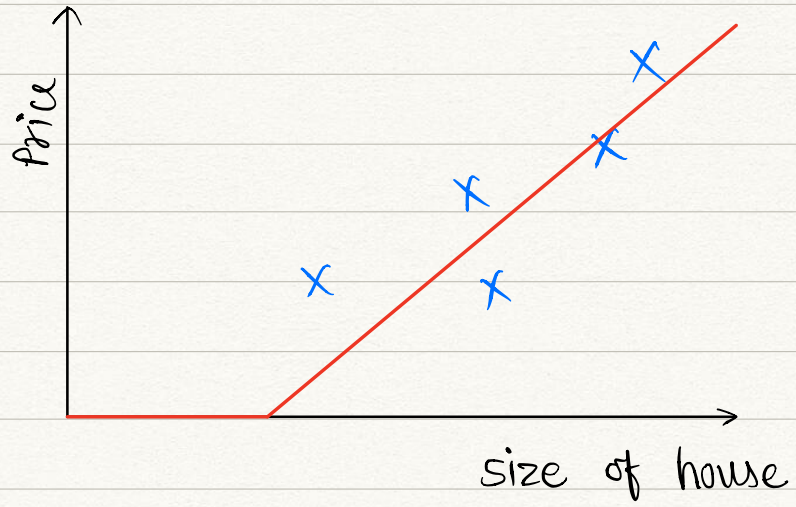
Q. What is Neural Network?



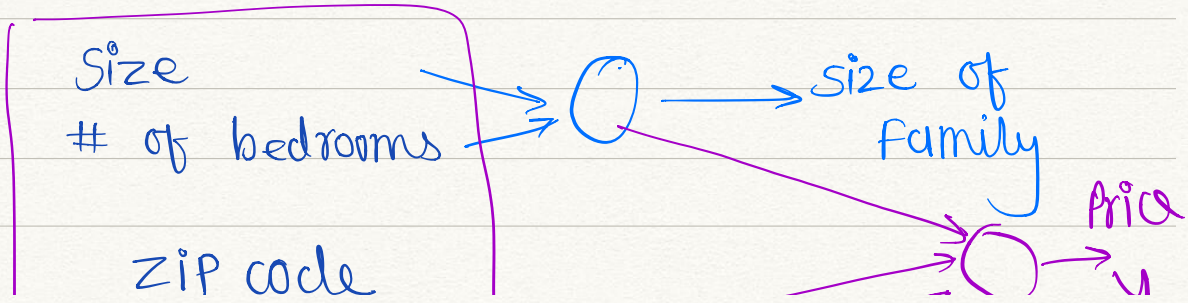
size $\xrightarrow{x}$ ○ $\xrightarrow{y}$ price } Single Neuron Network

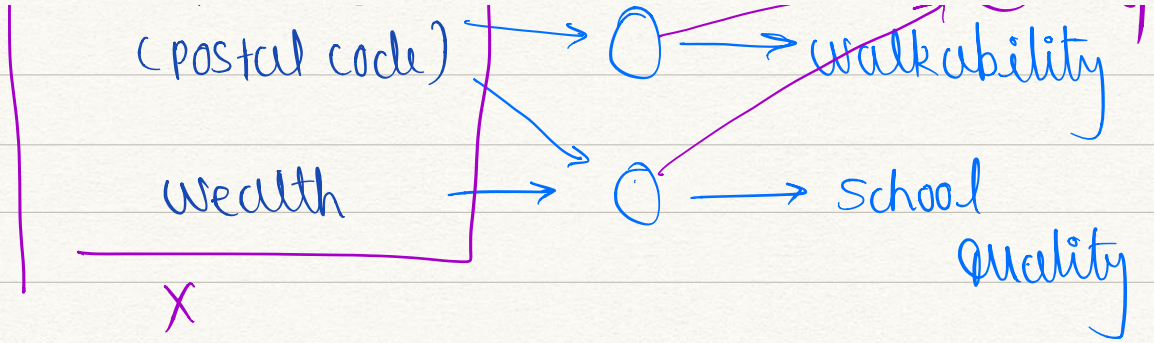
Neuron

} goes to zero sometimes & takes off as St. line.

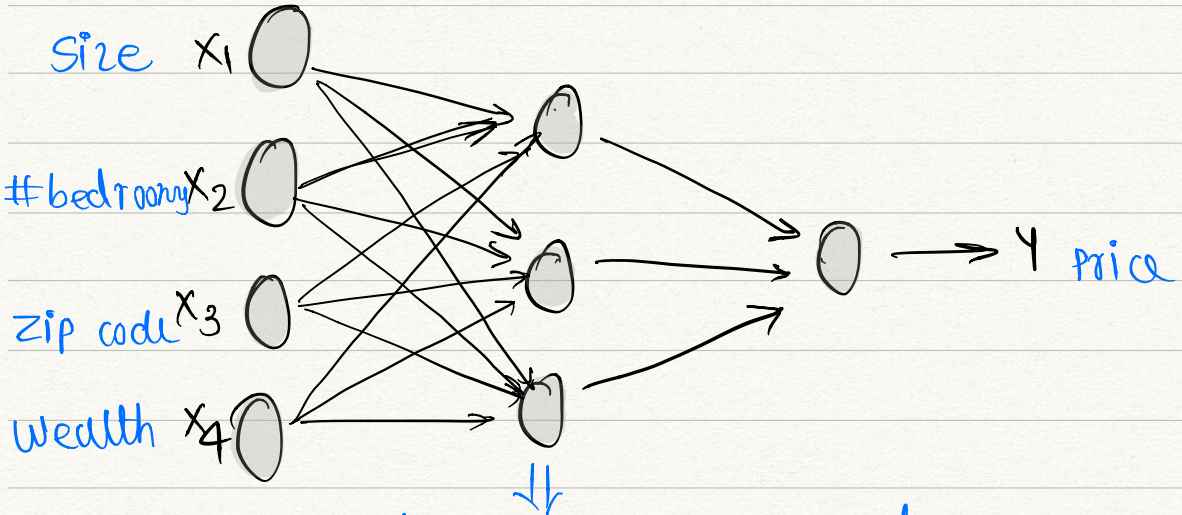
⇓

ReLU function
Rectified linear unit





give x & y, Network will figure out middle Neurons.



- Takes all four features as input
- Densly connected

✡ Supervised Learning:

X	Y	Application
Home features	Price	Real Estate
Ad, User info	Click on Ad? (0/1)	Online Advertising

Std NN

Image	Object	Photo tagging → CNN
Audio	Text Transcript	Speech recognition → RNN
English	Chinese	Machine Translation → RNN
Image, Radar info	Position of other car	Autonomous driving → custom / hybrid

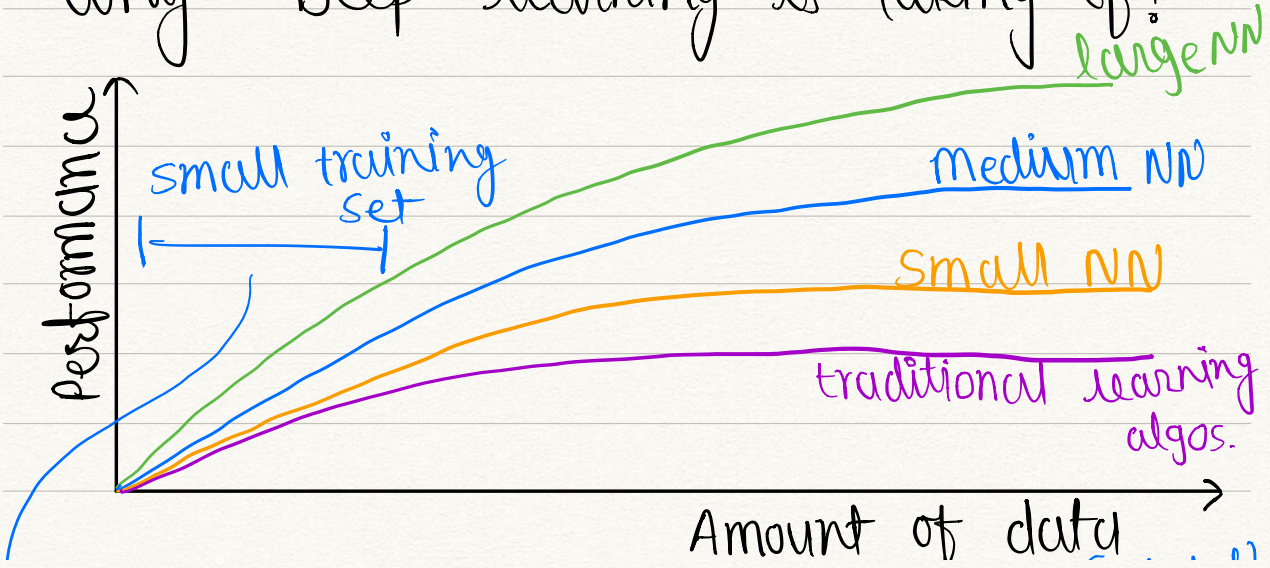
Structured Data
→ organised
tabular

Unstructured Data
- Audio
- Image
- Text

↓

Hard for computer

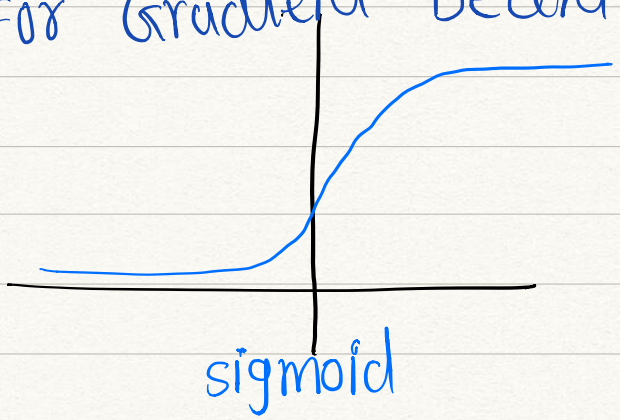
Why Deep learning is taking off?



→ Scale drives Deep learning progress (labelled)

Position of NN is not clearly defined.

For Gradient Decent



sigmoid

Decay of learning rate as we go forward.

ReLU

No decay of learning rate

→ Change in Algorithms.

⇒ Advances in Computation.

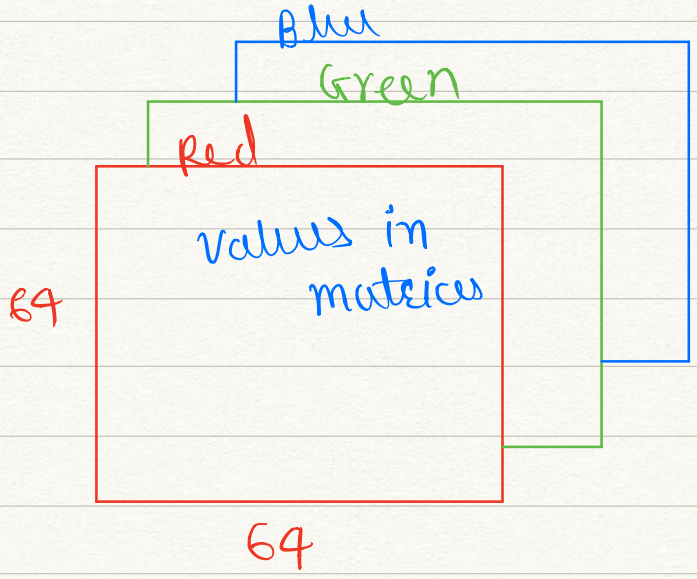
We can use for loop to go through entire training set but it will take lot of time complexity.

→ In Neural Networks we don't use for loops to go through training sets

Binary Classification :

(e.g.) Cat or No Cat in image

(64 × 64) px



Generate feature vector by putting all values in (n × 1) matrix

64 × 64 × 3

Here : (x, y)

$x \in R^{n_x}$, $y \in \{0, 1\}$

m training Example: $\{(x_1, y_1), (x_2, y_2), \dots\}$

$$X = \begin{bmatrix} | & | & | & & | \\ x_1 & x_2 & x_3 & \cdots & x_m \\ | & | & | & & | \end{bmatrix}_{n_x}$$

m columns

$$Y = \begin{bmatrix} y_1 & y_2 & \cdots & y_m \end{bmatrix}_{1 \times m}$$

Logistic Regression:

Given x, $\hat{y} = P(y=1|x)$

$x \in R^{n_x}$

Parameter: $w \in R^{n_x}$, $b \in R$

Output $\hat{y} = w^T x + b$

↳ not good

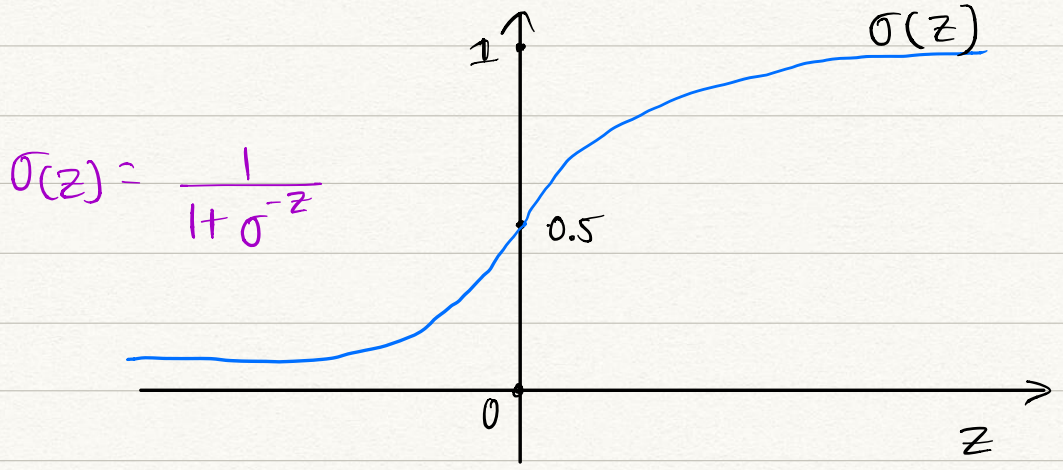
can't be binary (0 or 1)

so apply **sigmoid function**

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

$\downarrow$

z



In logistic Regression:
We try to fing w & b so that $\hat{y}$ become good estimate of chance of y being equal to 1.

Another Notation:

$x_0 = 1$, $x \in R^{n_x+1}$

$\hat{y} = \sigma(\theta^T x)$

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \rightarrow b \\ \} w \end{matrix}$

Logistic Regression Cost Function:

$$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

optimization problem becomes non convex. might have local optimum so gradient decent might not find global optimum.

Loss function

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

↓ small as possible

if $y=1$: $L(\hat{y}, y) = -\log \hat{y}$

large $\Rightarrow \hat{y} \rightarrow$ large

if $y=0$

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

large $\rightarrow \hat{y} \rightarrow$ small

Cost Function:

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

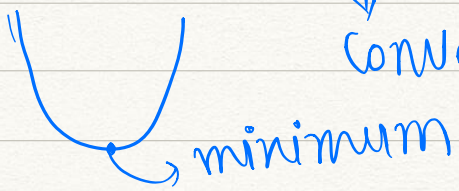
$$= \frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)}) \right]$$

Gradient Descent:

We want to find w, b for which J(w,b) is minimum.

↓

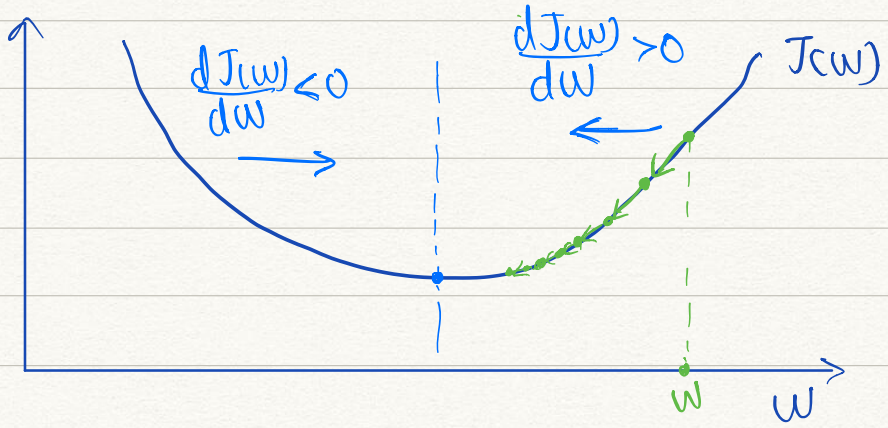
convex function



→ It will start at some initial point and then go down on the steepest decent in direction of $\nabla J(w,b)$.

→ It will converge to global minimum in case of convex function.

Gradient Descent for 1 D:



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

$\alpha \rightarrow$ learning rate

$\frac{dJ(w)}{dw} \Rightarrow$ in code "dw"

Coding term: $w := w - \alpha\, dw$

Actual in 2D:

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

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

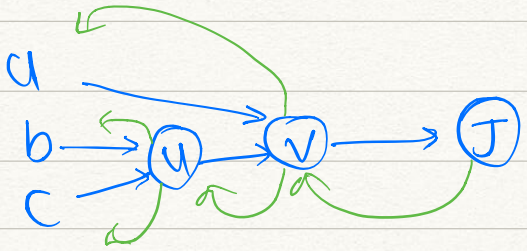
Computation Graph:

$J(a,b,c) = 3(a+bc)$

$u = bc$

$v = a+u$

$J = 3v$



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

$\frac{dJ}{dw} = 3$ $\qquad \frac{dJ}{da} = 3$

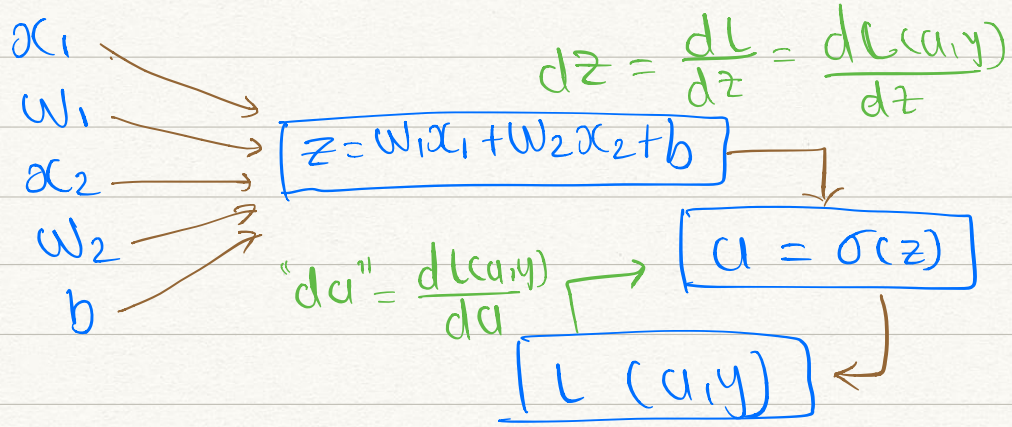
Logistic Regression Gradient Descent:

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

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

$$L = -\left(y \log \hat{y} + (1-y) \log (1-\hat{y})\right)$$

computational Graph:



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

$$dz = \frac{dL(a,y)}{dz} = \frac{dL}{da} \cdot \frac{da}{dz}$$

$$\frac{dL}{dw_1} = dw_1 = x_1 dz \qquad dw_2 = x_2 dz$$

$$db = dz$$

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

Logistic Regression on m examples:

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

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

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

$$dw_1^{(i)} \longrightarrow (x^{(i)}, y^{(i)})$$

Algorithm:

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

for loop:

For $i=1$ to m

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

for loop {

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

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

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

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

}

$J /= m$

$dw_1 /= m$, $dw_2 /= m$, $db /= m$

dw_1, dw_2, db → accumulator

$$dw_1 = \frac{\partial J}{\partial w_1}$$

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

Vectorization: get rid of for loop.

Logistic Regression Cost Function (Adv.)

If $y=1$ $\quad p(y|x) = \hat{y}$

$y=0$ $\quad p(y|x) = 1-\hat{y}$ } $p(y|x)$

$$p(y|x) = \hat{y}^{y} (1-\hat{y})^{(1-y)}$$ → function with above Boundary

conditions.

$$\log P(y|x) = \underbrace{y \log(\hat{y}) + (1-y)\log(1-\hat{y})}_{L(y,\hat{y})}$$

Cost Function:

$$P\left(\begin{matrix}\text{labels in} \\ \text{training set}\end{matrix}\right) = \prod_{i=1}^{n} P(y^{(i)}|x^{(i)})$$

we have to maximize it.

Put log & maximize it.

$$\log P(\,) = \sum_{i=1}^{n} \underbrace{\log P(y^{(i)}|x^{(i)})}_{-L(\hat{y}^{(i)}, y^{(i)})}$$

Maximum likelihood Estimate:

choose parameter that maximize $\log P(\cdots)$

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

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

↓so get rid of Negative sign.

$\frac{1}{m}$ → for better scale

By minimizing cost function, we are carrying out max. likelyhood estimation over logistic Reg. model under the assumption that our training examples are IRD (Identical, Randomly distri.)

Vectorization is useful in getting rid of for loop to run algorithm faster.

$z = w^T x + b$ $\quad w = \begin{pmatrix} \vdots \\ \end{pmatrix} \in \mathbb{R}^{n_x}$ $\quad x = \begin{pmatrix} \vdots \\ \end{pmatrix} \in \mathbb{R}^{n_x}$

Non Vectorized:

```
z = 0
for i in range (n-x)
    z += w(i) * x(i)
z += b
```

Vectorized: (numpy)

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

$\hookrightarrow w^T x$

Avoid for loops whenever possible

$u = Av$ $\quad u = np.dot(A,v)$

$u_i = A_{ij} v_j$

```
u = np.zeros((n,1))
for i ...
```

for j - - -
$u_i += A_{ij} \cdot v_j$

Vectors & Matrix valued Function:

e.g. Need to apply exponential operation on every element of matrix.

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

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

Using Numpy:

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

e.g. np.log , maximum
abs

Logistic Regression Derivatives:

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

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

$dw /= m$

Vectorizing Logistic Regression:

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

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

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

$1 \times m$

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

Broadcasting → b: Real number, 1×1

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

Vectorising logistic Reg's Gradient Output:

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

$$dz = [dz^{(1)}\ dz^{(2)} \cdots dz^{(m)}]$$
$$A = [a^{(1)}\ a^{(2)} \cdots a^{(m)}]$$
$$y = [y^{(1)}\ y^{(2)} \cdots y^{(m)}]$$

$$dz = A - Y$$

$dw = 0$
$dw += x^{(1)} dz^{(1)}$
$dw += x^{(2)} dz^{(2)}$
$\vdots$
$dw /= m$

$db = 0$
$db += dz^{(1)}$
$db += dz^{(2)}$
$\vdots$
$db /= m$

$$dw = \frac{1}{m} X dz^T$$
$$= \frac{1}{m} \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

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

Summing all up:

for i in range (10000):

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

$$A = \sigma(z)$$
$$dz = A - Y$$
$$dw = \frac{1}{m} x\, dz^T$$

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

$$w := w - \alpha\, dw$$
$$b := b - \alpha\, db$$

Makes your code run faster.

e.g.

	Apple	Beef	Eggs	Potatoes
carb	56	0	4.4	68
Prot.	1.2	104	52	8
Fat	1.8	135	99	0.9

$= A_{3\times4}$

Calculate % of calories from C, P & F for each of 4.

⇒ Cal = A.sum(axis=0)

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

no need to use it. Redundant

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

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

↓ Python will convert it to

$\begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix}$

||

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

General Principle:

(m,n) matrix $\pm$ $\times$ $/$ (1,n) $\longrightarrow$ (m,n)

(m,1) $\longrightarrow$ (m,n)

Operate element wise.

$\Rightarrow$ Sum values vertically : axis = 0
Operation Horizontally : axis = 1

Tips:

Never use rank 1 array.

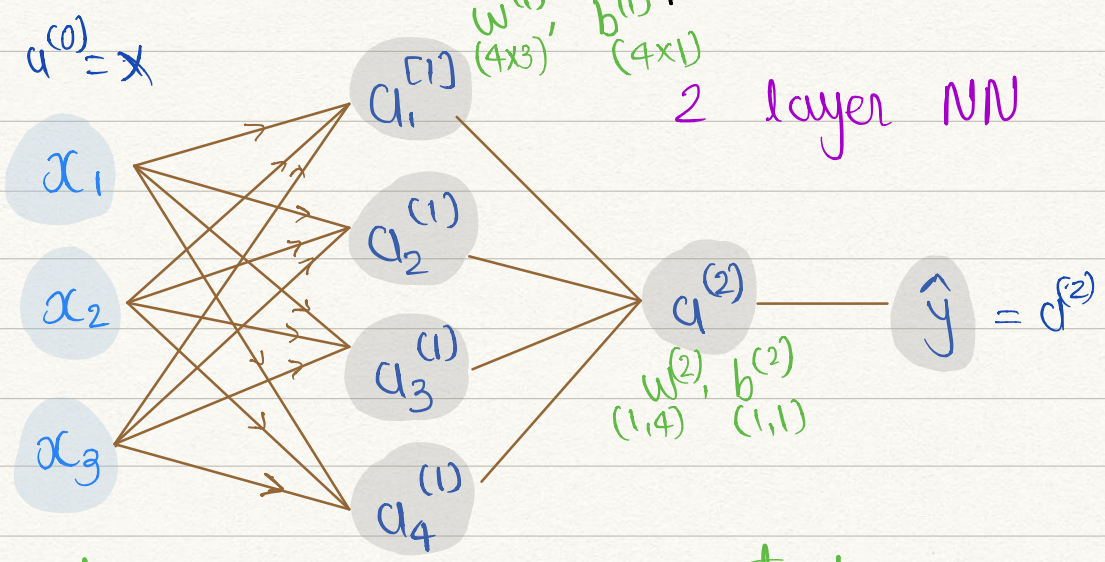
```
a = np.random.randn(5)
a.shape = (5,)
```

rank 1 array

use properly defined array:

```
a = np.random.randn(5,1)
a = np.random.randn(1,5)
```

Neural Network Representation:



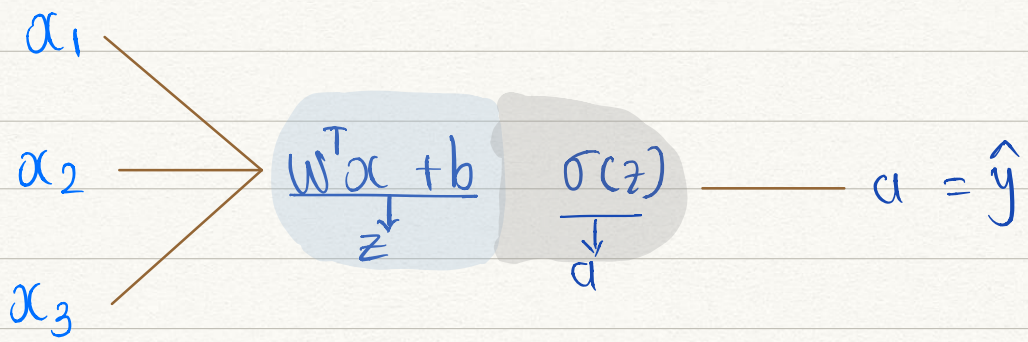
input layer (0) | hidden layer (1) | output layer (2)

$$a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_4^{(1)} \end{bmatrix}$$

$\hat{y} = a^{(2)}$ because in logistic reg. we only get single o/p from N.N.

$w^{(1)} \longrightarrow (4 \times 3) \longrightarrow$ 4 output unit
3 input unit

Computing NN's Output:



$a_i^{(l)}$ → layer number
→ node in layer

$$z_1^{(1)} = w_1^{(1)T} x + b_1^{(1)}$$
$$a_1^{(1)} = \sigma(z_1^{(1)})$$
} layer 1, node 1 computation

$$z_2^{(1)} = w_2^{(1)T} x + b_2^{(1)}$$
$$a_2^{(1)} = \sigma(z_2^{(1)})$$
} layer 1, node 2 computation

Vectorize the whole process:

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

$$= z^{(1)}$$

$$a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ \vdots \\ a_4^{(1)} \end{bmatrix} = \sigma(z^{(1)})$$

given input x:

$$\underset{(4\times1)}{z^{(1)}} = \underset{(4\times3)}{W^{(1)}} \underset{(3\times1)}{x} + \underset{(4,1)}{b^{(1)}} \qquad (x \to a^{(0)})$$

$$\underset{(4,1)}{a^{(1)}} = \sigma(\underset{(4,1)}{z^{(1)}})$$

Algorithm:

for $i=1$ to m:

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

} 4 lines of for loop

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

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

→ training examples.

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

↓ hidden units

Explaination for vectorized implementation:

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

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

$$w^{(1)} \begin{bmatrix} | & | & \\ x^{(1)} & x^{(2)} & \cdots \\ | & | & \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdots \\ \cdot & \cdot & \\ \cdot & \cdot & \\ \cdot & \cdot & \end{bmatrix} = \begin{bmatrix} | & | & \\ z^{(1)(1)} & z^{(1)(2)} & \cdots \\ +b^{(1)} & +b^{(1)} & \end{bmatrix}$$

$$w^{(1)} \cdot X \qquad Z^{(1)}$$

↓ Set of all training examples

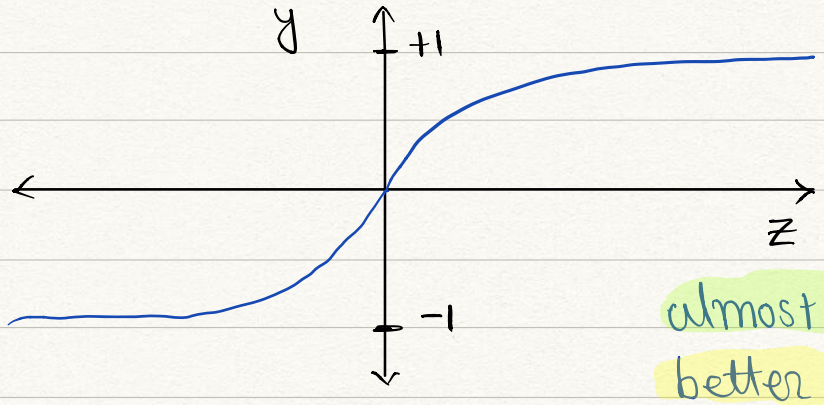
$$\left.\begin{aligned} Z^{(1)} &= W^{(1)} X + b^{(1)} \\ A^{(1)} &= \sigma(Z^{(1)}) \\ Z^{(2)} &= W^{(2)} A^{(1)} + b^{(2)} \\ A^{(2)} &= \sigma(Z^{(2)}) \end{aligned}\right\} \text{vectorization of 4 lines in for loop}$$

Activation Functions:

Sigmoid Function:

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

tanh Function:



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

almost always works better than sigmoid

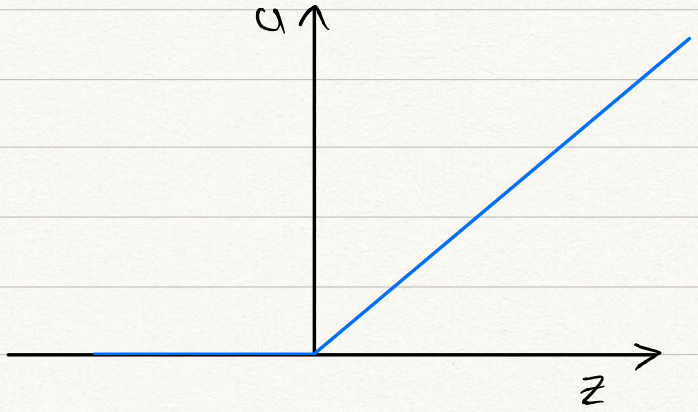
- have close to zero mean
- centering the data
- easier for further calculation

→ when you have binary classification output layer should be sigmoid.
keep hidden layers as tanh.

→ You can have different activation function for different layers.

→ when you have very large or very small i/p, slope of function is close to zero, so it can slow down gradient decent.

Rectified Linear Unit:



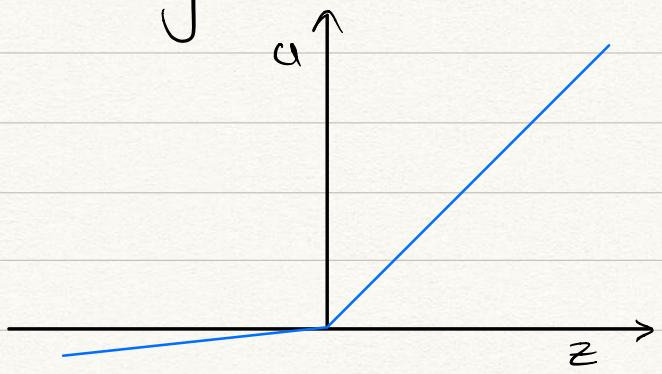
ReLU

$a = \max(0, z)$

Disadvantage:

→ Slope is 0 when i/p is negative

Leaky ReLU:



- works better than ReLU.

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

Why do we need non linear Activation Functions?

linear activation function:

$g(z) = z$

↳ identity function.

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

$$a^{(2)} = z^{(2)} = w^{(2)}a^{(1)} + b^{(2)}$$

$$a^{(2)} = w^{(2)}(w^{(1)}x + b^{(1)}) + b^{(2)}$$

$$= \underbrace{(w^{(2)}w^{(1)})}_{w'}x + \underbrace{(w^{(2)}b^{(1)} + b^{(2)})}_{b'}$$

$$a^{(2)} = w'x + b'$$

No matter how many layer NN has its just doing linear activation.
No benefit of doing this.

You can use linear activation on o/p layer. not recommended at hidden layer.

Derivatives of Activation Functions:

When we implement back prop. on NN we need to compute slope or the derivative of activation function.

Sigmoid Activation Function:

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

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

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

$$= a(1-a)$$

tanh activation:

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

$$g'(z) = \frac{d}{dz}(g(z)) = 1-(\tanh(z))^2$$

$$= 1-(g(z))^2$$

$$= \underline{1-a^2}$$

ReLU & Leaky ReLU:

$g(z) = \max(0, z)$ → ReLU

$g'(z) = 0$ if $z < 0$

1 if $z \geq 0$

~~undefined~~ if $z = 0$

↳ due to discontinuity.

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

$g'(z) = 0.01$ $z < 0$

1 $z \geq 0$

~~undefined~~ $z = 0$

Gradient Decent for NNs:

Parameters: $W^{(1)}$, $b^{(1)}$, $W^{(2)}$, $b^{(2)}$

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

$n_x = n^{(0)}, n^{(1)}, n^{(2)} = 1$

Cost Function: $J(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$

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

($\hat{y} = a^{(2)}$)

Repeat : { compute predictions: $(\hat{y}^{(i)}, i = 1, 2, \ldots, m)$

$dW^{(1)} = \frac{\partial J}{\partial W^{(1)}}$ $\frac{\partial J}{\partial b^{(1)}}$

$\frac{\partial}{\partial W^{(1)}}$, $\frac{\partial}{\partial b^{(1)}}$

update: $W^{(1)} = W^{(1)} - \alpha\, dW^{(1)}$

$b^{(1)} = b^{(1)} - \alpha\, db^{(1)}$

Formulas for Computing derivatives:

Forward Propogation:

$$Z^{(1)} = W^{(1)} X + b^{(1)}$$
$$A^{(1)} = g^{(1)}(Z^{(1)})$$
$$Z^{(2)} = W^{(2)} A^{(1)} + b^{(2)}$$
$$A^{(2)} = g^{(2)}(Z^{(2)}) = \sigma(Z^{(2)})$$

Back Propogation:

$$dZ^{(2)} = A^{(2)} - Y$$
$$Y = [y^{(1)}\ y^{(2)} \cdots y^{(m)}]$$

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

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

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

$$dZ^{(1)} = \underbrace{W^{(2)T} dZ^{(2)}}_{(n^{(1)}, m)} * g^{(1)'}(Z^{(1)}) \rightarrow (n^{(1)}, m)$$

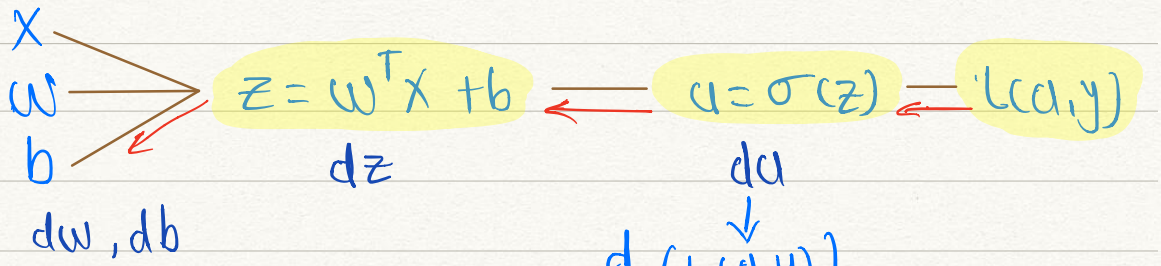
$*$ ↓ elementwise product

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

$$dw^{(1)} = \frac{1}{m} dz^{(1)} \cdot x^T$$

Backpropogation Intuition:

Computing Gradient:



$$\frac{d}{da}(L(a,y))$$

$$= \frac{d}{da}(-y \log a - (1-y) \log(1-a))$$

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

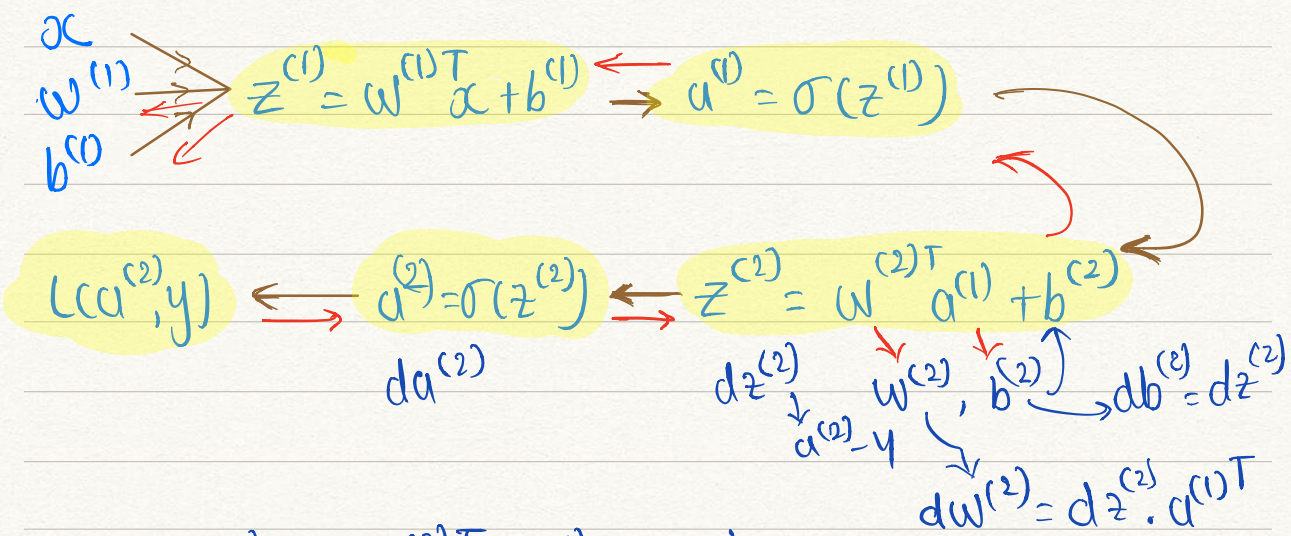
$$dz = da \cdot g'(z) \quad \rightarrow g(z) \rightarrow \text{sigmoid } \sigma(z)$$

$$a = \sigma(z)$$

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \quad \frac{d}{dz}(\sigma(z))$$

$$dw = dz \cdot x$$

$$db = dz$$



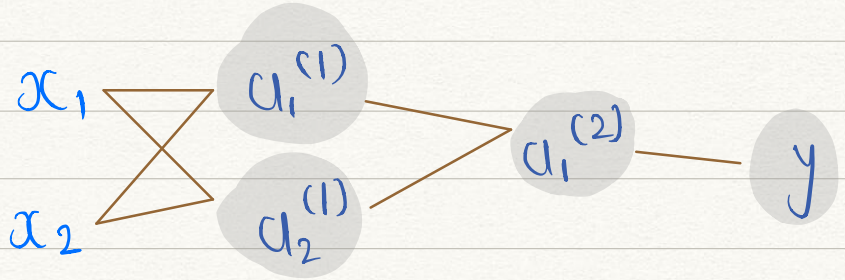
$$dz^{(1)} = w^{(2)T} dz^{(1)} * g^{(1)'}(z^{(1)})$$

$(n^{(1)}, n^{(2)})$ $(n^{(2)}, 1)$ $*$ $(n^{(1)}, 1)$ → element wise product

$w^{(2)}$ $(n^{(2)}, n^{(1)})$

Random Initialization:

What happens if you initialize weights to zero?



$$w^{(1)} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{(1)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$a_1^{(1)} = a_2^{(1)}$ $\quad dz^{(1)} = dz^{(2)}$

$dW = \begin{pmatrix} u & v \\ u & v \end{pmatrix}$ $\quad W^{(1)} = W^{(1)} - \alpha dW$

$W^{(1)} = \begin{bmatrix} - & - & - & - \\ - & - & - & - \end{bmatrix}$ identical rows.

If you initialize with zero, hidden layer will be symmetric and we will keep on computing same function, regardless of no. of layers.

So we do...

$W^{(1)} = np.random.randn(2,2) * 0.01$ — multiply with small no.

$b^{(1)} = np.zeros((2,1))$ — it is okay to have zero

↳ we do not have symmetry breaking problem

$W^{(2)} = random$

$b^{(2)} = 0$

→ If w values are too large, z will be very big or very small. in which case we will not get good gradient decent beccause it will have dw close to zero.
(end up at flat part of activation function)

Deep L-layer Neural Network

Total layers = hidden layers + output layers

$L = 4$ (No. of layers)

$n^{(l)}$ = No. of units in layer l

$n^{(1)} = 5 \quad n^{(2)} = 5 \quad n^{(3)} = 3 \quad n^{(L)} = 1$

$n^{(0)} = n_x = 3$

$a^{(l)} = g^{(l)}(z^{(l)})$ → activation in layer l

$w^{(l)}$ = weights for $z^{(l)}$

Forward Propogation in Deep Net.

General equation:

$$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$$

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

Vectorized:

$$Z^{(1)} = W^{(1)} A^{(0)} + b^{(1)}$$
$$A^{(1)} = g^{(1)}(Z^{(1)})$$

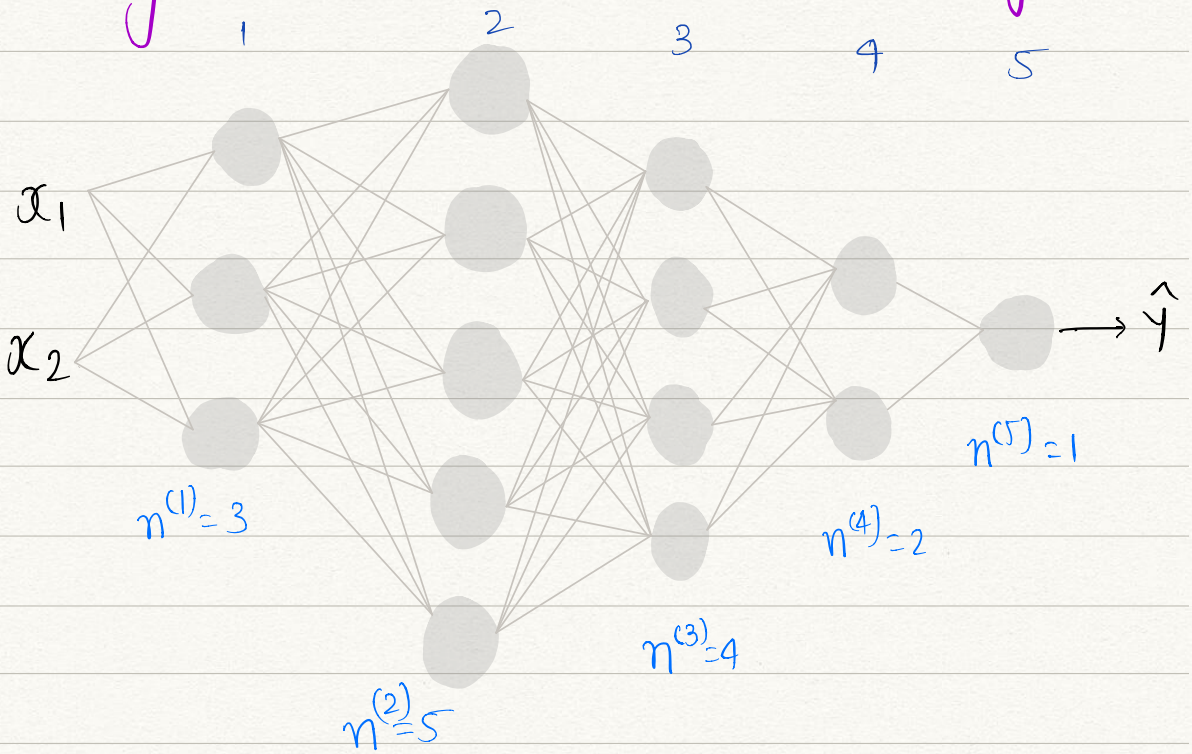
$$Z^{(2)} = W^{(2)} A^{(1)} + b^{(2)}$$
$$A^{(2)} = g^{(2)}(Z^{(2)})$$

$$\hat{y} = g^{(4)}(Z^{(4)}) = A^{(4)}$$

for loop

$0 \rightarrow l$

Getting Your Matrix dimension right:



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

$(3,1)$ $(3,2)$ $(2,1)$ $(3,1)$

$(n^{(1)},1)$ $(n^{(1)},n^{(0)})$ $(n^{(0)},1)$ $(n^{(1)},1)$

$$W^{(l)} = (n^{(l)}, n^{(l-1)})$$
$$b^{(l)} = (n^{(l)}, 1)$$

$W^{(1)} = (n^{(1)}, n^{(0)}) = (3,2)$ $\quad b^{(1)} = (3,1)$

$W^{(2)} = (5,3)$ $\quad b^{(2)} = (5,1)$

$W^{(3)} = (4,5)$ $\quad b^{(3)} = (4,1)$

$W^{(4)} = (2,4)$ $\quad b^{(4)} = (2,1)$

$W^{(5)} = (1,2)$ $\quad b^{(5)} = (1,1)$

$dw^{(l)} = (n^{(l)}, n^{(l-1)})$, $db^{(l)} = (n^{(l)}, 1)$

Vectorized Implimentation:

$$Z^{(1)} = W^{(1)} \cdot X + b^{(1)}$$

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

$$Z^{(1)}, A^{(1)} = (n^{(1)}, m) = dz^{(1)}, dA^{(1)}$$

Why deep Representation?

→ As we proceed deep into layers, they get more complex.

→ Initial hidden layers compute simple functions & as they move deeper they combine previous results & get more complex.

e.g. Audio → low levels (compose) → Phonems → (compose) → words → (compose) → Sentences

audio waveform

Circuit Theory & Deep learning:

Informally:

There are functions you can compute with a "small" L-layer deep neural Net that shallower nets require exponentially more hidden units to compute.

Building Blocks of deep Neural Nets:

Forward & Backward functions:

layer l : $W^{[l]}, b^{[l]}$

Forward:

input: $a^{[l-1]}$ o/p: $a^{[l]}$

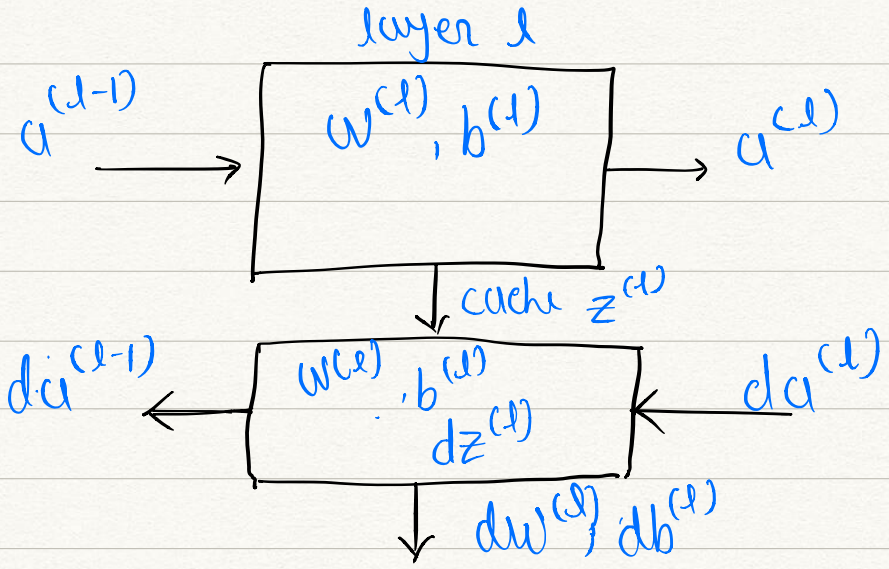
$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$ cache: $z^{[l]}$

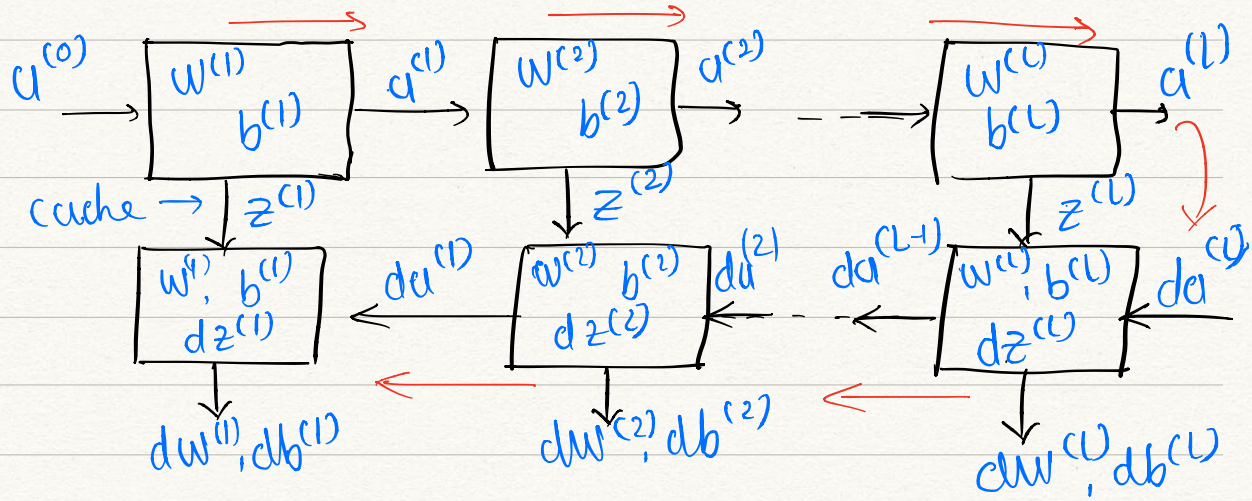
$a^{[l]} = g^{[l]}(z^{[l]})$

Backward:

i/p: $da^{[l]}$ o/p: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

cache: $z^{[l]}$





Backward Propagation for layer l:

Input : $da^{(l)}$

Output : $da^{(l-1)}, dW^{(l)}, db^{(l)}$

$$dz^{(l)} = da^{(l)} * g^{(l)'}(z^{(l)})$$

$$dW^{(l)} = dz^{(l)} \cdot a^{(l-1)}$$
$$db^{(l)} = dz^{(l)}$$

$$da^{(l-1)} = W^{(l)T} \cdot dz^{(l)}$$

$$dz^{(l)} = W^{(l+1)T} \cdot dz^{(l+1)} * g^{(l)'}(z^{(l)})$$

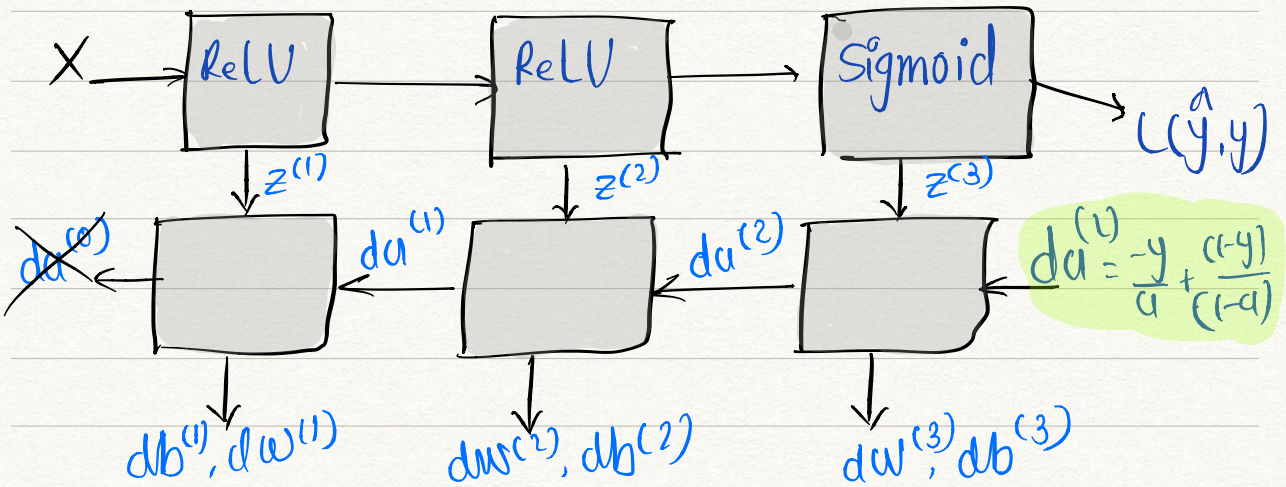
Vectorized:

$$dz^{(l)} = dA^{(l)} * g^{(l)'}(z^{(l)})$$

$$dW^{(l)} = \frac{1}{m} dz^{(l)} \cdot A^{(l-1)T}$$

$$db^{(l)} = \frac{1}{m} np.sum(dz^{(l)}, axis=1, keepdims=T)$$

$$dA^{(l-1)} = W^{(l)T} \cdot dz^{(l)}$$



Parameters Vs. Hyperparameters:

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

Hyperparameters: Learning Rate (α)

# of iterations of gradient decent to carry out.

# of hidden layer

# of hidden units

Activation function choice

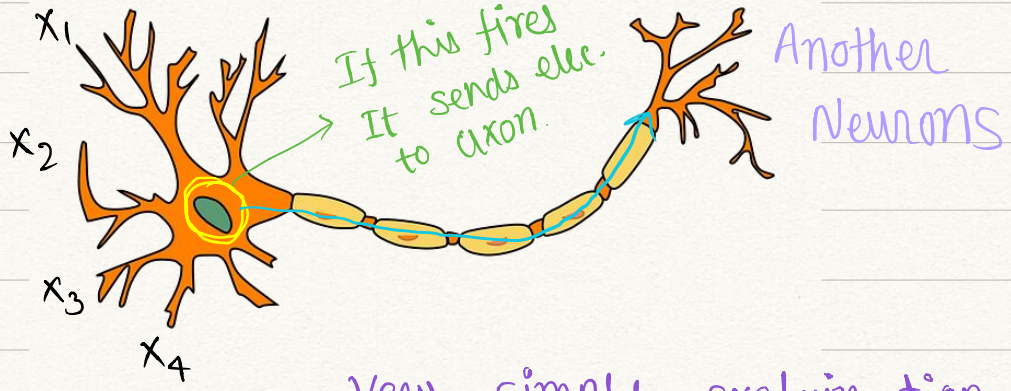
Other: Momentum, mini batch size, Regularization

Applied deep learning is a very empirical process.

idea → code → experiment

What does dep learning have to do with brain :

→ Not a lot...



Very simple explaination.

Quiz

Q:1 ✗ cache intermediate value forward to backward

Q:2 all but w^l, b^l, a^l

Q:3 Deeper layer compute more Complex

Q:4 False

Q:5 len (layer_dims)

W: (i, i-1)

Q:6 Layer : 4

hidden : 3

Q:7 False ✗ True

Q:8 True

Q:9 $W_1 = (4 \times 4)$ $W^2 = (3,4)$ $W^3 = (1,3)$

$b^1 = (4,1)$ $b^2 = (3,1)$ $b^3 = (1,1)$

Q:10 (n^l, n^{l-1})