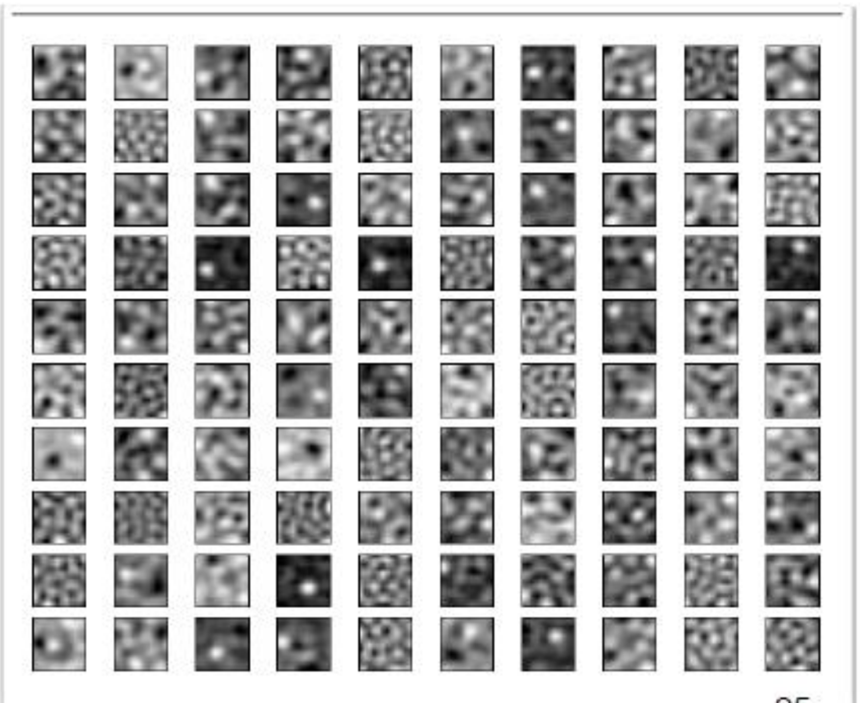
Filters

# Squared Error Loss

- Training on natural image patches, with squared loss
  - Not equivalent to weight decay



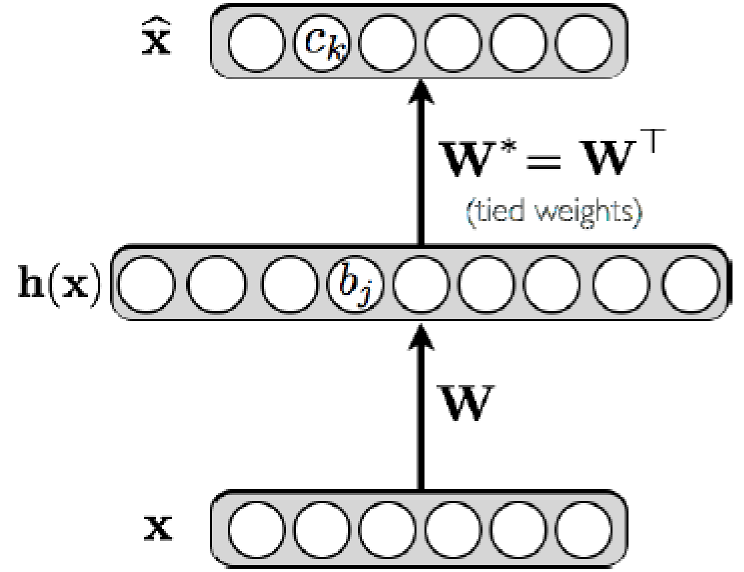
Data




Filters

# Contractive Autoencoder

- Alternative approach to avoid uninteresting solutions
  - add an explicit term in the loss that penalizes that solution
- We wish to extract features that only reflect variations observed in the training set
  - we'd like to be invariant to the other variations



# Contractive Autoencoder

- Consider the following loss function:

$$\underbrace{l(f(\mathbf{x}^{(t)}))}_{\text{Reconstruction Loss}} + \lambda \underbrace{||\nabla_{\mathbf{x}^{(t)}} \mathbf{h}(\mathbf{x}^{(t)})||_F^2}_{\text{Jacobian of Encoder}}$$

- For the binary observations:

$$l(f(\mathbf{x}^{(t)})) = -\sum_k \left( x_k^{(t)} \log(\widehat{x}_k^{(t)}) + (1 - x_k^{(t)}) \log(1 - \widehat{x}_k^{(t)}) \right)$$

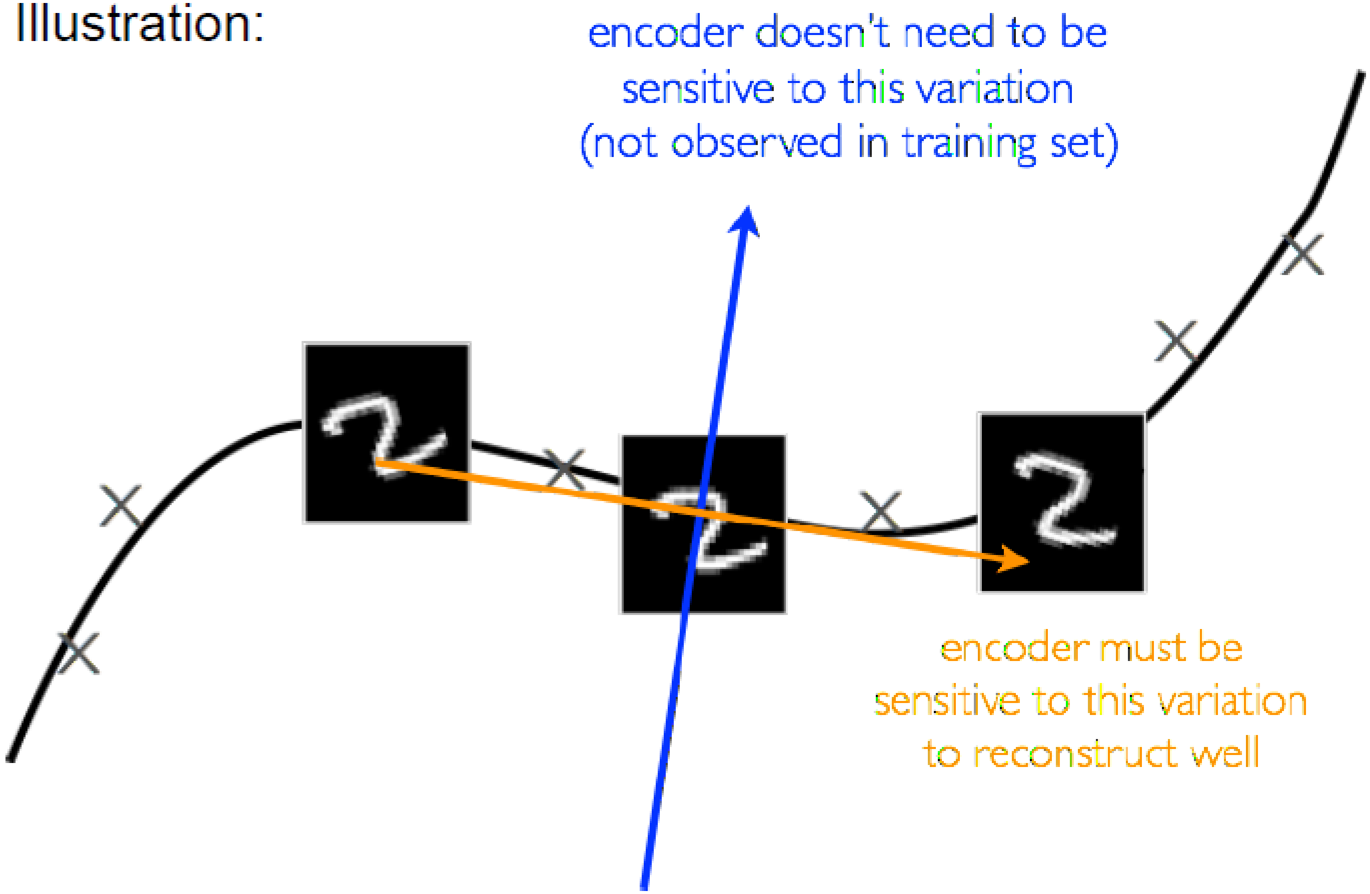
Autoencoder attempts to preserve all information

$$||\nabla_{\mathbf{x}^{(t)}} \mathbf{h}(\mathbf{x}^{(t)})||_F^2 = \sum_j \sum_k \left( \frac{\partial h(\mathbf{x}^{(t)})_j}{\partial x_k^{(t)}} \right)^2$$

Encoder throws away all information

# Contractive Autoencoder

- Illustration:

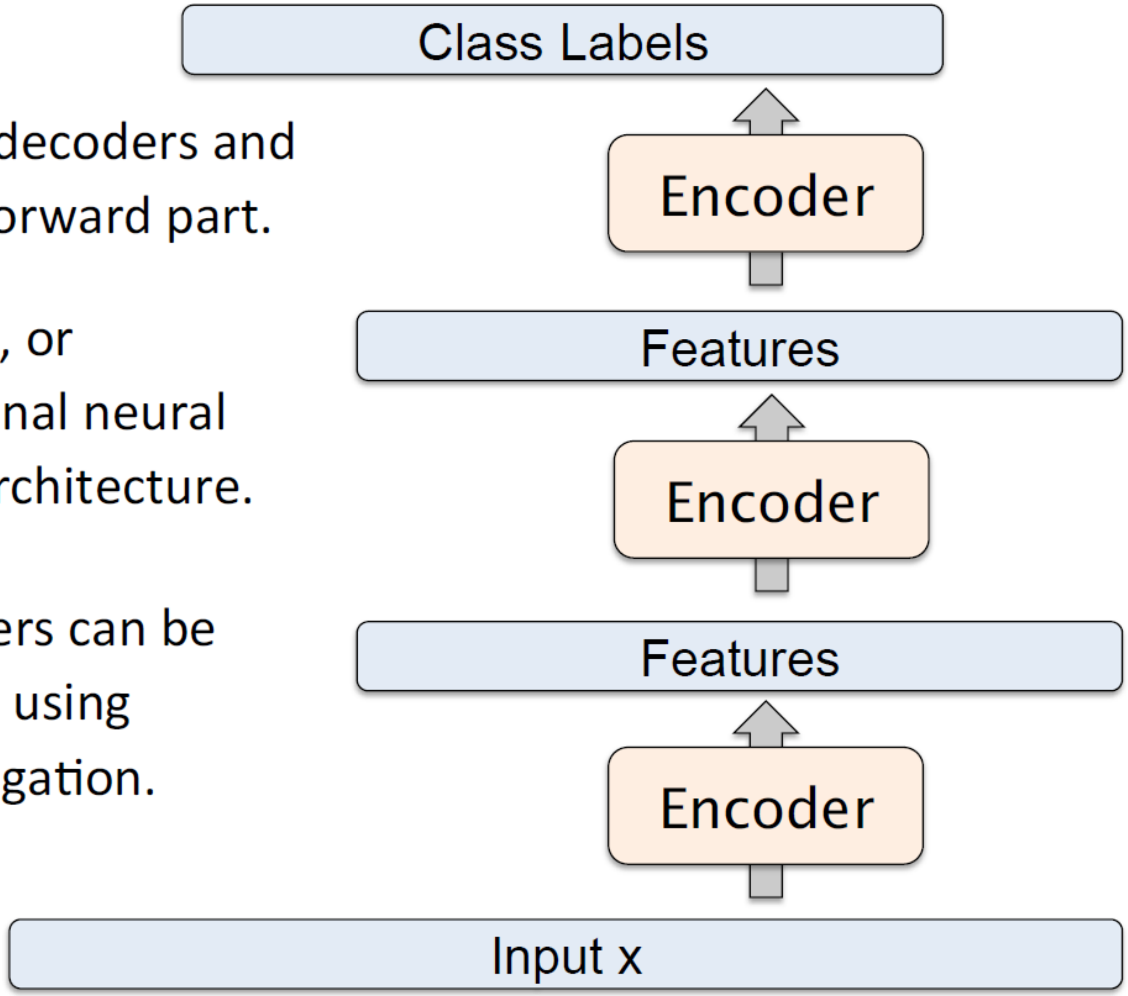


# Pros and Cons

- Advantage of denoising autoencoder: simpler to implement
  - requires adding one or two lines of code to regular autoencoder
  - no need to compute Jacobian of hidden layer
- Advantage of contractive autoencoder: gradient is deterministic
  - can use second order optimizers (conjugate gradient, LBFGS, etc.)
  - might be more stable than denoising autoencoder, which uses a sampled gradient

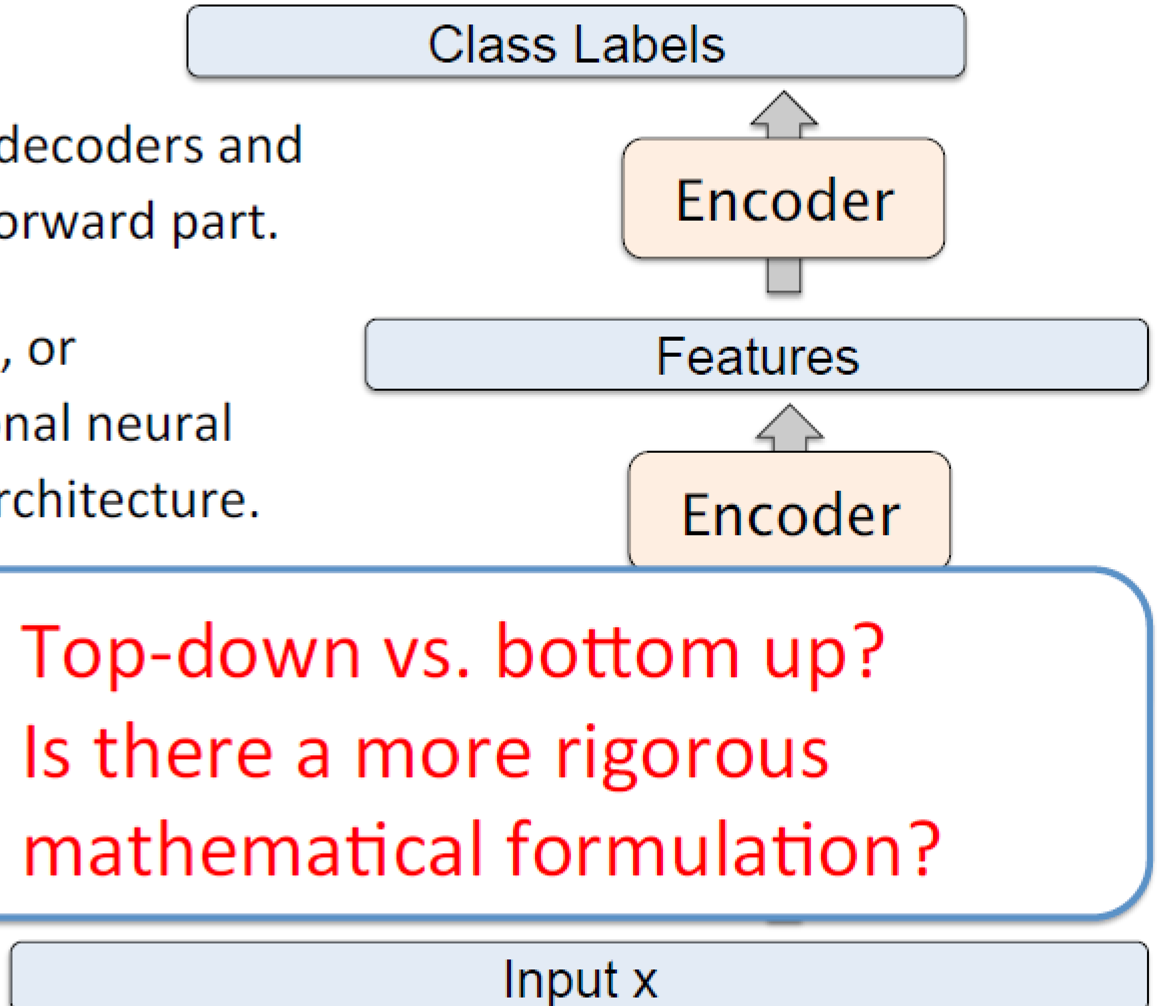
# Stacked Autoencoder

- Remove decoders and use feed-forward part.
- Standard, or convolutional neural network architecture.
- Parameters can be fine-tuned using backpropagation.



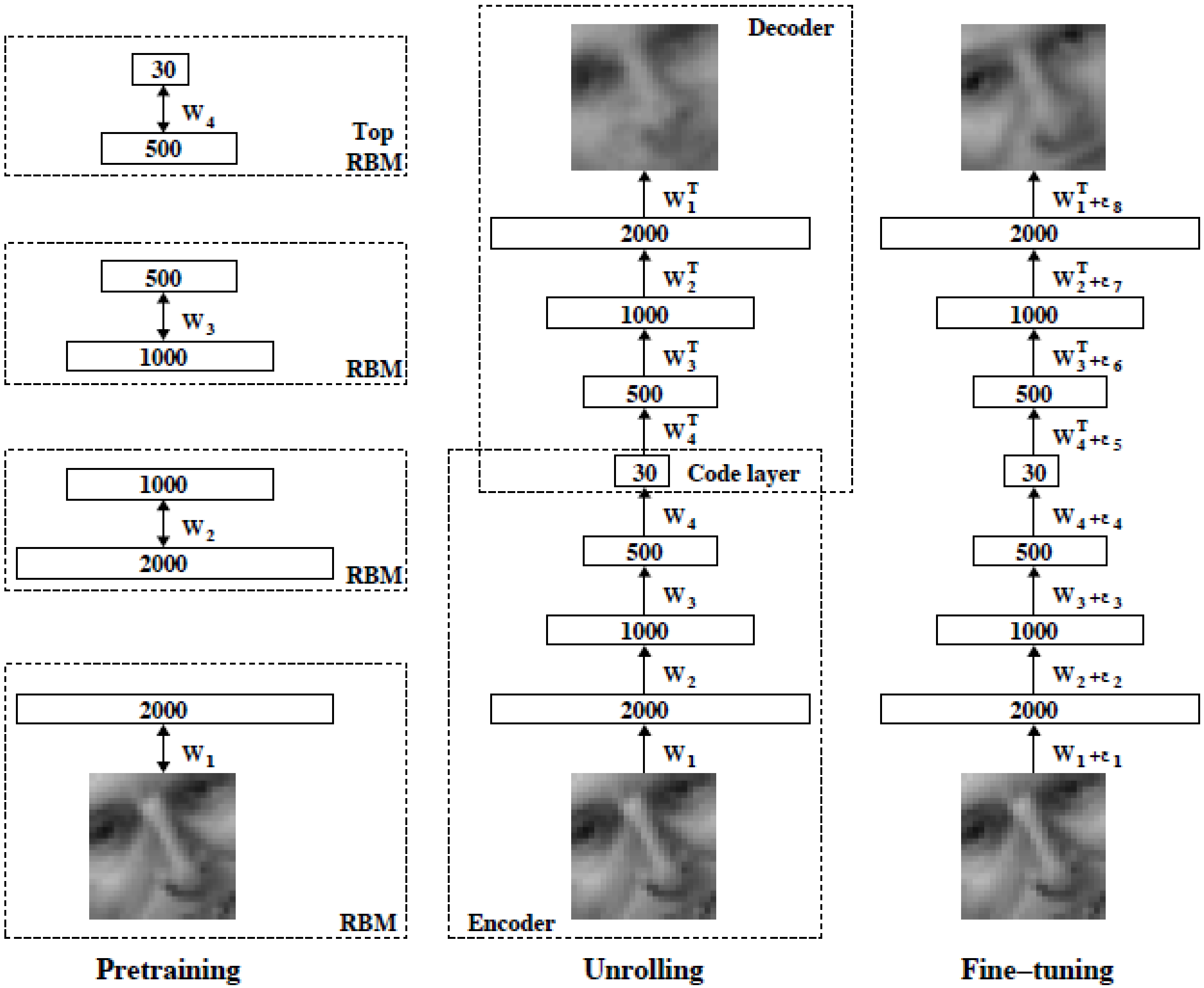
# Stacked Autoencoder

- Remove decoders and use feed-forward part.
- Standard, or convolutional neural network architecture.
- Param fine-tu backpr




Top-down vs. bottom up?
Is there a more rigorous mathematical formulation?

# Deep Autoencoder



# Deep Autoencoder

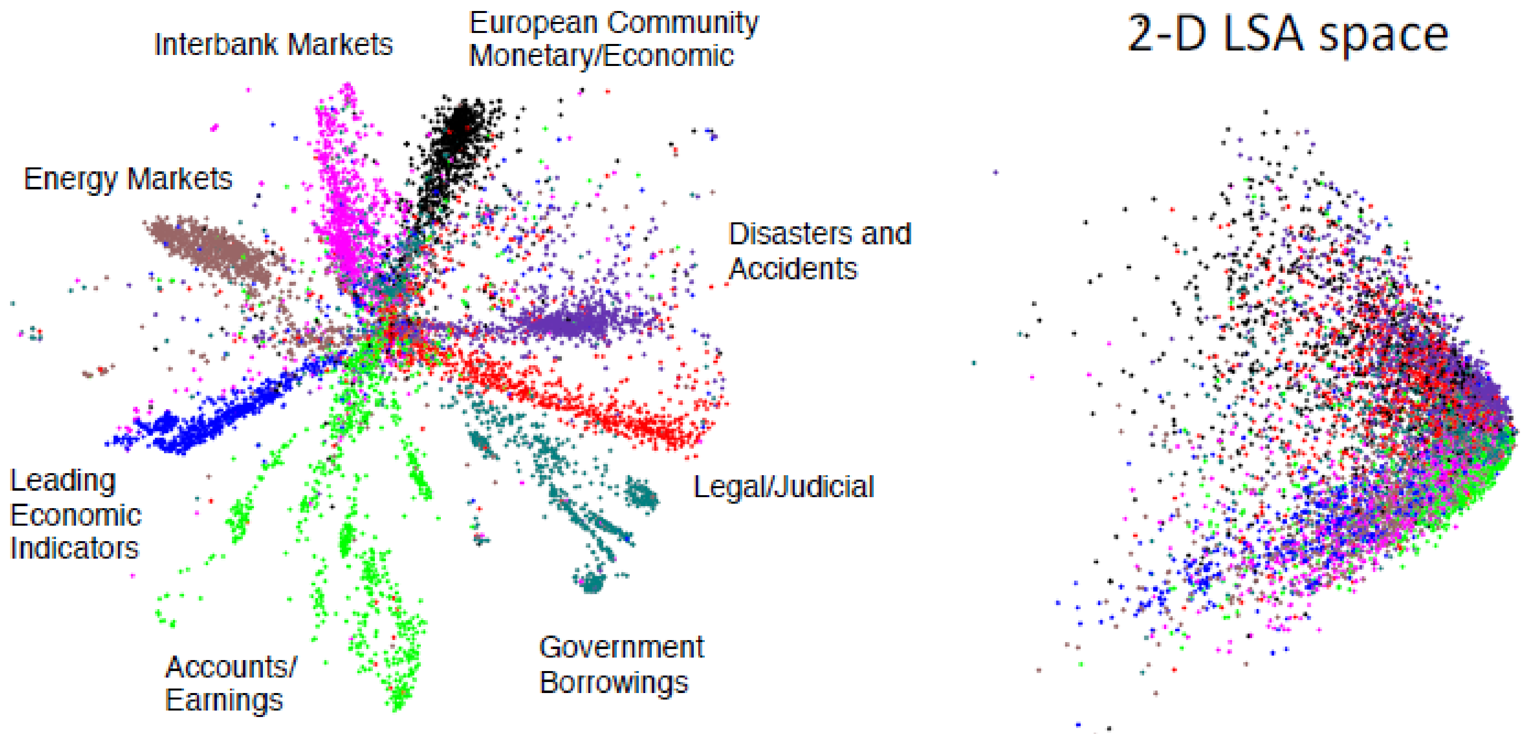
- We used 25x25 – 2000 – 1000 – 500 – 30 autoencoder to extract 30-D real-valued codes for Olivetti face patches.

- **Top**: Random samples from the test dataset.
- **Middle**: Reconstructions by the 30-dimensional deep autoencoder.
- **Bottom**: Reconstructions by the 30-dimensional PCA

# Deep Autoencoder

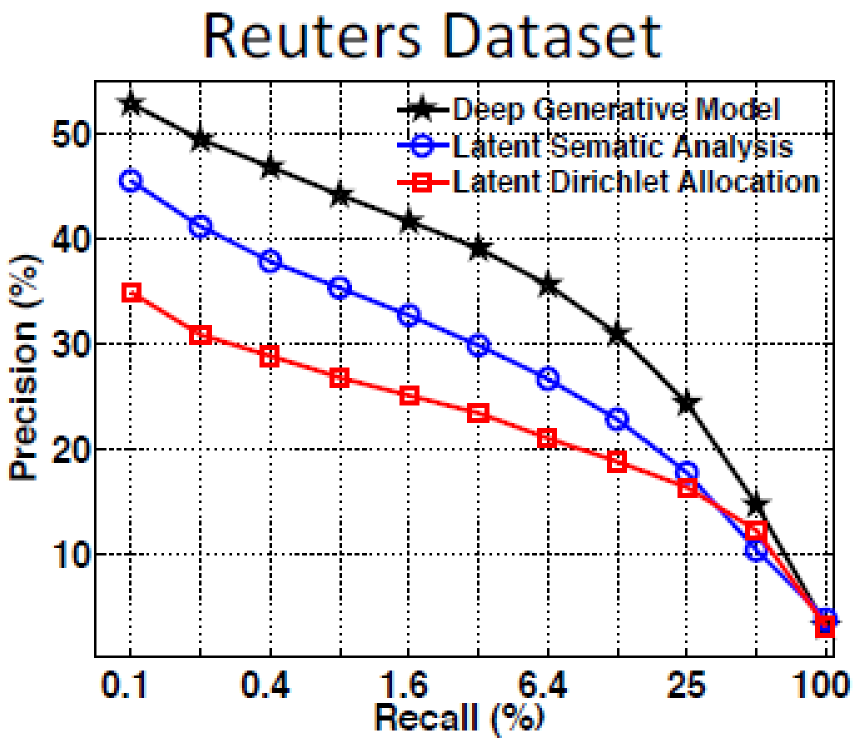
- Very difficult to optimize deep autoencoders using backpropagation
- Pre-training + fine-tuning
  - First train a stack of RBMs
  - Then “unroll” them
  - Then fine-tune with backpropagation

# Information Retrieval




- The Reuters Corpus Volume II contains 804,414 newswire stories (randomly split into **402,207 training and 402,207 test).**
- "Bag-of-words" representation: each article is represented as a vector containing the counts of the most frequently used 2000 words in the training set.

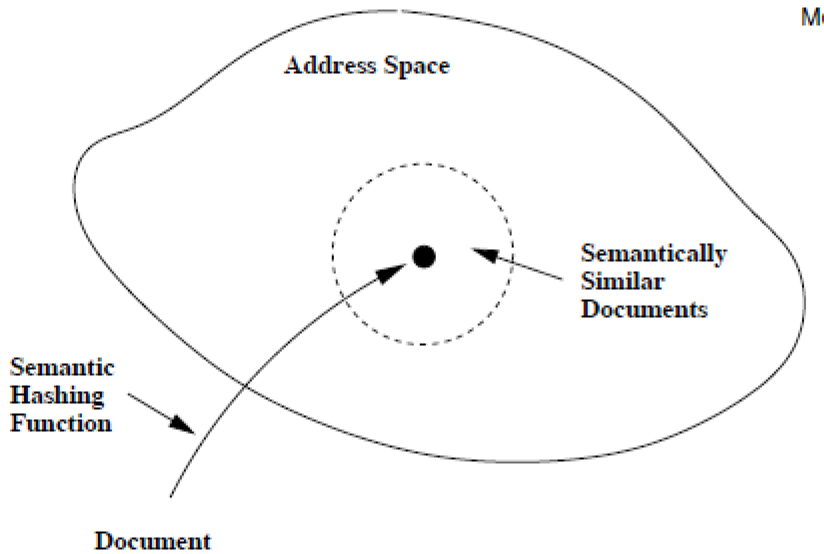
# Information Retrieval

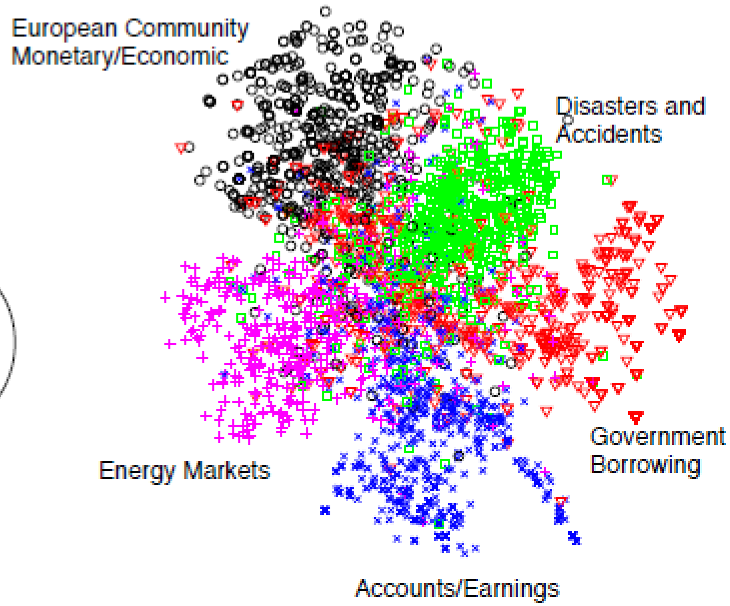



- Reuters dataset: 804,414 newswire stories.

- Deep generative model significantly outperforms LSA and LDA topic models

# Semantic Hashing

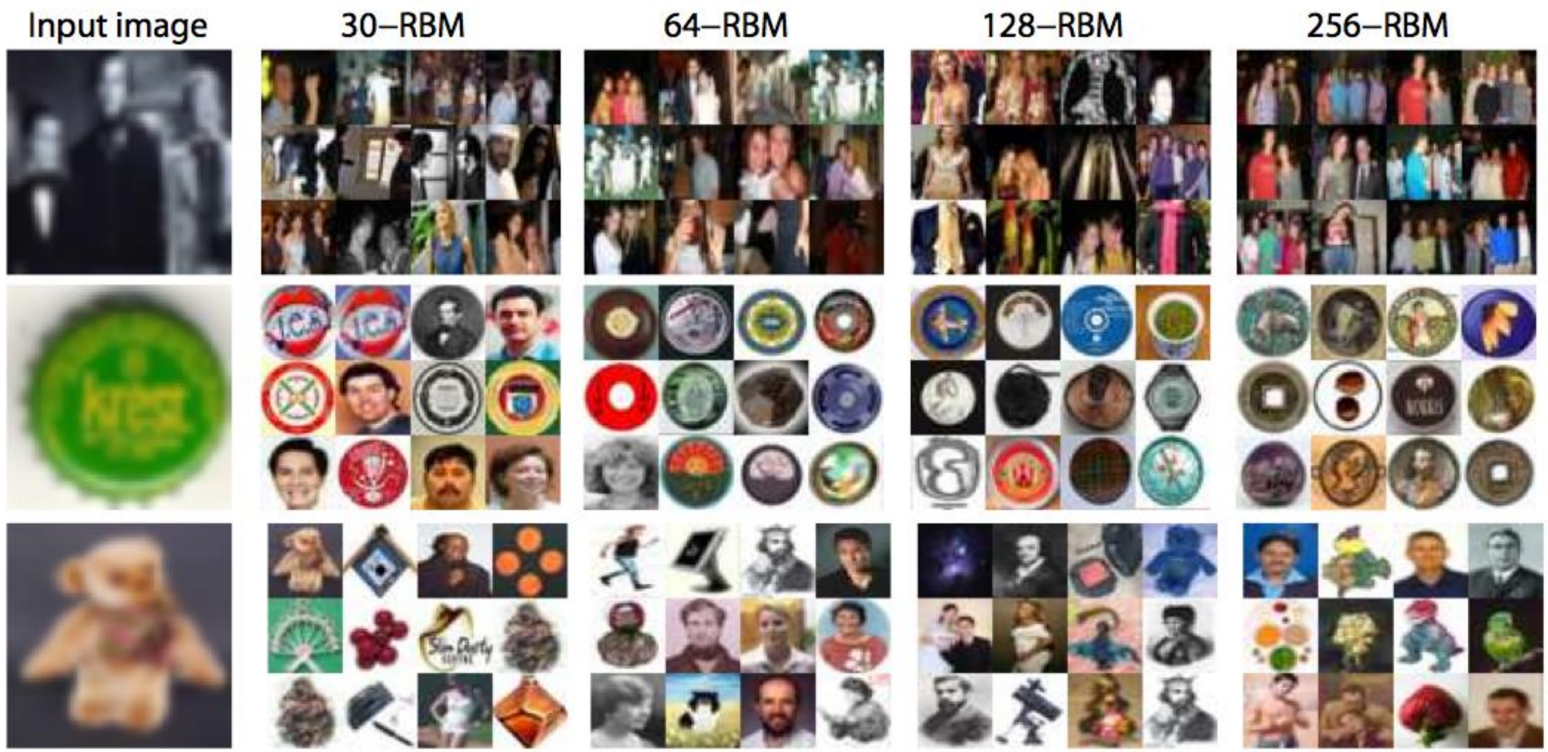



- Learn to map documents into **semantic 20-D binary codes.**
- Retrieve similar documents stored at the nearby addresses **with no search at all.**

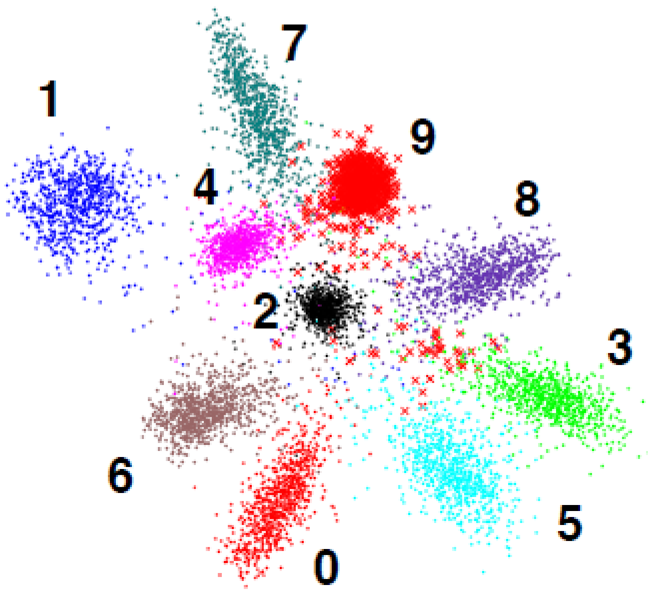
# Searching Image Database using Binary Codes

- Map images into binary codes for fast retrieval



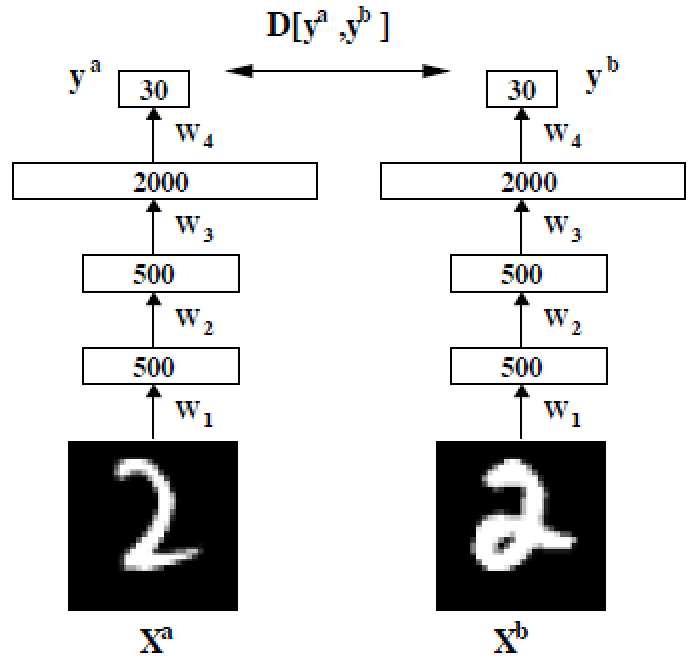

- Small Codes, Torralba, Fergus, Weiss, CVPR 2008
- Spectral Hashing, Y. Weiss, A. Torralba, R. Fergus, NIPS 2008
- Kulis and Darrell, NIPS 2009, Gong and Lazebnik, CVPR 20111
- Norouzi and Fleet, ICML 2011,

# Learning Similarity Measures



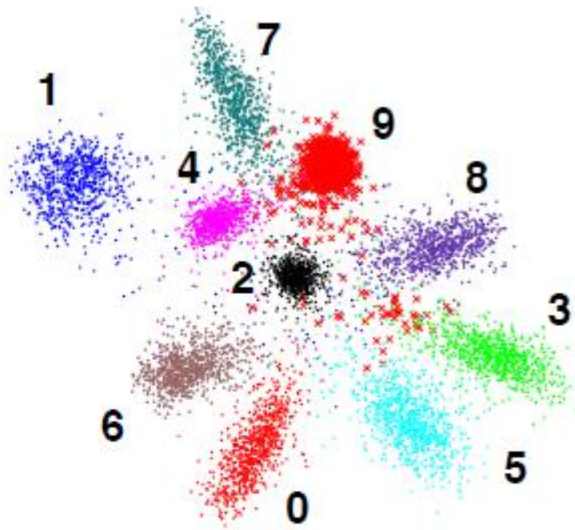

Related to Siamese Networks of LeCun.

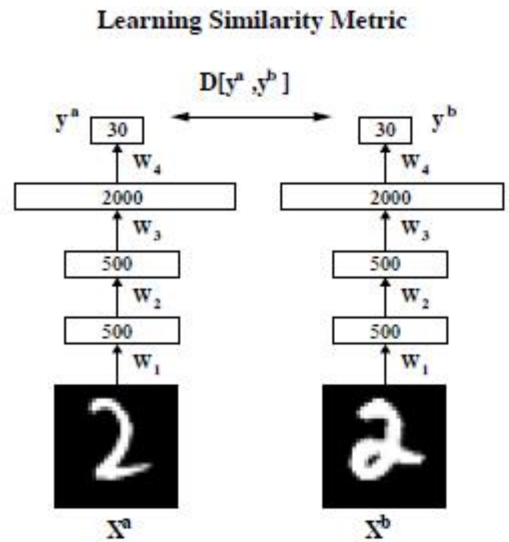
Maximize the Agreement



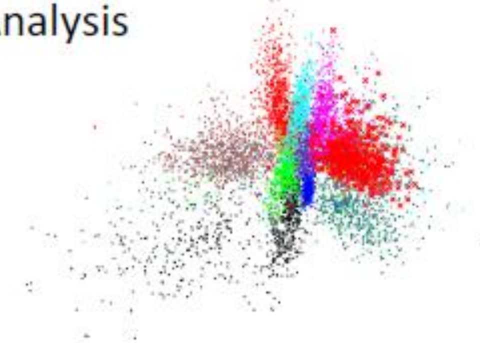

- Learn a nonlinear transformation of the input space.
- Optimize to make KNN perform well in the low-dimensional feature space

# Learning Similarity Measures

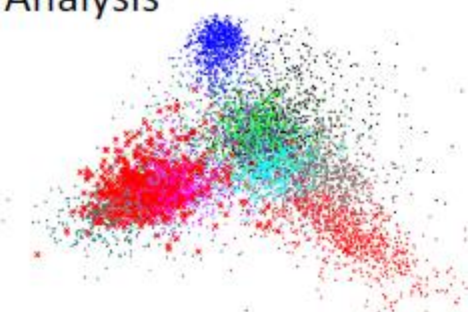



Neighborhood Component Analysis



Linear discriminant Analysis



PCA

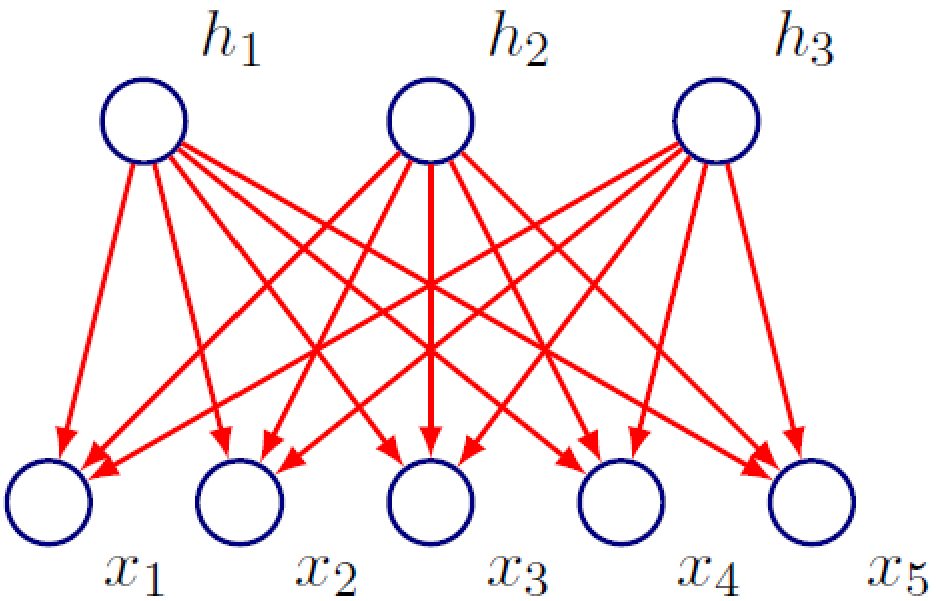
# Outline

1. Course Review
2. Linear Factor Model
3. Autoencoder
4. DBN and RBM

# More General Models

- Suppose $P$(h) can not be assumed to have a nice Gaussian form
- The decoding of the input from the latent states can be a complicated non-linear function
- Estimation and inference can get complicated!

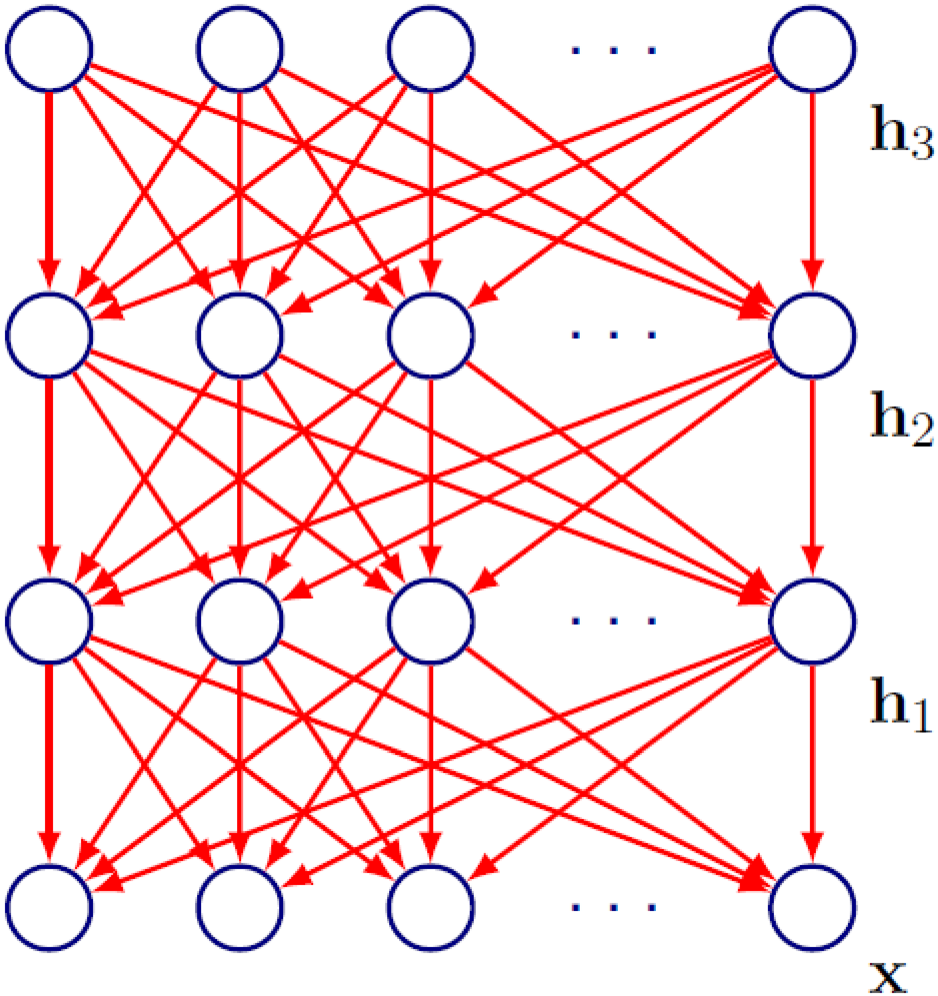
# Earlier we had:



# Quick Review

- Generative models can be modeled as directed graphical models
- The nodes represent random variables and arcs indicate dependency
- Some of the random variables are observed, others are hidden

# Sigmoid Belief Networks




- Just like a feedfoward network, but with arrows reversed.

# Sigmoid Belief Networks

- Let x = $h^0$. Consider binary activations, then:

$$P(\mathbf{h}_i^k = 1|\mathbf{h}^{k+1}) = sigm(b_i^k + \sum_j W_{i,j}^{k+1}\mathbf{h}_j^{k+1})$$

- The joint probability factorizes as:

$$P(\mathbf{x}, \mathbf{h}^1, \ldots, \mathbf{h}^l) = P(\mathbf{h}^l)\left(\prod_{k=1}^{l-1} P(\mathbf{h}^k|\mathbf{h}^{k+1})\right)P(\mathbf{x}|\mathbf{h}^1)$$

- Marginalization yields *P*(x), intractable in practice except for very small models

# Sigmoid Belief Networks

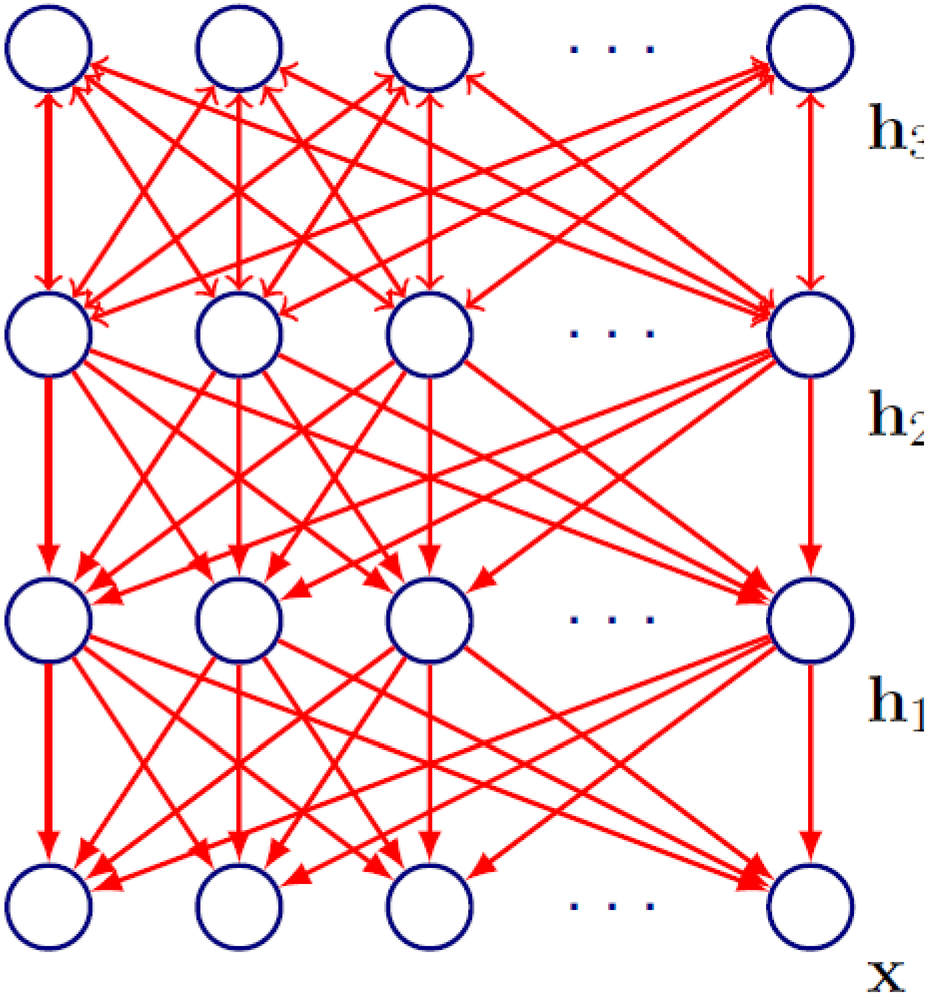
$$P(\mathbf{x}, \mathbf{h}^1, \ldots, \mathbf{h}^l) = P(\mathbf{h}^l)\Big(\prod_{k=1}^{l-1} P(\mathbf{h}^k|\mathbf{h}^{k+1})\Big)P(\mathbf{x}|\mathbf{h}^1)$$

- The top level prior is chosen as factorizable:
  $P(\mathbf{h}^l) = \prod_i P(\mathbf{h}_i^l)$
- A single (Bernoulli) parameter is needed for each $h_i$ in case of binary units
- Deep Belief Networks are like Sigmoid Belief Networks except for the top two layers

# Sigmoid Belief Networks

- General case models are called Helmholtz Machines
- Two key references:
  - G. E. Hinton, P. Dayan, B. J. Frey, R. M. Neal: The Wake-Sleep Algorithm for Unsupervised Neural Networks, In Science, 1995
  - R. M. Neal: Connectionist Learning of Belief Networks, In Artificial Intelligence, 1992

# Deep Belief Networks



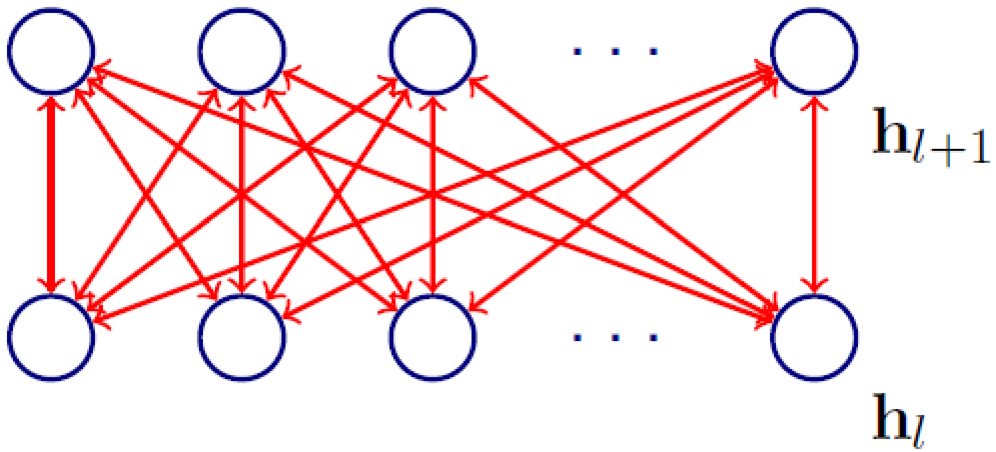

- The top two layers now have undirected edges

# Deep Belief Networks

- The joint probability changes as:

$$P(\mathbf{x}, \mathbf{h}^1, \ldots, \mathbf{h}^l) = P(\mathbf{h}^l, \mathbf{h}^{l-1}) \left( \prod_{k=1}^{l-2} P(\mathbf{h}^k | \mathbf{h}^{k+1}) \right) P(\mathbf{x} | \mathbf{h}^1)$$

# Deep Belief Networks




- The top two layers are a Restricted Boltzmann Machine
- A RBM has the joint distribution:

$$P(\mathbf{h}^{l+1}, \mathbf{h}^{l}) \propto \exp(\mathbf{b}'\mathbf{h}^{l-1} + \mathbf{c}'\mathbf{h}^{l} + \mathbf{h}^{l} W \mathbf{h}^{l-1})$$

- We will return to RBMs and training procedures in a while, but rst we look at the mathematical machinery that will make our task easier

# Greedy Layer-wise Training of DBNs

- Reference: G. E. Hinton, S. Osindero and Y-W Teh: A Fast Learning Algorithm for Deep Belief Networks, In Neural Computation, 2006.
- First Step: Construct a RBM with input **x** and a hidden layer **h**, train the RBM
- Stack another layer on top of the RBM to form a new RBM. Fix $W^1$, sample from $P(h^1|x)$, train $W^2$ as RBM
- Continue till *k* layers
- Implicitly defines *P*(x) and P(h) (variational bound justifies layerwise training)
- Can then be discriminatively fine-tuned using backpropagation

# Energy Based Models

- Energy-Based Models assign a scalar energy with every configuration of variables under consideration
- Learning: Change the energy function so that its final shape has some desirable properties
- We can define a probability distribution through an energy:

$$P(\mathbf{x}) = \frac{\exp^{-(\mathsf{Energy}(\mathbf{x}))}}{Z}$$

- Energies are in the log-probability domain:

$$\mathsf{Energy}(\mathbf{x}) = \log \frac{1}{(ZP(\mathbf{x}))}$$

# Energy Based Models

$$P(\mathbf{x}) = \frac{\exp^{-(\text{Energy}(\mathbf{x}))}}{Z}$$

- Z is a normalizing factor called the Partition Function

$$Z = \sum_{\mathbf{x}} \exp(-\text{Energy}(\mathbf{x}))$$

- How do we specify the energy function?

# Product of Experts Formulation

- In this formulation, the energy function is:

$$\text{Energy}(\mathbf{x}) = \sum_i f_i(\mathbf{x})$$

- Therefore:

$$P(\mathbf{x}) = \frac{\exp^{-(\sum_i f_i(\mathbf{x}))}}{Z}$$

- We have the product of experts:

$$P(\mathbf{x}) \propto \prod_i P_i(\mathbf{x}) \propto \prod_i \exp^{(-f_i(\mathbf{x}))}$$

# Product of Experts Formulation

$$P(\mathbf{x}) \propto \prod_i P_i(\mathbf{x}) \propto \prod_i \exp^{(-f_i(\mathbf{x}))}$$

- Every expert $f_i$ can be seen as enforcing a constraint on x
- If $f_i$ is large => $P_i(x)$ is small i.e. the expert thinks x is implausible (constraint violated)
- If $f_i$ is small => $P_i(x)$ is large i.e. the expert thinks x is plausible (constraint satisfied)
- Contrast this with mixture models

# Latent Variables

- x is observed, let's say **h** are hidden factors that explain x
- The probability then becomes:

$$P(\mathbf{x}, \mathbf{h}) = \frac{\exp^{-(\text{Energy}(\mathbf{x},\mathbf{h}))}}{Z}$$

- We only care about the marginal:

$$P(\mathbf{x}) = \sum_{\mathbf{h}} \frac{\exp^{-(\text{Energy}(\mathbf{x},\mathbf{h}))}}{Z}$$

# Latent Variables

$$P(\mathbf{x}) = \sum_{\mathbf{h}} \frac{\exp^{-(\text{Energy}(\mathbf{x},\mathbf{h}))}}{Z}$$

- We introduce another term in analogy from statistical physics: free energy:

$$P(\mathbf{x}) = \frac{\exp^{-(\text{FreeEnergy}(\mathbf{x}))}}{Z}$$

- Free Energy is just a marginalization of energies in the log-domain:

$$\text{FreeEnergy}(\mathbf{x}) = -\log \sum_{\mathbf{h}} \exp^{-(\text{Energy}(\mathbf{x},\mathbf{h}))}$$

# Latent Variables

$$P(\mathbf{x}) = \frac{\exp^{-(\text{FreeEnergy}(\mathbf{x}))}}{Z}$$

- Likewise, the partition function:

$$Z = \sum_{\mathbf{x}} \exp^{-\text{FreeEnergy}(\mathbf{x})}$$

- We have an expression for *P*(x) (and hence for the data log-likelihood). Let us see how the gradient looks like

# Data Log-Likelihood Gradient

$$P(\mathbf{x}) = \frac{\exp^{-(\mathsf{FreeEnergy}(\mathbf{x}))}}{Z}$$

- The gradient is simply working from the above:

$$\frac{\partial \log P(\mathbf{x})}{\partial \theta} = -\frac{\partial \mathsf{FreeEnergy}(\mathbf{x})}{\partial \theta} + \frac{1}{Z} \sum_{\tilde{\mathbf{x}}} \exp^{-(\mathsf{FreeEnergy}(\tilde{\mathbf{x}}))} \frac{\partial \mathsf{FreeEnergy}(\tilde{\mathbf{x}})}{\partial \theta}$$

- Note that $P(\tilde{\mathbf{x}}) = \exp^{-(\mathsf{FreeEnergy}(\tilde{\mathbf{x}}))}$

# Data Log-Likelihood Gradient

- The expected log-likelihood gradient over the training set has the following form:

$$\mathbb{E}_{\tilde{P}}\left[\frac{\partial \log P(\mathbf{x})}{\partial \theta}\right] = \mathbb{E}_{\tilde{P}}\left[\frac{\partial \mathrm{FreeEnergy}(\mathbf{x})}{\partial \theta}\right] + \mathbb{E}_{P}\left[\frac{\partial \mathrm{FreeEnergy}(\mathbf{x})}{\partial \theta}\right]$$

# Data Log-Likelihood Gradient

$$\mathbb{E}_{\tilde{P}}\left[\frac{\partial \log P(\mathbf{x})}{\partial \theta}\right] = \mathbb{E}_{\tilde{P}}\left[\frac{\partial \mathrm{FreeEnergy}(\mathbf{x})}{\partial \theta}\right] + \mathbb{E}_{P}\left[\frac{\partial \mathrm{FreeEnergy}(\mathbf{x})}{\partial \theta}\right]$$

- $\tilde{P}$ is the empirical training distribution
- Easy to compute!

# Data Log-Likelihood Gradient

$$\mathbb{E}_{\tilde{P}}\left[\frac{\partial \log P(\mathbf{x})}{\partial \theta}\right] = \mathbb{E}_{\tilde{P}}\left[\frac{\partial \text{FreeEnergy}(\mathbf{x})}{\partial \theta}\right] + \mathbb{E}_{P}\left[\frac{\partial \text{FreeEnergy}(\mathbf{x})}{\partial \theta}\right]$$

- P is the model distribution (exponentially many configurations!)
- Usually very hard to compute!
- Resort to Markov Chain Monte Carlo to get a stochastic estimator of the gradient

# A Special Case

- Suppose the energy has the following form:

$$\text{Energy}(\mathbf{x}, \mathbf{h}) = -\beta(\mathbf{x}) + \sum_i \gamma_i(\mathbf{x}, \mathbf{h}_i)$$

- The free energy, and numerator of log likelihood can be computed tractably!
- What is P(x)?
- What is the FreeEnergy(x)?

# A Special Case

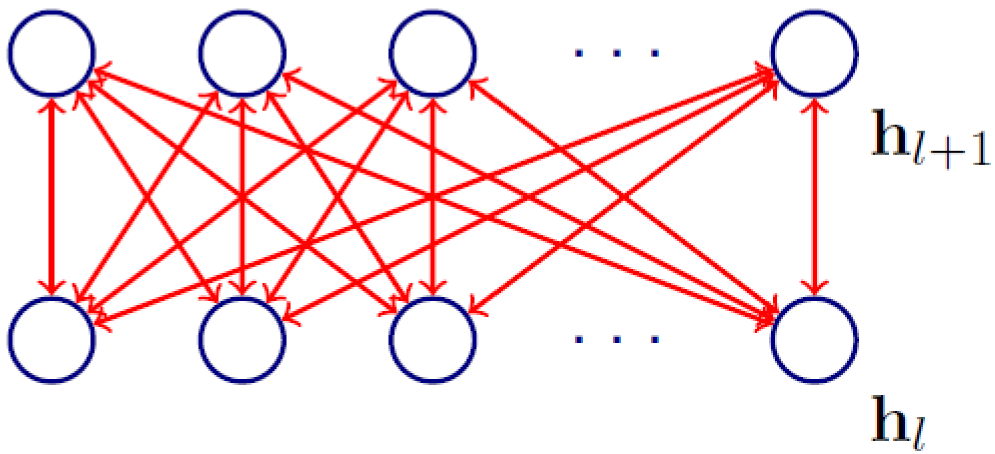
- The likelihood term:

$$P(\mathbf{x}) = \frac{\exp^{\beta(\mathbf{x})}}{Z} \prod_i \sum_{\mathbf{h}_i} \exp^{-\gamma_i(\mathbf{x},\mathbf{h}_i)}$$

- The Free Energy term:

$$\begin{aligned}\mathsf{FreeEnergy}(\mathbf{x}) &= -\log P(\mathbf{x}) - \log Z \\ &= -\beta - \sum_i \log \sum_{\mathbf{h}_i} \exp^{-\gamma_i(\mathbf{x},\mathbf{h}_i)}\end{aligned}$$

# Restricted Boltzmann Machines



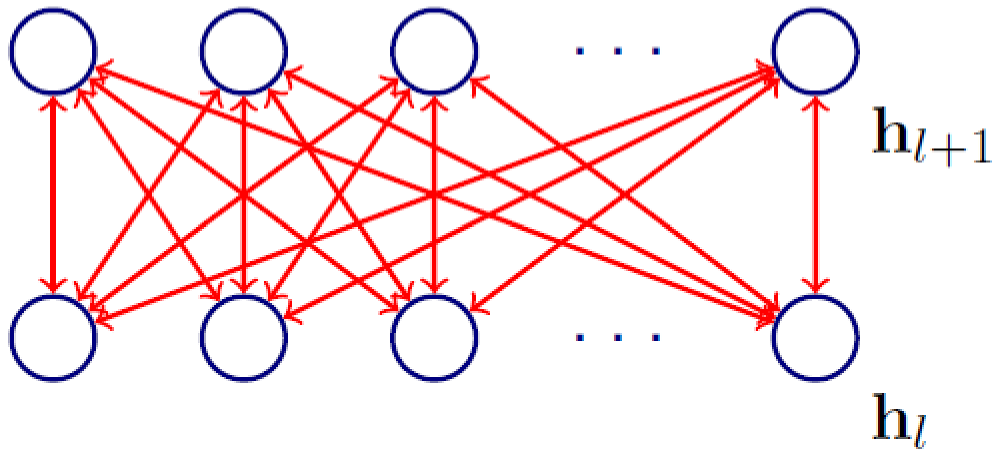

- Recall the form of energy:

$$\text{Energy}(\mathbf{x}, \mathbf{h}) = -\mathbf{b}^T\mathbf{x} - \mathbf{c}^T\mathbf{h} - \mathbf{h}^T W \mathbf{x}$$

- Takes the earlier nice form with (x) = $b^T$ x and
$$\gamma_i(\mathbf{x}, \mathbf{h_i}) = \mathbf{h}_i(\mathbf{c}_i + W_i\mathbf{x})$$

- Originally proposed by Smolensky (1987) who called them Harmoniums as a special case of Boltzmann Machines

# Restricted Boltzmann Machines



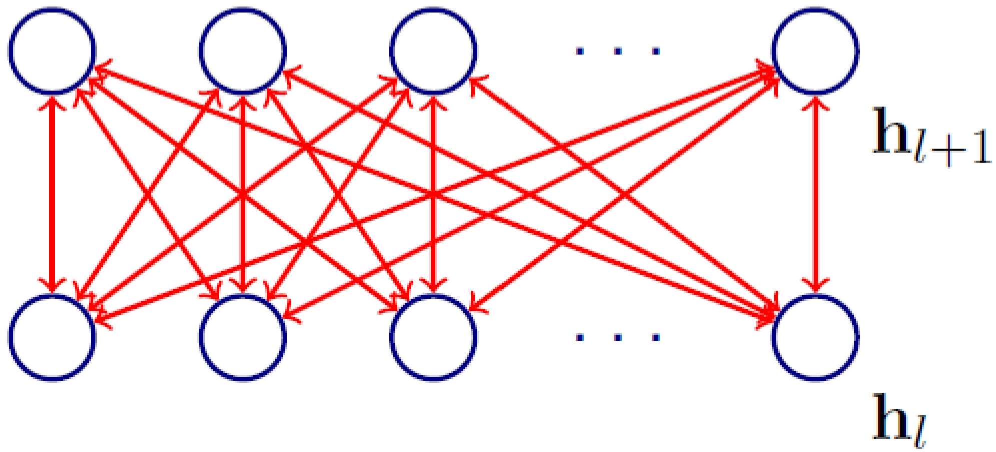

- As seen before, the Free Energy can be computed efficiently:

$$\text{FreeEnergy}(\mathbf{x}) = -\mathbf{b}^T\mathbf{x} - \sum_i \log \sum_{\mathbf{h}_i} \exp^{\mathbf{h}_i(\mathbf{c}_i + W_i\mathbf{x})}$$

- The conditional probability:

$$P(\mathbf{h}|\mathbf{x}) = \frac{\exp\left(\mathbf{b}^T\mathbf{x} + \mathbf{c}^T\mathbf{h} + \mathbf{h}^T W\mathbf{x}\right)}{\sum_{\tilde{\mathbf{h}}} \exp\left(\mathbf{b}^T\mathbf{x} + \mathbf{c}^T\tilde{\mathbf{h}} + \tilde{\mathbf{h}}^T W\mathbf{x}\right)} = \prod_i P(\mathbf{h}_i|\mathbf{x})$$

# Restricted Boltzmann Machines




- x and h play symmetric roles:

$$P(\mathbf{x}|\mathbf{h}) = \prod_i P(\mathbf{x}_i|\mathbf{h})$$

- The common transfer (for the binary case):

$$P(\mathbf{h}_i = 1|\mathbf{x}) = \sigma(\mathbf{c_i} + W_i \mathbf{x})$$

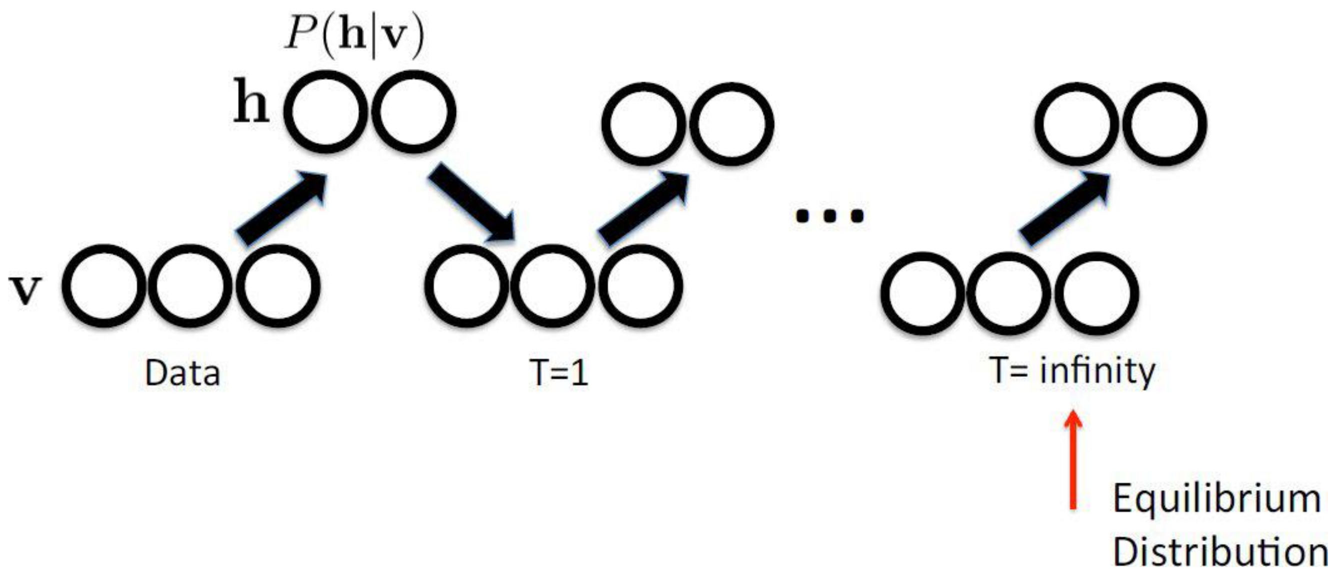
$$P(\mathbf{x}_j = 1|\mathbf{h}) = \sigma(\mathbf{b_j} + W_{:,j}^T \mathbf{h})$$

# Approximate Learning and Gibbs Sampling

$$\mathbb{E}_{\tilde{P}}\left[\frac{\partial \log P(\mathbf{x})}{\partial\theta}\right] = \mathbb{E}_{\tilde{P}}\left[\frac{\partial \text{FreeEnergy}(\mathbf{x})}{\partial\theta}\right] + \mathbb{E}_{P}\left[\frac{\partial \text{FreeEnergy}(\mathbf{x})}{\partial\theta}\right]$$

- We saw the expression for Free Energy for a RBM. But the second term was intractable. How do learn in this case?
- Replace the average over all possible input configurations by samples
- Run Markov Chain Monte Carlo (Gibbs Sampling):
- First sample $x_1 \sim P(x)$, then $h_1 \sim P(h|x_1)$, then $X_2 \sim P(x|h_1)$, then $h_2 \sim P(h|x_2)$ till $x_{k+1}$

# Approximate Learning, Alternating Gibbs Sampling




- We have already seen: $P(\mathbf{x}|\mathbf{h}) = \prod_i P(\mathbf{x}_i|\mathbf{h})$

$$P(\mathbf{h}|\mathbf{x}) = \prod_i P(\mathbf{h}_i|\mathbf{x})$$

- With: $P(\mathbf{h}_i = 1|\mathbf{x}) = \sigma(\mathbf{c_i} + W_i\mathbf{x})$ and

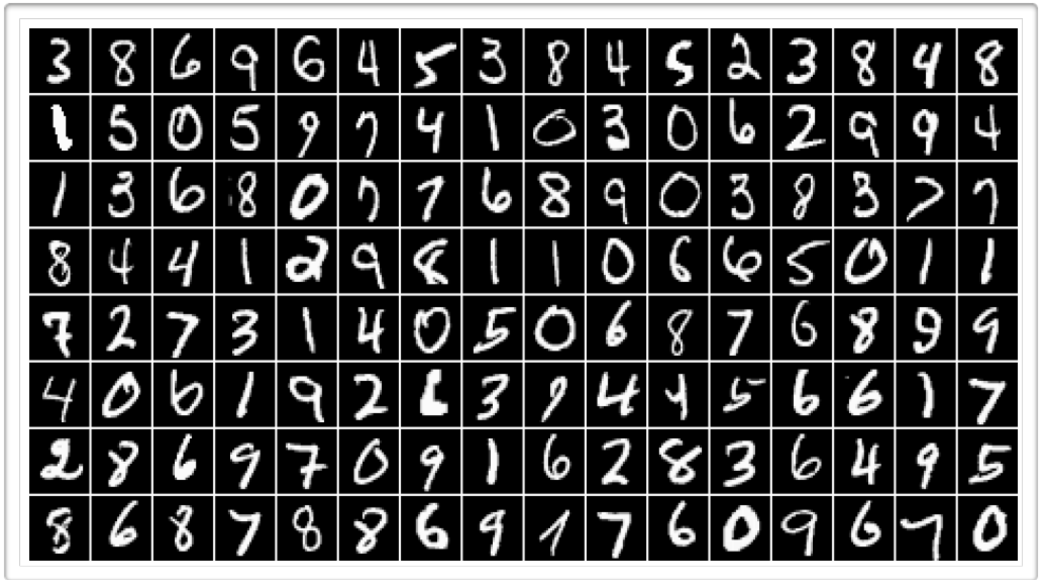
$$P(\mathbf{x}_j = 1|\mathbf{h}) = \sigma(\mathbf{b_j} + W_{:,j}^T\mathbf{h})$$

# Training RBM: Contrastive Divergence Algorithm

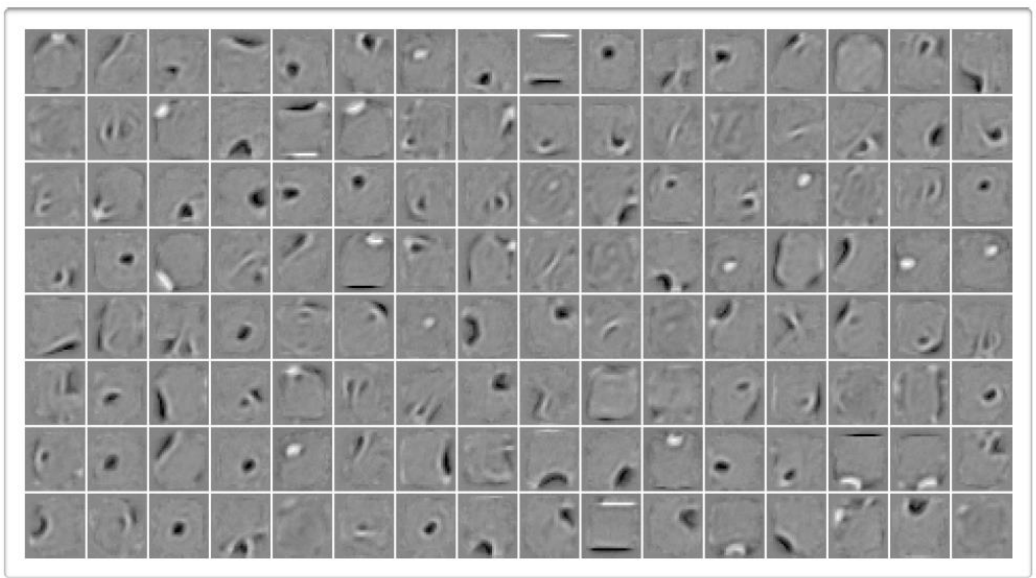
- Start with a training example on the visible units
- Update all the hidden units in parallel
- Update all the visible units in parallel to obtain a reconstruction
- Update all the hidden units again
- Update model parameters
- Aside: Easy to extend RBM (and contrastive divergence) to the continuous case

# Example: MNIST

Original images:



Learned features:



(Larochelle et al., JMLR 2009)

# Acknowledgement

Some of the materials in these slides are drawn inspiration from:

- Shubhendu Trivedi and Risi Kondor, University of Chicago, Deep Learning Course
- Hung-yi Lee, National Taiwan University, Machine Learning and having it Deep and Structured course
- Xiaogang Wang, The Chinese University of Hong Kong, Deep Learning Course
- Fei-Fei Li, Standord University, CS231n Convolutional Neural Networks for Visual Recognition course

# Next time

- Model Computation and Inference

# Questions?

## Thank You !