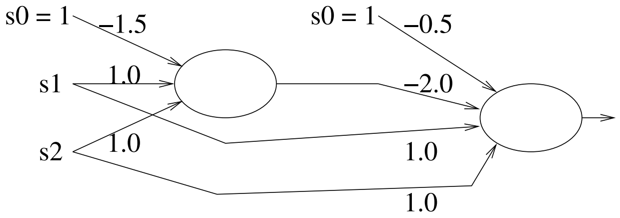
Multilayer Neural Networks

1 Fully connected, layered, feedforward networks

Networks of TLUs can encode more functions than can single TLUs.

A single TLU cannot encode the exclusive-or function. But two TLUs connected together, the output of the first feeding into the second, can do this.

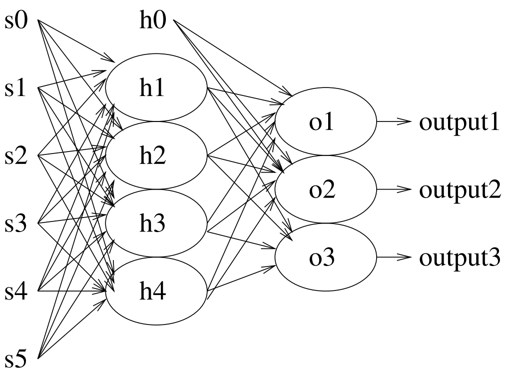


s1 XOR s2

What functions can neural nets compute in general? The answer is: if we place no restrictions on the network architecture, we can represent *any* function. In particular, if we allow *feedback* (where the output of one TLU is fed into that of an 'earlier' TLU), then ANNs have full computational power: they can compute all Turing-computable functions.

The full power is often not needed and it's not always desirable. It can be better to place some restrictions on the network architecture. (It can simplify the learning algorithms, for example.)

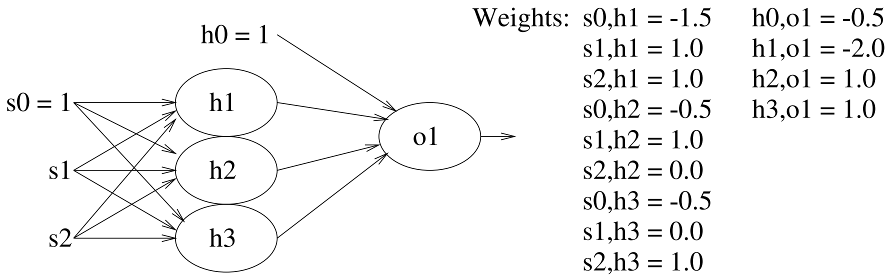
We will look at *fully connected, layered, feedforward networks* (for which there's a reasonably good learning algorithm). Here is an example of such a network:



- *Feedfoward* versus *recurrent* networks
  - In feedforward networks, links are unidirectional and there are no cycles. (Technically, they're DAGs.)
  - Recurrent networks have arbitrary topology. In particular, feedback loops are allowed.
- Layered feedforward networks
  - The units are arranged in a number of layers. A unit will send activation only to units in the next layer.
  - On the left, we see the *input units/input layer*. These units simply pass activation from the environment to the next layer.
  - On the right, we see the *output units/output layer*. The activation that these produce is taken as the output of the network.
  - In between are the *hidden units*, arranged in *hidden layers*.
  - The example is a two-layer network. The input layer isn't counted. (However, some people do not know this convention, and they would call this a three-layer network.)
- Fully connected, layered, feedforward networks
  - Each unit in a layer is connected to every unit in the next layer.

Our description henceforth will be kept simpler by assuming there is only one hidden layer. In general, there could be several.

Here, by way of an example is a fully-connected, layered, feedforward network for computing exclusive-or:



s1 XOR s2

Note that now that we have multiple output units, we can handle classification tasks in which there are more than two classes. If there are two clases, $|L| = 2$, we need only one output unit. if there are $|L| > 2$ classes, then we would use $|L|$ output units.

2 Training a Neural Network

How on earth would someone come up with an ANN? Consider all the decisions they have to make: the number of layers; the number of units in each layer; and the weights. (If we assume we've used the trick from the previous lecture, then we don't need to come up with thresholds.)

Quite simply, 'programming' these networks is in general impossible. However, there is an alternative: learning. We can present to an ANN a set of examples of function inputs and outputs (in our case, already-classified instances) and get it to learn the weights so that the ANN implements the function (or, at least, one quite close to it).

We will still have to decide on the number of hidden layers and the number of units in each layer (the 'architecture' or 'topology' of the network). And this remains something of an 'art'. But the automatic learning algorithm can take care of the weights. We can start with an ANN in which there is a random assignment of weights (usually in the range $[-0.5, 0.5]$), and then put the network through a period of training during which the weights will be adjusted.

During the training phase, we will present to the learning algorithm a set of example input vectors and their corresponding correct output vectors (i.e. the outputs we want the ANN to produce for these inputs). The learning algorithm adjusts the weights in the ANN in those cases when the ANN isn't currently producing the correct target output.

We follow common practice by first presenting a learning algorithm for a single TLU. Then we explain how to extend the ideas to apply to a fully connected, layered, feedforward ANN.

3 Training a TLU

As usual, we need a dataset of examples: each example consists of an input vector and the corresponding target output (which, since we have reverted to dealing with TLUs, is simply 0 or 1). The essence of the learning is this:

- If the actual TLU output is 1 when it should be 0, make each w_i smaller by an amount proportional to s_i.
- If the actual TLU output is 0 when it should be 1, make each w_i larger by a similar amount.

We must present the dataset of examples to the learning algorithm several times, each time being referred to as an *epoch*.

Class exercise. Why?

How many epochs are needed? Well, you can stop the algorithm when the TLU *converges*: it correctly predicts all the examples (or most of them) and the weights are not being changed any more. Alternatively, we might use a fixed number of epochs, or we might set a time limit on the learning.

How do we update the weights? First, we compute the error:

$$\text{Err} =_{\text{def}} \text{target} - \text{output}$$

Given that target and output can only be 0 or 1, the error can only be 0, 1 or -1. Second, we update each weight:

$$w_i := w_i + \alpha \times s_i \times \text{Err}$$

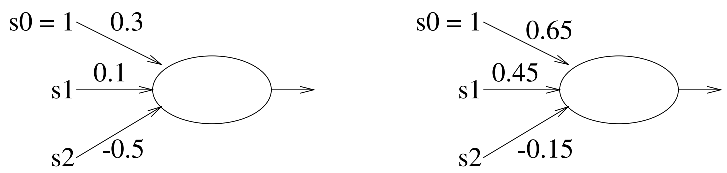
α is a constant called the *learning rate* and will be a real value, $0 < \alpha \leq 1$. (A typical value is around 0.35.)

By way of an example, imagine we're trying to train a TLU to compute $s_1 \wedge s_2$. Suppose the random weights assigned at the start of the algorithm are as shown in the left-hand diagram below. Suppose one example is as follows. The example inputs are $\langle 1, 1\rangle$. The target output is, of course, 1. The actual output from the TLU shown is 0. (Why? Because $1 \times 0.3 + 1 \times 0.1 + 1 \times -0.5 = -0.1$ and $-0.1 < 0$.)

So, the error is $1 - 0 = 1$. If we use a learning rate of 0.35, then we update the weights as follows:

$$\begin{aligned} w_{s_0} &:= 0.3 + 0.35 * 1 * 1 = 0.65 \\ w_{s_1} &:= 0.1 + 0.35 * 1 * 1 = 0.45 \\ w_{s_2} &:= -0.5 + 0.35 * 1 * 1 = -0.15 \end{aligned}$$

After this one example, the TLU is now as per the right-hand diagram.



We then go on to other examples from the training set and, once these are exhausted, we go through the examples all over again for the second epoch.

By way of summary, here's the training procedure in pseudocode:

Algorithm: TRAIN(*TLU*, *dataset*)

initialise *TLU* with random weights;
do
{ **for each** example $e \in$ *dataset*
  { UPDATE(*TLU*, e);
  }
}
while *TLU* hasn't converged

Algorithm: UPDATE(*TLU*, e)

inputVector := e's input vector;
targetOutput := e's output;
actualOutput := ACTIVATE(*TLU*, *inputVector*);
err := *targetOutput* − *actualOutput*;
for each of *TLU*'s weights w_i
{ $w_i := w_i + \alpha \times s_i \times$ *err*;
}

4 What Can a TLU Learn?

We saw that we can design TLUs to compute only certain functions. There's obviously no point trying to get a TLU to learn exclusive-or ($\oplus$) or other functions that it cannot compute. But what is the relationship between the set of functions a TLU can compute and the set we can get it to learn?

> *Whatever functions a TLU can compute, it can learn to compute.*

Of course, for it to learn a specific function requires that it be given a representative enough set of examples. It also requires that the learning rate, α, not be too large, otherwise a suitable set of weights can be 'overshot' (But if α is too small, learning can be very slow.)

How can we detect whether the TLU that we've been training has actually learned the function that we want it to? In the case of TLUs, we might just be able to do it by inspecting the weights. But, in general, following the training phase, we subject the TLU to a testing phase to find out what it has learned, as discussed in a previous lecture.