Introduction to Neural Networks
U. Minn. Psy 5038
Spring, 1998
Daniel Kersten

Lecture 3

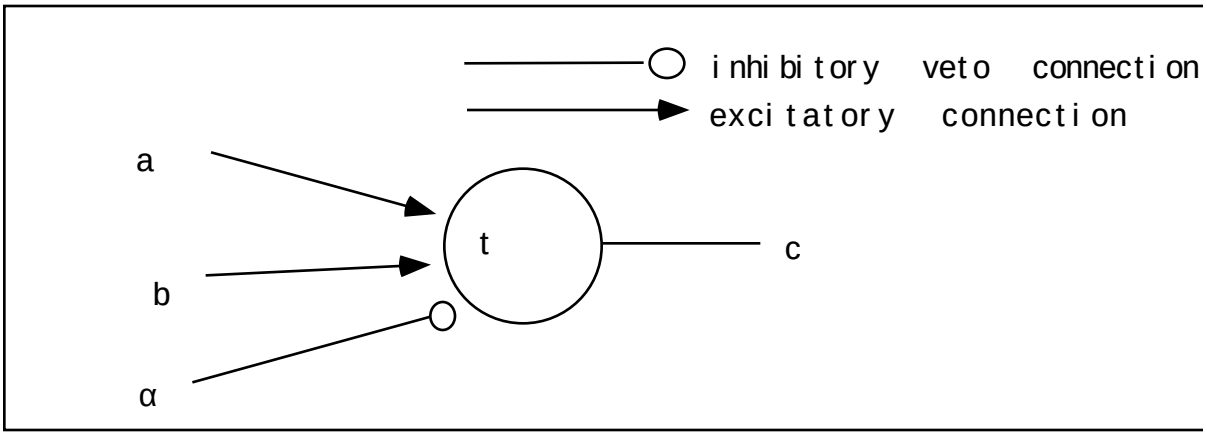
McCulloch-Pitts

Published in 1943, the McCulloch-Pitts model is famous and important because it showed that with a few simple assumptions, networks of of neurons may be capable of computing the full repertoire of logical operations. Although some of the basic facts about the physiology were wrong, the notion of neurons as computational devices remains with us.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5, 115-133.

Apart from the understanding of the biochemical basis of neural transmission, by the 1940's the basic signalling properties of neurons were well-known. In a seminal paper, McCulloch and Pitts formalized what was known, and developed a theory of neural networks that related the functioning of the brain to the then infant field of digital computers. They made the following assumptions:

- neuron signals were all-or-none
- a certain fixed number of synapses must be excited within a latent period of addition to excite the neuron
- the number of synapses is independent of previous activity and position on the neuron
- the only significant delay is synaptic delay
- there are excitatory and inhibitory synapses
- structure of the net does not change with time



McCulloch-Pitts: Inclusive OR, AND

For simplicity, let's set alpha to zero (no inhibition).The McCulloch-Pitts neuron sums its (binary) inputs, tests to see if the sum is bigger or less than the threshold. If bigger or equal, the output is set to 1, otherwise it is set to 0.

We can model the McCulloch-Pitts neuron's response like this:

```
McCullochPitts[a_, b_, t_] := If[a + b ≤ t, 1, 0];
```

The McCulloch-Pitts neuron is said to be computing *threshold logic*.

■ Inclusive OR

Let's set the threshold to 1, and find out what kind of function the neuron is computing:

$$c = \begin{cases} 1 & \text{if } a + b \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

```
McCullochPitts[1, 1, 1]
```

```
1
```

Try it out for various values of a and b. What happens if you put a semi-colon after the above expression, and then evaluate?

There are only 2 inputs with 2 possible values each, so let's list all the neuron's possible responses, defining a truth table. We use two *Mathematica* functions, **Table[]**, and **Flatten[]** to define a list, we'll call **truthtable**. **Table[]** is used to make lists, and because we have two indices (a, b), and a list as the first argument to **Table[]**, it makes a list of lists of lists.

```
Table[{a, b, McCullochPitts[a, b, 1]}, {a, 0, 1}, {b, 0, 1}] // StandardForm
```

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

We use **Flatten[,1]** to flatten truthtable to one level to put it in a standard "truth table" format, and view the result in "MatrixForm" or "TraditionalForm" which is the standard default in *Mathematica* 3.0.

```
truthtable =
 Flatten[Table[{a, b, McCullochPitts[a, b, 1]}, {a, 0, 1}, {b, 0, 1}], 1] //
  TraditionalForm
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Sidenote: *Mathematica* provides a wide range of list operations which you can read about in "**Lists and Matrices: List Operations**" in the **Built-in Functions** under **Help**. *Mathematica* also allows type-mixing in lists, so for example we can insert labels into our truthtable like this

```
Insert[truthtable, {"a", "b", "c"}, {1, 1}]
```

$$\begin{pmatrix} a & b & c \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Compare with *Mathematica*'s predefined **Or[]** function:

```
truthtable = Flatten[Table[{a, b, Or[a == 1, b == 1]}, {a, 0, 1}, {b, 0, 1}], 1]
```

$$\begin{pmatrix} 0 & 0 & \text{False} \\ 0 & 1 & \text{True} \\ 1 & 0 & \text{True} \\ 1 & 1 & \text{True} \end{pmatrix}$$

■ AND

Find a threshold value that will enable a McCulloch-Pitts neuron to realize the **And[]** function whose truth table is shown below.

```
truthtable = Flatten[Table[{a, b, And[a == 1, b == 1]}, {a, 0, 1}, {b, 0, 1}], 1]
```

$$\begin{pmatrix} 0 & 0 & \text{False} \\ 0 & 1 & \text{False} \\ 1 & 0 & \text{False} \\ 1 & 1 & \text{True} \end{pmatrix}$$

Main conclusions

It is straightforward to show that **Not** can be implemented with the inhibitory input. **And**, **Or**, and **Not** are a complete set in the sense that any logical operation can be built out of them.

Main result of 1943 paper:

Any finite logical expression can be realized by McCulloch-Pitts neurons.

This gave rise to the idea that the brain was very much like a digital computer. John von Neumann made explicit reference to the McCulloch-Pitts model in his famous 1945 technical report on the EDVAC.

How good is the McCulloch-Pitts model? Progress in theory and experimental findings led to a significant change in the way neural systems were seen to operate. What were these developments?

• information theory and statistical decision theory

emphasized the statistical nature of information transmission and processing in the presence of uncertainty and noise that was more characteristic of "real world" information processing.

• Psychophysics and signal detection theory showed the importance of false alarms in decision making

• Physiological data on inherent statistical and analog nature of sensory coding by neurons became apparent, e.g. input intensities are probably coded in terms of average frequency of spike trains, not in terms of a precise sequence--but this is still debated. The McCulloch-Pitts model is a structure-less discrete time model--sensory experiments suggested that "structure-less continuous signal, and continuous-time" models would be better.

• Hodgkin-Huxley model of neural discharge added substantial richness to our understanding of the mechanism of spike generation, and neural conduction.

Around 1956, from his deathbed, John von Neumann wrote:

"The language of the brain is not the language of mathematics" and

"..the message-system used in the nervous system, as described in the above, is of an essentially *statistical* character. In other words what matters are not the precise positions of definite markers, digits, but the statistical characteristics of their occurrence, i.e. frequencies of the periodic or nearly periodic pulse-trains, etc." Italics are his.

He predicted that brain theory would eventually come to resemble the physics of statistical mechanics and thermodynamics. Later on in this course, we'll see how von Neumann's prediction has come true, at least in theoretical modeling of brain functioning. (Von Neumann died in February, 1957)

Von Neumann, J. (1958). The Computer and the Brain. New Haven: Yale University Press.

Let us now examine more closely the rationale for continuous-response, and continuous-time models, and look at some simple models of the neuron that incorporate a frequency response. Then we will formally introduce the "generic neuron model" that we will use for most of this course.

Integrate and fire model of the neuron

One major limitation of the McCulloch-Pitts model is that it assumed that the fundamental language of neural communication was binary. From sensory studies, we know that neurons often encode information about stimulus intensity in terms of rate of firing.

Let's see how this might arise with some simple assumptions about how action potentials are generated.

Let u(t), and s(t) represent the membrane voltage potential and stimulus input (e.g. ionic current) to a neuron, respectively. We assume because of the membrane's capacitance, that the rate of change of the membrane potential is proportional to the input current, and so over time the potential, u(t) grows as more and more current pumps into the cell. When u(t) reaches some critical value, an action potential is generated, after which the neuron returns to it's resting state, and begins integrating again.

$$\frac{du}{dt} = s(t)$$

$$u(t) = \int_{t_1}^{t_2} s(t)dt$$

For small time intervals, and a smooth input, the integral is approximately the area of the rectangle under s:

$$\theta = \int_{t_1}^{t_2} s(t)dt \approx s(t_2 - t_1)$$

$$= s\Delta t$$

After time Δt, the neuron reaches threshold θ. So the time (or period) between spikes is θ/s. The frequency of firing is the reciprocal of the period:

$$\frac{1}{\Delta t} = \frac{s}{\theta} = f$$

So frequency of firing is proportional to the input current level.

As we will see below, this model assumes that the membrane integrates current with no leakage--i.e. it is a pure capacitance, with no resistance. We can improve the model by including both a resistive and capacitance elements to the equation--this is a leaky integrate-and-fire or "forgetful integrate-and-fire" model. The calculus gets a bit more sophisticated, so we'll use *Mathematica* to solve the equations for us.

So in a moment, we will derive the relationship between stimulus input level and the frequency of firing for the "forgetful integrate-and-fire" model. First, we go over some more *Mathematica* basics, and then develop a few tools.

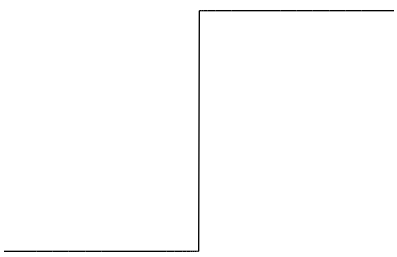
Rules and defining functions.

In *Mathematica*, you can define functions in terms of rules. One use of rules is to define functions over specific ranges. For example,

```
Clear[step]
step[x_] := 1 /;  x >= 0
step[x_] := -1 /; x < 0
```

Here **/;** means the function is defined with the following rule. The rules can be incompatible and *Mathematica* will evaluate the rules in a specific order, usually the order that you specified. You can find out what order the rules will be evaluated by typing **?step**. The **Clear[]** function clears any prior definition of the function **step[]**. **Clear** can take multiple arguments (e.g. **Clear[f,g,h]**).

```
Plot[step[x], {x, -3, 3}, Axes->False]
```



The rule for replacing a variable with a value in an expression is denoted by the operator **/.** meaning **given that** and **->** meaning **goes to**:

```
d + 2 e /. d->3 e
```

```
5 e
```

Derivatives and integrals.

The derivative of f[x], with respect to x is D[f[x], x]. For example, here is the derivative of x^3:

```
D[x^3, x]
```

$$3\ x^2$$

We can use *Mathematica* to calculate the indefinite integral of this function:

```
Integrate[3 x^2, x]
```

$$x^3$$

You can also do a numerical integration, which is particularly useful when a closed form solution isn't available:

```
NIntegrate[3 x^2, {x, 0, 2}]
```

```
8.000000000000000003
```

Differential equations.

■ Some demonstration examples of differential equation solution

The dynamics of many natural systems can be described in terms of differential equations. Later on we will see how the dynamics of models of large scale neural systems can be described in terms of coupled differential equations. A differential equation is a set of constraints on the rates of change of some dependent variables. Given these rates of change, we would often like to find out how the dependent variable itself changes with time. For example, basic electronics tells us that the charge across a capacitor is proportional to the voltage: q = CV, where C is a constant called the capacitance. Current, s, is the rate of change of charge:

s = dq/dt = C dV/dt. (Note that this is the same relation we used above, but with V replacing u, and C=1).

Just for demonstration purposes, suppose, we know that s[t] = cos(t), and that at time t=0, V=0, then what is V(t)?

```
DSolve[{V'[t] - (1/C) Cos[t] == 0,
    V[0] == 0}, V[t], t]
```

$$\left\{\left\{V(t) \to \frac{\sin(t)}{C}\right\}\right\}$$

You have probably noticed that this problem could have been easily solved by integration, (e.g. using **Integrate[]**). But as you will see below for integrate-and-fire neurons, you can't always simply solve an integral to find the solution. In particular, this happens when the rate of change of V depends on V itself. For example, here is a second order (second order because it involes a second derivative) equation for an oscillator. The acceleration of a mass on an ideal spring is proportional to the displacement: $d^2X/dt^2 = - k x$. Let k = 1, and assume initial conditions X[0] =0, X'[0] = 1, then DSolve tells us that the mass on the spring will oscillate sinusoidally:

```
DSolve[{X''[t] +  X[t] == 0,
    X[0] == 0, X'[0] == 1}, X[t], t]
```

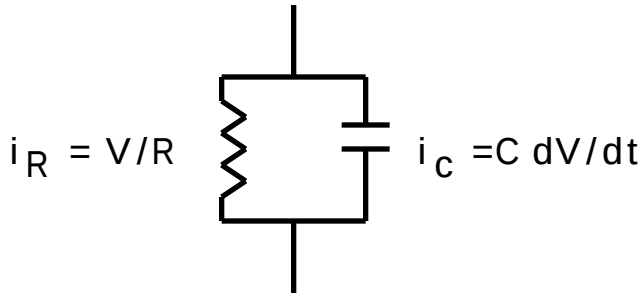
$$\{\{X(t) \to \sin(t)\}\}$$

The "Forgetful integrate-and-fire model" of the neuron using DSolve[]

The integrate and fire model of the neuron is a simple model which assumes (up until depolarization threshold) that the neuron membrane is a passive resistor and capacitor with some input current source. The current input leads to an increase in the resting potential until sufficient depolarization triggers an action potential.

We will develop the model in two parts. Our goal is to find how firing rate depends on the input current and threshold. First, we'll derive the relationship between membrane potential and time. Then we will derive the relationship between frequency of firing and input current, similar to how we did it for the simple integrate-and-fire model above.

The input current, s[t] is conserved and determines the sum of the current through the resistance and the capacitor, which using Ohm's law, and the definition of capacitance (q = CV) is summarized in the diagram.



By Kirchoff's current law, the current *in* equals the current *out*: :

$s = i_R + i_C$, or

s[t] = V/R + C dV/dt, or

dV/dt = s/C - V/(RC).

We can find the solution using the *Mathematica* function **DSolv**e, given the initial conditions that at time t=0, the voltage is **a**.

Let RC =1

```
DSolve[{V'[t] + V[t] - s/C == 0,
    V[0] == a}, V[t],t]
```

```
                  Exp[-t] (Exp[t] s + C (a - s/C))
{{V[t] -> ------------------------------------------}}
                                C
```

After t seconds, the voltage reaches threshold, theta, and a spike occurs. This occurs at time t. We can solve the above equation in terms of t:

```
Solve[theta == (a*C - s + E^t*s)/(C*E^t), t]
```

Solve::ifun : Inverse functions are being used by Solve, so some solutions may not be found.

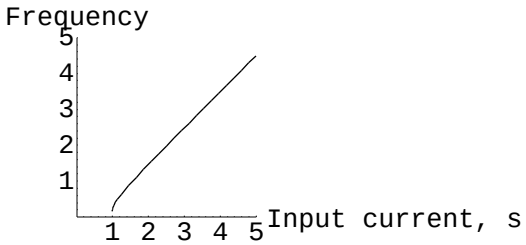
```
                    a C - s
{{t -> Log[-(-------------)]}}
                  s - C theta
```

E is 2.718..., and E^x is the same as Exp[x]. The frequency is 1/t and is thus given by the following function:

```
freq[s_,C_,a_,theta_] :=
    1/(Log[(-(a*C) + s)/(s - C*theta)]);
```

If we plot it for a capacitance of 1, and threshold of 1, frequency of firing as a function of input strength (current) looks like:

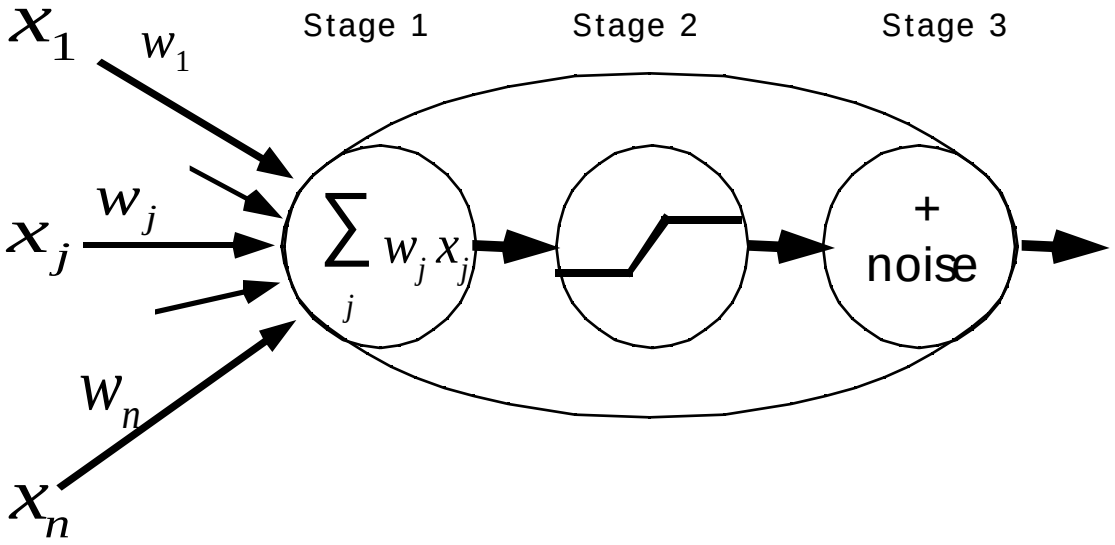
```
Plot[freq[s,1,0,1], {s, 0, 5},
    PlotRange->{{0,5},{0,5}},AspectRatio->1,AxesLabel->{"Input current,
s","Frequency"}];
```



To review what we've done, the "fire and integrate" neuron shows two properties characteristic of many neurons. First, it shows a threshold--it doesn't begin firing until the input current is sufficiently high. Second, once threshold is exceeded,

the frequency of firing grows in proportion to the input current. One characteristic we haven't modeled is the absolute refractory period. How would the shape of the above plot change if we included the effects of the absolute refractory period?

The generic connectionist model of the neuron



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

Stage 1:

Linear weighted sum of inputs

fixed bias term

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

Stage 2:

non-linearity

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

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

(Stage 3:)

noise

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

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

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

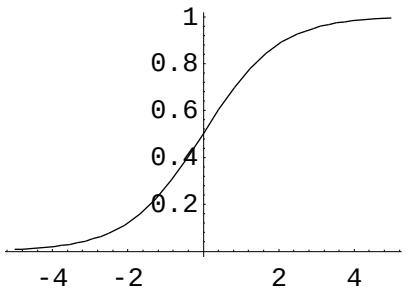
```
squash[x_] := N[1/(1 + Exp[-x])];
```

Recall, that the underscore, **x_ is important** because it tells Mathematica that x represents a slot, not an expression. Note that we've used a colon followed by equals (**:=**) instead of just an equals sign (=). When you use an equals sign, the value is calculated immediately. When there is a colon in front of the equals, the value is calculated only when called on later. So here we use **:=** because we need to define the function for later use.

Also note that our squashing function was defined with **N[]**. *Mathematica* trys to keep everything exact as long as possible and thus will try to do symbol manipulation if we don't explicitly tell it that we want numbers.

■ **Graphics.** Let's take the Exp (exponential function) and plot a graph of a non-linear squashing function:

```
Plot[squash[x], {x,-5, 5}];
```



This squashing function is often used to model the small-signal compression and large signal saturation characteristics of neural output.

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

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

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

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

Model linear neuron

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

```
y = w.x
```

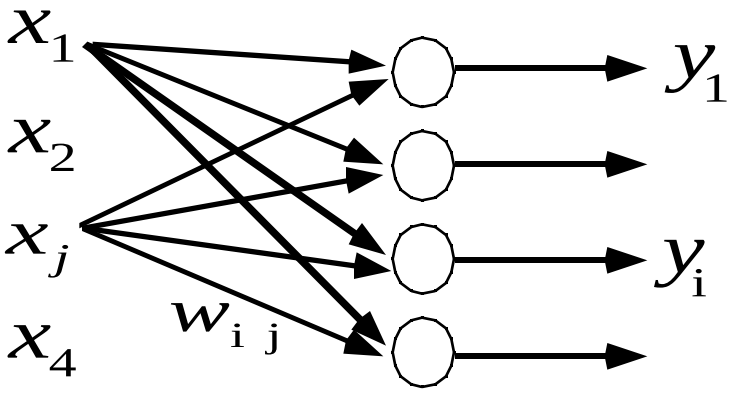
```
10
```

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

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

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

Modeling a simple neural network



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

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

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

```
y = M.x
```

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

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

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

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

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

```
squash[y]
```

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

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

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

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

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

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

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

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

```
0.999983
```

```
M[[2,3]]
```

```
-2
```

More on *Mathematica* functions for generating and plotting lists

Using Tables to make Lists.

Often we will have to define a list of input values to a neuron, or a list of synaptic weights. A convenient way of defining lists in *Mathematica* is to use the **Table[]** function.

For example, you can make a list whose elements are the squares of the element location.

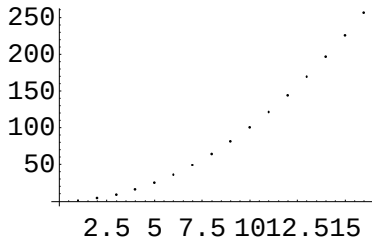
```
s = Table[x^2,{x, 1, 16}];
```

You can also use Table to make a list of lists, Here is a 16x16 matrix:

```
A = Table[x^2 + y^2, {x,1,16}, {y,1,16}];
```

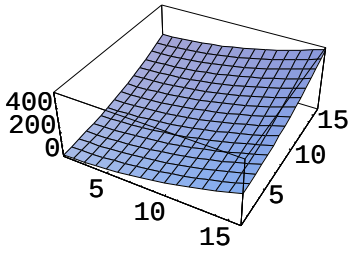
To graph a one-dimensional list, you have to use ListPlot:

```
ListPlot[s];
```



To graph a two-dimensional list, you have to use ListPlot3D:

```
ListPlot3D[A];
```



Optional Exercise

Write a *Mathematica* function that models the McCulloch-Pitts neurons for **Inclusive OR** and **AND** functions for inputs a, b, by defining a "step function", that is 1 for x ≤ theta, but 0 for x< theta: **step[x,theta]**