

# *The Perceptron and Friends*

## *Lesson 2*

### ■ Introduction

We've seen in our previous lesson that we can simulate a neural cell at the very low level of its electro-chemical processes. Indeed, this is quite helpful when trying to understand the functioning of individual cells and their transmission of the neural impulse (i.e. the progression of the action potential).

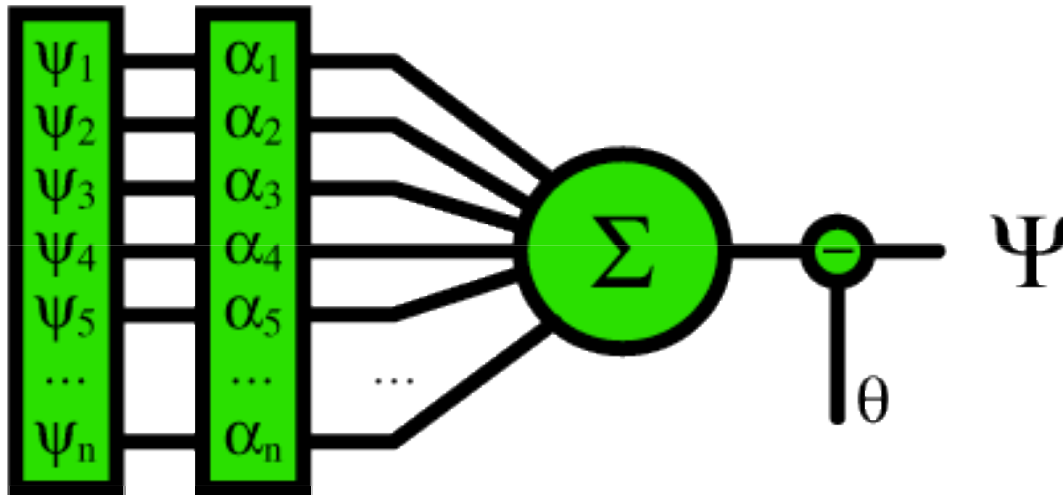
Our interests for this course lie mainly in the information processing actions of several (sometimes many many more) interconnected neurons. Computationally we can just ignore the *process* that causes the neuron to fire and, instead, concentrate on the net result of that firing. Should you protest our simplification, say, along the lines of “Well, couldn't the low level effects of the electro-chemical process have some slight effect on our result?” I present you with the following observation— Simulation and modelling must take place at some *scale* of analysis. Failure to fix our scale or scope predestines us to fail miserably. Should we model the electro-chemical processes? We're close to modelling some of the molecular processes when we do this, but what about the atomic processes themselves? What of the sub-atomic processes? Will our results hold up in different physical universes, those where *some* of our closely held rules don't apply?

I hope you appreciate that we're mainly interested in the *net result* of the electro-chemical and synaptic processes— that is, the firing of neurons and their connections to their neighbors, and how those firings can communicate information.

### □ The Perceptron

In the late 50's - early 60's Frank Rosenblat proposed several machines that laid the groundwork for the advent of artificial neural networks. These machines were called *Perceptrons*. These machines were based on the 1940's work of Warren McCulloch and Walter Pitts on the logical calculus of neural mechanisms. [1,2,3]

Rosenblat's machines were collections of units like those shown below. For this reason, the name *perceptron* has been used to denote not only the machines but (more commonly) the processing units themselves. Variations of these units go by the name *Adaline*, *Sigma-node*, and others.



**Figure 1.** A basic perceptron. Inputs,  $\psi_{1\dots n}$ , are multiplied by their respective weights,  $\alpha_{1\dots n}$ . These weighted inputs are summed and, if the resulting value exceeds some threshold,  $\theta$ , the output  $\Psi$  is a spike, usually represented by a '1'. Otherwise, there is no output, a '0'.

The perceptron is an analog to the biological neurons we described in the previous section. In essence, the perceptron is an artificial neuron that processes information from several inputs and, using some decision rule, and either activates its output or not. The inputs are analogous to the dendrites and the output represents the axon in a biological neuron. A collection of these assemblies will connect the outputs (axons) of some cells to the inputs of others, creating a virtual, artificial assembly.

In short, the perceptron collects several inputs, weights them, sums them, and then makes a decision to 'fire' or not based on this weighted sum.

Typically the inputs are binary – that is, they either are 1 or 0, representing the presence or absence of activity respectively. In fact, biological neurons behave largely in this manner. There is no differentiation between 'strengths' of firing, only the presence or absence thereof. Since the output of a perceptron is binary it follows that the resulting input to other cells will also be binary. The weights act to condition these inputs, emphasizing some and de-emphasizing others before they are summed. After this summation a decision is made to fire or not. The most basic type of decision is a so-called *threshold* decision. We will examine various sorts of decision functions in the upcoming sections.

#### **Notation**

There are many ways to notate the various inputs, weights, and other parameters. Here, we use the Greek lowercase psi,  $\psi$ , to denote the input n-tuple. The subscript selects one of the  $n$  elements from the list, so  $\psi_3$  is the 3rd input. (NB- Some folks refer to a list of this sort as a vector. As a point of fact, this is a semantic matter of interpretation, most notably with respect to magnitude. *Usually* the nature of the input of a neuron/perceptron is not interpreted to have a magnitude. Still, the term 'input vector' is frequently used and should be understood to be simply the list of inputs to the neuron. Some argue that the general form of these sorts of data structures follow more closely the idea of tensors. You are left to decide for yourself what makes the most sense and I welcome the feedback of any philosophical debates on the subject.)

Similarly, the weights are represented by the Greek lowercase letter alpha,  $\alpha$ . Subscripts represent individual elements of the list,  $\alpha_3$  being the third weight.

Summation is, as is typical, represented by the Greek uppercase sigma,  $\Sigma$ .

Finally, the input to the threshold function shown above is represented by the lowercase Greek letter theta,  $\theta$ .

The resulting state of the cell, on or off, is a binary value represented by the capital psi,  $\Psi$ .

Sometimes, for convenience, the input and output are represented by lowercase Roman letters 'i' and 'o' respectively with weights represented by 'w'.

Another common notation uses  $r_{in}$  to represent the input,  $w$  to represent the weights, and  $r_{out}$  to represent the output state of the cell.

Traditional mathematical notation for the above process is

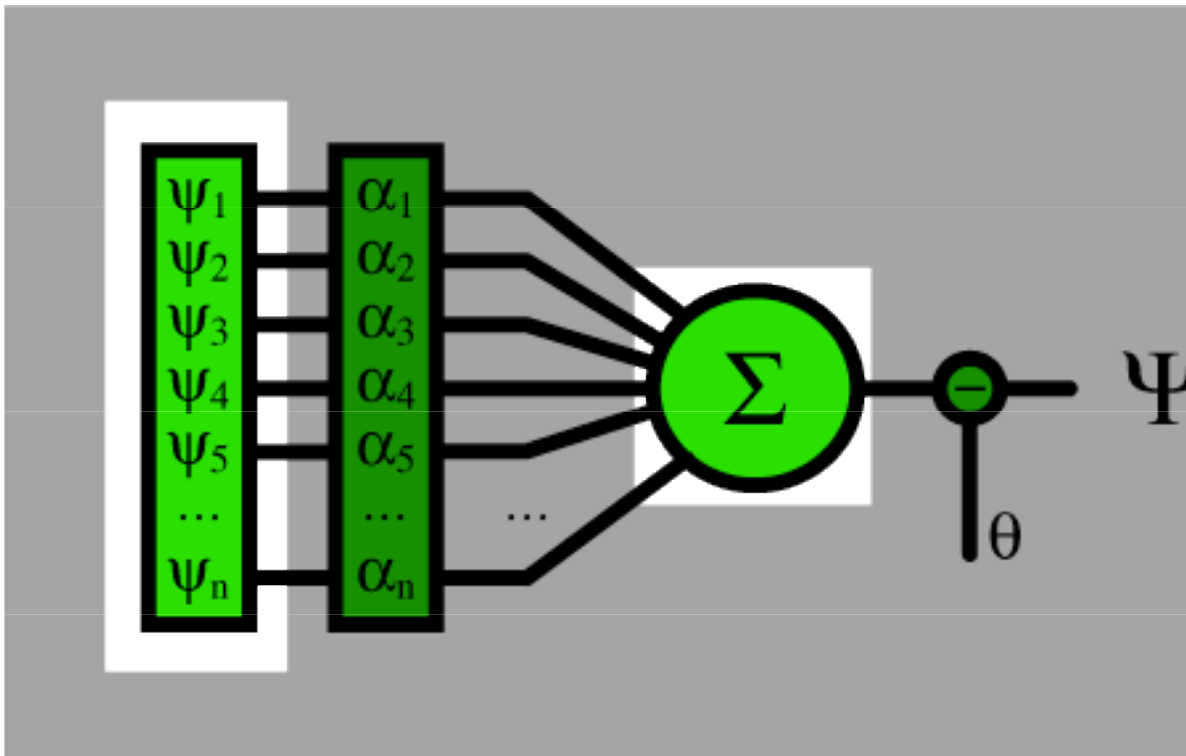
$$\Psi = \theta\left(\sum \psi \alpha\right) \quad (1)$$

Where  $\theta$  represents the Heaviside unit step function. In alternate implementations  $\theta$  is an adjustable threshold value. We will investigate several types of threshold functions (also called *activation functions*) below.

First, we'll decompose the process for those of you not totally familiar with this sort of calculation.

## ■ Simple Summation

To quote the philosopher Mr. T, "enough jibberjabber", let's get on with some computational examples. Let's start by examining the process of summation.



**Figure 2.** Examining the inputs and summation.

In *Mathematica* a list or vector or n-tuple is represented as a comma separated list, surrounded by curly-brackets (also known as braces or curly braces).

```
 $\psi = \{1, 0, 0, 1, 1\}$ 
```

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

Above we see a 5-tuple representing an input. The first and last two inputs are firing neurons, while the second and third are inactive.

We can sum the values of the list using this notation-

```
Plus @@  $\psi$ 
```

```
3
```

Here, Plus is said to be *applied* to the list (the two '@' signs denote application, shorthand for the *Mathematica* `Apply[]` function.) and we see the result, 3. So, in some overly simplistic, majority rules environment, we might say that, for a 5-input perceptron if 3 or more inputs are firing then we too shall fire.

This may be adequate but it doesn't allow any sort of fine-tuning of the inputs, it doesn't allow us to say that some inputs are more important than others, for example –

```
 $\psi = \{1, 0, 1, 1, 0\};$ 
```

(The semicolon at the end of the line suppresses output from *Mathematica*.  $\psi$  is set, we just don't see the results of setting it.)

```
Plus @@  $\psi$ 
```

```
3
```

In this case there are still three inputs firing but we can't differentiate the resulting sum from the previous example.

#### □ Explore

1. Try other combinations and different sizes of input vectors.
2. ☼ Can you figure out other ways to represent this process in *Mathematica*?

#### Example answers

1. Try other combinations and different sizes of input vectors.

```
 $\psi = \{1, 1, 1, 1, 0\};$ 
```

```
Plus @@  $\psi$ 
```

```
4
```

```
 $\psi = \{1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1\};$ 
```

```
Plus @@  $\psi$ 
```

```
8
```

2. ☼ Can you figure out other ways to represent this process in *Mathematica*?

```
Plus @@ {1, 1, 1, 0, 1, 1}
```

```
5
```

```
Apply[Plus, {1, 0, 1, 0, 1, 1}]
```

```
4
```

```

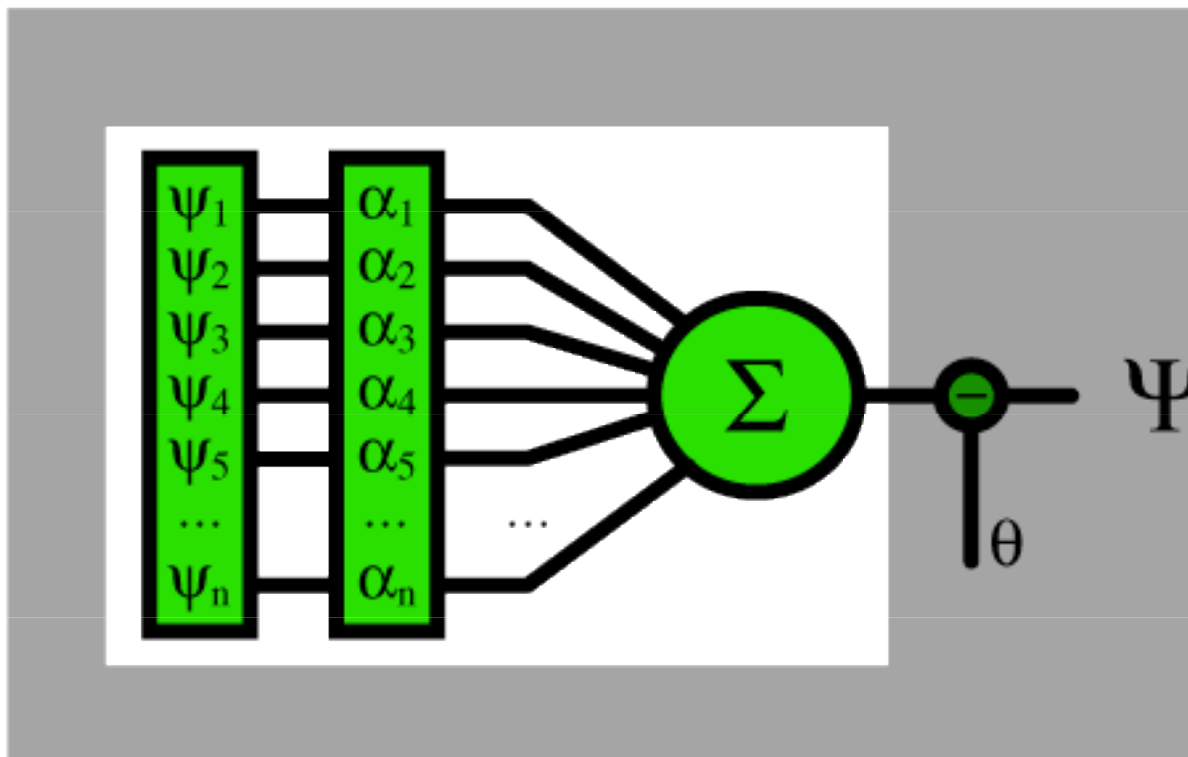
 $\psi = \{1, 0, 1, 0, 1, 1\};$ 
Apply[Plus,  $\psi$ ]
4

```

#### □ webMathematica

These exercises are in file L2E1.js (link!)

### ■ Weighted Summation



**Figure 3.** Weighting the inputs and summing them

Here we see our input  $\psi$  along with a set of weights  $\alpha$  —

```

 $\psi = \{1, 0, 0, 1, 1\};$ 
 $\alpha = \{-.2, 1.2, 3.3, -4.2, 1.3\};$ 

```

Multiplying the input vector by the weight vector

```

 $t = \psi \alpha$ 
 $\{-0.2, 0, 0, -4.2, 1.3\}$ 

```

results in a new set of values. This is sometimes called an *activation vector*. We sum the activation just like before —

```

Plus @@  $t$ 
-3.1

```

This final value (sometimes notated  $h$ ) is called the *activation value* or *net activation* of the perceptron.

In this case, different combinations of inputs will result in different output-

```
 $\psi = \{1, 0, 1, 1, 0\};$   
Plus @@ ( $\psi \alpha$ )
```

-1.1

```
 $\psi = \{1, 1, 1, 1, 0\};$   
Plus @@ ( $\psi \alpha$ )
```

0.1

So, you see here that the result of the summation isn't always binary (0,1) but rather takes on some real value as a result of the summation.

#### □ Explore

1. Try other combinations and different sizes of input vectors.
2. Try different weight vectors.
3. ☞ Can you figure out other ways to represent this process in *Mathematica*?

#### Example answers

1. Try other combinations of input vectors.

```
 $\psi = \{1, 1, 1, 1, 0\};$   
 $\alpha = \{-.2, 1.2, 3.3, -4.2, 1.3\};$   
 $t = \psi \alpha;$   
Plus @@  $t$ 
```

0.1

2. Try different weight vectors.

```
 $\psi = \{1, 1, 1, 1, 0\};$   
 $\alpha = \{-.2, 1.2, 3.3, -4.2, 1.3\};$   
 $t = \psi \alpha;$   
Plus @@  $t$ 
```

0.1

```
 $\alpha = \{-1.2, -1.2, 0.3, -0.42, 1.3\};$ 
```

```
 $t = \psi \alpha;$   
Plus @@  $t$ 
```

-2.52

3. ☞ Can you figure out other ways to represent this process in *Mathematica*?

```
Plus @@ ( $\{1, 1, 1, 0, 1\} \alpha$ )
```

-0.8

```
Apply[Plus,  $\{1, 0, 1, 0, 1, 1\} * \{.12, .12, -.2, -.4, 1.1, -0.02\}$ ]
```

1.

```
 $\psi = \{1, 1, 1, 0, 1, 1\};$   
 $\alpha = \{.12, .12, -.2, -.4, 1.1, -0.02\};$   
Apply[Plus,  $\psi \alpha$ ]
```

1.12

If students do this directly, sing!

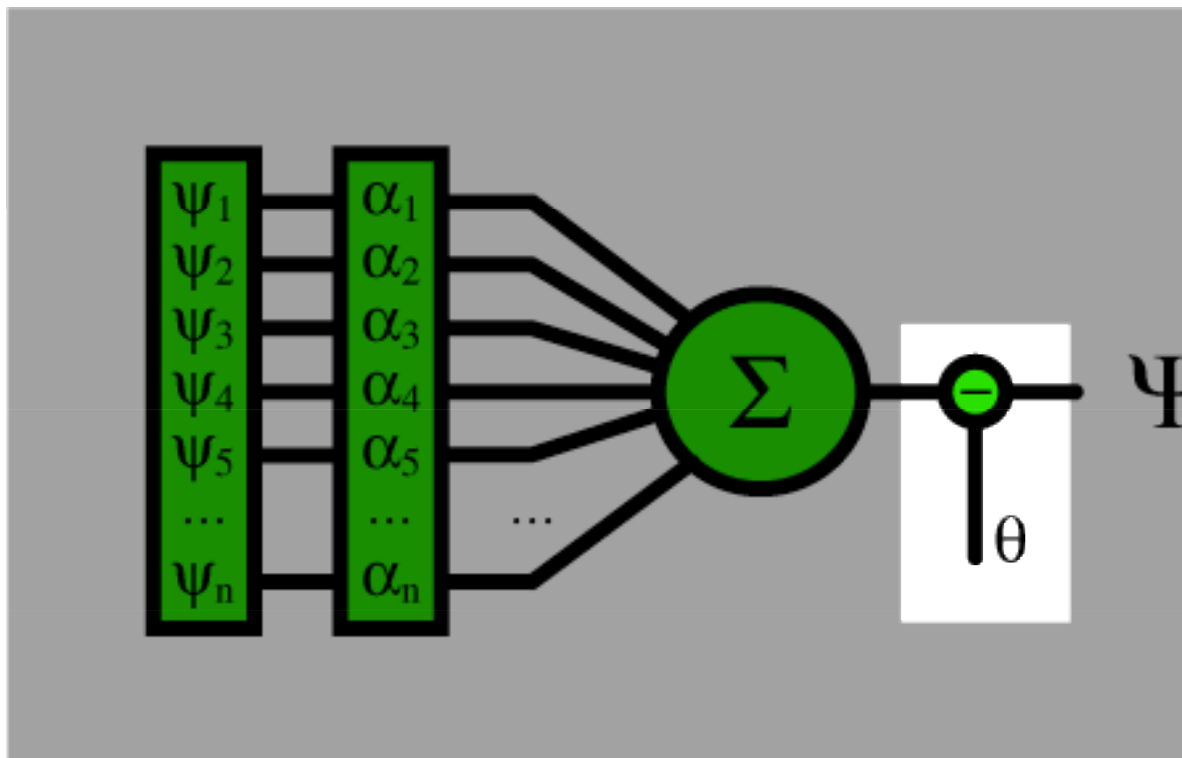
$\psi, \alpha$ 

1.12

#### □ webMathematica

These exercises are in file L2E2.js (link!)

#### ■ Weighted Summation with Threshold

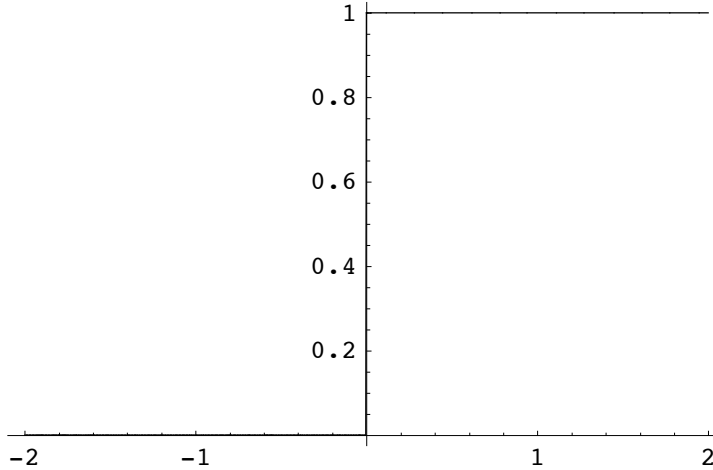


**Figure 4.** Adding the threshold function.

In a neural network, both physical and artificial, the outputs of neurons (axons) serve as inputs for other neurons (synapses). You may recall that we decided for this experiment to constrain ourselves to an all-or-nothing perceptron— i.e. either the artificial neuron fires or it does not. We represented these two states numerically by 1 and 0 respectively. We have a net activation value of  $h = -3.1$  in this example but we need to convert it to one of the two states above, 0 or 1.

*Mathematica's* `UnitStep[]` function returns 0 if its input is less than 0 and 1 otherwise. Therefore, we can use it as a simple decision function: If the sum of the weighted inputs is greater than 0 then we 'fire' (i.e. we pass a value of 1 down to other perceptron inputs), otherwise, we do not. As we stated above, the `UnitStep` function is an implementation of the Heaviside unit step function.

```
Plot[UnitStep[x], {x, -2, 2}];
```



Here is the math we're already familiar with —

```
 $\psi = \{1, 1, 1, 0, 0\};$   
 $\alpha = \{-2, 1.2, 3.3, -4.2, 1.3\};$ 
```

the weighted inputs —

```
 $t = \psi \alpha$   
{-0.2, 1.2, 3.3, 0, 0}
```

summed —

```
 $h = \text{Plus} @@ t$   
4.3
```

and finally passed through the UnitStep function —

```
 $\Psi = \text{UnitStep}[h]$   
1
```

Thus, for the given input  $\psi$  and weights  $\alpha$  our perceptron fires!

Notationally, as seen in Equation 1, we represent the unit step function with  $\theta(x)$ .

Mechanically, this is what the function does —

$$\theta(x) = \begin{cases} 1 & : x > 0 \\ 0 & : x \leq 0 \end{cases} \quad (2)$$

Other neural network notate the step function as a special type of *output function* (we'll look at different output functions below). In this case the 'generic' output function is notated  $g(x)$  and the specific case of the step function  $g_{\text{step}}(x)$ .

#### □ Explore

1. Try other combinations and different sizes of input vectors, weight vectors.
2. Modify the example to have an adjustable threshold,  $\theta$ .
3. Pick a set of random input values and weights. Hand tweak the inputs to get the neuron to fire or not. Then, hand tweak the weights to achieve the same results.
4. ☼ Can you figure out other ways to represent this process in *Mathematica*?

**Example answers**

1. Try other combinations and different sizes of input vectors, weight vectors.

```

ψ = {1, 1, 1, 1, 0};
α = {1.2, -1.2, 0.3, -0.2, 1.3};
t = ψ α;
h = Plus @@ t;
Ψ = UnitStep[h]

1

```

2. Modify the example to have an adjustable threshold,  $\theta$ .

Here, the threshold is

```

ψ = {1, 0, 1, 1, 0};
α = {1.2, -1.2, 0.3, -0.2, 1.3};
t = ψ α;
h = Plus @@ t;
θ = 1.1;
Ψ = UnitStep[h - θ]

1

```

3. Pick a set of random input values and weights. Hand tweak the inputs to get the neuron to fire or not. Then, hand tweak the weights to achieve the same results.

Generate a random table of inputs

```

ψ = Table[Random[Integer, {0, 1}], {5}]

{0, 1, 0, 0, 1}

```

and weights

```

α = Table[Random[Real, {-4, 4}], {5}]

{-3.97306, -3.78723, 0.389626, -3.5145, -0.410137}

```

and carry out the process-

```

t = ψ α;
h = Plus @@ t;
Ψ = UnitStep[h]

0

```

It didn't fire. Tweak the input vector (turn off the two bad values and turn on a positive weighted one)

```

ψ = {0, 0, 1, 0, 0}

{0, 0, 1, 0, 0}

```

carry out the process —

```

t = ψ α;
h = Plus @@ t;
Ψ = UnitStep[h]

1

```

It fires!

Reset  $\psi$  to its original value, modify one member of  $\alpha$  and carry out the process —

```

ψ = {0, 1, 0, 0, 1};
α = {-3.9, 3.8, 0.4, -3.5, -0.41};

```

```

t = ψ α;
h = Plus @@ t;
Ψ = UnitStep[h]

```

1

4. ☼ Can you figure out other ways to represent this process in *Mathematica*?

```

ψ = {1, 0, 1, 1, 0};
α = {1.2, -1.2, 0.3, -0.2, 1.3};
θ = 0.5;

```

```
UnitStep[Apply[Plus, ψ α] - θ]
```

1

```
UnitStep[ψ.α - θ]
```

1

NB- To write a *Mathematica* function to do the thresholding for us we could say

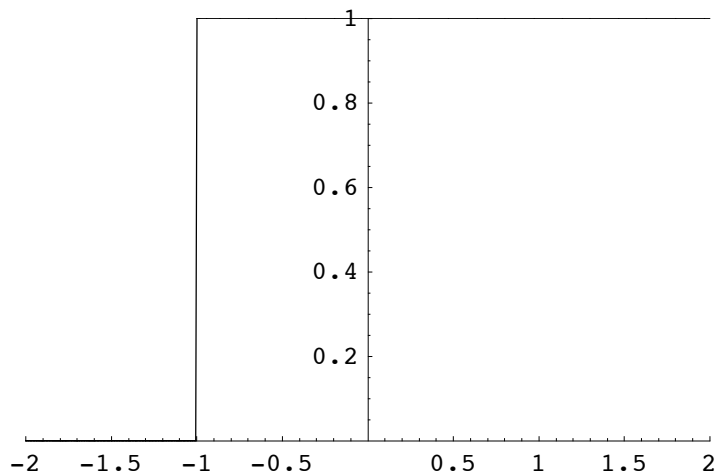
```
ourThreshold[h_, θ_] := UnitStep[h - θ]
```

This function takes two arguments, the value to threshold and the threshold value.

Evaluate the following and double-click on a graphic to see an animation of the function as  $\theta$  changes on the range  $[-1, 1]$ .

Table[

```
Plot[ourThreshold[x, θ], {x, -2, 2}, PlotRange -> {{-2, 2}, {0, 1}},
{θ, -1, 1, .2}];
```



#### □ webMathematica

These exercises are in file L2E3.js (link!)

#### ■ A Little Shortcut

Recall that, based on Equation 1, the net activation,  $h$  is

$$h = \sum \psi \alpha \quad (3)$$

There is a convenient mathematical shortcut for generating the net activation according to the above equation. The *dot-product* does exactly what the above equation specifies - it multiplies two vectors together and sums the resulting values. (Can you think of other mathematical / geometric applications where you've seen this process before? That is, multiplying things together and adding them all up?)

In notation, the dot product is denoted

$$\psi \cdot \alpha \quad (4)$$

The dot-product takes two vectors as an input and returns a single value (called a *scalar*)

$$\begin{aligned} \psi &= \{1, 0, 1\}; \\ \alpha &= \{.5, .23, -.32\}; \end{aligned}$$

$$h = \psi \cdot \alpha$$

$$0.18$$

#### □ Explore

1. Try other combinations and different sizes of input vectors, weight vectors.
2. What happens if the two input vectors have different lengths? Does this even make sense?
3. The dot product has an interesting geometric interpretation. Use *Mathematica* to explore these.

#### **Example answers**

1. Try other combinations and different sizes of input vectors, weight vectors.

$$\begin{aligned} \psi &= \{1, 1, 1\}; \\ \alpha &= \{.5, .23, -.32\}; \end{aligned}$$

$$h = \psi \cdot \alpha$$

$$0.41$$

$$\begin{aligned} \psi &= \{0, 1, 1\}; \\ \alpha &= \{.5, .23, -.32\}; \end{aligned}$$

$$h = \psi \cdot \alpha$$

$$-0.09$$

$$\begin{aligned} \psi &= \{0, 1, 1, 0, 1\}; \\ \alpha &= \{.5, .23, -.32, .42, -.22\}; \end{aligned}$$

$$h = \psi \cdot \alpha$$

$$-0.31$$

2. What happens if the two input vectors have different lengths? Does this even make sense?

```

ψ = {0, 1, 1};
α = {.5, .23, -.32, .42, -.22};

```

```
h = ψ.α
```

Dot::dotsh : Tensors {0, 1, 1} and {0.5, 0.23, -0.32, 0.42, -0.22} have incompatible shapes. [More...](#)

```
{0, 1, 1}.{0.5, 0.23, -0.32, 0.42, -0.22}
```

Of course there are unequal amounts of things to multiply together and eventually add. Therefore no.

3. The dot product has an interesting geometric interpretation. Use *Mathematica* to explore these.

to come.

#### □ webMathematica

These exercises are in file L2E4.js (link!)

## ■ The Output Function

In a neural network, both physical and artificial, the outputs of neurons (axons) serve as inputs for other neurons (synapses). You may recall that we decided for this experiment to constrain ourselves to an all-or-nothing network— either the neuron fired or it did not. We represented these two states numerically by 1 and 0 respectively. We have a net activation value of  $h = -3.1$  in this example but we need to convert it to one of the two states above, 0 or 1.

This transformation is achieved using an *output function* (also sometimes called a transfer function, or activation function). Notationally, we will generically use  $g(x)$  to indicate the output function.

#### □ Step Function

We've already looked at the simple unit step function,  $\theta(x)$  or  $g_{\text{step}}(x)$ , above.

As an equation the step function is simply —

$$g_{\text{step}}(x) = \theta(x) \tag{5}$$

In *Mathematica*, functions are denoted —

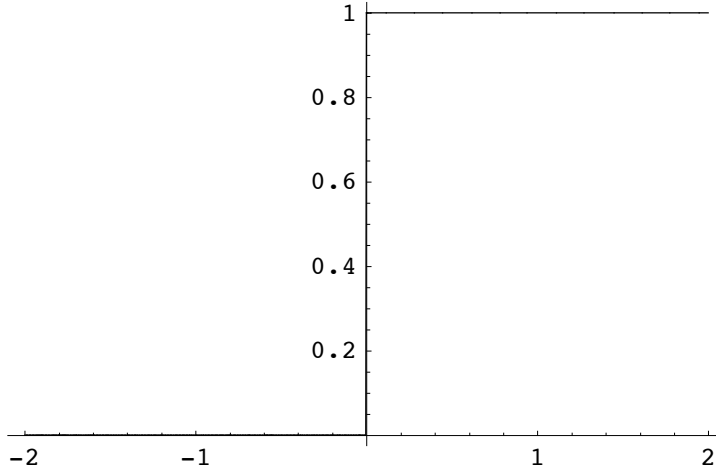
```
g_step[x_] := UnitStep[x]
```

We are using so-called Traditional Notation in *Mathematica*, so we can get away with a little prettier version—

```
g_step(x_) := θ(x)
```

Recall that the step function looks like this —

`Plot[gstep(x), {x, -2, 2}];`



### □ Linear Function

One frequently used activation function is the simple linear transfer —

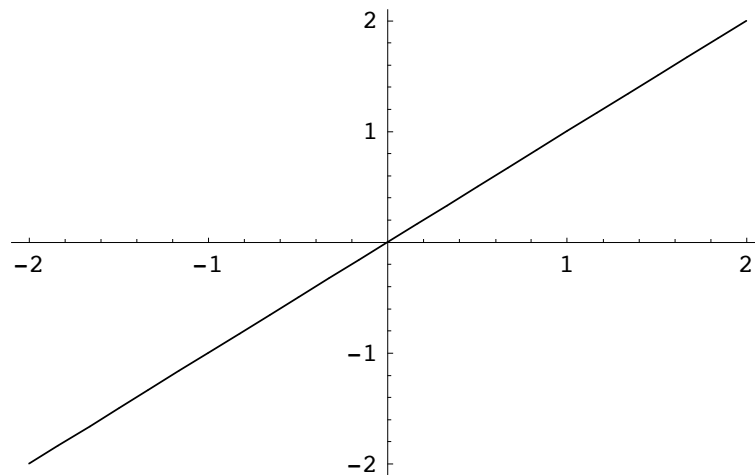
$$g_{\text{lin}}(x) = x \quad (6)$$

In *Mathematica* —

`glin(x_) := x`

and graphically —

`Plot[glin(x), {x, -2, 2}];`



Notice that this output function simply replicates its input. In this case notice that we've strayed from our original biologically-correct version of all-or-nothing activation. Still, these sorts of activation functions will have further utility down-the-road.

### □ Theta Function

Confusing the situation a little bit, there is a sometimes used function called  $g_{\theta}(x)$  that *isn't* just a unit step, instead it is a modification of the unit step as so —

$$g_{\theta}(x) = x \theta(x) \quad (7)$$

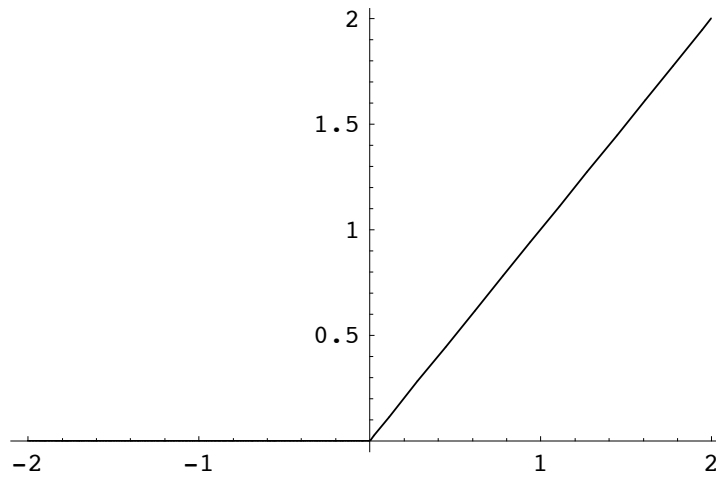
The result is a linear function, but only when  $x > 0$ .

In *Mathematica* —

$$g_{\theta}(x) := x \theta(x)$$

and graphically —

`Plot[gθ(x), {x, -2, 2}];`



#### □ Sigmoid Function

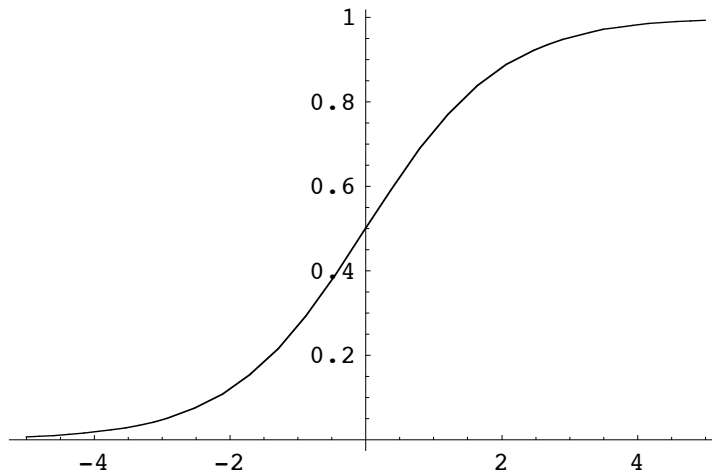
The sigmoid function is a commonly used activation function in artificial neural networks. Mathematically it is —

$$g_{\text{sig}}(x) = \frac{1}{1 + e^{-x}} \quad (8)$$

$$g_{\text{sig}}(x) := \frac{1}{1 + e^{-x}};$$

and graphically —

`Plot[gsig(x), {x, -5, 5}];`



### □ Gaussian Function

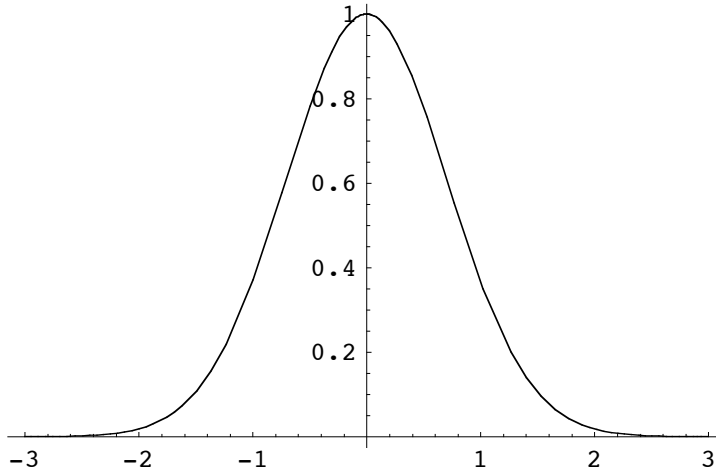
Another statistically interesting function is the Gaussian activation function —

$$g_{\text{gauss}}(x) = e^{-x^2} \quad (9)$$

$$g_{\text{gauss}}(x) := e^{-x^2}$$

Graphically —

`Plot[ggauss(x), {x, -3, 3}];`



### □ Explore

1. Speculate on the use of each of these functions. What are some of them good at? What are they bad at?
2. Speculate on the statistical nature of the last two activation functions (the Sigmoid and Gaussian functions). Why would we want such a thing?
3. Take one of the full example from above and apply the different output functions to it (instead of UnitStep[]) does it change the resulting activation value? If so how?

#### Example answers

1. Speculate on the use of each of these functions. What are some of them good at? What are they bad at?

to come

2. Speculate on the statistical nature of the last two activation functions (the Sigmoid and Gaussian functions). Why would we want such a thing?

to come

1. Take one of the full example from above and apply the different output functions to it (instead of UnitStep[]) does it change the resulting activation value? If so how?

$$\psi = \{0, 1, 0, 0, 1\};$$

$$\alpha = \{-3.9, 3.8, 0.4, -3.5, -0.41\};$$

$$t = \psi \alpha;$$

$$h = \text{Plus} @@ t;$$

$$\Psi = \text{UnitStep}[h]$$

$$\Psi = g_{\text{lin}}(h)$$

3.39

$$\Psi = g_{\text{step}}(h)$$

1

$$\Psi = g_{\theta}(h)$$

3.39

$$\Psi = g_{\text{sig}}(h)$$

0.967391

$$\Psi = g_{\text{gauss}}(h)$$

0.0000102104

#### □ webMathematica

These exercises are in file L2E5.js (link!)

#### ■ References

- [1] McCulloch, WS, Pitts, W. 1943, '[A logical calculus of the ideas immanent in nervous activity](#)', *Bulletin of Mathematical Biophysics*, vol. 5, pp 115–133.
- [2] Rosenblatt, F. 1959, 'Two theorems of statistical separability in the perceptron', *Proceedings of a Symposium on the Mechanization of Thought Processes*, Her Majesty's Stationary Office, London, pp. 421–456.
- [3] Rosenblatt, F. 1962, *Principles of Neurodynamics*, Spartan Books, New York.

#### ■ Revision History

1. October 2003 - Initial ramblings version
2. December 2003 - First release to CSAC
1. August 2004 - Second release to CSAC