Introduction to Neural Networks

Daniel Kersten

Lecture 4: The generic neuron and linear models

Introduction

Last time

Slow-potential qualitative neuron model.
Various types of neuron models: Levels of abstraction
McCulloch-Pitts threshold logic: Discrete time, discrete signal, no spatial structure assumed for neuron
Integrate-and-fire model: continuous time, continous signal, no spatial structure assumed for neuron

Today

We continue with the no spatial structure and continuous signal simplification, but discretize time, and from there build a network.
Review basic linear algebra. Motivate linear algebra concepts from neural networks.

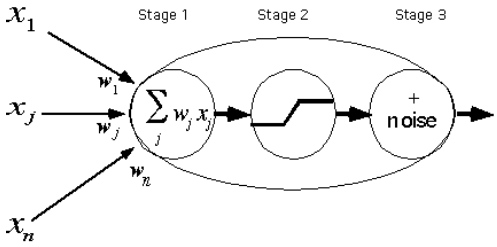
Motivation: What can be done with the linear modeling tools we cover today and in the next few lectures?

Filter images, filter sounds, model early sensory cortical cells, detect edges, textures, front-end for motion estimation, describe human sensory thresholds...

We'll develop tools to understand a range of theories from associative memory and recall to sparse coding representations of image or sound features.

Modeling one neuron

Three stages: dot product, point-wise non-linearity, additive noise



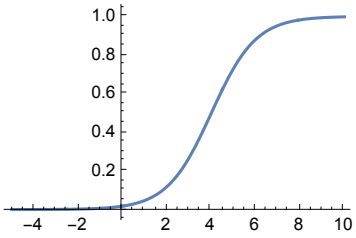
The generic neuron model abstracts the basic properties of the integrate and fire neuron, and makes provision for saturation as well. We'll assume we are concerned about the how the inputs determine the output at a particular time. We ignore differences in the timing of the inputs and assume no delays between the input and output. If we are thinking about firing rates, there is an implicit assumption of a time period over which the rate is relevant. E.g. if we assume 100 msec, then signal rates could vary between 0 and 50 or so spikes per second.

Stage 1: *Linear weighted sum of inputs*

The weights correspond to the synaptic efficiency of the inputs to the neuron which model the net effect on the input current. Sometimes models include a fixed additional bias term at the input, which will affect how the neuron behaves with the next non-linear term.

Stage 2: *Point-wuise non-linearity*

Popular forms are "sigmoidal" or approximately so: logistic function, arctan(), semi-linear function These are sometimes called "point" non-linearities. The non-linearity is a scalar function of a scalar--it isn't a non-linear combination of interacting inputs. The standard sigmoidal shaped non-linearity captures both the effects of small signal compression (e.g. recall threshold resulting from the leaky integrate and fire model) and large signal saturation on the output frequency of firing (recall the effect of the absolute refractory period).

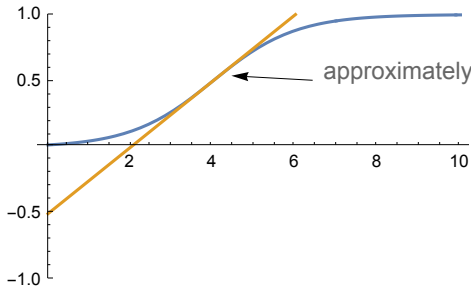


Stage 3: *Noise*

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. Note that putting the noise at the end is just one possibility. Noise can occur at the input and between Stage 1 and 2.

Much of our intuition and theory about neural networks is based on studies of *linear neural networks*. The rationale is that there is an intermediate input regime over which the output is an approximately linear function of the input.

The advantage of the linear approximation is simplification, and being able to exploit our knowledge of linear algebra and linear systems.



Use Plot to compare two sigmoidal non-linearities:

```
squash[x_] := N[1 / (1 + Exp[-x + 4])];
squash2[x_] := (2. / Pi) * N[ArcTan[x]];
```

Using *Mathematica* to make a generic neuron

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). We already introduced lists when we studied the McCulloch-Pitts model. Here is a four-dimensional vector which we'll call x. x could represent the input signals (e.g. firing rates) to a neuron.

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

Sidenote : While it doesn' t make sense for firing rates frequency, sometimes modelers let the input and outputs take on negative values, e.g. between - 1 and 1. The rationale is that the symmetry about zero can simplify the math.

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. Note that negative weights make sense -- recall that synapses can be excitatory or inhibitory.

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

Dot product (Stage 1)

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

```
y = w.x
```

```
10
```

You can also write:

```
y = Dot[w, x]
```

This kind of operation is a simple version of what is sometimes referred to as a "cross-correlator" or matched filter. It takes a signal **x**, and cross-correlates or tries to matches it with a template, **w**. In your homeowrk, you will show that for signals of fixed vector length (or "norm"), the dot product 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:

```
Clear[w1, w2, w3, w4, x1, x2, x3, x4];
y = {w1, w2, w3, w4}.{x1, x2, x3, x4}
```

```
w1 x1 + w2 x2 + w3 x3 + w4 x4
```

The sigmoidal non-linearity (Stage 2)

Now define a function to model the non-linearity of Stage 2 (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. Again, 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[]**. Why did we do that?

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.

Now let's include the non-linear squashing function to complete our model of the first two stages of the generic neuron:

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

Modeling noise (Stage 3)

We'd like to add a Stage 3 to our model of the neuron to take into account the noisiness of neural transmission. For this, we need the notions of *random variable* and *probability distribution*. We’ll cover just enough here to have some basic tools and concepts. We’ll go over probability theory in greater depth in later lectures.

The idea is that we want to simulate drawing of a random number to represent the variations in firing rate. Think of a hat filled with slips of paper whose proportions represent the probabilities of various deviations of firing rate values--e.g. “an extra few added spikes, or a few subtracted spikes each second”. The hat stands for a probability distribution. And the collection of all the values on the slips of paper represent the random variable associated with the distribution or hat. Once a piece of paper is drawn, the random variable “takes on” the value written on the paper. The value drawn determines the noise for that time period.

We could program the routines we need to generate particular types of noise using basic *Mathematica* functions. However, many of the probability functions we need are already built into *Mathematica*. As a first approximation the maintained action potential discharge can be modeled as a Poisson distribution, $p(a;\lambda)$ This is a discrete distribution that represents the probability of ***a*** events (e.g. ***a*** action potentials)

given an average rate of λ. The assumption behind the definition (see below) is that there is an average number of spikes that occur within a specific time interval, and that the discharge of a spike is independent of the time since the last one. Real neuron variability can be more complicated, but this provides a reasonable initial model. See Rieke et al. (1997).

Probability distributions and sampling

Statistical routines are useful for both theoretical aspects of modeling as well as for Monte Carlo simulations where one simulates drawing random samples to use as inputs to an algorithm. So it is worth a little effort to get acquainted with some fundamental tools and definitions. Let's define a Poisson distribution with a mean of λ. And then specify an average of 50 spikes per second, λ=50.

Discrete distributions

```
Clear[a];
PDF[PoissonDistribution[λ],a]
```

$$\begin{cases} \frac{e^{-\lambda}\,\lambda^{a}}{a!} & a \geq 0 \\ 0 & \text{True} \end{cases}$$

Let's specify a Poisson distribution with mean λ = 25:

```
pdist = PoissonDistribution[25];
```

The probability distribution function is given by:

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

$$\begin{cases} \frac{25^{a}}{e^{25}\,a!} & a \geq 0 \\ 0 & \text{True} \end{cases}$$

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]}
```

```
{25, 25, 5}
```

```
gpdist = DiscretePlot[PDF[pdist, x], {x, -10, 50}, AxesLabel → {"x", "p"}];
```

Continuous distributions, probability densities

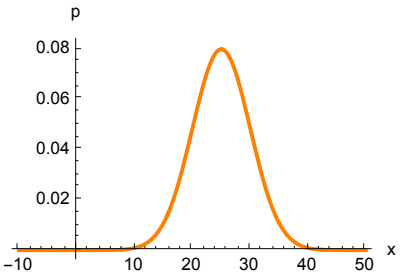
```
ndist = NormalDistribution[25.,Sqrt[25.]];
```

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

```
25., 25., 5.
```

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

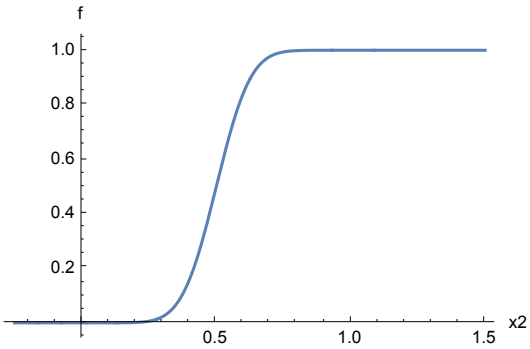
```
gndist = Plot[PDF[ndist, x], {x, -10, 50},
  AxesLabel → {"x", "p"}, PlotStyle → {Orange, Thick}]
```



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 a small interval *dx.* For a small interval, *dx*, the probability ≃ p(x)*dx, i.e.* the area under the curve. 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 x2:

```
Plot[CDF[ndist, x2], {x2, -0.25, 1.5}, AxesLabel → {"x2", "f"}]
```

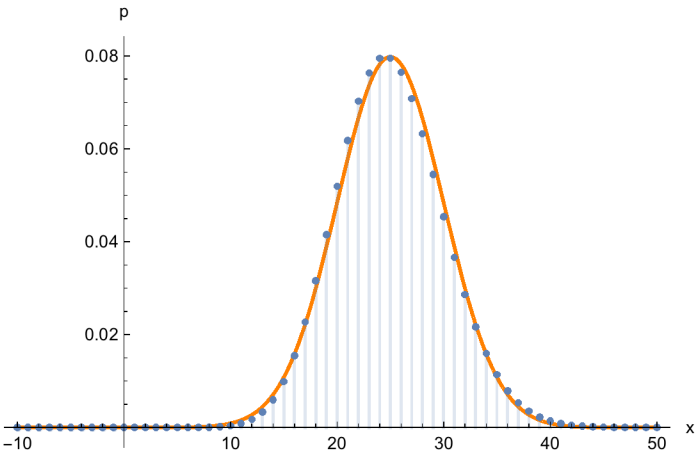


You can see from the graph that for this distribution, once x2 is greater than 0.7 or so, the probability of x being less than that is virtually certain, i.e. is essentially 1. If we set the mean = 0, and the standard deviation, we'd have a graph of the "cumulative **normal**".

Approximating a Poisson distribution by a Gaussian. As the mean of a Poisson distribution gets bigger, the form of the distribution gets closer to that of a Gaussian distribution. So sometimes it is convenient to approximate the noisiness of neural discharge with a Normal (Gaussian) distribution. The Gaussian distribution is continuous so is represented by a *density*. It is a fairly good approximation of a Poisson distribution for large values of the mean, and the math is usually easier. Further, if we represent the neuron output as a continuous value (e.g. frequency), then it is reasonable to use a continuous-valued model of noise. We do have to be careful tho', because the Gaussian is defined over negative numbers too.

As you can see in the graph below, at λ = 25, the Gaussian (in orange) is a fairly good fit

```
Show[{gndist, gpdist}]
```



Statistical Sampling

Having defined the normal distribution, how can we draw samples from it? You are no doubt familiar with the idea of picking out a colored ball from a jar without looking. If the jar has 100 red balls, and 50 blue balls, the probability of picking a red ball is 2/3rds. That is assuming we are putting the balls back after each draw. The process of randomly picking a ball is called *sampling*.

Or let's go back to a slightly more complicated version of our hat with slips of paper. Fill the hat with slips that each have a number written on it, say numbers from 1 to 9. Suppose there is 1 slip with the "1" written on it, 3 slips with "2" written on it, 3 slips with "3" written on it, 4 slips with "4" written on it, 5 slips with "5" written on it, 4 slips with "6" written on it, 3 slips with "7" written on it, 2 slips with "8" written on it, and 1 slip with "9 " written on it. If you randomly pick out a slip of paper, replace it, and do it again and again, you'd expect that the proportion of samples drawn would come to look like the proportions of slips of paper in the hat.

Let's see how to 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 this, called pseudo-random number generation. Unlike the Poisson or Gaussian distribution, the basic function is usually for a **uniformly distributed** random variable**--**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. *Mathematica* comes with standard functions, **RandomReal[] and RandomInteger[],** that enable us to generate random numbers that are uniformly distributed. With the appropriate argument, we can also define Poisson, Normal, and other kinds of random numbers using these functions**:**

```
RandomReal[ndist]
```

```
0.603919
```

But unless the distributions are uniform, it is better to use **RandomVariate[]** , which avoids you having to distinguish between discrete, continuous or mixed distributions:

```
0.591627
```

```
RandomVariate[ndist]
```

```
0.60025
```

Simulate drawing Poisson distributed integers with a mean value of 25 (See RandomVariate[])

Putting stages 1, 2 and 3 together

We can put the three stages together, 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:

```
Clear[x1,w,y];
w = {2,1,-2,3};
ndist2 = NormalDistribution[0.0,.1];

y[x1_] := N[squash[w.x1] + RandomVariate[ndist2]];

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

```
-0.170732 + squash[10.]
```

Note that this is an *additive* model of noise with constant variance. The noise level doesn't depend on the activity value of the deterministic part. Although often a good first approximation and starting point for a model, additive noise is often an over-simplication. Not surprisingly spiking behaves more like a Poisson where the variance grows with firing rate.

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

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

```
-0.0196075 + squash[10.]
```

Note that the above noise term can produce negative values. There are various ways of fixing this like setting samples below zero to zero. Or picking a suituable standard distribution that is only defined for positive values.

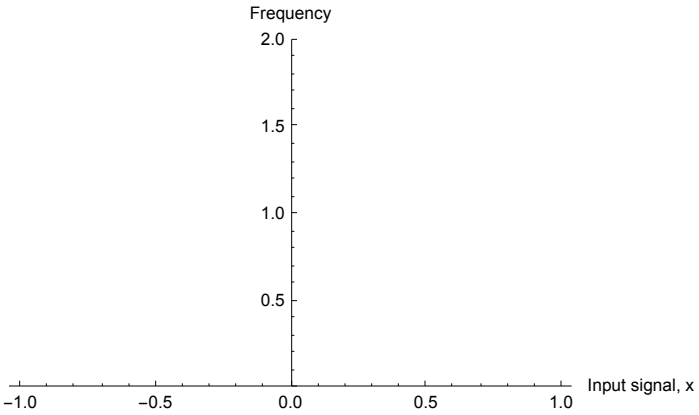
To sum up, the model you should have in mind is that at any given time interval (which is implicit in this continuous-response, discrete-time model), the neuron computes the sum of its weighted inputs, and the output signal, y, is a spike rate over this interval. Even with the same input, the spike count can vary with repeated measurements because of noise. With a sigmoidal non-linearity, there is small-signal suppression, and large-signal saturation. But there is an near-linear regime.

Exercise

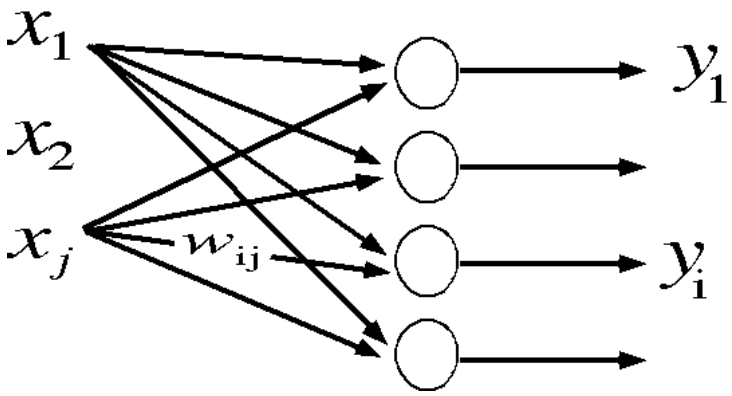
Suppose all the inputs except the first are clamped at zero. What does the response, y look like as a function of x for various levels of input signal?

Fill in the argument for Plot[]:

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



Modeling a simple neural network



Linear matrix model

What if the input

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

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}}
```

```
{{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 or TraditionalForm

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

```
y = W.x
```

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

In traditional mathematical form, this matrix multiplication is written as:

$$y_i = \sum_{j=1}^{n} w_{i,j}\, x_j$$

```
Sum[x_j w_i,j]
```

So to multiply an input vector by a matrix, we take the dot product of the input vector with each successive *row* of the matrix. Note that in *Mathematica*, 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 vector length:

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

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

Take the dot product of x2 above with itself to confirm normalization

---

Applying the non-linear squashing function

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.

```
y
squash[y]
```

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

```
{0.997527, 0.999089, 1., 0.000911051}
```

By default, our function **squash[]** is a **listable** function. This means that even though it was first 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]
```

```
{0.997527, 0.999089, 1., 0.000911051}
```

There we have it--a model for a simple neural network. It is also "feedforward" in that it maps inputs to outputs, in contrast to a network in which outputs get fed back to inputs. Variations on this equation will occur many times in the rest of the course, so it is worth taking some time to understand it.

Most of the time, we will assume the noise is negibilte. But if we want to add noise to the model, we could add a random number generator. Note that second argument of RandomVariate specifies the dimensions of the input:

```
y = squash[W.x] +RandomVariate[NormalDistribution[0.0, .1], 4]
```

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 w_{ij}, and the outputs y_i.

It is straightforward to turn the model into a function:

y[x_, W_, μ_, σ_] := squash[W.x] + RandomVariate[NormalDistribution[μ, σ], Dimensions[x][[1]]]

Mathematica sidenote: Accessing elements of vectors, matrices

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 **W**:

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

```
0.999089
```

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

```
-2
```

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

```
-2
```

```
W[[1]]
```

```
{2, 1, -2, 3}
```

You can use **Span[]** or its shorthand ;; to get rows, columns or submatrices. First two rows of W3x3:

```
W3x3[[1 ;; 2]] // MatrixForm
```

$$\begin{pmatrix} w11 & w12 & w13 \\ w21 & w22 & w23 \end{pmatrix}$$

Second and third rows of W3x3:

```
W3x3 = {{w11, w12, w13}, {w21, w22, w23}, {w31, w32, w33}};
```

```
W3x3[[ ;; , 2 ;; 3]] // MatrixForm
```

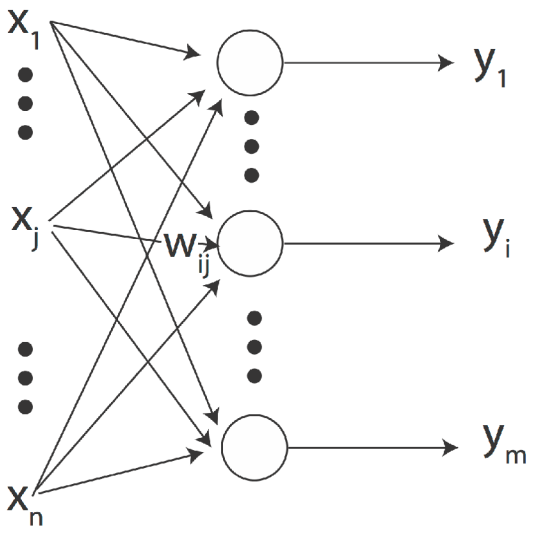
$$\begin{pmatrix} w12 & w13 \\ w22 & w23 \\ w32 & w33 \end{pmatrix}$$

Lower right 2x2 of W3x3:

```
W3x3[[2 ;; 3, 2 ;; 3]] // MatrixForm
```

$$\begin{pmatrix} w22 & w23 \\ w32 & w33 \end{pmatrix}$$

In general for a real neural network the number of inputs is highly unlikely to equal the number of outputs. In summary, the mapping from inputs to outputs is given by an mxn matrix, where the matrix output is followed by a sigmoid :



$$y_i = \sigma\left(\sum_{j=1}^{n} w_{ij}\, x_j\right) \quad \text{for } i = 1 \text{ to } m$$

Here are three examples with matrices **Wmxn**, where m = number of rows, and n = number of columns:

```
Clear[x1, x2, x3];
xin = {x1, x2, x3};
xin2 = {x1, x2};
W2x3 = {{w11, w12, w13}, {w21, w22, w23}};
W3x2 = {{w11, w12}, {w21, w22}, {w31, w32}};
W3x3 = {{w11, w12, w13}, {w21, w22, w23}, {w31, w32, w33}};
```

More inputs than outputs, more outputs than inputs, and then m = n:

```
W2x3.xin // MatrixForm
W3x2.xin2 // MatrixForm
W3x3.xin // MatrixForm
```

$$\begin{pmatrix} w11\,x1 + w12\,x2 + w13\,x3 \\ w21\,x1 + w22\,x2 + w23\,x3 \end{pmatrix}$$

$$\begin{pmatrix} w11\,x1 + w12\,x2 \\ w21\,x1 + w22\,x2 \\ w31\,x1 + w32\,x2 \end{pmatrix}$$

$$\begin{pmatrix} w11\,x1 + w12\,x2 + w13\,x3 \\ w21\,x1 + w22\,x2 + w23\,x3 \\ w31\,x1 + w32\,x2 + w33\,x3 \end{pmatrix}$$

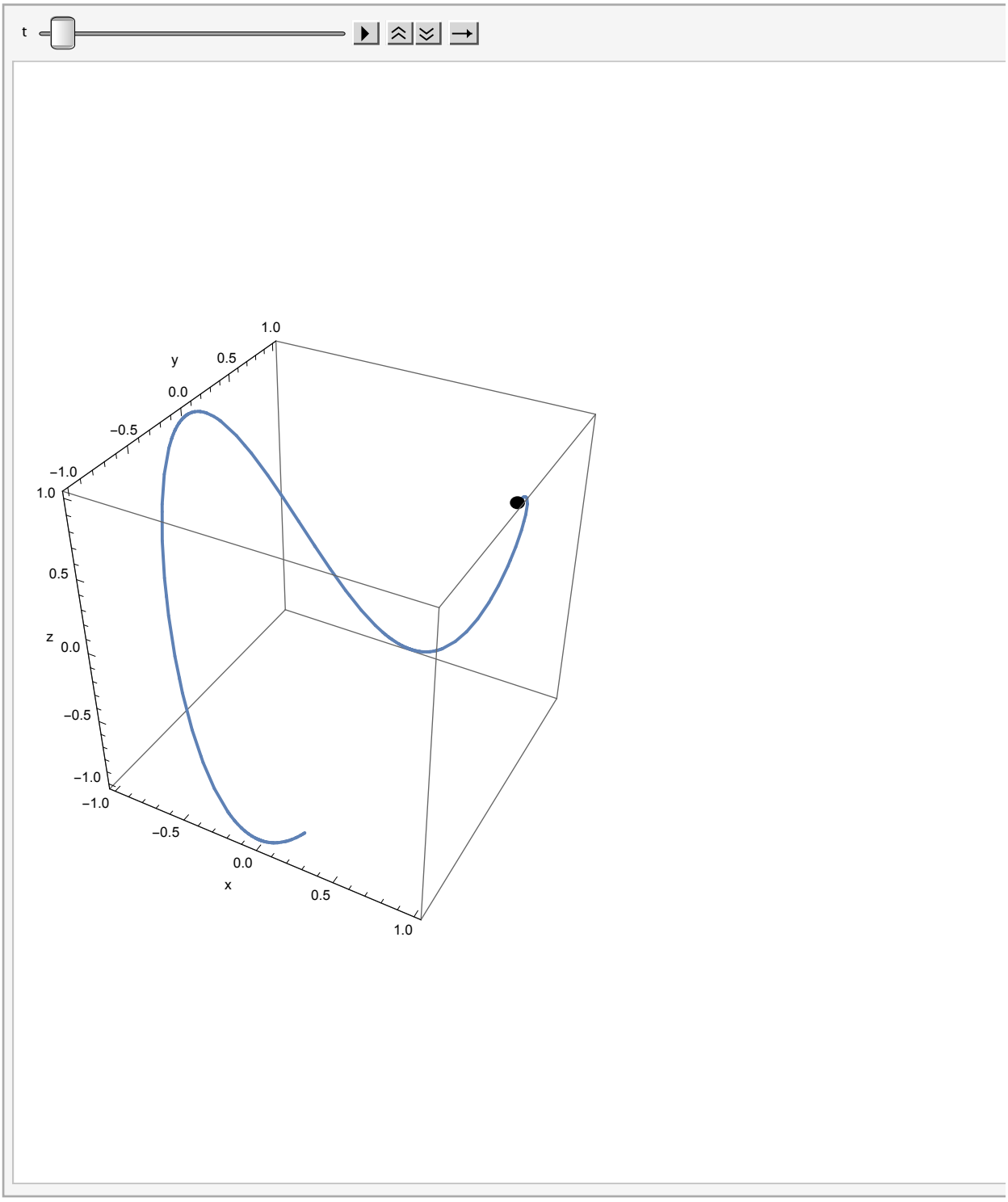
Some neuro-terminology: If one output neuron receives inputs from n neurons, these input neurons are said to comprise the output neuron's "*receptive field*". If one input neuron feeds into m output neurons, these m output neurons constitute the input neuron's "*projective field*".

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 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 as 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], Cos[2*t]}. We can use the Mathematica function, ParametricPlot3D[] to visualize how this state vector evolves in time through state space:

```
Clear[y];
y[t_] := {Cos[t], Sin[t], Cos[2 * t]};
Animate[Show[ParametricPlot3D[y[t], {t, 0, 5}, AxesLabel -> {"x", "y", "z"}],
  Graphics3D[{PointSize[.025], Point[y[t]]}]],
 {t, 0, 5}, AnimationRunning -> False]
```



Dimension of a vector.

```
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
```

Use Dimensions[ ] to print the dimensions of v

Compare Length[M] with Dimensions[M]. Compare Length[v] with Dimensions[v].

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, which is the Euclidean distance from the origin to the end of the vector. But **Norm[]** does. To get the length of a vector, you calculate the Euclidean distance from the origin to the end-point of the vector. squaring each component, adding up the squares, and taking the square root.

```
Remove[x1, x2, x3]
Norm[{x1, x2, x3}]
```

$$\sqrt{\text{Abs}[x1]^2 + \text{Abs}[x2]^2 + \text{Abs}[x3]^2}$$

To get a little more practice with *Mathematica*, you can also do this with the **Apply[ ]** function, where the **Plus** operation is applied to all the elements of the list. Note that the operation of exponentiation (raising to the power of 2) is "listable", that is it is applied to each element of the vector:

```
{x1,x2,x3}^2
Sqrt[Apply[Plus, {x1,x2,x3}^2]]
```

$$\{x1^2, x2^2, x3^2\}$$

$$\sqrt{x1^2 + x2^2 + x3^2}$$

Norm[*expr*] generalizes to **Norm[*expr, p*].** The second argument says that you want the vector 2-norm (i.e. Euclidean length). **Norm[x,1]** would return the "city-block" norm.

```
Norm[u,1]//TraditionalForm
```

$$\|u\|_1$$

Compare: {x1,x2,x3} {x1,x2,x3}, {x1,x2,x3}*{x1,x2,x3}, {x1,x2,x3}^2, {x1,x2,x3}.{x1,x2,x3}

The norm or vector 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

We just saw this. To calculate the dot product of two vectors, you multiply the corresponding components and add them up:

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

$$u1\ v1 + u2\ v2 + u3\ v3 + u4\ v4$$

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

a.b or **[a,b]**, 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 vector length of a vector.

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

```
3.74166
```

```
Sqrt[u.u]
```

$$\sqrt{u1^2 + u2^2 + u3^2 + u4^2}$$

Define your own function that will return the L2-norm (regular Euclidean length of a vector) x: Vectorlength[x_] := N[Sqrt[x.x]]

Projection

Projection is an important concept in linear algebra, and 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 vector with each *row* of the weight matrix.

In programming assignment 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, so **w.w = 1**. Recall that you can normalize any vector to unit length by dividing by its length:

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

or using the bult-in function Normalize[]:

```
vn = Normalize[v]
```

$$\left\{\frac{v1}{\sqrt{\text{Abs}[v1]^2+\text{Abs}[v2]^2+\text{Abs}[v3]^2+\text{Abs}[v4]^2}}, \frac{v2}{\sqrt{\text{Abs}[v1]^2+\text{Abs}[v2]^2+\text{Abs}[v3]^2+\text{Abs}[v4]^2}}, \frac{v3}{\sqrt{\text{Abs}[v1]^2+\text{Abs}[v2]^2+\text{Abs}[v3]^2+\text{Abs}[v4]^2}}, \frac{v4}{\sqrt{\text{Abs}[v1]^2+\text{Abs}[v2]^2+\text{Abs}[v3]^2+\text{Abs}[v4]^2}}\right\}$$

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

|**a**| |**b**| cos(angle between **a** and **b**)

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 perpendicular (orthogonal) 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 dot product exercise 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. This kind of comparison between input and the weight vector is also sometimes called “template matching”, where the weight vector represents a “template” stored in memory. As mentioned earlier, you may also see the phrase “matched filter”.

Consider the simple case of a 2-dimensional input onto one neuron. The output lives in a 1-dimensonal space, i.e. a line pointing in the direction of the weight vector **w**. 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". The magnitude of vector **z**, represents the output activity level of the neuron.

```
x = .85*{1,2};
w = N[{2/Sqrt[5],1/Sqrt[5]}];
z = (x.w) w
```

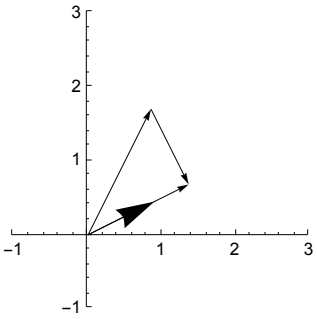
```
{1.36, 0.68}
```

Projection is a fundamental concept, and *Mathematica* provides a function for it:

```
z = Projection[x, w]
```

```
{1.36, 0.68}
```

```
Show[Graphics[{Tooltip[Arrow[{{0, 0}, x}], "x"], Tooltip[Arrow[{{0, 0}, z}], "z"],
   Tooltip[{Arrowheads[Large], Arrow[{{0, 0}, w}]}, "w"],
   Tooltip[Arrow[{x, z}], "z-x"]}],
 PlotRange → {{-1, 3}, {-1, 3}}, Axes → True, AspectRatio → 1]
```



Try moving your mouse over the arrows above. **Tooltips[]** provides rollover labels for **x**, **w**, **z**, and **z - x** on the graph.

Similarity measures between patterns: Angle between two vectors and orthogonality

Often we will want some measure of the similarity between two patterns, and more specifically two patterns of neural firing frequencies in a neural network. As we have just seen, one measure of comparison is the degree to which the two state vectors point in the same direction. Thus the cosine of the angle between two vectors is one possible measure:

```
cosine[x_,y_] := x.y/(Norm[x] Norm[y])
```

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

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

```
0
```

```
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};
VectorAngle[a, b]
cosine[a,b]
```

```
π
```

```
-1
```

And if the vectors **a2**, **b2**, are *orthogonal*, then:

```
a = {-2, -1, -3, -6};
b = {6, 3, 9, 12};
{a2, b2} = Orthogonalize[{a, b}];
cosine[a2, b2]
VectorAngle[a2, b2]
```

0

$\frac{\pi}{2}$

Similarity measures between patterns: 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":

```
N[Norm[a - b]]
```

23.4094

Or you can use:

```
EuclideanDistance[a, b]
```

$2\sqrt{137}$

Later on, we'll consider other measures of similarity, such as the normalized cross-correlation, *Mathematica*'s **CorrelationDistance[]**, among others.

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

Orthogonality

The case where a collection of vectors are at right angles to each other is an important special case that is worth spending a little time on. (As reminder, this is the same as being "perpendicular to each other", or "orthogonal to each other".) 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. This is the familiar Cartesian set.

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. This second set is a discrete representation from the set of 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};
```

The above example is just one of an infinite number of possible orthogonal sets.

You can calculate the inner products between any two Walsh vectors, 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.

Sidenote: See example section in the UnitStep[] Mathematica reference for general code for Walsh functions defined on a continuum:

```
Walsh[{n_, k_}, y_] :=
 Module[{li = Split[Extract[Nest[Sequence @@@ {Join[Riffle[#, #] & /@ #],
           Join[Riffle[#, -#] & /@ Reverse[#]]} &, {{1}}, n], {k}]]},
  -1 - 2 Total[MapIndexed[(-1)^#2 UnitStep[Mod[y, 2^n] - #1] &,
      Most[FoldList[Plus, 0, Length /@ li]]], Infinity]]
```

A brief digression into the "neural code": Grandmother cell coding

Later on, we'll spend considerable time thinking about the problem of *neural representation* and the *neural code*. What is the relationship between a pattern of activity and what it means or represents? Let's pursue this further. Suppose we have a pattern of activity, represented by a vector, at the output stage of a neural network. Let's assume that the network has done a good job of learning to distinguish a discrete categories of inputs. We might expect then that there is a small set of patterns, i.e. set of vectors, that are "far apart" that map onto the categories. One pattern can be identified with "category A", another with "category B" and so forth.

To be concrete, suppose there are 8 neurons in my brain that can fire in response to me viewing one of my relatives. Let's assign specific meanings to each of the patterns--each pattern is a code for some thing, like "grandma Tompkins", "grandma Wilke", "uncle Heine", "aunt Mabel", and so forth. If we consider the **u**'s, the Cartesian set, then to decide who I'm looking at (to "read my mind"), one could look for the one neuron that lights up to find out which relative it is representing--then the neuron activity represented, for example, by the third element of the pattern could mean "grandma Wilke". The third unit $\mathbf{u}_{3[3]}$ is 1 if and only if it is grandma Wilke.

This strategy wouldn't work if we encoded a collection of relatives using the **v**'s. Suppose relatives are represented by coding in the Walsh set, and that grandma Wilke is represented by the third pattern of activity $\mathbf{v}_3$, then although grandma Wilke implies that $\mathbf{v}_3[3]$ is -1 (or in general that $\mathbf{v}_3[i]$ is a particular value for any of the is), the reverse isn't true--the activity level of any single neuron does not uniquely specify which relative I'm looking at.

The mapping of **v**'s to labels representing my 8 relatives give us a simple example of what is sometimes referred to as a **distributed code**. The **u**'s are examples of a **grandmother cell code**. The reason for this obscure terminology can be traced to 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 the vectors in the set are not each 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/Norm[v1];
w2 = v2/Norm[v2];
w3 = v3/Norm[v3];
w4 = v4/Norm[v4];
w5 = v5/Norm[v5];
w6 = v6/Norm[v6];
w7 = v7/Norm[v7];
w8 = v8/Norm[v8];
```

Representations of patterns

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. In vision, one worries about the relationship between a pattern of firing and the various ways the image information could be represented in terms of features. What is the *meaning* of the pattern in terms of how it relates to what is in the world *and* how the pattern is to be used?

This will be our first introduction to the notion of a generative model which is a description of how patterns are made or “encoded”. This is in contrast to a description of how patterns are analyzed or “decoded”. Generative models specify how to synthesize patterns. And later we’ll see how discriminative models specify how to analyze a pattern.

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.

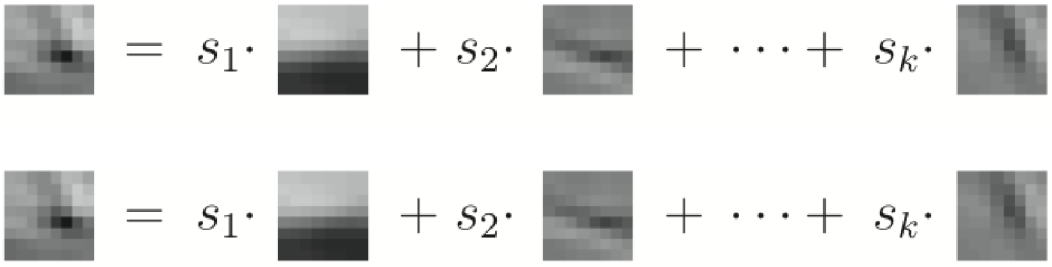
Motivation: representation of visual information

The idea in linear algebra of a “basis set” has provided insight into how populations of neurons might represent high-dimensional information, such as “an image” transmitted to cortex.

We first build some informal intuitions to distinguish between “features” *A* that *explain or generate* an input *I*, and the weights *W* that produce a pattern of neural activity *s*.

Let’s assume that an input image I can be represented by a linear combination of “feature” or basis vectors consisting of the columns of A:

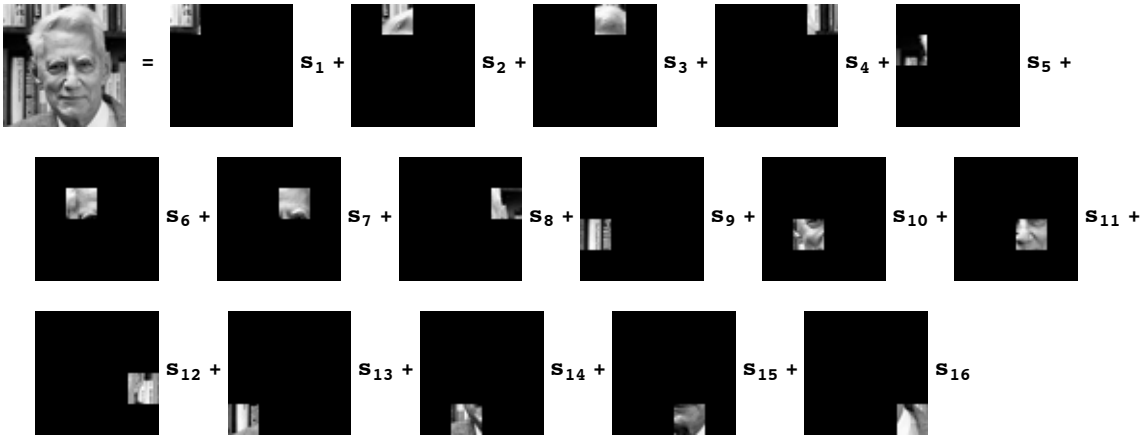
$$I(x,y) = \sum_{i=1}^{n} A_i(x,y) s_i$$



Now imagine the A's provide a vocabulary (or featrures) useful to describe any image (perhaps in a restricted set, say "human faces") in terms of a weighted sum. This is a generative model for the patterns in that set.

See: Hyvärinen, A. (2010). Statistical Models of Natural Images and Cortical Visual Representation. Topics in Cognitive Science, 2(2), 251–264. doi:10.1111/j.1756-8765.2009.01057.x

Here is a simple example where the features are non-overlapping patches:



Now let's reformulate the example in terms of simple vectors and matrices. We could "flatten" $I(x,y)$ and each $A_i(x,y)$ to turn them into vectors and then write:

I = A.s,

where the features $A_i(x,y)$ are now the "columns" of A, $\{A_j\}$, and the elements of s, $\{s_j\}$, represent the activity of the neurons representing how much each of those features contribute to (or explain) I:

I = A_1*s1 + A_2 *s2 +...

We use the word "features" because we think of the input as being made up of combinations of these particular patterns in the world.

(This is related to the machine learning "dictionaries" for a class of patterns. In this case, the goal is to find a small set of "words" that can be used to efficiently approximate any of the patterns in the class. Efficiency can be interpreted in terms of "sparseness", but more on this later.)

So the actual neural representation of an image I is only implicit in the firing rates and a definition of what each A_j means. In other words, the activities s are a simple "code" for I which can be recovered using the above formula...IF we know A.

The problem "inverse" to the above, is given an image I(x,y) and a set of receptive fields $\{W_i(x,y)\}$, determine the activity of the population of neurons.

$$s_i = \sum_{x,y} W_i(x,y) I(x,y)$$

Now we are analyzing rather than synthesizing an image. Again, switching to a vector representation of input I and a set of receptive fields W, the activity of the population of neurons can be written:

s = W.I.

where again s and I are vectors, and W is a matrix. W.I describes how a set of effective weights (e.g. synaptic weights or "receptive field" weights) turns the image input I into a pattern of neural responses--a "neural image" s.

OK, now let's make some very strong, but greatly simplifying assumptions.

From a linear algebra perspective, if I and s have the same number of elements, A is "square", and if A is invertible, then W = A^{-1}.

Further, there is a theorem that says if the columns of A are orthogonal, then W is just the transpose of A. This means that the "features" or basis vectors used to represent I (columns of A) are the same as the receptive field weights (rows of W). Under these assumptions, we can think of the pattern of a receptive field as representing an image feature. Let's now summarize the basic linear algebra of vector representation in terms of basis vectors.

Basis sets

To be concrete, as earlier, assume our vectors live in an 8-dimensional space.

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
```

$$\left\{\frac{-\frac{37}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{15}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{\frac{29}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{\frac{31}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{31}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\right.$$
$$\frac{-\frac{25}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{5}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{\frac{69}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}},\ \frac{-\frac{37}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{15}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-$$
$$\frac{\frac{29}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{\frac{31}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{31}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{25}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{5}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{\frac{69}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}},$$
$$\frac{-\frac{37}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{15}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{\frac{29}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{\frac{31}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{31}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{25}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+$$
$$\frac{-\frac{5}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{\frac{69}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}},\ \frac{-\frac{37}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{15}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{\frac{29}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-$$
$$\frac{\frac{31}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{31}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{25}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{5}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{\frac{69}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}},$$
$$-\frac{-\frac{37}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{15}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{\frac{29}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{\frac{31}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{31}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}-$$
$$\frac{-\frac{25}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{5}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{\frac{69}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}},\ -\frac{-\frac{37}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{15}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-$$
$$\frac{\frac{29}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{\frac{31}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{31}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{25}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{5}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{\frac{69}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}},$$
$$-\frac{-\frac{37}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{15}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{\frac{29}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{\frac{31}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{31}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+$$
$$\frac{-\frac{25}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{5}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{\frac{69}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}},\ -\frac{-\frac{37}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{15}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+$$
$$\left.\frac{\frac{29}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}+\frac{\frac{31}{2\sqrt{2}}-\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{31}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}-\frac{-\frac{25}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{-\frac{5}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}+\frac{\frac{69}{2\sqrt{2}}+\sqrt{2}}{2\sqrt{2}}\right\}$$

Looks complicated, but if you simplify it:

```
Simplify[%]
```

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

you get back the original vector.

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

Spectrum

The projections, {**g.w$_i$**} are sometimes called the **spectrum** of **g**. This terminology comes from the Fourier basis set used in Fourier analysis. Also recall white light is composed of a combination of light of different frequencies (colors). 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.

Representing the output in the column space of the network

We've seen that the output can be represented as the projections of the input onto each of the row vectors of the weight matrix:

```
{{w11, w12, w13}, {w21, w22, w23}}.{x1, x2, x3} // MatrixForm
```

$$\begin{pmatrix} w11\ x1 + w12\ x2 + w13\ x3 \\ w21\ x1 + w22\ x2 + w23\ x3 \end{pmatrix}$$

Our image dictionary example showed that it is also useful to think of the output in terms of its of the column vectors. Suppose we have 3 inputs and 2 outputs to our network as above. Inputs to the network live in a 3-dimensional space. Outputs live in 2 dimensions.

With a 3-dimensional vector that represents the 3 inputs onto each of 2 neurons, one can visualize the output as a 2-dimensional vector whose length and direction is determined by the 2-dimensional vector sum of three column weight vectors: $\begin{pmatrix} w_{11} \\ w_{21} \end{pmatrix}$, $\begin{pmatrix} w_{12} \\ w_{22} \end{pmatrix}$, $\begin{pmatrix} w_{13} \\ w_{23} \end{pmatrix}$, where the amplitude of each vector is scaled by the input activity levels x_1, x_2, x_3, to give: $\begin{pmatrix} w_{11} \\ w_{21} \end{pmatrix} x_1 + \begin{pmatrix} w_{12} \\ w_{22} \end{pmatrix} x_2 + \begin{pmatrix} w_{13} \\ w_{23} \end{pmatrix} x_3$.

Neural models of image representation in the primary visual cortex have been analyzed in terms of whether the high-dimensional images received (at the retina) are projected into lower or higher dimensional spaces in cortex, and what the consequences might be for biological image processing (Hyvärinen, 2010).

Show that : $\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} . \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} w_{11} \\ w_{21} \end{pmatrix} x_1 + \begin{pmatrix} w_{12} \\ w_{22} \end{pmatrix} x_2 + \begin{pmatrix} w_{13} \\ w_{23} \end{pmatrix} x_3$

Some more linear algebra terminology

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 all 8-dimensional vectors in terms of this basis set. The seven vector set would be said to be **incomplete**. Imagine you want to specify the position of a point in your room, but you can only say how far the point is on the floor

from two walls. You don’t have enough measurements to also say how high the point is.

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. So you could specify the position of any point on the floor of your room with just two numbers that represent how far the point is from a corner in your room along one direction, and then how far from the corner it is in an orthogonal direction. The points on the floor span this two-dimensional subspace of your room.
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 (cf. Hyvärinen, 2010). 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. Think about representing points in your room, where in addition to saying how far from the corner the point is along two walls, you can also say how far the point is along some arbitrary ruler laying on the floor. The additional measurement from the arbitrary ruler doesn’t give you any information you couldn’t get from the other two measurements.
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?

We've just spent quite a bit of time focusing on linear neural networks. Linear neural network models are identical to discrete linear systems models, and thus provide the advantage of bringing a large body of knowledge from mathematics and signal processing to bear on their behaviors. Later in Lecture 6, we'll learn more about linear systems and systems analysis.

We've noted that more realistic models take into account the properties than neurons have thresholds and a limited range of firing, which we modeled using the above squashing function.
From a computational standpoint, the squashing function has both advantages and disadvantages. It is what makes our simple linear neural network model *non-linear*. We will see later that this non-linearity enables networks to compute functions, solve problems, that can't be computed with a linear network. On the other hand, non-linearities make the analysis more complicated because we leave the well-understood domain of linear algebra. However, 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 simpler.

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

A mathematica-based linear algebra course. http://library.wolfram.com/infocenter/MathSource/4611/
Olver, Peter J. and Shakiban, Chehrzad (2005) Applied Linear Algebra, Prentice-Hall, Upper Saddle River, N.J., 2005
http://www.math.umn.edu/~olver/ala.html
Hyvärinen, A. (2010). Statistical Models of Natural Images and Cortical Visual Representation. Topics in Cognitive Science, 2(2), 251–264. doi:10.1111/j.1756-8765.2009.01057.
Rieke, F. et al. (1997). Spikes : exploring the neural code. Cambridge, Mass., MIT Press.
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.
Also, take a look at Strang's lecture notes on youtube, which are nicely organized here: http://web.mit.edu/18.06/www/videos.shtml