29/11/20
Target

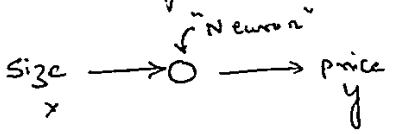
Deep learning . AI

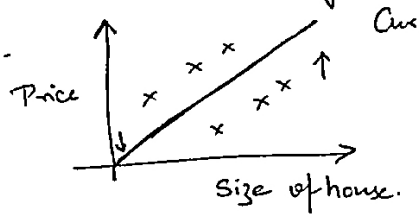
Coursera Specialization

* Courses:-
1. Neural Networks and Deep Learning
2. Im
3.
4.
5.

* Course 1 : Neural Networks & Deep Learning.

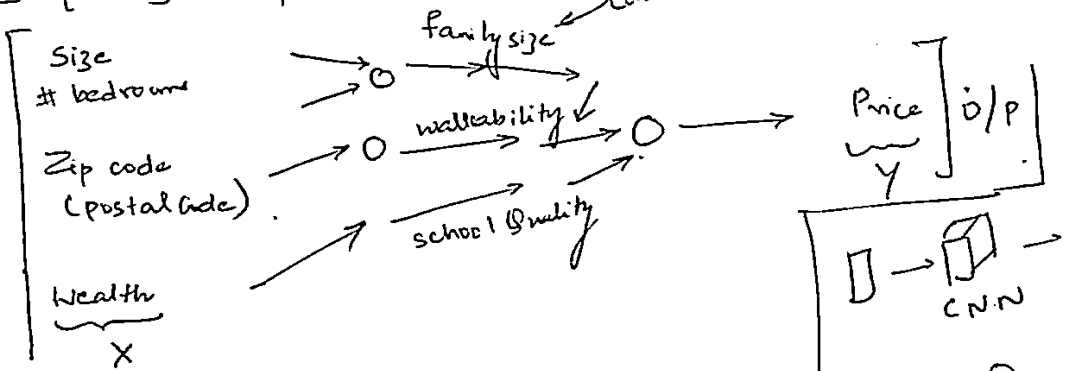
Housing price prediction :-

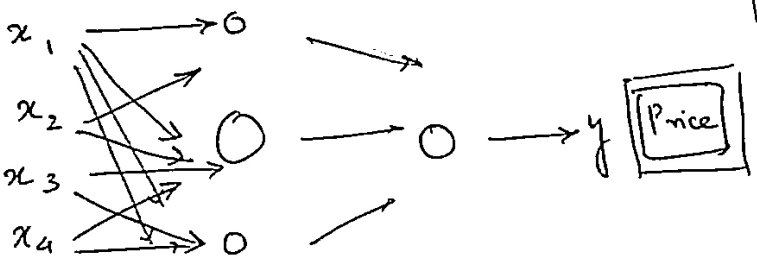




Curve fit linear regression.

[ReLU] : Rectified Linear unit.





II Supervised learning :

I/p(x)	O/p(Y)	Application	
Home feat..	Price	Real Estate	Standard NN
Ad, User info	Click	On! Adv	
Image	Obj (...)	Photo tagging	CNN (Convoluted)
Audio	Text transcript	Speech Recognition	Audi RNN (Recurred)
		Htc - translation	
Image/A-	--	Autonomous Driving	Custom/Hybrid

Supervised learning:

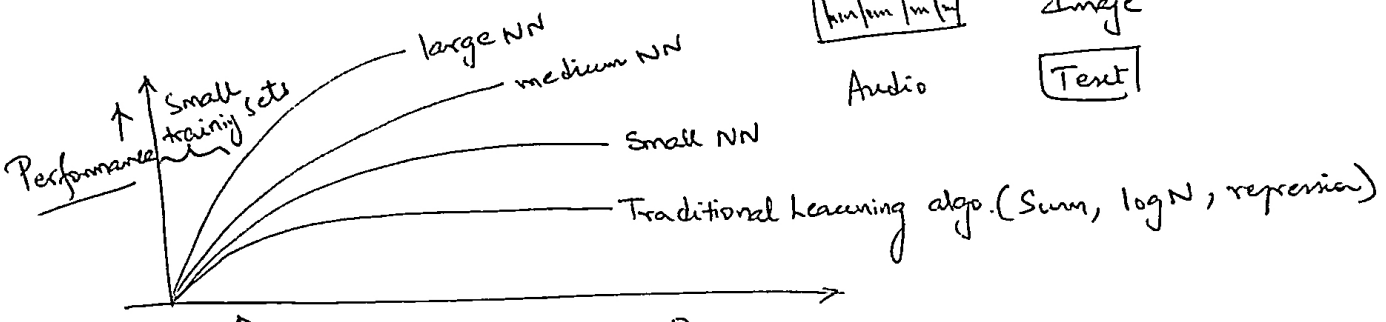
Structured data :

Unstructured Data

Image

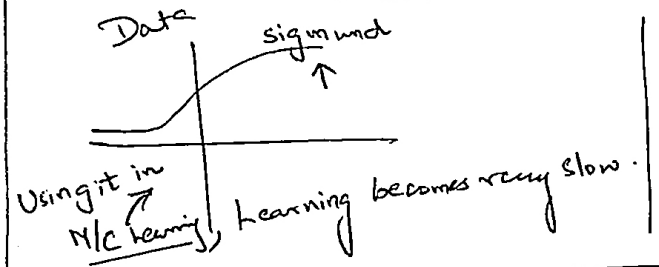
Audio

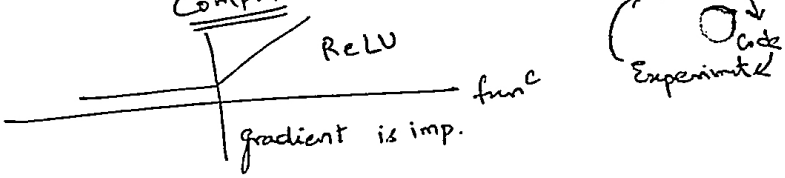
Text



Digitization of Data (Collecting More & More Data).

Performance increases with large NN. > medium > small NN.



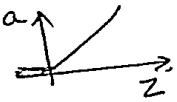


INTRODUCTION TO NEURAL NETWORKS.

1. Neural Networks & Deep learning

Week 1 : Intro.
Week 2 : Basics of NN programming
Week 3 : One hidden layer NN
Week 4 : Deep Neural Networks.

Quiz:

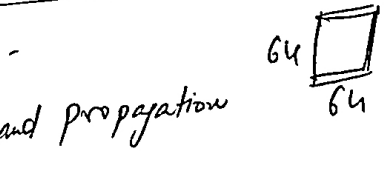
1. 'AI' is the new electricity :- Similar to electricity starting about 100 yrs ago, AI is transforming multiple industries.
2. Deep learning requires multiple iterations; train models, experience is 2nd
3. 

Increasing the size of NN, training set doesnot hurt algorithm performa & it may help significantly.

---

* Binary Classification :-

FPP / BPP : Forward / Backward propagation



$x = \begin{bmatrix} \vdots \end{bmatrix} = 64 \times 64 \times 3 = 12288$ $\quad \boxed{n = n_x = 12288}$

$(x, y) \quad x \in \mathbb{R}^{n_x}, \; y \in \{0, 1\}$ m training examples : $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})\}$

$X = \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(m)} \\ | & | & | & & | \end{bmatrix} \updownarrow n_x \qquad \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ & \vdots & \end{bmatrix}.$

$\leftarrow m \rightarrow$

$x \in \mathbb{R}^{n_x \times m}$ $\quad X.\text{shape} = (n_x, m)$; $\quad Y = [\,y^{(1)}\ y^{(2)} \dots y^{(m)}\,]$.

Logistic Regression

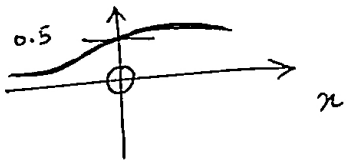
$y \in \mathbb{R}^{1 \times m}$

$y.\text{shape} = (1, m)$

Given X, want $\hat{y} = P(y=1|x)$ $\quad x \in \mathbb{R}^{n_x}$ (x dim vector). n_x

Par. $\boxed{w} \in \mathbb{R}^{n_x}$ $\quad \boxed{b} \in \mathbb{R}$.

Output $\boxed{\hat{y} = w^T x + b}$ $\quad 0 \le \hat{y} \le 1$



Sigmund func.: $\hat{y} = \sigma(w^T x + b)$

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

if z large $\sigma(z) \approx \frac{1}{1+0} = 1$.

if z large negative num, $\sigma(z) = \frac{1}{1+e^{-z}} \approx$

$\approx \frac{1}{1+\text{Big num}} \approx 0$.

alternative notation :-

$x_0 = 1$, $\quad x \in \mathbb{R}^{n_x+1}$, $\quad \hat{y} = \sigma(\theta^T x)$.

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{R_x} \end{bmatrix} \begin{matrix} \} b \leftarrow \\ \\ \\ \} w \leftarrow \end{matrix}$$

Summary: Parameters : $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$; $\hat{y} = w^T x + b$; $\hat{y} = \sigma(w^T x + b)$

$\hat{y} = \sigma(\theta^T x)$; $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_R \end{bmatrix} \begin{matrix} \} b \leftarrow \\ \} w \leftarrow \end{matrix}$

Logistic Regression Cost function

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

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$.

* loss (error) func: $\boxed{\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2}$ (efficiency of Alg.)

how good $\boxed{\hat{y}}$ is ?? given by

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

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

if $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$

* Cost function $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n} \ldots$

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

Summary:

∴ w, b minimize cost func^c;
loss func^c for efficiency of Algorithm

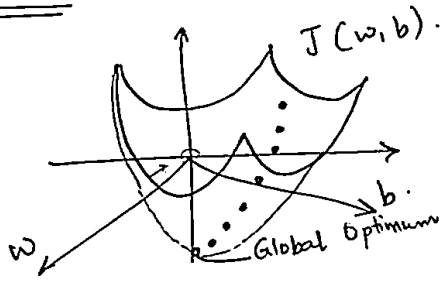
Logistic Regression

Note: The loss func^c computes the error for a single training example, the cost func^c is the average of the loss func^{es} of the entire training set.

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

Gradient Descent: (train / learn w, b on training set).

until now:



Cost func^c: $J(w,b)$

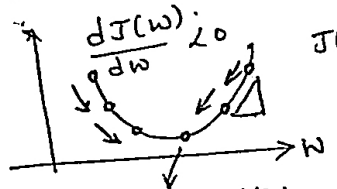
Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$.

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

$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)}).$$

∴ Want to find w, b that minimize $J(w,b)$??? *** ○

Convex Non-convex



Slope: dw

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

learning rate; update on w: "dw".

$w := w - \alpha\, dw$ Gradient Descent Update.

∂ = partial der.

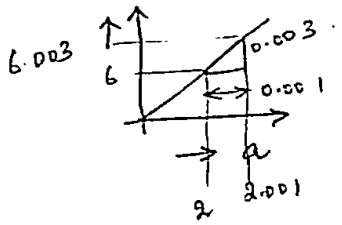
$$w := w - \alpha\frac{dJ(w,b)}{dw} \qquad \frac{\partial J(w,b)}{\partial w}\, dw$$

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

Update rule for Gradient Descent

∴ A Convex func^c has single local optima

Derivatives :-



$f(a) = 3a$

$a = 2, f(a) = 6$

$a = 2.001, f(a) = 6.003$

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

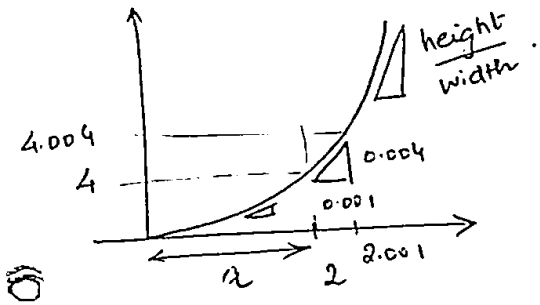
$a = 5.001 \quad f(a) = 15.003$

Slope at $a = 5$ is also 3

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

---

* Examples: Intuition about derivatives:



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

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

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

$\frac{d}{da} f(a) = 4$, when $a = 2$

$\because$ if $a = 5 \quad f(a) = 5$

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

$\frac{d}{da}(f(a)) = 10$ when $a = 5.$

$$\frac{d}{da} f(a) = \frac{d}{da} a^2 = 2a.$$

as $a \uparrow \quad \frac{d}{da} f(a) = \uparrow\uparrow\uparrow\ldots$

$f(a) = a^2 \quad \frac{d}{da} f(a) = 2a$

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

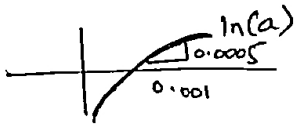
$a = 2.001 \quad f(a) \approx 4.004$

$f(a) = a^3 \quad \frac{d}{da} f(a) = 3a^2; 3 \times 2^2 \quad a = 2 \quad f(a) = 8$

$3 \times 2.001^2 \quad a = 2.001 \quad f(a) \approx 8.012.$

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

$\ln(a)$



$\frac{d}{da} f(a) = \frac{1}{a}$

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

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

$0.0005 \Big/ \frac{d}{da} f(a) = \frac{1}{2}$

$+0.0005$

Derivative is slope of a line;

* Slope is constant for linear lines whereas varies for exponential, logarithm.

---

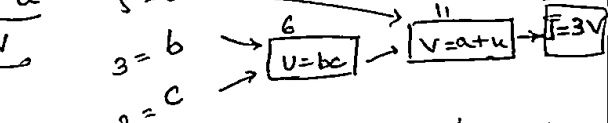
*

Computation graph :- FPP/BPP.

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

$\because u = bc$

$\underbrace{\quad\quad}_{J}$ $\underbrace{\quad\quad}_{V}$



$\because$

Note: One step of backward propagation on a computation graph yields derivative of final output variable.

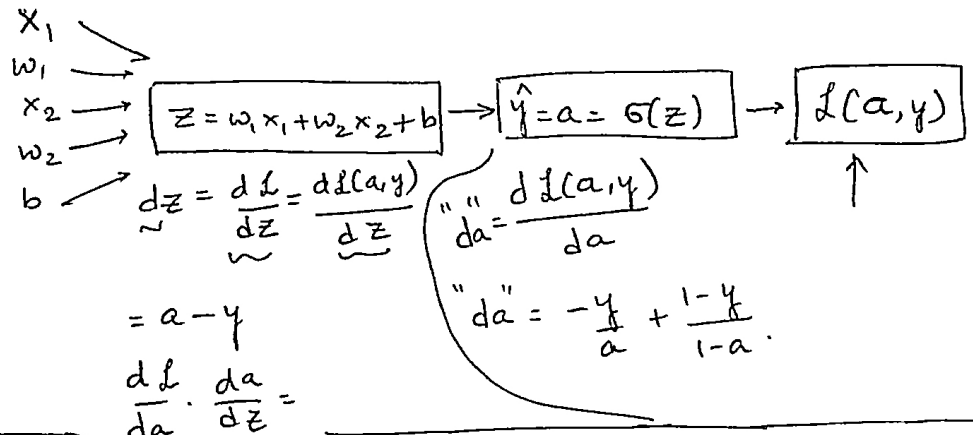
Derivatives with a computation graph:-

Logistic regression recap.

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

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

Loss fun^n: $\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$.



$\frac{d\mathcal{L}}{dw_1}$ "dw_1"

$= x_1 \, dz$

$dw_2 = x_2 \cdot dz$

$db = dz$

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

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

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

* Gradient Descent on 'm' Examples:

training example $(x^{(i)}, y^{(i)})$.

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

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

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

$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$

$dw_1^{(i)} - (x^{(i)}, y^{(i)})$

Lets say

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

$db = 0$.

for $i = 1$ to m

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

$dz^{(i)} = a^{(i)} - y^{(i)}$; $dw_1 += x_1^{(i)} dz^{(i)}$; $n = 2$;

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

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

$J /= m$; $dw_1 /= m$; $dw_2 /= m$;

$db /= m$.

$i = 1$ to m

2 for loops

dw_1, dw_2 .. features.

* $dw_1 = \frac{\partial J}{\partial w_1}$; $w_1 := w_1 - \alpha \, dw_1$; $w_2 := w_2 - \alpha \, dw_2$; $b := b - \alpha \, db$.

Vectorization °. to avoid for loops.

Time 12:00

* Derivation of dL/dz 8, 6, 10, 7, 9, 11, 6, 3, 7,

$$\frac{dL}{dz} = a - y$$

Deeplearning. AI.

Basics of Neural Network Programming

* Vectorization :-

$z = w^T x + b$.

if vectorize your code a Deep learning runs much faster

Normal	Vectorize
	a = np.random.rand(1000000)
	b = "
tic = time.time()	tic = time.time()
for i in range(1000000): c += a[i] * b[i]	c = np.dot(a,b)
toc = time.time()	toc = time.time()
481.3110203 9 ms	1.5051 ms.

300 times faster than normal

GPU (Graphical P.U. has many cores & used for parallel computing (Simult. many threads)
CPU (few cores & less parallelization)

999. GPU vs CPU

GPU, CPU } SIMD - single instruction multiple data. [Note : if you use In-Built func it enables python to efficient Parallelism]

* Examples :-

AVOID Explicit "FOR" Loops.

$u = Av$, $u_i = \sum_j A_{ij} v_j$

Normal

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

Vectorize

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

* Vectors & Matrix valued functions

Apply the exponential operation on every element of a matrix/vector.

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

Normal

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

```
import numpy as np
u = np.exp(v)
np.log(v) ; np.abs(v), np.maximum(v,0)
v**2, 1/v.
```

∴ dw = np.zeros((n_x, 1)) (n_x dimensional vector).

dw += x^(i) dz^(i).

dw /= m

dw = dw/m

for m training examples **Vectorizing Logistic Regression**

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

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

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

$Z = np.dot(w.T, X) + b$ ("Broadcasting" in Python) $\quad (1,1)$

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

VECTORIZING LOGISTIC REGRESSION'S GRADIENT OUTPUT:-

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

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

$A = [a^{(1)} \cdots a^{(n)}]$. $\quad Y = [y^{(1)} \cdots y^{(m)}]$.

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

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

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

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

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

$= \frac{1}{m} [x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)}]$.

$n_x \times 1$.

```
dw = 0
dw += x(1) dz(1)
dw += x(2) dz(2)
...
dw /= m
```

```
db = 0
db += dz(1)
db += dz(2)
...
db /= m
```

Broadcasting in Python (faster).

```
A = np.array([[.,.,.],[....],[....]])
cal = A.sum(axis=0)
percentage = 100 * A/cal.reshape(1,4)
```

Matlab/Octave: bsxfun

$= \begin{bmatrix} | & | & | & | \end{bmatrix}$

0 axis ↓

→ axis 1.

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

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

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

(m,n) $\quad (1,3)$ but Python realizes it (m,n)

GENERAL PRINCIPLE of B.C

(m,n) $+ - * /$ $(1,n) \rightsquigarrow (m,n)$

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

$(m,1)$ $+ - * /$ $\mathbb{R} \rightsquigarrow (m,1)$ (100)

$(1,n)$ $+ - * /$ $\mathbb{R} \rightsquigarrow (1,n)$

A Note on python/numpy vector.

never use data structures

* a = np.random.randn(5) X ← Print(a.T) a transpose.

(5,) rank 1 array

np.dot(a, a.T) → value outerproduct of a vector

Use in NN

a = np.random.randn(5,1). → [ ] [ ] [...]

Note Never use rank 1 array.

Always use, vector

a = np.random.randn(5,1) → a.shape = (5,1) Column vector

a = np.random.randn(1,5) → a.shape = (1,5) row vector

Reshape an array

a = a.reshape((5,1)).

assert(a.shape == (5,1)) ← inexpensive assertion

Tour of J.N.ipy

Explanation of logistic regression Cost func

* y|x y given x. Note: Minimizing the loss corresponds with maximizing log P(y|x).

Exercises: Week 2

Q1: A neuron computes a linear func ($z = Wx + b$) followed by an activation func

We generally say that the output of a Neuron is a $g(Wx+b)$ where g is the activation func (Sigmoid, tanh, ReLU, ...).

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

Q2. Logistic Loss :- $\mathcal{L}^{(i)}(\hat{y}^{(i)}, y^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1-y^{(i)}) \log(1-\hat{y}^{(i)}))$.

Q3 :img = (32,32,3) array ≈ x = img.reshape((32*32*3, 1)). Column vector.

Q4 a = np.random.randn(2,3); b = " (2,1); c = a+b; c.shape = (2,3) [Broadcasting of b. to match a.]

Q5 a = np.random.randn(4,3) b = " (3,2); c = a*b; "Error" [The * operator in Numpy, indicates element-wise multiplication.

It is different from "np.dot()". If you would try "c = np.dot(a,b)" you would get c.shape = (4,2)

Q6. Suppose you have n_x input features per example. $X = [x^{(1)} x^{(2)} \dots x^{(m)}]$.
What is the dimension of X? $\boxed{(n_x, m)}$

Q7 a = np.random.randn(12288, 150)
b = " (150, 45). $c = np.dot(a,b)$; c.shape = (12288, 45).

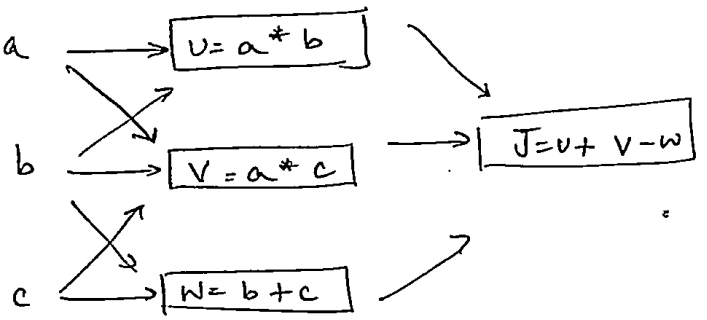
Q8. a.shape = (3,4).
b.shape = (4,1)

for 'i' in range (3):
  for j in range(4)
    c[i][j] = a[i][j] + b[j].

Vectorize $C = a + b^T$.

Q9. a = np.random.randn(3,3).
b = np.random.randn(3,1)
$c = a * b$
This will invoke broadcasting, so b is copied three times to become (3,3) and * is an element-wise product so c.shape will be (3,3).

10)



$\therefore J = (a-1) * (b+c)$.

* Programming exercise. Q. Build a fun^c returns $\boxed{\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}}$ Logistic fun^c

math.exp(x)

Note: We rarely use "math" library in deep learning because the inputs of the fun^c are real numbers. In deep learning we mostly use matrices and vectors.
This is why "numpy" is more useful.
Furthermore, if 'x' is a vector, then Py operation such as $s = x+3$ or $s = \frac{1}{x}$ will output s as a vector of the same size as x.

x = np.array([1,2,3]).

Note: ML/DL Normalize our data because Gradient Descent converges faster after Normalization.

* gradient of Sigmoid fun^c = Slope of Sigmoid fun^c.

$\frac{x}{\|x\|}$ ⇒ np.linalg.norm(a, axis=1, keepdims=True) $= \left[\sqrt{\sum}\right]$

Broadcasting changes matrix dimensions.

np.dot() → matrix-matrix multiplication
np.multiply() → element-wise multiply

Programming Exercise Part 2

Logistic Regression to identify cats.

Centre & Standardize your Data Set in Machine Learning

255 (maximum value of a pixel Channel).

* m_train images labelled as Cat ($y=1$) & non-cat ($y=0$)
* (num_px), (num_px, 3) 3 → 3 channels (RGB).
* Logistic regression is actually a very simple neural network
* $\omega^T x^{(i)} + b \mid \sigma$
* $$J = \frac{1}{m} \sum_{i=1}^{m} L(a^{(i)}, y^{(i)}).$$

The main steps for building a Neural Networks are:

1. Define the model structure (such as number of input features).
2. Initialize the model's parameters.
3. Loop:- Calculate current loss (forward propagation).
   :- Calculate current gradient (backward propagation).
   :- Update parameters (gradient descent).

You often build 1-3 separately and integrate them into one func^n we call model().

w = np.zeros((dim, 1))

Propagate(): w, b, X, Y.

$$A = \sigma(\omega^T X + b) = (a^{(1)}, a^{(2)}, \ldots a^{(m-1)}, a^{(m)})$$

$$J = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1-y^{(i)}) \log(1-a^{(i)}).$$

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A-Y)^T \qquad \frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}).$$

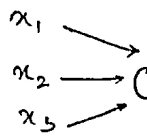
* $\theta = \theta - \alpha d\theta$. α = learning rate

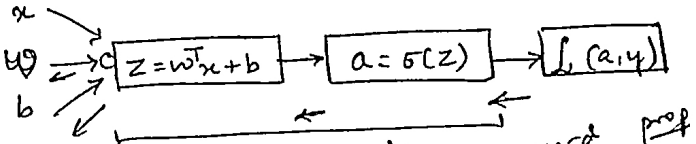
Glimpse

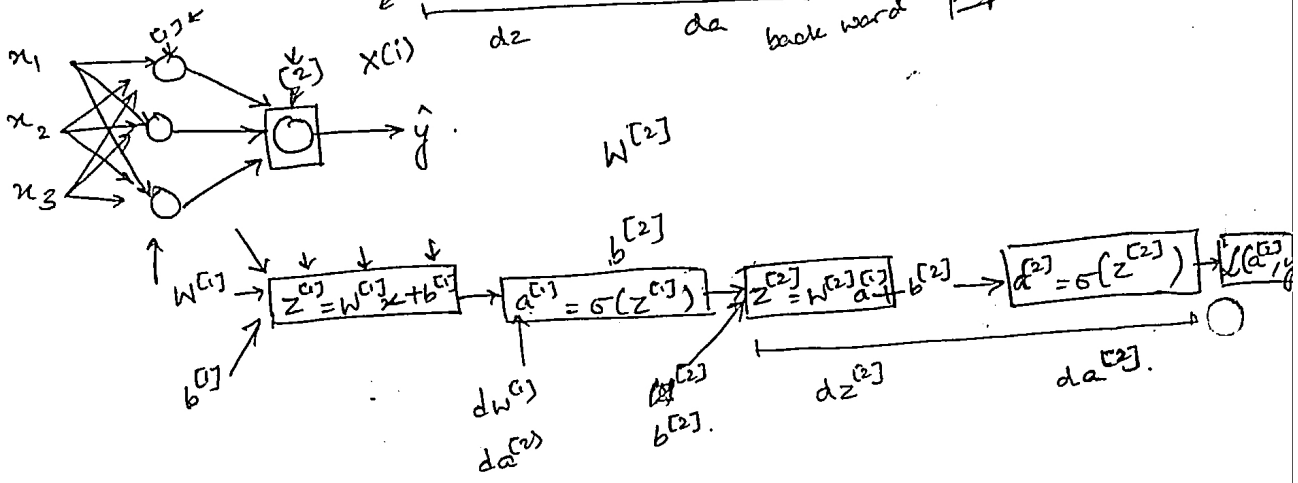
NN & DL

Week 3

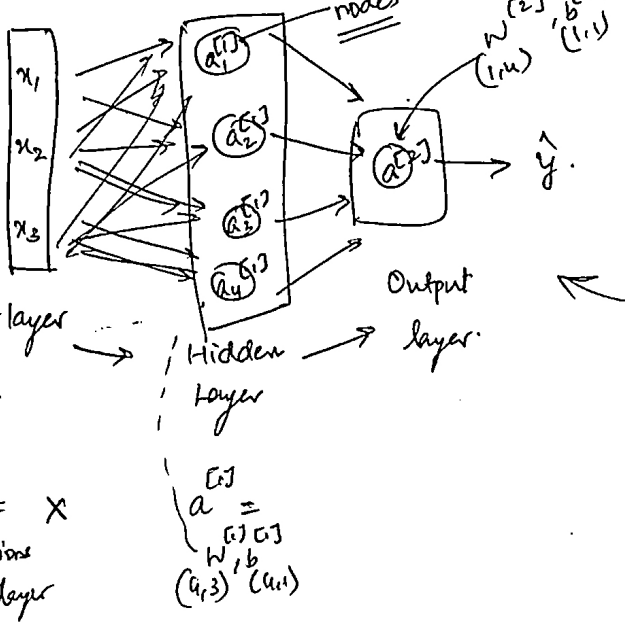
Neural Network Summary :



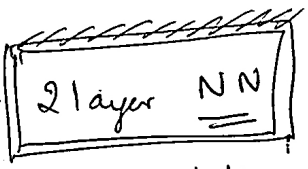




One Hidden Layer of Neural Network



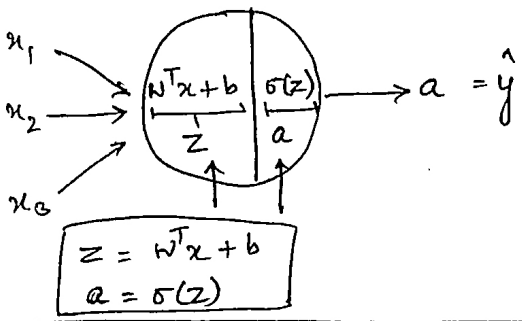
$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

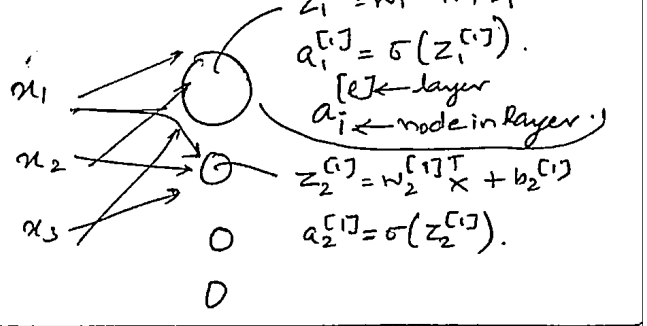


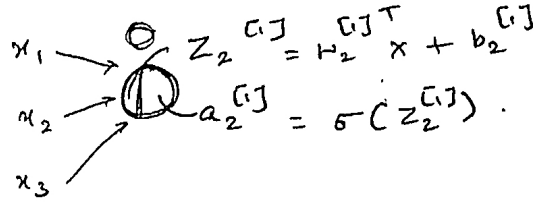
The input layer is not counted as an official layer

Computing a NN output : 2 steps

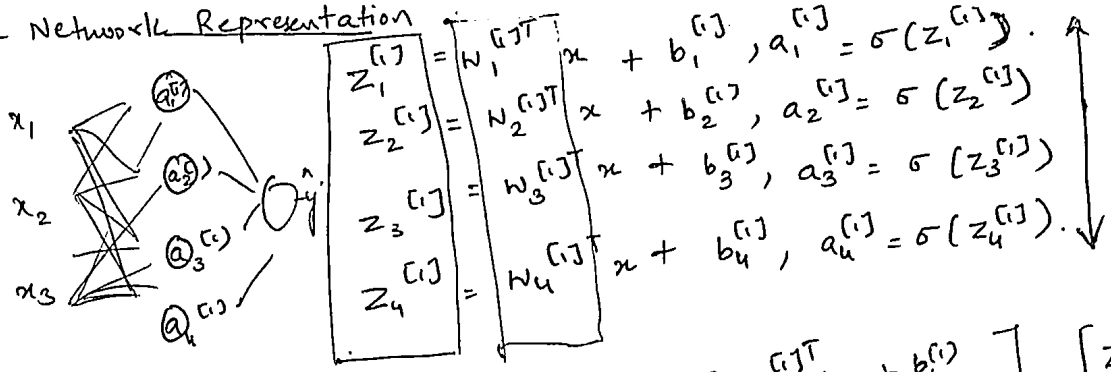
1. Compute z
2. Compute Activation(a) from sig fun of z







* Neural Network Representation



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

$(4,3)$ $(4,1)$

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

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

given input x:

$\rightarrow z^{[1]} = W^{[1]} x + b^{[1]}$
$(4,1)$ $(4,3)(3,1)$ $(4,1)$

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

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

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

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

7, 10, 11, 12, 13, 14

Vectorize across multiple examples :-

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

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

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

$$\begin{cases} z^{[1]} = W^{[1]} x + b^{[1]} \\ a^{[1]} = \sigma(z^{[1]}) \\ z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \\ a^{[2]} = \sigma(z^{[2]}) \end{cases}$$

for $i = 1$ to m,

$$\begin{aligned} z^{[1](i)} &= W^{[1]} x^{(i)} + b^{[1]} \\ a^{[1](i)} &= \sigma(z^{[1](i)}) \\ z^{[2](i)} &= W^{[2]} a^{[1](i)} + b^{[2]} \\ a^{[2](i)} &= \sigma(z^{[2](i)}) \end{aligned}$$

$a^{[2](i)}$ ← example i, ← layer 2

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x, m) ↑ training examples

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

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

hidden units.

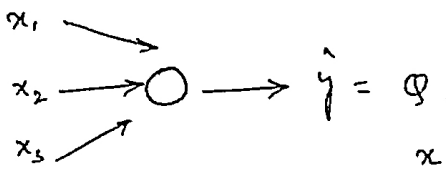
$$Z^{[1]} = W^{[1]} X + b^{[1]}.$$

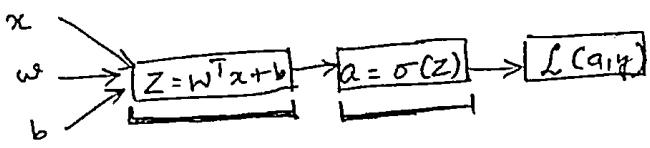
$$A^{[1]} = \sigma(Z^{[1]}).$$

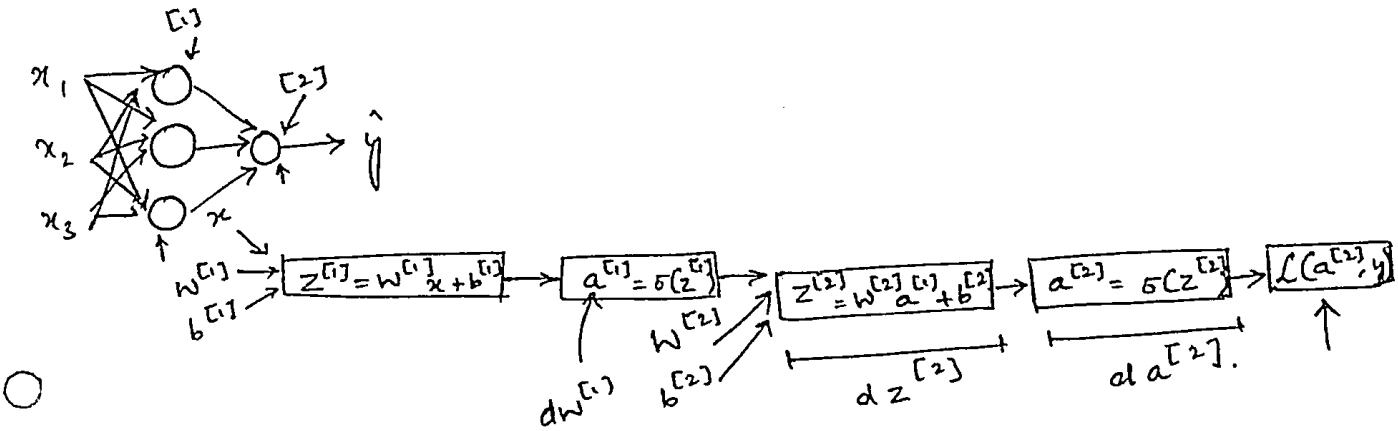
Explanation for Vectorized implementation

∵ first training example $z^{[1](1)} = W^{[1]} x^{(1)} + b^{[1]}$, $z^{[1](2)} = W^{[1]} x^{(2)} + b^{[1]}$.

$W^{[1]} = \begin{bmatrix} — \\ — \\ — \end{bmatrix}$.







Activation (Networks) functions:-

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

$a^{[1]} = \sigma(z^{[1]}) \rightarrow g(z^{[1]})$ until now Activation func is the sigmoid func

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

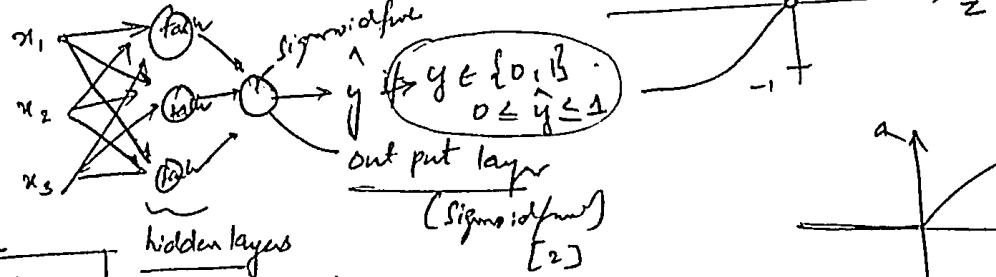
$a^{[2]} = \sigma(z^{[2]}) \rightarrow g(z^{[2]})$.

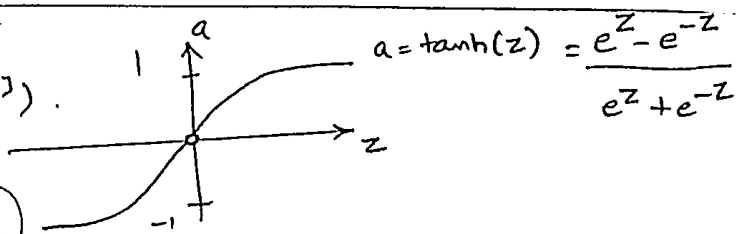
Nonlinear fun (Not sigmoid).

Better: tanh > sigmoid fun

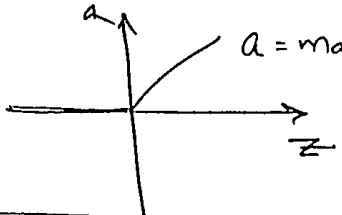
Now on Activation func is tanh func

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



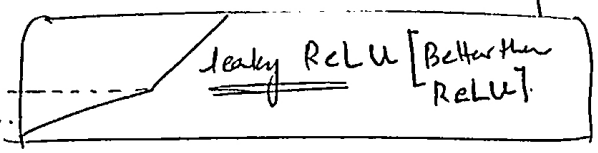


Note: hidden layers have data closer to zero mean so, tanh better for Output layer (Sigmoid) better.



ReLU: Rectified Linear Unit

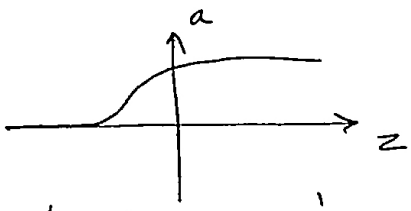
Default Choice of Activation fun for hidden layers if not sure frame... tanh Activation fun



NN is much faster when using ReLU Activation fun than tanh or sigmoid Activation fun

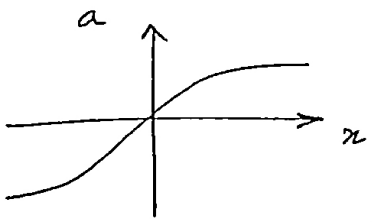
23, 24, 25, 26, 27, 28, 29, 30, 1, 2, 3, 4, 5, 6, 7.
15 days.

* Pros & Cons of Activation func.



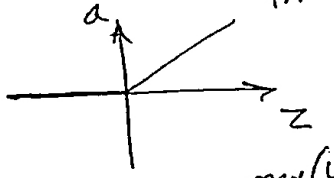
Sigmoid : $a = \frac{1}{1+e^{-z}}$

[Never for hidden layers & always for Output layers].



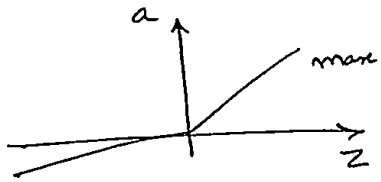
tanh: $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$.

[best for hiddenlayers]



ReLU $a = \max(0, z)$

(Default) for hidden layers.



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

[Sometimes better than ReLU].

* Why do u need a N.L Activation func.? :

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

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

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

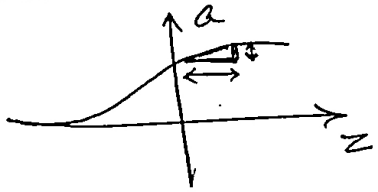
$g(z) = z$

"linear orientation function"

∵ A linear Hidden layer is more or less useless. N.L is better

* Derivatives of Activation Functions

Sigmoid



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

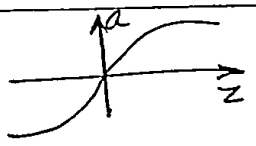
∵ $\frac{d}{dz}(g(z))$ = slope of $g(z)$ at z

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

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

$= a\,(1-a)$.

tanh



$g(z) = \tanh(z)$.

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

$g'(z) = \frac{d}{dz}(g(z))$

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

∴

$z = 10 \quad \tanh(z) \simeq 1$

$g'(z) \simeq 0$.

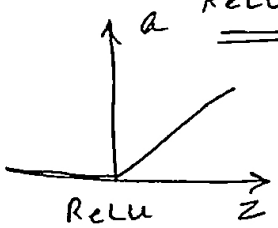
$z = -10 \quad \tanh(z) \simeq -1$

$g'(z) \simeq 0$.

$z = 0 \quad \tanh(z) = 0$

$g'(z) = 1$.

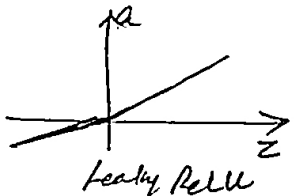
ReLU & Leaky ReLU — (3) —



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

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

$z = 0.000\ldots0$



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

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

Gradient Descent for Neural Networks :-

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

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

Cost fun^{n} : $J(w^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{n} \mathcal{L}(\hat{y}, y)$.

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

○ Gradient Descent :

Repeat- { Compute predicts $(\hat{y}^{(i)}, i = 1 \ldots, m)$.

$$dw^{[1]} = \frac{\partial J}{\partial W^{[1]}}, \quad db^{[1]} = \frac{\partial J}{\partial b^{[1]}}, \ldots$$

Update rule

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

Update Rule for Gradient Descent

○ Formulas for computing derivatives :

Forward Propagation :

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

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

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

$$A^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]}).$$

Correct vectorised Implementation of forward propagation for layer 'l'; $1 \leq l \leq L$:

$$z^{[l]} = w^{[l]} A^{[l-1]} + b^{[l]}.$$

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

Backward Propagation ○

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

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

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

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

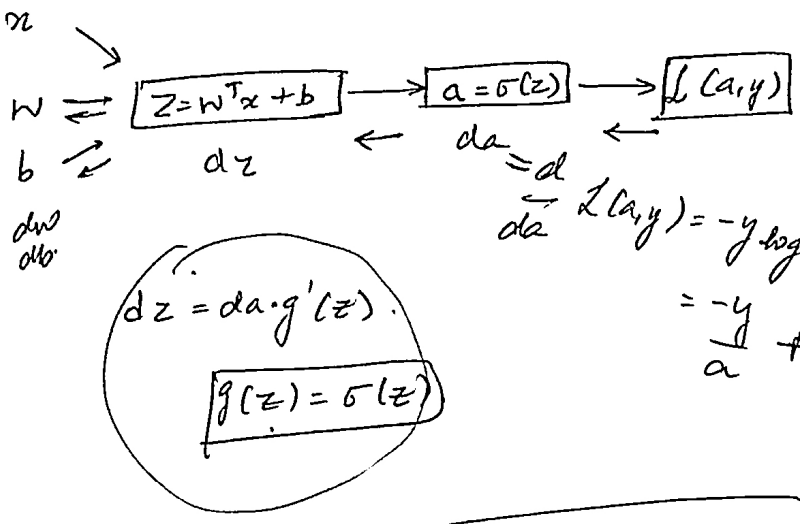
↑ element wise product $(n^{[1]}, m)$

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

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

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

Logistic regression

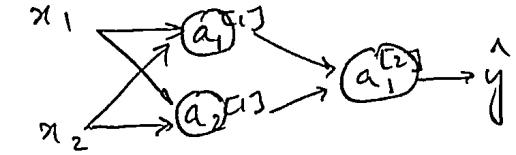


$$= -y\log a - (1-y)\log(1-a)$$

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

Random Initialization :-

* What happens if you initialize weights to zero?



$n^{[0]} = 2$ $\quad n^{[1]} = 2.$

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Weights. bias

$a_1^{[1]} = a_2^{[1]} \qquad dz_1^{[1]} = dz_2^{[1]}.$

Symmetric:-

$$dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix} \qquad \boxed{W^{[1]} = W^{[2]} - \alpha dW.} \qquad W^{[1]} = \begin{bmatrix} \cdots \\ \cdots \end{bmatrix}.$$

all the values of 'W'

∴ If you initialize with zero, W1 after every iteration, will have the first row equal to the second row. ∴ They all are computing the same thing

Initialize Randomly :-

$W^{[1]} = np.random.randn((2,2)) * 0.01$ — if it too large then ?!

$b^{[1]} = np.zero((2,1))$. ∵ Initialize to very small random values.

$W^{[2]} = \ldots$ ∵ W is random, so no longer there is a problem of symmetry breaking.

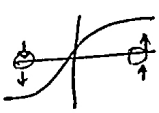
$b^{[2]} = 0.$

① Initialize parameters
② predict w/ forward prop.
③ Gradient Descent w/ Backward prop

if it is very large, 'W' will be large, correspondingly $Z^{[1]} = W^{[1]} x + b^{[1]}$

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

'Z' will be very large, or very small



the slope of the gradient will be very small so Gradient Descent will be very small & so learning slow

Week 3

Lab Planar_Data_Classification_with_one hidden_layer_v6c.

np.random.seed(1).

2-class dataset.

Mathematically,

For one example,

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

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

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

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

$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases}$

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(y^{(i)} \log(a^{[2](i)}) + (1-y^{(i)}) \log(1-a^{[2](i)}) \right).$$

General Met^4 to build NN

1. Define NN structure (# of inp units, # of hidden units, etc...)
2. Initialize the Model parameters.
3. Loop:
   - Implement forward propagation
   - compute loss
   - Implement backward prop. to get the gradients
   - Update parameters. (Gradient descent)

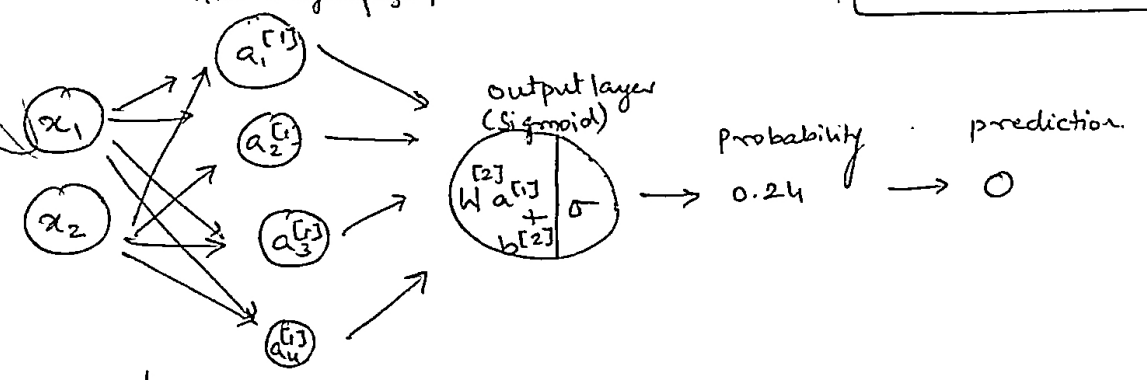
Define NN structure:

Define 3 variables:

- (n_x): the size of the input layer
- n_h: the size of the hidden layer (set this to 4)
- (n_y): the size of the output layer.

Use shapes of X & Y to find (n_x) and (n_y)

Also, hard code the hidden layer size to be 4.



sigmoid()

np.tanh()

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right)+(1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right)$$

Implement compute_cost() to compute the value of the cost J:

$$-\sum_{i=0}^{m} y^{(i)}\log\left(a^{[2](i)}\right):$$

```
logprobs = np.multiply(np.log(A2), Y)
cost = -np.sum(logprobs).
```

Hardest Part in Deep Learning is Backward Propagation

Summary of Gradient Descent:-

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

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

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

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]}).$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}.$$

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

- Functions Used
  - X, Y
  - * layer_sizes(X, Y)
  - * initialize_parameters(n_x, n_h, n_y)
  - * forward_propagation(X, parameters)
  - * compute_cost(A2, Y, parameters)
  - * backward_propagation(parameters, cache, X, Y)
  - * update_parameters(parameters, grads, learning_rate = 1.2)

Summary

Forward propagation

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

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

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

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

$$\vdots$$

$$A^{[L]} = g^{[L]}(Z^{[L]}) = \hat{y}$$

○

Backward Propagation

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

$$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]T}$$

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

$$dZ^{[L-1]} = W^{[L]T} \cdot dZ^{[L]} * g'^{[L-1]}(Z^{[L-1]})$$

$$\vdots$$

$$dZ^{[1]} = aW^{[2]} dZ^{[2]} * g'^{[1]}(Z^{[1]}).$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}.$$

Note that $A^{[0]T}$ is another way to . . . . See previous page.

Implementation info

Cost function:

Compute cost fun: to check if your model is actually learning.-

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

⇒ Backward Propagation: -

○ $$dW^{[L]} = \frac{\partial J}{\partial W^{[L]}} = \frac{1}{m} dZ^{[L]} A^{[L-1]T}.$$

$$db^{[L]} = \frac{\partial J}{\partial b^{[L]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[L](i)}.$$

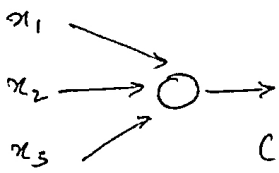
$$dA^{[L-1]} = \frac{\partial L}{\partial A^{[L-1]}} = W^{[L]T} \cdot dZ^{[L]}.$$

Week 4

Deep Neural Networks – Deep L-layer Neural Network

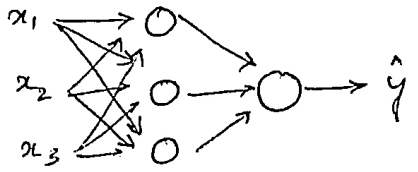
Deep Neural Network

"1 layer NN"



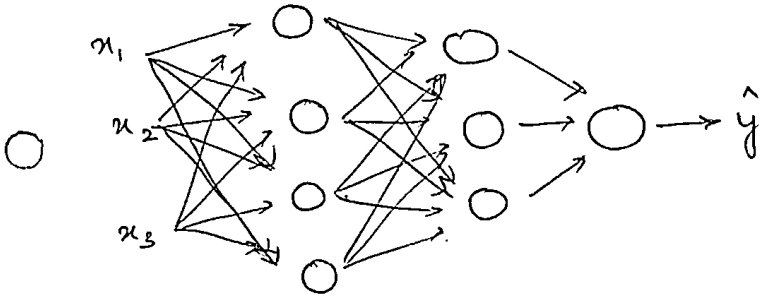
(shallow)

logistic regression "shallow"

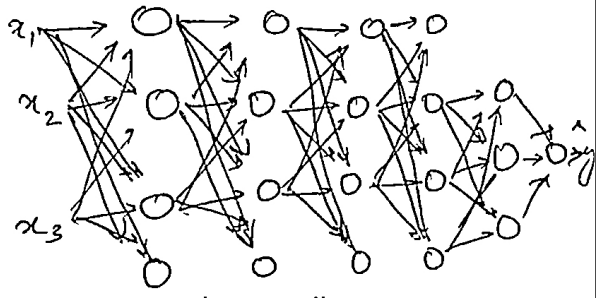


1 hidden layer.

2 Layer NN

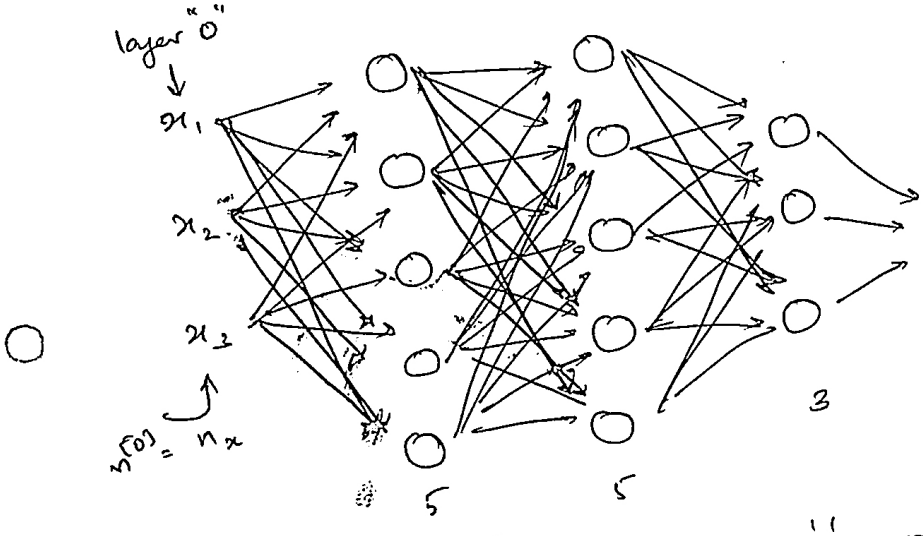


2 hidden layers.



"deep"

5 – hidden layers.



4 layer NN

$L = 4$ (# layers); $n^{[l]}$ = # units in layer 'l'.

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

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

$a^{[l]}$ = activations in layer 'l'.

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

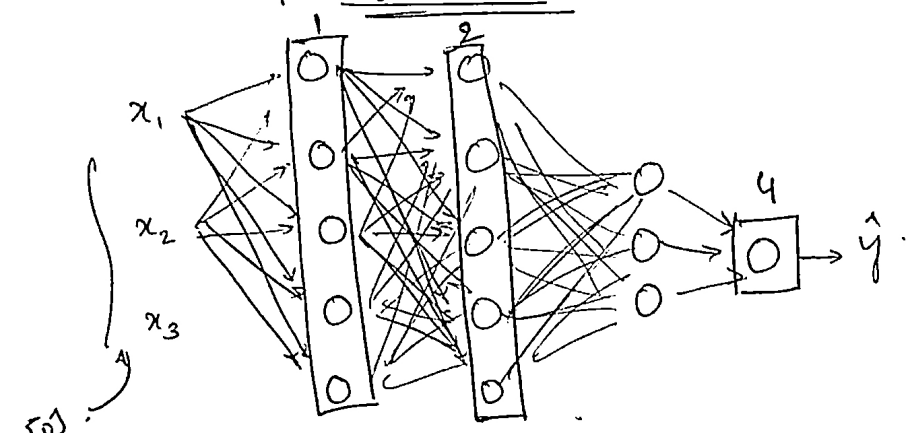
↳ Activation index

$b^{[l]}$ = biases " "

finally, $x = a^{[0]}$

↳ input layer & also activation at index 0.

* Forward propagation in a deep network



General Rule

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

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

General forward propagation Equ^n.

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

effect the activations in layer "1"

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

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

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

activation func^c

$$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}.$$

vector bias from [4] layer

Parameters from [4] layer

activation from previous layer

$$a^{[4]} = g^{[4]}(z^{[4]}).$$

Estimated Output $\hat{y}$ is computed.

∴ Vectorized -

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

$$z^{[1]} = W^{[1]} A^{[0]} + b^{[1]}$$

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

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

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

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

Seems like for $l = 1 \ldots 4$

But there is no way to compute for layers Acts 1, 2, 3 ... Perfectly OK for Explicit for loops.

Stacking columns for m training examples.

$$\left[z^{[2](1)} \quad z^{[2](2)} \quad \cdots \quad z^{[2](m)} \right] = Z^{[2]}.$$

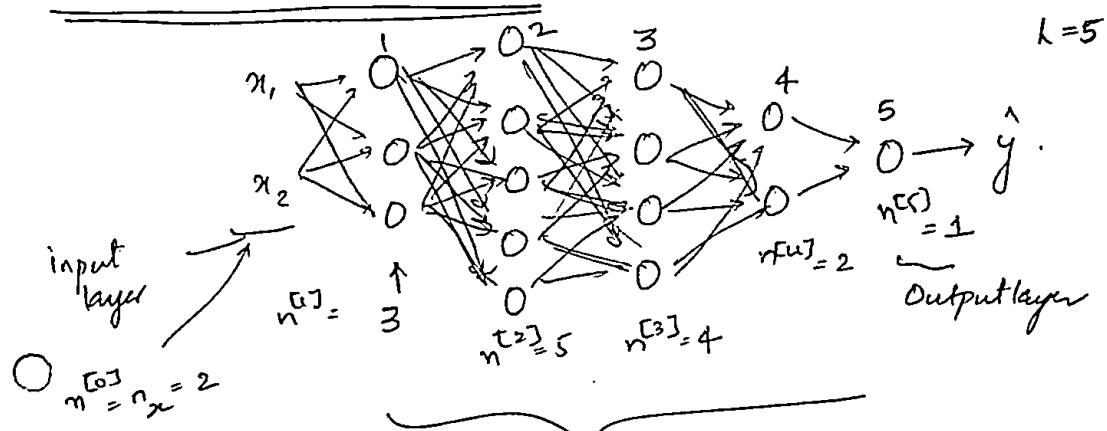
Correction for Video

$a^{[l]} = g^{[l]}(z^{[l]})$. a, z have dimensions $(n^{[l]}, 1)$.

Week 4-3

Getting your matrix dimensions right

Parameters $W^{[l]}$ and $b^{[l]}$



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

$(3,1) \leftarrow (3,2)(2,1) + (3,1)$.

$(n^{[1]}, 1) \leftarrow (n^{[1]}, n^{[0]})(n^{[0]}, 1) + (n^{[1]}, 1)$.

$\begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}_{3\times1} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix}_{3\times2} \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}_{2\times1} +$

from Matrix Multiplication

$W^{[1]} : (n^{[1]}, n^{[0]})$.

$W^{[2]} : (5, 3) : (n^{[2]}, n^{[1]})$

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

$(5,1) \quad (5,3) \quad (3,1) + (5,1)$. $(n^{[2]}, 1)$.

$W^{[3]} : (4,5) \quad (n^{[3]}, n^{[2]})$

$W^{[4]} : (2,4)$, $W^{[5]} : (1,2)$.

$W^{[l]} : (n^{[l]}, n^{[l-1]})$; $b^{[l]} : (n^{[l]}, 1)$ Dimensions of your matrices.

for Back propagation:

same as $dW^{[l]} : (n^{[l]}, n^{[l-1]})$.

$db^{[l]} : (n^{[l]}, 1)$. same as.

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

Vectorized Implementation :-

Normally (Non vectorized)

$$z^{[i]} = w^{[i]} \cdot x + b^{[i]}$$

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

for Vectorized (Stacking them).

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

$$z^{[i]} = w^{[i]} \cdot X + b^{[i]}$$

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

through Python Broadcasting

$(n^{[i]}, m)$

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

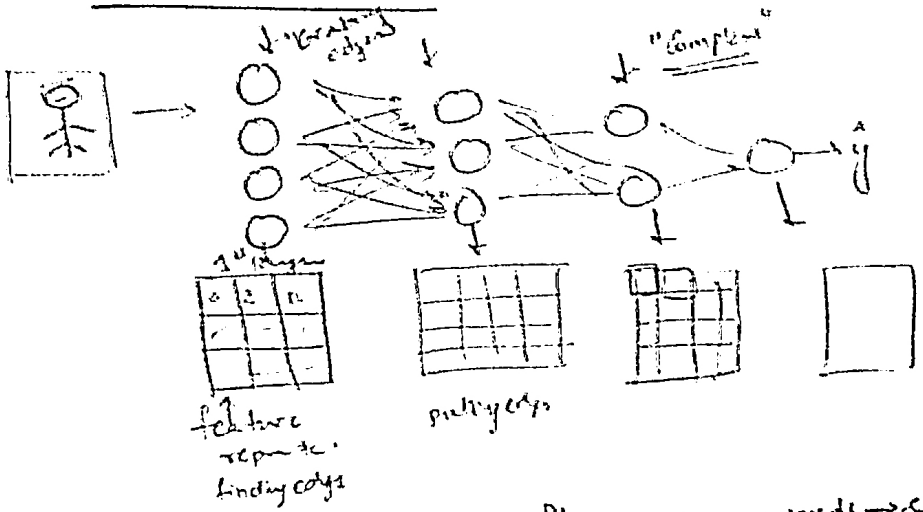
$Z^{[l]}, A^{[l]} : (n^{[l]}, m)$.

(capital (vectorized))

if $l = 0 \rightarrow A^{[0]} = X = (n^{[0]}, m)$

$dZ^{[l]}, dA^{[l]} : (n^{[l]}, m)$.

Intuition about deep representation



Audio → low level → Phonemes → words → sentence/phrases

Audio waveform

CAT

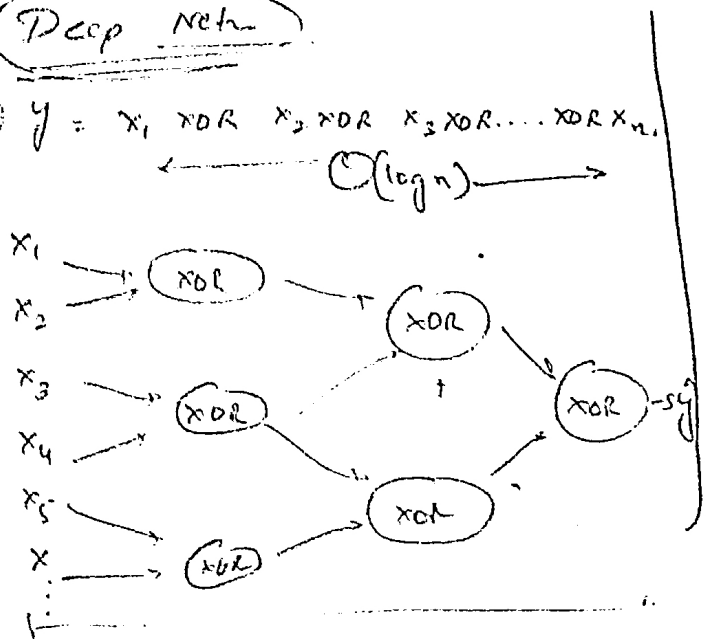
Circuit Theory and Deep learning

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.

Deep Neth

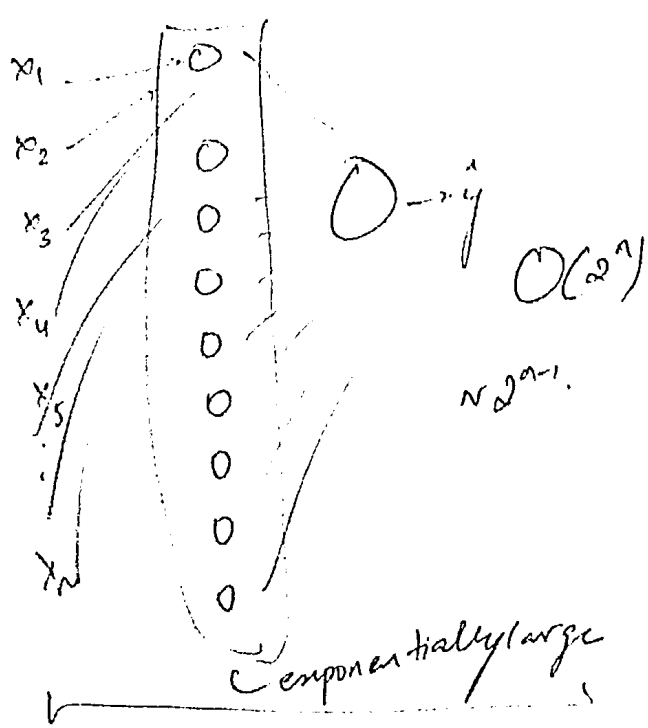
$y = x_1 \text{ XOR } x_2 \text{ XOR } x_3 \text{ XOR } \dots \text{ XOR } x_n$

$O(\log n)$



More efficient

esc: Shallower neth



* Building blocks of Deep Neural Networks:

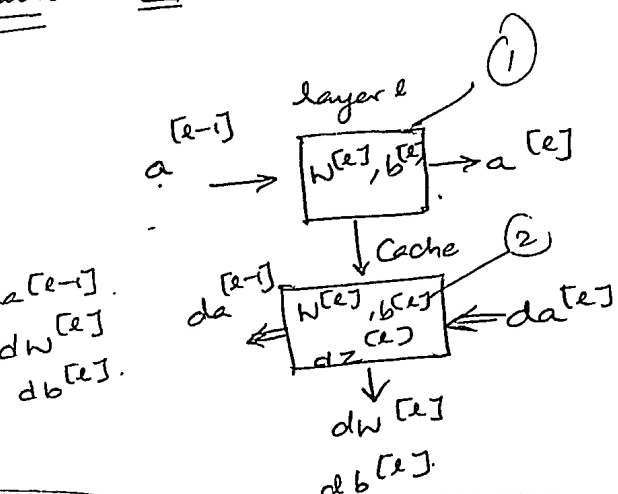
layer l: $W^{[l]}, b^{[l]}$

Forward: Input $a^{[l-1]}$, output $a^{[l]}$.

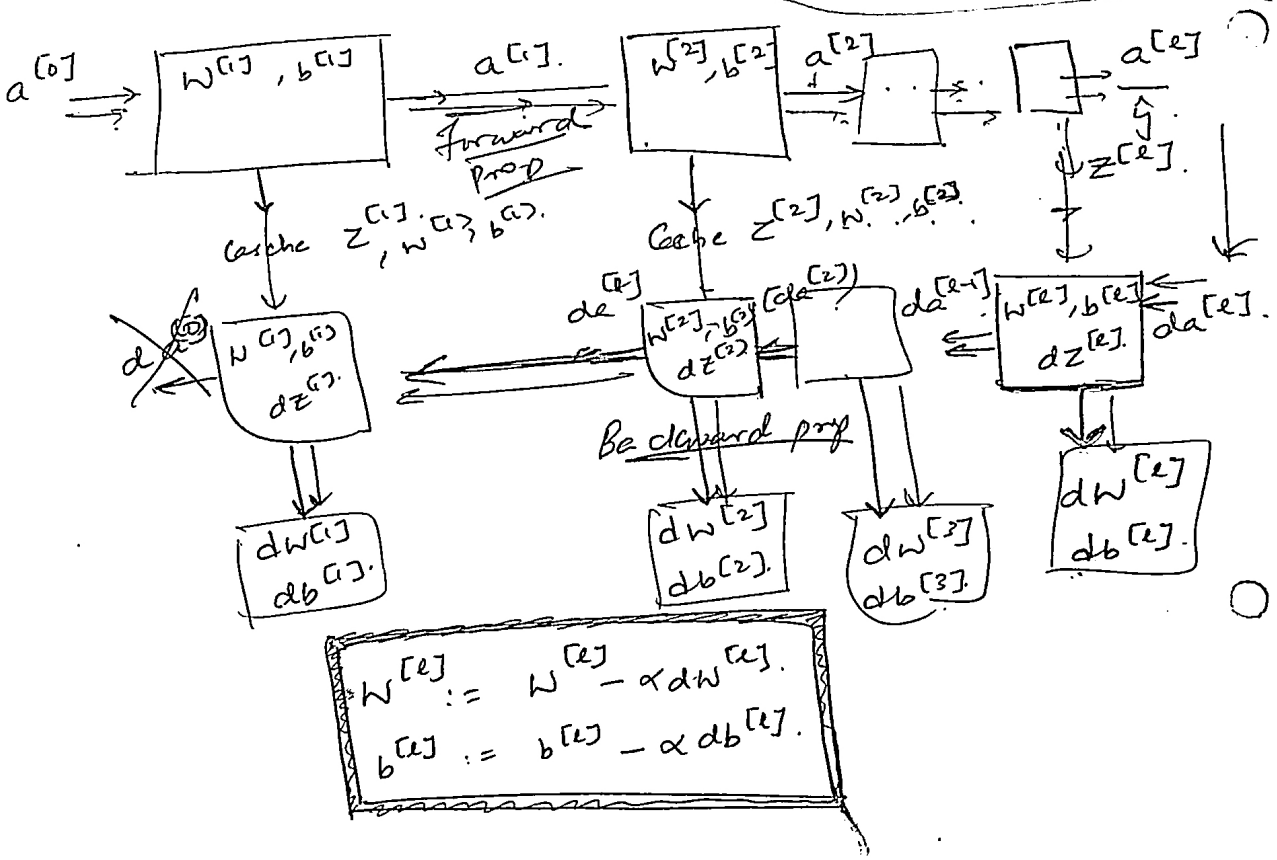
x_1, x_2, x_3, x_4 → $\hat{y}$

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$. Cache $z^{[l]}$.

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



Backward: Input $da^{[l]}$, output $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$. Cache $(z^{[l]})$



$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$

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

Correct Video

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

* Forward propagation for layer l

→ Input $a^{[l-1]}$.

→ Output $a^{[l]}$, Cache $(z^{[l]})$. ← $W^{[l]}, b^{[l]}$.

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

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

Vectorized

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

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

$X = A^{[0]} \rightarrow \square \rightarrow \square \rightarrow \square \rightarrow \cdots$

Backward Propagation for layer l:

○ → Input $da^{[l]}$.

→ Output $da^{[l-1]}, dW^{[l]}, db^{[l]}$.

eqn 4

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

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

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

$$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

Vectorized

$$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$$

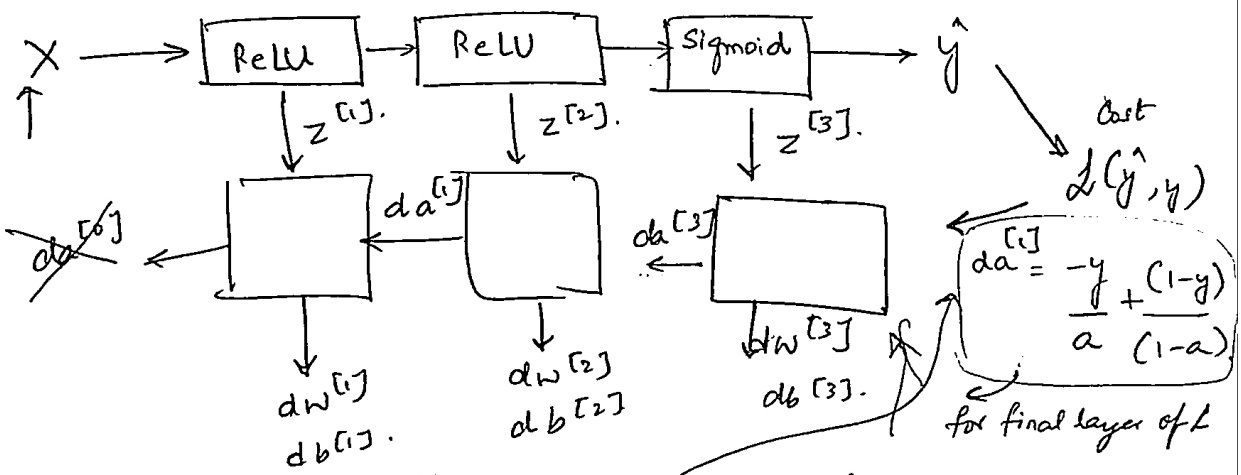
$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True)$$

$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$$

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

○

SUMMARY



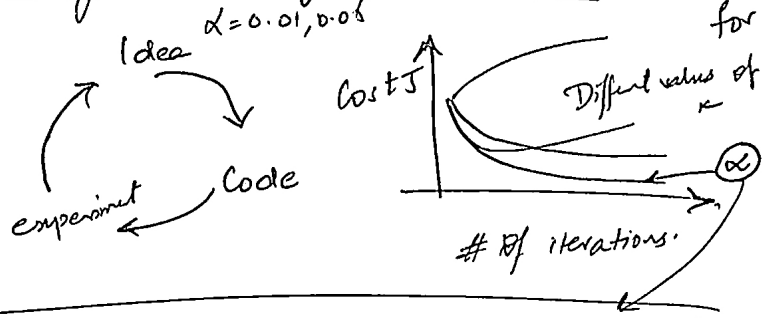
* Parameters Vs Hyperparameters:

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

Hyperparameters:
- Learning rate α
- # iterations.
- # hidden layers L
- # hidden units $n^{[1]}, n^{[2]} \ldots$
- Choice of activation function.

Later: Momentum, mini batch size, regularizations....

Note
* Applied deep learning is a very empirical process.



for checking out if the learning rate you think is the best value, try one value of α & see the graph

The value of α gives me pretty fast learning allows me to converge to a lower cost function.

(1) Try out many different values & check out which is best.

(2)

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

$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]T}$

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

$dZ^{[L-1]} = W^{[L]T} dZ^{[L]} * g'^{[L-1]}(Z^{[L-1]})$.

⋮ (*) element wise multiplication

$dZ^{[1]} = W^{[2]} dZ^{[2]} * g'^{[1]}(Z^{[1]})$

$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$

Note that $A^{[0]T}$ is another way to denote the input features, which is also written as X^T. $db^{[1]} = \frac{1}{m}$ np.sum$(dZ^{[1]}$, axis = 1, keepdims = True).