

Introduction to Neural Networks

U. Minn. Psy 5038

Spring, 1999

Self-organization

Decorrelation, and Principal Components Analysis with Neural Networks

Initialization

```
Off[SetDelayed::write]
Off[General::spell1]
<<Statistics`ContinuousDistributions`
```

Introduction

Unsupervised learning--what does it mean to learn without a teacher?

Learning as probability density estimation. From a relatively small number of samples (human faces), what can we conclude about the probabilistic structure of the whole ensemble (of human faces)?

Single random variable: mean, standard deviation, histogram

Vector random variable: means, autocovariance, higher-order statistics, histogram

Problem: How can we generalize from a relatively small number of samples to characterize the essential structure of the whole ensemble?

One solution:

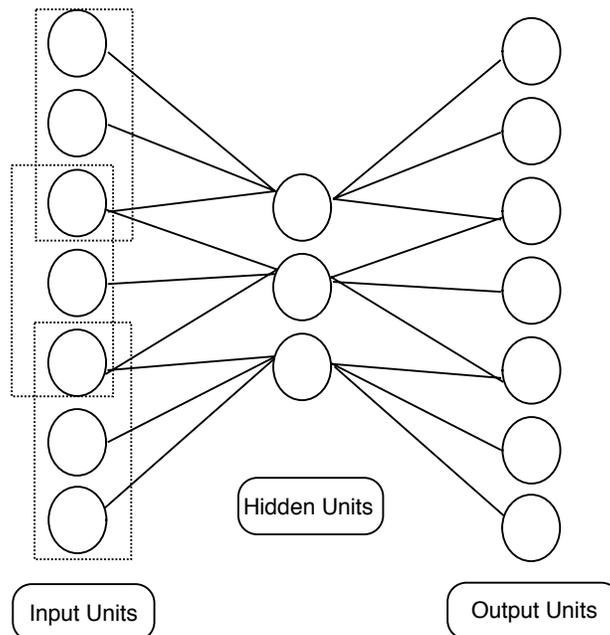
Assume a parametric model (e.g. Gaussian), because then we only need to estimate a few well-known statistics (e.g. mean and autocovariance matrix)

Unsupervised learning, self-organization, and data compression

In an earlier lecture, we saw an example of the increased computational power of a multi-layer network. The reason for the increased power is that the hidden units discover effective ways of representing contingencies in the training data set. For example, a solution to the XOR problem in effect discovers how to do AND as well as OR relations, and piece these together.

One of the problems with multi-layer nets is understanding exactly what they have discovered and are representing in the hidden layers. It is perhaps easiest to begin tackling this problem by setting up a network to do autoassociation.

Let's turn the supervised 3-layer backprop network into an unsupervised learning algorithm simply by setting the output equal to the input. Then we are seeking weights that achieve the goal that the outputs come as close as possible to matching the inputs, in a least squares sense. If we have a smaller number of hidden units than input or output units, we can ask: What has the network discovered about the input ensemble that is captured with the smaller dimensionality of the hidden unit layer?



A theoretical result (Baldi and Hornik, 1988) showed that for the linear case this kind of network is closely related to a standard statistical technique called "Principal Components Analysis", (PCA) that dates back to 1933 (Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. J. Educ. Psych., 24, 417-441, 498-520). The idea is that the variability in a data set consisting of n -dimensional vectors may concentrate along certain axes or subspaces of dimension $m < n$. Principal components analysis is one standard technique for finding the dimensions which capture the most variation. Suppose there are n input units, and m hidden units. Baldi and Hornik showed that the hidden units were finding the m -dimensional sub-space that capture the most variance. This is not exactly principal components analysis, but it is closely related.

In this notebook you will learn about PCA, and then see how it can be done by neural-like systems that use local hebbian learning, and avoid error back-propagation.

In order to understand PCA, let's start with the following simple two neuron system. The rationale is that the two neuron

system will give us insight into the problem of finding structure in data sets with really high dimensionality, such as images or speech.

Modeling the input: Statistical model of the correlations of a 2D input ensemble

Consider a "two-neuron" system whose inputs are correlated. The random variable, \mathbf{rv} , is a 2D vector. The scatter plot for this vector has a slope of $\text{Tan}[\theta] = 0.41$. The variances along the axes are 4^2 and $.25^2$ (.0625). **gprincipalaxes** is a graph of the principal axes which we will use for later comparison with simulations.

ContinuousDistributions.m is a *Mathematica* package that you have seen before and that provides routines for sampling from a Gaussian (or Normal) distribution, rather than the standard uniform distribution that **Random[]** provides.

```
ndist = NormalDistribution[0,1];theta = Pi/8;
bigstd = 4.0; smallstd = 0.25;
alpha = N[Cos[theta]]; beta = N[Sin[theta]];
rv :=
{bigstd x1 alpha + smallstd y1 beta, bigstd x1 beta -
smallstd y1 alpha} /.
{x1-> Random[ndist],y1-> Random[ndist]};

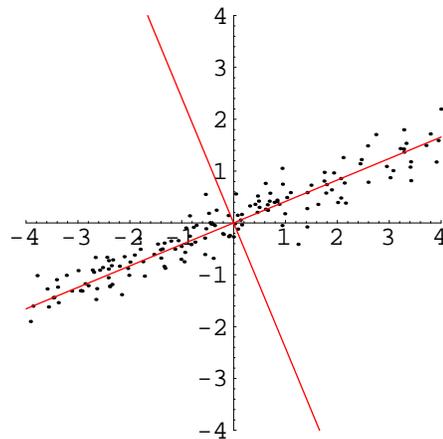
gprincipalaxes = Plot[{x (beta/alpha),
x (-1/(beta/alpha))}, {x,-4,4},
PlotRange->{{-4,4},{-4,4}},
PlotStyle->{RGBColor[1,0,0]},
AspectRatio->1,DisplayFunction->Identity];
```

$x1$ and $y1$ are said to be *correlated*. Let's view a scatterplot of samples from these two correlated Gaussian random variables.

```
npoints = 200;
rvsamples = Table[rv,{n,1,npoints}];
```

```
g1 = ListPlot[rvsamples,PlotRange->{{-4,4},{-4,4}},
AspectRatio->1, DisplayFunction->Identity];
```

```
Show[g1,gprincipalaxes,
  DisplayFunction-> $DisplayFunction];
```



Principal Components Analysis (PCA) using standard statistical methods

In the previous section, we developed a model for synthetic data--so we know the statistical structure. Usually, we don't have a good model of the statistical structure of an ensemble, but want to discover something about it. It is usually too hard to find a complete model of the underlying distribution, but we can still analyze the data to find means, variances and so forth. When dealing with a high-dimensional ensemble, sometimes most of the variation is occurring in some subspace. In the example above, most of the variation is occurring in the 1D subspace defined by a line of slope **theta**. How can we discover that without having the model before us?

PCA provides the method. PCA seeks out a new coordinate system that is just a rotation of the original that does two very interesting things: 1) The data when projected onto the new rotated coordinates are no longer correlated--in fact the autocovariance matrix (see below) is diagonal; 2) The new coordinates can be ordered so that the main or "principal component" has the most variance, the next has the second most, and so forth. How is this done? It turns out that the eigenvectors of the autocovariance matrix are the principal components, and the eigenvalues are the variances of the data when projected onto the new axes.

What good is PCA for a data set? If the variance of some of the projections is near zero, one can in fact dispense with these components and achieve a good approximate coding of the data with just the remaining coordinates. We are going to see that in the 2D example, we can get an economical coding of the data with just one number, rather than two.

■ Calculate the autocovariance matrix

Let $E[\bullet]$ stand for the expected or average of a random variable, \bullet . The autocovariance matrix of a vector random variable, \mathbf{x} , is: $E[(\mathbf{x}-E[\mathbf{x}])(\mathbf{x}-E[\mathbf{x}])^T]$. Let's compute the autocovariance matrix for \mathbf{rv} . The calculations are simpler because the average value of \mathbf{rv} is zero. As we would expect, the matrix is symmetric:

```

autolist = Table[
  Outer[Times, rvsamples[[i]], rvsamples[[i]],
    {i, Length[rvsamples]}];
MatrixForm[auto =
  Sum[autolist[[i]],
    {i, Length[autolist]}/Length[autolist]]
Clear[autolist];

```

$$\begin{pmatrix} 12.5964 & 5.07562 \\ 5.07562 & 2.11868 \end{pmatrix}$$

The variances of the two inputs (the diagonal elements) are due to the projections onto the horizontal and vertical axis of the generating random variable. (The autocorrelation matrix is expected value of the outer product without subtracting off the means.)

■ Calculate the principal components (eigenvectors of the autocovariance matrix)

Now we will calculate the eigenvectors of the autocovariance matrix

```

MatrixForm[eigauto = Eigenvectors[auto]]

```

$$\begin{pmatrix} 0.926878 & 0.375362 \\ -0.375362 & 0.926878 \end{pmatrix}$$

Remember that the eigenvectors of a symmetric matrix are orthogonal. You can check that these are.

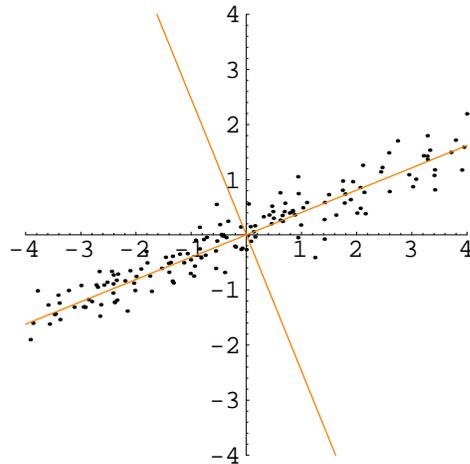
Let's graph the principal axes corresponding to the eigenvectors of the autocovariance matrix together with the scatterplot we plotted earlier.

```

gPCA = Plot[{eigauto[[1,2]]/eigauto[[1,1]] x,
  eigauto[[2,2]]/eigauto[[2,1]] x},
  {x,-4,4}, AspectRatio->1,
  DisplayFunction->Identity,
  PlotStyle->{RGBColor[1,.5,0]};

```

```
Show[g1,gPCA,DisplayFunction->$DisplayFunction];
```



You can see that PCA has *discovered* important structure in the input ensemble.

■ Calculating the variances (eigenvalues)

Before calculating the eigenvectors, you should be able to figure out what they should be, if PCA is doing what we think it is.

The eigenvalues give the ratio of the variances of the projections of the random variables $\mathbf{rv}[[1]]$, and $\mathbf{rv}[[2]]$ along the principal axes:

```
eigvalues = Eigenvalues[auto]
```

```
{14.6519, 0.0631816}
```

The projections along the principal axes are now **decorrelated**. We can see this by calculating the autocovariance matrix of the projected values:

```
autolist = Table[
  Outer[Times,eigauto.rvsamples[[i]],
    eigauto.rvsamples[[i]],{i,Length[rvsamples]}];
MatrixForm[Chop[
  Sum[autolist[[i]],
    {i,Length[autolist]}/Length[autolist]]
Clear[autolist];
```

```
( 14.6519    0
   0 0.0631816)
```

Note that the off-diagonal elements (the terms that measure the covariation of the transformed random variables) are zero. Further, the diagonal elements are estimates of the population variances along the principal axes. The population variances

are given by the **bigstd**², and **smallstd**² from our synthetic data model.

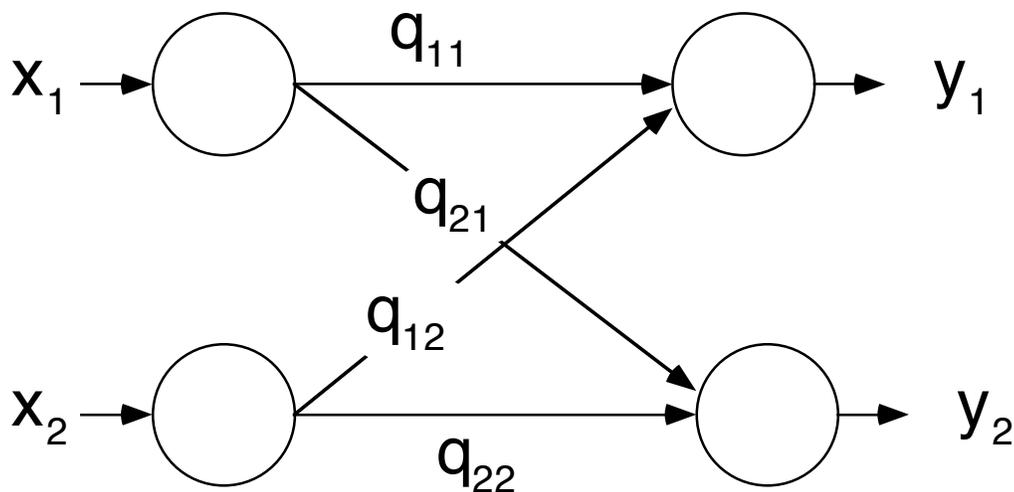
How can we do PCA using brain-style computation?

Principal Components Analysis (PCA) with neural networks

Neural network model using Hebb together with Oja's rule for extracting the dominant principal component

■ Introduction to Oja's network: weight normalization

Consider the following linear neural network. The input and output values are represented by vectors \mathbf{x} , and \mathbf{y} respectively. The connection weights are represented by matrix \mathbf{Q} .



We will combine the outer product form of Hebb's rule, together with Oja's modification. Without Oja's rule, the Hebb rule does not place a limit on the size of the weights.

$$\Delta q_{ij} = \alpha (x_j y_i - q_{ij} y_i^2)$$

Exercise: Show, when the weights are no longer changing, that:

$$\sum_{i,j} w_{ij}^2 = 1$$

■ Implementing Oja's network

Oja's rule constrains the sum of the squares of the weights to approach 1. We will set the initial values of the weight matrix to random values between 0 and 1.

```

npoints = 400; p1 = {};  $\alpha$  = 0.01;
size = 2;
Q = Table[Random[], {size}, {size}];

```

Note that a space in *Mathematica* between two expressions does an element by element multiplication. We use this notation as economical way of writing Oja's rule. An example is:

```

MatrixForm[{ {a,b}, {c,d} } {x,y} ]

```

```

a x    b x
c y    d y

```

Note that this different from standard matrix multiplication.

```

For[i=1,i<=npoints,i++,
    x = rv; y = Q.x;
    Q = Q +  $\alpha$  (Outer[Times,y,x] - Q y y);

(*p1 keeps track of the evolution of the weights
in terms of the slopes of the rotated coordinates*)
If[Mod[i,5]==0,
    p1 = Join[p1,{ {Q[[1,2]]/Q[[1,1]],
                  Q[[2,2]]/Q[[2,1]] } }]];
];

```

■ Graph the evolution of the network: slopes of the projection axes

Let's plot the slopes of projection axes as a function of iterations. We've sampled every 5th value, using `Mod[i,5]`, and stored it in `p1`. These values should approach the slope of the scatter plot, `Tan[theta]`.

```

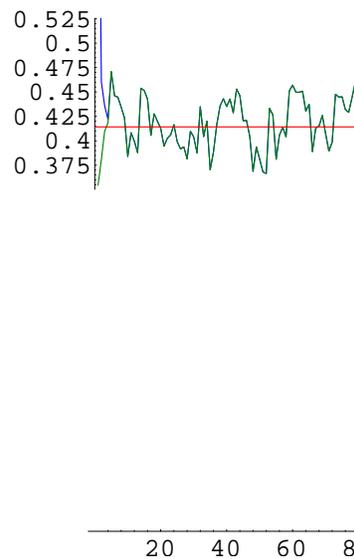
Plot[ Tan[theta], {x, 0, Length[p1]}, AxesOrigin->{0,0},
  DisplayFunction->Identity,
  PlotStyle->{RGBColor[1,0,0]}}];

ListPlot[Map#[[2]]&,p1], AxesOrigin->{0,0},
  PlotJoined->True, DisplayFunction->Identity,
  PlotStyle->{RGBColor[0,.5,0]}}];

ListPlot[Map#[[1]]&,p1], AxesOrigin->{0,0},
  PlotJoined->True, DisplayFunction->Identity,
  PlotStyle->{RGBColor[0,0,1]}}];

Show[%,%%,%%%, DisplayFunction->$DisplayFunction];

```



There is some random fluctuation in the weights. We can obtain more stability by having a time constant over which the Hebbian term and the variance of y are averaged.

■ Graph the slope of the slope of the projection axes (ratio of network weights) together with the data

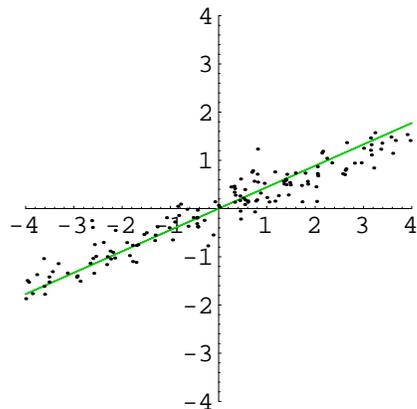
We can see how well the coordinate transformation fits the principal axes of a sample scatter plot:

```

gnetwork = Plot[
  {s p1[[Length[p1]]][[1]], s p1[[Length[p1]]][[2]]},
  {s, -4, 4}, PlotRange->{{-4,4},{-4,4}},
  AspectRatio->1,PlotStyle->{RGBColor[0,.8,0]},
  DisplayFunction->Identity];

```

```
Show[gnetwork,g1,DisplayFunction->$DisplayFunction];
```



You can verify that the network does a good job of extracting the principal component. Recall that the slope for the population distribution is $\text{Tan}[\theta]$:

```
N[Tan[theta]]
```

```
0.414214
```

The only problem with this network is that having two output neurons is redundant--they both pull out the same principal component--the dominant axis. The slopes for both are:

```
p1[[Length[p1]]]
```

```
{0.443491, 0.443491}
```

This isn't surprising because the network was symmetrical--both output neurons saw the same inputs and updated their weights using the same rule. How can this be fixed to pick out the other principal components? Some kind of asymmetry has to be introduced.

A generalization of Oja's rule for extracting all of the principal components (Sanger, 1989)

■ Introduction to Sanger's network for PCA

The problem with Oja's network is that it extracts just the main principal component. Our example had two outputs, but the network is symmetric, and both outputs were the same--the projection of the input onto the main principal axes. Sanger proposed a modification to the Oja network that can extract *all* of the principal components.

We will use the same network as in the above example. However, the normalization part of learning rule will be asymmetric. The generalization of Oja's term to update weights q , is given by:

$$\Delta q_{ij} = \alpha \left(x_j y_i - y_i \sum_{k=1}^i q_{kj} y_k \right)$$

■ Implementing the Sanger network

The above learning rule can be evaluated in *Mathematica* as `LT Outer[Times,y,y].Q`, where `LT` is a lower triangular matrix. The entries above the diagonal are all zero, and the entries below and including the diagonal are one. You can verify the Sanger weight update formula with the following expressions:

```
n = 3;
LT = Table[If[i>=j,1,0],{i,n},{j,n}];
Q = Array[q,{n,n}];
yy = Array[y,{n}];
(LT Outer[Times,yy,yy]).Q
```

$$\begin{pmatrix} q(1,1)y(1)^2 & q(1,2)y(1)^2 & & & \\ q(2,1)y(2)^2 + q(1,1)y(1)y(2) & q(2,2)y(2)^2 + q(1,2)y(1)y(2) & & & \\ q(3,1)y(3)^2 + q(1,1)y(1)y(3) + q(2,1)y(2)y(3) & q(3,2)y(3)^2 + q(1,2)y(1)y(3) + q(2,2)y(2)y(3) & & & \\ q(3,3)y(3)^2 & & & & \end{pmatrix}$$

OK, let's try Sanger's network out on our synthetic data.

```
npoints = 1200;
p1 = {};  $\alpha$  = 0.02;
size = 2;
LT = Table[If[i>=j,1,0],{i,size},{j,size}];
Q = Table[.3 Random[], {size}, {size}];
```

```
For[i=1,i<=npoints,i++,
  x = rv; y = Q.x;
  deltaQ = (Outer[Times,y,x] - (LT Outer[Times,y,y]).Q);
  Q = Q +  $\alpha$  deltaQ;
  If[Mod[i,10]==0,
    p1 = Join[p1,{Q[[1,2]]/Q[[1,1]], Q[[2,2]]/Q[[2,1]] }]];
];
```

You may have to adjust the learning constant. It can take 1000's of iterations to converge, so don't give up easily.

■ Graph the evolution of the network weights in terms of the slope

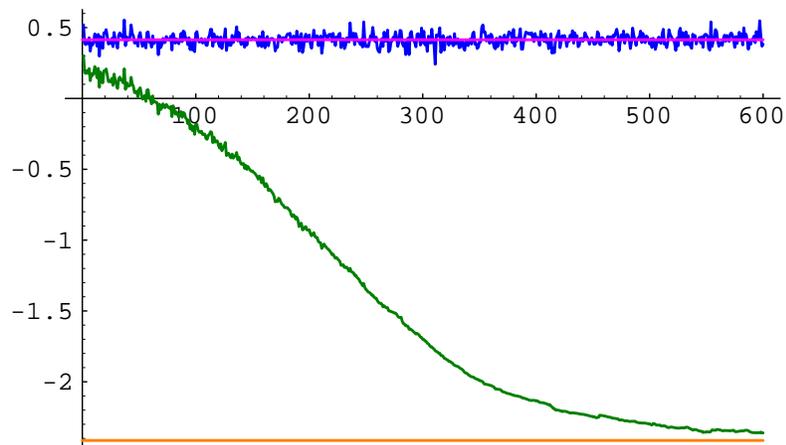
From the model of our synthetic data, the two slopes should be:

Tan[theta] and **-1/Tan[theta]**. Let's take a look at the slopes of the transformed coordinates in the list **p1**:

```
Plot[-1/Tan[theta], {x, 0, Length[p1]}, DisplayFunction->Identity,
  PlotStyle->{RGBColor[1, .5, 0]};
Plot[Tan[theta], {x, 0, Length[p1]}, DisplayFunction->Identity,
  PlotStyle->{RGBColor[1, 0, 1]};

ListPlot[Map[#[[2]]&, p1],
  PlotJoined->True, DisplayFunction->Identity,
  PlotStyle->{RGBColor[0, .5, 0]};

ListPlot[Map[#[[1]]&, p1],
  PlotJoined->True, DisplayFunction->Identity,
  PlotStyle->{RGBColor[0, 0, 1]};
Show[%, %, %, %, DisplayFunction->${DisplayFunction};
```



Note that the number of iterations is plotted in multiples determined by the Mod[] function above.

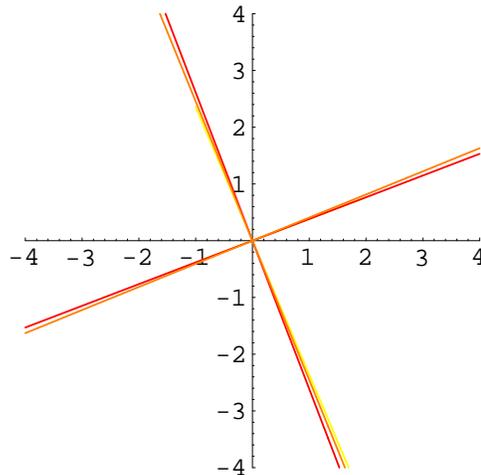
■ Graph the transformed axes of the Sanger network and compare them to those from the underlying distribution

Let's plot up the transformation axes of the Sanger network (**gnetwork2**), and compare them with the axes from the population distribution (**gprincipalaxes**), the calculated principal component axes (**gPCA**):

```

gnetwork2 = Plot[
{s p1[[Length[p1]]][[1]], s p1[[Length[p1]]][[2]]},
{s, -1, 2}, PlotRange->{{-4,4},{-4,4}},PlotStyle->{RGBColor[1,1,0]},
AspectRatio->1, DisplayFunction->Identity];
Show[gnetwork2, gprincipalaxes,gPCA,
DisplayFunction->$DisplayFunction];

```



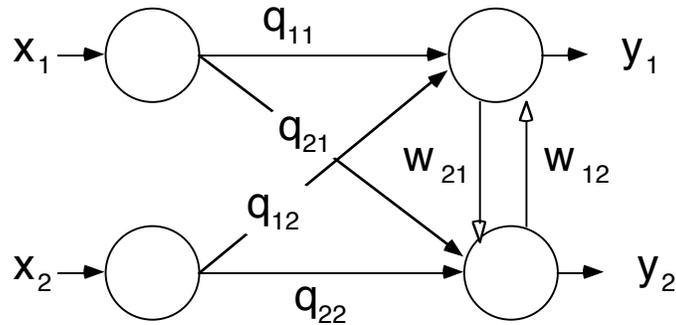
Non-orthogonal decorrelation

■ Foldiak's scheme combining Hebbian and anti-Hebbian learning

Rigid rotations aren't the only possible transformations that decorrelate the inputs. Further, one might want a new coordinate system that shares the variance equally--after all, we do not have strong evidence that neurons vary greatly in their ability to code the range of variation. This section looks at a network due to Peter Foldiak.

Foldiak and Barlow devised a neural network that combined a Hebbian learning rule on the forward connections with anti-Hebbian learning on the inhibitory connections between the output units. Oja's rule was used to normalize the weights. It can be shown that decorrelated output values are steady-state solutions for the weight changes.

One of the reasons for interest in this kind of model are the potential relations with the physiology. Inhibitory links are well-known, and evidence for anti-Hebbian learning is something to be looked for empirically.



Independence and decorrelation

Two random variables, x , y are uncorrelated if $E(xy) = E(x)E(y)$. They are independent if $p(x,y) = p(x)p(y)$. For a multivariate Gaussian density, these two are equivalent. But in general, independence is the stronger condition, in that independence implies no correlation (but the converse is not true). Suppose a neural network recodes the input (x,y) into outputs (x',y') . The above networks "decorrelated" the inputs-- $E(x'y') = E(x')E(y') = 0$. If the inputs are not Gaussian random variables, is there a way to find an independent representation? There has been recent discussion of the utility of both decorrelated and statistically independent representations. We don't have time to go into the details, but the interested student should take a look at Barlow (1990), Bell and Sejnowski (1995), and Olshausen and Field (1996). As an application, Olshausen and Field (1996) showed that the pattern of simple cell receptive field weights in visual cortex can be obtained by seeking a coordinate transformation of image space that decorrelates the output values using two simple constraints: 1) keep the discrepancy between the image data and its linear reconstruction (as a weighted sum of receptive fields) small, and, 2) keep the overall activity level (e.g. total variance) small. The result is a sparse code for images.

References

- Baldi, P., & Hornik, K. (1989). Neural networks and principal components analysis: Learning from examples without local minima. *2*, 53-58.
- Barlow, H. (1990). Conditions for versatile learning, Helmholtz's unconscious inference, and the task of perception. *Vision Research*, *30*(11), 1561-1572.
- Bell A. J. and Sejnowski T. J. . An information-maximization approach to blind separation and blind deconvolution. *Neural Computation*, *7*(6):1129-1159, 1995.
- Roweis, S., & Ghahramani, Z. (1999). A unifying review of linear gaussian models. *Neural Comput*, *11*(2), 305-45.
- Barlow, H. B., & Foldiak, P. (1989). Adaptation and decorrelation in the cortex. In C. Miall, R. M. Durban, & G. J. Mitchison (Ed.), *The Computing Neuron* Addison-Wesley.
- Hinton, G. E., & Ghahramani, Z. (1997). Generative models for discovering sparse distributed representations. *Philos Trans R Soc Lond B Biol Sci*, *352*(1358), 1177-90.
- Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *J. Educ. Psych.*, *24*, 417-441, 498-520
- Oja, E. (1982). A simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, *15*, 267-273
- Olshausen, B. A., & Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, *381*, 607-609.
- Sanger, T. (1989). Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks*, *2*, 459-473.
- Zhu, S. C., Wu, Y., & Mumford, D. (1997). Minimax Entropy Principle and Its Applications to Texture Modeling. *Neural Computation*, *9*(8), 1627-1660.