Introduction to Neural Networks

Sampling, generative models
Summed vector memories

Initialization

```
Off[SetDelayed::write]
Off[General::spell1]
SetOptions[ListPlot, Joined → True];
SetOptions[ArrayPlot, ColorFunction → "GrayTones",ImageSize->Tiny];
```

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

- **Introduction to statistical learning**
- **Generative modeling and statistical sampling**

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 shared regularities among the inputs.

Statistical learning theory (or machine learning) provides a theoretical foundation for neural networks. In particular, neural networks can be interpreted as solving several standard problems in statistics: regression, classification, and probability density estimation. Here is a summary:

- **Supervised learning:**

Supervised learning: Training set $\{f_i, g_i\}$, where i =1,..,N.

Regression: Find a function ϕ: f -> g, i.e. where g takes on continuous values. Fitting a straight line to data is a simple case. The weight matrix W in the generic neural network model is a particular case of ϕ.

Below there is a simple exercise to illustrate the how the linear associator deals with inputs that it hasn't seen before.

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

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

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

ϕ: f -> {0,1}

and step back in history to see how the *Perceptron* learns to find the weights that determine the mapping ϕ.

- **Unsupervised learning:**

Unsupervised learning: Training set $\{f_i\}$

Estimate probability density: p(f), e.g. so that the statistics of p(f) match those of $\{f_i\}$, but generalizes

beyond the data.

If f is a one dimensional scalar, then estimating p(f) is the familiar problem if compiling a histogram, i.e. a table of the frequency of occurrence of the values of f that lie in a small interval Δf. Or if f is discrete, the frequency of occurrence of values of f.

In general, f is a vector with many elements that may depend on each other. Then density estimation becomes 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!) The solution to this problem is to assume a "parametric form" for the distribution in which just a few parameters (e.g. mean and standard deviation, or mean and covariance) need to be estimated.

Synthesis of random textures or mountain landscapes is an example from computer graphics where histograms have been measured for key features, and then given the histograms, one can draw random samples, where each sample is a new texture, etc.. Having the histograms and a method for drawing samples is an example of a **generative model** for the data.

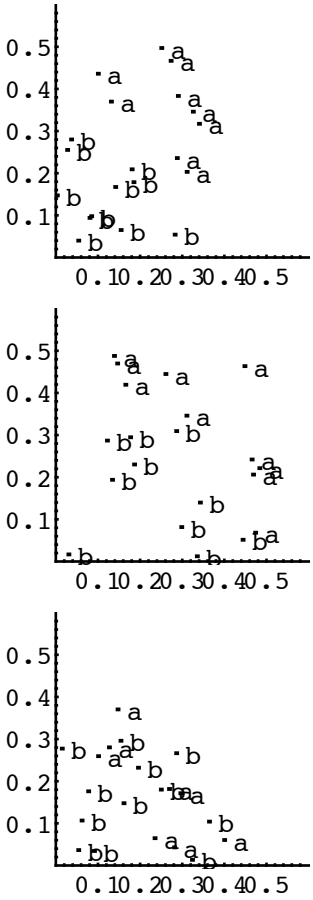
Generative modeling and Statistical sampling

■ Generative modeling

Whether and how well a particular learning method works depends on how the data is generated. Usually we don’t know. However, it is important to understand why a learning model succeeds or fails. It helps 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". Counts of the relative proportions of the different types of slips of paper in the hat represents a histogram.

We will be 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. And later we'll see how the structure of the generative model determines optimal inference.

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, where a straight line can be drawn between the two classes, 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, one can show 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 get better that the cosine of the angle between any two random vectors is close to zero. The histograms for the distributions of the cosines of random vectors for dimensions 10, 50 , and 250 get progressively narrower.

```
Dot[r1 = RandomReal[{-1, 1}, 2], r2 = RandomReal[{-1, 1}, 2]] /
 (Norm[r1] * Norm[r2])
Dot[vr1 = RandomReal[{-1, 1}, 5000], vr2 = RandomReal[{-1, 1}, 5000]] /
 (Norm[vr1] * Norm[vr2])
```

```
0.43314
```

```
0.00173772
```

■ Probability densities and discrete distributions

As we noted earlier, most 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 (or particular Δx for densities) is the same over the sampling range.

(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.

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

```
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_] := RandomReal[{0, 1}, ntimes];
```

Or like this:

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

Now let us do a sampling experiment to get a list of 1000 *identically distributed* (also in this case uniformly), *identically drawn*, random samples.

```
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.

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

```
498
```

You can also use the Map function:

```
Count[Map[# ≤ .5 &, z], True]
Count[# ≤ .5 & /@ z, True]
(*Short hand for the above*)
```

```
498
```

```
498
```

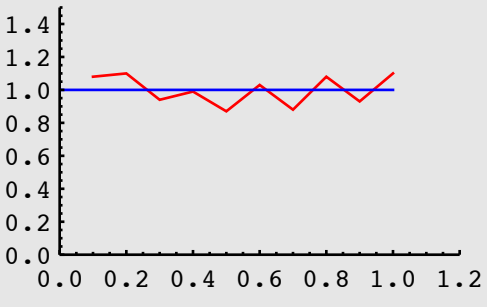
So far, we have good agreement with what we expect--about half (500/1000) of the samples should be less than or equal to 0.5. We can make a better comparison by comparing the plots of the histogram from the sampling experiment with the theoretical prediction. Let's make a table that summarizes the frequency.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

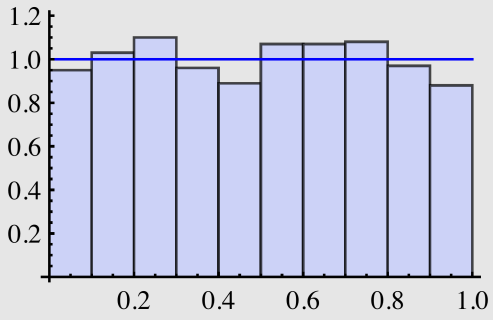
```
i = 1; theoreticalz = Table[{x, PDF[udist, x]}, {x, 0, 0.99 + bin, bin}];
simulatedz = Table[{x, N[Freq/Length[z]][[i++]]/bin}, {x, bin, 1, bin}];
theoreticalg = ListPlot[theoreticalz, Joined → True,
  PlotStyle → {RGBColor[0, 0, 1]}, PlotRange → {{0, 1.2}, {0, 1.5}}];
simulatedg = ListPlot[simulatedz, Joined → True,
  PlotStyle → {RGBColor[1, 0, 0]}, PlotRange → {{0, 1.2}, {0, 1.5}}];
```

```
Show[simulatedg, theoreticalg, ImageSize → Small]
```



We calculated the histogram using several basic functions, but *Mathematica* has a built-in function Histogram[] to do it for us and produce a bar chart:

```
Show[{Histogram[z, {.1}, "ProbabilityDensity"], theoreticalg},
 ImageSize → Small]
```



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

■ Demonstration of the *Central Limit Theorem*

Now we'd like to see what happens when we make new random numbers by adding up several uniformly distributed numbers.

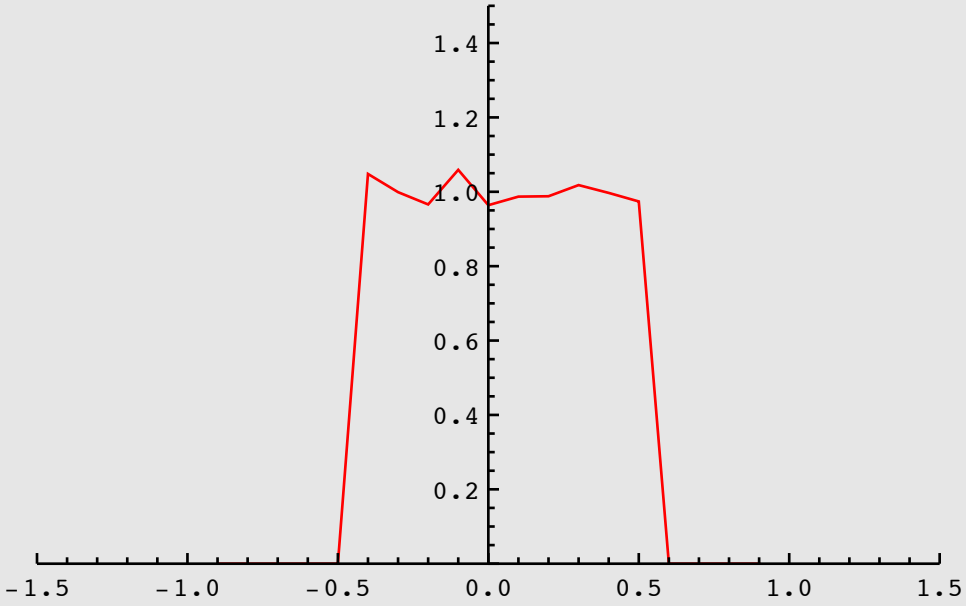
Let's define a function, **rv**, that generates random **nrv**-dimensional vectors whose elements are uniformly distributed between 0.5 and -0.5. **nrv** is the number of uniformly distributed samples that we add together. Try **nrv** =1. Then try **nrv**=3.

```
nrv=1;
udist = UniformDistribution[{-.5,.5}];
rv := Table[Random[udist],{i,1,nrv}]; (*Each rv has dimension nrv*)
ipsample = Table[Apply[Plus,rv],{10000}];(*We draw a sample rv, and then add up a
    elements of rv. Then do it again, for a total of 10,000 times *)
```

ipsample is a list of 10000 elements. Let's calculate the histogram as we did above.

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

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



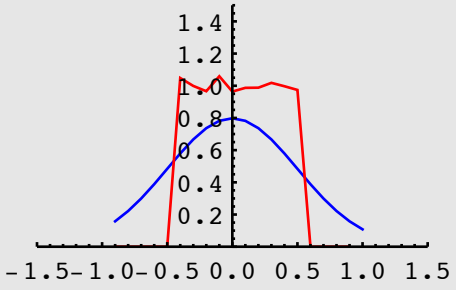
Now what is the theoretical distribution?

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 (finite mean and variance)-- the uniform distribution is reasonable, so is for example the distribution of the random variable z = x.y, where x and y are uniform random variables, and many others that you could make up. The second condition that needs to be satisified is that the variables should be from the same distribution--i.e. "identically distributed". These two conditions are often called ***i.i.d.*** which stands for *independent and identially distributed*. Or in other words, each random number or vector is drawn (as in “pulled out of a hat”) independently of any of the other draws, and each one comes from same underlying distribution.

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 the distribution should approach 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 = 0.5`; ndist = NormalDistribution[0, standdev];
theoreticalz = Table[{x, PDF[ndist, x]}, {x, -1 + bin, 1, bin}];
theoreticalg = ListPlot[theoreticalz, Joined → True,
  PlotStyle → {RGBColor[0, 0, 1]}, DisplayFunction → Identity,
  PlotRange → {{-1.5`, 1.5`}, {0, 1.5`}}];
```

```
Show[theoreticalg, simulatedg, DisplayFunction → $DisplayFunction]
```



Now try making nrv the sum of three draws, instead of just one, i.e. let 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 drawn, and identically distributed ones.

Next time: Introduction to non-linear models

■ Perceptron (Rosenblatt, 1958)

The original Perceptron was a neural network architecture in which the neuron models were threshold logic units (TLU)--i.e. the generic connectionist unit with a step threshold function.

The perceptron proposed by Rosenblatt was fairly complicated, consisting of an 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).

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

```
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, and f2 to g2:

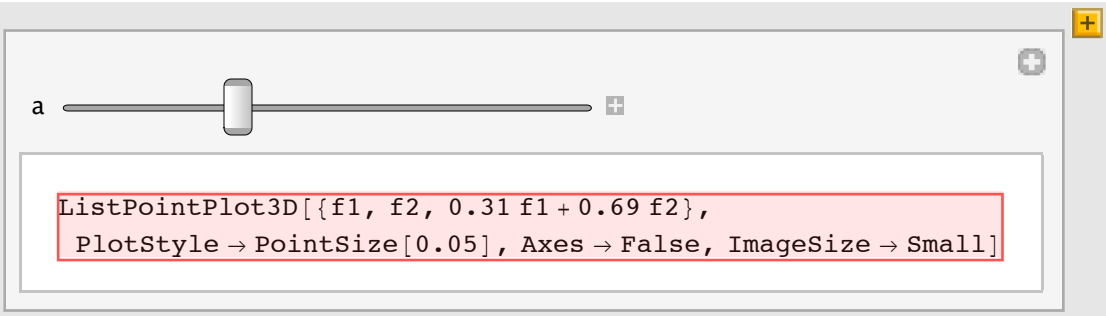
```
Wt = Outer[Times, g1, f1] + Outer[Times, g2, f2];
Wt.f1
Wt.f2
```

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

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

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

```
Manipulate[
 fi = a * f1 + (1 - a) * f2;
 aa = a;
 ListPointPlot3D[{f1, f2, fi}, PlotStyle → PointSize[0.05],
  Axes → False, ImageSize → Small], {{a, .4}, 0, 1}]
```



Now define an interpolated point gt between g1 and g2 using the value of **aa**:

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

Show that Wt maps fi to gt:

```
Wt.fi
gt
```

```
{0.69, 0.31, 4.38}
```

```
{0.69, 0.31, 4.38}
```

Appendix

Summed vector memories

We' ve assumed a Hebbian learning rule : $\triangle W = f_i\, g_j$. How could we model a more complex, or simpler rule? $\triangle W\left(f_i\, g_j\right)$?

Generalized Hebb rule

■ Taylor series expansion

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

```
Clear[h, g, f, x, i, j]
```

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

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

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. If we include terms up to second order, then the approximation would be a quadratic line.

What if we have a surface h[x,y]? We can approximate it near (0,0) by a quadratic surface:

```
Series[h[x, y], {x, 0, 2}, {y, 0, 2}]
```

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

Using Normal[ ] to remove the order expressions:

```
Normal[%]
```

$$y^2\left(\frac{1}{4}x^2\,h^{(2,2)}(0,0)+\frac{1}{2}x\,h^{(1,2)}(0,0)+\frac{1}{2}h^{(0,2)}(0,0)\right)+$$
$$y\left(\frac{1}{2}x^2\,h^{(2,1)}(0,0)+x\,h^{(1,1)}(0,0)+h^{(0,1)}(0,0)\right)+\frac{1}{2}x^2\,h^{(2,0)}(0,0)+x\,h^{(1,0)}(0,0)+h(0,0)$$

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}]
```

$$\left(h(0,0)+y\,h^{(0,1)}(0,0)+O\left(y^2\right)\right)+x\left(h^{(1,0)}(0,0)+h^{(1,1)}(0,0)\,y+O\left(y^2\right)\right)+O\left(x^2\right)$$

■ The "generalized Hebb rule":

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

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

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

(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 highest, i.e. second-order term.

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

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

Learning is easy, but it seems that the information about the set of input vectors is pretty messed up due to superposition. However, there are two cases where a template matching operation (i.e. dot product) could pull out useful information in the sense of giving a large response to a particular vector, say $\mathbf{f}^l$. Consider,

$$\mathbf{w}.\mathbf{f}^l=\left(\sum_k \mathbf{f}^k\right).\mathbf{f}^l=\sum_k \mathbf{f}^l.\mathbf{f}^k=\mathbf{f}^l.\mathbf{f}^l+\sum_{l\neq k}\mathbf{f}^l.\mathbf{f}^k \qquad (2)$$

We know that $\mathbf{f}^l.\mathbf{f}^l \geq \mathbf{f}^l.\mathbf{f}^k$ 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. So for example, compare the strength of the response of **W** to **f1, f2, f3, f4, f5** as compared to an arbitrary **f0** for 100-dimensional vectors:

```
f := Table[RandomReal[{-1, 1}], {100}];
```

```
f1 = f; f2 = f; f3 = f; f4 = f; f5 = f;
f0 = f;
W = f1 + f2 + f3 + f4 + f5;
```

Compare

```
{W.f1, W.f2, W.f3, W.f4, W.f5}
```

```
{25.7308, 28.2348, 30.487, 36.8068, 32.4119}
```

with

```
{f1.f1, f2.f2, f3.f3, f4.f4, f5.f5}
```

```
{31.0188, 31.6316, 30.6742, 36.5456, 31.522}
```

Now if we run a random pattern through W that it hasn't seen before, we tend to get a much lower value:

```
W.f0
```

```
13.1435
```

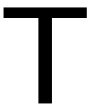
Further, suppose one of the inputs appears more frequently than the others, say f^l, then this term would dominate the sum **w**, and we might expect that a template matching operation ($\mathbf{w}.f^l$) could provide information that a high output neuron in effect is saying "yes, this input pattern looks like something I've seen frequently"), as in sensitization. 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.

DEMO with pictures of letters: How familiar is X, compared to what has been seen before?

■ Learning: input vector sums

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

```
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, 1, 1, 1, 1, 1, 1, 1, 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, 0, 0}};
```

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

```
Tv = N[Normalize[Flatten[Tmatrix]]];
Iv = N[Normalize[Flatten[Imatrix]]];
Pv = N[Normalize[Flatten[Pmatrix]]];
Xv = N[Normalize[Flatten[Xmatrix]]];
```

Let **sv** be the weight vector (**W**).

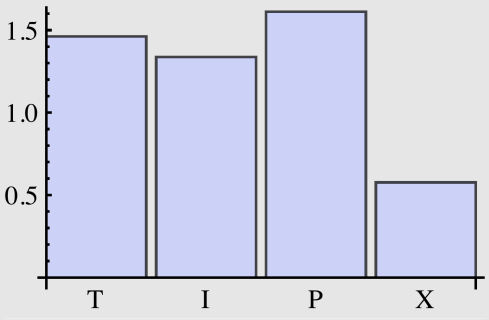
```
sv = Tv+Iv+Pv;
```

```
ArrayPlot[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.

```
matchedfilterout = {sv.Tv, sv.Iv, sv.Pv, sv.Xv};
BarChart[matchedfilterout, ChartLabels → {"T", "I", "P", "X"},
 ImageSize → Small]
```



■ 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.

```
(sv.Tv)^2 / Mean[{(sv.Tv)^2, (sv.Iv)^2, (sv.Pv)^2}]
```

0.983338

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

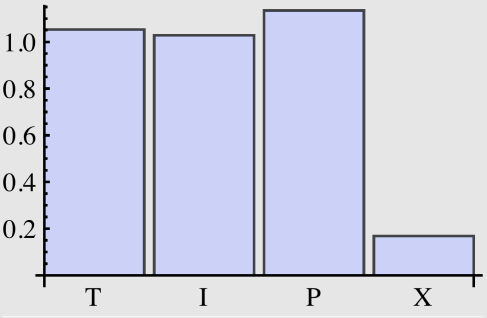
In the above representation of the letters, all the input vectors lived in the positive "quadrant", so their dot products are all positive. What if we center the vectors about zero?

```
Tv = Tv - Mean[Tv];
Iv = Iv - Mean[Iv];
Pv = Pv - Mean[Pv];
Xv = Xv - Mean[Xv];
```

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

```
ArrayPlot[Partition[sv, 10]]
```

```
matchedfilterout = {sv.Tv, sv.Iv, sv.Pv, sv.Xv};
BarChart[matchedfilterout, ChartLabels → {"T", "I", "P", "X"},
 ImageSize → Small]
```

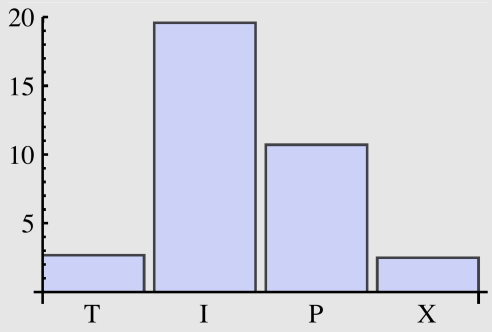



```
(sv.Xv)^2 / (Mean /@ {{(sv.Tv)^2, (sv.Iv)^2, (sv.Pv)^2}})
```

{0.0245508}

■ **What happens if the summed 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, ChartLabels → {"T", "I", "P", "X"},
 ImageSize → Small]
```



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: 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 predetermined 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 simple 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 histogram information in a single model neuron, in the sense that its response is a function of the number of times a particular synapse has been activated. The neuron behaves like a "look up table" in which we input a pattern that it has seen before, and the output reflects how often that pattern has occurred.

But we might want to have a mechanism that told us how often Ts, Is, Ps occur, in a way that doesn't muddle up their representational elements. The effectiveness of the summed vector memory depends on the dependencies of the patterns it has seen, and they may not be orthogonal.

Later we will look at a much more general framework for self-organization and the problem of measuring histograms which we can treat as estimates of the underlying probablity distributions or densities. In machine learning and statistics this kind of learning is called "density estimation".

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.

© 1998, 2001, 2003, 2005, 2007, 2009, 2011, 2012 Daniel Kersten, Computational Vision Lab, Department of Psychology, University of Minnesota.