Introduction to Neural Networks
U. Minn. Psy 5038
Spring, 1998

Principal Components Analysis with Neural Networks

Initialization

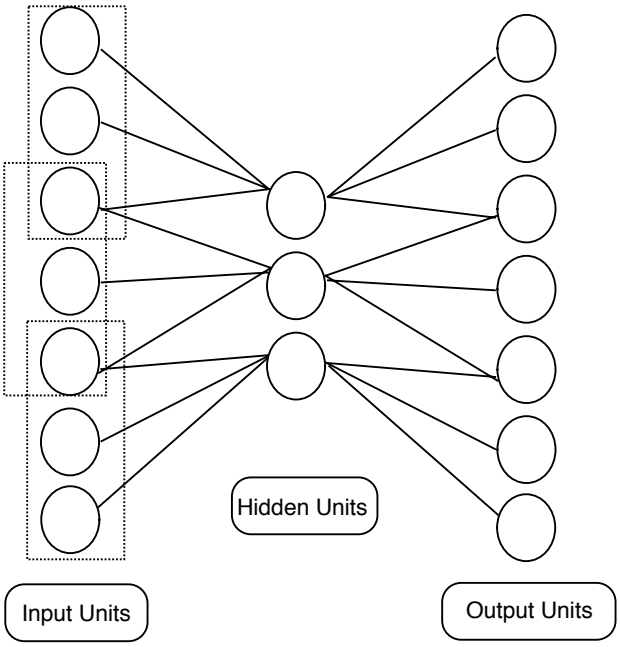
```
Off[SetDelayed::write]
Off[General::spell1]
<<Statistics`ContinuousDistributions`
```

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 "Principle Components Analysis", 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.

Statistical model of the correlations of a 2D input ensemble

Consider a "two-neuron" system whose inputs are correlated. The random variable, **rv**, is a 2D vector. The scatter plot for this vector has a slope of 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.

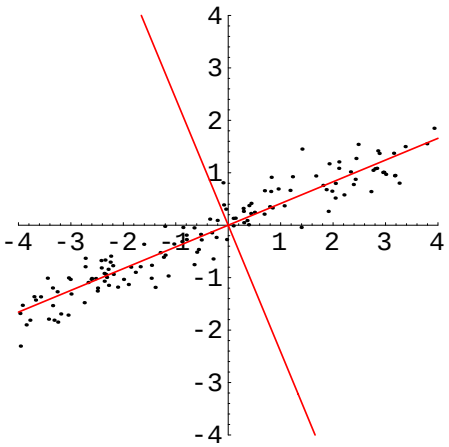
```
ndist = NormalDistribution[0,1];theta = Pi/8;
bigstd = 4.0; smallstd = 0.25;
alpha = N[Cos[theta]]; beta = N[Sin[theta]];
rv :=
{bigstd x1 alpha + smallstd y1 beta, bigstd x1 beta -
smallstd y1 alpha} /.
    {x1-> Random[ndist],y1-> Random[ndist]};

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];
```



Standard Principal Components Analysis (PCA)

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 occuring in some subspace. In the example above, most of the variation is occuring 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[•] stand for the expected or average of a random variable, •. The autocovariance matrix of a of vector random variable, **x**, is: E[[**x**-E[**x**]][**x**-E[**x**]]^T]. Let's compute the autocovariance matrix for **rv**. The calculations are simpler because the average value of **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];
```

```
13.8967   5.64077
5.64077   2.35884
```

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.

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

Now we will calculate the eigenvectors of the autocovariance matrix

```
MatrixForm[eigauto = Eigenvectors[auto]]
```

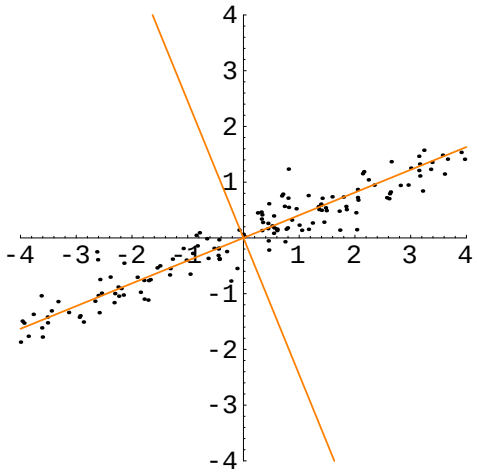
```
0.926014    0.377488
-0.377488   0.926014
```

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 **rv[[1]]**, and **rv[[2]]** along the principal axes:

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

```
{16.1962, 0.0593863}
```

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];
```

```
15.3486     0
0           0.0533513
```

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

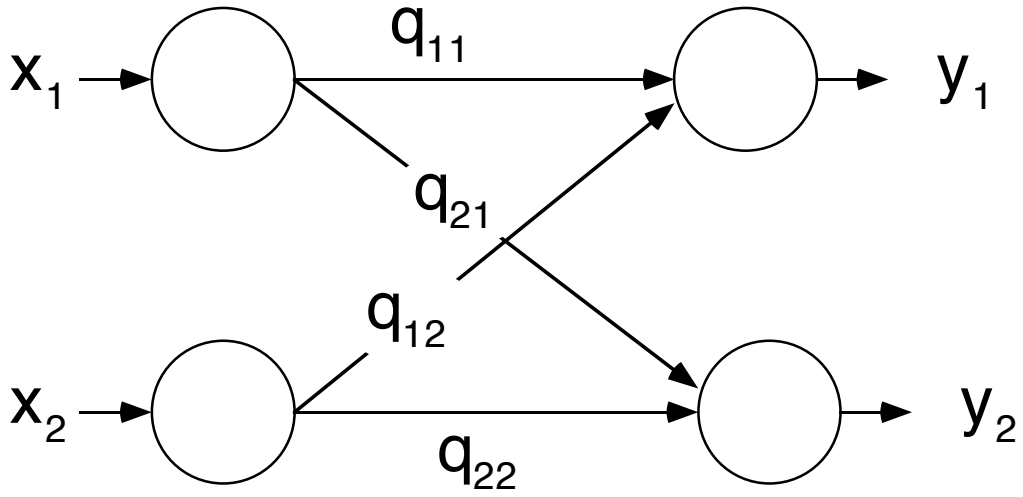
smallstd^2 from our synthetic data model.

How can we do PCA using brain-style computation?

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 **x**, and **y** respectively. The connection weights are represented by matrix **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.

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

■ **Exercise: Show that 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 intial values of the weight matrix to random values between 0 and 1.

```
npoints = 400; p1 = {}; η = 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
```

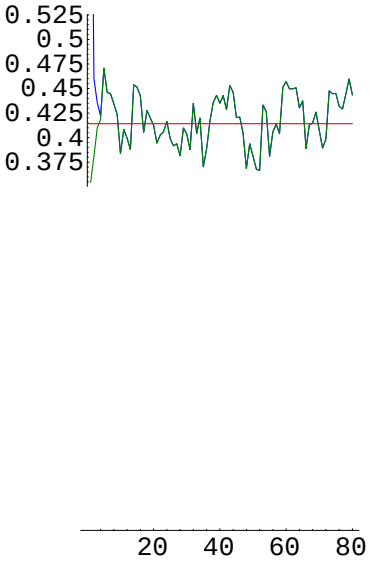
Note that this different from standard matrix multiplication.

```
For[i=1,i<=npoints,i++,
            x = rv; y = Q.x;
            Q = Q + η (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];
```



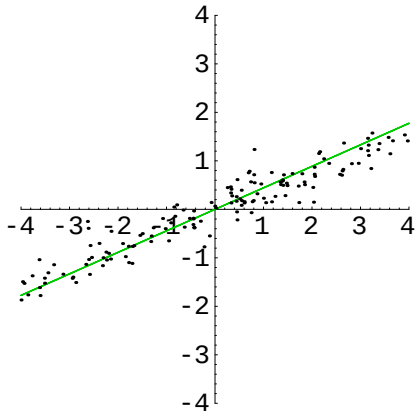
There is some random fluctuation in the weights. We can obtain more stability by having a time constant over which the Hebbian term and the variance of y are averaged.

■ 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 **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 = 2;
LT = Table[If[i>=j,1,0],{i,n},{j,n}];
WW = Array[w,{n,n}];
yy = Array[y,{n}];
(LT Outer[Times,yy,yy]).Q
```

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

```
npoints = 1200;
p1 = {}; α = 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 + α deltaQ;
    If[Mod[i,10]==0,
                p1 = Join[p1,{{Q[[1,2]]/Q[[1,1]], Q[[2,2]]/Q[[2,1]] }}]];

];
```

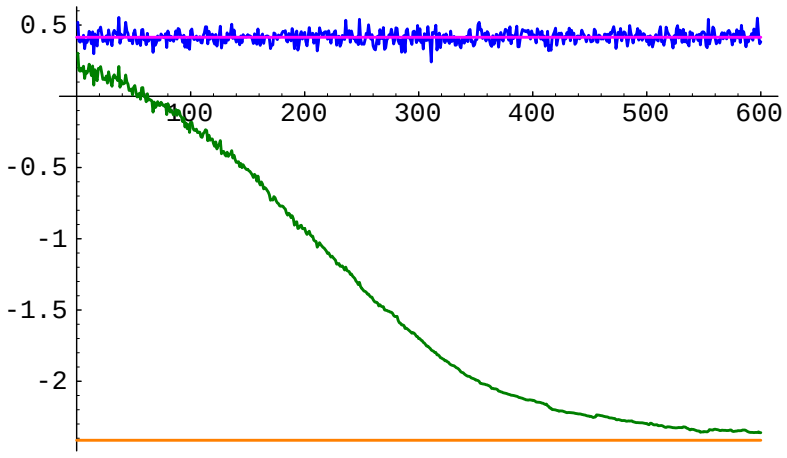
You may have to adjust the learning constant. It can take 1000's of iterations to converge, so don't give up easily.

■ 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];
```

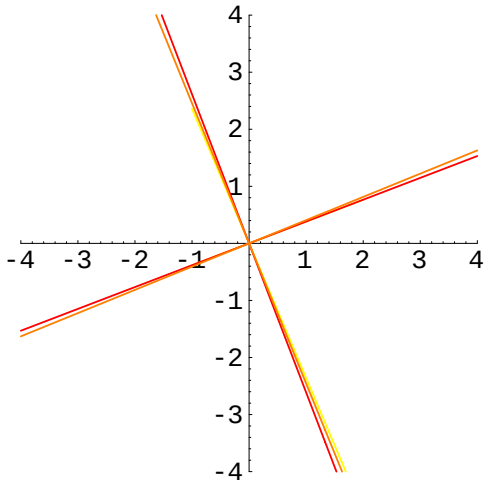


Note that the number of iterations is plotted in multiples determined by the Mod[] function above.

■ 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];
```



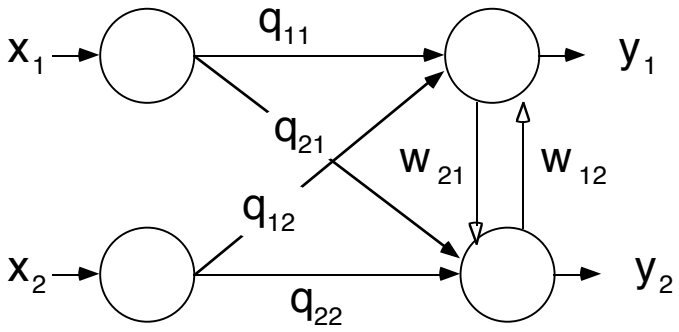
Optional: Foldiak's decorrelating scheme using Hebbian and anti-Hebbian learning

Rigid rotations aren't the only possible transformations that decorrelate the inputs. Further, one might one 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.

As far as I know, the weight space dynamics of this network have not been formally characterized and it may have multiple distinct steady-states. View this section on Foldiak's algorithm as something to explore and play with.

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.



For a fixed set of weights, the steady state of the network unit values is the solution to:

y = Q.x + W.y, which is **y = T.x**, where

T = **Inverse[I - W].Q**.

For the simulations, we will use **Wconnections** to specifiy the connectivity. It wouldn't be very efficient with a large network, but it enables us to keep the code simple. We will also update the weights after averaging over **timeconstant** stimulus presentations.

```
npoints = 256; timeconstant = 4;
β = 0.01; p1 = {}; α = 0.01;
```

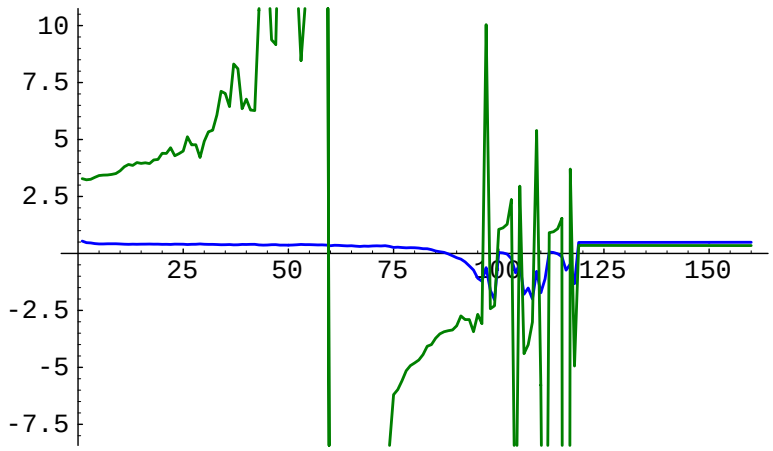
```
size = 2;
Wconnections = Table[1,{size},{size}];
Wconnections[[1,1]] = 0; Wconnections[[1,2]] = 1;
Wconnections[[2,1]] = 1; Wconnections[[2,2]] = 0;
```

```
one = IdentityMatrix[size]; zero = Table[0,{size},{size}];
W = Wconnections zero;
Q = Table[Random[],{size},{size}];
varY = Table[0.0,{size}];
deltaQ = zero; deltaW = zero;
```

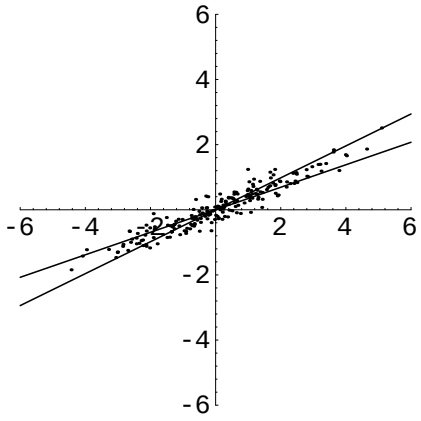
```
For[i=1,i<=npoints,i++,
        T = Inverse[one - W].Q;
        deltaW = zero; deltaQ = zero; varY = 0 varY;
     For[j=1,j<=timeconstant,j++,
            x = rv; y = T.x;
            deltaW = deltaW + Wconnections Outer[Times,y,y];
            deltaQ = deltaQ + Outer[Times,y,x];
            varY = varY + (y y);];
        W = W - α deltaW/timeconstant;
        Q = Q + β (deltaQ - Q varY)/timeconstant;
        If[Mod[i,8]==0,
                p1 = Join[p1,{{T[[1,2]]/T[[1,1]],
                    T[[2,2]]/T[[2,1]] }}]];
];
```

Below, I show the results after runing the above loop five times.

```
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];
```



```
gnetwork = Plot[
{s p1[[Length[p1]]][[1]], s p1[[Length[p1]]][[2]]},
    {s, -6, 6}, PlotRange->{{-6,6},{-6,6}},
    AspectRatio->1,
    DisplayFunction->Identity];
Show[gnetwork,g1,DisplayFunction->$DisplayFunction];
```



Independence and decorrelation

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).

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.

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.

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.