Scientific Computing

April 25, 2025

Announcements

→ Homework 6 due Friday, May 2, 11:59pm

→ Final exam is take-home only
assigned Fri, May 2
due Fri, May 9

Today

→ Coding NN and Batching

→ Loss Functions

Office Hours:

Mon + Fri

9:30am - 10:30am

Cudahy 307

Structure:

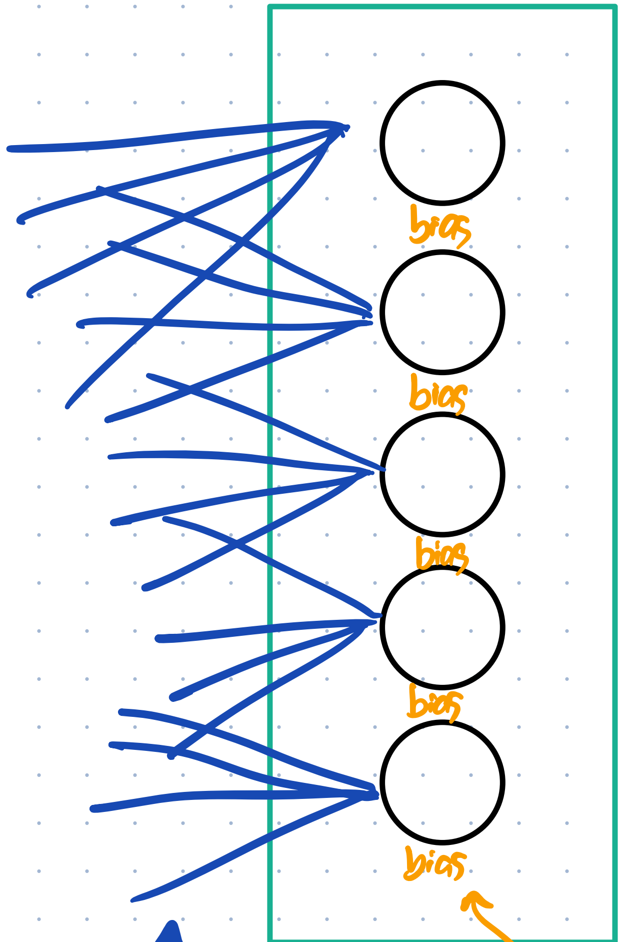
* Object Oriented

* Objects:

    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

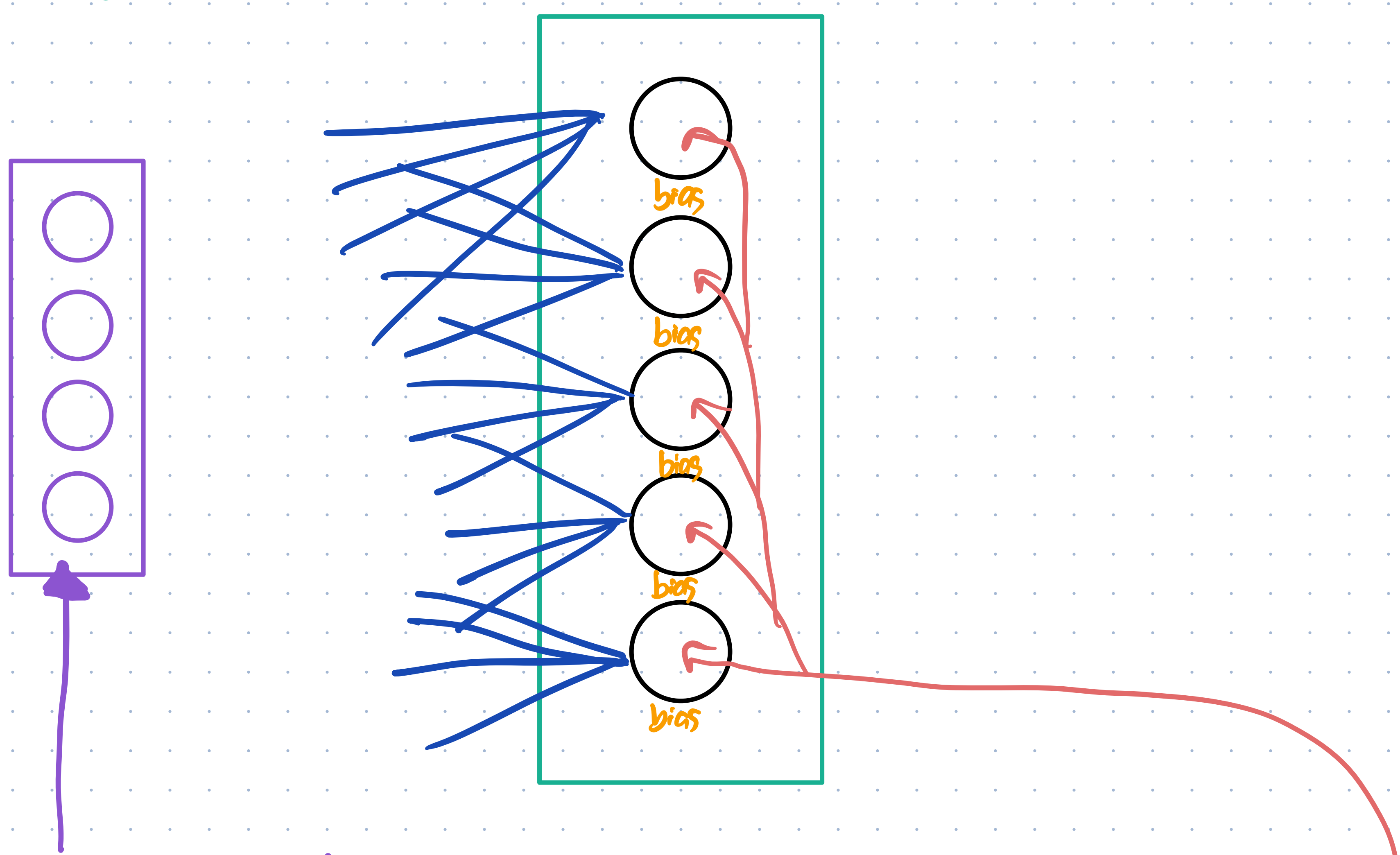
Layer:



knows the biases (vector)

knows the weights on these edges
(matrix)

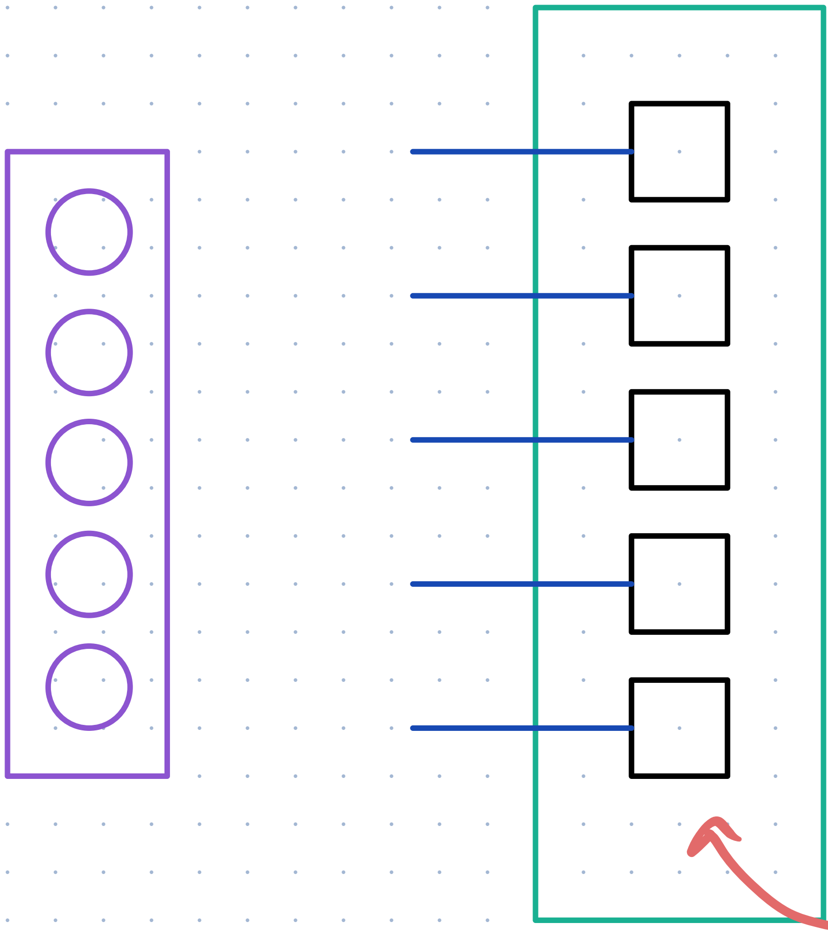
Layer:



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

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

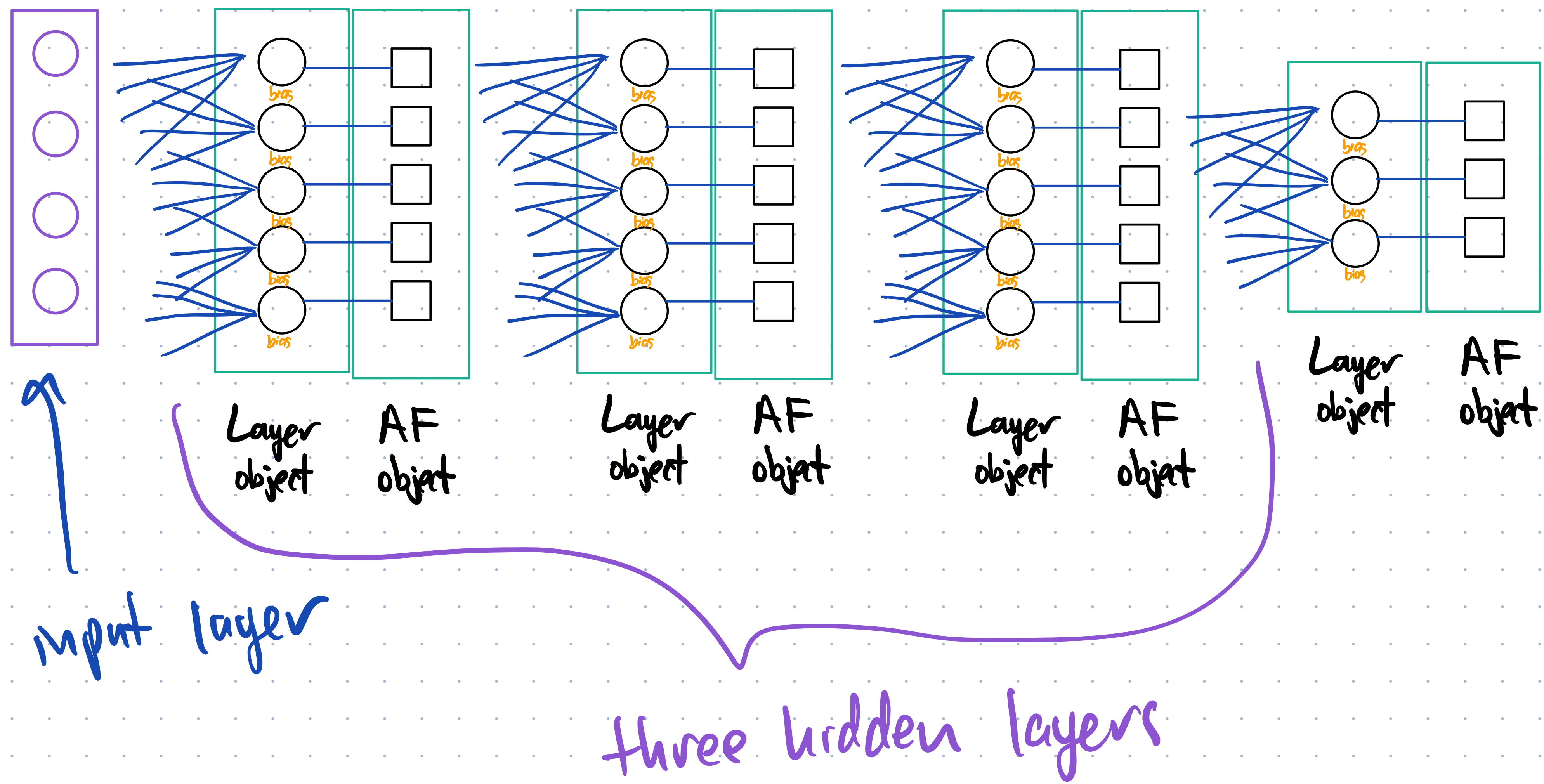
Activation Function classes (one for each activation function)



$\max(0, v)$

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

Coding time:

First: the "numpy" Python library

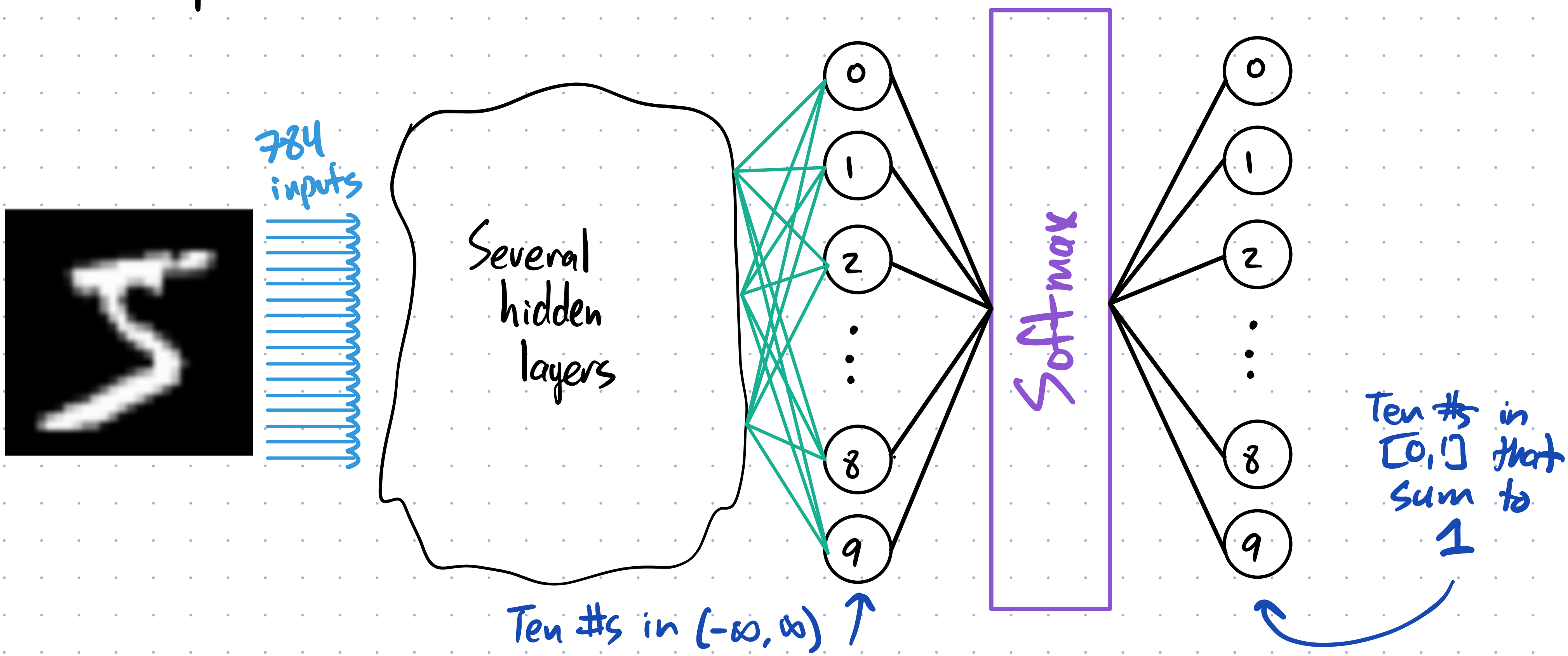
* Jupyter notebook demo

Next: Coding our first layers together from scratch.

(very heavy inspiration from the "Neural Networks from Scratch" book!)

New Activation Function: Softmax

* Turns a vector of #s into a probability distribution
  (a vector of #s in [0,1] that sums to 1)
* Useful for the output layer in a classification problem.



Unlike our other activation functions, Softmax works on the whole vector at once, not one value at a time individually.

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} e^{x_1}/S \\ e^{x_2}/S \\ e^{x_3}/S \\ \vdots \\ e^{x_k}/S \end{bmatrix}$$

where

$$S = \sum_{i=1}^{k} e^{x_i}$$

Obvious these add up to 1 because the denominator is their sum.

Obviously > 0 because $e^x > 0$.

Ex:

$$\begin{array}{c} 0\\ 1\\ 2\\ 3\\ 4\\ 5\\ 6\\ 7\\ 8\\ 9 \end{array} \begin{bmatrix} -0.8\\ -0.7\\ 0.3\\ 0.1\\ 0.8\\ 0.5\\ 0.2\\ -0.3\\ 0\\ 0.6 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} 0.037\\ 0.041\\ 0.111\\ 0.091\\ 0.184\\ 0.136\\ 0.101\\ 0.061\\ 0.082\\ 0.150 \end{bmatrix}$$

$$\sum_{i=0}^{9} e^{x_i} \approx 12.06$$

* 18.4% chance the digit is a 4

Numeric Stability (because e^x gets big!)

Let $m = \max(\vec{x})$

Then do $\dfrac{e^{x_i - m}}{\sum_{j=1}^{k} e^{x_j - m}}$ all components ≤ 0

$$\frac{e^{x_i - m}}{\sum_{j=1}^{k} e^{x_j - m}} = \frac{\frac{e^{x_i}}{e^m}}{\sum_{j=1}^{k} \frac{e^{x_j}}{e^m}} = \frac{\cancel{\frac{1}{e^m}} \cdot e^{x_i}}{\cancel{\frac{1}{e^m}} \cdot \sum_{j=1}^{k} e^{x_j}}$$

* Let's add this as a new Activation Function class

Batching

So far we've done the linear algebra and coding for feeding forward one input vector at a time.

one input vector

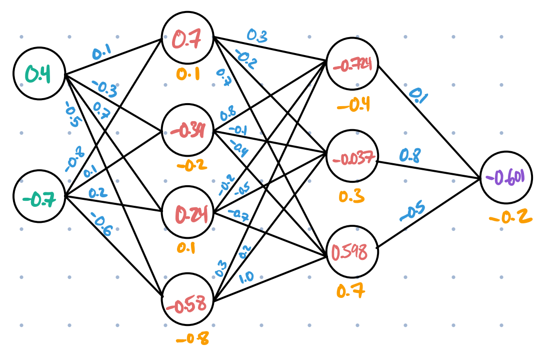
(ignoring activation functions!)

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

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

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

$$= \begin{bmatrix} -0.601 \end{bmatrix}$$



* Numpy does vector calculations faster than individual number calculations

* In the same way, it does matrix calculations faster than one vector at a time!

* We can feed forward <u>many</u> input vectors simultaneously by making one big matrix with many columns.

Let $\begin{bmatrix} \vec{v}_1 & \vec{v}_2 & \vec{v}_3 \end{bmatrix}$ denote the matrix with columns $\vec{v}_1$ $\vec{v}_2$ $\vec{v}_3$.

Fact: $M \cdot \begin{bmatrix} \vec{v}_1 & \vec{v}_2 & \vec{v}_3 \end{bmatrix} = \begin{bmatrix} M\vec{v}_1 & M\vec{v}_2 & M\vec{v}_3 \end{bmatrix}$

$$\begin{bmatrix} 0.1 & -0.8 \\ -0.3 & 0.1 \\ 0.7 & 0.2 \\ -0.5 & -0.6 \end{bmatrix} \underbrace{\begin{bmatrix} 0.4 & -0.1 & 0.8 \\ -0.7 & 0.2 & 0.6 \end{bmatrix}}_{\text{3 columns}} + \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ -0.2 & -0.2 & -0.2 \\ 0.1 & 0.1 & 0.1 \\ -0.8 & -0.8 & -0.8 \end{bmatrix} = \begin{bmatrix} 0.7 & -0.7 & -0.3 \\ -0.39 & -0.15 & -0.38 \\ 0.24 & 0.07 & 0.78 \\ -0.58 & -0.87 & -1.56 \end{bmatrix}$$

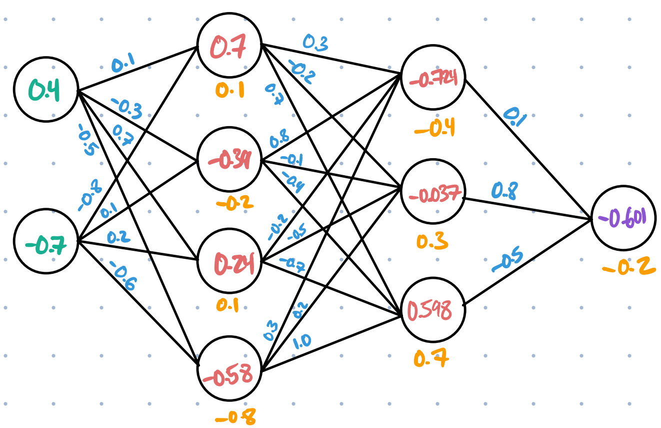
$$\begin{bmatrix} 0.3 & 0.8 & -0.2 & 0.3 \\ -0.2 & -0.1 & -0.5 & 0.2 \\ 0.7 & -0.4 & -0.7 & 1.0 \end{bmatrix} \begin{bmatrix} 0.7 & -0.7 & -0.3 \\ -0.39 & -0.15 & -0.38 \\ 0.24 & 0.07 & 0.78 \\ -0.58 & -0.87 & -1.56 \end{bmatrix} + \begin{bmatrix} -0.4 & -0.4 & -0.4 \\ 0.3 & 0.3 & 0.3 \\ 0.7 & 0.7 & 0.7 \end{bmatrix}$$

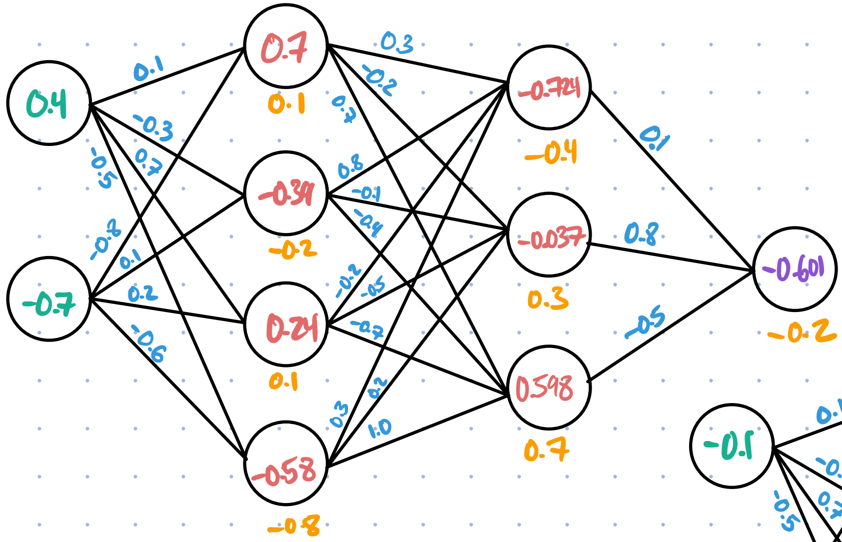
$$= \begin{bmatrix} -0.724 & -0.816 & -1.418 \\ -0.037 & 0.12 & -0.304 \\ 0.598 & -0.208 & -1.464 \end{bmatrix}$$

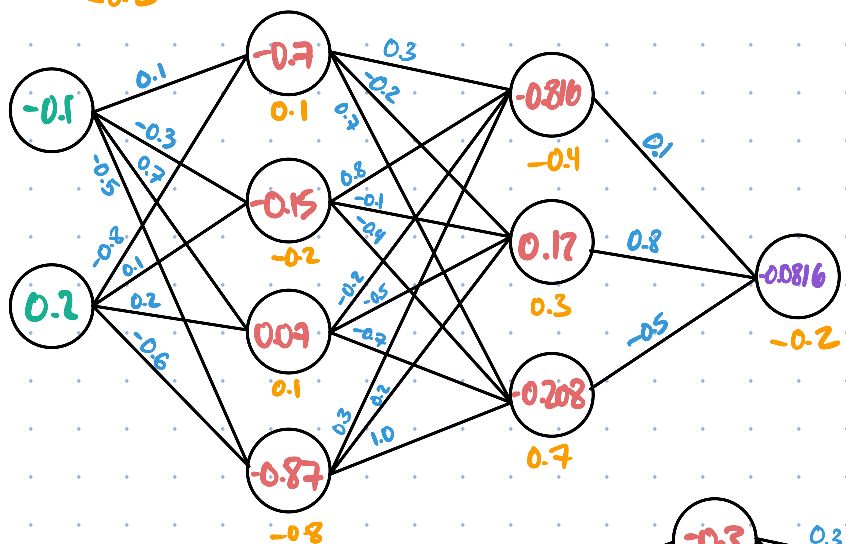
$$\begin{bmatrix} 0.1 & 0.8 & -0.5 \end{bmatrix} \begin{bmatrix} -0.724 & -0.816 & -1.418 \\ -0.037 & 0.12 & -0.304 \\ 0.598 & -0.208 & -1.464 \end{bmatrix} + \begin{bmatrix} -0.2 & -0.2 & -0.2 \end{bmatrix} = \begin{bmatrix} -0.601 & -0.0816 & 0.147 \end{bmatrix}$$

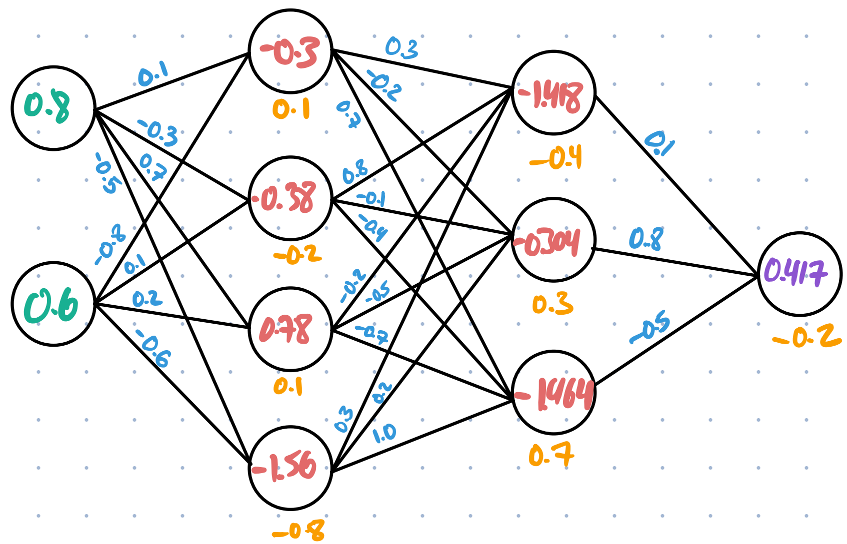
Batch of 3 inputs

(ignoring activation functions!)









Three inputs fed forward all in one calculation!

Could pass all 60,000 MNIST digits through in one calculation!

* Let's update our code to support batching.

* Let's update our code to support batching.

* Big speed update!

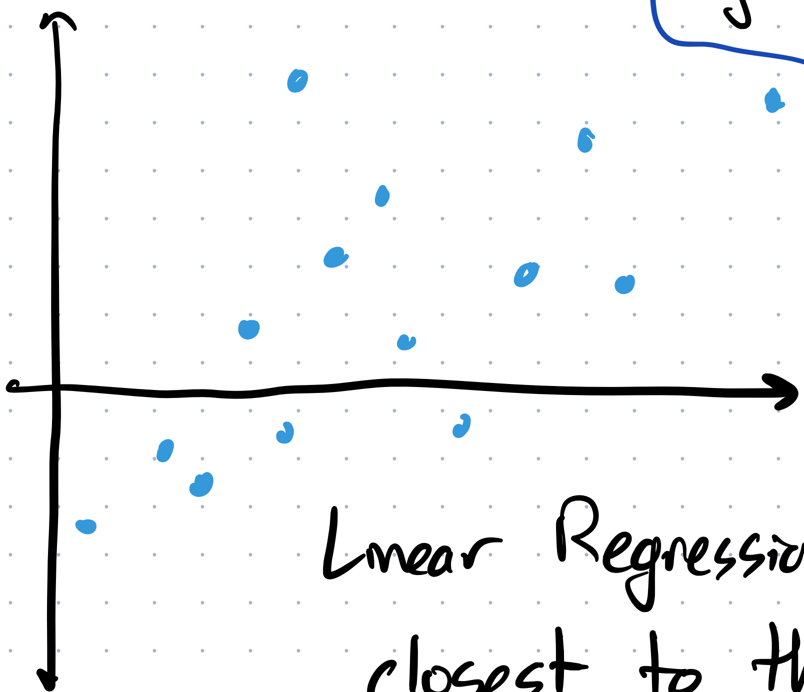
* Demo: - Training on MNIST in browser

- Use in our new classes to predict digits

Topic 15 - Loss Functions

Remember our analogy with Linear Regression.

x: time since Jan 1 this year
y: temperature on my outdoor thermometer



We only have sporadic readings.

Linear Regression asks "what line is closest to these points?"

"Closest" means minimizing the sum of

$$([\text{actual } y \text{ value}] - [\text{predicted } y \text{ value}])^2$$

over all known points.

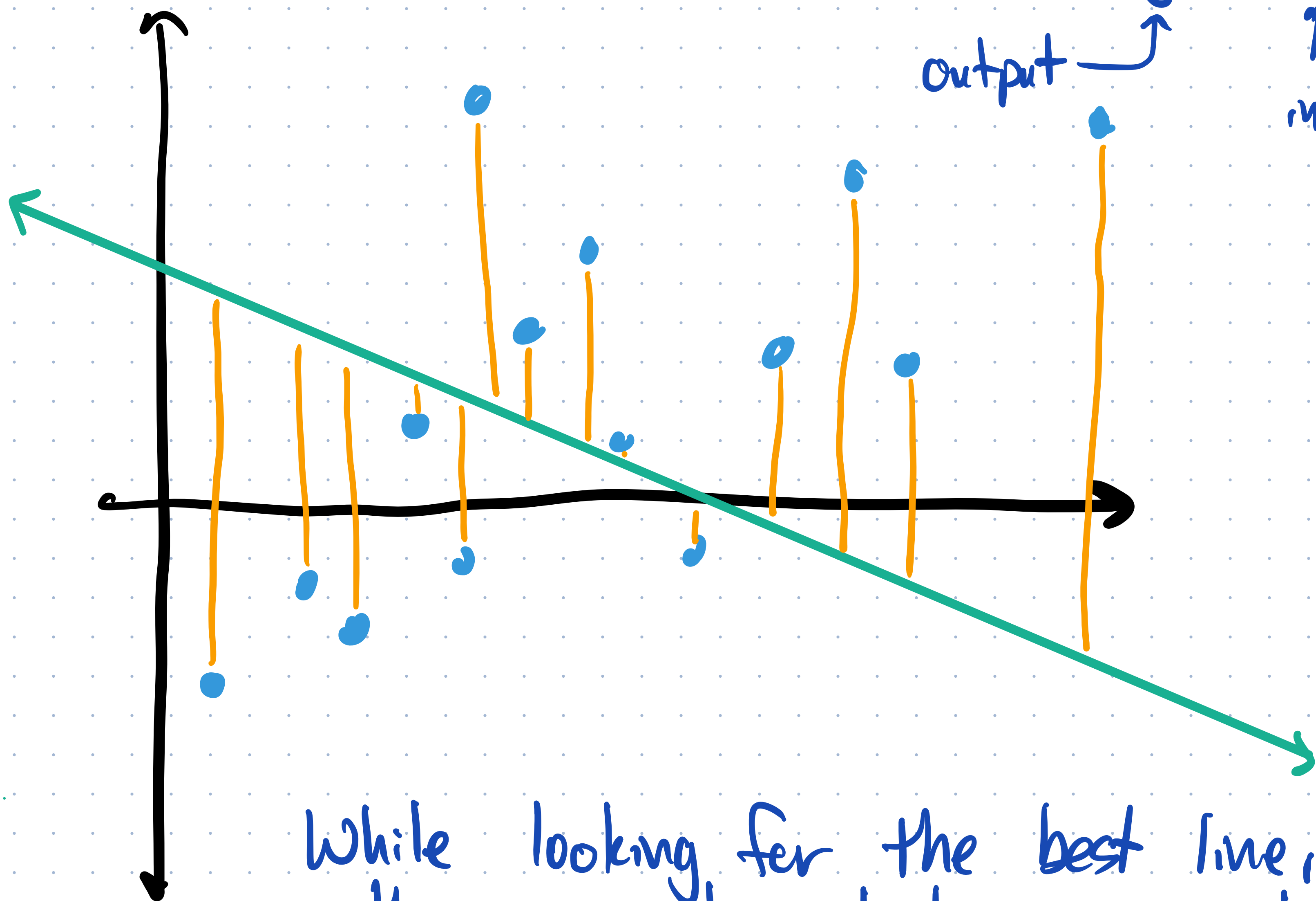
Strong Analogy: Linear Regression

knobs

$$y = mx + b$$

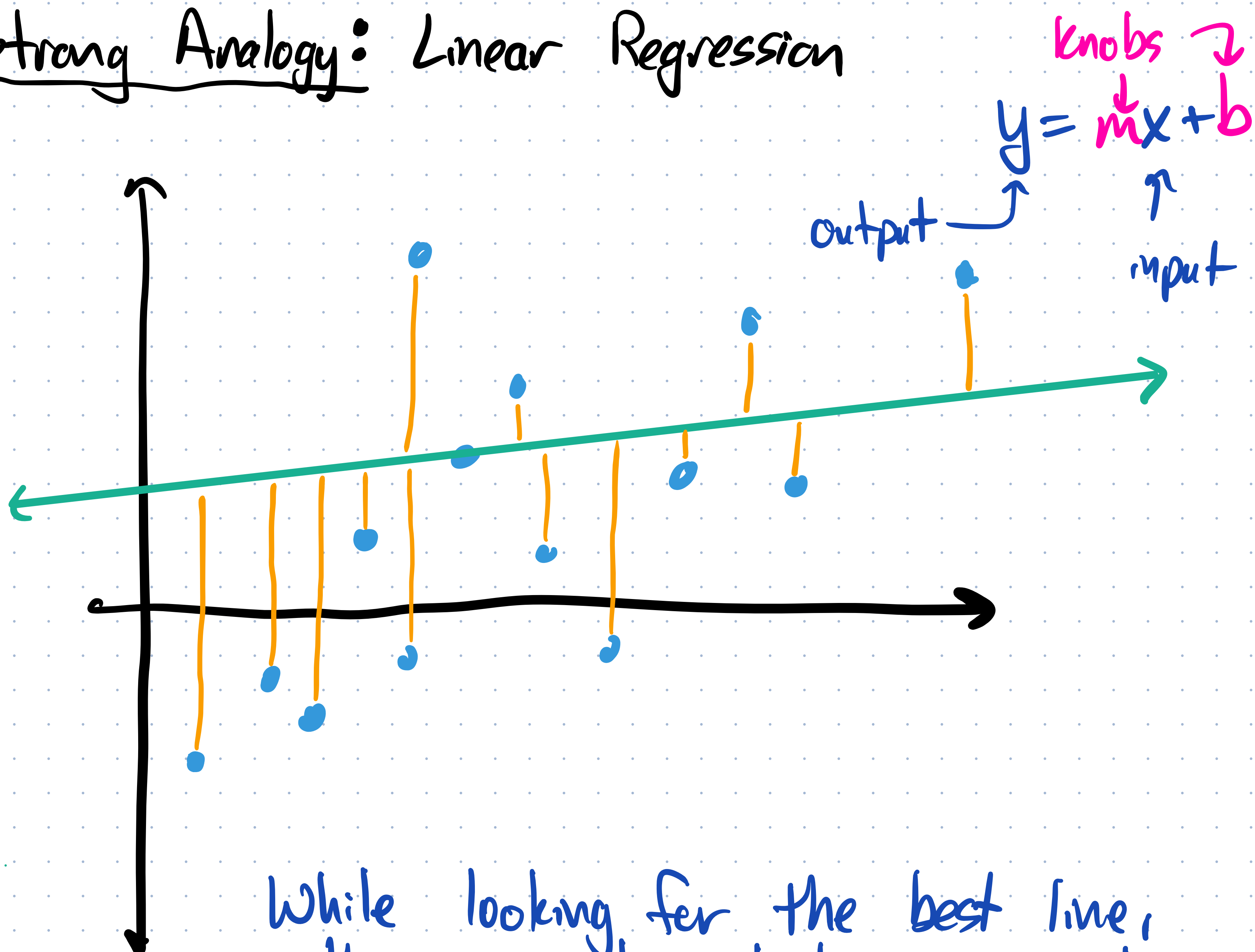
output

input



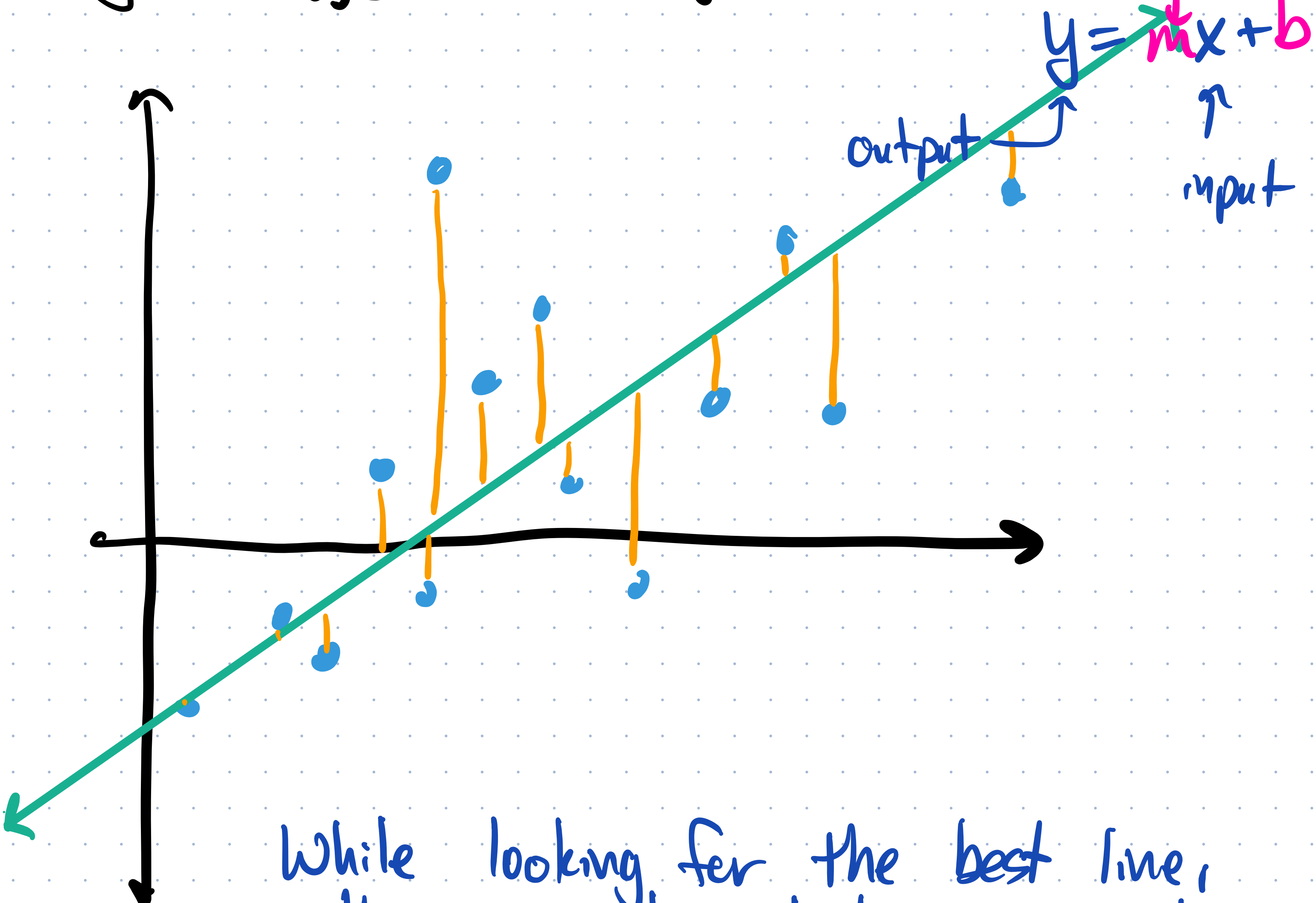
While looking for the best line, there are two knobs we can turn: m and b

Strong Analogy: Linear Regression



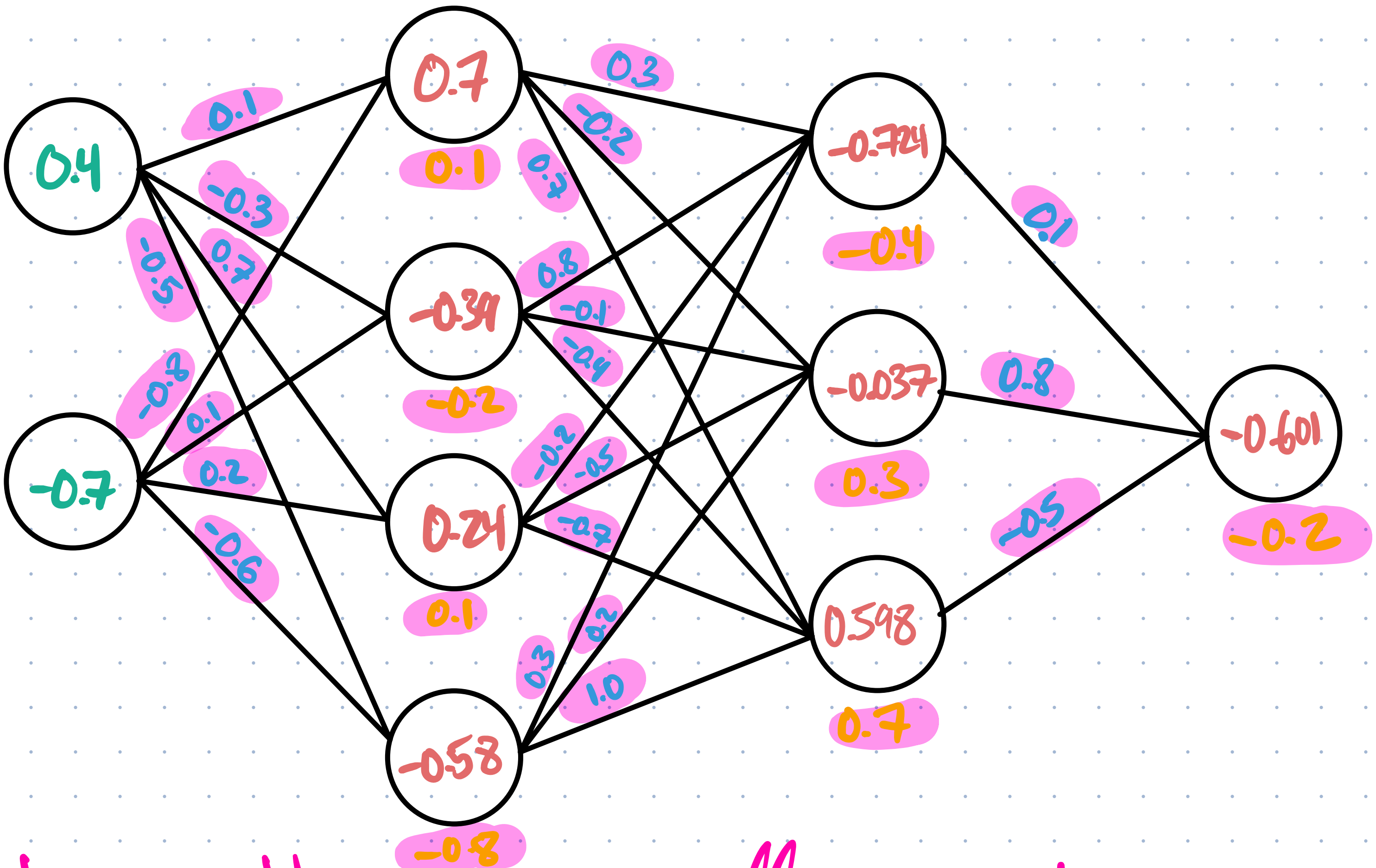
While looking for the best line, there are two knobs we can turn: m and b

Strong Analogy: Linear Regression



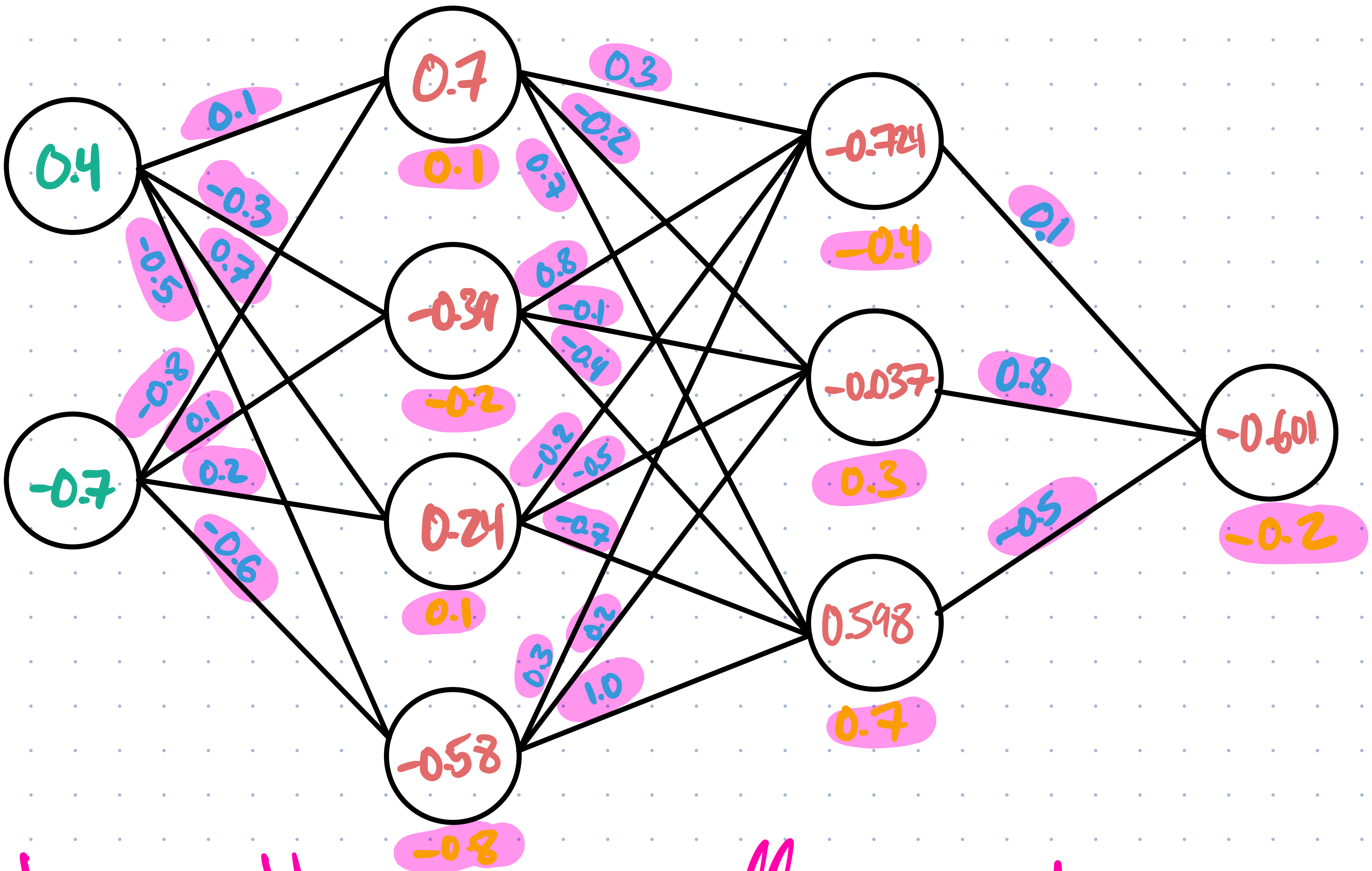
While looking for the best line, there are two knobs we can turn: m and b

Neural Networks are — like lines — just functions with knobs to turn to make them match known data. lots more



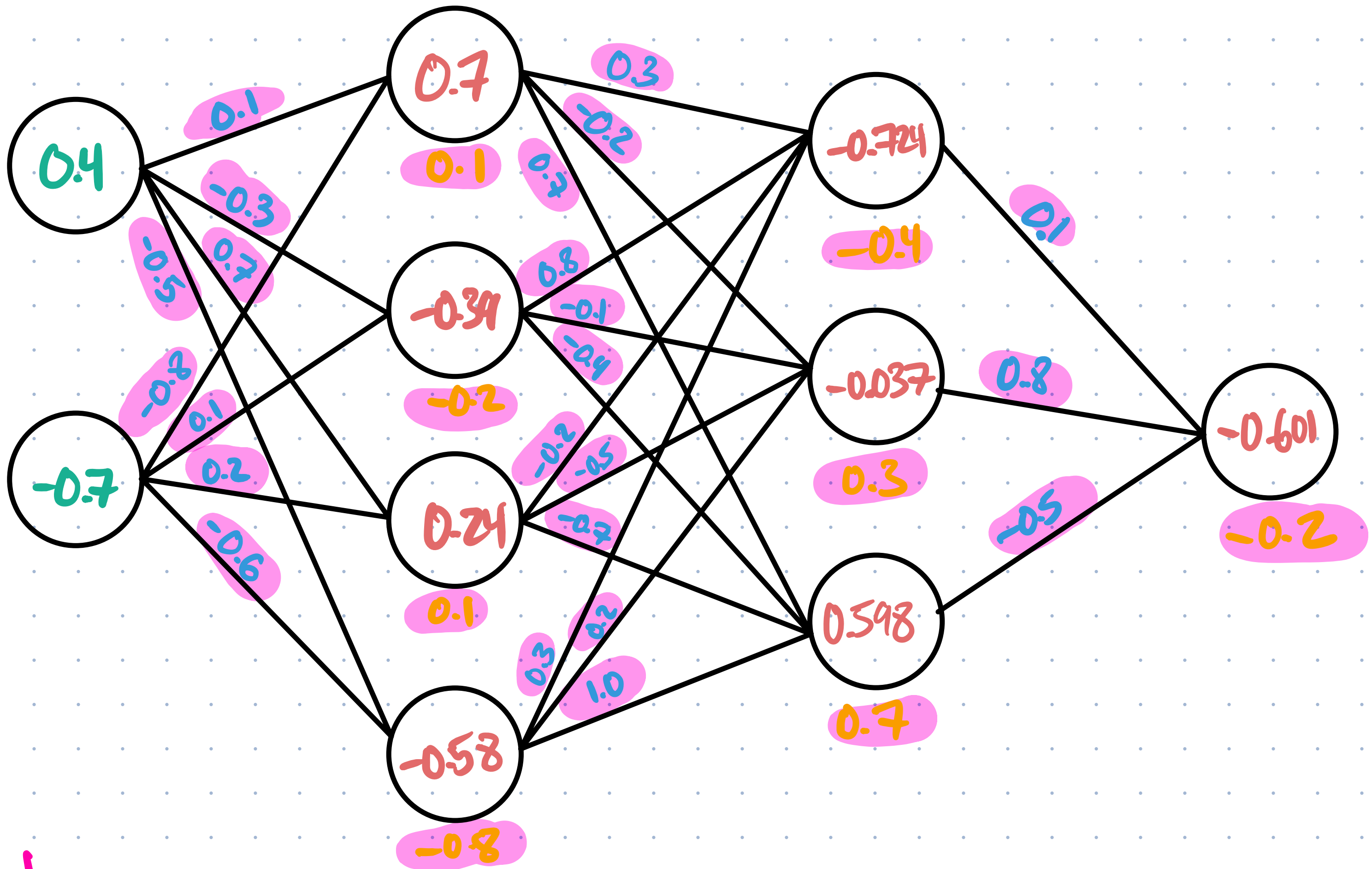
32 knobs in this very small example

Goal: Find the best way to tune these 32 knobs so that when you feed it known input, you get the known output.



32 knobs in this ver small exam le

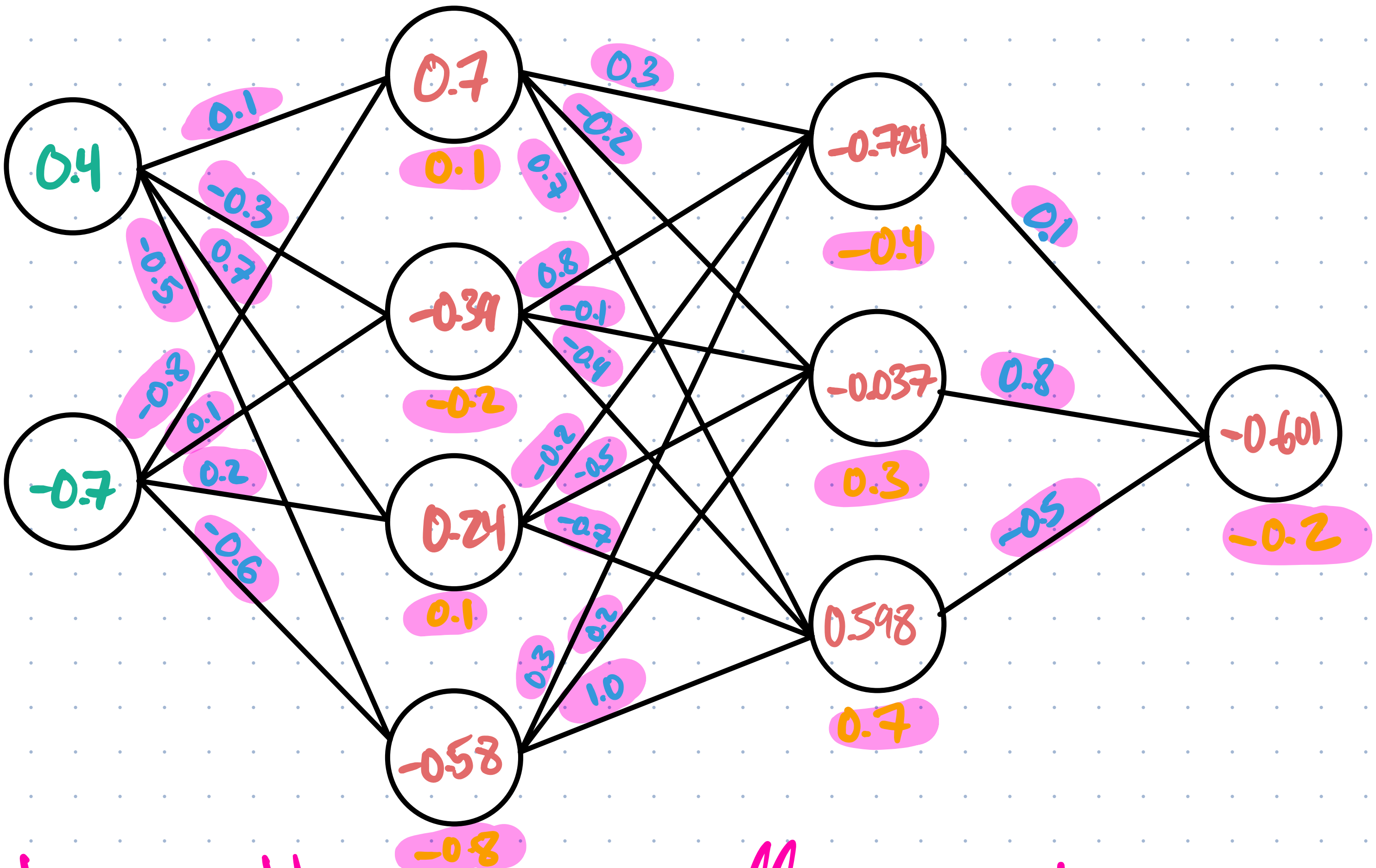
<u>Actual</u> Goal: Find the best way to tune the knobs so that when the NN sees <u>new</u> input data, it produces correct output.



32 knob

How do we measure how <u>good or bad</u> a NN is in terms of matching known data?

[Linear Regression: Mean Squared Error, $\sum_{pts} (\text{actual} - \text{predicted})^2$]



32 knobs in this very small example

Loss

The "score" of a NN relative to particular training data.

Change weights or biases: loss goes up (bad) or down (good)

Loss

Two types of problems we'll use NNs for:

(1) Regression – predict output values based on input values

* Predict home price based on zip code, sq. ft, # bedrooms, # bathrooms, crime rate, school quality, etc
* Predict # of bike rentals based on day, weather, holiday, etc.

(2) Classification – classify input into categories

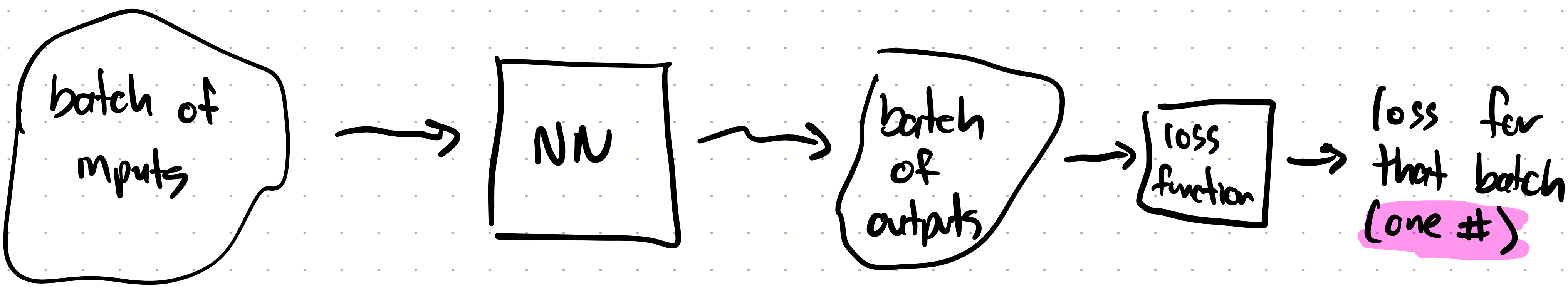
* MNIST digits
* Predict whether a patient has diabetes, prediabetes, or neither, based on health and lifestyle data

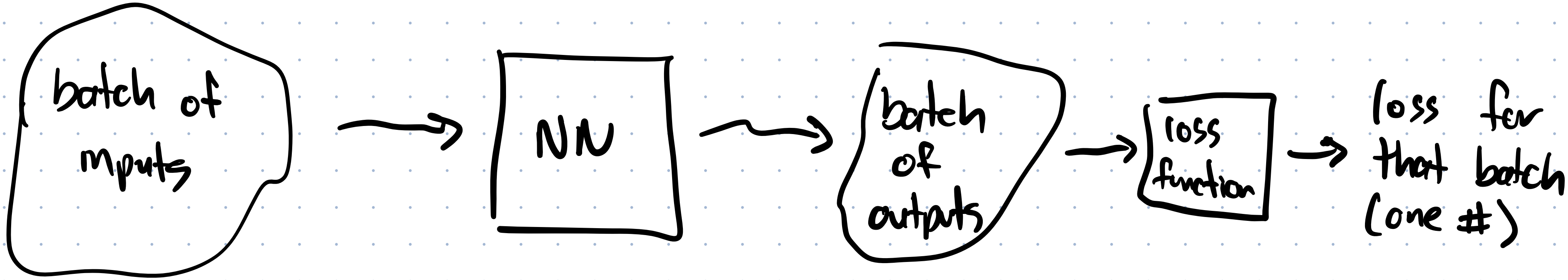
We use different <u>loss functions</u> for each type of problem.

↳ Scoring function for a NN, smaller is better

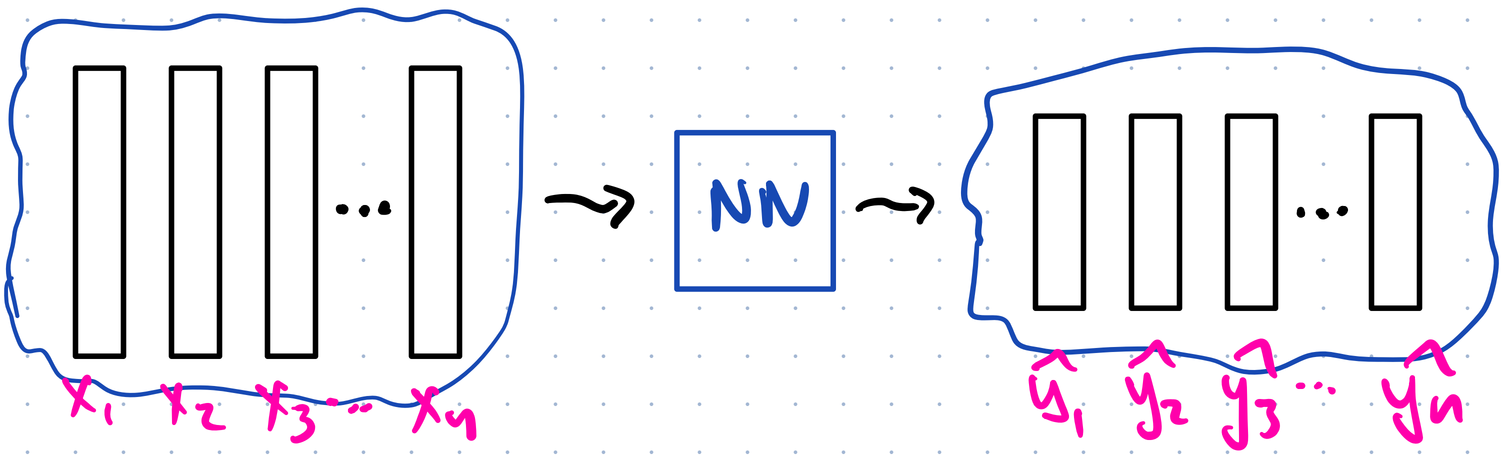
★ Regression.

- Actually, first some notation.
- The loss function is defined <u>not</u> for one input/output pair at a time, but for a whole <u>batch</u> of input/output pairs. [Remember "batching" from our last lecture]





Remember each input is a whole vector (one # per input neuron) and same for each output.



For a batch of size n, we call the input vectors $x_1, x_2, \dots x_n$, the <u>expected</u> output $y_1, y_2, \dots y_n$, and the actual output $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$.

all vectors

Training data: $\{(x_1, y_1), (x_2, y_2) \dots\}$

The goal of a loss function is to measure how far apart the actual output $\hat{y}$ is from the desired output y, and "training" = "make the loss get smaller"

★ Regression.

Loss function #1: Mean Squared Error (MSE)

For a NN whose output layer has 1 neuron
and for a batch of n input/output pairs:

$$\text{loss} = \underbrace{\frac{1}{n}\left(\sum_{i=1}^{n} \underbrace{(y_i - \hat{y}_i)^2}_{\text{squared diff between actual and expected output}}\right)}_{\text{mean of that, over the whole batch}}$$

For a NN whose output layer has k neurons, and
for a batch of n input/output pairs:

$$\text{loss} = \frac{1}{n \cdot k}\left(\sum_{i=1}^{n} \sum_{j=1}^{k} (y_{ij} - \hat{y}_{ij})^2\right)$$

y_{ij} is the j^{th} component of the vector y_i

Example:

```
>>> y = np.round(np.random.randn(3,5),2); y
array([[ 1.9 ,  0.24, -0.38, -0.86,  2.49],
       [-0.22,  0.17, -0.74,  1.12,  1.06],
       [-2.39,  0.98, -1.87, -1.62,  0.23]])
>>> yhat = np.round(np.random.randn(3,5),2); yhat
array([[ 0.03,  0.43, -0.25,  0.61, -0.92],
       [-1.84, -0.35,  2.32,  0.82,  0.51],
       [-1.11, -0.19,  0.48,  2.1 ,  0.6 ]])
>>> (y - yhat)**2
array([[ 3.4969,  0.0361,  0.0169,  2.1609, 11.6281],
       [ 2.6244,  0.2704,  9.3636,  0.09  ,  0.3025],
       [ 1.6384,  1.3689,  5.5225, 13.8384,  0.1369]])
>>> np.sum( (y-yhat)**2 ) / 15
np.float64(3.4996599999999995)
>>> 
```

3 output neurons, batch of 5 input/output pairs

* Regression.

Loss function #2: Mean Absolute Error (MAE)

For a NN whose output layer has 1 neuron and for a batch of n input/output pairs:

$$loss = \frac{1}{n}\left(\sum_{i=1}^{n} |y_i - \hat{y}_i|\right)$$

For a NN whose output layer has k neurons, and for a batch of n input/output pairs:

$$loss = \frac{1}{n \cdot k}\left(\sum_{i=1}^{n}\sum_{j=1}^{k} |y_{ij} - \hat{y}_{ij}|\right)$$

y_{ij} is the j^{th} component of the vector y_i

★ Regression.

MSE:

- More common
- Comes from linear regression, where it has more motivation
- Penalizes outliers more (one really bad prediction is much worse than two medium bad predictions)

MAE:

- Less common, but not uncommon
- Penalizes outliers equally to other points

★ Regression.

Example:

$$y = \begin{bmatrix}1\\1\end{bmatrix}, \quad \hat{y} = \begin{bmatrix}5\\1\end{bmatrix} \leftarrow \text{off by } 4$$

$$MSE = \frac{1}{2}\left((5-1)^2 + (1-1)^2\right) = 8$$

$$MAE = \frac{1}{2}\left(|5-1| + |1-1|\right) = 2$$

versus

each off by 2

$$y = \begin{bmatrix}1\\1\end{bmatrix} \quad \hat{y} = \begin{bmatrix}3\\3\end{bmatrix}$$

$$MSE = \frac{1}{2}\left((3-1)^2 + (3-1)^2\right) = 4$$

smaller

$$MAE = \frac{1}{2}\left(|3-1| + |3-1|\right) = 2$$

same