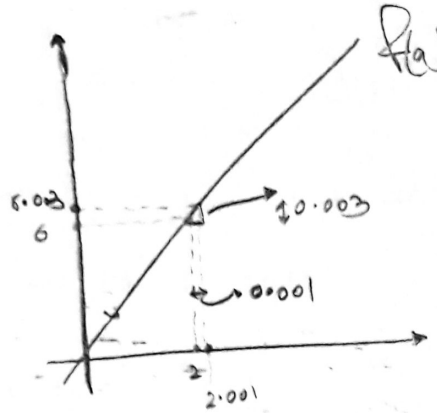
Intuition about derivatives



$f(a) = 3a$

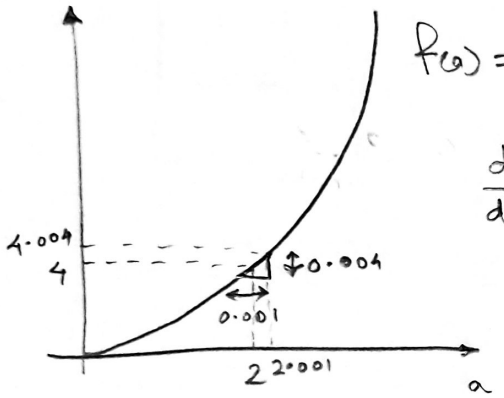
$a = 2 \rightarrow f(a) = 6$

$a = 2.001 \rightarrow f(a) = 6.003$

slope (derivative) of $f(a)$ at $a = 2$ is 3

$$\frac{\text{height}}{\text{width}} = \frac{0.003}{0.001} = 3$$

$$\frac{df(a)}{da} = 3$$



$f(a) = a^2$

$$\frac{df(a)}{da} = 2a$$

$a = 2$ $\quad f(a) = 4$

$a = 2.001$ $\quad f(a) \approx 4.004$ (reality: 4.004001)

Slope (derivative) of $f(a)$ at $a = 2$ is 4.

$2a$ → $\frac{0.004}{0.001}$

$a = 5$ $\quad f(a) = 25$

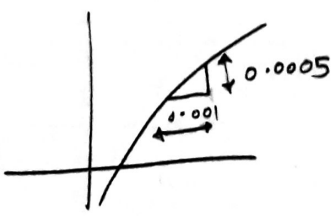
$a = 5.001$ $\quad f(a) = 25.010$

Slope (derivative) of $f(a)$ at $a = 5$ is 10

$2a$ → $\frac{0.010}{0.001}$

$f(a) = \log_e(a) = \ln(a)$

$$\frac{df(a)}{da} = \frac{1}{a}$$



$a = 2$ $\quad f(a) \approx 0.69315$

$a = 2.001$ $\quad f(a) \approx 0.69365$

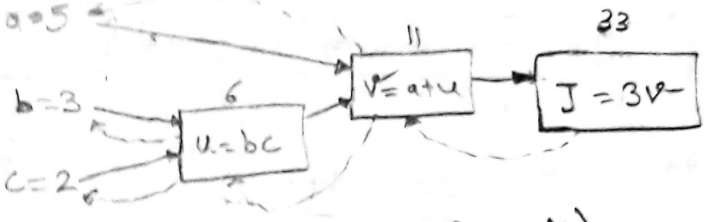
$\frac{1}{a}$

Computation graph

$J(a,b,c) = 3(a + \underbrace{bc}_{u})_{v} = 3(5 + 3\times 2) = 33$

J

eg:-



$u = bc$

$v = a + u$

$J = 3v$

(right to left) ← for derivatives (to minimize the cost function)

$\frac{dJ}{dv}$ change in J when v is nudged slightly

i.e. $J = 3v$

$v = 11 \rightarrow 11.001$, $J = 33 \rightarrow 33.003$ $\Big\}$ $\frac{0.003}{0.001} = 3$

$\therefore \frac{dJ}{dv} = 3$

$\frac{dJ}{da} = 3 = \frac{dJ}{dv} \cdot \underbrace{\frac{dv}{da}}_{\frac{dv}{da} = 1} = 3 \cdot 1 = 3$

$\frac{d \text{ final output var}}{d \text{ var}}$ can be written as "dvar" in code (to simplify things as we always have d final output/d)

for example:- $\frac{dJ}{dv}$ can be "dv" = 3

$\frac{dJ}{du} = \underbrace{\frac{dJ}{dv}}_{3} \cdot \underbrace{\frac{dv}{du}}_{1} = 3$

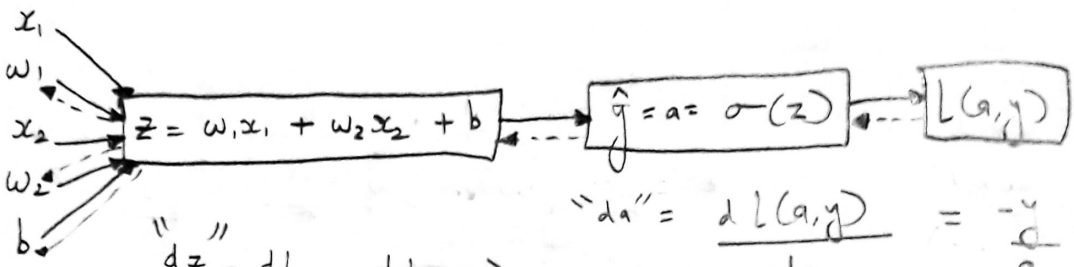
$\frac{dJ}{db} = \underbrace{\frac{dJ}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{db}}_{c \text{ (which is 2)}} = 6$ $\quad \frac{dJ}{dc} = 9$

Logistic Regression Derivatives

$z = w^T x + b$

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

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



"da" $= \dfrac{dL(a,y)}{da} = -\dfrac{y}{a} + \dfrac{1-y}{1-a}$

"dz" $= \dfrac{dL}{dz} = \dfrac{dL(a,y)}{dz}$

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

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

$da = \dfrac{e^{-z}}{(1+e^{-z})^2} dz$

$\dfrac{da}{dz} = \dfrac{e^{-z}}{(1+e^{-z})^2} = a^2 \cdot e^{-z}$

but $\left(1 + e^{-z} = \dfrac{1}{a} \right.$

$\left. e^{-z} = \dfrac{1}{a} - 1 = \dfrac{1-a}{a} \right)$

$\dfrac{da}{dz} = a^2 \cdot \dfrac{(1-a)}{a} = a \cdot (1-a)$

"dz" $= \dfrac{dL}{da} \cdot \dfrac{da}{dz} = \left(\dfrac{-y}{a} + \dfrac{1-y}{1-a} \right) a \cdot (1-a)$

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

$= -y + ya + a - ya$

"dz" $= -y + a = \underline{a - y}$

$\frac{dL}{dw_1} = "dw_1" = x_1 \cdot dz"$ $\left(\frac{dL}{da} \cdot \frac{da}{dz} \cdot \frac{dz}{dw_1} = (a-y) \cdot x_1 = x_1 \cdot dz\right)$

$"dw_2" = x_2 \cdot dz"$

$"db" = "dz"$

one step of gradient descent:

$w_1 = w_1 - \alpha\, dw_1$

$w_2 = w_2 - \alpha\, dw_2$

$b = b - \alpha\, db$

Logistic regression on m examples

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

(1st for loop) for i=1 to m:

$z^{(i)} = w^T \cdot x^{(i)} + b$ → dot product

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

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

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

(2nd for loop)

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

$dw_2 \mathrel{+}= x_2^{(i)} \cdot dz^{(i)}$ ↕ here n=2

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

dw_3

dw_n

$J /= m;$

$dw_1 /= m;$

$dw_2 /= m;$

$db /= m;$

} average over all of m

$w_1 = w_1 - \alpha\, dw_1$

$w_2 = w_2 - \alpha\, dw_2$

$b = b - \alpha\, db$

} gradient descent

for next step

To speed up the "for" loop implementation we use Vectorization techniques.

Logistic regression – removing 1 for loop

$J = 0$, $dw = np.zeros((n_x, 1))$, $db = 0$

for $i = 1$ to m:

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

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

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

$$dz^{(i)} = a^{(i)}(1 - a^{(i)})$$

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

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

$J = J/m$

$dw = dw/m$

$db = db/m$

Can be further reduced as:—

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

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

$$= w^T \cdot X + b$$

Can be written as → $Z = np.dot(w.T, X) + b$

$$A = [a^{(1)} \; a^{(2)} \cdots a^{(m)}] = \sigma(Z)$$

$z = w^T x + b$

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

can be written as (without a for loop)

$A = \sigma(z)$

$dz = A - Y$

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

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

In python

axis = 0

axis = 1

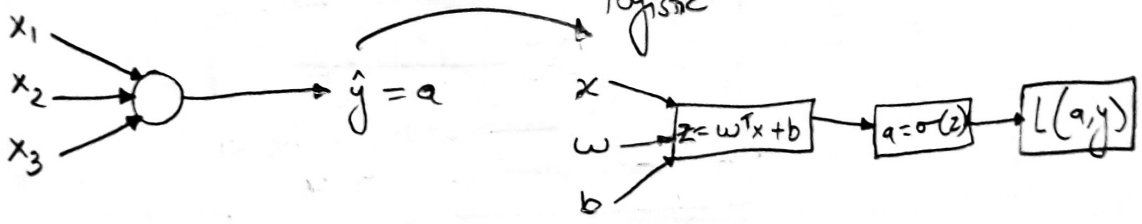
Don't use a rank 1 array in python

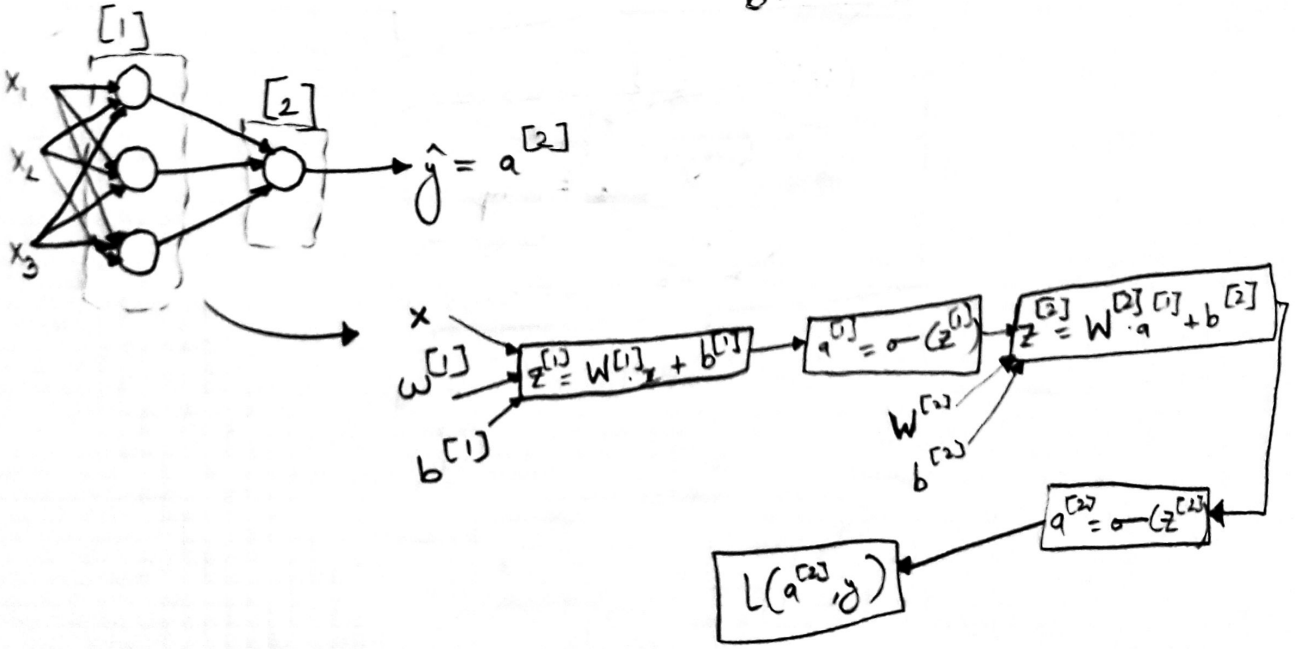
$a = np.array([1, 2, 3, 4])$ shape = (4,)

$a = np.array([[1, 2, 3, 4]])$

Use nx1 or 1xn matrices

What is a neural network?





In conventional usage :-

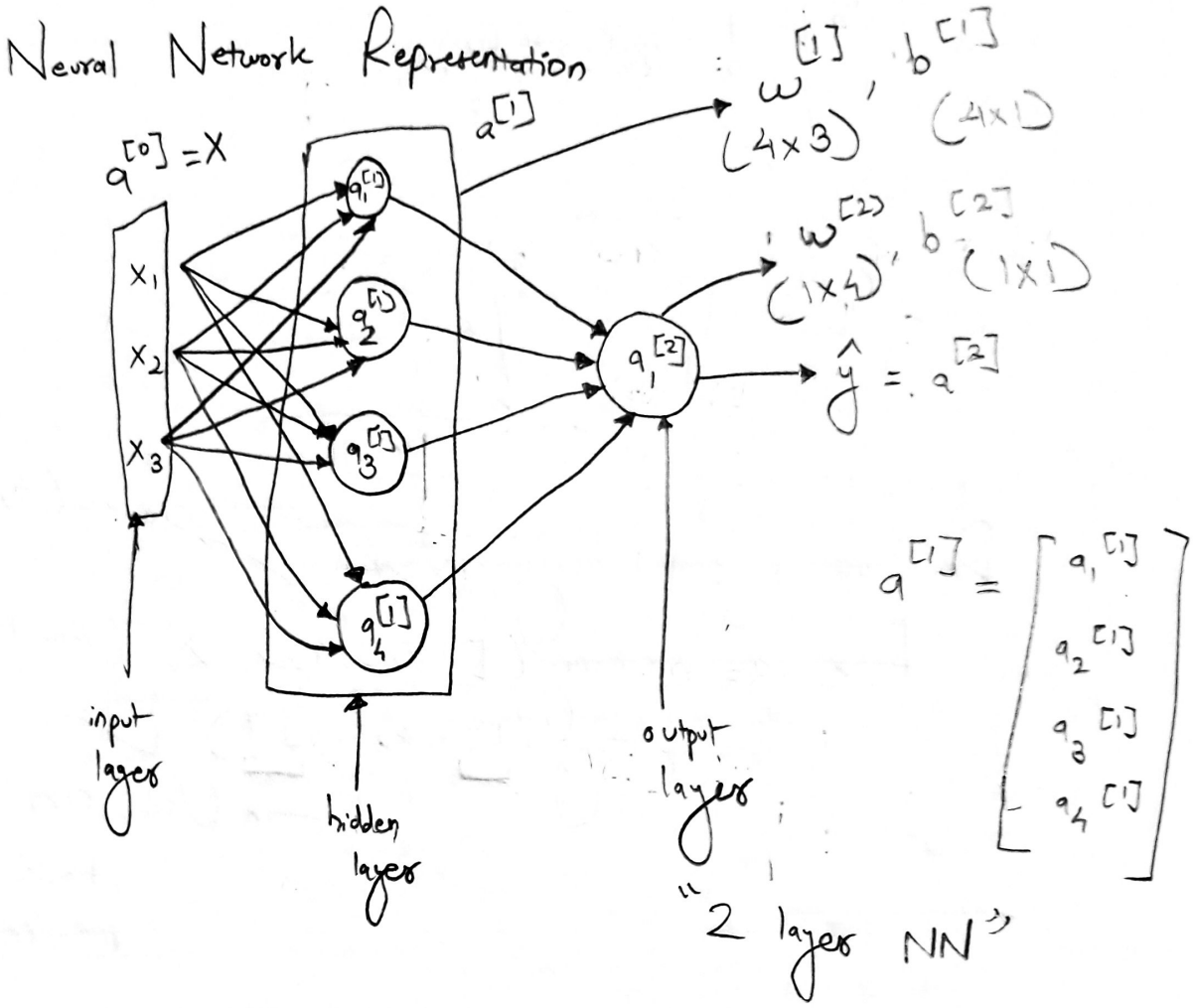
we don't count the input layer.

∴ N layer neural network

↳ N includes (hidden + output) layers

NOT input layer

Neural Network Representation



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

"2 layer NN"

$a^{[l]}_i$: $[l]$ → layer; i → node in that layer

$a^{[l](k)}_i$: $[l]$ → layer; (k) → training example; i → node in that layer

for $i = 1$ to m (for m training examples)

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

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

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

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

Vectorizing across multiple examples

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

(n_x, m)

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

$\therefore$ Vectorized version is:—

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

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

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

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

for A, Z, X

↑ different nodes/units

→ different training examples

Justification for vectorized implementation

$$z^{[1](1)} = w^{[1]} \cdot x^{(1)} + b^{[1]} \qquad z^{[1](2)} = w^{[1]} \cdot x^{(2)} + b^{[1]}$$

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

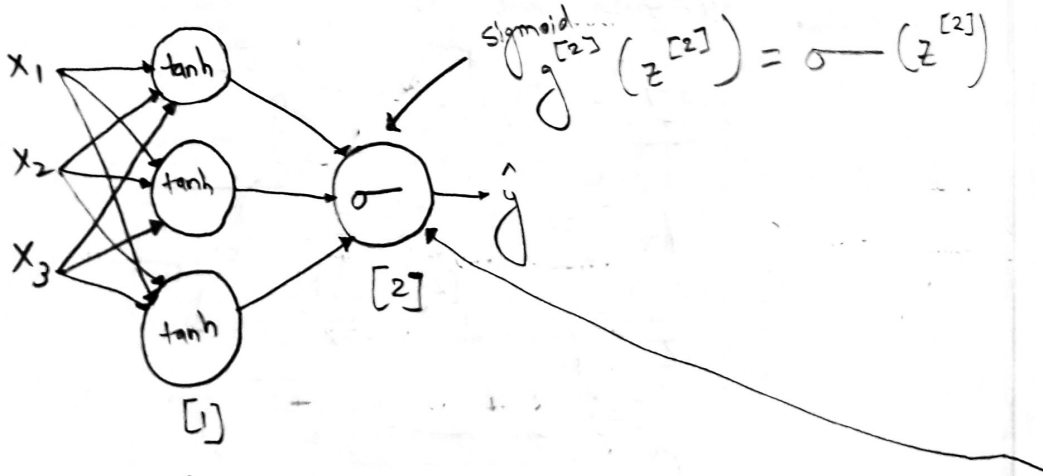
$$w^{[1]} = \begin{bmatrix} — \\ — \\ — \\ — \end{bmatrix} \qquad w^{[1]} x^{(1)} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \qquad w^{[1]} \cdot x^{(2)} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

$$w^{[1]} \cdot x^{(3)} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

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

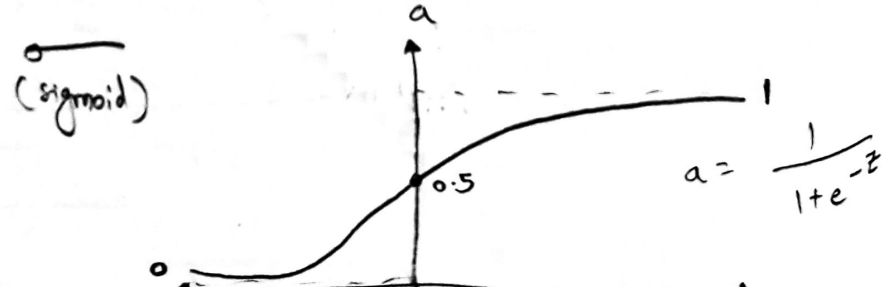
$$= \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & \cdots \\ | & | & | \\ +b^{[1]} & +b^{[1]} & \end{bmatrix} = Z^{[1]}$$

Activation functions

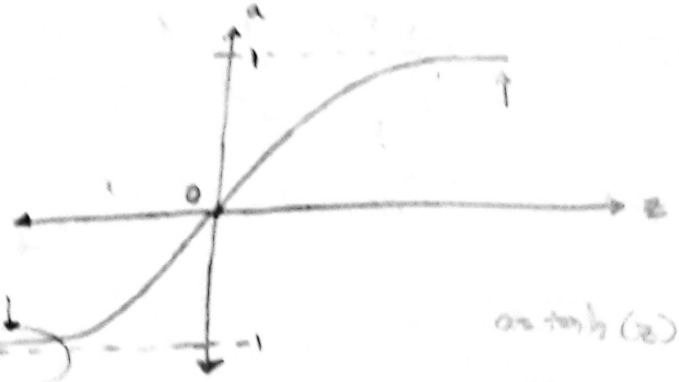


$$g^{[1]}(z^{[1]}) = \tanh(z^{[1]})$$

g can be



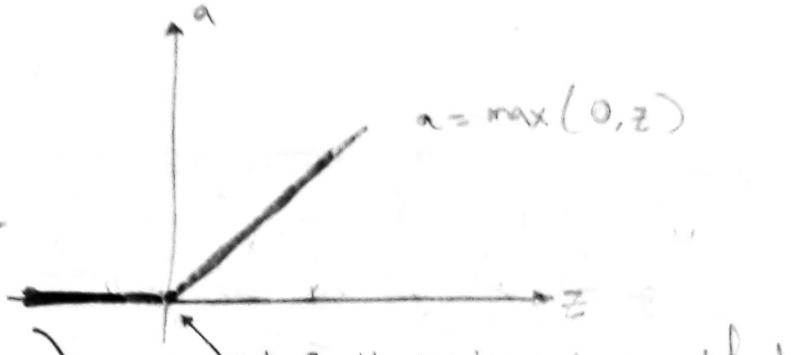
tanh



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

ReLU
(rectified linear unit)

To prevent gradient becoming small, we use ReLU (fast learning)



at 0 it is in reality undefined

tanh → • Can be used when we want to center our data and have 0 mean for our data (not possible for sigmoid function)

→ which has mean closer to 0.5 closer to 0

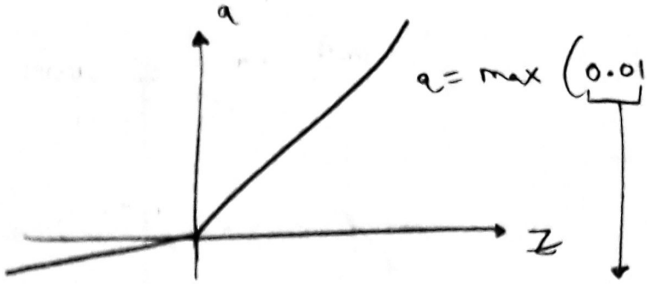
• Due to above point, it makes learning a bit easier (for next layer).

between tanh and sigmoid

"I pretty much never use sigmoid activation function anymore, the tanh function is almost always superior." — Lecturer

Exception: — in output layer, for binary classification, using sigmoid activation function can be useful.

Leaky ReLU



Can be used as a hyperparameter

Why not use a linear activation function?

eg:- $z^{[1]} = W^{[1]} \cdot x + b^{[1]}$

$a^{[1]} = z^{[1]}$ (g it is)

⋮

if we do the above then:—

$a^{[1]} = z^{[1]} = W^{[1]} \cdot x + b^{[1]}$

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

$a^{[2]} = W^{[2]} \cdot \underbrace{\left(W^{[1]} \cdot x + b^{[1]}\right)}_{a^{[1]}} + b^{[2]}$

$= \underbrace{\left(W^{[2]} \cdot W^{[1]}\right)}_{w'} \cdot x + \underbrace{\left(W^{[2]} \cdot b^{[1]} + b^{[2]}\right)}_{b'}$

$= \underbrace{w'x + b'}$ → this is a linear equation

(model is no more expressive than standard logistic regression models without any hidden layer.)

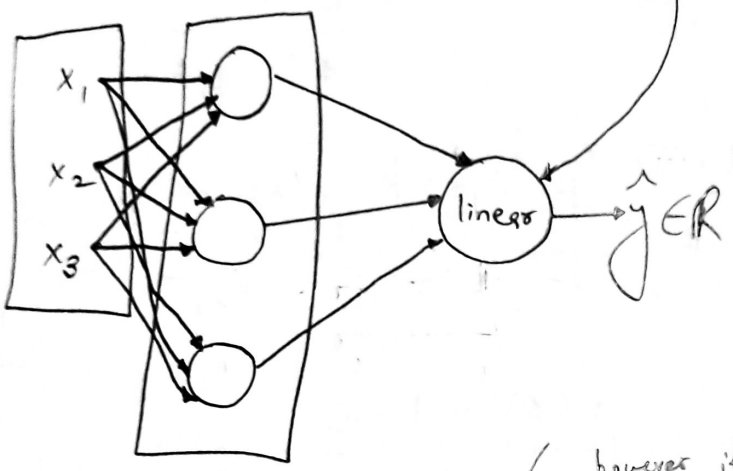
One place where we can use linear activation function

↓ is

regression problems

↓ used in

output layer

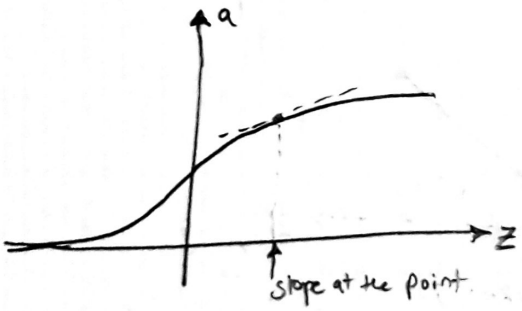


non linear activation function

(however if range is non negative, ReLU can also be used.)

Derivatives of activation function

Sigmoid



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

$$g'(z) = \frac{d\,g(z)}{dz} = \text{slope of } g(x) \text{ at } z$$

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

$$= g(z)\cdot(1-g(z))$$

$$g'(z) = a\cdot(1-a)$$

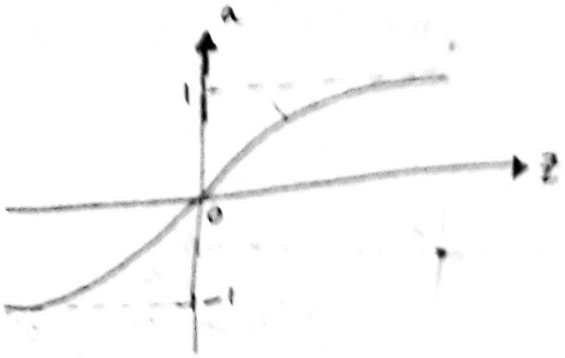
for $z = 10$, $g(z) \approx 1 \approx a$

$g'(z) \approx 1(1-1) \approx 0$

for $z = -10$, $g(z) \approx 0$

$g'(z) \approx 0\cdot(1-0) \approx 0$

tanh



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

$$g'(z) = \frac{dg(z)}{dz} \Rightarrow \text{slope of } g(z) \text{ at } z$$

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

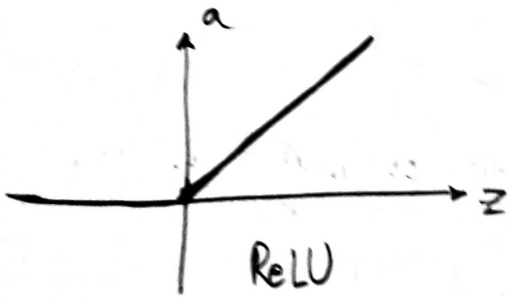
$$a = g(z), \quad g'(z) = 1 - a^2$$

if $z = 10 \rightarrow \tanh(z) \approx 1 \longrightarrow g'(z) \approx 0$

if $z = -10 \rightarrow \tanh(z) \approx -1 \longrightarrow g'(z) \approx 0$

if $z = 0 \rightarrow \tanh(z) \approx 0 \longrightarrow g'(z) \approx 1$

ReLU

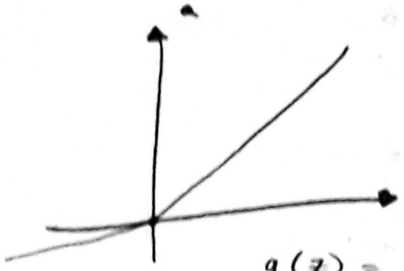


$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \\ \text{undefined} & \text{(Technically for } z = 0\text{)} \end{cases}$$

but we can write it as

Leaky ReLU



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

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

Gradient descent for neural networks

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

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

$n_x = n^{[0]}, n^{[1]}, n^{[2]}$

$n^{[2]} = 1$ ← # output layer

Cost function: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]})$

$$= \frac{1}{m} \sum_{i=1}^{n} L(\hat{y}, y)$$

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

→ for binary classification can be same as logistic regression one!

Gradient descent

- randomly initialize weights and biases
- repeat {

  Compute predictions $(\hat{y}^{(i)}, i = 1, 2, 3, \ldots, m)$ ← training examples

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

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

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

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

  ⋮

}

Formulas for computing derivatives

Forward propogation:-

$$z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(z^{[2]})$$

Backpropogation :-

$$dz^{[2]} = A^{[2]} - Y \qquad (Y = [y^{(1)}\ y^{(2)}\ y^{(3)} \ldots\ y^{(m)}])$$
$$dw^{[2]} = \frac{1}{m} \cdot dz^{[2]} \cdot A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \cdot np.sum(dz^{[2]}, axis=1, keepdims=True)$$
$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$$

$W^{[2]T} dz^{[2]}$: $(n^{[1]}, m)$; $g^{[1]\prime}(z^{[1]})$: $(n^{[1]}, m)$; $*$: element wise product

$$dw^{[1]} = \frac{1}{m} dz^{[1]} \cdot X^T$$
$$db^{[1]} = \frac{1}{m} \cdot np.sum(dz^{[1]}, axis=1, keepdims=True)$$

What happens if we initialize weights to zero?

- All of the hidden units remain symmetric.
- No matter how long we run gradient descent, they will continue to compute exactly the same function.

(bias terms can be initialized to zero)

∴ we initialize them randomly.

We prefer to initialize weights to very small random values.

↳ np.random.randn((x,y)) * 0.01

↳ in lecture (hyperparameter)

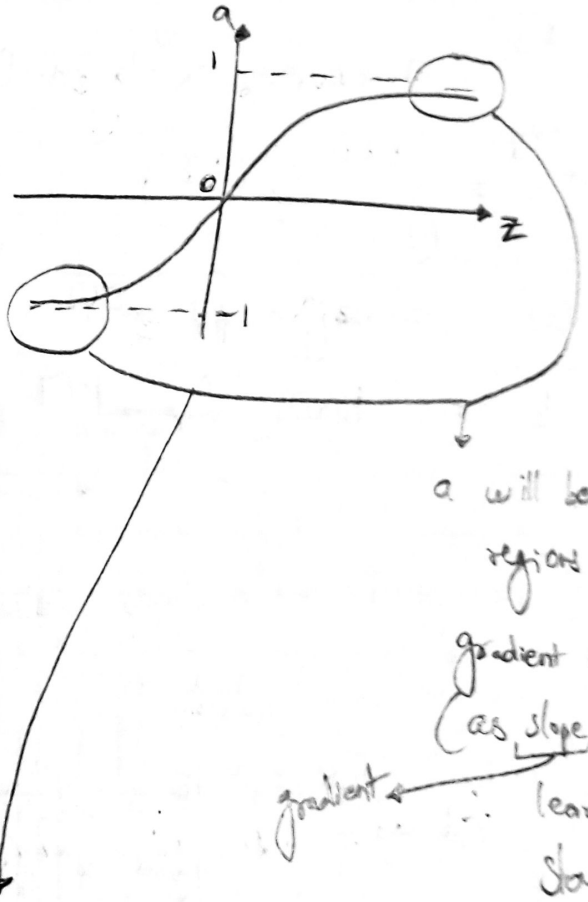
if weights are large (↑)

then in

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

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

and in curves like tanh



a will be near those regions making gradient descent slower (as slope is less) and ∴ learning will be slow.

gradient

Both tanh and sigmoid suffer from Vanishing gradient problem.