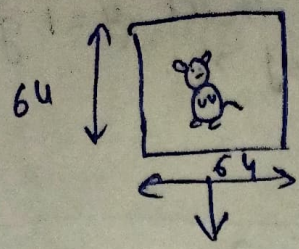
Week 2

★ Binary Classification.

↳ logistic Regression algorithm.

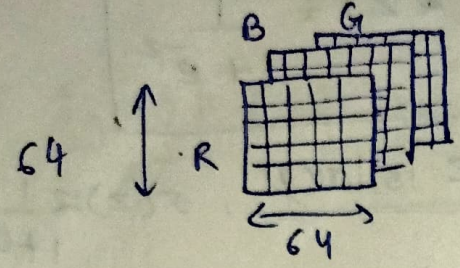
eg.

An image of a cat



output:

$\longrightarrow$ 1 (cat) vs 0 (non cat).



$\longrightarrow$ x = [input feature vector]

$$x = \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_{4096} \\ B_1 \\ B_2 \\ \vdots \\ B_{4096} \\ G_1 \\ G_2 \\ \vdots \\ G_{4096} \end{bmatrix} \quad \text{12288}$$

$= n_x$

↓

dimension of x.

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

↓

data input

m train examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

↓

m_{train}, m_{test} = no. of test examples.

$$X = \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(m)} \\ | & | & | & & | \end{bmatrix} \quad n_x \text{ (rows)}, \quad m \text{ (columns)}$$

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

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \cdots & y^{(m)} \end{bmatrix} \Rightarrow Y \in \mathbb{R}^{1 \times m}$$

• Logistic Regression.

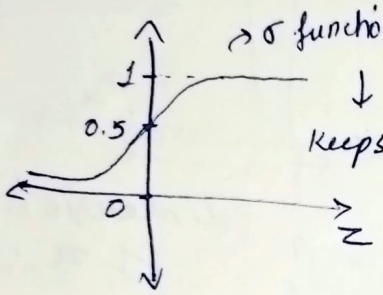
Given x, $\hat{y} = P(y=1|x)$ → Probability that $y=1$ given the value of x.

$x \in \mathbb{R}^{n_x}$ ↓ output $0 \le \hat{y} \le 1$

Parameters :- $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$.

output :- $\hat{y} = \sigma(\underbrace{w^T x + b}_{z})$

w transpose



Keeps value between 0 and 1.

W = weights (how much importance do we give to a particular feature)

b = bias (decides the threshold of the neuron)

no of
W = no of features × no of examples
b = no of ~~examples~~ neurons

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

If z is large, $\sigma(z) \simeq \frac{1}{1+0} = 1$

If z is large negative no., $\sigma(z) = \frac{1}{1+\text{Big no.}} \simeq 0$

Basically we need to find the values of parameters w and b such that $\hat{y}$ will give an accurate probability of ~~prediction~~ y being equal to 1.

• Logistic Regression cost function.

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

Given $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$, we want

$\hat{y}^{(i)} \simeq y^{(i)}$

Loss (error) function :- $L(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$

If $y=1$: $L(\hat{y}, y) = -\log \hat{y}$ ← we want $\log \hat{y}$ Large.

↓ should be small

$\hat{y} \rightarrow$ Large.

$(0 \leq \hat{y} \leq 1) \Rightarrow \hat{y}$ should be closer to 1.

$\hat{y} \simeq 1$ ← be closer to 1.

If $y=0$: $L(\hat{y}, y) = -\log(1-\hat{y})$ ← $\log(1-\hat{y})$ should be large.

⇓

$1-\hat{y} \rightarrow$ large.

$\hat{y} \rightarrow$ small.

⇓

$\hat{y} \simeq 0$

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

↓

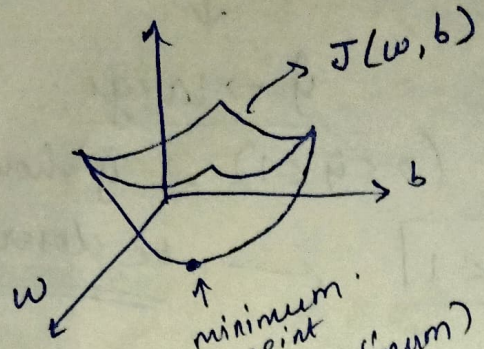
average loss over all the data examples.

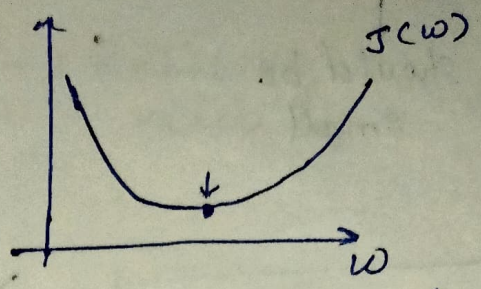
$= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)}) \right]$

(Whereas, Loss function is defined over a single example.

Note → We want to find parameters w, b for which the cost function will be minimum.

★ Gradient Descent.



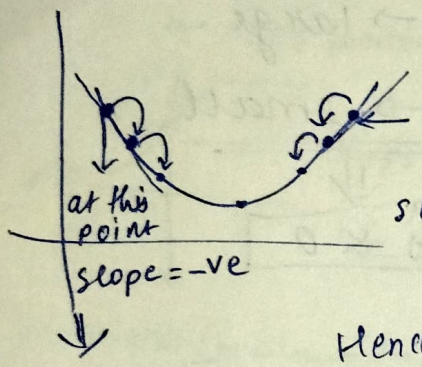


Repeat {

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

(learning rate)

}



$\frac{dJ(w)}{dw}$ ↑ slope $\Rightarrow dw \Rightarrow$ $w := w - \alpha\, dw$

Hence, $w := w - \alpha\, dw$ → we are subtracting from w, eventually reaching the minimum point.

Hence,

$w := w - \alpha(-dw)$

$w := w + \alpha\, dw$

↳ we are adding to w, eventually reaching the minimum point.

---

for $J(w,b)$

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

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

Gradient descent tells us what changes to which weights matters the most or results in fastest decrease in cost function.

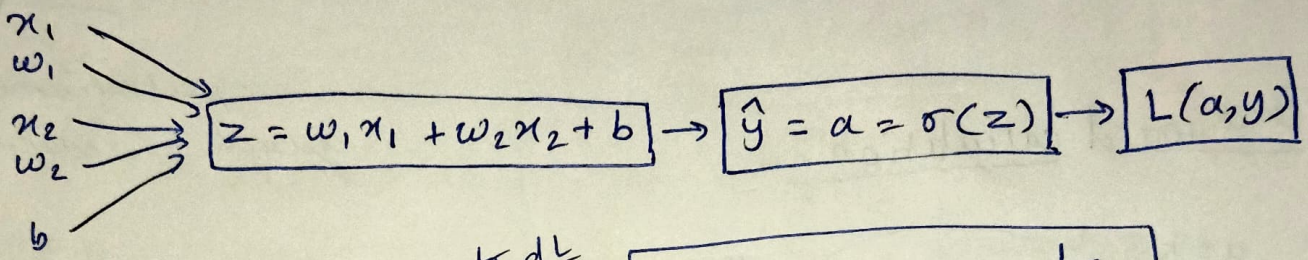
★ Logistic Regression Gradient Descent :

$z = w^T x + b$ $\qquad \sigma(z) = \frac{1}{1+e^{-z}} = a$

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

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

eg:-



$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

$dw_1 = \frac{dL}{dw_1} = \frac{\partial L}{\partial w_1} = x_1 \cdot dz$

$dw_2 = \frac{\partial L}{\partial w_2} = x_2 \cdot dz$

$db = \frac{\partial L}{\partial b} = dz$

$dz = \frac{dL}{dz} = \frac{dL}{da} \cdot \frac{da}{dz}$

$dz = a - y$

$\frac{dL}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$

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

- Logistic regression on 'm' examples.

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

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

★ Computation Graph.

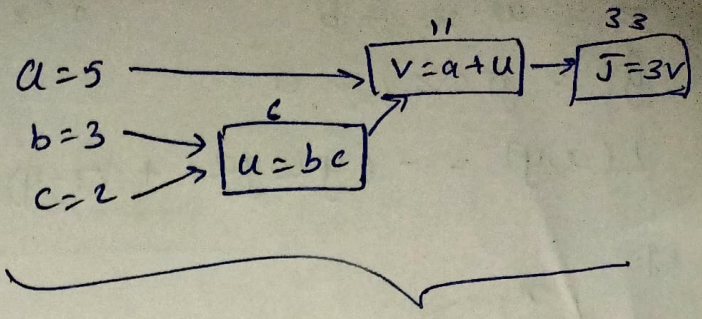
Forward :-

ex:- $J(a,b,c) = 3(a+bc) = 33$

$u = bc$

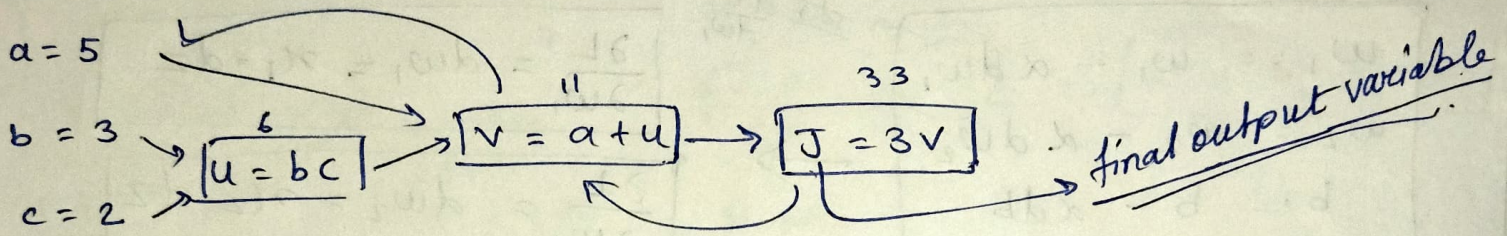
$v = a+u$

$J = 3v$



computational graph (forward)

Backward calculation :-



$\frac{dJ}{dv} = 3 \Rightarrow$ $J = 3v$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

(increase in J = 3 times increase in v)

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

$= 3 \cdot 1 = 3$

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

$\{ a \rightarrow v \rightarrow J \}$

Usually in code, we are interested in calculating the ~~derivative~~ derivative of final output variable.

$\frac{d \text{ FinalOutputVar}}{d \text{ var}}$ ← this is too long.

So, we could replace it with just "d var", which is derivative of final output variable with respect to 'var'.

* Vectorization

$z = w^T x + b$.

$w = \begin{bmatrix} \vdots \end{bmatrix}$ $x = \begin{bmatrix} \vdots \end{bmatrix}$ $w, x \in \mathbb{R}^{n_x}$

non-vectorized:

```
z = 0
for i in range(n_x):
    z += w[i] * x[i]
z += b
```

Vectorized:

```
z = np.dot(w, x) + b
```

$w^T x$.

This version is almost 300 times faster!

eg:- we want to apply exponential operation on every element of a matrix/vector.

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \rightarrow u = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

non-vectorized:

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

vectorized:

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

(Easy!)

Hence, in vectorized version,

$dw_1 = 0$, $dw_2 = 0$ ✗ → $dw = np.zeros((n_x, 1))$

and $dw_1 = dw_1/m$, $dw_2 = dw_2/m$ → $dw /= m$

✗

for $(x^{(i)}, y^{(i)}) \rightarrow dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

$$\Rightarrow \boxed{\frac{\partial J(w,b)}{dw_1} = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial L(a^{(i)}, y^{(i)})}{\partial w_1^{(i)}}}_{dw_1^{(i)}}}$$

---

initialize $J=0;\ dw_1=0;\ dw_2=0;\ db=0$

For $i=1$ to m:

$$z^{(i)} = w^T x^{(i)} + b$$
$$a^{(i)} = \sigma(z^{(i)})$$
$$J\ +\!= -\left[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})\right]$$

$$dz^{(i)} = a^{(i)} - y^{(i)}$$
$$dw_1\ +\!= x_1^{(i)} * dz^{(i)}$$
$$dw_2\ +\!= x_2^{(i)} * dz^{(i)}$$
$$db\ +\!= dz^{(i)}$$

$n=2$ ↕

→ has many loops. Some large datasets might have a large n. Then this method gets inefficient.

$J\ /\!= m$
$dw_1\ /\!= m$
$dw_2\ /\!= m$
$db\ /\!= m$

Hence,

$$w_1 := w_1 - \alpha\, dw_1$$
$$w_2 := w_2 - \alpha\, dw_2$$
$$b := b - \alpha\, db$$

} 1 step of Gradient descent.

To get rid of inefficient loops, we use **Vectorization.**

✱ Vectorizing Logistic Regression.

→ Let's say we have m data examples.

$$z^{(1)} = w^T x^{(1)} + b \qquad z^{(2)} = w^T x^{(2)} + b \qquad z^{(3)} = w^T x^{(3)} + b \quad \cdots$$

$$a^{(1)} = \sigma(z^{(1)}) \qquad a^{(2)} = \sigma(z^{(2)}) \qquad a^{(3)} = \sigma(z^{(3)})$$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad (n_x \times m) \qquad w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \} n_x \qquad w^T = \underbrace{[\cdots]}_{n_x}$$

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

$$\downarrow$$

$$\underbrace{[\cdots]}_{w^T\ 1 \times n_x} \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}_{n_x \times m} = \underbrace{[\cdots]}_{1 \times m}$$

$$\therefore \underbrace{[z^{(1)}\ z^{(2)}\ z^{(3)} \cdots z^{(m)}]}_{Z} = [\underbrace{w^T x^{(1)} + b}_{z_1} \quad \underbrace{w^T x^{(2)} + b}_{z_2} \quad \cdots \quad \underbrace{w^T x^{(m)} + b}_{z_m}]$$

↳ This can be done using :- ~~np.do~~

$$\boxed{Z = np.dot(w.T, X) + b}$$

→ here, python automatically adds b to all numbers in the $w^T x^{(i)}$ matrix.

↓

This is called broadcasting

Transpose

a real number

★ Vectorizing Logistic Regression:

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

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

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

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

$$dw = \frac{\sum_{i=1}^{m} x^{(i)} \cdot dz^{(i)}}{m} \qquad db = \frac{\sum_{i=1}^{m} dz^{(i)}}{m}$$

$\rightarrow db = [np.sum(dz)]/m$

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

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

$$dw = \frac{1}{m} \begin{bmatrix} x^{1} dz^{1} + x^{2} dz^{2} + \cdots + x^{(m)} dz^{(m)} \\ \vdots \end{bmatrix}_{n_x \times 1}$$

Updated: implementing logistic Regression:

$Z = w^{T}X + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

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

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

Gradient Descent :- $w := w - \alpha dw$, $b := b - \alpha db$ } we might need a for loop here tho.

* Broadcasting in Python.

We want to find % of calories from carb, protein, fat.

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
fat	1.8	135.0	99.0	0.9

$= A_{(3\times4)}$

cal = A.sum(axis=0) → ~~horizo~~ vertical addition (axis=1 → horizontal)

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

Broadcasting.

this is not really necessary since A.sum(axis=0) already made the cal → (1,4).

Broadcasting example:-

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

$$\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}_{(2,3)} + \begin{bmatrix}100\\200\end{bmatrix}_{(2,1)\rightarrow(2,3)} \rightarrow \begin{bmatrix}101&102&103\\204&205&206\end{bmatrix}$$