Scientific Computing

Wednesday, April 8

Announcements

* HW 5 due Friday, 11:59pm

* Friday, April 17: pre-recorded lecture
  no office hours

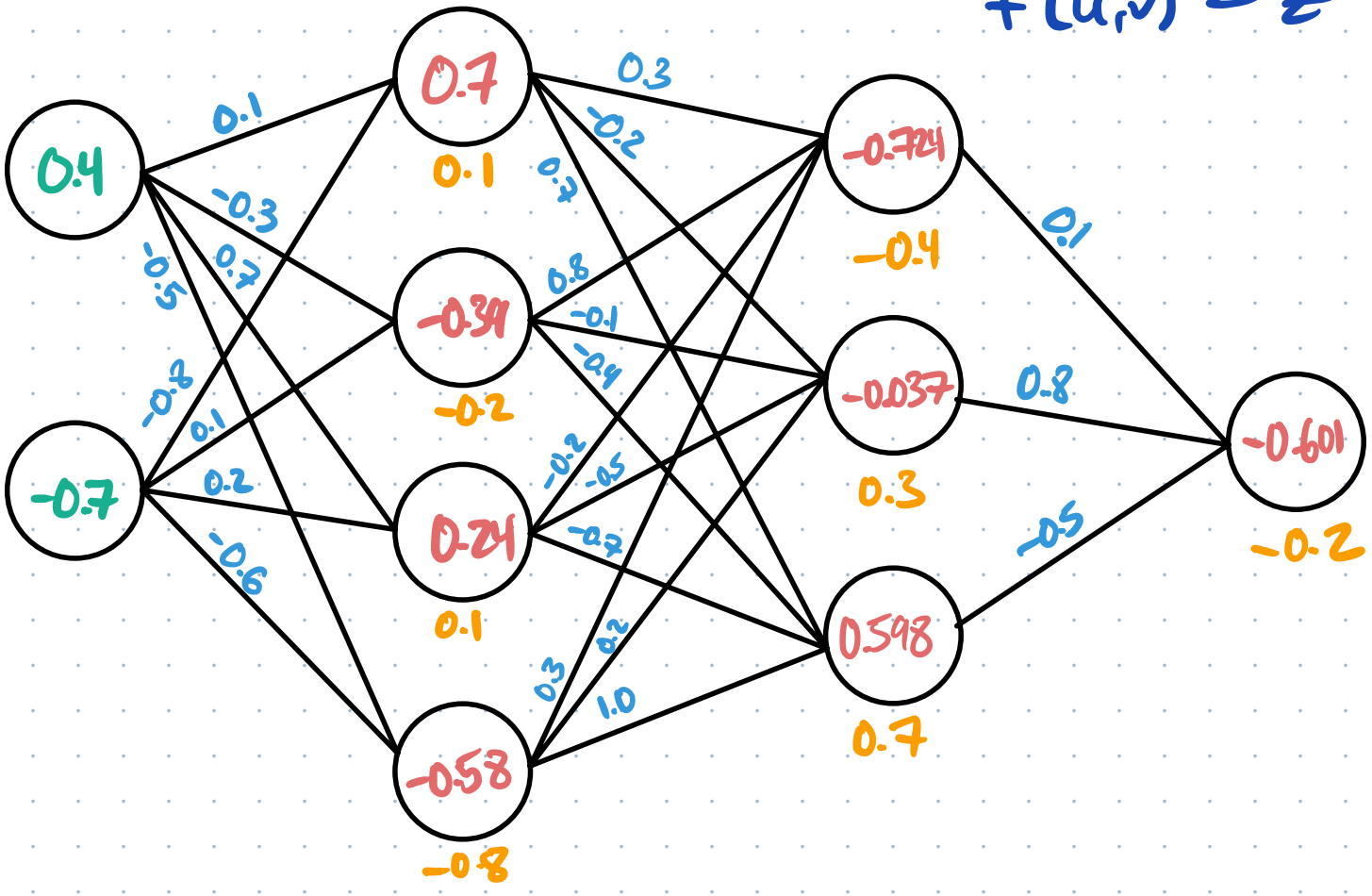
Office Hours:

Mon, 9:30-10:30
Fri, 2:00-3:00

Cudahy 307

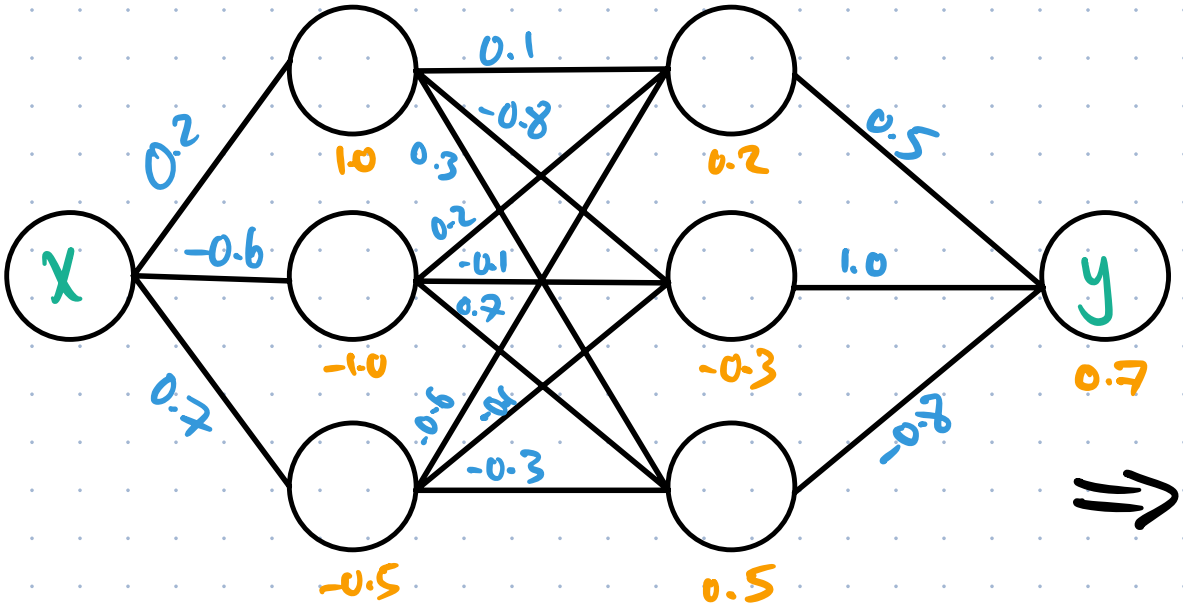
Example: A NN that takes 2 inputs and has 1 output

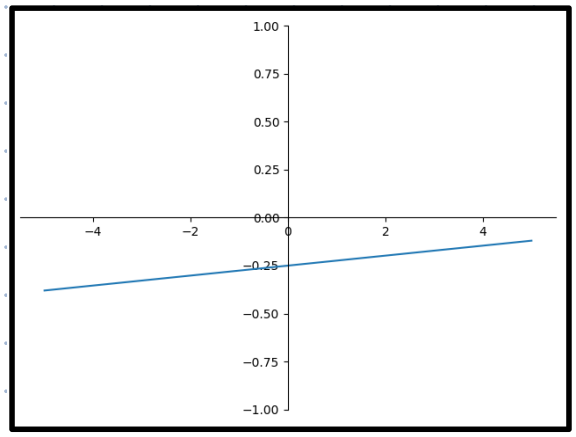
$f(u,v) = z$



So, for this particular neural network, with these weights and biases, the inputs $(0.4, -0.7)$ produce output -0.601

$$f(0.4, -0.7) = -0.601$$





$\Rightarrow y = 0.026x - 0.25$

Understanding check:

* x is the input variable, it changes
* y is the output variable, it changes as x changes

The weights and biases stay fixed. They are what define this particular function, like "m" and "b" in a line $y = mx + b$. If we change them, that's a different NN.

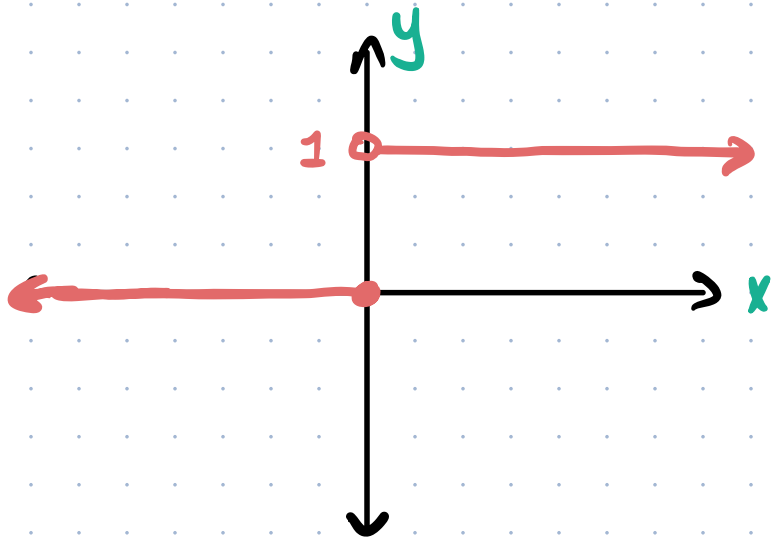
Activation Functions

In order to make NNs capable of representing non-linear functions, we pass the output of each neuron through an "activation function" before passing it on to the next layer.

The activation functions themselves are non-linear.

AF #1: Step Function (old school, not used often anymore)

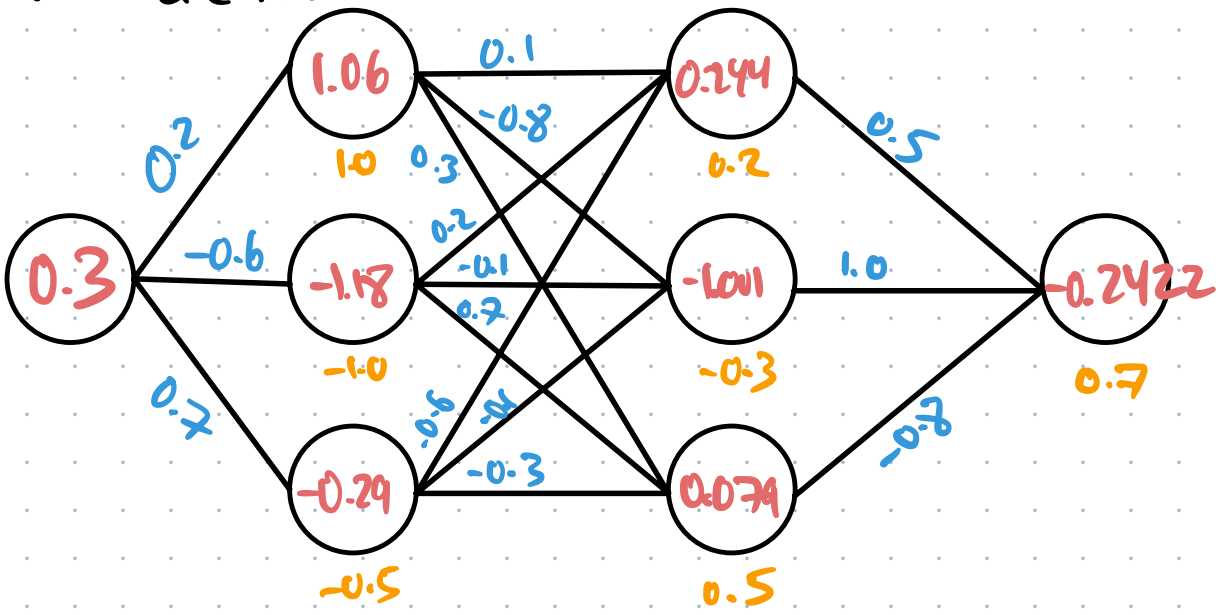
$$f(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

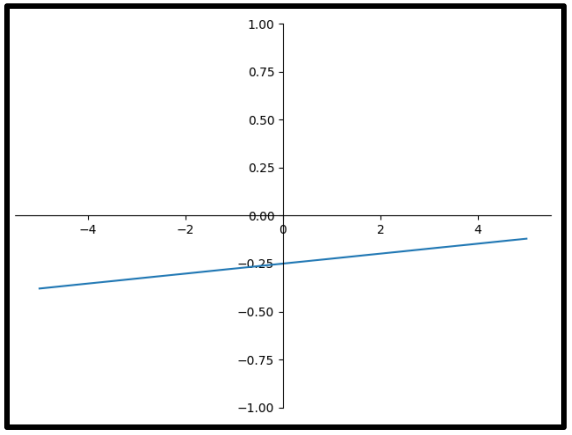


With this AF each neuron only outputs 0 or 1, not any decimal.

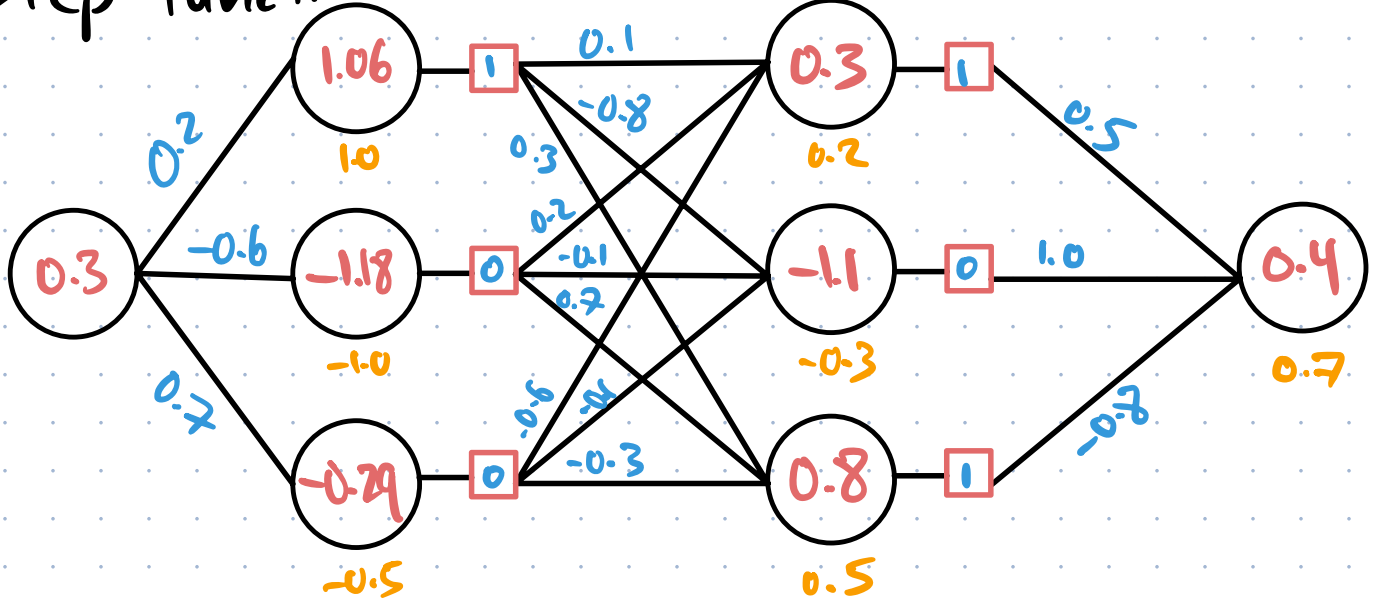
(off → 0, on → 1)

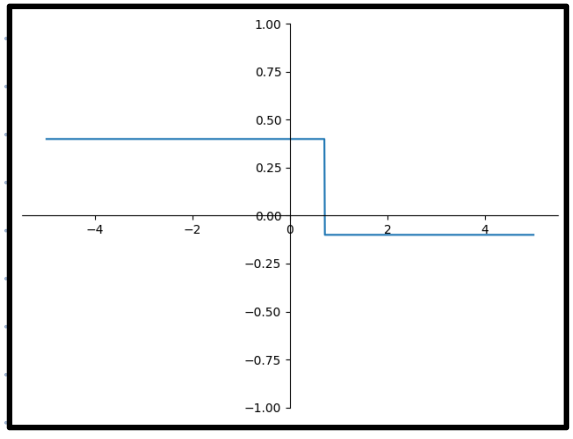
no activation:





step function:



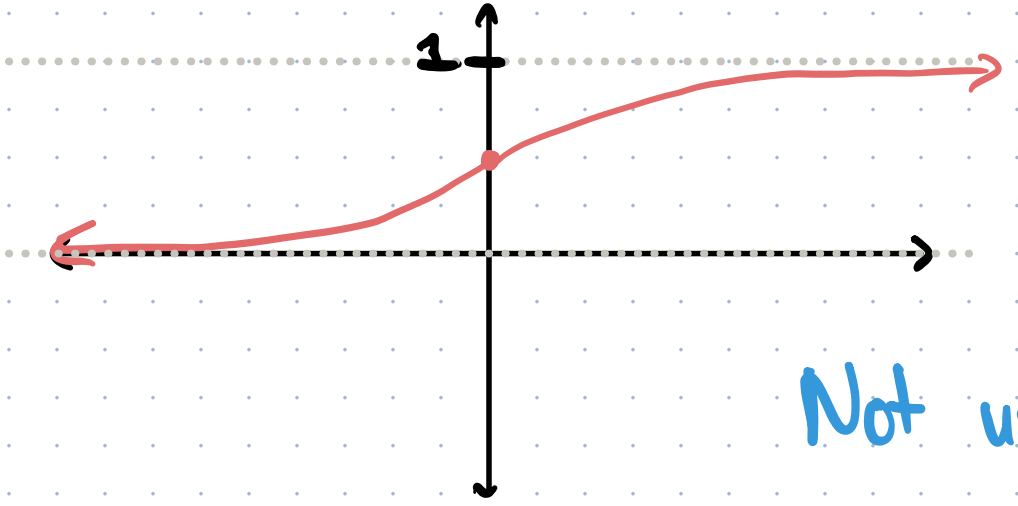


AF #2 : Sigmoid

A problem with the step function ⎍ is that it throws away a lot of information.

$f(0.001) = f(100)$

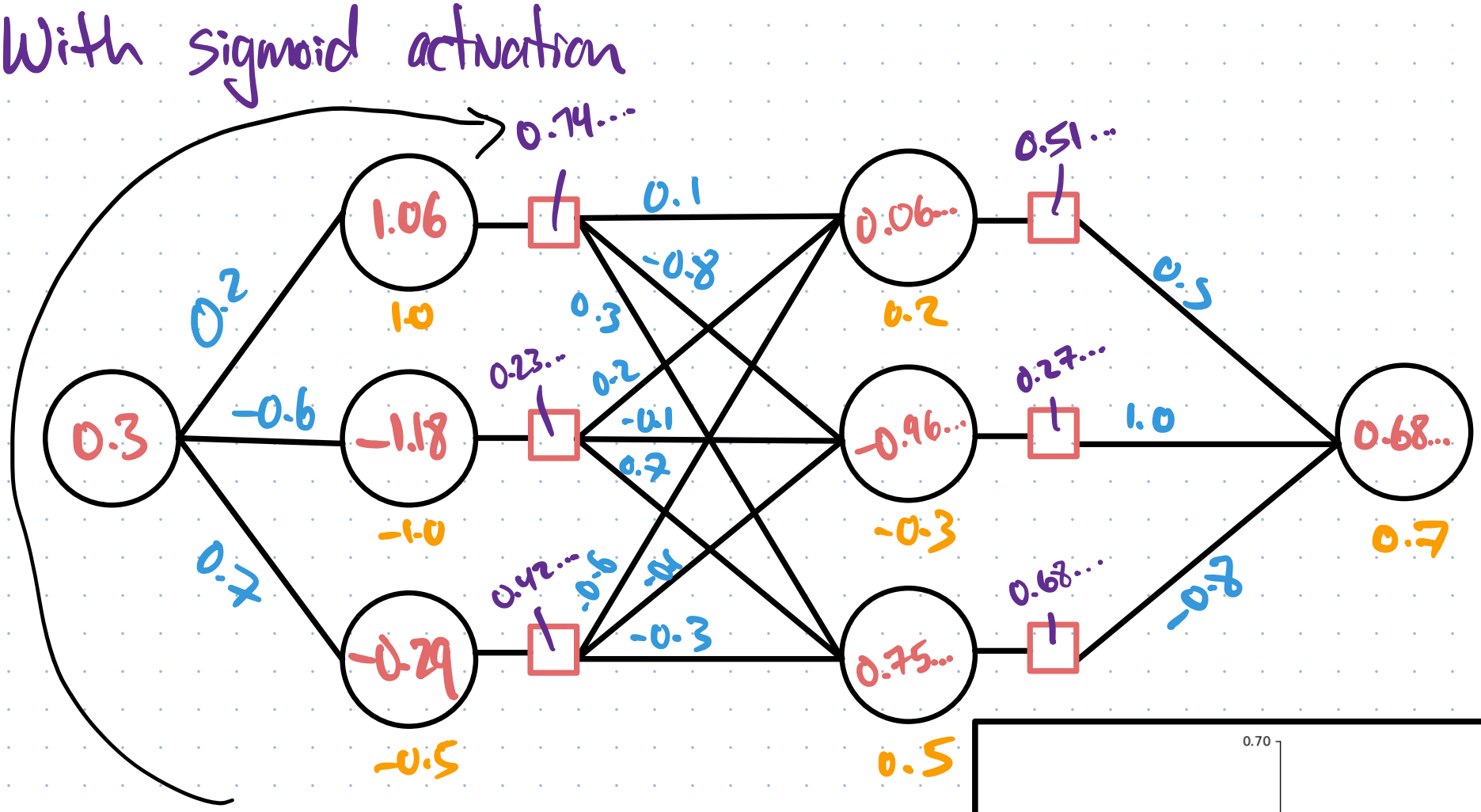
Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$



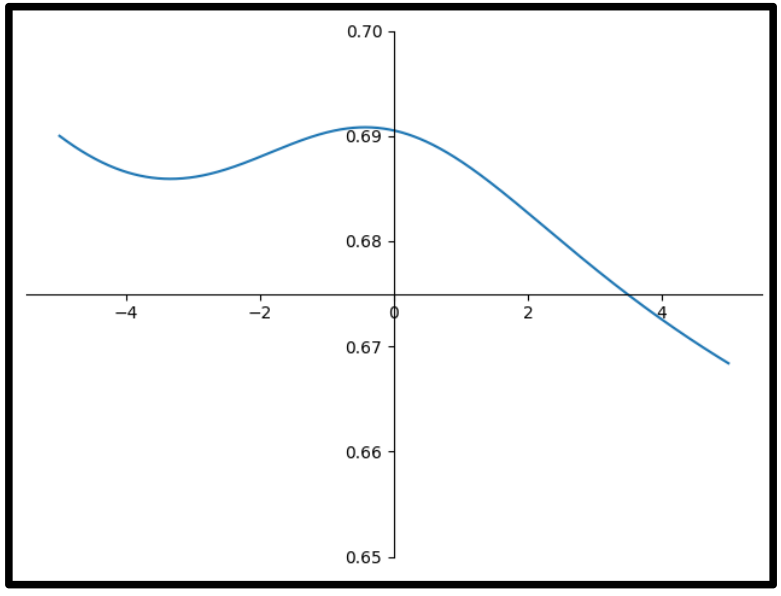
More gentle, and not as lossy.

Not used much anymore

With sigmoid activation

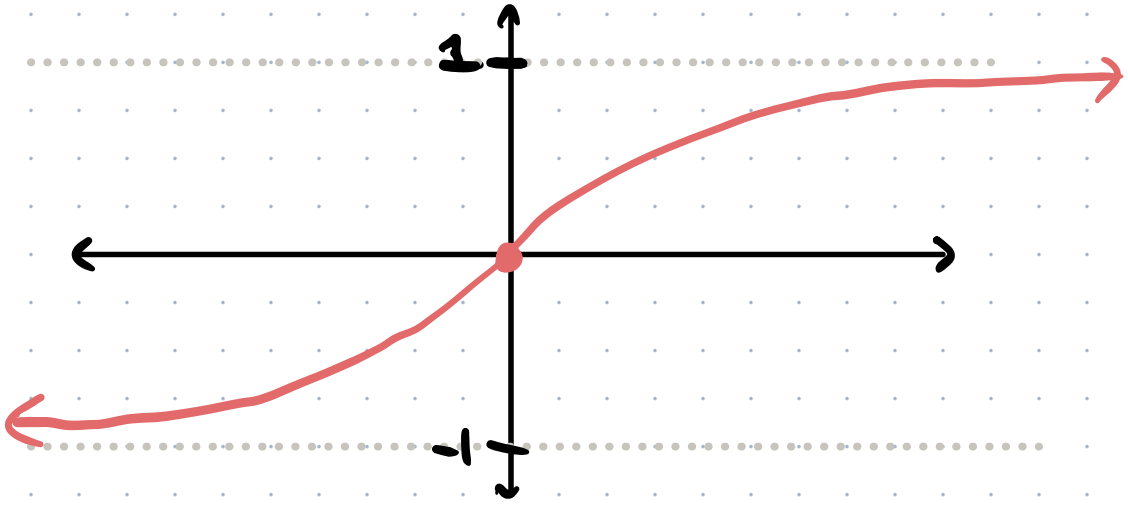


$$\frac{1}{1+e^{-1.06}} \approx 0.74$$



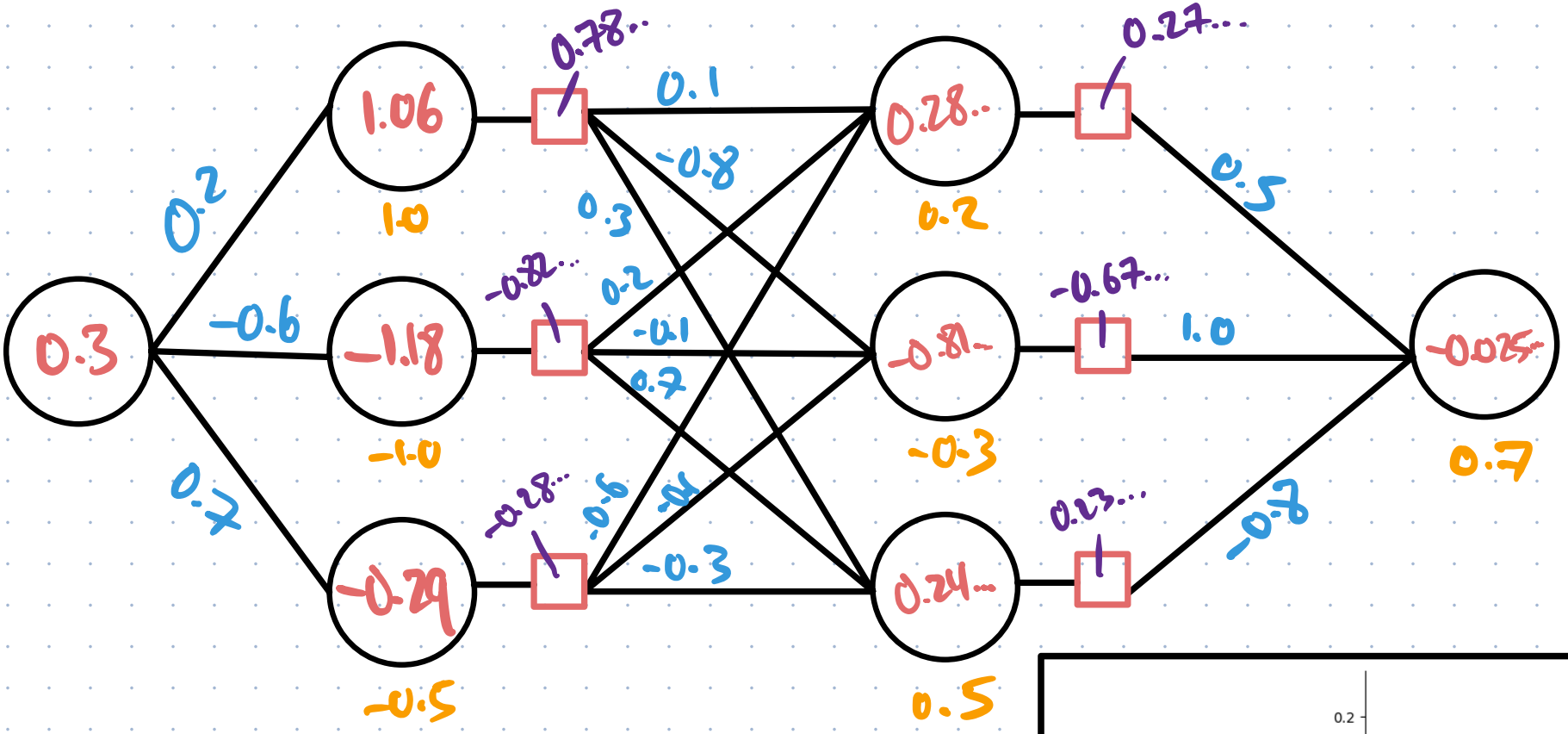
AF #3: Hyperbolic Tangent (tanh)

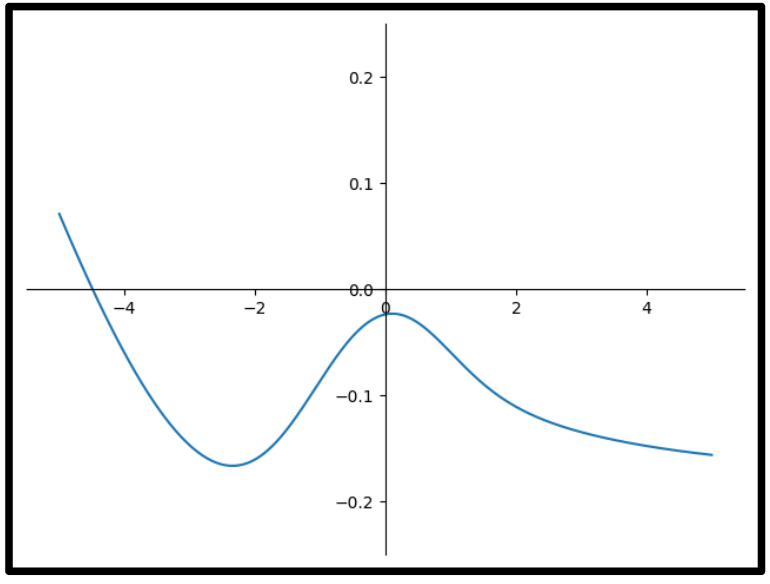
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



a bit like sigmoid, but can be negative

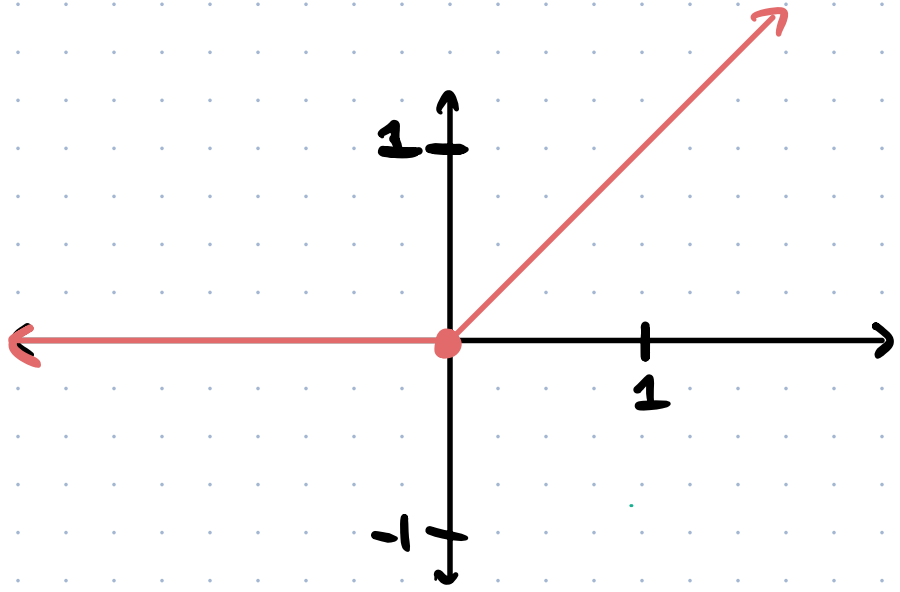
With tanh activation





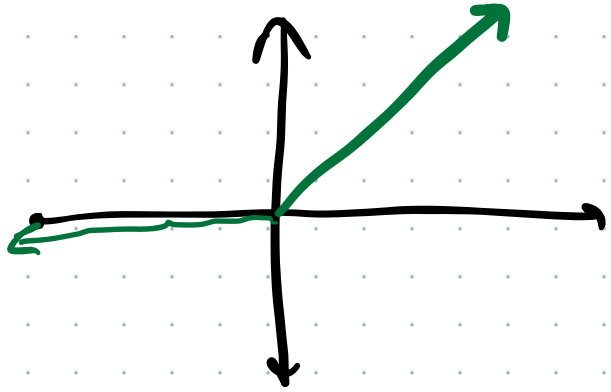
AF #4: Rectified Linear Unit (ReLU)

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$



Neuron is "on" if ≥ 0, and keeps its value, and "off" if < 0, and returns no value

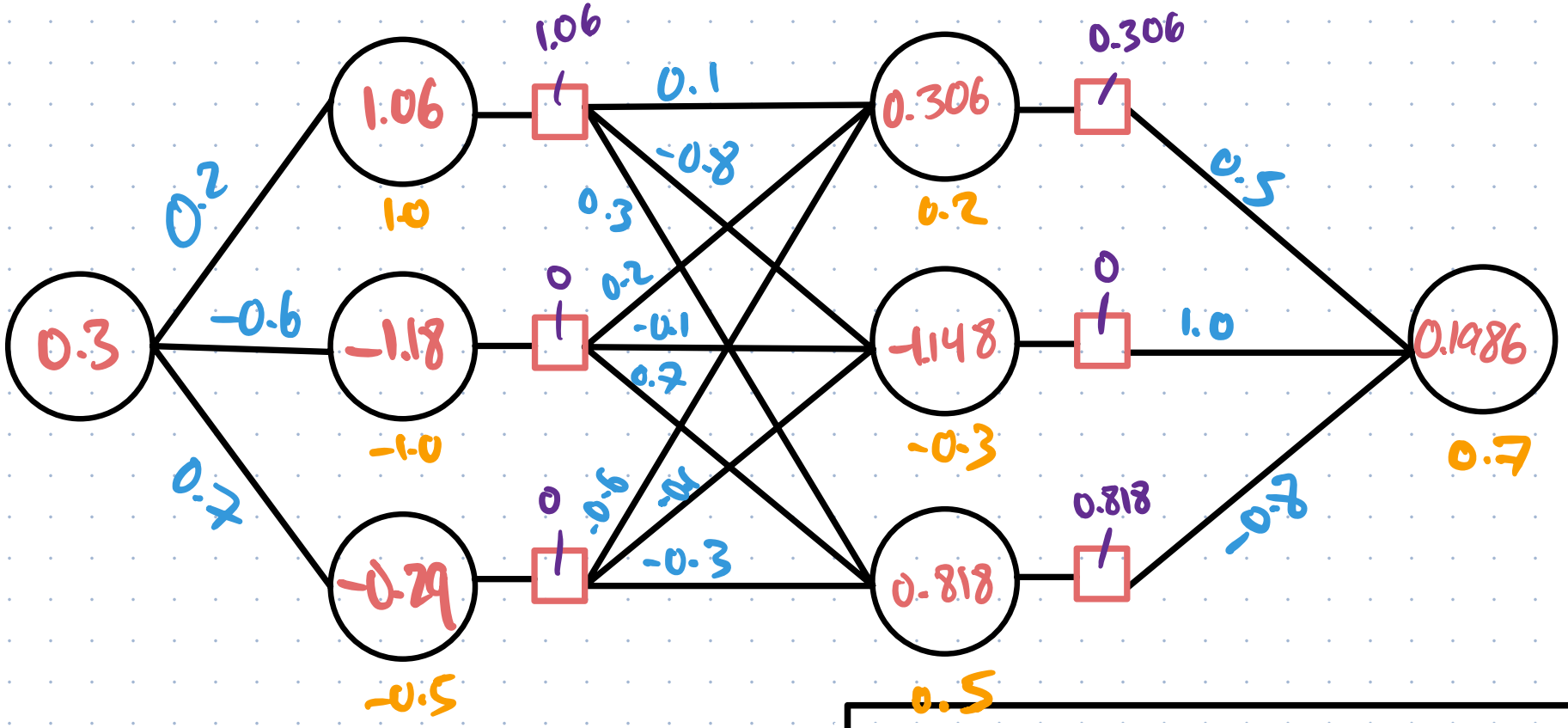
Leaky ReLU

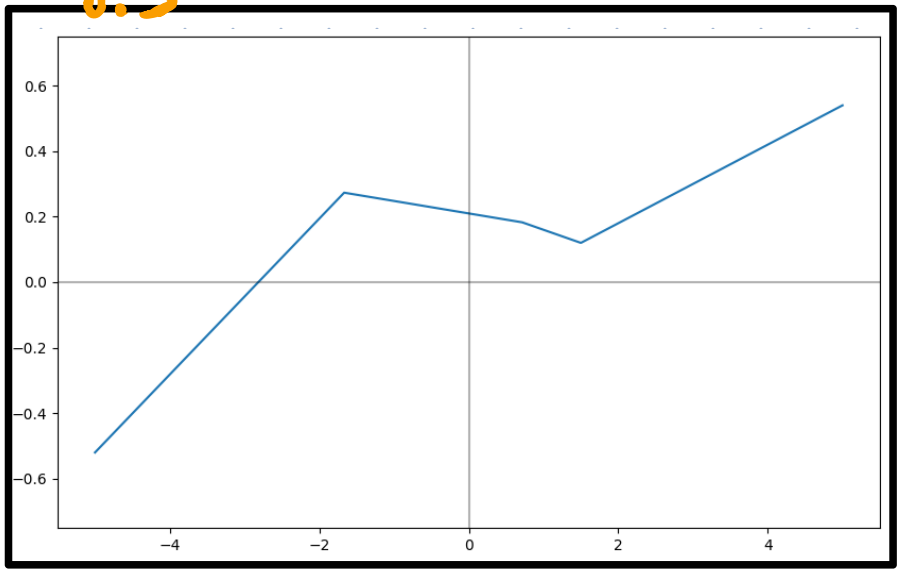


$$\begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$$

α is small, 0.1

With ReLU activation





Linear Algebra

* You may have noticed that the operations done during the forward pass feel like dot products and matrix multiplications.

* This is why GPUs work so well with NNs.

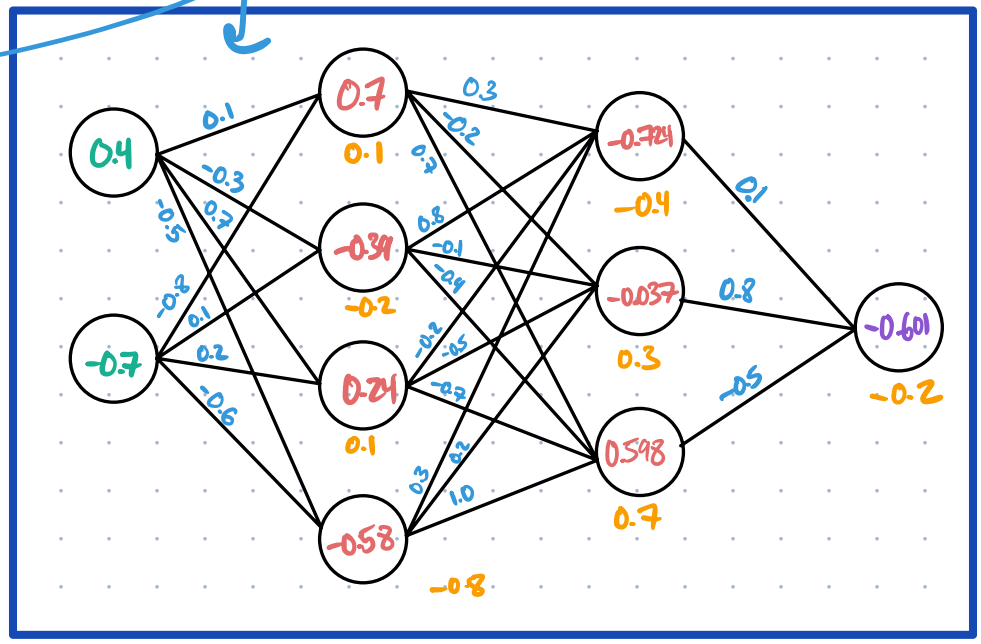
* Let's explore.

Think of each layer as a vector of values.

$$\begin{bmatrix} 0.4 \\ -0.7 \end{bmatrix} \quad \begin{bmatrix} 0.7 \\ -0.39 \\ 0.24 \\ -0.58 \end{bmatrix} \quad \begin{bmatrix} -0.724 \\ -0.037 \\ 0.598 \end{bmatrix} \quad \begin{bmatrix} -0.601 \end{bmatrix}$$

How is each layer computed from the previous one?
Make matrices for the weights between each layer and vectors for the biases.

$$M_1 = \begin{bmatrix} 0.1 & -0.8 \\ -0.3 & 0.1 \\ 0.7 & 0.2 \\ -0.5 & -0.6 \end{bmatrix} \qquad b_1 = \begin{bmatrix} 0.1 \\ -0.2 \\ 0.1 \\ -0.8 \end{bmatrix}$$



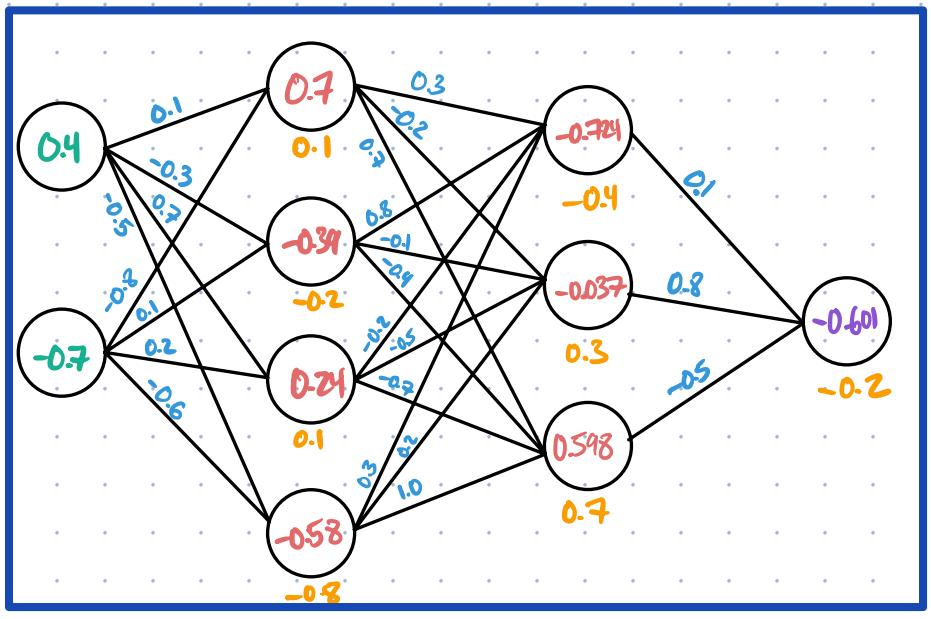
Rearrange!

(ignoring activation functions!)

$$\begin{bmatrix} 0.1 & -0.8 \\ -0.3 & 0.1 \\ 0.7 & 0.2 \\ -0.5 & -0.6 \end{bmatrix} \cdot \begin{bmatrix} 0.4 \\ -0.7 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \\ 0.1 \\ -0.8 \end{bmatrix} = \begin{bmatrix} 0.7 \\ -0.39 \\ 0.24 \\ -0.58 \end{bmatrix}$$

$M_1 =$

$b_1 =$



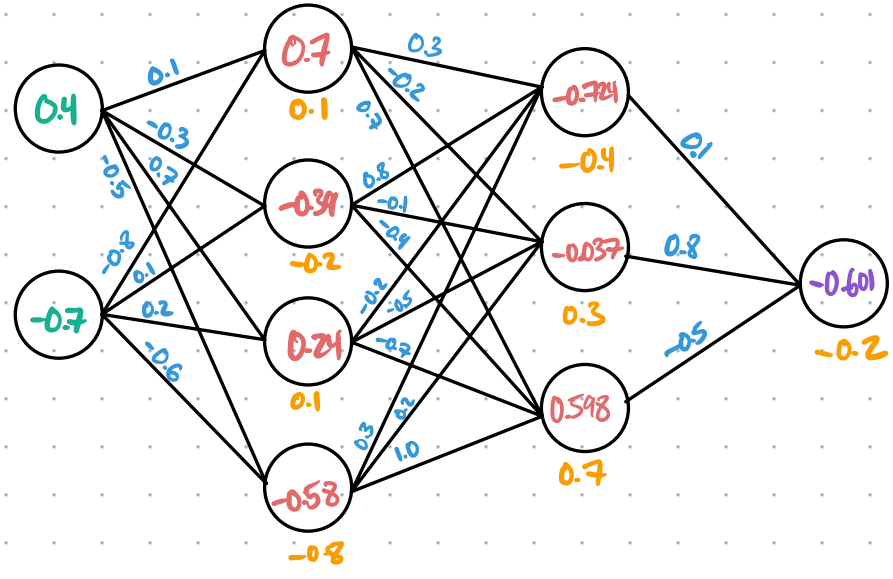
(ignoring activation functions!)

$$\begin{bmatrix} 0.1 & -0.8 \\ -0.3 & 0.1 \\ 0.7 & 0.2 \\ -0.5 & -0.6 \end{bmatrix} \begin{bmatrix} 0.4 \\ -0.7 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \\ 0.1 \\ -0.8 \end{bmatrix} = \begin{bmatrix} 0.7 \\ -0.39 \\ 0.24 \\ -0.58 \end{bmatrix}$$

$$\begin{bmatrix} 0.3 & 0.8 & -0.2 & 0.3 \\ -0.2 & -0.1 & -0.5 & 0.2 \\ 0.7 & -0.4 & -0.7 & 1.0 \end{bmatrix} \begin{bmatrix} 0.7 \\ -0.39 \\ 0.24 \\ -0.58 \end{bmatrix} + \begin{bmatrix} -0.4 \\ 0.3 \\ 0.7 \end{bmatrix} = \begin{bmatrix} -0.724 \\ -0.037 \\ 0.598 \end{bmatrix}$$

$$\begin{bmatrix} 0.1 & 0.8 & -0.5 \end{bmatrix} \begin{bmatrix} -0.724 \\ -0.037 \\ 0.598 \end{bmatrix} + [-0.2]$$

$$= [-0.601]$$



$$\begin{bmatrix} 0.3 & 0.8 & -0.2 & 0.3 \\ -0.2 & -0.1 & -0.5 & 0.2 \\ 0.7 & -0.4 & -0.7 & 1.0 \end{bmatrix} \begin{bmatrix} 0.7 \\ -0.39 \\ 0.24 \\ -0.58 \end{bmatrix} + \begin{bmatrix} -0.4 \\ 0.3 \\ 0.7 \end{bmatrix} = \begin{bmatrix} -0.724 \\ -0.037 \\ 0.598 \end{bmatrix}$$

$$Wx_1 + b = x_2$$

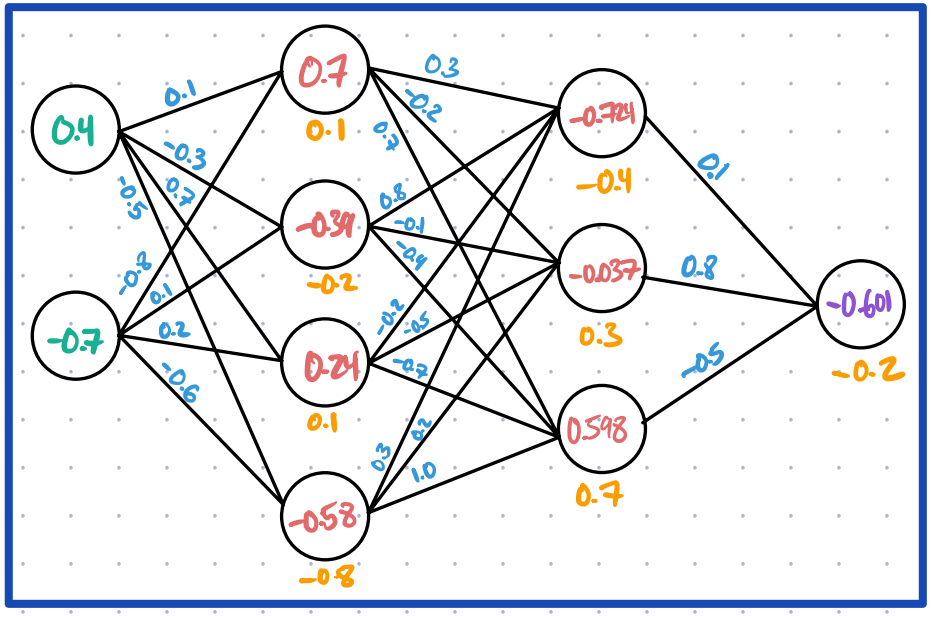
What about activation function?

Suppose our activation function is σ.

We'll use the notation

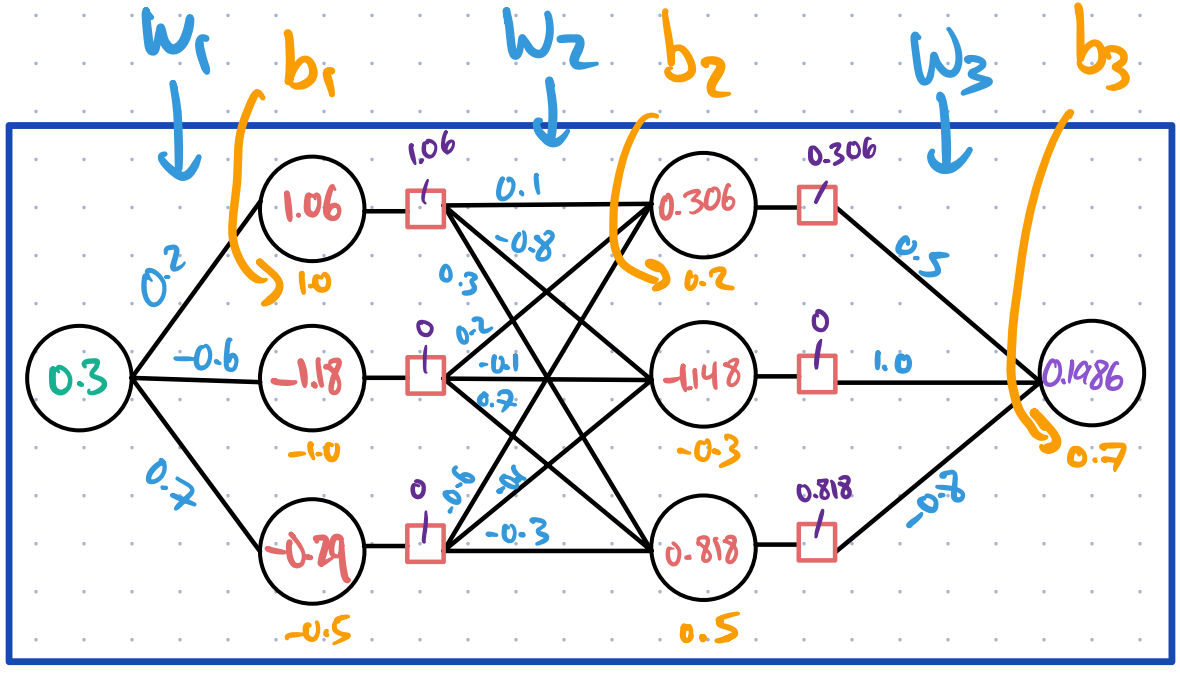
$$\sigma\left(\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_k \end{bmatrix}\right) = \begin{bmatrix} \sigma(v_1) \\ \sigma(v_2) \\ \vdots \\ \sigma(v_k) \end{bmatrix}.$$

$$\sigma(Wx_1 + b) = x_2$$



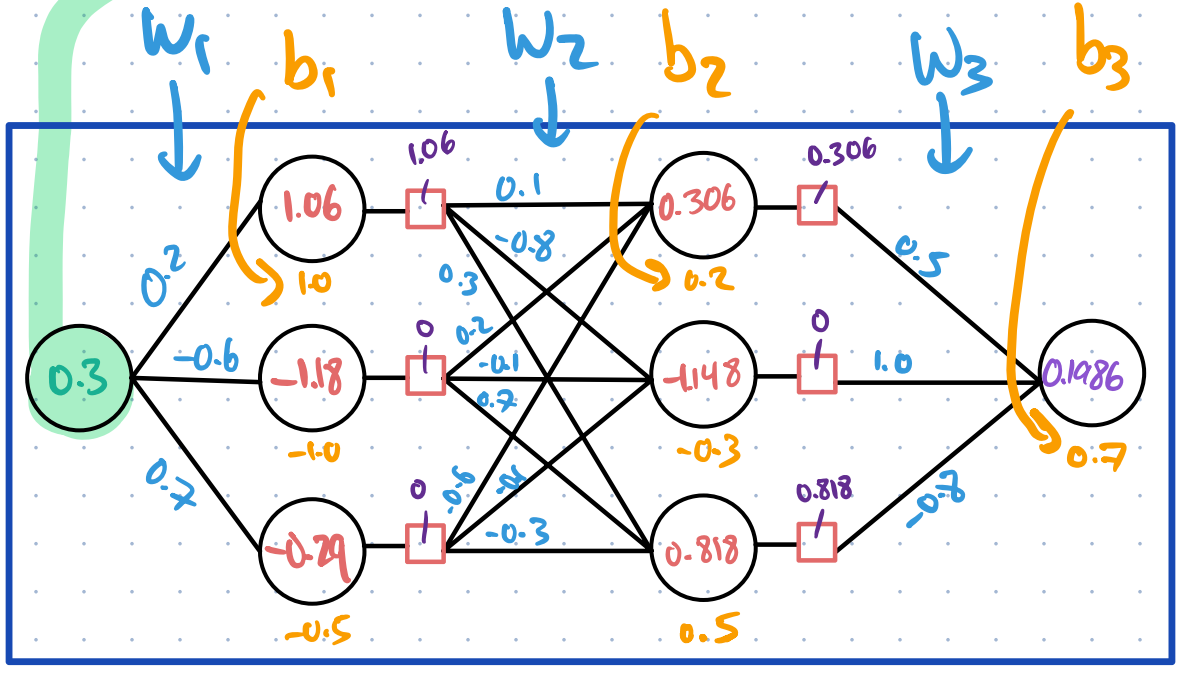
So this neural network, with activation function σ for each layer, is the function:

$$f(x) = W_3\sigma(W_2\sigma(W_1[x] + b_1) + b_2) + b_3$$



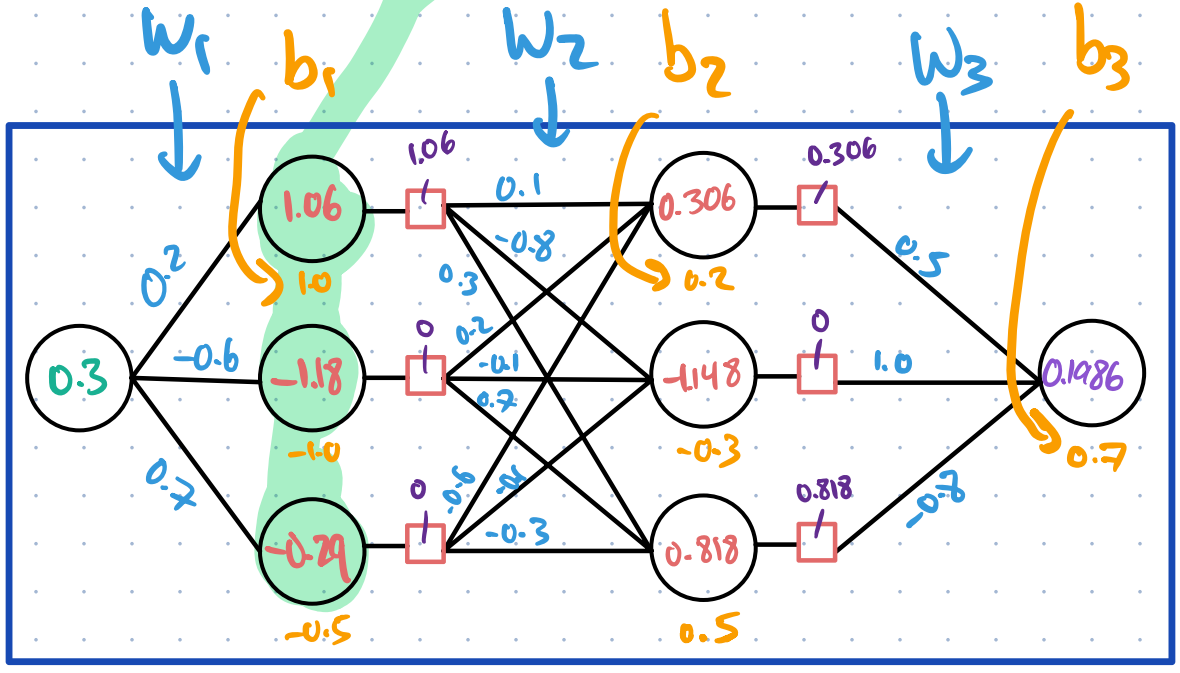
So this neural network, with activation function σ for each layer, is the function:

$$f(x) = \sigma(W_3\sigma(W_2\sigma(W_1[x] + b_1) + b_2) + b_3)$$



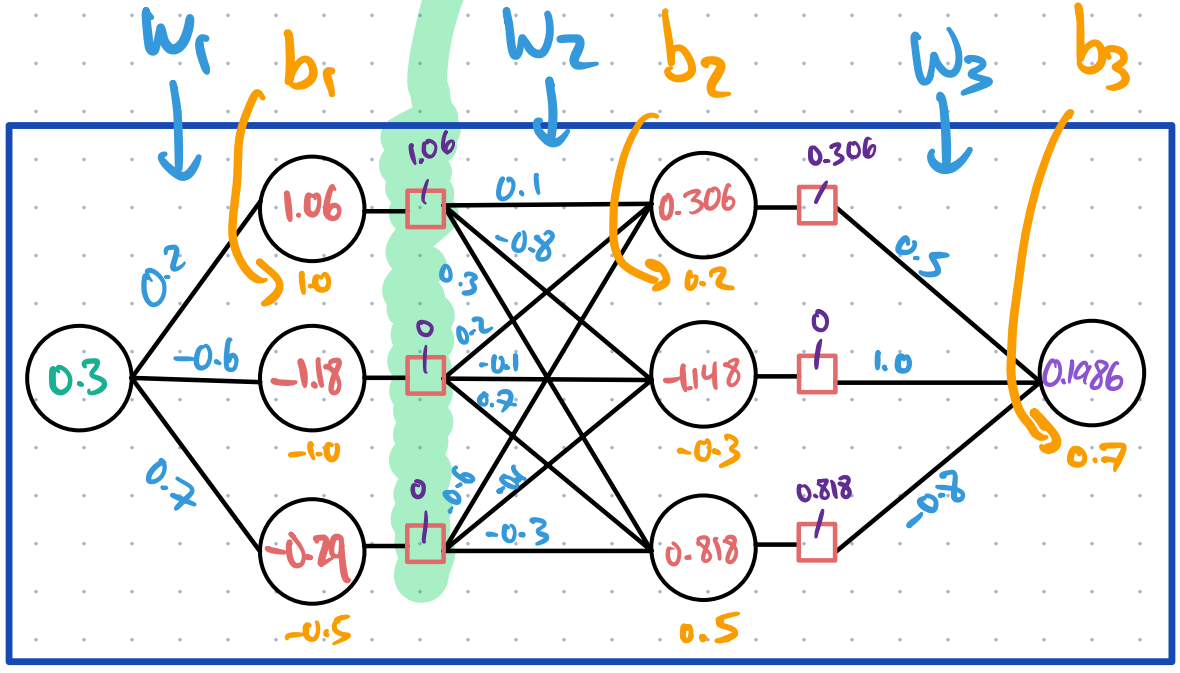
So this neural network, with activation function σ for each layer, is the function:

$$f(x) = \sigma(W_3\sigma(W_2\sigma(W_1[x] + b_1) + b_2) + b_3)$$



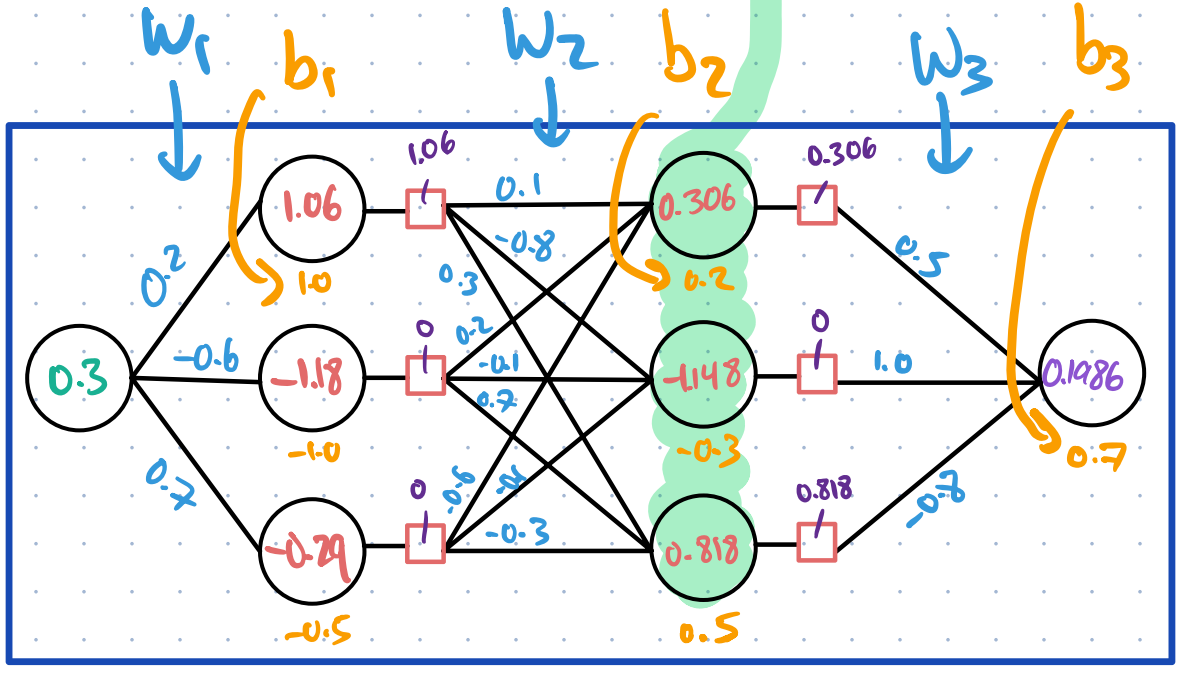
So this neural network, with activation function σ for each layer, is the function:

$$f(x) = \sigma(W_3\sigma(W_2\sigma(W_1[x] + b_1) + b_2) + b_3)$$



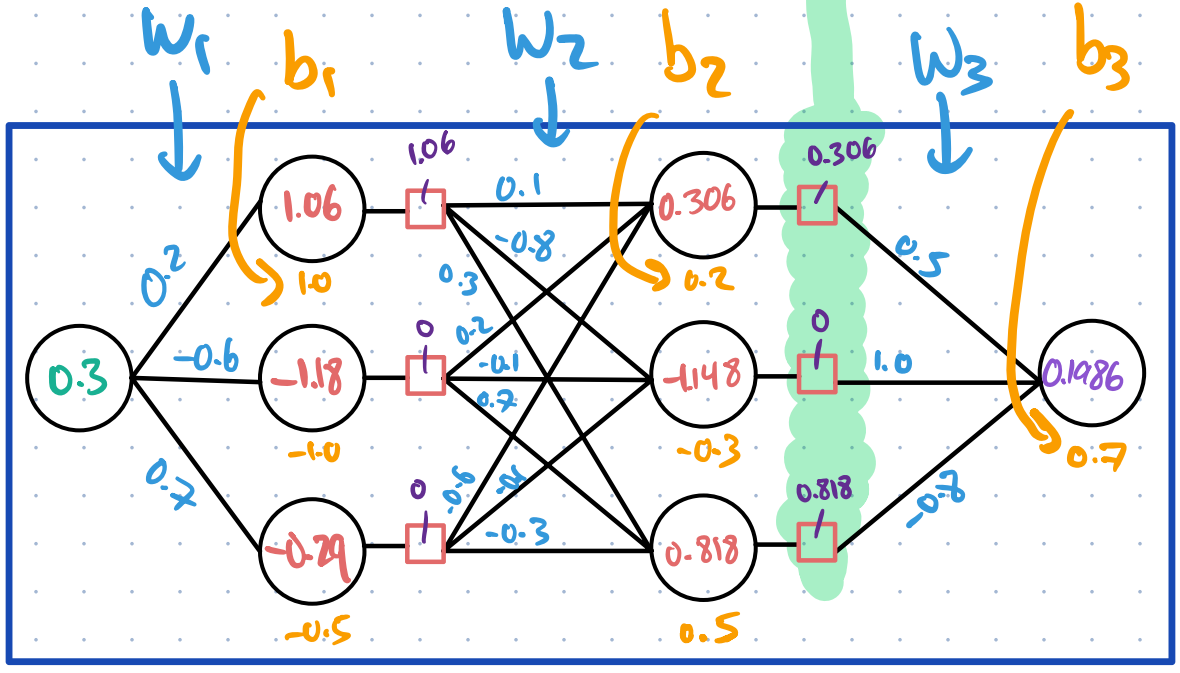
So this neural network, with activation function σ for each layer, is the function:

$$f(x) = \sigma(W_3 \sigma(W_2 \sigma(W_1[x] + b_1) + b_2) + b_3)$$



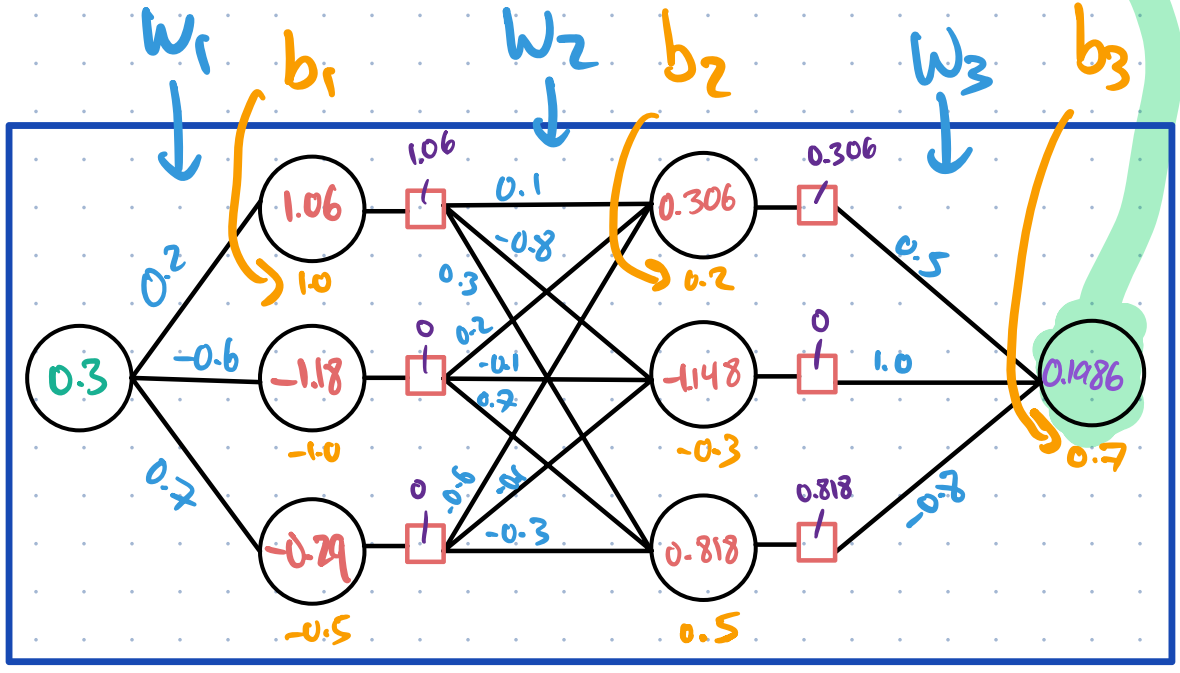
So this neural network, with activation function σ for each layer, is the function:

$$f(x) = \sigma(W_3 \sigma(W_2 \sigma(W_1[x] + b_1) + b_2) + b_3)$$



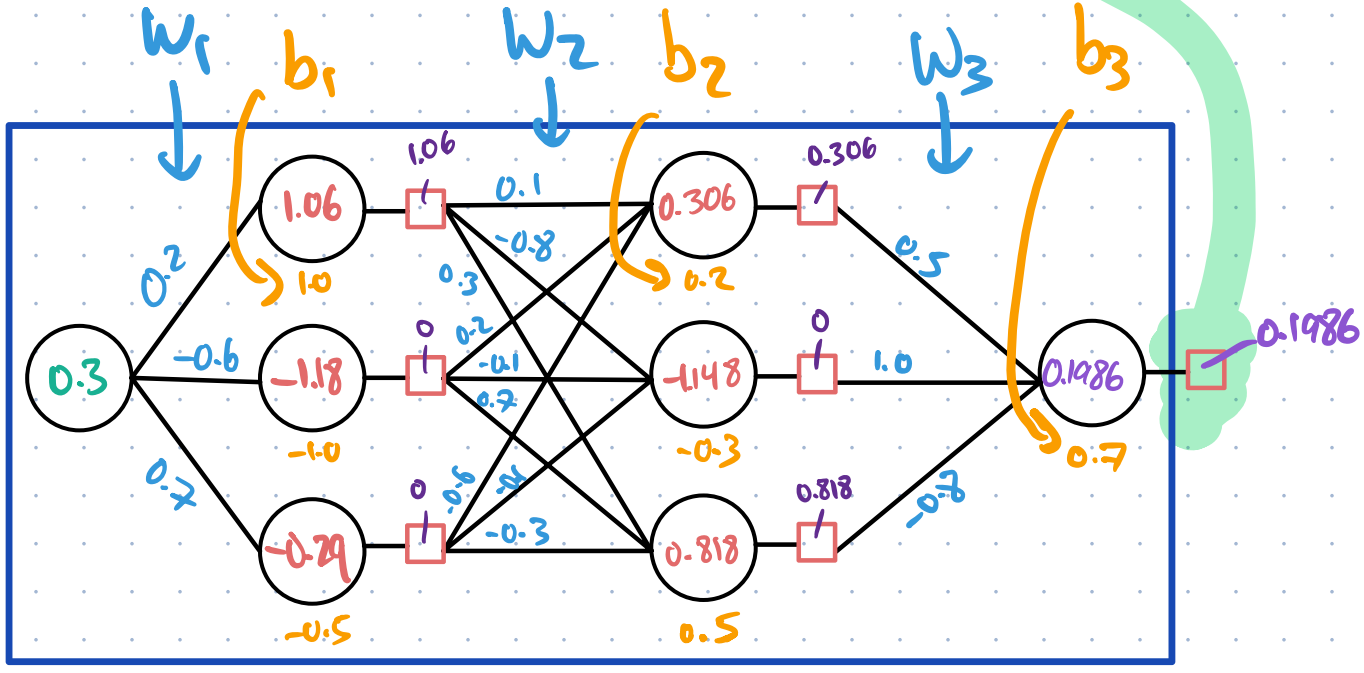
So this neural network, with activation function σ for each layer, is the function:

$$f(x) = \sigma(W_3\sigma(W_2\sigma(W_1[x] + b_1) + b_2) + b_3)$$



So this neural network, with activation function σ for each layer, is the function:

$$f(x) = \sigma(W_3\sigma(W_2\sigma(W_1[x] + b_1) + b_2) + b_3)$$



* Sometimes the output layer has a different activation function, or no AF.

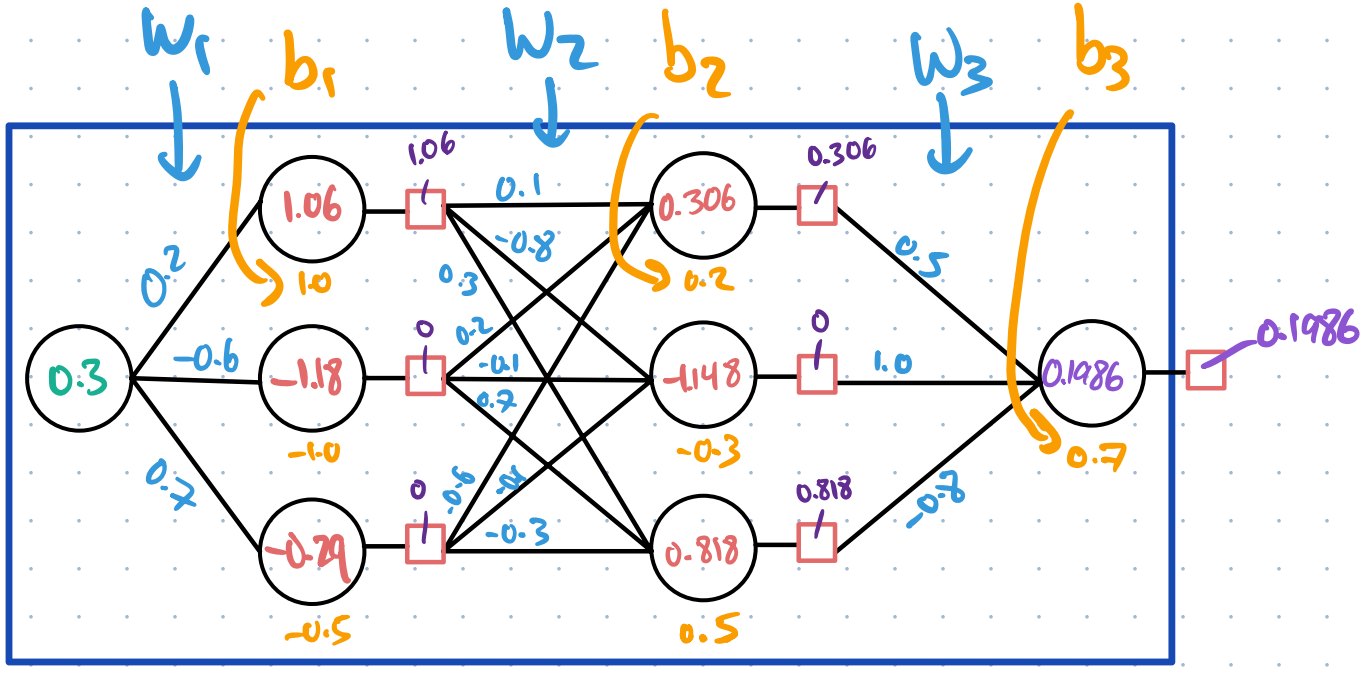
$$W_1 = \begin{bmatrix} 0.2 \\ -0.6 \\ 0.7 \end{bmatrix} \quad W_2 = \begin{bmatrix} 0.1, & 0.2, & -0.6 \\ -0.8 & -0.1 & -0.1 \\ 0.3 & 0.7 & -0.3 \end{bmatrix} \quad W_3 = \begin{bmatrix} 0.5 & 1.0 & -0.8 \end{bmatrix}$$

each column of a W is the weights coming out of one neuron

$$b_1 = \begin{bmatrix} 1.0 \\ -1.0 \\ -0.5 \end{bmatrix}, \quad b_2 = \begin{bmatrix} 0.2 \\ -0.3 \\ 0.5 \end{bmatrix}, \quad b_3 = \begin{bmatrix} 0.7 \end{bmatrix}$$

$$\sigma(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} = \max(0, x)$$

ReLU



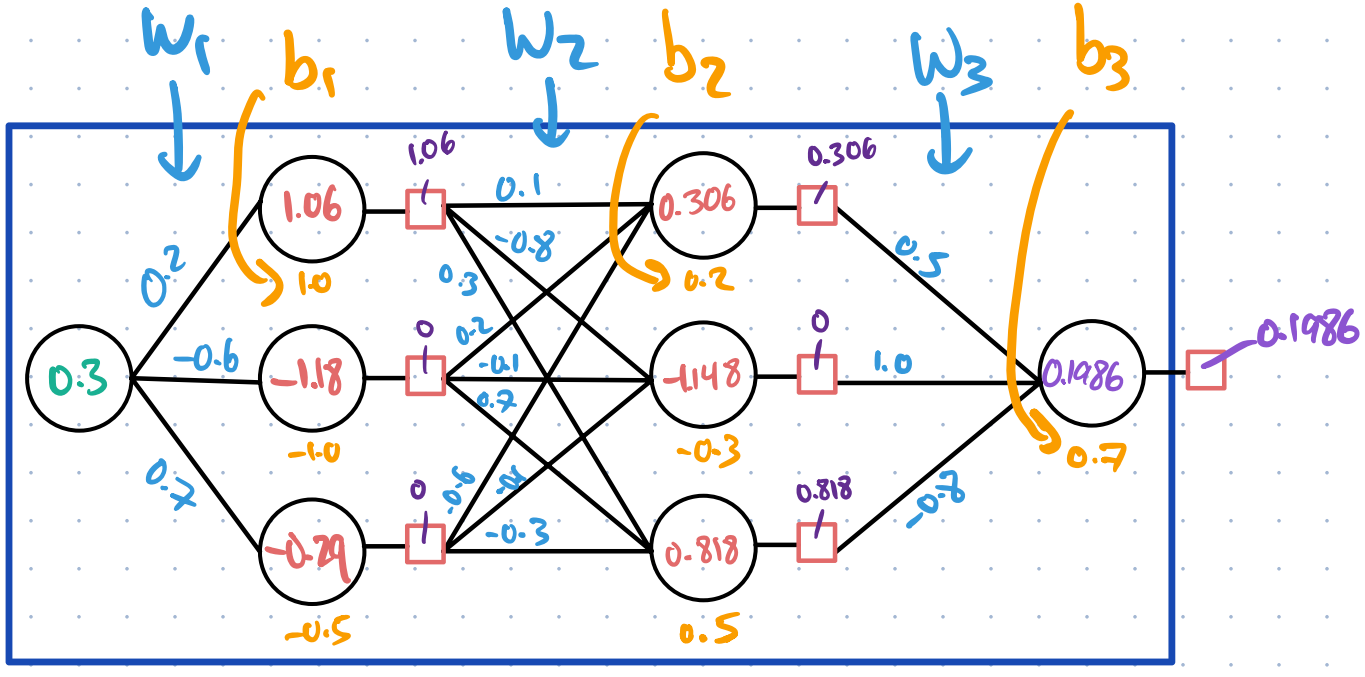
$$W_1 = \begin{bmatrix} 0.2 \\ -0.6 \\ 0.7 \end{bmatrix} \quad W_2 = \begin{bmatrix} 0.1, & 0.2, & -0.6 \\ -0.8 & -0.1 & -0.1 \\ 0.3 & 0.7 & -0.3 \end{bmatrix} \quad W_3 = \begin{bmatrix} 0.5 & 1.0 & -0.8 \end{bmatrix}$$

$$b_1 = \begin{bmatrix} 1.0 \\ -1.0 \\ -0.5 \end{bmatrix}, \quad b_2 = \begin{bmatrix} 0.2 \\ -0.3 \\ 0.5 \end{bmatrix}, \quad b_3 = [0.7]$$

each column of a W is the weights coming out of one neuron

$$f([0.3]) = \max(0, [0.5 \ 1.0 \ -0.8] \max(0, \begin{bmatrix} 0.1, & 0.2, & -0.6 \\ -0.8 & -0.1 & -0.1 \\ 0.3 & 0.7 & -0.3 \end{bmatrix} \max(0, \begin{bmatrix} 0.2 \\ -0.6 \\ 0.7 \end{bmatrix} [0.3] + \begin{bmatrix} 1.0 \\ -1.0 \\ -0.5 \end{bmatrix}) + \begin{bmatrix} 0.2 \\ -0.3 \\ 0.5 \end{bmatrix}) + [0.7])$$

$$= 0.1986$$



Now you understand exactly what a neural network is:

a fancy way to define a function

But not yet:

* How to find a good one (training)
* How NNs accomplish different tasks (what good is a fancy function?)

Next lecture: Coding our first NN, and batching the forward pass

Topic 15 - Implementing our first NN and Batching

* Goal: Write code to perform the forward pass of a NN, with activation functions

The purpose of writing this code is to understand the topic.

If you needed a fast NN for research, you would probably use a Python library like PyTorch

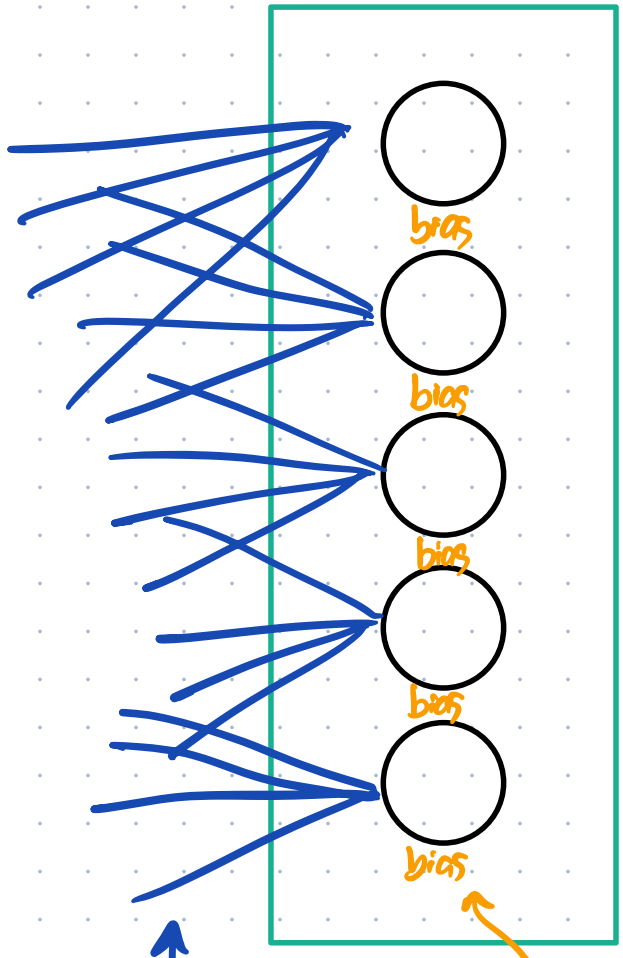
Structure:

* Object Oriented
* Objects:

without activation

- Layer
  - knows the weights that feed into it from the previous layer
  - knows the biases of its neurons
  - can take input data and compute output data
- ActivationFunction
  - can take input data and compute output data

Layer:



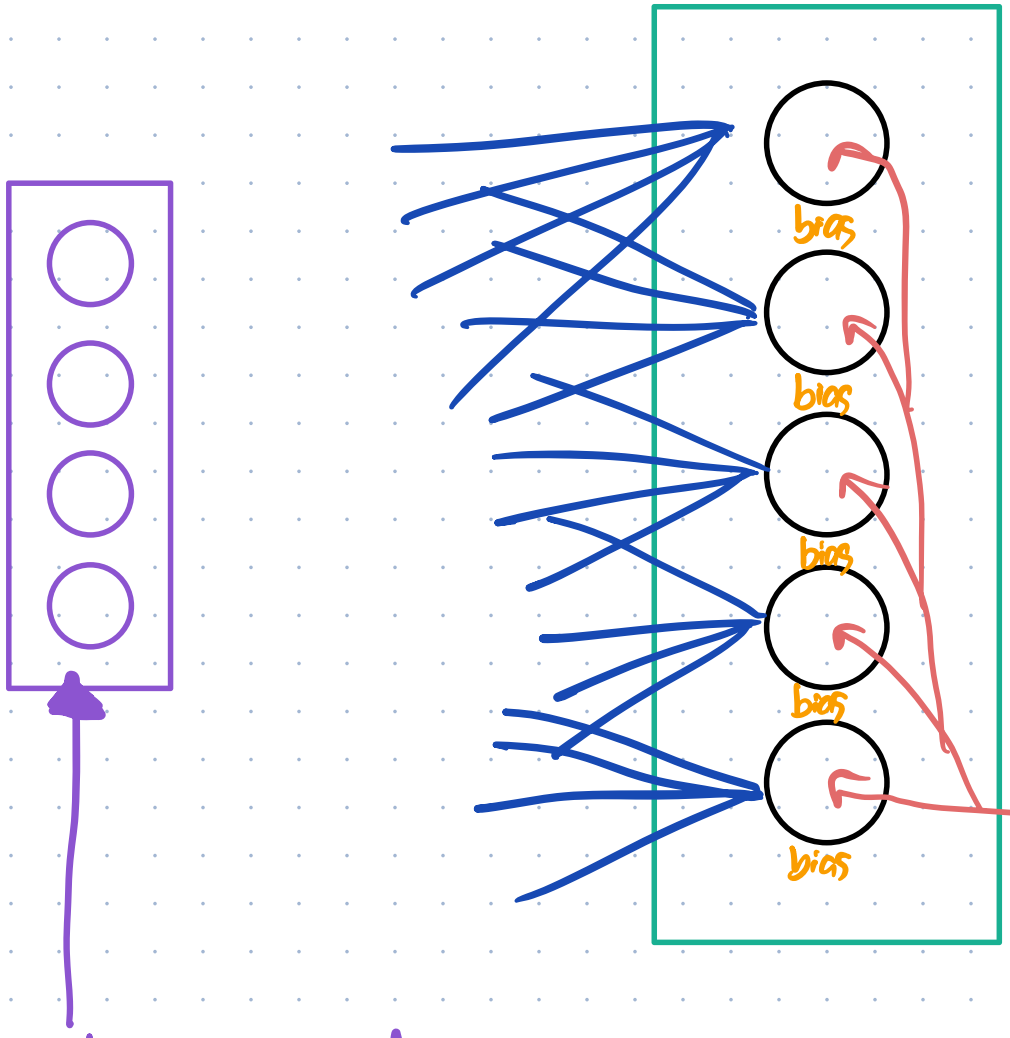
matrix for the weights

vector for the biases

knows the biases

knows the weights on these edges

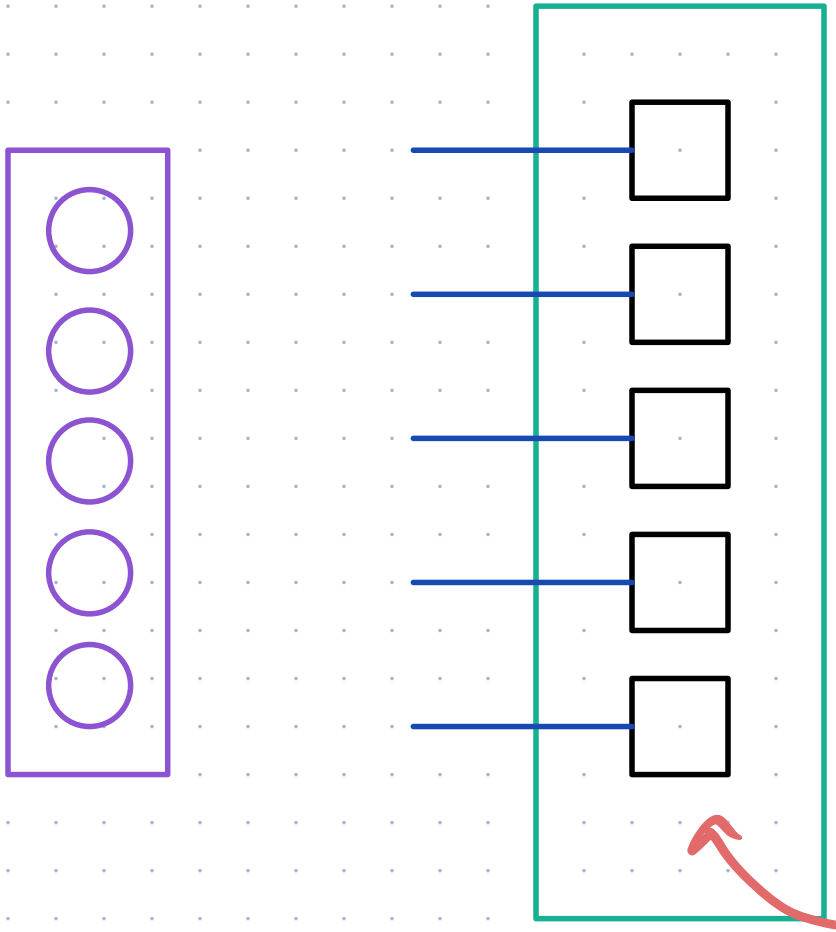
Layer:



input: value of previous layer's neurons
(actually, the activation function of those neurons)

output: value of this layer's neurons (pre-activation)

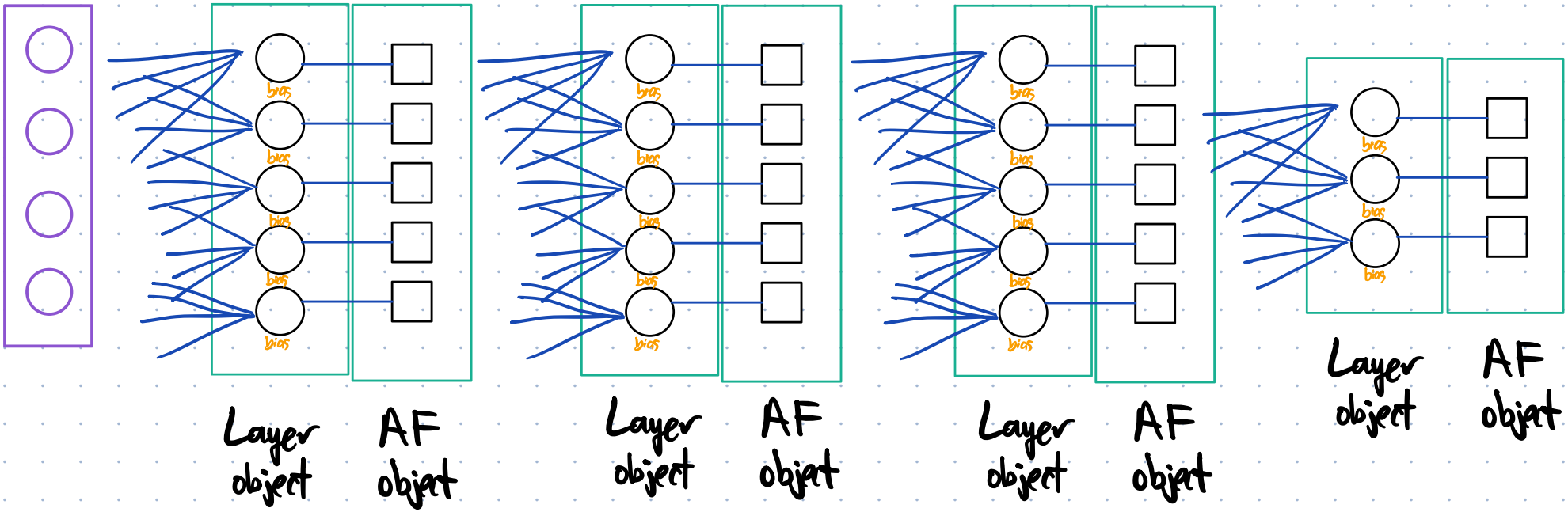
Activation Function classes (one for each activation function)



inputs: values of neurons we're activating

outputs: activated values

Then we can chain instances of these objects together to make a full NN.



Coding time:

First: the "numpy" Python library

* Jupyter notebook demo