Scientific Computing

Announcements

Friday, April 17

[recording]

* Homework 6 assigned,
  due Monday, April 27, 11:59pm

* Final Exam: Monday, 5/4,
  1pm-3pm
  Johnston Hall 417

Office Hours:

Mon, 9:30-10:30

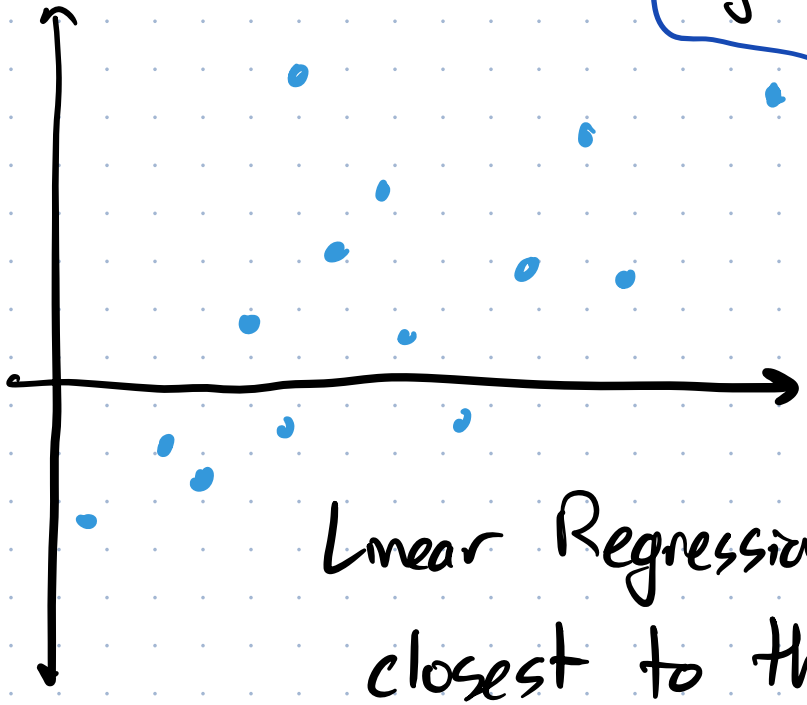
~~Fri, 2:00-3:00~~

Cudahy 307

Topic 16 - 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

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

over all known points.

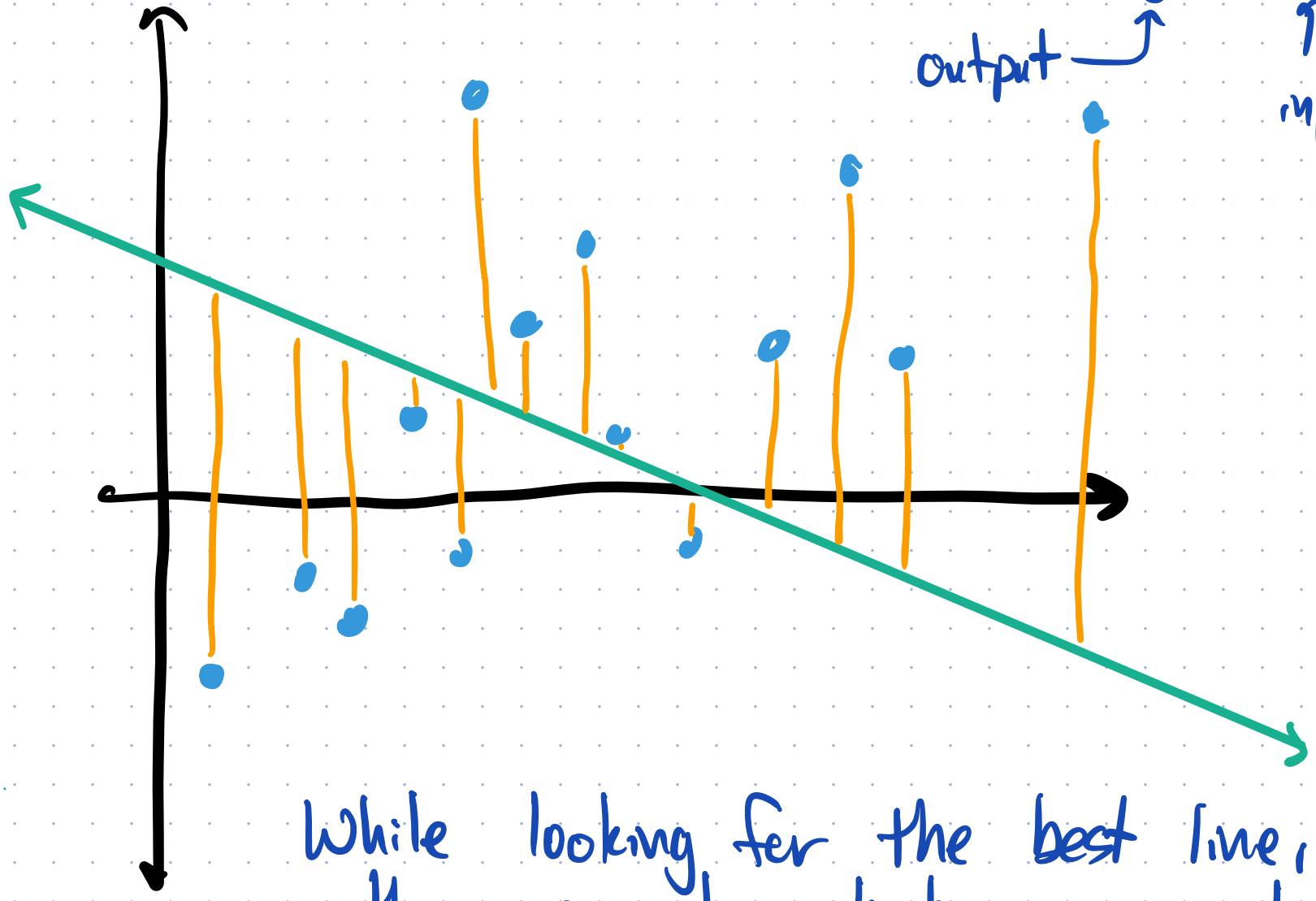
Strong Analogy: Linear Regression

knobs

$y = mx + b$

output

input



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

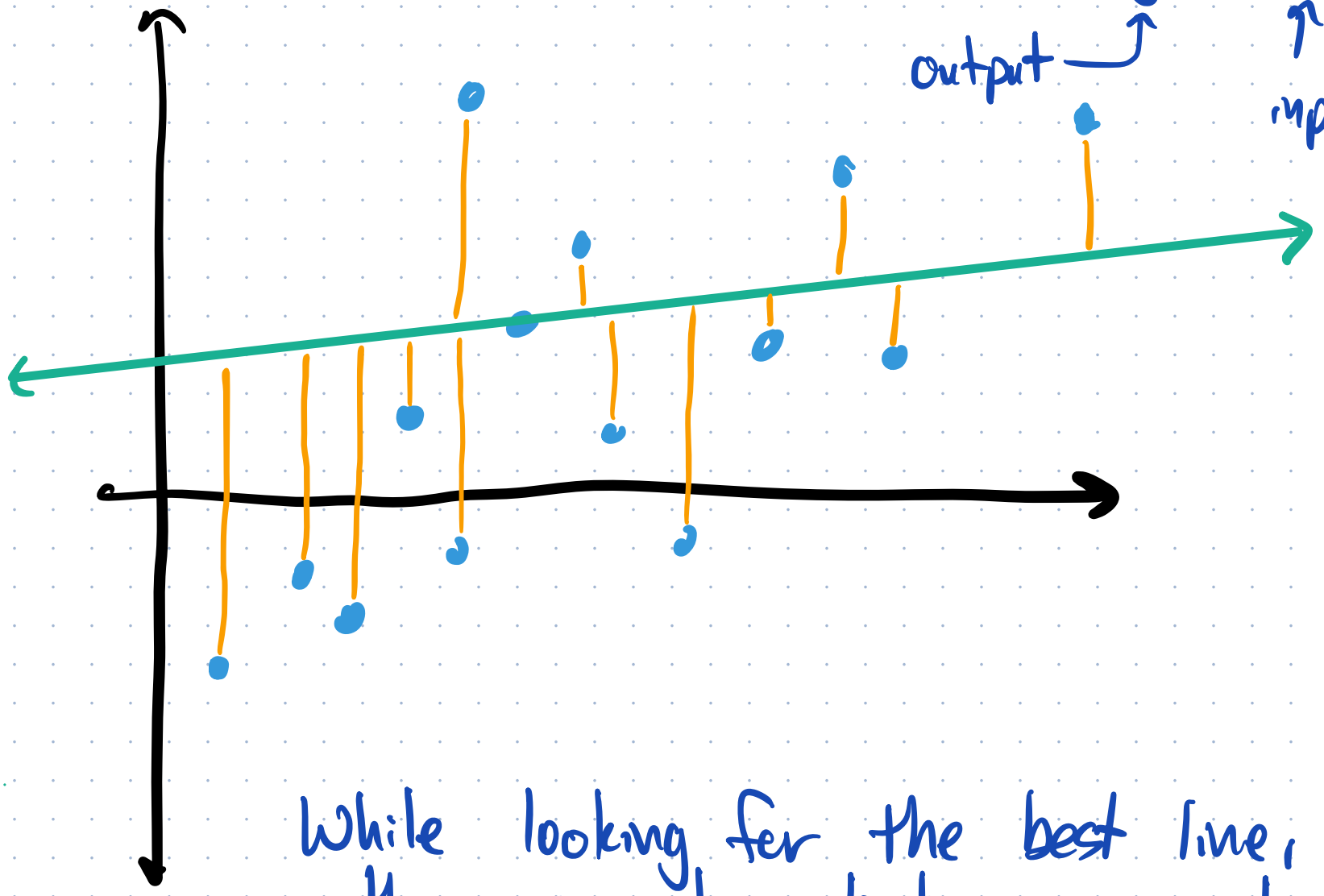
Strong Analogy: Linear Regression

knobs

$y = mx + b$

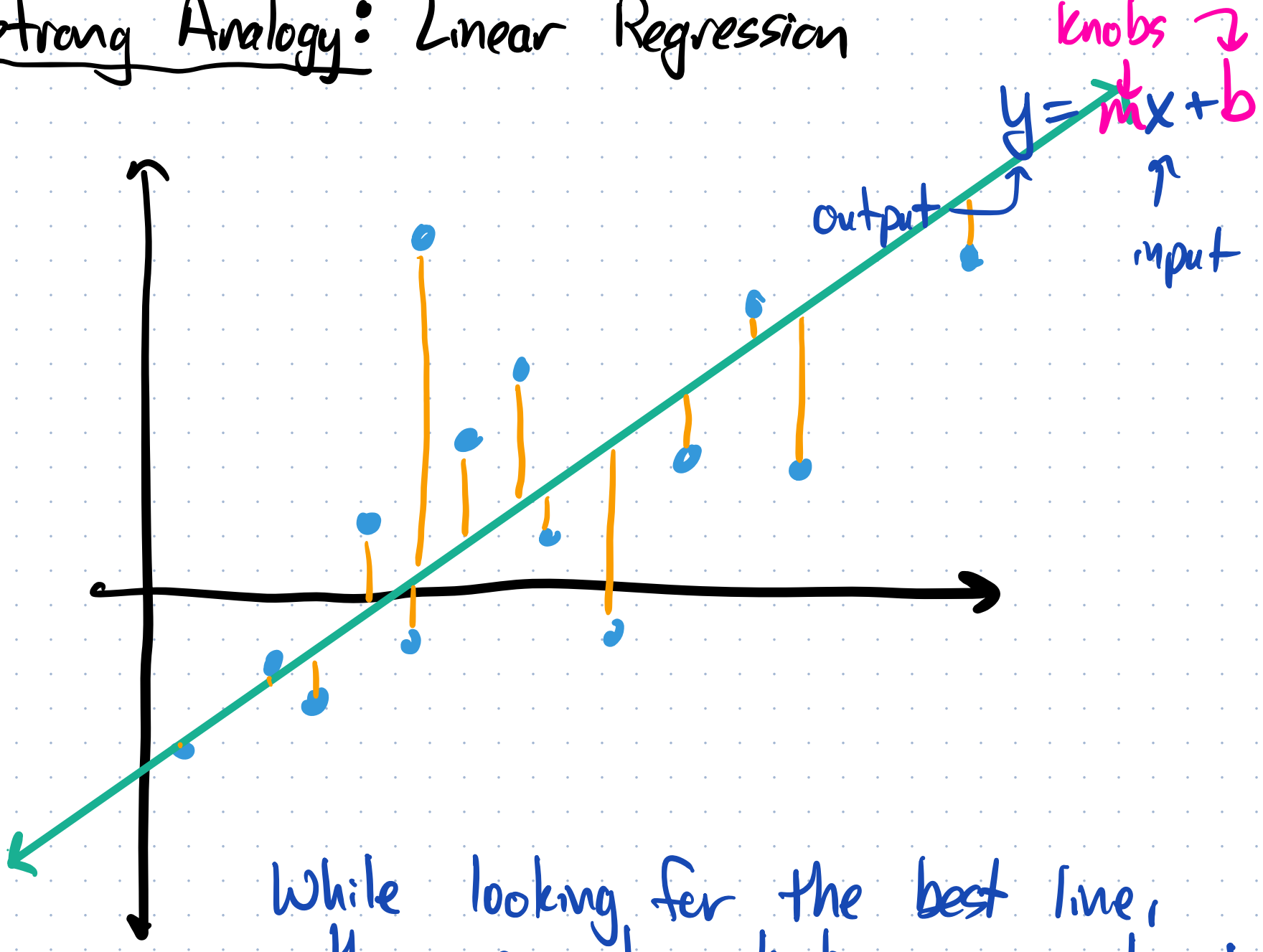
output

input



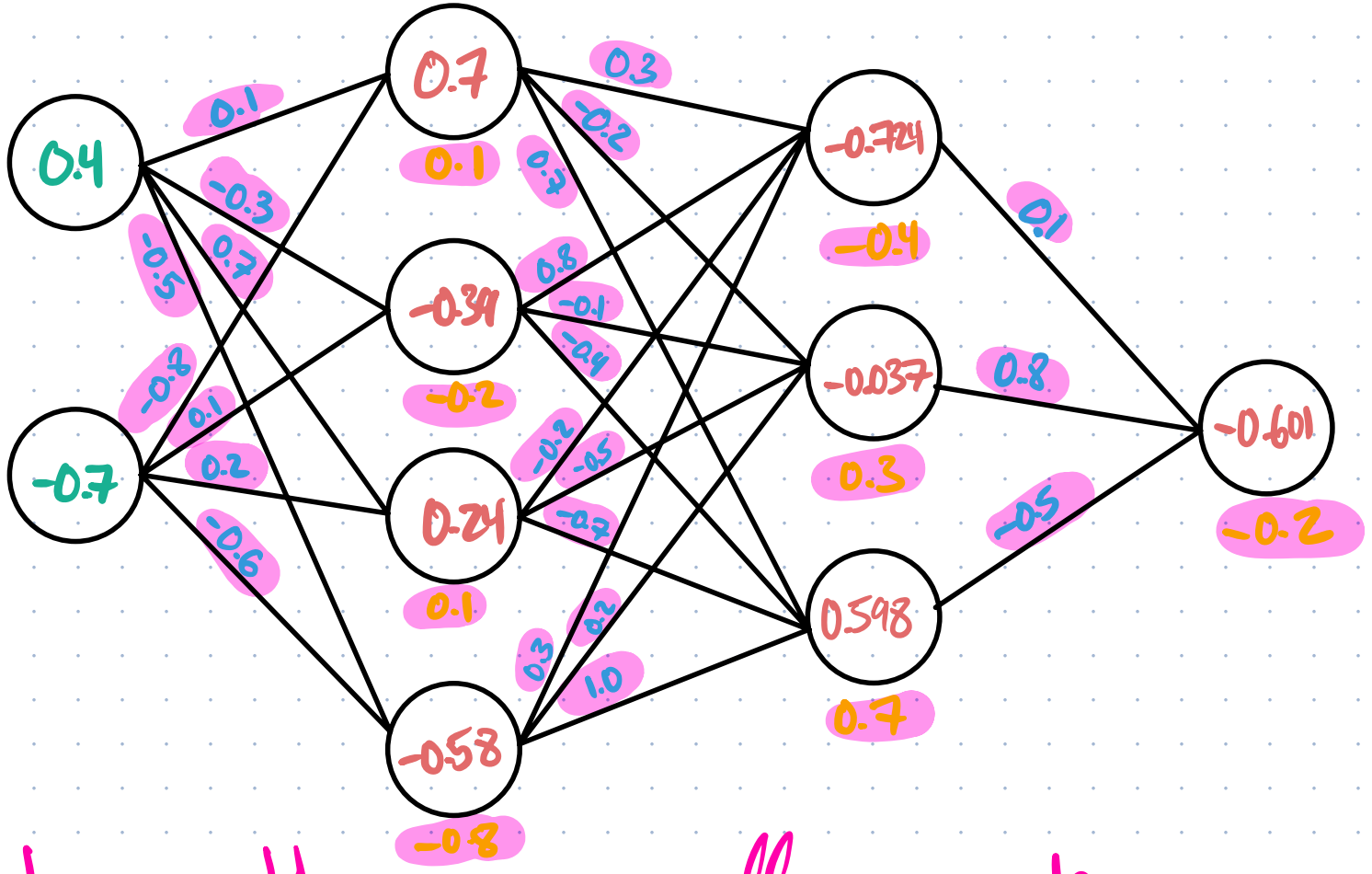
While looking for the best line, there are two knobs we can turn: m and b

Strong Analogy: Linear Regression



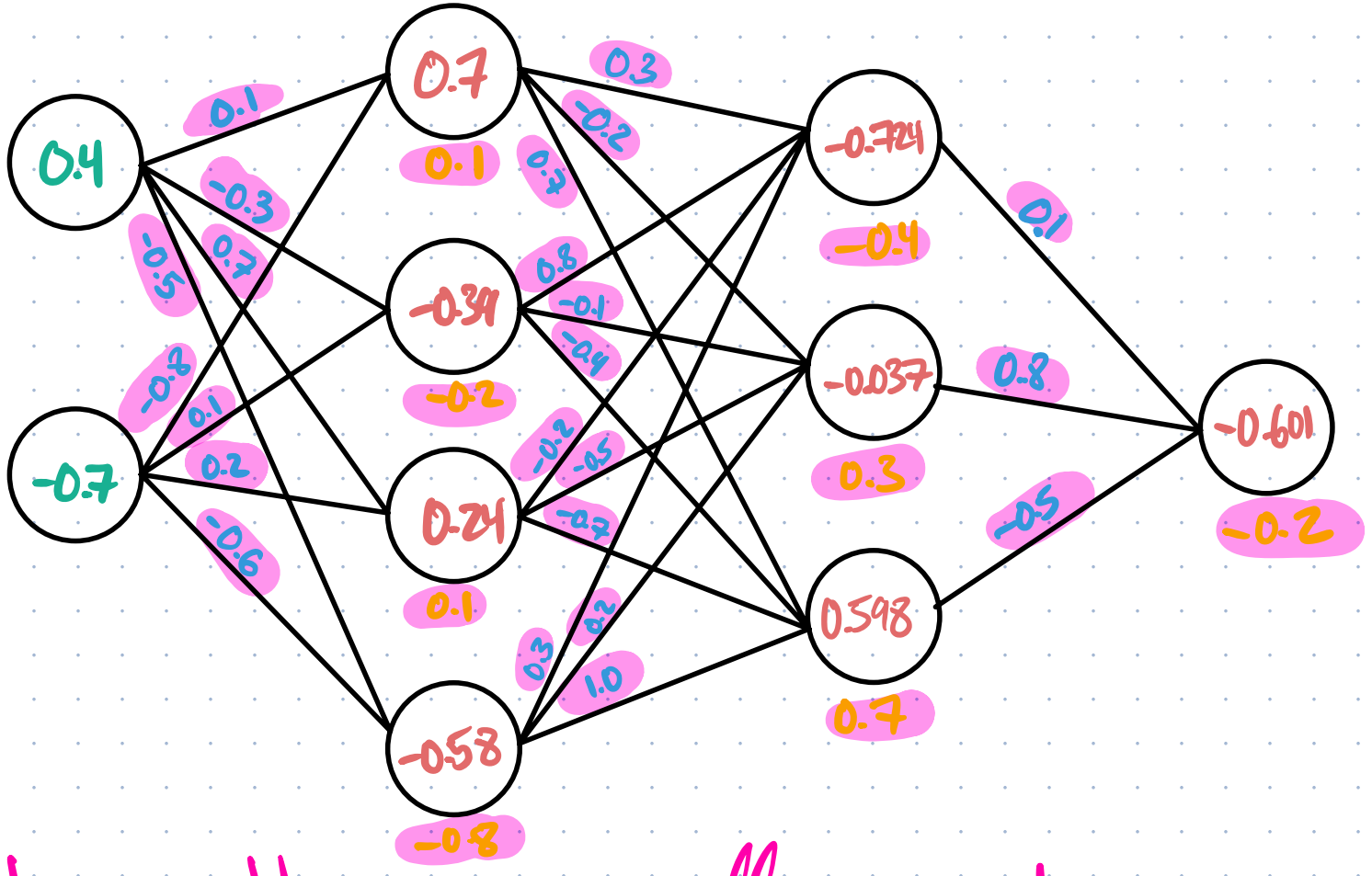
While looking for the best line, there are two knobs we can turn: m and b

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



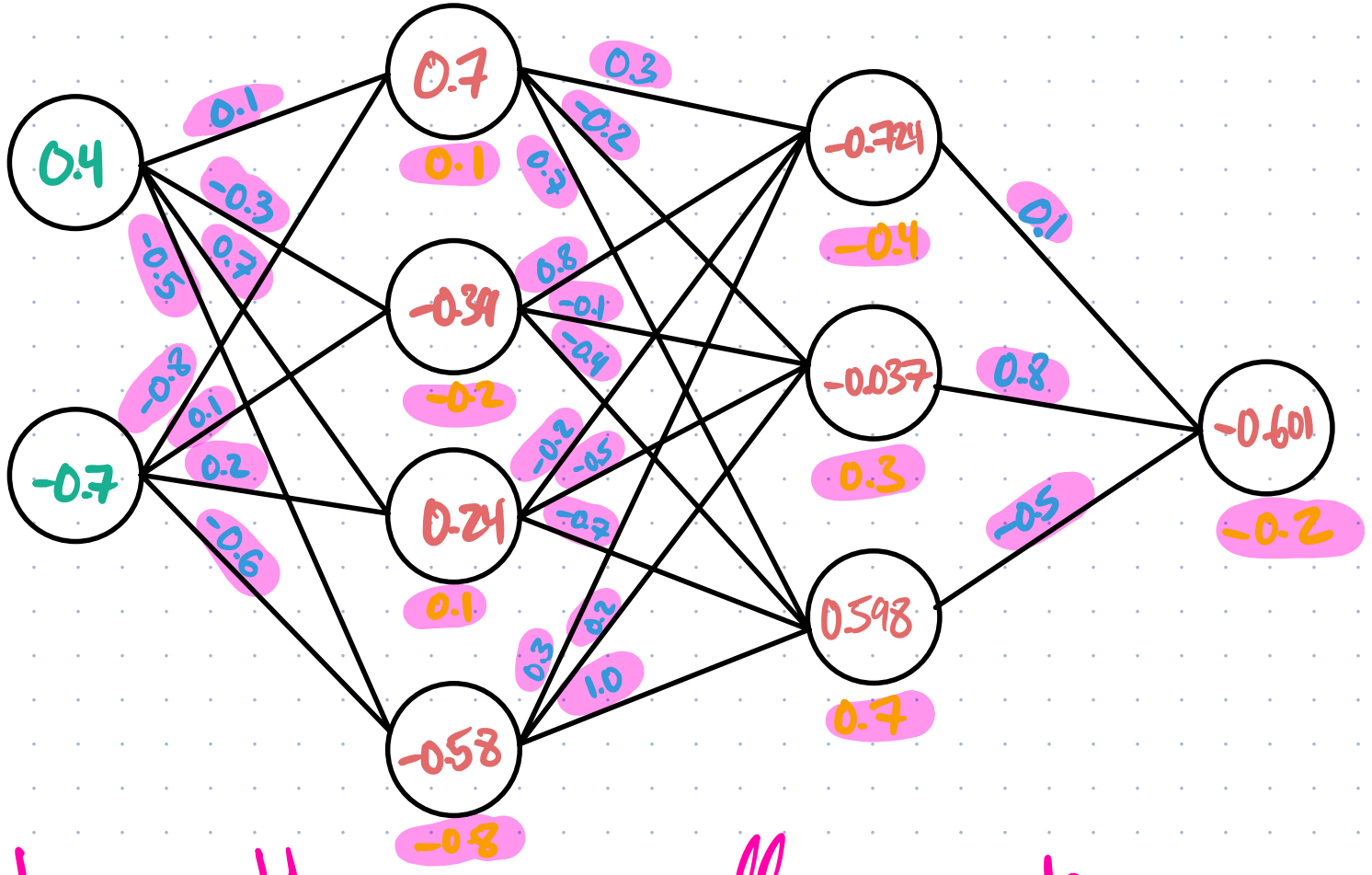
32 knobs in this very small example

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 very small example

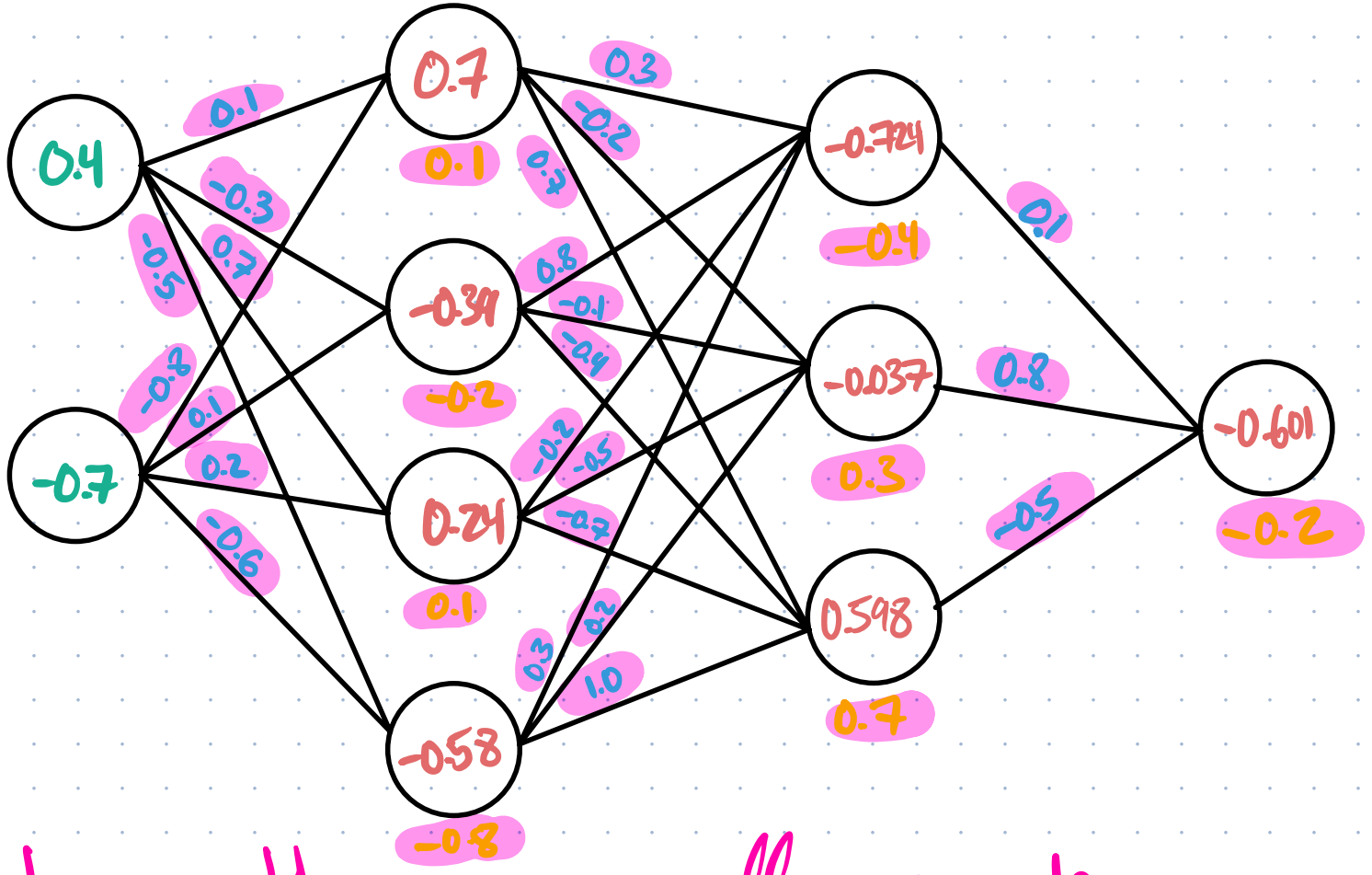
Actual Goal: Find the best way to tune the knobs so that when the NN sees new input data, it produces correct output.



32 knobs in this very small example

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

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



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

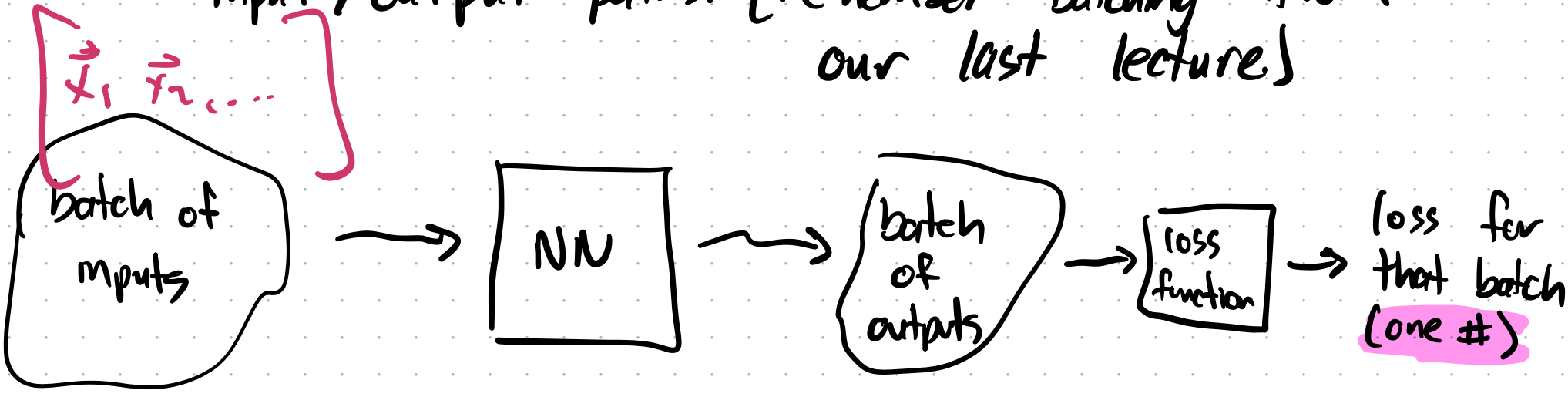
We use different <u>loss functions</u> for each type of problem.

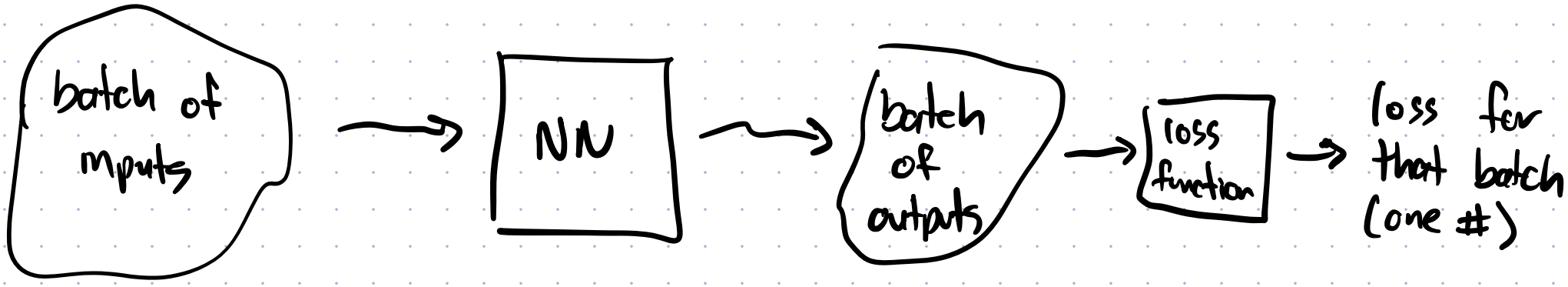
↳ Scoring function for a NN, smaller is better

★ Regression.

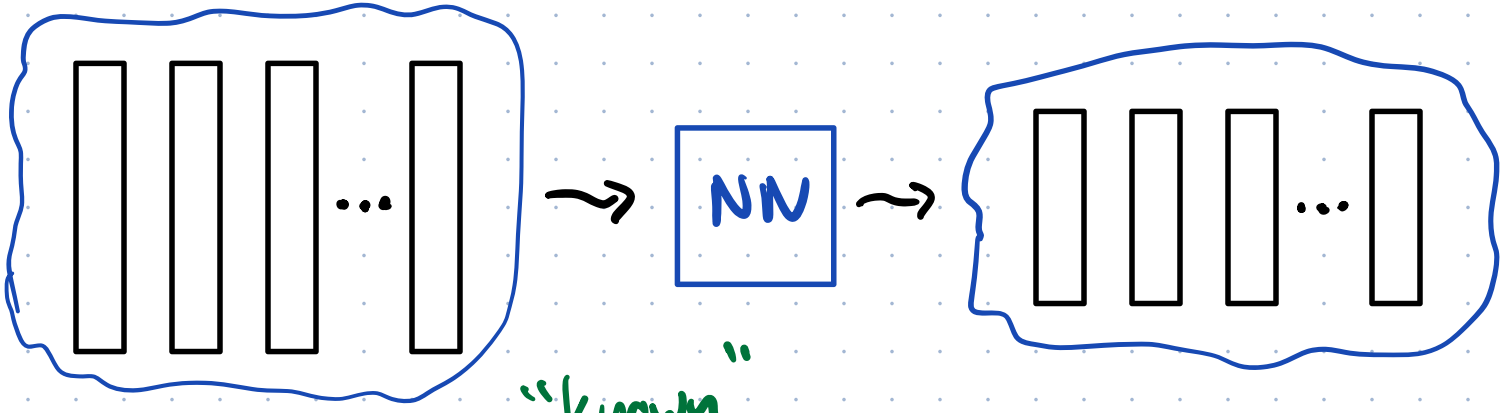
$T = \{ (\vec{x}_1, \vec{y}_1), (\vec{x}_2, \vec{y}_2), \ldots \}$

- 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 expected output $y_1, y_2, \dots y_n$, and the actual output $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$.

"known"

all vectors

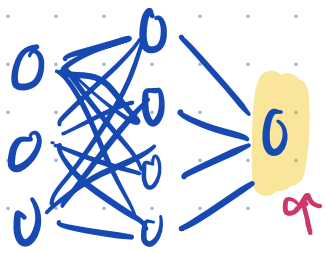
Training data: $\{(x_1, y_1), (x_2, y_2) \dots\}$

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:



$$loss = \frac{1}{n}\left(\sum_{i=1}^{n}(y_i - \hat{y}_i)^2\right)$$

single #

squared diff between actual and expected output

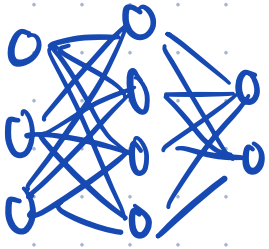
mean of that, over the whole batch

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \rightarrow \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \hat{y}_3 \end{bmatrix}$$

n=3

expected $\begin{bmatrix} y_{11} & y_{21} \\ y_{12} & y_{22} \end{bmatrix}$

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



actual $\begin{bmatrix} \hat{y}_{11} & \hat{y}_{21} \\ \hat{y}_{12} & \hat{y}_{22} \end{bmatrix}$

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

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

Example:

expected output

actual output

```
>>> 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(3.499659999999995)
>>>
```

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:

$$y = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \hat{y} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$ ← 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

$$y = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \hat{y} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$ ← each off by 2

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

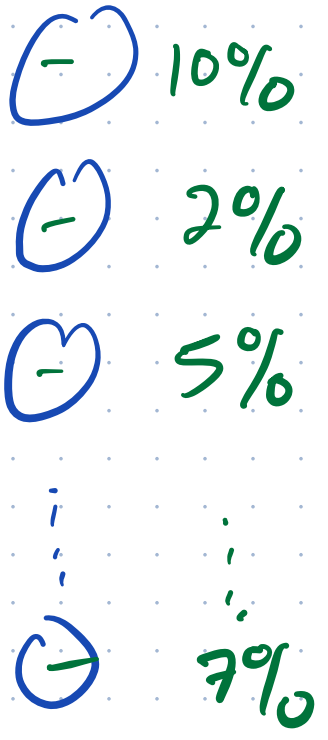
$$MAE = \frac{1}{2}\left(|3-1| + |3-1|\right) = 2$$ ← same

* Classification

Recall that if we are sorting inputs into k classes (e.g. 10 digits for MNIST) then there are k output neurons.

We pass them collectively through the softmax activation function to turn them into a probability distribution.

- Each output in $[0,1]$
- Outputs sum to 1



Previously:

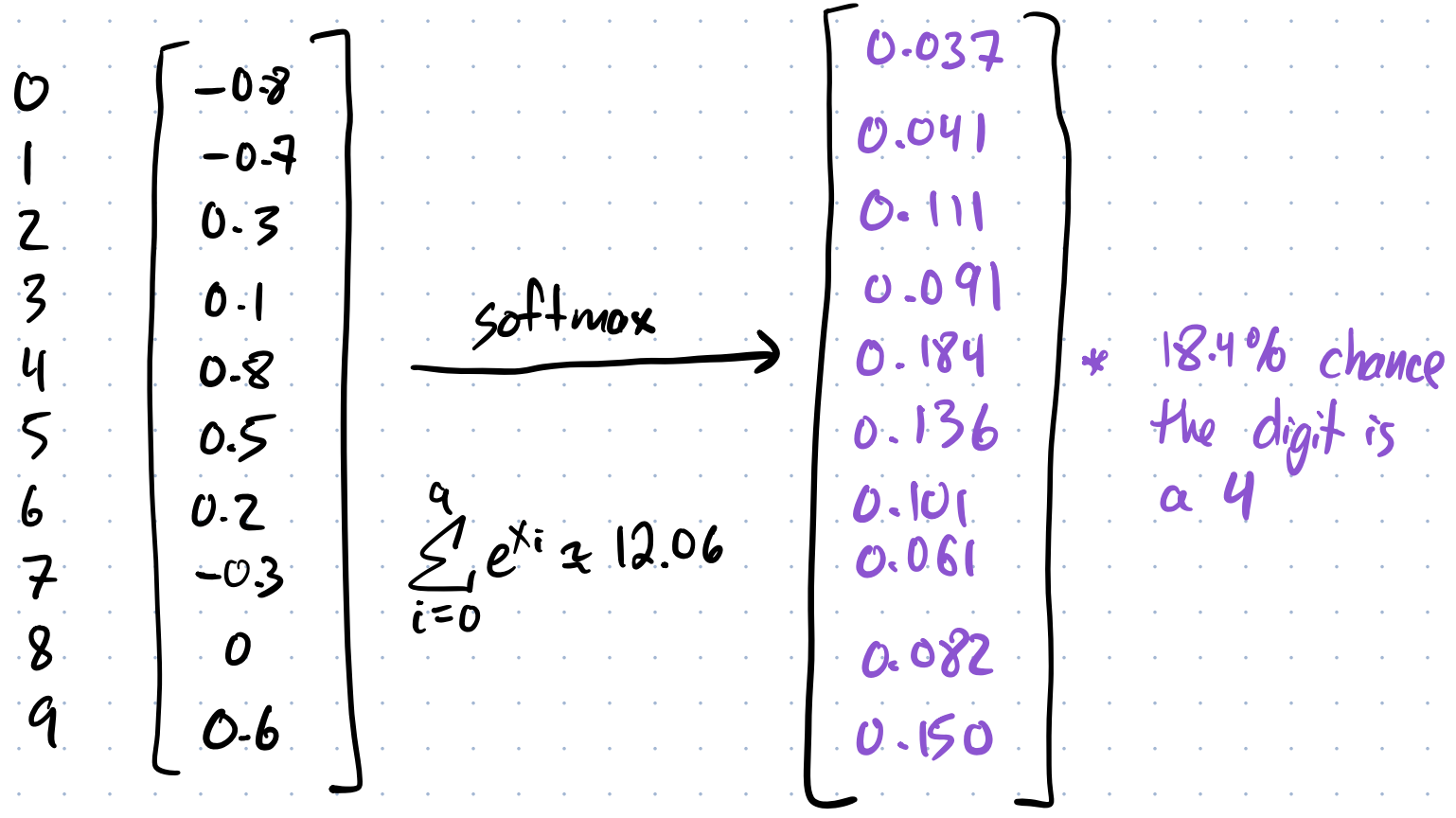
Unlike our other activation functions, Softmax works on the whole vector at once, not one value at a time individually.

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} e^{x_1}/S \\ e^{x_2}/S \\ e^{x_3}/S \\ \vdots \\ e^{x_k}/S \end{bmatrix} \quad \text{where} \quad S = \sum_{i=1}^{k} e^{x_i}$$

Obvious these add up to 1 because the denominator is their sum.
Obviously > 0 because $e^x > 0$.

Previously:

Ex:



* Classification

Goal of a loss function: measure the distance between the actual classification and the predicted probability distr.

(1-hot encoding)

Ex: Actual: [0 0 0 0 1 0 0 0 0 0]

Predicted: [0.01 0.03 0.1 0.07 0.33 0.11 0.01 0.21 0.03 0.1]

Another predicted: [0 0 -01 0.03 0.92 0.02 0.01 0 0.01 0]

Second one is much closer to the actual answer, so the loss should be lower.

* Classification

Training data

output of NN

Ex: Actual: [0 0 0 0 1 0 0 0 0 0]

Predicted: [0.01 0.03 0.1 0.07 0.33 0.11 0.01 0.21 0.03 0.1]

Another predicted: [0 0 -.01 0.03 0.92 0.02 0.01 0 0.01 0]

Lots of ways to devise a loss function to capture this closeness.

The most popular is "Categorical Cross-Entropy".

For 1 sample with k outputs: ← $k=10$ for digits

$$loss = -\sum_{i=1}^{k} y_i \cdot \log(\hat{y}_i)$$

$$= -\left(y_1 \cdot \log(\hat{y}_1) + y_2 \cdot \log(\hat{y}_2) + \cdots + y_k \cdot \log(\hat{y}_k)\right)$$

* Classification

Ex: Actual: [0 0 0 0 1 0 0 0 0 0]

Predicted: [0.01 0.03 0.1 0.07 0.33 0.11 0.01 0.21 0.03 0.1]

Another predicted: [0 0 -01 0.03 0.92 0.02 0.01 0 0.01 0]

$$loss = -\sum_{i=1}^{k} y_i \cdot \log(\hat{y}_i)$$

0 for the "wrong classes"
1 for the "correct class"

So, this simplifies to $loss = -\log(\hat{y}_c)$
where $\hat{y}_c$ is the confidence level of the correct class

* Classification

Ex: Actual: [0 0 0 0 1 0 0 0 0 0]

Predicted: [0.01 0.03 0.1 0.07 0.33 0.11 0.01 0.21 0.03 0.1]

Another predicted: [0 0 -01 0.03 0.92 0.02 0.01 0 0.01 0]

$$loss = -\log(\hat{y}_c)$$

→ $loss = -\log(0.33) \approx 0.48$

→ $loss = -\log(0.92) \approx 0.03$ ← smaller!

* Classification

For the loss of a whole batch of inputs/outputs, just average the loss of each (input, output) pair individually.
(same thing we did for regression)

Why this formula?

- Connections to statistics (distance between discrete probability distributions)
- Works well in practice

So, whether we're doing regression or classification, we have a "loss function" that measures:

for a batch of inputs, how far is the actual from what the training data says it should be

Lower is better.

Goal: pick weights and biases that make the loss as low as possible on your training data, and hope that the resulting NN performs well on whatever new data you have for it.

How do we find good weights and biases?

Bad idea: Pick random #s for them over and over again until some combination is good.

Interesting Idea: Do Hill Climbing or Simulated Annealing. Change some/all of the weights and biases by a little, see if the loss of the new NN is better or worse.

This is not ever done in practice, and I've never seen it discussed much. Why?

My guesses:
* too slow
* so many parameters that it's hard to go downhill
* the method people do use is good and fast

Next topic: Backpropagation