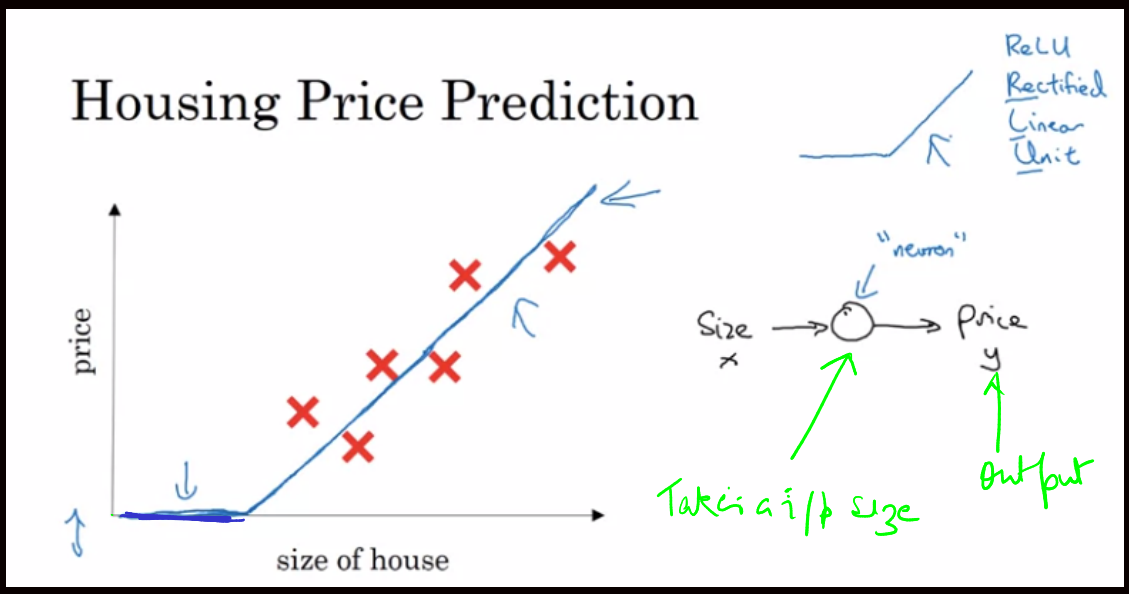
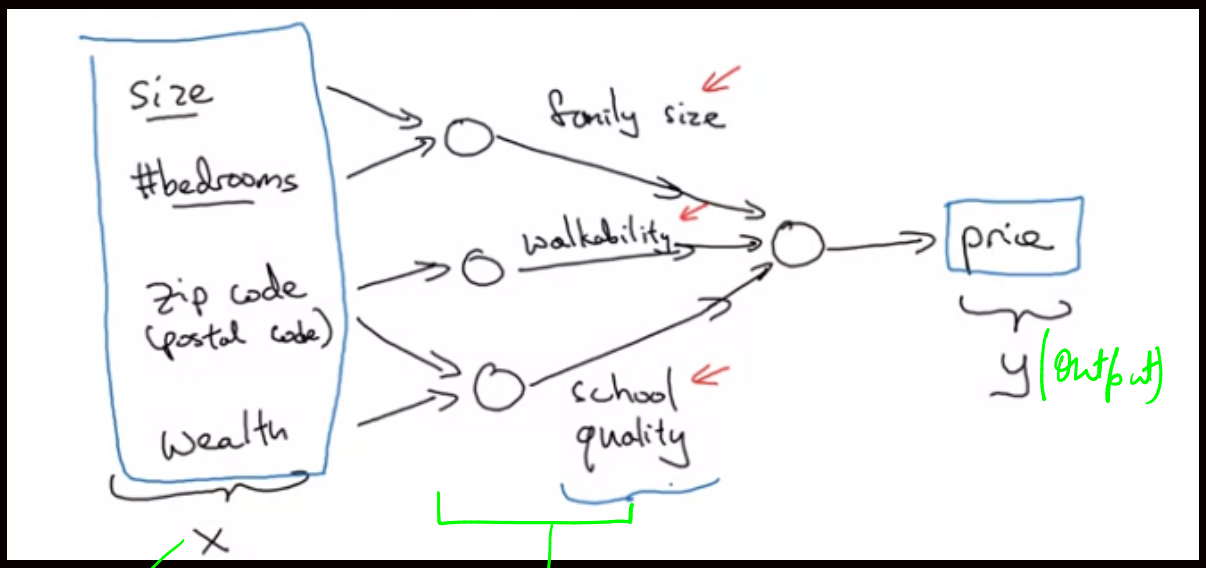
Week 1 Neural Networks & Deep Learning

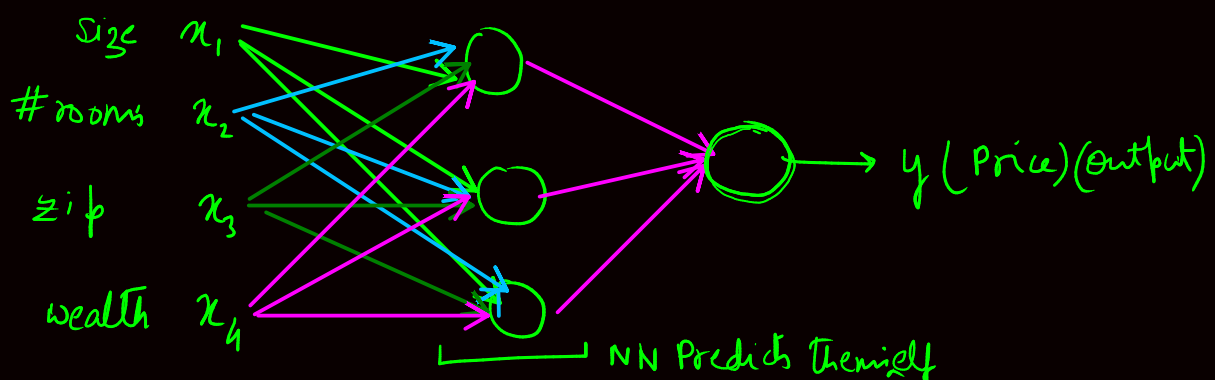


Smallest Possible Neural Network



Input which we will give.

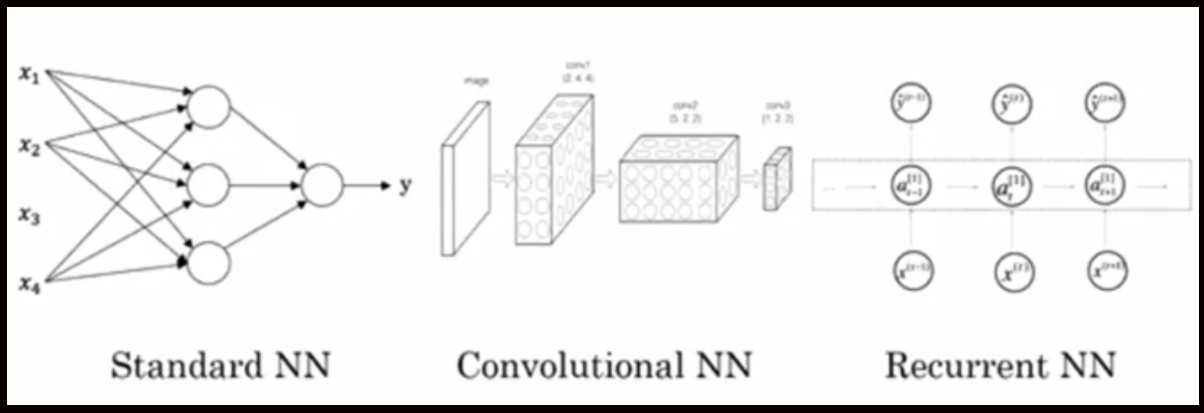
These features will be predicted by the network itself.



Supervised learning

Applications

Input(x)	Output (y)	Application	
Home features	Price	Real Estate	} Standard NN
Ad, user info	Click on ad? (0/1)	Online Advertising	} Standard NN
Image	Object (1,...,1000)	Photo tagging	} CNN
Audio	Text transcript	Speech recognition	} RNN (Sequence data)
English	Chinese	Machine translation	} RNN
Image, Radar info	Position of other cars	Autonomous driving	} Custom Hybrid Network



Standard NN | Convolutional NN | Recurrent NN

Other Classification

Structured Data

Size	#bedrooms	...	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Unstructured Data

Audio

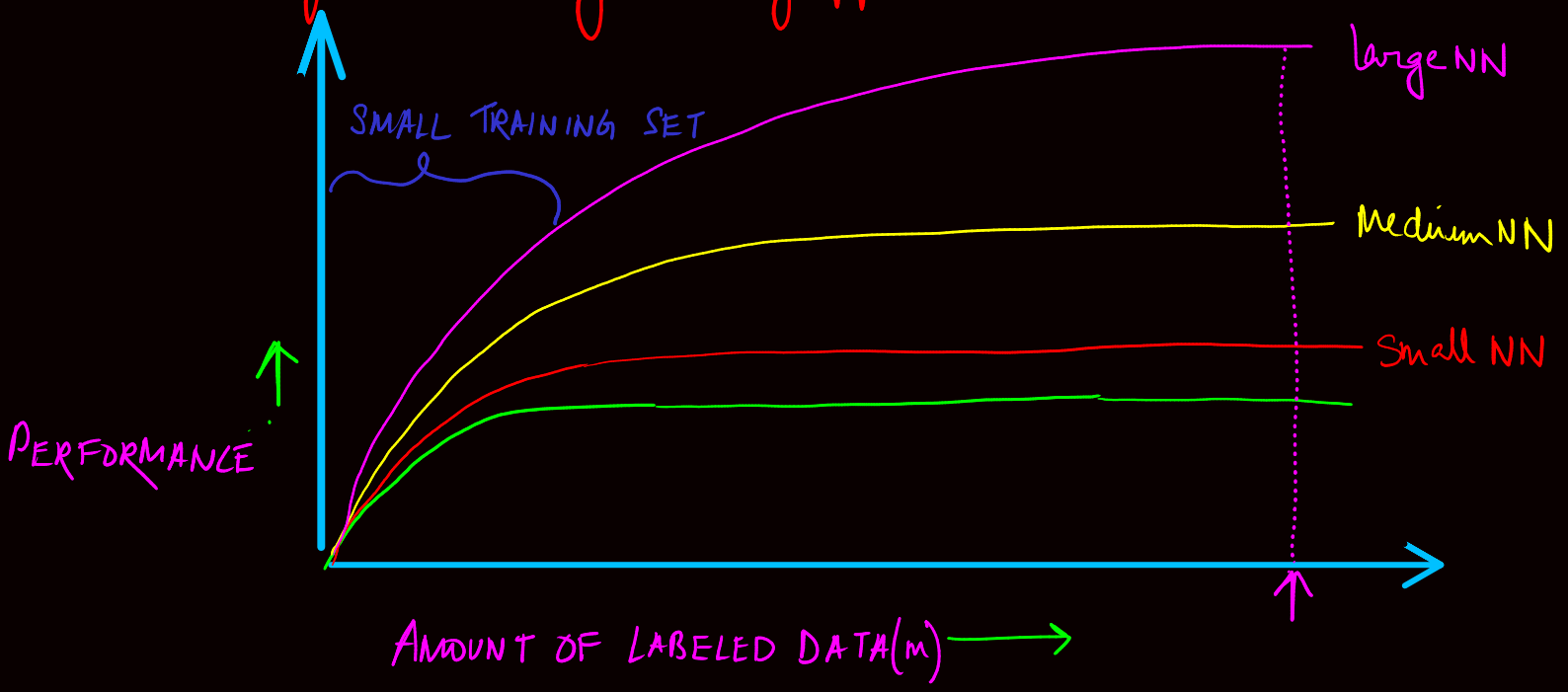
Image

Four scores and seven years ago...

Text

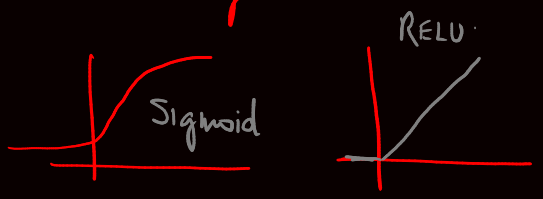
NN focuses more on Unstructured Data

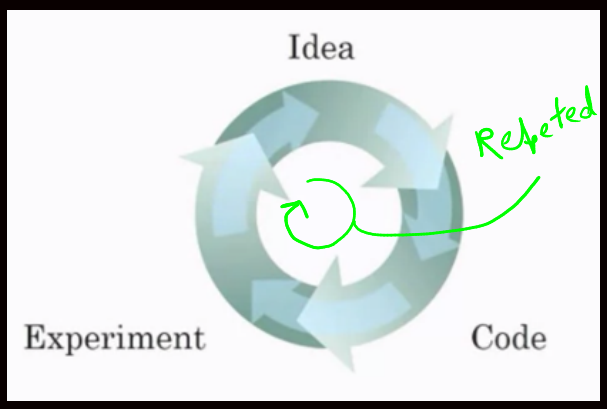
Why deep learning is taking off now?



Driving forces:

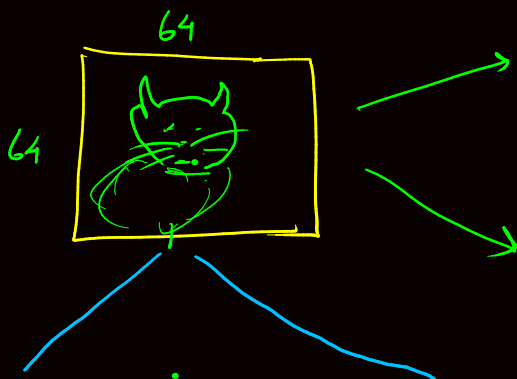
- Data. (↑ in data)
- Computation (Hardware)
- Algorithms. Eg





Week 2

BINARY CLASSIFICATION



feature vector. $x =$

$$\begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \end{bmatrix}$$

12288

dimension $64 \times 64 \times 3 = 12288$

$n_x = 12288$

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

m training example $= \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots (x^{(m)}, y^{(m)})\}$

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

1 feature vector (1st image)

$(n_x \times m)$

m

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

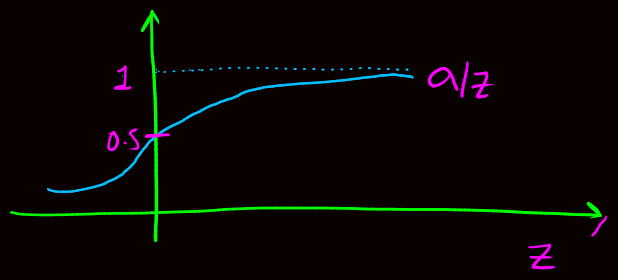
$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}_{1 \times m}$$

LOGISTIC REGRESSION

Given x, want $\hat{y} = P(y=1|x)$; $0 < \hat{y} \leq 1$

Parameters $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output $\hat{y} = \sigma(\underbrace{w^T x + b}_{z})$



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

when $z \to \infty$, $\sigma(z) = 1$

$z \to -\infty$, $\sigma(z) = 0$

COST FUNCTION

Prediction on the training example i^{th}:

$$y^{(i)} = \sigma(w^T x^{(i)} + b) \quad , \quad \sigma(z^{(i)}) = \frac{1}{1+e^{z^{(i)}}}$$

Loss fun^n (error fun^n):

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

If $y=1$ $\quad L(\hat{y}, y) = -\log \hat{y} \rightarrow \hat{y}$ must be large

$y=0$; $L(\hat{y}, y) = -\log(1-\hat{y}) \rightarrow \hat{y}$ be be as small as possible.

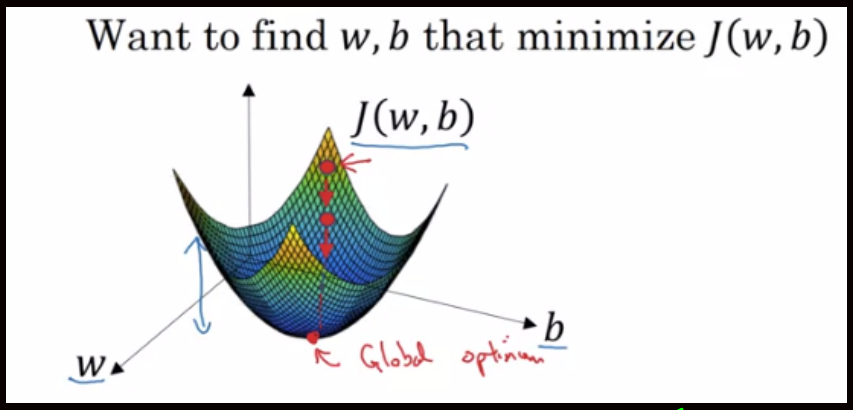
Loss fun^n $\rightarrow$ Single Training example

Cost fun^n: (for the entire training set)

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

* Convex fun^n ∪ → one global minima

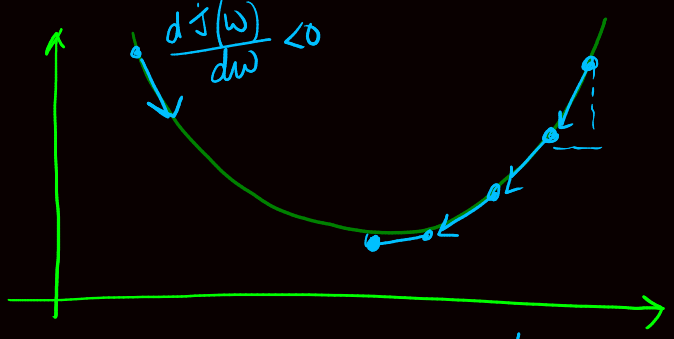
Non convex fun^n ∿∿∿ → Multiple global minima.



Gradient Descent

learning rate, dw

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

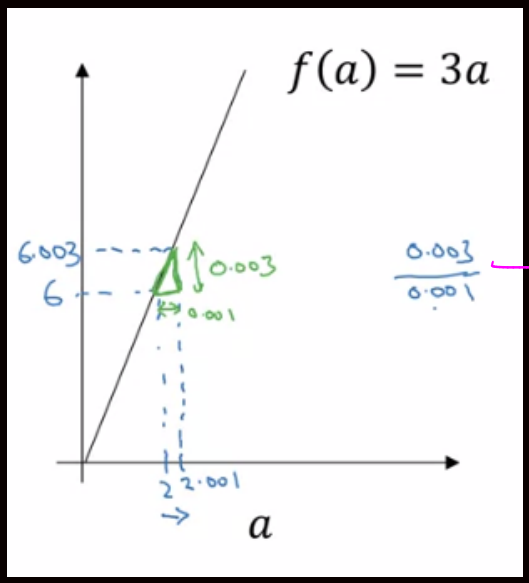


Update:

$$w = w - \alpha\frac{dJ(w,b)}{dw} \quad \text{"dw"}$$

$$b = b - \alpha\frac{dJ(w,b)}{db} \quad \text{"db"}$$

Derivatives



$\frac{0.003}{0.001}$ → Slope

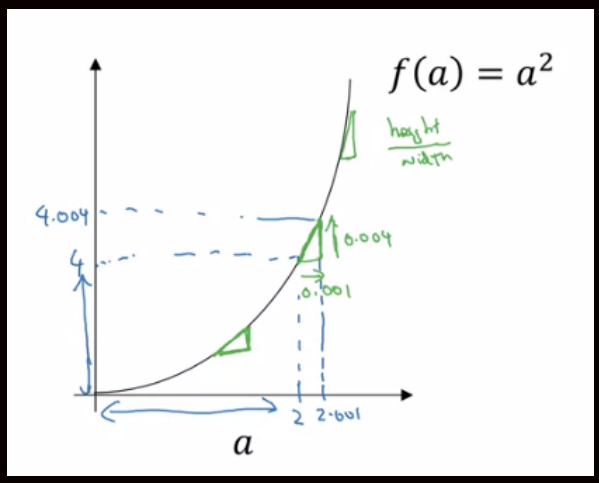
$$\frac{df(a)}{da} = 3 = \frac{d}{da} f(a)$$

i.e if we tweak the variable a 1 time, f(a) is tweaked 3 times that value

$a = 5$, $f(a) = 15$

$a = 5.001$, $f(a) = 15.003$

When slope is diff throughout the function



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

$\frac{df(a)}{da} = 4$ when $a = 2$

$\frac{df(a)}{da} = 10$ when $a = 5$

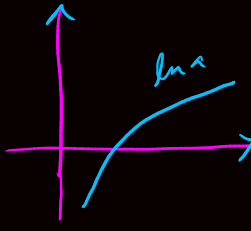
Similar fun^n's are

① $f(a) = a^3$

$\frac{df(a)}{da} = 3a^2$

② $\log_a a = f(a)$

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

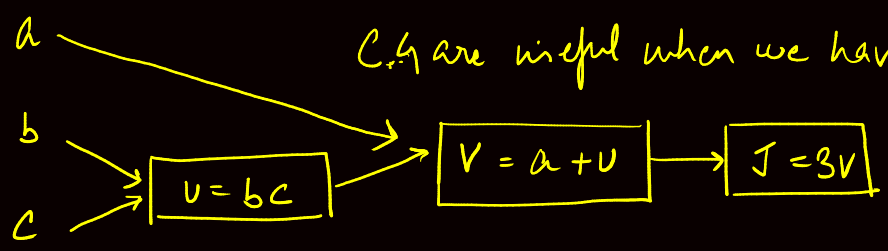


* Derivative of a fun^n = Slope of a fun^n

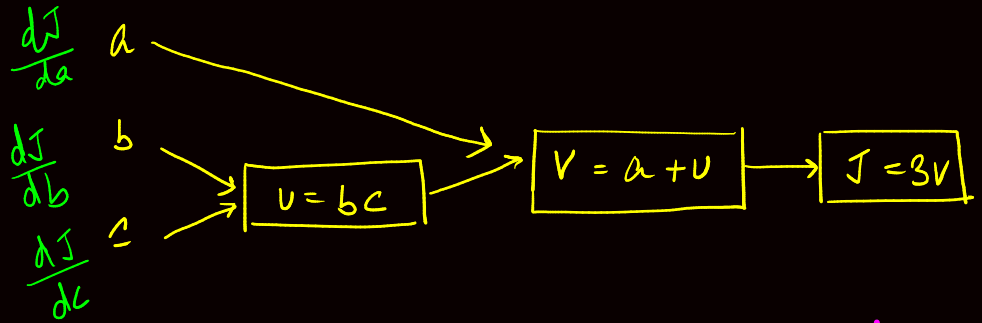
COMPUTATION GRAPH

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

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



C.G are useful when we have a specific o/p variable like J which we want to optimize



$\frac{dJ}{dv}$ = → $J = 3v$

$v = 11 \longrightarrow 11.001$ {x}

$J = 33 \longrightarrow 33.003$ {3x}

$\therefore \frac{dJ}{dv} = \frac{d(3v)}{dv} = 3$

$\frac{dJ}{da}$ = $a = 5 \longrightarrow 5.001$

$v = 11 \longrightarrow 11.001$

$J = 33 \longrightarrow 33.001$

a effects → v effects → J

Chain rule → $\frac{dJ}{da} = \frac{dJ}{dv} \times \frac{dv}{da}$ (3 × 1)

$= 3 \times 1$

Final O/P Variable to optimize = J

$\frac{d(\text{Final O/P Variable})}{d(\text{var})} = dJdvar = "dvar"$

Smart Notation for code

$* \frac{dJ}{d\,var}$

$$\frac{dJ}{db} = \frac{dJ}{dv} \cdot \frac{dv}{db}$$

$b = 3 \rightarrow 3.001$

$v = b.c \rightarrow 6 \rightarrow 6.002$

$$= \frac{dJ}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{db}$$

$= 3.x\,1 \times 2$

$= 6$

$\frac{dJ}{dc} = 6$ (Same

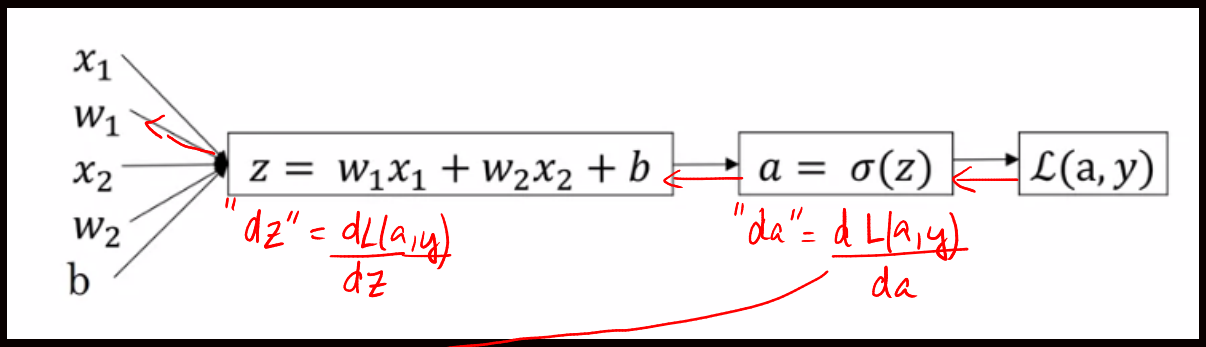
* The most good way to find derivatives is to do a right to left computation. by chain Rule

LOGISTIC REGRESSION GRADIENT DESCENT

(1) For 1 training example

Recap $z = w^T x + b$ $\quad L(a,y) = -(y \log a + (1-y)\log(1-a))$

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



$$\frac{dL}{dz} = \frac{dL}{da} \times \frac{da}{dz} \rightarrow a(1-a)$$

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

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

$$= (a - y)$$

$$* \quad \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial w_1}$$

$$(a - y) \times x_1$$

$$dz \times x_1$$

$$\therefore \quad \frac{\partial L}{\partial w_1} = "dw_1" = x_1 dz \quad , \frac{\partial L}{\partial w_2} = x_2 dw_2 \quad , \frac{dL}{db} = dz$$

Updates (learning rate → α)

$$w_1 = w_1 - \alpha dw_1 \; ; \; w_2 = w_2 - \alpha dw_2 \; ; \; b := b - \alpha db$$

(2) For 'm' training example

Recap $\quad z = w^T x + b \qquad L(a, y) = -(y \log a + (1-y) \log(1-a))$

$$\hat{y}^{(i)} = a^{(i)} = a(z)$$

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

Steps

① Initialization

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

For $i = 1$ to m

$$z^{(i)} = w^T x^{(i)} + b$$
$$a^{(i)} = a(z^{(i)})$$
$$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$$
$$dz^{(i)} = a^{(i)} - y^{(i)}$$
$$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$

This is easier as only 2 features w_1 & w_2

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

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

$J /= m$
$dw_1 /= m$
$dw_2 /= m$
$db /= m$

This technique is very tedious

∴ For such calculation we adopt what we know as vectorization,

VECTORIZATION

$z = w^T x + b$

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

```
a = np.random.rand(1000000)
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")

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")
```

```
250286.989866
Vectorized version:1.502752304077148ms
250286.989866
For loop:474.29513931274414ms
```

for loops are much slower.

GPU
CPU
→ SIMD → Single Instruction multiple data

AVOID USING EXPLICIT FOR LOOPS wherever possible

Examples

Suppose

① $v = Av$

$v = np.dot(A, v)$ → Vectorized.

$v_i = \sum_j A_{ij} v_j$

```
v = np.zeros((n,1))
for i ..... :
    for j ..... :
        v[i] += A[i][j] * v[j]
```

} Not Vectorized

②

Say you need to apply the exponential operation on every element of a matrix/vector.

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

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

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

Vectorized faster.

③

Original For Loop Technique.

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

Alternate

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

For $i = 1$ to m

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

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

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

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

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

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

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

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

$J /= m$

$dw_1 /= m$

$dw_2 /= m$

$db /= m$

$dw /= m$

VECTORIZING LOGISTIC REGRESSION

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

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

$$Z = \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} = w^T X + \begin{bmatrix} b & b & \cdots & b \end{bmatrix}_{1 \times m}$$

w^T → Row vector

$$\begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} = \begin{bmatrix} w^T x^{(1)} + b & w^T x^{(2)} + b & \cdots & w^T x^{(n)} + b \end{bmatrix}_{1 \times m}$$

$Z = np.dot(w.T, X) + b$ → Real no broadcasted to $(1 \times m)$ matrix to add to the previous term

Similarly, $A = \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix} = \sigma(Z)$

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

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

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

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

* $\boxed{db = \frac{1}{m}\ np.sum(dZ)}$
* $\boxed{dw = \frac{1}{m} X \times dz^T}$

$$= \frac{1}{m} \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & \ldots \\ | & | & | \end{bmatrix}_{n \times m} \begin{bmatrix} dz^{(1)} \\ dz^{(2)} \\ \vdots \\ dz^{(m)} \end{bmatrix}_{m \times 1}$$

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

* Code for iter in range(1000): → {epochs}

```
Z = np.dot(w.T, X) + b
A = σ(Z)
dZ = A - Y
dw = 1/m X dZ^T
db = 1/m np.sum(dZ)
w = w - α dw
b = b - α db
}
```

Vectorized. imp of Gradient Descent

BROADCASTING IN PYTHON

Calories from Carbs, Proteins, Fats in 100g of different foods:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

Sum

2% 94%

Q Find the % of calories for carb, protein & fat for each of the food items individually

```
In [1]: import numpy as np

In [2]: A = np.array([[56.0,0.0,4.4,68.0],
                      [1.2,104.0,52.0,8.0],
                      [1.8,135.0,99.0,0.9]])
        print(A)

[[ 56.    0.    4.4  68. ]
 [  1.2 104.   52.    8. ]
 [  1.8 135.   99.    0.9]]

In [3]: cal = A.sum(axis=0)
        print(cal)

[ 59.  239.  155.4  76.9]

In [4]: percentage = 100* A/cal.reshape(1,4)
        print(percentage)

[[94.91525424  0.          2.83140283 88.42652796]
 [ 2.03389831 43.51464435 33.46203346 10.40312094]
 [ 3.05084746 56.48535565 63.70656371  1.17035111]]
```

↗ reshape command is redundant here but good to be safe

* ① $\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100 \xrightarrow{\text{Python Broadcasting}} \begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix}$

② $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$

$1 \times n \rightarrow m \times n$

etc...

* (m,n) matrix $\pm$ $(1,n) \rightarrow (m,n)$
  $*$ $(m,1) \rightarrow (m,n)$
  $/$

* Tip: Don't use data structure.

* ```
a = np.random.randn(5)
```
  a.shape = (5,)
  "rank 1 array"
  → if such rank 1 arrays come
  } Don't use a = a.reshape((5,1))

```
a = np.random.randn(5,1)  → a.shape = (5,1)   column vector ✓
a = np.random.randn(1,5)  → a.shape = (1,5)   row vector ✓
assert(a.shape == (5,1))
```

* Logistic Regression cost fn

$\hat{y} = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$

*

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

$P(y|x) = \hat{y}^y (1-\hat{y})^{(1-y)}$ → summary

$\log P(y|x) = \log \hat{y}^y (1-\hat{y})^{(1-y)}$

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

maximize ↑ $\log(P(y|x)) = -L(\hat{y}, y)$

to minimize the loss
```

Cost on m examples

$$\log P(\text{labels in training set}) = \log \prod_{i=1}^{m} P\left(y^{(i)} \mid x^{(i)}\right)$$

$$= -L\left(\hat{y}^{(i)}, y^{(i)}\right)$$

By maximum likelihood estimation

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

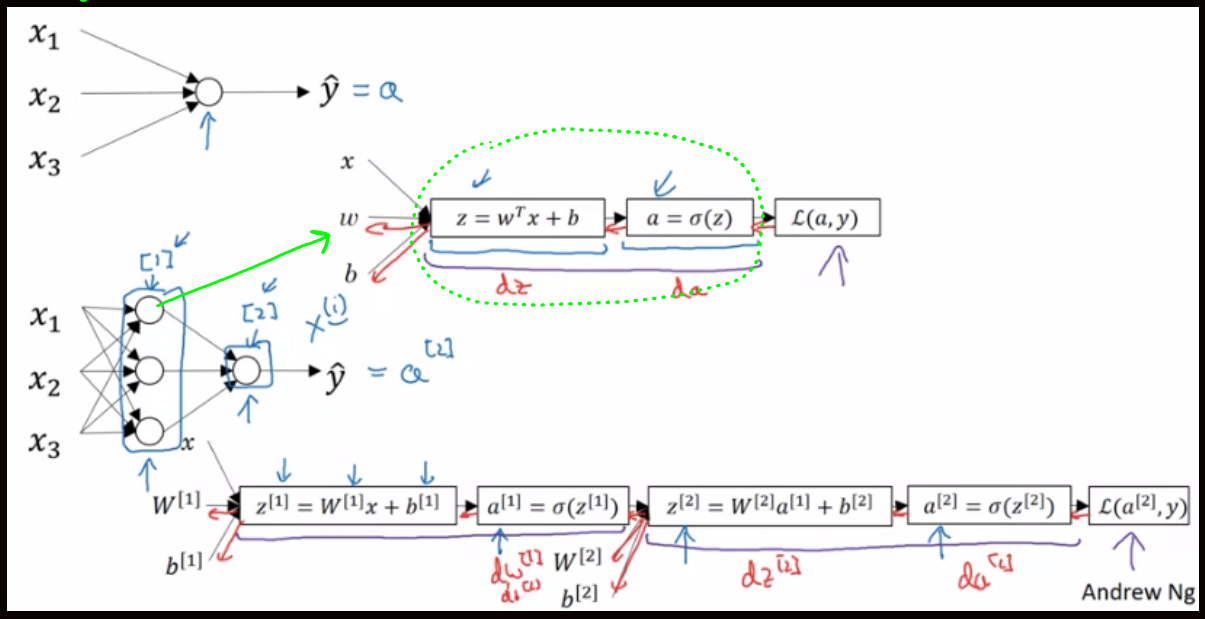
Cost : $J(w,b) = \frac{1}{m} \sum L\left(\hat{y}^{(i)}, y^{(i)}\right)$ ← maximum

minimize

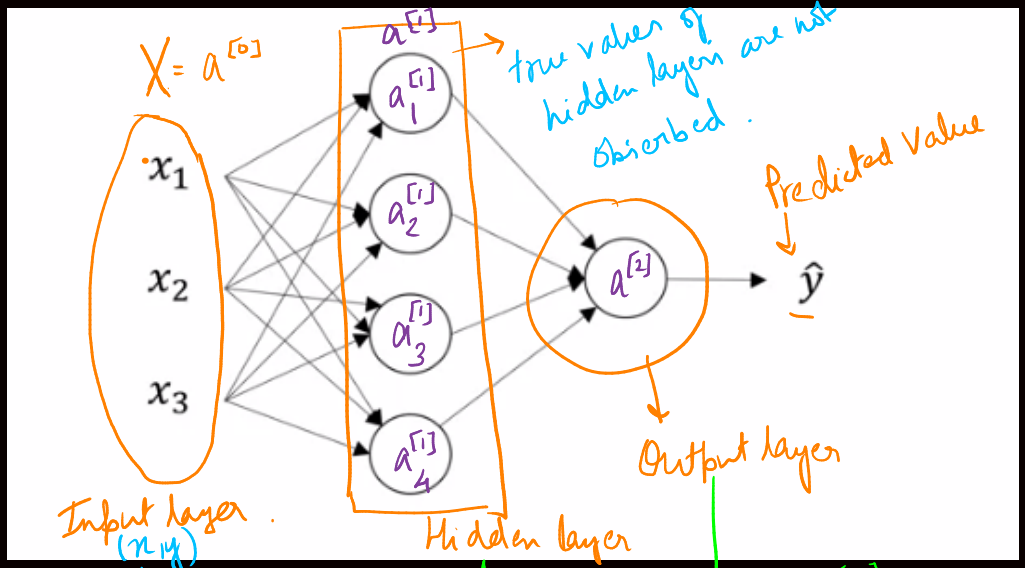
//

WEEK 3

Overview



2-layer Network [i/p excluded]



(x, y)

i/p image — label

$w^{[1]}, b^{[1]}$ — (4,3) (4,1)

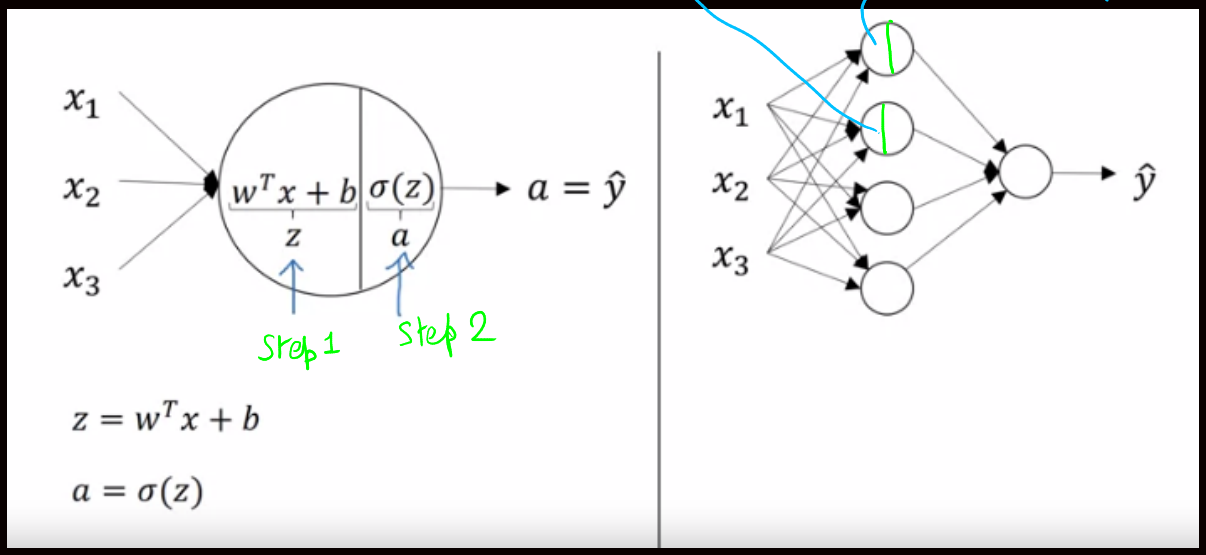
$w^{[2]}, b^{[2]}$ — (1,4) (1,1)

$z_2^{[1]} = w_2^{[1]} x + b_2^{[1]}$

$a_2^{[1]} = \sigma(z_2^{[1]})$

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

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



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

$$Z = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = Z^{[1]}$$

$$Z = \begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix}_{4\times 3} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{3\times 1} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = W^{[1]} \quad\quad b^{[1]}$$

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

$\therefore$

Given input x:

$$z^{[1]} = W^{[1]} x + b^{[1]} \qquad (a^{[0]} = x)$$
$$(4,1) \quad (4,3) \quad (3,1) \quad (4,1)$$

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

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

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

For m training example.

$a^{[2](i)}$ → i^{th} training example

↓ layer 2

Vectorized Solution

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

(training example)

$n_x \times m$ (n_x = no. of pixels)

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

where, m = no. of training examples

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

← no. of training examples →

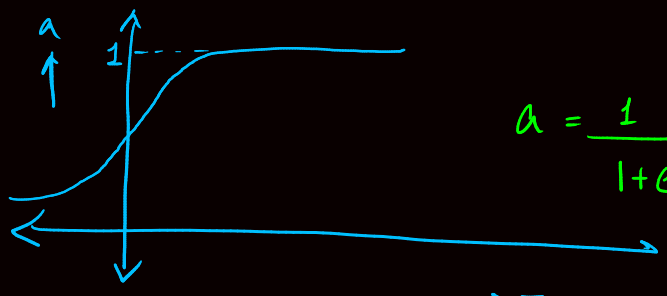
$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ a_2^{[1](1)} & & & \\ \vdots & & & \end{bmatrix}$$

↕ No. of nodes in that particular layer.

ACTIVATION FUNCTIONS

Sigmoid

The tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer.

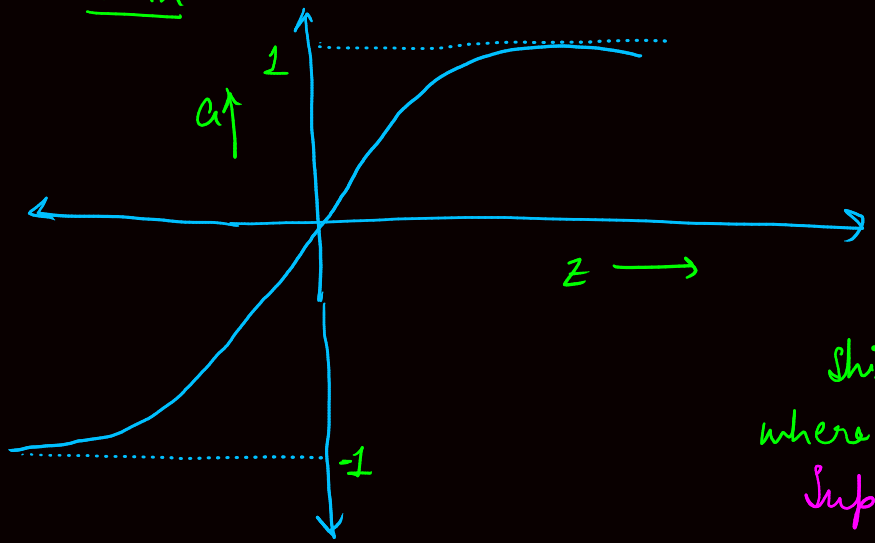


$$a = \frac{1}{1+e^{-z}}$$

$0 < a < 1$

Can be used for binary classification task

Tanh



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

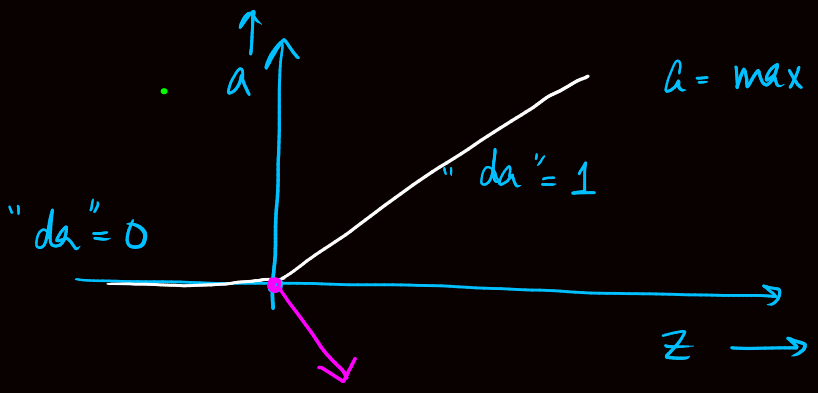
"Zero mean"

Shifted version of sigmoid where $-1 < a < 1$

Superior than sigmoid

* Drawback of sigmoid & tanh is that if the activation fun^n value is very large or very small, the slope ≈ 0, thus learning becomes very slow.

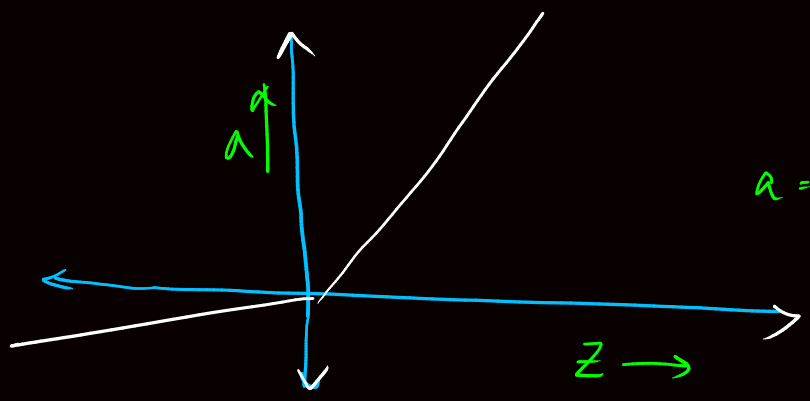
ReLU (Rectified Linear Unit)



technically this point is undefined but in computer the value of this point accounts for 0.0000000.....1

* If our model is for binary classification, let the o/p layer activation fun^n by sigmoid & all the rest layers, ReLU must be the default choice.

Leaky ReLU

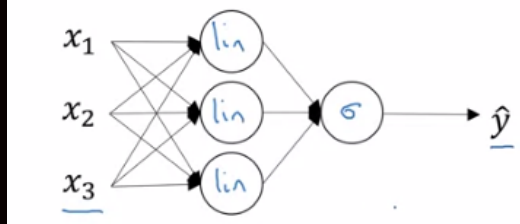


can be used as hyperparameter

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

Why non-linear activation fun^n?

* Combining 2 or more linear fun^n will in turn give an linear fun^n Thus no learning



Given x:

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

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

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

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

$g(z) = z$ "linear activation function"

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

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

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

$= (W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]})$

$= W'x + b'$

Andrew Ng

One place where we use linear activation fun^n are regression problem (ML)

Eg Predict house price

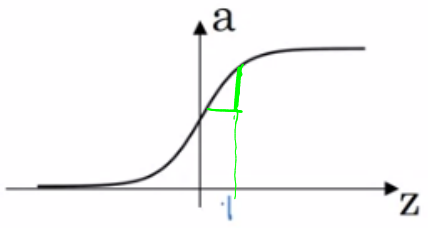
$y \in \mathbb{R}$ { \$0, ... \$100000

Use linear here

But all other hidden layers must use ReLU

Derivative of Activation fun^n

① Sigmoid



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

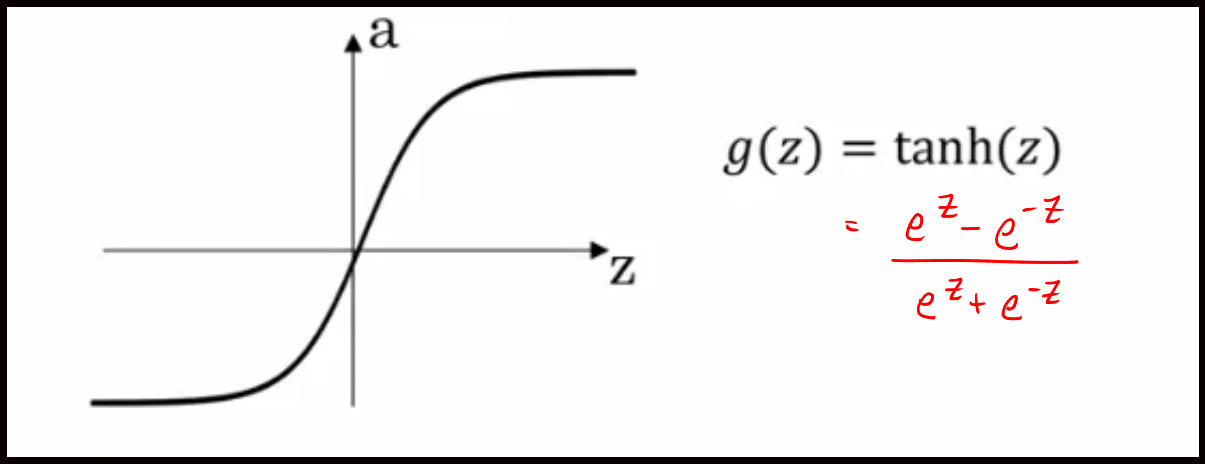
$g'(z) = \frac{d\,g(z)}{dz}$ = slope of $g(x)$ at z

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

$= a(1-a)$

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

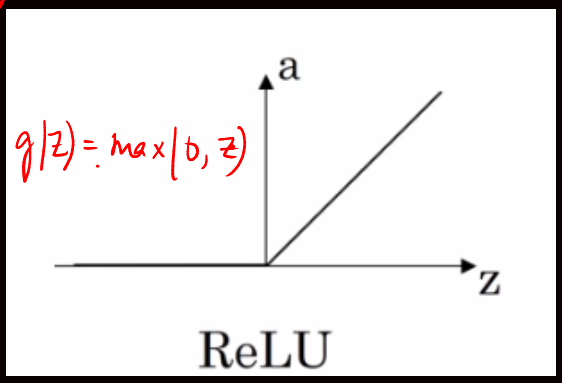
② Tanh



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

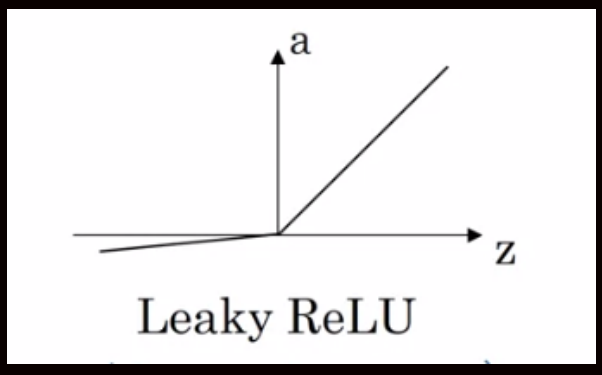
$$a = g(z) \Rightarrow g'(z) = 1 - a^2$$

③ ReLU



$$g'(z) = \begin{cases} 0 & \text{if } z<0 \\ 1 & \text{if } z>0 \\ 0.00\ldots1 & \text{if } z=0 \end{cases}$$

④ 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 Network

Parameters = $W^{[1]}$ $(n^{[1]} \times n^{[0]})$, $b^{[1]}$ $(n^{[1]} \times 1)$, $W^{[2]}$ $(1 \times n^{[1]})$, $b^{[2]}$ $[n^{[2]}, 1]$ $\quad n_x = n^{[0]}, n^{[1]}, n^{[2]} = 1$

Cost fun^n $= J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{n} L(\hat{y}, y)$ $\quad (\hat{y} \to a^{[2]})$

Gradient Descent

Repeat { (Till parameters converge)

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

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

③ $W^{[1]} = W^{[1]} - \alpha\, dW^{[1]}$; $b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$

$W^{[2]} = W^{[2]} - \alpha\, dW^{[2]}$; $b^{[2]} = b^{[2]} - \alpha\, db^{[2]}$

}

Formulas

Forward Propagation

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

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

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

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

Backward Propagation

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

$$dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$$

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

Automatically changes any rank 1 array to a vector

$(5,) \to (5,1)$

element wise product

$$dz^{[1]} = \underbrace{W^{[2]T} dz^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]'}\left(z^{[1]}\right)}_{(n^{[1]}, m)}$$

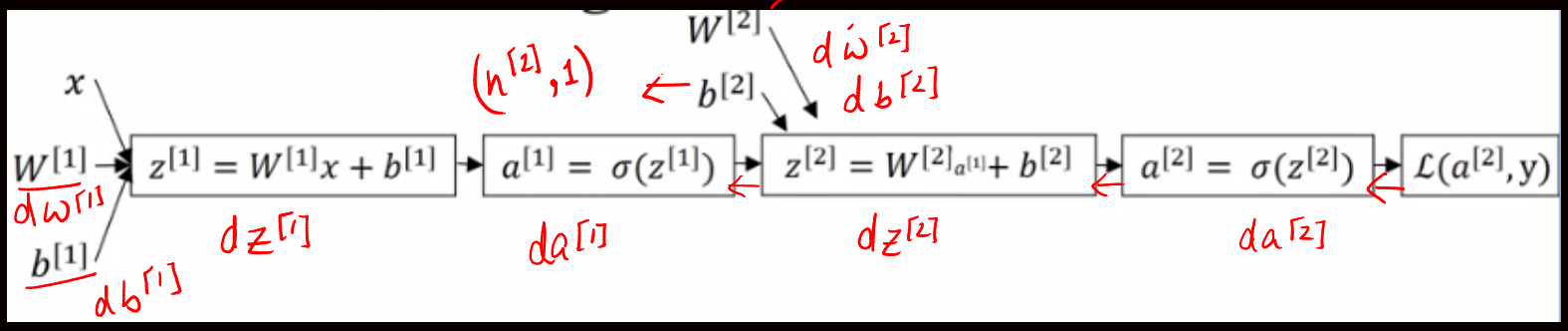
$$dw^{[1]} = \frac{1}{m} dz^{[1]} X^T$$

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

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

Intuition behind the Back Prop formulas

$W^{[2]}$ & $dw^{[2]}$ $(n^{[2]}, n^{[1]})$ → same dimension



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

$$dw^{[2]} = dz^{[2]} a^{[1]T} \approx \{ \text{Logistic Regn. } dw = dz\, x \}$$

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

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

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

$$dw^{[1]} = dz^{[1]} x^T$$

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

Summary of gradient descent

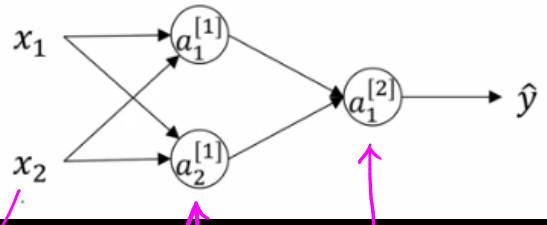
Vectorized Implementation

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

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$ $(n^{[1]},1)$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$ $(n^{[1]},m)$ $(n^{[1]},m)$ $(n^{[1]},m)$ — element-wise product
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

RANDOM INITIALIZATION

What happens if you initialize weights to zero?



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

let, $w^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ $b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Then $a_1^{[1]} = a_2^{[1]}$ and $dz_1^{[1]} = dz_2^{[1]}$

$dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$ $\quad W^{[1]} = W^{[1]} - \alpha\, dW = \begin{bmatrix} \text{——} \\ \text{——} \end{bmatrix}$

row 1 = row 2

No matter how many times we train, all the hidden units will be computing the same fun^n (all the hidden units will be symmetric

Solution

Initialize with random values:

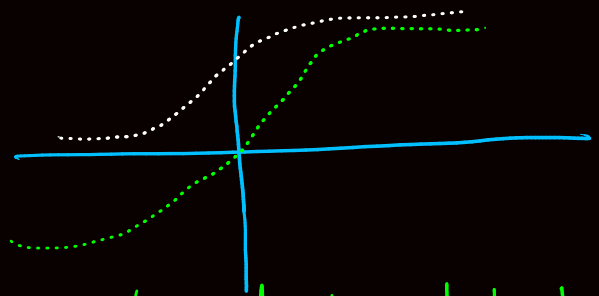
$W^{[1]} = np.random.randn((2,2)) \times 0.01$ * ↙ to initialize W by very small values *

$b^{[1]} = np.zeros((2,1))$ { No symmetry issue }

$W^{[2]} =$ . . . . . same as $W^{[1]}$

$b^{[1]} =$ . . . . . .

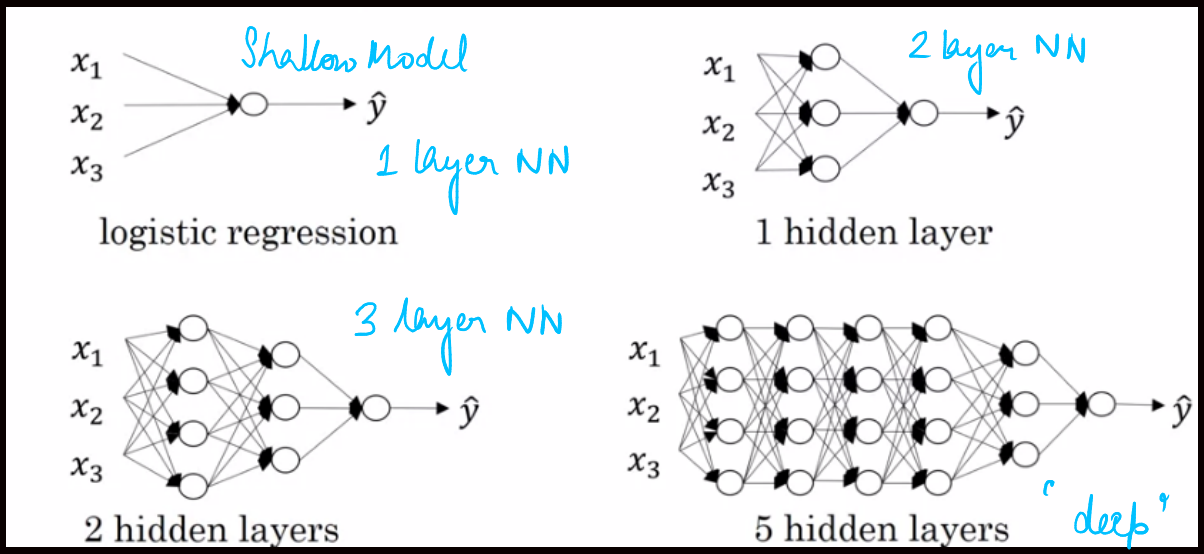
* If W are large $g(z)$ will be big & we will end up in a flat slope if we use tanh or sigmoid as $g(z)$ fun^n

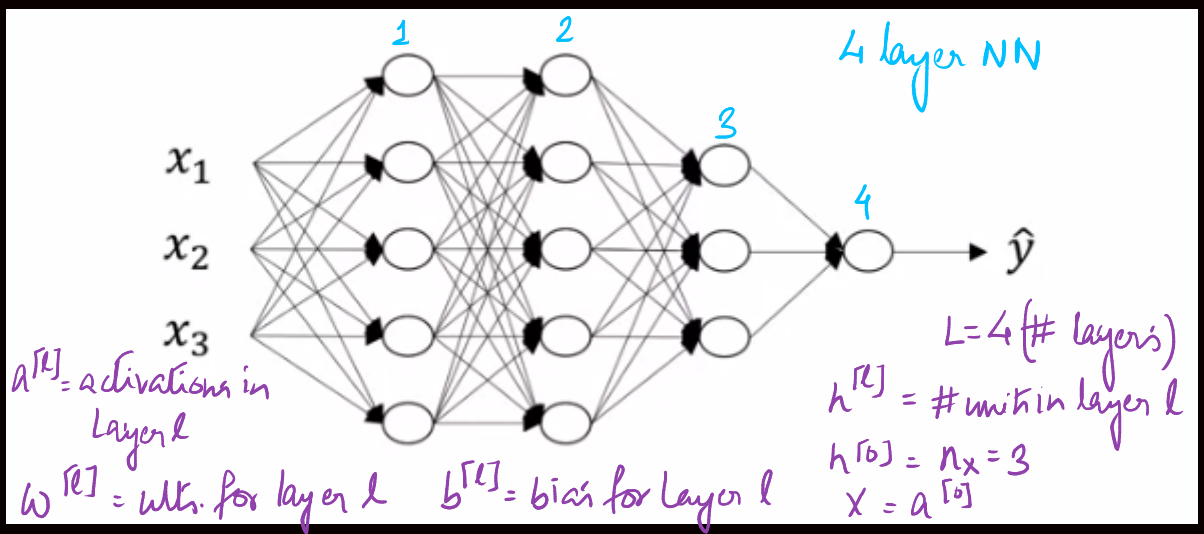


* 0.01 can be used as a hyperparameter

Week 4

Deep L-Layer neural network





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

Forward Propagation

$X: \; z^{[1]} = w^{[1]} X + b^{[1]}$

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

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$
$$\vdots$$
$$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$$

General = $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$
$a^{[l]} = g^{[l]}(z^{[l]})$

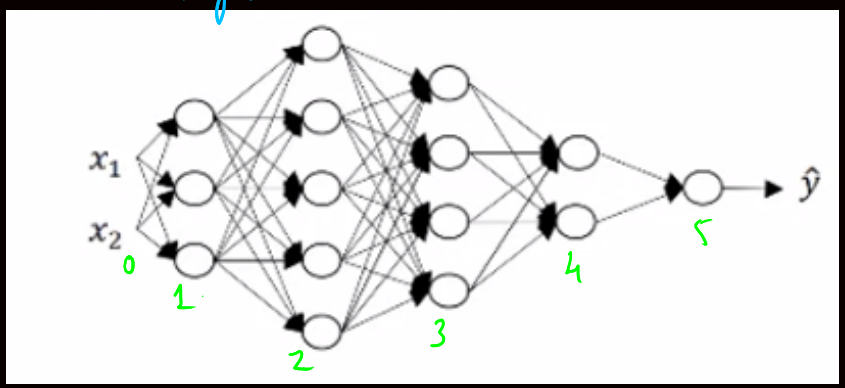
Vectorized

Capital Z as defined in previous slides $\longleftarrow$ $Z^{[l]} = W^{[l]} A^{[l-1]} + B^{[l]}$

$A^{[l]} = G^{[l]}(Z^{[l]})$

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

Getting the dimensions right



Eg. $z^{[1]}_{3\times1} = W^{[1]}_{(3\times2)} x_{(2\times1)} + b^{[1]}_{(3\times1)}$

$\vdots$

General

$w^{[l]} : (n^{(l)}, n^{(l-1)})$

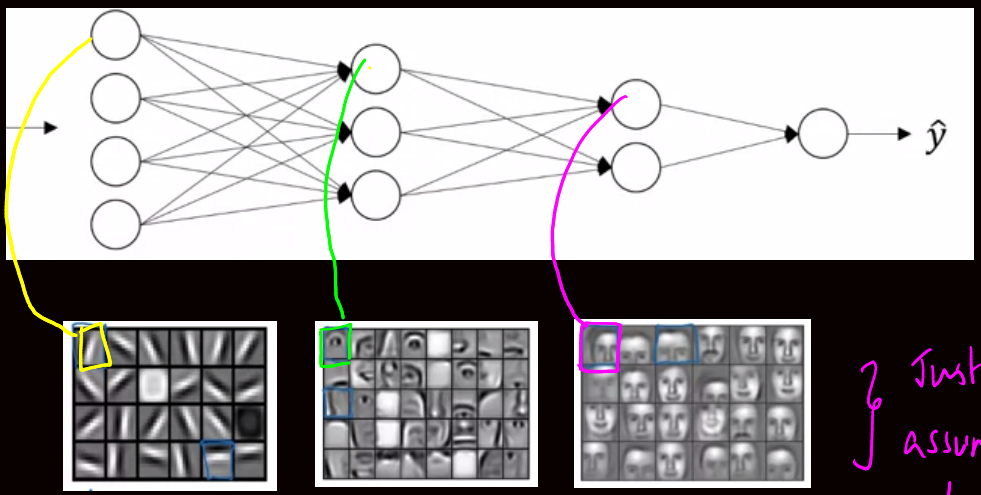
$b^{[l]} : (n^{[l]}, 1)$

$dw^{[l]}$: same as $w^{[l]}$

$db^{[l]}$: " " $b^{[l]}$

$z^{[l]}, a^{[l]} : (n^{[l]}, 1)$

9 Why deep representation ?



Just an arbitrary assumption about what a layer does

Audio Signal → low level sounds → Phonemes 'C' 'A' 'T' → Words → Sentences → find

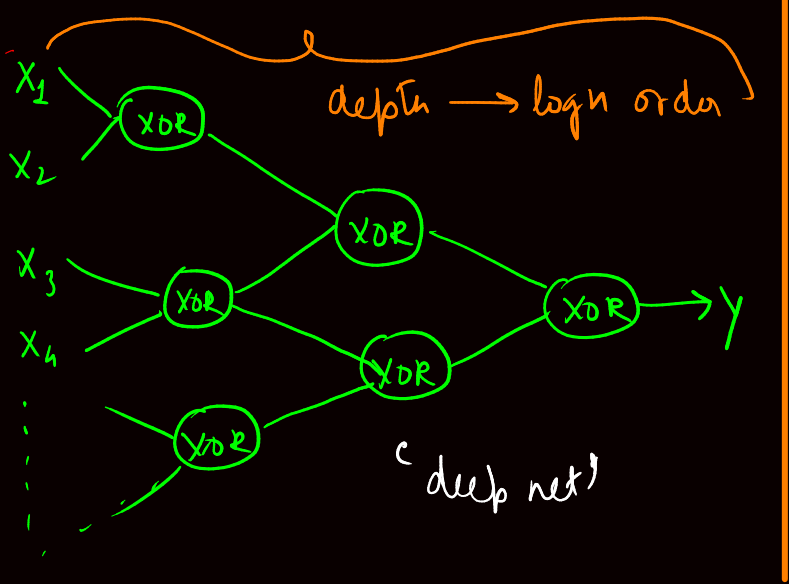
low level features → high level features

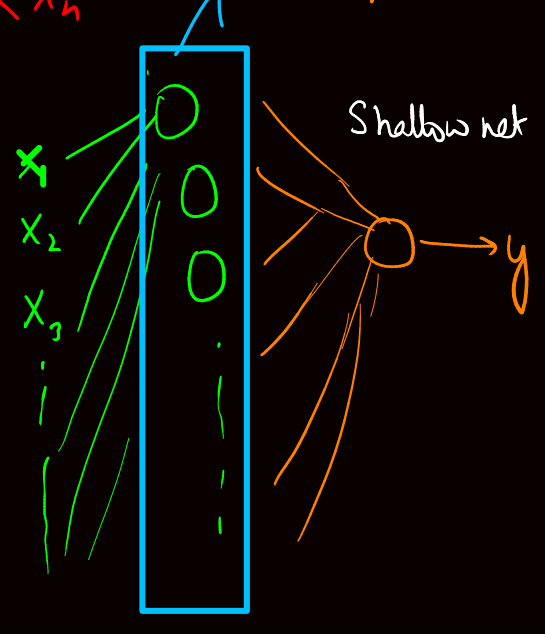
Circuit theory and deep learning

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

Ex

x_1 XOR x_2 XOR x_3 . . . . . XOR x_n

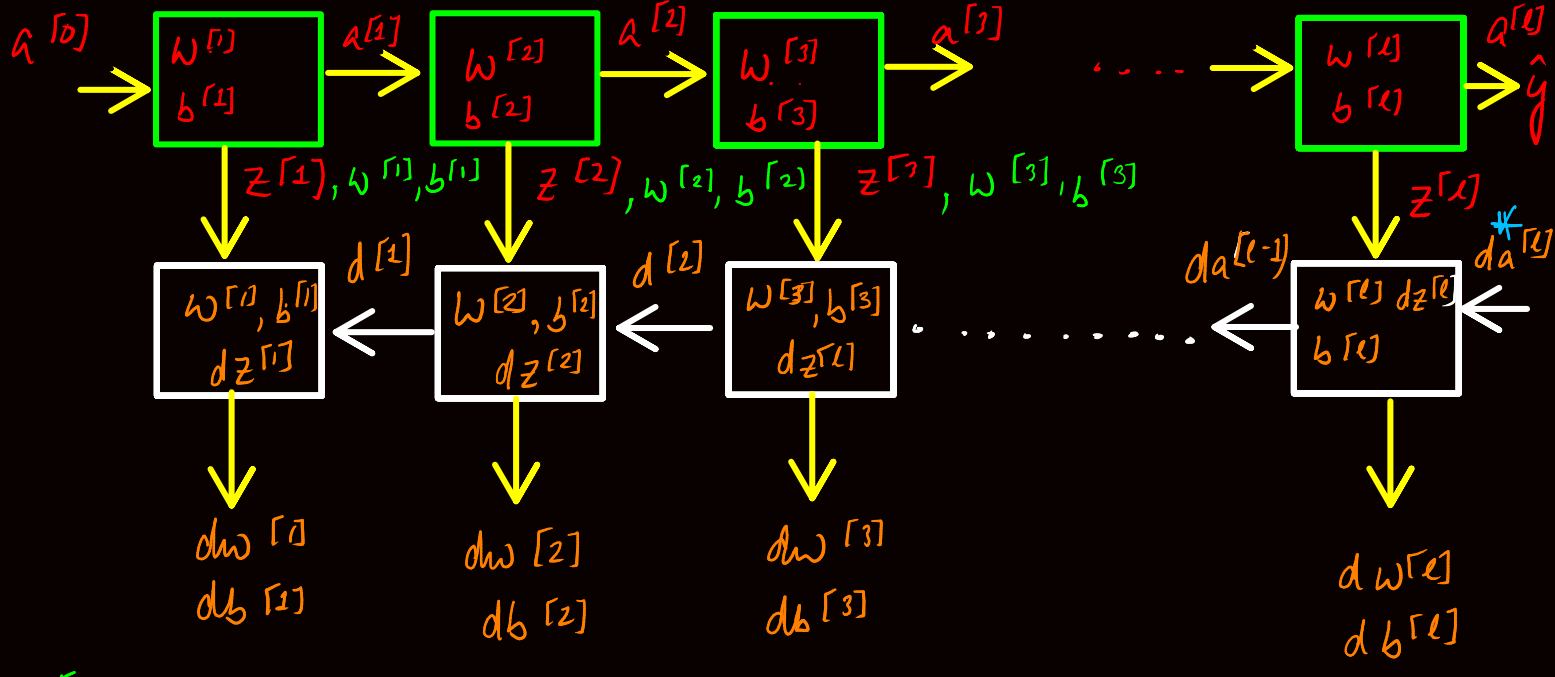




Building Blocks of Deep Neural Network

for logistic Regression

* $da^{[l]} = \frac{-y}{a} + \frac{(1-y)}{(1-a)}$



Formulas:

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

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

$$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$$

$$dw^{[l]} = dz^{[l]} * a^{[l-1]T}$$

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

$$da^{[l-1]} = w^{[l]T} dz^{[l]}$$

Parameters & Hyperparameters

PARAMETERS: $w^{[l]}, b^{[l]}$

HYPERPARAMETERS:

- Learning rate, α
- # iterations
- # hidden layer L
- # hidden units, $n^{[1]}, n^{[2]} \ldots$
- Choice of activation fun^n. etc.

Try with all types of hyperparameters. DL is an empirical process.