Computing Graph: For logistic Regression.

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

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

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

One traing example.

$$\begin{matrix} x_1 \\ w_1 \\ x_2 \\ w_2 \\ b \end{matrix} \rightarrow z = w_1 x_1 + w_2 x_2 + b \rightarrow a = \sigma(z) \rightarrow \mathcal{L}(a, y)$$

reduce $\mathcal{L}$ modifying w & b.

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

$$\text{"}da\text{"} = \frac{d\mathcal{L}(a, y)}{da}$$

"da" in code.

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

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

logistic example.

$$= a - y \qquad a(1-a)$$

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

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

$$= -(\cancel{y} - ay) + a - ya$$

$$= a - y$$

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

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

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

Going backwards ~~to~~ to find out how much you need to change "dw1", "dw2" & "db" to ~~find out~~ change Loss func a little bit.

Backward propagation.

Logistic Regression on m examples.

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

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

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

$dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

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

"$dw_1^{(i)}$" → shown on single training example.

↓

overall cost function derivative.

*. Procedure :

$J=0,\ dw_1=0,\ dw_2=0,\ db=0.$

For $i=1$ to m → first for loop

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

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

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

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

$dw_j += x_j^{(i)} dz^{(i)}$

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

accumulators

refer to only one single training example.

For loop in calculating $w_1, w_2 \cdots w_n$.

→ second for loop.

↓

mitigated by vectorization to reduce for loop.

$J /= m$

$dw_1 /= m$

$dw_2 /= m$

$\vdots$

$db /= m$

get final answer.

$$\left.\begin{aligned} w_1 &:= w_1 - \alpha\, dw_1 \\ w_2 &:= w_2 - \alpha\, dw_2 \\ &\vdots \\ b &:= b - \alpha\, db \end{aligned}\right\}.$$

Vectorization (numpy lib)

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

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

Non-Vec :

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

$z += b$.

slow.

Vectorized:

$z = $ np.dot(w.x) + b.

↓

$w^T x$.

GPU } SIMD - single instruction multiple data.
CPU }

good at.

Guideline:

avoid for loop whenever necessary.

$u = Av$

$u_i = \sum_j A_{ij} v_j$.

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

for i ...
  for j ....
    $u[i] += A[i][j] * v[j]$

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

every time see if it is possible to call a numpy command instead of using for loop.

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

$u = $ np.zeros((n,1))
for i in range(n) :

import numpy as np

$u = $ np.exp(v)
np.log()
np.abs()
np.maximum(v, 0,

① $dw_1 = 0$, $dw_2 = 0$, ⋮ $\rightarrow$ dw = np.zeros((n_x, 1))

for $j = 1 \dots n_x$

dw_j += … (gets updated)

gets optimised by

$dw_j += x_j^{(i)} dz^{(i)}$

② dw += $x^{(i)} dz^{(i)}$ $\rightarrow$ using np command.

③ $dw_1 = dw_1/m \dots$

$\rightarrow dw = dw/m$.

For m trains example:

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

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

$z^{(2)} = \dots$

$z^{(3)} \dots$

carry out forward propagation simultaneously with m examples without for loop.

$\rightarrow$ matrices

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

$\underbrace{\qquad}_{m}$

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

$[a_1, a_2 \dots a_m] = \sigma(Z)$

how to 'sigmoid' all the vectorized Z. shown in assignment. ?

↳ using numpy command np.exp().

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

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

$\uparrow (1,1) \rightarrow$ broadcasting

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

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

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

$Y = [y^{(1)} \ldots y^{(m)}]$

$dZ = A - Y = [a^{(1)} - y^{(1)}, \ldots]$.

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

$= \frac{1}{m} np.sum(dZ)$ ✓

$dw = \frac{1}{m} X dZ^T$ ✓

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

$= \frac{1}{m} [\underbrace{x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)}}_{n \times 1}]$

Single step

$Z = w^T X + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

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

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

still need a for loop to do multiple iterations

Broadcasting example:

① cal = A. sum(axis = 0)

want to sum up vertically

horizontally → axis = 1

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

cheep to call

always confirm if not sure.

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

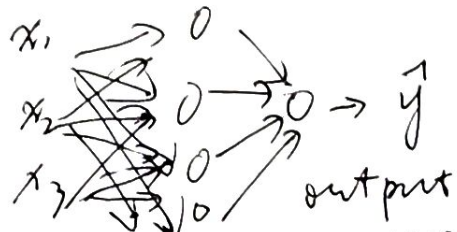
③ $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + [100 \ 200 \ 300]$

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

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

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

Neural Network Representation



output layer

2L NN.

do not count input layer.

Input layer.

Hidden layer

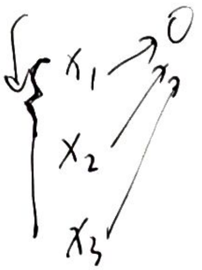
actual value of this layer is not observed.

Input

Input activation $\rightarrow \boxed{a^{[0]} = X}$

Hidden : $a_1^{[1]}\ a_2^{[1]} \cdots$

Given



$z_1^{[1]} = w_1^{[1]T} X + b^{[1]}$

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

$a_i^{[l]}$ $\leftarrow$ $[l]$ $\leftarrow$ Layer. $i \Leftarrow$ node in Layer.

$z^{[1]}$

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ \vdots \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$

(3.1)

4×3

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

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

$\Downarrow$ 2L NN

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

(1.1) $\quad [1.4]$ $\quad [4.1]$ $\quad [1.1]$

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

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

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

$\vdots$

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

layer 2 $\quad$ example i

For i from 1 to m :

$\vdots$

vectorized to avoid for loop for training examples

$z^{[1](1)} = w^{[1]}x^{(1)} + b^{[1]}$, $z^{[1](2)} = w^{[1]}x^{(2)} + b^{[1]}$, $z^{[1](3)} = w^{[1]}x^{(3)} + b^{[1]}$

$$w^{[1]} = \begin{bmatrix} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{bmatrix}. \quad w^{[1]}x^{(1)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}, \quad w^{[1]}x^{(2)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix} \quad w^{[1]}x^{(3)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

$$w^{[1]}X = w^{[1]} \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

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

$$A = \sigma(Z) \rightarrow A \left.\underbrace{\begin{bmatrix} | & | & \\ a^{[1](1)} & a^{[1](2)} & \cdots \\ | & | & \end{bmatrix}}_{\text{training example.}}\right\} \text{hidden units in 1st Layer}$$

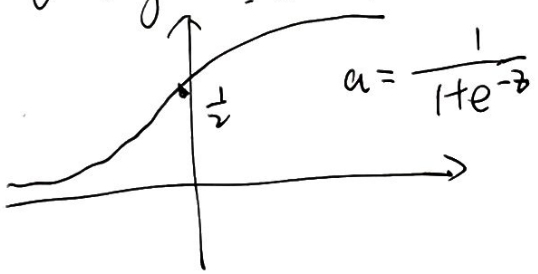
$$* \quad Z^{[i]} = W^{[i]}A^{[i-1]} + b^{[i]}$$

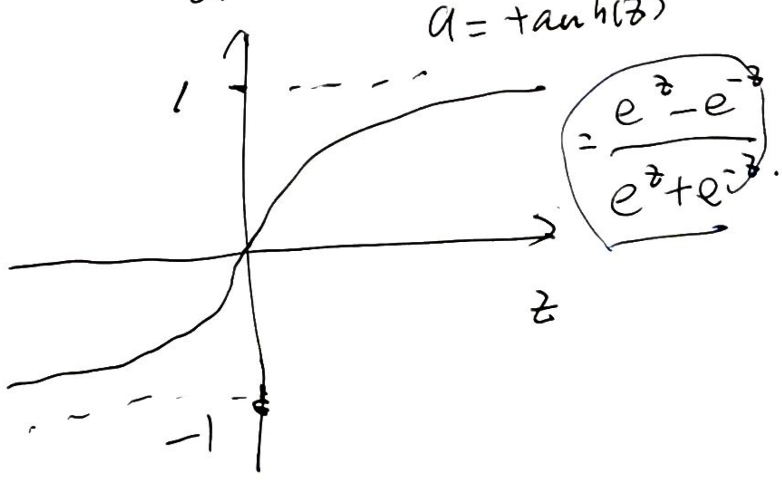
Forward propagation fully vectorized.

$$\begin{cases} A^{[0]} = X \quad (\text{input Layer}) \\ a^{[0](i)} = x^{(i)} \end{cases}$$

Activation Function:

σ sigmoid:





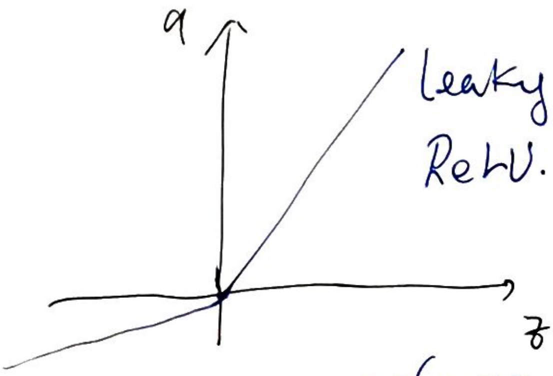
Tanh function almost always outperforms sigmoid function since. it has 0 mean makes learning for 2nd Layer easier.

Sigmoid function → used for binary classification of

it requires the probability. so it is used for output Layer

Different function for different Layer.

$g^{[i]}$ ← Layer.



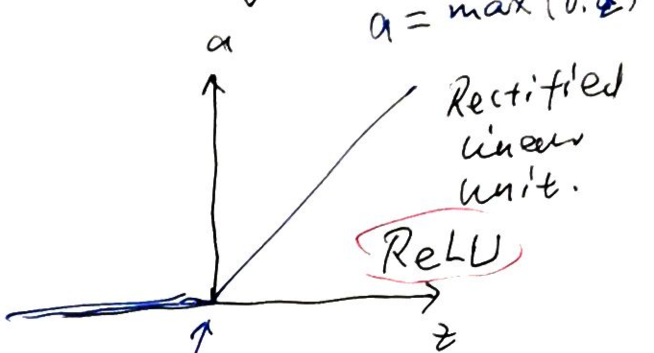
$a = \max(0.01z, z)$

ReLU: less effect the slope really small when going to zero → usually much faster than tanh & sigmoid

common drawback:

slope is small, z is either very large or very small

⇓

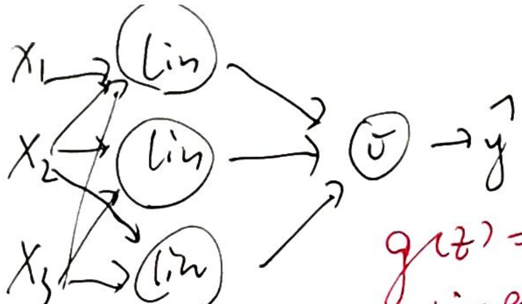


$z = 0.000\ldots$

pretend the derivative when $z = 0$ is either 0 or 1. works just fine.

Increasingly default activation function. (Hidden Layer)

If not sure what to use. ReLU is a great default choice.



$g(z) = z$
linear activation func.

$\Downarrow$

no more expressive than a standard Logistic Regression.

linear activation func always calculate the same → Layers does not have their use all → linear function.

$\downarrow$

one use: doing linear regression on housing price, etc.

$\Downarrow$

use linear function for output layer.

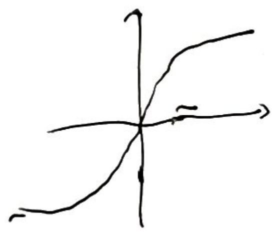
$$\begin{cases} z \to \infty, & g'(z) \to 0 \\ z \to -\infty, & g'(z) \to 0 \\ z = 0, & g'(z) = \frac{1}{2} \times \frac{1}{2} = \frac{1}{4} \end{cases}$$

$$a' = a(1-a)$$

Tanh activation func.

$$g(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}.$$

$$g'(z) = 1 - (\tanh(z))^2 = 1 - g^2(z)$$



$$a' = 1 - a^2$$

$$\begin{cases} g'(z) \to 0. \text{ when } z \to \infty \\ g'(z) \to 0, \text{ when } z \to -\infty \\ z = 0 \to g'(z) = 1 \end{cases}$$

For sigmoid func.

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

$$g'(z) = g(z)(1 - g(z))$$

(leaky) ReLU:

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undef} & z = 0. \end{cases}$$

$$g'(z) = \begin{cases} 1 & z > 0 \\ 0.01 & z < 0 \\ \dots \end{cases}$$

Gradient Descent for One Hidden layer NN.

2 L NN.

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

$$= dz^{[2]} \cdot W^{[2]} \cdot g'(z)$$

Paras: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$

cost func: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum \mathcal{L}(\hat{y}, y)$

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

Gradient descent: d.

troublesome.

Repeat: {

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

$dw^{[1]} = \frac{\partial J}{\partial w^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, ...

$W^{[1]} = W^{[1]} - \alpha dW^{[1]}$

⋮

} until converging.

Vectorized:

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

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

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

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

element wise product.

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

$db^{[1]} =$ np.sum($dZ^{[1]}$, axis=1, keepdims = T

Forward. & Backward Graph: $w^{[2]}\ b^{[2]}$.

similar to logistics.

$x, w^{[1]}, b^{[1]} \rightarrow z^{[1]} = W^{[1]}x + b^{[1]} \rightarrow a^{[1]} = \sigma(z^{[1]}) \rightarrow z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \rightarrow a^{[2]} = \sigma(z^{[2]}) \rightarrow \mathcal{L}(\hat{y}, y)$

$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$

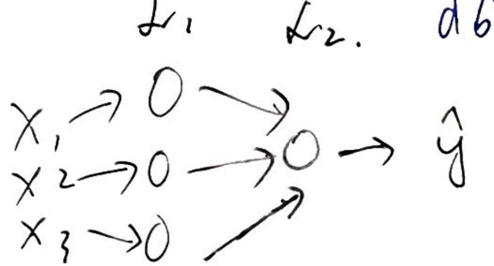
$dz^{[2]} = a^{[2]} - y$

$dW^{[2]} = a^{[1]T} \cdot dz^{[2]}$

$db^{[2]} = dz^{[2]}$.

$dW^{[1]} = dz^{[1]} \cdot X^T$

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



If weights are initialised as zero.

$w^{[1]}_R = \begin{bmatrix}0&0\\0&0\end{bmatrix}$. $dw = \begin{bmatrix}u&v\\u&v\end{bmatrix}$

no matter how many neurons are in the layer. they are essentially computing the same thing. → as one neuron.

$w^{[1]} = \begin{bmatrix}---\\---\end{bmatrix}\Big\}$ equal.

works for the shallow neural network.

deep one different meth.

Initial with small value.

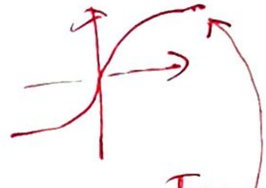
Random Initialisation:

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

$b^{[1]}$ = np.zeros((2,1)) → no symetric problem

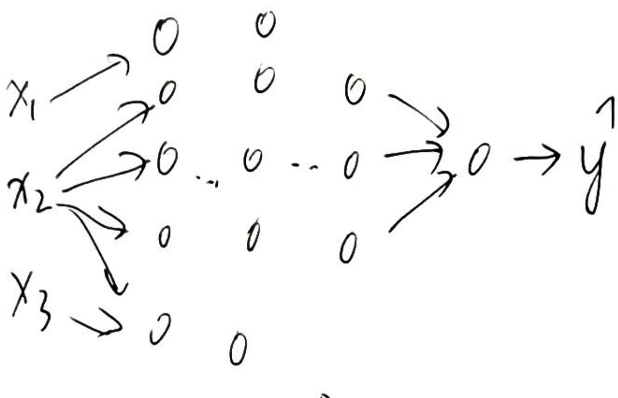
$w^{[2]}$ = ... rand.. ((1,2)) * 0.01 ...

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



if its Too big learning will be very slow.

Deep Neural Network.



$L=4$ (# layer)

$n^{[l]}$ # units in layer L.

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

$a^{[l]}$ → activation in layer l.

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

$W^{[l]}$ weights for $z^{[l]}$.

$n^{[0]} = n_x = 3$.

* Forward propagation in deep NN

$$\begin{cases} Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]} \\ A^{[l]} = g^{[l]}(Z^{[l]}) \end{cases}$$

$A^{[0]} = X$ train example.

$$Z^{[l]} = \Big[\, Z^{[l](1)} \quad Z^{[l](2)} \quad \cdots \quad Z^{[l](m)} \,\Big]$$ node in each layer.

For loop for different layer is perfectly suitable.

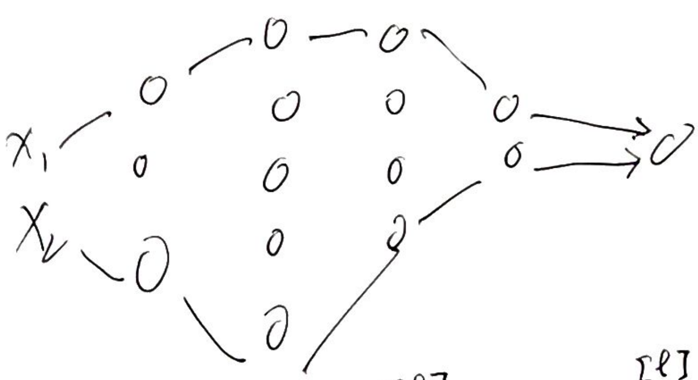
("not avoidable" — Andrew Ng).

* $\boxed{\text{Dimensions Check}}$.

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

$(3,1)$. $\quad (3,2)\ (2,1) \quad [3,1]$

$(3,2) \to (n^{[1]}, n^{[0]})$.



$$W^{[l]} \Rightarrow (n^{[l]}, n^{[l-1]})$$

$$\begin{cases} b^{[l]} : (n^{[l]}, 1) \\ dW^{[l]} : (n^{[l]}, n^{[l-1]}) \\ db^{[l]} : (n^{[l]}, 1) \end{cases}$$

Z & a — same dimension

↓

vectorized implementation

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

$(n^1, m) \quad (n^{[1]}, n^{[0]})(n^{[0]}, m)$

$b^{[1]}$: $(n^{[1]}, 1)$ ↓ bd. $(n^{[1]}, m)$ broadcast

pic of a human → find edges → parts of faces

{horizontal, vertical, ⋮} {eyes, noses, ⋮}

→ different type of faces.

compose hierarchically.

another eg: sound recog.

Some functions require exponentially more hidden ~~layers~~ units if layers are limited.

x_1 XOR x_2 XOR x_3 ... XOR x_n

Circuit theory

x_1, x_2 > XOR > ... ($\log n$)

x_3 > XOR

→ elongate horizontally

$x_1, x_2, \ldots, x_n$ → o → y, 2^n

Backward functions.

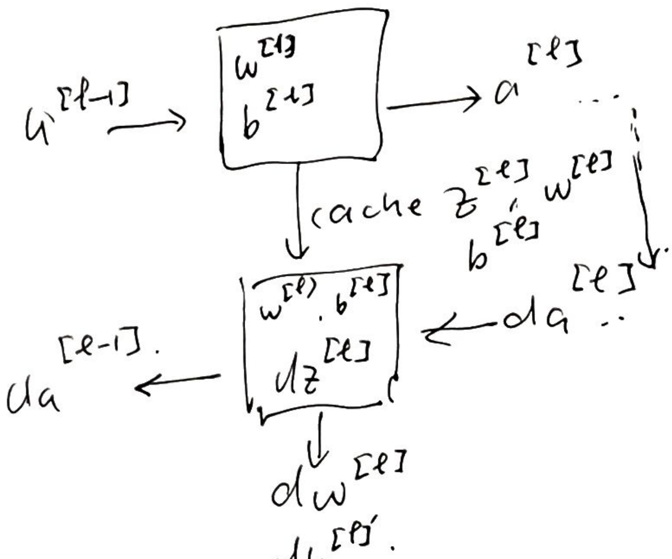
Input: $da^{[l]}$.

cache ($z^{[l]}$)

output: $da^{[l-1]}$

$dw^{[l]}$, $db^{[l]}$.

layer l.



$$\begin{cases} w^{[l]} = w^{[l]} - \alpha\, dw^{[l]} \\ b^{[l]} = b^{[l]} - \alpha\, db^{[l]} \end{cases}$$

Backward Propagation for layer l.

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

$$dz^{[l]} = w^{[l+1]T} dz^{[l+1]} * g^{[l]'}(z^{[l]})$$

Parameters & Hyperparameters

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

Hyperparameters:

- learning rate α
- # iterations
- # hidden layers L
- # hidden units. $n^{[1]}, n^{[2]} \ldots$
- choice of activation func.
- many more ...

empirical process

try to tune the parameters.

Hyperpara Tune might be different in different industry.