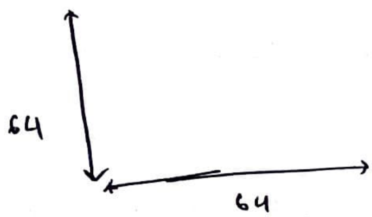
THROUGHOUT THESE NOTES, I'M USING SUBSCRIPT WHERE ANDREW IS USING SUPERSCRIPT ~~[illegible]~~ COZ I'M RACIST, ALSO, COPYRIGHT.

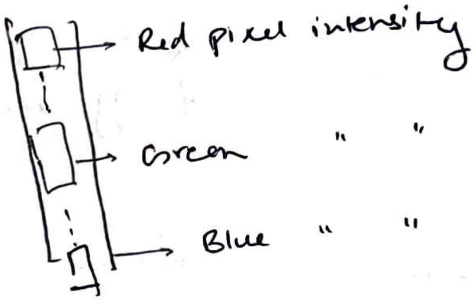
CNN Notes

Binary Classification

Eg: Image → 1 (cat) vs 0 (non cat)



⇒ 3 matrices (RBG) → unroll into feature vector

$x =$ 

Dimensions of x:

$(64 \times 64 \times 3) \times 1 \Rightarrow 12288 \times 1$

$n = n_x = 12288$

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

m training examples: $(x^1, y^1), (x^2, y^2), \ldots\ldots, (x^m, y^m)$

m_{train}, m_{test}

$\xleftarrow{\quad m \quad}$ → no. of training examples

Now define $X = \begin{bmatrix} | & | & | & & | \\ x_1 & x_2 & x_{3000} & \cdots & x_m \\ | & | & | & & | \end{bmatrix}$ $\updownarrow n_x$ - ies

matrix X is just all training examples stacked up in rows

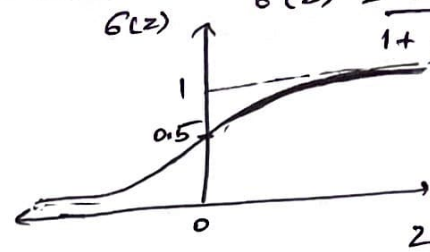
$Y = [\, y_1, y_2 \ldots\ldots y_m \,]$

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

$Y = \mathbb{R}^{1 \times m}$

$\sigma \to 0,\ z \to -\infty$
$\to 1,\ z \to \infty$

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



Logistic Regression

Given: x, you want $\hat{y} = \mathbb{P}(y=1 \mid x)$

$x \in \mathbb{R}^{n_x}$ parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output $\hat{y} = w^T x + b$ but this not have range $(0,1)$.

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

Our task is to have good w and b so that $\hat{y}$ is a very good estimation of y being 1

—

~~Logisti~~

Logistic Regression cost function.

Given: $\{(x_1, y_1) \cdots (x_m, y_m)\}$ we want $\hat{y}_i^{(i)} \approx y_i^{i}$

Loss (error) function:

$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ is reasonable but gradient descent goes bonkers.

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

If $y=1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$. You want small loss function so large $\hat{y}$ $\therefore \hat{y} \to 1$

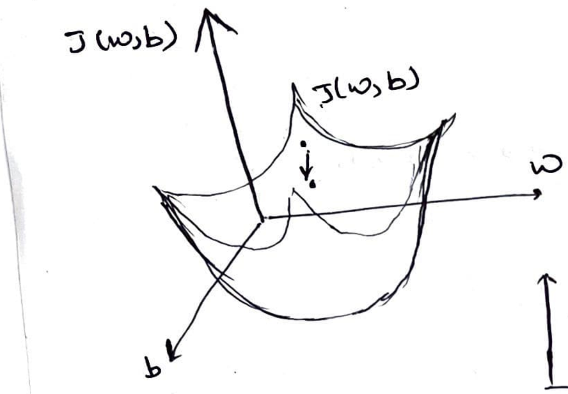
If $y=0$: .... $= \cancel{(1-y)} -\log(1-y) \to \hat{y}$ small, i.e. $\hat{y} \to 0$

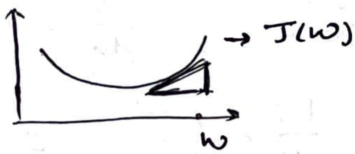
Cost function: $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}_i, y_i) = -\frac{1}{m} \sum_{i=1}^{m} (y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i))$

Now we need to find suitable w, b which minimizes value of cost function ~~J(w,b)~~ $J(w, b)$.

Gradient Descent:

Gradient descent takes a random ~~val~~ w, b and in iterations, moves in the steepest downward slope available.





Repeat {

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

}

Basically the intuition is just to keep approaching minima.

—

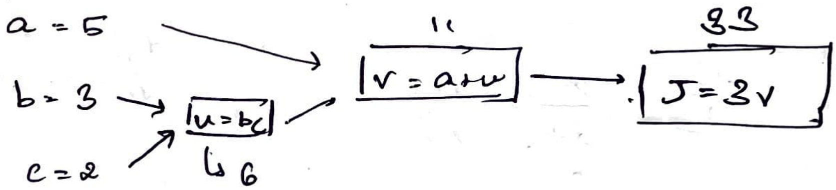
Then he explains derivatives for 17 minutes because American students are dumb

—

Computation Graph

$$J(a,b,c) = \underbrace{3(\underbrace{a+\underbrace{bc}_{u}}_{v})}_{J}$$

$u = bc$ $v = a+u$ $J = 3v$



COMPUTATION GRAPH : L → R gets output

You CAN GO BACKWARD TO GET DERIVATIVE.

$a = 5$, $b = 3$, $c = 2$

$b, c \rightarrow \boxed{u = bc} \rightarrow \boxed{v = a + u} \rightarrow \boxed{J = 3v}$, with $a \rightarrow \boxed{v = a + u}$

To find $\frac{dJ}{da}$, $\frac{dJ}{db}$, $\frac{dJ}{dc}$ ... use chain rule,

So reverse and find $\underbrace{\frac{dJ}{dv}}_{3} \times \underbrace{\frac{dv}{da}}_{1}$ $\quad \therefore \frac{dJ}{da} = 3$

Similarly $\frac{dJ}{du} = 3$, $\frac{dJ}{db} = \frac{dJ}{du} \times \frac{du}{db} = 3c$

Logistic Regression Gradient Descent

$\hat{y} = a$

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

$x_1, w_1, x_2, w_2, b \rightarrow \boxed{z = w_1x_1 + w_2x_2 + b} \rightarrow \boxed{a = \sigma(z)} \rightarrow \boxed{\mathcal{L}(a,y)}$

$\frac{dL}{dz} = \frac{dL}{da} \times \frac{da}{dz}$ $\qquad \frac{d\mathcal{L}}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$

$= a - y$

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

"dw_1" "dw_2" "db"

$w_1 = w_1 - \alpha\, dw_1$

$w_2 = w_2 - \alpha\, dw_2$

$b = b - \alpha\, db$

Gradient descent on m examples:

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

$$a^i = \hat{y}_i = \sigma(z_i) = \sigma(w^T x_i + b)$$

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} \mathcal{L}(a_i, y_i)}_{dw_i \;\rightarrow\; (x_i,\, y_i)}$$

for $i = 1$ to m

$z_i = w^T x_i + b$

$a_i = \sigma(z_i)$

$J \mathrel{+}= -[y_i \log a_i + (1-y_i)\log(1-a_i)]$

$\frac{dJ}{dz_i} = a_i - y_i$

$\overbrace{\frac{dJ}{dw_1} \mathrel{+}= x_{1i}\, dz_i \qquad \frac{dJ}{dw_2} \mathrel{+}= x_{2i}\, dz_i}^{\text{for } n=2} \qquad db \mathrel{+}= dz_i$

.... dw_n

$w_1 = w_1 - \alpha \frac{dJ}{dw_1}$; $w_2 = w_2 - \alpha \frac{dJ}{dw_2}$; $w_3 = w_3 - \alpha \frac{dJ}{dw_3}$

fe

6 ✓

8f –

2

Vectorization

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

Vectorizing Logistic Regression.

Mon

Training examples

$z_1 = w^T x^1 + b \qquad z_2 = w^T x_2 + b \qquad z_3 = w^T x_3 + b$

$a_1 = \sigma(z_1) \qquad a_2 = \sigma(z_2) \qquad a_3 = \sigma(z_3)$

$$X = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix} \quad (n_x, m)$$

$$[z_1 z_2 \cdots z_m] = w^T X + \underbrace{[b\ b\ b \cdots b]}_{1 \times m}$$

$$Z \rightarrow [z_1 z_2 \cdots z_m] = [w_T x_1 + b \quad w_T x_2 + b \ \cdots \ w_T x_m + b]$$

$Z = np.dot(w.T, X) + b$ → makes an appropriate matrice with all elements b.

$A = [a_1\ a_2 \ \cdots \ a_m] = \sigma(z)$.

$dz_1 = a_1 - y_1 \quad dz_2 = a_2 - y_2 \quad \ldots$

$dZ = [dz_1 \; dz_2 \; \ldots \; dz_m]$.

$1 \times m$.

$A = [a_1 \ldots a_m] \quad Y = [y^1 \ldots y^m]$.

$dZ = A - Y. = [a_1 - y_1 \quad a_2 - y_2 \quad \ldots \quad a_m - y_m]$

Removed first for loop.

$dw = 0$

$dw += x_1 dz_1$

$dw += x_2 dz_2$

$\vdots$

$dw /= m$

$db = 0$

$db += dz_1$

$db += dz_2$

$\vdots$

$db /= m$.

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

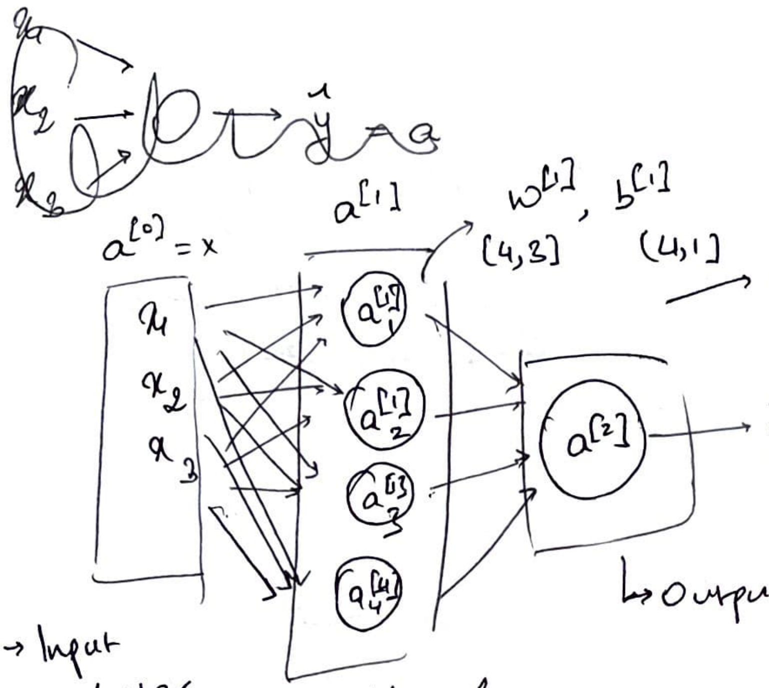
$$dw = \frac{1}{m} X \, dZ^T$$

$$= \frac{1}{m} [\, x_1 dz_1 + \ldots x_m dz_m]$$

$n \times m$

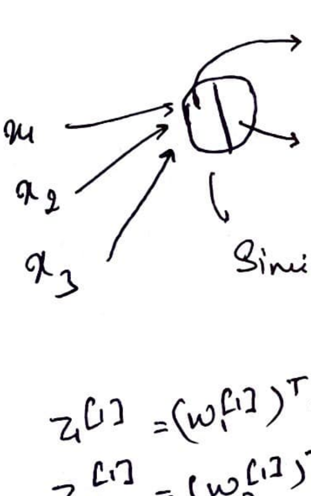
(I)

What are Neural Networks?



$W^{[1]}, b^{[1]}$
$(4,3)$ $(4,1)$

→ 2 Layer Neural Network.



$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$

$a_1^{[1]} = \sigma(z_1^{[1]})$

$a_i^{[l]}$ → layer
↳ node in layer

Similarly 4 nodes

$$z_1^{[1]} = (w_1^{[1]})^T x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = (w_2^{[1]})^T x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = (w_3^{[1]})^T x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = (w_4^{[1]})^T x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

writing a for loop here is inneficient.

$$\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

II

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

⇒ Given input x

$$z^{[1]} = W^{[1]} \underbrace{a^{[0]}}_{x} + b^{[1]} \qquad a^{[1]} = \sigma(z^{[1]})$$

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

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

Multiple ~~examp~~ training examples.

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

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

$\vdots$ m training examples.

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

for $i = 1$ to m

$$\left.\begin{aligned} z^{[1](i)} &= W^{[1]} x^{i} + b^{[1]} \\ a^{[1](i)} &= \sigma(z^{[1](i)}) \\ z^{[2](i)} &= W^{[2]} a^{[1](i)} + b^{[2]} \\ a^{[2](i)} &= \sigma(z^{[2](i)}) \end{aligned}\right\}$$ add (i) to all training examples.

$$X = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix} \quad (n_x, m)$$

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

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

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

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

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

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

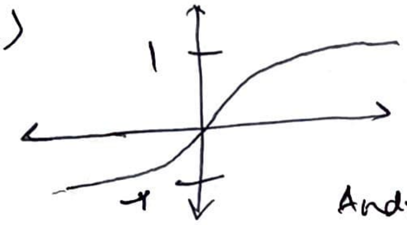
III

Activation function:

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

sigmoid here is called the activation function.

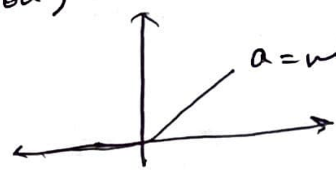
More activation functions

1) 

$a = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$: generally much better ~~so that~~ coz mean $= 0$

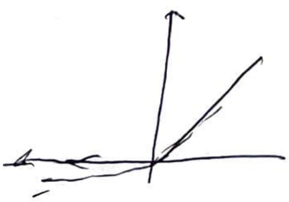
Andrew loves tanh and says it is much more superior except in final layer as sigmoid fn can be interpreted as a probability.

2) Rectified linear unit (ReLU function)



$a = \max(0, z)$

Generally: In Binary classification systems, sigmoid is preferred. and ~~the~~ if you don't know what to do, use ReLU.



} → Leaky ReLU ($a = \max(0.01z, z)$ or $(0.001z, z)$ you get the hint.

IV

Gradient descent for neural networks:

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ — $n_x = n^{[0]}, n^{[1]}, n^{[2]} = 1$

$w^{[1]} \rightarrow (n^{[1]}, n^{[0]})$, $b^{[1]} \rightarrow (n^{[1]}, 1)$, $w^{[2]} \rightarrow (n^{[2]}, n^{[1]})$, $b^{[2]} \rightarrow (n^{[2]}, 1)$

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

Gradient descent:

For i = 1 to m:

Compute: $(\hat{y}^{(i)}, i = 1, \dots m)$

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, ...

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

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

Random Initialization

w(weights) needs to be initialized randomly coz lets say

$w^{[1]}$ = zeroes $a_1^{[1]} = a_2^{[1]}$ $b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

In this case $a_1^{[1]} = a_2^{[1]}$ $dz_1^{[1]} = dz_2^{[1]}$

And no matter how much you update, you'll keep computing the exact same function.

∴ $w^{[1]}$ = np.random.randn((2,2)) * 0.01

$b^{[1]}$ = zeroes

$w^{[2]}$ = np.random. * 0.01

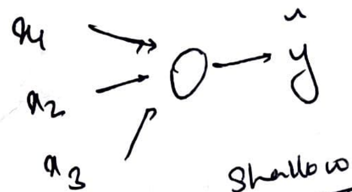
$b^{[2]}$ = zeroes

you keep weights small so sigmoid/tanh doesn't tend to 1 or 0

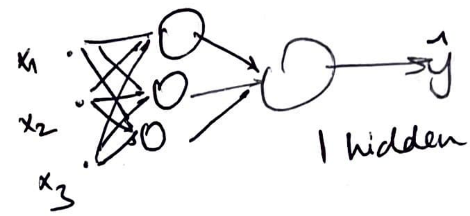
(i)

Week 4

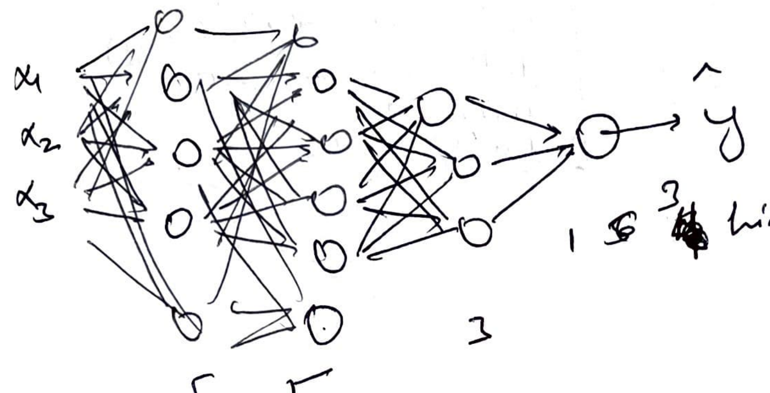
DEEP LAYER NEURAL NETWORK



Shallow



1 hidden layer



"deep"

1 ~~5~~ 3 hidden layer

$n^{[1]} = 5$, $n^{[2]} = 5$, $n^{[3]} = 3$, $n^{[4]} = n^{[L]} = 1$

4 Layer NN

$a^{[l]}$ = activation = $g(z^{[l]})$, $w^{[l]}$ = weights for $z^{[l]}$

$b^{[l]}$ = bias for $z^{[L]}$

Forward Propagation

x: $z^{[1]} = w^{[1]}x + b^{[1]}$; $a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$; $a^{[2]} = g^{[2]}(z^{[2]})$

:

$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$; $a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$

Generic: $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$

$a^{[l]} = g^{[l]}(z^{[l]})$

Vectorized: $Z^{[1]} = w^{[1]}A^{[0]} + b^{[1]}$

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

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

$\hat{Y} = g(z^{[4]}) = A^{[4]}$

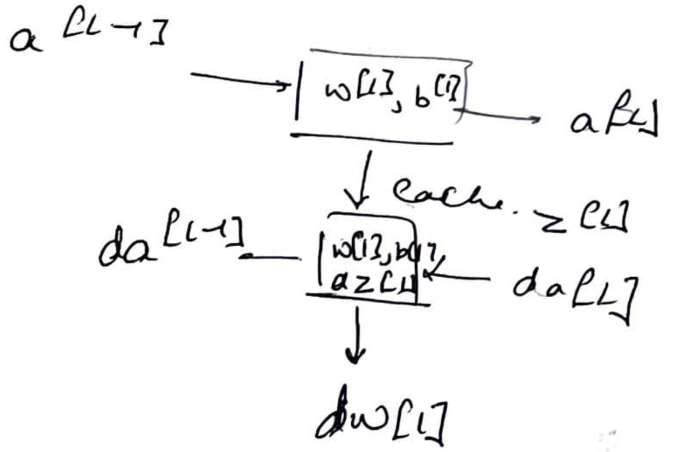
For $l = 1 \ldots 4$

There is no way to remove this for loop

(ii)

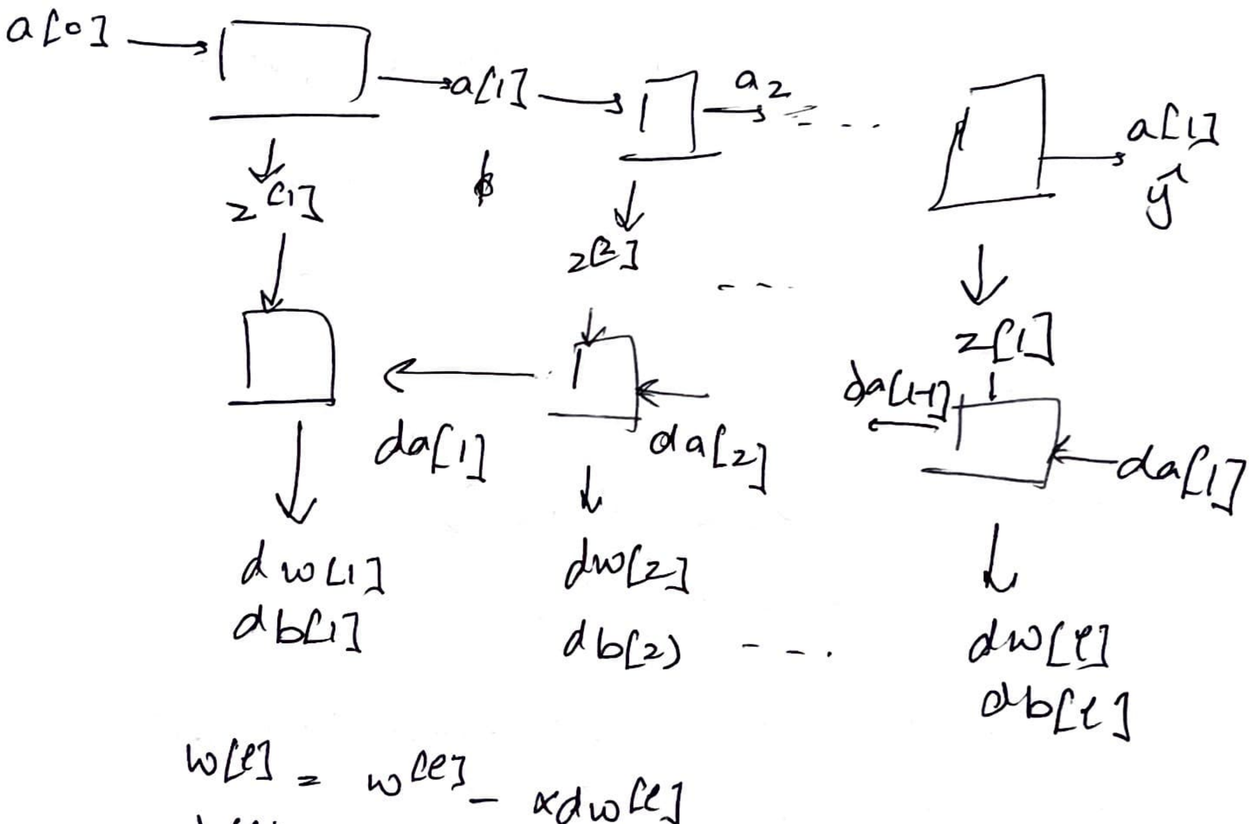
Matrice dimansion → get them right.

backward functions.



element wise multiplica

$$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$$
$$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]T}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$$



$$w^{[l]} = w^{[l]} - \alpha dw^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

(iii)

Hyper Parameters :

Parameters : $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]} \ldots$

Hyperparameters : Learning rate α

\# iterations, # hidden layers L, # hidden units $n^{[1]}, n^{[2]} \ldots$

choice of activation function.

Later : Momentum, mini batch size, etc.

Gradient descent for neural networks:

Parameters: $w^{[1]}$, $b^{[1]}$, $w^{[2]}$, $b^{[2]}$ $\qquad n_x = n^{[0]}, n^{[1]}, n^{[2]} = 1$

$w^{[1]} \to (n^{[1]}, n^{[0]})$, $b^{[1]} \to (n^{[1]}, 1)$, $w^{[2]} \to (n^{[2]}, n^{[1]})$, $b^{[2]} \to (n^{[2]}, 1)$

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

Gradient descent:

for $i=1$ to m:

Compute: $(\hat{y}^{(i)}, i=1, \dots m)$

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, ...

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

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

Random Initialization

w (weights) needs to be initialized randomly cos lets say

$w^{[1]}$ = zeroes $a_1^{[1]} = a_2^{[1]}$ $\qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

In this case $a_1^{[1]} = a_2^{[1]}$ $\quad dz_1^{[1]} = dz_2^{[1]}$

And no matter how much you update, you'll keep computing the exact same function.

∴ $w^{[1]}$ = np.random.randn((2,2)) * 0.01

$b^{[1]}$ = zeroes

$w^{[2]}$ = np.random. * 0.01

$b^{[2]}$ = zeroes

You keep weights small so sigmoid/tanh doesn't tend to 1 or 0