

NNs

---

Henny Ons

---

---

---



## Prerequisites

- Basic Linear algebra
- Multivariable calculus
- Differential
- Jacobian's / gradients
- Basic ML knowledge

## Agenda

- Multivariable calculus review
- Neurons as a function
- Jacobians & NNs
- Gradient descent
- Back propagation
- BP as Matrix calculus

## Basic Notation

Define  $m = \#_{\text{data}}$  of training set

Define  $n = \# \text{ of input variables } x_1, x_2, \dots$

Define  $L = \# \text{ of layers in NN}$

Define  $l = \text{specific layer}$

$w^L = \text{weight}(l)$   $b^L = \text{bias}$  going into layer  $l$



Define  $x_i = \text{single input variable}$

## Big picture

NNs are functions (Linear functions of weights & biases)

Big calculus problem (Linear approximations)

$$\rightarrow \frac{\partial \text{cost}}{\partial w} \quad (\text{want this})$$

Find how much

each weight & bias

contributes to the

cost

$$x_1 \rightarrow$$

$$x_2 \rightarrow$$

$$x_3 \rightarrow$$

$$x_4 \rightarrow$$

$$x_n \rightarrow$$

with respect to  
every weight and  
bias

## Matrix Calculation Review

Gradients, Jacobians, Jacobian Chain Rule

Vector function

$$\text{eg. } f(x, y) = x^2 + \cos y$$

= Gradient is a horizontal vector of partial derivatives

$$\begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x \\ -\sin(y) \end{bmatrix} = \nabla f(x, y)$$

Notation:  
upside down triangle  
Triangle for change  
Gradient of  $f(x, y)$

$$f(x, y) = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 2x + y^3 \\ e^y - 13x \end{bmatrix} \rightarrow \text{Breaks up into 2 scalar functions}$$

$$\mathbb{R}^2 \rightarrow \mathbb{R}^2$$

$$\begin{aligned} \textcircled{1} \quad f_1 &= 2x + y^3 \rightarrow \frac{\partial f_1}{\partial x} = 2 \quad \frac{\partial f_1}{\partial y} = 3y^2 \\ \textcircled{2} \quad f_2 &= e^y - 13x \rightarrow \frac{\partial f_2}{\partial x} = -13 \quad \frac{\partial f_2}{\partial y} = e^y \end{aligned}$$

small functions  
composing one large function

## Create Jacobian

denoted also as:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} \begin{bmatrix} \nabla f_1 \\ \nabla f_2 \end{bmatrix} = \begin{pmatrix} 2 & 3y^2 \\ -13 & e^y \end{pmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

= Transposed  
defined rigorously from original function

Gradients:  
 $\mathbb{R}^n \rightarrow \mathbb{R}^1$  vector to scalar

$$x = 2, y = 0$$

$$\begin{bmatrix} x \\ y \end{bmatrix} \xrightarrow{f(x, y)} s \text{ for scalar}$$

$$\text{eg. } \begin{bmatrix} 2 \\ 0 \end{bmatrix} = (5)$$

$$\mathbb{R}^2 \rightarrow \mathbb{R}^1$$

Jacobian:

$$\mathbb{R}^n \rightarrow \mathbb{R}^m$$

partial derivative,

$$\frac{\partial f}{\partial x} = 2x$$

$$\frac{\partial f}{\partial y} = -\sin(y)$$

vector to another vector

Since we can vary the input size and size of the Jacobian.

From last page

$$f(x, y) = \begin{bmatrix} 2x - y^3 \\ e^x - 13y \end{bmatrix}$$

$$\frac{\partial f_1}{\partial x} = 2 \quad \frac{\partial f_1}{\partial y} = -3y^2$$

$$\frac{\partial f_2}{\partial x} = e^x \quad \frac{\partial f_2}{\partial y} = -13$$

$$\mathbb{R}^2 \Rightarrow \mathbb{R}^2$$

$$f_1 = 2x - y^3$$

$$f_2 = e^x - 13y$$

$$\left. \begin{array}{l} \frac{\partial f_1}{\partial x} = 2 \\ \frac{\partial f_1}{\partial y} = -3y^2 \\ \frac{\partial f_2}{\partial x} = e^x \\ \frac{\partial f_2}{\partial y} = -13 \end{array} \right\} = J = \begin{bmatrix} 2 & -3y^2 \\ e^x & -13 \end{bmatrix}^T = \begin{bmatrix} \nabla f_1^T \\ \nabla f_2^T \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix}$$

Transpose the matrix to "factor" out  $x$  &  $y$ .

### Scalar Chain Rule

Let  $f(x) = \sin(x^2)$

$$f'(x) = 2x \cos(x^2)$$

Let  $g = x^2$

$f = \sin(g)$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} = 2x \cos g \quad \text{technical breakdown}$$

Now, extend to:  $f = g \cdot h(c(a \cdot x))$

as a general concept: use product of partial derivatives

① How does  $g$  change as  $\begin{bmatrix} x \\ y \end{bmatrix}$  change?

$$\vec{x} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \frac{\partial \vec{g}}{\partial \vec{x}} = \begin{bmatrix} 2x & 1 \\ 0 & 3y^2 \end{bmatrix}$$

② How does  $f$  change as  $(g_1, g_2)$  change

$$\frac{\partial \vec{f}}{\partial \vec{g}} = \begin{bmatrix} \cos(g_1) & 0 \\ 0 & \frac{1}{g_2} \end{bmatrix}$$

③ Apply the Scalar chain rule

vector notation

$$\frac{\partial \vec{f}}{\partial \vec{x}} = \frac{\partial \vec{f}}{\partial \vec{g}} \frac{\partial \vec{g}}{\partial \vec{x}} = \begin{bmatrix} \cos(g_1) & 0 \\ 0 & \frac{1}{g_2} \end{bmatrix} \begin{bmatrix} 2x & 1 \\ 0 & 3y^2 \end{bmatrix}$$

Matrix multiplication

$$\frac{\partial \vec{f}}{\partial \vec{x}} = \begin{bmatrix} 2x \cos(g_1) & \cos(g_1) \\ 0 & \frac{3y^2}{g_2} \end{bmatrix} = \begin{bmatrix} 2x \cos(x^2 + y) & \cos(x^2 + y) \\ 0 & \frac{3}{y} \end{bmatrix}$$

### Jacobian Chain Rule

Define the following:

$$f(x, y) = \begin{bmatrix} \sin(x^2 + y) \\ \ln(y^3) \end{bmatrix}$$

$$g = \begin{bmatrix} x^2 + y \\ y^3 \end{bmatrix}_{g_1, g_2} \quad f = \begin{bmatrix} \sin(g_1) \\ \ln(g_2) \end{bmatrix}$$

In side functions      out side functions

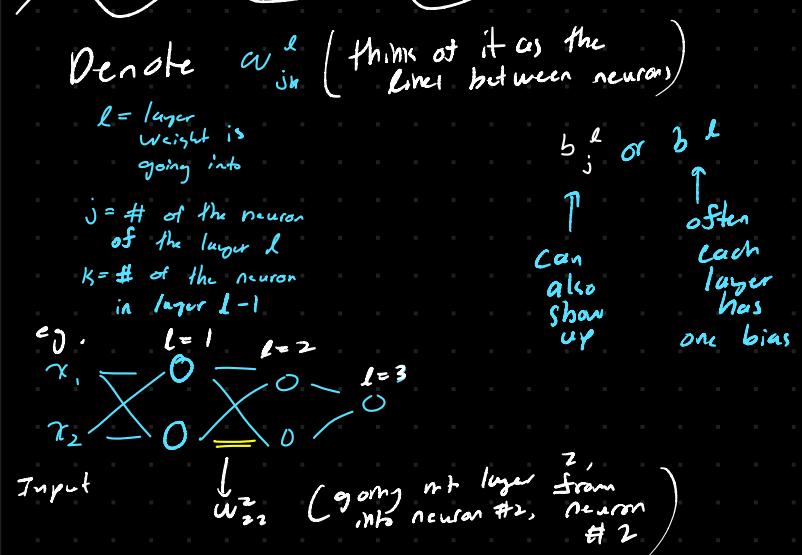
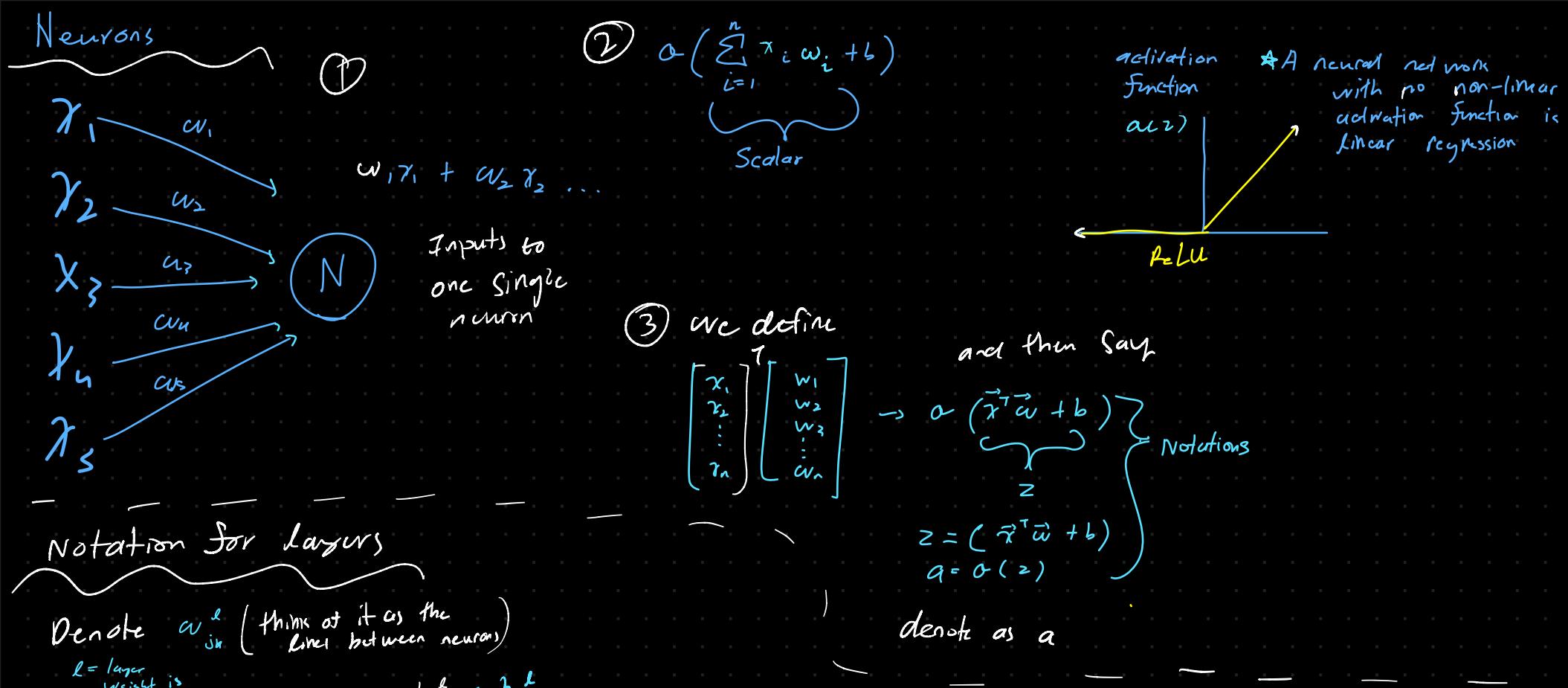
what if we cannot use chain rule?

$$f(x, y) = \begin{bmatrix} \sin(x^2 + y^2) \\ \cos(g_2^3) \\ xy^3 \end{bmatrix}$$

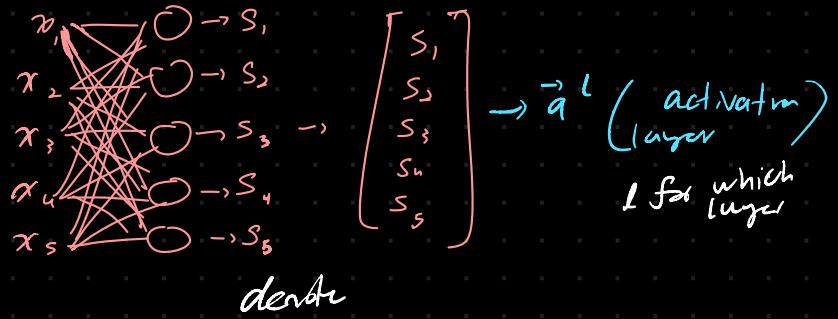
$$g = \begin{pmatrix} x^2 + y^2 \\ y^3 \\ 1 \end{pmatrix}$$

Leave as is if no intermediate

put all when you are not using chain rule aka / no intermediate function



For one layer: Define scalars



( $a$  for activation)      input goes into

of weights

From single neuron to full layer:

Single node

$$\alpha(z^T w + b)$$

Entire layer

$W^l$

$w_{jk}^l$  is equivalent to:

Define # of neurons in layer =  $k = j$  (usually square)  
 in training examples

$$j \left[ \begin{array}{c} a^1 \\ \vdots \\ a^k \end{array} \right] \leftarrow k \left[ \begin{array}{c} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{array} \right]$$

comes from the  $k$ th neuron

and goes into the  $j$ th neuron

e.g.

-  $\alpha$ :  $\alpha(w^l \cdot a^{l-1} + b^l)$

-  $W^l$ : construct the weights matrix

$$W^l = \begin{bmatrix} (1,1) & (1,2) \\ (2,1) & (2,2) \end{bmatrix} = \begin{bmatrix} 5 & 2 \\ 3 & 6 \end{bmatrix}$$

$$\alpha(w^l \cdot a^{l-1} + b^l)$$

activation

of the

previous

layer

calculates  
 = the  
 "weighted"  
 sum of  
 the input

$$= \begin{bmatrix} 5 & 2 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$z = \begin{bmatrix} 90 \\ 150 \end{bmatrix} + \begin{bmatrix} 6 \\ 6 \end{bmatrix} = \begin{bmatrix} 96 \\ 156 \end{bmatrix}$$

$$\alpha(z) = \alpha\left(\begin{bmatrix} 96 \\ 156 \end{bmatrix}\right)$$

### Activation Function Calculation

General:  $a_n = \alpha(w^T a^n + b^l)$

$$a_0 \Rightarrow \alpha(w^T a^0 + b^l)$$

from  $a_0$ ,  
 $a_1 = \alpha(w^T a^1 + b^l)$

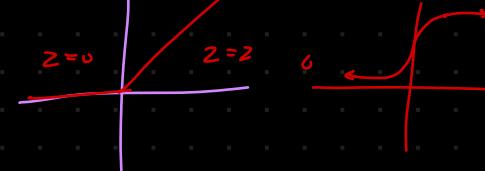
vector  $\rightarrow$  vector

output is  
 a vector

$$= \begin{bmatrix} 96 \\ 156 \end{bmatrix}$$

$$\alpha\left(\begin{bmatrix} 96 \\ 156 \end{bmatrix}\right) = \begin{bmatrix} 96 \\ 156 \end{bmatrix}$$

relu      Sigmoid



feed forward through  
 an NN  
 input through the network

$$O - O - O$$

$$O - O -$$

forward

$$O - O -$$

backward

$$\alpha(z) = \begin{bmatrix} 96 \\ 156 \end{bmatrix}$$

stays  
 the same

## Jacobians of NNs

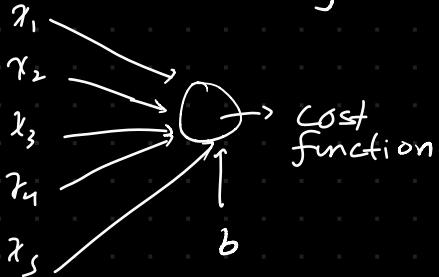
Minimize the cost of the output (error)

Mean Squared error

$$\frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2$$

↓ answer we got  
real answer

$\pm$  does not matter (for the power rule)  
 $m$  is the # of training examples



## Jacobians of Operations

Binary element-wise operations

vector to vector

$F(\vec{v}, \vec{w}) \rightarrow \vec{z}$

$$\textcircled{1} \quad \begin{bmatrix} \vec{v} \\ \vec{w} \end{bmatrix} = \begin{bmatrix} \vec{z} \end{bmatrix}$$

element wise operators  
(can be any kind, here if general)

Hadamard product ( $\otimes$ )  
is a binary element-wise operation.

$$\textcircled{2} \quad J = \frac{\partial F}{\partial \vec{v}} = \begin{bmatrix} v_1 \otimes w_1 \\ v_2 \otimes w_2 \\ v_3 \otimes w_3 \\ \vdots \\ v_n \otimes w_n \end{bmatrix} = \begin{bmatrix} b \end{bmatrix}$$

## ② $F(\vec{v}, \vec{w})$ (general)

$$= (f(\vec{v})) \circ g(\vec{w}) = \begin{bmatrix} f_1(\vec{v}) \circ g_1(\vec{w}) \\ f_2(\vec{v}) \circ g_2(\vec{w}) \\ \vdots \\ f_n(\vec{v}) \circ g_n(\vec{w}) \end{bmatrix}$$

operations on the functions before

Can get 2 Jacobians  
one for each vector

$$\frac{\partial F}{\partial \vec{w}} \quad \frac{\partial F}{\partial \vec{v}}$$

first element of  $v$ , second of  $w$

$$\begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{bmatrix}$$

In the most general form (if  $F(\vec{v})$  has multiple element wise types of operations)

General representation  
increase  $f_i$  and  $g_i$ , don't change the vector

$$\begin{bmatrix} \frac{\partial}{\partial v_1} f_1(\vec{v}) \circ g_1(\vec{w}) \\ 0 \\ \frac{\partial}{\partial v_2} f_2(\vec{v}) \circ g_2(\vec{w}) \\ 0 \\ \vdots \\ \frac{\partial}{\partial v_n} f_n(\vec{v}) \circ g_n(\vec{w}) \\ 0 \end{bmatrix}$$

If  $f(\vec{v}) = \vec{v}$  and  $g(\vec{w}) = \vec{w}$

Scalar:

$$\begin{bmatrix} \frac{\partial}{\partial v_1} (\vec{v}) \circ (\vec{w}) & 0 & \dots & 0 \\ 0 & \frac{\partial}{\partial v_2} (\vec{v}) \circ (\vec{w}) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial}{\partial v_n} (\vec{v}) \circ (\vec{w}) \end{bmatrix}$$

## Derivatives of Neuron Operators (Hadamard product)

→ Hadamard product

$$F(\vec{v}, \vec{w}) = \begin{bmatrix} f_1(\vec{v}) \otimes g_1(\vec{w}) \\ f_2(\vec{v}) \otimes g_2(\vec{w}) \\ \vdots \\ f_n(\vec{v}) \otimes g_n(\vec{w}) \end{bmatrix} = \begin{bmatrix} v_1 \otimes w_1 \\ v_2 \otimes w_2 \\ \vdots \\ v_n \otimes w_n \end{bmatrix} \Rightarrow \text{much better description}$$

$$\frac{\partial F}{\partial \vec{v}} = \begin{bmatrix} \frac{\partial F_1}{\partial v_1} & \frac{\partial F_1}{\partial v_2} & \dots & \frac{\partial F_1}{\partial v_n} \\ \frac{\partial F_2}{\partial v_1} & \frac{\partial F_2}{\partial v_2} & \dots & \frac{\partial F_2}{\partial v_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial v_1} & \frac{\partial F_n}{\partial v_2} & \dots & \frac{\partial F_n}{\partial v_n} \end{bmatrix} = \begin{bmatrix} cw_1 & 0 & \dots & 0 \\ 0 & cw_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & cw_n \end{bmatrix}$$

\* For all element wise functions, the Jacobian will be a diagonal matrix

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \rightarrow \text{diag}(a, b, c)$$

$$\frac{d}{dt}(xw) = x \quad (\text{easier way to understand})$$

## Derivative of a Scalar expansion

$$2 \begin{bmatrix} \vec{v} \\ 1 \end{bmatrix} = \begin{bmatrix} 2\vec{v}_1 \\ 2\vec{v}_2 \\ 2\vec{v}_3 \\ \vdots \\ 2\vec{v}_n \end{bmatrix}$$

$$F(\vec{v}, x) = f(\vec{v}) \circ g(x)$$

Defining for  $g(\vec{u}) = \vec{I} \cdot \vec{u}$  (expands into a vector)

The act of multiplying  $\vec{x}$  by the ones vector is an act (expands to a diag matrix or a Jacobian) of broadcasting itself

$$\begin{bmatrix} 2 \\ \vec{v} \\ 2 \end{bmatrix} \circ \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

When we get the Jacobian, we get a gradient if  $x$  is a scalar

$$\frac{\partial F}{\partial \vec{v}} = \begin{bmatrix} f_1(\vec{v}) \circ g_1(\vec{x}) \\ f_2(\vec{v}) \circ g_2(\vec{x}) \\ \vdots \\ f_n(\vec{v}) \circ g_n(\vec{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial v_1} & \frac{\partial f_1}{\partial v_2} & \cdots & \frac{\partial f_1}{\partial v_n} \\ \frac{\partial f_2}{\partial v_1} & \frac{\partial f_2}{\partial v_2} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial v_1} & \frac{\partial f_n}{\partial v_2} & \cdots & \frac{\partial f_n}{\partial v_n} \end{bmatrix}$$

$$\text{Gradient with respect to } x \quad \nabla F_x = \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \\ \vdots \\ \frac{\partial f_n}{\partial x} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

don't forget, this is the scalar vector.

$$\begin{bmatrix} x & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & x \end{bmatrix}$$

$g$  is element wise

$$\text{Derivative of a Sum}$$

e.g.  $\vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \sum_{i=1}^n (g_i(\vec{v}))_i = \begin{bmatrix} \text{---} s \text{ ---} \end{bmatrix}$  for  $s$  to denote the sum

$$= \begin{bmatrix} \frac{\partial s}{\partial v_1} & \frac{\partial s}{\partial v_2} & \cdots & \frac{\partial s}{\partial v_n} \end{bmatrix}$$

$$J = \left[ \frac{\partial}{\partial v_i} \sum_{i=1}^n g_i(v) \right] = \frac{\partial}{\partial v_1} \sum_{i=1}^n g_i(v) \cdots \frac{\partial}{\partial v_n} \sum_{i=1}^n g_i(v)$$

Easy to generalize

\* The derivative of a sum is the sum of the derivatives

$$J = \left[ \sum_{i=1}^n \left( \frac{\partial}{\partial v_i} g_i(v) \right) \right] = \left[ \sum_{i=1}^n \frac{\partial}{\partial v_1} g_i(v) \cdots \sum_{i=1}^n \frac{\partial}{\partial v_n} g_i(v) \right]$$

If  $g(\vec{v}) = 2\vec{v}$

Consider  $g(\vec{v}) = \vec{v}$

$$J = \left[ \sum_{i=1}^n \frac{\partial}{\partial v_i} (v_i) \sum_{i=1}^n \frac{\partial}{\partial v_2} (v_i) \cdots \sum_{i=1}^n \frac{\partial}{\partial v_n} (v_i) \right] = \begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix}$$

Basically, a horizontal line of what  $g$  does if it does anything

# Derivative of a neuron

activation  
 $a = \sigma(\vec{w}^T \vec{x} + b)$

$\vec{z}$  dot product  
 = Hadamard product

wrt  
 $\frac{\partial a}{\partial w} = \frac{\partial a}{\partial z} \frac{\partial z}{\partial w}$

$\frac{\partial a}{\partial b} = \frac{\partial a}{\partial z} \frac{\partial z}{\partial b}$

Define an activation slice:

$a = \sigma(\text{sum}(w \otimes x) + b)$

$\vec{z}$  Hadamard product

$\vec{w}$  SCH

$\frac{\partial a}{\partial w} = \frac{\partial a}{\partial z} \frac{\partial z}{\partial s} \frac{\partial s}{\partial H} \frac{\partial H}{\partial w}$

either 0 or 1

$1^T$  previous slide

$$[-] \begin{bmatrix} 1 \\ | \\ | \\ m \end{bmatrix}, [-] \begin{bmatrix} 1 \\ | \\ | \\ n \end{bmatrix}$$

$\sigma(\sum(w \otimes x) + b)$

$\vec{z}$

$\frac{\partial z}{\partial b} = 1$

By Jacobian convention,  $H = w \otimes x$

①  $\frac{\partial H}{\partial w} = \begin{bmatrix} x_1 & \dots & 0 \\ \vdots & x_2 & \vdots \\ 0 & \dots & x_n \end{bmatrix}$   
 diag ( $x_1, \dots, x_n$ )

②  $Z = S(H) + b$   
 $\frac{\partial Z}{\partial H} = 1$

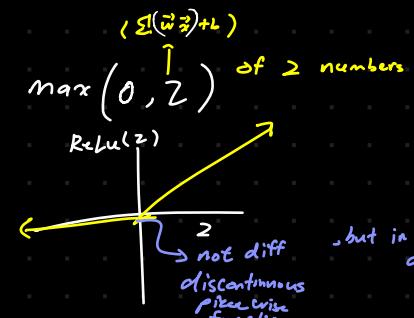
③  $Z = S^{-1}$

$\frac{\partial Z}{\partial s} = 1$   
 for  $SCH$   
 $\frac{\partial s}{\partial H} = (1 1 1 1 1)$  (Scalar)

④ Derivative of ReLU wrt the activation

$\frac{\partial a}{\partial z} = \frac{\partial a}{\partial z} [1^T]$

Simplifies  
 after all  
 3 substitutions



Need to find this

To solve, we split up into piecewise functions

taking the derivative

$$\frac{\partial a}{\partial w} = \max \left\{ \begin{array}{l} z \leq 0, 0 \rightarrow 0 \times \frac{\partial z}{\partial w} = 0 = [0] \\ z > 0, z \rightarrow 1 \times \frac{\partial z}{\partial w} = \frac{\partial z}{\partial w} = [1^T] \end{array} \right.$$

For some  $w$ ,  $\frac{\partial a}{\partial w}$  gives the best approximation for the instant rate of change of the activation, wrt that specific  $w$   
 also putting all derivatives together

important so that the vector operations make sense dimensionally

$\frac{\partial a}{\partial w} = [0 0 0 \dots 0]$

or  
 $\frac{\partial a}{\partial w} = [x_1, x_2, x_3, \dots, x_n]$

depending on  $\vec{w}^T \vec{x} + b$

\* The derivative of the cost wrt a weight is directly proportional to the input  $\vec{x}$

# The Gradient of the Loss Function

Cost function

$$MSE : \frac{1}{2m} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$\downarrow$  answer activation  $a^L$



We want to find:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a} \frac{\partial a}{\partial w} \quad (\text{Chain rule})$$

$\frac{\partial C}{\partial b}$  know this already

Recall that

$$\frac{\partial a}{\partial w} = \begin{cases} [0]^T & \text{if } \vec{w}^T \vec{x} + b \leq 0 \\ [1]^T & \text{if } \vec{w}^T \vec{x} + b > 0 \end{cases}$$

$$\frac{\partial a}{\partial b} = \begin{cases} [0]^T & \text{if } \vec{w}^T \vec{x} + b \leq 0 \\ [1]^T & \text{if } \vec{w}^T \vec{x} + b > 0 \end{cases}$$

$$\frac{1}{m} \left\{ \begin{array}{l} \vec{0}^T \\ \sum_{i=1}^m (\vec{w}^T \vec{x} + b - y_i) \vec{x}^T \end{array} \right\}$$

$$\text{one training example} \quad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

$$\frac{1}{2m} \sum_{i=1}^m (y_i - a^L)^2$$

$$\text{Several training examples } C_m \quad \begin{bmatrix} x_{11} & x_{21} & \dots & x_{n1} \\ x_{12} & x_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ x_{1n} & x_{2n} & \dots & x_{nn} \end{bmatrix}$$

$$\text{Define and say: } v = \underset{\text{correct answer}}{y} - a^L$$

$$\frac{1}{2m} \sum_{i=1}^m (y_i - a^L)^2 = \frac{1}{2m} \sum_{i=1}^m (v)^2$$

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial v} \frac{\partial v}{\partial a} \frac{\partial a}{\partial w}$$

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial v} \frac{\partial v}{\partial w}$$

$$\frac{\partial a}{\partial w}$$

From ReLU,

$$\theta(w^T x + b) \rightarrow \max(0, w^T x + b)$$

$$\Rightarrow \frac{1}{m} \sum_{i=1}^m \left\{ \begin{array}{l} [0]^T \quad \text{if } ... \\ -\{y_i - (\max(0, w^T x + b))\} \vec{x}^T \quad \text{if } w^T x + b > 0 \end{array} \right.$$

$$= -(y_i - (w^T x + b)) \vec{x}^T$$

$$\frac{\partial C}{\partial w} = \frac{1}{m} \sum_{i=1}^m (\vec{w}^T \vec{x} + b - y_i) \vec{x}^T$$

= scalar

$$\text{① } \frac{\partial v}{\partial w} = \frac{\partial}{\partial w} (y - a^L) \quad \text{don't forget that } L \text{ is not actually a constant in the equation}$$

$$\frac{\partial v}{\partial w} = -(\vec{1})(\frac{\partial a}{\partial w})$$

$$\text{② } = \frac{\partial}{\partial w} \left( \frac{1}{2m} \sum_{i=1}^m (v)^2 \right)$$

$$\begin{aligned} \text{Take out the constant} &= \frac{1}{2m} \left( \sum_{i=1}^m \frac{\partial}{\partial w} (v^2) \right) \left( \frac{\partial v^2}{\partial v} = \frac{\partial v^2}{\partial v} \frac{\partial v}{\partial w} \right) \\ &= \frac{1}{2m} \sum_{i=1}^m 2v \frac{\partial v}{\partial w} \\ &= \frac{1}{m} \sum_{i=1}^m (v) \left( \frac{\partial v}{\partial w} \right) \left( \frac{\partial v}{\partial w} = -\frac{\partial a}{\partial w} \right) \\ \text{what does this mean? taking } \vec{v} \text{ at a piecewise} & \\ \downarrow & \text{make the options negative} \\ = \frac{1}{m} \sum_{i=1}^m \left\{ \begin{array}{l} [0]^T \quad \text{if } ... \\ -v \frac{\partial a}{\partial w} \quad \text{if } ... \end{array} \right. & \text{★ DOES not actually change the piecewise} \end{aligned}$$

$$\text{Subbing in the variables,} = \frac{1}{m} \sum_{i=1}^m \left\{ \begin{array}{l} [0]^T \quad \text{if } ... \\ -(\vec{y}_i - a^L) \vec{x}^T \quad \text{if } ... \end{array} \right.$$

Very important to understand that it is just attached to the result of the piecewise

if  $w^T x + b > 0$   
always the case if it went down this path.

max function is redundant

What did we find out?  
 $\frac{\partial C}{\partial w} = \frac{1}{m} \sum_{i=1}^m (w^T x + b - y_i) x^i$  if  $w^T x + b \leq 0$   
 Want a Jacobian for back propagation

$$\left[ \begin{array}{c} \frac{\partial C}{\partial w_1} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{array} \right] \text{ called total weights}$$

$$\Rightarrow \frac{1}{m} \sum_{i=1}^m e_i(x)$$

where  $e_i = w^T x - y_i$  (scalar)

Consider  $m=1$  and 5 inputs, transposing our  $e_i | x^T$  so it becomes a column,

$$\frac{\partial C}{\partial w} = \begin{bmatrix} e_1 x_1 \\ e_2 x_2 \\ \vdots \\ e_5 x_5 \end{bmatrix} \frac{\partial C}{\partial w_1} \text{ has } C \text{ changes wrt } w_1, \dots \text{ how } C \text{ changes wrt } w_2, \dots$$

when you change  $w_i$  by some amount, the corresponding change is ...

$$\begin{aligned} \text{consider } m \sum_{i=1}^m e_i x^T \\ = \frac{1}{m} e_1 x^T + e_2 x^T + e_3 x^T \dots e_5 x^T \\ = \frac{1}{m} \left( \begin{bmatrix} e_1 x_1 \\ e_2 x_2 \\ \vdots \\ e_5 x_5 \end{bmatrix} + \begin{bmatrix} e_1 x_1 \\ e_2 x_2 \\ \vdots \\ e_5 x_5 \end{bmatrix} + \dots + \begin{bmatrix} e_1 x_1 \\ e_2 x_2 \\ \vdots \\ e_5 x_5 \end{bmatrix} \right) \end{aligned}$$



Consider a general case

$$\frac{1}{m} \begin{bmatrix} e_1 x_1 + e_2 x_1 \dots e_5 x_1 \\ e_1 x_2 + e_2 x_2 \dots e_5 x_2 \\ e_1 x_3 + e_2 x_3 \dots e_5 x_3 \\ e_1 x_4 + e_2 x_4 \dots e_5 x_4 \\ e_1 x_5 + e_2 x_5 \dots e_5 x_5 \end{bmatrix}$$

random averages

$$\Rightarrow \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \end{bmatrix} = \frac{\partial J}{\partial w} \text{ all } w's, \text{ averaged out over } \underline{\text{all training examples}}$$

## Derivative of the bias

$$\frac{\partial C}{\partial b} = \frac{1}{2m} \sum_{i=1}^m (\gamma - a^L)^2$$

$$v = \gamma - a^L \quad (-1) \quad \checkmark$$

From  $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial v} \frac{\partial v}{\partial a^L} \frac{\partial a^L}{\partial b}$

$$= \frac{1}{m} \sum_{i=1}^m v \begin{cases} 0 & \text{if } \dots \\ -1 & \text{if } \dots \end{cases} \quad (\text{plug into our piecewise})$$

$$= \frac{1}{m} \sum_{i=1}^m \begin{cases} 0 & \text{if } \dots \\ -(v - a^L) & \text{if } \dots \end{cases}$$

$\downarrow$   
max redundancy again, can remove

$$\frac{\partial v}{\partial b} = -1 \begin{cases} 0 & \text{if } w^T x + b \leq 0 \\ 1 & \text{if } w^T x + b > 0 \end{cases}$$

\* scalar \*

$$= \frac{1}{m} \sum_{i=1}^m \begin{cases} (w^T x + b - \gamma) & \dots \end{cases}$$

$$\frac{\partial C}{\partial b} = \frac{1}{2b} \left( \frac{1}{2m} \sum_{i=1}^m (v^2) \right)$$

$$= \frac{1}{2m} \left( \sum_{i=1}^m \frac{\partial^2}{\partial b^2} (v^2) \right) \left( \frac{\partial v}{\partial b} \right) \left( \frac{\partial v}{\partial b} \right)$$

$$= \frac{1}{2m} \sum_{i=1}^m \cancel{v} \frac{\partial v}{\partial b}$$

$$= \frac{1}{m} \sum_{i=1}^m v \frac{\partial v}{\partial b} \rightarrow \text{piecewise}$$

$$\frac{\partial v}{\partial b} = \frac{\partial v}{\partial a^L} \frac{\partial a^L}{\partial b}$$

$$= (-1) \begin{cases} 0 & \text{if } \dots \\ 1 & \text{if } \dots \end{cases}$$

Now generally,

$$= \begin{cases} 0 & \text{if } \dots \\ \frac{1}{m} \sum_{i=1}^m (w^T x + b - \gamma) & \text{if } \dots \end{cases}$$

$e_i$

$$\frac{\partial C}{\partial b} = \frac{1}{m} \sum_{i=1}^m e_i$$

derivative of the cost wrt to the bias, averaged over all training examples.

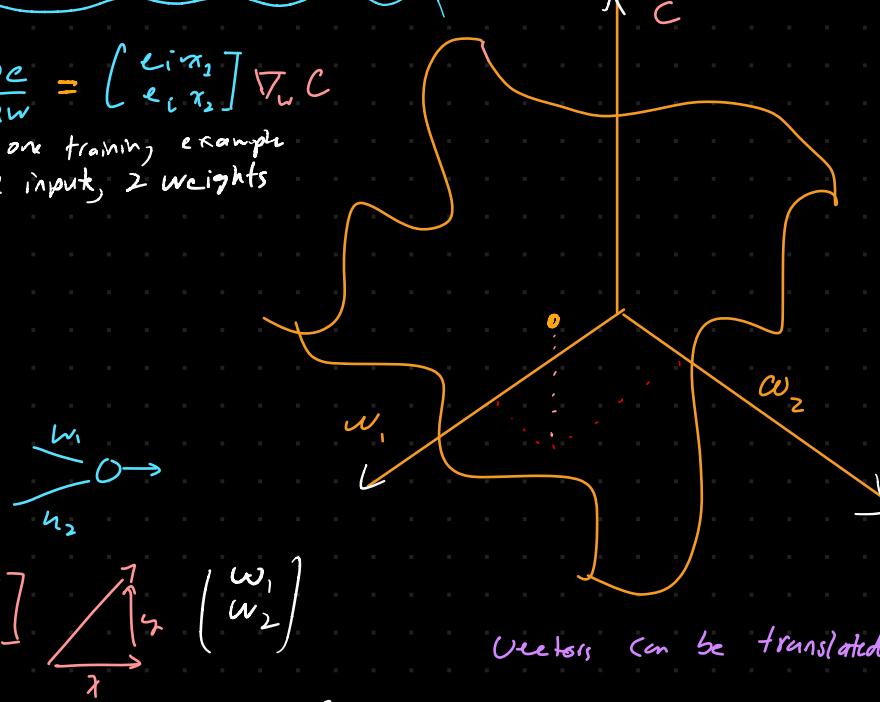
$$\boxed{\frac{\partial C}{\partial b} = \frac{1}{m} [e_1 + e_2 + \dots + e_n]}$$

Just a scalar

## Gradient Descent Intuition

$$\frac{\partial C}{\partial w} = \begin{bmatrix} e_i x_1 \\ e_i x_2 \end{bmatrix} \nabla_w C$$

one training example  
2 inputs, 2 weights



The gradient of a function points in the direction of steepest ascent

The value of the input decides the derivative  
want to change the weights of the corresponding input that causes the greatest effect on the cost function

$\begin{bmatrix} e_i x_1 \\ e_i x_2 \end{bmatrix}$  will point in the direction of the weights that have the most impact

$\nabla_w C$  goes in a higher direction of cost (evaluate  $e_i x_i$  first)  
error is magnified

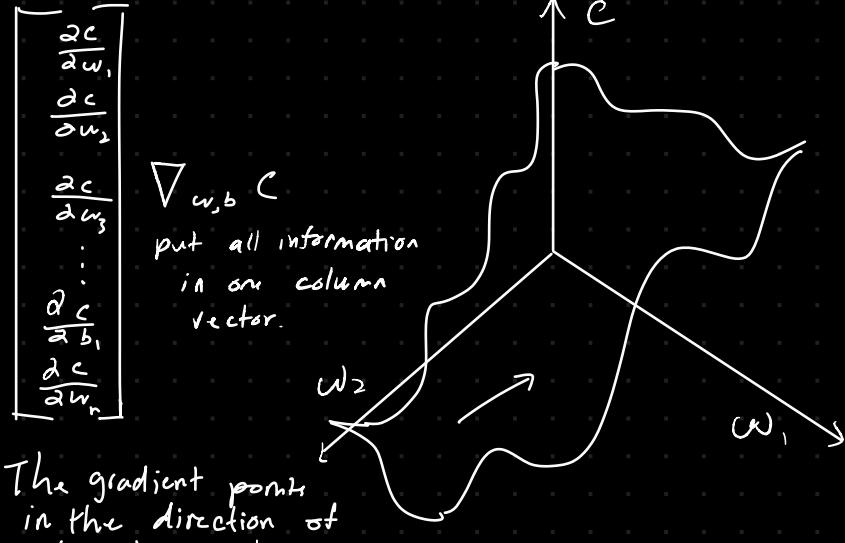
- $\nabla_w C$  tells us how to most quickly decrease the cost

The vector points in the direction that have the most impact (cost activation is  $x_i$ , error is magnified by  $x_i$ )

This we want to do  
gradient descent in the opposite direction

All machine learning models optimize to point in the direction of steepest descent

## Gradient Descent & Stochastic Gradient Descent



$\theta$  (defined to be all weights and biases)

Initially starts as randomly assigned

$$\theta = \theta - \alpha \nabla_{w,b} C$$

↓  
Learning rate (Step size)

$$\begin{bmatrix} w_1 - \alpha \frac{\partial C}{\partial w_1}, \\ w_2 - \alpha \frac{\partial C}{\partial w_2}, \\ \vdots \\ w_n - \alpha \frac{\partial C}{\partial w_n}, \\ b_1 - \alpha \frac{\partial C}{\partial b_1}, \\ b_2 - \alpha \frac{\partial C}{\partial b_2}, \\ \vdots \\ b_r - \alpha \frac{\partial C}{\partial b_r} \end{bmatrix}$$

$\alpha$  (Learning rate)  
depends on research

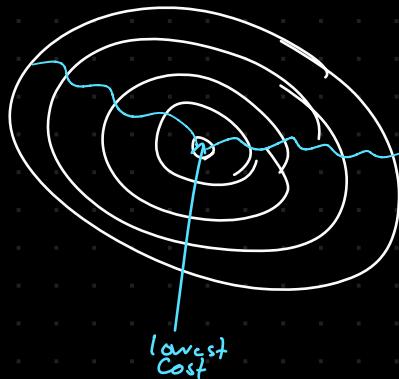
Training batch: m  
↳ Can be very computationally expensive.

$$\begin{bmatrix} 1 \\ \frac{1}{m} \\ \vdots \\ 1 \end{bmatrix}$$

over m examples

Mini-batch Gradient descent:

SGD → batching smaller datasets and then, running gradient descent on each batch  
Shuffle the data well. Batch should represent data as a whole.

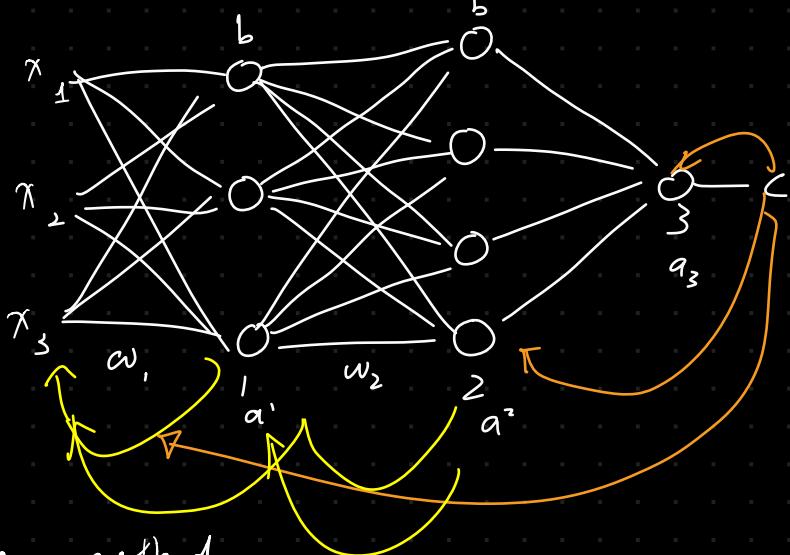


$$\left\{ x_1 \right\} \quad \left\{ x_2 \right\}$$

↓      ↓  
SGD   SGD

then, you can take as many steps, as you have batches instead of one step over all examples.

Finding derivatives of entire layers (why it is also different from one neuron)



Back propagation - saves efficiency by not having to go back and forth

Accumulate our "error"

One method

$$\frac{\partial C}{\partial w_1} = \left( \frac{\partial a'}{\partial w_1} \right) \left( \frac{\partial a^2}{\partial a_1} \right) \left( \frac{\partial a^3}{\partial a^2} \right) \left( \frac{\partial C}{\partial a^3} \right)$$

$$\left( \frac{\text{first layer}}{\text{weights}} \right) \left( \frac{\text{second layer}}{\text{first layer}} \right) \left( \frac{\text{third layer}}{\text{second layer}} \right) \left( \frac{\text{cost}}{\text{third layer}} \right)$$

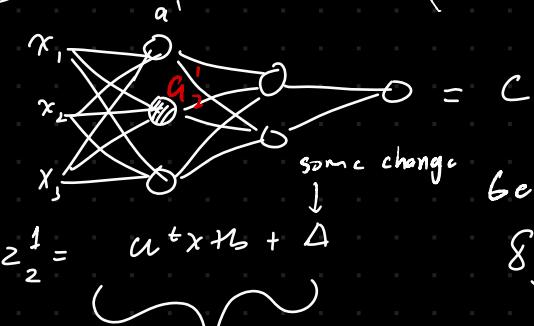
$$\frac{\partial C}{\partial w_2} = \frac{\partial a^2}{\partial w_2} \frac{\partial a^3}{\partial a^2} \frac{\partial C}{\partial a^3}$$

Doing these calculations to compute the derivatives going forward is resource-intensive  
Not often done, but possible

error of node  $j$ , of layer  $l$

$$\delta_j^l$$

The error of a node



$$z_2^1 = a^t x + b + \Delta$$

Activation of node 2 of  
layer 1, before ReLU

$(z_2^1 + \Delta)$  is the "total error"

1st equation - error of last node  $a^L$

$$a^L = \sigma(z^L)$$

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

$$\delta_j^L = \left( \frac{\partial C}{\partial a_j^L} \right) \left( \frac{\partial a_j^L}{\partial z_j^L} \right) \text{ chain rule again}$$

If we add  $\Delta$  to a node,  
how does the cost  
function change?

General form  $\delta_j^L = \frac{\partial C}{\partial z_j^L}$  (impact of a node on the cost)

if close to 0,  
this means that changing the value of  
 $z$  has little effect on the cost  
Vice versa.

How much "error" is being caused  
by a node  
"Bang for your buck"

Now, do the entire last layer:

$$\nabla_{a^L} C \quad (\text{gradient of cost wrt } a^L)$$

$$\delta^L = \nabla_{a^L} C \odot \sigma'(z^L)$$

↑ element-wise derivative  
operation

$$\begin{bmatrix} \frac{\partial C}{\partial a_1^L} \\ \frac{\partial C}{\partial a_2^L} \\ \vdots \\ \frac{\partial C}{\partial a_n^L} \end{bmatrix} \otimes \begin{bmatrix} \sigma'(z_1^L) \\ \sigma'(z_2^L) \\ \vdots \\ \sigma'(z_n^L) \end{bmatrix}$$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L) \quad \text{general form}$$

2nd equation: error of any node:  $\delta^L = ((w^{L+1})^T \otimes \delta^{L+1}) \otimes \sigma'(z^L)$

= weights going into layer  $L+1$

↓ weight matrix connecting  $L, L+1$

↓ total error of  $L+1$  layer

derivatives of the activations

$$\delta^L = \begin{bmatrix} \vdots \\ \delta^L \end{bmatrix}$$

why does this work:

If we know errors at layer  $L+1$

$(w^{L+1})^T$  is like going back and getting that error

e.g. Find  $\frac{\partial C}{\partial z_1}$ , given  $\frac{\partial C}{\partial z_2}$

$$\delta(z_1) + \delta(w_2 a_1 + b_2)$$

given

using the chain rule:

$$\frac{\partial C}{\partial z_1} = \frac{\partial z_2}{\partial z_1} \frac{\partial C}{\partial z_2}$$

$$\frac{\partial C}{\partial z_1} = \frac{\partial a_1}{\partial z_1} \frac{\partial z_2}{\partial a_1} \frac{\partial C}{\partial z_2}$$

$\delta(z_2) \otimes w_2^T \otimes \delta^2$

$$z_1 = w_2 a_1$$

$$\frac{\partial z_1}{\partial a_1} = w_2^T \leftarrow \text{Transposed}$$

$$\delta^{L+1} \quad C = \begin{bmatrix} \vdots \\ \delta^{L+1} \end{bmatrix}_j \quad u^T = \begin{bmatrix} \vdots \\ u^T \end{bmatrix}_K$$

Recursing through the networks

Generally, derivative of matrix-vector product wrt the vector is,

$$\frac{\partial Ab}{\partial b} = A^T$$

works for any layer

Equation 3 - derivative of cost wrt any bias

$$\frac{\partial C}{\partial b_1} = \delta^L$$

$$\begin{bmatrix} 4 \\ 4 \\ 4 \\ 4 \\ 4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \end{bmatrix}$$

Given  $\sigma(w, a_0 + b_1) \Rightarrow \sigma(w_2 a_1 + b_2)$

$$\delta^L = \frac{\partial C}{\partial z_1}$$

given  $\delta'$  (error of  $z_1$ )

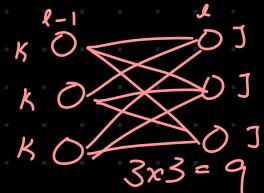
$$\frac{\partial C}{\partial b_1} = \left( \frac{\partial C}{\partial z_1} \right) \left( \frac{\partial z_1}{\partial b_1} \right)$$

$\delta^L$       1

$$\boxed{\frac{\partial C}{\partial b_1} = \delta^L}$$

Vectorized form:

Example:



$$w_{lk} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}_{j,k} \quad (3,3)$$

$$\frac{\partial C}{\partial w_{lk}} \begin{bmatrix} \frac{\partial C}{\partial w_{11}} & \frac{\partial C}{\partial w_{12}} & \frac{\partial C}{\partial w_{13}} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$(n, l) \quad (l, m)$

$$\frac{\partial C}{\partial w_1} = \begin{bmatrix} a'_1 \delta_1^2 & a'_2 \delta_1^2 & a'_3 \delta_1^2 \\ a'_1 \delta_2^2 & a'_2 \delta_2^2 & a'_3 \delta_2^2 \\ a'_1 \delta_3^2 & a'_2 \delta_3^2 & a'_3 \delta_3^2 \end{bmatrix}$$

$$\begin{matrix} \tilde{\theta}_1 \\ \tilde{\theta}_2 \\ \tilde{\theta}_3 \\ \tilde{\theta}_4 \\ \tilde{\theta}_5 \end{matrix} \quad \begin{matrix} \tilde{\theta}_1 \\ \tilde{\theta}_2 \\ \tilde{\theta}_3 \\ \tilde{\theta}_4 \\ \tilde{\theta}_5 \end{matrix}$$

Calculate  
like:

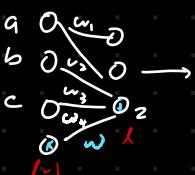
$$\delta^2 = \begin{bmatrix} \delta_1^2 \\ \delta_2^2 \\ \delta_3^2 \end{bmatrix} \quad a_1 = \begin{bmatrix} 1 \\ a'_1 \\ a'_2 \\ a'_3 \end{bmatrix}$$

$$\boxed{(\vec{\delta}^2)(\vec{a}_1)^T = \frac{\partial C}{\partial w_1}}$$

by matrix  
vector  
multiplication

Equation 4 - find the derivative of the cost wrt any weight

$$\frac{\partial C}{\partial w_{jk}} = a_{k-1}^T \delta_j^L \quad (\text{Scalar})$$



$$\text{have } \frac{\partial C}{\partial z_j^L} = \delta_j^L$$

$$z_j^L = (a_{k-1} + b_k a_k + c_k a_{k+1} + \dots + w_k \delta_j^L)$$

$$\delta_j^L = (a_{k-1} + b_k a_k + c_k a_{k+1} + \dots + w_k \delta_j^L)$$

$$\frac{\partial z_j^L}{\partial w} = k$$

What is  
k?  
activation  
of node  
l-1 from  
node k

Tying everything together

$$\frac{\partial C}{\partial b} \begin{cases} \vec{0} & \text{if } w^T x \leq 0 \\ \frac{1}{m} \sum_{i=1}^m (w^T x + b - y_i) e_i & \text{if } w^T x > 0 \end{cases}$$

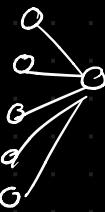
$$\frac{\partial C}{\partial w} \begin{cases} \vec{0} \\ \frac{1}{m} \sum_{i=1}^m (w^T x + b - y_i) x^T e_i \end{cases}$$

$$\frac{\partial C}{\partial b} = e^i = \frac{\partial C}{\partial b} = \delta^L$$

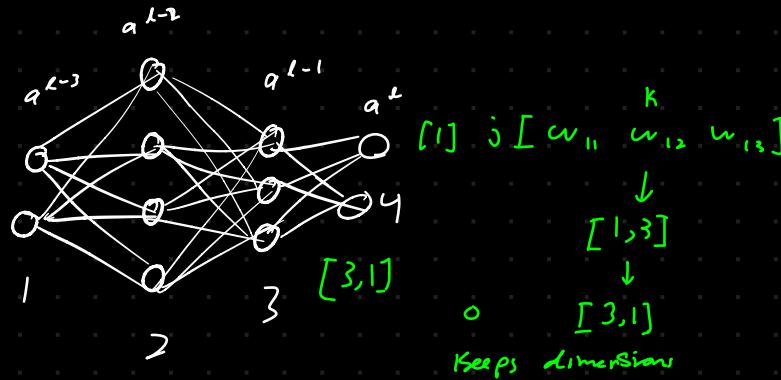
$$\frac{\partial C}{\partial w} = e_i x^T = \frac{\partial C}{\partial w^i} = \delta^L (a^{L-1})^T$$

\* These are equivalent

$e^i$   
 $e_i x^T$  are wrt only one node



① Feed forward and store  $z^l$  and  $a^l$  values at all  $l$  layers



② compute the cost

③ use eq 1 to calculate  $\delta^L$

④ use eq 2 to compute all  $\delta^L$  terms

⑤ use eq 3 to find bias derivatives  $\frac{\partial C}{\partial b^L}$

⑥ use eq 4 to find weight derivatives  $\frac{\partial C}{\partial w^i}$

$$\theta = \theta - \alpha \nabla_{w,b} C$$

forward pass  $\xrightarrow{\hspace{2cm}}$   
 back propagation

$\xleftarrow{\hspace{2cm}}$   
 back propagation

$$\nabla_{w,b} C = \begin{bmatrix} \quad \\ \quad \\ \quad \end{bmatrix}$$

unroll all matrices / flatten

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}$$

C U

$$① \delta^L = \nabla_a C \circ \sigma'(z^L) \quad (\text{last layer})$$

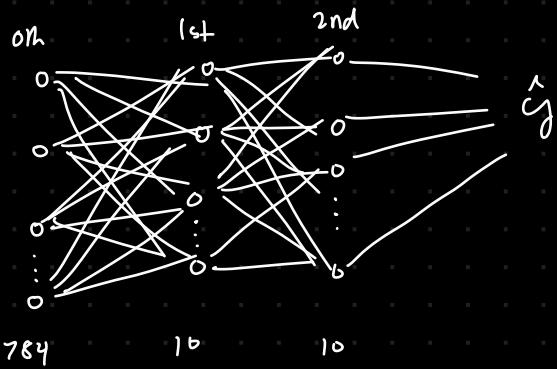
$$② \delta^L = ((w^{L+1})^T \delta^{L+1}) \circ \sigma'(z^L)$$

$$③ \frac{\partial C}{\partial b^L} = \delta^L$$

$$④ \frac{\partial C}{\partial w^i} = \delta^L (a^{L-1})^T$$

$$x = \begin{bmatrix} x^1 & x^2 & \dots & x^m \end{bmatrix} \Rightarrow 0, 1, 2, 3, \dots 9$$

10 classes



### ① Forward propagation

$$A^0 = X \quad (\text{first layer})$$

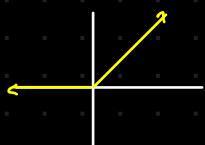
$$Z^1 = W^1 A^0 + b^1$$

$10 \times m$        $10 \times 784$        $784 \times m$        $10 \times m$

Apply activation function (otherwise layers are just linear combinations of previous layers)

use ReLU ( $Z^1$ )

adds complexity rather than a linear model



$$Z^2 = W^2 A^1 + b^2$$

$$A^2 = \text{softmax}(Z^2) \quad \frac{e^{z_{2i}}}{\sum_{i=1}^K e^{z_{2i}}}$$

our outputs  
are converted  
to a probability  
between 0 and 1

# 3Blue|Brown - Deep Learning Series

Neurons hold a number called an activation

Activations of one layer determine the

activations of the next layer

$$a^{(l)} = \frac{1}{1 + e^{-x}} \quad \text{column } a^l = a \begin{pmatrix} a_1^l \\ a_2^l \\ a_3^l \\ \vdots \\ a_n^l \end{pmatrix} = \begin{pmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & w_{k,n} \\ w_{k,0} & \dots & \ddots & \vdots \\ \vdots & \ddots & \ddots & w_{n,n} \end{pmatrix}^T + b^l = \begin{pmatrix} a^{l+1} \\ | \\ | \end{pmatrix}$$

Applying sigmoid elementwise to each of the entries

Each neuron is a function  
Network is a function of functions

$\frac{1}{m} \sum_{i=1}^m (y - y^i)^2$  cost function  
over all  $n$  training examples

$$\frac{\partial C}{\partial w} = 0 \text{ does not always work}$$

Global minimum -> crazy hard

The gradient of a function gives you direction of steepest ascent.

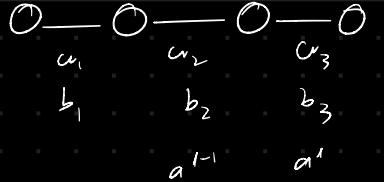
Compute  $\nabla C$ .

$$-\nabla C \vec{w} = \begin{bmatrix} \dots \\ \dots \\ \dots \end{bmatrix}$$

↑  
negative  
(descent)

magnitude tells = which direction gives you more "bang for your buck"

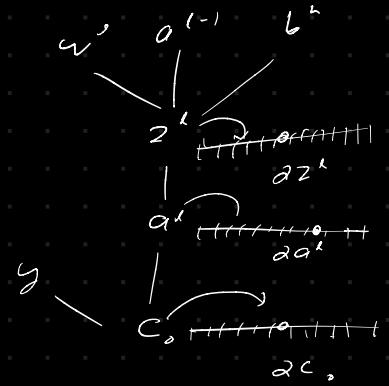
eg.



$$c = (wz)^2 + c_0$$

$$z^l = a^l \cdot a^{l-1} + b^0$$

$$a^l = \alpha(z^l)$$



$$\frac{\partial c}{\partial w^l} = \frac{\partial z^l}{\partial w^l} \frac{\partial a^l}{\partial z^l} \frac{\partial c_0}{\partial a^l} = (a^{l-1})(\alpha'(z^l))(2)(a^l - y)$$

chain rule

$$\frac{\partial c}{\partial a^l} = 2(a^l - y)$$

$$\frac{\partial a^l}{\partial z^l} = \alpha'(z^l)$$

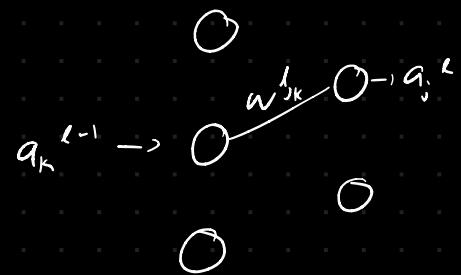
$$\frac{\partial z^l}{\partial w^l} = a^{l-1}$$

$$\frac{\partial c}{\partial b^1} = \frac{\partial z^l}{\partial b^1} \frac{\partial a^l}{\partial z^l} \frac{\partial c_0}{\partial a^l}$$

$$= 1 \cdot \alpha'(z^l) (a^l - y)$$

$\nabla C =$

$$\begin{bmatrix} \frac{\partial c}{\partial w^1} \\ \frac{\partial c}{\partial b^1} \\ \vdots \\ \frac{\partial c}{\partial w^l} \\ \frac{\partial c}{\partial b^l} \end{bmatrix}$$



$$\frac{\partial C}{\partial x_1} \rightarrow \boxed{\text{labour}} \quad \frac{\frac{\partial y_1}{\partial C}}{\frac{\partial C}{\partial y_2}} = \frac{x_2}{\frac{\partial C}{\partial x_2}} \rightarrow \boxed{\text{labour}} \quad \frac{\frac{\partial y_2}{\partial C}}{\frac{\partial C}{\partial y_3}} = \frac{x_3}{\frac{\partial C}{\partial x_3}}$$

$$\frac{\partial L}{\partial w} = \frac{\partial C}{\partial y} \frac{\partial y}{\partial w}$$

$$\frac{2C}{2x} = \frac{2E}{2y} \frac{2y}{2x}$$

class Lawyer :

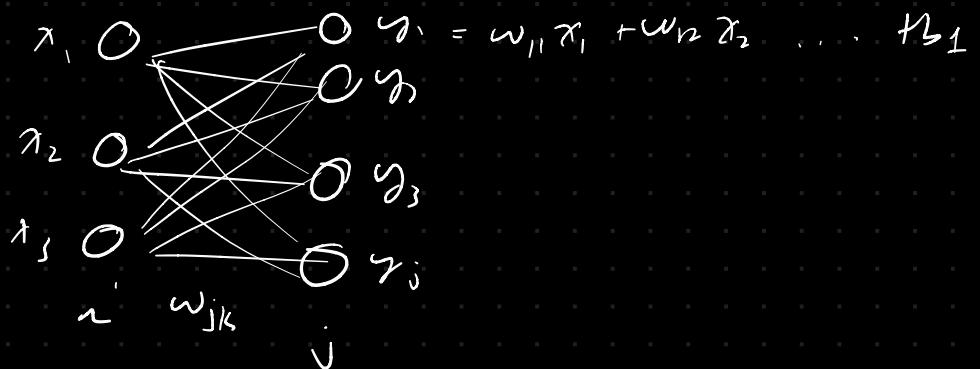
self.input

## self-output

forward (self, input )

backward (self, output-gradient, learning rate)

$$K \left( \begin{array}{cccc} w_{11} & w_{12} & \cdots & \\ w_{21} & \ddots & & \\ \vdots & & \ddots & \\ \end{array} \right) \left( \begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array} \right) + \left( \begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_j \end{array} \right)$$



$$j \begin{bmatrix} w_{11} & w_{12} & \dots \\ w_{21} & \ddots & \vdots \\ \vdots & \vdots & \ddots \\ w_{K1} & w_{K2} & \dots \end{bmatrix} k \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_J \end{bmatrix}$$

dot product

$$\frac{\partial c}{\partial y} = \begin{bmatrix} \frac{\partial c}{\partial y_1} \\ \vdots \\ \frac{\partial c}{\partial y_j} \end{bmatrix}$$

$$\frac{\partial c}{\partial w_{ik}} = \frac{\partial c}{\partial y_i} x_i$$

$$\frac{\partial c}{\partial w} = \begin{bmatrix} \frac{\partial c}{\partial w_{11}} & \frac{\partial c}{\partial w_{12}} \\ \frac{\partial c}{\partial w_{21}} & \ddots & \vdots \\ \vdots & \ddots & \frac{\partial c}{\partial w_{KK}} \end{bmatrix}$$

$$\frac{\partial c}{\partial w} = \begin{bmatrix} 1 \\ 1 \\ \vdots \end{bmatrix} (x_1 \ x_2 \ \dots \ x_K)$$

## Chapter 4 - what is a GPT?

Generative pre-trained transformer

```
graph TD; A[Generative] --> B[pre-trained]; B --> C[transformer]; C --> D[Neural net work]
```

Voice to text }  
Text to image } transformers

text → prediction that comes next

Give small section of data

add word to text (prediction)

repeat for new text

Input is broken up into tokens

words / character combinations

Associate token with a vector (unique)

words with similar meanings should land  
close to each other in the space

Attention matrix / blocks is used to update the "meaning" of a word

Feed to multilayer perceptron

Go back and forth between  
attention & multilayer perceptron

Want essential meaning of last vector to  
be used to create a probability distribution  
of tokens, predicting the next token

Blocks box of trainable parameters

Input must be an array of numbers / tensor

→ Input is transformed into array of real numbers

ChatGPT uses 8 categories of matrices for different purposes

GPT has a predefined vocabulary 50000

embedding matrix

$$\begin{pmatrix} \text{word \#1} & \text{word \#2} & \text{word \#3} & \dots \end{pmatrix} = \mathbf{W}_E$$

one column per word

Determines what word each vector turns into  
in the first step

Begin random

word  $\Rightarrow$  embedding

6 geometric points in a space

$\begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}$  12,728 coordinates

Directions have a semantic meaning

$$E(\text{King}) - E(\text{Queen}) \approx E(\text{man}) - E(\text{woman})$$

works exactly like math

Dot product of 2 vectors is a measure of how well they align

How words get embedded is learned through data

vocab size

$$50,257 \times 1288 = 617558216$$

embedding  
dimension

A word is informed by its surroundings

Empower word to consider the context

Initially the vectors are words without context  
we hope that the end, vectors adjust and represent the meaning of the word with respect to the context

fixed total words

2048

out put:

outputs are referred as logits.

Map matrix of tokens, and then apply softmax  
(for next word prediction)

Un embedding matrix  $W_u$

one row / word in vocab

Each row has same dim as embedding dimension

6 1 7 5 5 8 0 1 6

Soft max: probability distribution

All values between 0, and 1, and all  
values add to 1.

$$\frac{e^{x_i}}{\sum e^{x_n}}$$

picking maximizing input

GPT adds a constant to this

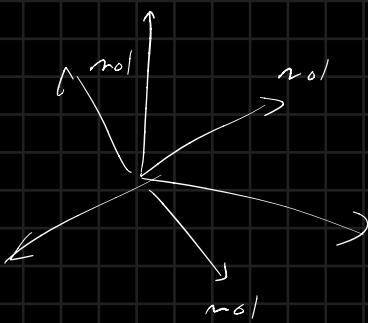
$$\frac{e^{x_i/T}}{\sum e^{x_n/T}}$$

as  $T \uparrow$ , more weight on  
low values

as  $T \downarrow$ , more weight on  
high values

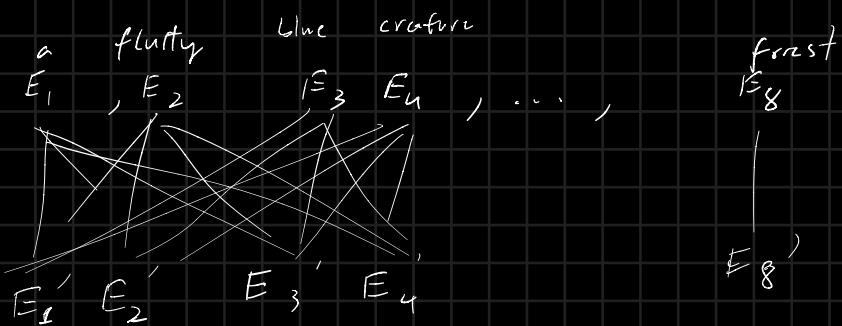
## Chapter 6 - Attention transformers

The initial tokens represent no context



Transforming = moves? the vector so that it takes in context

The last word encodes the entire block



Query (question) encoded as a vector

$$W_Q \left( \underbrace{\quad}_{\text{Question}} \right) \left| \begin{array}{c} \vec{E}_1 \\ \vdots \\ \vec{E}_n \end{array} \right. = \left( \begin{array}{c} \vec{Q}_1 \\ \vdots \\ \vec{Q}_n \end{array} \right)$$

By all  
embeddings  
 $\Downarrow$   
Do backpropagation

key matrix

$$a \rightarrow \vec{k} \in \mathbb{R}^{c_{\text{vk}} \times k}$$

$$\begin{bmatrix} & \\ & \end{bmatrix}$$

Answers the question from  
the query

Query & key dimensions are the same

How much do they match?

$E_1$

$$E_1 K^T Q_1$$

as value ↑, embeddings "attend to each other"

$$E_2 K^T \theta$$

$$E_3 K^T Q_1$$

compute soft max along columns of same grid dimension

how relevant is word on the left, to corresponding value at the top

Attention pattern

Applies  
column wise

Attention ( $Q, K, V$ )

$$\left( \begin{matrix} Q_1 \\ Q_2 \\ \vdots \end{matrix} \right) \quad \left( \begin{matrix} K_1 \\ K_2 \\ \vdots \end{matrix} \right)$$

$$= \text{softmax}\left(\frac{Q K^T}{\sqrt{d_k}}\right) V$$

Never want later words to influence later words

Want the bottom triangle of matrix to be zero

=> Solution

Set all features here to be  $-\infty$

$$\begin{array}{c|cc} -\infty & 1 & 2 \\ \hline -\infty & -\infty & 3 \\ -\infty & 1 & -\infty \end{array}$$

"masking"

Update embeddings

use value matrix to add to word without context

Apply to all tokens

$$\vec{v}_1, \vec{v}_2, \dots, \vec{v}_8$$

Multiply all dot products by corresponding row  $\vec{v}$

$$\vec{E} + \Delta \vec{E}_4 = \vec{E}'_4$$

Apply row wise, add column wise to original embedded word

This is called one head of attention.

$$\begin{matrix} Q & K & V \\ 12288 \times 128 & 12888 \times 128 & 12288 \times 12888 \end{matrix}$$

Value =  $K + Q$  (Smaller amount)

Split Value matrix into 2 smaller matrices

overall linear map

12288  $\xrightarrow{\quad}$  12288

value down matrix (dimensions)

value matrix (dimensions)

12288  $\xrightarrow{\quad}$  128

128  $\xrightarrow{\quad}$  12288 = output

value down matrix (dimensions)

value matrix (dimensions)

Cross attention (two types of data)

Self attention (one type of data)

Full attention block  $\Rightarrow$  Multiple self attention with

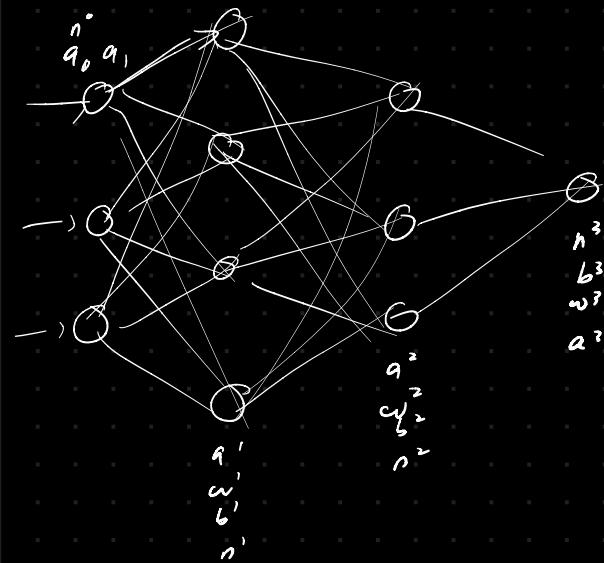
distinct key and queries

GPT uses 96 self attention heads

All heads give a proposed change. Add all changes to the original embedded word. This embedded vector is one output of the attention block.

Run in parallel





Denoting  $n^l$  # of neurons in the layer

$w^l$  weight matrix associated with the layers  $L$  and  $L-1$ , where  $w_{jk}^L$  represents the neuron  $j$  in  $L$ , and  $k$  in  $L-1$ .

$b^l$  vector of biases associated with layer  $L$ , of the size  $n^{L-1}$ .

$a^l$  activations of size  $n^l \times 1$

$z^l$  weighted output of neurons of size  $n^{L-1}$ .

$\sigma^l$  Activation function applied to the output of the neurons,  $a^l = \sigma^l(z^l)$

Let  $x$  be the input vector:  $a[0] = x$ .

$$z^L = (w^L)[a^{L-1}] + [b^L]$$

$$a^L = \sigma^L(z^L)$$

choose cross entropy loss for

example

$$C = -(y \log \hat{y} + (1-y) \log (1-\hat{y}))$$

denote  $y$  as the actual output

$\hat{y}$  as the predicted output

$$\begin{aligned} & z^0 = a^0 \\ & z^1 = w^1 a^0 + b^1 \\ & a^1 = \sigma^1(z^1) \\ & z^2 = w^2 a^1 + b^2 \\ & a^2 = \sigma^2(z^2) \\ & z^3 = w^3 a^2 + b^3 \\ & a^3 = \sigma^3(z^3) \end{aligned}$$

$$\begin{aligned} & z^0 = a^0 \\ & z^1 = w^1 a^0 + b^1 \\ & a^1 = \sigma^1(z^1) \\ & z^2 = w^2 a^1 + b^2 \\ & a^2 = \sigma^2(z^2) \\ & z^3 = w^3 a^2 + b^3 \\ & a^3 = \sigma^3(z^3) \end{aligned}$$

$$\frac{\partial C}{\partial w^3} = \frac{\partial C}{\partial a^3} \frac{\partial a^3}{\partial z_3} \frac{\partial z_3}{\partial w^3}$$

$\underbrace{\gamma}_{\text{if we shift } w, \text{ how does the cost change?}} \quad (a^2)^\top$

$$\frac{\partial C}{\partial b^3} = \frac{\partial C}{\partial a^3} \frac{\partial a^3}{\partial z_3} \frac{1}{\partial b^3}$$

of the very last layer before computing the cost.

$$a = a'(z)$$

$$z^3 = w^3 a^2 + b^3$$

of any layer:

$$\frac{\partial C}{\partial w^2} = \gamma^l (a^{l-1})^\top$$

$$\frac{\partial C}{\partial b^L} = \gamma^L$$

Now continue backpropagating our data pipeline:

$$\frac{\partial C}{\partial w_2} = \left( \frac{\partial C}{\partial a^3} \right) \left( \frac{\partial a^3}{\partial z^3} \right) \left( \frac{\partial z^3}{\partial a^2} \right) \left( \frac{\partial a^2}{\partial z^2} \right) \left( \frac{\partial z^2}{\partial w^2} \right)$$

describe as functions of each other.

$$\frac{\partial C}{\partial b_2} = \left( \frac{\partial C}{\partial a^3} \right) \left( \frac{\partial a^3}{\partial z^3} \right) \left( \frac{\partial z^3}{\partial a^2} \right) \left( \frac{\partial a^2}{\partial z^2} \right) \left( \frac{\partial z^2}{\partial b^2} \right) 1$$

$$\frac{\partial C}{\partial w_1} = \left( \frac{\partial C}{\partial a^3} \right) \left( \frac{\partial a^3}{\partial z^3} \right) \left( \frac{\partial z^3}{\partial a^2} \right) \left( \frac{\partial a^2}{\partial z^2} \right) \left( \frac{\partial z^2}{\partial a^1} \right) \left( \frac{\partial a^1}{\partial z^1} \right) \left( \frac{\partial z^1}{\partial w^1} \right)$$

$$\frac{\partial C}{\partial b_1} = \left( \frac{\partial C}{\partial a^3} \right) \left( \frac{\partial a^3}{\partial z^3} \right) \left( \frac{\partial z^3}{\partial a^2} \right) \left( \frac{\partial a^2}{\partial z^2} \right) \left( \frac{\partial z^2}{\partial a^1} \right) \left( \frac{\partial a^1}{\partial z^1} \right) \left( \frac{\partial z^1}{\partial b^1} \right) 1$$

We want to define our error recursively so that we don't have to keep calculating our cost derivatives from the start over and over again

$$\frac{\partial c}{\partial w^l} = \frac{\partial c}{\partial z^l} \frac{\partial z^l}{\partial w^l} \quad \text{for any layer } l$$

have intermediates  
 that we  
 need to  
 calculate

$$\frac{\partial c}{\partial b^l} = \frac{\partial c}{\partial z^l} \frac{\partial z^l}{\partial b^l} \quad \text{for any layer } l$$

For calculating the partial derivatives, we

should calculate

$$\begin{array}{cccc} \textcircled{1} & \textcircled{2} & \textcircled{3} & \textcircled{4} \\ \frac{\partial c}{\partial z^l} & \frac{\partial c}{\partial z^{l+1}} & \frac{\partial z}{\partial w^l} & \frac{\partial z}{\partial b^l} \end{array}$$

$\uparrow$   
 of  
 last  
 layer  
 $\downarrow$   
 layer  $l$

w.l. need  
to define  
this in  
terms of

$$\frac{\partial c}{\partial z^{l+1}}$$

$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L} \frac{\partial a^L}{\partial z^L} = \frac{\partial C}{\partial a^L} \otimes \sigma'(z^L)$$

$$\frac{\partial C}{\partial a^L} = \frac{2 - (\gamma \log a^L + (1-\gamma) \log (1-a^L))}{2a^L}$$

$$= -\left(\frac{\gamma}{a^L} - \frac{1-\gamma}{1-a^L}\right)$$

$$\sigma'(z) = a^L(1-a^L)$$

All together:

$$\frac{\partial C}{\partial z^L} = a^L - \gamma \left( \text{also called the error of a layer} \right)$$

Want  $\frac{\partial C}{\partial z^{L+1}}$ , to calculate  $\frac{\partial C}{\partial z^L}$

We can express recursively:

$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial z^{L+1}} \frac{\partial z^{L+1}}{\partial a^L} \frac{\partial a^L}{\partial z^L}$$

Proven by looking at our pipeline.

First, calculate  $\frac{\partial C}{\partial z^L}$ , to calculate  $\frac{\partial C}{\partial z^L}$ ,  $L=2-1, L-2 \dots$

From

$$z^{L+1} = w^{L+1} a^L + b^{L+1}$$

$$\frac{\partial z^{L+1}}{\partial a^L} = w^{L+1}$$

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L)$$

All together:  $\frac{\partial C}{\partial z^L} = \left( (w^{L+1})^T \left( \frac{\partial C}{\partial z^{L+1}} \right) \otimes \sigma'(z^L) \right)$

$$z^l = w^l \cdot a^{l-1} + b^l$$

$$\frac{\partial z^l}{\partial w^l} = a^{l-1} \quad \frac{\partial L}{\partial w^l} = \frac{\partial L}{\partial z^l} \cdot a^{l-1)^T}$$

$$\frac{\partial z^l}{\partial b^l} = 1 \quad \frac{\partial L}{\partial b^l} = \frac{\partial L}{\partial z^l}$$

for a batch of  $m$  training examples, we need to average out  
our derivatives over  $m$ .

compute gradient of cost function using the chain rule from  
multivariable calculus.

# Michael Nielsen: Neural Networks & Deep Learning

## Chapter 1:

$V_1$ : primary visual cortex:

Neural networks: infer rules for recognizing hand written digits.

### 1.1

perceptron: artificial neuron

used in production  $\rightarrow$  sigmoid neuron

perceptron: takes in several binary inputs, and returns one binary output



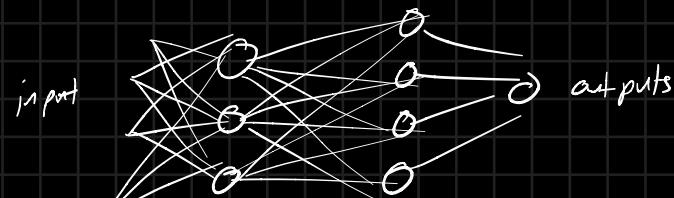
Binary as in input/output are 0/1.

i.e.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Larger weights mean that input has more impact.

Wish up evidence to make decisions.



A perceptron produces an output and makes a decision. This output is inputted to several other perceptrons.

Rewrite:

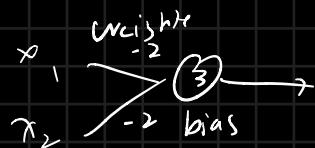
$$\sum_j w_j x_j = \vec{w} \cdot \vec{x}$$

= how easy is it to get the perception to "fire"

Move threshold to other side  $\rightarrow$  called the bias

$b \Rightarrow \uparrow$  fire chance  
small  $\Rightarrow \downarrow$  fire chance

$$\text{output} = \begin{cases} 0 & \text{if } \vec{w} \cdot \vec{x} + b \leq 0 \\ 1 & \text{if } \vec{w} \cdot \vec{x} + b > 0 \end{cases}$$



Logical gates can be implemented by neural networks.

? ? ? NAND (pg 5-6)

Think of input perceptions as special units defined to output desired

values

output, but no inputs (pg. 6)

$$\theta_i \rightarrow$$

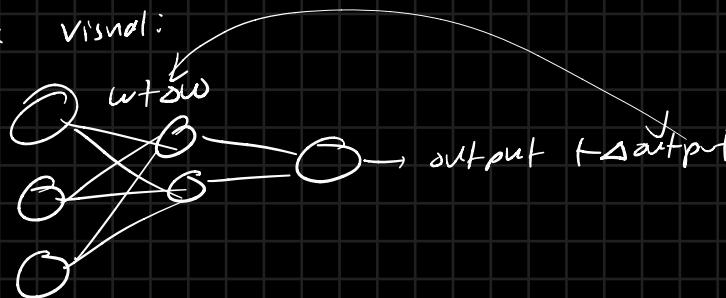
Improvement  $\rightarrow$  logic gates that learn to solve problems.

## 1.2 - Sigmoid neurons

Suppose we make a small change in a weight/bias in the network.

↪ observe how this change causes a corresponding change in the output from the network.

As a visual:

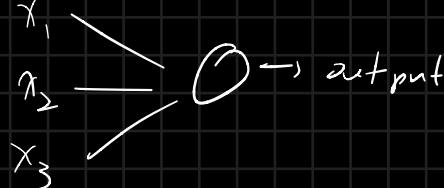


Using this, we try to adjust weights & biases to set a model to behave in the way we want.  
repeat again and again.

↪ hard to do with binary output

Introduce: the sigmoid neuron

small change to input causes small change to output.



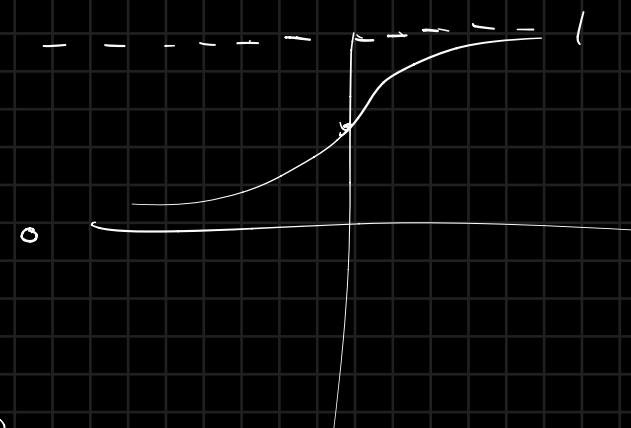
Any value  $0 \leq x \leq 1$ ,  $x \in \mathbb{R}$

$$\sigma(w \cdot x + b)$$

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (\text{sigmoid function})$$

Very positive  $\rightarrow$  approaches 1

Very negative  $\rightarrow$  approaches 0



From Calculus,

$$z = \sum w_i x_i + b$$

$$\Delta z = \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b \quad (\text{linear approximation!})$$

Rate of change  $\times$  Change in ind variable of  
partial derivative

Any activation function applies  $f(wx + b)$

Scaling weights & biases have no effect, since the decision made by a perceptron is influenced by its other weights. This means that the output should be scaled, meaning the behaviour does not change.

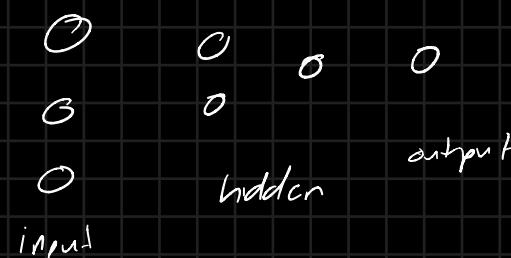
As  $C \rightarrow \infty$ , for sigmoid neurons, both should output a very positive number, meaning that both will make the same decision to output a 1.

When  $wx + b = 0$ , the other perceptrons' "output / decision" may get scaled, and thus, lead to a slight nudge in the output, leading to a different decision before scaling  $C \rightarrow \infty$ .

# 1.3 - Architecture of a neural network

Input layer } input neurons

out put layer } output neurons  
hidden layer → hidden neurons



Feed forward NNs

Digit classification tasks:



$$784 = 28 \times 28 \text{ pixels}$$

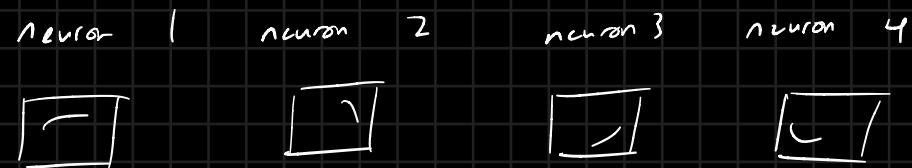
gray scale values, between 0 and 1.

Why do neurons  $\Rightarrow$  Inefficient?

Test out to find out.  $n^{neurons}$  is better

From first principles:

e.g. first output neuron  $\rightarrow$  decides if digit is a zero.  
 $\rightarrow$  weighs up all evidence from hidden layer of neurons.



If we had 4 outputs for example, the first output neuron must be trying to figure out the most significant section of the digit.

Heuristic.

Binary representation from 0-9

4 bits max.  
weights corresponding to correct bit should ↑  
weights corresponding to incorrect bit should ↓

# 1.5 - Gradient descent.

$\vec{x}$  training input.  $784 \times 1$

$\vec{y}$  corresponding desired output  $10 \times 1$

(Cost function: change weights and biases so output from network approximates  $\vec{y}_{\text{true}}$  for all  $n$  training examples)

$$C(w, b) = \frac{1}{2n} \sum_{x=1}^n (y(x) - a)^2$$

$\vec{a}$  vector of outputs

$n$  training examples

$w, b$  collection of all weights and biases.

aka MSE

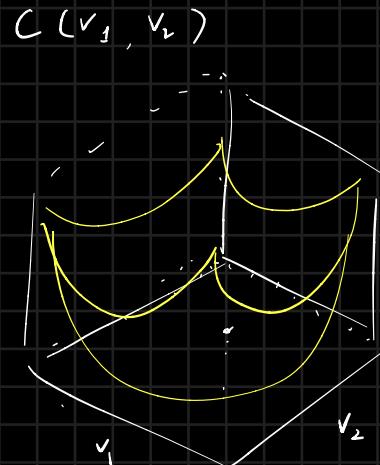
Small cost  $\rightarrow$  high accuracy

high cost  $\rightarrow$  low accuracy

The # of images is not a smooth function of the weights and biases.

Focus: Isolate the idea of minimizing a function of many variables

e.g.  $C(v)$ , real-valued function



Find where  $C$  achieves a global minimum.

Cannot use calculus to find the extremum  $\rightarrow$  many calculations!

first and second derivatives

??

Simulate the motion of a ball, rolling down a valley.

small  $v_1$ ,  $v_2$ , and  $v_3$  movement:

$$\Delta C = \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2$$

$$\Delta v = (\Delta v_1, \Delta v_2)^T$$

choose  $\Delta v_1, \Delta v_2$  to make  $\Delta C$  negative.

define gradient of the vector  $C$

means gradient

$$\nabla C = \left( \frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T$$

dot product

$$\Delta C \approx \nabla C \cdot \Delta v$$

Suppose we choose  $\Delta v = -n \nabla C$

$n$  is a small positive parameter called the learning rate.

$$n > 0 \quad \nabla C > 0$$

$$\Delta C = \nabla C \cdot (-n \nabla C) = -n \|\nabla C\|^2$$

This guarantees  $\Delta C$  to be a negative number

$$\Delta C \leq 0$$

positive position 2

$$v \rightarrow v' = v - n \nabla C$$

choose small  $n$  so that  $-n \|\nabla C\|^2$  approximates well.

works for  $m$  variables,  $v_1, v_2, \dots, v_m$

$$\nabla C = \left( \frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m} \right)^T$$

Gradient descent is the most optimal strategy.

Suppose we want to make a move  $\Delta v$  as to decrease  $C$  as much as possible.

i.e. we want to minimize  $\Delta C = \nabla C \cdot \Delta v$

Define  $\theta$ , s.t. we can constrain  $\|\Delta v\| = \theta$ , for  $\theta \geq 0$ .

The choice of  $\Delta v$  that minimizes  $\Delta C$ , is  $\Delta v = -n \nabla C$  where  $n = \frac{\theta}{\|\nabla C\|}$

Lanchy-Schwarz inequality

upper bound on the inner products between 2 vectors in an inner product space, in terms of the product of the vector magnitudes.

for all  $u, v$  of an inner product space.

$$|u \cdot v|^2 \leq (u \cdot u) \cdot (v \cdot v)$$

$$|\nabla C \cdot \Delta v| \leq \underbrace{\|\nabla C\| \|\Delta v\|}_{\theta}$$

To minimize, we need to maximize the cosine between  $\nabla C$  and  $\Delta v$ .

$\Delta v$  points in the opposite direction of  $\nabla C$ .

Replace the vector  $\vec{v}$  with  $\vec{w}$  and  $\vec{b}$

$$\nabla C \frac{\partial C}{\partial w_k} \dots \nabla C = \dots \frac{\partial C}{\partial b_1}$$

$$\vec{w}_k = \vec{w}'_k = w_k - n \frac{\partial C}{\partial w_k}$$

$$\vec{b}_1 = \vec{b}'_1 = b_1 - n \frac{\partial C}{\partial b_1}$$

Because  $C$  is of the form  $C = \frac{1}{n} \sum_x C_x$ ,  $C_x = \frac{1}{n} \log(g(x) - a)$ ,

we need to compute the cost for each training input separately, then average them.

Stochastic gradient descent  $\Rightarrow$  estimate  $\nabla C$ , by computing  $\nabla C_x$  for a small sample of randomly chosen outputs.

Randomly pick small #  $m$  training inputs, called  $x_1, x_2, \dots, x_m$ .

We hope that  $\frac{\sum_{j=1}^m C_x}{m} = \frac{\sum_x \nabla C_x}{n} = \nabla C$

$$\therefore \nabla C = \frac{1}{n} \sum_{j=1}^m \nabla C_{x_j}$$

for SGD:

$$w_k - \frac{n}{m} \sum_j \frac{\partial C_{x_j}}{\partial w_k}$$

$$b_1 - \frac{n}{m} \sum_j \frac{\partial C_{x_j}}{\partial b_1}$$

Repeating SGD for multiple batches

→ Called one epoch.

Removing  $\frac{1}{m}$  and  $\frac{1}{n}$  are equivalent to scaling  
the learning factor.

mini-batch size 1 → More training examples to work with,  
but, will be significantly slower.

60,000 images  $\Rightarrow$  50,000 images (training)  
 $\Rightarrow$  10,000 images (validation)

```
class Networks (object):  
    def __init__(self, sizes):
```

```
        self.num_layers = len(sizes);  
        self.sizes = sizes
```

```
        self.biases = [np.random.rand(y, 1) for y in sizes[1:]]  
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
```

randn → mean 0, std dev 1, Normal distribution.

Denoting matrix  $W$ , with  $w_{jk}$  weights

from the  $k$ th neuron into the  $j$ th neuron. 10 neurons

$$a' = \sigma(Wa + b)$$

↓  
Element-wise  
Second layer  
of neurons

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = 10 \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \sigma \left( \begin{matrix} & 784 \\ w^T & \xrightarrow{j} \\ & \downarrow k \end{matrix} \right) \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} 784 + \begin{pmatrix} b \\ \vdots \\ b \end{pmatrix} 10$$

```
def Sigmoid(z):  
    return 1.0 / (1.0 + np.exp(-z))
```

This is applied element wise.

```
def feed forward(self, a):  
    for b, w in zip(self.biases, self.weights):  
        a = Sigmoid(np.dot(w, a) + b);  
        # must pick the correct dimensions  
    return a
```

SGD function => pg 26.

Full Code on pg 28-30.

Once trained, our model can easily run

Iterating over hyper-parameters → fine tuning

Problems:

Initializing  $w$  and  $b$  wrong for network to learn

Don't have enough training data to get meaningful learning

Impossible tasks.

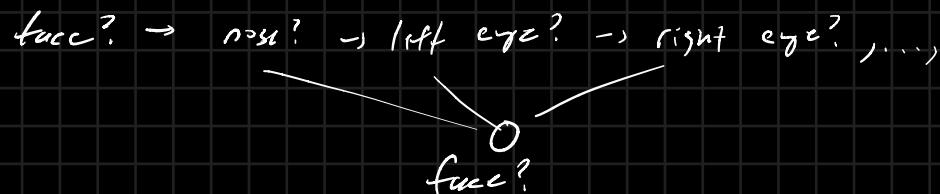
Learning rate too high/low

## Support Vector machines: SVMs

skikit-learn → LIBSVM

Simple algorithm + good\_training-data ≥ sophisticated algorithm

If we can solve sub-problems, we solve large problems with neural networks.



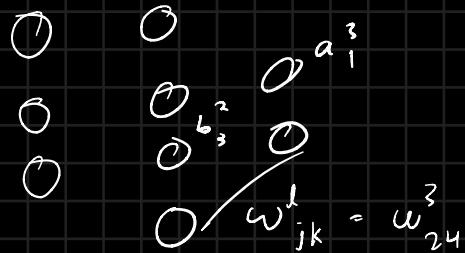
even further → eye? → eye brow? → eyelashes → iris

⇒ general term: Deep neural networks

Hidden layers allow you to build up a hierarchy of concepts

Back propagation: how quickly the cost changes wrt weights and biases.

$w_{jk}^l$   $k^{th}$  neuron from  $l-1$  layer, into  $j^{th}$  neuron of  $l^{th}$  layer.



$b_j^l$ ,  $a_i^l$

$$a_i^l = \sigma \left( \sum_k w_{ik}^l a_k^{l-1} + b_i^l \right)$$

$\downarrow \quad \downarrow \quad \downarrow$   
vectors      vectors

As a matrix,  $w^l$  is the weights matrix at weights

that connect to the  $l^{th}$  layer of neurons.

to the  $j^{th}$  row — , from the  $k^{th}$  column  $\downarrow$

To calculate an element wise function,

$$\sigma \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \sigma(z_3) \end{pmatrix}$$

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

Define  $z^l$  as the weighted input to the neurons of layer  $l$

$$a^l = \sigma(z^l)$$

Goal: compute  $\frac{\partial C}{\partial w}$  or  $\frac{\partial C}{\partial b}$  wrt any  $w$  /  $b$

$$C = \frac{1}{n} \sum_x (y(x) - a^L(x))^2, x \text{ is a training example}$$

↓  
desired output      vector activations  
when  $x$  is input.

2 statements to clarity.

$$\textcircled{1} C = \frac{1}{n} \sum_x C_x \text{ average over cost function}$$

compute  $\frac{\partial C_x}{\partial w}$   $\frac{\partial C_x}{\partial b}$ , then get  $\frac{\partial C}{\partial w}$   $\frac{\partial C}{\partial b}$  by averaging all training examples

(2) cost, as a function of the output activation  $a^L$

for a single training ex  $x$ :

$$C(a^L) = \frac{1}{2} \sum_j (y_j - a_j^L)^2$$

Not a function of  $x$  or  $y$ !

$\Rightarrow x$  and thus  $y$ , are fixed before, and we want to model based on the network's outputs, as a determining variable for the "accuracy" of a model.

$\Rightarrow$  Not something we can change and not something a model learns.

## 2.3 - Hadamard product

Element wise product of 2 vectors

$$(\vec{s} \circ \vec{t})_j = s_j t_j$$

## 2.4 - Four equations of backpropagation

Introduce  $\delta_j^l$  called the error of  $j$ th neuron into the  $l$ th layer

sits at the  $j$ th neuron at layer  $l$ . It adds a change  $\Delta z_j^l$  to the neurons weighted input

rather than outputting  $a(z_j^l)$ , the neuron outputs  $a(z_j^l + \Delta z_j^l)$ , which propagates through the network

causing overall cost to change by  $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$

Choose  $\Delta z_j^l$  to have opposite sign of  $\frac{\partial C}{\partial z_j^l}$

$\frac{\partial C}{\partial z_j^l}$  is a measure of the error of a neuron

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

$\delta^l$ , error of layer  $l$

$$\frac{\partial C}{\partial a_j^l} = \frac{\partial C}{\partial z_j^l} \text{ depending on activation function chosen}$$

$$\textcircled{1} \quad \delta_j^l = \frac{\partial C}{\partial a_j^l} \sigma'(z_j^l)$$

★ output layer error only  
 deactivation function  
 w.r.t weighted sum  
 cost w.r.t output activation

↙ for  $j$  neurons in a layer  
 => correct notation would be  $\delta_j$ .

$$\frac{\partial C}{\partial a_j^l} = (a_j^l - y_j)$$

⇒ Matrix-based representation

$$\delta^l = \nabla_a C \odot \sigma'(z^l)$$

Gradient components are partial derivatives  $\frac{\partial C}{\partial a_j^l}$

$$\frac{\partial C}{\partial a^l} = \nabla_a C = (a^l - y)$$

$$\delta^l = (a^l - y) \odot \sigma'(z^l)$$

② Error  $\delta^l$  in terms of error at next layer

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Moving error back through the network of weights, then, through the activation function in layer  $l$ , to give us  $\delta^l$

Now, we can compute  $\delta^l$  for any layer in the network.

③ Rate of change of cost w.r.t any bias in the network

$$\frac{\partial C}{\partial b_j} = \delta_j^l$$

$$\frac{\partial C}{\partial b} = \delta$$

④ Rate of change of cost w.r.t any weight in the network

$$\frac{\partial C}{\partial w_{jk}} = a_k^{l-1} \delta_j^l$$

$$\frac{\partial C}{\partial w} = a_{in} \delta_{out}$$

$$a_{in} \times \delta_{out}$$

⑤              

weights outputs from low activation neurons learn slowly.

Depending on activation func, neurons learn differently, and they

can be predicted based on activation function graph behaviours.

## 2.6 - Back propagation algorithm

1. Input  $x$ , compute  $a^1, l=1$
2. Feed forward for  $l=2, 3, \dots, L$ , compute  $z^l = w^l a^{l-1} + b^l$  and  $a^l = \sigma(z^l)$
3. Compute  $\delta^L = \nabla_a C \odot a'(z^L)$  at the end
4. For each previous layer: Compute  $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot a'(z^l)$
5. The gradient of a cost function is given by  $\frac{\partial C}{\partial w_{jk}^l} = a^{l-1} \delta_j^l$ ,  $\frac{\partial C}{\partial b_j^l} = \delta_j^l$

For SGD,

$$w^l - \frac{1}{m} \sum_x \delta^{x,l} \odot a^{(x,l-1)}^T$$

$$b^l - \frac{1}{m} \sum_x \delta^{x,l}$$

Rather than propagating a matrix of examples, we can do it with a matrix.

$$\Delta C \approx \frac{\partial C}{\partial a_1^l} \frac{\partial a_1^l}{\partial a_n^{l-1}} \frac{\partial a_n^{l-1}}{\partial a_r^{l-2}}, \dots, \frac{\partial a_1^l}{\partial a_2^l} \frac{\partial a_2^l}{\partial a_j^l} \Delta w_{jk}^l$$

$$\Delta C = \sum_{mnpq} \frac{\partial C}{\partial a_1^l} \frac{\partial a_1^l}{\partial a_n^{l-1}} \frac{\partial a_n^{l-1}}{\partial a_r^{l-2}}, \dots, \frac{\partial a_1^l}{\partial a_2^l} \frac{\partial a_2^l}{\partial a_j^l} \Delta w_{jk}^l$$

for all intermediate neurons affecting the cost, when we

change one weight  $w_{jk}$

If we think of all intermediate edges connecting to the cost, from a small change in a weight, the partial derivative is just a rate and the path to the cost function is the sum of all rate factors  $\times$  paths from the initial change in a weight.