STA 414/2104

Statistical Methods for Machine Learning and Data Mining

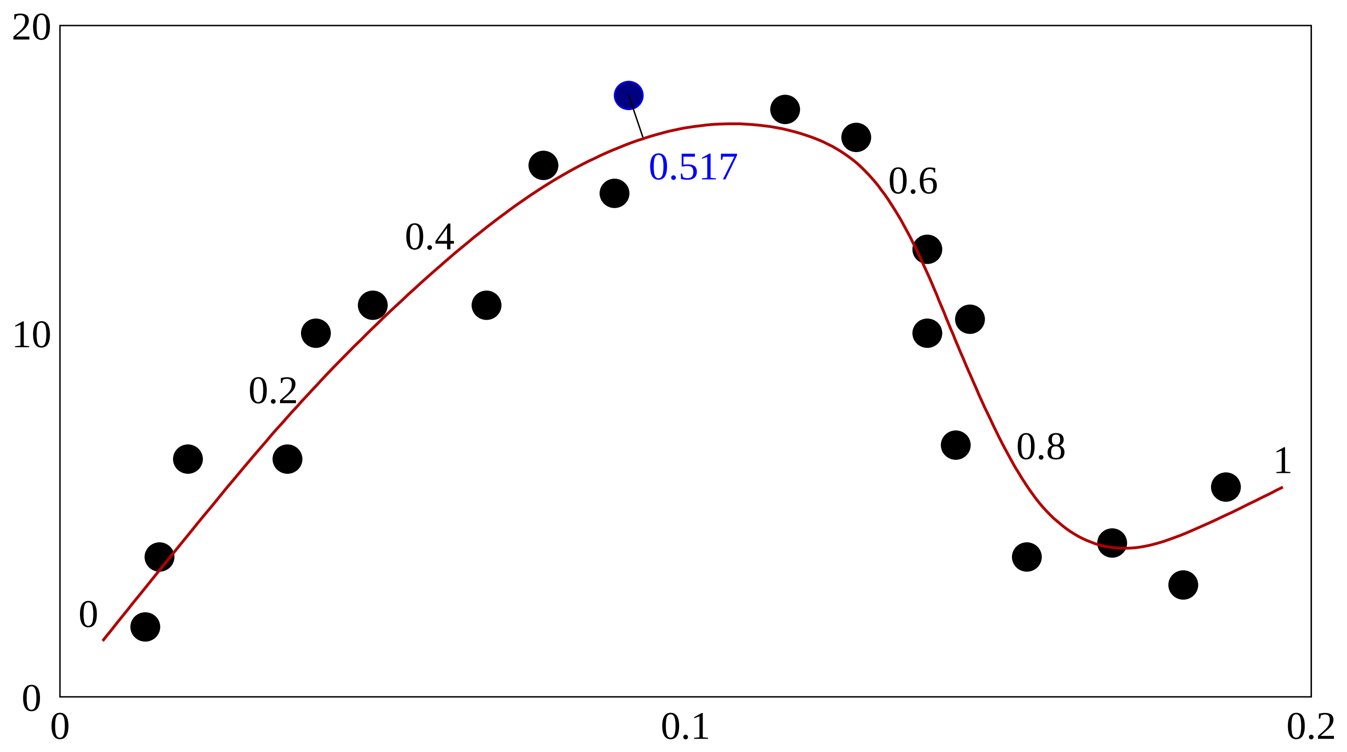
Radford M. Neal, University of Toronto, 2012

Week 11

Dimensionality Reduction

Dimensionality Reduction

High dimensional data is often “really” lower-dimensional: For example:



These points all lie near a curve. Perhaps all that matters is where the points lie on this curve, with the small departures from the curve being unimportant.

If so, we can reduce this 2D data to one dimension, by just projecting each point to the nearest point on the curve. Specifying a point on the curve requires just one coordinate. For example, the blue point at $(0.9, 18)$ is replaced by 0.517.

Manifolds and Embedding

In general, the p-dimensional data points might lie near some M-dimensional surface, or *manifold*.

Points in an M-dimensional manifold can (in each local region) be specified by M coordinates. Eg, points on a sphere can be described by "latitude" and "longitute" coordinates, so the sphere is a 2D manifold.

A p-dimensional "embedding" of the manifold is a map from points on the manifold to p-dimensional space.

Finding an embedding of a lower-dimensional manifold that the data points lie near is one form of unsupervised learning. We'd like to be able to map each point to the coordinates of the point closest to it on the manifold.

Some methods don't really find a manifold and an embedding — they just assign M coordinates to each p-dimensional training case, but don't have any way of assigning low-dimensional coordinates to new test cases. Such methods may still be useful for visualizing the data.

Hyperplanes

In one simple form of dimensionality reduction, the manifold is just a hyperplane.

An M-dimensional hyperplane through the origin can be specified by a set of M basis vectors in p-dimensional space, which are most conveniently chosen to be orthogonal and of unit length.

If $u_1, \ldots, u_M$ are such a basis, the point in the hyperplane that is closest to some p-dimensional data point x is the one with the following coordinates (in terms of the basis vectors):

$$u_1^T x, \ \ldots, \ u_M^T x$$

If we want a hyperplane that doesn't go through the origin, we can just translate the data so that this hyperplane does go through the origin.

Principal Component Analysis

Principal Component Analysis

Principal Component Analysis (PCA) is one way of finding a hyperplane that is suitable for reducing dimensionality.

With PCA, the first basis vector, u_1, points in the direction in which the data has maximum variance. In other words, the projections of the data points on u_1, given by $u_1^T x_1, \ldots, u_1^T x_N$, have the largest sample variance possible, for any choice of unit vector u_1.

The second basis vector, u_2, points in the direction of maximum variance subject to the constraint that u_2 be orthogonal to u_1 (ie, $u_2^T u_1 = 0$).

In general, the i'th basis vector, also called the i'th principal component, is the direction of maximum variance that is orthogonal to the previous $i-1$ principal components.

There are p principal components in all. Using all of them would just define a new coordinate system for the original space. But if we use just the first M, we can reduce dimensionality. If the variances associated with the remaining principal components are small, the data points will be close to the hyperplane defined by the first M principal components.

Finding Principal Components

To find the principal component directions, we first centre the data — subtracting the sample mean from each variable. (We might also divide each variable by its sample standard deviation, to eliminate the effect of arbitrary choices of units.) We put the values of the variables in all training cases in the $n \times p$ matrix X.

We can now express p-dimensional vectors, v, in terms of the eigenvectors, $u_1, \ldots, u_p$, of the $p \times p$ matrix $X^T X$. Recall that these eigenvectors will form an orthogonal basis, and the u_k can be chosen to be unit vectors. I'll assume they're ordered by decreasing eigenvalue. We'll write

$$v \;=\; s_1 u_1 + \cdots + s_p u_p$$

If v is a unit vector, the projection of a data vector, x, on the direction it defines will be $x^T v$, and the projections of all data vectors will be Xv. The sample variance of the data projected on this direction is

$$\begin{aligned}(1/n)(Xv)^T(Xv) \;&=\; (1/n)v^T(X^T X)v \;=\; (1/n)v^T(s_1\lambda_1 u_1 + \cdots + s_p\lambda_p u_p) \\ &=\; (1/n)(s_1^2\lambda_1 + \cdots + s_p^2\lambda_p)\end{aligned}$$

where λ_k is the eigenvalue associated with the eigenvector u_k, and $\lambda_1 \geq \cdots \geq \lambda_p$.

Finding Principal Components (Continued)

We just saw that the sample variance of the data projected in direction of a unit vector, v, is

$$(1/n)(s_1^2\lambda_1 + \cdots + s_p^2\lambda_p)$$

To find the first principal component direction, we maximize this, subject to v being of unit length, so $s_1^2 + \cdots + s_p^2 = 1$. The maximum occurs when $s_1^2 = 1$ and other $s_j = 0$, so that $v = \pm u_1$.

To find the second principal component, we look at unit vectors orthogonal to u_1 — ie, with $s_1 = 0$. The unit vector maximizing the variance subject to this constraint is $\pm u_2$.

Similarly, the third principal component direction is $\pm u_3$, etc.

More on Finding Principal Components

So we see that we can find principal components by computing the eigenvectors of the $p \times p$ matrix $X^T X$, where the $n \times p$ matrix X contains the (centred) values for the p variables in the n training cases. We choose eigenvectors that are unit vectors, of course. The signs are arbitrary.

Computing these eigenvectors takes time proportional to p^3, after time proportional to np^2 to compute $X^T X$.

What if p is big, at least as big as n? Eg, gene expression data from DNA microarrays often has $n \approx 100$ and $p \approx 10000$. Then $X^T X$ is singular, with $p - n + 1$ zero eigenvalues. There are only $n - 1$ principal components, not n.

We can find the $n - 1$ eigenvectors of $X^T X$ with non-zero eigenvalues from the eigenvalues of the $n \times n$ matrix XX^T, in time proportional to $n^3 + pn^2$. If v is an eigenvector of XX^T with eigenvalue λ, then $X^T v$ is an eigenvector of $X^T X$ (not necessarily of unit length), with the same eigenvalue:

$$(X^T X)(X^T v) \ = \ X^T (XX^T) v \ = \ X^T \lambda v \ = \ \lambda (X^T v)$$

So PCA is feasible as long as *either* of p or n is no more than a few thousand.

What is PCA Good For?

Seen as an unsupervised learning method, the results of PCA might be used just to gain insight into the data.

For example, we might find the first two principal components, and then produce a 2D plot of the data. We might see interesting structure, such as clusters.

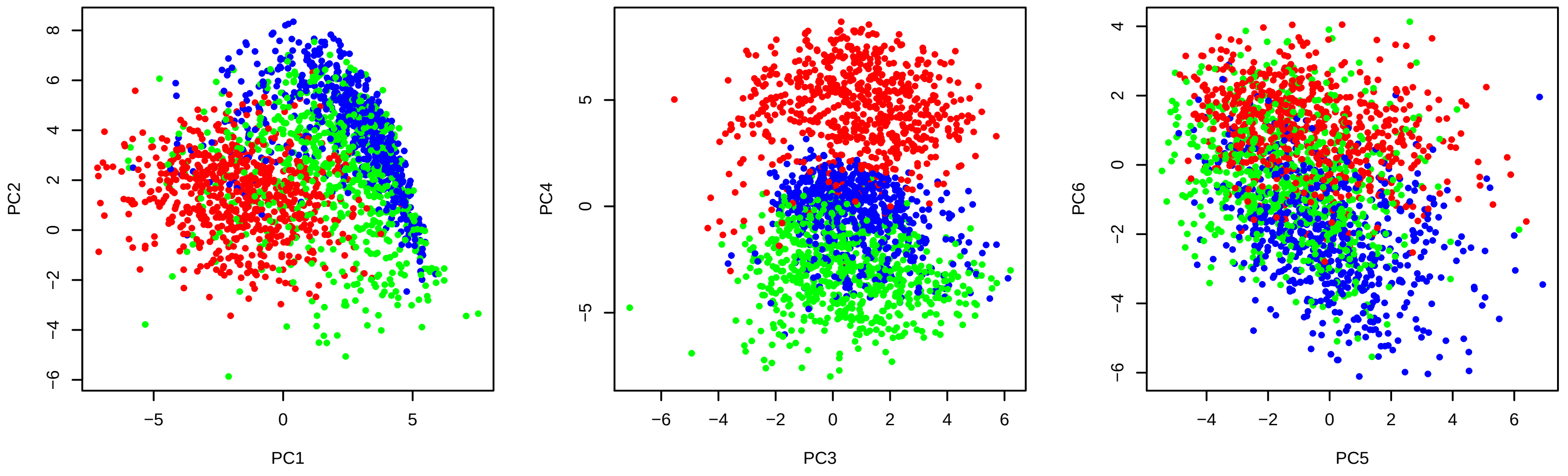
PCA is also used as a preliminary to supervised learning. Rather than use the original p inputs to try to predict y, we instead use the projections of these inputs on the first M principal components. This may help avoid overfitting.
It certainly reduces computation time.

There is no guarantee that this will work — it could be that it is the small *departures* of x from the M dimensional hyperplane defined by the principal components that are important for predicting y.

Example: Zip Code Recognition

I tried finding principal componenents for data on handwritten zip codes (US postal codes). The inputs are pixel values for an 8×8 image of the digit, so there are 256 inputs. There are 7291 training cases.

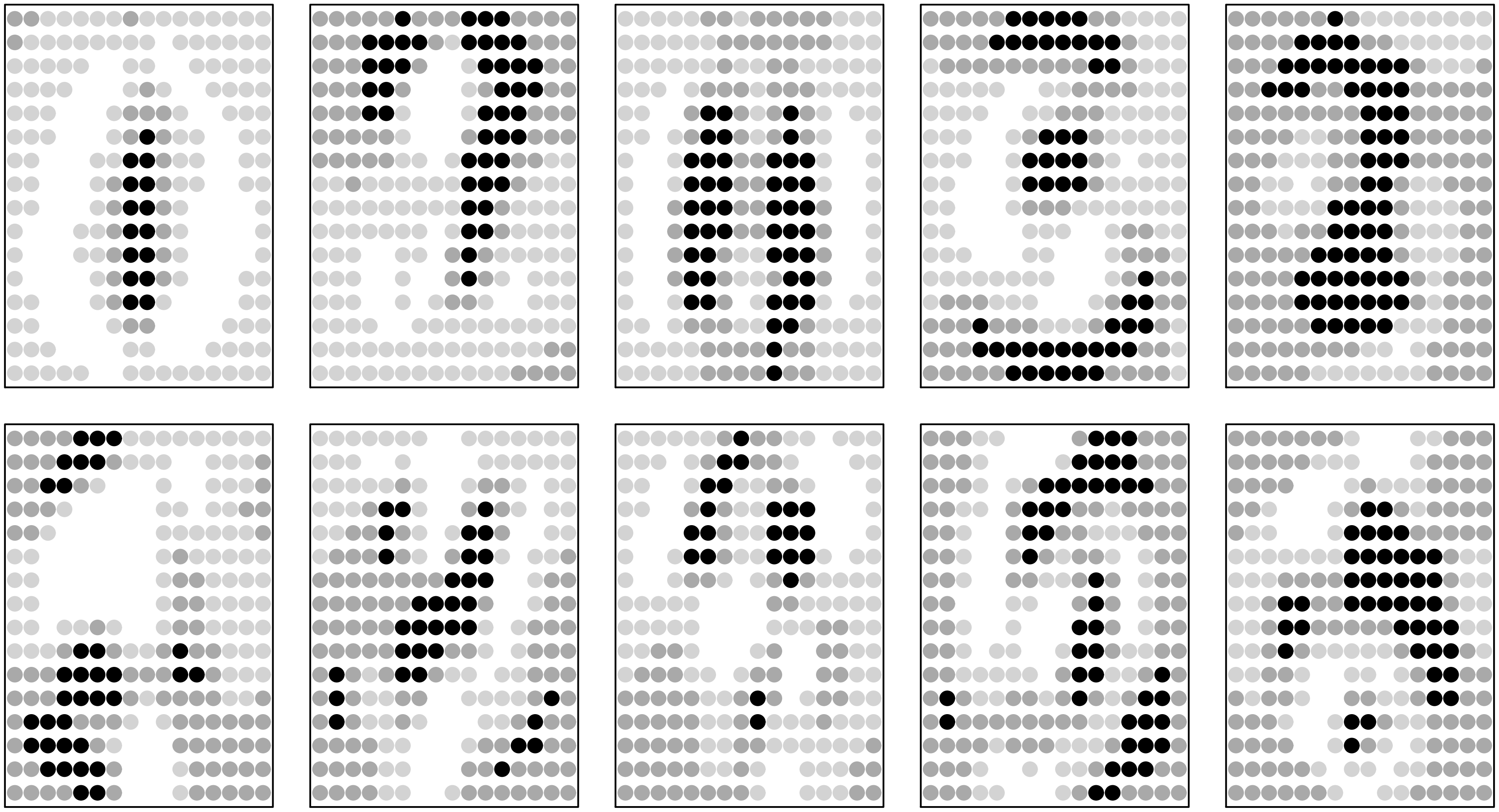
Here are plots of 1st versus 2nd, 3rd versus 4th, and 5th versus 6th principal components for training cases of digits "3" (red), "4" (green), and "9" (blue):



Clearly, these reduced variables contain a lot of information about the identity of the digit — probably much more than we'd get from any six of the original inputs.

Pictures of What the Principal Components Mean

Directions of principal components in input space are specified by 256-dimensional unit vectors. We can visualize them as 16×16 “images”. Here are the first ten:



Factor Analysis

Factor Analysis — A Probabilistic Model Related to PCA

PCA doesn't provide a probabilistic model of the data. If we use $M = 10$ principal components for data with $p = 1000$ variables, it's not clear what we're saying about the distribution of this data.

A latent variable model called *factor analysis* is similar, and does treat the data probabilistically.

We assume that each data item, $x = (x_1, \ldots, x_p)$ is generated using M latent variables $z_1, \ldots, z_M$. the relationship of x to z is assumed to be linear.

The z_i are independent of each other. They all have Gaussian distributions with mean 0 and variance 1. (This is just a convention — any mean and variance would do as well.)

An observed data point, x, is obtained by

$$x \;=\; \mu \;+\; Wz \;+\; \epsilon$$

where μ is a vector of means for the p components of x, W is a $p \times M$ matrix, and ϵ is a vector of p "residuals", assumed to be independent, and to come from Gaussian distributions with mean zero. The variance of ϵ_j is σ_j^2.

The Distribution Defined by a Factor Analysis Model

Since the factor analysis model expresses x as a linear combination of independent Gaussian variables, the distribution of x will be multivariate Gaussian. The mean vector will be μ. The covariance matrix will be

$$\begin{aligned} E\Big((x-\mu)(x-\mu)^T\Big) &= E\Big((Wz+\epsilon)(Wz+\epsilon)^T\Big) \\ &= E\Big((Wz)(Wz)^T \;+\; \epsilon\epsilon^T \;+\; (Wz)\epsilon^T \;+\; \epsilon(Wx)^T\Big) \end{aligned}$$

Because ϵ and z are independent, and have means of zero, the last two terms have expectation zero, so the covariance is

$$E\Big((Wz)(Wz)^T \;+\; \epsilon\epsilon^T\big) \;=\; WE(zz^T)W^T \;+\; E(\epsilon\epsilon^T) \;=\; WW^T + \Sigma$$

where Σ is the diagonal matrix containing the residual variances, σ_j^2.

This form of covariance matrix has $Mp + p$ free parameters, as opposed to $p(p+1)/2$ for a unrestricted covariance matrix. So when M is small, factor analysis is a restricted Gaussian model.

Fitting Factor Analysis Models

We can estimate the parameters of a factor analysis model (W and the σ_j) by maximum likelihood.

This is a moderately difficult optimization problem. There are local maxima, so trying multiple initial values may be a good idea. One way to do the optimization is by applyng EM, with the z's being the unobserved data.

When there is more than one latent factor ($M > 1$), the result is non-unique, since the latent space can be rotated (with a corresponding change to W) without affecting the probability distribution of the observed data.

Sometimes, one or more of the σ_j are estimated to be zero. This is maybe not too realistic.

Factor Analysis and PCA

If we constrain all the σ_j to be equal, the results of maximum likelihood factor analysis are essential the same as PCA. The mapping $x = Wz$ defines an embedding of an M-dimensional manifold in p-dimensional space, which corresponds to the hyperplane spanned by the first M principal components.

But if the σ_j can be different, factor analysis can produce much different results from PCA:

- Unlike PCA, maximum likelihood factor analysis is not sensitive to the units used, or other scaling of the variables.
- Lots of noise in a variable (unrelated to anything else) will not affect the result of factor analysis except to increase σ_j for that variable. In contrast, a noisy variable may dominate the first principle component (at least if the variable is not rescaled to make the noise smaller).
- In general, the first M principal components are chosen to capture as much *variance* as possible, but the M latent variables in a factor analysis model are chosen to explain as much *covariance* as possible.

Other Dimensionality Reduction Methods

Non-Gaussian and Non-linear Latent Variable Models

In factor analysis, the M latent variables, $z_1, \ldots, z_M$, have independent Gaussian distributions, and the relationship of the observed variables, $x_1, \ldots, x_p$, to z is assumed to be linear.

We could change either or both of these assumptions:

Independent Component Analysis (ICA) keeps the linear relationship of x to z, and $z_1, \ldots, z_M$ are still independent, but it assumes that each z_k has *anything but* a Gaussian distribution.

With this change, the non-uniqueness of factor analysis (when $M > 1$) goes away. It turns out that spherical Gaussian distributions are the *only* ones that are rotationally symmetrical and have independent components.

Nonlinear latent variable models may (or may not) keep the Gaussian distribution for z, but they assume that the relationship of x to z may be nonlinear.

Challenges of Nonlinear Latent Variable Modeling

Care is needed to avoid overfitting. Allowing arbitrarily peculiar functions from z to x would fit the training data well, but not result in good predictions, nor in valid insight into the nature of the data.

Bayesian or maximum penalized likelihood methods could be used.

Inverting the model may be computationally difficult. The model may directly specify how x relates to z. But if we observe a new x, we may want to infer what z produced it (or a distribution over possible values for z). This can be difficult, whereas for factor analysis, the distribution of z given x is Gaussian, with a mean that is a linear function of x.

Estimating the parameters can be computationally difficult. If maximum likelihood is used, there may be many local optima.

Auto-Encoder Neural Networks

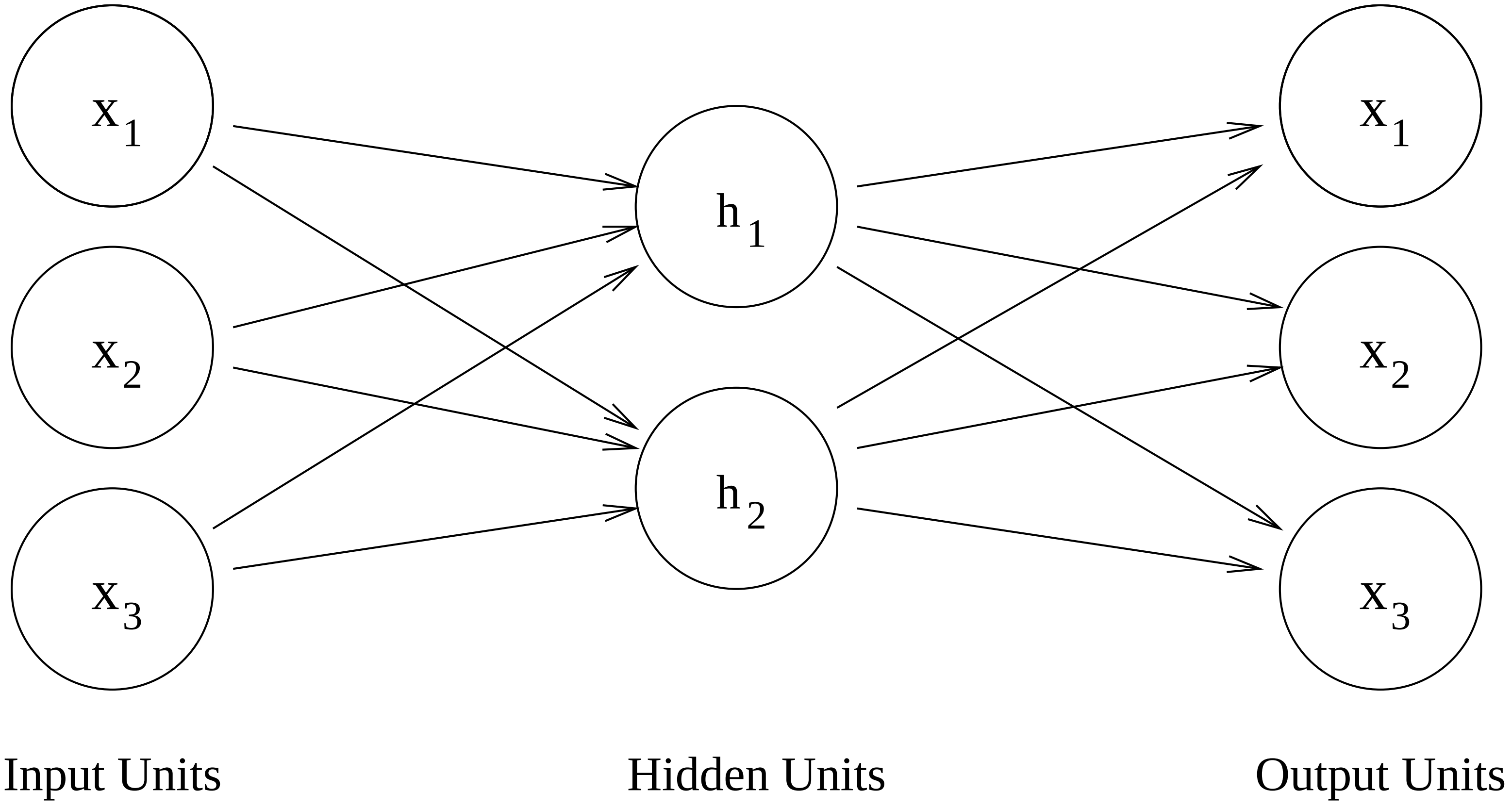
One approach to these problems is to train an *auto-encoder*, which combines a non-linear mapping of the M-dimensional latent variable to the p-dimensional observation with a non-linear mapping from a p-dimensional observation to the M-dimensional latent space.

To make an auto-enoder, we choose some supervised learning procedure — eg, a multilayer perceptron network — but rather than have it predict some response y from x, we instead train it to predict x from x,

Of course, this is trivially easy if no constraint is put on how the predictions can be done. We need to put a"bottleneck" in the model that prevents it from just predicting x perfectly by saying it's equal to x.

A Neural Network for Principal Component Analysis

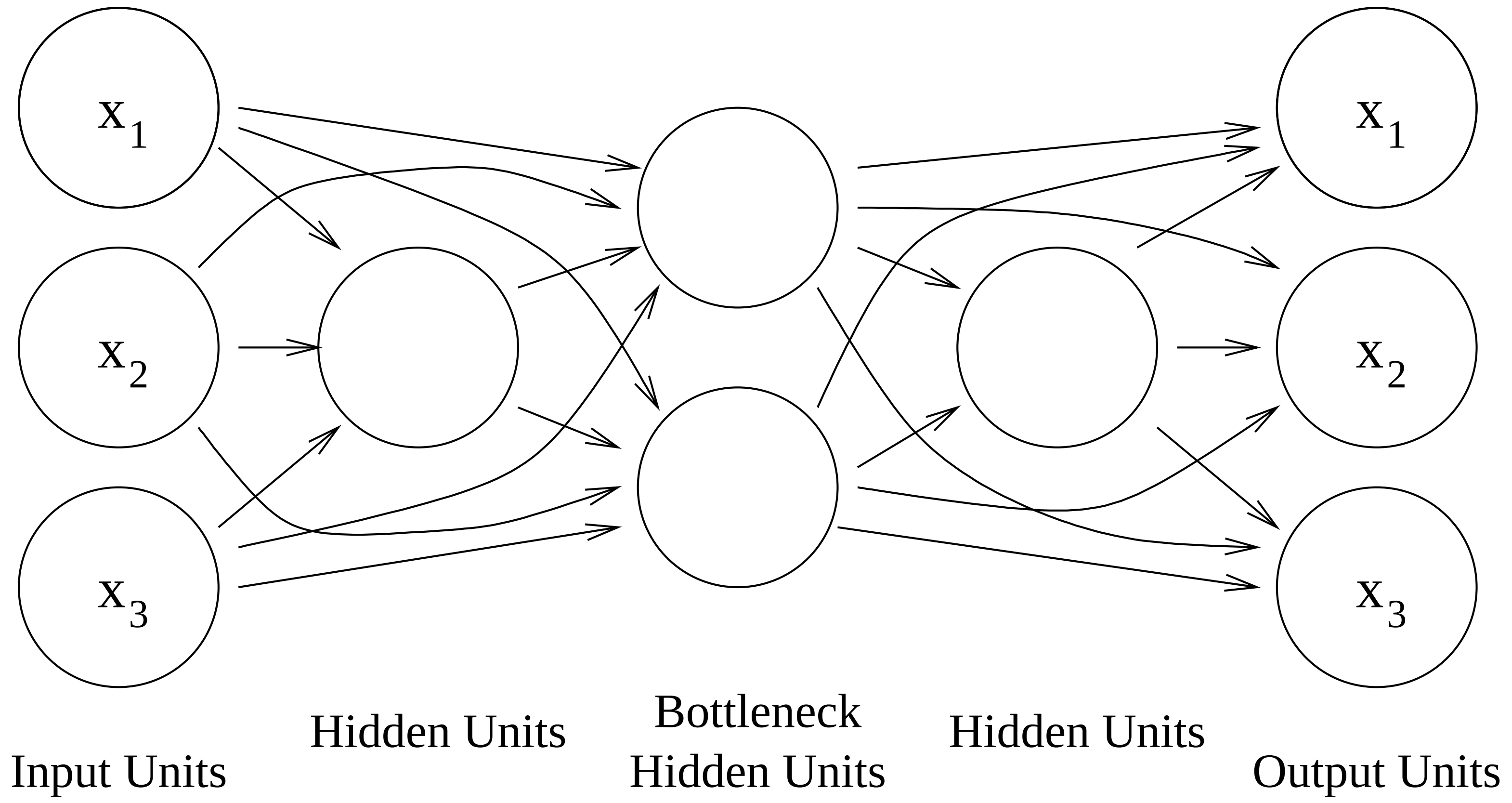
Here's a simple auto-encoder network that finds vectors that span the space of the first M principle components:



We train this network to minimize the sum of the squared reconstruction errors. The hidden units (which have identity activation functions) will compute $E(z|x)$.

Making the Auto-Encoder Nonlinear

We can change this network so that the mappings to and from the bottleneck are nonlinear:



Here, I've put in direct connections from the inputs to the middle hidden layer (the bottleneck) and from the bottleneck to the output units. I show only one hidden unit in the extra hidden layers, but usually there would be more.

Constrained Linear Dimensionality Reduction

Another direction for modifying PCA is to introduce constraints.

Often, we think that data is a non-negative combination of some non-negative patterns. For example, the spectrum of a serum sample is the sum of contributions from the various molecules in the serum. The points in each molecule's spectrum are non-negative, and the amounts of each molecule in the serum are also non-negative.

Non-negative Matrix Factorization finds such a decomposition. It finds a way of writing the $n \times p$ matrix, X, of observed data in the form

$$X \;=\; WH \;+\; E$$

where W is $n \times M$, H is $M \times p$, and E is $n \times p$. W and H are non-negative. The matrix E represents "noise" that isn't accounted for by the factorization. We might aim to minimize the sum of squares of values in E.

This decomposition isn't unique, but may nevertheless provide insight into the data. In contrast, results of PCA may be hard to interpret, since positive and negative components can cancel.