NNs

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 propogation
BP as Matrix Calculus

Basic Notation

Define m = size of training set

Define n = # of input variables $x_1, x_2, \ldots$ $\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$

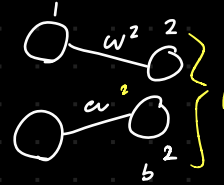
Define ^nL = amount of layers in NN

Define l = specific layer

w^l = weight (l) l = layer weights going into

b^l - bias for layer l

eg.



Define x_i = single input variable

Big picture

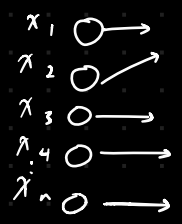
NNs are functions (linear functions of weights & biases)

Big calculus problem (linear approximations)

→ $\frac{\partial cost}{\partial w}$ (want this)

Find how much each weight & bias contributes to the cost

with respect to every weight and bias



$$f(x,y) = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 2x + y^3 \\ e^y - 13x \end{bmatrix}$$

→ Breaks up into 2 scalar functions

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

① $f_1 = 2x + y^3 \to \frac{\partial f_1}{\partial x} = 2 \quad \frac{\partial f_1}{\partial y} = 3y^2$

② $f_2 = e^y - 13x \to \frac{\partial f_2}{\partial x} = -13 \quad \frac{\partial f_2}{\partial y} = e^y$

small functions composing one large function

create Jacobian

denoted also as: $\nabla f_1, \nabla f_2$

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

"Transposed" defined rigorously from original function

Matrix Calculation Review

Gradients, Jacobians, Jacobian Chain Rule

Vector function

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 $f(x,y)$"

Gradients:

vector to scalar

$\mathbb{R}^n \to \mathbb{R}^1$

$x = 2, \; y = 0$

$\begin{bmatrix} x \\ y \end{bmatrix} \xrightarrow{f(x,y)} (5)$ → 5 for Scalar

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

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

Jacobian:

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

partial derivatives,

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

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

Vector to another Vector

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

From last page

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

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

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

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

$f_1 = 2x - y^3$

$f_2 = e^x - 13y$

Scalar Chain Rule

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

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

Let $g = x^2$

$f = \sin(g)$

technical breakdown

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} = 2x\cos g \quad \text{(substitute back)} \quad = 2x\cos(x^2)$$

Now, extend to: $f = g(h(c(a(x))))$ as a general concept: use product of partial derivatives

also from 3Blue1Brown chain rule video

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

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

What if we cannot use chain rule?

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

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

leave as is if no intermediate

put a 1 when you are not using chain rule also / no intermediate function

---

Jacobian chain Rule

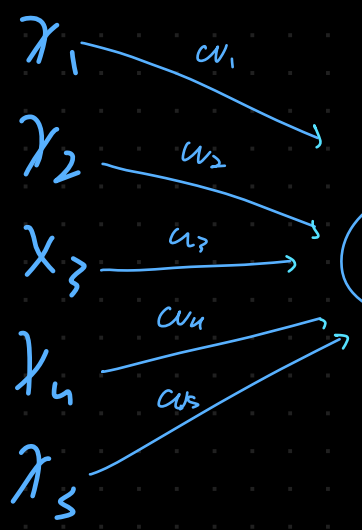
Define the following:

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

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

Inside functions — outside functions

Neurons



① $w_1 x_1 + w_2 x_2 \dots$

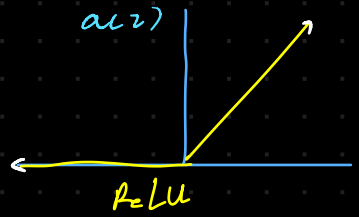
Inputs to one single neuron

② $a\left(\sum_{i=1}^{n} x_i w_i + b\right)$

Scalar

activation function

$a(z)$



★ A neural network with no non-linear activation function is linear regression

③ We define

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^T \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix} \rightarrow a(\underbrace{\vec{x}^T \vec{w} + b}_{z})$$

and then say

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

$a = a(z)$

Notations

denote as a

Notation for layers

Denote w^{l}_{jk} (think of it as the lines between neurons)

l = layer weight is going into

j = # of the neuron of the layer l

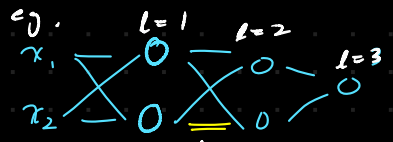
k = # of the neuron in layer $l-1$

b^{l}_{j} or b^{l}

↑ Can also show up

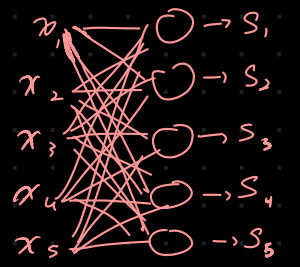
↑ often each layer has one bias

e.g.



w^{2}_{22} (going into layer 2, into neuron #2, from neuron #2)

For one layer: Define scalars

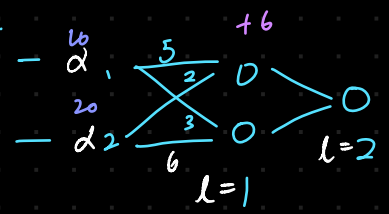


$\rightarrow \vec{a}^{\,l}$ (activation layer) — l for which layer

denote

(a for activation) input goes into

eg.



w^l_{jk} construct the weights matrix

$$w^1 = \begin{bmatrix} w^1_{(1,1)} & w^1_{(1,2)} \\ w^1_{(2,1)} & w^1_{(2,2)} \end{bmatrix} = \begin{bmatrix} 5 & 2 \\ 3 & 6 \end{bmatrix} \quad \begin{matrix} j \\ \\ \end{matrix} \quad \overset{k}{\text{(top)}}$$

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

activation of the previous layer

Calculates the "weighted" sum of the input

$$= \begin{bmatrix} 3 & 2 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \end{bmatrix} + \begin{bmatrix} b_1 \\ 1 \end{bmatrix}$$

b_1

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

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

From single neuron to full layer:

Single node

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

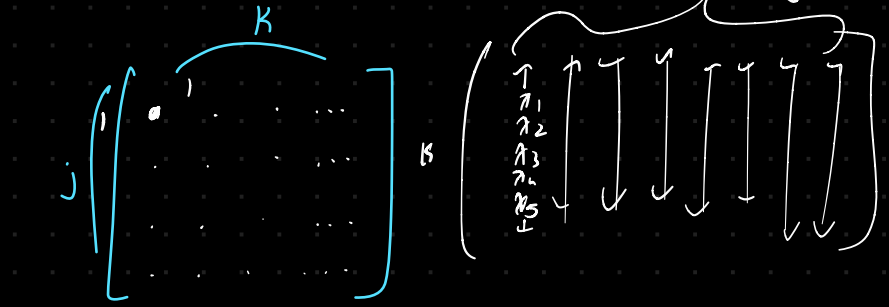
Entire layer

w^l

w^l_{jk} is equivalent to:

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

eg. Coming from the 1st neuron in layer $l-1$, and going into the 1st neuron in layer l



Comes from the kth neuron and goes into the jth neuron

Activation Function Calculation

General: $a_n = \sigma(w^{l+1} a^n + b^l)$

$a_0 \Rightarrow \sigma(w^1 a^0 + b^1)$

Vector → vector — output is a vector

from a_0,

$a_1 = \sigma(w^2 a^1 + b^2)$

$a_2 \rightarrow \dots$

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

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

ReLU: $z=0$, $z=z$

Sigmoid: 0, 1

feed forward through an NN

input through the network

forward

backward

$\sigma(z) = \begin{bmatrix} 96 \\ 156 \end{bmatrix}$ stays the same

Jacobians & NNs

Minimize the cost of the output (error)

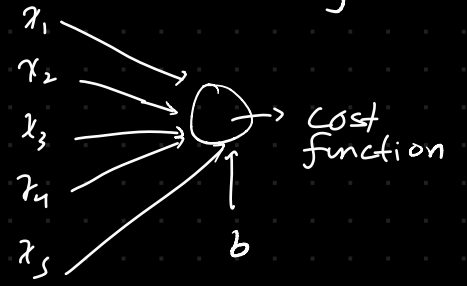
Mean squared error

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

↓ ↓ ↓ answer we got, real answer

$\frac{1}{2}$ does not matter (for the power rule)

m is the # of training examples



Jacobians of Operations

Binary element-wise operations

vector to vector

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

① $\begin{bmatrix}\vec{v}\end{bmatrix} \begin{matrix}\otimes\\ \otimes\\ \otimes\end{matrix} \begin{pmatrix}\vec{w}\end{pmatrix} = \begin{bmatrix}\vdots\end{bmatrix}$

element wise operators (can be any thing, keep it general)

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

③ $J = \frac{\partial \vec{F}}{\partial \vec{v}} = \begin{matrix}F_1\\ F_2\\ \vdots\\ F_n\end{matrix}$

$$\begin{bmatrix} v_1 \otimes w_1 \\ v_2 \otimes w_2 \\ v_3 \otimes w_3 \\ \vdots \\ v_n \otimes v_n \end{bmatrix} = \begin{pmatrix} b \end{pmatrix}$$

② $F(\vec{v},\vec{w})$ (general)

$= (f(\vec{v})) \circ (g(\vec{w})) =$ "operations on the functions before"

$$\begin{bmatrix} f_1(\vec{v}) \circ g_1(\vec{w}) \\ f_2(\vec{v}) \circ g_2(\vec{w}) \\ \vdots \\ f_n(\vec{v}) \circ g_n(\vec{w}) \end{bmatrix} \begin{matrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{matrix}$$

first element of v_0 second element of w

Can get 2 Jacobians one for each vector

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

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

General representation

incase f_1 and g_1 don't change the vector

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

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

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

↑ scalar

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

↳ $\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix}$ → diag (a, b, c)

Derivatives of Neuron Operators (Hadamond product)

→ Hadamard Product

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

$$\frac{\partial F}{\partial \vec{v}} = \begin{matrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{matrix} \begin{bmatrix} \frac{\partial F_1}{\partial v_1} & \frac{\partial F_1}{\partial v_2} & \cdots & \frac{\partial F_1}{\partial v_n} \\ \frac{\partial F_2}{\partial v_1} & \frac{\partial F_2}{\partial v_2} & \cdots & \frac{\partial F_2}{\partial v_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial v_1} & \frac{\partial F_n}{\partial v_2} & \cdots & \frac{\partial F_n}{\partial v_n} \end{bmatrix} = \begin{bmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & & \vdots \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & w_n \end{bmatrix}$$

$\frac{\partial F}{\partial \vec{v}}$ ↓ with respect to $\vec{v}$

$\frac{d}{dw}(xw) = x$ (easier easy to understand)

Derivative of a Scalar expansion

$$2\begin{bmatrix} \\ \vec{v} \\ \\ 1 \end{bmatrix} = \begin{bmatrix} 2v_1 \\ 2v_2 \\ 2v_3 \\ \vdots \\ v_n \end{bmatrix}$$

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

defining for $g(\vec{v}) = \vec{1} \cdot x$ (expands into a vector)

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

$$\begin{bmatrix} 2 \\ 2 \\ \vdots \\ 2 \end{bmatrix} \circ \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} f_1(\vec{v}) \circ g_1(\vec{x}) \\ f_2(\vec{v}) \circ g_2(\vec{x}) \\ \vdots \\ f_n(\vec{v}) \circ g_n(\vec{x}) \end{bmatrix} \begin{matrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{matrix}$$

don't forget, this is the Scalar vector.

$$\frac{\partial F}{\partial \vec{v}} = \begin{bmatrix} \frac{\partial f_1}{\partial v_1} & \frac{\partial f_1}{\partial v_2} & \cdots & \frac{\partial f_1}{\partial v_n} \\ \frac{\partial f_2}{\partial v_1} & \frac{\partial f_2}{\partial v_2} & & \vdots \\ \vdots & & \ddots & \\ \frac{\partial f_n}{\partial v_1} & \frac{\partial f_n}{\partial v_2} & \cdots & \frac{\partial f_n}{\partial v_n} \end{bmatrix} = \begin{bmatrix} x & & & 0 \\ & x & & \\ & & \ddots & \\ 0 & & & x \end{bmatrix}$$

When we get the Jacobian, we set a gradient (x is a Scalar)

Gradient with respect to x

$$\nabla_x F = \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \\ \vdots \\ \frac{\partial f_n}{\partial x} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Derivative of a Sum

g is element wise

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

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

$$J = \begin{bmatrix} \frac{\partial}{\partial v_1} \sum_{i=1}^{n} g_i(v) & \frac{\partial}{\partial v_2} \sum_{i=1}^{n} g_i(v) & \cdots & \frac{\partial}{\partial v_n} \sum g(v) \end{bmatrix}$$

★ The derivative of a sum is the sum of the derivatives

$$J = \begin{bmatrix} \sum_{i=1}^{n} \frac{\partial}{\partial v_1} g_i(v) & \sum_{i=1}^{n} \frac{\partial}{\partial v_2} g_i(v) & \cdots & \sum_{i=1}^{n} \frac{\partial}{\partial v_n} g_i(v) \end{bmatrix}$$

Easy to generalize

If $g(\vec{v}) = 2\vec{v}$ → $[2\ 2\ 2\ \dots\ 2]$

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

$$J = \begin{bmatrix} \sum_{i=1}^{n} \frac{\partial}{\partial v_1}(v_i) & \sum_{i=1}^{n} \frac{\partial}{\partial v_2}(v_i) & \cdots & \sum_{i=1}^{n} \frac{\partial}{\partial v_n}(v_i) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

1 + 0 + 0 + ... 0 + 1 + 0 + ... 0 + 0 + ... + 1

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

Derivative of a neuron

activation
$a = \sigma(\vec{w}^T \vec{x} + \vec{b})$ dot product = Hadamard product
(underbrace: z)

we want:

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

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

Define an activation like:

$$a = \sigma(\mathrm{Sum}(w \otimes x) + b)$$

H → Hadamard product
$S(H)$

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

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

② $z = S(H) + b$
$\frac{\partial z}{\partial H} = 1$
for $S(H)$
$\frac{\partial s}{\partial H} = [1\ 1\ 1\ 1\ 1\ 1]$

③ $z = s + b$
$\frac{\partial z}{\partial s} = 1$ (scalar)

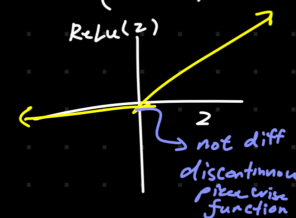
④ Derivative of ReLU wrt the activation

$$\frac{\partial a}{\partial w} = \frac{\partial a}{\partial z}[\vec{x}^T]$$

↓ Simplifies after all 3 substitutions

Need to find this

$\max(0, z)$ of 2 numbers ($\Sigma(\vec{w}\vec{x}) + b$)



not diff
discontinuous piecewise function
, but in practice, does not matter

To solve, we split up into piecewise functions

taking the derivative

$$\frac{\partial a}{\partial w} = \max \begin{cases} z \le 0,\ 0 \to \vec{0} \times \frac{\partial z}{\partial w} = \vec{0} = [\vec{0}] \\ z > 0,\ z \to 1 \times \frac{\partial z}{\partial w} = \frac{\partial z}{\partial w} = [\vec{x}^T] \end{cases}$$

↓ also putting all derivatives together

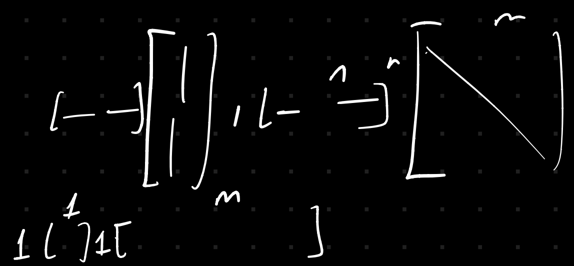
For some w, $\frac{\partial a}{\partial w}$ gives the best approximation for the instant rate of change of the activation, wrt that specific w

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

(④ ③ ② ①)

$\frac{\partial a}{\partial z}$ → either 0 or 1

1^T → previous slide



our final answer is:

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

or

$$\frac{\partial a}{\partial w} = (x_1, x_2, x_3 \ldots x_n)$$

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

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

$\sigma(\Sigma(w \otimes x) + b)$ (underbrace: z)

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

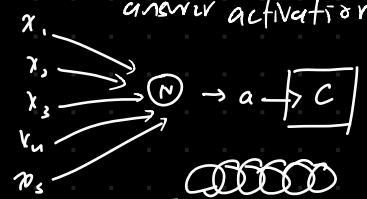
$$\frac{\partial a}{\partial b} = \begin{cases} 0 \times 1 = \vec{0} \text{ if } z \le 0 = [\vec{0}] \\ \vec{1} \times \vec{1} = [\vec{1}] \text{ if } z > 0 \end{cases}$$

(important so that the vector operations make sense dimensionally)

The Gradient of the Loss Function

Cost function

MSE: $\frac{1}{2m}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ ← answer, ↓ activation a^L



we want to find:

$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial w}$ (chain rule)

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

Recall that

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

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

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

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

several training examples (m)

$\begin{bmatrix} x_{11} & x_{21} & \cdots & x_{n1} \\ x_{12} & x_{22} & & \\ \vdots & \vdots & \ddots & \\ x_{1n} & x_{2n} & \cdots & x_{nn} \end{bmatrix}$

correct answer

Define and Say: $v = y - a^L$

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

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

$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial v}\frac{\partial v}{\partial w}$ ← we have $\frac{\partial a}{\partial w}$

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

Find $\frac{\partial C}{\partial v}$ $\frac{\partial v}{\partial w} = -(1)\left(\frac{\partial a}{\partial w}\right)$

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

Take out the constant $= \frac{1}{2m}\left(\sum_{i=1}^{m}\frac{\partial}{\partial w}(v^2)\right)$ $\left(\frac{dv^2}{dw} = \frac{dv^2}{dv}\frac{dv}{dw}\right)$ (known)

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

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

what does this mean? ↓ looking at a piecewise → make the options negative

$= \frac{1}{m}\sum_{i=1}^{m}\begin{cases} [\vec{0}]^T & \text{if } \dots \\ -v\frac{\partial z}{\partial w} & \text{if } \dots \end{cases}$ ✱ DOES not actually change the piecewise

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

Subbing in the variables, $= \frac{1}{m}\sum_{i=1}^{m}\begin{cases} [0]^T & \text{if } \dots \\ -(y_i - a^L)[\vec{x}^T] \end{cases}$

From ReLU,

$\sigma(w^T x + b) \to \max(0, w^t x + b)$

$\Rightarrow \frac{1}{m}\sum_{i=1}^{m}\begin{cases} [0]^T & \text{if } \dots \\ -\{y_i - (\max(0, \vec{w}^t\vec{x} + b))\}[\vec{x}^T] \end{cases}$

if $w^t x + b > 0$ ← always the case if it went down this path. max function is redundant

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

$\frac{\partial C}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}(\vec{w}^t\vec{x} + \vec{b} - \vec{y}_i)[\vec{x}^t]$

$= \text{scalar} \quad 1 \quad [\ ] \quad 1 \quad [\ \ \ ] \quad n$

$\frac{1}{m}\begin{cases} \vec{0}_t \\ \sum_{i=1}^{m}(w^t x + b - y)(x^t) \end{cases}$

what did we find out?

want a Jacobian for back propagation

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

$(w^t x + b - y_i)$: e_i (called error)

Want:

$$\begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \frac{\partial C}{\partial w_3} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}^T$$

w_n total weights

$$\Rightarrow \boxed{\frac{1}{n}\sum_{i=1}^{n} e_i [x]}$$

where $e_i = w^t \vec{x} - y_i$ (scalar)

Consider $n=1$ and 5 inputs, transposing our $e_i[x]^T$ so it becomes a column,

$$\frac{\partial C}{\partial \vec{w}} = \begin{bmatrix} e x_1 \\ e x_2 \\ \vdots \\ e x_5 \end{bmatrix} \begin{matrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_3} \end{matrix}$$

how C changes wrt w_1 ...

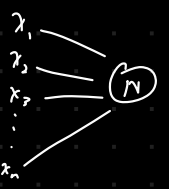
how C changes wrt w_2 ...

when you change w_1 by some amount, the corresponding change is ...

consider $\frac{1}{n}\sum_{i=1}^{n} e_i x^T$

$$= \frac{1}{n}\left(e_1 x^T + e_2 x^T + e_3 x^T \dots e_5 x^T\right)$$

$$\frac{1}{n}\left(\begin{bmatrix} e_1 x_1 \\ e_2 x_1 \\ \vdots \\ e_5 x_1 \end{bmatrix} + \begin{bmatrix} e_1 x_1 \\ e_2 x_1 \\ \vdots \\ e_5 x_2 \end{bmatrix} \dots + \begin{bmatrix} e_5 x_1 \\ e_5 x_2 \\ \vdots \\ e_5 x_5 \end{bmatrix}\right)$$



Consider a general case

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

random averages

$$\Rightarrow \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \end{bmatrix} = \frac{\partial C}{\partial \vec{w}}$$

derivative of the cost with respect to all w's, averaged out over all training examples

Derivative of the bias

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

$v = y - a^L$

From $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial v}\frac{\partial v}{\partial a^L}\frac{\partial a^L}{\partial b}$ (-1) ✓

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

* Scalar *

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

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

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

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

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

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

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

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

↓ max redundancy again, can remove

$$= \frac{1}{m}\sum_{i=1}^{m}\left\{(w^t x + b - y)\right.$$

$$= \begin{cases} 0 & \text{if } \dots \quad w^t x \le 0 \\ \frac{1}{m}\sum_{i=1}^{m}\underbrace{(w^t x + b - y)}_{c_i} & \text{if } \dots \quad w^t x + b > 0 \end{cases}$$

Now generally, $\frac{\partial C}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} c_i$

$$\boxed{\frac{\partial C}{\partial b} = \frac{1}{m}\left[c_1 + c_2 + \dots c_n\right]}$$

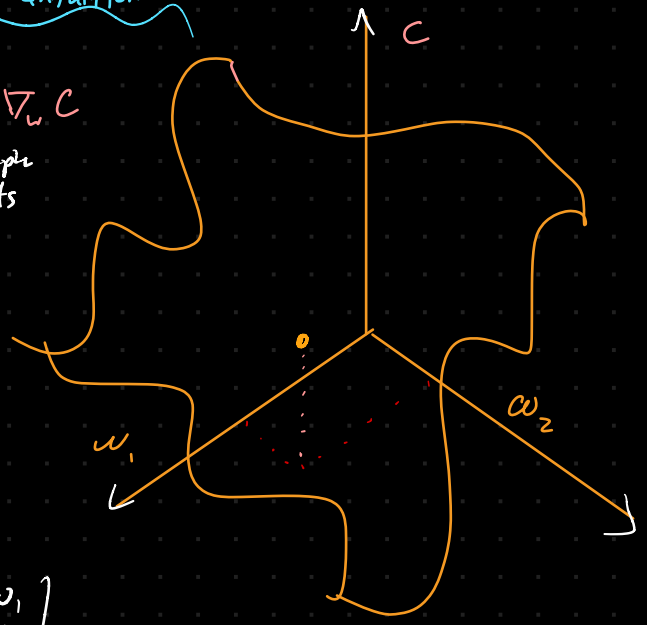
Just a scalar

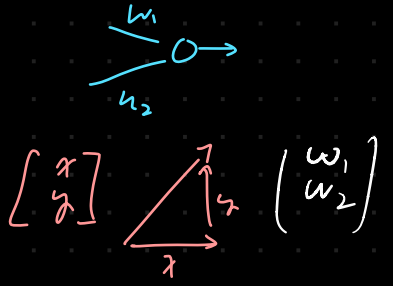
derivative of the cost wrt to the bias, averaged over all training examples.

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

Vectors can be translated

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 (evaluate $e_i x_1$ first)

$\nabla_w C$ goes in a higher direction of cost — 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

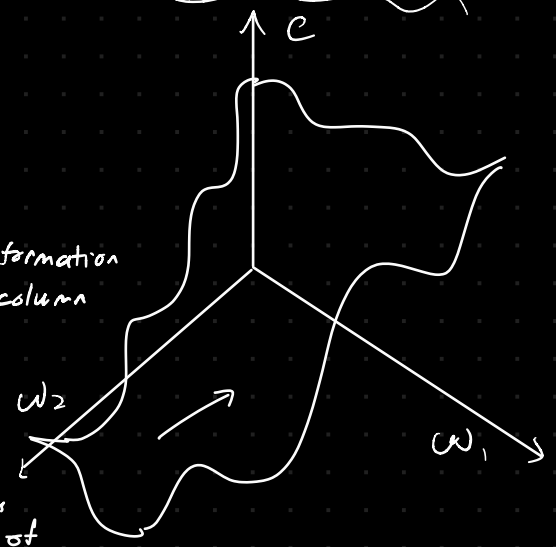
All machine learning models optimize to point in the direction of steepest descent

Gradient Descent & Stochastic Gradient Descent

$$\begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \frac{\partial C}{\partial w_3} \\ \vdots \\ \frac{\partial C}{\partial b_1} \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

$\nabla_{w,b} C$

put all information in one column vector.



The gradient points in the direction of steepest ascent

θ (defined to be all weights and biases)

Initially starts as randomly assigned

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

↓ learning rate (step size)

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

α (learning rate) depends on research

Training batch: m
↳ Can be very computationally expensive.

$$\begin{bmatrix} | \\ \vec{q} \\ | \end{bmatrix}$$ over m examples

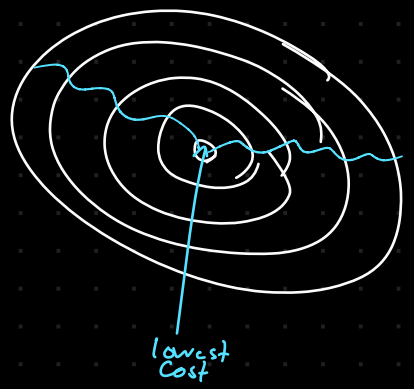
"Mini-batch" Gradient descent:

SGD → batching smaller datasets and then, running gradient descent on each batch

Shuffle the data well. Batch should represent data as a whole.

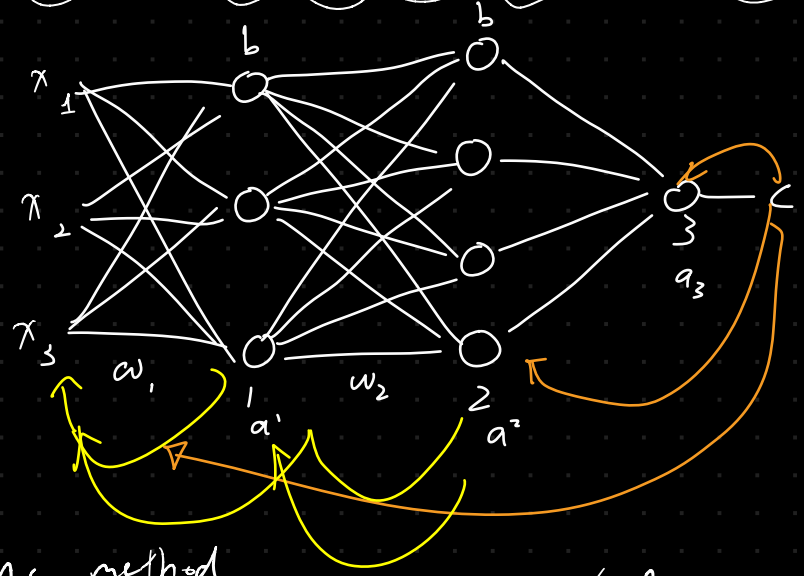
$\{x_1\}$ $\{x_2\}$

↓ 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

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

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

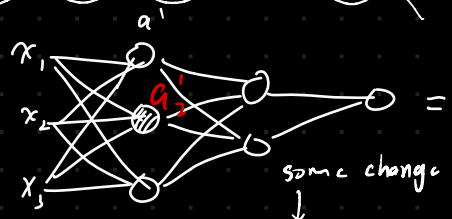
$$\frac{\partial C}{\partial w_2} = \frac{\partial a^2}{\partial w^2}\frac{\partial a^3}{\partial a^2}\frac{\partial C}{\partial a^3}$$

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

error of node j, of layer l

δ_j^l

The error of a node



Activation of node 2 of layer 1, before ReLU

$(z^1_2 + \Delta)$ is the "total error"

If we add Δ to a node, how does the cost function change?

General form error of a node (impact of a node on the cost)

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}$$

$$\delta^1_2 = \frac{\partial C}{\partial z^1_2}$$

→ 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"

1st equation – error of last node a^L

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

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}$$

$$\delta^1_j = \left(\frac{\partial C}{\partial a^1_j}\right)\left(\frac{\partial a^1_j}{\partial z^1_j}\right)$$ chain rule again

$$\delta^1_j = \frac{\partial C}{\partial a^1_j} \cdot \sigma'(z^1_j)$$ general form

Now, do the entire last layer:

$\nabla_{a^l} C$ (gradient of cost wrt a^l)

$$\begin{bmatrix} \frac{\partial C}{\partial a^l} \\ \frac{\partial C}{\partial a^l} \\ \frac{\partial C}{\partial a^l} \\ \vdots \end{bmatrix} \otimes \begin{bmatrix} \sigma'(z_1) \\ \sigma'(z_2) \\ \sigma'(z_3) \\ \vdots \end{bmatrix}$$

element wise ↑ derivative ↓

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

element-wise operation

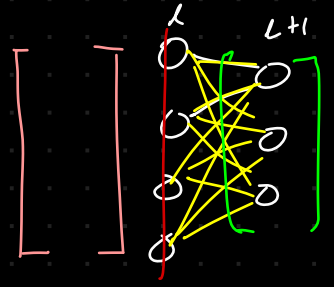
2nd equation: error of any node: $\delta^l = ((w^{l+1})^T \otimes \delta^{l+1}) \otimes \sigma'(z^l)$

= weights going into layer $l+1$

↓ weight matrix connecting $l, l+1$

↘ total error of $l+1$ layer

derivatives of the activations

$\delta^l =$ 

why does this work:

If we know errors of layer $l+1$

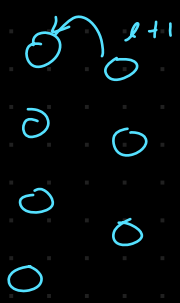
$(w^{l+1})^T$ is like going back and getting that error

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

z_1 z_2

$\sigma(w_1 a_0 + b_1) + \sigma(w_2 a_1 + b_2)$

given



$w = \begin{bmatrix} & \\ & \end{bmatrix}$ (K columns, j rows)

$w^T = \begin{bmatrix} & \\ & \end{bmatrix}$ (j columns, K rows)

Reversing through the network

using the chain rule:

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

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

$$\sigma'(z_1) \otimes w_2^T \otimes \delta^2$$

$z_1 = w_2 a_1$

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

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

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

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

eg. $\sigma(w_1 a_0 + b_1) \Rightarrow \sigma(w_2 a_1 + b_2)$

z_1; a_1, a_2

given δ^1 (error of z_1)

$$\delta^1 = \frac{\partial C}{\partial z_1}$$

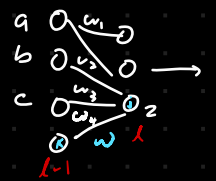
$$\frac{\partial C}{\partial b_1} = \left(\frac{\partial C}{\partial z_1}\right)\left(\frac{\partial z_1}{\partial b_1}\right)$$

$\frac{\partial C}{\partial z_1} = \delta^1$, $\frac{\partial z_1}{\partial b_1} = 1$

$$\boxed{\frac{\partial C}{\partial b_1} = \delta^1}$$

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

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad (\text{Scalar})$$



have $\frac{\partial C}{\partial z^l_j}$ δ

know $\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{jk}}$

(Taking derivative)

$z^l_j = (a w_1 + b w_2 + c w_3 + k w + b)$

dot product

$z^l_j = (a w_1 + b w_2 + c w_3 + k w + b)$

$\frac{\partial z^l_j}{\partial w} = k$

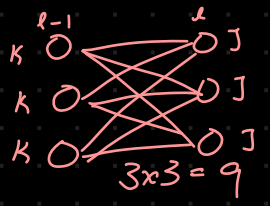
what is k? activation of node $l-1$ from node k

Vectorized form:

Example:

from kth, into jth

error of the node its coming from in l



$$w_l = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{matrix} K \\ \\ J \end{matrix}$$

(3, 3)

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

$$\frac{\partial C}{\partial w_2} = \begin{bmatrix} a^1_1 \delta^2_1 & a^1_2 \delta^2_1 & a^1_3 \delta^2_1 \\ a^1_1 \delta^2_2 & a^1_2 \delta^2_2 & a^1_3 \delta^2_2 \\ a^1_1 \delta_3 & a^1_2 \delta_3 & a^1_3 \delta_3 \end{bmatrix}$$

Calculate like:

$$\delta^2 = \begin{bmatrix} \delta^2_1 \\ \delta^2_2 \\ \delta^2_3 \end{bmatrix}^3 \qquad a_1 = \begin{bmatrix} a^1_1 \\ a^1_2 \\ a^1_3 \end{bmatrix}^1$$

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

by matrix vector multiplication

(n, 1) (1, m)

n m

0 0

0 0

0 0

0 0

Tying everything together

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

(e_i)

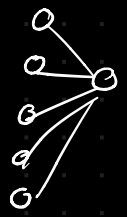
$$\frac{\partial C}{\partial w} \begin{cases} \vec{0} & \\ \frac{1}{M}\sum_{i=1}^{m} (w^T x + b - y)\, x^T & \end{cases}$$

(e_i)

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

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

These are equivalent



e^i

$e_i x^T$ are only wrt one node

BP Equations

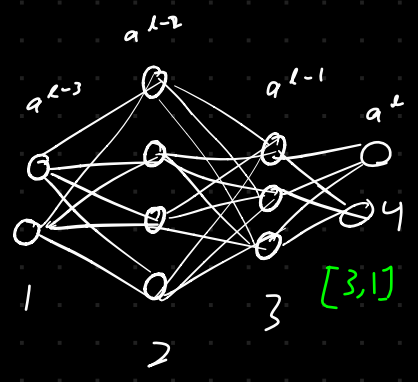
(1) $\delta^L = \nabla_a C \circ \sigma'(z^L)$ (last layer)

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

(3) $\frac{\partial C}{\partial b^l} = \delta^l$

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

(1) Feed forward and store z^l and a^l values of all l layers



$[1] \; j \; [w_{11} \; w_{12} \; w_{13}]$ (k)

$\downarrow$

$[1,3]$

$\downarrow$

$[3,1]$

0 keeps dimension

(2) compute the cost

(3) use eq 1 to calculate δ^L

(4) use eq 2 to compute all δ^l terms

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

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

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

unroll all matrices / flatten

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

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

forward pass ⟶

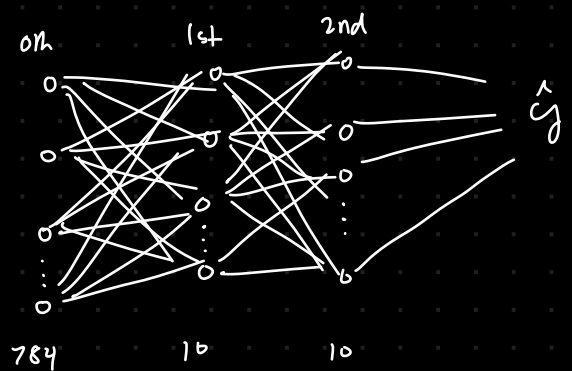
back propagation ⟵

back propagation ⟵

C ∪

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

10 classes



(1) Forward propogation

$A^0 = X$ (first layer)

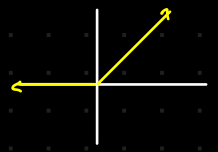
$Z^1 = W^1 A^0 + b^1$

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

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

Use ReLU (Z^1)

adds complexity rather than a linear model



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

$A^2 = \text{Softmax}(A^2)$ $\quad \frac{e^{z_i}}{\sum_{i=1}^{n} e^{z_i}}$

our outputs are converted to a probability between 0 and 1

3Blue1Brown - Deep Learning Series

Neurons hold a number called an activation

Activations of one layer determine the activations of the next layer

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\text{colum } a^{(l)} = \sigma\left(\begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ a_3^{(l)} \\ \vdots \\ a_n^{(l)} \end{bmatrix}_n \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & & & \\ \vdots & & \ddots & \\ w_{k,n} & \cdots & & w_{k,n} \end{bmatrix}_k + \begin{bmatrix} b_1 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} \\ a^{(l+1)} \\ \\ \end{bmatrix}\right)$$

Applying sigmoid elementwise to each of the entries

Each neuron is a function
Network is a function of functions

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

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

Global minimum -> crazy hard

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

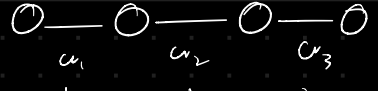
Compute ∇C.

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

↑ negative (descent)

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

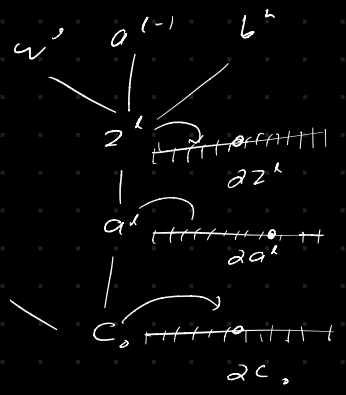
eg.



$$C = \frac{1}{2}(y - a^L)^2 = C_0$$

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

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



$$\frac{\partial C_0}{\partial w^l} = \frac{\partial z^l}{\partial w^l}\frac{\partial a^l}{\partial z^l}\frac{\partial C_0}{\partial a^l} = (a^{l-1})(\sigma'(z^l))(2)(a^L - y)$$

chain rule

$$\frac{\partial C}{\partial a^L} = 2(a^L - y)$$

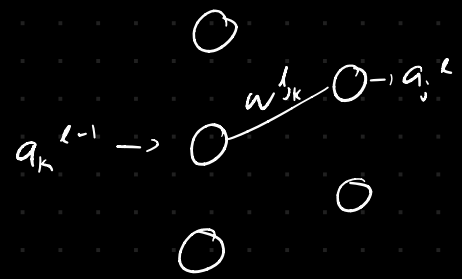
$$\frac{\partial a^l}{\partial z^l} = \sigma'(z^l)$$

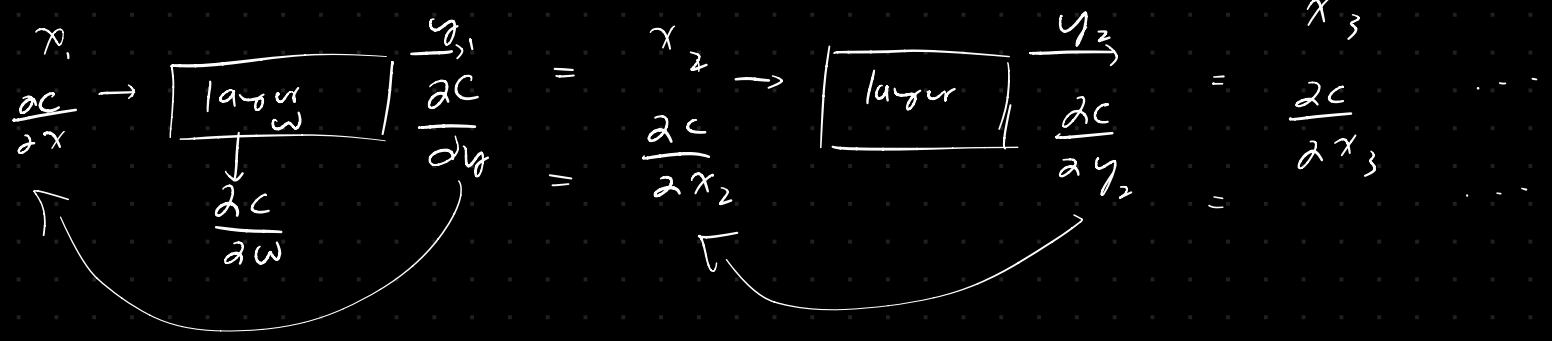
$$\frac{\partial z^l}{\partial w^l} = a^{l-1}$$

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^1} \\ \frac{\partial C}{\partial b^1} \\ \vdots \\ \frac{\partial C}{\partial w^l} \\ \frac{\partial C}{\partial b^L} \end{bmatrix}$$

$$\frac{\partial C_0}{\partial b^L} = \frac{\partial z^l}{\partial b^l}\frac{\partial a^l}{\partial z^l}\frac{\partial C_0}{\partial a^l}$$

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





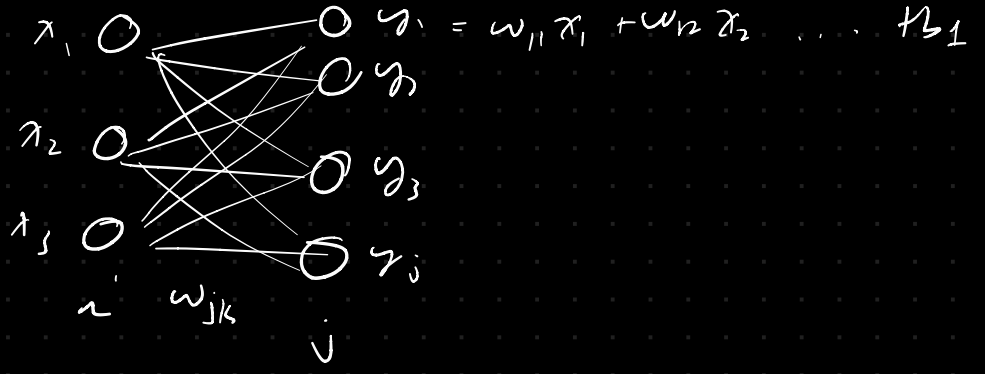
$$\frac{\partial E}{\partial w} = \frac{\partial C}{\partial y} \frac{\partial y}{\partial w}$$

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

```
class Layer:
    self.input
    self.output
    forward(self, input)
    backward(self, output_gradient, learning_rate)
```

$$\begin{pmatrix} w_{11} & w_{12} & \cdots \\ w_{21} & \cdots & \\ \vdots & & \\ j & & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \end{pmatrix}$$

$j \times i$ (k) $\quad i \times 1 \quad j \times 1$



$$y_1 = w_{11} x_1 + w_{12} x_2 \cdots + b_1$$

$$\begin{matrix} & & K & & 1 & & 1 \\ j & \begin{bmatrix} w_{11} & w_{12} & \cdots \\ w_{21} & \ddots & \end{bmatrix} & K & \begin{bmatrix} x_1 \\ x_2 \\ \\ x_K \end{bmatrix} & + & \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \end{bmatrix} & j \end{matrix}$$

dot product

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

$$\frac{\partial C}{\partial w_{jk}} = \frac{\partial C}{\partial y_j} x_i$$

$$\frac{\partial C}{\partial w} = \begin{bmatrix} \frac{\partial C}{\partial w_{11}} & \frac{\partial C}{\partial w_{12}} & \cdots \\ \frac{\partial C}{\partial w_{12}} & \ddots & \frac{\partial C}{\partial w_{jk}} \end{bmatrix}$$

$$\frac{\partial C}{\partial w} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_i \end{bmatrix}$$

Chapter 4 – What is a GPT?

Generative → Generated

pre-trained → trained

transformer → Neural network

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 eachother in the space

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

Feed to multilayer perceptron

Go back at 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

Black box of tunable 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 predetermined vocabulary 50000

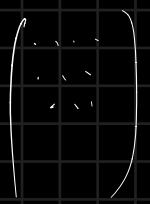
Embedding matrix $\begin{pmatrix} \text{word \#1} & \text{word \#2} & \text{word \#3} & \cdots \end{pmatrix} = W_E$

one column per word
Determines what word each vector turns into
in the first step

Begin random

word => embedding

Geometric points in a space

 12,288 coordinates

Directions have a semantic meaning

E(king) − E(queen) ≈ E(man) − E(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 × 12288 = 617558016
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

output:

Map matrix of tokens, and then apply softmax

Un embedding matrix W_u

one row / word in vocab

Each row has same dim as embedding dimension

617558016

Softmax: probability distribution

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

$$\frac{e^{x_i}}{\sum_n e^{x_n}}$$ picking maximizing input

GPT adds a constant to this

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

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

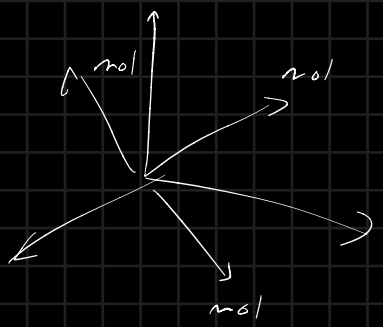
as $T \downarrow$, more weight on high values

outputs are referred as logits.

(for next word prediction)

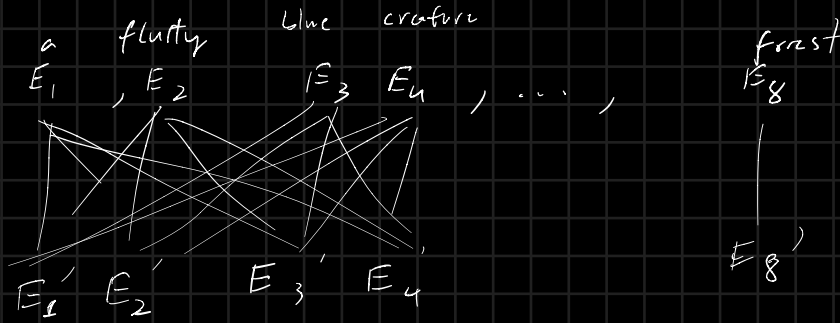
Chapter 6 – Attention transformers

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

$$W_Q \begin{pmatrix} \\ \\ \\ \end{pmatrix} \begin{pmatrix} \\ \\ \vec{E}_4 \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \vec{Q}_4 \\ \\ \end{pmatrix}$$

Question

By all embeddings

↓

Do backpropagation

Key matrix

$a \rightarrow \vec{E}_1 \xrightarrow{W_K} \vec{K}_1$

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

Answers the question from the query

Query & Key dimensions are the same

How much do they match?

E_1

E_1 $K_1 \cdot Q_1$ . . .

E_2 $K_2 \cdot Q_1$

E_3 $K_3 \cdot Q_1$

as value ↑, embeddings "attend to each other"

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

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$\begin{bmatrix} Q_1 & Q_2 & \cdots \\ | & | & \end{bmatrix}$ $\begin{bmatrix} K_1 & K_2 & \cdots \\ | & | & \end{bmatrix}$

Never want later words to influence later words

Want the bottom triangle of matrix to be zero

=> Solution

set all features here to be $-\infty$

$-\infty$	1	2
$-\infty$	$-\infty$	3
$-\infty$	$-\infty$	$-\infty$

"masking"

Update embeddings

use value matrix to add to word without context

Apply to all tokens

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

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

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

Apply row wise, add column wise to original embedded word

This is called one head of attention.

Q: 12288×128

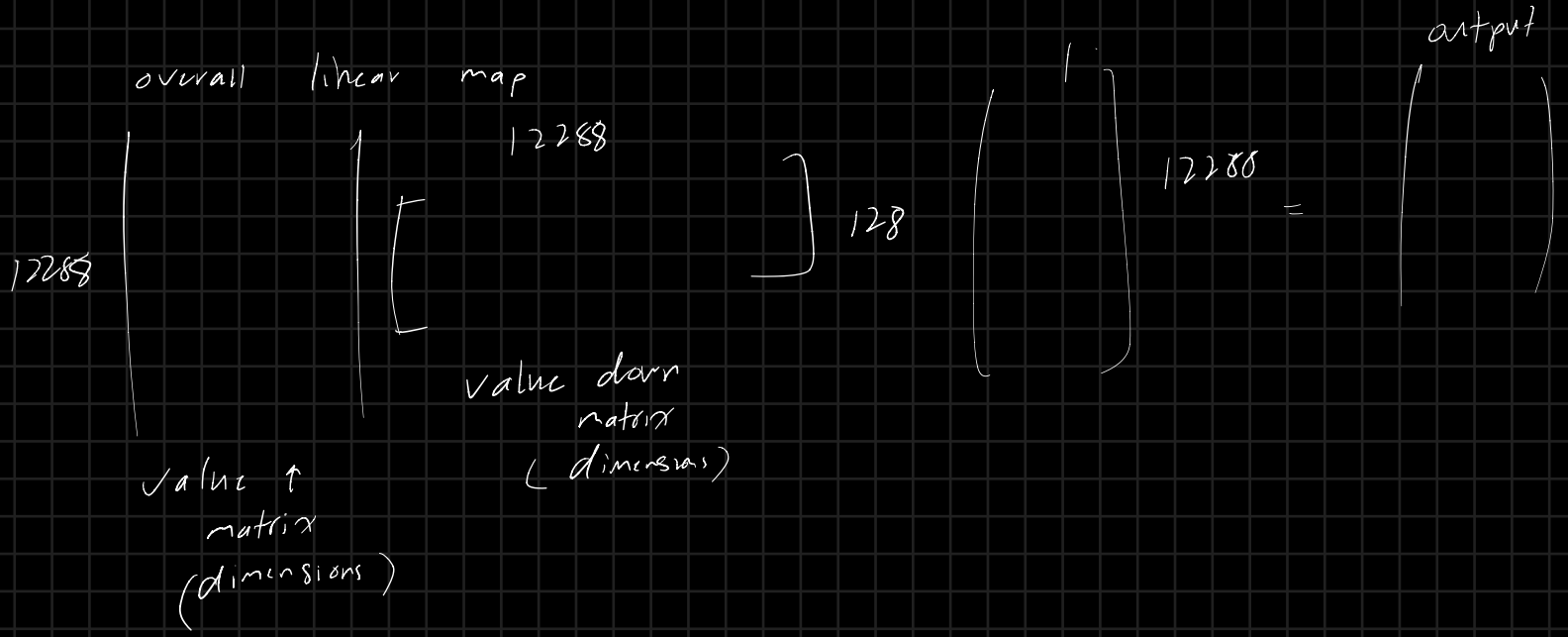
K: 12288×128

V: 12288×12288

Value = K + Q (Smaller amount)

Split value matrix into 2 smaller matrices

overall linear map



Cross attention (two types of data)

Self attention (one type of data)

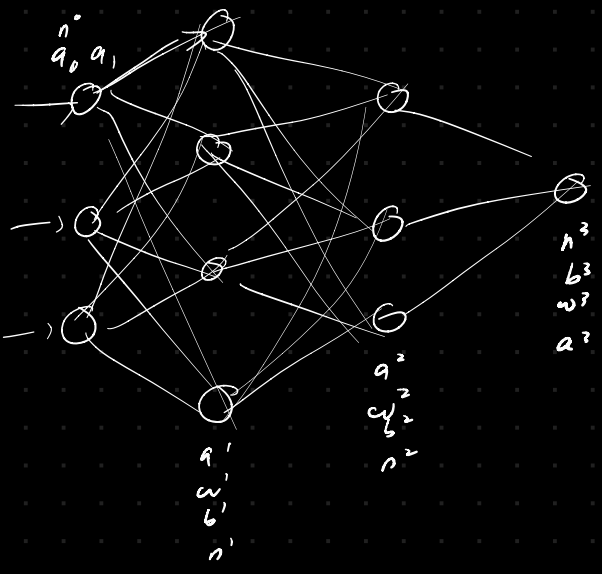
Full attention block => Multiple self attention with distinct key and queries

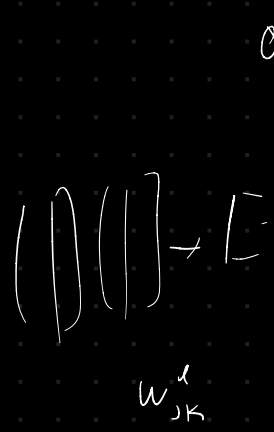
GPT uses 96 self attention heads

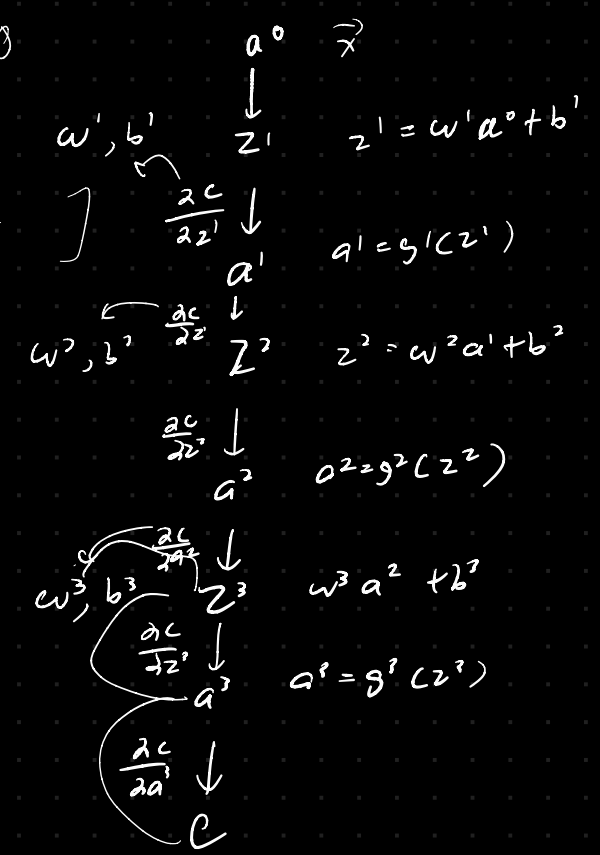
All heads give a proposed change. Add all changes to the original embedded word. This embedded vector is one output of the attention block. } Run in parallel

In practice

All value up matrices are stuck together







Denoting n^l # of neurons in the layer

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

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

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

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

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

Let $\vec{x}$ be the input vector: $a[0] = x$.

$$z^l = (w^l)(a^{l-1}) + |b^l|$$

$$a^l = g^l(z^l)$$

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

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

δ $(a^2)^T$

If we shift w, how does the cost change?

of the very last layer before computing the cost.

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

δ $\quad$ 1

$a = \sigma(z)$

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

of any layer:

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

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

Now continue backpropogating our data pipeline:

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

δ^3 $\quad$ δ^2 $\quad$ $(w^3)^T$ $\quad$ $\sigma'(z)$ $\quad$ (a^1)

describe as functions of eachother.

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

δ $\quad$ 1

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

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

δ^3 $\quad$ δ^2 $\quad$ δ^3

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

δ $\quad$ δ^3 $\quad$ 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

$$\frac{\partial C}{\partial w^l} = \frac{\partial C}{\partial z^L} \frac{\partial z^l}{\partial w^l}$$ for any layer l

$$\frac{\partial C}{\partial b^l} = \frac{\partial C}{\partial z^L} \frac{\partial z^l}{\partial b^l}$$ for any layer l

} have intermediates that we need to calculate

For calculating the partial derivatives, we should calculate

① $\frac{\partial C}{\partial z^L}$ ② $\frac{\partial C}{\partial z^l}$ ③ $\frac{\partial z^l}{\partial w^l}$ ④ $\frac{\partial z^l}{\partial b^l}$

① of last layer

② of layer l

we need to define this in terms of $\frac{\partial C}{\partial z^{l+1}}$

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

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

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

$$\sigma'(z) = a^2(1-a^2)$$

All together:

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

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 $\frac{\partial C}{\partial z^l}$, $l = L-1, L-2 \ldots$

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

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

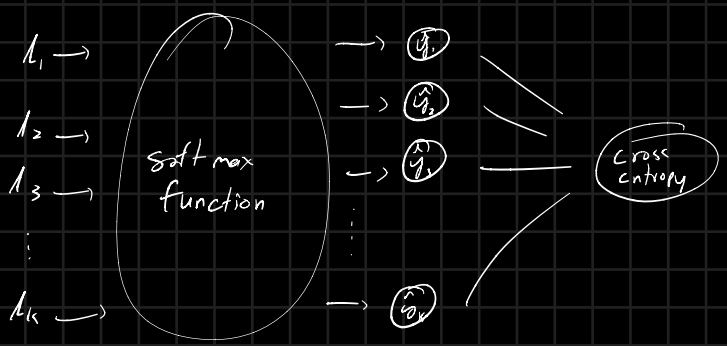
$$\frac{\partial z^l}{\partial w^l} = a^{l-1} \qquad \frac{\partial C}{\partial w^l} = \frac{\partial C}{\partial z^l} a^{(l-1)^T}$$

$$\frac{\partial z}{\partial b^l} = 1 \qquad \frac{\partial C}{\partial b^l} = \frac{\partial C}{\partial z^l}$$

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.

Softmax + Cross entropy



$$\frac{\partial L}{\partial l_n} = \hat{y}_n - y_n$$

$$\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx}$$

$$\frac{d}{dx}\left(\ln(x)\right) = \frac{1}{x}$$

Deriving Softmax

$$\hat{y}_i = \frac{e^{l_i}}{\sum_{j=1}^{k} e^{l_j}}$$

want $\frac{\partial \hat{y}_i}{\partial l_n}$ $\quad i \in (0, k)$ $\quad n \in (0, k)$

① $i = n$
② $i \neq n$

$i = n$,

$$\frac{\partial \hat{y}_i}{\partial l_n} = \frac{e^{l_i} \sum e^{l} - e^{l_i} e^{l_n}}{\left(\sum e^{l}\right)^2}$$

$$= \frac{e^{l_i}}{\sum e^{l}} \cdot \left(1 - \frac{e^{l_n}}{\sum e^{l}}\right)$$

Softmax with idx i $\quad$ Softmax with index n

$$= \hat{y}_i (1 - \hat{y}_n)$$

But, $i = n$

$$= \hat{y}_n (1 - \hat{y}_n)$$

$i \neq n$

$$\frac{\partial \hat{y}_i}{\partial l_n} = \frac{0 \times \sum e^{l} - e^{l_i} e^{l_n}}{\left(\sum e^{l}\right)^2}$$

$$= \frac{-e^{l_i}}{\sum e^{l}} \cdot \frac{e^{l_n}}{\sum e^{l}}$$

$$= -\hat{y}_i \hat{y}_n$$

Cross entropy:

$$L_i = -y_i \log(\hat{y}_i)$$

$$L = -\sum_i y_i \log(\hat{y}_i)$$

$$\frac{\partial L}{\partial l_n} = -\sum y_i \frac{1}{\hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial l_n}$$

Case 1, $i = n$

$$\frac{\partial L}{\partial l_n} = -\sum_{i=n} \frac{y_i}{\hat{y}_i} (\hat{y}_n)(1 - \hat{y}_n)$$

Since $i = n$

$$= -\sum_{i=n} y_n + y_n \hat{y}_n$$

$$= \hat{y}_n y_n - y_n$$

Case 2:

$i \neq n$

$$\frac{\partial L}{\partial l_n} = +\sum \frac{y_i}{\hat{y}_i} (-\hat{y}_i y_n)$$

$$\frac{\partial L}{\partial l_n} = \sum_{i \neq n} y_i y_n$$

Total: $-y_n + \sum_i y_i \hat{y}_n$

If y_i is one hot encoded,

$$\sum y_i \hat{y}_n = \hat{y}_n$$

$$= \hat{y}_n - y_n$$

Optimizers :

① GD

$$\theta = \theta - \alpha \nabla_\theta J(\theta)$$

② SGD (same)

③ SGD + momentum

$$v = \gamma v + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - \alpha v$$

④ SGD + momentum + acceleration

$$v = \gamma . v + \eta \nabla_\theta J(\theta - \gamma . v)$$

$$\theta = \theta - \alpha v$$

Ada Grad (learn more in 1 dir)

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \nabla_{\theta_{t,i}} J(\theta_{t,i})$$

$$G_{t,ii} = G_{t-1,ii} + \nabla^2_{\theta_{t,i}} J(\theta_{t,i})$$

⑤ Adadelta

⑥ Adam (Ada delta + Momentum)

⑦ NAdam (Adadelta, Momentum, acceleration)

Michael Nielson: Neural Networks & Deep Learning

Chapter 1:

V_1: primary visual cortex:

Neural networks: infer rules for recognizing hand written digits.

1.1

perceptron: artifical neuron

Used in production -> sigmoid neuron

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

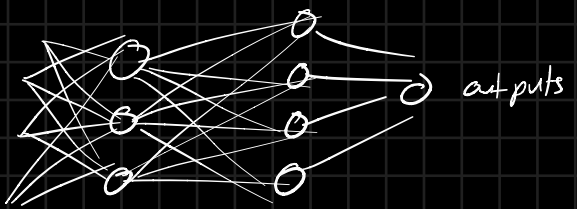
Binary as in input/output are 0/1.

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.

Weigh up evidence to make decisions.



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

rewrite:

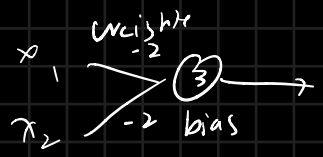
$\sum_j w_j x_j = \vec{w} \cdot \vec{x}$

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

Move threshold to other side -> called the bias

b is => ↑ fire chance
small => ↓ fire chance

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



Logic gates can be implimented by neural networks.

??? NAND (pg 5-6)

Think of input perceptrons as speicial units defined to output desired values

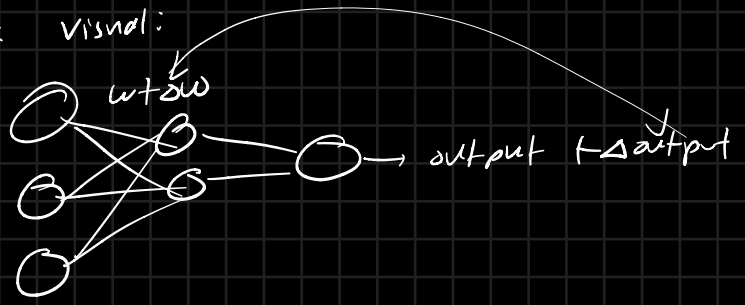
output, but no inputs (pg. 6)

Improvement => logic gates that learn to solve problems.

1.2 – Sigmoid neurons

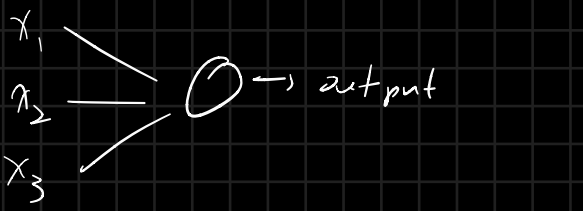
Suppose we make a small change in a weight / bias in the network.
↳ observe how this change causes a corresponding change in the output from the network.

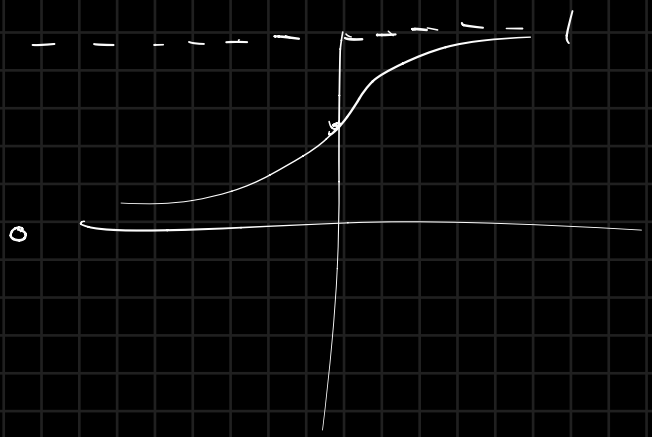
As a visual:



Using this, we try to adjust weights & biases to get 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 $0 \leq x \leq 1$, $x \in \mathbb{R}$

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

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

very positive → approaches 1

very negative → approaches 0

From Calculus,

$$z = \sum w x + b$$

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

rate of change $\times$ change in ind variable of partial derivative

Any activation function applied $f(w \cdot x + b)$

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

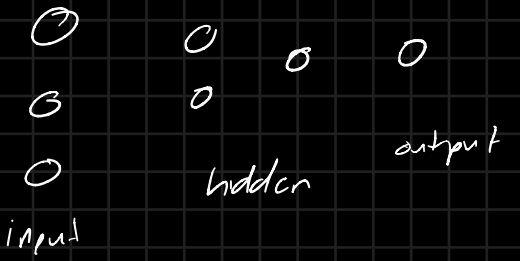
when $w \cdot x + b = 0$, the other perceptrons "output/decision" may get scaled, and thus, lead to a slight nudge in the output, leading to a different decision before scaling $c \to \infty$.

1.3 – Architecture of a neural network

Input layer → input neurons

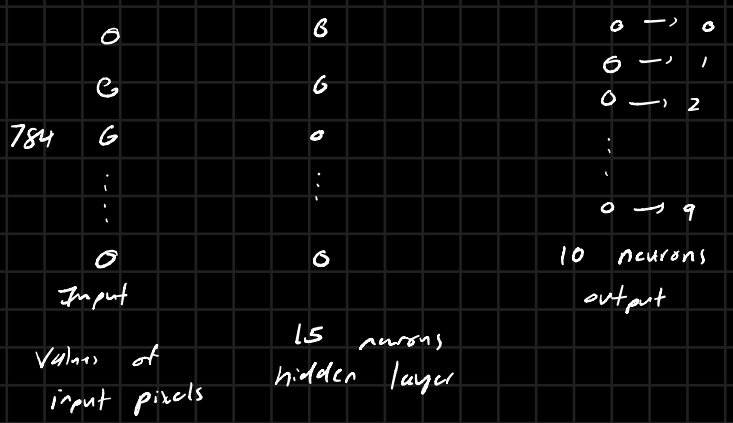
output layer → output neurons

hidden layer → hidden neurons



Feed forward NNs

Digit classification tasks:



784 = 28 × 28 pixels

grey scale values, between 0 and 1.

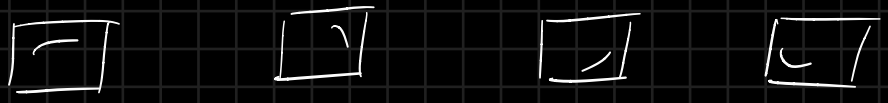
Why 10 neurons => Inefficient?

Test out to find out. 10 output neurons is better

From first principles:

eg. first output neuron → decides if disit is a zero.
→ weighs up all evidence from hidden layer of neurons.

Neuron 1 neuron 2 neuron 3 neuron 4



If all 4 fire, then we conclude disit is a zero.
overlapping pixels fire more.

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

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. 784×1

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

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

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

$\vec{a}$ vector of outputs
n training examples
w, b collection of all weights and biases.

aka MSE

Small cost → high accuracy

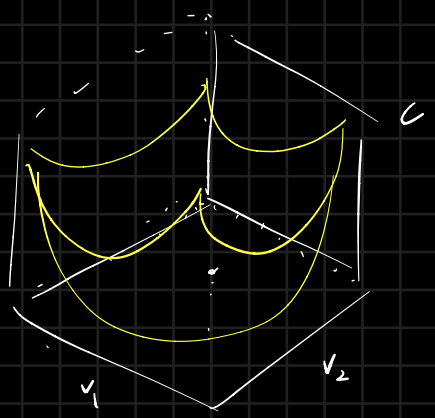
high cost → low accuracy

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

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

eg. $C(\vec{v})$, real-valued function

$C(v_1, v_2)$



Find where C achieves a global minimum.

Cannot use calculus to find the extremum → too many calculations!

first and second derivatives

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

Small v_1, and v_2 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 ΔC 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 $\Delta v = -\eta \nabla C$

η is a small positive parameter called the learning rate.

$\eta > 0$ $\nabla C > 0$

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

This guarantees ΔC to be a negative number

$$\Delta C \leq 0$$

positional position 2

$$v \to v' = v - \eta \nabla C$$

Choose small η so that $-\eta \|\nabla C\|^2$ approximates well.

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

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

Gradient descent is the most optimal strategy.

Suppose we want to make a move Δv as to decrease C as much as possible.

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

Define ϵ, st we can constrain $\|v\| = \epsilon$, for $\epsilon > 0$.

The choice of Δv that minimizes ΔC, is $\Delta v = -\eta \nabla C$ where $\eta = \frac{\epsilon}{\|\nabla C\|}$

Cauchy 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.

$$|\langle u, v\rangle|^2 \leq \langle u, u\rangle \cdot \langle v, v\rangle$$

$$|\nabla C \cdot \Delta v| \leq \|\nabla C\| \|\Delta v\|$$

ϵ

to minimize, we need to maximize the cosine between ∇C and ΔC

v points in the opposite direction of ∇C.

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

$\nabla C \quad \frac{\partial C}{\partial w_k} \cdots \quad \nabla C = \cdots \frac{\partial C}{\partial b_l}$

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

$$b_l = b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

Because C is of the form $C = \frac{1}{n}\sum_x C_x$, $C_x = \frac{1}{2}\|y(x) - a\|$,

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

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

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

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

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

for SGD:

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

$$b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{x_j}}{\partial b_l}$$

Repeat 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 ↓ → more training examples to work with, but, will be significantly slower.

60,000 images => 50000 images (training)
=> 10,000 images (validation)

```
class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(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 kth neuron into the jth neuron.

10 neurons

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

element-wise · second layer of neurons

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}_{10\times 1} = \sigma\left(\begin{bmatrix} & \xrightarrow{784} & \\ & \cdots & \\ \downarrow & & \\ 10 & & \end{bmatrix} \begin{pmatrix} \\ \\ \\ \end{pmatrix}_{784} + \begin{pmatrix} \\ \\ \\ \end{pmatrix}_{10} \right)$$

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

This is applied element wise.

```
def feedforward(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

sk: kit - learn → LIBSVM

Simple algorithm + good_training_data ≥ sophisticated algorithm

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

face? → nose? → left eye? → right eye? , ...)

face?

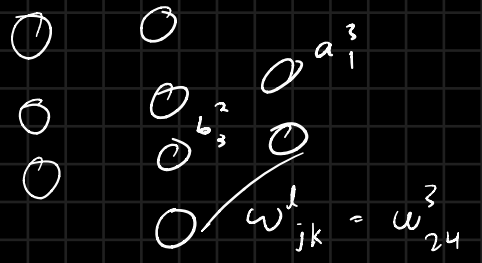
even further → eye? → eye brow? → eyelashes → iris

⇒ 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^l_{jk} kth neuron from $l-1$ layer, into jth neuron of lth layer.



b^l_j, a^l_j

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

a^l_j: vector; w^l_{jk}: matrix; a^{l-1}_k: vectors; b^l_j: vector

As a matrix, w^l is the weights matrix of weights that connect to the lth layer of neurons.

to the jth row, from the kth column

To vectorize an element wise function,

$$\sigma\left(\begin{bmatrix}2\\3\end{bmatrix}\right) = \begin{bmatrix}\sigma(2)\\ \sigma(3)\end{bmatrix}$$

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

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 $\frac{\partial C}{\partial b}$ with any w / b

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

$y(x)$: desired output; $a^L(x)$: vector activations when x is input.

2 Statements to clarify.

① $C = \frac{1}{n} \sum_x C_x$ 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!

=> 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.

=> Not something we can change and not something a model learns.

2.3 - Hadamard product

Element wise product of 2 vectors

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

2.4 - Four equations of backpropagation

Introduce δ_j^l called the error of jth neuron into the lth layer

sits at the jth neuron of layer l. It adds a change Δz_j^l to the neurons weighted input

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

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

Choose Δz_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

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

δ^l, error of layer l

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

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

cost with output activation

derivative activation function with weighted sum

* output layer error only

$\forall$ for j neurons in a layer

$\Rightarrow$ correct notation would be k.

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

$\Rightarrow$ 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 δ^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 δ^L

Now, we can compute δ^L for any layer in the network.

(3) Rate of change of cost wrt any bias in the network

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

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

(4) Rate of change of cost wrt any weight in the network

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

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

weight outputs from low activation neurons learn slowly.

Depending on activation func, neurons learn differently, and they can be predicted based on activation function graph behaviors.

2.6 – Backpropogation 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 $\delta^L = \nabla_a C \odot \sigma'(z^L)$ at the end
4. For each previous layer: compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
5. The gradient of a cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$, $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

For SGD,

$$w^l = \frac{\eta}{m} \sum_x \delta^{x,l} \odot \sigma'(a^{x,l-1})^T$$

$$b^l = \frac{\eta}{m} \sum_j \delta^{x,l}$$

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

$$\Delta C \approx \frac{\partial C}{\partial a^l_n} \frac{\partial a^l_n}{\partial a^{l-1}_m} \frac{\partial a^{l-1}_m}{\partial a^{l-2}_r} \cdots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk}$$

$$\Delta C = \sum_{mnpq\dots} \frac{\partial C}{\partial a^l_n} \frac{\partial a^l_n}{\partial a^{l-1}_m} \frac{\partial a^{l-1}_m}{\partial a^{l-2}_r} \cdots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk}$$

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, for 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.

Chapter 3

Cross entropy function

Regularization methods (L1, L2)

Dropout

Initializing weights

Choosing good hyper parameters.

5.1

Learning is slow: partial derivatives are small

when neuron output is close to 1, curve gets flat and $\sigma'(z)$ is small.

Replace cost with cross entropy

$$C = -\frac{1}{n} \sum_x [y \ln a + (1-y) \ln(1-a)]$$

n: total # of items in training data
x: all training inputs
y: corresponding desired output

If actual output $\approx$ desired output
cost ≈ 0

cross entropy avoids the problem of learning slowly

Deriving the cost:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y)$$

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y)$$

The rate of learning is controlled by $(a-y)$

The larger error, the faster the neuron learns

When we use the cross entropy, the $\sigma'(z)$ term gets cancelled out (no need to worry about vanishing gradients)

$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y)$$

with the cross entropy: The neuron learns fast regardless of if it is unambiguously wrong

As a vector, per example:

$$\sum_j [y_j \ln(a_j^L) + (1-y_j) \ln(1-a_j^L)]$$

over all j classes

Two classical probability distributions

If output neurons are from Sigmoid activation, then, cross entropy is better.

Exercises:

if $a = \sigma(z) \approx y$

Then

$$C = -\frac{1}{n} \sum_x [y \ln y + (1-y) \ln (1-y)]$$

Consider inside only:

$$= y \ln(y) + (1-y)(\ln(1-y))$$

$$= y \ln(y) + \ln(1-y) - y \ln(1-y)$$

$$= \ln(y^y) + \ln(1-y) - \ln(1-y)^y$$

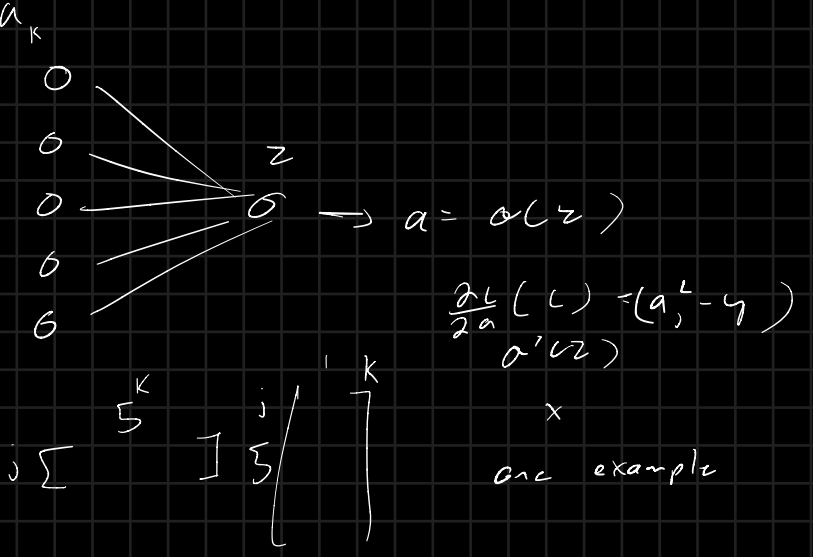
$$= \ln\left(\frac{y^y (1-y)}{(1-y)^y}\right)$$

$$= \ln\left(\frac{y^y}{(1-y)^{y-1}}\right)$$

Binary cross entropy

Quadratic cost:

$$\frac{\partial C}{\partial w_{jk}} = \frac{1}{n} \sum_x a_k^{L-1} (a_j^L - y_j) \sigma'(z_j^L)$$



$$= [\quad]$$

$$\delta^L = a^L - y \quad \text{(element wise, one example)}$$

what is $\nabla_a C$?

$$C = -\frac{1}{n} \sum_x \left[y \ln a + (1-y) \ln (1-a) \right]$$

$$\frac{\partial C}{\partial a} = -\frac{1}{n} \sum_x \left(\frac{1}{a} + (1-y)\left(\frac{1}{1-a}\right)(-1) \right)$$

$$= -\frac{1}{n} \sum_x \left(\frac{1}{a} + (y-1)\left(\frac{1}{1-a}\right) \right)$$

$$= -\left(\frac{1}{a} + (y-1)\left(\frac{1}{1-a}\right) \right)$$

$$= (1-y)\left(\frac{1}{1-a}\right) - \frac{1}{a}$$

$$= \frac{1-y}{1-a} - \frac{1}{a}$$

$$\frac{(1-y)(a)}{(1-a)(a)} - \frac{(1-a)}{(a)(1-a)}$$

origin of the cross entropy:

want backprop to cancel out $\sigma'(z)$

$$\frac{\partial c}{\partial b} = \frac{\partial c}{\partial a}\sigma'(z)$$

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

$$\frac{\partial c}{\partial b} = \frac{\partial c}{\partial a}a(1-a)$$

Need $\frac{\partial c}{\partial a} = \frac{a-y}{a(1-a)}$ $a(1-a)$ cancels out

Integrating:

$$C = -[y\ln(a) + (1-y)\ln(1-a)] + C$$

one example

Average over all examples

$$C = -\frac{1}{n}\sum_x \ldots$$

cross entropy measures how "surprised we are" when we learn the true value of y.

To eliminate x_j means that $\frac{\partial c}{\partial a} = \frac{1}{x_j}$. This is not possible because the cost doesn't take in the input to the last layer.

$a_j^L = \frac{e^{z_j^L}}{\sum e^{z_j^L}}$ forms a probability distribution

Log-likelihood cost function

$C = -\ln a_j^L$

prob ≈ 1, small cost

Can show that

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j)$$

Softmax + log-likelihood cost
=
sigmoid activation + cross entropy

$\frac{\partial C}{\partial w}$
$\frac{\partial C}{\partial b}$ Same

As probabilities

$a_j^L = \frac{e^{cz_j^L}}{\sum_k e^{cz_k^L}}$ $c \in \mathbb{R}$

outputs follow a prob dist
since they add up to 1.

If $c \to \infty$, the limiting value is still 1

Sets max value to highest value over its
classes.

Show for Softmax + log-likelihood

$$\delta^L_j = a^L_j - y_j$$

$$C = -\ln a_j \qquad \frac{\partial C}{\partial a} = -\frac{1}{a_j}$$

$$\frac{\partial C}{\partial z} = \underbrace{\frac{\partial C}{\partial a}}_{\text{loss}} \underbrace{\frac{\partial a}{\partial z}}_{\text{loss}} \underbrace{}_{\text{softmax}}$$

$$= \left(-\frac{1}{a_j}\right)\Big($$

Vector

Softmax: $\mathbb{R}^n \to \mathbb{R}^n$

Jacobian of Softmax: $\neq 0$ if diagonal $\neq 0$

Vector function

Take partial derivative of the log of the output.

$$\frac{\partial}{\partial z_j} \ln(a_j) = \frac{1}{a_j} \frac{\partial a_j}{\partial z_j}$$

$$a_j \frac{\partial}{\partial z_j} \ln(a^L_j)$$

$$\ln(a_j) = \ln\left(\frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}\right) = z_i - \ln\left(\sum_{i=1}^{n} e^{z_i}\right)$$

$$\frac{\partial}{\partial z_j} \ln(a_j) = \frac{\partial z_i}{\partial z_j} - \frac{\partial}{\partial z_j} \ln\left(\sum_{i=1}^{n} e^{z_i}\right)$$

$$\downarrow$$

$i = j$, 1

0, otherwise

$$-\left(\frac{1}{\sum_{i=1}^{n} e^{z_i}}\right)\left(\frac{\partial}{\partial z_j} \sum_{i=1}^{n} e^{z_i}\right)$$

$$= \left(\frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}\right)$$

$$= 1 - a_j^2$$

Jacobian:

$$\begin{pmatrix} a_1(1-a_1) & -a_1 \cdot a_2 & \cdots \\ -a_2 a_1 & \ddots & \\ \vdots & & \end{pmatrix}$$

$$\frac{\partial C}{\partial \text{softmax}} = 1 - a_j^2$$

$$\frac{\partial C}{\partial z_j} = -\sum_{i=1}^{C} \frac{y_i}{a_j}(1 - a_j)$$

$$\frac{\partial C}{\partial z_j} = \sum_{i=1}^{C} y_i a_j - \sum_{i=1}^{C} y_i$$

$$\frac{\partial C}{\partial z_j} = s_j \sum_{i=1}^{C} y_i - y_j = s_j - y_j$$

3.2 - Overfitting & Regularization

The test of a model should be to make predictions in situations where it hasn't before.

Bad job generalizing: no improvement and hockey-stick like validation accuracy while loss is "decreasing"

overfitting / overtraining

test cost ↑, while train cost ↓

Always regard the classification accuracy over the cost measure for analysis

"Memorizing training set"

Stop training at max validation accuracy

Also can use validation data for hyperparameter tuning

=> If we use test data for hyperparameter tuning, we may overfit our hyperparameters

Use test dataset as a measure of network's generalization

Held out method:

Reduce overfitting: Increase the size of training data.