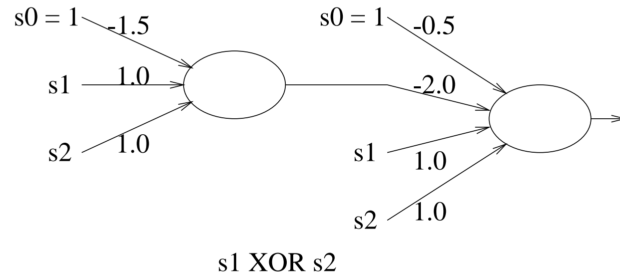
# Multilayer Neural Networks

## 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.

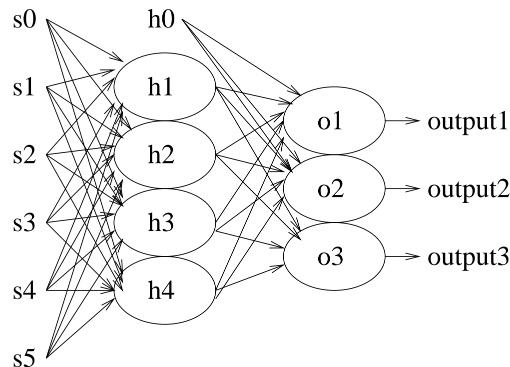



s1 XOR s2

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:

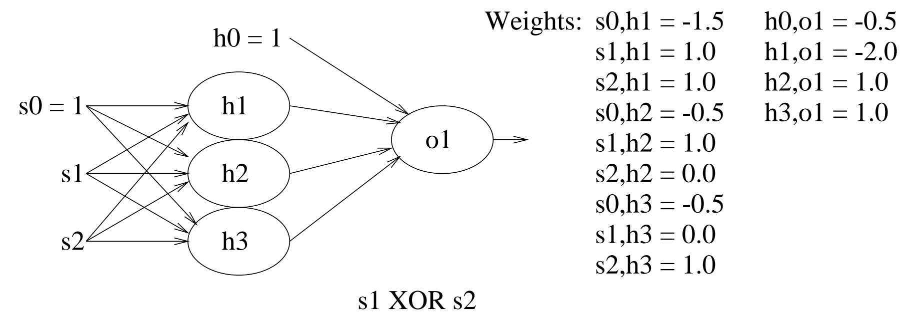





- *Feedfoward* versus *recurrent* networks
  - In feedforward networks, links are unidirectional and there are no cycles. (Technically, they're DAGs.)
  - Recurrent networks have arbitrary topology. In particular, feedback loops are allowed.
- Layered feedforward networks
  - The units are arranged in 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. (They are our sensors.)
  - On the right, we see the *output units*/*output layer*. The activation that these produce is taken as the output of the network (e.g. to choose an action for a reactive agent).
  - In between are the *hidden units*, arranged in *hidden layers*.
  - The example is a two-layer network. The input layer (the sensors) aren'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

## 2 Implementing ANNs

We saw the code for a single TLU (`StandaloneTLU.java`). This included a method `activate(double[] s)` for taking inputs from sensors and working out the output of the TLU. Perhaps the only significant change that we need when we come to implement networks of TLUs is to allow TLUs to take their input from the output of other TLUs, as well as from the sensor. Here's some code in which a TLU in the output layer takes its input from the output of TLUs in the hidden layer:

```
public double activate()
{  double in = 0.0;
   for (int i = 0; i < numOfInputs; i++)
   {  inputs[i] = hiddenUnits[i].getOutput();
      in += weights[i] * inputs[i];
   }
   output = g(in);
   return output;
}
```

# 3 Supervised Learning of ANNs

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 learning algorithm a set of examples of function inputs and outputs and get it to learn an ANN that 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.

Any form of learning where the inputs and also their corresponding target outputs can be perceived by the learner is referred to as *supervised learning*. Typically, the target outputs will come from some 'teacher' (but this could be any other agent or even the environment itself). This is in contrast to other forms of learning where the target outputs are not known or not available.

Of course, we don't really want the 'teacher' to have to supply an absolutely comprehensive set of examples (*every* possible input and its target output). We want it to be the case that the learner will be able to *generalise* (to give good outputs on previously unseen data) if given a sufficiently representative set of training examples.

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.

# 4 Learning in TLUs

So we present the learning algorithm with a series 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 algorithm 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 examples to the learning algorithm several times, each time being referred to as an *epoch*. 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:

$$\text{Err} \triangleq \text{target} - \text{output}$$

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:

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

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

If `tlu` is a variable that contains a TLU with randomly-assigned weights, `numOfInputs` is the number of input lines coming into this TLU (including the extra one that was used to get rid of the threshold), and `examples` is a set of examples, then the learning algorithm is:

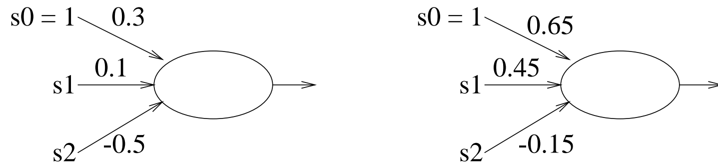
```
for (each epoch, e)
{  /* Within each epoch, we deal with each example in turn.
    */
   for (each ex in the set of examples)
   {  exampleInput = ex.getInput();
      targetOutput = ex.getTargetOutput();
      /* Compute the actual output we get for this example input.
       */
      output = tlu.activate(exampleInput);
      /* Compute error between target and actual output.
       */
      err = targetOutput - output;
      /* Adjust each incoming weight.
       */
      for (int i = 0; i < numOfInputs; i++)
      {  tlu.setWeight(i,
            tlu.getWeight(i) + theLearningRate * s[i] * err;
      }
   }
}
```

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.

## 5 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, $\alpha$, not be too large, otherwise a suitable set of weights can be 'overshot' (But if $\alpha$ 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. We'll discuss this further after we've looked at learning in fully connected, layered, feedforward ANNs.

## Exercise (Past-Exam Question)

This question is about TLUs and neural networks. Throughout, assume that the *activation function* of the TLUs, $g$, is defined as follows:

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

where $\theta$ is the threshold of the TLU.

1. Design a TLU which has two inputs $s_1$ and $s_2$. The inputs can take values of 0 or 1. The TLU should compute $s_1 \downarrow s_2$, where $\downarrow$ is the NOR operator, i.e. $s_1 \downarrow s_2 \equiv \neg(s_1 \vee s_2)$.
2. Your answer to part 1 is to be converted into a TLU that has a threshold of 0 but it has an extra input $s_0$. At what value will you fix the input $s_0$ and what weight will you use on $s_0$'s input wire to continue to compute $s_1 \downarrow s_2$?
3. The kind of conversion that you carried out in part 2 is a useful precursor to training a neural net using a learning algorithm. Why is it useful?
4. Below we have tabulated eight Boolean-valued functions:

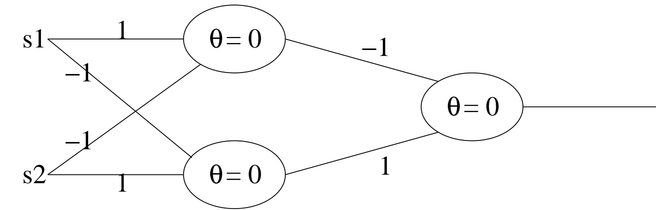


| $s_1$ | $s_2$ | $s_1 \boxtimes s_2$ | $s_1 \boxdot s_2$ | $s_1 \boxplus s_2$ | $s_1 \boxminus s_2$ | $s_1 \otimes s_2$ | $s_1 \odot s_2$ | $s_1 \oplus s_2$ | $s_1 \ominus s_2$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Suppose you intend to design eight TLUs, each having two inputs, $s_1$ and $s_2$, that can take values of only 0 or 1.

For how many of the eight functions can such TLUs be designed? Explain your answer convincingly and in detail. (There is no need to give any actual TLUs.)

5. Here is a fully connected, layered, feedforward neural network:




What will the output of this network be when the values of its inputs, $s_1$ and $s_2$, are both 1. Show your working.

6. Design a fully connected, layered, feedforward neural network for the following. There are four inputs, each of which can take a value of 0 or 1. When any two of the inputs are 0 and the other two inputs are 1, the output of the network is 1. Otherwise the output is 0.