(1) Lec #15

→ let see Computational graph for logistic regression algorithm & do the forward & backward pass on the graph.
→ using graph we can compute gradient / derivative / slope efficiently.

Logistic Regression Recap:

$$\underset{[---]}{w^T}\ \underset{\begin{bmatrix}1\\1\end{bmatrix}}{x}$$

(1) $z = w^T x + b$

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

$z = \theta^T x + \theta_0$

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

(3) $\mathcal{L}(a, y) = -[y \log(a) + (1-y)\log(1-a)]$

(for 1 example)

$x_1, w_1, x_2, w_2, b \to$ (1) $\boxed{z = w_1 x_1 + w_2 x_2 + b} \to$ (2) $\boxed{\hat{y} = a = \sigma(z)} \to$ (3) $\boxed{\mathcal{L}(a, y)}$

$dz = \dfrac{d\mathcal{L}}{dz}$

$= a - y$

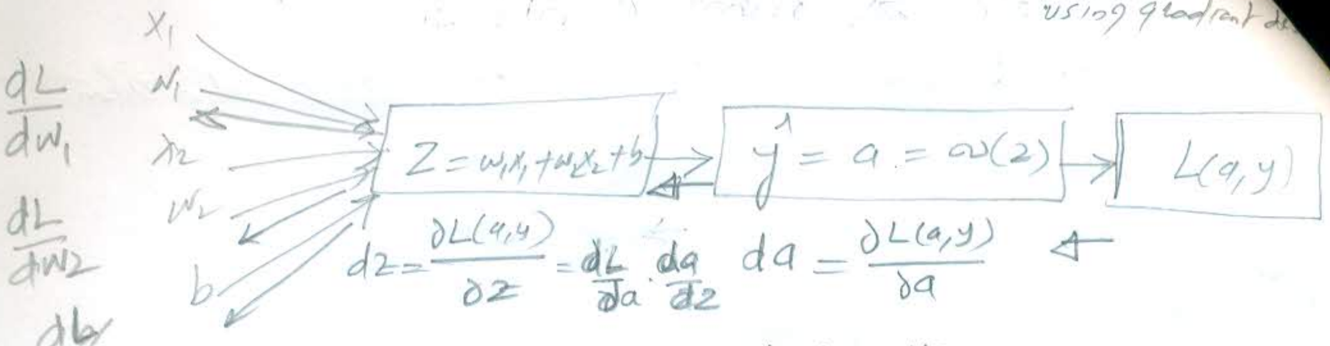
$\dfrac{d\mathcal{L}(a,y)}{da} = da = \dfrac{-y}{a} + \dfrac{1-y}{1-a}$

$dz = \underbrace{\dfrac{d\mathcal{L}}{da}}\cdot\dfrac{da}{dz} = a - y$ $\quad \dfrac{da}{dz} = (a(1-a))$

– In order to reduce the loss, we need to update w_1, w_2 & b — so we need to compute the derivative

z

- In order to Reduce the loss ($\mathcal{L}(a,y)$) for a single Training example, we have to modify the parameters using gradient descent



- In forward pass we calculate the loss
- In Backward pass we calculate the gradients
- Use those gradients & modify or update the parameters (w_1, w_2, b) to reduce the loss

$\Rightarrow \frac{\partial L(a,y)}{\partial a} = ?$

$$\frac{\partial L(a,y)}{\partial a} = \frac{\partial}{\partial a}\left[\left(y \log(a) + (1-y)\log(1-a)\right)\right]$$

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

$$\frac{\partial L(a,y)}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$$

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

$\Rightarrow \frac{\partial L(a,y)}{\partial z} = ?$

$$\frac{\partial L(a,y)}{\partial z} = \frac{\partial L}{\partial a} * \frac{\partial a}{\partial z} = da \cdot \frac{\partial a}{\partial z}$$

(chain Rule)

As $a = \frac{1}{1+e^{-z}}$

$$\boxed{\frac{\partial a}{\partial z} = a(1-a)}$$ // property of sigmoid function derivative

$$dz = \frac{\partial L}{\partial z} = \left(\frac{-y}{a} + \frac{1-y}{1-a}\right) * a(1-a)$$

$$= -y(1-a) + (1-y)a$$

$$= -y + \cancel{ya} + a - \cancel{ya}$$

$$\boxed{dz = a - y}$$

$$dw_1 = \frac{\partial L(a,y)}{\partial w_1} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial w_1} = dz \cdot \left(\frac{\partial z}{\partial w_1}\right)$$

$$z = w_1 x_1 + w_2 x_2 + b$$

$$\boxed{\frac{\partial z}{\partial w_1} = x_1}$$

$$\therefore \boxed{dw_1 = (a-y) \cdot x_1}$$

$$dw_2 = \frac{\partial L(a,y)}{\partial w_2} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_2} = dz \cdot \frac{\partial z}{\partial w_2}$$

$$a) \quad z = w_1 x_1 + w_2 x_2 + b$$

$$\boxed{\frac{\partial z}{\partial w_2} = x_2}$$

$$\boxed{dw_2 = (a-y) x_2}$$

$$db = \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b}$$

as $z = w_1 x_1 + w_2 x_2 + b$

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

$$\therefore \boxed{db = a - y}$$

$\therefore$ * Gradient descent updates for logistic regression for a single training example.

$\boxed{(h_\theta(x) - y)}$

$$\begin{aligned}
w_1 &:= w_1 - \alpha\, dw_1 = w_1 - \alpha (a-y) x_1 = w_1 - \alpha\, dz\, x_1 \\
w_2 &:= w_2 - \alpha\, dw_2 = w_2 - \alpha (a-y) x_2 = w_2 - \alpha\, dz\, x_2 \\
b &:= b - \alpha\, db = b - \alpha (a-y) = b - \alpha\, dz
\end{aligned}$$

$\boxed{(h_\theta(x) - y)} \Rightarrow$ it is similar to our old proof

* Take away message

  * Use of computation graph makes the calculation of gradients/derivatives very efficient.

→ m examples

→

Gradient descent on m examples

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

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

$$d\omega_1^{(i)},\ d\omega_2^{(i)},\ db^{(i)}$$

$$d\omega_1 \Rightarrow \frac{\partial}{\partial \omega_1} J(\omega, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \omega_1} \mathcal{L}(a^{(i)}, y^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) x_1^{(i)}$$

$$d\omega_2 = \frac{\partial}{\partial \omega_2} J(\omega, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \omega_2} \left(\mathcal{L}(a^{(i)}, y^{(i)}) \right) = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) x_2^{(i)}$$

$$db = \frac{\partial}{\partial b} J(\omega, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b} \left(\mathcal{L}(\hat{a}^{(i)}, \hat{y}^{(i)}) \right) = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

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

// forward pass

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

// dimension / Backward pass

$$dw_1 += x_1^{(i)} dz^{(i)}$$
$$dw_2 += x_2^{(i)} dz^{(i)}$$
$$db += dz^{(i)}$$

// backward pass. $n = 2$; if $n > 2$ loop

for j in range: $dw_j += x_j^{(i)} dz^{(i)}$

$J /= m$

$dw_1 /= m$

$dw_2 /= m$

$db /= m$

// average the gradients

$$\omega_1 := \omega_1 - \alpha dw_1$$
$$\omega_2 := \omega_2 - \alpha dw_2$$
$$b := b - \alpha db$$

// update weights

2 weakness: • 2 For loops → make code less efficient

• In Deep learning Era, we have bigger dataset, so need to do training w/o loops

⇒ Vectorization of training code helps to remove explicit for loop.

$$\boxed{Z = w^T x + b} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_{n_x} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{n_x}$$

w & x are large vector $w \in R^{n_x}$, $x \in R^{n_x}$

eg

$$w = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad x = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$$

- Non vectorized Implementation

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

same

- Vectorized Implementation

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

$$\boxed{Z = w^T x + b}$$

(1, 2) (2, 1)

$w_1 * x_1 + w_2 * x_2 + b$

(single Instruction multiple data)

// np.dot() numpy functions both in cpu or gpu mode, uses SIMD instruction that do computation in parallel, which ultimately run the code very fast.

$w = w - \alpha dw$

$b = b - \alpha db$

<u>code</u> / Demo for vectorized result

c = np.dot(a, b)   vs   for i in range(1000000)
        c += a[i] * b[i]

```
import time

a = np.random.rand(1000000)      // uniform distribution

b = np.random.rand(1000000)

tic = time.time()

c = np.dot(a, b)

toc = time.time()

print(c)
print("vectorized version:" + str(1000 * (toc - tic)) + "ms")   // ~1.5ms

// Let see Non vectorized version
c = 0
tic = time.time().
for i in range(1000000)
        c += a[i] * b[i]

toc = time.time()

print c
print("for loop:" + str(1000 * (toc - tic)) + "ms")   // 300 ms
```

vs

* Take away
use built function to run code faster as they use SIMD instructions

→ Whenever possible, avoid expl... A ... V

$$U = \underset{\text{matrix}}{A}\ \underset{\text{vector}}{V} \qquad \underset{3\times3}{[\ ]}\ \underset{3\times1}{[\ ]} \ U$$

$$U = \sum_j \underset{i\times j}{A_{ij}}\ \underset{j\times1}{V_j} \qquad = \underset{3\times1}{[\ ]}$$

Non vectorized Imp

```
u = np.zeros((n,1))
for i in range(3)
    for j in range(3)
        u[i] += A[i][j] * v[j]
```

VS

vectorized Imp

$$\boxed{u = np.dot(A, V)}$$ // it eliminates 2 For loops

$$\Rightarrow \underset{A}{\underset{3\times3}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}}}\ \underset{V}{\underset{3\times1}{\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}}} = \underset{U}{\underset{3\times1}{\begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix}}}$$

U = np.dot(A, V)

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

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

* Vectors & matrix valued functions

→ If we need to apply the exponential operation on every element of a matrix/vector

$$V = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} \Rightarrow u = \begin{bmatrix} e^{u_1} \\ \vdots \\ e^{u_n} \end{bmatrix}$$

→ **Non vectorized Imp:**

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

→ **Vectorized Imp.:**

- Numpy have many built in function that work in vectorized fashion

```
u = np.exp(V)
```

```
np.log(V)
np.abs(V)
np.maximum(V, 0)
```

```
V ** 2      // elementwise square
1/V         // element wise inverse
```

⇒ Python operators do element square efficiently

Logistic regression algorithm to make loop free code.

First Attempt

$J = 0$, $\boxed{dw_1 = 0, \cancel{dw_2 = 0}}$, $db = 0$ $w = [\ ]_{n_x}$ $x^{(i)} = [\ ]_{n_x}$

accumulator

$dw = np.zero((n_x, 1))$

for $i = 1$ to m:

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

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

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

$$dz^{(i)}_{(1\times1)} = a^{(i)} - y^{(i)}$$

if $n_x = 2$

$$\text{for } j \text{ in range}(n_x): \quad dw_j += x^{(i)}_j dz^{(i)}$$

$$dw_1 += x^{(i)}_1 dz^{(i)}$$

$$dw_2 += x^{(i)}_2 dz^{(i)}$$

vs

vector e scalar multiply

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

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

$J = J/m$, $\boxed{\cancel{dw_1 = dw_1/m}, \cancel{dw_2 = dw_2/m}}$, $db /= m$

$$dw /= m$$

- Now we have one loop instead of 2 loop which make the program much faster.

$w = w - \alpha dw$

$b = b - \alpha db$

Vectorized logistic Regression

* To speed up Implementation of Training process — process the entire training set — using the idea of vectorization

Forward Pass

* currently forward propagation steps for m training example inside a loop

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

$$a^{(1)} = \sigma(z^{(1)}) \quad\Big|\quad a^{(2)} = \sigma(z^{(2)}) \quad\Big|\quad a^{(3)} = \sigma(z^{(3)})$$

- w/o loop, How we perform computation?

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \in R^{n_x, m}$$

(n_x, m)

$$W = \begin{bmatrix} \\ \\ \\ \end{bmatrix}_{(n_x, 1)} \Rightarrow W^T = [\quad\quad]_{(1, n_x)} \quad \& \quad b = [b \; b \cdots b]_{(1 \times m)}$$

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

broadcasting

$$\underset{1 \times n_x}{W^T} * \underset{(n_x, m)}{\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}} + \underset{1 \times m}{[b \; b \cdots b]}$$

$$= [W^T x^{(1)} + b \quad W^T x^{(2)} + b \quad \cdots \quad W^T x^{(m)} + b]$$ // automatic // stack of z's for all // training examples

$$= [z^{(1)} \quad z^{(2)} \quad \cdots \quad z^{(m)}] = Z$$

(13)

$$Z = [z^{(1)} \quad z^{(2)} \quad \ldots \quad z^{(m)}] \Rightarrow A = \sigma(Z)$$

$$A = [a^{(1)} \quad a^{(2)} \quad \ldots \quad a^{(m)}]$$

$$A = \sigma(Z)$$

$$\boxed{A = 1/(1 + np.exp(-Z))}$$

vectorized solution ✓

NOTE: Dimension

$X \Rightarrow (n_x, m)$

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

need

$dw \Rightarrow (n_x, 1)$

Backward Pass

- As we know

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

$$\therefore \quad dZ = [dz^{(1)} \quad dz^{(2)} \quad \ldots \quad dz^{(m)}]$$ // stacking horizontally

$1 \times m$

$$A = [a^{(1)} \quad \ldots \quad a^{(m)}]$$ ✓ // stack horizontally

$$Y = [y^{(1)} \quad \ldots \quad y^{(m)}]$$ // stack horizontally

$(1 \times m)$

$$\boxed{dZ = A - Y}_{m \times 1} = [a^{(1)} - y^{(1)} \quad a^{(2)} - y^{(2)} \quad \ldots \quad a^{(m)} - y^{(m)}]$$

- we do

$(n_x, m) * (m \times 1)$

$\frac{1}{m} * np.dot(X, dZ.T)$

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

$(n_x, 1)$

need to do transpose dZ as it is $(1, m)$ matrix

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

or

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

G.D

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

Broad casting in python

- calories from carbs, proteins, Fats in 100 grams of different food.

$$
\begin{array}{r c}
 & \begin{array}{cccc} \text{Apple} & \text{Beef} & \text{Eggs} & \text{potatos} \end{array} \\
\begin{array}{r} \text{carb} \\ \text{protein} \\ \text{Fat} \end{array} & \begin{bmatrix} 56.0 & 0.0 & 4.4 & 68.0 \\ 1.2 & 104.0 & 52.0 & 8.0 \\ 1.8 & 135.0 & 99.0 & 0.9 \end{bmatrix} \overset{\text{let}}{=} A
\end{array}
$$

T.cal = $\overline{59}$ cal in apple

- calculate % of calories from carb, protein, and fat.
- can we do it without explicit for loop?

cal = A. sum (axis = 0) // vertically add.

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

More examples regarding broad casting

$$(w^T x + b)$$

① let we ⇒ $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}_{4\times1} + 100$, python does broadcasting operation & generate a [4x1] matrix as shown below. Then perform the Addition operation

$$= \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}_{4\times1} + \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix}_{4\times1} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}_{4\times1}$$

② Similarly

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2\times3} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix}_{1\times3}$$

python performs broadcasting operation & generate [2x3] matrix corresponds to [1x3]

$$= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2\times3} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix}_{2\times3}$$

$$= \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix}$$

(2,3) (2,1)

Python make 2x3 from 2,1 matrix by copying the column values 3 time.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 100 & 100 \\ 200 & 200 & 200 \end{bmatrix}$$

$$= \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}$$

- $\boxed{\text{General principle for broadcasting}}$

1) If (m, n) matrix & perform arithmetic operation with (1, n) matrix $\begin{bmatrix} + , - , * , / \end{bmatrix} \xrightarrow{\text{broadcasting it}} (m, n)$

2) If (m, n) matrix & $\begin{bmatrix} + , - , * , / \end{bmatrix}$ operation (m, 1) matrix $\xrightarrow{\text{broadcast}}$ (m, n)

3) If (m, 1) & $\begin{bmatrix} + , - , * , / \end{bmatrix}$ with Real num $\xrightarrow{\text{broadcast}}$ (m, 1) by copying the real value

eg $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + 100 \longrightarrow \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 100 \\ 100 \\ 100 \end{bmatrix}$

4) (1, n) & $\begin{bmatrix} + , - , * , / \end{bmatrix}$ with real no $\rightarrow$ (1, n)

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

- Tip to avoid buggs in code

```
import numpy as np
a = np.random.randn(5)
```
└ Gaussian variable
```
print(a)
print(a.shap)      // (5,) is Rank 1 array
print(a.T)
```
↳ not a row or a column vector
```
print(np.dot(a, a.T)  // is a (no), but we may expect vector
```

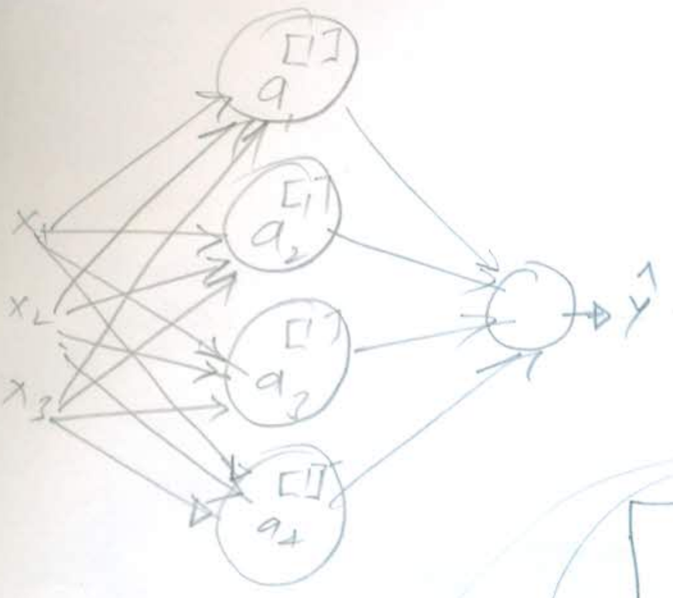
Soln

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

5,1 / 1,5 ↳ (5,5)

```
assert(a.shap == (5,1)))   // to check AssertionError
```

* Neural NN Representation



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_2^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

$$\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_2^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$

is a vector

$W^{[1]}$

$b^{[1]}$

$(4, 1)$

$$= \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]} x + b_3^{[1]} \\ w_4^{[1]} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = z^{[1]}$$

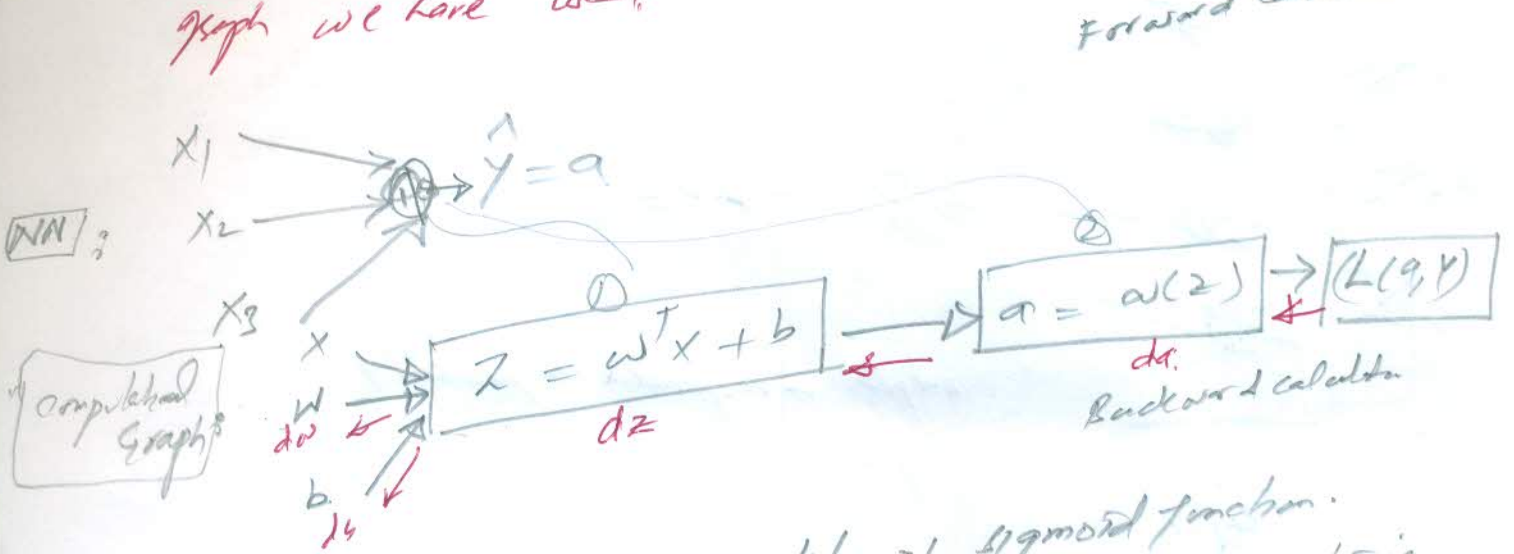
⇒ Rule of Thumb: stack the z vertically as nodes are placed vertically in NN

$$\sigma(z_1^{[1]}) = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = a^{[1]}$$

What is NN ?

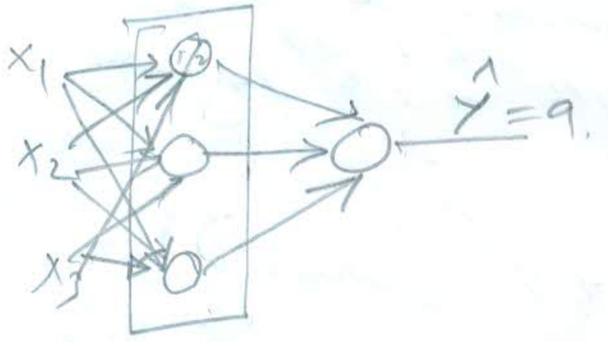
25/04/2024

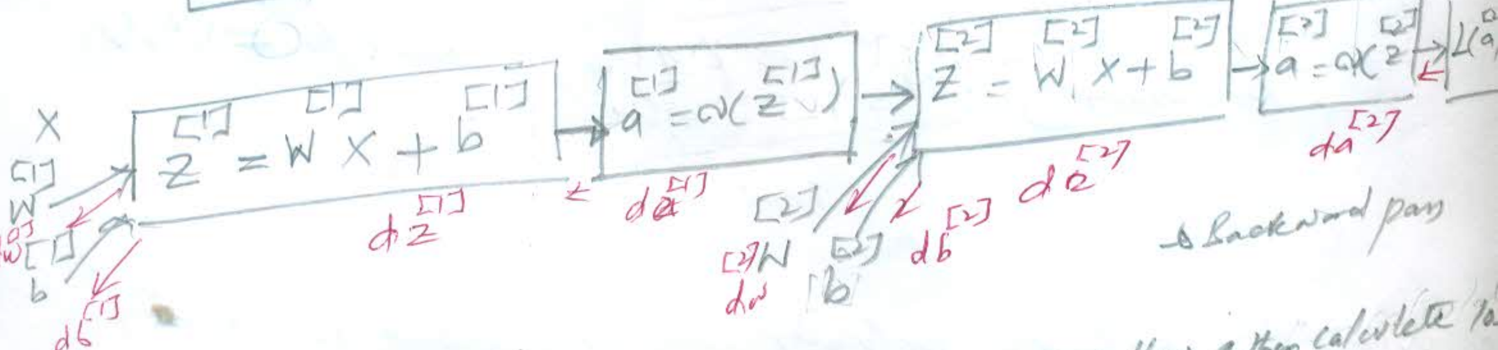
- For logistic regression algorithm, the following computational graph we have used.



NN : • stacking together lot of sigmoid function.

- $\boxed{z}$ followed by $\boxed{a}$ calculation is done in multiple times
- logistic regression repeating twice.

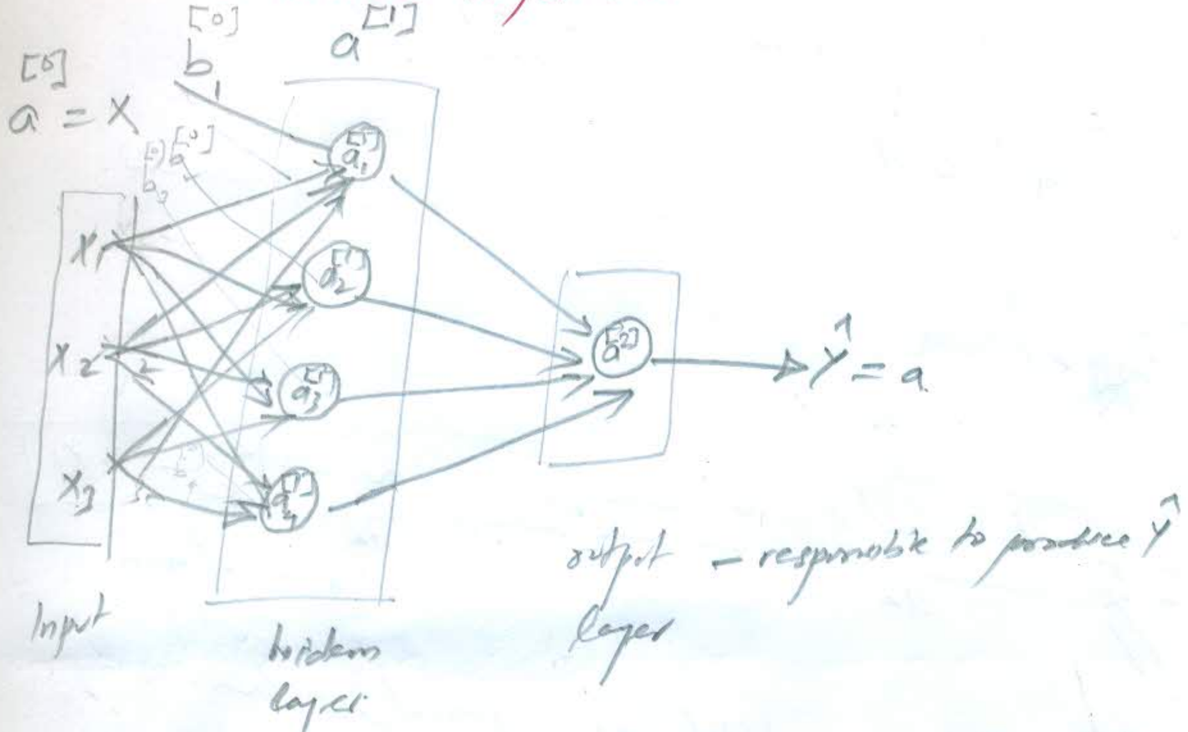




- In NN, we have multiple z follow a, z follow a, then calculate loss
- Similarly backward pass is to calculate derivatives

26 / C1W3L02

- Neural Network presentation : Details
- One hidden layer NN



- what are calculation $a^{[0]} = X$, $a^{[1]}$, $a^{[2]} = Y$

- it is 2 layer NN, not count input layer
- hidden layer associated parameters $w^{[1]}$, $b^{[1]}$, whereas output layer is associated with $w^{[2]}$, $b^{[2]}$ parameters

- Dimension of $w^{[1]}$ (matrix) is $(4, 3)$ (matrix), where $b^{[1]} = (4, 1)$
- Dimension of $w^{[2]}$ is $(1, 4)$, where $b^{[2]} = (1, 1)$

1 unit 4 unit

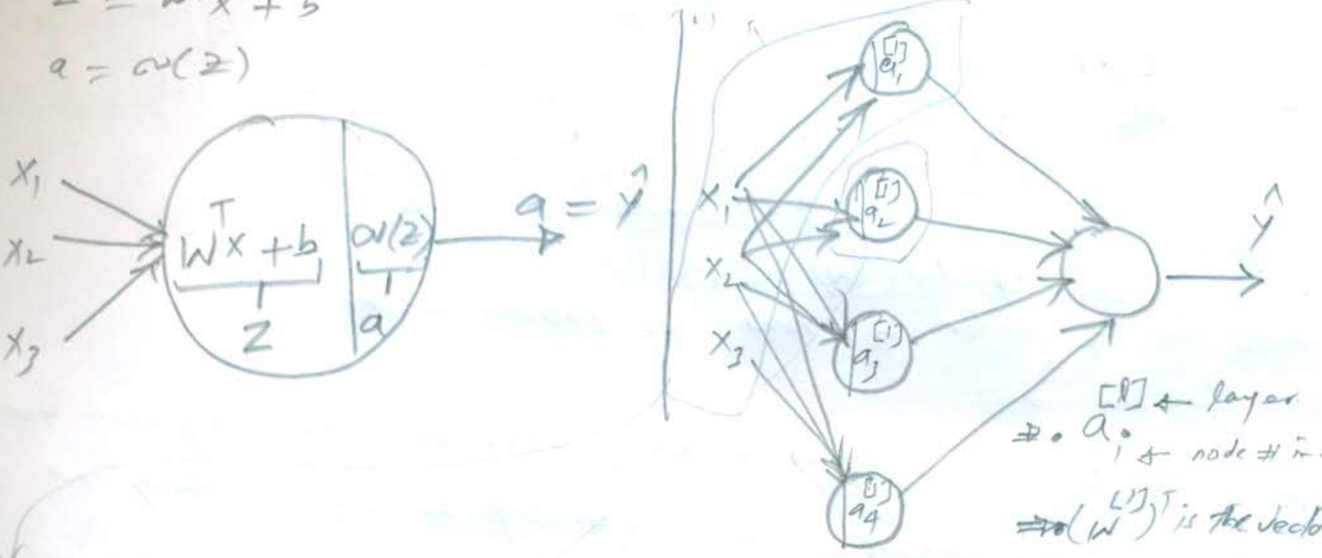
- What is the network actually computing, see in Next page.

27_ Details of what NN with one hidden layer is Computing:

→ In logistic Regression a single unit compute two things ① z followed by ② a

$z = w^T x + b$

$a = \sigma(z)$



⇒ $a_i^{[l]}$ ← layer, ← node # in a layer

⇒ $(w^{[l]})^T$ is the vector transpose

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

Do calculation using for loop is inefficient, so take 4 equation & vectorized them.

$$z^{[1]} = \underbrace{\begin{bmatrix} \text{---} & w_1^{[1]T} & \text{---} \\ \text{---} & w_2^{[1]T} & \text{---} \\ \text{---} & w_3^{[1]T} & \text{---} \\ \text{---} & w_4^{[1]T} & \text{---} \end{bmatrix}}_{(4\times3)\ \text{matrix}}^{W^{[1]}} \underset{3\times1}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}} + \underbrace{\underset{4\times1}{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}}_{b^{[1]}} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$\boxed{Z1 = np.dot(W1, x) + b1}$

- Stack the 4 vector in a row as shown above.

→ Vectorization Rule of thumb, when we have different node in a layer, we stack them vertically — so similarly we stack their results vertically in matrix

- we call (4,3) weight matrix as $W^{[1]}$ and (4,1) bias matrix as $b^{[1]}$ at hidden unit

- Similarly

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

element

- $\boxed{A1 = np.tanh(Z1)}$

Given input x ($a^{[0]}$)

$$z^{[1]}_{4,1} = W^{[1]}_{(4,3)} \, x_{(3,1)} + b^{[1]}_{(4,1)}$$

$$a^{[1]}_{(4,1)} = \sigma\left(z^{[1]}_{(4,1)}\right)$$

$$z^{[2]}_{(1,1)} = W^{[2]}_{(1,4)} \, a^{[1]}_{(4,1)} + b^{[2]}_{(1,1)}$$

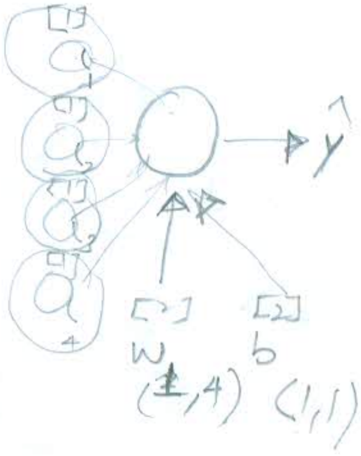
$$a^{[2]}_{(1,1)} = \sigma\left(z^{[2]}_{(1,1)}\right)$$

z1 = np.dot(W1,x) + b1

a1 = np.tanh(z1)

z2 = np.dot(w,a1) + b2

a2 = sigmoid(z2)



$$W^T = W^{[2]}$$

$$b = b^{[2]}$$

* If we just think the output unit as the logistic regression unit

$$\therefore \quad W^{[2]} = W^T$$

$$b^{[2]} = b$$

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

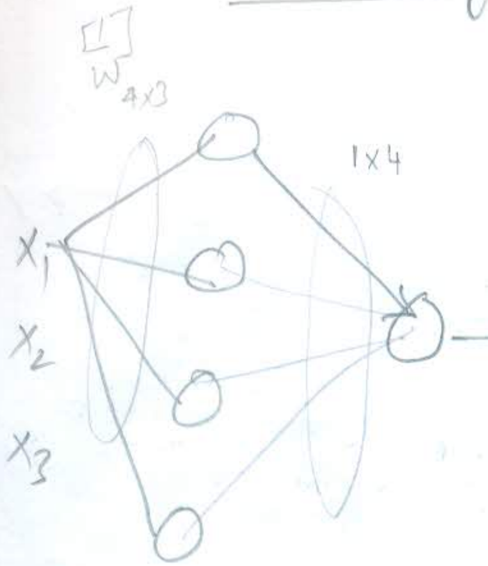
$$\hat{y} = a = \sigma(z)$$

* We have to compute only above 4 equations for computing $\hat{y}$ for a <u>single training example.</u>

* Now let see vectorization across multiple examples.

ONE hidden layer Neural Network

Vectorizing across multiple examples



$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

$$x \longrightarrow a^{[2]} = \hat{y}$$

$$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$$ (for a single Training examples)

$$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$$ ~ $a^{[2](i)}$ — (i): example i; $[2]$: layer 2

⋮

$$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$$

$\boxed{\cancel{xxx}}$

For $i = 1$ to m:

$$z^{[1](i)} = W^{[1]}_{(4\times3)} x^{(i)}_{(3\times1)} + b^{[1]}$$
$$a^{[1](i)}_{4\times1} = \sigma(z^{[1](i)}_{(4\times1)})$$
$$z^{[2](i)} = W^{[2]}_{(1\times4)} a^{[1](i)}_{(4\times1)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

* We can also avoid this for loop also using vectorization.

vectorized across multiple examples

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

$3 \quad 100$

punch line: we have to compute the following

$Z^{[1]} = W^{[1]} X + b^{[1]}$ (if $m = 100$)

$Z_{4\times 100}$ $(4,3)$ $(3,100)$

$A^{[1]}_{4\times 100} = \sigma(Z^{[1]})_{(4,100)}$

$Z^{[2]} = W^{[2]}_{1\times 4} A^{[1]}_{4\times 1} + b^{[2]}$

$A^{[2]} = \sigma(Z^{[2]})_{(1\times 1)}$

previously we have compute

for $i = 1$ to m:

$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$

$a^{[1](i)} = \sigma(z^{[1](i)})$

$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$

$a^{[2](i)} = \sigma(z^{[2](i)})$

$$Z^{[1]} = \begin{bmatrix} Z_1^{[1](1)} & Z_1^{[1](2)} & & Z_1^{[1](m)} \\ Z_2^{[1](1)} & Z_2^{[1](2)} & \cdots & Z_2^{[1](m)} \\ Z_3^{[1](1)} & Z_3^{[1](2)} & & Z_3^{[1](m)} \\ Z_4^{[1](1)} & Z_4^{[1](2)} & & Z_4^{[1](m)} \end{bmatrix}_{(4\times 100)}$$

($Z^{[1]}$ obtain by stacking $Z^{[1](i)}$ horizontally) [it could be stacked ... training examples]

training examples →

↓ hidden units at hidden unit

$$A^{[1]} = \begin{bmatrix} a_1^{[1](1)} & a_1^{[1](2)} & & a_1^{[1](m)} \\ a_2^{[1](1)} & a_2^{[1](2)} & \cdots & a_2^{[1](m)} \\ a_3^{[1](1)} & a_3^{[1](2)} & & a_3^{[1](m)} \\ a_4^{[1](1)} & a_4^{[1](2)} & & a_4^{[1](m)} \end{bmatrix}$$

($a^{[1]}$ obtained stacking $a^{[1]}$ horizontally)

$A1 = np.tanh(Z1^{[1]})$

(5)

Justification for correctness of vectorized implementation

$$\boxed{Z^{[1]} = W^{[1]}X + b^{[1]}}$$

$\Rightarrow$ let consider 3 training examples

- For 1st training example

$$Z^{[1](1)} = \underset{(4,3)}{W^{[1]}}\,\underset{(3,1)}{X^{(1)}} + \cancel{b^{[1]}} \rightarrow 0$$

- For 2nd training example

$$Z^{[1](2)} = \underset{4\times3}{W^{[1]}}\,\underset{3\times1}{X^{(2)}} + \cancel{b^{[1]}} \rightarrow 0$$

- For 3rd training example

$$Z = \underset{4\times3}{W^{[1]}}\,\underset{3\times1}{X^{(3)}} + \cancel{b^{[1]}} \rightarrow 0$$

(A)

$$W^{[1]} = \begin{bmatrix} & & \\ & & \end{bmatrix}_{4,3}$$

$$W^{[1]}X^{(1)} = \underset{(4,3)}{\begin{bmatrix} & & \\ & & \end{bmatrix}}\underset{3\times1}{\begin{bmatrix} \\ \\ \end{bmatrix}} = \begin{bmatrix} \circ \\ \circ \\ \circ \\ \circ \end{bmatrix}_{4\times1}$$

$$W^{[1]}X^{(2)} = \begin{bmatrix} \circ \\ \circ \\ \circ \\ \circ \end{bmatrix}_{4\times1}$$

$$W^{[1]}X^{(3)} = \begin{bmatrix} \circ \\ \circ \\ \circ \\ \circ \end{bmatrix}_{4\times1}$$

(B)

$$X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} \\ x_3^{(1)} & x_3^{(2)} & x_3^{(3)} \end{bmatrix}_{3\times3}$$

$$\underset{4\times3}{W^{[1]}}\underbrace{\begin{bmatrix} | & | & | \\ X^{(1)} & X^{(2)} & X^{(3)} \\ | & | & | \end{bmatrix}}_{X}{}_{3\times3} = \begin{bmatrix} \circ & \circ & \circ \\ \circ & \circ & \circ \\ \circ & \circ & \circ \\ \circ & \circ & \circ \end{bmatrix}_{4\times3} = \begin{bmatrix} | & | & | \\ Z^{[1](1)} & Z^{[1](2)} & Z^{[1](3)} \\ | & | & | \end{bmatrix}_{4\times3} = Z^{[1]}$$

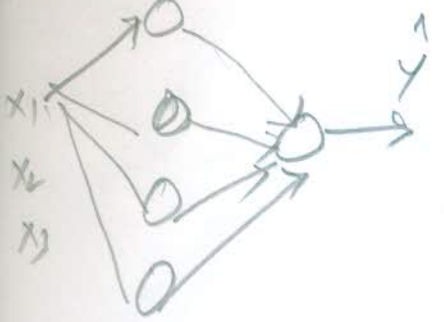
$$+b^{[1]} \quad +b^{[1]} \quad +b^{[1]}$$

using broadcasting

we can add b using broadcasting

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

* Recap of vectorizing across multiple examples.



for $i = 1$ to m:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

let

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}_{3\times m}$$

$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}_{4\times m}$$

show correctness of following the following impl

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

⇒ Gradient Descent for Neural Network with single hidden unit

- Parameters & dimensions: $w^{[1]}$, $b^{[1]}$, $w^{[2]}$, $b^{[2]}$

$(n^{[1]}, n^{[0]})$, $(n^{[1]}, 1)$, $(n^{[2]}, n^{[1]})$, $(n^{[2]}, 1)$

- $n^{[0]}$ are no input features
- $n^{[1]}$ are no of hidden units at layer [1]
- $n^{[2]}$ are the no of hidden units at layer [2]

Cost function: for binary classification

$$J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$

Gradient Descent: // Initialize the parameters randomly, instead of initialize them with zeros

Repeat {

forward pass to compute ① compute prediction $(\hat{y}^{(i)}, \; i = 1, \ldots m)$

compute Derivative ② $dw^{[1]} = \frac{dJ}{dw^{[1]}}, \; db^{[1]} = \frac{dJ}{db^{[1]}} \ldots$

③ update parameters

$$w^{[1]} = w^{[1]} - \alpha \, dw^{[1]}$$
$$b^{[1]} = b^{[1]} - \alpha \, db^{[1]}$$
$$w^{[2]} = w^{[2]} - \alpha \, dw^{[2]}$$
$$b^{[2]} = b^{[2]} - \alpha \, db^{[2]}$$

}

→ Equations for Forward & backward propagations

* Forward pass // all are vectorized

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[1]} A^{[1]} + b^{[1]}$$

as it classification problem

$$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

* $\boxed{\text{Backward pass to calculate derivatives}}$

let $Y = \{ y^{(1)}, y^{(2)} -- y^{(m)} \}$ // Ground Truth

$$dZ^{[2]} = \frac{dL}{dZ^{[2]}} = A^{[2]} - Y$$

$(n^{[2]},)$

$(n^{[2]}, 1)$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis=1, keepdims=True)$$

similar to logistic Regression

$$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{(n^{[1]}, m)}$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis=1, keepdims=True)$$

$(n^{[1]}, 1)$

$(n^{[1]},)$

Computational Graph for Logistic Regression

$x, w, b \rightarrow \boxed{z = w^T x + b} \rightarrow \boxed{a = \sigma(z)} \rightarrow \boxed{\mathcal{L}(a,y)}$

$dw, db \leftarrow dz \leftarrow da$

As $\mathcal{L}(a,y) = -(y \log a + (1-y)\log(1-a))$

$da = \frac{\partial \mathcal{L}(a,y)}{\partial a} = -\left(\frac{y}{a} + \frac{1-y}{1-a}(-1)\right) = \boxed{\frac{-y}{a} + \frac{1-y}{1-a}}$

As $dz = \frac{\partial \mathcal{L}}{\partial z} = \boxed{\frac{\partial \mathcal{L}}{\partial a} * \frac{\partial a}{\partial z}} = da \cdot \frac{da}{dz}$

(chain rule of calculus)

As for logistic regression we use sigmoid fn $a = \frac{1}{1+e^{-z}}$

$\therefore \boxed{\frac{\partial a}{\partial z} = a(1-a)} = \frac{1}{(1+e^{-z})}\cdot\left(1-\frac{1}{1+e^{-z}}\right)$

$\therefore dz = \left(\frac{-y}{a}+\frac{1-y}{1-a}\right)a(1-a) = \cancel{ay} - y + a - \cancel{ay} = a-y$

$\boxed{dz = a-y}$

$dw = \frac{\partial \mathcal{L}(a,y)}{\partial w} = \frac{\partial \mathcal{L}(a,y)}{\partial z}\cdot\frac{\partial z}{\partial w} = dz\cdot\frac{\partial z}{\partial w}$

As $z = w^T x + b \Rightarrow \boxed{\frac{\partial z}{\partial w} = x}$

$\therefore \boxed{dw = dz \cdot x}$

$db = \frac{\partial \mathcal{L}(a,y)}{\partial b} = \frac{\partial \mathcal{L}(a,y)}{\partial z} * \frac{\partial z}{\partial b} = dz \cdot \frac{\partial z}{\partial b}$

$\boxed{\frac{\partial z}{\partial b} = 1}$ $\therefore \boxed{db = dz}$

Gradients with 2 layer NN

- Computation Graph

$$\begin{array}{c} x,\ W^{[1]},\ b^{[1]} \\ \xleftarrow{dW^{[1]},\ db^{[1]}} \end{array} \to \boxed{z^{[1]} = W^{[1]}x + b^{[1]}} \to \boxed{a^{[1]} = \sigma(z^{[1]})} \to \boxed{z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}} \to \boxed{a^{[2]} = \sigma(z^{[2]})} \to \boxed{\mathcal{L}(a^{[2]}, y)}$$

$$W^{[2]} \xleftarrow{dW^{[2]}},\quad b^{[2]} \xleftarrow{db^{[2]}}$$

$$dz^{[1]} \leftarrow da^{[1]} \leftarrow dz^{[2]} \leftarrow da^{[2]} \leftarrow$$

$$da^{[2]} = \frac{\partial L(a,y)}{\partial a^{[2]}} = \frac{\partial}{\partial a^{[2]}}\left[-\left(y\log a^{[2]} + (1-y)\log(1-a^{[2]})\right)\right]$$

$$= -\frac{y}{a^{[2]}} - (1-y)\cdot\frac{1}{1-a^{[2]}}\cdot(-1)$$

$$da^{[2]} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

$$dz^{[2]} = \frac{\partial L(a,y)}{\partial z^{[2]}} = \frac{\partial L(a,y)}{\partial a^{[2]}} * \frac{\partial a^{[2]}}{\partial z^{[2]}} = da^{[2]} * \frac{\partial a^{[2]}}{\partial z^{[2]}}$$

$$a^{[2]} = \frac{1}{1+e^{-z^{[2]}}}$$

$$\frac{\partial a^{[2]}}{\partial z^{[2]}} = a^{[2]}(1 - a^{[2]})$$

$$\therefore dz^{[2]} = \left(\frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right) * a^{[2]}(1-a^{[2]})$$

dimensions of $z^{[2]}, a^{[2]}$, $dW^{[2]}, b^{[2]}$ etc

$$dz^{[2]} = -y + \cancel{y a^{[2]}} + a^{[2]} - \cancel{y a^{[2]}} = a^{[2]} - y$$

$$\boxed{dz^{[2]} = a^{[2]} - y}$$

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial W^{[2]}} = dz^{[2]} * \frac{\partial z^{[2]}}{\partial W^{[2]}}$$

as $Z = W a + b$

previously:

$\Rightarrow$ "$dw = dz \cdot x$"

$w^{[1]} = [\quad]$

$$\frac{\partial z^{[2]}}{\partial w^{[2]}} = a^{[1]}$$

$$dw^{[2]} = \frac{\partial L}{\partial w^{[2]}} = (a^{[2]} - y) * a^{[1]T}$$

dim ?

$$\boxed{dw^{[2]} = (a^{[2]} - y) * a^{[1]T}}$$

w is row vector so need Transpose in NN

But in logistic Regression no Transpose is required

→ dim of w & dw are same.

$$\frac{\partial L}{\partial b^{[2]}} = \frac{\partial L}{\partial z^{[2]}} * \frac{\partial z^{[2]}}{\partial b^{[2]}} = dz^{[2]} * \frac{\partial z^{[2]}}{\partial b^{[2]}}$$

$w^{[2]} = [- \; - \; -]$

$dw^{[2]} = [- \; - \; -]$

need Transpose for vector

not need for a single value.

$$\frac{\partial z^{[2]}}{\partial b^{[2]}} = \cancel{a} \; 1$$

$$db^{[2]} = \frac{\partial L}{\partial b^{[2]}} = dz^{[2]} * 1 = dz^{[2]}$$

where in case of

$dw = dz \cdot x$

dw is Coln vector $[\;]$

so no need of Transpose.

$$\boxed{db^{[2]} = dz^{[2]}}$$

$$da^{[1]} = \frac{\partial L}{\partial a^{[1]}} = \frac{\partial L}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} = dz^{[2]} * \frac{\partial z^{[2]}}{\partial a^{[1]}}$$

As $z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

$$\therefore \frac{\partial z^{[2]}}{\partial a^{[1]}} = W^{[2]}$$

$$\therefore da^{[1]} = dz^{[2]} * W^{[2]} = (a^{[2]} - y) W^{[2]}$$

$$\boxed{da^{[1]} = (a^{[2]} - y) \; w^{[2]}}$$

(32)

as $z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

$$\frac{\partial z^{[2]}}{\partial w^{[2]}} = a^{[1]}$$

Previously:

$\Rightarrow$ "$dw = dz \cdot x$"

$w^{[1]} = [\quad]$

$$\therefore dw^{[2]} = \frac{\partial L}{\partial w^{[2]}} = (a^{[2]} - y) \cancel{*} a^{[1]T}$$

dim ?

$$\boxed{dw^{[2]} = (a^{[2]} - y) * a^{[1]T}}$$

→ w is row vector so need Transpose in NN

But in logistic Regression no Transpose is required

→ dim of w & dw are same

$$\therefore \frac{\partial L}{\partial b^{[2]}} = \frac{\partial L}{\partial z^{[2]}} * \frac{\partial z^{[2]}}{\partial b^{[2]}} = dz^{[2]} * \frac{\partial z^{[2]}}{\partial b^{[2]}}$$

$$\frac{\partial z^{[2]}}{\partial b^{[2]}} = 1$$

$W^{[2]} = [- \; - \; -]$

$dw^{[2]} = [- \; - \; -]$

need Transpose for vector

not need for a single value

$$db^{[2]} = \frac{\partial L}{\partial b^{[2]}} = dz^{[2]} * 1 = dz^{[2]}$$

where in case of

$dw = dz \cdot x$

dw is Colm vector $[\;]$

so no need of Transpose

$$\boxed{db^{[2]} = dz^{[2]}}$$

$$da^{[1]} = \frac{\partial L}{\partial a^{[1]}} = \frac{\partial L}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} = dz^{[2]} * \frac{\partial z^{[2]}}{\partial a^{[1]}}$$

As $z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

$$\therefore \frac{\partial z^{[2]}}{\partial a^{[1]}} = W^{[2]}$$

$$\therefore da^{[1]} = dz^{[2]} * W^{[2]} = (a^{[2]} - y) W^{[2]}$$

$$\boxed{da^{[1]} = (a^{[2]} - y) \; w^{[2]}}$$

(33)

$$dz^{[1]} = \frac{\partial L}{\partial z^{[1]}} = \frac{\partial L}{\partial a^{[1]}} * \frac{\partial a^{[1]}}{\partial z^{[1]}} = da^{[1]} * \frac{\partial a^{[1]}}{\partial z^{[1]}}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$ Any Activation Function

$$\frac{\partial a^{[1]}}{\partial z^{[1]}} = g^{[1]'}(z^{[1]})$$

$$dz^{[1]} = da^{[1]} * g^{[1]'}(z^{[1]})$$

$$dz^{[1]} = dz^{[2]} W^{[2]} * g^{[1]'}(z^{[1]})$$?

$$\boxed{dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})}$$

$$dw^{[1]} = \frac{\partial L}{\partial w^{[1]}} = \frac{\partial L}{\partial z^{[1]}} * \frac{\partial z^{[1]}}{\partial w^{[1]}} = dz^{[1]} * \frac{\partial z^{[1]}}{\partial w^{[1]}}$$

As $z^{[1]} = W^{[1]} x + b^{[1]}$

$$\frac{\partial z^{[1]}}{\partial w^{[1]}} = x \quad \therefore \quad \boxed{dw^{[1]} = dz^{[1]} * x^T}$$

dim ?

$$\boxed{db^{[1]} = dz^{[1]}}$$