Shallow Autoencoder

Ke Chen

Department of Computer Science, The University of Manchester

Ke.Chen@manchester.ac.uk

Outline

Overview

History and main properties

Traditional and Denoising Autoencoder

Architecture, Loss function, learning algorithm, illustrative example, manifold perspective

Sparse Autoencoder

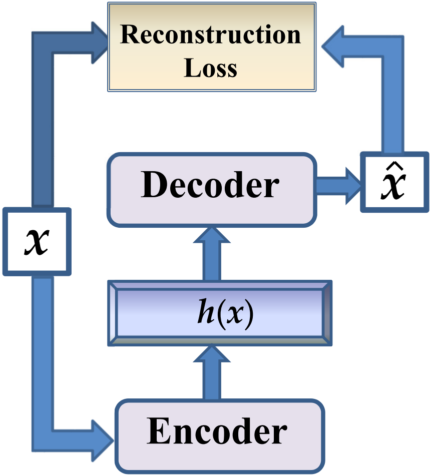
Architecture, Loss function, learning algorithm, illustrative example

Restricted Boltzmann Machine

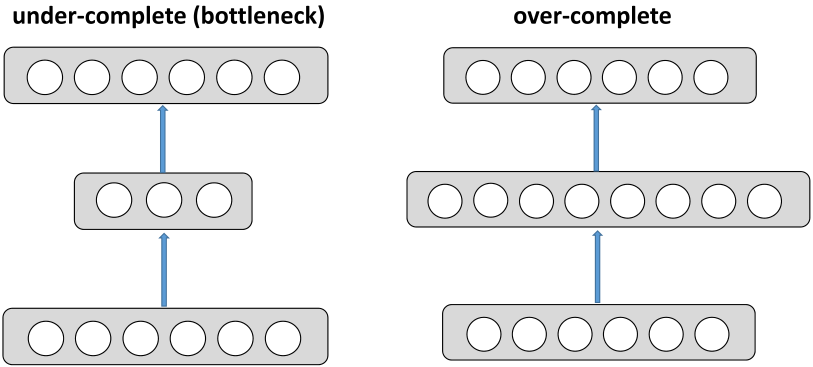
Architecture, energy function, learning algorithm, continuous input, illustrative example

Overview

- Autoencoder (AE) is a specific type of neural networks, originally named auto-associator in 1980s for dimension reduction and feature extraction.
- In general, AE consists of two components, encoder and decoder, to learn reconstruction of data itself in a self-supervised learning manner.
- There are a variety of AEs; discriminative vs. probabilistic, static vs. dynamic, structured vs. unstructured, $\cdots$
- Nowadays, AE is a centre of representation learning and closely related to many areas ranging from manifold learning to generative modelling.



- In general, AEs may generate two different types of representations.
  - Under-complete (bottleneck): the dimension of learned representation is lower than that of data for dimension reduction
  - Over-complete: the dimension of learned representation is higher than that of data to discover and capture intrinsic structures underlying data



Traditional Autoencoder (AE)

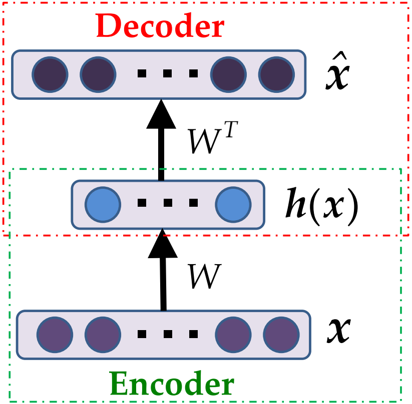
- Architecture: a MLP of a single bottleneck hidden layer ($|\boldsymbol{h}| < |\boldsymbol{x}|$) and tied weight matrix to generate an under-complete representation
- The hidden layer is often named coding layer.
- **Encoder**

$$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{a}_h), \ \boldsymbol{a}_h = W\boldsymbol{x} + \boldsymbol{b}_h \qquad \boxed{\boldsymbol{f}(\boldsymbol{a}_h) = \left\{ f(a_{h,j}) \right\}_{j=1}^{|\boldsymbol{h}|}}$$

- **Decoder**

$$\hat{\boldsymbol{x}} = \boldsymbol{g}(\boldsymbol{a}_o), \ \boldsymbol{a}_o = W^T\boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{b}_o \qquad \boxed{\boldsymbol{g}(\boldsymbol{a}_o) = \left\{ g(a_{o,j}) \right\}_{j=1}^{|\boldsymbol{x}|}}$$

$W_{|\boldsymbol{h}| \times |\boldsymbol{x}|}$: (tied) weight matrix
$\boldsymbol{b}_h, \boldsymbol{b}_o$: biases for hidden and output layers



Fact: PCA is a special case of then traditional AE when $f(\cdot)$ and $g(\cdot)$ are linear activation function.

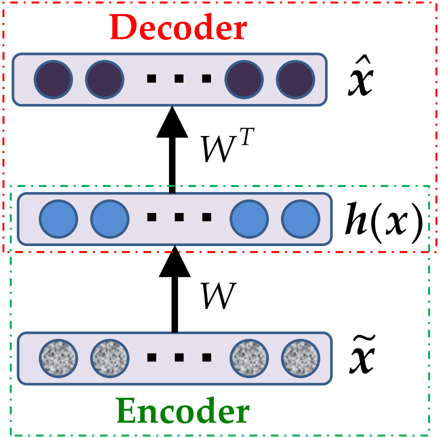
Denoising Autoencoder (DAE)

- Archhitecture: a MLP of a hidden layer and tied weight matrix to generate either under-complete or over-complete representation.
- Learn denoising by recovering a data point, $\boldsymbol{x}$, from its corrupted noisy version, $\tilde{\boldsymbol{x}}$
- In deployment phase, it can generate a proper representation directly from test data.
- **Encoder**

$$\boldsymbol{h}(\tilde{\boldsymbol{x}}) = \boldsymbol{f}(\boldsymbol{a}_h), \ \ \boldsymbol{a}_h = W\tilde{\boldsymbol{x}} + \boldsymbol{b}_h.$$

- **Decoder**

$$\hat{\boldsymbol{x}} = \boldsymbol{g}(\boldsymbol{a}_o), \ \ \boldsymbol{a}_o = W^T\boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{b}_o.$$



Fact: In DAE, untied weight matrices are used occasionally in applications.

Noisy Training Data Generation

- Given a training dataset, $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^{|\mathcal{D}|}$, generate noisy data, $\tilde{\boldsymbol{x}}_i$, to train DAE by corrupting $\boldsymbol{x}_i$ with
  - Gaussian noise: for real-valued $\boldsymbol{x}_i$, $\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{\epsilon}$, $\boldsymbol{\epsilon}$ randomly drawn from $N(\mathbf{0}, \sigma^2 I)$ and the amount of noise controlled by σ
  - Salt-and-peper noise: for discrete-valued $\boldsymbol{x}_i$, generate $\tilde{\boldsymbol{x}}_i$ by flipping some randomly chosen elements' value of $\boldsymbol{x}_i$ to either maximum or minimum of the domain range in $\mathcal{D}$
  - Masking noise: for discrete-valued $\boldsymbol{x}_i$, generate $\tilde{\boldsymbol{x}}_i$ by setting some randomly chosen elements of $\boldsymbol{x}_i$ to zero

Loss Function

Given a training dataset, $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{x}_i)\}_{i=1}^{|\mathcal{D}|}$ (AE) or $\mathcal{D} = \{(\tilde{\boldsymbol{x}}_i, \boldsymbol{x}_i)\}_{i=1}^{|\mathcal{D}|}$ (DAE), loss functions for AE/DAE are defined based on real-valued or binary-valued input

- Mean squared error (MSE) loss for real-valued or categorical-valued input (**loss-1**)

$$\mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o; \mathcal{D}) = \frac{1}{2|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} ||\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i||^2. \quad (1)$$

$\hat{\boldsymbol{x}}_i$: output of AE/DAE for input $\boldsymbol{x}_i$
$g(\cdot)$: linear activation function in output layer (see Slides 5 and 6)

- Cross-entropy loss for binary-valued input (**loss-2**)

$$\mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o; \mathcal{D}) = -\frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \sum_{j=1}^{|\boldsymbol{x}_i|} \Big(x_{ij} \log \hat{x}_{ij} + (1 - x_{ij}) \log(1 - \hat{x}_{ij}) \Big), \; x_{ij} \in \{0, 1\}. \quad (2)$$

$\hat{\boldsymbol{x}}_i = (\hat{x}_{i1}, \cdots, \hat{x}_{ij}, \cdots, \hat{x}_{i|\boldsymbol{x}_i|})$: output of AE/DAE for input $\boldsymbol{x}_i = (x_{i1}, \cdots, x_{ij}, \cdots, x_{i|\boldsymbol{x}_i|})$
$g(\cdot)$: sigmoid activation function in output layer (see Slides 5 and 6)

Learning Algorithm

Input a training set, $\mathcal{D} = \{(\boldsymbol{z}_i, \boldsymbol{x}_i)\}_{i=1}^{|\mathcal{D}|}$ where $\boldsymbol{z}_i = \boldsymbol{x}_i$ for AE or $\boldsymbol{z}_i = \tilde{\boldsymbol{x}}_i$ for DAE
Randomly initialize $W, \boldsymbol{b}_h$ and $\boldsymbol{b}_o$ and pre-set a learning rate η and batch size $|\mathcal{B}|$

- **Forward Computation**

  For the input $\boldsymbol{z}_i$ $(i = 1, \cdots, |\mathcal{B}|)$, output of the hidden layer is

  $$\boldsymbol{h}(\boldsymbol{z}_i) = \boldsymbol{f}\left(\boldsymbol{a}_h(\boldsymbol{z}_i)\right), \quad \boldsymbol{a}_h(\boldsymbol{z}_i) = W\boldsymbol{z}_i + \boldsymbol{b}_h.$$

  And output of the output layer is

  $$\hat{\boldsymbol{x}}_i = \boldsymbol{g}\left(\boldsymbol{a}_o(\boldsymbol{z}_i)\right), \quad \boldsymbol{a}_o(\boldsymbol{z}_i) = W^T\boldsymbol{h}(\boldsymbol{z}_i) + \boldsymbol{b}_o.$$

  where $g(\cdot)$ is linear and sigmoid activation function for real-valued/categorical-valued and binary-valued input, respectively.

Learning Algorithm

- **Backward Gradient Computation**

  For $i = 1, 2, \cdots, |\mathcal{B}|$, compute gradients of loss function with respect to parameters
  - Gradients for the output layer depend on loss functions.

$$\textbf{loss-1}: \boldsymbol{\delta}_o(\boldsymbol{z}_i, \boldsymbol{x}_i) = \frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{a}_o(\boldsymbol{z}_i)} = \hat{\boldsymbol{x}}_i - \boldsymbol{x}_i$$

$$\textbf{loss-2}: \boldsymbol{\delta}_o(\boldsymbol{z}_i, \boldsymbol{x}_i) = \frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{a}_o(\boldsymbol{z}_i)} = \boldsymbol{x}_i - \hat{\boldsymbol{x}}_i$$

  - Gradients for hidden layer can be computed with gradient of output layer.

$$\frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{h}(\boldsymbol{z}_i)} = W\boldsymbol{\delta}_o(\boldsymbol{z}_i), \quad \boldsymbol{\delta}_h(\boldsymbol{z}_i, \boldsymbol{x}_i) = \frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{a}_h(\boldsymbol{z}_i)} = \Big(\boldsymbol{f}'\big(\boldsymbol{a}_h(\boldsymbol{z}_i)\big)\Big) \odot \frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{h}(\boldsymbol{z}_i)}$$

  - Gradients for biases are computed as follows:

$$\frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{b}_o} = \boldsymbol{\delta}_o(\boldsymbol{z}_i, \boldsymbol{x}_i), \quad \frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{b}_h} = \boldsymbol{\delta}_h(\boldsymbol{z}_i, \boldsymbol{x}_i).$$

Learning Algorithm

- **Parameter Update**

  Applying the gradient descent method and tied weight matrix leads to update rules:

$$W \leftarrow W - \frac{\eta}{2|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}\Big(\underbrace{\boldsymbol{\delta}_h(\boldsymbol{z}_i,\boldsymbol{x}_i)\boldsymbol{z}_i^T}_{\text{encoder}} \; + \; \underbrace{\boldsymbol{h}(\boldsymbol{z}_i)\left(\boldsymbol{\delta}_o(\boldsymbol{z}_i,\boldsymbol{x}_i)\right)^T}_{\text{decoder}}\Big),$$

$$\boldsymbol{b}_o \leftarrow \boldsymbol{b}_o - \frac{\eta}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}\boldsymbol{\delta}_o(\boldsymbol{z}_i,\boldsymbol{x}_i),$$

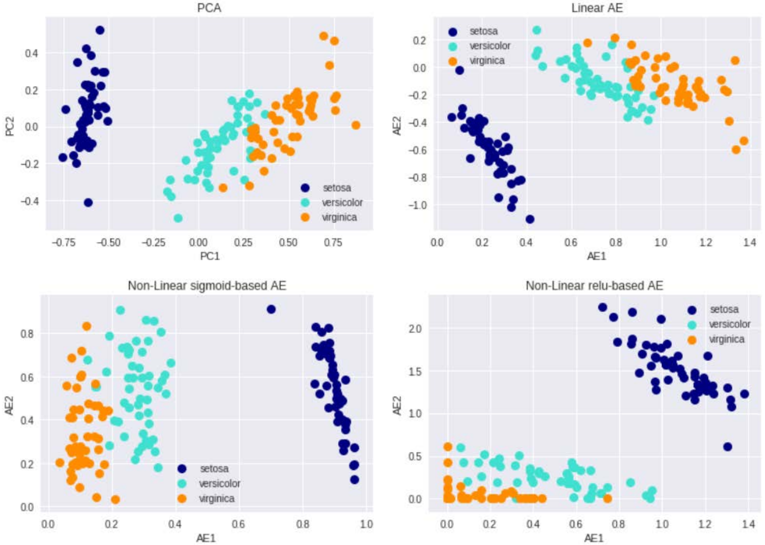
$$\boldsymbol{b}_h \leftarrow \boldsymbol{b}_h - \frac{\eta}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}\boldsymbol{\delta}_h(\boldsymbol{z}_i,\boldsymbol{x}_i).$$

The above three steps repeat for all mini-batches until terminated with early stopping.

Fact: While the traditional AE always uses the tied weights, DAE may use untied weights in applications.

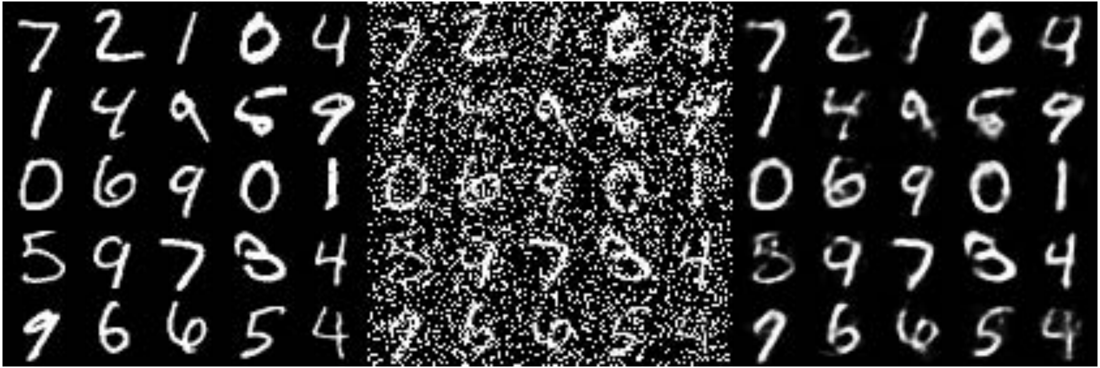
Illustrative Example

- Visualisation: AE ($|\boldsymbol{h}| = 2$) versus PCA ($p = 2$) on Iris dataset ($|\boldsymbol{x}| = 4$)



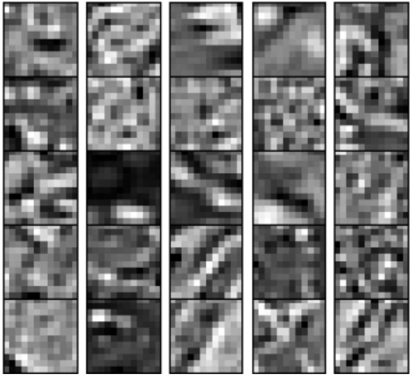
Illustrative Example

- Denoising: bottleneck DAE; original (left), noisy (middle), recovered (right)



Illustrative Example

- Sparse representation: over-complete DAE (200 hidden neurons, Gaussian noise $\sigma = 0.5$; 12×12 image patches (left), 12×12 weights associated with each of first 100 hidden neurons (right) that learns Gabor-like local oriented edge detectors



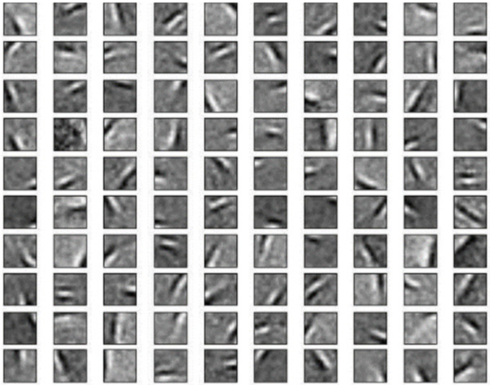
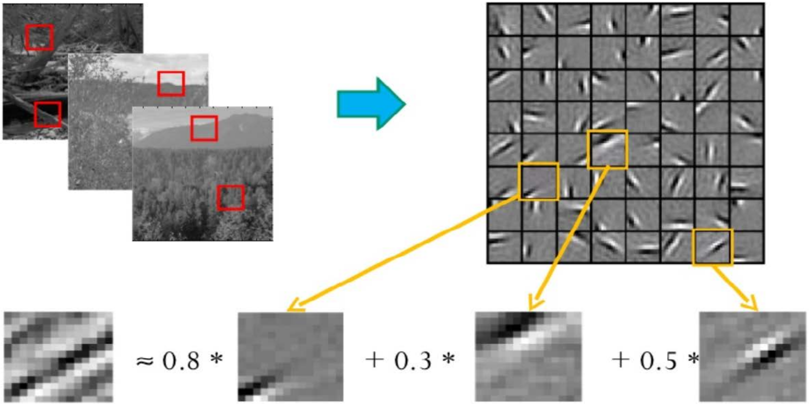


Illustration of Sparse Representation

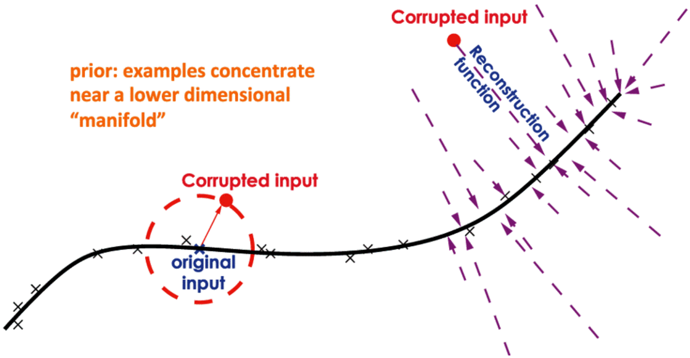
- Sparse representation: compact, interpretable, biologically plausible



$[a_1, \ldots, a_{64}] = [0, 0, \ldots, 0, \mathbf{0.8}, 0, \ldots, 0, \mathbf{0.3}, 0, \ldots, 0, \mathbf{0.5}, 0]$
(feature representation)

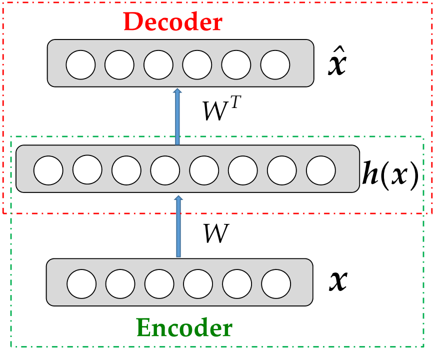
Manifold perspective of DAE

- Manifold assumption: natural high-dimensional data often concentrated close to a nonlinear low-dimensional manifold
- DAE: learn modelling manifold and capture main variations along the manifold
- Output of encoder: interpreted as a coordinate system on the manifold



Sparse Autoencoder (SAE)

- Architecture: NNs of encoder and decoder that generate the over-complete representation
- AE extensible to SAE by adding a regularisation penalty to avoid learning identity function
- Learning with regularised loss leads to sparse representation reflecting intrinsic structure underlying data.



- **Encoder**

$$\boldsymbol{h}(\boldsymbol{x}) = f(\boldsymbol{a}_h), \ \boldsymbol{a}_h = W\boldsymbol{x} + \boldsymbol{b}_h.$$

- **Decoder**

$$\hat{\boldsymbol{x}} = g(\boldsymbol{a}_o), \ \boldsymbol{a}_o = W^T\boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{b}_o.$$

Fact: In SAE, untied weight matrices may be used in applications.

KL-sparsity Penalty

- Motivation: make hidden neurons inactive in most of time
- Given a training dataset, $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^{|\mathcal{D}|}$, the averaged activation of hidden neuron j:

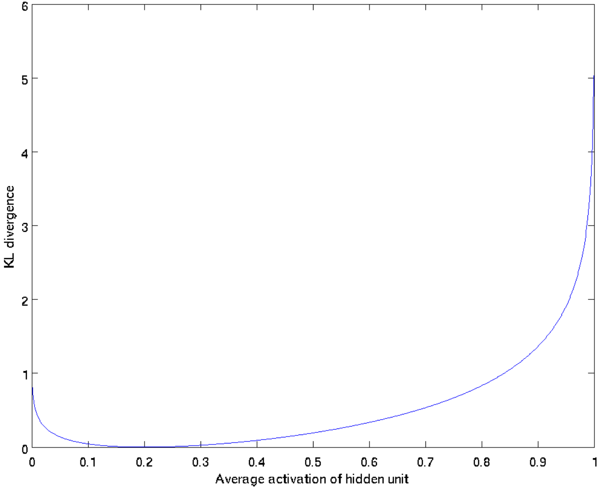
$$\hat{\rho}_j = \frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|} h_j(\boldsymbol{x}_i) = \frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|} f\left(a_{h,j}(\boldsymbol{x}_i)\right), \quad j = 1, 2, \cdots, |\boldsymbol{h}|.$$

- To ensure the sparsity, set up a constraint: $\hat{\rho}_j = \rho$, where ρ (sparsity degree) is a constant of small value close to zero, e.g., 0.05.
- KL-sparsity penalty

$$\mathcal{R}_{KL}(W, \boldsymbol{b}_h) = \sum_{j=1}^{|\boldsymbol{h}|} KL(\rho||\hat{\rho}_j) = \sum_{j=1}^{|\boldsymbol{h}|}\left(\rho\log\frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}\right).$$

KL-sparsity Penalty

- Property: $KL(\rho||\hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$ or increases monotonically as $\hat{\rho}_j$ diverges from ρ. For instance, set $\rho = 0.2$ for one sigmoid hidden neuron



SAE Learning

- Based on **loss-1** or **loss-2** , the regularised loss for SAE is

$$\mathcal{L}_R(W, \boldsymbol{b}_h, \boldsymbol{b}_o; \mathcal{D}, \rho) = \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o; \mathcal{D}) + \lambda \mathcal{R}_{KL}(W, \boldsymbol{b}_h),$$

where λ is a trade-off coefficient (hyper-parameter to be tuned during learning).

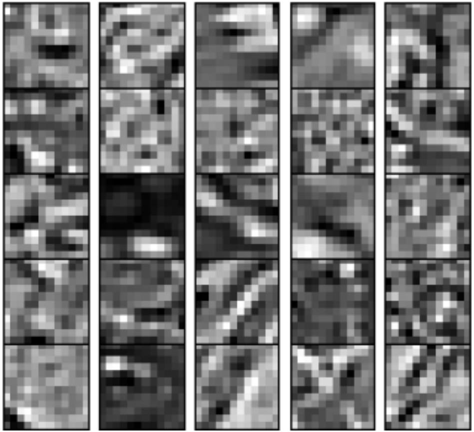
- Adapt the learning algorithm for AE to SAE with the following modification:
  1. **Forward computation**: further compute the averaged activations of all $|\boldsymbol{h}|$ hidden neurons, $\hat{\rho}_1, \hat{\rho}_2, \cdots, \hat{\rho}_{|\boldsymbol{h}|}$, on training dataset $\mathcal{D}$
  2. **Backward gradient computation**: add the gradient of $\mathcal{R}_{KL}(W, \boldsymbol{b}_h)$ to that associated with the hidden layer

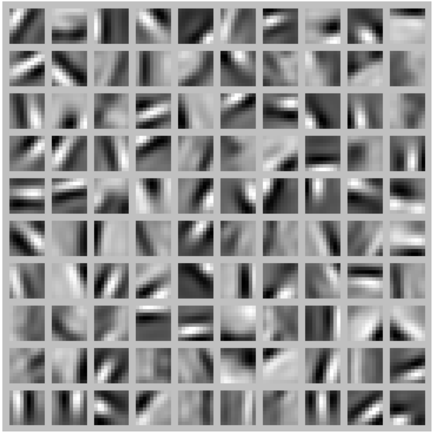
$$\boldsymbol{\delta}_h(\boldsymbol{x}_i, \boldsymbol{x}_i) = \frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{a}_h(\boldsymbol{x}_i)} = \left(\boldsymbol{f}'\left(\boldsymbol{a}_h(\boldsymbol{x}_i)\right)\right) \odot \left(\frac{\partial \mathcal{L}(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{h}(\boldsymbol{x}_i)} + \lambda \boldsymbol{\delta}_{KL}\right),$$

where $\boldsymbol{\delta}_{KL} = \left\{\delta_{KL}^{(j)}\right\}_{j=1}^{|\boldsymbol{h}|}$, $\delta_{KL}^{(j)} = \frac{\partial \mathcal{R}(W, \boldsymbol{b}_h)}{\partial a_{h,j}(\boldsymbol{x}_i)} = -\frac{\rho}{\hat{\rho}_j} + \frac{1-\rho}{1-\hat{\rho}_j}$ and $\boldsymbol{a}_h(\boldsymbol{x}_i) = \{a_{h,j}\}_{j=1}^{|\boldsymbol{h}|}$.

Illustrative Example

- Sparse representation: SAE (200 hidden neurons, $\rho = 0.05$); 12×12 image patches (left), 12×12 weights associated with each of first 100 hidden neurons (right) that learns Gabor-like local oriented edge detectors





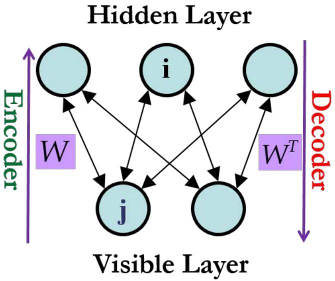
Restricted Boltzmman Machine (RBM)

- Architecture: 2-layer probabilistic NN of encoder and decoder with bi-directional weight matrix: $W_{ij} = W_{ji}$
- Connections: only between visible and hidden layer
- Probabilistic neuron: output probability for hidden state and "reconstruction" for binary-valued hidden and visible units, $h_i \in \{1, 0\}$, $v_j \in \{1, 0\}$, $P(h_i = 1|\boldsymbol{v})$, $P(v_j = 1|\boldsymbol{h})$
- **Encoder**

$$P(h_i|\boldsymbol{v}) = \phi\Big(\sum_{j=1}^{|\boldsymbol{v}|} W_{ij} v_j + b_{h,i}\Big), \ P(\boldsymbol{h}|\boldsymbol{v}) = \prod_{i=1}^{|\boldsymbol{h}|} P(h_i|\boldsymbol{v}).$$

- **Decoder**

$$P(v_j|\boldsymbol{h}) = \phi\Big(\sum_{i=1}^{|\boldsymbol{h}|} W_{ji} h_i + b_{v,j}\Big), \ P(\boldsymbol{h}|\boldsymbol{v}) = \prod_{j=1}^{|\boldsymbol{v}|} P(v_j|\boldsymbol{h}).$$



Sigmoid activation function

$$\boxed{\phi(x) = \frac{1}{1 + e^{-x}}}$$

Fact: RBM inspired by Physics was invented by P. Smolensky in 1986 and rose to prominence after G. Hinton in 2002.

- Energy function: associate a scalar energy to each configuration of states in system

$$E(\boldsymbol{v}, \boldsymbol{h}; \Theta) = -\boldsymbol{v}^T W \boldsymbol{h} - \boldsymbol{b}_v^T \boldsymbol{v} - \boldsymbol{b}_h^T \boldsymbol{h} = -\sum_{i=1}^{|\boldsymbol{h}|} \sum_{j=1}^{|\boldsymbol{v}|} W_{ij} h_i v_j - \sum_{j=1}^{|\boldsymbol{v}|} v_j b_{v,j} - \sum_{i=1}^{|\boldsymbol{h}|} h_i b_{h,i},$$

  where $\Theta = \{W, \boldsymbol{b}_h, \boldsymbol{b}_v\}$.

- Boltzmann distribution: joint probability of all random variables with energy function

$$P(\boldsymbol{v}, \boldsymbol{h}) = \frac{e^{-E(\boldsymbol{v}, \boldsymbol{h}; \Theta)}}{Z}, \quad \text{Partition: } Z = \sum_{\boldsymbol{v}} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h}; \Theta)}$$

- Loss function: learn "reconstructing" $\boldsymbol{x}$ from a training dataset $\mathcal{D} = \{\boldsymbol{x}_t)\}_{t=1}^{|\mathcal{D}|}$

$$\mathcal{L}(\Theta; \mathcal{D}) = -\log P(\boldsymbol{v}) = -\log \sum_{\boldsymbol{h}} P(\boldsymbol{v}, \boldsymbol{h}) = -(\log \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h}; \Theta)} - \log Z).$$

  The optimisation of the loss function is intractable due to the partition Z. Different sampling-based approximation algorithms were proposed for RBM learning.

Gibbs sampling: one step ($k = 1$)

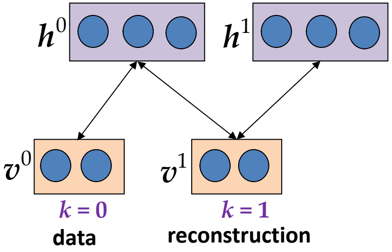
- set $\boldsymbol{v}^0 = \boldsymbol{x}$, $\boldsymbol{x} \in \mathcal{D}$.
- estimate $\{P(h_i^0|\boldsymbol{v}^0)\}_{i=1}^{|\boldsymbol{h}|}$ with encoder and then form realisation of $\boldsymbol{h}^0$ by sampling with these probabilities

$$P(h_i^0 = 1|\boldsymbol{v}^0) = \phi\Big(\sum_{j=1}^{|\boldsymbol{v}|} W_{ij} v_j^0 + b_{h,i}\Big)$$

- use decoder to estimate $\Big\{P(v_j^1|\boldsymbol{h}^0)\Big\}_{j=1}^{|\boldsymbol{v}|}$ and then generate "reconstruction", $\boldsymbol{v}^1$, via sampling

$$P(v_j^1 = 1|\boldsymbol{h}^0) = \phi\Big(\sum_{i=1}^{|\boldsymbol{h}|} W_{ji} h_i^0 + b_{v,j}\Big)$$

- with the "reconstruction", $\boldsymbol{v}^1$, use encoder to estimate probabilities $\{P(h_i^1|\boldsymbol{v}^1)\}_{i=1}^{|\boldsymbol{h}|}, \cdots$



Realisation via sampling

$$\boxed{h_i = 1/0, \text{ if } P(h_i^0|\boldsymbol{v}^0) \gtrless u \sim U[0,1]}$$

$$P(h_i^1 = 1|\boldsymbol{v}^1) = \phi\Big(\sum_{j=1}^{|\boldsymbol{v}|} W_{ij} v_j^1 + b_{h,i}\Big)$$

Contrastive Divergence (CD-1) Learning Algorithm

Input: a training dataset, $\mathcal{D} = \{\boldsymbol{x}_t)\}_{t=1}^{|\mathcal{D}|}$, randomly initialise $W, \boldsymbol{b}_h$ and $\boldsymbol{b}_v$, and pre-set a learning rate, η, and a mini-batch size, $|\mathcal{B}|$.

For $t = 1, 2, \cdots, |\mathcal{B}|$, do steps 1 & 2 as follows:

1. **Data Phase**
   - present an instance to the visible layer, i.e., $\boldsymbol{v}_t^0 = \boldsymbol{x}_t$.
   - estimate probabilities with encoder: $P(\boldsymbol{h}_t^0|\boldsymbol{v}_t^0) = \Big(P(h_{ti}^0 = 1|\boldsymbol{v}_t^0)\Big)_{i=1}^{|\boldsymbol{h}|}$
2. **Model Phase**
   - Form a realisation of $\boldsymbol{h}_t^0$ by sampling with probabilities $P(\boldsymbol{h}_t^0|\boldsymbol{v}_t^0)$.
   - With the realisation of $\boldsymbol{h}_t^0$, apply the decoder to estimate probabilities: $P(\boldsymbol{v}_t^1|\boldsymbol{h}_t^0) = \Big(P(v_{tj}^0 = 1|\boldsymbol{h}_t^0)\Big)_{j=1}^{|\boldsymbol{v}|}$, and then produce a "reconstruction" $\boldsymbol{v}_t^1$ via sampling with probabilities $P(\boldsymbol{v}_t^1|\boldsymbol{h}_t^0)$.
   - With the encoder and the "reconstruction", $\boldsymbol{v}_t^1$, estimate probabilities: $P({\boldsymbol{h}_t}^1|\boldsymbol{v}_t^1) = \Big(P(h_{ti}^1 = 1|\boldsymbol{v}_t^1)\Big)_{i=1}^{|\boldsymbol{h}|}$.

Contrastive Divergence (CD-1) Learning Algorithm (cont.)

- **Parameter Update**
  Based on Gibbs sampling results in the data and the model phases, parameters are updated as follows:

$$W \leftarrow W + \frac{\eta}{|\mathcal{B}|} \sum_{t=1}^{|\mathcal{B}|} \Big(\underbrace{P(\boldsymbol{h}_t^0 | \boldsymbol{v}_t^0) \left(\boldsymbol{v}_t^0\right)^T}_{\text{data}} - \underbrace{P(\boldsymbol{h}_t^1 | \boldsymbol{v}_t^1) \left(\boldsymbol{v}_t{}^1\right)^T}_{\text{model}} \Big),$$

$$\boldsymbol{b}_h \leftarrow \boldsymbol{b}_h + \frac{\eta}{|\mathcal{B}|} \sum_{t=1}^{|\mathcal{B}|} \Big(P(\boldsymbol{h}_t^0 | \boldsymbol{v}_t^0) - P(\boldsymbol{h}_t^1 | \boldsymbol{v}_t^1) \Big),$$

$$\boldsymbol{b}_v \leftarrow \boldsymbol{b}_v + \frac{\eta}{|\mathcal{B}|} \sum_{t=1}^{|\mathcal{B}|} \left(\boldsymbol{v}_t^0 - \boldsymbol{v}_t^1 \right).$$

The CD-1 learning algorithm runs iteratively (for several epochs) until it converges.

Gaussian–Bernoulli RBM for Continuous Input

- Energy function ($\boldsymbol{v}$: real-valued input; $\boldsymbol{h}$: binary-valued hidden state)

$$E(\boldsymbol{v}, \boldsymbol{h}; \Theta) = \sum_{j=1}^{|\boldsymbol{v}|} \frac{(v_j - b_{v,j})^2}{2\sigma_j^2} - \sum_{i=1}^{|\boldsymbol{h}|} \sum_{j=1}^{|\boldsymbol{v}|} W_{ij} h_i \frac{v_j}{\sigma_j} - \sum_{i=1}^{|\boldsymbol{h}|} h_i b_{h,i},$$

  where $\Theta = \{W, \boldsymbol{b}_h, \boldsymbol{b}_v\}$ and σ_j is standard deviation in Gaussian for visible neuron j.

- Probabilistic neurons

$$P_B(h_i = 1|\boldsymbol{v}) = \phi\left(\sum_{j=1}^{|\boldsymbol{v}|} W_{ij} \frac{v_j}{\sigma_j} + b_{h,i}\right)$$

  where $\phi(\cdot)$ is the sigmoid activation function.

$$P_G(v_j = x|\boldsymbol{h}) = \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{\left(x - b_{v,j} - \sigma_j \sum_{i=1}^{|\boldsymbol{h}|} W_{ji} h_i\right)^2}{2\sigma_j^2}\right)$$

Gaussian–Bernoulli RBM for Continuous Input

- Contrastive Divergence (CD-1) Learning Algorithm

  For $t = 1, 2, \cdots, |\mathcal{B}|$, $i = 1, 2, \cdots, |\boldsymbol{h}|$ and $j = 1, 2, \cdots, |\boldsymbol{v}|$ do steps 1 & 2

  1. Data Phase: estimate probabilities with $P_B(h^0_{ti} = 1|\boldsymbol{v}^0_t)$ where $\boldsymbol{v}^0_t = \boldsymbol{x}_t$
  2. Model Phase
     - Form a realisation of h^0_{ti} by sampling with $P_B(h^0_{ti} = 1|\boldsymbol{v}^0_t)$
     - With h^0_{ti}, estimate the "reconstruction": $v^1_{tj} = \sigma_j \sum_{i=1}^{|\boldsymbol{h}|} W_{ji} h^0_{ti} + b_{v,j}$
     - With v^1_{tj}, estimate $P_B(h^1_{ti} = 1|\boldsymbol{v}^1_t)$.

  Parameter Update: $i = 1, 2, \cdots, |\boldsymbol{h}|$ and $j = 1, 2, \cdots, |\boldsymbol{v}|$ do

$$W_{ij} \leftarrow W_{ij} + \frac{\eta}{|\mathcal{B}|} \sum_{t=1}^{|\mathcal{B}|} \left(P_B(h^0_{ti} = 1|\boldsymbol{v}^0_t) \frac{v^0_{tj}}{\sigma_j} - P_B(h^1_{ti} = 1|\boldsymbol{v}^1_t) \frac{v^1_{tj}}{\sigma_j} \right)$$

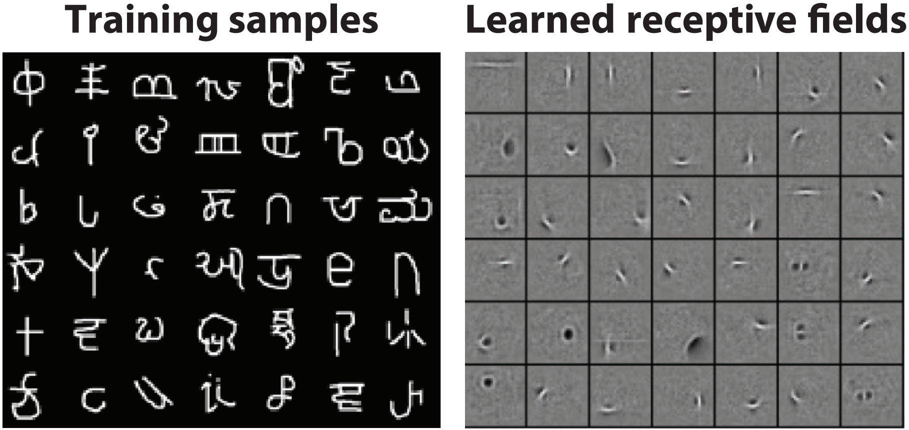
$$b_{h,i} \leftarrow b_{h,i} + \frac{\eta}{|\mathcal{B}|} \sum_{t=1}^{|\mathcal{B}|} \left(P_B(h^0_{ti} = 1|\boldsymbol{v}^0_t) - P_B(h^1_{ti} = 1|\boldsymbol{v}^1_t) \right)$$

$$b_{v,j} \leftarrow b_{v,j} + \frac{\eta}{|\mathcal{B}|} \sum_{t=1}^{|\mathcal{B}|} \left(v^0_{tj} - v^1_{tj} \right)$$

Tip: Each input feature is often standardised to $N(0, 1)$ to avoid setting σ_j.

Illustrative Example

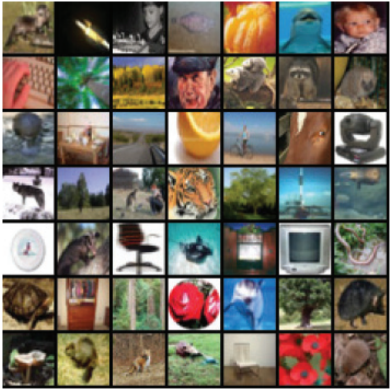
- Handwritten characters: binary-valued RBM of 200 hidden neurons
  28×28 image size (left), 28×28 weights associated with each of 42 randomly chosen hidden neurons (right) that learns receptive fields



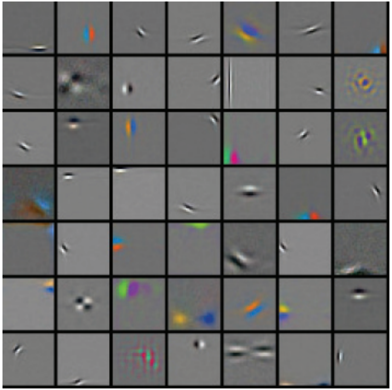
Illustrative Example

- CIFAR-100 image dataset: real-valued RBM of 300 hidden neurons
  32×32 image size (left), 32×32 weights associated with each of 49 randomly chosen hidden neurons (right) that learns receptive fields

Training samples



Learned receptive fields



Reference

If you want to deepen your understanding and learn something beyond this lecture, you can self-study the optional references below.

[Goodfellow et al., 2016] Goodfellow I., Bengio Y., and Courville A. (2016): *Deep Learning*, MIT Press. (Chapter 14, Sections 18.1-18.2 & 20.2)

[Hinton, 2010] Hinton G. (2010): A practical guide to training restricted Boltzmann machines. *Technical Report: UTML TR 2010(003)*, Department of Computer Science, University of Toronto. Online: https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf

[Chen, 2015] Chen K. (2015): Deep and modular neural networks. In *Springer Handbook of Computational Intelligence*, Chapter 28, pp. 473-492. (Sections 28.1-28.2)