Perceptrons, Part 2 – Training

Eckel, TJHSST AI2, Spring 2020

# Background & Explanation

Last week, we laid the groundwork – now it’s time to train our perceptrons and see what they can, and can’t, model.

A video explaining the perceptron training algorithm can be found here: <https://www.loom.com/share/768b875388fa474098c5a389550ee7d5>

The formulas mentioned in the video are:

To calculate current output:

To update the weight vector:

To update the bias:

Remember to keep looping until either your target number of generations has been reached OR two consecutive epochs have the same identical outcome.

If that isn’t clear, please re-watch the video!

A few important notes about this algorithm:

* I wasn’t clear about this in the video, but it’s possible for two consecutive epochs to have identical outcomes while the accuracy still isn’t 100%. It just means that any changes during that epoch get reversed by later inputs. Bottom line: **AFTER THIS PROCESS COMPLETES, YOU MUST CHECK ACCURACY SEPARATELY. You can’t just assume it’s 100% correct.**
* In case you’re curious – this will certainly converge to a 100% correct answer *if such an answer can be found*, but it *isn’t* guaranteed to converge to the *best* answer if 100% isn’t available. So, we can definitively answer “which Boolean functions are 100% reproducible”, but not “what is the closest we can get to a Boolean function that isn’t reproducible.”
* Finally, don’t forget that I mention 100 epochs being sufficient in the video. Don’t go higher than that; it’ll inflate your runtimes unnecessarily!

# Required Tasks 1

Your goal is to write code that will determine how many *n*-bit Boolean functions can be perfectly modeled by a perceptron, for (in theory) any value of *n* (though *n* > 4 is impractical). So, your code should:

1. Take a number of bits as an argument.
2. Generate every truth table possible with that number of bits.
3. For each truth table:
   1. Start a perceptron model with a zero vector for weight and a zero value for bias and run the perceptron training algorithm until it completes, as shown in the video.
   2. **TEST THE ACCURACY** of the completed perceptron by looping over the truth table one more time and calculating the percentage of input vectors that are categorized correctly by your final perceptron.
4. Output the total number of possible functions, and how many of those truth tables could be 100% correctly modeled. For example, for *n* = 2, the answer should be “16 possible functions; 14 can be correctly modeled.”

Find the answers for 3 bits and 4 bits. Four bits will take several minutes to run; don’t be impatient!

The rest of this assignment involves pretty different thinking; it’s totally ok to stop reading here, do this part, and come back later to read the rest!

# Explorations: What Can’t Perceptrons Model?

Ok, so you’ve got perceptron training working (or you at least understand how it works). The next question is: which functions can’t be modeled?

(In ideal circumstances, we’d have a class discussion about this, but it seems like that’s not happening, so I’ll explain.)

The only two 2-bit functions that can’t be modeled are #6 and #9, also known as XOR and XNOR.

To answer the question “why don’t these work”, let’s first consider one that *does* work. We saw last week that AND can be modeled with this perceptron:

or, in other words, if otherwise .

I want to show you a different way of writing this. If we break apart the input and weight vectors and call the individual values , , , and then we can rewrite the perceptron formula as follows:

This time, the dot just means numerical multiplication. Now, when does the step function return “1”? Because we’re using the step function, it returns 1 any time the expression inside the parentheses is greater than zero. So let’s write this as an inequality, then – this perceptron returns “1”, or “true”, when the following condition is met:

Or, to rewrite:

This is a simple linear inequality on two variables, input 1 and input 2! For the specific AND perceptron above, it makes:

Let’s visualize! Examine the following graph:



This graph contains two things:

* The inequality written above, graphed with on the x-axis and on the y-axis.
* The four input vectors to the AND Boolean function graphed as ordered pairs, with the color RED representing inputs that should be evaluated as “0” or “false”, and the color GREEN representing inputs that should be evaluated as “1” or “true”.

Notice that the inequality contains the green point and all the red points are outside its solution zone!

As another example, this diagram shows the four possible inputs evaluated using the OR function, and the inequality representation of a perceptron that models OR correctly:



Specifically, this is the inequality .

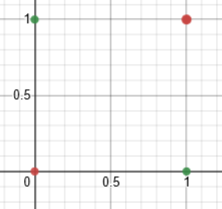
At this point, it should make sense that any perceptron with a step activation function is equivalent to a linear inequality. These diagrams are all on two variables, but the same logic applies to 3, 4, etc.

Ok…so why can’t we model XOR?

This is the truth table for XOR:

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

…and when we graph those points, this is what we get:



Do you see the problem? It’s *impossible* to write a single linear inequality that will place the green points on one side and the red points on the other side!

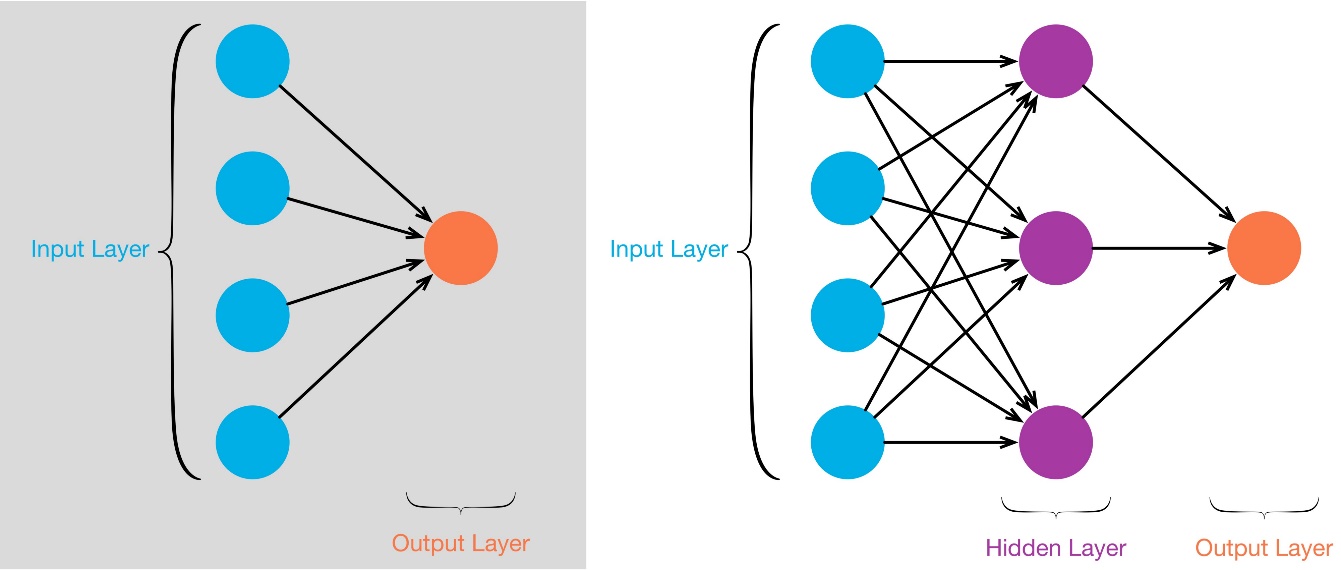
This is a **CRUCIAL** concept to understanding the capabilities of a perceptron. We say that the outputs of the AND function are **linearly separable**, and the outputs of the XOR function are **not linearly separable**. (If you’d like further reading, there’s a whole Wikipedia article on “linear separability”, which I highly recommend!)

**If we were in class right now, I’d have a chance to check in with people and see that the idea of linear separability makes sense. We can’t do that today, so just make sure that this is clear! I’m trusting you to ask questions if it isn’t.**

# Ok… So… What next?

Well, if *one* perceptron won’t model XOR correctly, we’ll have to use more than one. Specifically, we’ll have to use a **perceptron network.**

A visual aid:



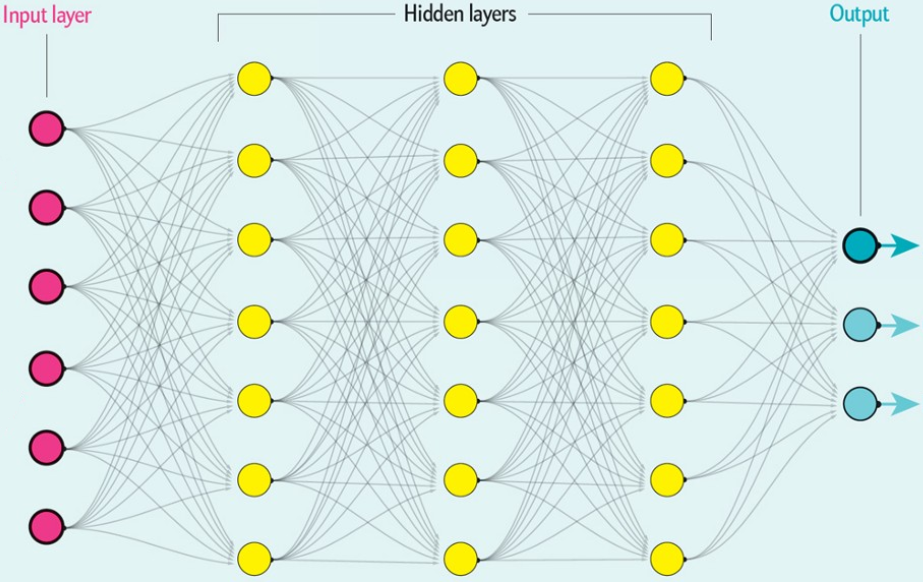
(Source: <https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f>)

On the left, you see the kind of perceptron “networks” we’ve been building so far – some number of inputs go directly into a perceptron, which gives one output.

On the right, you see what we’re going to make next – we have a **hidden layer** of perceptrons, and the outputs from the hidden layer feed into the final perceptron. The picture on the right is of a 4 – 3 – 1 network (4 inputs, 3 hidden perceptrons, 1 final perceptron).

Please note: **EVERY INPUT goes into EVERY HIDDEN PERCEPTRON**. This is a fundamental characteristic of perceptron networks, and it will continue onward when we have multiple hidden layers – every perceptron from each layer is connected to every perceptron in the next layer, always.

Our networks will look like this, eventually, though we don’t have the notation or programming knowhow to model this efficiently yet. A brief vision of your future, then; specifically, this is a 6 – 7 – 7 – 7 – 3 perceptron network:

**

# Required Tasks 2

Your next task is to **create a 2 – 2 – 1 perceptron network that models XOR correctly**. Specifically, that’s 2 inputs (which should be expected), 2 hidden perceptrons, and 1 output perceptron.

**I am NOT asking you to use the training algorithm on page 1 to accomplish this! I haven’t taught you how to train a perceptron \*network\* yet… in fact, I haven’t taught you how to do this at all! I’m asking you to figure it out in any way that makes sense to you. Feel free to work together with another student!**

Ok. Before I can specify what we’ll need, here, we need some better notation. Call the two inputs and . Call the two hidden perceptrons and . Call the output perceptron .

We need a weight vector and bias value for each perceptron; let’s start with . The weight vector will contain specific values that correspond to the connection from to and to . So this is the notation we’ll use for those values – for , . Or, in words, the weight vector of perceptron 3 is the weight from input 1 to perceptron 3 and the weight from input 2 to perceptron 3. Similarly, to distinguish perceptron 3’s bias value from the other perceptron biases, we’ll call it .

To completely specify a 2 – 2 – 1 perceptron network, then, we need all of the following values:

With that established, let me be specific. This is a paper challenge – either draw it and take a picture, or find a digital way to create a diagram – as well as a coding challenge. Your task comes in two pieces.

On paper, I need these things:

* A diagram / drawing of what the network looks like
* Every value above specified in a list near the diagram

In your code, I need these things:

* A function to run your XOR network that should visibly include three separate calls to the perceptron function specified in Perceptrons Part 1.
* A comment in your code that contains the phrase “XOR HAPPENS HERE” (so I can search easily) so I can find your code for the above bullet point.
* (Once again, your code can have all the values hardcoded; this is not a training challenge.)

# Get Your Code & Answers Ready to Submit

You’ll be submitting two files to me – a document and a .py file. There are **a lot of things I need to check** on this assignment, so please **follow these instructions carefully!**

In the document, I need:

1. All three outputs for Required Tasks 1. That is, you must include *how many Boolean functions there are* and *how many of them can be 100% modeled by a single perceptron* for each of the values *n* = 2,  *n* = 3, and *n* = 4.
2. A picture or screenshot of your 2 – 2 – 1 XOR network that includes all the specific values you used.

Your code also needs to have multiple functionalities.

1. If you receive *one command-line input*, then it will be a string of a tuple containing a pair of Boolean inputs, for example “(1, 0)”. In this case, run those inputs through your XOR network and print out the result.
2. If you receive *two command-line inputs*, then the first one will represent a number of bits and the second one will specify the canonical integer representation of a specific Boolean function. For example, 4 60800 would specify 4-bit Boolean function #60800. Your code should train a perceptron using the process described on the first page and, after 100 epochs or stability is reached, output:
   1. The final weight vector
   2. The final bias value
   3. The accuracy of the perceptron as a decimal or percent
3. Do not forget to add the comment specified on the previous page to your code!

# Specification

Submit your **code and document** to <https://tinyurl.com/S20EckelPerceptrons2>.

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 document and code match the specifications above.

For **resubmission**:

* Complete the specification correctly.

# Specification for Outstanding Work: Graph Your Trained Perceptrons

This is one of my favorite OW specifications for the whole year and I hope lots of people do it!

Install the **matplotlib** package if you haven’t already. Find a tutorial somewhere that shows you what you need to accomplish the following.

Then, write code that will loop over each 2-bit Boolean function’s truth table and:

* Completely train a perceptron as described on page 1.
* Output an inequality graph on a **domain and range of (-2,2)** to show the results of your training! Your graph should have:
  + A *small* dot every 0.1 units in both directions. **Color each small dot according to whether or not its coordinates return 1 or 0 when passed through your perceptron.** (Note that this means you’ll pass non-integer values into your perceptron; as shown on page 2, this should be fine – it’s just a linear inequality after all!)
  + A *large* dot at each of the four actual input vectors from the truth table, just like you see in the Desmos graphs on pages 2 and 3. **The large dots should not be colored according to the perceptron output; they should be colored according to the truth table.**
* In other words, what you’ll see on each graph is big green dot(s) surrounded by little green dots, and big red dot(s) surrounded by little red dots, *except* for functions #6 and #9, where we should see at least one big dot surrounded by the wrong color!

In all, your code should output 16 separate graphs.

Submit your **code** to <https://tinyurl.com/S20EckelPerceptrons2OWA>.

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 matches the specification above. (No command line arguments; it just runs.)

For **resubmission**:

* Complete the specification correctly.

# Specification for Outstanding Work: Animate the Perceptron Training Process

Write code that accepts **two command line arguments**. The first is the canonical integer representation of a 2-bit Boolean function. The second is a training rate, lambda (see formulas on page 1).

Your code should train a perceptron to match that particular Boolean function, outputting exactly the same thing that is described in the previous OW spec (little dots, big dots.) But: this time, you should output your graph after *every single step* in the training process (4 times per epoch). Specifically, the little dots should change color and the big dots shouldn’t. As a result, we should be able to see your perceptron’s solution space bounce around until it lines up with the 4 big dots correctly!

Control the animation speed so that the training process is clearly viewable with a lambda of 0.1.

(A word on packages – I don’t actually know if **matplotlib** will do animation well. You may have to resort to **tkinter**, which I know will work perfectly, if a little less naturally for making graphs!)

Submit your **code** to <https://tinyurl.com/S20EckelPerceptrons2OWB>.

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 matches the specification above. (Two command line arguments as specified above.)

For **resubmission**:

* Complete the specification correctly.