Pattern Recognition and Machine Learning

James L. Crowley

ENSIMAG 3 - MMIS Fall Semester 2020
Lessons 5 9 December 2020

Generative Networks: EigenSpace Coding, Auto-Encoders, Variational Autoencoders and Generative Adversarial Networks

Outline:

Background Reading

- Turk, M. and Pentland, A., Eigenfaces for Recognition. *Journal of cognitive neuroscience*, vol. 3, no 1, p. 71-86, 1991.
- Kingma, D.P., Mohamed, S., Rezende, D.J. and Welling, M., Semi-supervised learning with deep generative models. In Advances in neural information processing systems (pp. 3581-3589), NIPS 2014.
- Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks.

Notation

W(i,j)	An RxC image window (Imagette).
$\vec{W}$	A flattened 1-D Vector representing the imagette.
$\{\vec{W}_m\}$	A training set of M imagettes
$\vec{\mu} = E\{\vec{W}_m\}$	Average (mean) imagette
$\vec{V}_m = \vec{W}_m - \vec{\mu}$	Zero-mean normalized imagette
$\mathbf{V} = \begin{pmatrix} \vec{V}_1 & \vec{V}_2 & \cdots & \vec{V}_N \end{pmatrix}$	Training Matrix of Zero-mean imagettes
$\Sigma = \mathbf{V}\mathbf{V}^T$	Covariance of imagettes
ϑ	NxN rotation matrix (N Eigenvectors of Σ)
Λ	Diagonal matrix of eigen values for Σ
x_d	A feature. An observed or measured value.
$\vec{X}$	A vector of D features.
D	The number of dimensions for the vector $\vec{X}$
$\{\vec{X}_m\}$ $\{y_m\}$	Training samples for learning.
M	The number of training samples.
$a_j^{(l)}$	the activation output of the j^{th} neuron of the l^{th} layer.
$w_{ij}^{(l)}$	the weight for the unit *i* of layer *l–1* and the unit *j* of layer *l*.
b_j^l	the bias term for j^{th} unit of layer *l*.
ρ	The sparsity parameter

Key Equations

Principal Components Analysis: $\vartheta^T \Sigma \vartheta = \Lambda$

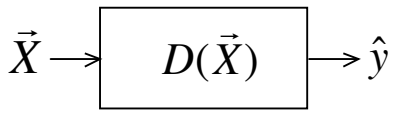
The average activation at layer *l*: $\hat{\rho}_j = \frac{1}{M}\sum_{m=1}^{M} a_{j,m}^{(1)}$

The autoencoder cost function: $L_{sparse}(W, B; \vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(2)} - \vec{X}_m)^2 + \beta \sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$

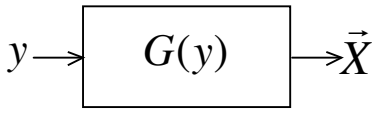
the Kullback-Leibler Divergence: $\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j) = \sum_{j=1}^{N^{(1)}} \left(\rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j} \right)$

Generative Networks

Deep learning was originally invented for recognition. The same technology can be used for generation. Up to now we have looked at what are called "discriminative" techniques. These are techniques that attempt to discriminate a class label, y from a feature vector, $\vec{X}$.



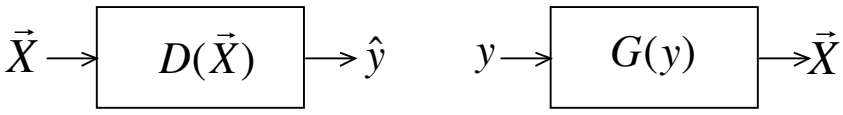
The same process can be used to learn a network that generates $\vec{X}$ given a code y. This is called a "generative" process.



Given an observable random variable $\vec{X}$, and a target variable, gradient descent allows us to learn a joint probability distribution, $P(\vec{X},\vec{Y})$, where $\vec{X}$, is generally composed of continuous variables, and $\vec{Y}$ is generally a discrete set of classes represented by a binary vector.

A discriminative model gives a conditional probability distribution $P(\vec{Y} \mid \vec{X})$.
A generative model gives a a conditional probability $P(\vec{X} \mid \vec{Y})$

We can combine a discriminative process for one data set with a generative process from another and use these to make synthetic outputs.



A classic example is an autoencoder. But before we see how to learn an autoencoder, I would like to review a similar classic technique from computer vision:
Eigen-Space Coding.

Principal Components Analysis and Eigen-Space Coding

Principal Components Analysis (PCA) is a popular method to reduce the number of dimensions in high dimensional feature vectors. In some cases this can provide an important reduction in computing time with little or no impact on recognition rates. It can also be used to determine an orthogonal basis set for highly redundant features, such as the raw pixels in small windows extracted from images.

While PCA is primarily a data compression method for encoding, it has been successfully used as a method for generating features for detection and recognition. An important example occurred in 1991, with the thesis of Mathew Turk at MIT Media Lab. (Turk and Pentland - CVPR 1991). This paper won "best paper" at CVPR and marked the beginning of the use of appearance-based techniques in Computer Vision.

To use the method we require a training set of M face windows (imagettes) $\{W_m\}$ of size R rows and C columns. We will use the set $\{\vec{W}_m\}$ to learn a set of D orthogonal basis images $\vec{\varphi}_d$. These can serve as feature detectors for detection, and recognition.

We can then project an imagette onto basis vectors to obtain a feature vector.

$\vec{X}_m = \varphi^T \vec{W}_m$ where each component is $x_{dm} = \langle W_m, \varphi_d \rangle = \sum_{i=1}^{C} \sum_{j-1}^{R} W_m(i,j)\ \varphi_d(i,j)$

We can reconstruct the imagette by as a weighted sum of basis images.

$$\hat{W}(i,j) = \mu(i,j) + \sum_{d=1}^{D} x_d \varphi_d(i.j)$$

where $\mu(i,j)$ is the "average" imagette from the training data.

The energy of the difference between the original imagette and reconstructed imagette is a measure of similarity of the imagette to a face.

$$\varepsilon_R^2 = \sum_{j=1}^{R} \sum_{i=1}^{C} R(i,j)^2 \quad \text{where } R(i,j) = W(i,j) - \hat{W}(i,j)$$

We can also train a classifier for subsets of face images, for example all images of a particular person. Other recognition techniques are also possible.

Principal Components Analysis of Face Imagettes.

For notation reasons, it is often convenient to map (or flatten) the 2D window *W(i,j)* imagettes onto a 1-D vector $\vec{W}$. This allows us to express using classical vector algebra.

For an imagette of size C columns by R rows,
allocate a vector W(n) with N=R x C coefficients
For any pixel (i,j), compute $n = j*C + i$
Then W(n) = W(i, j).

Principal components analysis is a method to determine a **linear subspace** that is optimal for reconstructing a set of vectors.

Assume a set of M training vectors (imagettes) $\{\vec{W}_m\}$ of N pixels.

The training data are used to compute an orthogonal basis set of D vectors to represent the training set, $\vec{\varphi}_d$ where D ≤ N.

This basis is provided by the principal components of the covariance matrix of $\{\vec{W}_m\}$.

First, normalize the imagettes to zero mean:

Compute the average vector: $\vec{\mu} = E\{\vec{W}_m\}$ that is: $\mu(n) = \frac{1}{M}\sum_{m=1}^{M} W_m(n)$

Normalize to zero mean the training data: $\vec{V}_m = \vec{W}_m - \vec{\mu}$

The covariance matrix is then constructed from the matrix of normalized training imagettes. Compose the matrix **V** as

$$\mathbf{V} = \left(\vec{V}_1, \vec{V}_2, ..., \vec{V}_m\right) = \begin{pmatrix} V_1(1) & V_2(1) & \dots & V_M(1) \\ V_1(2) & V_2(2) & \dots & V_M(2) \\ \vdots & \vdots & \ddots & \vdots \\ V_1(n) & V_2(n) & \dots & V_M(n) \end{pmatrix}$$

V has N lines and M columns. Each column is a training image, $\vec{V}_m$. Each row is a pixel n← (i,j).

The outer product $V \cdot V^T$ is a covariance matrix, Σ:

$$\Sigma = V \cdot V^T$$

This covariance matrix has N x N = N^2 coefficients.

$$\Sigma = \mathbf{V}\mathbf{V}^T = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

The coefficients of $\mathbf{\Sigma}$ are the covariances for the pixels, i and j.

$$\sigma_{ij}^2 = \frac{1}{M} \sum_{m=1}^{M} V_m(i) V_m(j)$$

For small imagettes, the covariance matrix, $\mathbf{\Sigma}$, is easily diagonalized using standard algorithms for diagonalizing large matrices, such as Householder's method. (See numerical recipes in C or any other numerical methods toolkit). This will work for up to 32 x 32 imagettes (covariance matrices of 1024 x 1024).

$(\varphi, \Lambda) \leftarrow \text{PCA}(\mathbf{\Sigma})$. Giving: $\varphi^T \Sigma \varphi = \Lambda$

Where φ, is an NxN rotation matrix and Λ is a diagonal matrix with N non-zero diagonal values.

The N columns of the rotation matrix, φ, are the eigenvectors of $\mathbf{\Sigma}$. These eigenvectors provide an orthogonal set of N normalized vectors, $\vec{\varphi}_n$, that provide a set of orthogonal code vectors for describing the imagettes of size N.

The square root of the diagonal terms of Λ are the eigenvalues, λ_n. The N eigenvalues tell the average energy for each eigenvector for the imagettes in the training set. This can be used to indicate the average squared error that would result from reconstructing an image without the corresponding eigenvector .

Diagonalisation generally work well for matrices up to 1024 x 1024. Thus we can easily use this method for imagettes up to 32 x 32 pixels. Other more exotic algorithms can be used for larger matrices.

The eigenvectors provide an orthogonal basis for the training data. We can project any imagette $\vec{W}$ onto this basis with:

$$\vec{X} = \varphi^T \vec{W} = \sum_{n=1}^{N} W(n) \cdot \vec{\varphi}(n)$$

where the coefficients are $x_d = \sum_{n=1}^{N} W(n) \cdot \varphi_d(n)$ for $d=1,\ldots,D$ with $D \leq N$

This projection acts as a "code" for the imagette. We can reconstruct an imagette this code as a weight sum of the bases plus the average image.

Reconstruction: $\hat{W}(n) = \mu(n) + \sum_{d=1}^{D} x_d \varphi_d(n)$

This reconstruction will only produce imagettes that resemble the training data. Patterns not in the training data will not appear in the reconstruction! This is classically used as a filter to eliminate noise. It can also be used as a pattern detector!

We can determine an error image:

Error image: $R(n) = W(n) - \hat{W}(n)$

Error Energy: $\varepsilon_R^2 = \sum_{n=1}^{N} R(n)^2$

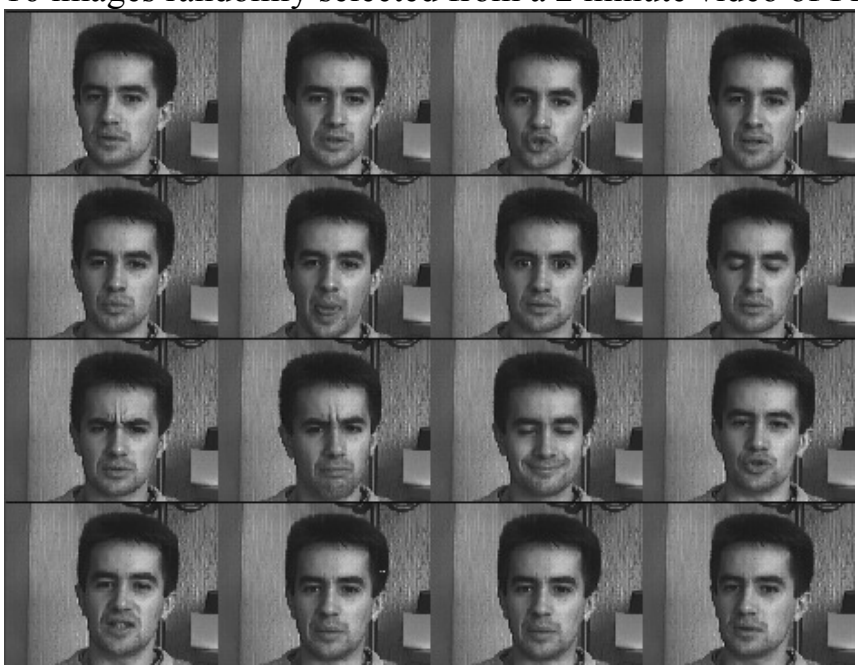
For $x_d = \langle W(n), \varphi_d(n) \rangle = \sum_{n=1}^{N} W(n) \cdot \varphi_d(n)$

The error energy gives an indication of how similar the input imagette is to the training data. The reconstructed image shows where the difference occurs.

When used with a set of face imagettes, the error energy is called the "difference from Face Space".

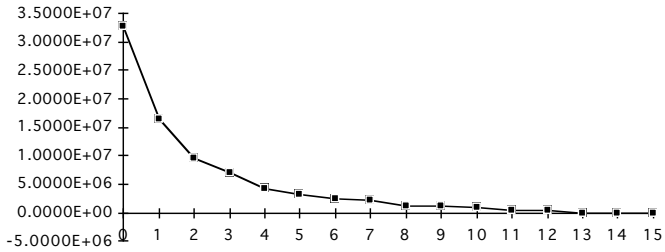
Example

16 images randomly selected from a 2 minute video of Francois Berard. (1995).

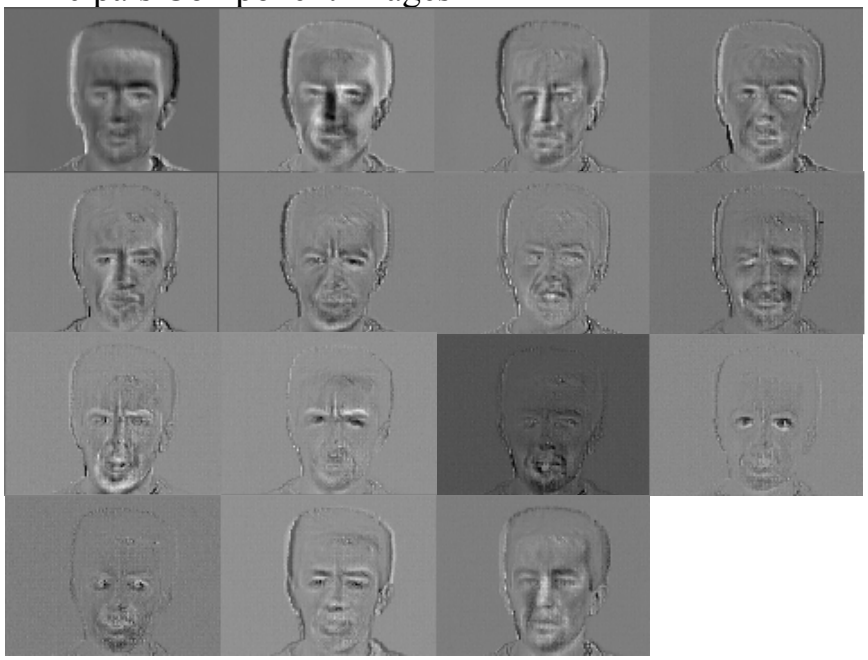


Average Image

Eigen Values



Principals Component images



Note that for this to work well for face images, the images should be normalized in position and scale. This is generally accomplished by aligning the eyes to standard positions.

If the images are note aligned, then the eigenvalues will remain large and a larger code is required.

Reconstruction

The code can be used to reconstruct an image: $\hat{W}(n) = \mu(n) + \sum_{d=1}^{D} x_d \varphi_d(n)$

Example reconstructed image (120 bytes) Error Image.

Reconstruction (120 bytes)

Image Error

Face Detection using Distance from Face Space

The residue image can be used to determine if a new face imagette, W(n), is "similar" to the eigenspace (linear subspace). In this case, the residue is called the "Distance from Face Space" (DFS)

The distance from Face Space can be used as a face detector!

We scan the image with different size windows, texture map each window to a standard size, then computer the residue distance from face space. If the distance is small, the window contains a face similar to the Face space.

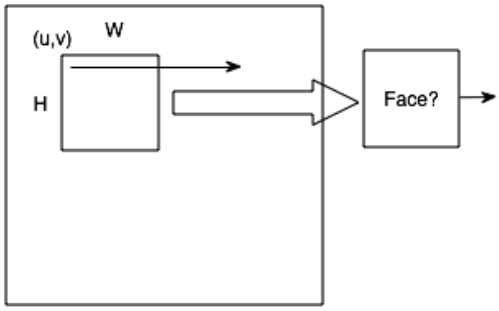
If all N bases used, then any imagette from $\{\vec{W}_m\}$ can be perfectly reconstructed (except for round off error).

Reconstruction: $\hat{W}(n) = \mu(n) + \sum_{d=1}^{D} x_d \varphi_d(n)$

Residue image: $R(n) = W(n) - \hat{W}(n)$

Residue Energy: $\varepsilon_R^2 = \sum_{n=1}^{N} R(n)^2$

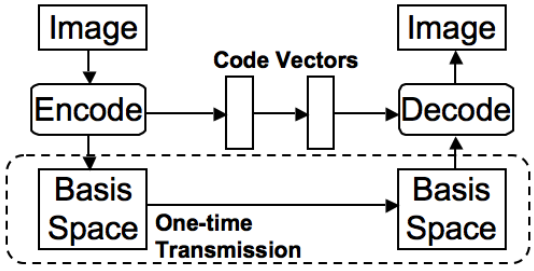
Test if an imagette is a face: if (ε_R^2 < Threshold) THEN Face ELSE NOT Face



In practice, this method is less effective and more expensive than the cascade classifiers seen last lecture.

Eigenspace coding for transmission.

Eigen space coding is a very effective method for signal compression!



In 1996 we were able to demonstrate video telephony in real time (video rates) over a 9600 baud serial line! We ran a video conf with MIT with very low band-width phone line. In this demo, we used 32 coefficients per image! (32 x 4 - 128 bytes/image).

Eigenspace Coding for Face Recognition.

We can use the coefficients $\vec{X} = \langle W(n), \vec{\varphi}(n) \rangle$ as a feature vector for recognition.

This technique typically used with Gaussian Mixture models,
Let us define the parameters for an I component model as $\nu = (\alpha_i, \vec{\mu}_i, \Sigma_i)$
then

$$p(\vec{X};\nu) = \sum_{i=1}^{I} \alpha_i \mathcal{N}(\vec{X};\vec{\mu}_i,\Sigma_i) \quad \text{where : } \mathcal{N}(\vec{X};\vec{\mu},\Sigma) = \frac{1}{(2\pi)^{\frac{D}{2}} \det(\Sigma)^{\frac{1}{2}}} e^{-\frac{1}{2}(\vec{X}-\vec{\mu})^T \Sigma^{-1} (\vec{X}-\vec{\mu})}$$

We use an algorithm such as EM to learn a model ν_k for M_k samples of images $\{W_m^k(n)\}$ of the face for each individual, k, from a population of K individuals.

Given an unknown face, W(n): $\vec{X} = \langle W(n), \vec{\varphi}(n) \rangle = \sum_{n=1}^{N} W(n) \cdot \vec{\varphi}(n)$

from Bayes rule we can determine the most probable individual, C_k as :

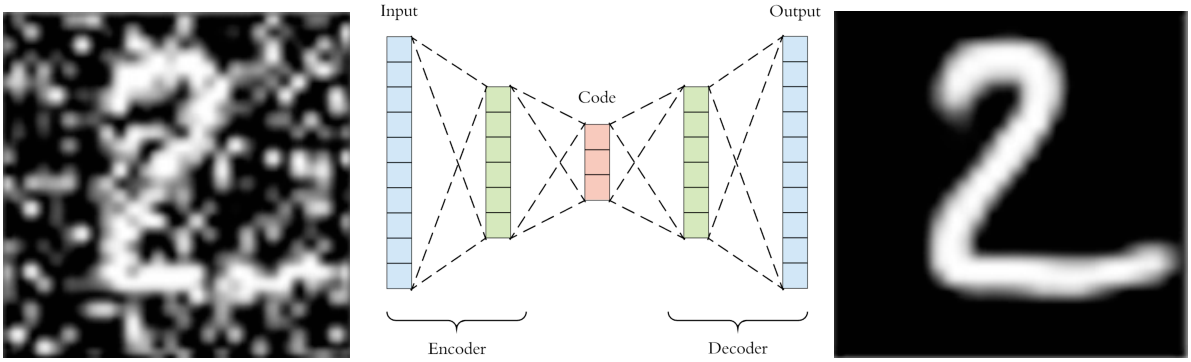
$$P(C_k \mid \vec{X}) = \frac{p(\vec{X} \mid C_k) P(C_k)}{\sum_{k=1}^{k} p(\vec{X} \mid C_k) P(C_k)}$$

In 1991, Eigen-space coding was a revolutionary technique for face recognition, because it was the first technique to actually seem to work!

However, testing soon revealed that it could only work with:
1) Controlled lighting
2) Pre-defined face orientation (i.e. Cooperating Subject)
3) Normalized image position and size.
4) Minimal occlusions
5) Limited size population
In other conditions the results are unreliable.

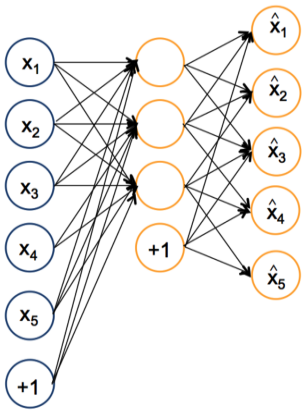
In practice, if there are variations in illumination, these will dominate the first eigenvectors. In this case the corresponding eigenvectors are not useful for recognition and can be omitted.

AutoEncoders



An auto-encoder is an unsupervised learning algorithm that uses back-propagation to learning a sparse set of features for describing the training data. Rather than try to learn a target variable, y_m, the auto-encoder tries to learn to reconstruct the input X using a minimum set of features (latent variables).

An Autocoder learns to reconstruct (generate) clean copies of data without noise. The Key concepts are:
1) The training data is the target. The error is the difference between input and output
2) Training is with standard back-propagation (or gradient descent), with the addition of a "sparsity term" to the loss function
3) Sparsity encodes the data with a minimum number of independent hidden units (Code vectors)



Using the notation from our 2 layer network, given an input feature vector $\vec{X}_m$ the auto-encoder learns $\{w_{ij}^{(1)}, b_j^{(1)}\}$ and $\{w_{jk}^{(2)}, b_k^{(2)}\}$ such that for each training sample, $\vec{a}_m^{(2)} = \hat{X}_m \approx \vec{X}_m$ using as few hidden units as possible.

Note that $N^{(2)} = D$ and that $N^{(1)} << N^{(2)}$

When the number of hidden units $N^{(2)}$ is less than the number of input units, D,

$\vec{a}_m^{(2)} = \hat{X}_m \approx \vec{X}_m$ is necessarily an approximation. The hidden unites provide a "lossy" encoding for $\vec{X}_m$. This encoding can be used to suppress noise!

The error for back-propagation for each unit is a vector $\vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$
with a component $\delta_{i,m}$ for component $x_{i,m}$ of the training sample $\vec{X}_m$

The hidden code is composed of independent "features" that capture some component of the input vector. Each cell of the code vector is driven by a receptive field whose sum of products with the receptive fields of other code cells is almost zero. That is, the code vectors are almost orthogonal. However, rather than minimizing the product of code vectors, sparsity seeks to generate the smallest set of code vectors that can reconstruct the training data without the noise. With an autoencoder the components may have some slight overlap. The average degree of independence is captured by a "sparsity parameter", $\hat{\rho}$.

The Sparsity Parameter
The sparsity $\hat{\rho}_j$ is the average activation for each of the hidden units $j=1$ to $N^{(1)}$.
The auto-encoder will learn weights subject to a sparseness constraints specified by a target sparsity parameter ρ, typically set close to zero.

The simple, 2-layer auto-encoder is described by:

Level 0: $\vec{X}_m = \begin{pmatrix} x_{1,m} \\ \vdots \\ x_{D,m} \end{pmatrix}$ an input vector

level 1: $\vec{Y}_m = a_{j,m}^{(1)} = f(\sum_{i=1}^{D} w_{ij}^{(1)} x_{i,m} + b_j^{(1)})$ the code vector

level 2: $\hat{X}_m = a_{k,m}^{(2)} = f(\sum_{j=1}^{N^{(1)}} w_{jk}^{(2)} a_{j,m}^{(1)} + b_k^{(2)})$ the reconstruction of the input.

The output should approximate the input.

$$\vec{a}_m^{(2)} = \begin{pmatrix} a_1^{(2)} \\ \vdots \\ a_D^{(2)} \end{pmatrix} = \hat{X}_m \approx \vec{X}_m, \quad \text{with error } \vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$$

The sparsity $\hat{\rho}_j$ for each hidden unit (code component) is computed as the average activation for the M training samples:

$$\hat{\rho}_j = \frac{1}{M}\sum_{m=1}^{M} a_{j,m}^{(1)}$$

The auto-encoder is trained to minimize the average sparsity. This is accomplished using back propagation, with a simple tweak to the cost function.

Standard back propagation tries to minimize a loss based on the sum of squared errors. The loss for each sample is.

$$L_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - y_m)^2$$

For an auto-encoder, the target output is the input vector, and the loss is squared difference from the input vector:

$$L_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - \vec{X}_m)^2$$

To impose "sparsity" we add an additional term to the loss.

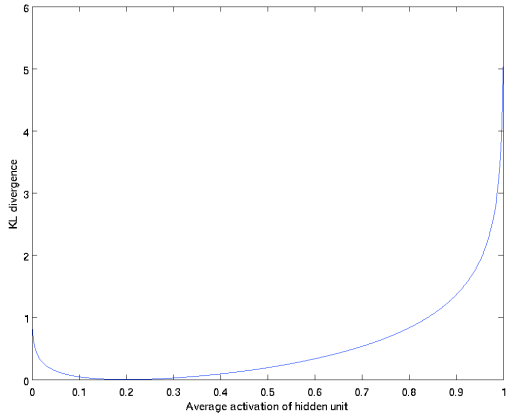
$$L_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - \vec{X}_m)^2 + \beta\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$$

where $\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$ is the Kullback-Leibler Divergence of the vector of hidden unit activations and β controls the importance of the sparsity parameter.

Kullback-Leibler Divergence

The KL divergence between the desired and average activation is:

$$\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j) = \sum_{j=1}^{N^{(1)}} \left(\rho \log\frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j} \right)$$



To incorporate the KL divergence into back propagation, we replace

$$\delta_j^{(1)} = \frac{\partial f(z_j^{(1)})}{\partial z_j^{(1)}} \sum_{k=1}^{N^{(2)}} w_{jk}^{(2)} \delta_k^{(2)}$$

with

$$\delta_j^{(1)} = \frac{\partial f(z_j^{(1)})}{\partial z_j^{(1)}} \left(\sum_{k=1}^{N^{(2)}} w_{jk}^{(2)} \delta_k^{(2)} + \beta \left(-\frac{\rho}{\hat{\rho}_j} + \frac{1-\rho}{1-\hat{\rho}_j} \right) \right)$$

where $N^{(2)} = D$, the size of the size of the input and output vectors.
(The network output has the same number of components as the input).

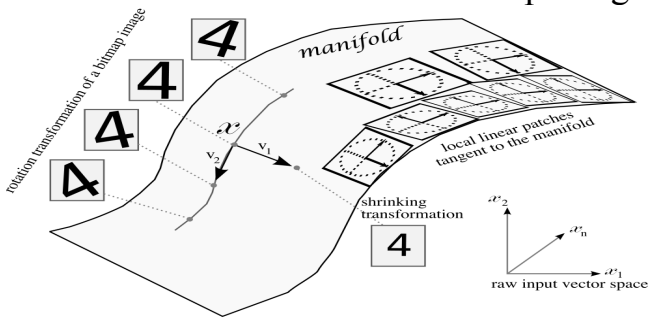
The average activation $\hat{\rho}_j$ is used to compute the correction. Thus you need to compute a forward pass on a batch of training data, before computing the back-propagation. Thus learning is necessarily batch mode.

The auto-encoder forces the hidden units to become approximately orthogonal, allowing a small correlation determined by the target sparsity, ρ. Thus the hidden units act as a form of basis space for the input vectors. The values of the hidden code layer are referred to as latent variables. The latent variables provide a compressed representation that reduces dimensionality and eliminates random noise.

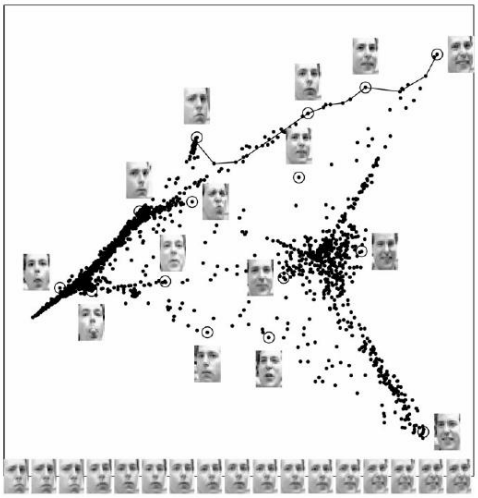
Auto-Encoders vs Principal Components Analysis

What is the difference between an Auto-Encoder and Principle Components analysis? Both techniques project a high-dimensional data set onto a lower dimensional manifold (variété différentielle in french).

Affine Transformations of a Bitmap Image

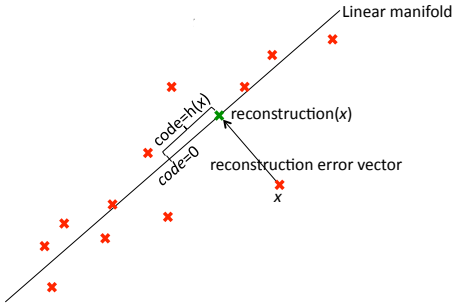


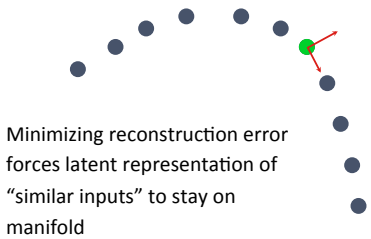
Face Expressions for an individual



(Illustration from the NAACL 2013 lecture from R. Socher and C. Manning)

PCA projects the data onto a high-dimensional linear manifold. AutoEncoders project the data onto a non-linear manifold that (should) provide a better representation of the latent space.





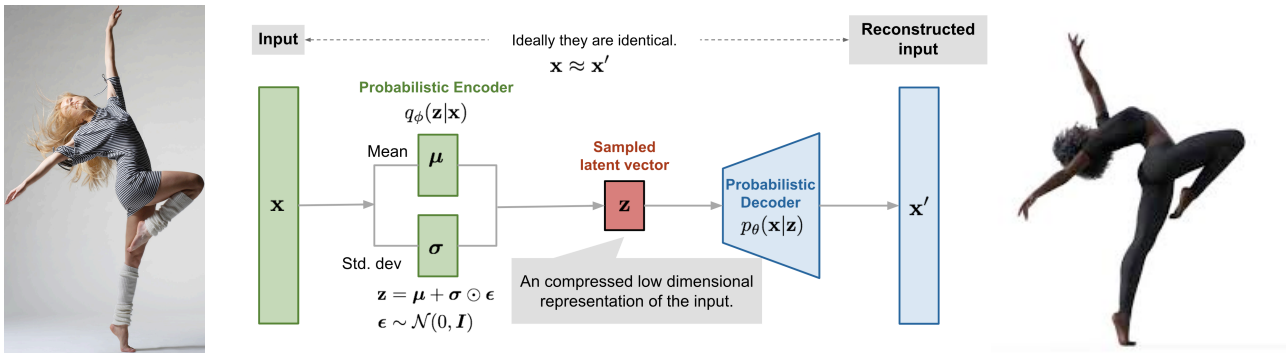
AutoEncoder

So which one works better as a general face detector? Try it and see.

An experimental comparison of face detection using difference from Face Space computed with PCA vs AutoEncoders is work +2 points on the second programming project.

Variational Autoencoders

The output of an auto-encoder can be used to drive a decoder to produce a filtered version of the encoded data or of another training set. However, the output from an auto-encoder is discrete. We can adapt an auto-encoder to generate a *nearly* continuous output by replacing the code with a probabilistic code represented by a mean and variance.



This is called a Variational Autoencoder (VAE). VAEs combine a discriminative network with a generative network. VAEs can be used to generate "deep fake" videos sequences.

For a fully connected network, decoding is fairly obvious. The network input is a binary vector $\vec{Y}$ with k binary values y_k, with one for each target class. This is a code. The output for a training sample $\vec{Y}_m$ is an approximation of a feature vector belonging to the code class, $\hat{\vec{X}}_m$

$$\vec{a}_m^{(2)} = \hat{X}_m \approx \vec{X}_m$$

and the error is the difference between a output and the actual members of the class.

$$\vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$$

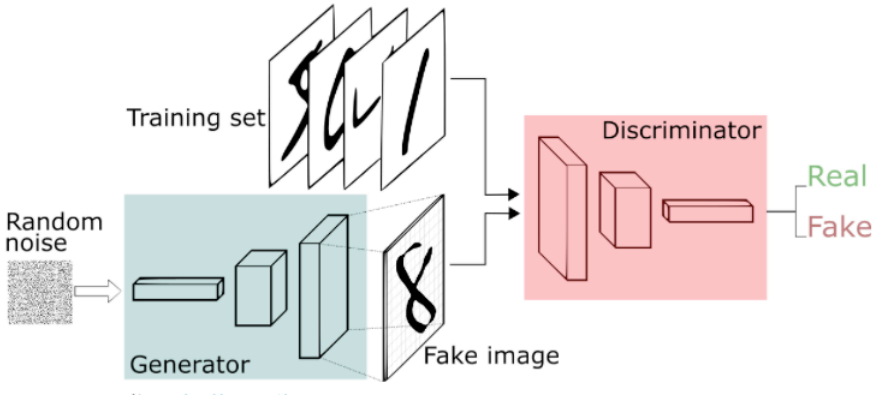
The average error for at training set $\{\vec{Y}_m\}$, $\{\vec{X}_m\}$ can be used to drive back-propagation.

Generative Adversarial Networks.

Generative Networks

It is possible to put a discriminative network together with a generative network and have them train each other. This is called a Generative Adversarial Network (GAN).

A Generative Adversarial Network places a generative network in competition with a Discriminative network.



The two networks compete in a zero-sum game, where each network attempts to fool the other network. The generative network generates examples of an image and the discriminative network attempts to recognize whether the generated image is realistic or not. Each network provides feedback to the other, and together they train each other. The result is a technique for unsupervised learning that can learn to create realistic patterns. Applications include synthesis of images, video, speech or coordinated actions for robots.

Generally, the discriminator is first trained on real data. The discriminator is then frozen and used to train the generator. The generator is trained by using random inputs to generate fake outputs. Feedback from the discriminator drives gradient ascent by back propagation. When the generator is sufficiently trained, the two networks are put in competition.

GAN Learning as Min-Max Optimization.

The generator is a function $\hat{\vec{X}} = G(\vec{z},\theta_g)$, where $G()$ is a differentiable function computed as a multi-layer perceptron, with trainable parameters, θ_g, and z is an input random vector with model $p_z(\vec{z})$, and $\hat{\vec{X}}$ is a synthetic (fake) pattern.

The discriminator is a differentiable function $D(\vec{X},\theta_d)$ computed as a multi-layer perceptron with parameters θ_d that estimates the likelihood that $\vec{X}$ belongs to the set described by the model θ_d.

The generator $\hat{\vec{X}} = G(\vec{z}, \theta_g)$ is trained to minimize $Log(1 - D(G(\vec{z}, \theta_g)))$

The perceptrons $D()$ and $G()$ play a two-player zero-sum min-max game with a value function *V(D, G)*:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

In practice, this may not give sufficient gradient to learn. To avoid this, the discriminator is first trained on real data. The generator is then trained with the discriminator held constant. When the generator is sufficiently trained, the two networks are put in competition, providing unsupervised learning.

The discriminator is trained by ascending the gradient to seek a max:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

The generator is trained by seeking a minimum of the gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$