Perceptrons, Part 1 – Constructing & Verifying

Eckel, TJHSST AI2, Spring 2022

What Is a Perceptron?

We'll continue our study of **supervised machine learning** with a study of **perceptrons**. A perceptron is a simple trainable algorithm for **binary classification**. Binary classification means our concept has only two possible output values. As the unit goes on, we will make networks of perceptrons that can approximate complex concepts, but we'll start small.

A perceptron is effectively a simple function on a set number of inputs. It consists of an **activation function** $A(t)$, a **weight vector** $\vec{w}$, and a **bias scalar** b and then for any input vector $\vec{x}$ it returns $A(\vec{w} \cdot \vec{x} + b)$. The multiplication symbol here represents the dot product – the sum of pairwise multiplications of each pair of corresponding values in the two vectors. If you see the dot product and have a vague feeling that you might have forgotten what that means, I encourage you to take a moment to google vector dot product to refresh your memory; you'll need a clear mental image of what's happening here.

In any case: that's the whole idea; a perceptron comprises quite a simple calculation! Let's investigate in detail what a this single perceptron can, and can't, do.

Your first task is to use perceptrons to model Boolean functions. After a review of Boolean functions, I'll explain why, and what we'll learn as a result.

Boolean Functions and Canonical Integer Representation

To be general, a Boolean function takes in any number of inputs, each valued as either True or False, and outputs a single True or False value. By now, you're familiar with using "0" to denote False and "1" to denote True; you're probably also familiar with common two-variable Boolean functions like AND and OR:

AND:

In_1	In_2	Out
1	1	1
1	0	0
0	1	0
0	0	0

OR:

In_1	In_2	Out
1	1	1
1	0	1
0	1	1
0	0	0

As you'll see later, we're going to need a way to specify Boolean functions efficiently; let's explore this idea further and find one.

Let's start by asking – how many possible Boolean functions are there on 2 inputs, or stated equivalently, on a 2-bit input vector? In other words, how many distinct truth tables can I create with possible input vectors 11, 10, 01, and 00, as above? The answer is $16 = 2^{2^2}$ because the truth table is size 2^2 and each entry has 2 possible values. What about 3 bit functions? Answer: $2^{2^3} = 256$. (Remember, exponents evaluate from the top down.) Check yourself: do you understand this calculation makes sense? When thinking about 3-bit inputs, what does the 3 represent? The 2^3?

Of course, some Boolean functions are logically more useful than others, and so we have given a few of them specific names. You know names for many of the 16 2-bit functions, such as AND shown above, but say you wrote a truth table and then made all the outputs "1". It's a valid function – each input produces a consistent, specific output – but it's not a function we'd use very often so it doesn't have a name. This is even more true with higher numbers of bits; most 3-bit and 4-bit Boolean functions have no common name at all.

Without individual names to use, we'll want a more systematic way of uniquely identifying any possible specific Boolean function on any number of input bits. For this purpose, we'll use the **canonical integer representation** of each function. Note that the truth tables above start with all "1"s and count down to all "0"s in the input columns. If we standardize this way of writing truth tables, then to describe a Boolean function to you, I would only need to tell you the particular digits in the output column and the number of input bits (which would allow you to write all the input possibilities in this standardized order on the left). So, for example, the 2-bit AND function on the previous page could be defined by the binary string 1000. We could call that "2-bit Boolean function 1000" and, since we've decided on a standardized order in which to write out the input possibilities, that specifically defines the function.

But: we don't usually think in binary, so let's convert. "1000" in binary is "8" in decimal. So, let's call AND "2-bit Boolean function #8". OR, also shown on the previous page, would be represented as 1110, and thus becomes "2-bit Boolean function #14". 2-bit Boolean function #15 would refer to the functioned I mentioned that just outputs all "1"s; 2-bit Boolean function #0 would output all "0"s.

This convention is what we'll use, then. I should be able to tell you "use 4-bit Boolean function #561" and your code should be able to create the entire truth table – the complete, 16-item list of input / output pairs, where input is a 4-value tuple/vector and the output is a single 1 or 0.

It is important to remember that we need to specify the number of bits! Below, on the left, is 3-bit Boolean function #8 which is **different** from 2-bit AND. For comparison, the 3-bit version of AND is on the right. It turns out to be 3-bit Boolean function #128.

3-bit function #8:

In_1	In_2	In_3	Out
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	0

3-bit function #128 (3-bit AND):

In_1	In_2	In_3	Out
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	0

Later, you'll need to write a function that takes the number of bits and canonical integer representation of a Boolean function and generates its truth table. Make sure you understand canonical integer representation well enough to do this!

Write Code to Generate Boolean Functions

Before we get back to Perceptrons, let's just make sure we can use this canonical integer representation to make truth tables correctly. Code up the following two functions:

- `truth_table(bits, n)` – this takes a number of bits and the canonical integer representation of a Boolean function on that number of bits and returns a truth table for that function. For example, `truth_table(2, 5)` would return `(((1,1),0), ((1,0),1), ((0,1),0), ((0,0),1))`. You can pick the data structure – tuples of tuples or a dictionary matching tuples to integers both seem reasonable.

- `pretty_print_tt(table)` – this takes a truth table as output by the previous function and prints it nicely. It should look something like the truth tables I wrote as examples on this assignment sheet, though obviously with cruder formatting. Be sure that the input values are ordered the same way as mine – all "1"s first.

Using Perceptrons to Model Boolean Functions

And that brings us to the description of this lab: we want to see which Boolean functions a single perceptron is capable of modeling.

This is an excellent introductory exercise for coding perceptrons as it will demonstrate both the versatility and limits of the perceptron architecture on a data set that we can generate ourselves easily. We already understand the complexity of the set of all possible *n*-digit Boolean functions, so we don't need familiarity with a particular data set to run the exercise. Also, we can train on a **complete** set of possibilities; our training set can be 100% of the entire observation space. So we can see how a perceptron behaves in this idealized setting.

Later, we will use much messier real-world data sets, of course, but we'll need networks that are more complex than a single perceptron in order to interpret them.

For this part of the assignment, **we won't be training our perceptrons yet**. This part of the assignment is just the ground work – code to model a perceptron, code to model Boolean functions, and code to compare them with each other.

For example, consider the following perceptron specification:

$A(t) = step(t)$ or, in other words, $A(t) = 1$ if $t > 0$ otherwise $A(t) = 0$.
$\vec{w} = \langle 1, 1 \rangle$
$b = -1.5$

This precisely models the AND function shown on the previous page. Consider the input vector $\vec{x} = \langle 1, 0 \rangle$:

$$\begin{aligned} A(\vec{w} \cdot \vec{x} + b) &= step(\langle 1, 1 \rangle \cdot \langle 1, 0 \rangle - 1.5) \\ &= step\big((1 \cdot 1 + 1 \cdot 0) - 1.5\big) \\ &= step(1 - 1.5) \\ &= step(-0.5) \\ &= 0 \end{aligned}$$

…just as it should be, matching the second row on the truth table above. (Note that I have represented the input vector in blue so you can more easily see how the dot product functions in this example.)

The bottom line: **this particular specification of activation function, weight vector, and bias scalar gives us a function that reproduces the output of an AND gate exactly.** If that sentence doesn't make sense, go back and rethink this example more carefully (or ask me or a peer some questions!)

Goals of This Exploration

Ultimately, with part 2 of this assignment, our goal will be to write code that trains a perceptron to model a particular Boolean function. In other words, we'll want to simply start from the truth table shown on the previous page and have our code generate a successful weight vector and bias scalar to recreate that truth table; in the parlance of machine learning, we'll want our code to generate a perceptron that successfully classifies each observation as either "1" or "0" to match the data in the truth table.

But that's next week. This assignment has a lot of moving parts. For this week, I just want each part to work. So, **for now, we're just going to generate a truth table, specify a perceptron, and check how well the truth table and the perceptron match.**

Model Perceptrons & Verify Against Boolean Functions

Now please write the following functions:

- `perceptron(A, w, b, x)` – this takes as parameters a function `A`, a vector `w`, a scalar `b`, and an input vector `x`, outputting the result of the perceptron's calculation. Note we are passing a function as a variable – Python lets you do this, as long as the function's name is written without parentheses. So, for example, if everything is correctly written, this code:

```
def step(num):
    # write code to correctly model a step function (make sure to return an int)

def perceptron(A, w, b, x):
    # write code to correctly model a perceptron

print(perceptron(step, (1,1), -1.5, (1,0)))
```

  …should print 0 (see example on previous page).

  Inside the perceptron function, you should use "A" as the name of the activation function. By passing "step" in as "A", the "A" calls will run the "step" code. (If this doesn't make sense right away, just know that you **cannot** write "step" anywhere inside the perceptron function, and if you keep that in mind I have a feeling that you'll figure out the rest as you go.)

- `check(n, w, b)` – this takes a canonical Boolean integer representation `n` and weight vectors `w` and `b` and calls `truth_table` using `n` to create the actual truth table of the Boolean function in question. Then, it tries all of the inputs in the truth table on the perceptron created with `w` and `b` and compares the output to the actual truth table, keeping track of how many inputs are classified correctly and how many are classified incorrectly. It should return the accuracy of the perceptron as a decimal from 0 to 1.

  Note we do *not* need to pass in the number of bits to the check function! Since you can't dot-product vectors of unequal lengths, the number of bits (ie, the length of an input vector) *must* be equal to the length of the weight vector!

Your code will need to accept three command line arguments and then pass them along to "check". You know how to read integers and floats from string inputs, but the second argument will be written as a tuple, and it needs to be read into your code as an actual tuple instead of a string. Check out this little piece of code to accomplish that quickly:

```
import ast

x = "(1, 2, 3, 4, 5)"
t = ast.literal_eval(x)
print(t, type(t))
```

Check Yourself Before Submitting!

```
> yourcode.py 8 "(1, 1)" -1.5
```

…should print `1.0` since that perceptron correctly models the AND gate in all cases, or, in other words, is 100% accurate. (See example on page 3.)

```
> yourcode.py 9 "(1, 1)" -1.5
```

…should print `0.75` since now only three rows will be correctly matched.

Finally here's a random test case to check a larger size of input vector:

```
> yourcode.py 50101 "(3, 2, 3, 1)" -4
```

This should make 4-bit Boolean function #50101 and compare it to the 4-input perceptron defined in the other two arguments (we know it's 4-bit because that tuple has length 4). The output here should be `0.3125`, as it happens.

I also encourage you to find a few other test cases and run them, and you can feel free to share with other students. Either way, **make sure it runs correctly on 3, 4, 5, etc length weight vectors as well!**

Specification

Submit **a single python script** to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code accepts **three command line arguments** – n, w, and b as described in the explanation for the "check" function. See above examples. **Again: it should be possible for w to be any length, not just 2!**
- Your code prints the percentage of the time that the truth table and the perceptron match.