STA 414/2104

Statistical Methods for Machine Learning and Data Mining

Radford M. Neal, University of Toronto, 2014

Week 10

Neural Networks

"Curse of Dimensionality" for Linear Basis Function Models

Modeling a general non-linear relationship of y to x with a linear basis function model seems attractive when x is of low dimension, but when there are many inputs, we would seem to need a huge number of local basis functions to "cover" the high dimensional input space.

This is at least a computational problem — part of the "curse of dimensionality".

We saw, though, that it's actually possible to use an infinite number of basis functions — in the Gaussian process framework.

Another possibility is to use a relatively small number of basis functions, that cover only the actual area where x values are found, which may be the vicinity of a manifold of much lower dimension. We might either:

- pick a subset of data points as centres for basis functions, or
- make the basis functions depend on parameters that adapt to the data.

We'll look now at one class of models of the second type.

"Neural Networks" with One "Hidden Layer"

The term "neural network" can refer to many models that were originally inspired by thoughts on how the brain might work. We'll look here at the most common one — a "multilayer perceptron" network (MLP) with one "hidden layer".

This can be seen as a linear basis function model extended to make the basis functions depend on some additional parameters. (So the model is no longer linear with respect to these additional parameters.)

As before, we model the response y for a case with inputs x as

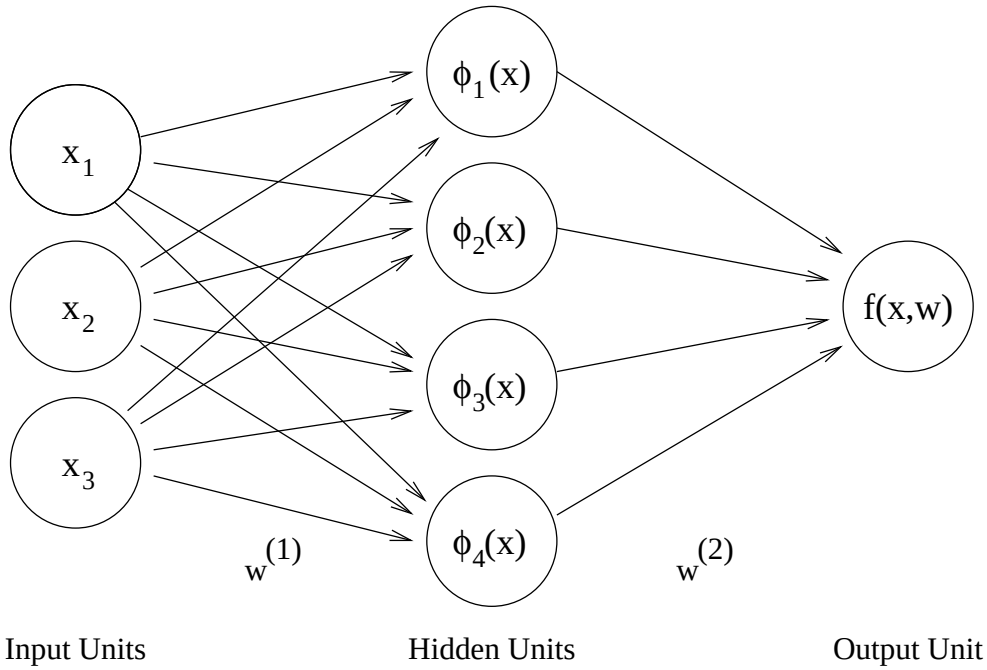
$$y \;=\; f(x, w) \;+\; \text{noise}$$

but now the parameters w — called "weights" — come in two groups:

- A matrix $w^{(1)}$ defines the basis functions.
- A vector $w^{(2)}$ defines a linear combination of these basis functions.

The Architecture of a Multilayer Perceptron Network

A multilayer perceptron with one layer of four "hidden" units looks like this:



Each element of $w^{(1)}$ and $w^{(2)}$ is associated with one of the arrows (connections) above. Each hidden unit computes a value that is a function of a linear combination of the values of units that point into it, with weights given by $w^{(1)}$.

More layers of hidden units can be added — the hidden unit values in one layer are computed from linear combinations of hidden unit values in the previous layer. There could be more than one response variable, in which case $w^{(2)}$ will be a matrix.

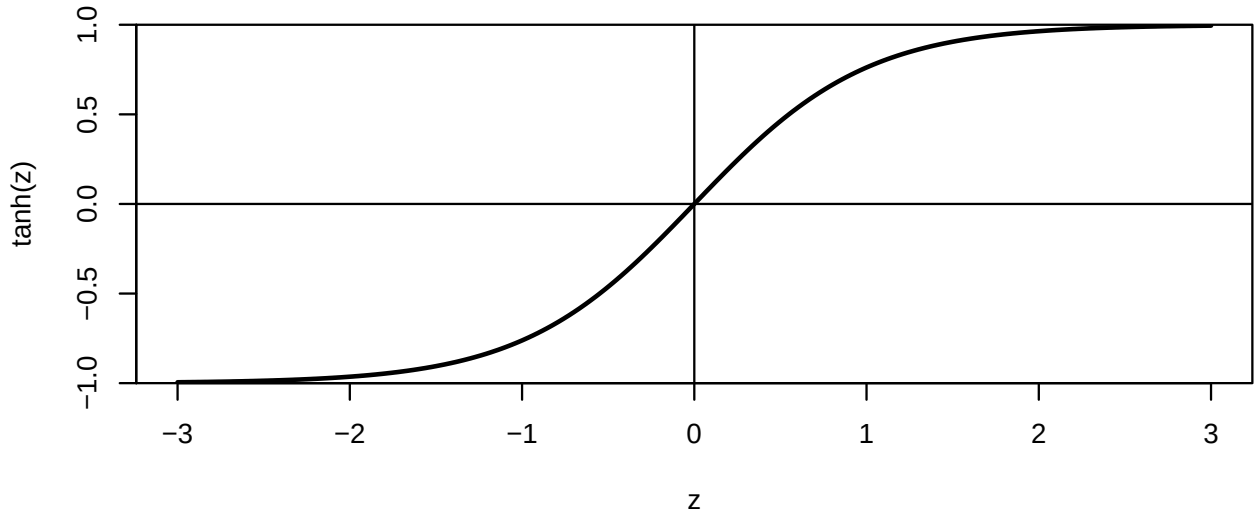
The Function Computed by the Network

The function, $f(x, w)$, computed by a multilayer perceptron network with one hidden layer of m units and one output unit can be written as follows:

$$f(x,w) \;=\; w_0^{(2)} \;+\; \sum_{j=1}^{m} w_j^{(2)} \phi_j(x,w), \qquad \phi_j(x,w) \;=\; h\Big(w_{0j}^{(1)} \;+\; \sum_{k=1}^{p} w_{kj}^{(1)} x_k\Big)$$

The network can approximate any function (better as m increases) if the *activation function*, $h(a)$, is any non-polynomial function.

A traditional choice of activation function (which I'll assume in later slides) is $\tanh(a) = (e^a - e^{-a}) \,/\, (e^a + e^{-a})$. It looks like this:



Training the Network by Maximum Likelihood

The simplest training procedure for an MLP network is to simply adjust the parameters (w) to maximize some measure of fit to the training data. The most obvious measure of fit is likelihood.

For a regresson problem, we may model the target values in training cases, $y_1, \ldots, y_n$, as being independent (given $x_1, \ldots, x_n$ and w), with responses distributed as $y_i \sim N(f(x_i, w), \sigma^2)$, for some noise variance σ^2.

The log of the likelihood (ignoring terms not involving w) is then

$$-\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - f(x_i, w))^2$$

So maximum likelihood estimation for w just minimizes total squared error. But this is a complicated function of w, with no analytical way to find the maximum.

In real problems, the dimension of w, which is $m(p+2)+1$, may be in the hundreds, thousands, tens of thousands, ...

There usually are many local maxima, and finding a global maximum is hopeless. Fortunately, good results are often obtained from fairly good local maxima.

Multiple Ways of Fitting the Data with an MLP

We may not need to find the global maximum of the likelihood because an MLP can fit the data in multiple ways — some producing identical results, others producing nearly identical results.

First, there are exact symmetries:

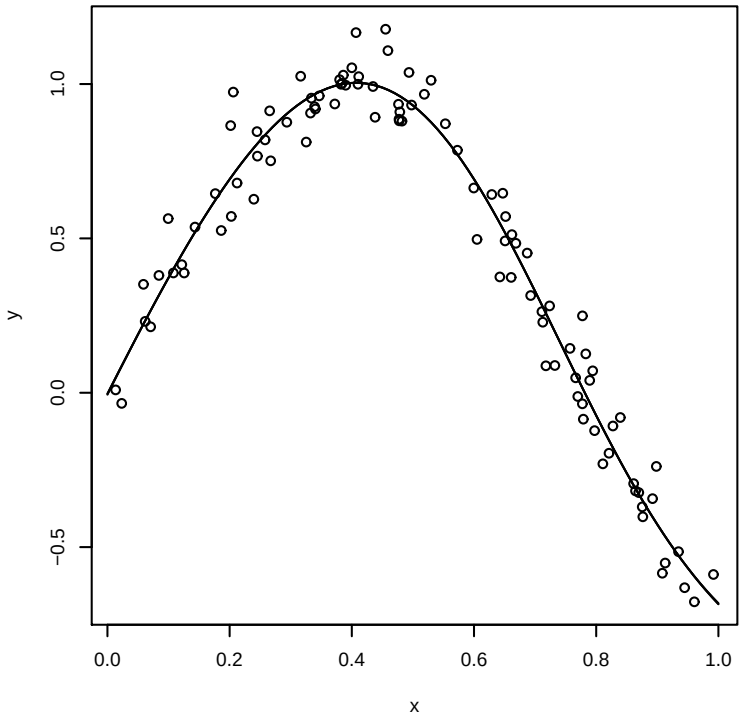
- Permuting the order of hidden units.
- For any hidden unit, j, negating $w_j^{(2)}$ and $w_{kj}^{(1)}$ for all k (including 0).

There are also similar, but different, ways to fit the data.

Suppose we have 10 hidden units, but 3 hidden units are enough to roughly fit the data. The other 7 hidden units could fit various different slight wiggles, with any of these overall fits being fairly good.

Example of Fitting a Network

Here are 100 training points (circles) that I artificially generated as $y = \sin(4x) + \text{noise}$, and the functions computed by three MLP networks with three hidden units found by fitting this data, with different starting points of the optimization procedure. The three fits are indistinguishable. The biggest difference in predicted value over the range $(0, 1)$ is about 0.003.



Here are the parameter values for the three fits:

$w_{11}^{(1)}$	$w_{12}^{(1)}$	$w_{13}^{(1)}$	$w_{01}^{(1)}$	$w_{02}^{(1)}$	$w_{03}^{(1)}$	$w_1^{(2)}$	$w_2^{(2)}$	$w_3^{(2)}$	$w_0^{(2)}$
-1.807	1.422	-2.923	-0.159	-0.224	2.052	-1.565	1.162	1.894	-1.829
-1.906	3.018	-0.218	0.005	-2.132	0.436	-2.140	-1.690	-0.530	-1.418
2.924	0.621	1.906	-2.078	-0.435	0.004	-1.867	0.885	1.987	-1.459

Iteratively Finding the MLP Parameters that Maximize the Likelihood

There is no analytical way of finding the parameters, w, of a multilayer perceptron network that maximize the likelihood, so iterative methods are used.

To start, we set all the network parameters to small random values — eg, values drawn uniformly from $[-0.1, +0.1]$. (Except we might set $w_0^{(2)}$ to the mean of y in the training cases, so it won't be far off if this mean is large.)

Setting all the $w_j^{(2)}$ and $w_{kj}^{(1)}$ to zero would not work, since the hidden units are then all identical, and an iterative method that treats them symmetrically could never make them non-identical.

We stop trying to increase the likelihood when either (a) the likelihood seems to be at a local maximum, or (b) it seems that trying to increase the likelihood further would actually make performance on test cases worse, or (c) our patience runs out. For (b), we need to use a held-out set of "validation" cases.

Optimizing neural network parameters may be quick for simple problems, but for the most difficult problems, network training can take days (or even weeks).

Three Approaches to Optimization

We'll see that it's easy to compute the partial derivatives of the log likelihood. Second derivatives can also be computed.

Using these derivatives, several iterative approaches to maximizing the log likelihood are possible:

- Newton and quasi-Newton methods — use the matrix of second derivatives (or estimates) to approximate the log likelihood as a quadratic function of the parameters. Move to the maximum of this approximation at each iteration. *Infeasible if there are many parameters (matrix too big); may not work well anyway, if the likelihood is very non-quadratic.*

- Conjugate gradient methods — cleverly choose good directions to look for the maximum, moving to the maximum along that direction at each iteration.

- Gradient descent — always move in the direction of the gradient (vector of partial derivatives). In the simplest scheme, we change w by some constant times the gradient of the log likelihood. *The original, and crudest, method, but sometimes the best — it doesn't need a big matrix of second derivatives, and can be used "on-line", without looking at all training cases at once.*

Computing Derivatives of the Log Likelihood

Partial derivatives of the log likelihood with respect to the network parameters are found by applying the chain rule backwards from the network output. Since training cases are assumed to be independent, we calculate derivates separately for each training case, then just sum them. I'll denote **minus** the log likelihood for just one training case by E, so that smaller E is better.

We first do "forward propagation", computing the summed input to each hidden unit, a_j, then the values of the hidden units, z_j, and finally the value of the output unit, f:

$$\begin{aligned} a_j &= w_{0j}^{(1)} + \sum_{k=1}^{p} w_{kj}^{(1)} x_k \\ z_j &= \tanh(a_j) \\ f &= w_0^{(2)} + \sum_{j=1}^{m} w_j^{(2)} z_j \end{aligned}$$

We then use "backpropagation" to find the derivatives of E with respect to f, with respect to z_j, and with respect to a_j. From these, we can find the derivatives with respect all the components of w.

Derivatives of E With Respect to f, z_j, and a_j

To start, we find the derivative of E with respect to the network output, f. If the response, y, is modeled as Gaussian with mean f and variance σ^2, then minus the log likelihood for just one case is $E = (y - f(x, w))^2 / 2\sigma^2$, and its derivative with respect to f is

$$\frac{\partial E}{\partial f} \;=\; (f - y) \,/\, \sigma^2$$

Once we have computed $\partial E / \partial f$, we work backward to (for all j) compute $\partial E / \partial z_j$, found assuming that $w_j^{(2)}$ and the $z_{j'}$ for $j' \neq j$ are fixed:

$$\frac{\partial E}{\partial z_j} \;=\; \frac{\partial E}{\partial f}\,\frac{\partial f}{\partial z_j} \;=\; w_j^{(2)}\,\frac{\partial E}{\partial f}$$

Next, we use the fact that $\tanh'(a) \;=\; 1 - \tanh(a)^2$ to work back to the derivative of E with respect to a_j, the summed input to hidden unit j:

$$\frac{\partial E}{\partial a_j} \;=\; \frac{\partial E}{\partial z_j}\,\frac{\partial z_j}{\partial a_j} \;=\; (1 - z_j^2)\,\frac{\partial E}{\partial z_j}$$

If we had more than one hidden layer, we would continue working backwards, finding the derivatives of E with respect to the values of all hidden units, and with respect to the summed inputs for these hidden units.

Derivatives of E With Respect to w

We can find the derivative of E w.r.t. a parameter on a connection from unit A to unit B by multiplying the value of A by the derivative of E w.r.t. the summed input to B.

For $w_1^{(2)}, \ldots, w_m^{(2)}$:

$$\frac{\partial E}{\partial w_j^{(2)}} = \frac{\partial E}{\partial f}\frac{\partial f}{\partial w_j^{(2)}} = z_j \frac{\partial E}{\partial f}$$

Also, $\partial E \,/\, \partial w_0^{(2)} = \partial E \,/\, \partial f.$

Similarly, for $w_{1j}^{(1)}, \ldots, w_{pj}^{(1)}$:

$$\frac{\partial E}{\partial w_{kj}^{(1)}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{kj}^{(1)}} = x_k \frac{\partial E}{\partial a_j}$$

Also, $\partial E \,/\, \partial w_{0j}^{(1)} = \partial E \,/\, \partial a_j.$

Maximizing the Likelihood by Simple Gradient Ascent

After randomly initializing the parameters, w, to small values, we maximize the log likelihood by repeatedly doing the following, using some suitably chosen "learning rate" or "stepsize", η:

1. For each training case:

   – Compute the values of the units by forward propagation.
   – Compute the derivatives of E w.r.t. the unit values by backpropagation.
   – Compute the derivatives of E w.r.t. the parameters from these results.

2. Sum these derivatives over all training cases, to get the gradient vector, g, of minus the total log likelihood.

3. Change w by moving in the direction of (minus) this gradient, replacing w by $w - \eta g$.

If η is too big, the changes will "overshoot", and end up decreasing the likelihood rather than increasing it. We have to set η by trial-and-error.

On-line Learning

The procedure in the previous slide is sometimes called "batch" gradient ascent learning, since the derivatives from the training cases are handled as one batch.

It's also possible to do "on-line" learning, in which we update the parameters based on each training case in turn. This may work better if the training cases are redundant — many cases provide the same information. Batch learning then wastes time looking at them all before doing anything.

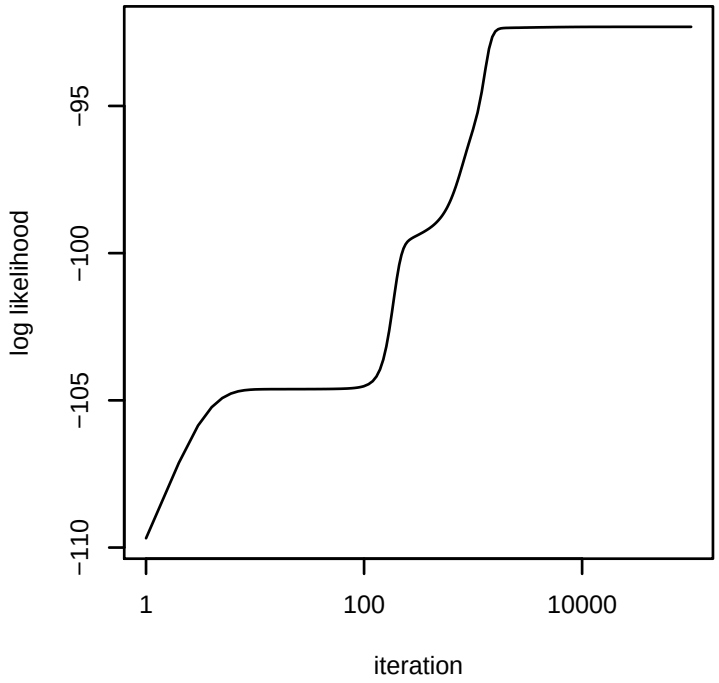
However, for on-line learning to converge to a (local) maximum, we will have to decrease η with time (or switch to batch learning once we're near the maximum).

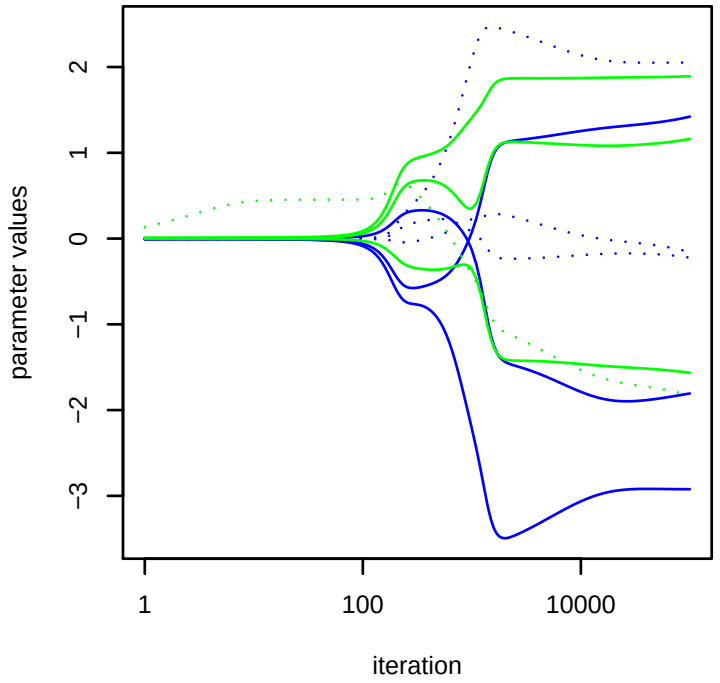
Actually, *true* on-line learning uses a new training case for each update — there is no finite training set, but rather a continuous stream of training cases. This is the situation for perceptual learning in humans.

Example of Simple Gradient Ascent Learning

Recall the previous example, with one input (ranging over $(0, 1)$), one real target (equal to $\sin(4x) + \text{noise}$), and 100 training cases.

I trained a MLP network with three hidden units for 100000 iterations of simple gradient descent ($\eta = 0.3$). Here are plots of the log likelihood and of the 10 parameters as a function of iteration (log scale):

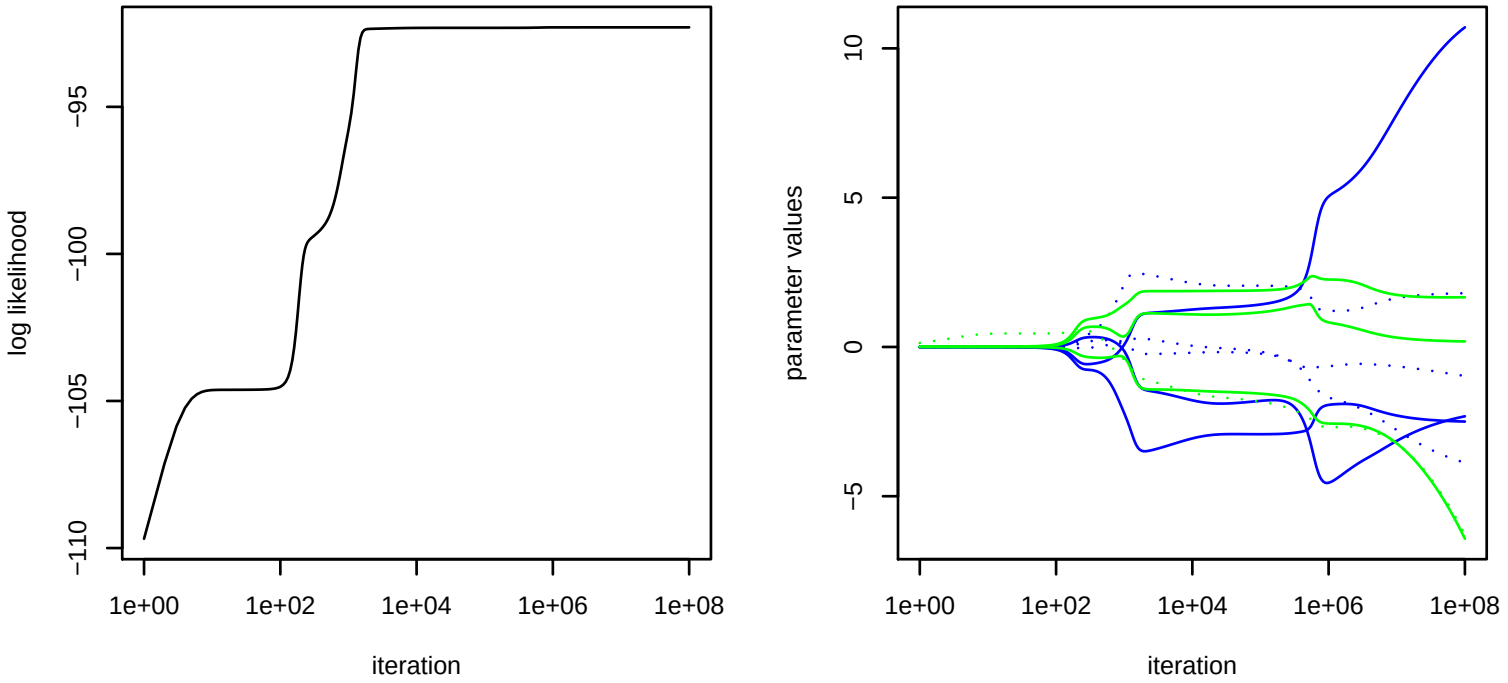




Blue lines are $w_{kj}^{(1)}$ ($k = 0$ dotted). Green lines are $w_j^{(2)}$ ($j = 0$ dotted).

Continuing the Example...

It may seem like the optimization has converged after 100000 iterations. But what happens if we continue training for many more iterations?

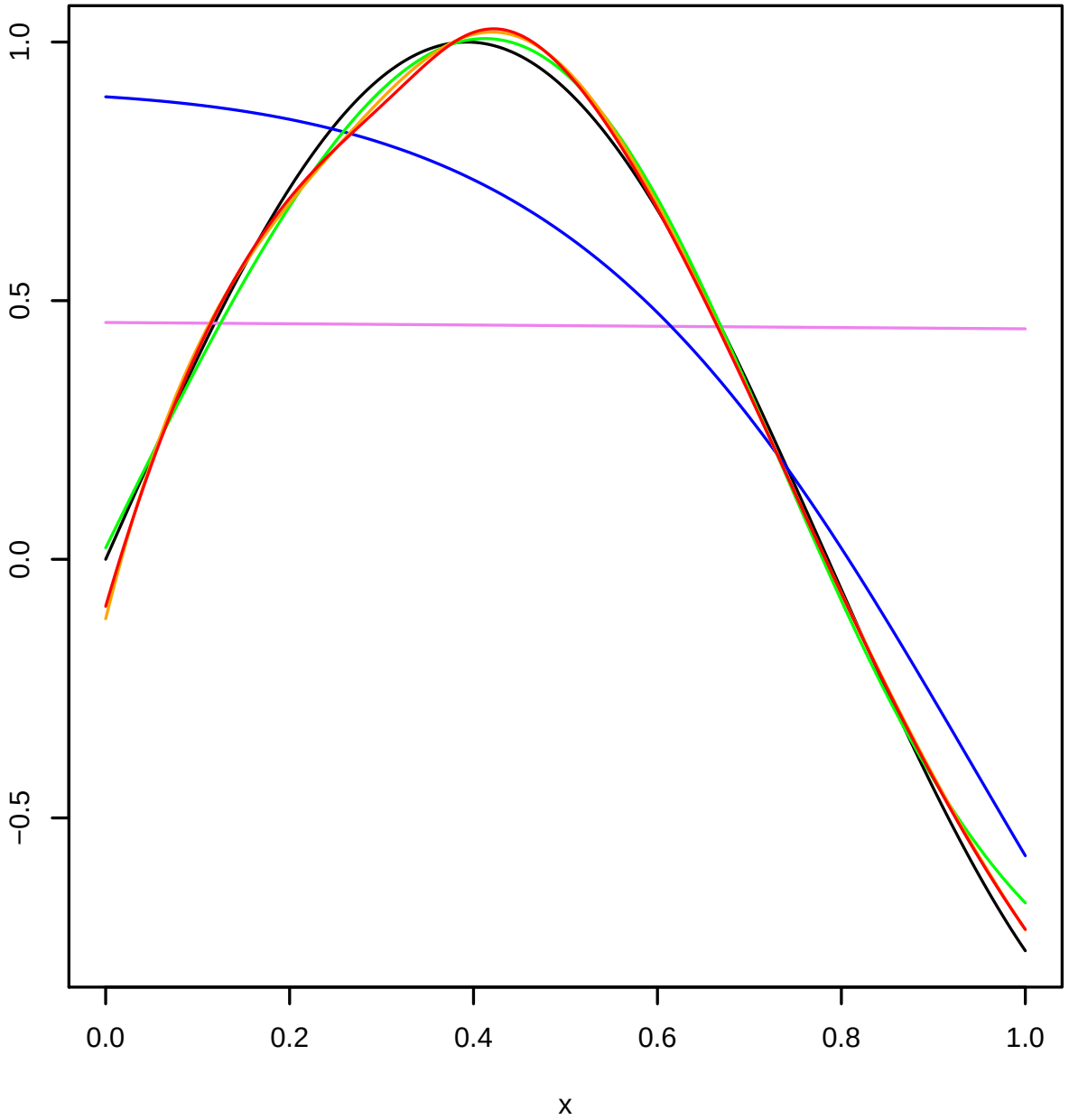


This shows how tricky the MLP likelihood function can be! In practice, we don't try to find the absolute maximum, since with complex networks it is sure to overfit the training data anyway...

Overfitting in This Example

Here is the true regression function, $\sin(4x)$, in black, and the regression functions defined by the networks after training for 10^2 (violet), 10^3 (blue), 10^4 (green), 10^6 (orange), and 10^8 (red) iterations.

The last two curves may have slightly overfit the data. Much more severe overfitting can occur when there are more inputs and/or more hidden units.



Avoiding Overfitting Using a Penalty

As for linear basis function models, we can try to avoid overfitting by maximizing the log likelihood plus a penalty function.

For an MLP with one hidden layer, a suitable penalty to add to minus the log likelihood might be

$$\lambda_1 \sum_{k=1}^{p} \sum_{j=1}^{m} \left[w_{kj}^{(1)} \right]^2 \; + \; \lambda_2 \sum_{j=1}^{m} \left[w_{j}^{(2)} \right]^2$$

(Sometimes the penalty is multiplied by 1/2, to get rid of a factor of 2 in the derivative of the penalty.)

We need to select two constants controlling the penalty, λ_1 and λ_2. Setting $\lambda_1 = \lambda_2$ isn't always reasonable, since a suitable value for λ_1 depends on the measurement units used for the inputs, whereas a suitable value for λ_2 depends on the measurement units for the response.

We might try S-fold cross-validation, but it may not work well, if each training run goes to a different local maximum. So we might use a single split into estimation and validation sets, with no re-training on the whole training set.

Neural Network Classification Models

We can also use a multilayer perceptron network as a discriminative classification model. The output of the network, $f(x, w)$, can be used in a logistic model for the probability that a binary class variable, y, is 1:

$$P(y = 1 \,|\, x,\, w) \quad = \quad \Big[1 + \exp(-f(x, w))\Big]^{-1} \quad = \quad \sigma(f(x, w))$$

where $\sigma(a) = 1/(1 + e^{-a})$.

The derivative of σ is $\sigma'(a) = \sigma(a)\,(1-\sigma(a)) = \sigma(a)\sigma(-a)$. We use this when computing derivatives of minus the log likelihood by backpropagation.

We can use a probit model for a binary class instead, if we prefer it.

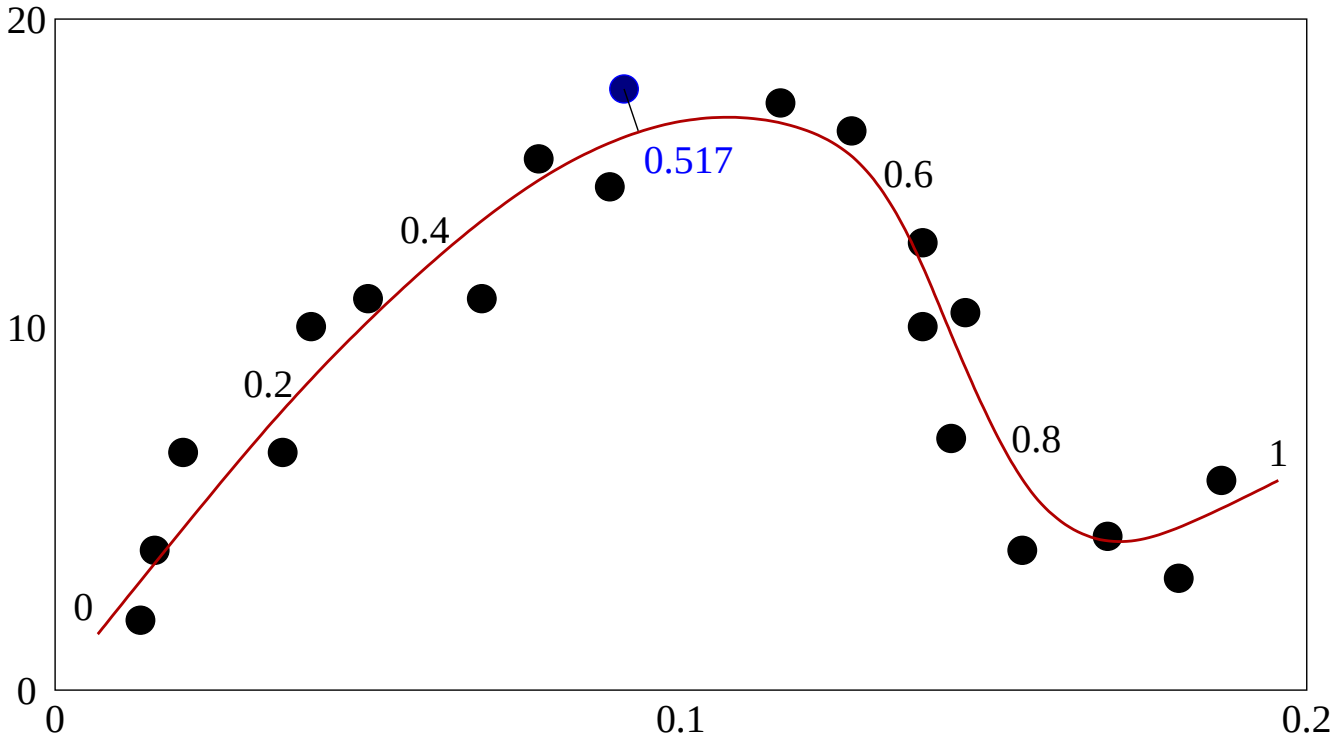
When there are more than two classes, we can use a "multinomial logit" (also called "softmax") model. With K classes, we use a network with K outputs, $f_k(x, w)$ for $k = 1, \ldots, K$. The class probabilities are then defined to be

$$P(y = k \,|\, x,\, w) \quad = \quad \frac{\exp(f_k(x, w))}{\sum_{k'=1}^{K} \exp(f_{k'}(x, w))}$$

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 "longitude" 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 p.

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.

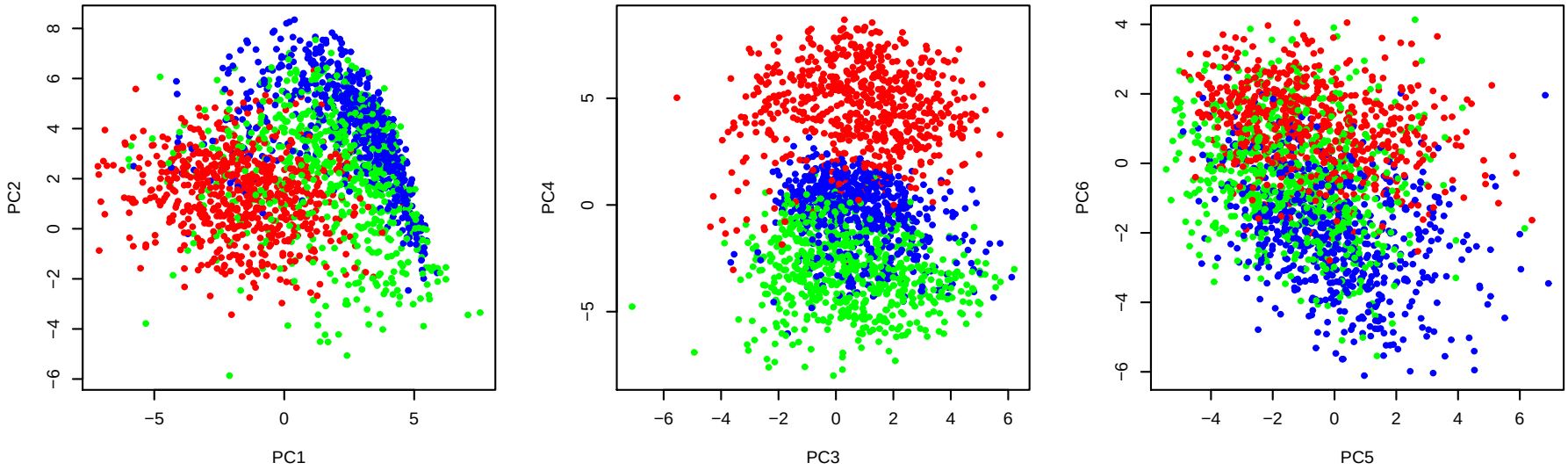
PCA is also used as a preliminary to supervised learning. Rather than use the original p inputs to try to predict y, we instead use the projections of these inputs on the first M principal components. This may help avoid overfitting.
It certainly reduces computation time.

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 16×16 image of the digit, so there are 256 inputs. There are 7291 training cases.

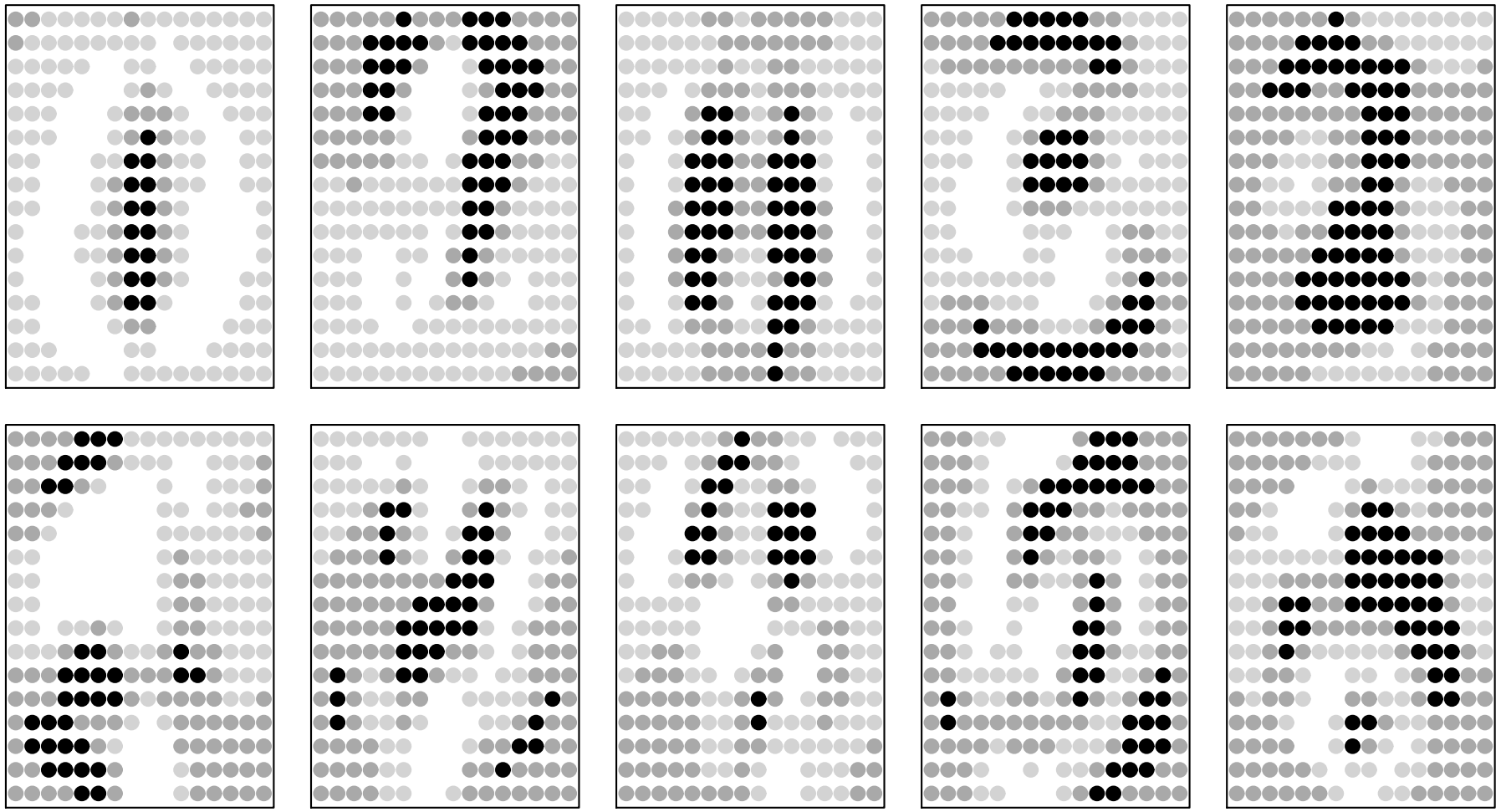
Here are plots of 1st versus 2nd, 3rd versus 4th, and 5th versus 6th principal components for training cases of digits "3" (red), "4" (green), and "9" (blue):



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 small and 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.