Pyceptron exercises

Olivia Guest

October 14, 2015

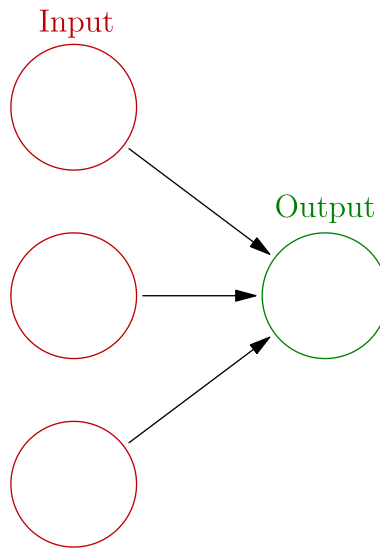## 1 Running a perceptron on paper

Figure 1: A perceptron that you can run on paper.

Using the activation propagation Equation 1 as presented and applied in the handouts calculate the output of the perceptron in Figure 1:

$$y = f\left(\sum_{j=1}^{N} x_j \times w_{ji}\right) \tag{1}$$

Firstly, because this is just a practice run to help you understand how activations propagate randomly, choose some values for the weights and inputs (simple small numbers will make the arithmetic easier). Then, using the three inputs and the three connection weights you just picked, calculate the value of the single output unit. If you wish to calculate the so-called post-synaptic activation for the output unit you can use an activation function. A simple version of this function, denoted by f, is to change the output unit's value to 1.0 if the total of all inputs to it is above 0, otherwise set it to 0. In other words:

$$y = \begin{cases} 1 & \text{if } \sum_{j=1}^{N} x_j \times w_{ji} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

What does f help the network do?

2 Playing around with the network

Now you will have a go playing around, but hopefully not breaking, the network depicted in Figure 2. However, don't worry if you do break it, there is an infinite supply.
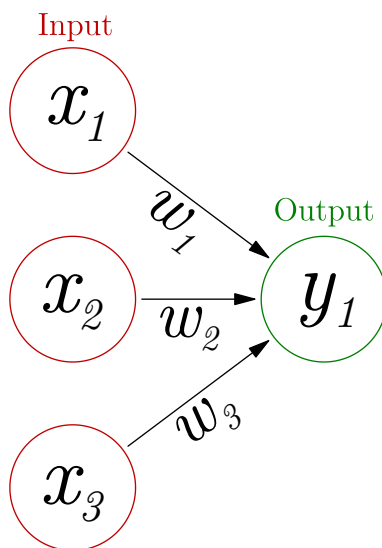

Figure 2: A perceptron

Have a look at the code that makes it learn. Open the file called `pyceptron.py` and go to the part that says `def Train(self):`. This function, as I probably already explained, does a bunch of things that essentially implement the two basic equations. The first equation is that for calculating the activation of the output unit(s):

$$y_j = f\left(\sum_1^N w_i \times x_i\right) \tag{3}$$

where y_j is the (output) unit whose state you want to calculate, N is the number of units on the previous layer, w_i is the weight on the connection between i and j, and f is a function that the unit applies. In the code you can write that in many ways, just like you can say the same or similar things in natural language in a number of different ways. I decided on this:

```
for i in range(N+1):
        y += x[i] * w[i]

y = f(y)
```

and hopefully you can guess what each variable is based on the equation. Find this snippet of code within the `Train` function. Using your own judgement, add some comments to explain what is going on, for example:

```
for i in range(N+1):
# loop through all the indices of input units
        y += x[i] * w[i]
        # sum the product of each input unit x[i]
        # and weight w[i] and save it in output
        # unit y

y = f(y)
# calculate the post-synaptic value for y
```

Ask for help if something is not making sense straight away.

Can you find where `f` is defined?

The second, and equally important equation, you need is the one that trains the weights:

$$\Delta w_i = \eta(d_j - y_j) \times x_i \tag{4}$$

where d is what you want y to be given x, and η is the learning rate. In other words d is the target, y is as before the output unit (or units), and x are the unit units. As before, there are a number of different ways to say this in Python. Hopefully, my choice is a clear translation:

```
error = d[p][0] - y

for i in range(N+1):
        w[i] += h * error * x[i]
```

Now that you have a slightly clearer idea (if not — ask me or a demonstrator for clarification) of how the network works, have a go changing things to see what happens. Before you change anything though, try running the network to see what happens with the current settings.

1. Can you figure out what it is learning based on the output it prints to screen?

2. After how many iterations does it learn (i.e., have no error)?

Now try the following:

1. Change the number of training iterations as set in `Main()`. What happens?

2. Change the value of the learning rate as set in `Main()`. What happens?

3. What does `N.Run()` do? Try commenting it back in to see.

4. `N.Run()` calls the function `def Run(self):` (below `Train`) — can you tell now what it might do?

5. Can you spot similarities (and differences) between `Run` and `Train`? Imagine if it was called something less informative than `Run`, would you still be able to reverse-engineer it? Add comments to help you if you need to return to it.

6. Change the graphs parameter in the initialisation of the network `N` to `True`. What happens?

7. Is there something with respect to the actual running or training of the network that you do not understand? Try to use `print` to see what is going on in such a case.

With respect to the graphs, they represent what the network knows about the inputs. Inputs that are classed as 1.0 (their target is 1.0) are represented with one colour and inputs in the other class are coloured with another. The line that moves around during training is a function of the weights and shows how the network is separating the data into two groups (i.e., y is either 1.0 or 0.0).

Based on these graphs can you tell when the network has learned?

3 Teaching the network logic

To explore some of the very basic uses of neural networks, you will first attempt to teach the 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 (i.e., 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 (recall this allows for both as well as either). Such formalisation is useful because it is an easy way to avoid ambiguity, thus helping us gain a clear understanding of what a network is telling us.

3.1 Not

The first logical operator you will teach your 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 you give the network a 0 you want it to return a 1, and when you give the network a 1 you 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

3.2   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 you 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 because when I wrote the code I made no assumptions about the number of input units. 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**.

3.3   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**.

3.4   Could the network buy me fruit?

Instead of only learning the outcomes of logival operators the network can try to clasify some fruit based on two of their features, colour and taste, into ones I do and ones I do not want to eat. Can the network learn which of the fruit in Table 4 I like?

Name	Colour	Taste	Does Olivia like it?
	Yellow-Red	Sour-Sweet	Yes-No
Loquat	0.1	0.5	1.0
Lemon	0.0	0.0	0.0
Apple	1.0	0.8	0.0
Strawberry	1.0	0.9	1.0

Table 4: Some fruit for the network to categorise.

Can the network classify the fruit based on my personal preferences? Yes / No

Is it an easy classification for the network to learn? Yes / No

Does changing the learning rate help? Yes / No

Does training for longer help? Yes / No

3.5   Xor

Things start to become interesting when you 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 5 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 5: Truth table for logical **xor**.

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

Is it an easy classification for the network to learn? Yes / No

Does changing the learning rate help? Yes / No

Does training for longer help? Yes / No

Describe what is going on here.