Pyceptron exercises

Olivia Guest

April 22, 2014

1 Teaching the network logic

To explore some of the very basic uses of neural networks, we will first attempt to teach our perceptron logical operators. Logical operators were touched on in the first part of the workshop, when we talked about **not**, **and**, **or**, etc., and applied them to truth values (e.g., true and false).

Even though logical operators might seem very abstract, saying things like "true **and** true is true" is just a formalised way of saying much more common expressions such as "I agree **and** you agree, therefore we both agree". Natural language is full of such logical operations, for example when we ask questions, e.g., "do you want sugar **or** milk in your tea?" is equivalent to "sugar **or** milk" in formal logic. Such formalisation is useful because it is an easy way to avoid ambiguity which is required when programming.

1.1 Not

The first logical operator we will teach our perceptron is the **not** operation. This requires one input unit and one output unit because negation (a synonym for not) is applied to one variable (or in our case input unit) at a time. So when we give the network a 0 we want it to return a 1, and when we give the network a 1 we need a 0 on the output. That means our training patterns are 0 and 1, and our targets are 1 and 0, in that order. Table 1 represents our patterns and their required targets.

Input	Output
0	1
1	0

Table 1: Truth table for logical **not**.

Once you have made the required changes, run the network to see what happens and how long it takes to train.

Can the network learn **not**? Yes / No

1.2 And

Now let's teach it something a little more complex, the most basic form of logical **and** is that with two inputs (e.g., false **and** true is false) and it always has one

output. Meaning that the network needs to learn the input-output mapping given in Table 2. By looking at Table 2, you might notice that now there are three units, two input units and the same number of output units as before. So we need to modify the network to have two input units, leaving the single output unit untouched. Thankfully, the code can figure out the number of input and output units required based on the patterns and targets you give it. This is something I wrote myself, if you want to see how that is done see, but do not get bogged down with understanding this Once the patterns and targets are set, run the network like before and see what happens.

Can the network learn **and**? Yes / No

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Table 2: Truth table for logical **and**.

1.3 Or

Now let's do the same for **or**. Logical **or** returns a 1, or true, if either input is on. In other words, it only returns 0 when neither are 1, as shown in Table 3. Remember that at any point you can check what any logical operator does directly in the Python shell, e.g., by typing `False or True`, Python will say `True` (corresponding to the 3rd line of Table 3).

Can the network learn **or**? Yes / No

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Table 3: Truth table for logical **or**.

1.4 Xor

Things start to become interesting when we teach the network **xor**, also known as exclusive or. Like its namesake **or**, **xor** returns true when either one or the other inputs are true, *but it does not* return true when both are true (compare Table 4 with Table 3).

In Python one way of specifying **xor** is by using `(a and not b) or (b and not a)`. You do not need to know this to program the network, because the network figures out how to map inputs onto outputs itself, or at least tries to...

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 4: Truth table for logical **xor**.

Can the network learn **xor**? Yes / No