Introduction to Neural Networks

Daniel Kersten

Lecture 4: The generic neuron and linear models

Introduction

Last time

Levels of abstraction in neural models
McCulloch-Pitts threshold logic: Discrete time, discrete signal, no spatial structure assumed for neuron
Integrate-and-fire model: continuous time, continuous signal, no spatial structure assumed for neuron

Today

We continue with the structure-less and continuous signal simplification, but discretize time, and from there build a network.
Introduce vector and matrix representations used for representing the pattern of neural firing rates and synaptic weights respectively.
Review basic linear algebra of vectors.

(Later in Lecture 6, we will review the properties of matrices. And in Lecture 7, we will see how to exploit the linear algebra of vectors and matrices to empirically analyze the linear properties of a neural system.)

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

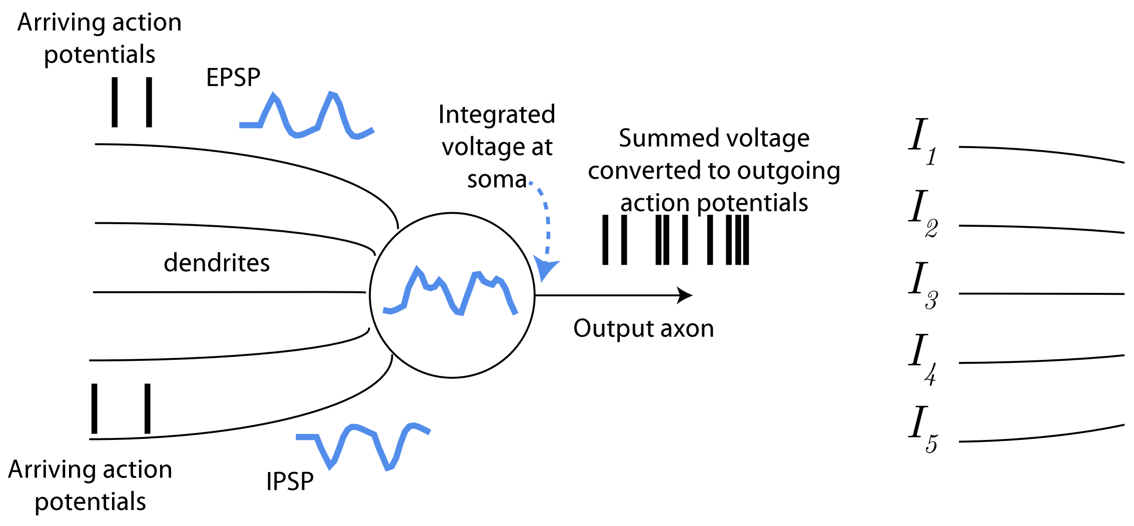
Build neural networks that filter images, filter sounds, model early sensory organ and cortical cells, detect edges, textures, front-end for motion estimation, describe human sensory thresholds...

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

Two stages: dot product, point-wise non-linearity

In[433]:= Image[{


Arriving action potentials
EPSP
Integrated voltage at soma
Summed voltage converted to outgoing action potentials
dendrites
Output axon
Arriving action potentials
IPSP
I_1
I_2
I_3
I_4
I_5

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 input levels 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. For example, the time period could be 100 msec, and signal rates might vary between 0 and 50 or so spikes per second.

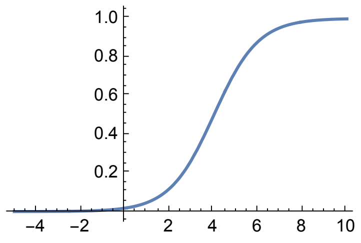
The right hand panel in the graph illustrates the basic model: The output of a generic neuron is the result of two stages: a weighted sum of the inputs, $\sum_i w_i\ I_i$ which is then fed through a non-linearity, $\sigma()$.

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. A fixed bias can be modeled as a fixed input, so the math is the same.

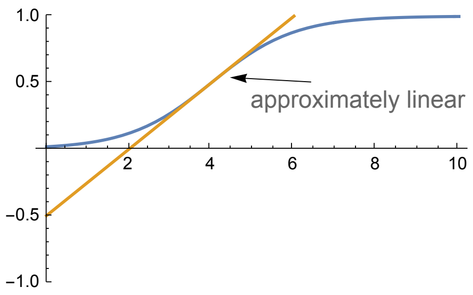
Stage 2: *Point-wise non-linearity*

The linear weighted sum is then fed through a simple non-linearity. Popular forms are “sigmoidal” or approximately so: logistic function, arctan(), semi-linear function. These are sometimes called "point" non-linearities, because the non-linearity is a scalar function of a scalar input (the sum of the synaptic inputs)--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).



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. However, additional computational power comes from the sigmoidal non-linearities.

The advantage of the linear approximation is simplification, and being able to exploit our knowledge of linear algebra and linear systems. So the linear assumption is a practical first step in modeling.



▶ 1. Use Plot to compare two sigmoidal non-linearities:

```
In[683]:= 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 four input signals (e.g. firing rates) to a neuron.

```
In[690]:= x = {2,3,0,1};
```

Sidenote : While it doesn't make sense for firing rate frequency, sometimes modelers let the input and outputs take on negative values, e.g. between - 1 and 1, with the assumption is that -1 and +1 can always be mapped on to numbers representing the smallest and biggest possible firing rates. 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.

```
In[691]:= w = {2,1,-2,3};
```

Dot product

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:

```
In[692]:= y = w.x
Out[692]= 10
```

You can also write:

```
In[693]:= y = Dot[w, x]
Out[693]= 10
```

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 homework, 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:

```
In[694]:= Clear[w1, w2, w3, w4, x1, x2, x3, x4];
          y = {w1, w2, w3, w4}.{x1, x2, x3, x4}
Out[695]= w1 x1 + w2 x2 + w3 x3 + w4 x4
```

The sigmoidal non-linearity

Now define a function to model the non-linearity of Stage 2 (in this case, the “logistic function” mentioned earlier):

```
In[696]:= 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?

► 2. Graphics. Define squash2[x_,s_] with a scale parameter that stretches or compresses the x dimension. Adjust the input scale, s, 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:

```
In[519]:= w
          x
Out[519]= {2, 1, -2, 3}
Out[520]= {2, 3, 0, 1}
```

Putting things together

Let’s put the the two stages together, producing the output of a generic neuron, with synaptic weights **w** and input **x** = {.2,.3,0,.2}:

In[521]:=

```
In[697]:= Clear[y];
          y[x_] := squash[w.x];
```

```
In[699]:= w.x
          y[x]
Out[699]= 10
Out[700]= 0.999955
```

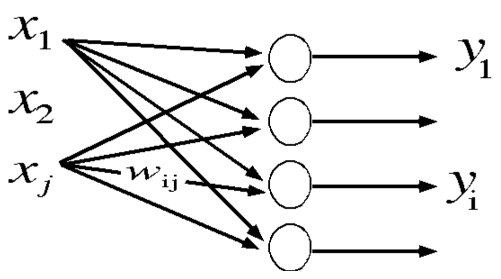
▸ 3. Exercise
Suppose all the inputs except the first are clamped at zero. What does the response, **y** look like as a function of x for various levels of input signal? Fill in the argument for Plot[]:

```
In[445]:= Plot[, {x, -2, 2}, PlotRange → {0, 2}, AxesLabel → {"Input signal, x", "Frequency"}]
```

Anything important missing from our model neuron?

Lots of details are missing, but one thing we haven’t done is to take into consideration a basic fact noted in the previous lecture: neuron responses are inherently variable. Given repetitions of identical inputs, the firing rate is variable. To a reasonable first approximation, the variance of firing rate is proportional to the mean. This suggests a Poisson model of firing rates. In later lectures, we will develop probability models in more detail. In the mean time, the Appendix of this notebook shows a simple way to add a third noise stage to the generic neuron model.

Modeling a simple neural network



Linear matrix model

What if the input

```
In[526]:= 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 a list of the four weight lists or vectors. Here is a 4x4 matrix W:

```
In[703]:= W = {{2,1,-2,3}, {3,1,-2,2}, {4,6,5,-3},{1,-2,2,1}}
Out[703]= {{2, 1, -2, 3}, {3, 1, -2, 2}, {4, 6, 5, -3}, {1, -2, 2, 1}}
```

► 4. 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 the product of the matrix **W** times the input vector **x**:

```
In[704]:= y = W.x
Out[704]= {10, 11, 23, -3}
```

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

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

$\mathrm{Sum}[x_j\, w_{i,j}]$

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:

```
In[705]:= c = 1/Sqrt[x.x];
          x2 = N[c x]
Out[706]= {0.534522, 0.801784, 0., 0.267261}
```

► 5. 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.

```
In[708]:= y
          squash[y]
Out[708]= {10, 11, 23, -3}

Out[709]= {0.999955, 0.999983, 1., 0.0474259}
```

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**:

```
In[711]:= y = squash[W.x]
Out[711]= {0.999955, 0.999983, 1., 0.0474259}
```

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 negligible. But if we want to add noise to the model, we could use a random number generator to add noise to each of the four outputs (see Appendix). Note that second argument of **RandomVariate** specifies the dimensions of the input:

```
y = squash[W.x] + RandomVariate[NormalDistribution[0.0, .1], 4]
{0.930125, 0.828765, 1.00453, -0.0880496}
```

► 6. 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:

```
        Clear[y];
         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,

```
In[712]:= y
Out[712]= {0.999955, 0.999983, 1., 0.0474259}

In[713]:= y[[2]]
Out[713]= 0.999983
```

Given a matrix w:

```
In[714]:= MatrixForm[W]
Out[714]//MatrixForm=
```

$$\begin{pmatrix} 2 & 1 & -2 & 3 \\ 3 & 1 & -2 & 2 \\ 4 & 6 & 5 & -3 \\ 1 & -2 & 2 & 1 \end{pmatrix}$$

we can access the the element in the second row, third column of **W**:

```
In[715]:= W[[2,3]]
Out[715]= -2
```

which you can also access as:

```
In[716]:= W[[2]][[3]]
Out[716]= -2
```

The first row is:

```
In[718]:= W[[2]]
Out[718]= {3, 1, -2, 2}

In[719]:= {3, 1, -2, 2}[[3]]
Out[719]= -2
```

You can use **Span[]** or its shorthand ;; to get rows, columns or submatrices.

(For Matlab or Python users, compare with “slicing” syntax.)

```
In[536]:= W3x3 = {{w11, w12, w13}, {w21, w22, w23}, {w31, w32, w33}};

In[537]:= W3x3 // MatrixForm
Out[537]//MatrixForm=
```

$$\begin{pmatrix} w11 & w12 & w13 \\ w21 & w22 & w23 \\ w31 & w32 & w33 \end{pmatrix}$$

Here are the first two rows of W3x3:

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

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

Second and third columns of W3x3:

```
In[538]:= W3x3[[ ;; , 2 ;; 3]] // MatrixForm
Out[538]//MatrixForm=
```

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

or

```
In[539]:= W3x3[[All, 2 ;; 3]] // MatrixForm
Out[539]//MatrixForm=
```

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

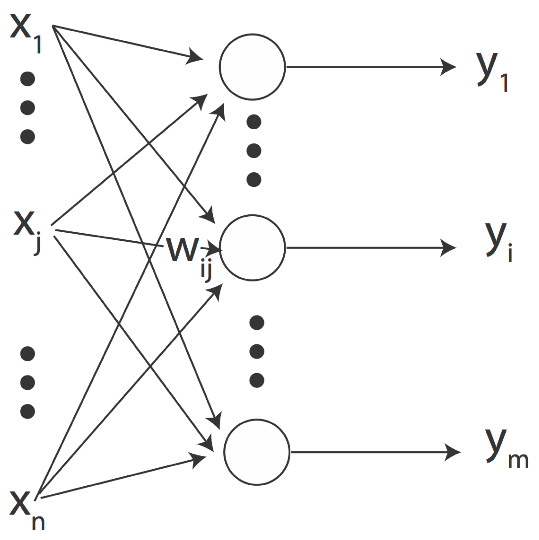
Lower right 2x2 submatrix of W3x3, which is rows 2 to 3, and columns 2 to 3:

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

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

Rectangular matrices

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 n inputs to m outputs is given by an m x n 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:

```
In[720]:= 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 the “square matrix” case, m = n:

```
In[726]:= W2x3.xin // MatrixForm
          W3x2.xin2 // MatrixForm
          W3x3.xin // MatrixForm
Out[726]//MatrixForm=
          ( w11 x1 + w12 x2 + w13 x3 )
          ( w21 x1 + w22 x2 + w23 x3 )

Out[727]//MatrixForm=
          ( w11 x1 + w12 x2 )
          ( w21 x1 + w22 x2 )
          ( w31 x1 + w32 x2 )

Out[728]//MatrixForm=
          ( w11 x1 + w12 x2 + w13 x3 )
          ( w21 x1 + w22 x2 + w23 x3 )
          ( w31 x1 + w32 x2 + w33 x3 )
```

▸ 7. If there are k inputs to a neuron, how many rows and columns should the weight matrix have?

Some 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*".

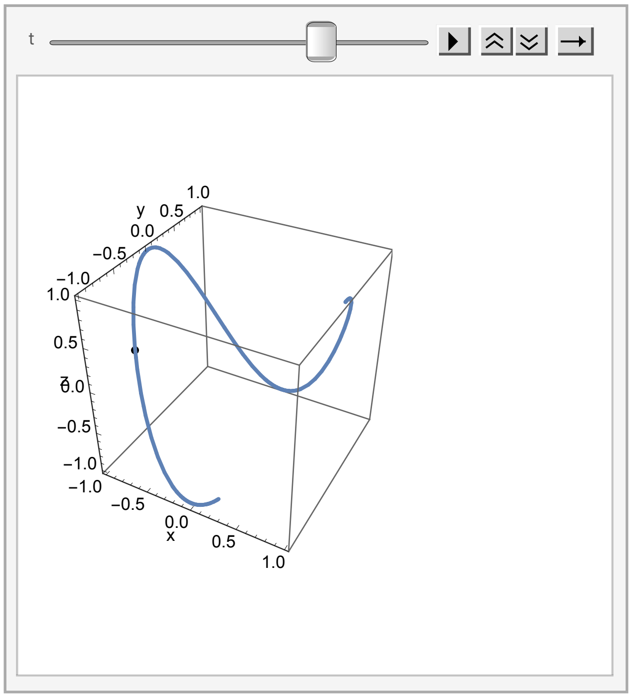
Vector operations and patterns of neural activity

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 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:

```
In[729]:= Clear[y];
y[t_] := {Cos[t], Sin[t], Cos[2 * t]};
Animate[Show[ParametricPlot3D[y[t], {t, 0, 5}, AxesLabel -> {"x", "y", "z"},
    ImageSize → Small], Graphics3D[{PointSize[.025], Point[y[t]]}]],
 {t, 0, 5}, AnimationRunning -> False]
```

Out[731]=



This simple state space model shows that although a network may have 3 outputs, its actual dynamics can be constrained to a smaller space, which in the above is the one dimension along the curved line.

Next we review some basic definitions, properties and operations on vectors and their implementations in Mathematica.

Dimension of a vector.

```
In[732]:= v = {2.1, 3, -0.45, 4.9};

In[735]:= Dimensions[v]

Out[735]= {4}
```

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

```
In[736]:= Length[v]

Out[736]= 4

In[737]:= M = {{2,4,2}, {1,6,4}};

In[617]:= Length[M]

Out[617]= 2

In[739]:= Dimensions[M][[2]]

Out[739]= 3
```

► 8. 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, unlike Matlab, 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 $\mathbf{x}^{\mathrm{T}}$. You can see a vector in column form by typing **v//MatrixForm**, or:

```
In[618]:= MatrixForm[v]

Out[618]//MatrixForm=
```

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

Vector addition

Vector addition accomplished by simply adding the components of each vector to make a new vector. Note that the vectors all have the same dimension.

```
In[740]:= a = {3,1,2};
          b = {2,4,8};
          c = a + b

Out[742]= {5, 5, 10}
```

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

```
In[745]:= 2*a
          2 a
Out[745]= {6, 2, 4}

Out[746]= {6, 2, 4}
```

Euclidean "length" of a vector

It is potentially confusing 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.

```
In[625]:= Remove[x1, x2, x3]
          Norm[{x1, x2, x3}]
```

Out[626]= $\sqrt{\text{Abs}[x1]^2 + \text{Abs}[x2]^2 + \text{Abs}[x3]^2}$

The euclidean length of a vector of neural activities is one way to summarize the overall signal “strength” of a population. The direction of the vector represents the “pattern” of activity.

To get a little more practice with *Mathematica*, you can also calculate the norm of a vector 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]]
```

Out[748]= $\{x1^2, x2^2, x3^2\}$

Out[749]= $\sqrt{x1^2\ x2^2\ x3^2}$

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.

```
In[564]:= Norm[u,1]//TraditionalForm
Out[564]//TraditionalForm=
          1
```

► 9. 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

Here is a bit more information about dot products. We’ve seen that to calculate the dot product of two vectors, you multiply the corresponding components and add them up:

```
In[565]:= Clear[u1,u2,u3,u4,v1,v2,v3,v4];
          u = {u1,u2,u3,u4};
          v = {v1,v2,v3,v4};
          u.v

Out[568]= 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:

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

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.

```
In[569]:= N[Sqrt[a.a]]

Out[569]= 3.74166

In[570]:= Sqrt[u.u]
```

Out[570]= $\sqrt{u1^2 + u2^2 + u3^2 + u4^2}$

▸ 10. 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 is said to *"project"* 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:

```
In[571]:= vn = v/Sqrt[v.v] ;
```

or using the built-in function Normalize[]:

```
Clear[v, v1, v2, v3, v4];
v = {v1, v2, v3, v4};

In[635]:=

vn = Normalize[v]
```

Out[635]= $\left\{\frac{v1}{\sqrt{\text{Abs}[v1]^2 + \text{Abs}[v2]^2 + \text{Abs}[v3]^2 + \text{Abs}[v4]^2}}, \frac{v2}{\sqrt{\text{Abs}[v1]^2 + \text{Abs}[v2]^2 + \text{Abs}[v3]^2 + \text{Abs}[v4]^2}}, \frac{v3}{\sqrt{\text{Abs}[v1]^2 + \text{Abs}[v2]^2 + \text{Abs}[v3]^2 + \text{Abs}[v4]^2}}, \frac{v4}{\sqrt{\text{Abs}[v1]^2 + \text{Abs}[v2]^2 + \text{Abs}[v3]^2 + \text{Abs}[v4]^2}}\right\}$

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 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-dimensional 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.

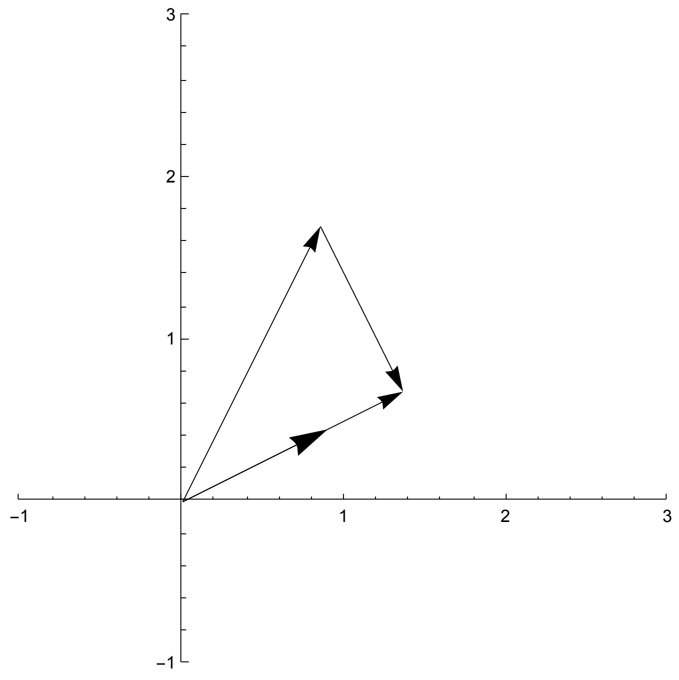
```
In[751]:= x = .85*{1,2};
          w = N[{2/Sqrt[5],1/Sqrt[5]}];
          z = (x.w) w
Out[753]= {1.36, 0.68}
```

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

```
In[754]:= z = Projection[x, w]
Out[754]= {1.36, 0.68}
```

```
In[755]:= 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]
```

Out[755]=



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:

```
In[578]:= 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:

```
In[651]:= a = {2,1,3,6};
          b = {6, 3, 9, 18};
          VectorAngle[a, b]
          cosine[a,b]

Out[653]= 0

Out[654]= 1
```

▶ 11. Verify 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:

```
In[583]:= a = {-2,-1,-3,-6};
          b = {6, 3, 9, 18};
          VectorAngle[a, b]
          cosine[a,b]

Out[585]= π

Out[586]= -1
```

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

```
In[587]:= a = {-2, -1, -3, -6};
          b = {6, 3, 9, 12};
          {a2, b2} = Orthogonalize[{a, b}];
          cosine[a2, b2]
          VectorAngle[a2, b2]

Out[590]= 0

Out[591]= π/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 vector lengths. Another measure of similarity is the Euclidean length of the difference between two vectors, or the "distance between the tips of their vectors":

```
In[655]:= N[Norm[a - b]]
Out[655]= 14.1421
```

Or you can use:

```
In[658]:= EuclideanDistance[a, b]
```

Out[658]= $10\sqrt{2}$

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

▶ 12. Just by thinking about the development of geometry of ancient Greece, 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:

```
In[659]:= 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};

In[667]:= cartesianset = {u1, u2, u3, u4, u5, u6, u7, u8};
```

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, which can be defined using the built-in HadamardMatrix function:

```
In[758]:= walshset = 2 Sqrt[2] HadamardMatrix[8] // MatrixForm
```

Out[758]//MatrixForm=

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \end{pmatrix}$$

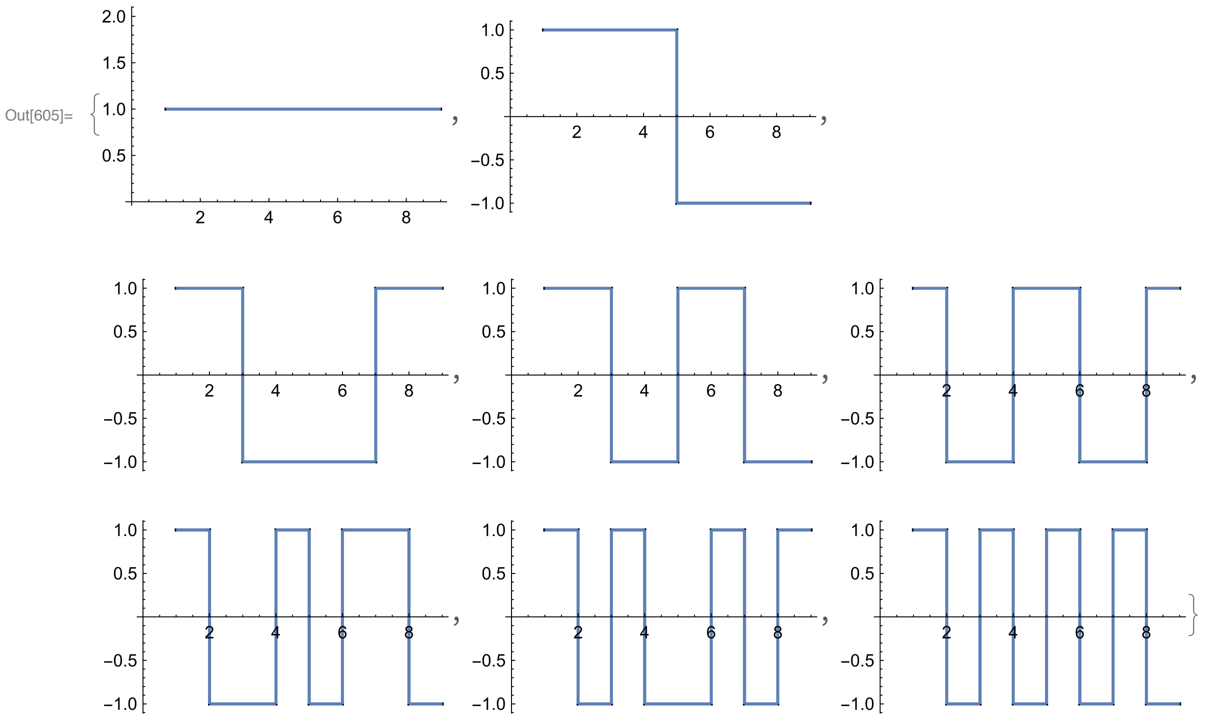
An important, decades-old, technique for the analysis of neural systems is to study their response to periodic inputs, in particular to sine waves of various frequencies. This is part of what is called ***linear***

systems analysis, which we will develop in detail in Lecture 7.

Walsh functions are the binary-valued analog to sine waves, which one can see by plotting out their forms:

In[605]:=
```
Table[ListStepPlot[walshset[[i]]], {i, 1, 8}]
```

Out[605]=



The above Cartesian and Walsh examples are just two of an infinite number of possible orthogonal sets.

▶ 13. Calculate the inner products between any two Walsh vectors. Which pairs are zero?

Note that with the first set of vectors, the cartesian set, {$\mathbf{u}_i$}, you can tell which vector it is just by looking for where the 1 is. For the second set, Walsh 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. Let’s look at this distinction from the point of view of the “neural code”.

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 firing rates, 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 identity, 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 value of $\mathbf{u}_3$[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 i's), 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 non-identical vectors in the Walsh set we generated above are 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:

```
In[670]:= {v1, v2, v3, v4, v5, v6, v7, v8} = walshset;

In[671]:= 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];
```

Note that as the default, the built-in function Hadamard[8] produces an orthonormal set.

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.

▸ 14. 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?

Next time we'll go into a bit more depth to show how the columns of weight matrices can be interpreted as "features" in a sensory input, and neural activity can represent how much weight these features get given a sensory input.

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

Appendix

Modeling a noisy neuron

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.

We'd like to add a third stage 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;λ) 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

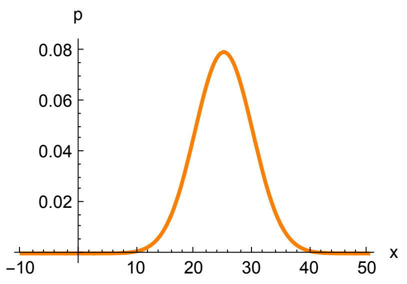
```
ndist = NormalDistribution[25.,Sqrt[25.]];
```

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

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

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

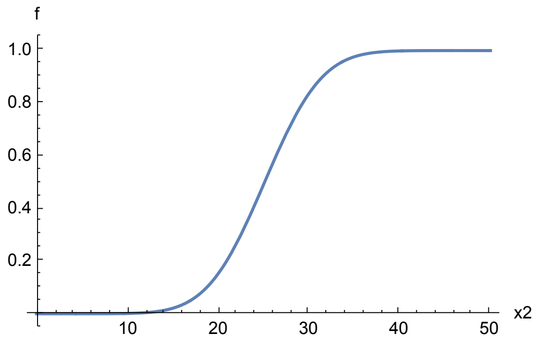
```
gndist = Plot[PDF[ndist, x], {x, -10, 50},
  AxesLabel → {"x", "p"}, PlotStyle → {Orange, Thick}]
```



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, 50}, AxesLabel → {"x2", "f"}]
```

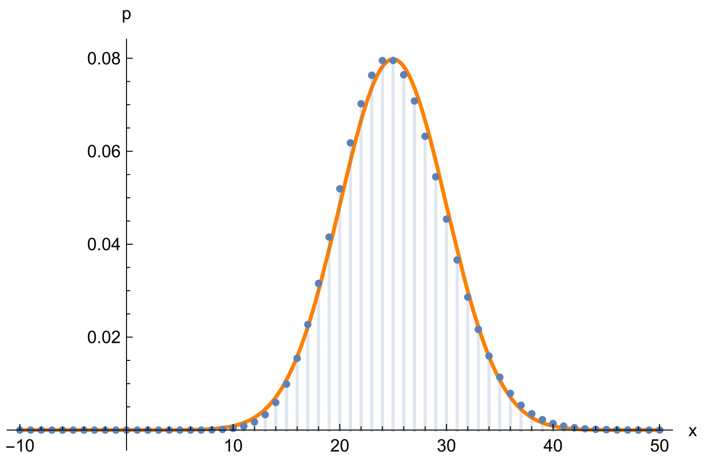


You can see from the graph that for this distribution, once x2 is greater than 0.7 or so, the probability of x being less than that is virtually certain, i.e. is essentially 1. If we set the mean = 0, and the standard deviation, we'd have a graph of the "cumulative **normal**".

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 continuousvalued 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]
```

```
32.6511
```

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]
```

```
25.4723
```

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

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

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

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}]
```

```
1.0266
```

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.