10707
Deep Learning: Spring 2021

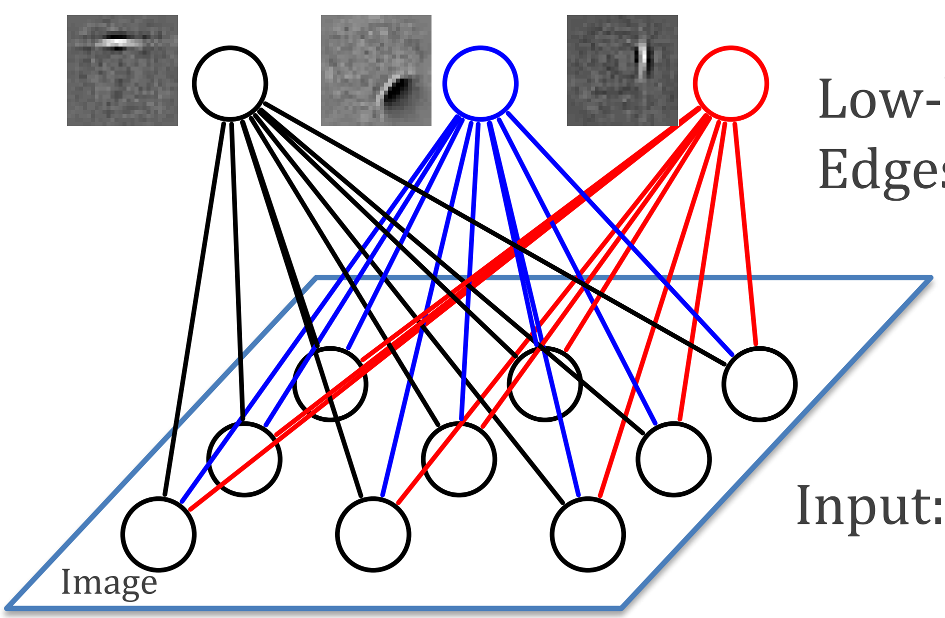
Andrej Risteski

Machine Learning Department

Lecture 13:
More applications of variational methods: DBNs, VQ-VAEs, NVAEs

Part I: Learning Deep Belief Networks (DBNs)

Deep Belief Network



Low-level features:
Edges

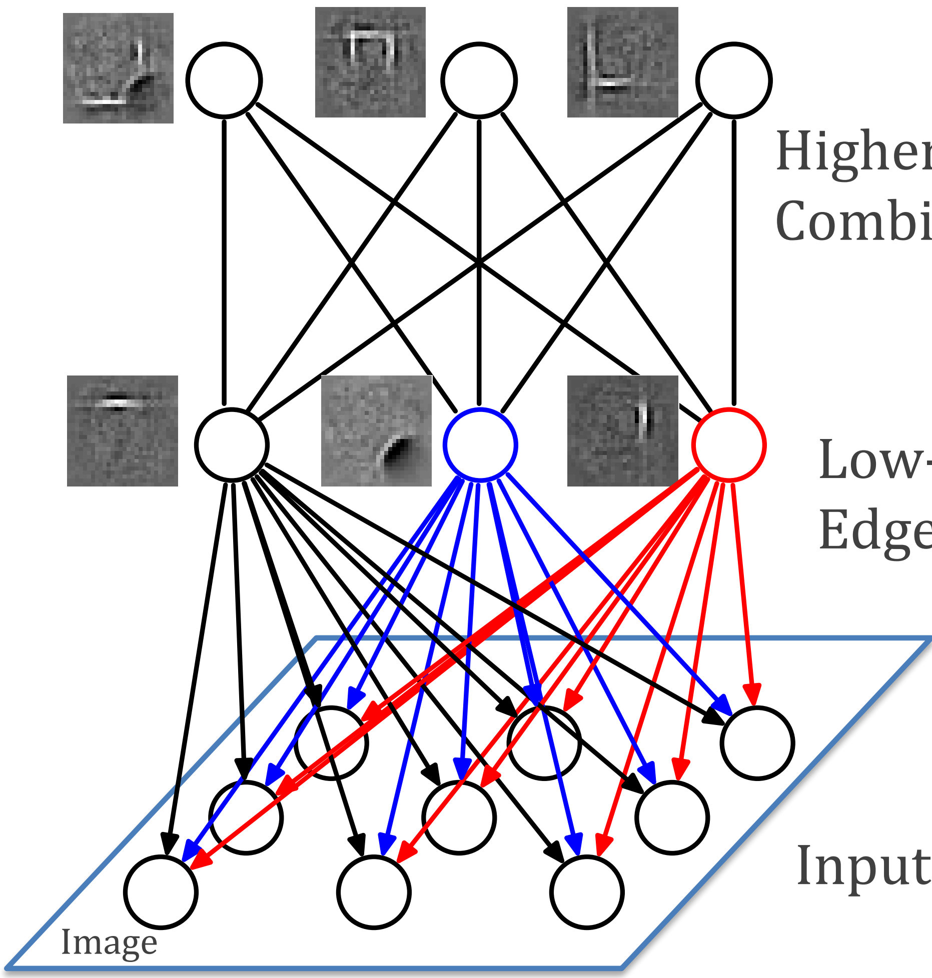
Built from **unlabeled** inputs.

Input: Pixels

(Hinton et.al. Neural Computation 2006)

Deep Belief Network

Internal representations capture higher-order statistical structure



Higher-level features:
Combination of edges

Low-level features:
Edges

Built from **unlabeled** inputs.

Input: Pixels

(Hinton et.al. Neural Computation 2006)

Deep Belief Network

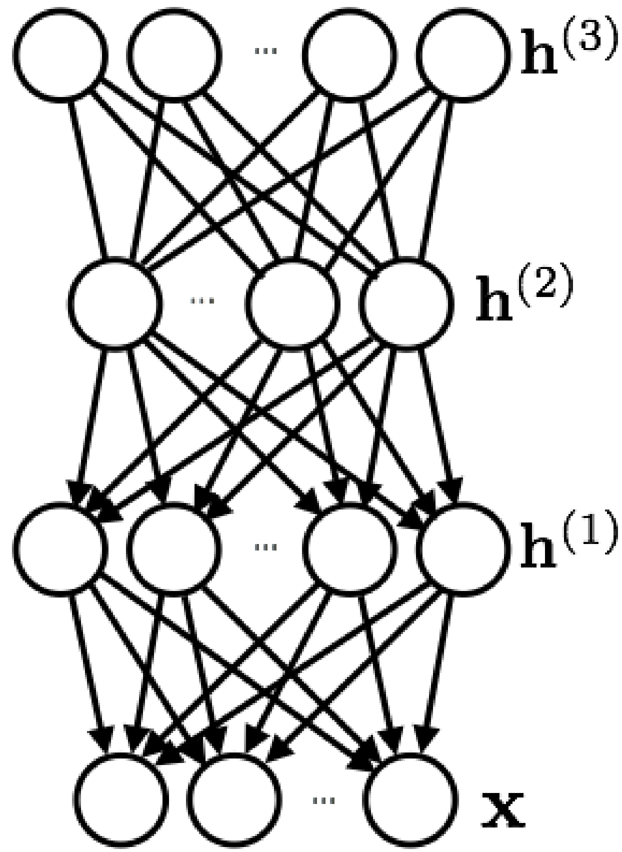
- it is a generative model that mixes undirected and directed connections between variables

- top 2 layers' distribution $p(\mathbf{h}^{(2)}, \mathbf{h}^{(3)})$ is an RBM!

- other layers form a Bayesian network with conditional distributions:

$$p(h_j^{(1)} = 1|\mathbf{h}^{(2)}) = \mathrm{sigm}(\mathbf{b}^{(1)} + \mathbf{W}^{(2)^\top}\mathbf{h}^{(2)})$$

$$p(x_i = 1|\mathbf{h}^{(1)}) = \mathrm{sigm}(\mathbf{b}^{(0)} + \mathbf{W}^{(1)^\top}\mathbf{h}^{(1)})$$



Deep Belief Network

The joint distribution of a DBN is as follows

$$p(\mathbf{x}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{h}^{(3)}) = p(\mathbf{h}^{(2)}, \mathbf{h}^{(3)})\, p(\mathbf{h}^{(1)}|\mathbf{h}^{(2)})\, p(\mathbf{x}|\mathbf{h}^{(1)})$$

where

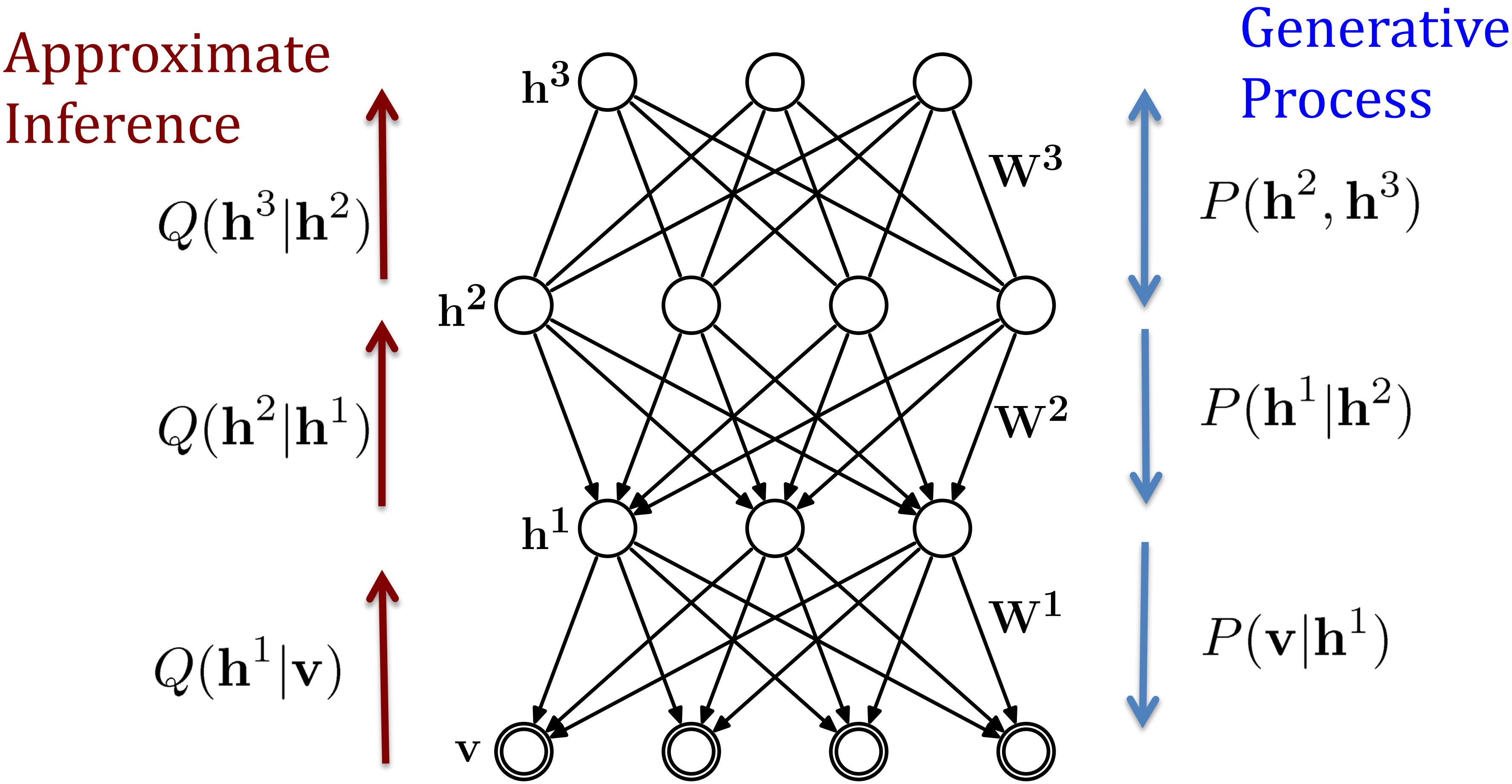
$$p(\mathbf{h}^{(2)}, \mathbf{h}^{(3)}) = \exp\left(\mathbf{h}^{(2)\top}\mathbf{W}^{(3)}\mathbf{h}^{(3)} + \mathbf{b}^{(2)\top}\mathbf{h}^{(2)} + \mathbf{b}^{(3)\top}\mathbf{h}^{(3)}\right)/Z$$

$$p(\mathbf{h}^{(1)}|\mathbf{h}^{(2)}) = \prod_j p(h_j^{(1)}|\mathbf{h}^{(2)})$$

$$p(\mathbf{x}|\mathbf{h}^{(1)}) = \prod_i p(x_i|\mathbf{h}^{(1)})$$

(I realize this looks odd.)

DBN Layer-wise Training

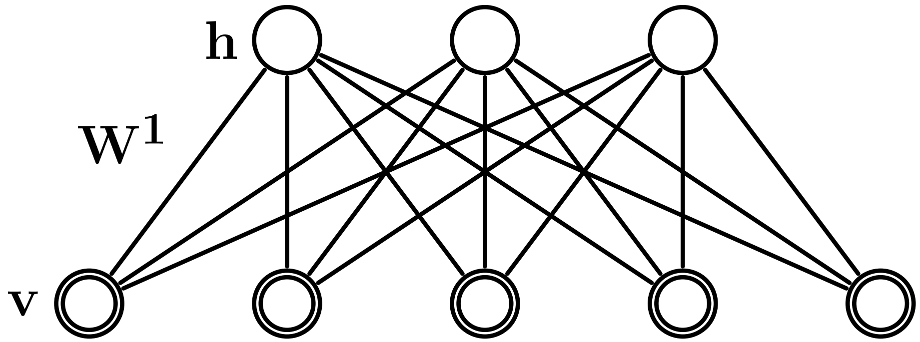


$Q(h^t|h^{t-1}), P(h^{t-1}|h^t)$ are product distributions, s.t.:

$$Q\left((h^t)_j = 1 \middle| h^{t-1}\right) = \frac{1}{1 + \exp(W_{t,.}h^{t-1})} \quad P\left((h^{t-1})_j = 1 \middle| h^t\right) = \frac{1}{1 + \exp((h^t)^T W_{.,t})}$$

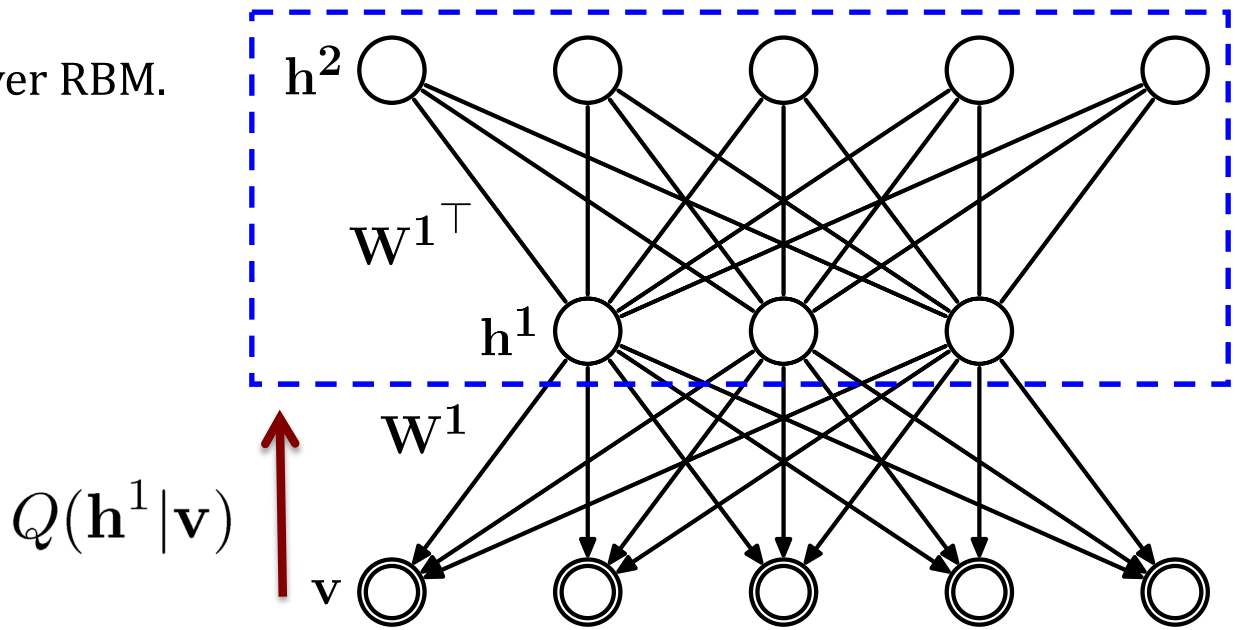
DBN Layer-wise Training

- Learn an RBM with an input layer v=x and a hidden layer h.



DBN Layer-wise Training

- Learn an RBM with an input layer v=x and a hidden layer h.
- Treat inferred values $Q(\mathbf{h}^1|\mathbf{v}) = P(\mathbf{h}^1|\mathbf{v})$ as the data for training 2nd-layer RBM.
- Learn and freeze 2nd layer RBM.

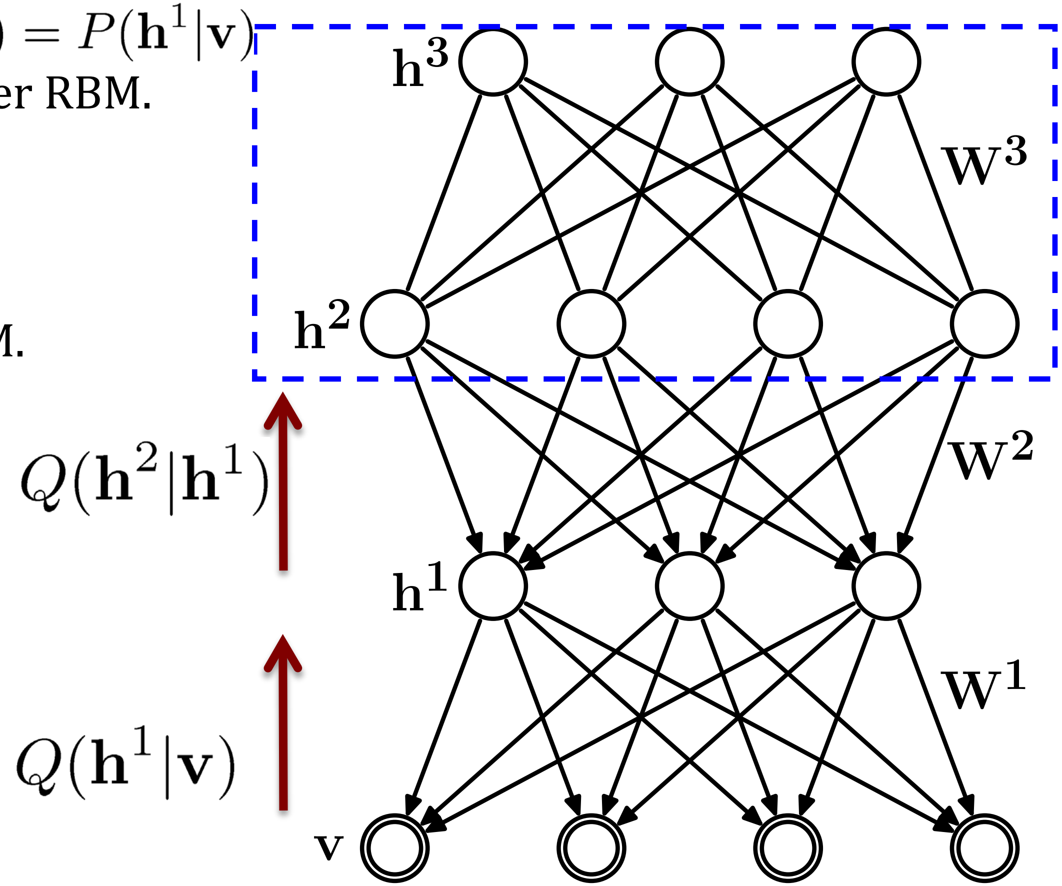


DBN Layer-wise Training

- Learn an RBM with an input layer v=x and a hidden layer h.

Unsupervised Feature Learning.

- Treat inferred values $Q(\mathbf{h}^1|\mathbf{v}) = P(\mathbf{h}^1|\mathbf{v})$ as the data for training 2nd-layer RBM.
- Learn and freeze 2nd layer RBM.
- Proceed to the next layer.

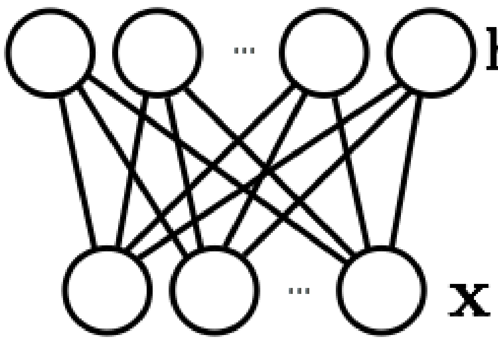


Where does this training come from??

Variational intuitions

Let's write the marginal p(x) in terms of the Gibbs variational principle.

Recall, we have:



For every distribution $q(\mathbf{h}^{(1)}|\mathbf{x})$:

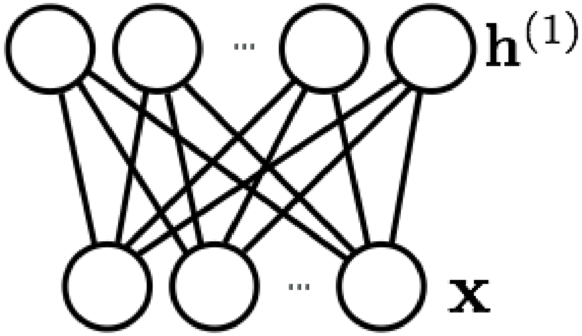
$$\log p(\mathbf{x}) \geq \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log p(\mathbf{x}, \mathbf{h}^{(1)}) - \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log q(\mathbf{h}^{(1)}|\mathbf{x})$$

Equality is attained if $q(\mathbf{h}^{(1)}|\mathbf{x}) = p(\mathbf{h}^{(1)}|\mathbf{x})$.

Variational intuitions

Let's write the marginal p(x) in terms of the Gibbs variational principle.

Recall, we have:



For every distribution $q(\mathbf{h}^{(1)}|\mathbf{x})$:

$$\log p(\mathbf{x}) \geq \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log p(\mathbf{x}, \mathbf{h}^{(1)}) - \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log q(\mathbf{h}^{(1)}|\mathbf{x})$$

The idea will be to add layers, s.t. we improve the ***variational bound (i.e. the right-hand side)***

Variational intuitions

adding 2nd layer means
untying the parameters

$$\log p(\mathbf{x}) \geq \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \left(\log p(\mathbf{x}|\mathbf{h}^{(1)}) + \log p(\mathbf{h}^{(1)}) \right) - \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log q(\mathbf{h}^{(1)}|\mathbf{x})$$

- When adding a second layer, we model $p(\mathbf{h}^{(1)})$ using a separate set of parameters

  - they are the parameters of the RBM involving $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$
  - $p(\mathbf{h}^{(1)})$ is now the marginalization of the second hidden layer

$$p(\mathbf{h}^{(1)}) = \sum_{\mathbf{h}^{(2)}} p(\mathbf{h}^{(1)}, \mathbf{h}^{(2)})$$

Variational intuitions

adding 2nd layer means
untying the parameters

$$\log p(\mathbf{x}) \geq \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \left(\log p(\mathbf{x}|\mathbf{h}^{(1)}) + \log p(\mathbf{h}^{(1)}) \right)$$

$$- \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log q(\mathbf{h}^{(1)}|\mathbf{x})$$

- we can train the parameters of … the bound. This is equivalent to m… . terms are constant:

Layerwise training improves variational lower bound

$$- \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log p(\mathbf{h}^{(1)})$$

- this is like training an RBM on data generated from $q(\mathbf{h}^{(1)}|\mathbf{x})$!

Stacking the layers

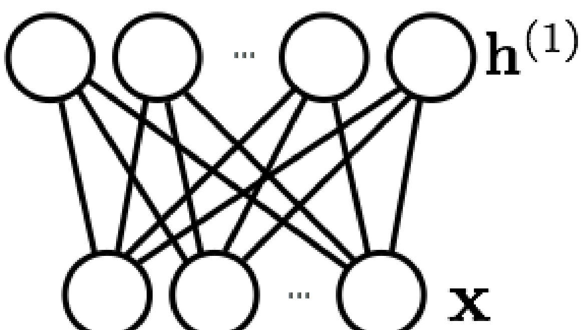
This is where the RBM stacking procedure comes from:

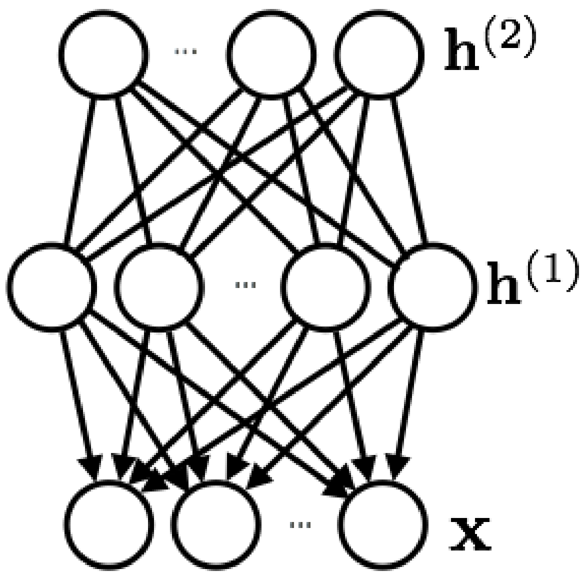
- **idea**: improve prior on last layer by adding another hidden layer

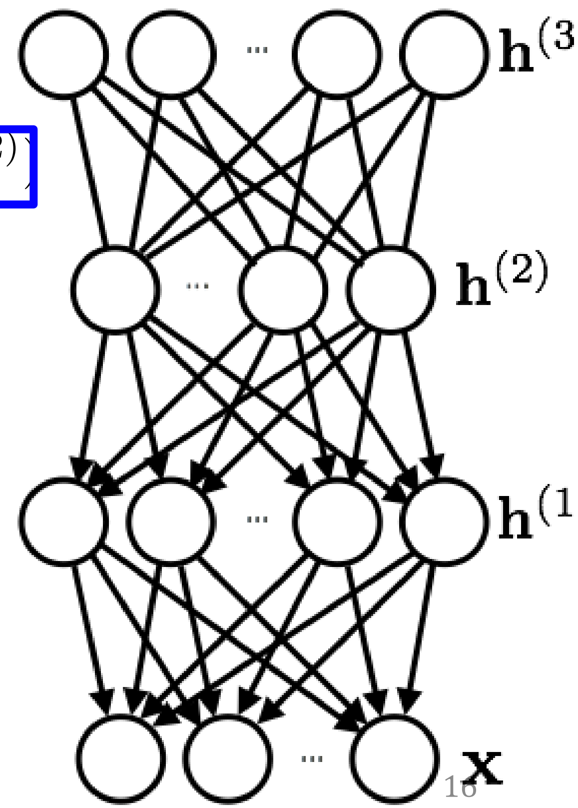
$$p(\mathbf{h}^{(1)}, \mathbf{h}^{(2)}) = p(\mathbf{h}^{(1)}|\mathbf{h}^{(2)}) \sum_{\mathbf{h}^{(3)}} p(\mathbf{h}^{(2)}, \mathbf{h}^{(3)})$$

$$p(\mathbf{x}, \mathbf{h}^{(1)}) = p(\mathbf{x}|\mathbf{h}^{(1)}) \sum_{\mathbf{h}^{(2)}} \boxed{p(\mathbf{h}^{(1)}, \mathbf{h}^{(2)})}$$

$$p(\mathbf{x}) = \sum_{\mathbf{h}^{(1)}} \boxed{p(\mathbf{x}, \mathbf{h}^{(1)})}$$

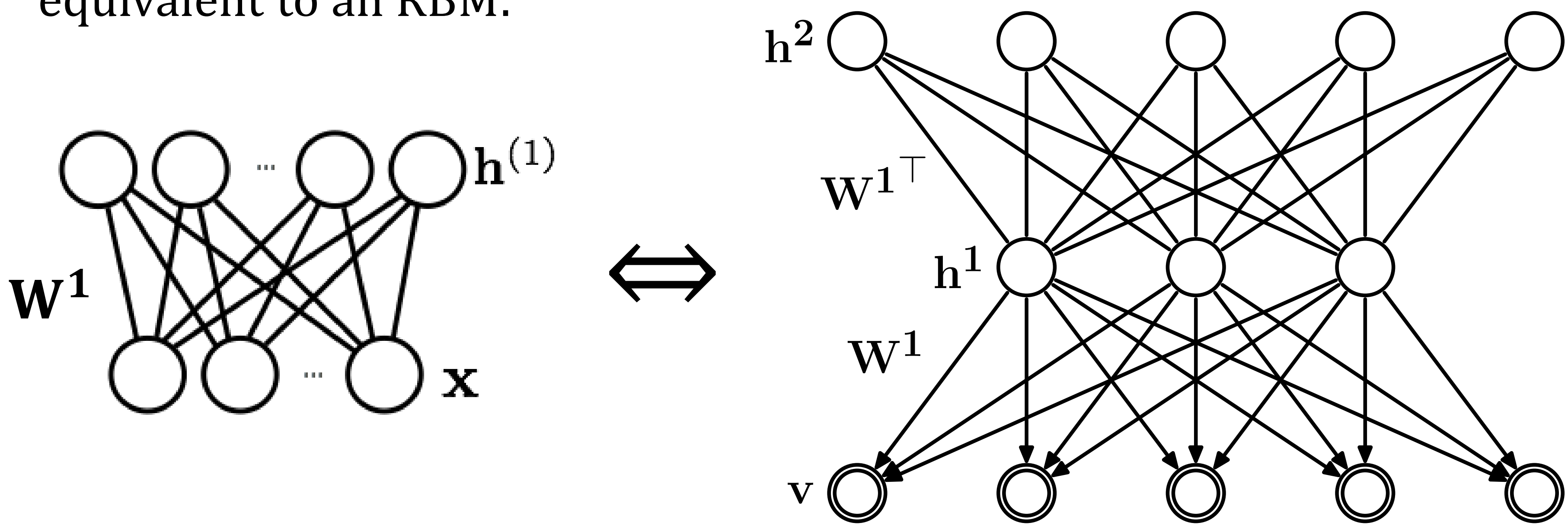






Improvement at initialization: weight-tied DBN is equivalent to a RBM

Observation: a two-layer DBN with appropriately tied weights is equivalent to an RBM:



Formal proof is a little annoying. Intuition:

- Gibbs sampling converges to model distribution in first case.
- Gibbs sampling on top two layers, plus one last sample of x given $\mathrm{h}^{(1)}$ converges to model distribution in second.
- The steps in these two random walks are *exactly* the same.

Improvement at initialization: weight-tied DBN is equivalent to a RBM

adding 2nd layer means untying the parameters

$$\log p(\mathbf{x}) \geq \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \left(\log p(\mathbf{x}|\mathbf{h}^{(1)}) + \log p(\mathbf{h}^{(1)})\right) - \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log q(\mathbf{h}^{(1)}|\mathbf{x})$$

- for $q(\mathbf{h}^{(1)}|\mathbf{x})$ we use **the posterior of the first layer RBM**.
- by initializing the weights of the second layer RBM as the transpose of the first layer weights, **the bound is initially tight!** (As we showed, a 2-layer DBN with tied weights is equivalent to a 1-layer RBM)
- Need not keep being tight:
  as $p(\mathbf{h}^{(1)})$ changes, so does $p(\mathbf{h}^{(1)}|\mathbf{x})$, and so does the KL to $q(\mathbf{h}^{(1)}|\mathbf{x})$

Deep Belief Networks

This process of adding layers can be repeated recursively

- we obtain the greedy layer-wise pre-training procedure for neural networks

We now see that this procedure corresponds to maximizing a bound on the likelihood of the data in a DBN

- in theory, if our approximation $q(\mathbf{h}^{(1)}|\mathbf{x})$ is very far from the true posterior, the bound might be very loose
- this only means we might not be improving the true likelihood
- we might still be extracting better features!

Fine-tuning is done by the Up-Down algorithm

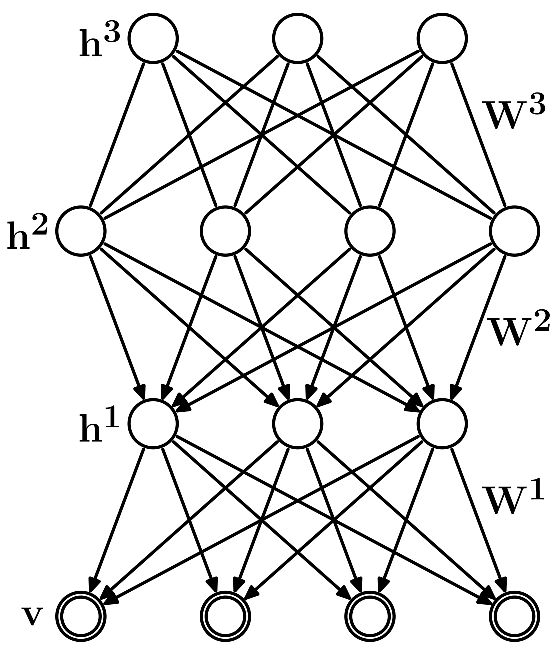
- A fast learning algorithm for deep belief nets. Hinton, Teh, Osindero, 2006.

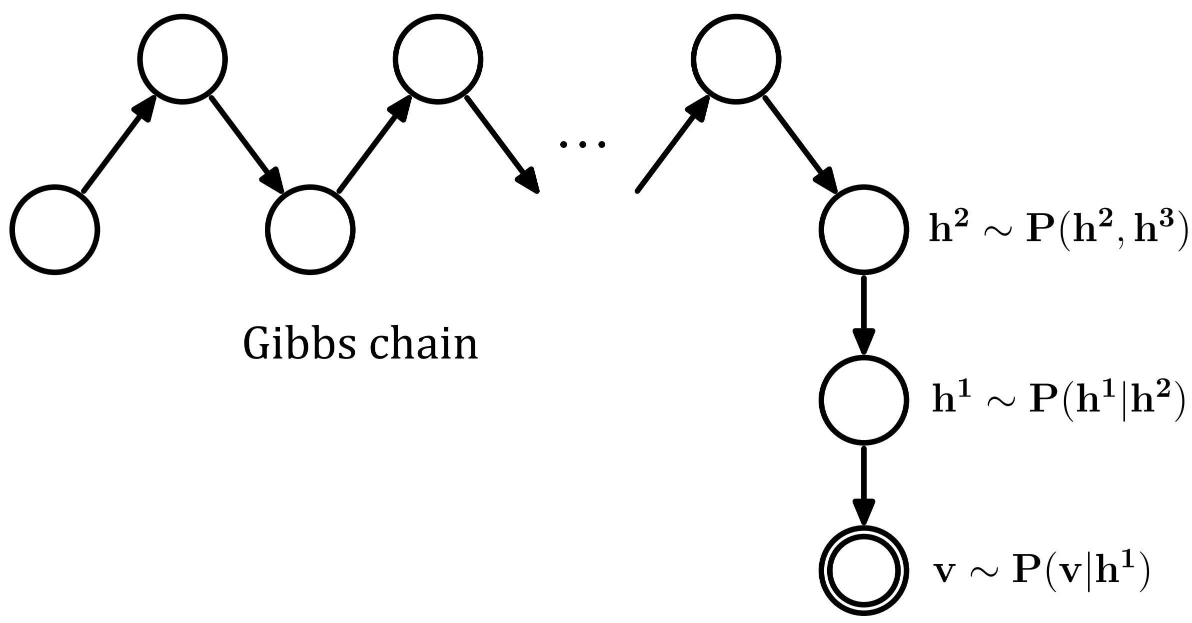
Sampling from DBNs

- To sample from the DBN model:

$$P(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2, \mathbf{h}^3) = P(\mathbf{v}|\mathbf{h}^1)P(\mathbf{h}^1|\mathbf{h}^2)P(\mathbf{h}^2, \mathbf{h}^3)$$

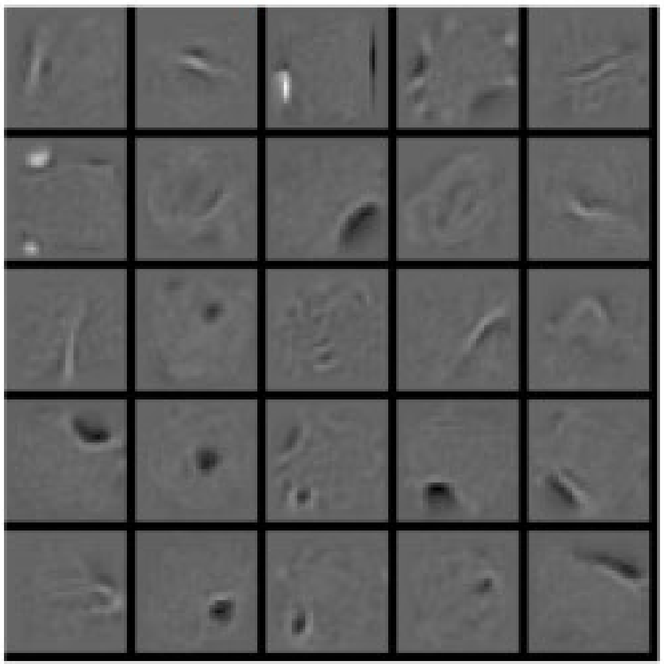
- Sample h^2 using alternating Gibbs sampling from RBM.
- Sample lower layers using sigmoid belief network.



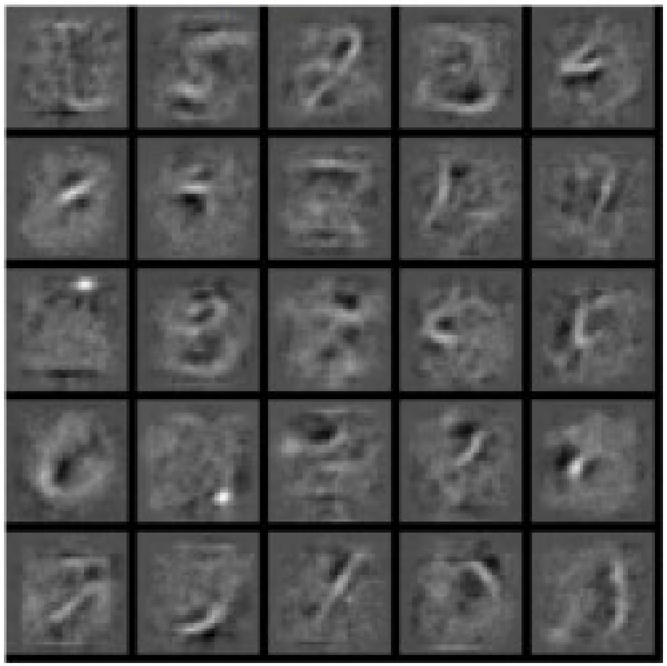


Learned Features

1^{st}-layer features

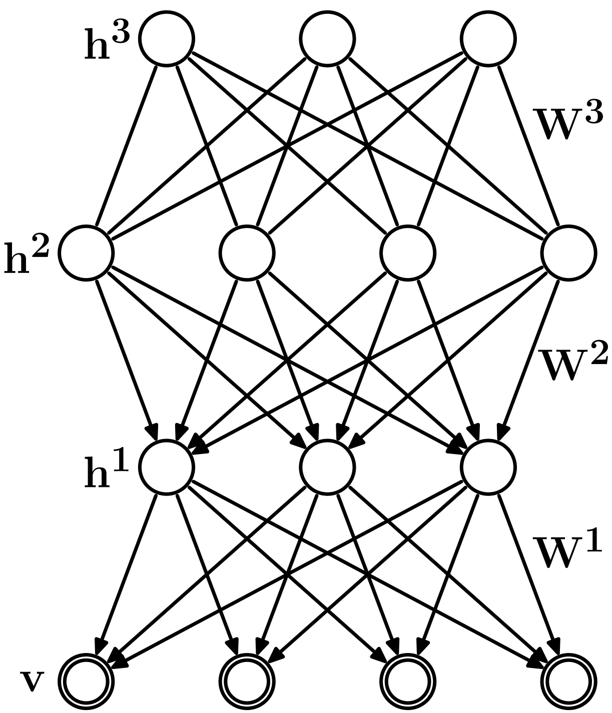


2^{nd}-layer features

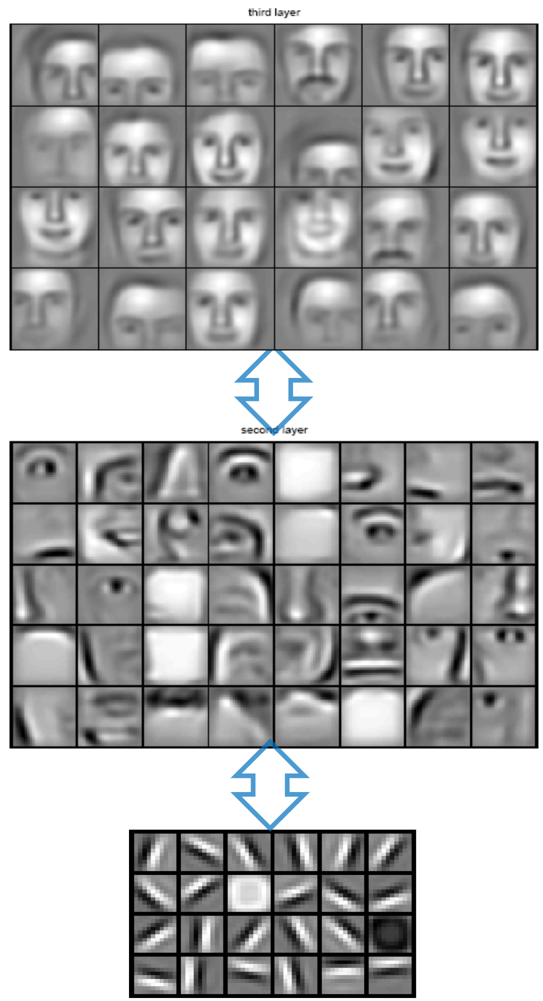


Learning Part-based Representation

Convolutional DBN



Faces



Groups of parts.

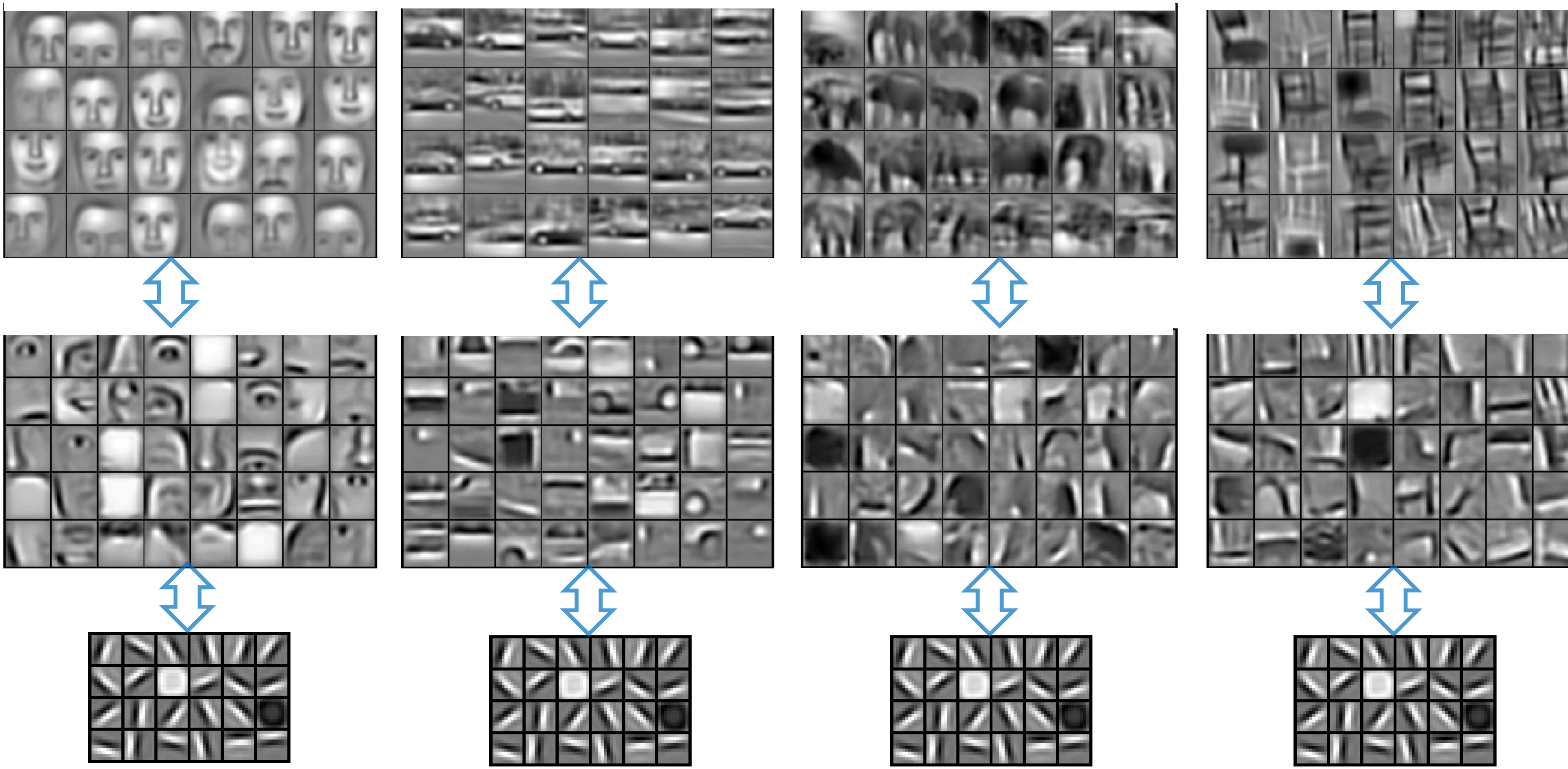
Object Parts

Trained on face images.

Lee et.al., ICML 2009

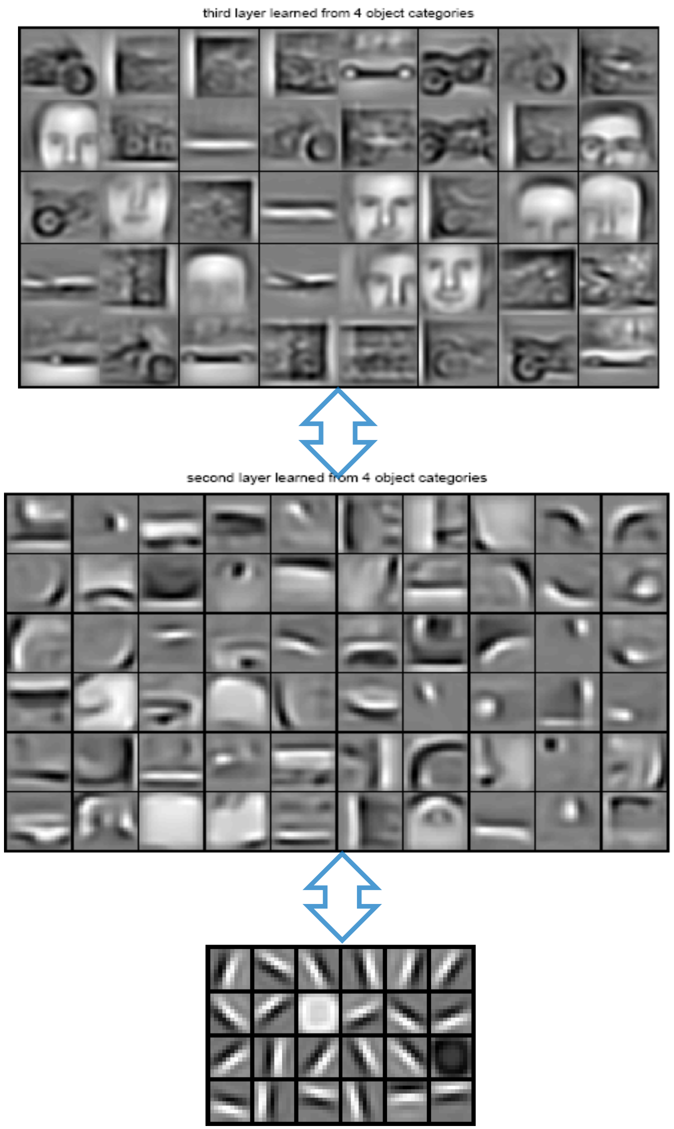
Learning Part-based Representation

Faces Cars Elephants Chairs



Lee et.al., ICML 2009

Learning Part-based Representation



Groups of parts.

Class-specific object parts

Trained from multiple classes (cars, faces, motorbikes, airplanes).

Lee et.al., ICML 2009

Part II: Variants of VAEs
(VQ-VAEs, NVAEs)

VQ-VAE, Oord et al ‘17, Razavi et al ‘19

Figure from Razavi et al ‘19

Basic idea: discrete latent space

Idea: perform k-means on the recovered latent vectors to discretize

Map the latent vectors to the closest mean (nearest-neighbor lookup):

$$q(z = k|x) = \begin{cases} 1 & \text{for } \mathrm{k} = \mathrm{argmin}_j \|z_e(x) - e_j\|_2, \\ 0 & \text{otherwise} \end{cases}$$

Latent embedding space

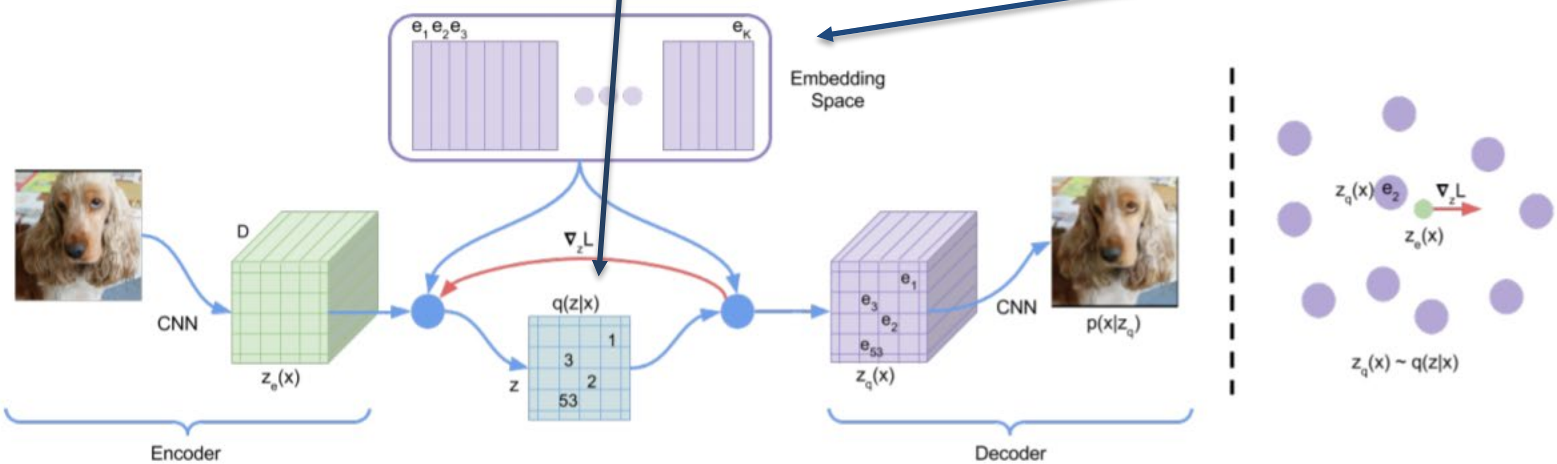


Figure 1: Left: A figure describing the VQ-VAE. Right: Visualisation of the embedding space. The output of the encoder $z(x)$ is mapped to the nearest point e_2. The gradient $\nabla_z L$ (in red) will push the encoder to change its output, which could alter the configuration in the next forward pass.

The loss

Idea: perform k-means on the recovered latent vectors to discretize

Map the latent vectors to the closest mean (nearest-neighbor lookup):

$$q(z=k|x) = \begin{cases} 1 & \text{for } \mathrm{k} = \mathrm{argmin}_j \|z_e(x) - e_j\|_2, \\ 0 & \text{otherwise} \end{cases}$$

Variational posterior q(z|x) is a distribution over a domain of size K, and a point mass. Let's denote $z_q(x) = \mathrm{argmin}_j \left\|z_e(x) - e_j\right\|_2$.

Loss (first try): $$L(\theta, e) = \mathbb{E}_x[\underbrace{\log(p_\theta(x|z_q)}_{\text{Reconstruction loss}} + \underbrace{KL(q(\cancel{h|x)}||p(h))}_{\text{Regularization towards prior}}$$

The authors drop the regularization term.

Problem: the mapping $z_e \to z_q$ involves argmin and is not differentiable.

The loss: straight-through estimator

Idea: perform k-means on the recovered latent vectors to discretize

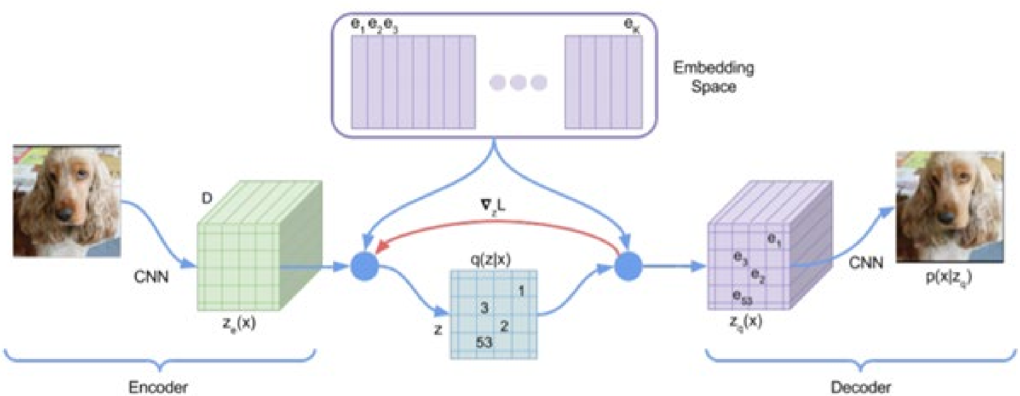
Map the latent vectors to the closest mean (nearest-neighbor lookup):

$$q(z=k|x) = \begin{cases} 1 & \text{for } \mathrm{k} = \mathrm{argmin}_j \|z_e(x) - e_j\|_2, \\ 0 & \text{otherwise} \end{cases}$$

Loss (first try): $L(\theta, e) = \mathbb{E}_x[\log(p_\theta(x|z_q))]$

Problem: the mapping $z_e \to z_q$ involves argmin and is not differentiable.

Solution: use straight-through estimator; copy gradients from decoder input $z_q(x)$ to encoder output $z_e(x)$



The loss: quantization

Idea: perform k-means on the recovered latent vectors to discretize

Map the latent vectors to the closest mean (nearest-neighbor lookup):

$$q(z = k|x) = \begin{cases} 1 & \text{for } k = \text{argmin}_j \|z_e(x) - e_j\|_2, \\ 0 & \text{otherwise} \end{cases}$$

Problem: the cluster means $\{e_j\}$ are not getting updated.

Loss (second try): $L(\theta, e) = \mathbb{E}_x[\underbrace{\log(p_\theta(x|z_q)}_{\text{Reconstruction loss}} + \underbrace{\left\|SG(z_e(x)) - z_q(x)\right\|^2}_{\text{Quantization loss}}]$

$SG(z_e(x))$: stop-gradient operator; identity at forward computation; has zero derivative, so argument doesn't get update at backward computation

This term only updates cluster means $\{e_j\}$!

The loss: quantization

Idea: perform k-means on the recovered latent vectors to discretize

Map the latent vectors to the closest mean (nearest-neighbor lookup):

$$q(z = k|x) = \begin{cases} 1 & \text{for } \mathrm{k} = \mathrm{argmin}_j \|z_e(x) - e_j\|_2, \\ 0 & \text{otherwise} \end{cases}$$

Problem: the cluster means $\{e_j\}$ are not getting updated.

Loss (second try): $L(\theta, e) = \mathbb{E}_x[\underbrace{\log(p_\theta(x|z_q)}_{\text{Reconstruction loss}} + \underbrace{\left|\left|SG(z_e(x)) - z_q(x)\right|\right|^2}_{\text{Quantization loss}}]$

(Alternatively, this term can be rewritten as follows. If $z_{1,i}, \ldots, z_{n_i,i}$ are the decoded samples closest to e_i, this term is $\sum_i \sum_{j=1}^{n_i} \left|\left|z_{j,i} - e_i\right|\right|^2$.
New e_i can just be set to $e_i := \sum_{j=1}^{n_i} z_{j,i}$.)

The loss: commitment penalty

Idea: perform k-means on the recovered latent vectors to discretize

Map the latent vectors to the closest mean (nearest-neighbor lookup):

$$q(z = k|x) = \begin{cases} 1 & \text{for } \mathrm{k} = \mathrm{argmin}_j \|z_e(x) - e_j\|_2, \\ 0 & \text{otherwise} \end{cases}$$

Loss (third try):

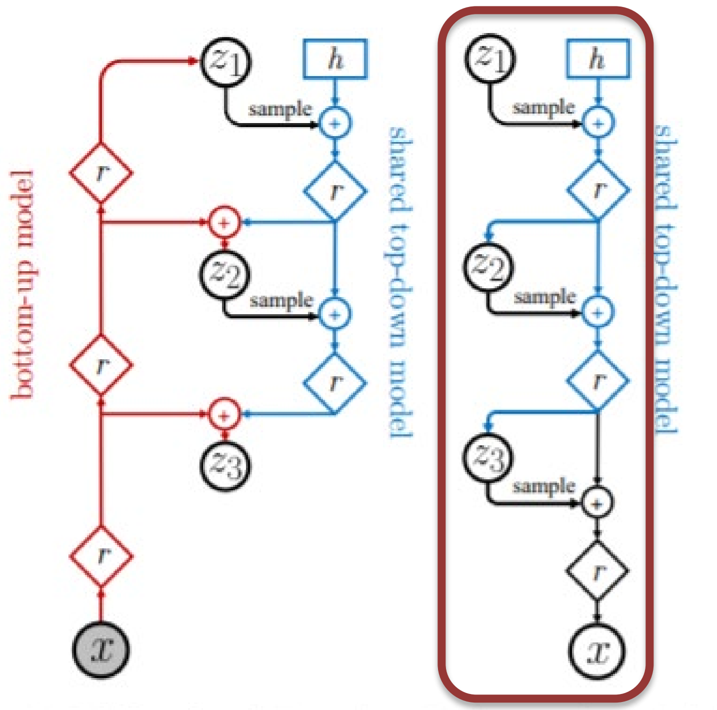
$$L(\theta, e) = \mathbb{E}_x[\underbrace{\log(p_\theta(x|z_q)}_{\text{Reconstruction loss}} + \underbrace{\left|\left|SG(z_e(x)) - z_q(x)\right|\right|^2}_{\text{Quantization loss}} + \underbrace{\beta \left|\left|z_e(x) - SG(z_q(x))\right|\right|^2}_{\text{Commitment loss}}]$$

Authors add a commitment loss: this "regularizer" attempts to keep unquantized $z_e(x)$ close to current means $\{e_i\}$.

NVAE, Vahdat-Kautz ’21

Figure from Vahdat-Kautz ‘21

Basic idea: careful changes in architecture



(a) Bidirectional Encoder (b) Generative Model

Figure 2: The neural networks implementing an encoder $q(\boldsymbol{z}|\boldsymbol{x})$ and generative model $p(\boldsymbol{x}, \boldsymbol{z})$ for a 3-group hierarchical VAE. ⟨r⟩ denotes residual neural networks, ⊕ denotes feature combination (e.g., concatenation), and $\boxed{h}$ is a trainable parameter.

Main idea: hierarchical model for the generative and inference direction, with careful choice of architecture;

Generative model:

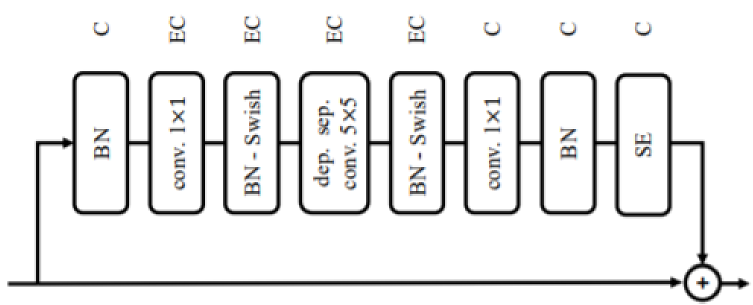
Use residual networks from layer to layer; the dimensions of the z's gradually increase to gradually add more detail to image.

Basic idea: careful changes in architecture

Generative model:

Use residual networks from layer to layer; the dimensions of the z's gradually increase to gradually add more detail to image.

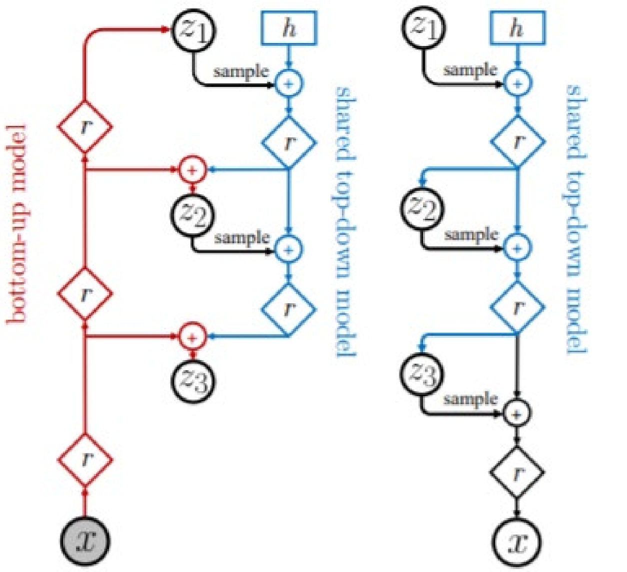
Hyperparamter	**MNIST** 28×28	**CIFAR-10** 32×32	**ImageNet** 32×32	**CelebA** 64×64	**CelebA HQ** 256×256	**FFHQ** 256×256
# latent variable scales	2	1	1	3	5	5
# groups in each scale	5, 10	30	28	5, 10, 20	4, 4, 4, 8, 16	4, 4, 4, 8, 16
spatial dims of $\boldsymbol{z}$ in each scale	$4^2, 8^2$	16^2	16^2	$8^2, 16^2, 32^2$	$8^2, 16^2, 32^2, 64^2, 128^2$	$8^2, 16^2, 32^2, 64^2, 128^2$



(a) Residual Cell for NVAE Generative Model

Basic idea: careful changes in architecture

Encoder: weight tied w/ decoder for better behavior of KL term



(a) Bidirectional Encoder (b) Generative Model

Figure 2: The neural networks implementing an encoder $q(\boldsymbol{z}|\boldsymbol{x})$ and generative model $p(\boldsymbol{x}, \boldsymbol{z})$ for a 3-group hierarchical VAE. $\Diamond$ denotes residual neural networks, $\oplus$ denotes feature combination (e.g., concatenation), and $\boxed{h}$ is a trainable parameter.

Recall, there is a term $KL(q(z|x)||p(z))$ which will be large if q and p are far.

Weight tying: if we parametrize $p(z)$ s.t.

$$p(z_l^i|\boldsymbol{z}_{<l}) := \mathcal{N}\left(\mu_i(\boldsymbol{z}_{<l}), \sigma_i(\boldsymbol{z}_{<l})\right)$$

we parametrize $q(z|x)$ correspondingly as

$$q(z_l^i|\boldsymbol{z}_{<l}, \boldsymbol{x}) := \mathcal{N}\left(\mu_i(\boldsymbol{z}_{<l}) + \Delta\mu_i(\boldsymbol{z}_{<l}, \boldsymbol{x}), \sigma_i(\boldsymbol{z}_{<l}) \cdot \Delta\sigma_i(\boldsymbol{z}_{<l}, \boldsymbol{x})\right)$$

("relative to p" parametrization)

Then, we have:

$$\mathrm{KL}\left(q(z^i|\boldsymbol{x})||p(z^i)\right) = \frac{1}{2}\left(\frac{\Delta\mu_i^2}{\sigma_i^2} + \Delta\sigma_i^2 - \log\Delta\sigma_i^2 - 1\right)$$