Perceptrons 4: Matrices and Non-Linearity

Eckel, TJHSST AI2, Spring 2023

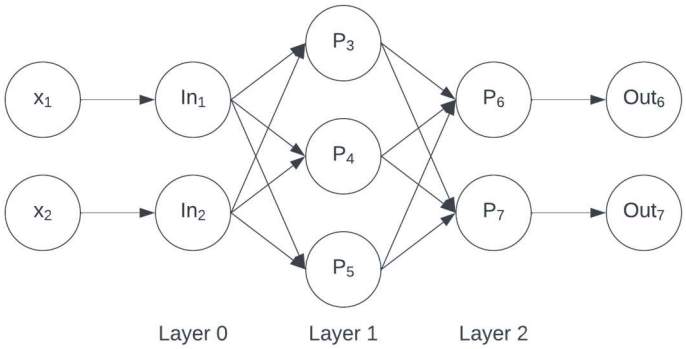
Background & Explanation

We're getting closer and closer to being ready to put perceptrons into huge neural networks that can train themselves to do complicated tasks. This assignment is about adding in the last couple of bits of understanding:

- Matrix representation of perceptron calculations, and familiarity with numpy so we can do them easily.
- Continuous nonlinearity in the activation function, and its benefits.

Matrix Representation of Perceptrons

Neural networks start to get hard to keep track of when we assign each weight and bias to an individual variable. Consider even a fairly small network, a 2-3-2 network:



Think of all the variables we need to keep track of all this!

Layer 0	Layer 1	Layer 2
In_1 Input: x_1 Output: a_1	P_3 Inputs: a_1 a_2 Weights: $w_{1,3}$ $w_{2,3}$ Bias: b_3 Output: a_3	P_6 Inputs: a_3 a_4 a_5 Weights: $w_{3,6}$ $w_{4,6}$ $w_{5,6}$ Bias: b_6 Output: a_6
In_2 Input: x_2 Output: a_2	P_4 Inputs: a_1 a_2 Weights: $w_{1,4}$ $w_{2,4}$ Bias: b_4 Output: a_4	P_7 Inputs: a_3 a_4 a_5 Weights: $w_{3,7}$ $w_{4,7}$ $w_{5,7}$ Bias: b_7 Output: a_7
	P_5 Inputs: a_1 a_2 Weights: $w_{1,5}$ $w_{2,5}$ Bias: b_5 Output: a_5	

There has to be an easier way to keep track of it all! Thankfully, there is – this is what matrices, and matrix multiplication, were made for. If you've taken / are taking Linear Algebra, this will be quite familiar! This is exactly the sort of situation that course's techniques were crafted for. If you haven't, don't worry – I'll take you through it.

Let's represent the *input* and *output* at each layer using a *column vector* (equivalently, a matrix with a single column.)

Network input:	Layer 0 output / Layer 1 input:	Layer 1 output / Layer 2 input:	Layer 2 output:
$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$	$a^o = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$	$a^1 = \begin{bmatrix} a_3 \\ a_4 \\ a_5 \end{bmatrix}$	$a^2 = \begin{bmatrix} a_6 \\ a_7 \end{bmatrix}$

Note the use here of *superscript notation* to represent each *matrix*. This is confusing because there are a lot of other mathematical situations where we use superscripts to represent exponents, but that's not what we're using them for in this particular instance. For the purposes of notation in this assignment, when we see a_1 we think/say "a subscript 1" or "a sub 1", and this means "the numerical output of *individual perceptron* 1". When we see a^o we think/say "a superscript 0" or "a super 0", and this means "the matrix of all the outputs of all the perceptrons on *layer* 0".

Once again: *subscripts* stand for *individual perceptrons*, while *superscripts* stand for *layers*.

Let us also make sure we understand the idea of *vectorized functions* – functions that take a matrix input and a matrix output and apply to each value in the matrix individually. For example, if I have a function $f(x) = 3x + 1$ and I vectorize it so that it may take a matrix input instead, that would look like this:

$$f\left(\begin{bmatrix} 2 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} 3 \cdot 2 + 1 \\ 3 \cdot 3 + 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 10 \end{bmatrix}$$

With superscript notation and vectorized functions established, let us represent weights and biases in matrices as well:

Layer 1 weights:	Layer 1 biases:	Layer 2 weights:	Layer 2 biases:
$w^1 = \begin{bmatrix} w_{1,3} & w_{2,3} \\ w_{1,4} & w_{2,4} \\ w_{1,5} & w_{2,5} \end{bmatrix}$	$b^1 = \begin{bmatrix} b_3 \\ b_4 \\ b_5 \end{bmatrix}$	$w^2 = \begin{bmatrix} w_{3,6} & w_{4,6} & w_{5,6} \\ w_{3,7} & w_{4,7} & w_{5,7} \end{bmatrix}$	$b^2 = \begin{bmatrix} b_6 \\ b_7 \end{bmatrix}$

Note that the weight matrices are designed so that each *column* corresponds to a single perceptron from the *previous* layer, and each *row* corresponds to a single perceptron from the *current* layer. Why is all of this organized in this particular way? Well, recall how matrix multiplication works – row by column. (If you don't recall how matrix multiplication works, take a moment to google / ask a friend / refer to old notes / figure it out; this next part won't make any sense otherwise.) This allows us to represent a lot of calculations in one matrix operation.

Consider this calculation, where $A(x)$ is used as both vectorized and non-vectorized versions of our activation function:

$$A(w^1 \cdot a^0 + b^1) = A\left(\begin{bmatrix} w_{1,3} & w_{2,3} \\ w_{1,4} & w_{2,4} \\ w_{1,5} & w_{2,5} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} b_3 \\ b_4 \\ b_5 \end{bmatrix}\right) = A\left(\begin{bmatrix} w_{1,3} \cdot a_1 + w_{2,3} \cdot a_2 + b_3 \\ w_{1,4} \cdot a_1 + w_{2,4} \cdot a_2 + b_4 \\ w_{1,5} \cdot a_1 + w_{2,5} \cdot a_2 + b_5 \end{bmatrix}\right) = \begin{bmatrix} A(w_{1,3} \cdot a_1 + w_{2,3} \cdot a_2 + b_3) \\ A(w_{1,4} \cdot a_1 + w_{2,4} \cdot a_2 + b_4) \\ A(w_{1,5} \cdot a_1 + w_{2,5} \cdot a_2 + b_5) \end{bmatrix}$$

This recreates each individual perceptron's calculation on layer 1 using 3 matrices instead of 11 different individual variables. Use your knowledge of matrix multiplication to verify this! Bottom line: the matrix operations encode the regularity in the calculation here, so we can represent the entire situation with much simpler notation.

The goal now is to use matrix representations in our code. It will be incredibly helpful to only need to write a single line of code that looks like the version of this calculation on the left, and let Python perform the matrix operations for us. This notation also has another advantage – the code we write won't need to change if the sizes of the matrices change. No matter how many perceptrons are on each layer, the matrix calculation looks the same!

Get Familiar with numpy

The next step, then, is to learn numpy – the pre-eminent Python package for mathematical operations, including matrices. Using numpy matrices, our code will be cleaner and shorter, and thus easier to both write and debug. Download the numpy orientation Python file on the course website and play around – everything you need is there.

Have you read it? Played around with it? Read all the comments carefully? Excellent!

Let's be clear about what we want to write, then. We want code that looks like this pseudocode:

```
function A(x) = (our activation function)
function A_vec(x) = A(x) vectorized

weights = [Python list of 2d numpy matrices of weights at each layer]
biases = [Python list of 2d numpy matrices of biases at each layer]
input = a single 2d numpy column matrix representing the input vector

function p_net(A_vec, weights, biases, input):
  let a be a Python list of numpy matrices that will hold the output at each layer
  a[0] = input
  n = the number of perceptron layers in this network
  for each layer i of the network from 1 to n:
    a[i] = A(weights[i] @ a[i-1] + biases[i])  # @ is dot product
  return a[n]

output = p_net(A_vec, weights, biases, input)
```

How clean is that! A whole network represented in just a handful of lines of code. The especially delightful thing about this is that the code, in addition to being quite simple, is universal! No matter how big our network is, no matter how many perceptrons at each layer, this function won't need any further modification.

A few **crucial** implementation notes:

- I **strongly** recommend having weights[0] and biases[0] each be None – there is no weight or bias matrix at layer 0, so adding these placeholders means you don't have to adjust which index you're using. With these None values in place, weights[1] is the weight matrix at layer 1; much easier to keep track of than trying to subtract 1 from your index each time.
- The @ symbol represents a dot product, not an itemwise product. Make sure you can tell the difference in the numpy demo.
- **Make sure that all matrices are 2-d, even if one dimension is size 1. The 1x1 matrix is still [[number]].** Numpy will very kindly try to figure out what you meant to do if you mess this up, and your code will still run, it just won't work – the math that numpy does won't be the math you're expecting.

Time for some challenges!

Matrix Representation of Perceptrons: Challenge 1, Recreating XOR

Recreate your XOR code from the last assignment. Instead of storing values individually, make a single list of the weight matrices (you should have a 2x2 matrix and a 1x2 matrix). Make a single list of the bias matrices (you should have a 2x1 matrix and a 1x1 matrix). **Once again I say - make sure that all matrices are 2-d, even if one dimension is size 1. The 1x1 matrix is still [[number]]. This is a very common mistake.** Run your network with a single call to the p_net function described above, passing it the step function, an input vector, and the weights list and biases list that you've hardcoded.

This should match the previous assignment's specification for what occurs if the code is passed a single command line variable, but now the comment "XOR HAPPENS HERE" should be next to a single p_net call as described.

After you've recreated your XOR code, move on to the next challenge.

Matrix Representation of Perceptrons: Challenge 2, The Diamond

Imagine the rotated square made by connecting (1,0), (0,1), (-1,0), and (0,-1) on the coordinate plane. A diamond centered at the origin. Your next task is to make a 2-4-1 network that will correctly classify any point as being inside or outside this diamond. Some notes:

- There is an easy way to tell if your network is correct. If the coordinates of a point are (x,y), then it is inside this diamond if and only if $|x| + |y| < 1$. In this way, it's easy to generate random (x,y) pairs and test your network.
- As a hint about architecture, this network is really seeing if your point satisfies four simultaneous inequalities (that is, being on the correct side of each of the four lines around the perimeter of the diamond). And each perceptron is a single linear inequality. This should at least tell you why I picked "4" for the number of perceptrons in the hidden layer!
- So… if the hidden perceptrons are checking each inequality individually, what should the last one do?

Your code should use a matrix representation of your network. That is, the work in specifying the network, as with the previous challenge, comes in hardcoding a weights list and a biases list. Then, you should be able to call your network with a single call to the p_net function, passing the step function, an input (x,y) pair of decimal values, and the weights list and biases list you've specified.

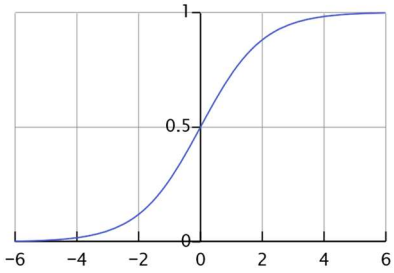
Get this so it's working perfectly, then move on to the next challenge.

Introducing Non-Linearity: Challenge 3, The Circle

Now, let's change the activation function. Instead of the step function, let's use the sigmoid function.

$$f(x) = \frac{1}{1 + e^{-x}}$$

The graph of this is as follows:



I want to be completely clear here: **this will no longer output a 0 or 1 from each perceptron**. It will output something between 0 and 1, though. When it comes to perceptron network **outputs**, we can simply round to whichever is closest – 1 or 0 – and achieve a definitive output that way. But **within the network, we will not round**; that is to say, in any case where a perceptron feeds into another perceptron, **we will pass along the unrounded decimal values between 0 and 1**.

Why do we do this? Well, the sigmoid function isn't linear. Which means that it can be used… non-linearly. Now, instead of a single stark line on a graph, yes on one side and no on the other, we have a non-linear gradient, and when two non-linear gradients are overlapped their intersection may be *curved*! (It is also true that next week we'll discover that a differentiable activation function is necessary for training, but more on that later.)

Anyway, uh, check *this* out.

In order for this to work, you need a particular solution to the previous challenge. Your final perceptron needs to have a weight vector that is all 1s. If you found a different solution, that's awesome; now, find a solution where the final perceptron's weight vector is <1, 1, 1, 1> and b is whatever it needs to be to make that work.

Once you're good there, do this:

- Copy your previous diamond network so you can modify it for a new challenge. Instead of the diamond centered at (0,0), let us consider the *circle* centered at (0,0). Now say we want your network to correctly classify whether or not a point is inside this *circle*. (Again, easy to check if the network is correct – just see if $\sqrt{x^2 + y^2} < 1$.)
- Generate 500 random points where $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$. Then, mess with the bias values on the hidden perceptrons to shift the lines to improve accuracy slightly. See how accurate you can get. You obviously can't get 100% accuracy! Four straight lines can't make a circle!
- Now, here's the fun part. Modify this network to use the *sigmoid function instead of the step function*. (If you've been doing this correctly, this should be a miniscule change – just change what function is passed in as argument A.)
- Modify the bias value in the output perceptron, and see how accurate you can get your network to be now. Without modifying any weights, just modifying biases, it should be possible to outperform the step function network by a noticeable margin – perhaps even get 100% accuracy, though 100% accuracy is not required.

(Tips for getting this working if you're having trouble are on the next page.)

Some tips if you're having trouble with the circle challenge:

- Just to be clear: **you should not modify the weight values from the previous challenge at all**. You are only modifying biases. Technically, any of the 5 bias values is fair game, though it doesn't make sense to vary one hidden perceptron without varying the others; a symmetry argument should be pretty clear here! So you're really only playing with 2 values. So: all four hidden perceptrons should have the same bias value. Let's call that i. The final perceptron's bias value I'll call j.
- Either i or j, or perhaps both, will need to be set to a value with a high degree of precision to get this to work. So: pick one, or both, and find a value that classifies every point as inside the circle. Then, find a value that classifies every point as outside the circle. In between these two values, vary by 10ths. Get as close as you can to where your code switches from classifying everything as inside to classifying everything as outside. Then vary within that range by hundredths. Then find the best outcome and vary by thousandths. Etc.
- If you aren't getting above 95% accuracy, you aren't done yet.

Save and hardcode the values of your best network.

Get Your Code Ready to Submit

There are **a lot of things I need to check** on this assignment, so please **follow these instructions carefully!**

Your code again 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. (Make sure you've commented "XOR HAPPENS HERE" again so I can see you're using matrices now!) This is how I will check **challenge 1**.
2. If you receive *two command-line inputs*, then each one will be a decimal value. Use the first as *x* and the second as *y*, and output "inside" or "outside" based on your *step-function diamond code* (not the circle challenge). This is how I will check **challenge 2**.
3. If you receive *zero command-line inputs*, then generate 500 random points as described in the circle challenge. Run them all through your hardcoded best results sigmoid circle network. Output the coordinates of any *misclassified* points, and print the percentage of the 500 that were classified correctly. This is how I will check **challenge 3**.

Specification

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

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code matches the specifications above.