Neural Networks

1 Introduction

We're still studying reactive agents. But we're now going to look at another way of implementing the action function that maps from percepts to actions: (artificial) neural networks (ANNs).

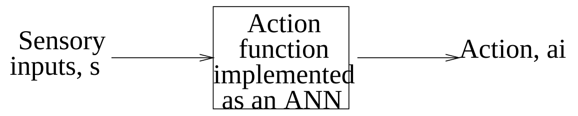
The brain comprises around 10^{11} neurons, connected into a complex network. ANNs similarly comprise numerous simple computational devices (here called Threshold Logic Units or TLUs), connected into complex networks.

The advantages of using ANNs include:

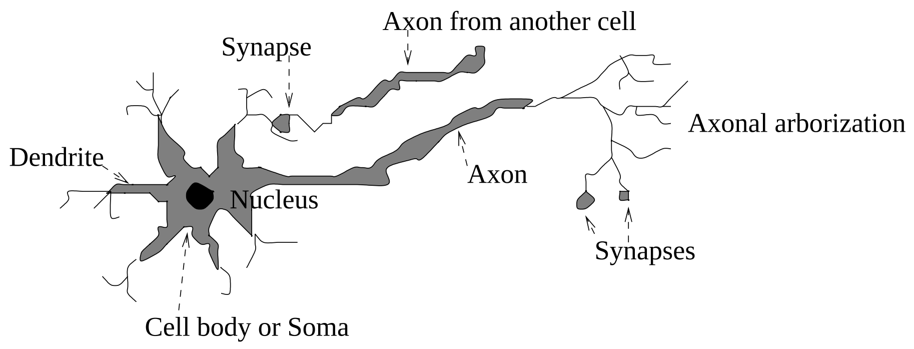
- They can handle real-valued inputs and outputs;
- They can handle noisy inputs;
- They can be trained: they learn functions from example data.

These advantages may make them useful in all sorts of situations in AI where we need to represent functions, not just for the implementation of the action function of a reactive agent. For example, they have been successful in learning to recognise handwritten characters, spoken words and human faces.

Their ability to handle real-valued noisy inputs (the kinds of values that sensors deliver) means that we may not even need a 'perceptual preprocessing' phase (or, at the least, we can get by with a much reduced preprocessing phase):



2 Neurons

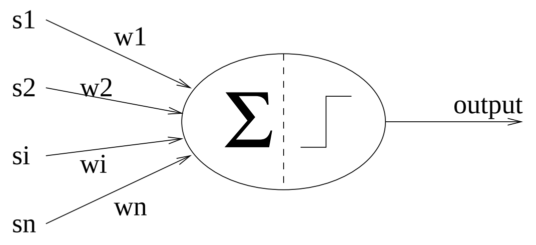


Sufficient electrical activity on a neuron's dendrites causes an electrical pulse to be sent down the axon, where it may activate other neurons.

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

3 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.



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

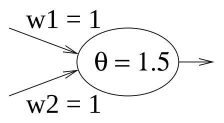
$$\text{output} \triangleq g(\text{in})$$

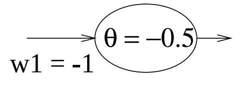
where, for the moment,

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

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.

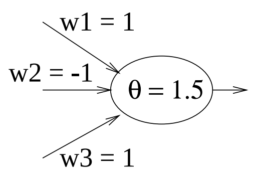
As well as having the flexibility of handling real-number inputs, if we allow inputs of only 0 (**false**) or 1 (**true**), TLUs can simulate many Boolean operators and expressions. Here are TLUs that compute $s_1 \wedge s_2$ and $\neg s_1$:





(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 $s_1 \wedge \neg s_2 \wedge s_3$



(You might already be objecting to the use of TLUs for implementing the action functions of agents. The output of a single TLU is either 0 or 1. So a single TLU is only capable of choosing between two actions, a_0 and a_1. But, in general, agents have many more than two actions. This is a problem we can overcome by using several TLUs. If we use l TLUs, each outputting 0 or 1, then we get 2^l different outputs.

We'll postpone further discussion of use of more than one TLU to the next lecture. For now, note that the issues I'm about to discuss in the next section apply, not just to single TLUs, but also to using multiple TLUs in the way we have just discussed. Only if we build networks of TLUs in which the outputs of some TLUs are fed in as inputs to other TLUs can we start to overcome the limitations described in the next section.)

4 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.

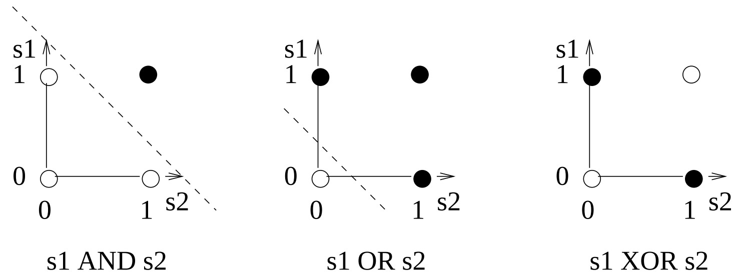
s_1	s_2	output
0	0	0
0	1	1
1	0	1
1	1	0

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:



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.

Question. *Can you come up with another example of a 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).

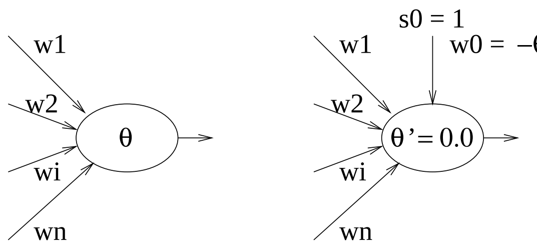
Unfortunately, there are not many linearly separable functions. So single TLUs may not be of much use to us. As we'll see, *networks* of TLUs can overcome these limitations. Before we look at networks, we'll look at a Java implementation of TLUs.

5 A TLU Implementation

TLUs are simple circuits, and therefore one implementation would be to construct them physically, e.g. from silicon. This is perhaps a long-term goal of some ANN researchers. But, for the moment, most TLUs and ANNs are simulated in software. We'll see a Java implementation.

We'll make a simplification first. This simplification makes the implementation simpler. But its real motivation is that it makes the learning algorithm that is coming up later easier too.

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.

Here's some Java for this simplified TLU:

```
public class StandaloneTLU
{
   public StandaloneTLU(double theThreshold, int theNumOfInputs,
      double[] theWeights)
   {  /* The threshold is converted to an extra weighted input.
       */
      numOfInputs = theNumOfInputs + 1;
      weights = new double[numOfInputs];
      weights[0] = - theThreshold;
      for (int i = 1; i < numOfInputs; i++)
      {  weights[i] = theWeights[i - 1];
      }
   }
   public double activate(double[] s)
   {  double in = 0.0;
      in += weights[0] * 1.0; // the extra input for the threshold line
      for (int i = 1; i < numOfInputs; i++)
      {  in += weights[i] * s[i - 1];
      }
      return g(in);
   }

   public double g(double x)
   {  return x >= 0 ? 1 : 0;
   }

   private int numOfInputs;
   private double[] weights;
}
```

The code will be explained in the lectures.

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.