

Week 2

m training example \rightarrow process without explicit for loop.

after forward propagation \rightarrow backward propagation

binary classification - cat or non-cat

$$x \rightarrow y$$

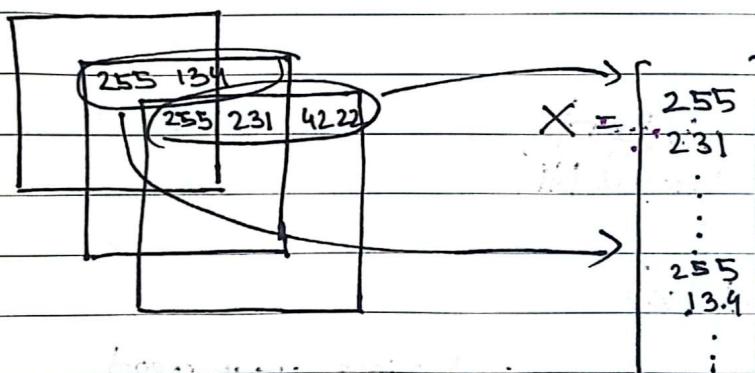
64x64 image

rgb \rightarrow all have separate 64x64 matrix



pixel intensity values

unroll pixel intensity value into feature vector



total dimension:

$$64 \times 64 \times 3 = 12288$$

$$n \cdot x = 12288$$

Notation

$$(x, y) \quad x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$$



single training example

training set comprises of m training examples

$(x^{(1)}, y^{(1)})$ \rightarrow input & output for first training example

entire training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \dots\}$

$$m = M_{\text{train}} \quad m_{\text{test}} = \# \text{ test examples}$$

\downarrow
no. of training examples

compact notation of entire training set:

$$X = \begin{bmatrix} | & | & | & | \\ X^{(1)} & X^{(2)} & X^{(3)} & \dots & X^{(m)} \\ | & | & | & | \end{bmatrix}$$

$\xleftarrow[m \text{ cols}]{} \quad \xrightarrow[n_x \text{ rows}]{} \quad \xrightarrow[X \in \mathbb{R}^{n_x \times m}]{} \quad \xleftarrow[\text{Stack in columns}]{} \quad$

~~$X = [X^{(1)} \ X^{(2)} \ X^{(3)} \dots \ X^{(m)}]$~~ wrong
use this
notation

$x.\text{shape} = (n_x, m)$

$$Y = [y^{(1)} \ y^{(2)} \ y^{(3)} \dots \ y^{(m)}]_{1 \times m}$$

$$Y.\text{shape} = (1, m)$$

viewed
Logistic Regression \rightarrow Small NN

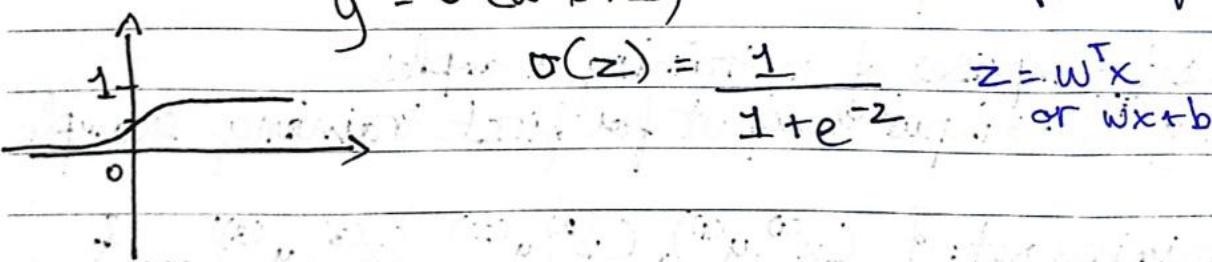
given x , want $\hat{y} = P(y=1|x)$

$$x \in \mathbb{R}^{n_x}$$

parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$ bad since you need output b/w 0 & 1

output $\hat{y} = w^T x + b$ not good for classification

$\hat{y} = \sigma(w^T x + b)$ so we use sigmoid function



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

if z is very large: $\sigma(z) \approx \frac{1}{1+0} = 1$ (close to 1)

if z is very small/very large -ve number:

$$\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{1+\text{big no.}} \approx 0 \text{ (close to 0)}$$

job in logistic regression is to learn parameters w & b , so that
 y becomes a good estimate of the chance of y being
 equal to 1.

Keep w & b parameters separate:

different notation: → not going to be used in this course

- define extra feature $x_0 = 1$, so $x \in \mathbb{R}^{n+1}$
- $\hat{y} = \sigma(w^T x)$

$$w = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$$

⇒ for implementing neural networks, it is best to keep
 w & b separate, so don't use the x_0 notation for this
 course.

Cost function - Logistic Regression

$$\hat{y} = \sigma(w^T x + b), \text{ where } \sigma(z) = \frac{1}{1+e^{-z}}$$

Given $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$,
want $\hat{y}^{(i)} \approx y^{(i)}$

m training example

$$z^{(i)} = w^T x^{(i)} + b \quad \left. \begin{array}{l} x^{(i)} \\ y^{(i)} \\ \hat{y}^{(i)} \end{array} \right\} \begin{array}{l} \text{denotes data associated} \\ \text{with } i\text{th training example} \end{array}$$

Loss (error) function:

$$L(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2 \quad \times \text{we don't do this in logistic regression, as optimization problem becomes non-convex.}$$

Loss function:

$$L(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y})) \quad \left. \begin{array}{l} \text{local-optima problem} \\ \text{convex} \end{array} \right.$$

$$\text{if } y=1: L(\hat{y}, y) = -\log \hat{y} \quad \text{want } \hat{y} \text{ large or } =1, \text{ as it can't be bigger than } 1$$

$$\text{if } y=0: L(\hat{y}, y) = -\log(1-\hat{y}) \quad \text{want } \log(1-\hat{y}) \dots \text{want } \hat{y} \text{ small}$$

Cost function:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})]$$

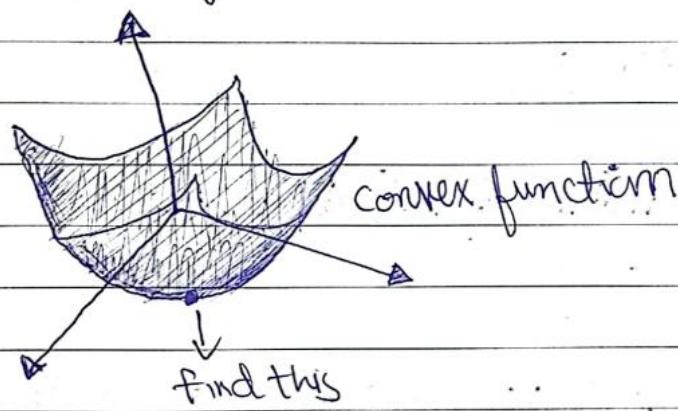
Loss function → applied to single training example

cost function → cost of your parameter (entire training set)

cost function \rightarrow avg of the loss function of the entire training set

Gradient Descent

\rightarrow want to find w, b that minimizes $J(w, b)$



slope of a function at a point

$$w = w - \alpha \frac{dJ(w)}{dw}$$

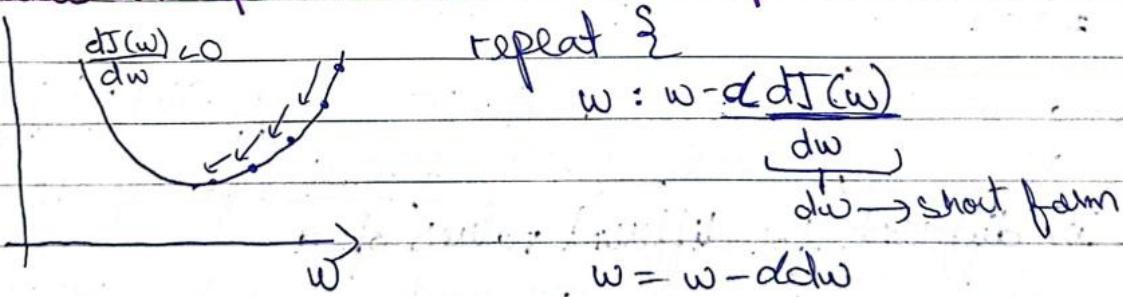
learning rate α
 $dJ(w)$ variable name
 for derivative term in code

$$w = w - \alpha dw$$

derivative is +ve
 so you decrease

derivative is -ve
 so you add
 $-(-dw)$
 $+dw$

GD takes a step downward in the steepest descent.



$$\underline{dJ(w)} = ?$$

$\frac{dw}{}$
slope

J(w, b)

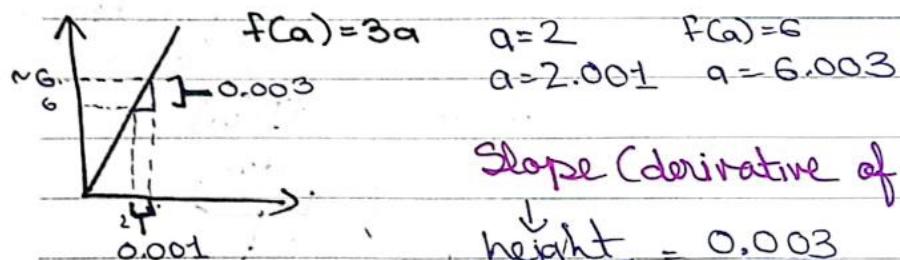
$$w := w - \alpha \frac{dJ(w, b)}{dw} \rightarrow \underline{\frac{\partial J(w, b)}{\partial w}} \rightarrow \boxed{dw}$$

$$b := b - \alpha \frac{dJ(w, b)}{db} \rightarrow \underline{\frac{\partial J(w, b)}{\partial b}} \rightarrow \boxed{db}$$

in code: small $d : \partial$

partial derivative:

when J is a function of 2 or more variables, we use partial derivative.



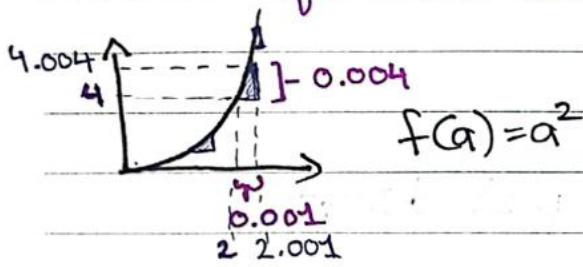
Slope (derivative of $f(a)$) at $a=2$ is 3

$$\frac{\text{height}}{\text{width}} = \frac{0.003}{0.001}$$

$$\begin{array}{lll} a=5 & f(a)=15 & \Rightarrow \text{slope at } a=5 \text{ is also 3} \\ a=5.001 & f(a)=15.003 & \frac{d f(a)}{da} = 3 = \frac{d f(a)}{da} \end{array}$$

on a straight line:

the function's derivative doesn't change

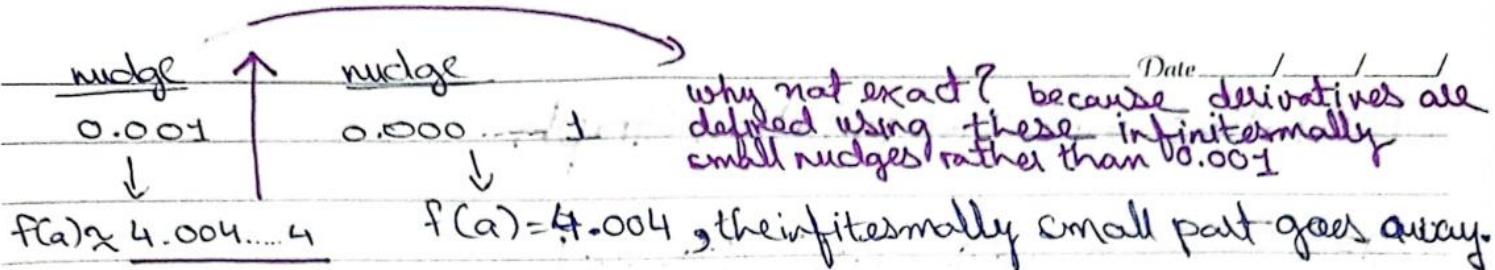


$$\begin{array}{lll} a=2 & f(a)=4 & \text{slope (derivative) of } f(a) \text{ at } a=2 \text{ is 4.} \\ a=2.001 & f(a) \approx 4.004 & \therefore \frac{d f(a)}{da} = 4 \text{ when } \underline{a=2} \end{array}$$

$$\therefore \boxed{\frac{d f(a)}{da} = \frac{d}{da} a^2 = 2a}$$

$$\begin{array}{lll} a=5 & f(a)=25 & \frac{d f(a)}{da} = 10 \text{ when} \\ a=5.001 & f(a)=25.010 & \underline{a=5} \end{array}$$

→ Slope is different, for different values of a
for any given value of a if you nudge it up by 0.001,
 $f(a)$ goes up by $\downarrow \text{derivative} \times \downarrow \text{nudge}$



$$f(a) = a^2 \quad \frac{d}{da} f(a) = 2a$$

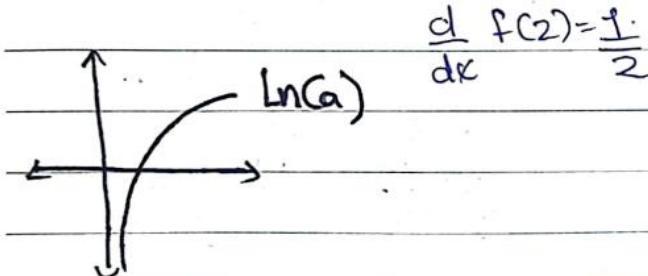
$a=2 \quad f(a)=4$
 $a=2.001 \quad f(a) \approx 4.004$

$$f(a) = a^3 \quad \frac{d}{da} f(a) = 3a^2$$

$a=2 \quad f(a)=8$
 $a=2.001 \quad f(a) \approx 8.012$

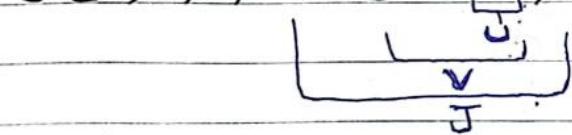
$$f(a) = \log_e(a) \quad \frac{d}{da} f(a) = \frac{1}{a}$$

$a=2 \quad f(a)=0.69315$
 $a=2.001 \quad f(a) \approx 0.69365$



derivative is just a slope of line.

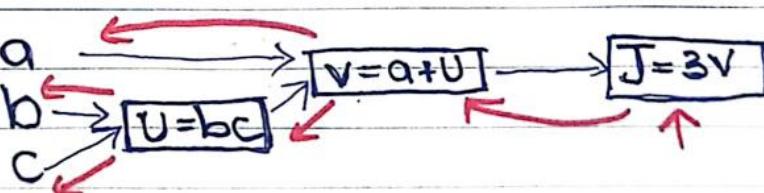
Computation Graph → organizes computation with blue arrow, left to right computation.



simpler graph:

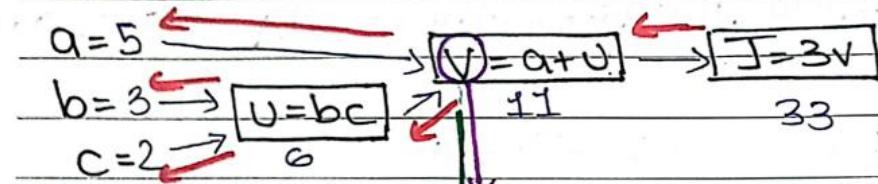
3 distinct steps:

- 1) $u = bc$
- 2) $v = a + u$
- 3) $J = 3v$



for computing derivative, go in opposite direction

Derivatives with Computation Graph



$$\frac{dJ}{dv} = ? \quad 3$$

$$f(a) = 3a$$

$$\frac{df(a)}{da} = \frac{df}{da} = 3$$

$$J = 3v$$

$$\frac{dJ}{dv} = 3$$

if we change value v, how does it change J?

$$J = 3v$$

$$v = 11 \rightarrow 11.001$$

$$J = 33 \rightarrow 33.003$$

→ to compute derivative of final output variable wrt. to v, then we have done one step of backward propagation.

$$\frac{dJ}{da} = ? \quad 3$$

$$\begin{aligned} a &= 5 \rightarrow 5.001 \\ v &= 11 \rightarrow 11.001 \\ J &= 33 \rightarrow 33.003 \end{aligned}$$

change a → changes

chain rule

$$\frac{dJ}{da} = \frac{dv}{da} \cdot \frac{dJ}{dv}$$

$$\downarrow \quad \downarrow$$

$$1 \quad 3$$

$$1 \times 3 \quad 1 \times 3$$

final output variable (you really care about)
 $\rightarrow J$ in this case

will compute derivative with J of intermediate variables like a, b, c so name the final variable (J) like `dFinalOutputVar`, `dJVar` in code.
`'dVar'`

$$\frac{dJ}{du} = 3 = \frac{dJ}{dv} \cdot \frac{dv}{du}$$

$\underbrace{}_3 \quad \underbrace{}_1$

$$v=6 \rightarrow 6.00 \pm$$

$$v=11 \rightarrow 11.00 \pm$$

$$J=33 \rightarrow 33.00 \underline{3}$$

$$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db}$$

$\underbrace{}_3 \quad \underbrace{}_2$

$$b=3 \rightarrow 3.00 \pm$$

$$u=bc \rightarrow 6 \rightarrow 6.00 \pm$$

$$c=2$$

$$\frac{dJ}{dc} = \frac{dJ}{du} \cdot \frac{du}{dc}$$

$\underbrace{}_3 \quad \underbrace{}_3$

$$= 6$$

* when computing derivatives, the most efficient way to do so, is a right to left computation, following the direction of arrow.

Gradient Descent for Logistic Regression

$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z)$$

$$L(a, y) = - (y \log(a) + (1-y) \log(1-a))$$

Date / /

$$\begin{aligned}
 & x_1, x_2, w_1, w_2, b \\
 & \xrightarrow{\quad z = w_1 x_1 + w_2 x_2 + b \quad} \xrightarrow{\hat{y} = a = \sigma(z)} \xrightarrow{\mathcal{L}(a, y)} \\
 & dz = \frac{d\mathcal{L}}{dz} \\
 & = \frac{d\mathcal{L}(a, y)}{da} \\
 & = \frac{-y}{a} + \frac{1-y}{1-a} \\
 & = \frac{dy}{da} \cdot \frac{da}{dz} \rightarrow a(1-a)
 \end{aligned}$$

compute how much you need to change the parameters.

Simplified to

$$w_1 = w_1 - \alpha \frac{d\mathcal{L}}{dw_1}$$

$$w_2 = w_2 - \alpha \frac{d\mathcal{L}}{dw_2}$$

$$b = b - \alpha \frac{d\mathcal{L}}{db}$$

Gradient Descent on m Examples

cost function: $J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(a^{(i)}, y^{(i)})$

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

$$(x^{(i)}, y^{(i)})$$

$$\underbrace{dw_1^{(i)}, dw_2^{(i)}, db^{(i)}}_{\text{notation for one training example}}$$

$$\frac{\partial}{\partial w_1} J(w, b) = \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$$

$$dw_1^{(i)} \rightarrow (x^{(i)}, y^{(i)})$$

implements 1 step of gradient descent,
so you have to do multiple times:

Date _____ / _____ / _____

$J=0; dw_1=0; dw_2=0; db=0$ // initialization

for $i=1$ to m → 2nd for loop

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

$$a^{(i)} = \sigma(z^{(i)})$$

$$J += [y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log (1-a^{(i)})]$$

$$dz^{(i)} = a^{(i)} - y^{(i)}$$

$$dw_1 += x_1^{(i)} dz^{(i)} \uparrow \text{assuming } n=2$$

$$dw_2 += x_2^{(i)} dz^{(i)}$$

$$db += dz^{(i)}$$

$$J/m$$

$$dw_1/m; dw_2/m; db/m$$

used as accumulators, so they
do not have superscript

$$dw_1 = \frac{\partial J}{\partial w_1}$$

$$w_1 = w_1 - \alpha dw_1$$

// to implement 1

$$w_2 = w_2 - \alpha dw_2$$

step of GD, you do,

$$b = b - \alpha db$$

2 weaknesses here:

- 1) you need to write 2 for loops, 1 loop over the features
- 2) having explicit for loops in your code for deep learning algorithms, makes your algorithm run less efficiently.

Solution? → **Vectorization Techniques** allows you to get rid of explicit for loops.

$$Z = w^T x + b \quad w = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}; x = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix} \quad w \in \mathbb{R}^{n \times 1} \quad x \in \mathbb{R}^{1 \times n}$$

non vectorized:

$$z=0$$

for i in range(n-x):

$$z += w[i] * x[i]$$

$$z += b$$

(vectorized):

$$Z = np.dot(w, x) + b$$

$$w^T x$$

MINDFUL NOTES

say you can/need to apply the exponential operation on every element of a matrix

$$V = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \quad U = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

import numpy as np
 $U = np.exp(V)$

vectorized
 short code
 for applying
 exp. op on
 every element

$U = np.zeros((n, 1))$

for i in range(n):

$U[i] = \text{math.exp}(V[i])$

$np.log(V)$

$np.abs(V)$

$np.maximum(V, 0) \rightarrow \text{element wise max}$

$V * V^2 \rightarrow \text{square max}$

$1/V \rightarrow \text{inv}$

other useful functions

Vectorizing Logistic Regression's Gradient output

Vectorizing Logistic Regression

$$Z^{(1)} = w^T x^{(1)} + b$$

$$a^{(1)} = \sigma(Z^{(1)})$$

$$Z^{(2)} = w^T x^{(2)} + b$$

$$\sigma^{(2)} = \sigma(Z^{(2)})$$

$$Z^{(m)} = w^T x^{(m)} + b$$

$$\sigma^{(m)} = \sigma(Z^{(m)})$$

how to do it without explicit for loop?

$$X = \begin{bmatrix} | & | & | & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & | & | \end{bmatrix} (n_x, m) \text{ meaning }$$

$$\underbrace{w^T X}_{\mathbb{R}^{n_x \times m}} \begin{bmatrix} |^{(1)} & |^{(2)} & |^{(m)} \\ x^{(1)} & x^{(2)} & x^{(m)} \\ | & | & | \end{bmatrix} \text{ row vector}$$

$$Z = [Z^{(1)} \ Z^{(2)} \ \dots \ Z^{(m)}] = w^T X + [b \ b \ \dots \ b]_{1 \times m}$$

$$= [w^T x^{(1)} + b \ w^T x^{(2)} + b \ \dots \ w^T x^{(m)} + b]_{1 \times m}$$

$$Z = np.dot(w.T, X) + b$$

$\nwarrow (1, 1)$ real no. here,

$$A = [a^{(1)} \ a^{(2)} \ \dots \ a^{(m)}] = \sigma(Z)$$

$a^{(1)}, a^{(2)}, \dots, a^{(m)}$ for b, g, \dots

python automatically converts it to $(1, m)$ row vector. (Broadcasting)

MINDFUL NOTES

Vectorizing Logistic Regression's Gradient Output

$$dz^{(1)} = a^{(1)} - y^{(1)} \quad dz^{(2)} = a^{(2)} - y^{(2)} \quad \dots$$

$$dZ = [dz^{(1)} \ dz^{(2)} \ \dots \ dz^{(m)}] \quad \text{A } m \times m \text{ or } m \text{ dimensional row vector}$$

$$A = [a^{(1)} \ \dots \ a^{(m)}], \quad Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(m)}]$$

$$dZ = A - Y = [a^{(1)} - y^{(1)} \ a^{(2)} - y^{(2)} \ \dots]$$

We now have to work to remove the second for loop

$$\begin{aligned} dw &= 0 \\ dw + &= x^{(1)} dz^{(1)} \\ dw + &= x^{(2)} dz^{(2)} \\ dw / &= m \end{aligned} \quad \text{for loop} \quad \left| \begin{array}{l} db = 0 \\ db + dz^{(1)} \\ \vdots \\ db + dz^{(m)} \\ db / m \end{array} \right. \quad \text{for loop}$$

Vectorized sol:

$$db = \frac{1}{m} \text{np.sum}(dZ) \quad // = \frac{1}{m} \sum_{i=1}^m dz^{(i)}$$

$$dw = \frac{1}{m} X \cdot dZ^T \quad // = \frac{1}{m} \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ dz^{(2)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

single iteration of gradient

descent for logistic regression // $\frac{1}{m} \left[x^{(1)} dz^{(1)} + \dots + x^{(m)} dz^{(m)} \right]$
 $\therefore (Z = w^T X + b)$ // not code, just explanation
 $Z = \text{np.dot}(w.T, x) + b$

$$A = \sigma(Z)$$

$$dZ = A - Y$$

$$dw = \frac{1}{m} X \cdot dZ^T$$

$$db = \frac{1}{m} \text{np.sum}(dZ)$$

$$\begin{aligned} w &= w - \alpha dw \\ b &= b - \alpha db \end{aligned}$$

// for multiple iterations, you still need a for loop over the no. of iterations.

Broadcasting in Python

e.g.: Calories from Carbs, Proteins, Fats in 100g of different foods.

	Apples	Beef	Eggs	Potatoes	
carb	56.0	0.0	4.4	68.0	$= A_{3 \times 4}$
protein	1.2	104.0	52.0	8.0	\downarrow
Fats	1.8	135.0	99.0	0.9	
	59 cal				

$$\frac{56}{59} \times 94.9\%$$

Q. Calculate % of calories from carbs, proteins, fats.

Sum each of the columns, then divide throughout the matrix to get % of calories from carbs, proteins, fats
 Can you do this without an explicit for loop?

Code:

```
import numpy as np
A = np.array([ [56.0, 0.0, 4.4, 68.0], ... ])
print(A)
cal = A.sum(axis=0)
print(cal)
percentage = 100 * A / cal.reshape(1,4)
print(percentage)
```

broadcasting 1×4
 (redundant here although but takes $O(1)$.)

eg1: $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100$ add in python,
it will auto-expand = $\begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$

works for
parameters,

eg: b (constant)

⇒ this type of broadcasting works for both row vector & column vector

ex2: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{m \times n} + [100 \ 200 \ 300]$ python turns to = $\begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$

$(1, n) \rightsquigarrow (m, n)$

ex3: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{m \times n} + \begin{bmatrix} 100 \\ 200 \end{bmatrix}_{(m \times 1)}$ in python? = $\begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}$

$(m, 1) \rightsquigarrow (m, n)$

General Principle

<u>operand</u>	<u>op</u>	<u>operand</u>
1) (m, n) matrix	+	$(1, n) \rightsquigarrow (m, n) = (m, n)$
2) (m, n) matrix	*	$(m, 1) \rightsquigarrow (m, n) = (m, n)$
3) $(m, 1)$	+	IR

eg: $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + 100 = \begin{bmatrix} 101 \\ 102 \\ 103 \end{bmatrix}$

$$[1 \ 2 \ 3] + 100 = [101 \ 102 \ 103]$$

* read documentation of numpy for fully general view of broadcasting in python, more of what it

Tips & Tricks

The ability of python to allow you to use broadcasting operations & more generally, the greater flexibility of python numpy programming language, is both a weakness & a strength

introduces a lot
of bugs with this
amount of flexibility

↓
can get a lot of work done
with a single line of code

e.g.: column + row
vector + vector,
can throw a dimension
mismatch or type error
something, or you may
get a matrix or a sum
of a row & a column vector

(there is an internal logic
to these strange effects
of python)

Couple of Tips & Tricks to eliminate & simplify all
of the strange looking bugs:

```
import numpy as np # funny data structure
a = np.random.randn(5) # five random gaussian variables
print(a.shape) # (5,) stored in Array A.
print(a.T)
# a is a transpose (rank 1 array,
# look the same here (nor row vector, nor column vector)
print(np.dot(a, a.T)) # ax^T = get a number; when
# you shouldn't (9.06571)
```

key takeaway:
don't use rank 1 array as they cause bugs.

Sol:

```
a = np.random.rand(5,1) // returns column vector
print(a.T) // now it looks diff. from a
print(np.dot(a,a.T)) // [[...]] gives a 1x5 matrix
now
```

don't use rank 1 array,

so make it a column or row vector

```
a = np.random.rand(5,1) // a.shape = 5,1 col vector
a = np.random.rand(1,5) // a.shape = 1,5 row vector
(So behaviour of vectors are understandable)
assert(a.shape == (5,1)) // serves as documentation
as well
```

Explanation of Logistic Regression Cost Function

$$\hat{y} = \sigma(\omega^T x + b) \text{ where } \sigma(z) = \frac{1}{1+e^{-z}}$$

Interpret $\hat{y} = P(y=1|x)$

$$\left. \begin{array}{l} \text{if } y=1 : P(y|x) = \hat{y} \\ \text{if } y=0 : P(y|x) = 1 - \hat{y} \end{array} \right\} P(y|x)$$

$$P(y|x) = \hat{y}^{y} (1-\hat{y})^{(1-y)}$$

$$\text{if } y=1 : P(y|x) = \hat{y}^1 \frac{(1-\hat{y})^0}{(1-\hat{y})^0} = 1 = \hat{y}$$

$$\text{if } y=0 : P(y|x) = \frac{\hat{y}^0}{1} \frac{(1-\hat{y})^1}{(1-\hat{y})^1} = 1 - \hat{y}$$

$$\begin{aligned}\log p(y|x) &= \log \hat{y}^T y (1-\hat{y})^{1-y} \\ &= y \log \hat{y} + (1-y) \log (1-\hat{y}) \\ &= -\hat{\ell}(\hat{y}, y)\end{aligned}$$

minimize the loss function
corresponds to maximizing the log
of probabilities

Cost on m examples:

$$\log P(\text{Labels in training set}) = \prod_{i=1}^m p(y^{(i)}|x^{(i)})$$

$$\begin{aligned}\log P(\cdot) &= \sum_{i=1}^m \log p(y^{(i)}|x^{(i)}) \quad \text{maximum likelihood estimation} \\ &= -\sum_{i=1}^m \hat{\ell}(y^{(i)}, \hat{y}^{(i)})\end{aligned}$$

$$\text{Cost: } J(w, b) = \frac{1}{m} \sum_{i=1}^m \hat{\ell}(y^{(i)}, \hat{y}^{(i)})$$