Introduction to Neural Networks
U. Minn. Psy 5038

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

Initialization

In[262]:=
```
Off[SetDelayed::write]
Off[General::spell1]
```

Introduction

Last time

A common distinction in neural networks is between supervised and unsupervised learning ("self-organization"). The heteroassociative network is 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 a useful internal representation based the inputs. What "useful" means depends on the application. We explored the idea that if memories are stored with autoassociative weights, it is possible to later "recall" the whole pattern after seeing only part of the whole.

■ Simulations

Heterassociation

Autoassociation

Superposition and interference

Today

- **Summed vector memories**

- **Introduction to statistical learning**

- **Generative modeling and statistical sampling**

Summed vector memories

Generalized Hebb rule

Taylor series expansion

Recall that a smooth function h(x) can be expanded in a Taylor's series:

```
Series[h[x], {x, 0, 3}]
```

$$h(0) + h'(0)\,x + \frac{1}{2}\,h''(0)\,x^2 + \frac{1}{6}\,h^{(3)}(0)\,x^3 + O(x^4)$$

where we've used **Series[ ]** to write out terms to order 3, and `O[x]`^{4} means there are more terms (potentially infinitely more), but whose values are fall off as the fourth power of x (and higher), so are small for x<1. `h`$^{(n)}$`[0]` means the nth derivative of h evaluated at x=0.

If we only include terms up to first order, this corresponds to approximating h[x] near x=0 by a straight line. What if we have a surface h[x,y]? We can approximate it near (0,0) by a quadratic surface:

In[265]:=
```
Series[h[x, y], {x, 0, 2}, {y, 0, 2}]
```

Out[265]=
$$\left(h(0,0)+h^{(0,1)}(0,0)\,y+\frac{1}{2}h^{(0,2)}(0,0)\,y^2+O(y^3)\right)+\left(h^{(1,0)}(0,0)+h^{(1,1)}(0,0)\,y+\frac{1}{2}h^{(1,2)}(0,0)\,y^2+O(y^3)\right)x+\left(\frac{1}{2}h^{(2,0)}(0,0)+\frac{1}{2}h^{(2,1)}(0,0)\,y+\frac{1}{4}h^{(2,2)}(0,0)\,y^2+O(y^3)\right)x^2+O(x^3)$$

Using Normal[ ] to remove the order expressions:

In[266]:=
```
Normal[%]
```

Out[266]=
$$\frac{1}{2}h^{(2,0)}(0,0)\,x^2+h^{(1,0)}(0,0)\,x+h(0,0)+y\left(\frac{1}{2}h^{(2,1)}(0,0)\,x^2+h^{(1,1)}(0,0)\,x+h^{(0,1)}(0,0)\right)+y^2\left(\frac{1}{4}h^{(2,2)}(0,0)\,x^2+\frac{1}{2}h^{(1,2)}(0,0)\,x+\frac{1}{2}h^{(0,2)}(0,0)\right)$$

Note above that the terms with x and y never appear with exponents bigger than 2, hence "quadratic".

But we can approximate h with even fewer terms, as a plane around the origin:

```
Series[h[x, y], {x, 0, 1}, {y, 0, 1}]
```

■ The "generalized Hebb rule":

In general, we might model the change in synaptic weights between neuron i and j by `ΔW[fi, gj]`, where as before `fi, gj` are scalars representing the pre- and post-synaptic neural activities. Then with `ΔW[fi, gj]` playing the role of h[x,y] in the above expansion, we have that `ΔW[fi, gj]` is approximately equal to:

```
Expand[Normal[Series[ΔW[fi, gj], {fi, 0, 1}, {gj, 0, 1}]]]
```

$$\Delta W(0,0)+g_j\,\Delta W^{(0,1)}(0,0)+f_{20}\,\Delta W^{(1,0)}(0,0)+f_{20}\,g_j\,\Delta W^{(1,1)}(0,0)$$

(where again we've used **Normal[ ]** to remove O[ ] terms from the expression, and **Expand[ ]** to expand out the products). By generalizing the learning rule to any smooth function, we see that the Hebbian rule used in the linear associator models (both auto and heteroassociation) corresponds to using only the last term.

What if we used just the first term? In other words, suppose learning depended only on the input strength `fi`? Is there any useful function for such "strengthening-by-use" synapses? Suppose we have a set of k *normalized* input vectors $\{f^k\}$. The learning rule says that the synaptic weights **w**, for a single neuron would be

$$\mathbf{w}=\sum_{\mathbf{k}}\mathbf{f}^{\mathbf{k}} \qquad (1)$$

Learning is easy, but it seems that the information about the set of input vectors is pretty messed up due to superposition. There are two cases where a template matching operation (i.e. dot product) could pull out useful information. Consider,

$$\mathbf{w}.\mathbf{f}^l = \sum_k \mathbf{f}^l.\mathbf{f}^k = \mathbf{f}^l.\mathbf{f}^l + \sum_{l\neq k} \mathbf{f}^l.\mathbf{f}^k \tag{2}$$

We know that $\mathbf{f}^l.\mathbf{f}^l > \mathbf{f}^l.\mathbf{f}^k$ for l≠k, but potentially we have lots of terms in the sum that could swamp out $\mathbf{f}^l.\mathbf{f}^l$. However, if their directions are randomly distributed, we could have many cancellations. Later you'll see that random large dimensional vectors tend to be orthogonal.

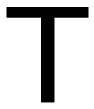
Further, suppose one of the inputs appears more frequently than the others, say $\mathbf{f}^l$, then this term would dominate the sum **w**, and we might expect that a template matching operation ($\mathbf{w}.\mathbf{f}^l$) could provide information that a high output neuron in effect is saying "yes, this input pattern looks like something I've seen frequently"). Conversely, an unusually low value of the dot product would mean that "...mm this is novel, maybe I should pay more attention to this one". One potential disdvantage is that a high rate of firing would be the norm, and there is substantial evidence that the cortex of the brain is very economical when it comes to "spending" spikes.

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

■ Learning: input vector sums

Let's simulate the case where $\triangle W[f_i, g_j] \propto f_i$.

In[267]:=
```
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}};
```



In[268]:=
```
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}};
```

P

In[269]:=
```
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}};
```

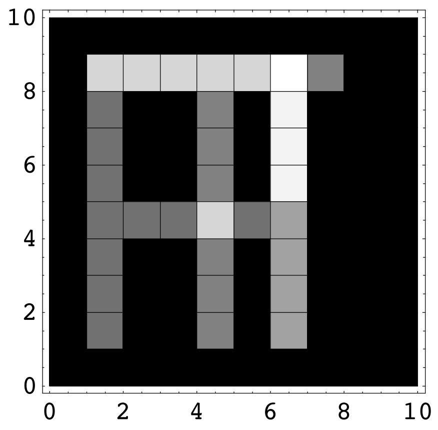
X

In[270]:=
```
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}};
```

In[271]:=
```
normalize[x_] := N[x/Sqrt[x.x]];
Tv = normalize[Flatten[Tmatrix]];
Iv = normalize[Flatten[Imatrix]];
Pv = normalize[Flatten[Pmatrix]];
Xv = normalize[Flatten[Xmatrix]];
```

In[276]:=
```
sv = Tv+Iv+Pv;
```

In[277]:=
```
ListDensityPlot[Partition[sv,10]];
```

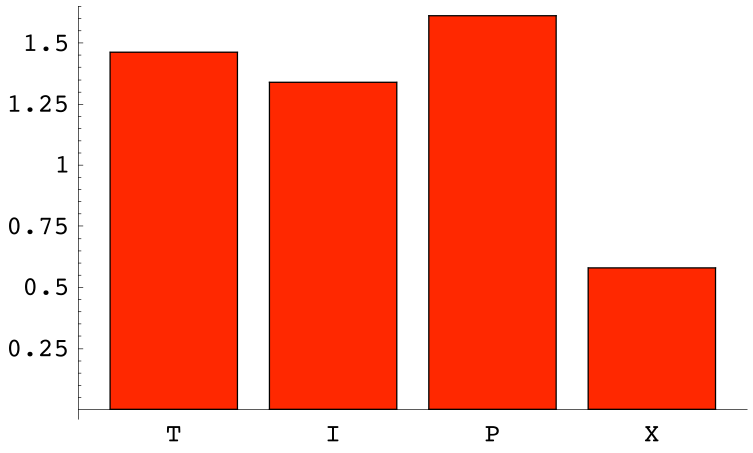


■ Recall: Matched filter (cross-correlator)

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[]**.

In[278]:=
```
<<Graphics`Graphics`
```

In[279]:=
```
matchedfilterout = {sv.Tv,sv.Iv,sv.Pv,sv.Xv};
BarChart[matchedfilterout,BarLabels->{"T","I","P","X"}];
```



■ Signal-to-noise ratio

How well does the output separate the familiar from the unfamiliar (X)? We'd like to compare the output of the model neuron when the input is the novel stimulus X, vs. the output we might expect for familiar inputs. There are several ways of summarizing performance, but one simple formula calculates the ratio of the squared output to the average squared input.

We first read in an add-on statistics package that provides us with extra functions, including **Mean[ ]**. (**Mean[ ]** is a built-in function from version 5 on).

```
In[281]:= << Statistics`DescriptiveStatistics`
```

```
In[284]:= (sv.Xv) ^2 / Mean [{(sv.Tv) ^2, (sv.Iv) ^2, (sv.Pv) ^2}]

Out[284]= 0.153137
```

■ Center vectors about zero , i.e. each has zero mean

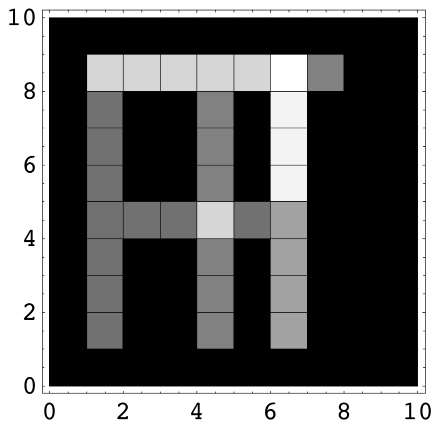
In the above representation of the letters, all the input vectors lived in the positive "quadrant", so their dot products are all positive. What if we center the vectors about zero?

```
In[285]:= normalize[x_] := N[x/Sqrt[x.x]];
          Tv = normalize[Flatten[Tmatrix]];
          Iv = normalize[Flatten[Imatrix]];
          Pv = normalize[Flatten[Pmatrix]];
          Xv = normalize[Flatten[Xmatrix]];
```

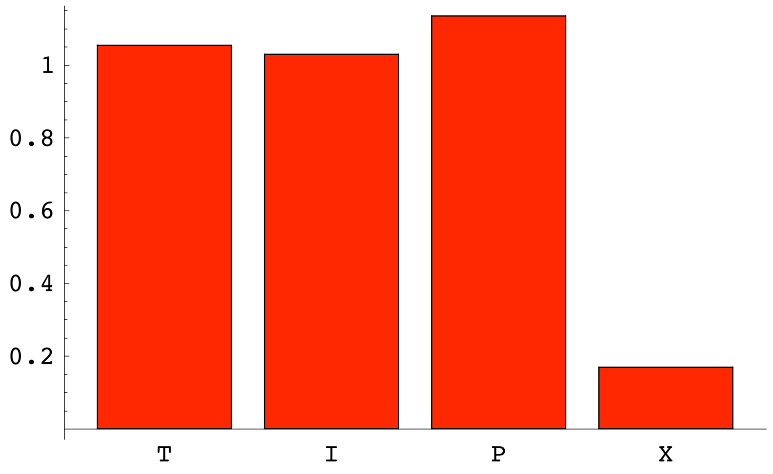
```
In[290]:= Tv = Tv - Mean[Tv];
          Iv = Iv - Mean[Iv];
          Pv = Pv - Mean[Pv];
          Xv = Xv - Mean[Xv];
```

```
In[294]:= sv = Tv+Iv+Pv;
```

In[295]:=
```
ListDensityPlot[Partition[sv,10]];
```



In[296]:=
```
matchedfilterout = {sv.Tv,sv.Iv,sv.Pv,sv.Xv};
BarChart[matchedfilterout,BarLabels->{"T","I","P","X"}];
```

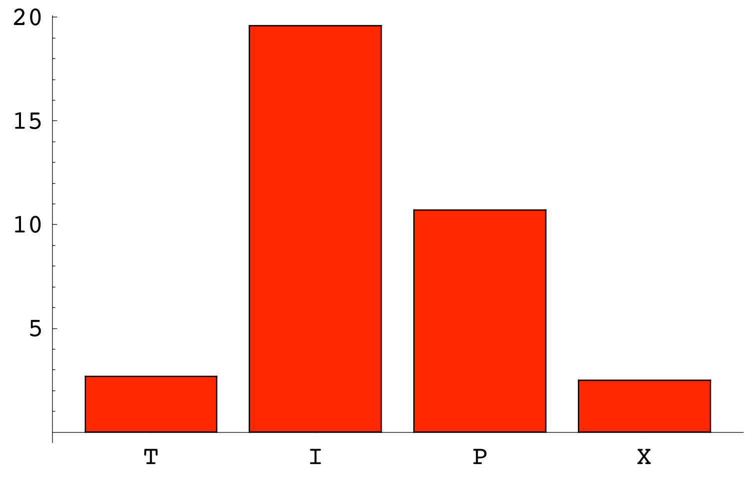


In[298]:=
```
(sv.Xv)^2/(Mean /@ {{(sv.Tv)^2, (sv.Iv)^2, (sv.Pv)^2}})
```

Out[298]=
```
{0.0245508}
```

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

In[299]:=
```
sv = Tv + 20 Iv+ 10 Pv;
matchedfilterout = {sv.Tv,sv.Iv,sv.Pv,sv.Xv};
BarChart[matchedfilterout,BarLabels->{"T","I","P","X"}];
```



Side-note: Optimality of matched filter

The field of signal detection theory has shown that if one is given a vector input **x**, and required to detect whether it is due to a signal in noise (**s+n**), or just noise (**n**), then under certain conditions, one cannot do any better than to base one's decision on the dot product **x.s**. The conditions are simple: the elements of the noise vector are assumed to be identical and independently distributed gaussian random variables, and **s** is assumed to be known exactly (i.e. is represented by a vector whose elements have fixed values).

Learning information about the relative frequencies

We've seen how a very simple form of the generalized Hebbian learning rule can capture useful information about the relative frequencies of stimulus occurrence. This is a crude form of self-organization. We know from statistics that there are standard devices for estimating frequency of occurrence--namely, histograms. The vector sum has accumulated a kind of histogram, in the sense that it counts the number of times a particular synapse has been activated. But it is sub-optimal for our function, because what we'd like to have ideally is a device that told us how often **T**s, **I**s, **P**s occur, in a way that doesn't muddle up their representational elements.

Later we will look at the statistical framework for self-organization and the problem of measuring histograms and using these data to model probablity densities, or "density estimation" as it is called.

Overview of Statistical learning theory

Statistical learning theory

We've noted that 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 on the inputs.

Over the past decade or so, 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:

Supervised learning: Training set $\{\mathbf{f_i},\mathbf{g_i}\}$

Regression: Find a function ϕ: $\mathbf{f}$->$\mathbf{g}$, i.e. where $\mathbf{g}$ takes on continuous values. Fitting a straight line to data is a simple case.

Classification: Find a function ϕ:$\mathbf{f}$->$\{\mathbf{0,1,2,...,n}\}$, i.e. where $\mathbf{g_i}$ takes on discrete values or labels. Face recognition is an example.

Many problems require discrete decisions. A problem with linear regression networks that we've studied so far is that they don't. Below there is a simple exercise to illustrate the how the linear associator deals with inputs that it hasn't seen before.

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

ϕ: $\mathbf{f}$ -> $\{0,1\}$

and see how the Perceptron solves it:

■ Unsupervised learning:

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 beyond the data.

In general, **f** is a vector with many elements that may depend on each other, so density estimation is a hard problem, and involves much more than compiling histograms of the frequency of the individual vector elements. One also needs the histograms for the joint occurrence of all pairs, all triplets, etc.. One quickly runs into so-called combinatorial explosion for the number of bins. But the problem is further compounded by the lack of enough samples to fill them. (Imagine you want to build a probability table for all 4x4 patches sampled from on-line digital 8 bit pictures. The pictures are really tiny, and there are only 16 pixels for each picture. But your histogram would need 256^16 bins. Calculate it!)

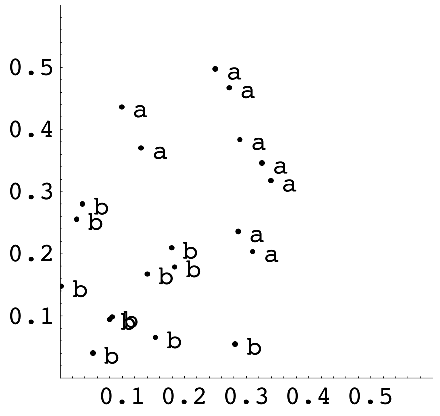
Synthesis of random textures or mountain landscapes is an example from computer graphics.

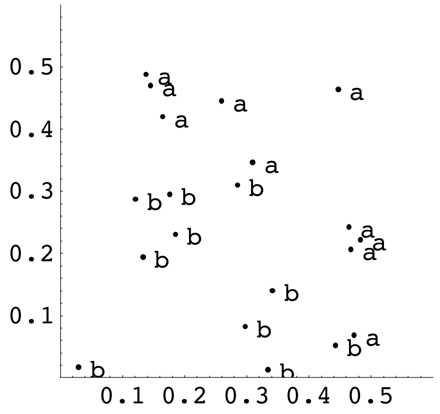
Generative modeling and Statistical sampling

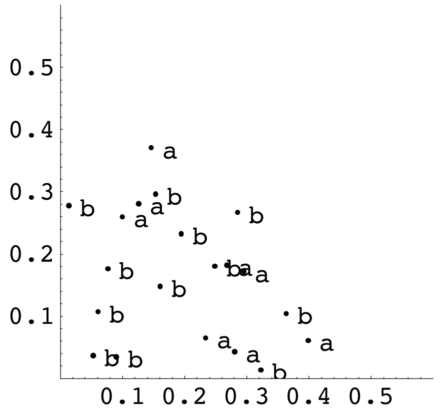
■ Generative modeling

Whether and how well a particular learning method works depends on how the data is generated. We spend most of our time thinking about how to model learning and inference, i.e. estimation and classification. But it is also important to understand how to model the regularities in incoming data. A powerful way to do this is to develop "generative models" that when implemented produce artificial data that resembles what the real data looks like. This corresponds to the statistics problem of "filling a hat with the appropriate slips of paper" and then "drawing samples from the hat".

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 for regression or classification. 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 determine 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.

In[312]:=
```
<<Statistics`DiscreteDistributions`
<<Statistics`ContinuousDistributions`
```

In[314]:=
```
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:

In[315]:=
```
sample[ntimes_] :=
    Table[Random[Real],{ntimes}];
```

Or like this:

In[316]:=
```
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.

In[317]:=
```
z = sample[1000];
```

Count up how many times the result was 0.5 or less. To do this, we will use two built-in functions: **Count[]**, and **Thread[]**. You can obtain their definitions using the ?? query.

In[318]:=
```
Count[Thread[z<=.5],True]
```

Out[318]=
```
520
```

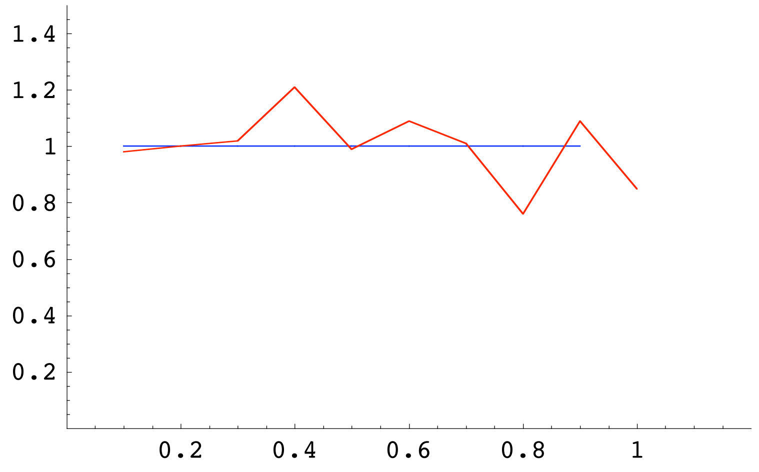
So far, we have good agreement with what we expect--about half (500/1000) of the samples should be less than 0.5. We can make a better comparison by comparing the plots of the histogram from the sampling experiment with the theoretical prediction. Let's make a table that summarizes the frequency. We 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.

In[323]:=
```
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[]**.

In[325]:=
```
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}}];
```

In[330]:=
```
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 several uniformly distributed numbers.

Let's define a function, **rv**, that generates random **nrv**-dimensional vectors whose elements are uniformly distributed between 0.5 and -0.5. Try nrv =1. Then try nrv=3.

In[359]:=
```
nrv=1;
udist = UniformDistribution[-.5,.5];
rv := Table[Random[udist],{i,1,nrv}];
ipsample = Table[Apply[Plus,rv],{10000}];
```

ipsample is a list of 10000 elements.

In[363]:=
```
bin = 0.1;
Freq = Table[Count[Thread[x<ipsample<=x+bin],True],{x,-1,1-bin,bin}];
```

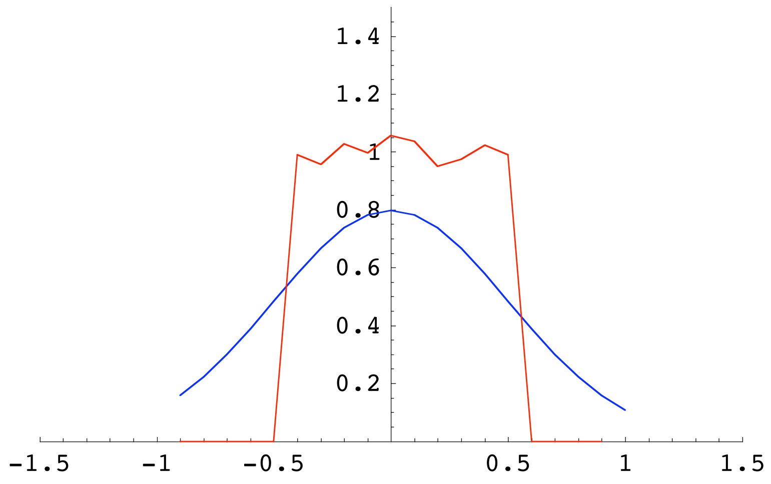
In[365]:=
```
i=1;
simulatedz = Table[{x,(1/bin) N[Freq/Length[ipsample]][[i++]]},
    {x,-1+bin,1-bin,bin}];
simulatedg = ListPlot[simulatedz,
        PlotJoined->True, PlotStyle->{RGBColor[1,0,0]},
        DisplayFunction->Identity, PlotRange->{{-1.5,1.5},{0,1.5}}];
```

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 can do some theory to find out, see below) 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:

In[368]:=
```
standdev=0.5;
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}}];
```

In[372]:=
```
Show[theoreticalg,simulatedg,
    DisplayFunction->$DisplayFunction];
```



Now try nrv = 3 above.

Again, our basic observation--tendency towards a bell-shaped distribution for sums--doesn't depend on the type of random number distribution-we'll get a distribution of the new random number that looks like a bell-shaped curve if we add up enough of the independently and identically sampled ones (short hand is i.i.d. for *independently and identically distributed*).

Exercises

Exercise

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 above using your answer for the theoretical distribution.

Answer for the standard deviation is in the closed cell below.

Exercise

Try using `LaplaceDistribution[ mu, beta ]` instead of the UniformDistribution as the source of the i.i.d. random variables.

Linear interpolation interpretation of linear heteroassociative learning and recall

```
In[373]:= << Graphics`Graphics3D`
```

```
In[374]:= f1 = {0, 1, 0};
f2 = {1, 0, 0};
g1 = {0, 1, 3};
g2 = {1, 0, 5};
```

```
W = Outer[Times, g1, f1];
```

W maps **f1** to **g1**:

```
W.f1
```

```
{0, 1, 3}
```

W maps f2 to g2:

```
W = Outer[Times, g2, f2];
W.f2
```

```
{1, 0, 5}
```

Because of the orthogonality of f1 and f2, the sum **Wt** still maps **f1** to **g1**:

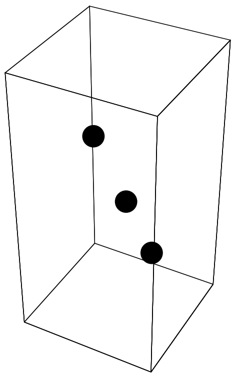
```
Wt = Outer[Times, g1, f1] + Outer[Times, g2, f2];
Wt.f1
```

```
{0, 1, 3}
```

Define an interpolated point **fi** somewhere between **f1** and **f2**, the position being determined by parameter a:

```
a = 0.4;
fi = a * f1 + (1 - a) * f2;
```

```
ScatterPlot3D[{f1, f2, fi}, PlotStyle -> PointSize[.1], Axes → False];
```



Now define an interpolated point **gt** between **g1** and **g2**

```
gt = a * g1 + (1 - a) * g2;
```

Show that **Wt** maps **fi** to **gt**:

```
Wt.fi
gt
```

Next time: Introduction to non-linear models

■ Perceptron (Rosenblatt, 1958)

The original Perceptron was a neural network architecture for

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

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

These networks were difficult to analyze theoretically, but a simplified single-layer perceptron can be analyzed. Next lecture we will look at classification, 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 University 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.