Machine Learning for Robotics
Intelligent Systems Series
Lecture 5

Georg Martius

MPI for Intelligent Systems, Tübingen, Germany

May 22, 2017

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

MAX-PLANCK-GESELLSCHAFT

Unsupervised Learning
Dimensionality Reduction – continued

Dimensionality Reduction – reminder

Given: data

$$X = \{x^1, \dots, x^N\} \subset \mathbb{R}^d$$

Dimensionality Reduction – Transductive

Task: Find a lower-dimensional representation

$$Y = \{y^1, \dots, y^N\} \subset \mathbb{R}^n$$

with $n \ll d$, such that Y "represents X well"

Dimensionality Reduction – Inductive

Task: find a function $\phi : \mathbb{R}^d \to \mathbb{R}^n$ and set $y_i = \phi(x_i)$

(allows computing $\phi(x)$ for $x \neq X$: "out-of-sample extension")

Dimensionality Reduction – Overview

Optimizing a cost for parametric transformations:

Model "Y represents X well" as a cost function and optimize for it.

For instance minimize: $\sum_{i=1}^{N} \|x_i - \psi(y_i)\|^2$ where $y = \phi(x_i), \phi : \mathbb{R}^d \to \top^n$ and $\psi : \mathbb{R}^n \to \mathbb{R}^d$.

Optimizing a cost for parametric transformations:

Model "Y represents X well" as a cost function and optimize for it.

For instance minimize: $\sum_{i=1}^{N} \|x_i - \psi(y_i)\|^2$ where $y = \phi(x_i), \phi : \mathbb{R}^d \to \top^n$ and $\psi : \mathbb{R}^n \to \mathbb{R}^d$.

- for linear ϕ, ψ: Principal Component Analysis (PCA)
- for kernelized ϕ: Kernel Principal Component Analysis (KPCA)
- for neural networks for ϕ: Selforganizing Maps (SOM)
- for neural networks for ϕ, and ψ: Autoencoder

Optimizing a cost for parametric transformations:

Model “Y represents X well” as a cost function and optimize for it.

For instance minimize: $\sum_{i=1}^{N} \|x_i - \psi(y_i)\|^2$ where $y = \phi(x_i), \phi : \mathbb{R}^d \to \top^n$ and $\psi : \mathbb{R}^n \to \mathbb{R}^d$.

- for linear ϕ, ψ: Principal Component Analysis (PCA)
- for kernelized ϕ: Kernel Principal Component Analysis (KPCA)
- for neural networks for ϕ: Selforganizing Maps (SOM)
- for neural networks for ϕ, and ψ: Autoencoder

Optimizing a Cost for non-parametric transformations:

For instance minimize: $\sum_{i=1,j=1}^{N} \|\|x_i - x_j\|^2 - \|y_i - y_j\|^2\|^2$ where $y \in \mathbb{R}^n$.

Dimensionality Reduction – Overview

Optimizing a cost for parametric transformations:

Model "Y represents X well" as a cost function and optimize for it.

For instance minimize: $\sum_{i=1}^{N} \|x_i - \psi(y_i)\|^2$ where $y = \phi(x_i), \phi : \mathbb{R}^d \to \top^n$ and $\psi : \mathbb{R}^n \to \mathbb{R}^d$.

- for linear ϕ, ψ: Principal Component Analysis (PCA)
- for kernelized ϕ: Kernel Principal Component Analysis (KPCA)
- for neural networks for ϕ: Selforganizing Maps (SOM)
- for neural networks for ϕ, and ψ: Autoencoder

Optimizing a Cost for non-parametric transformations:

For instance minimize: $\sum_{i=1,j=1}^{N} \|\|x_i - x_j\|^2 - \|y_i - y_j\|^2\|^2$ where $y \in \mathbb{R}^n$.

- Multidimensional Scaling, Local linear Embedding, Isomap

$$U, W = \mathop{\mathbf{argmin}}_{U \in \mathbb{R}^{n \times d}, W \in \mathbb{R}^{d \times n}} \sum_{i=1}^{N} \|x_i - UWx_i\|^2 \qquad \text{(PCA)}$$

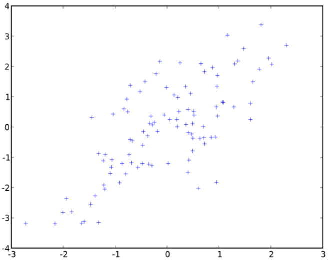
Solution: $U = \big(u_1|u_2|\cdots|u_n\big)$ and $W = U^\top$ with $u_1, \ldots, u_n$: eigenvectors (with largest eigenvalues) of correlation/covariance matrix $cov(X)$.

Principal Component Analysis (PCA) (reminder)

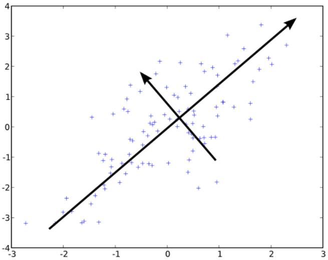
$$U, W = \underset{U \in \mathbb{R}^{n \times d}, W \in \mathbb{R}^{d \times n}}{\textbf{argmin}} \sum_{i=1}^{N} \|x_i - UWx_i\|^2 \qquad \text{(PCA)}$$

Solution: $U = \left(u_1 | u_2 | \cdots | u_n\right)$ and $W = U^\top$ with $u_1, \ldots, u_n$: eigenvectors (with largest eigenvalues) of correlation/covariance matrix $cov(X)$.

Data

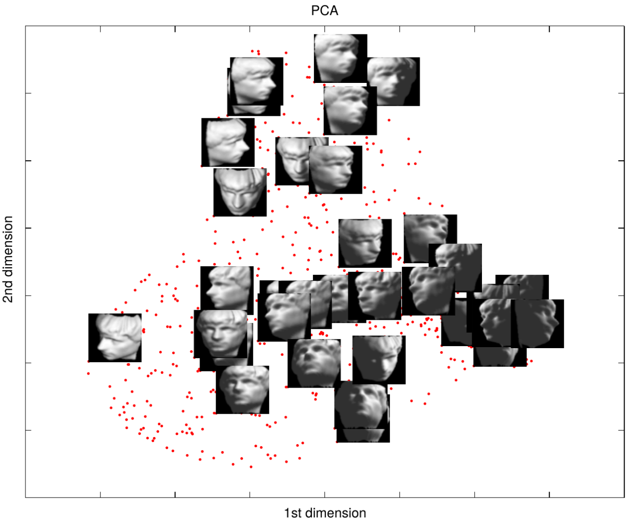


PCA



Images: 64×64
Dim: $n = 4096$
Number: $N = 698$

Different head orientations.



PCA analysis does not correspond to orientation

Kernel-PCA (reminder)

Given samples $x_i \in \mathcal{X}$, kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ with an implicit feature map $\phi : \mathcal{X} \to \mathcal{H}$. **Do PCA in the (implicit) feature space** $\mathcal{H}$.
Kernel trick (reformulation by inner products):

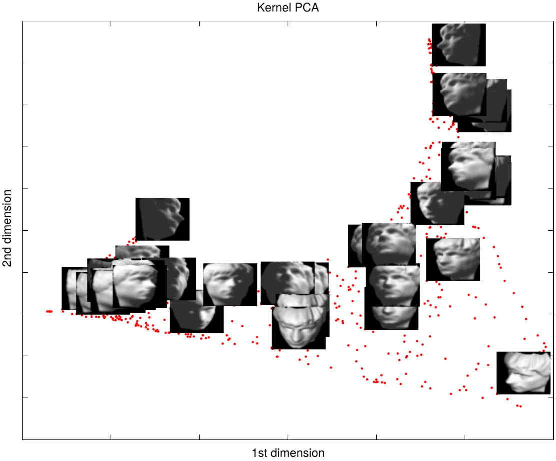
use Eigenvalues of $K_{ij} = \langle \phi(x_i), \phi(x_j) \rangle = k(x_i, x_j)$

Kernel-PCA (reminder)

Given samples $x_i \in \mathcal{X}$, kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ with an implicit feature map $\phi : \mathcal{X} \to \mathcal{H}$. **Do PCA in the (implicit) feature space $\mathcal{H}$.**

Kernel trick (reformulation by inner products):

use Eigenvalues of $K_{ij} = \langle \phi(x_i), \phi(x_j) \rangle = k(x_i, x_j)$

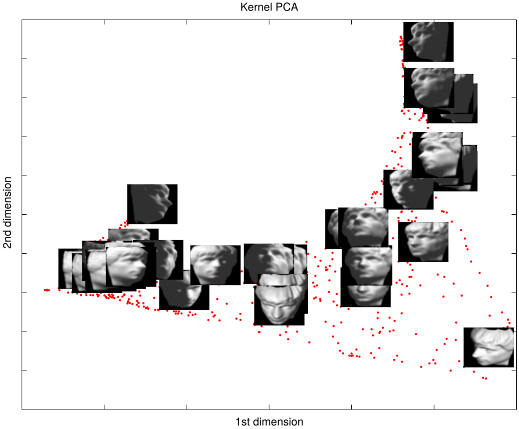


Kernel-PCA (reminder)

Given samples $x_i \in \mathcal{X}$, kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ with an implicit feature map $\phi : \mathcal{X} \to \mathcal{H}$. **Do PCA in the (implicit) feature space** $\mathcal{H}$.

Kernel trick (reformulation by inner products):

use Eigenvalues of $K_{ij} = \langle \phi(x_i), \phi(x_j) \rangle = k(x_i, x_j)$



Kernel-PCA (rbf): Coordinate 1: left-right orientation, 2: brightness

Multidimensional Scaling (MDS)

Given: data $X = \{x^1, \ldots, x^N\} \subset \mathbb{R}^d$

Task: find embedding $y^1, \ldots, y^N \subset \mathbb{R}^n$ that **preserves pairwise distances** $\Delta_{ij} = \|x^i - x^j\|$.

Solve, e.g., by gradient descent on

$$J(y) = \sum_{i<j} \left(\|y^i - y^j\|^2 - \Delta_{ij}^2\right)^2$$

Multidimensional Scaling (MDS)

Given: data $X = \{x^1, \ldots, x^N\} \subset \mathbb{R}^d$

Task: find embedding $y^1, \ldots, y^N \subset \mathbb{R}^n$ that **preserves pairwise distances** $\Delta_{ij} = \|x^i - x^j\|$.

Solve, e.g., by gradient descent on (normalized)

$$J(y) = \frac{1}{\sum_{i<j} \Delta_{ij}^2} \sum_{i<j} \quad (\|y^i - y^j\|^2 - \Delta_{ij}^2)^2$$

Derivative is given by:

$$\frac{\partial J(y)}{\partial y_k} = \frac{2}{\sum_{i<j} \Delta_{ij}^2} \sum_{j \neq k} (\|y^k - y^j\|^2 - \Delta_{kj}^2) \frac{y^k - y^j}{\Delta_{kj}}$$

Multidimensional Scaling (MDS)

Given: data $X = \{x^1, \dots, x^N\} \subset \mathbb{R}^d$

Task: find embedding $y^1, \dots, y^N \subset \mathbb{R}^n$ that **preserves pairwise distances** $\Delta_{ij} = \|x^i - x^j\|$.

Solve, e.g., by gradient descent on (normalized)

$$J(y) = \frac{1}{\sum_{i<j} \Delta_{ij}^2} \sum_{i<j} \quad (\|y^i - y^j\|^2 - \Delta_{ij}^2)^2$$

Derivative is given by:

$$\frac{\partial J(y)}{\partial y_k} = \frac{2}{\sum_{i<j} \Delta_{ij}^2} \sum_{j \neq k} (\|y^k - y^j\|^2 - \Delta_{kj}^2) \frac{y^k - y^j}{\Delta_{kj}}$$

Good starting positions: use first n PCA-projections

MDS is equivalent to PCA for Euclidean distance

Although mathematically very different both methods yield the same result if Euclidean distance is used:
Distance matrix Δ can be written as inner products (kernel matrix)

$$X^\top X = -\frac{1}{2} H \Delta H \quad \text{with } H = \mathbb{I} - \frac{1}{N}\vec{1}\vec{1}^\top$$

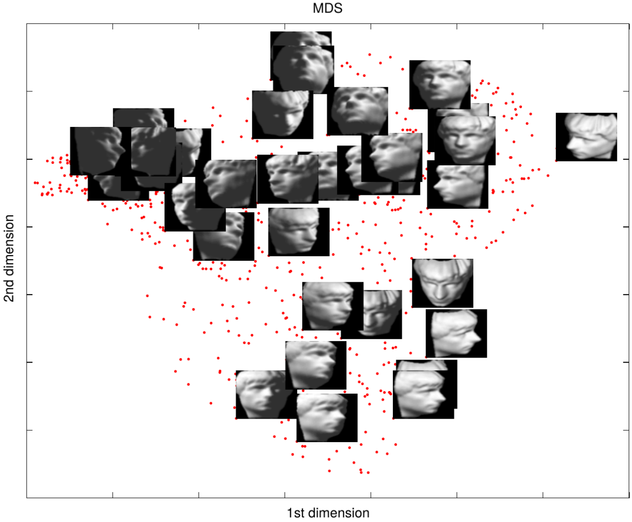
Thus we can rewrite the minimum of J as

$$\operatorname*{\mathbf{argmin}}_Y J(y) = \operatorname*{\mathbf{argmin}}_Y \sum_i \sum_j (x_i^\top x_j - y_i^\top y_j)^2$$

with solution: $Y = \Lambda^{1/2} V^\top$ with Λ: top n eigenvalues of $X^\top X$ and V corresponding eigenvalues, like in PCA.

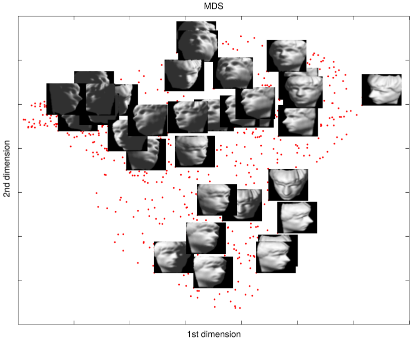
But different distance metrics can be used.

MDS on head-pictures

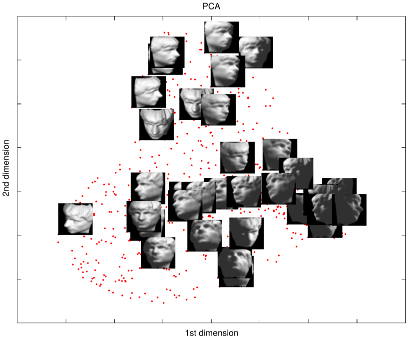


MDS on head-pictures

MDS

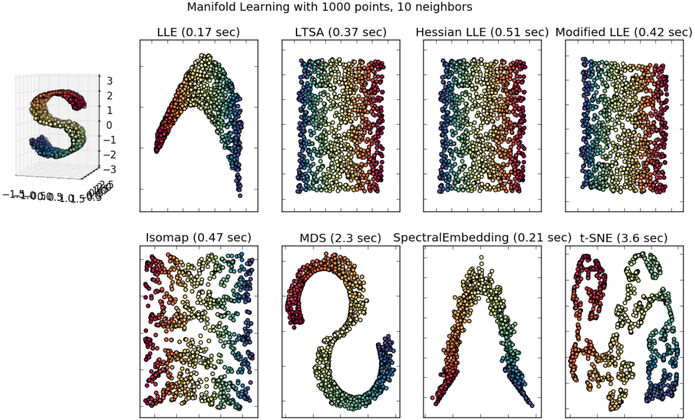


PCA



MDS same as PCA up to sign

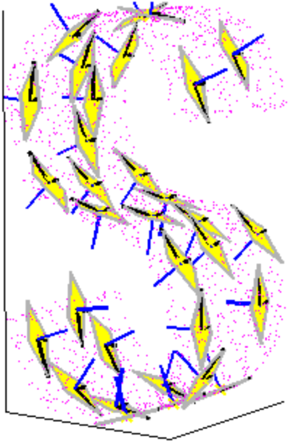
Other methods for dimensionality reduction and manifold learning



Tod

write relation of methods

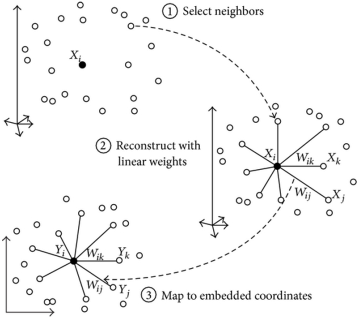
- Assumes that data on a manifold
  ➭ **Locally linear**, i.e. each sample and its neighbors lie on approximately linear subspace
- Idea:
  1. approximate data by a bunch of linear patches
  2. glue patches together on a low dimensional subspace s.t. neighborhood relationships between patches are preserved.



by S.Roweis and L.K. Saul, 2000

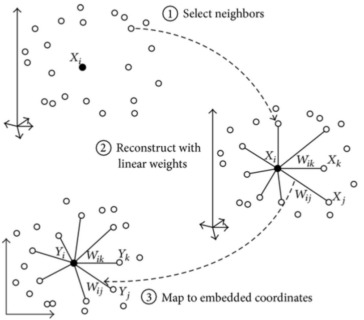
Local Linear Embedding (LLE) – Algorithm

1. identify nearest neighbors B_i for each x_i (either fixed k or fixed radius ϵ)



1. identify nearest neighbors B_i for each x_i (either fixed k or fixed radius ϵ)
2. compute weights to best linearly reconstruct x_i from B_i

$$\min_w \sum_{i=1}^{N} \left\| x_i - \sum_{j=1}^{k} w_{ij} x_{B_i(j)} \right\|^2$$



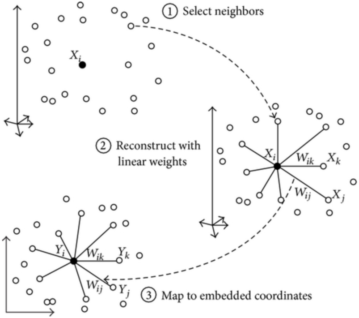
Local Linear Embedding (LLE) – Algorithm

1. identify nearest neighbors B_i for each x_i (either fixed k or fixed radius ϵ)
2. compute weights to best linearly reconstruct x_i from B_i

$$\min_w \sum_{i=1}^{N} \left\| x_i - \sum_{j=1}^{k} w_{ij} x_{B_i(j)} \right\|^2$$

3. Find low-dim embedding vector y_i best reconstructed by weights

$$\min_Y \sum_{i=1}^{N} \left\| y_i - \sum_{j=1}^{k} w_{ij} y_{B_i(j)} \right\|^2$$



Local Linear Embedding (LLE) – Algorithm (continued)

3. Find low-dim embedding vector y_i best reconstructed by weights

$$\min_Y \sum_{i=1}^{N} \left\| y_i - \sum_{j=1}^{k} w_{ij} y_{B_i(j)} \right\|^2$$

Reformulated as:

$$\min_Y \mathbf{Tr}\left(Y^\top Y L\right) \qquad L = (\mathbb{I} - W)^\top (\mathbb{I} - W)$$

3. Find low-dim embedding vector y_i best reconstructed by weights

$$\min_Y \sum_{i=1}^{N} \left\| y_i - \sum_{j=1}^{k} w_{ij} y_{B_i(j)} \right\|^2$$

Reformulated as:

$$\min_Y \mathbf{Tr}\left(Y^\top Y L\right) \qquad L = (\mathbb{I} - W)^\top (\mathbb{I} - W)$$

Solution is arbitrary in origin and orientation and scale.

- constraint 1: $Y^\top Y = \mathbb{I}$ (scale)
- constraint 2: $\sum_i y_i = 0$ (origin at 0)

3. Find low-dim embedding vector y_i best reconstructed by weights

$$\min_Y \sum_{i=1}^{N} \left\| y_i - \sum_{j=1}^{k} w_{ij} y_{B_i(j)} \right\|^2$$

Reformulated as:

$$\min_Y \mathbf{Tr}\left(Y^\top Y L\right) \qquad L = (\mathbb{I} - W)^\top (\mathbb{I} - W)$$

Solution is arbitrary in origin and orientation and scale.

- constraint 1: $Y^\top Y = \mathbb{I}$ (scale)
- constraint 2: $\sum_i y_i = 0$ (origin at 0)
- minimize only with constraint 1:
  ➡ rows of Y are Eigenvalues of L associated with **smallest** Eigenvalues
- Constraint 2 is satisfied if u associated with $\lambda = 0$ is discarded

Local Linear Embedding (LLE) – Algorithm (continued)

3. Find low-dim embedding vector y_i best reconstructed by weights

$$\min_{Y} \sum_{i=1}^{N} \left\| y_i - \sum_{j=1}^{k} w_{ij} y_{B_i(j)} \right\|^2$$

Reformulated as:

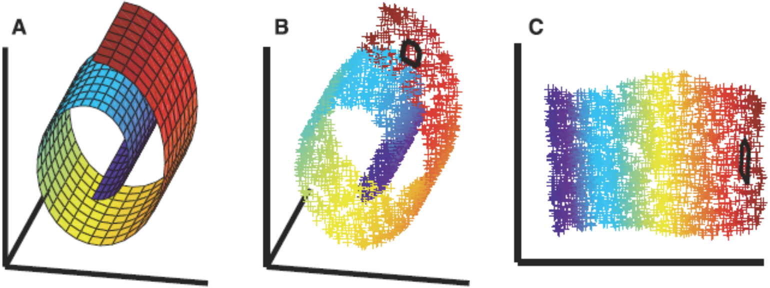
$$\min_{Y} \mathbf{Tr}\left(Y^\top Y L\right) \qquad L = (\mathbb{I} - W)^\top (\mathbb{I} - W)$$

Solution is arbitrary in origin and orientation and scale.

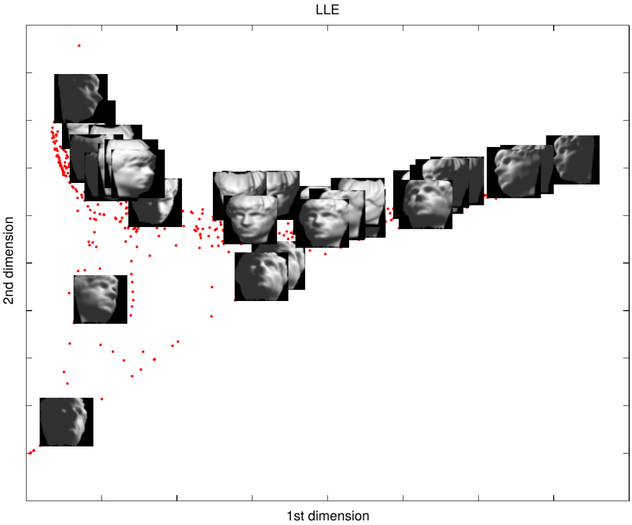
- constraint 1: $Y^\top Y = \mathbb{I}$ (scale)
- constraint 2: $\sum_i y_i = 0$ (origin at 0)
- minimize only with constraint 1:
  ➡ rows of Y are Eigenvalues of L associated with **smallest** Eigenvalues
- Constraint 2 is satisfied if u associated with $\lambda = 0$ is discarded

LLE is global dimensionality reduction while preserving local structure

Local Linear Embedding (LLE) – Example I



Local Linear Embedding (LLE) – Examples



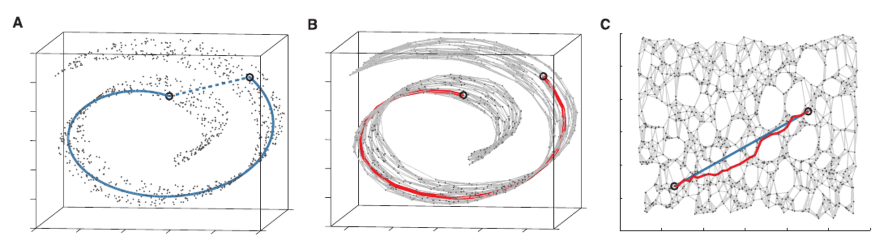
LLE (k=5): Coordinate 1: left-right orientation, 2: ∼ up-down

Isomap (Tenenbaum, de Silva, Langfort 2000)

Main Idea: Perform **MDS on geodesic distances**

Isomap (Tenenbaum, de Silva, Langfort 2000)

Main Idea: Perform **MDS on geodesic distances**

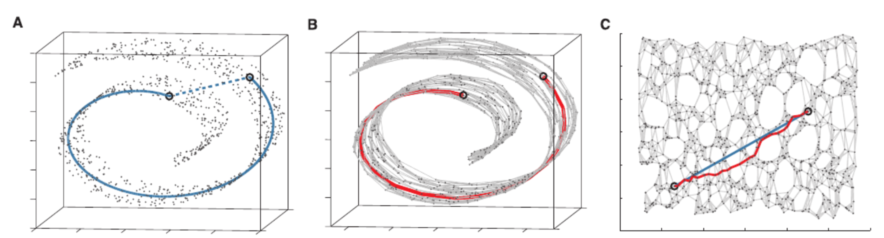


Geodesic: shortest path on a manifold

Isomap – Nonlinear extension of MDS

Isomap (Tenenbaum, de Silva, Langfort 2000)

Main Idea: Perform **MDS on geodesic distances**

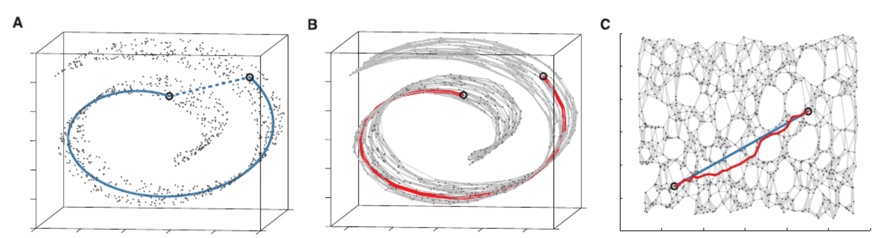


Geodesic: shortest path on a manifold

1. identify nearest neighbors B_i for each x_i
   (either fixed k or fixed radius ϵ)

Isomap (Tenenbaum, de Silva, Langfort 2000)

Main Idea: Perform **MDS on geodesic distances**

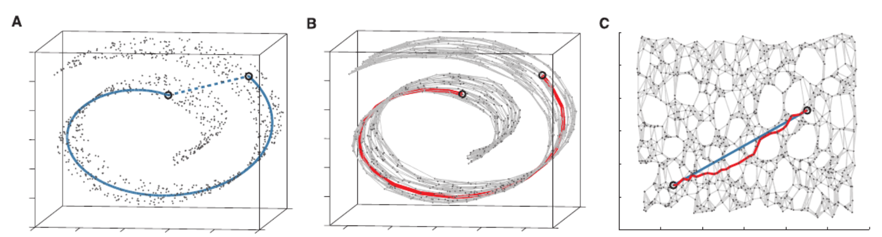


Geodesic: shortest path on a manifold

1. identify nearest neighbors B_i for each x_i (either fixed k or fixed radius ϵ)
2. compute pairwise geodesic distances: shortest paths in nearest neighbor graph

Isomap (Tenenbaum, de Silva, Langfort 2000)

Main Idea: Perform **MDS on geodesic distances**



Geodesic: shortest path on a manifold

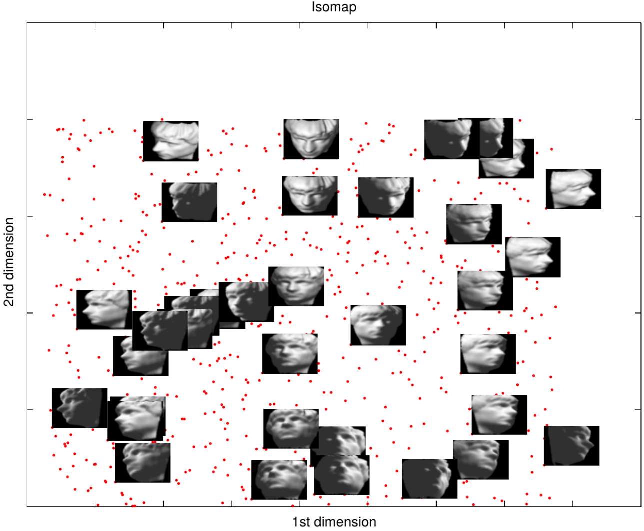
1. identify nearest neighbors B_i for each x_i (either fixed k or fixed radius ϵ)
2. compute pairwise geodesic distances: shortest paths in nearest neighbor graph
3. perform MDS to preserve these distances

Remark: Different than nonlinear forms of PCA

Anecdotal: both papers appeared in *Science* in the same issue!

Tenenbaum: “Our approach [Isomap], based on estimating and preserving global geometry, may distort the local structure of the data. Their technique [LLE], based only on local geometry, may distort the global structure,” he said.

Isomap – Example



Isomap (k=6): Coordinate 1: left-right orientation, 2: up-down

Step 2 of Isomap requires to find all shortest paths.

Floyd–Warshall algorithm

- finds all shortest distances in a graph in $\Theta(|V|^3)$
- dynamic programming solution that iteratively improves current estimates

Step 2 of Isomap requires to find all shortest paths.

Floyd–Warshall algorithm

- finds all shortest distances in a graph in $\Theta(|V|^3)$
- dynamic programming solution that iteratively improves current estimates

Given: Graph with vertices V numbered from $1, \dots, |V|$.
Let $s(i, j, k)$ denote the shortest path from i to j using vertices $\{1, \dots, k\}$

What is $s(i, j, k+1)$**?**

Step 2 of Isomap requires to find all shortest paths.

Floyd–Warshall algorithm

- finds all shortest distances in a graph in $\Theta(|V|^3)$
- dynamic programming solution that iteratively improves current estimates

Given: Graph with vertices V numbered from $1, \ldots, |V|$.
Let $s(i, j, k)$ denote the shortest path from i to j using vertices $\{1, \ldots, k\}$

What is $s(i, j, k+1)$?

1. a path using only vertices $\{1, \ldots, k\}$
2. a path going from i to $k+1$ and from $k+1$ to j

Step 2 of Isomap requires to find all shortest paths.

Floyd–Warshall algorithm

- finds all shortest distances in a graph in $\Theta(|V|^3)$
- dynamic programming solution that iteratively improves current estimates

Given: Graph with vertices V numbered from $1, \ldots, |V|$.
Let $s(i, j, k)$ denote the shortest path from i to j using vertices $\{1, \ldots, k\}$

What is $s(i, j, k+1)$?

1. a path using only vertices $\{1, \ldots, k\}$
2. a path going from i to $k+1$ and from $k+1$ to j

$$s(i, j, k+1) = \mathbf{min}\left(\ s(i, j, k), \quad s(i, k+1, k) + s(k+1, j, k)\ \right)$$

Algorithm evaluates $s(i, j, k)$ for all i, j for $k = 1$, then $k = 2, \ldots, |V|$.

Floyd–Warshall algorithm

Reminder: $s(i,j,k+1) = \mathbf{min}\left(\; s(i,j,k), \quad s(i,k+1,k) + s(k+1,j,k) \;\right)$

input V, $w(u,v)$ (weight matrix)
$s[u][v] = \infty \qquad \forall u,v \in [1,\ldots,|V|]$ minimum distances so far

Floyd–Warshall algorithm

Reminder: $s(i,j,k+1) = \mathbf{min}\left(\ s(i,j,k), \quad s(i,k+1,k) + s(k+1,j,k)\ \right)$

input V, $w(u,v)$ (weight matrix)

$s[u][v] = \infty \qquad \forall u,v \in [1,\ldots,|V|]$ minimum distances so far

for each vertex v

$\quad s[v][v] \leftarrow 0$

for each edge (u,v)

$\quad s[u][v] \leftarrow w(u,v)$

Reminder: $s(i,j,k+1) = \mathbf{min}\left(s(i,j,k), \quad s(i,k+1,k) + s(k+1,j,k) \right)$

input V, $w(u,v)$ (weight matrix)
  $s[u][v] = \infty \quad \forall u,v \in [1,\ldots,|V|]$ minimum distances so far
  for each vertex v
    $s[v][v] \leftarrow 0$
  for each edge (u,v)
    $s[u][v] \leftarrow w(u,v)$
  for k from 1 to $|V|$
    for i from 1 to $|V|$
      for j from 1 to $|V|$
        if $s[i][j] > s[i][k] + s[k][j]$
          $s[i][j] \leftarrow s[i][k] + s[k][j]$

Visualization: https://www.cs.usfca.edu/~galles/visualization/Floyd.html

- Advantages
  - works for nonlinear data
  - preserves global data structure
  - performs global optimization
- Disadvantages
  - works best for swiss-roll type of structures
  - not stable, sensitive to “noise” examples
  - computationally expensive $O(|V^3|)$

Autoencoder

Idea: Use a neural network that learns to **reproduce the input** from a **lower-dimensional intermediate** representation

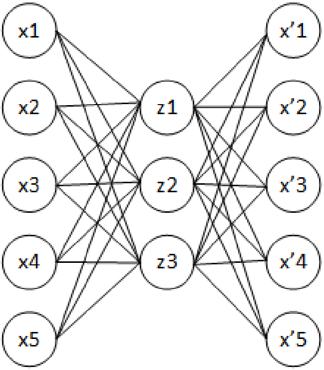
Idea: Use a neural network that learns to **reproduce the input** from a **lower-dimensional intermediate** representation

Self-supervised learning

Input: $x \in \mathbb{R}^d$
Output x
hidden layer $z \in \mathbb{R}^n$ $(n < d)$
(bottleneck)

Encoder: $x \mapsto z$
Decoder: $z \mapsto x$

Trained to minimize reconstruction error.



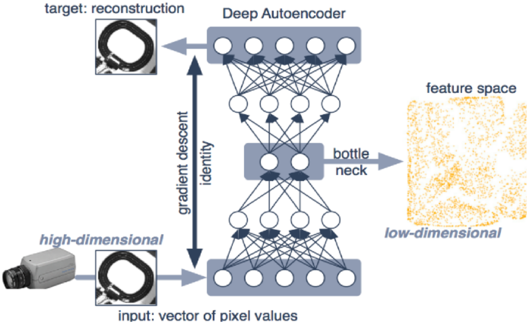
Idea: Use a neural network that learns to **reproduce the input** from a **lower-dimensional intermediate** representation

Self-supervised learning

Input: $x \in \mathbb{R}^d$
Output x
hidden layer $z \in \mathbb{R}^n$ $(n < d)$
(bottleneck)

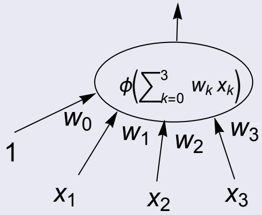
Encoder: $x \mapsto z$
Decoder: $z \mapsto x$

Trained to minimize reconstruction error.



Inspired by biological neurons, but extremely simplified:

Simple artificial Neuron

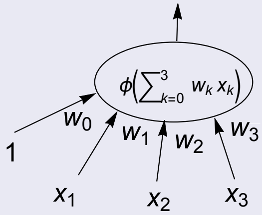


$$\hat{y}_i = \phi\Big(\sum_{j=1}^{d} w_{ij} x_j\Big)$$

$$\phi(z) = \frac{1}{1+e^{-z}} \qquad \text{sigmoid}$$

Inspired by biological neurons, but extremely simplified:

Simple artificial Neuron



$$\hat{y}_i = \phi\Big(\sum_{j=1}^{d} w_{ij} x_j\Big)$$

$$\phi(z) = \frac{1}{1+e^{-z}} \qquad \text{sigmoid}$$

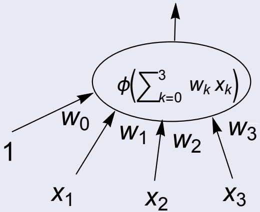
Like in regression problems we use squared error:

$$\mathcal{L}(w) = \frac{1}{2}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

(plus regularization)

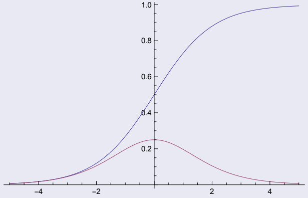
Delta Rule

Perform gradient descent in L: $w^t = w^{t-1} - \epsilon \frac{\partial \mathcal{L}(w)}{\partial w}$



$$\mathcal{L}(W) = \frac{1}{2} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

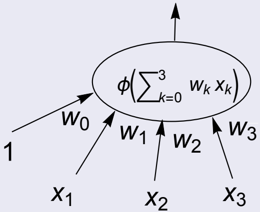
Sigmoid ϕ:



Artificial Neural Networks – a short introduction

Delta Rule

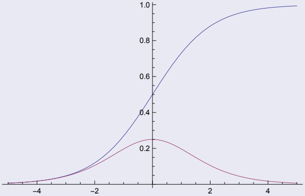
Perform gradient descent in L: $w^t = w^{t-1} - \epsilon \frac{\partial \mathcal{L}(w)}{\partial w}$



$$\mathcal{L}(W) = \frac{1}{2} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

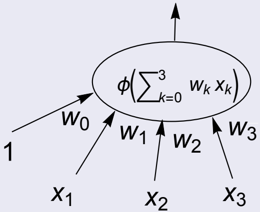
$$\frac{\partial \mathcal{L}(w)}{\partial w} = \underbrace{(\hat{y} - y)}_{\delta} \phi'(z) x$$

Sigmoid ϕ:



Delta Rule

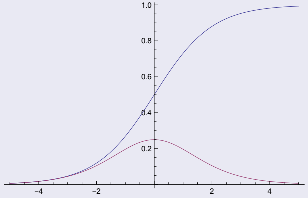
Perform gradient descent in L: $w^t = w^{t-1} - \epsilon \frac{\partial \mathcal{L}(w)}{\partial w}$



$$\mathcal{L}(W) = \frac{1}{2} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \underbrace{(\hat{y} - y)}_{\delta} \phi'(z) x$$

Sigmoid ϕ:

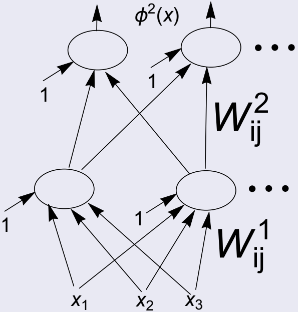


$$\Delta w = -\epsilon \frac{\partial \mathcal{L}(w)}{\partial w}$$

$$w := w + \Delta w$$

Multilayer Network – Backpropagation

Stack layers of neurons on top of each other.

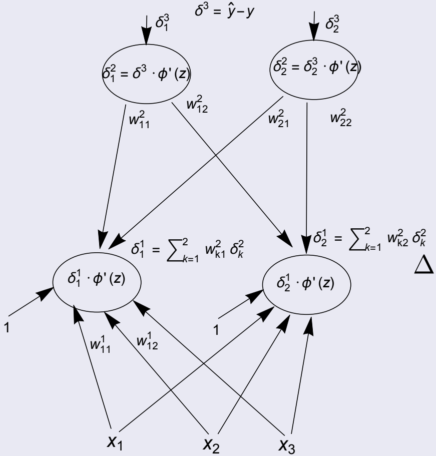


$$\hat{y} = \dots \phi^2(W^2 \phi(W^2 x))$$

$$\mathcal{L}(W) = \frac{1}{2} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

Multilayer Network – Backpropagation

Stack layers of neurons on top of each other.



$$\hat{y} = \ldots \phi^2(W^2 \phi(W^2 x))$$

$$\mathcal{L}(W) = \frac{1}{2} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

$$\Delta W^l = -\epsilon \sum_i^N \delta_i^{l+1} \mathsf{Diag}[\phi'(z_i)](x_i^{l-1})^\top$$

input: x^0, input of layer l: x^{l-1}.

Backpropagation of the error signal:
$$\delta^l = (W^{l+1})^\top \delta^{l+1}$$

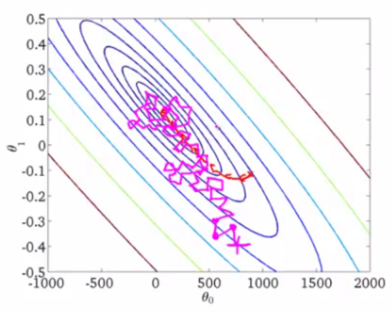
Stochastic gradient descent (SGD)

- Loss/Error is expected empirical error: sum over examples (batch)

- SGD: update parameters on every example:

$$\Delta W^l = -\epsilon \cancel{\sum_i^N} \delta_i^{l+1} \mathrm{Diag}[\phi'(z_i)](x_i^{l-1})^\top$$

- Minibatches: average gradient over a small # of examples



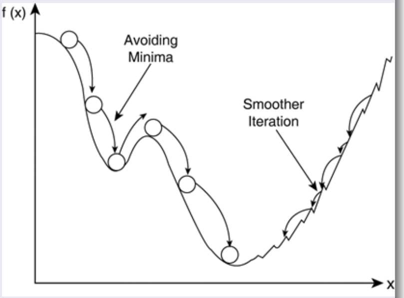
Advantages: many updates of parameters, noisier search helps to avoid flat regions

Momentum

Speed up gradient descent

- Momentum: add a virtual mass to the parameter-particle

$$\Delta W_t = -\epsilon \frac{\partial L(x_t)}{\partial W} + \alpha \Delta W_{t-1}$$

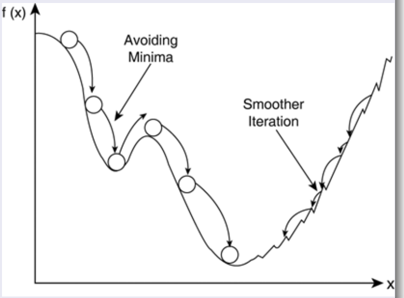


Momentum

Speed up gradient descent

- Momentum: add a virtual mass to the parameter-particle

$$\Delta W_t = -\epsilon \frac{\partial L(x_t)}{\partial W} + \alpha \Delta W_{t-1}$$



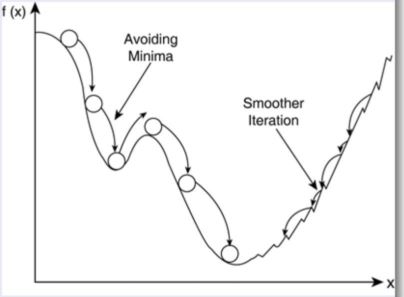
Advantages: may avoids some local minima, faster on ragged surfaces
Disadvantages: another hyperparameter, may overshoot

Momentum

Speed up gradient descent

- Momentum: add a virtual mass to the parameter-particle

$$\Delta W_t = -\epsilon \frac{\partial L(x_t)}{\partial W} + \alpha \Delta W_{t-1}$$



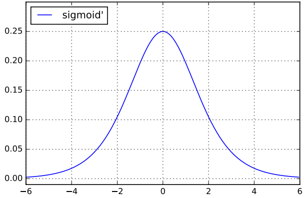
Advantages: may avoids some local minima, faster on ragged surfaces
Disadvantages: another hyperparameter, may overshoot

Adam (2014)

Rescale gradient for each parameter to unit size:

$W_t = W_{t-1} - \epsilon \frac{\langle \nabla W \rangle_{\beta_1}}{\sqrt{\langle (\nabla W)^2 \rangle_{\beta_2}} + \lambda}$ with moving averages: $\langle \cdot \rangle_\beta$

- Derivative of sigmoid vanished for large absolute input (saturation)
- For deep networks (many layers)
  ➡ gradient vanishes



ReLU

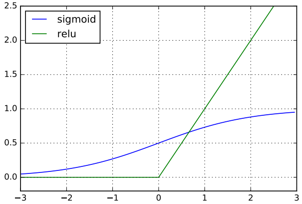
Use a simpler non-linearity:

$$\phi(z) = \mathbf{max}(0, z)$$

CRelu: concatenate positive and negative

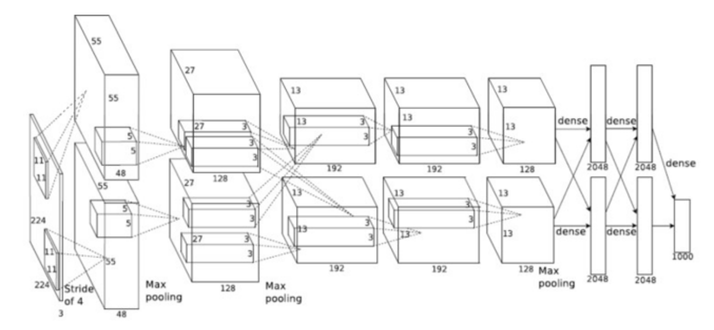
$$\phi(z) = (\mathbf{max}(0, z), -\mathbf{max}(0, -z))$$

Unit-derivative everywhere



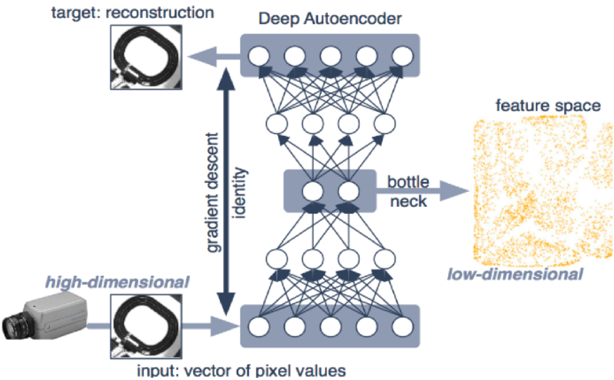
- Trainability and more computer power
  - ➡ larger and deeper networks (>6 layers)
- Breakthrough in performance in many ML applications
  Vision, NLP, Speech,. . .

Convolutionary Network (CNN) – for vision



[Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012]

Back to Autoencoder



- Force a low-dimensional intermediate representation z, with which a good reconstruction can be achieved
- non-linear dimensionality reductions
- But: need to know size of z and sometimes hard to train

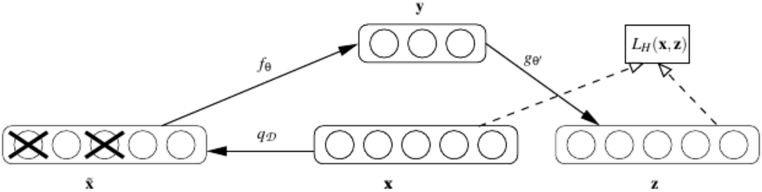
Stacked Denoising Autoencoder

- Idea 1: use a large z but regularize (easier to train)
- Idea 2: make z robust to perturbations (denoising)

Vincent et al, 2010

Input: noise corrupted input $\hat{x}$, target noise free x

$$\mathcal{L}_i = (\phi(\hat{x}_i) - x_i)^2$$



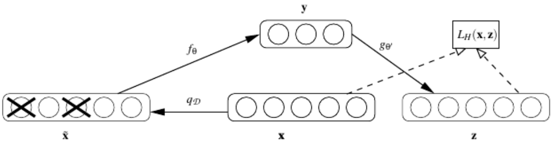
Stacked Denoising Autoencoder

- Idea 1: use a large z but regularize (easier to train)
- Idea 2: make z robust to perturbations (denoising)

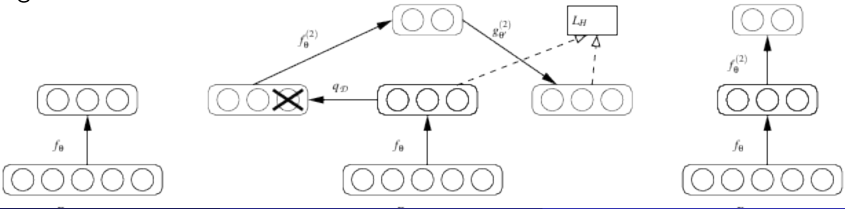
Vincent et al, 2010

Input: noise corrupted input $\hat{x}$, target noise free x

$$\mathcal{L}_i = (\phi(\hat{x}_i) - x_i)^2$$

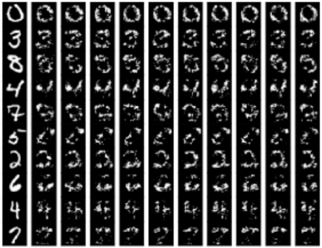


Stacking:



Mnist: generation of samples

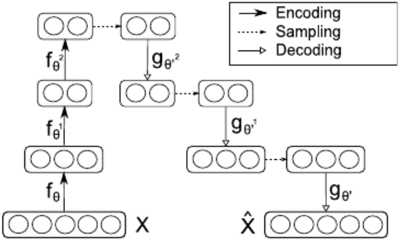
Stacked autoencoder:



Stacked denoising autoencoder:

Sample generation:

- Encode input
- Bernoulli sampling in latent state of each layer



Manifold learning and dimensionality reduction

Summary:

- Linear methods are quite useful already (PCA etc.)
- For nonlinear methods: Isomap and autoencoders are the most useful methods

Dimensionality reduction is important for:

- data visualization
- representation learning
- generative models