m ⇒ No. of training examples

* sigmoid to ReLU

Subject * structured & unstructured data. Date

WEEK 2: Binary classification

Training set of 'm' training examples, process entire training set without using for loop.

① Forward propagation

② Backward propagation step/ Backward pass

LOGISTIC REGRESSION

- Algorithm for binary classification (Input (x) → output (y) Y/N)
- Images: 3 separate matrices (R G B)
- Input: Feature vector x

$$x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \\ 255 \\ 134 \\ 93 \\ \vdots \end{bmatrix} \begin{matrix} \} R \\ \\ \\ \} G \\ \\ \\ \} B \\ \\ \end{matrix}$$

... Pixel values

eg:

R, G, B ⇒ 64 × 64

x ⇒ 64 × 64 × 3

= 12288

(Total no. of pixels)

- $n = n_x$ = Dimension of input feature vector

input feature vector x → Binary class. → output 0 or 1

→ NOTATION

1) (x, y)
   - $x \in \mathbb{R}^{n_x}$ (n_x dimensional feature vector)
   - $y \hookrightarrow 0$ or 1

2) "m" training examples → m_{train}
   - $(x_1, y_1), (x_2, y_2) \ldots (x_m, y_m)$
   - m_{test} = no. of test examples

3) X = x_1 x_2 x_3 ...

$$X = \begin{bmatrix} | & | & | & & | \\ x^{1} & x^{2} & x^{3} & \cdots & x^{m} \\ | & | & | & & | \end{bmatrix} \quad n_x \uparrow \quad \leftarrow m \rightarrow$$

$X = \mathbb{R}^{n_x \times m}$

Python: X.shape = (n_x, m)

(n_x, m) dimensional matrix

rows ↓ columns →

4) $Y = [y^{(1)}\ y^{(2)} \dots y^{(m)}] : Y \in \mathbb{R}^{1 \times m}$

Y.shape = (1, m)

row ← → column

Subject _____ Date _____ → (x or y)

IMP: Take data associated with different training examples and stack them in different columns.

LOGISTIC REGRESSION

- used for binary classification problems (output label y = 0/1)
- Given x, want $\hat{y} = P(y=1|x)$

  $x \in \mathbb{R}^{n_x}$ $\quad 0 \le \hat{y} \le 1$
- Logistic Regression:

  $w \in \mathbb{R}^{n_x}$ → vector (n-dimension)

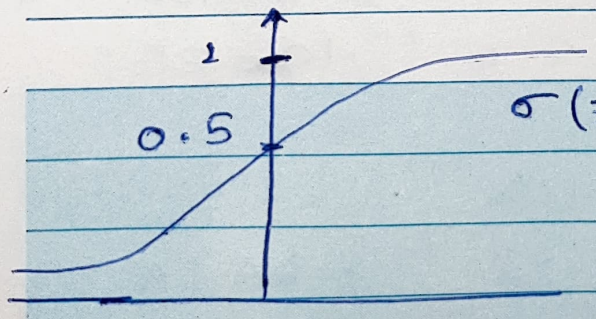
  b → real number
- output: $\boxed{\hat{y} = w^T x + b}$

  ↳ Not good for bin. class.

  $w^T x + b$ can be -ve or >>> 1

→ Actually: $\boxed{\hat{y} = \sigma(w^T x + b)}$

z



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

if z is large

$\sigma(z) \approx \frac{1}{1+0} = 1$

if z is small or a very large -ve no,

$\sigma(z) = \frac{1}{1+e^{-z}} \approx 0$

Alternative: $x_0 = 1$ & $x \in \mathbb{R}^{n_x+1}$

$\hat{y} = \sigma(\theta^T x)$

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$ $\left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\}$ $\begin{matrix} b \\ \\ \\ \\ \end{matrix}$ $\left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\}$ $\begin{matrix} \\ \\ \\ \\ w \end{matrix}$

significance of 'b' & 'w'

NOTE: parameters of logistic regression → "w" [n_x = Dimensional vector]
"b" → a real number

COST FUNCTION:

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

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

Training set: m parameters

$\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

output expected → $\hat{y}^{(i)} \approx y^{(i)}$

Training set ↵ Ground truth label

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

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

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

$x^{(i)}$
$y^{(i)}$
$z^{(i)}$ } DATA

data associated with i^{th} example (superscript (i) in parenthesis)

Loss (error) function

$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ → as small as possible

Subject _____ Date

How good output $\hat{y}$ is when true label is y

→ squared error doesn't make gradient descent work well

→ Loss function in Logistic Regression makes optimisation easy ⇒ $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$ as small as possible

eg: If $y=1$; $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$

want $\log \hat{y}$ to be >>> & $-\log \hat{y}$ <<<<

⇒ want $\hat{y}$ >>>> (close to 1 because $0 \leq \hat{y} \leq 1$)

If $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$

For loss function to be small, $-\log(1-\hat{y})$ ⇒ <<<<

$\therefore \log(1-\hat{y})$ ⇒ >>>>

$(1-\hat{y})$ ⇒ >>>>

⇒ $\hat{y}$ <<<< (as small as possible)

Loss fx? ⇒ single training ex.

cost fx? ⇒ entire training set

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

$$= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})\right]$$

In training: find w & b that minimize J

NOTE: Logistic Regression is a v.v. small neural network

GRADIENT DESCENT

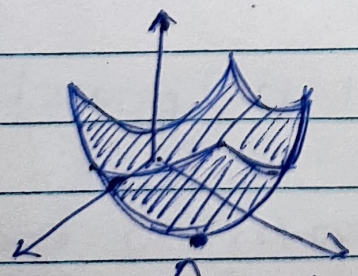
$$\hat{y} = \sigma(w^T x + b)$$

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

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

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

Find w, b to minimise cost function $J(w,b) \Rightarrow$ convex fx?

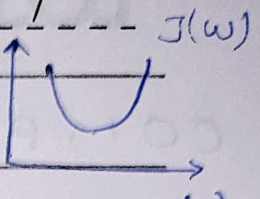


Least value of $J(w,b)$

① Initialize w & b

② Gradient descent start at pt 1 & move downwards on the steepest path by 1 step.

~~NEURAL NETWORK~~



- Gradient descent

Repeatedly update 'w'

Repeat {

$w := w - \alpha \frac{dJ(w)}{dw}$ → Learning rate

updating w — derivative, update/change (dw)

Learning rate controlls how big of a step is taken on each iteration of gradient descent

CODE: $w := w - \alpha dw$

For $J(w, b)$

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

$b := b - \alpha \frac{dJ(w,b)}{\partial b}$

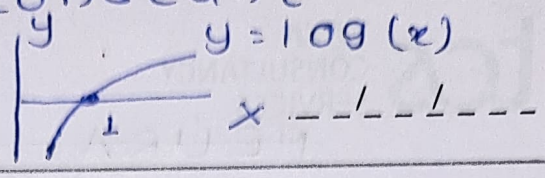
$b := b - \alpha db$ & $w := w - \alpha dw$

⇒ convex fx? only has 1 optima.

CALCULUS & DERIVATIVES:

① st. line ⇒ slope = constant

Log:



COMPUTATION GRAPH:

- To explain forward & back propagation

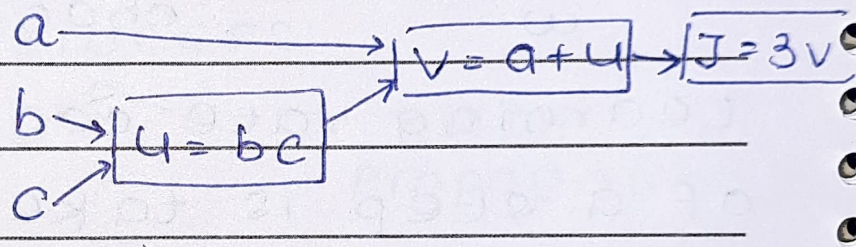
→ Left to right computation (forward pass)

eg: $J(a, b, c) = 3(a + bc)$

$u = bc$

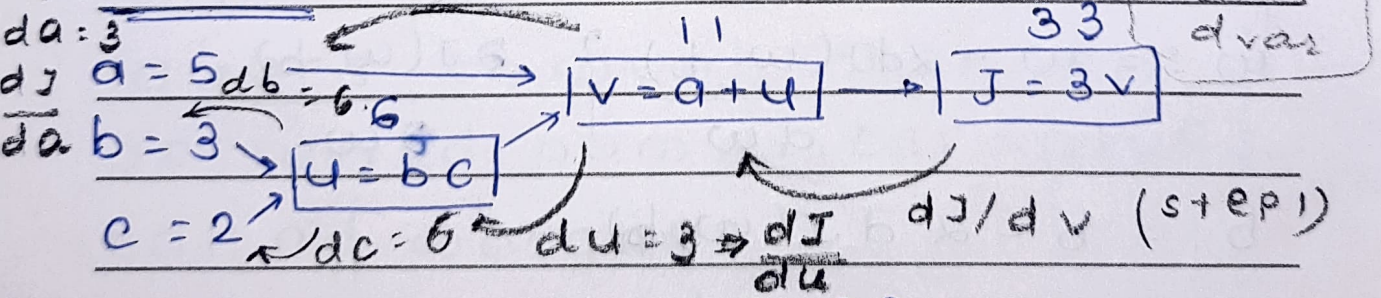
$v = a + u$

$J = 3v$



NOTE: "Backward" propagation yields final output variable derivatives

DERIVATIVES WITH COMPUTATION GRAPH:



$J = 3v$ & $v = 11$ & $J = 33$

$\therefore \frac{dJ}{dv} = 3 \qquad \frac{dJ}{da} = 3v = 3(a + u) = 3$

or $\frac{dJ}{da} = \frac{dJ}{dv} \times \frac{dv}{da} = 3 \times 1 = 3$

CODE: $\frac{d\,\text{output final var}}{d\,\text{var}} = dJ_{var} = \boxed{d\text{var}}$

TATA CONSULTANCY SERVICES

Forward ⇒ cost fx? (minimize)

Backward ⇒ derivatives

GRADIENT DESCENT FOR LOGISTIC REGRESSION:

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

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

$\mathcal{L}(a, y) = -(y \log(a) + (1-y)\log(1-a)) \Rightarrow \boxed{a = \hat{y}} \Rightarrow$ Loss fx?

computation graph:

x_1, w_1, x_2, w_2, b → $\boxed{z = w_1 x_1 + w_2 x_2 + b}$ → $\boxed{a = \sigma(z)}$ → $\boxed{\mathcal{L}(a, y)}$

$dw_1 = \frac{d\mathcal{L}}{dw_1} = x_1 dz$; $dw_2 = \frac{d\mathcal{L}}{dw_2} = x_2 dz$; $db = dz$

$\frac{d\mathcal{L}}{da} = \frac{-y}{a} + \frac{1-y}{1-a}$

$\frac{da}{dz} = a(1-a)$

$dz = \frac{d\mathcal{L}(a,y)}{dz} = \frac{d\mathcal{L}}{dz}$ $\quad da = \frac{d\mathcal{L}(a,y)}{da}$

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

$dz = \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$

updates → $w_1 := w_1 - \alpha\, dw_1$ $\quad b := b - \alpha\, db$

$w_2 := w_2 - \alpha\, dw_2$

LOGISTIC REGRESSION ON m EXAMPLES

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

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

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

Initialize: $J = 0$, $dw_1 = 0$, $dw_2 = 0$, $db = 0$ $\quad dw = np.zeros((n_x, 1))$

for $i = 1$ to m: For loop #1

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

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

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

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

$dw_1 += x_1^{(i)} dz^{(i)}$

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

for loop #2 → for $j = 1$ to n_x: $dw_j += \ldots$; $n = 2$

$db += dz^{(i)}$ $\quad dw += x^{(i)} dz^{(i)}$

2 for loops

① for loop over 'm' training examples

② for loop over features

$J /= m$ $\quad J = J/m$

$dw_1 /= m$ $\quad dw_1 = dw_1/m$

$dw_2 /= m$ $\quad dw_2 = dw_2/m$

$\Rightarrow dw /= m$ or $dw = \frac{dw}{m}$

$db /= m$ $\quad db = b/m$

↓

accumulators

(sum over entire training set)

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

UPDATES

* vectorization helps us get rid of for loops

$\sigma(z) = a$ ← activation

↑

sigmoid activation function

*DERIVATIVES:

$\frac{da}{dz} = \frac{d}{dz}\sigma(z) = \sigma(z) \times (1 - \sigma(z)) = a(1-a)$; $\frac{dL}{da} = \frac{a-y}{a(1-a)} \Rightarrow \frac{dL}{dz} = a - y$

DERIVATIVE OF SIGMOID.

° VECTORIZATION : Getting rid of for loops

$z = w^T x + b$

$w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $\quad x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$

$w = \mathbb{R}^{n_x}$ $\quad x = \mathbb{R}^{n_x}$

vectorized

$z = \underbrace{np.dot(w, x)}_{w^T x} + b$

non-vectorized

```
z = 0
for i = 1 to (n-x)
(i in range (n-x)):
    z += w[i] * x[i]
z += b
```

⇒ VECTORISED CODE RUNS MUCH FASTER

GPU, CPU } SIMD

single instruction & multiple data stream (PARALLELISM)

• VECTORIZATION

$u = A \sqrt{v}$ ← matrix

↑ vector

$u_i = \sum_j A_{ij} v_j$

```
u = np.zeros((10,1))
for i ....
    for j ....
u[i] += A[i][j] * v[j]
```

Alternative

```
u = np.dot(A,v)
```

* np.log(v) ← logarithm
* np.abs(v) ← absolute
* np.maximum(v,0) ← max.
* v**2 ← square
* $\frac{1}{v}$ ← inverse

ex: $v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$ to have $u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$

```
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(v[i])
```

Alternative

```
import numpy as np
u = np.exp(v)
```

• VECTORIZING LOGISTIC REGRESSION:

$z^{(1)} = w^T x^{(1)} + b$ & $a^{(1)} = \sigma(z^{(1)})$ ... $z^{(m)} = w^T x^{(m)} + b$ & $a^{(m)} = \sigma(z^{(1)})$

ACTIVATION

$$X = \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(n)} \\ | & | & | & & | \end{bmatrix} \Rightarrow (n_x, m) \text{ matrix}$$

$\mathbb{R}^{n_x \times m}$

$$[z^{(1)}\ z^{(2)}\ z^{(3)} \dots z^{(m)}] = w^T X + [b\ b \dots b]$$

$(1 \times m)$

$$= [\underbrace{w^T x^{(1)} + b}_{z_1} \quad \underbrace{w^T x^{(2)} + b}_{z_2} \dots \underbrace{w^T x^{(n)} + b}_{z^n}]$$

$1 \times m$

command: `z = np.dot(w.T, x) + b` ← real no. (1x1 matrix)

BROADCASTING ↳ to calculate all 'z' at same time.

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

• VECTORIZING LOGISTIC REGRESSION:

$dz^{(1)} = a^{(1)} - y^{(1)}$ $dz^{(2)} = a^{(2)} - y^{(2)}$ ... $dz^{(m)} = a^{(m)} - y^{(m)}$

$\therefore dz = [dz^{(1)}\ dz^{(2)} \dots dz^{(m)}]$

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

$\therefore dz = A - Y = [a^{(1)} - y^{(1)} \dots a^{(m)} - y^{(m)}]$

```
dw = 0
dw += x(1) dz(1)
dw += x(2) dz(2)
   :
dw += x(m) dz(m)
dw = dw/m
⇒ dw /= m
```

```
db = 0
db += dz(1)
db += dz(2)
   :
db += dz(m)
db = db/m
⇒ db /= m
```

$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$

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

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

$dw = \frac{1}{m} \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$

$dw = \frac{1}{m} [x^{(1)} dz^{(1)} + \dots + x^{(m)} dz^{(m)}]$

• IMPLEMENTING LOGISTIC REGRESSION:

Traditional approach:

- $J = 0$; $dw_1 = 0$, $dw_2 = 0$, $db = 0$
- for $i = 1$ to m:
  - $z^{(i)} = w^T x^{(i)} + b$
  - $a^{(i)} = \sigma(z^{(i)})$
  - $J += -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log (1 - a^{(i)})]$
  - 100p
    - $dz^{(i)} = a^{(i)} - y^{(i)}$
    - $dw_1 += x_1^{(i)} dz^{(i)}$
    - $dw_2 += x_2^{(i)} dz^{(1)}$
    - $db += dz^{(i)}$
- $J = J/m$, $dw_1 = dw_1/m$, $dw_2 = dw_2/m$, $db = db/m$

code implementation:

$z = w^T X + b$

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

$A = \sigma(z)$

$dz = A - Y$

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

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

$w := w - \alpha dw$

$b := b - \alpha db$

- **BROADCASTING:-**

axis = 0 → vertical

axis = 1 → Horizontal

cal = A.sum(axis = 0)

percentage = 100 * A / (cal.reshape(1, 4))

A → (3, 4) & (1, 4) vector divisor

eg. of broadcasting → (it is dimensions)

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

⊛ (m, n) matrix

$+, -$ → (1, n) ⇝ (m, n)

$*, /$ → (m, 1) ⇝ (m, n)

- JUPYTER
  - Rank 1 array ⇒ 1 square bracket
  - vector ⇒ 2 square bracket

```
import numpy as np
a = np.random.randn(5)   → creates 5 random Gaussian variables stored in array a
print(a)
print(a.shape) ⇒ (5,)    ⇒ Rank 1 vector (Neither row nor column)
print(a.T)   → Transpose
print(np.dot(a, a.T))   → a multiplied by a^T, gives a number
```

NOT EFFICIENT (Rank 1 vectors/array)

USE COLUMN {(n,1) arrays} or ROW {(1,n) array} vector instead : import numpy as np

```
a.shape=(5,1)  a = np.random.randn(5,1)  ← 5×1 vector (column)
               print(a)
               print(a.T)  → row vector
```

assertions ⇒ documentation for code

```
assert (a.shape == (5,1))
```