Scientific Computing

April 28, 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

→ Loss Functions

→ Backpropagation

Office Hours:

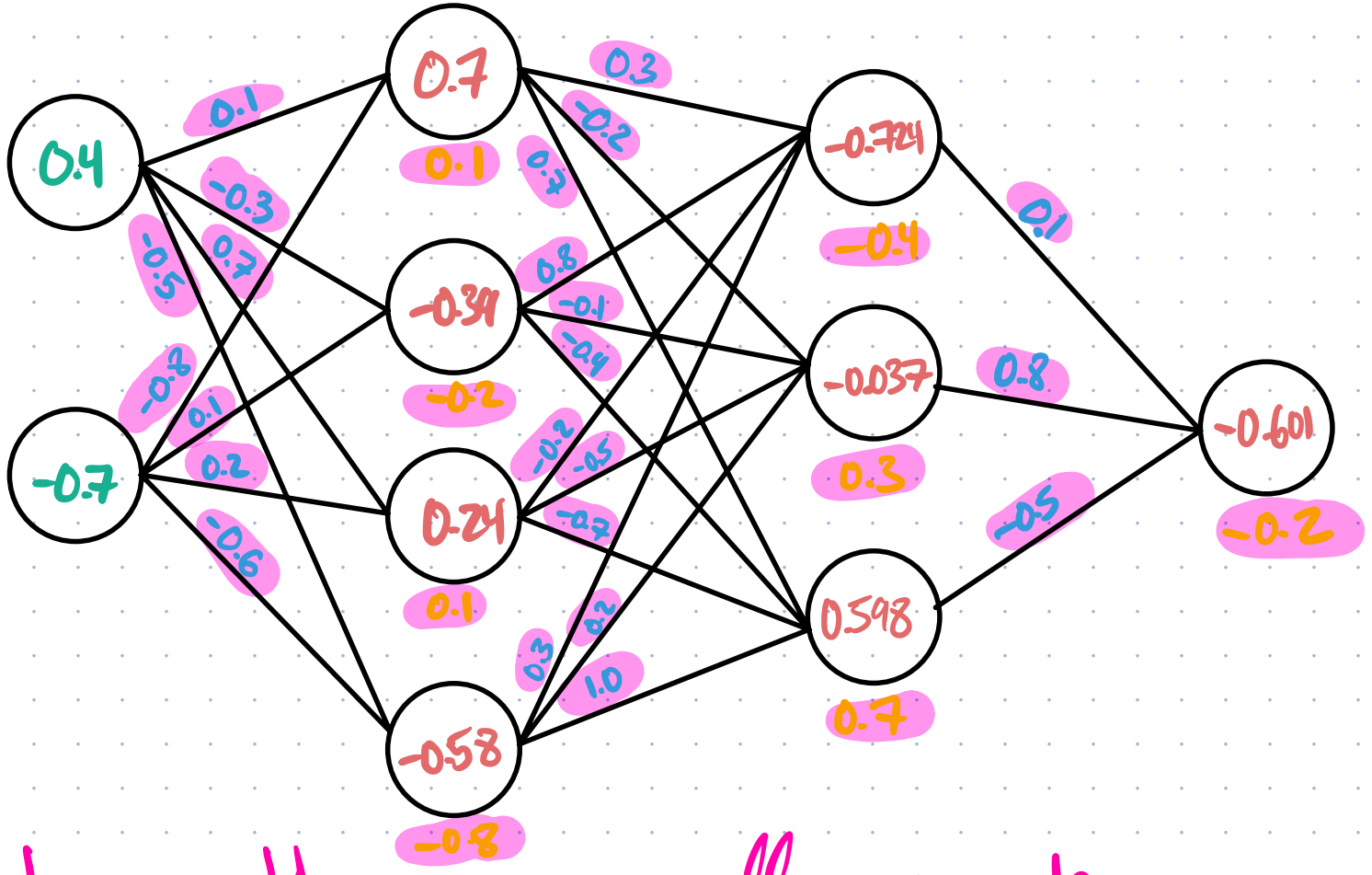
Mon + Fri

9:30am - 10:30am

Cudahy 307

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

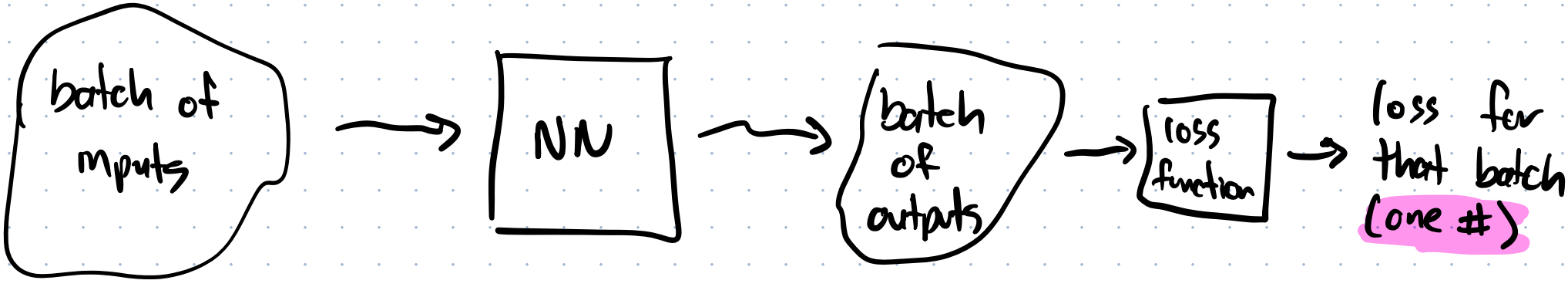
* MNIST digits
* Predict whether a patient has diabetes, prediabetes, or neither, based on health and lifestyle data

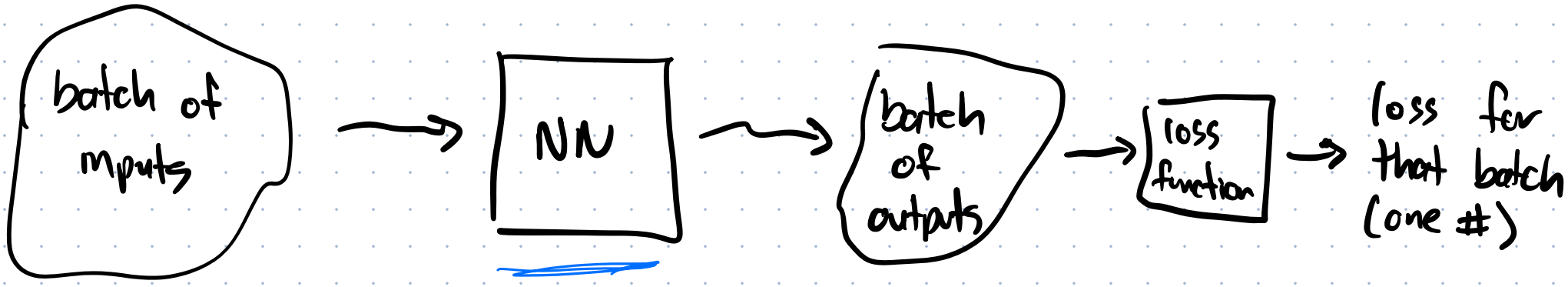
We use different loss functions for each type of problem.

↳ Scoring function for a NN, smaller is better

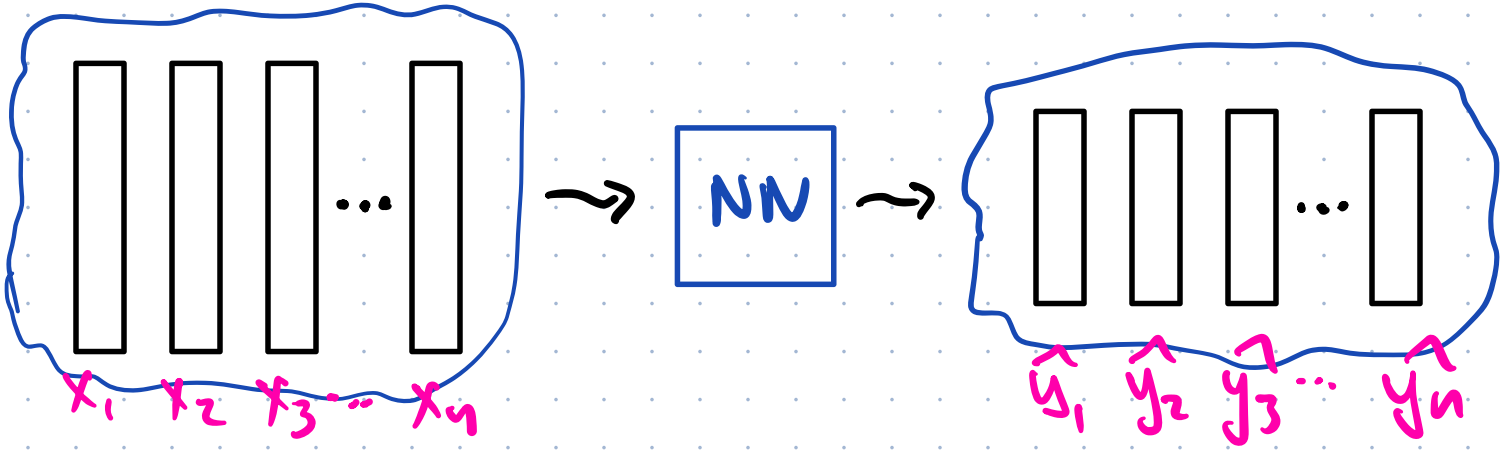
★ Regression.

- Actually, first some notation.
- The loss function is defined not for one input/output pair at a time, but for a whole batch 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 $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$.

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:

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

$(y_i - \hat{y}_i)^2$: squared diff between actual and expected output

$\frac{1}{n}$: 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:

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

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(3.499659999999995)
>>>
```

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.

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.

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

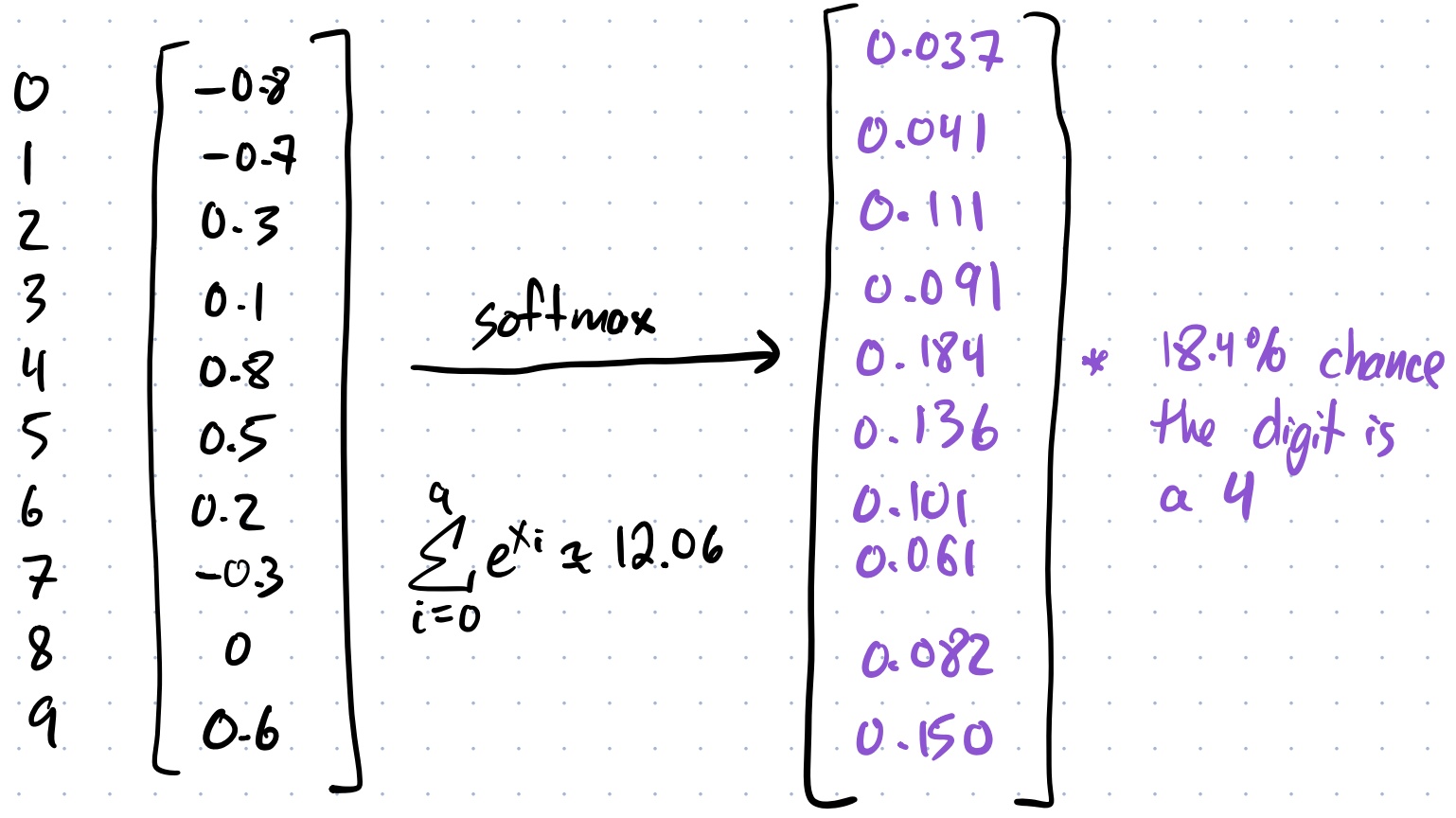
where $S = \sum_{i=1}^{k} e^{x_i}$

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

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

Previously:

Ex:



* Classification

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

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

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

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

y_i: 0 or 1

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

$$= -\log(\hat{y}_4) \quad \text{(in this sample)}$$

* 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

Topic 16 - Backpropagation

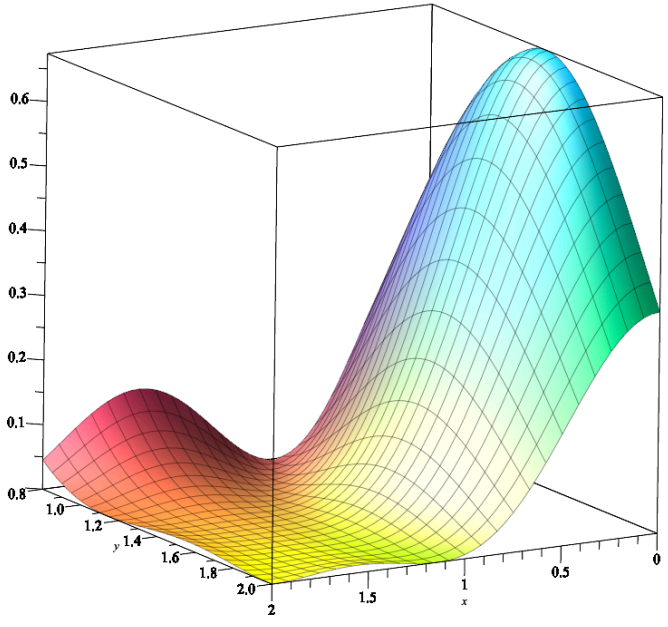
Big picture:

* Start with a NN with random weights and biases
* Feed all of the known inputs (training data) through in one batch and get all their outputs
* Compute the loss between the expected outputs (y) and the actual outputs ($\hat{y}$)
* Use **CALCULUS** to figure out how to tweak the weights + biases a little to make the loss go down a little.
* Repeat until your NN is good enough.

CALCULUS

Let's recall the idea of Gradient Descent.

Suppose you have a function $f(x,y)=z$.



The vector $\begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \nabla f$ tells you the direction of steepest ascent from any point (x,y).

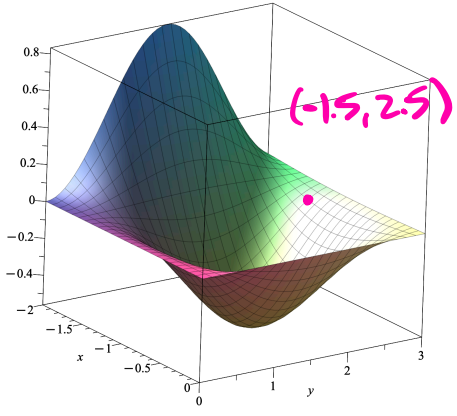
The negative version $-\nabla f = \begin{bmatrix} -\frac{\partial f}{\partial x} \\ -\frac{\partial f}{\partial y} \end{bmatrix}$ is the direction of steepest descent.

CALCULUS

The negative version $\begin{bmatrix} -\frac{\partial f}{\partial x} \\ -\frac{\partial f}{\partial y} \end{bmatrix}$ is the direction of steepest descent.

This means if you're only going to walk 1 step in the mountains, then one step in that direction will get you as low as possible among all possible one steps.

Ex: $f(x,y) = x\cos(x)(\sin(y))^2$.



$$\begin{bmatrix} -\frac{\partial}{\partial x}\left(x\cos(x)\cdot(\sin(y))^2\right) \\ -\frac{\partial}{\partial y}\left(x\cos(x)(\sin(y))^2\right) \end{bmatrix}$$

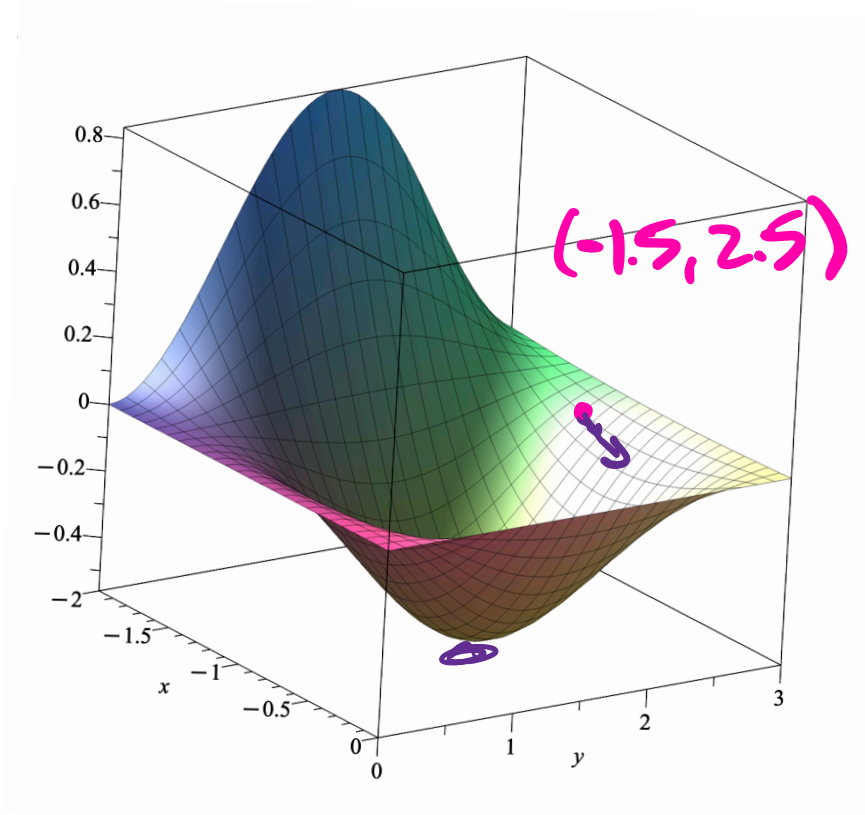
$(\sin(3))^2$

CALCULUS

Ex: $f(x,y) = x\cos(x)(\sin(y))^2$.

$$\frac{\partial f}{\partial x} = (\sin(y))^2 \cdot (\cos(x) - x\sin(x))$$

$$\frac{\partial f}{\partial y} = x\cos(x) \cdot 2\sin(y)\cos(y)$$



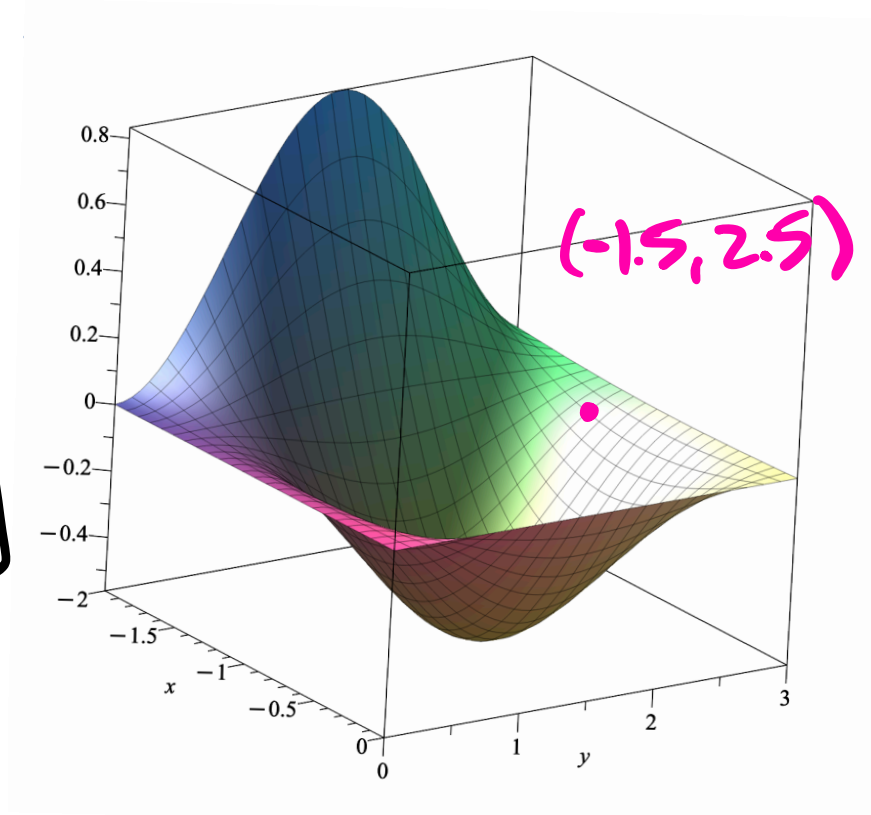
$$(-\nabla f)(-1.5, 2.5) = -\begin{bmatrix} \sin(2.5)^2 \cdot (\cos(-1.5) + 1.5\sin(1.5)) \\ (-1.5)\cos(-1.5) \cdot 2 \cdot \sin(2.5)\cos(2.5) \end{bmatrix}$$

$$= \begin{bmatrix} 0.5105\ldots \\ -0.1017\ldots \end{bmatrix}$$

CALCULUS

Ex: $f(x,y) = x\cos(x)(\sin(y))^2$

$$(-\nabla f)(-1.5, 2.5) = \begin{bmatrix} 0.5105\ldots \\ -0.1017\ldots \end{bmatrix}$$



This doesn't mean you move +0.51 on the x-axis an −0.10 on the y-axis. That would be a huge step

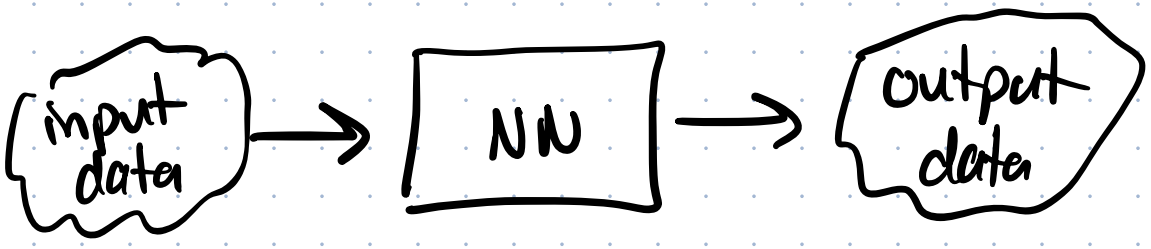
Instead you move a small step, maybe $\frac{1}{100}$ of that.

$$\begin{bmatrix} -1.5 \\ 2.5 \end{bmatrix} + \begin{bmatrix} 0.0051\ldots \\ -0.0010\ldots \end{bmatrix} = \begin{bmatrix} -1.5051 \\ 2.4990 \end{bmatrix}$$ and repeat.

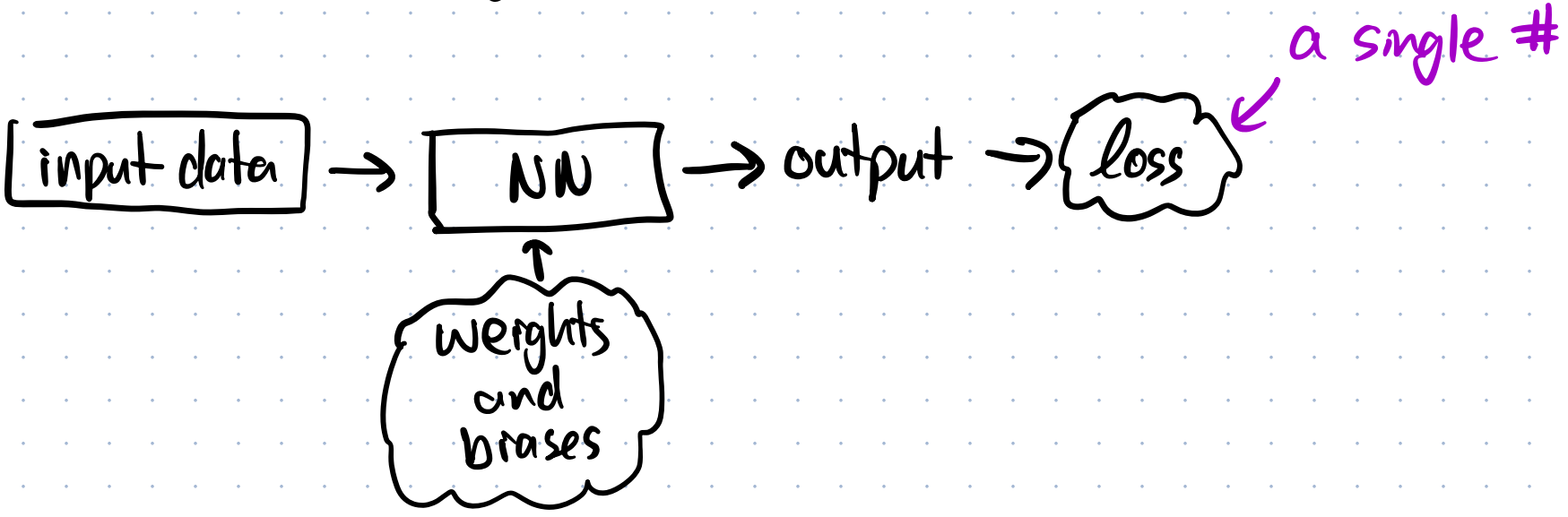
$-\nabla f$ is the same! Just new points.

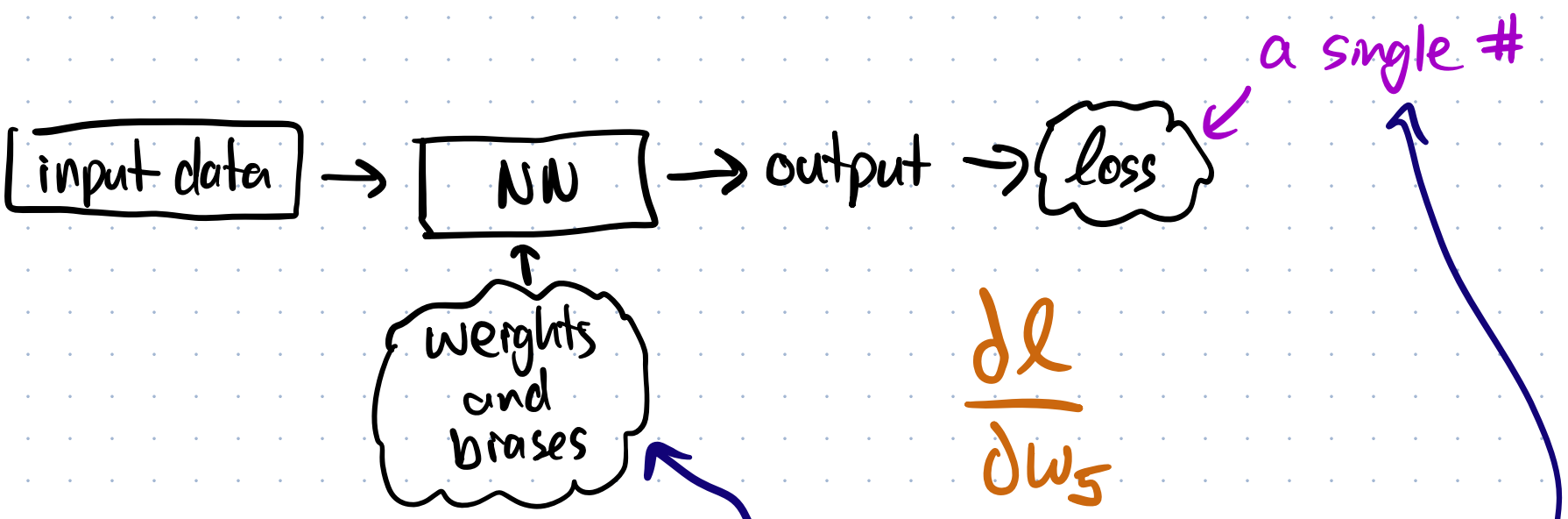
* Gradient Descent Demo

We have been thinking about Neural Networks as big functions



For now, instead of thinking of the input data as the variables, we'll think of all the training data as a fixed constant and the weights and biases as the variables.





$$\frac{\partial \ell}{\partial w_5}$$

How can we find the inputs that minimize the output?

Gradient Descent!

Suppose the neural network has weights $w_1, w_2, \ldots, w_m$ and biases $b_1, b_2, \ldots, b_n$. Let ℓ be the mean loss over the whole batched input.

What is $-\nabla NN(w_1, w_2, \ldots w_m, b_1, b_2, \ldots, b_n)$?

$$\ell = NN(w_1, w_2, \ldots, w_m, b_1, b_2, \ldots b_n).$$

How will we compute

$$\nabla NN = \begin{bmatrix} \frac{\partial \ell}{\partial w_1} \\ \vdots \\ \frac{\partial \ell}{\partial w_m} \\ \frac{\partial \ell}{\partial b_1} \\ \vdots \\ \frac{\partial \ell}{\partial b_n} \end{bmatrix} ?$$

CALCULUS

specifically
the CHAIN RULE

the CHAIN RULE

$f(t)$

$\frac{df}{dx}(3) = 5$ means if you change go from

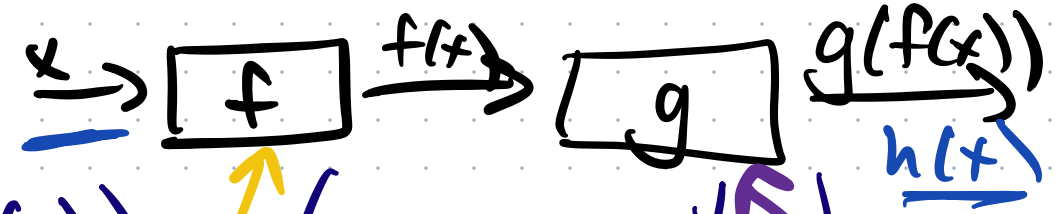
$x = 3$ to $x = 3 + [\text{a little more}]$

ε

then $f(t)$ goes from

$f(3)$ to $f(3) + 5 \cdot [\text{a little more}]$.

the CHAIN RULE



Suppose $h(x) = g(f(x))$ (a composition).

Chain rule: $h'(x) = g'(f(x)) \cdot f'(x)$

or $\frac{dh}{dx} = \frac{dh}{df} \cdot \frac{df}{dx}$

how much h changes when x changes a little

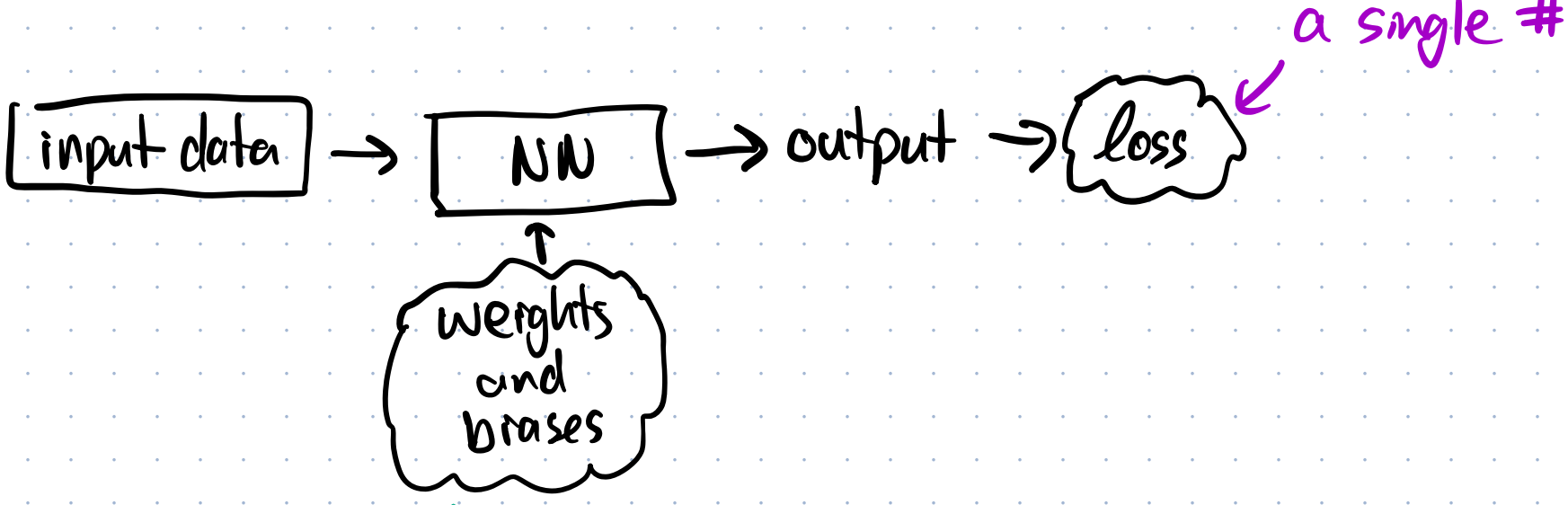
how much f changes when x changes a little

how much g changes when f changes a little

the CHAIN RULE

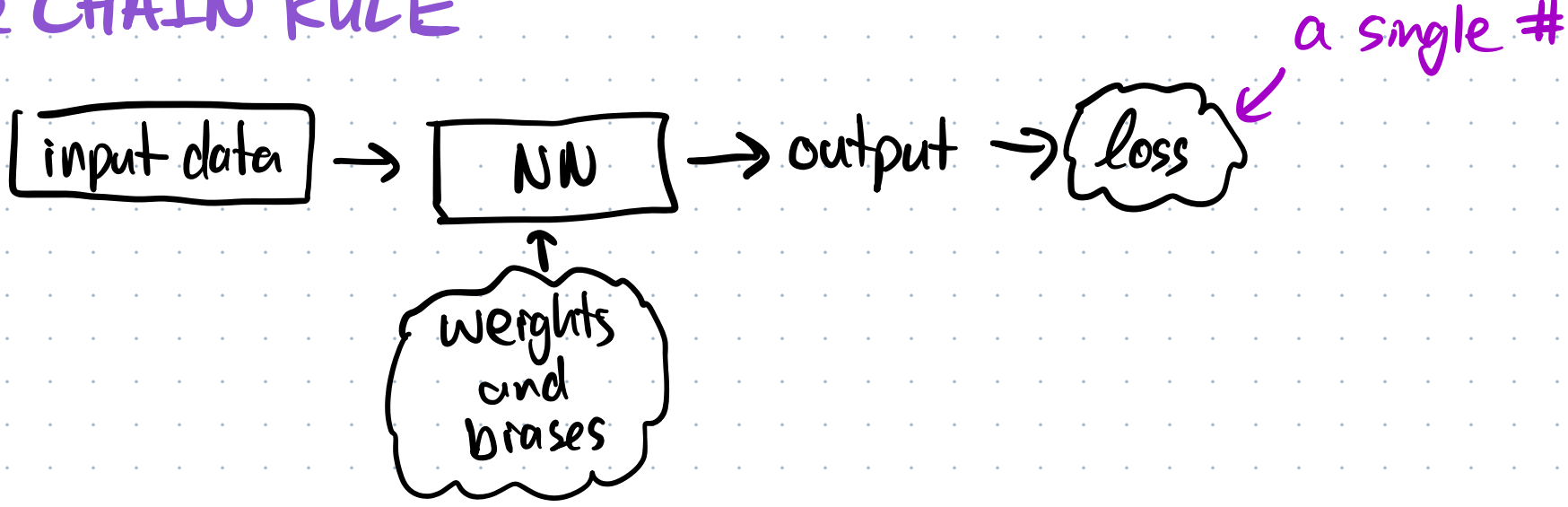
Neural Networks are just really big multivariate compositions of simple functions

(addition, multiplication, activation functions, loss functions)



We can apply the chain rule a lot to see how changing any weight or bias individually affects the loss

the CHAIN RULE

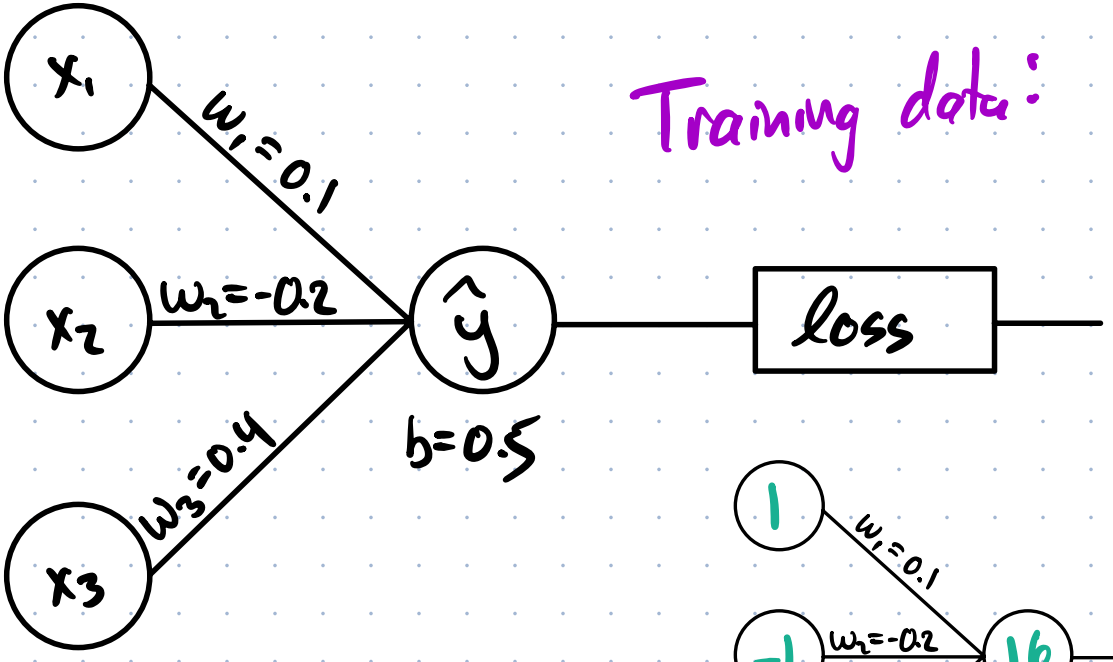


We can apply the chain rule a lot to see how changing any weight or bias individually affects the loss

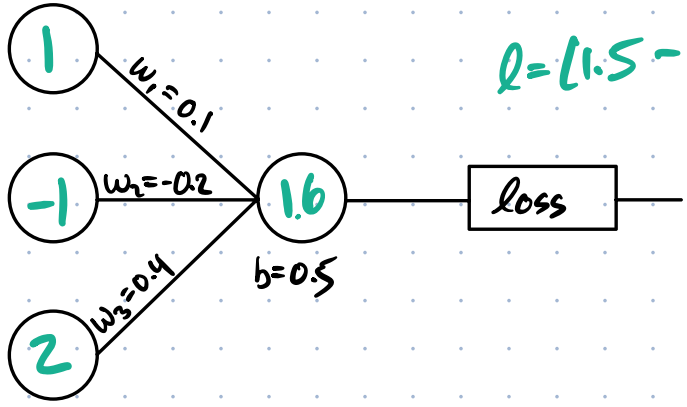
$$\nabla NN = \begin{bmatrix} \frac{\partial \ell}{\partial w_1} \\ \vdots \\ \frac{\partial \ell}{\partial w_m} \\ \frac{\partial \ell}{\partial b_1} \\ \vdots \\ \frac{\partial \ell}{\partial b_n} \end{bmatrix}.$$

That means we'll know the gradient and we can do gradient descent.

First example: No AF, no hidden layers, 3 input, 1 output, only one point of training data

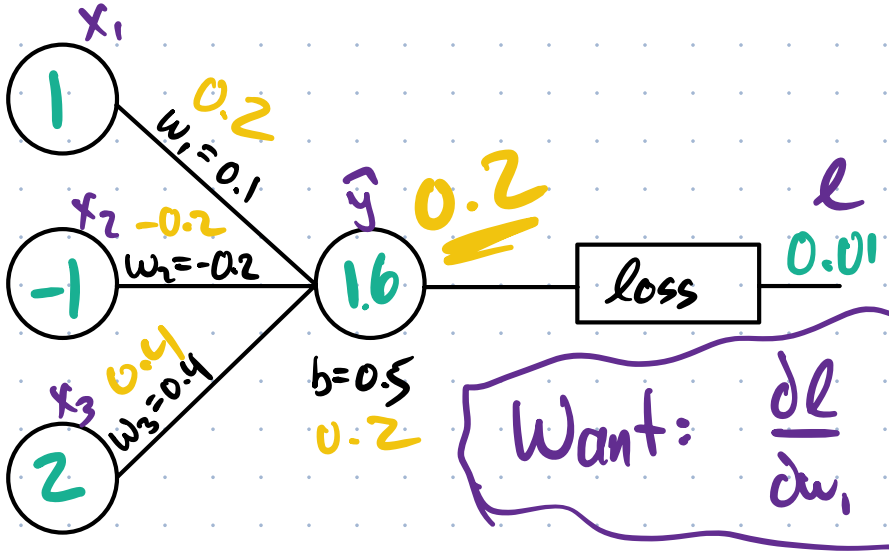


Training data: $(1, -1, 2) \rightarrow 1.5$



$\ell = (1.5 - 1.6)^2 = 0.01$

How do we adjust w_1, w_2, w_3, b to make ℓ go down for <u>this</u> training data?



Training data: $(1, -1, 2) \to 1.5$

Want: $\frac{\partial \ell}{\partial w_1}, \frac{\partial \ell}{\partial w_2}, \frac{\partial \ell}{\partial w_3}, \frac{\partial \ell}{\partial b}$

$\ell = (\hat{y} - 1.5)^2$.

So, $\frac{\partial \ell}{\partial \hat{y}} = 2(\hat{y} - 1.5)$. ($\hat{y} = 1.6$, $1.6 - 1.5$)

$= 2 \cdot (0.1) = 0.2$

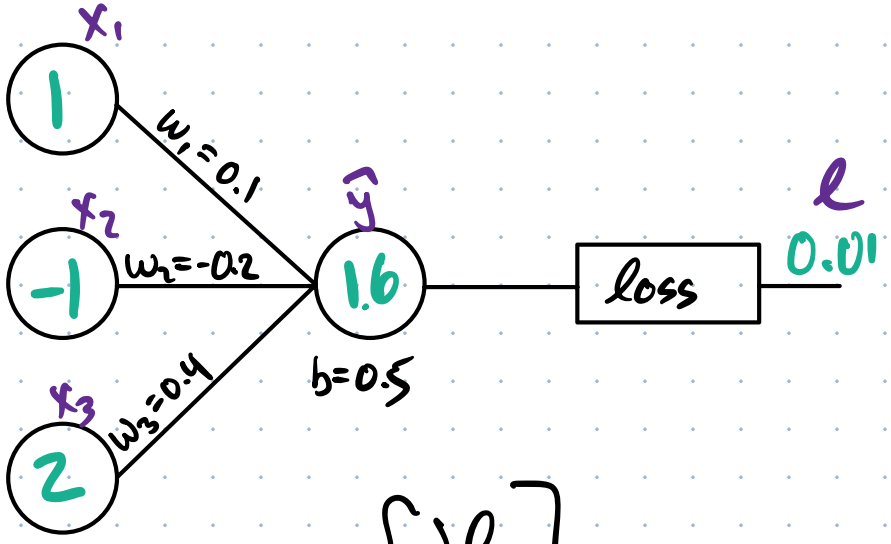
$\hat{y} = 1 \cdot w_1 - 1 \cdot w_2 + 2 \cdot w_3 + b$

So, $\frac{\partial \hat{y}}{\partial w_1} = 1$, $\frac{\partial \hat{y}}{\partial w_2} = -1$, $\frac{\partial \hat{y}}{\partial w_3} = 2$, $\frac{\partial \hat{y}}{\partial b} = 1$

$\frac{\partial \ell}{\partial w_1} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1} = 0.2 \cdot 1 = 0.2$

chain rule!

$\frac{\partial \ell}{\partial w_2} = -0.2$ $\frac{\partial \ell}{\partial w_3} = 0.4$ $\frac{\partial \ell}{\partial b} = 0.2$



Training data: $(1, -1, 2) \to 1.5$

$$-\nabla NN = \begin{bmatrix} \frac{\partial \ell}{\partial w_1} \\ \frac{\partial \ell}{\partial w_2} \\ \frac{\partial \ell}{\partial w_3} \\ \frac{\partial \ell}{\partial b} \end{bmatrix} = \begin{bmatrix} -0.2 \\ 0.2 \\ -0.4 \\ -0.2 \end{bmatrix}$$

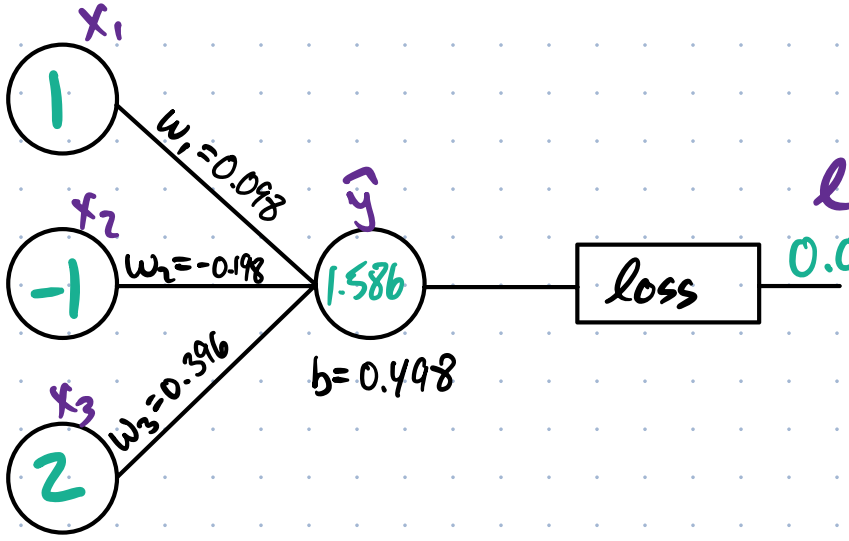
Now adjust each weight a small amount in this direction. Let's do $\frac{1}{100}$.

w_1: $0.1 \to 0.098$

w_2: $-0.2 \to -0.198$

w_3: $0.4 \to 0.396$

b: $0.5 \to 0.498$



Training data: $(1, -1, 2) \to 1.5$

(prev. 0.01)

Now repeat. Compute $-\nabla NN$ with these altered weights, adjust by $\frac{1}{100}$ of it, and so on.

* Python demo.