Introduction to Neural Networks
U. Minn. Psy 5038

Daniel Kersten

Lecture 4

Introduction

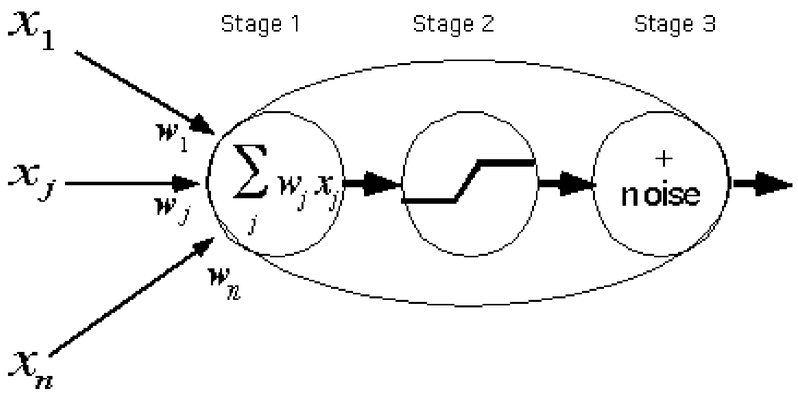
Last time

- **Slow-potential qualitative neuron model.**
- **Various types of neuron models: Levels of abstraction**
- **McCulloch-Pitts threshold logic: Discrete time, discrete state, no spatial structure**

Today

- **We develop a "structure-less, continuous signal, and discrete time" generic neuron model and and from there build a network.**
  This "connectionist" model is one of several abstractions that we saw in the previous lecture.
- **We review basic linear algebra. Motivate linear algebra concepts from neural networks.**

The generic neuron model



The generic connectionist model abstracts the basic properties of the integrate and fire neuron, and makes provision for saturation as well.

Stage 1:

Linear weighted sum of inputs

fixed bias term ()

The weights correspond to the synaptic efficiency of the inputs to the neuron which model the net effect on the input current. The bias term models a threshold.

Stage 2:

non-linearity

Popular forms are: logistic function, arctan(), limit function

A point non-linearity models both the effects of small signal compression (e.g. threshold) and large signal saturation on the output frequency of firing.

(Stage 3:)

noise

The number of spikes in a neuron's discharge is not strictly determined by the input, but varies statistically. This can be modeled assuming some form of additive (or other) stochastic component to the neural discharge frequency.

Now we will develop some *Mathematica* tools to model Stage 1 and 2 of the generic connectionist model of the neuron.

- **Defining functions.** Let's define a function to model the non-linearity (in this case, the "logistic function" mentioned earlier):

```
squash[x_] := N[1/(1 + Exp[-x])];
```

Recall, that the underscore, **x_ is important** because it tells Mathematica that x represents a slot, not an expression. Note that we've used a colon followed by equals (**:=**) instead of just an equals sign (=). When you use an equals sign, the value is calculated immediately. When there is a colon in front of the equals, the value is calculated only when called on later. So here we use **:=** because we need to define the function for later use. Also note that our squashing function was defined with **N[]**. *Mathematica* trys to keep everything exact as long as possible and thus will try to do symbol manipulation if we don't explicitly tell it that we want numbers.

Graphics. Adjust the input scale of squash[ ] to plot a very steep squashing function for -5<x<5. I.e. it should look like the step function we used when modeling the McCulloch-Pitts neuron.

- **Lists.** We already introduced lists when we studied the McCulloch-Pitts model. In this course, we are going to do a lot of work with lists, in particular with vectors (a vector is a list of scalar elements) and matrices (a matrix is a list of vector elements). Here is a four-dimensional vector which we'll call **x**. **x** could represent the input signals to a neuron.

```
x = {2,3,0,1};
```

As you may have discovered earlier, by ending a line with a semi-colon, you suppress the output after hitting the return key. Now let's make another vector, this one will be a list of "weights", say, representing the efficiency with which the inputs at the synapses are transmitted to the neuron hillock (we'll allow negative weights for the time being):

```
w = {2,1,-2,3};
```

- **Model linear neuron**

Now the output of a model neuron that simply takes a weighted sum of the inputs is just the dot product of the input with the weights:

```
y = w.x
```

```
10
```

This kind of operation is sometimes referred to as a "cross-correlator". It takes a signal **x**, and cross-correlates it with a template, **w**. Later on in an exercise you will show that for signals of fixed length, the cross-correlator gives the biggest response to the signal that exactly matches the template.

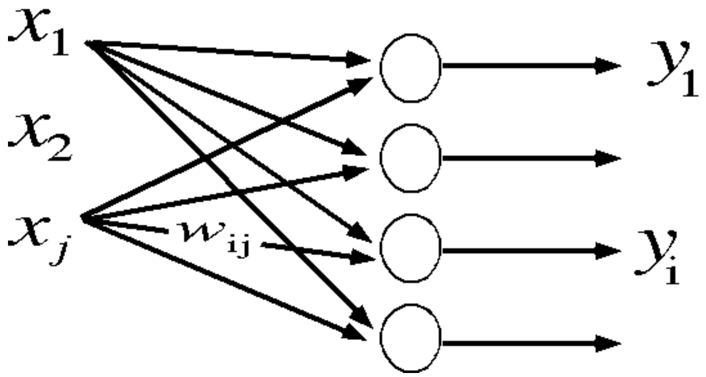
We can see what the dot product does algebraically by defining the input and weights algebraically:

```
y = {w1, w2, w3, w4}.{x1, x2, x3, x4}
```

Now let's add the non-linear squashing function to complete our model of the generic connectionist neuron:

```
y = squash[w.x];
```

Modeling a simple neural network



What if the input is applied to four neurons, each with a different set of weights? We can represent the weights by a "weight matrix", which is just a list of the four weight lists or vectors. Here is a 4x4 matrix W:

```
W = {{2,1,-2,3}, {3,1,-2,2}, {4,6,5,-3},{1,-2,2,1}};
```

Each successive element of the list W is a *row* of matrix W. Verify this by displaying W in MatrixForm

---

Now what are the outputs of the four neurons? It is just the product of the matrix **M** times the input vector **x**:

```
y = W.x
```

```
{10, 11, 23, -3}
```

In traditional form, this matrix multiplication is written as:

$$y_i = \sum w_{i,j}\, x_j$$

So to multiply an input vector by a matrix, we take the dot product of the input with each successive *row* of the matrix.

Note that a dot is used for multiplying vectors by themselves, vectors by a matrix, or to multiply two matrices together. If you want to multiply a vector or matrix by a scalar, c, you don't use a dot. For example, to normalize x by its length:

```
c = 1/Sqrt[x.x];
x2 = N[c x]
```

```
{0.534522, 0.801784, 0, 0.267261}
```

Now let's apply our squashing function to the output **y**. Note how the big positive values are set close to one, and the negative value is set close to zero.

```
squash[y]
```

```
{0.999955, 0.999983, 1., 0.0474259}
```

By default, our function **squash[]** is a **listable** function. This means that even though it was defined to operate on a scalar, when applied to a list, it automatically gets applied to each element of the list in turn.

We can do everything at once in our four-neuron network, producing the four outputs of four generic neurons to an input **x**:

```
y = squash[W.x]
```

```
{0.999955, 0.999983, 1., 0.0474259}
```

There we have it--a model for a simple four-neural network! This equation will occur many times in the rest of the course, so it is worth taking some time to understand it. Our example has four inputs, and four outputs. Try making a graphical sketch of the net to illustrate what is connected to what, label the inputs x_j.; the weights M_{ij}, and the outputs y_i.

You can access the components of vectors. For example here is the second element of y, and the element in the second row, third column of M:

```
y[[2]]
```

```
0.999983
```

```
W[[2,3]]
```

```
-2
```

Modeling noise (Stage 3): Generic neuron plus noise

We'd like to add a Stage 3 to our model of the neuron in which we take account of the noisiness of neural transmission. For this, we need the notion of a *probability distribution*. We could develop the routines we need using basic *Mathematica* functions. However, much of the work has been done for us in the *Standard Mathematica Packages*(Add-ons, in the Help menu). These packages have to be read in when you need the function definitions they contain. As a first approximation the maintained action potential discharge can be modeled as a Poisson distribution. But to use the Poisson distribution in a *Mathematica* model, you have to read in the Statistics package **DiscreteDistributions** as shown below.

Statistics and stochastic processes

Statistical routines are useful for both theoretical aspects of modeling as well as for Monte Carlo simulations. So it is worth a little effort to get acquainted with some fundamental tools and definitions. Let's start by reading in one of the statistics packages and defining a Poisson distribution with a mean of λ. And then specify λ=50 (e.g. 50 spikes per second of a neuron).

■ Discrete distributions

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

```
PDF[PoissonDistribution[λ],a]
```

$$\frac{e^{-\lambda}\,\lambda^{a}}{a\,!}$$

Let's specify a Poisson distribution with mean λ = 50:

```
pdist = PoissonDistribution[50];
```

The probability distribution function is given by:

```
PDF[pdist,a]
```

The output shows *Mathematica*'s definition of the function. You can obtain the mean, variance and standard deviation (which is the square root of the variance) of the distribution we've defined. Try it:

```
Mean[pdist]
Variance[pdist]
StandardDeviation[pdist]
```

What is your guess of the general relationship between the mean and variance for the Poisson distribution?

We are going to approximate the noisiness of neural discharge with a Normal or Gaussian distribution. The Gaussian distribution is continuous, rather than discrete. It is a fairly good approximation of a Poisson distribution for large values of the mean. To model the Gaussian, we need to read in the following package:

■ Continuous distributions, probability densities

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

```
ndist = NormalDistribution[0.5,.1];
```

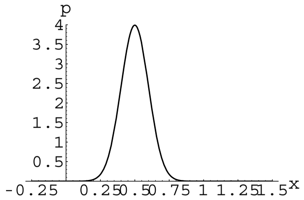
```
General::spell1 :  Possible spelling error: new symbol name "ndist" is similar to existing symbol "pdist".
```

```
Print[Mean[ndist],", ",Variance[ndist],", ",
    StandardDeviation[ndist]]
```

```
0.5, 0.01, 0.1
```

A plot of the probability distribution function for this normal distribution looks like:

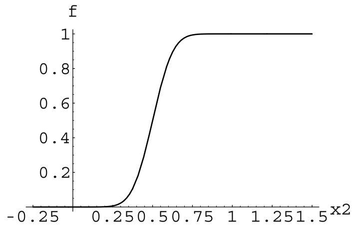
```
Plot[PDF[ndist,x],{x,-.25,1.5}, PlotRange->{0,4},AxesLabel->{"x","p"}];
```



A given ordinate value is a "density", rather than probability. But we can talk about the probability that x takes on some value in an interval, say dx. For a small interval, dx, the probability ≃ p(x)dx. What is the probability that x takes on some value between +∞ and -∞? What is area under this curve?

The cumulative distribution, f(x2) =p(x<x2), tells us the probability of x being less than a particular value of x2:

```
Plot[CDF[ndist,x2],{x2,-.25,1.5},AxesLabel->{"x2","f"}];
```



You can see from the graph that for this distribution, once x2 is greater than 0.7 or so, the probability of x being less than that is virtually certain, i.e. is essentially 1.If we set the mean =0, and the standard deviation, we'd have a graph of the "cumulative normal".

■ Statistical Sampling

Having defined the normal distribution, how can we draw samples from it? In other words, can we simulate a process in which we fill a hat with slips of paper in such a way that the proportions for each value mimic what we obtain from a theoretical distribution?

Most standard programming languages come with subroutines for doing pseudo-random number generation. Unlike the Poisson or Gaussian distribution, these numbers are usually **uniformly distributed--**that is, the probability of being a certain value (or within a tiny range) is constant over the entire sampling range of the random variable.

This is like filling the hat with slips of paper where the number of slips is the same for each value.

As we noted earlier, *Mathematica* comes with a standard function, **Random[]** that enables us to generate random numbers that are uniform. With the appropriate argument, we can also define Poisson, Normal, and other kinds of random numbers. (There are some other possible distributions in the packages too, like the **ChiSquareDistribution**).

```
Random[ndist]
```

```
0.701706
```

Putting together stages 1, 2 and 3 together

We can do everything at once, producing the output of a generic neuron, with synaptic weights **w**, neural noise with a mean of 0.0 and std. dev. 0.1 to an input x:

```
w = {2,1,-2,3};
ndist2 = NormalDistribution[0.0,.1];
y[x1_] := N[squash[w.x1] + Random[ndist2]];
y[{2,3,0,1}]
```

```
0.911057
```

If we invoke the **y[]** function again, we get a different response:

```
y[{2,3,0,1}]
```

```
0.890749
```

To sum up, the model you should have in mind is that at any given time interval (which is implicit in this continuous-response, discrete-time model), the neuron computes the sum of its weighted inputs, and the output signal, y, is a spike rate over this interval. With a sigmoidal non-linearity, there is small-signal suppression, and large-signal saturation.

Exercise

Suppose all the inputs except the first are clamped at zero. What does the response, **y** look like as a function of x for various levels of input signal? Fill in the argument for Plot[]:

```
Plot[, {x, -2, 2}, PlotRange -> {0, 2},
  AxesLabel -> {"Input signal, x", "Frequency"}];
```

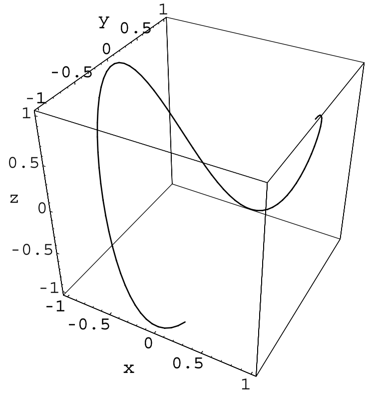
Vector operations and patterns of neural activity

■ State space and state vectors.

In neural networks, we are often concerned with a vector whose components represent the activities of neurons which are changing in time. So sometimes we will talk about state vectors.There isn't anything profound about this terminology--it just reflects that we are interested in the value of the vector when the system is in a particular state at time t. It is often very useful to think of an n-dimensional vector as a point in an n-dimensional space. This space is often referred to as state space. Suppose, we have a 3 neuron system. We can describe the state of this system as a 3-dimensional vector where each component represents the activity of the neuron. Further, suppose just for the sake of an example to visualize, the activities of the first, second, and third neurons (i.e. components) of a 3-dimensional vector are given by: y = {Cos[t], Sin[t], t}. We can use the Mathematica function, ParametricPlot3D[] to get a picture of how this state vector evolves in time through state space:

```
Clear[y];
y[t_] := {Cos[t], Sin[t], Cos[2 t]};
ParametricPlot3D[y[t], {t,0,5}, AxesLabel->{"x","y","z"}];
```

```
ParametricPlot3D::ppcom : Function y[t] cannot be
   compiled; plotting will proceed with the uncompiled function.
```



■ Dimension of a vector.

Obtain the dimensionality of a vector using Dimensions[], or Length[].

```
v = {2.1, 3, -0.45, 4.9};
```

Dimensions[], will give you the dimensions of a matrix, while **Length[]** tells you the number of elements in the list. For example,

```
M = {{2,4,2}, {1,6,4}};
```

```
Length[M]
```

```
2
```

Compare Length[M] with Dimensions[M].

■ Transpose of a vector.

The transpose of a column vector is just the same vector arranged in a row. However, because of the way Mathematica uses lists to represent vectors you don't have to distinguish between row and column vectors. In standard math notation, transpose of a vector **x**, is often written $\mathbf{x}^{\mathbf{T}}$. You can see a vector in column form by typing **v//MatrixForm**, or**:**

```
MatrixForm[v]
```

$$\begin{pmatrix} 2.1 \\ 3 \\ -0.45 \\ 4.9 \end{pmatrix}$$

■ **Vector addition** is accomplished by simply adding the components of each vector to make a new vector.

Note that the vectors all have the same dimension.

```
a = {3,1,2};
b = {2,4,8};
c = a + b
```

```
{5, 5, 10}
```

Vectors can be multiplied by a constant. We saw an example of this earlier.

```
2 a
```

```
{6, 2, 4}
```

■ Euclidean length of a vector

It is unfortunate terminology, but **Length[]** does NOT give you the metrical or Euclidean length of the vector. In order to get the length of a vector, you calculate the Euclidean distance from the origin to the end-point of the vector. We get this by squaring each component, adding up the squares, and taking the square root. First, we will do this using the **Apply[]** function, where the **Plus** operation is applied to all the elements of the list. Note that the operation of exponentiation is "listable", that is it is applied to each element of the vector:

```
a^2
```

```
{9, 1, 4}
```

What is **a a** ?

```
N[Sqrt[Apply[Plus, a^2]]]
```

```
3.74166
```

Alternatively, you can read in the package using: **<<LinearAlgebra`MatrixManipulation`** and then use the function **VectorNorm[N[a],2]**. The second argument says that you want the vector 2-norm (ie. Euclidean length). **VectorNorm[N[a],1]** would return the "city-block" norm.

If you wish, you can define your own function to apply to the list. What we have just calculated is the square root of the dot product or inner product of **a** with itself. The length of a vector **a** is often written as **|a|** in standard math notation. In the next section, we use the inner or dot product to calculate the Euclidean length of a vector.

- **Dot or Inner product**. To calculate the inner product of two vectors, you multiply the corresponding components and add them up:

```
u = {u1,u2,u3,u4};
v = {v1,v2,v3,v4};
u.v
```

```
u1 v1 + u2 v2 + u3 v3 + u4 v4
```

The **inner product** is also called the **dot product**. Later we will see what is meant by **outer product**. The inner product between two vectors **a** and **b** is written either as:

$$\mathbf{a.b} \text{ or } [\mathbf{a,b}], \text{ or } \mathbf{a^T b}$$

Mathematica uses the dot notation.

One use of the inner product is to calculate the length of a vector. **a.a** is just the sum of the squares of the elements of **a**, so gives us another way of calculating the length of a vector.

```
N[Sqrt[a.a]]
```

```
3.74166
```

Let's define a function that will return the length of a vector, x:

```
Vectorlength[x_] := N[Sqrt[x.x]]
```

Projection

Projection is an important concept in linear neural networks.

When a pattern of activity, **x**, is input to a linear neural network, the weight matrix **W** transforms the input pattern to a new output pattern **y** of activities. This linear transformation works by "projecting" the input onto a new set of dimensions by taking the dot product of the input with each *row* of the weight matrix. The columns of **W** can be thought of as describing a particular set of dimensions of the space within which output vectors of the network can live. Each output element's activity level says how much of the input activity got projected onto the vector specified by a *column* of **W**:

Suppose we have 3 inputs and 2 outputs to our network. Inputs to the network live in a 3-dimensional space. Outputs live in 2 dimensions.

$$\textbf{Prove :}\ \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} w_{11} \\ w_{21} \end{pmatrix} x_1 + \begin{pmatrix} w_{12} \\ w_{22} \end{pmatrix} x_2 + \begin{pmatrix} w_{13} \\ w_{23} \end{pmatrix} x_3$$

The dot product, **a.b**, is equal to:

$$|\mathbf{a}|\,|\mathbf{b}| \cos(\text{angle between } \mathbf{a} \text{ and } \mathbf{b})$$

In problem set 1, you calculate the output of a linear neuron model as the dot product between an input vector and a weight vector. Both the weight and input lists can be thought of as vectors in an n-dimensional space. Suppose the weight vector has unit length. Recall that you can normalize any vector to unit length by dividing by its length:

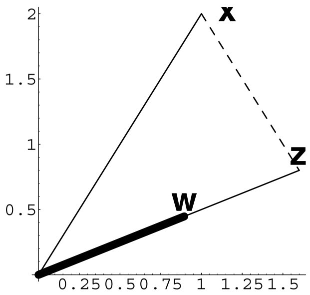
```
v = v/Sqrt[v.v] ;
```

Geometrically, we can think of the output of a neuron as the projection of the activity of the neuron input activity vector onto the weight vector direction. Suppose the input vector is already perpendicular to the weight vector, then the output of the neuron is zero, because the cosine of 90 degrees is zero. As you found or will find with the cross-correlator of Problem Set 1, the further the input pattern is away from the weight vector, as measured by the cosine between them, the poorer the "match" between input and weight vectors, and the lower the response.

Here are three lines of code that calculate the two-dimensional vector **z** in the direction of **w**, with a length determined by "how much of **x** projects in the **w** direction":

```
x = {1,2};
w = N[{2/Sqrt[5],1/Sqrt[5]}];
z = (x.w) w ;
```

```
Show[   Graphics[{Line[{{0,0}, x}],
            Line[{{0,0}, z}],
            {Dashing[{0.03,0.03}], Line[{x, z}] },
            Text[FontForm["w",{"Helvetica-Bold",18}], w, {0,-1}],
            Text[FontForm["x",{"Helvetica-Bold",18}], x, {-2,0}],
            Text[FontForm["z",{"Helvetica-Bold",18}], z, {0,-1}],
            {AbsoluteThickness[3], Line[{{0,0}, w}] }
        }],
        Axes->True, AspectRatio->1
];
```



■ Angle between two vectors and orthogonality: Similarity measure between patterns

Often we will want some measure of the similarity between two patterns of neural firings. As we have just seen, one measure of comparison is the degree to which the two state vectors point in the same direction. The cosine of the angle between two vectors is one possible measure:

```
cosine[x_,y_] := x.y/(Vectorlength[x] Vectorlength[y])
cosine[a,b]
```

```
0.758175
```

Note that if two vectors point in the same direction, the cosine of the angle between them is 1:

```
a = {2,1,3,6};
b = {6, 3, 9, 18};
cosine[a,b]
```

```
1.
```

Try verifying that **w** and **z** from the previous section point in the same direction.

If two vectors point in the opposite directions, the cosine of the angle between them is -1:

```
a = {-2,-1,-3,-6};
b = {6, 3, 9, 18};
cosine[a,b]
```

■ Euclidean distance between two vectors

Two vectors may point in the same direction, but could be quite different because they have different lengths. Another measure of similarity is the Euclidean length of the difference between two vectors, or the "distance between the tips of their vectors":

```
Vectorlength[a - b]
```

```
28.2843
```

By thinking about the geometry, what is Vectorlength[{3,0}-{0,4}]?

■ **Orthogonality.** The case where vectors are at right angles to each other is an important special case that is worth spending a little time on. Consider an 8-dimensional space. One very familiar set of orthogonal vectors is the following:

```
u1 = {1,0,0,0,0,0,0,0};
u2 = {0,1,0,0,0,0,0,0};
u3 = {0,0,1,0,0,0,0,0};
u4 = {0,0,0,1,0,0,0,0};
u5 = {0,0,0,0,1,0,0,0};
u6 = {0,0,0,0,0,1,0,0};
u7 = {0,0,0,0,0,0,1,0};
u8 = {0,0,0,0,0,0,0,1};
```

Each vector has unit length, and it is easy to see just by inspection that the inner product between any two is zero. On the other hand, here is another set of 8 vectors in 8-space for which it is not immediately obvious that they are all orthogonal. These vectors are called Walsh functions:

```
v1 = {1, 1, 1, 1, 1, 1, 1, 1};
v2 = {1,-1,-1, 1, 1,-1,-1, 1};
v3 = {1, 1,-1,-1,-1,-1, 1, 1};
v4 = {1,-1, 1,-1,-1, 1,-1, 1};
v5 = {1, 1, 1, 1,-1,-1,-1,-1};
v6 = {1,-1,-1, 1,-1, 1, 1,-1};
v7 = {1, 1,-1,-1, 1, 1,-1,-1};
v8 = {1,-1, 1,-1, 1,-1, 1,-1};
```

You can calculate the inner products between any two, and you will find out that they are all zero. Note that with the first set of vectors, {$\mathbf{u}_i$}, you can tell which vector it is just by looking for where the 1 is. For the second set, {$\mathbf{v}_i$}, you can't tell by looking at just one component. For example, the first component of all of the Walsh functions has a 1. You have to look at the pattern to tell which Walsh function you are looking at.

Suppose for the moment that we want to assign meaning to each of the patterns--each pattern is a code for some thing, like "grandma Tompkins", "grandma Wilke", and so forth. If we use the **u**'s, then we could look for the one neuron that lights up to find out which grandma it is representing--then neuron activity represented, for example, by the third element of the pattern could mean "grandma Wilke". This strategy wouldn't work if we encoded a collection of grandmothers using the **v**'s. The **v**'s give us a simple example of what is sometimes referred to as a **distributed code**. The **w**'s are examples of a **grandmother cell code**. The reason for this obscure terminology can be traced to earlier debates on whether there may be single cells in the brain whose firing uniquely determines the recognition of one's grandmother.

- **Orthonormality.** The Walsh set is orthogonal, but they are not of unit length. We have already seen some of the advantages of working with unit length vectors. The general issue of normalization comes up all the time in neural networks both in terms of limiting overall neural activity, and limiting synaptic weights. So it is sometimes convenient to normalize an orthogonal set, producing what is known as an *orthonormal* set of vectors:

```
w1 = v1/Vectorlength[v1];
w2 = v2/Vectorlength[v2];
w3 = v3/Vectorlength[v3];
w4 = v4/Vectorlength[v4];
w5 = v5/Vectorlength[v5];
w6 = v6/Vectorlength[v6];
w7 = v7/Vectorlength[v7];
w8 = v8/Vectorlength[v8];
```

Vector representations, linear algebra

The issue of how information is to be represented is fundamental in the information sciences generally, as well as for neural network theory. A pattern of activity over a set of neurons is presumed to mean something, and there are different ways of coding the same meaning. But different codes have different properties. A code may not be sufficient to uniquely code all the possible things we need to represent. A code could be redundant and have more than one way of representing the same thing. This section continues with our review of the basics of vector and linear algebra by going a little more deeply into the subject. The pay-off will be some mathematics that provides intuition about issues of neural representation. You can think of this as a first lesson in the "psychology of linear algebra".

■ Basis sets

It is pretty clear that given any vector whatsoever in 8-space, you can specify how much of it gets projected in each of the eight directions specified by the unit vectors v1, v2, ...v8. But you can also build back up an arbitrary vector by adding up all the contributions from each of the component vectors. This is a consequence of vector addition and can be easily seen to be true in 2 dimensions. We can verify it ourselves. Pick an arbitrary vector **g**, project it onto each of the basis vectors, and then add them back up again:

```
g = {2,6,1,7,11,4,13, 29};
```

```
(g.u1) u1  + (g.u2) u2  +(g.u3) u3  +(g.u4) u4  +
(g.u5) u5  +(g.u6) u6  +(g.u7) u7  +(g.u8) u8
```

```
{2, 6, 1, 7, 11, 4, 13, 29}
```

Exercise

What happens if you project **g** onto the normalized Walsh basis set defined by **{w1,w2,...}** above, and then add up all 8 components?

```
(g.w1) w1  + (g.w2) w2  +(g.w3) w3  +(g.w4) w4  +
(g.w5) w5  +(g.w6) w6  +(g.w7) w7  +(g.w8) w8
```

The projections, $\mathbf{g.u_i}$ are sometimes called the **spectrum** of **g**. This terminology comes from the Fourier basis set used in Fourier analysis. A discrete version of a Fourier basis set is similar to the Walsh set, except that the elements fit a sine wave pattern, and so are not binary-valued.

The orthonormal set of vectors we've defined above is said to be **complete**, because any vector in 8-space can be expressed as a linear weighted sum of these **basis vectors**. The weights are just the projections. If we had only 7 vectors in our set, then we would not be able to express any 8-dimensional vector in terms of this basis set. The seven vector set would be said to be **incomplete**. A basis set which is orthonormal and complete is very nice from a mathematical point of view. Another bit of terminology is that these seven vectors would not **span** the 8-dimensional space. But they would span some sub-space, that is of smaller dimension, of the 8-space.

There has been much interest in describing the effective weighting properties of visual neurons in primary visual cortex of higher level mammals (cats, monkeys) in terms of basis vectors. One issue is if the input (e.g. an image) is projected (via a collection of receptive fields) onto a set of neurons, is information lost? If the set of weights representing the receptive fields of the collection of neurons is complete, then no information is lost.

■ Linear dependence

What if we had 9 vectors in our basis set used to represent vectors in 8-space? For the u's, it is easy to see that in a sense we have too many, because we could express the 9th in terms of a sum of the others. This set of nine vectors would be said to be linearly dependent. A set of vectors is linearly dependent if one or more of them can be expressed as a linear combination of some of the others. Sometimes there is an advantage to having an "over-complete" basis set (e.g. more than 8 vectors for 8-space; cf. Simoncelli et al., 1992).

Theorem: A set of mutually orthogonal vectors is linearly independent.

However, note it is quite possible to have a linearly independent set of vectors which are not orthogonal to each other. Imagine 3-space and 3 vectors which do not jointly lie on a plane. This set is linearly independent.

If we have a linearly independent set, say of 8 vectors for our 8-space, then no member can be dropped without a loss in the dimensionality of the space spanned.

It is useful to think about the meaning of linear independence in terms of geometry. A set of three linearly independent vectors can completely span 3-space. So any vector in 3-space can be represented as a weighted sum of these 3. If one of the members in our set of three can be expressed in terms of the other two, the set is not linearly independent and the set only spans a 2-dimensional subspace. That is, the set can only represent vectors which lay on a plane in 3-space. This can be easily seen to be true for the set of u's, but is also true for the set of v's.

Thought exercise

Suppose there are three inputs feeding into three neurons in the simple linear network such as defined at the beginning of this lecture. If the weight vectors of the three neurons are not linearly independent, do we lose information?

Linearity, real neural networks, and what's up next time?

From a computational standpoint, the squashing function has both advantages and disadvantages. It is what makes our neural network model *non-linear*, and as we will see later, this non-linearity enables networks to compute functions that can't be computed with a linear network. On the other hand, non-linearities make the analysis complicated because we leave the well-understood domain of linear algebra. In fact, there are cases for which most of the neural activities are in the mid-range of the squashing function, and here one can approximate the network as a purely linear one--just matrix operations on vector inputs, and the analysis becomes relatively simple.

Compared to the complexity of real neurons and networks, assuming linearity might seem to be just too simple. But we will see in the next lecture, that a linear model can be quite good model for some biological subsystems. We will apply the techniques of linear vector algebra to model a network discovered in the visual system of the horseshoe crab. Later we'll see how some aspects of associative memory can be modeled using linear systems.

References

atmatica (http://www.mathsource.com/Content/Applications/Education/Other/0209-551?notables). A mathematica-based linear algebra course.

Simoncelli, E. P., Freeman, W. T., Adelson, E. H., & Heeger, D. J. (1992). Shiftable Multi-scale Transforms. IEEE Trans. Information Theory, 38(2), 587--607.

Strang, G. (1988). Linear Algebra and Its Applications (3rd ed.). Saunders College Publishing Harcourt Brace Jovanovich College Publishers.