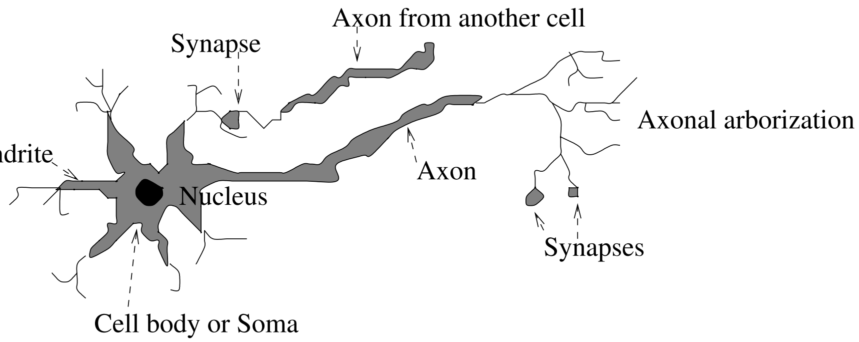
Neural Networks

We're now going to look at another way of building classifiers: (artificial) neural networks (ANNs).

The brain comprises around 10^{11} neurons, connected into a complex network. ANNs similarly comprise numerous simple computational devices (here called *Threshold Logic Units* or *TLUs*), connected into complex networks.

Among the advantages of neural networks are: they are very good at handling numerical inputs; and they can be trained: they can learn functions from examples. They have been successful in learning to recognise handwritten characters, spoken words and human faces. Their advantages make them useful for a wide variety of other tasks, in addition to classification.

1 Neurons

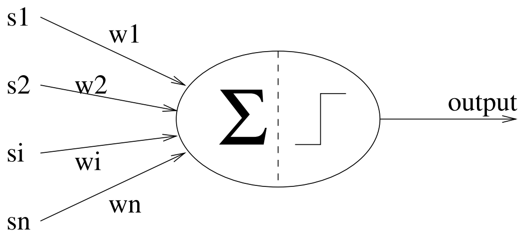


Sufficient electrical activity on a neuron's dendrites causes an electrical pulse to be sent down the axon, where it may activate other neurons.

Before looking at ANNs in general, we look at TLUs, which are computational units inspired by neurons.

2 Threshold Logic Units (TLUs)

A *threshold logic unit (TLU)* takes in some numerical inputs, computes a weighted sum of the inputs, compares the sum to a threshold value (θ), and outputs a 1 if the threshold is equalled or exceeded; otherwise, it outputs a 0.



$$\text{in} =_{\text{def}} \sum_{i=1}^{n} w_i s_i$$

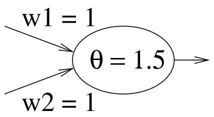
$$\text{output} =_{\text{def}} g(\text{in})$$

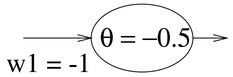
where, for the moment,

$$g(x) =_{\text{def}} \begin{cases} 1 & \text{if } x \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

The inputs $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$ and weights $\mathbf{w} = \langle w_1, \ldots, w_n \rangle$ are typically real values (positive or negative). *in* is a weighted sum of these inputs. g is called the *activation function*: it decides whether the weighted sum of the inputs is big enough for there to be an output value of 1.

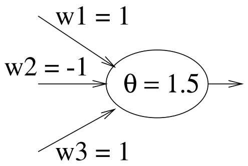
As well as having the flexibility of handling real-number inputs, if we allow inputs of only 0 (false) or 1 (true), TLUs can simulate many Boolean operators and expressions. Here are TLUs that compute $s_1 \wedge s_2$ and $\neg s_1$:





(Note that these are not the only weights and thresholds that could be used to simulate these operations.)

Here's a TLU that computes a more complex Boolean expression $s_1 \wedge \neg s_2 \wedge s_3$



3 What can TLUs represent?

We've seen a few example of functions that a single TLU can encode. But there are limitations on what functions TLUs can encode. They cannot even encode all Boolean operators.

For example, exclusive-or ($\oplus$) cannot be computed by a single TLU.

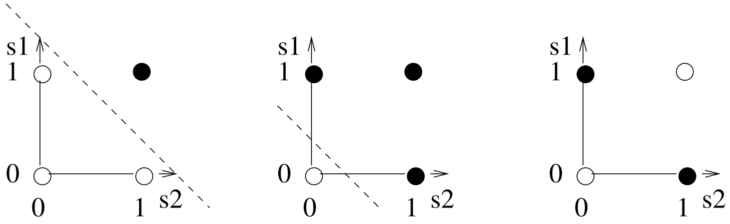
s_1	s_2	output
0	0	0
0	1	1
1	0	1
1	1	0

We would need weights w_1 and w_2 and threshold θ such that

$$\begin{array}{rcl} w_1 \times 0 + w_2 \times 0 & < & \theta \\ w_1 \times 0 + w_2 \times 1 & \geq & \theta \\ w_1 \times 1 + w_2 \times 0 & \geq & \theta \\ w_1 \times 1 + w_2 \times 1 & < & \theta \end{array}$$

There are no such weights and threshold.

A TLU can only encode *linearly separable functions*. Exclusive-or is not a linearly separable function. We can see what this means from these graphs:



s1 AND s2   s1 OR s2   s1 XOR s2

We have two inputs, which can be 0 or 1, drawn on the axes. Black dots indicate points in the input space where the output should be 1. But a TLU outputs 1 iff some threshold is exceeded, so this space is divided into two (those points where the weighted sum is above the threshold and those points where it isn't). So, in the cases where we can draw a straight line that separates the white dots from the black dots, we can use a TLU.

Class Exercise. *Give another Boolean operator that is not linearly separable.*

In general, where we have more than 2-dimensions (more than 2 inputs), the input space is divided into two by a hyperplane rather than by a straight line. But this doesn't change the fundamental point that we can only represent functions where such a dividing plane does exist (linearly separable functions).

Unfortunately, there are not many linearly separable functions. So single TLUs may not be of much use to us. As we'll see, *networks* of TLUs can overcome these limitations.

4 TLUs for Classification

You might already be objecting to the use of TLUs for implementing classifiers. Apart from the limitation to linearly separable functions, there seem to be two other limitations:

Only two classes? The output of a single TLU is either 0 or 1. So a single TLU is only capable of choosing between two classes. While two classes is very common (spam/ham, employee/visitor, etc.), there are applications where we will have more than two classes (beginner/intermediate/advanced).

This is a problem we can overcome by using several TLUs. We'll postpone further discussion of use of more than one TLU to the next lecture.

Only numeric inputs? The inputs to TLUs ($s_1, \ldots, s_n$) are numbers. How can we handle attributes whose types aren't numbers?

We've already seen how to handle Boolean-valued inputs: simply use 1 for true and 0 for false.

What about symbolic-valued attributes? If there are only two values (as with *sex*), then we can use 0 and 1 again.

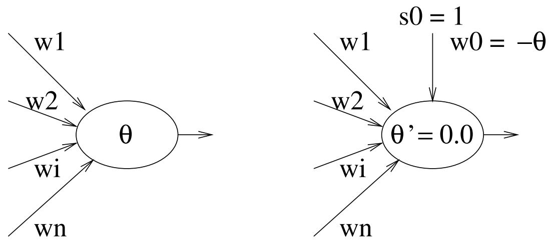
If there are more than two values (e.g. *none*, *snack*, *full*), we use more than one binary input. The simplest approach is to use one binary input per value. For example, to handle the three values of the *meal* attribute, we would use three inputs s_1, s_2 and s_3. To input an instance whose value for this attribute is *none*, $s_1 = 1, s_2 = 0, s_3 = 0$. For *snack*, $s_1 = 0, s_2 = 1, s_3 = 0$. For *full*, $s_1 = 0, s_2 = 0, s_3 = 1$. If there are k values, this scheme requires k inputs.

You will appreciate, however, that, were we to encode the values as consecutive binary numbers, only $\lceil \log_2 k \rceil$ would actually be needed. For example, for three values, two inputs gives four distinct bit patterns (one of which would never be used), whereas using three inputs as proposed above, gives eight distinct bit patterns (five of which would never be used). Although our scheme is not economical, it probably makes it easier to find weights.

5 A Simplification

This simplification makes the implementation simpler. But its real motivation is that it makes the learning algorithm that is coming up later simpler.

Suppose we have a TLU whose threshold is θ. The simplification is to construct a new but equivalent TLU whose threshold θ' is fixed at zero. But then we add an extra input line, s_0, to the new TLU. The input value on this line will be fixed at 1, and the weight on this line will be fixed at $-\theta$.



These give equivalent results.

Exercises

1. Design TLUs for
   (a) $s_1 \vee s_2$ and
   (b) $s_1 \Rightarrow s_2$.

2. A literal is any Boolean variable (such as s_1 in the above examples) or its negation (e.g. $\neg s_1$). A TLU can compute the value of any conjunction of literals, e.g. $s_1 \wedge \neg s_2 \wedge s_3$ (as earlier), or $\neg s_1 \wedge \neg s_2 \wedge s_3 \wedge s_4$, or $s_1 \wedge s_2 \wedge s_3 \wedge s_4 \wedge s_5$, etc.

   In general, what weights and threshold would you need if you were designing a TLU for some conjunction of literals? (Hint: See if you can come up with the answer by generalising from the TLU that appeared earlier in the notes for computing $s_1 \wedge \neg s_2 \wedge s_3$.)

3. Similarly, TLUs can compute the value of any disjunction of literals (i.e. using $\vee$). In general, what weights and thresholds would you need?

4. Design a TLU that computes a majority function. In other words, it outputs 1 only if half or more than half of its n inputs are 1.

5. Similarly design a TLU that computes a minority function: it outputs 1 only if half or fewer than half of its inputs are 1.