Pyceptron exercises

Olivia Guest

October 13, 2015

1 Running a perceptron on paper

Using the equations from the hand outs calculate the output of the perceptron in Figure 1. Firstly, because this is just a practice run to help your understand how activations propagate ramdomly choose some values for the weights and inputs (simple small numbers will make the arithmetic easier). 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 function for the output unit you can use an activation function. A simple version of this function, denoted by f in the handouts, is to change the output unit's value to 1.0 if the input to it above 0.5, otherwise set it to 0. What does this help the network do?

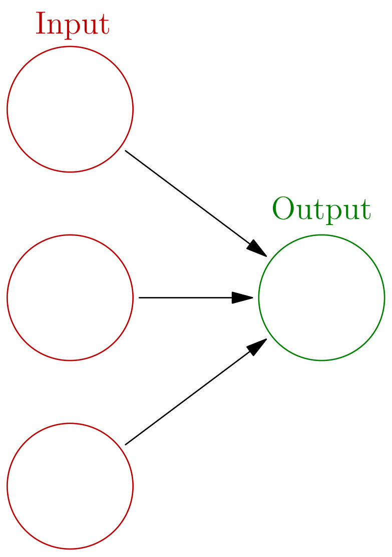

Figure 1: A perceptron that you can run on paper.

2 Playing around with the network

Now let's have a go playing around, but hopefully not breaking, the network (see Figure 1). However, don't worry if you do break it, there is an infinite supply.

So let's have a go at looking at the code that makes it learn. 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):

$$y_j = f\left(\sum_1^N w_i \times x_i\right) \tag{1}$$

where y_j is the (output) unit whose state we 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 we 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. Please ask for help if something is not making sense straight away.

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

$$\Delta w_i = \eta(d_j - y_j) \times x_i \tag{2}$$

where d is what we 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:

1. Change the value of the learning rate η, aka `h`. What happens?
2. Does printing out the values of the variables, e.g., `w`, `h`, `x`, `y`, etc. help you understand what's going on?
3. What does `self.Draw()` do? Try commenting it out.
4. What does the function `def Run(self):` (below `Train`) 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?

3 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 (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. Such formalisation is useful because it is an easy way to avoid ambiguity which is required when programming.

3.1 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

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 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 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?
	Red-Yellow	Sweet-Sour	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

On line BLAH comment out where it produces graphs. What are the graphs depicting?

Does changing the learning rate help? Yes / No

Does training for longer help? Yes / No

3.5 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 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

Does changing the learning rate help? Yes / No

Does training for longer help? Yes / No