Introduction to Neural Networks
U. Minn. Psy 5038
Spring, 1999

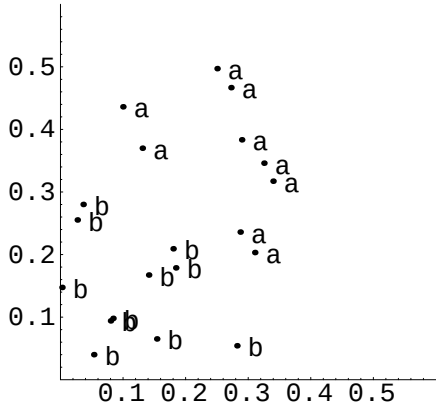
Lecture 9
Sampling, Summed vector memories
Intro. to non-linear models

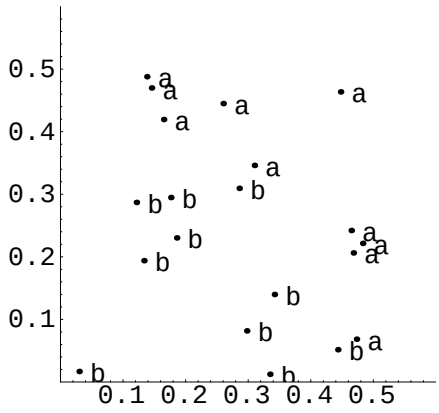
Initialization

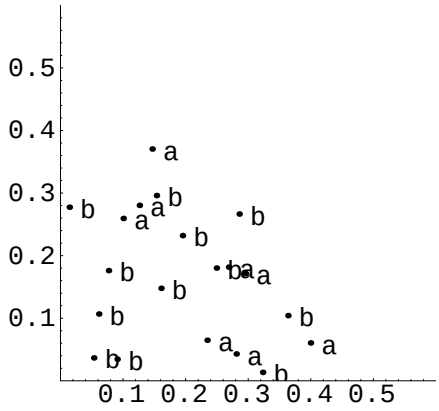
```
Off[SetDelayed::write]
Off[General::spell1]
```

Statistical sampling

We will begin doing "Monte Carlo" simulations of neural network behavior. This means that rather than using real data, we will use the computer to generate random samples for our inputs. Monte Carlo simulations help to see how the structure of the data determines network performance. It is useful to know how to generate random variables (or vectors) with the desired characteristics. For example, suppose you had a one unit network whose job was to take two scalar inputs, and from that decide whether the input belonged to group "a" or group "b". The complexity of the problem, and thus of the network computation depends on the data structure. The next three plots illustrate how the data determines the complexity of the decision boundary that separates the a's from the b's.







Later we'll see how the simplest Perceptron can always solve problems of the first category, but that we'll need more complex models to classify patterns whose separating boundaries are not straight.

■ Inner product of random vectors

In another application of Monte Carlo techniques, in the problem set you will see how the inner product of random vectors is distributed as a function of the dimensionality of the vectors.

The assumption of orthogonality for the input patterns for the linear associator would seem to make it useless as a memory advice for arbitrary patterns. However, if the dimensionality of the input space is large, the odds are pretty good that the cosine of the angle between any two random vectors is close to zero. In the exercise, you will calculate the histograms for the distributions of the cosines of random vectors for dimensions 10, 50 , and 250 to show that they get progressively narrower (see Anderson, p. 187).

■ Probability densities and discrete distributions

As we noted earlier, most standard programming languages come with standard subroutines for doing pseudo-random number generation. Unlike the Poisson or Gaussian distribution, these numbers are **uniformly distributed--**that is, the probability of the random variable taking on a certain value is the same over the sampling range. *Mathematica* comes with a standard function, **Random[]** that enables us to generate (pseudo) random numbers that are uniform, Poisson, Normal, and others. (Why are they "pseudo" random numbers?)

Later in the course, we'll see that there is a close connection between Gaussian random numbers and linear estimators.

There are two packages **DiscreteDistributions.m**, and **ContinuousDistributions.m** which contain the definitions of distributions, cumulative distributions, and provide the means to draw samples.

The alternative package function to the built-in function **Random[]**, is **UniformDistribution[]** that generates uniformly distributed random numbers.

```
<<Statistics`DiscreteDistributions`
<<Statistics`ContinuousDistributions`
```

```
udist = UniformDistribution[0,1];
```

We can define a function, **sample[]**, to generate **ntimes** samples, and then make a list of a 1000 values like this:

```
sample[ntimes_] :=
    Table[Random[Real],{ntimes}];
```

Or like this:

```
sample[ntimes_] :=
    Table[Random[udist],{ntimes}];
```

The second way is more general, because we can use other distributions in our simulations later.

Now let us do a sampling experiment to get the list.

```
z = sample[1000];
```

Count up how many times the result was 20 or less. To do this, we will use two built-in functions: **Count[]**, and **Thread[]**. You can obtain their definitions using the ?? query.

```
Count[Thread[z<=.5],True]
```

```
517
```

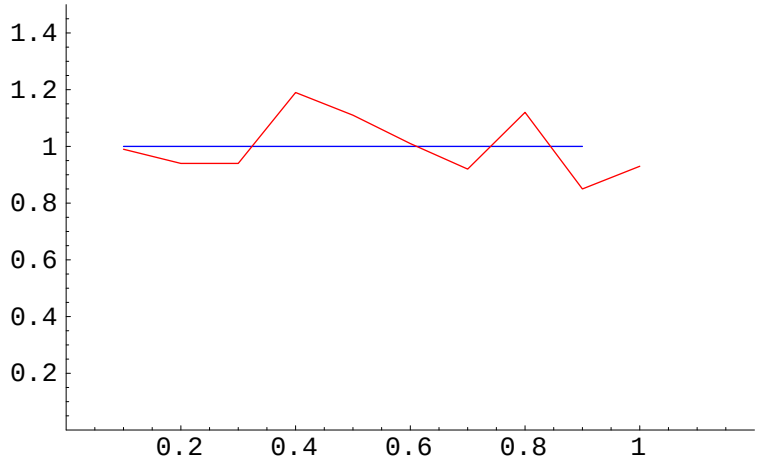
So far, we have good agreement with what we expect--about half (500/1000) of the samples should be less than 0.5. We can make a better comparison by comparing the plots of the histogram from the sampling experiment with the theoretical prediction. Let's make a table that summarizes the frequency. We do this by testing each sample to see if it lies within the bin range between x and x + 0.1. We count up how many times this is true to make a histogram.

```
bin = 0.1;
Freq = Table[Count[Thread[x<z<=x+bin],True],{x,0,1-bin,bin}];
```

Now we will plot up the results. Note that we normalize the **Freq** values by the number values in z using **Length[]**.

```
i=1;
theoreticalz = Table[{x,PDF[udist,x]}, {x,bin,.99,bin}];
simulatedz = Table[{x,(1/bin) N[Freq/Length[z]][[i++]]},
                        {x,bin,1,bin}];
theoreticalg = ListPlot[theoreticalz,
        PlotJoined->True, PlotStyle->{RGBColor[0,0,1]},
        DisplayFunction->Identity, PlotRange->{{0,1.2},{0,1.5}}];
simulatedg = ListPlot[simulatedz,
        PlotJoined->True, PlotStyle->{RGBColor[1,0,0]},
        DisplayFunction->Identity, PlotRange->{{0,1.2},{0,1.5}}];
```

```
Show[theoreticalg,simulatedg,
    DisplayFunction->$DisplayFunction];
```



As you can see, the computer simulation matches fairly closely what theory predicts.

■ Central Limit theorem Demonstration

Now we'd like to see what happens when we make new random numbers by adding up the squares of uniformly distributed ones. Why squares? It turns out that our observation below doesn't matter what we do the uniformly distributed numbers, and squaring is easy to do.

Let's define a function, **rv**, that generates random 20-dimensional vectors whose elements are uniformly distributed between 0.5 and -0.5.

```
rv := Table[Random[Real]-0.5,{i,1,20}];
ipsample = Table[rv.rv,{5000}];
```

ipsample is a list of 5000 elements, each of which is the dot product of a random vector.

```
bin = 0.1;
Freq = Table[Count[Thread[x<ipsample<=x+bin],True],{x,-1,1-bin,bin}];
```

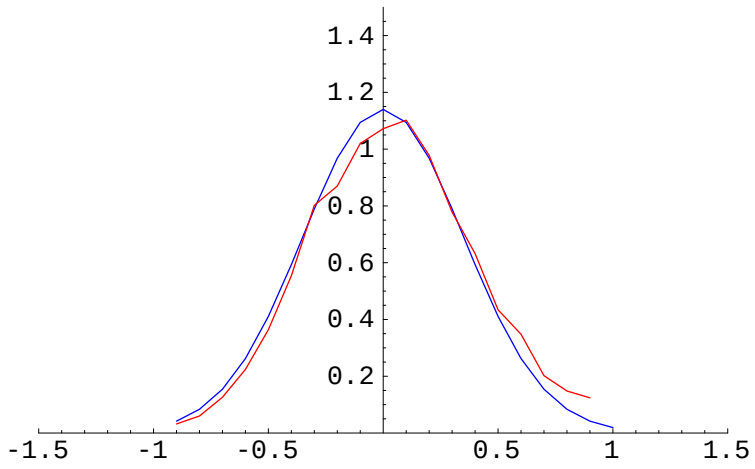
```
i=1;
simulatedz = Table[{x,(1/bin) N[Freq/Length[ipsample]][[i++]]},
    {x,-1+bin,1-bin,bin}];
simulatedg = ListPlot[simulatedz,
        PlotJoined->True, PlotStyle->{RGBColor[1,0,0]},
        DisplayFunction->Identity, PlotRange->{{-1.5,1.5},{0,1.5}}];
```

Now what is the theoretical distribution? The **Central Limit Theorem** states that the sum of n independent random variables approaches the Gaussian distribution as n gets large. The n independent random variables can come from any "reasonable" distribution-- the uniform distribution is reasonable, so is the distribution of the random variable z = x.y, where x and y are uniform random variables.

We don't know (although we could do some theory to find out) what the standard deviation of the theoretical distribution is, but it should be normal by the Central Limit Theorem. And we know the mean has to be zero, by symmetry. So we can try out various theoretical standard deviations to see what fits the simulation best:

```
standdev = 0.35;
ndist = NormalDistribution[0,standdev];
theoreticalz = Table[{x,PDF[ndist,x]}, {x,-1+bin,1,bin}];
theoreticalg = ListPlot[theoreticalz,
        PlotJoined->True, PlotStyle->{RGBColor[0,0,1]},
        DisplayFunction->Identity, PlotRange->{{-1.5,1.5},{0,1.5}}];
```

```
Show[theoreticalg,simulatedg,
    DisplayFunction->$DisplayFunction];
```



Exercise

Instead of generating samples of the inner product of random vectors, add up the elements:

```
rv := Table[Random[Real]-0.5,{i,1,20}];
ipsample = Table[Apply[Plus,rv],{5000}];
```

Calculate what the theoretical mean and standard deviation should be using the following rule:

1. The mean of a sum of independent random variables equals the sum of their means

2. The variance of a sum of independent random variables equals the sum of the variances

(And remember that the standard deviation equals the square root of the variance).

Plot up the simulated and theoretical distributions.

Summed vector memories
(see SVDMEMO in Anderson, chapter 7)

■ **How familiar is X, compared to what has been seen before?**

Matched filter (cross-correlator)

```
Imatrix = {
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

```
Tmatrix = {
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 1, 1, 1, 1, 1, 1, 1, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

```
General::spell1:
   Possible spelling error: new symbol name "Tmatrix"
     is similar to existing symbol "Imatrix".

General::spell1:
   Possible spelling error: new symbol name "Tmatrix"
     is similar to existing symbol "Imatrix".

General::spell1:
   Possible spelling error: new symbol name "Tmatrix"
     is similar to existing symbol "Imatrix".
```

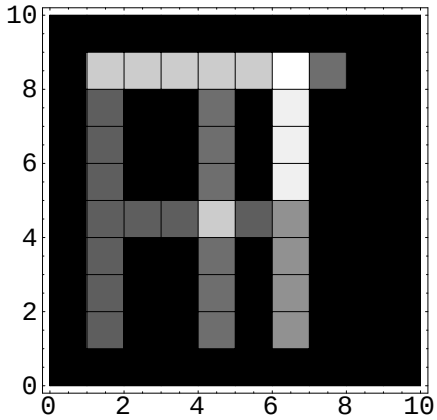
```
Pmatrix = {
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 1, 1, 1, 1, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 1, 1, 1, 1, 1, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

```
Xmatrix = {
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 0, 0, 1, 0},
  {0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
  {0, 0, 0, 1, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 1, 0, 0, 1, 0, 0, 0},
  {0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
  {0, 1, 0, 0, 0, 0, 0, 0, 1, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

```
normalize[x_] := N[x/Sqrt[x.x]];
Tv = normalize[Flatten[Tmatrix]];
Iv = normalize[Flatten[Imatrix]];
Pv = normalize[Flatten[Pmatrix]];
Xv = normalize[Flatten[Xmatrix]];
```

```
sv = Tv+Iv+Pv;
```

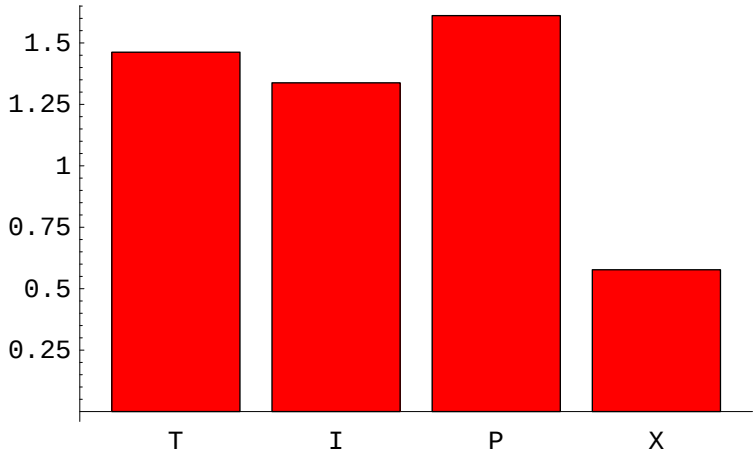
```
ListDensityPlot[Partition[sv,10]];
```



Let's look at the outputs of the summed vector memory to the three inputs it has seen before (T,I,P) and to a new input X. We will use a *Mathematica* graphics package that has some extra plot styles in it--in particular, the **BarChart[]**.

```
<<Graphics`Graphics`
```

```
matchedfilterout = {sv.Tv,sv.Iv,sv.Pv,sv.Xv};
BarChart[matchedfilterout,BarLabels->{"T","I","P","X"}];
```



What happens if the vector memory has seen many I's and P's, but only one T?

```
sv = Tv + 20 Iv+ 10 Pv;
matchedfilterout = {sv.Tv,sv.Iv,sv.Pv,sv.Xv};
BarChart[matchedfilterout,BarLabels->{"T","I","P","X"}];
```

Introduction to non-linear models

Statistical learning theory

A common distinction in neural networks is between supervised and unsupervised learning. The heteroassociative network was supervised, in the sense that a "teacher" supplies the proper output to associate with the input. Learning in autoassociative networks is unsupervised in the sense that they just take in inputs, and try to organize an internal representation based the inputs.

Over the past decade, there has been considerable progress in establishing the theoretical foundations of neural networks in the larger domain of statistical learning theory. In particular, neural networks can be seen to be solving several standard problems in statistics: regression, classification, and probability density estimation. Here is a summary:

Supervised learning: Training set $\{\mathbf{f}_i,\mathbf{g}_i\}$

Regression: Find a function ϕ: **f**->**g**, i.e. where **g** takes on continuous values.

Classification: Find a function ϕ:**f**->{**0,1,2,...,n**}, i.e. where $\mathbf{g}_i$ takes on discrete values or labels.

Unsupervised learning: Training set $\{\mathbf{f}_i\}$

Estimate probability density: p(**f**), e.g. so that the statistics of p(**f**) match those of $\{\mathbf{f}_i\}$, but generalizes well beyond the data.

In general, **f** is a vector whose elements may depend on each other, so density estimation is a hard problem, and involves much more than compiling histograms.

Many problems require discrete decisions. A problem with linear regression networks that we've studied so far is that they don't.

Next time we will take a look at the binary classification problem:

ϕ: **f** -> {0,1}

■ Perceptron (Rosenblatt, 1958)

The original perceptron was fairly sophisticated--input layer ("retina" of sensory units), associator units, and response units. There was feedback between associator and response units.

The neuron models were threshold logic units (TLU)--i.e. the generic connectionist unit with a step threshold function.

These networks were difficult to analyse theoretically, but a simplified single-layer perceptron can be analyzed. Next lecture we will look at linear separability , the perceptron learning rule, and the work of Minsky and Papert (1969).

References

Bishop, C. M. (1995). Neural Networks for Pattern Recognition. Oxford: Oxford Univeristy Press.

Duda, R. O., & Hart, P. E. (1973). Pattern classification and scene analysis. New York.: John Wiley & Sons.

Vapnik, V. N. (1995). The nature of statistical learning. New York: Springer-Verlag.