Perceptrons, Part 1 – Constructing & Verifying

Eckel, TJHSST AI2, Spring 2020

# The Big Picture: Supervised Machine Learning

The remainder of our course will focus on **supervised machine learning**, or training a program to make distinctions in data based on a set of pre-classified inputs. Some vocabulary:

* Supervised learning is designed to deal with **classification problems**.  For example, look at a photograph and determine if it is a cat, dog or porcupine.  Or, listen to a song and recognize if it is Beethoven, Bach, or the Beatles.  Analyze local weather data and decide to carry an umbrella or not.  Etc, etc.
* The vocabulary here is that we want to put **observations** into **classifications**.  So, one observation might be a picture, and its classification would be "CAT", or one observation might be several pieces of weather data, and its classification would be "CARRY AN UMBRELLA".
* Observations come in the form of **feature vectors**.  A vector is just a collection of values, remember - a tuple is roughly the same mental model - and so when I say feature vector, I mean a collection of values of various **explanatory variables**, also called **independent variables**.  For the umbrella situation, this might be discrete values of several variables like temperature, barometric pressure, wind reports, etc.  For the image, it would be each individual R, G, and B value of each individual pixel; a 50x50 image would be a 7500 variable feature vector!  (We'll make neural networks complex enough to handle that much input later in the quarter; for now we'll start smaller.) Remember: the *observation* is *the entire vector*. One observation is many variables.
* We assume that there is an ideal function, often called a **concept**, that will perfectly classify each feature vector into its correct classification.  Call this *f*.  Due to data or algorithmic limitations, in most cases we will not find *f*.
* Instead, we’re generating *an approximation of the concept*, a **classifier function**.  The usual notation is .
* With **supervised learning**, we begin with several observations that are already classified and we use them as a **training set** to get our as close as possible to the theoretical *f*. The **error rate** on any given set of inputs is the percentage of inputs incorrectly classified. The idea is that if we get a low error rate on the training set, we will likely have a low error rate on new inputs as well.

To summarize, the fundamental concept of machine learning needs to be clear: we are **approximating a concept with a classifier function**.  In supervised learning, we **train a classifier function on a known training set** before using it to classify unfamiliar data. (If you’re curious about unsupervised learning, you can join the optional Piazza class I created over the break; it contains an assignment on K-Means, an unsupervised learning algorithm, as well as another supervised learning algorithm that constructs decision trees.)

The preceding text is dense; take a moment to check yourself. Scan back over all the bolded words and fragments. Do they all make sense? Take a moment and talk through them in your mind, as if you were explaining them to someone else. Make sure you’re ready to continue!

# What Is a Perceptron?

We’ll begin 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 takes an **activation function** , a **weight vector** , and a **bias scalar** and then for any input vector it returns . The multiplication symbol here represents the dot product – the sum of pairwise multiplications of each pair of corresponding values in the two vectors. (Feel free to google dot product to refresh your memory if necessary.) This is quite a simple calculation! Let’s see what a 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.

# Boolean Functions and Canonical Integer Representation

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:

|  |  |  |
| --- | --- | --- |
| In1 | In2 | Out |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

OR:

|  |  |  |
| --- | --- | --- |
| In1 | In2 | Out |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

So, how many binary functions are there on 2 bit inputs? In other words, how many truth tables can I create with inputs 11, 10, 01, and 00, as above? The answer is because the truth table is size and each entry has 2 possible values. What about 3 bit functions? Answer: . Check yourself: do you understand why?

Of course, some Boolean functions are logically more useful than others. You know names for many of the 16 2-bit functions, but you don’t know a name for, for instance, 2-bit Boolean function 3. Make the truth table – note that this is simply NOT In1! As a result, it doesn’t have a particular name as a 2-bit function.

This is even more true with higher numbers of bits; most 3-bit and 4-bit Boolean functions have no name at all.

It will be convenient for the purposes of this lab to have an easy way of representing Boolean functions, though, so without individual names to use, we’ll want a more systematic way of uniquely identifying each function. 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. If we standardize this way of writing truth tables, then we only need the numbers in the out column. So, for example, AND could be defined by the binary string 1000, which in base 2 represents the number 8. So, AND can be represented as “2-bit Boolean function #8”. OR, represented as 1110, becomes “2-bit Boolean function #14”. 2-bit Boolean function #15 would output all “1”s; 2-bit Boolean function #0 would output all “0”s.

**Note 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:

|  |  |  |  |
| --- | --- | --- | --- |
| In1 | In2 | In3 | 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):

|  |  |  |  |
| --- | --- | --- | --- |
| In1 | In2 | In3 | 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!

# 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 (once we can build canonical integer representation truth tables). 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:

or, in other words, if otherwise .

This precisely models the AND function shown on the previous page. Consider the input vector :

…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 next week, 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.**

# Required Task

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.)

* 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.
* 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.

Think of at least two other clear-cut test cases and run them, checking that the answers are correct, before moving on.

Also, I’ll be running test cases on weight vectors that are larger than 2 inputs. **Make sure it runs correctly on 3, 4, 5, etc length weight vectors as well!**

# Specification

Submit your **code** to <https://tinyurl.com/S20EckelPerceptrons1>.

This assignment is **complete** if:

* The “First Name” field on the Dropbox submission form contains your **class period**, not your name.
* The “Last Name” field on the Dropbox submission form contains your **last name then a comma then your first name** (like, for example, “Eckel, Malcolm”).
* 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.

For **resubmission**:

* Complete the specification correctly.

# Extension Questions

Thinking about these will help you get ahold of the next assignment much more quickly, which, given the weirdness of our current format, might be really valuable.

1. By hand, figure out a perceptron that will exactly match NOR (2-bit canonical integer representation #1).
2. By hand, figure out a perceptron that will exactly match 2-bit canonical integer representation #13.
3. Are there any 2-bit truth tables that a perceptron can’t match? (Hint: yes. Which ones?)
4. **After you’ve found an answer to number 3 on your own**, look at the Wikipedia page for perceptron and find a mention of “linearly separable”. Do some reading. Why is your answer to number 3 correct? Did you miss any?

We’ll discuss next week!