Name :- Atishay Jain

atishay

This notebook contains the notes of :-

→ Deeplearning.ai course 1 (neural networks)
→ " " " 2 (optimization, etc)
→ " " " 3 (structuring ml projects)
→ " " " 4 (CNN and computer vision, neural style, transfer)
+ (A section which further explains anchor boxes)

→ Notes from the API project
(syntax notes on pandas, numpy and matplotlib)

→ Re-inforcement learning

→ R programming

→ RNN's from o'reily book

9321 44884

name, emp no.,

Q1 Random forest algorithm.

→ It is an ensemble method (based on divide and conquer approach) of decision trees generated on a randomly split dataset.

→ This collection of ~~dece~~ decision tree classifiers is ~~also~~ also known as forest.

→ The individual decision trees are generated using an attribute selection indicators such as information gain, gain ratio, and gini index for each attribute.

→ Each tree depends upon an independent random sample.

→ In a classification problem, each tree votes and the most popular class is chosen as the final result.

→ In case of regression, the average of all the tree outputs is considered as the final result.

The algorithm works in 4 steps :-

1.) Select random samples from the given dataset.
2.) Construct the decision tree for each sample and get a prediction result from each decision tree.

3.) Perform a vote for each predicted result.

4.) Select the prediction result ~~which~~ with the most votes as final prediction.

Advantages:-

→ Highly accurate as a lot of decision trees participate in the process.

→ Does not suffer from overfitting as the average of all the predictions are taken which cancels out the biases.

→ The algorithm can be used for both classification and regression problem.

→ We can get relative feature importance.

→ Random forests can also handle missing values. There are two ways to handle these:-

(i) Using median values the repl are continuous variables.

(ii) Computing the proximity-weighted average of missing values.

Disadvantages:-

→ Random forest is slow in generating predictions because it has multiple decision trees and it also has to perform

voting.

→ The model is difficult to interpret as compared to decision tree, where you can easily make a decision by following the path in the tree.

Deep learning

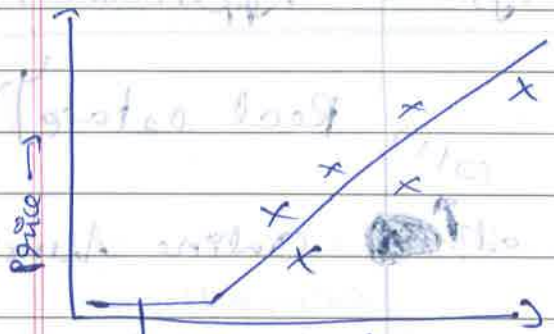
Part - 1

(1) Neural network and deep learning

(1.1) week (1)

(What is neural network)

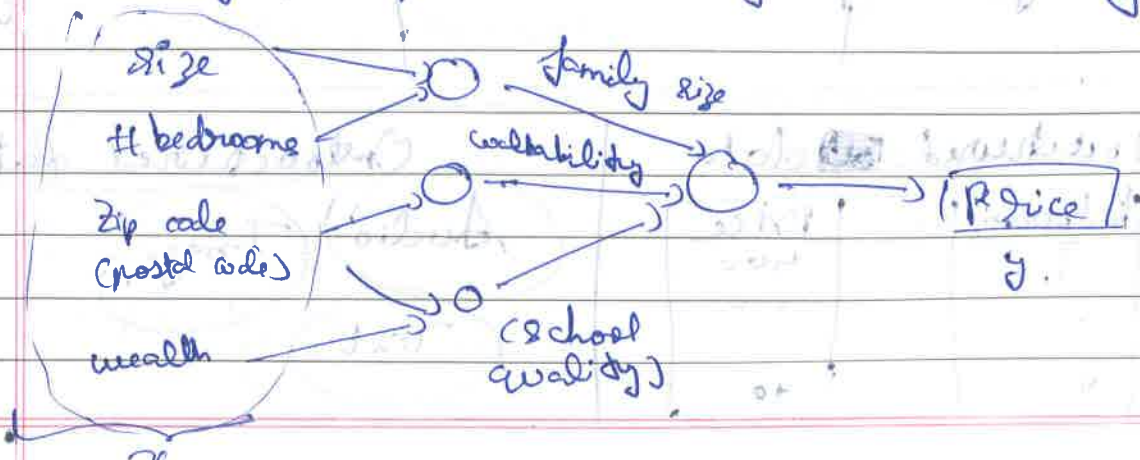
→ Housing price prediction.

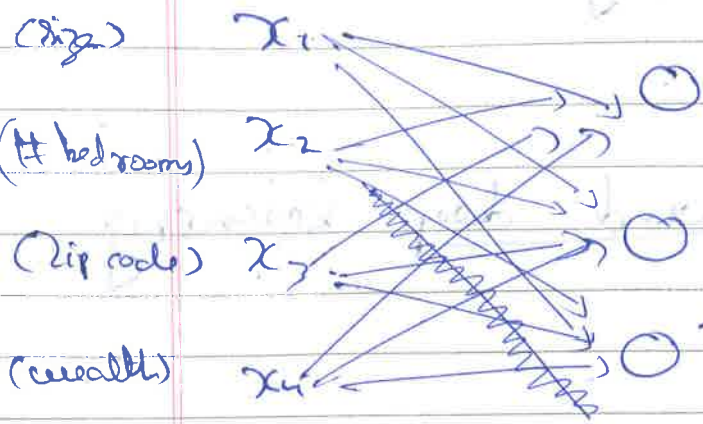


↳ this function is called Relu function

↳ Rectified linear unit

→ Housing price prediction using more than 1 features





→ Supervised learning.

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

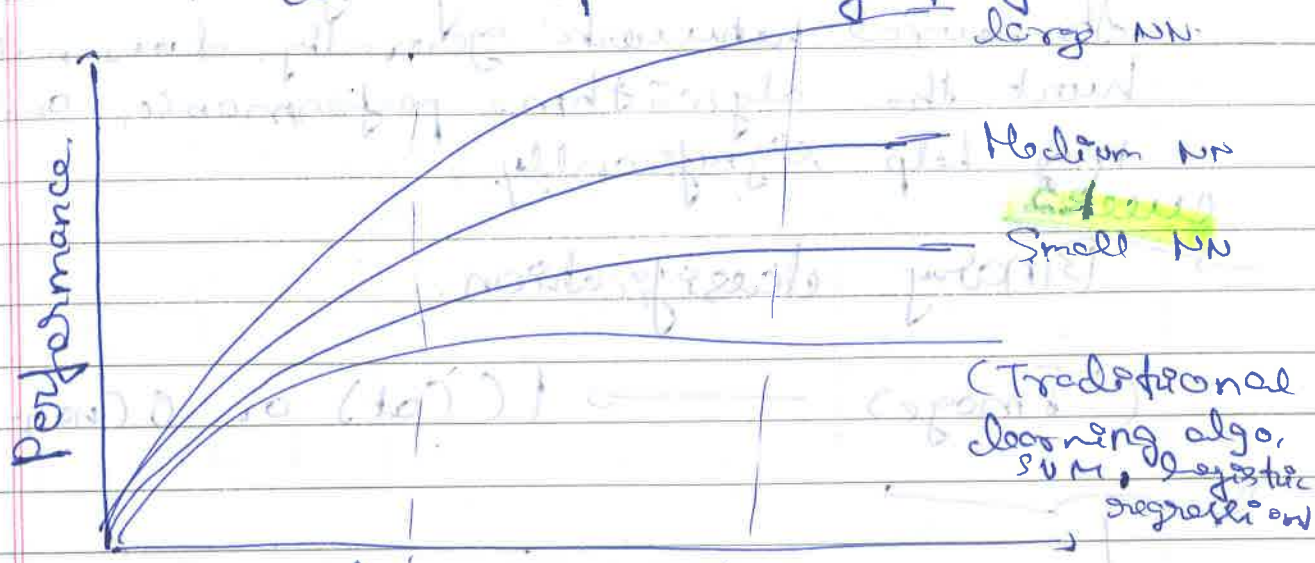
Structured data

Size	bedroom	...	Price
2104	3		400
1600	3		330
⋮	⋮		
3000	4		540

Unstructured data

Audio, Image, Text

→ Scale drives deep learning progress



l network

Small training sets → large NN may not dominate small NN, or traditional learning algos.

Her larger NN reach better consistently

Thus the reason that deep learning is taking off :-

(i) Deep learning has resulted in significant improvements in important applications such as online advertising, speech recognition and image recognition.

(ii) We have access to a lot more computational power

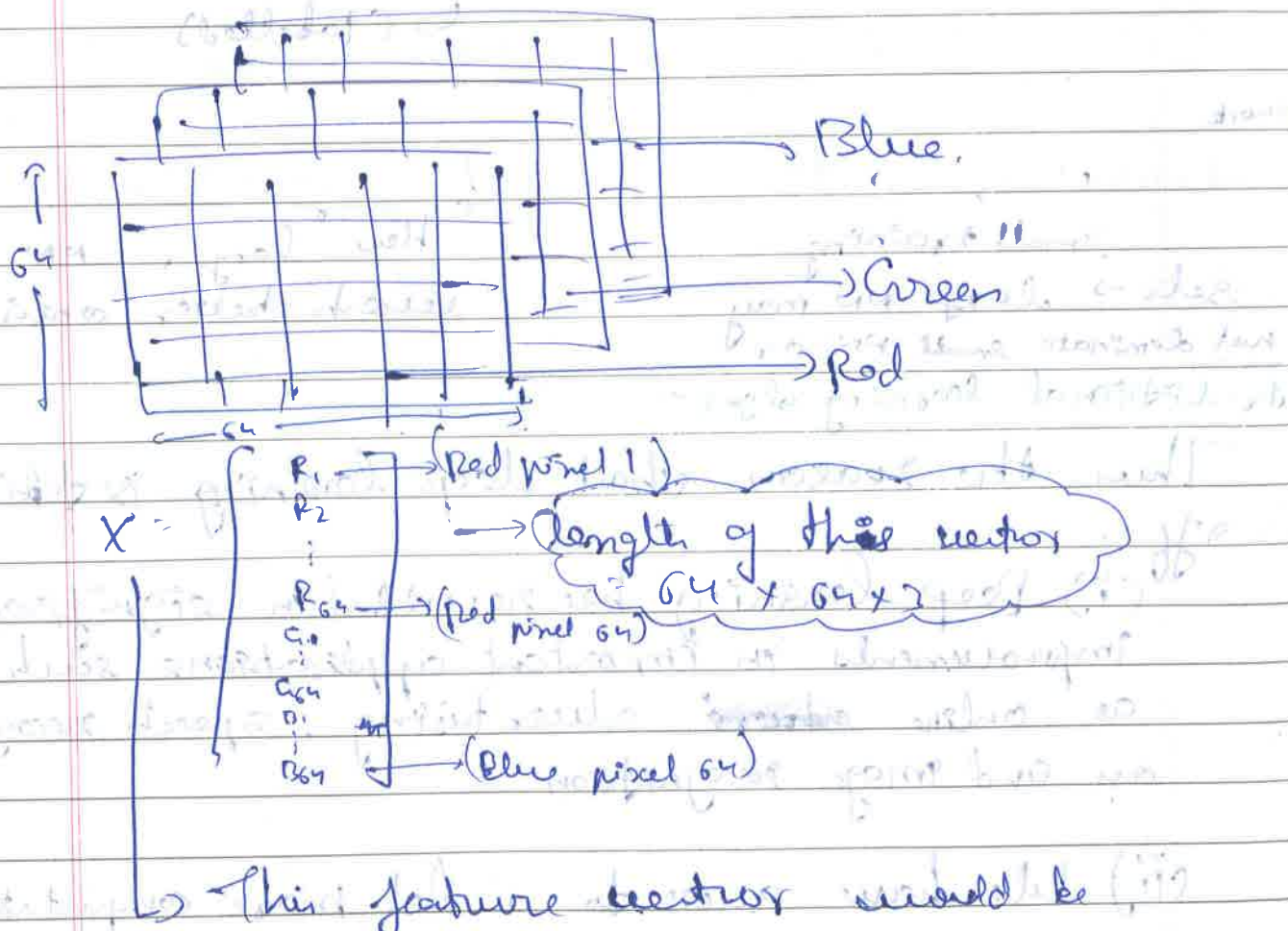
(iii) We have access to a lot more data.

And as a matter of fact increasing the neural network generally does not hurt the algorithms performance, and it may help significantly.

Week 2

→ Binary classification.

(Image) ⟶ 1 (cat) or 0 (non-cat)



→ This feature vector would be our input

→ Notation:-

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

m training example : $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \ldots (x^{(m)}, y^{(m)})\}$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad n_x \text{ (rows)}, \quad m \text{ (columns)}$$

$X \in \mathbb{R}^{(n_x) \times (m)}$ X shape = (n_x, m)

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

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

Y shape = $(1, m)$

→ logistic regression

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

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

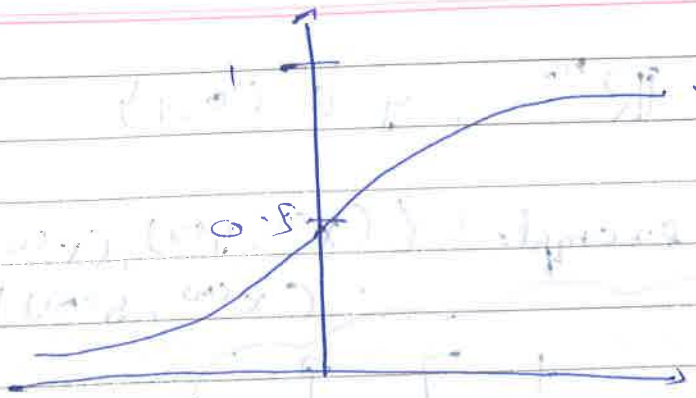
parameters :- $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

for linear regression we would have done:-

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

for logistic regression:

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



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

If z is large $\sigma(z) \approx \frac{1}{1+0} = 1$

If z is a large negative number

$$\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{\text{large number}} \approx 0$$

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

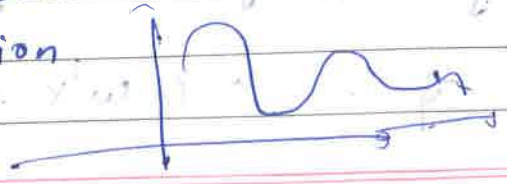
given $\{(x^{(1)}, y^{(1)}), \ldots (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$

loss (error) function :

we could have used

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

but we dont use that because it makes gradient descent not work well as it is a non-convex function.



PAGE NO.
DATE / /

then we use the following loss function:-

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

If $y = 1$: $L(\hat{y}, y) = -\log \hat{y}$ ← (want $\log \hat{y}$ large, want $\hat{y}$ large)

If $y = 0$: $L(\hat{y}, y) = -\log (1-\hat{y})$ ← (want $\log(1-\hat{y})$ large ∴ want $\hat{y}$ small)

→ cost function:

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

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

→ The loss function computes the error for a single training example; the cost function is the average of the loss loss functions of the entire training set.

→ We want to find w, b to minimize $J(w, b)$

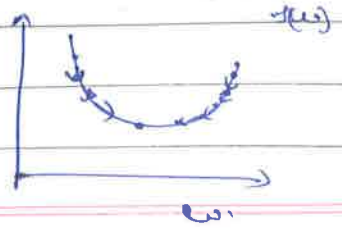
Gradient descent:-

Repeat {

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

(learning rate)

}



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

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

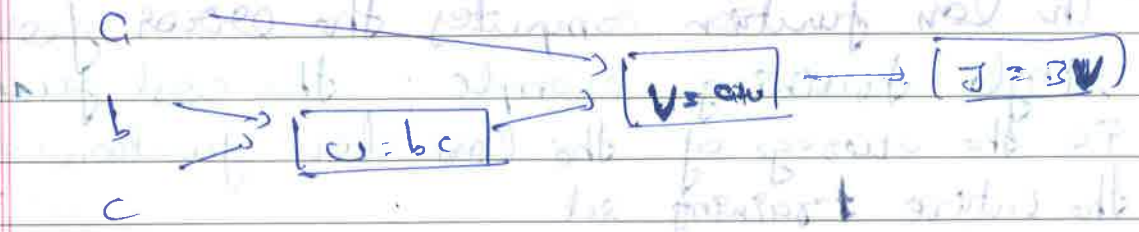
→ Computation graph

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

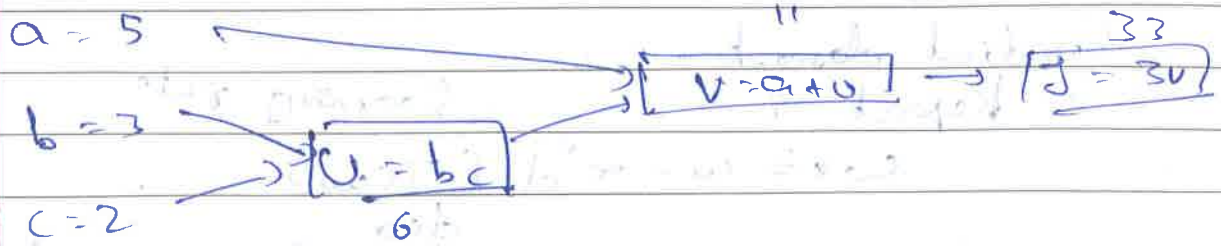
$u = bc$

$v = a + u$

$J = 3v$



→ Computing derivatives.



$J = 3v$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

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

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

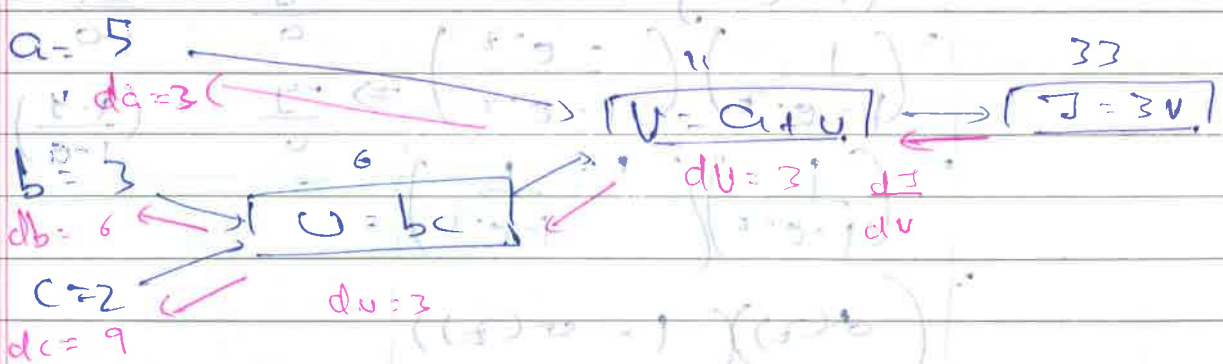
$J = 33 \rightarrow 33.009$ $\Rightarrow$ thus $\frac{dJ}{da} = 3$

$$\frac{dJ}{da} = \left(\frac{dJ}{dv}\right) \cdot \left(\frac{dv}{da}\right)$$

(The amount by which J changes when we change a, is the product by which J changes when we change v and (by which v changes when we change a)

now from now on we will denote

$\frac{d(\text{Final output variables})}{d(\text{variable})}$ as "d(variable)"



Here we are calculating derivatives from right to left.

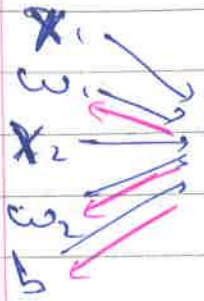
→ logistic regression recap:

$z = w^T x + b$

$y = a = \sigma(z)$

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

→ Logistic regression derivatives :



$z = w_1 x_1 + w_2 x_2 + b \rightarrow a = \sigma(z) \rightarrow L(a, y)$

$dz = \frac{dL(a,y)}{dz} = \left(\frac{dL(a,y)}{da}\right)\left(\frac{da}{dz}\right) = a - y$

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

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

$= \left(\frac{-1}{(1+e^{-z})^2}\right)(+e^{-z})$

$= \frac{-e^{-z}}{(1+e^{-z})^2}$

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

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

$= (\sigma(z))(1 - \sigma(z))$

$= a(1-a)$

then $\left(\frac{dL(a,y)}{da}\right)\left(\frac{da}{dz}\right)$

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

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

$= -y + ay + a - ay$

$= a - y$

$dw_1 = \frac{dL(a,y)}{dw_1} = x_1 dz$

$dw_2 = \frac{dL(a,y)}{dw_2} = x_2 dz$

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

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

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

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

$\Rightarrow -y \cdot \frac{d(\log a)}{da} - \frac{d(\log(1-a))}{da} + y \cdot \frac{d(\log(1-a))}{da}$

$\Rightarrow \frac{-y}{a} + \frac{1}{1-a} \cdot \frac{-y}{1-a}$

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

$w_1 := w_1 - \alpha . dw_1$

$w_2 := w_2 - \alpha . dw_2$

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

→ Logistic regression on m examples.

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

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

So lets initialize

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

For $i = 1$ to m

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

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

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

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

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

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

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

$J = J/m$

$dw_1 = dw_1/m$

$dw_2 = dw_2/m$

$db = db/m$

$w_1 := w_1 - \alpha . dw_1$

$w_2 := w_2 - \alpha . dw_2$

$b := b - \alpha . db$

After this we would be able to change w_1, w_2 & b once. So we see that we have to use for loop & this would reduce our efficiency.

hence we use vectorization to eliminate the use of for loops & increase the efficiency.

→ Vectorization :-

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

$z = w^T x + b$

non-vectorized :-

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

Vectorized :-

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

↳ $w^T x$

→ Python code for this in jupyter notebook :-

```
import numpy
a = np.array([1,2,3,4])
print(a)
```

```
[1 2 3 4]
```

```
import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)
```

```
tic = time.time()
c = np.dot(a, b)
toc = time.time()
print("Vectorized version:" + str(1000 * (toc - tic))
      + "ms")

c = 0
tic = time.time()
for i in range(1000000):
    c = c + a[i] * b[i]
toc = time.time()
print("For loop:" + str(1000 * (toc - tic))
      + "ms")
```

```
Vectorized version: 1.505..... ms
For loop : 481.3110..... ms
```

- This we see that vectorized version is around 300 times faster.

- The thing is that when we use vectorization and inbuilt libraries, they use the parallel lines in CPU's and CPU's much more efficiently than for loop.

Whenever possible avoid explicit for loops.

{P. T. O}

→ Vectors and matrix valued functions:

let $v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$

import numpy as np.

u = np.exp(v) ⟶ $u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$

u = np.log(v) ⟶ $u = \begin{bmatrix} \log v_1 \\ \log v_2 \\ \log v_n \end{bmatrix}$

u = np.abs(v) → $u = \begin{bmatrix} abs(v_1) \\ abs(v_n) \end{bmatrix}$

c = np.array([[c1, c2, ..., cn]])

u = np.maximum(v, c) → (returns the maximum element elementwise of both the arrays.)

v ** 2 → $u = \begin{bmatrix} v_1^2 \\ \vdots \\ v_n^2 \end{bmatrix}$

→ logistic regression derivatives.

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

for i = 1 to m:

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

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

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

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

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

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

J = J/m, dw = dw/m

db = db/m

→ Vectorized logistic regression:-

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

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

$$= [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad w^T x^{(3)} + b \cdots w^T x^{(m)} + b]$$

$$= Z$$

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

↳ Here python would automatically convert b "real number" to (1×m) matrix & add each element b to np.dot(w.T, x). This is called "broadcasting".

$A = [a^{(1)}\ a^{(2)}\ a^{(3)} \cdots a^{(m)}] = \sigma(Z)$

now till now we did

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

$dz^{(2)} = a^{(2)} - y^{(2)}$ ..... (5)

⋮

$dZ = [dz^{(1)}\ dz^{(2)}\ dz^{(3)} \cdots dz^{(m)}]$

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

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

till now we did :-

{p.T.O}

till now we did :-

→ $dw = 0$

$dw \mathrel{+}= x^{(1)} dz^{(1)}$

$dw \mathrel{+}= x^{(2)} dz^{(2)}$

⋮

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

$dw /= m$

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

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

$= \frac{1}{m} [x^{(1)} dz^{(1)} + x^{(2)} dz^{(2)} \cdots x^{(m)} dz^{(m)}]$

→ $db = 0$

$db \mathrel{+}= dz^{(1)}$

$db \mathrel{+}= dz^{(2)}$

$db \mathrel{+}= dz^{(3)}$

⋮

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

$db /= m$

$db = \frac{1}{m} \sum_{i=0}^{m} dz^{(i)}$

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

Then the code we will write is :-

```
for iter in range(0, 1000):-
    Z = w^T X + b
      = 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
```

→ Broadcasting examples.

Calories from carbs, proteins, Fats in 100 g 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

We have to calculate % of calories from Carb, protein, Fat :-

Ans) import numpy as np.

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]]

cal = A.sum(axis = 0)

now we will have cal = [59, 239, 155.4, 76.9]

percentage = 100 * A / cal

Then percentage:-

$$\begin{bmatrix} \frac{56}{59}\times 100 & \frac{0}{239}\times 100 & \frac{4.4}{155.4}\times 100 & \frac{68.0}{76.9}\times 100 \\ \frac{1.2}{59}\times 100 & \frac{104}{239}\times 100 & \frac{52.0}{155.4}\times 100 & \frac{8.0}{76.9}\times 100 \\ \frac{1.8}{59}\times 100 & \frac{135}{239}\times 100 & \frac{99.0}{155.4}\times 100 & \frac{0.9}{76.9}\times 100 \end{bmatrix}$$

This is the ans we require.

→ Some more broadcasting examples.

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

Python would automatically convert to :- $\begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix}$

(ii) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ (2x3) $+ \begin{bmatrix} 100 & 200 & 300 \end{bmatrix}$ (1x3)

Python would convert this to :- $\begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix}$

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

(iii) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ (2x3) $+ \begin{bmatrix} 100 \\ 200 \end{bmatrix}$ (2x1)

Python would convert this to

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

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

→ general principal.

$(m, n) \begin{matrix} + \\ - \\ * \\ / \end{matrix} (1, n) \rightsquigarrow (m, n)$

$(m, n) \begin{matrix} + \\ - \\ * \\ / \end{matrix} (m, 1) \rightsquigarrow (m, n)$

→ Tip:-

a = np.random.randn(5)
a.shape = (5,)
"rank 1 array"

"dont use"

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

(do this)

→ if for some reason we are getting a rank 1 array then we should do:

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

→ logistic regression cost function:-

so basically we have $\hat{y} = \sigma(w^T x + b)$

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

We interpret $\hat{y}$ as the probability that the image is a cat image given values x.

thus

$\hat{y} = P(y=1 \mid x)$

↳ probability that $y = 1$ given the values of x.

If $y=1$: $P(y|x) = \hat{y}$

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

→ the probability of $y=1$ given the parameters x is $\hat{y}$

→ The probability of $y=0$ given the parameters x is $(1-\hat{y})$

Thus to summarise we could write:-

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

(i) If $y=1$ then the probability that $y=1$ given $\hat{y}$ would be $(\hat{y})^{(1)}(1-\hat{y})^{(1-1)} = \hat{y}$

(ii) If $y=0$ then the probability that $y=$

(i) If $y=1$ then $P(y|x) = \hat{y}^{(1)} \cdot (1-\hat{y})^{(1-1)}$

$= \hat{y}$

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

(ii) If $y=0$ then $P(y|x) = \hat{y}^{0}(1-\hat{y})^{(1-0)}$

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

hence $P(y=0|x) = 1-\hat{y}$

Then we can write :-

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

now as log is a strictly increasing function thus $\log(P(y|x))$ would

also indicate the probability.

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

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

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

thus as $\log(P(y|x))$ ↑ , $L(\hat{y}, y)$ ↓

thus $P(y|x)$ ↑ , $L(\hat{y}, y)$ ↓

and we need $P(y|x)$ ↑ to go up as we want more certainty and hence more probability thus we want $L(\hat{y}, y)$ ↓. hence correctly termed as loss function.

→ Cost on m examples.

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

$\Rightarrow \log p(\cdots) = \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)})$

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

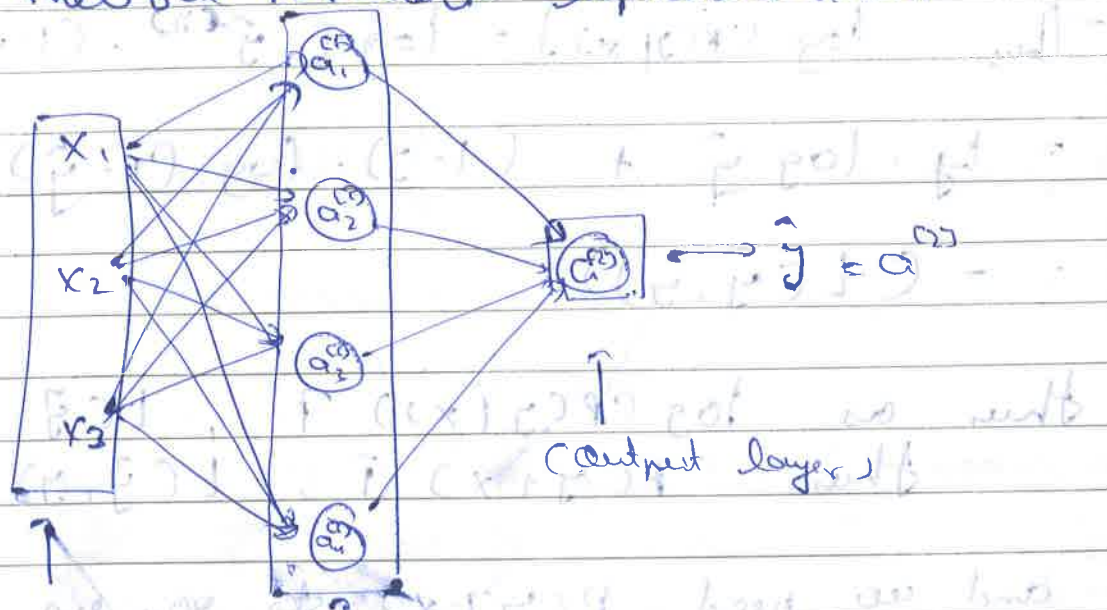
$= -\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$

Cost: $J(w, b) = \left(\frac{1}{m}\right) \cdot$ [redacted] $\left(\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})\right)$

(minimise)

week 3 (Shallow neural networks)

→ Neural network representation:



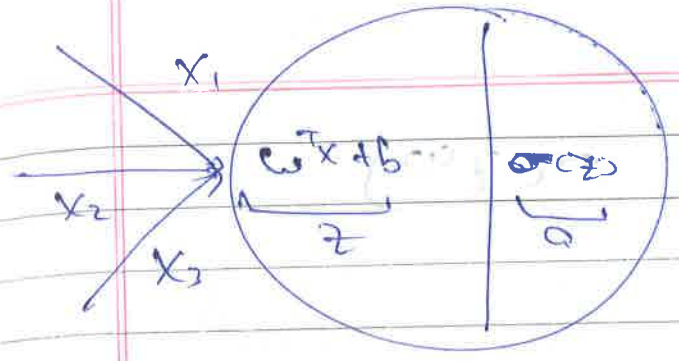
$(a^{[0]} = X)$ $(a^{[1]})$

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

- The above is a 2 layer neural network (the hidden layer & the output layer). we dont count the input layer as an official layer

$w^{[1]} \rightarrow (4, 3)$ → dimensions

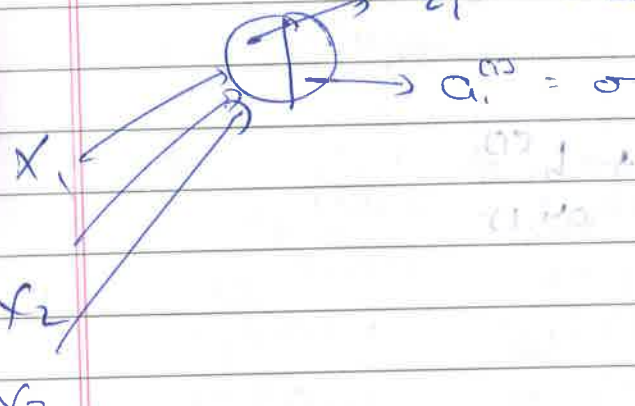
$b^{[1]} \rightarrow (4, 1)$ → dimensions



$z = w^T x + b$

$a = \sigma(z)$

now if we look at the first node of layer 1:-



($a_i^{[l]}$ → l^{th} layer, i → node in layer)

similarly:

$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$; $a_1^{[1]} = \sigma(z_1^{[1]})$

$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$; $a_2^{[1]} = \sigma(z_2^{[1]})$

$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}$; $a_3^{[1]} = \sigma(z_3^{[1]})$

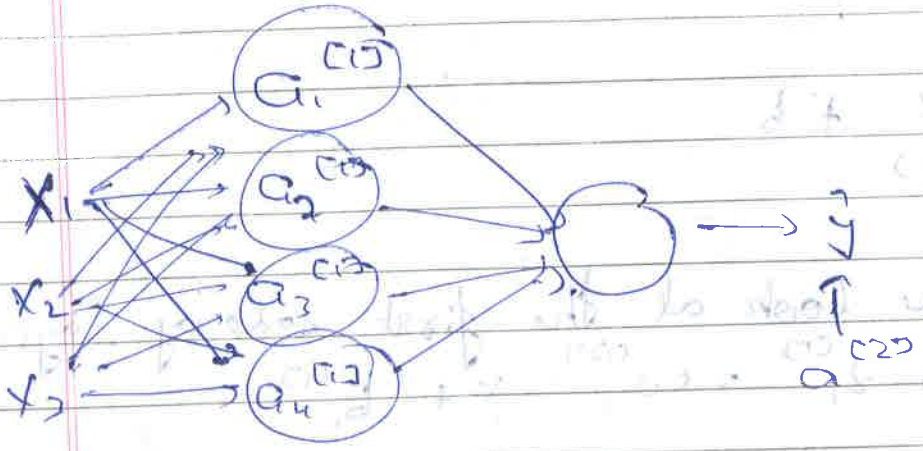
$z_4^{[1]} = w_4^{[1]T} \cdot x + b_4^{[1]}$; $a_4^{[1]} = \sigma(z_4^{[1]})$

we could write this as:-

$$\begin{bmatrix} \text{---} w_1^{[1]T} \text{---} \\ \text{---} w_2^{[1]T} \text{---} \\ \text{---} w_3^{[1]T} \text{---} \\ \text{---} w_4^{[1]T} \text{---} \end{bmatrix}_{(4,3)} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}_{(4,1)} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_3^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

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

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



Given input x:

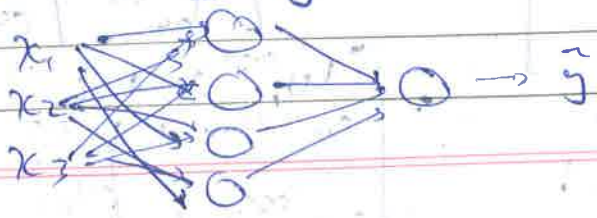
$$\rightarrow z^{[1]} = \underset{(4,1)}{W^{[1]}} \underset{(4,3)}{a^{[0]}} + \underset{(4,1)}{b^{[1]}} \quad \underset{(3,1)}{}$$

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

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

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

→ Vectorizing across multiple examples.



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

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

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

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

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

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

↳ layer 2

→ for $i = 1$ to m:

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

} non vectorized implementation.

so we do:

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

(n_x, m)

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

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

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

→ Justification for vectorized implementation.

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

$$w^{[1]} = \begin{bmatrix} \text{---} \\ \vdots \\ \text{---} \end{bmatrix} \quad x^{(1)} = \begin{bmatrix} | \\ | \end{bmatrix}$$

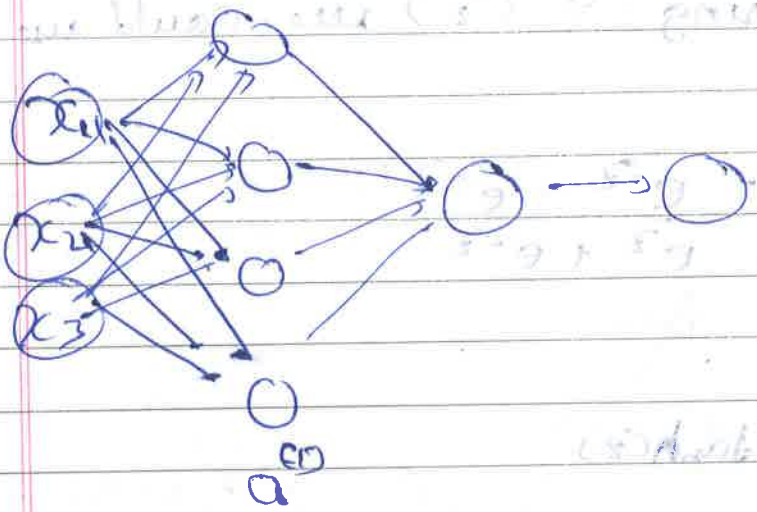
$$w^{[1]} x^{(1)} = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$$

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

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

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

→ Recap of ~~[illegible]~~ vectorization across multiple examples:-



$a^{[1]}$

$$W^{[1]} = \begin{bmatrix} — & w_1^{[1]} & — \\ — & w_2^{[1]} & — \\ — & w_3^{[1]} & — \\ — & w_4^{[1]} & — \end{bmatrix} \quad (4,3)$$

$$x = \begin{bmatrix} x_1^{(1)} \\ x_2^{(2)} \\ x_3^{(3)} \end{bmatrix}$$

$$X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \\ x_3^{(1)} & x_3^{(2)} & \cdots & x_3^{(m)} \end{bmatrix} \quad 3 \times m.$$

Un-Vectorized form:

for i in (0, m):

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

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

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

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

Vectorized form:

$$Z^{[1]}_{(4,m)} = W^{[1]}_{(4,3)} \cdot X^{[1]}_{(3,m)} + b^{[1]}_{(4,m)}$$

(b after broadcasting)

$$A^{[1]}_{(4,m)} = \sigma(Z^{[1]}_{(4,m)})$$

$$Z^{[2]}_{(1,m)} = W^{[2]}_{(1,4)} \cdot A^{[1]}_{(4,m)} + b^{[2]}_{(1,m)}$$

(b after broadcasting)

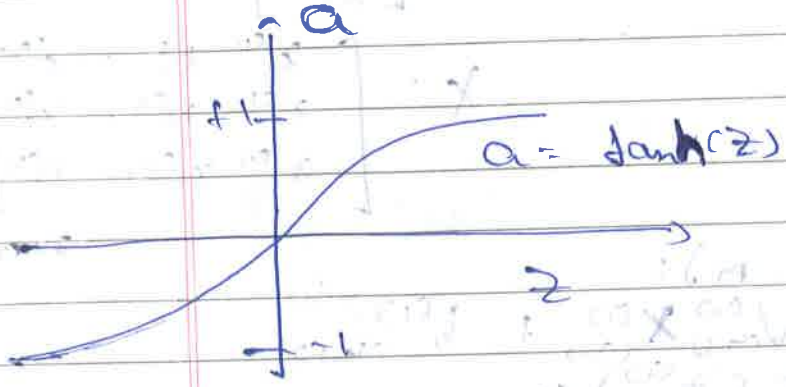
$$A^{[2]}_{(1,m)} = \sigma(Z^{[2]}_{(1,m)})$$

Note: $\tan^{-1}(x)$ and $\tanh(x)$ are two different expressions

→ Activation functions.

instead of using $\sigma(z)$ we could use $\tanh(z)$

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

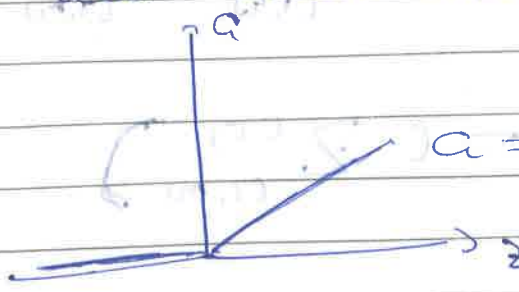


thus:

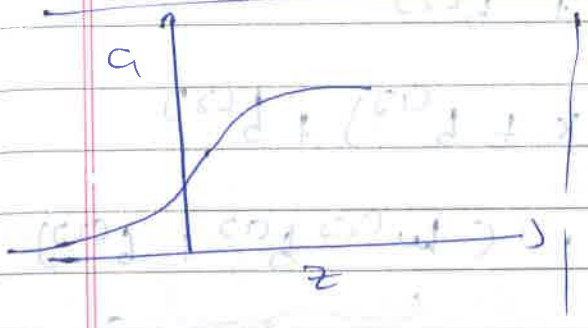
$$z^{[1]} = W^{[1]} x + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$

where g could be $\sigma()$ or $\tanh()$

→ we could also use Relu function:



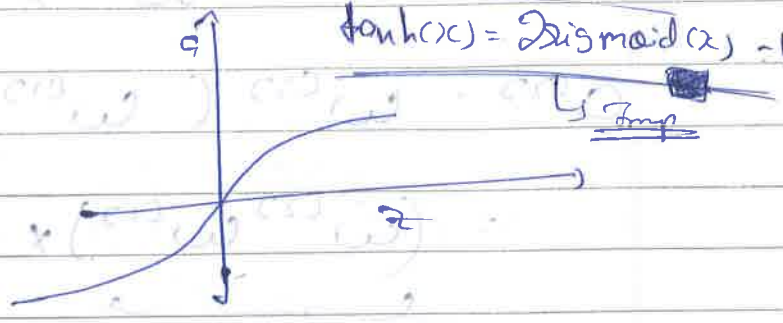
→ Pros and cons of activation function



Sigmoid:

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

(Do not use very less, only for binary classification)



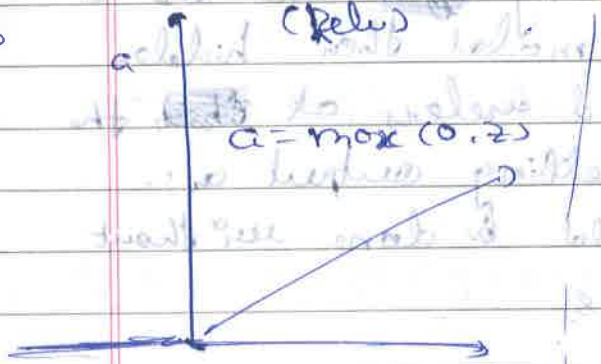
tanh(x) = 2sigmoid(x) - 1

Imp

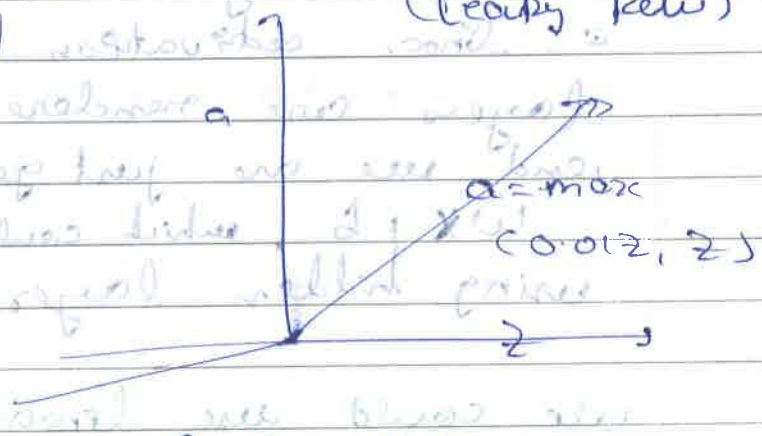
$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

(mostly all the time works better than sigmoid)

→



(By default we use this, mostly this works the best)



(Leaky Relu)

→ Why does neural network need non linear activation function.

⇒ If we do $g(z) = z$

∴ linear activation function"

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

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

$a^{[2]} = z^{[2]} = w^{[2]} x + b^{[2]}$

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

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

w_1 $\qquad$ b'

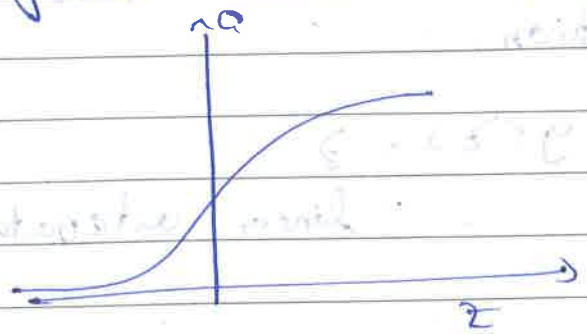
$= w' x + b'$

hence we get that if we use a linear activation model then hidden layers are rendered useless, at the end we are just getting output as: $wx + b$, which could be done without using hidden layers.

We could use linear activation function sometimes to compute the final output at the last hidden layer though.

→ Derivatives of activation function.

⇒ Sigmoid activation function



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

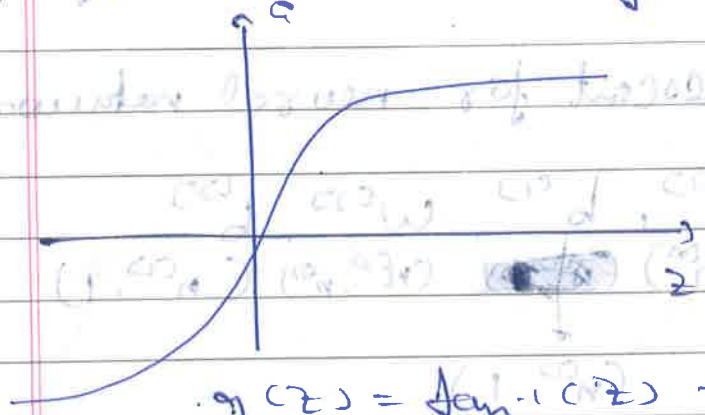
$$\frac{d(g(z))}{dz} = \text{slope of } g(z) \text{ at } z = g'(z)$$

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

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

Thus if $g(z) = a$ then $g'(z) = a(1-a)$

⇒ tan⁻¹ activation function.



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

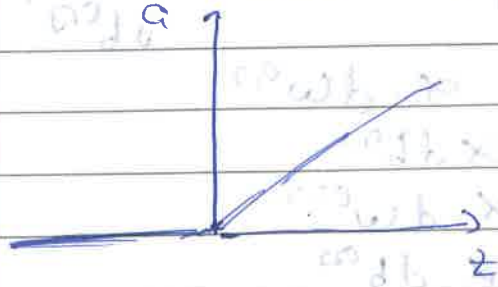
$$g'(z) = \frac{d(g(z))}{dz} = \text{slope of } g(z) \text{ at } z$$

$$= 1 - (\tan^{-1}(z))^2$$

Then if $a = g(z)$

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

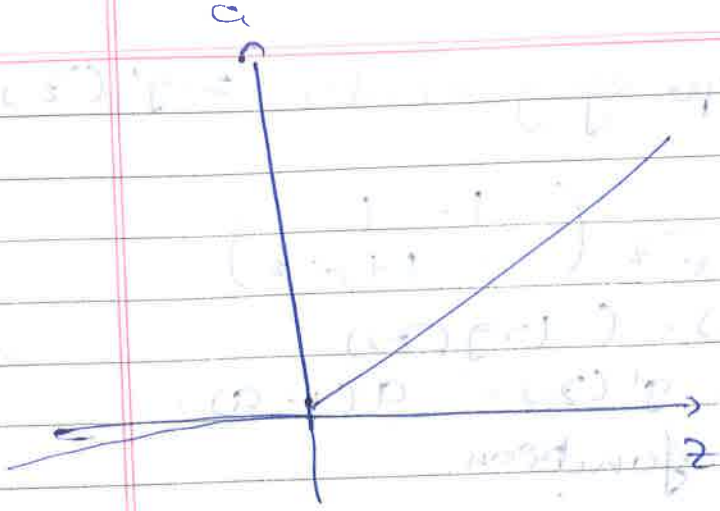
⇒ ReLU and Leaky ReLU



(ReLU)

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

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



Leaky ReLU

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

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

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

$= \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$ → $a^{[2]}$

Gradient descent:-

→ Repeat {

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

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

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

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

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

$b^{[2]} := b^{[2]} - \alpha \cdot db^{[2]}$

→ formulaes for computing derivatives.

Forward propagation :-

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

back propagation :-

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

↳ this thing means see that all that our dimensions are $(n^{[2]}, 1)$ & not $(n^{[2]},)$

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

$W^{[2]T} dZ^{[2]} \rightarrow (n^{[1]}, m)$

$g^{[1]'}(Z^{[1]}) \rightarrow (n^{[1]}, m)$

↳ element wise product.

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

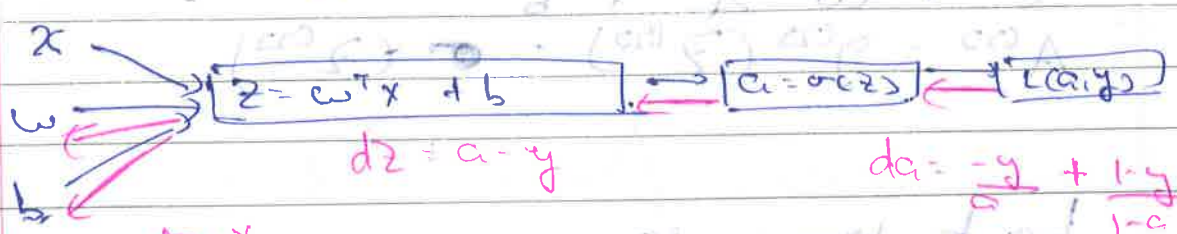
$= (n^{[1]}, 1)$

{p-10}

→ Backpropagation intuition (optional)

⇒ Computing gradients :-

logistic regression :-



we know that $a = g(z)$

$$g(z) = \sigma(z) = a$$

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

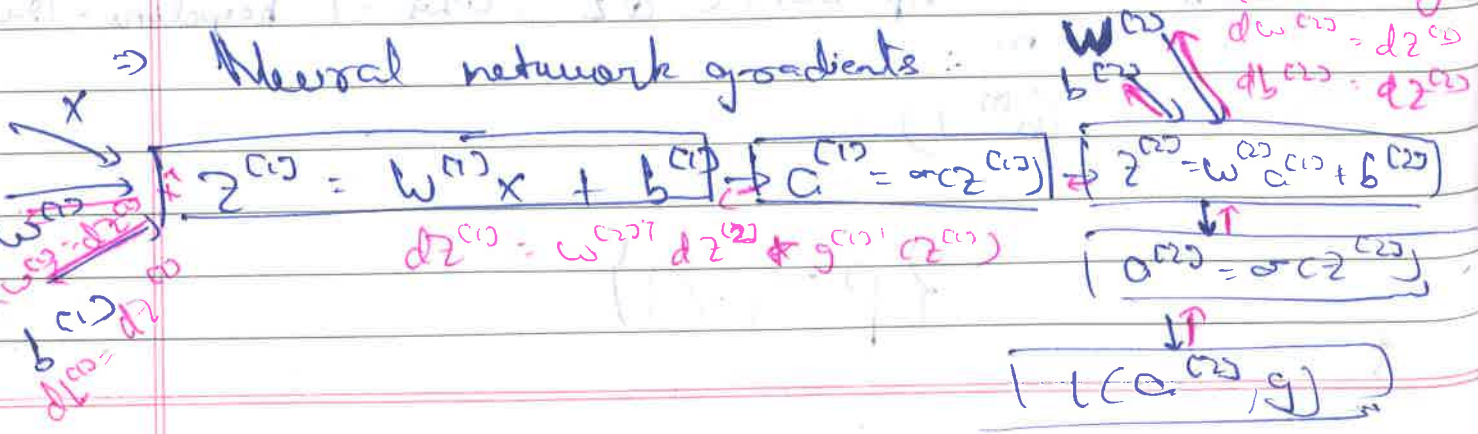
$$\Rightarrow g'(z) = \frac{da}{dz}$$

$$\Rightarrow g'(z) = \left(\frac{da}{dl}\right) \cdot \left(\frac{dl}{dz}\right)$$ → means dl/dz

$$\Rightarrow g'(z) = \frac{dz}{da}$$ → means $\frac{da}{dl}$

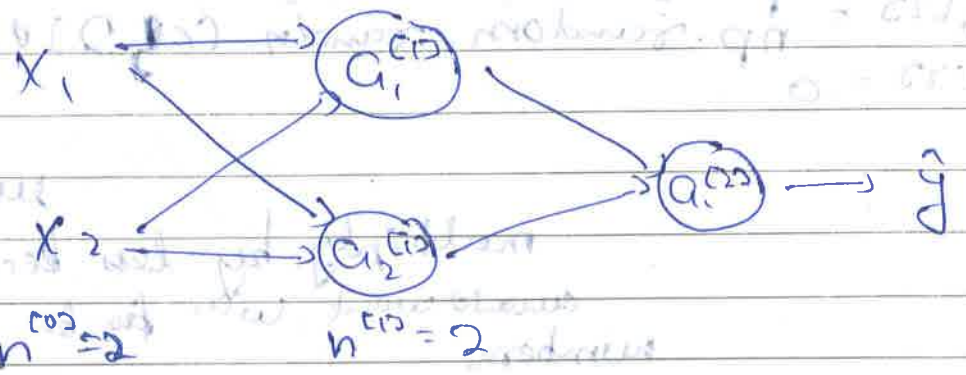
$$\Rightarrow dz = da \cdot g'(z)$$

⇒ Neural network gradients :-



→ Random initialization:

What happens if you initialize weights to zero??



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

then we get $a_1^{[1]} = a_2^{[1]}$

$w^2 = [0 \;\; 0]$

$dz_2^{[1]} = dz_2^{[1]}$

$dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$

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

So basically we will always have dw in the form of $\begin{bmatrix} \alpha & \beta \\ \alpha & \beta \end{bmatrix}$ &

always $a_1^{[1]} = a_2^{[2]}$ which would fail the purpose of hidden layers

so instead we should use :

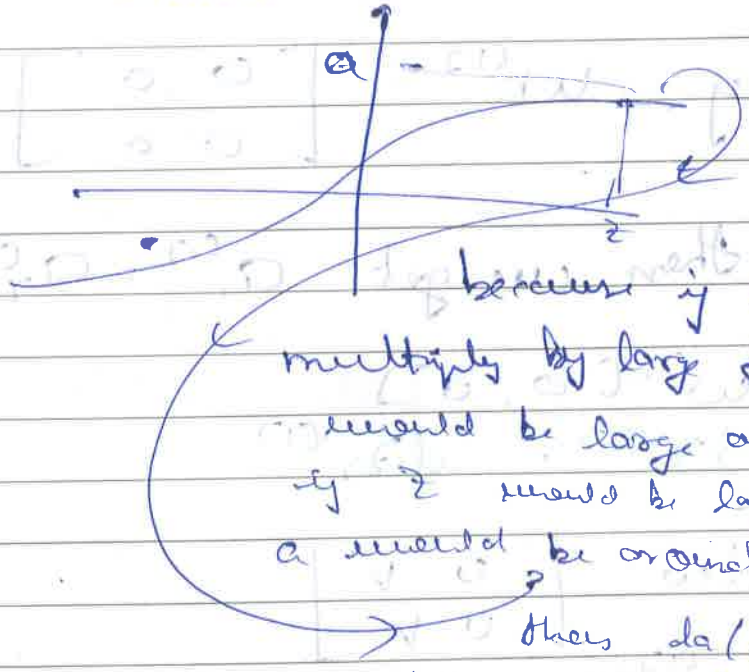
$w^{[1]} = np.random.randn((2,2)) * 0.01$

$b^{[1]} = np.zeros((2,1))$

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

$b^{[2]} = 0$

we would multiply by less because we would want w's to be small numbers



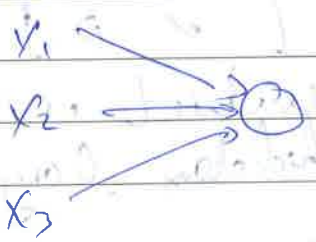
because if we would multiply by large then z would be large as $z = w^T x + b$

if z would be large then a would be original here

then $da/dz \approx 0$

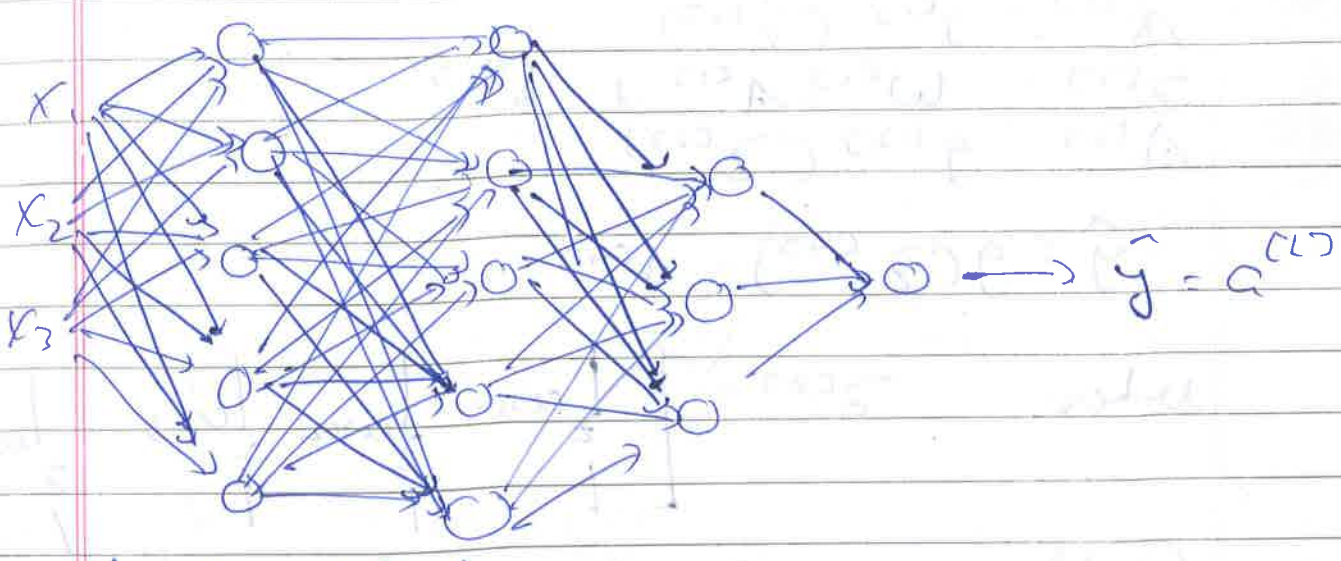
which would make our algorithm very slow.

week 5 @ Deep L-layer neural network



1 layer neural network (shallow)

Deep neural network



$L = 4$ (# layers)

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

$a^{[l]}$: activations in layer l

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

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

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

$a^{[0]} = x$

[redacted] $\rightarrow a^{[0]}$

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

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

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

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

$\vdots$

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

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

Then do generalize:-

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

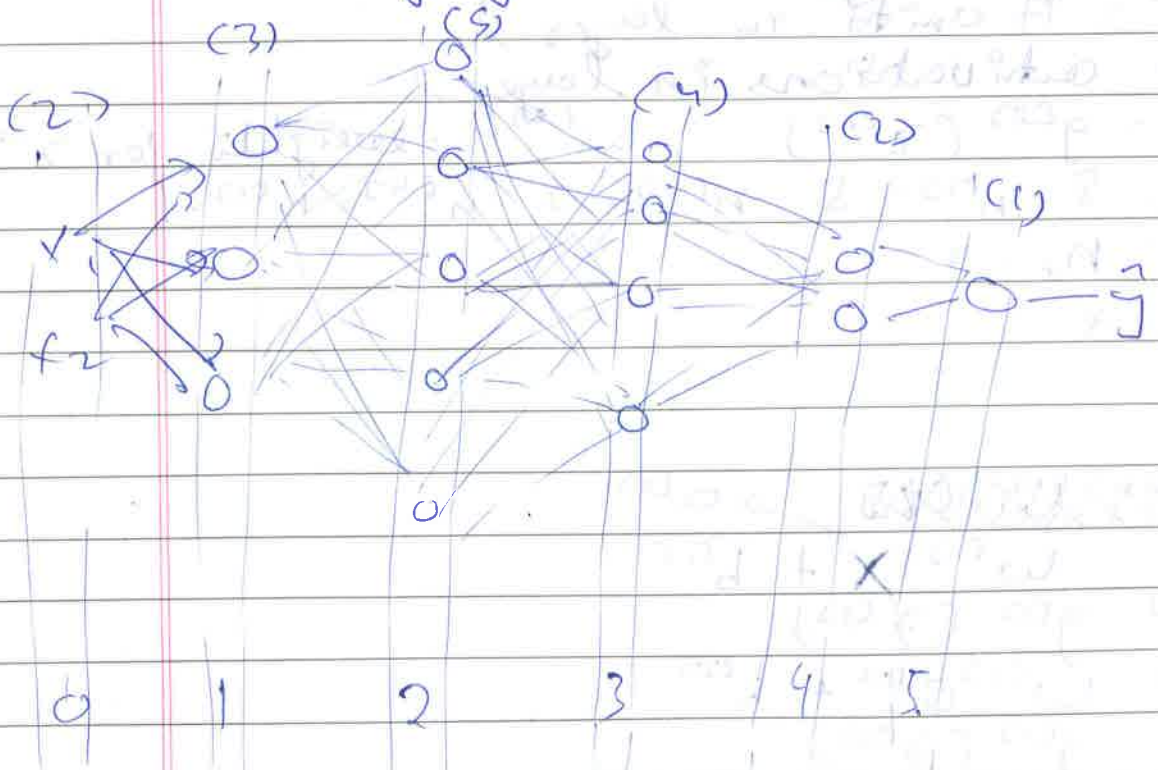
$a^{[l]} = g^{[l]}(z^{[l]})$

Vectorized :-

$$Z^{[1]} = W^{[1]} A^{[0]} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$
$$\hat{y} = g(Z^{[4]}) = A^{[4]}$$

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

→ Getting your matrix dimensions right.



$W^{[1]}$: $(n^{[1]}, n^{[0]})$ $(3, 2)$

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

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

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

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

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

and $z^{[l]} = (n^{[l]}, 1)$

for vectorized versions:-

(i) $Z^{[l]} = \left[z^{[l](1)} \; z^{[l](2)} \; z^{[l](3)} \; \cdots \; z^{[l](m)} \right]$

$Z^{[l]} = (n^{[l]}, m)$

(ii) $b^{[l]} = \left[b^{[l](1)} \; b^{[l](2)} \; \cdots \; b^{[l](m)} \right]$

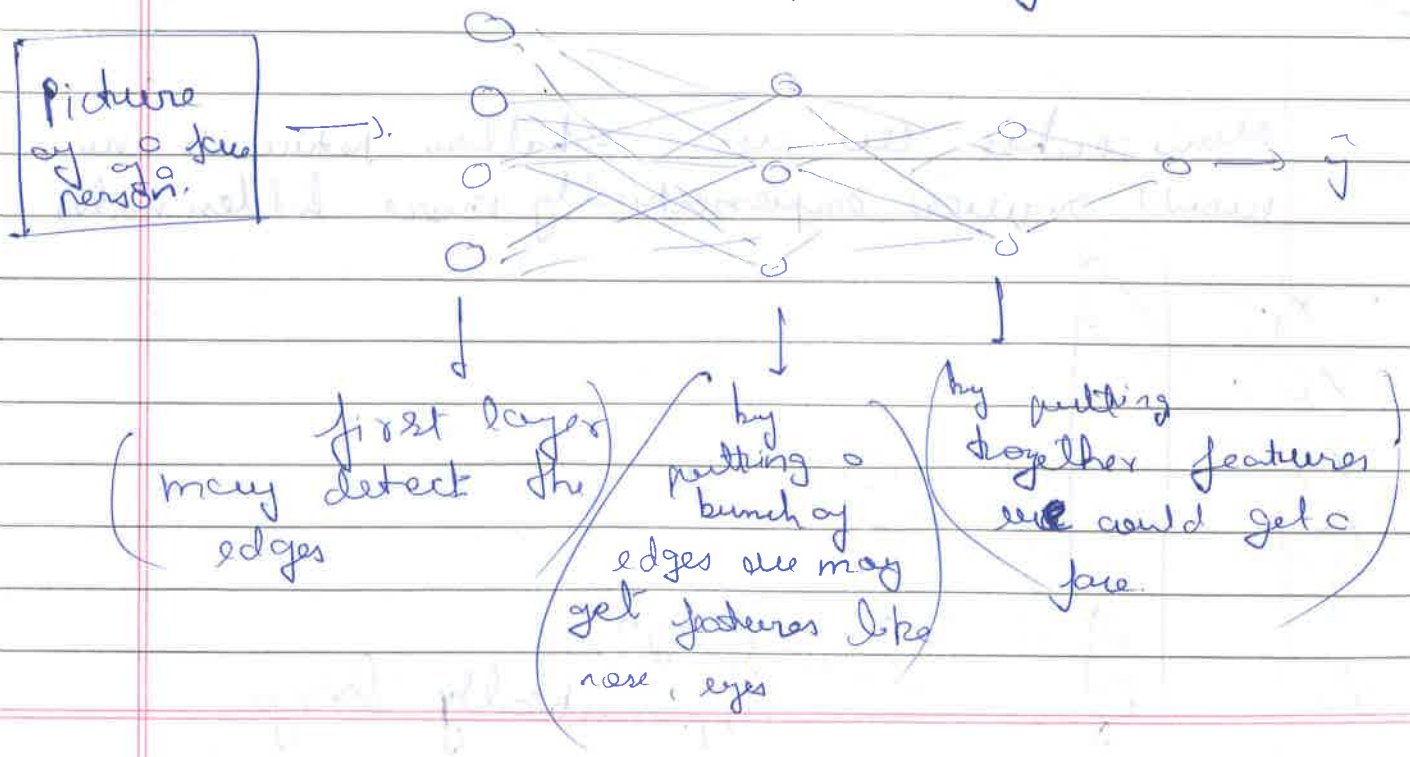
$b^{[l]} = (n^{[l]}, m)$

then $db^{[l]} = (n^{[l]}, m)$

(iii) $w^{[l]} = \begin{bmatrix} \text{---} \; w^{[l]}_{\text{node 1}} \; \text{---} \\ \text{---} \; w^{[l]}_{\text{node 2}} \; \text{---} \\ \vdots \\ \text{---} \; w^{[l]}_{\text{node n}} \; \text{---} \end{bmatrix}$ (number of nodes in previous layer) × (number of nodes)

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

→ Intuition about deep learning.



So basically deep learning is deep neural networks

PAGE NO.
DATE / /

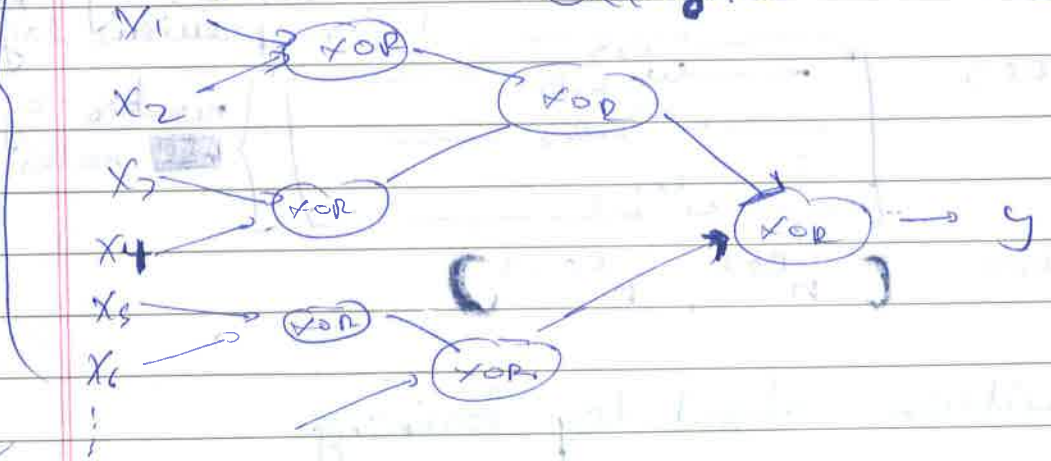
⇒ Circuit theory and deep learning :-

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.

Eg.

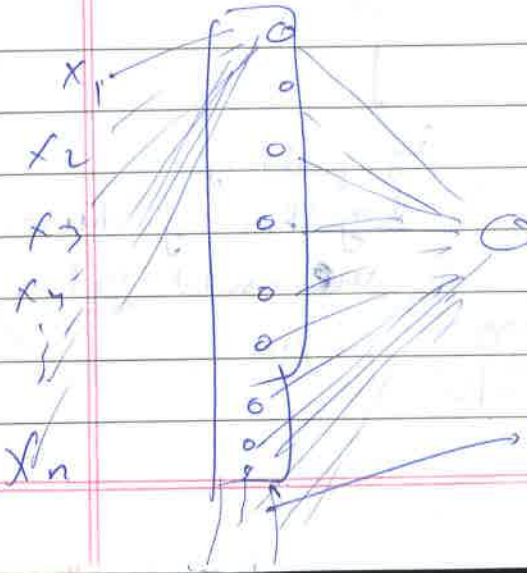
$y = (X_1) \text{ XOR } (X_2) \text{ XOR } (X_3) \text{ XOR } (X_4) \cdots \text{ XOR } (X_n)$

$\longleftarrow O(\log n) \longrightarrow$



This is when we are using a deep layered deep neural network

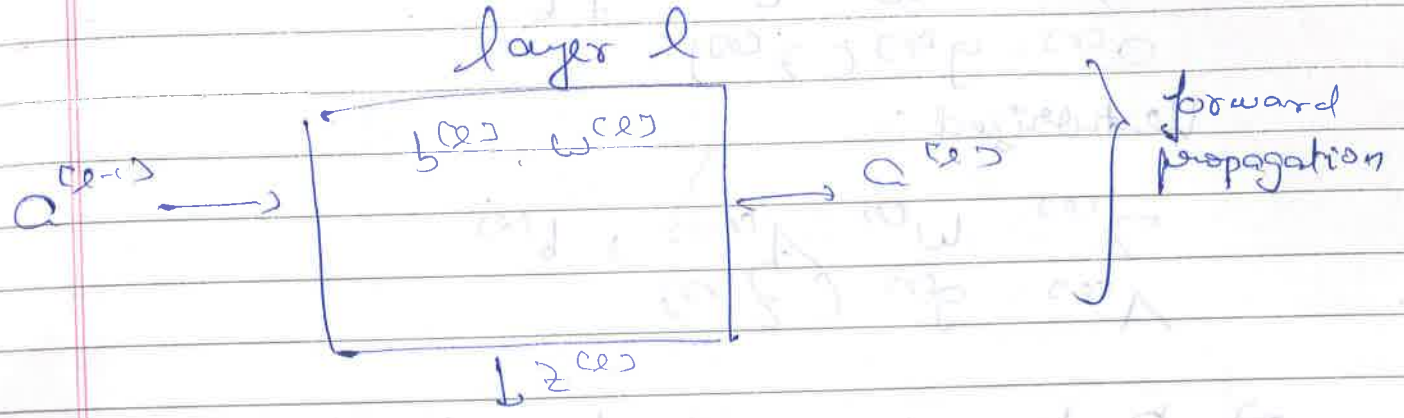
Now when we are a shallow network we would require exponentially more hidden nodes.

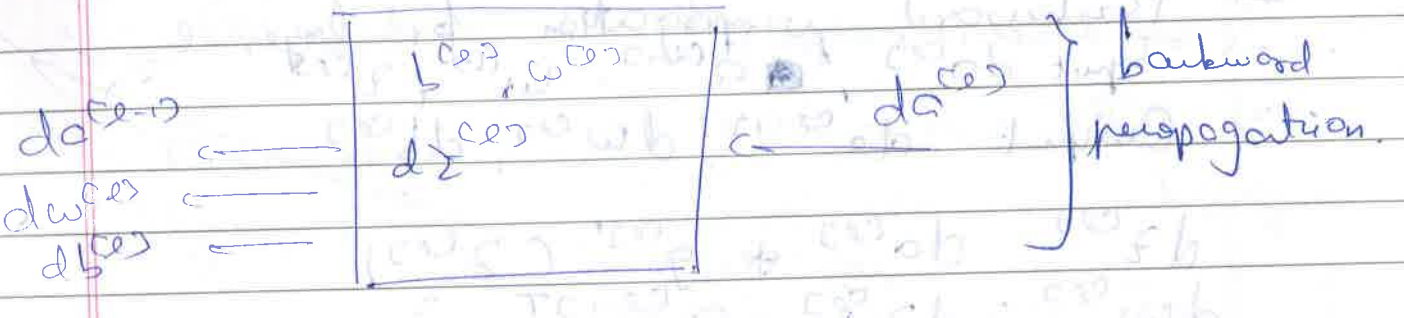


$O(2^n)$
exponentially large

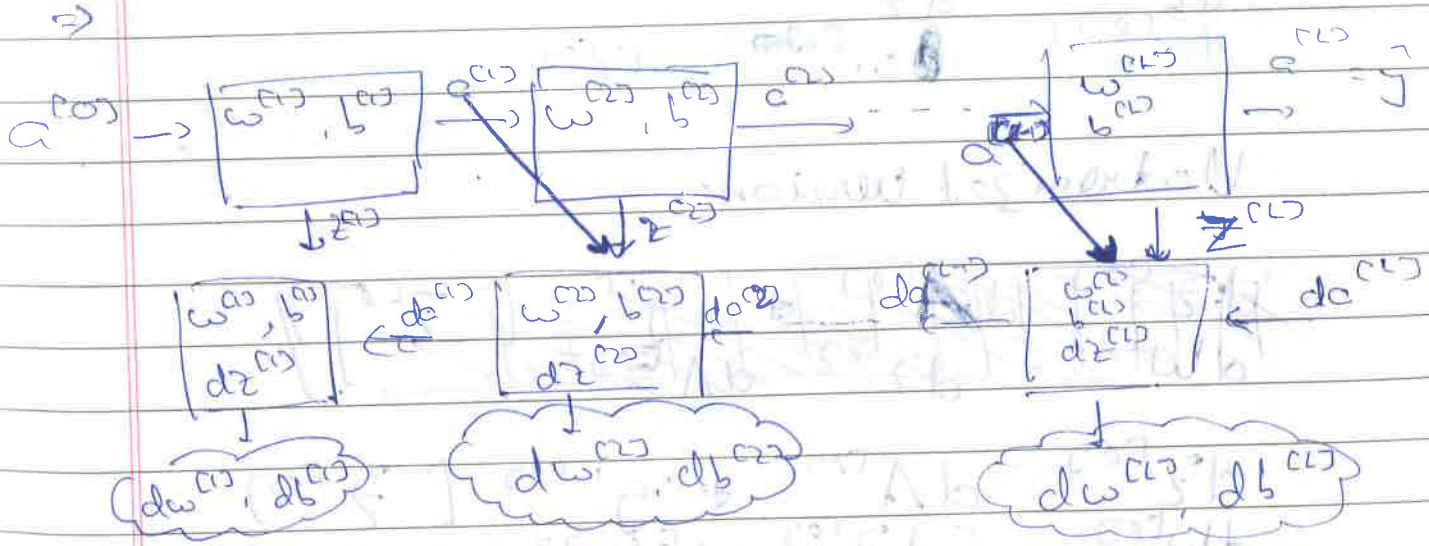
→ Building blocks of deep neural network.

⇒ Foroward and backward functions :-





⇒



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

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

→ Foward propagation for layer l.

⇒ Input $a^{[l-1]}$

⇒ Output $a^{[l]}$, (cache $(z^{[l]})$

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

Vectorized :-

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

⇒ Backward propagation for layer l.

⇒ Input $da^{[l]}$, $a^{[l-1]}$, $w^{[l]}$, $z^{[l]}$

⇒ Output $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$

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

Vectorized version:-

$$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$$
$$dW^{[l]} = (dZ^{[l]} \cdot A^{[l-1]T})(1/m)$$
$$db^{[l]} = \frac{1}{m} \cdot np.sum(dZ^{[l]}, axis=1, keepdims=True)$$
$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$$

→ To calculate

$$dA^{[2]} = \left(-\frac{y^{(1)}}{a^{(1)}} + \frac{(1-y^{(1)})}{1-a^{(1)}} \right. \quad \left. - \frac{y^{(m)}}{a^{(m)}} \right)$$

→ To calculate

$$dz^{[2]} = \left(-\frac{y}{a} + \frac{1-y}{1-a} \right)$$

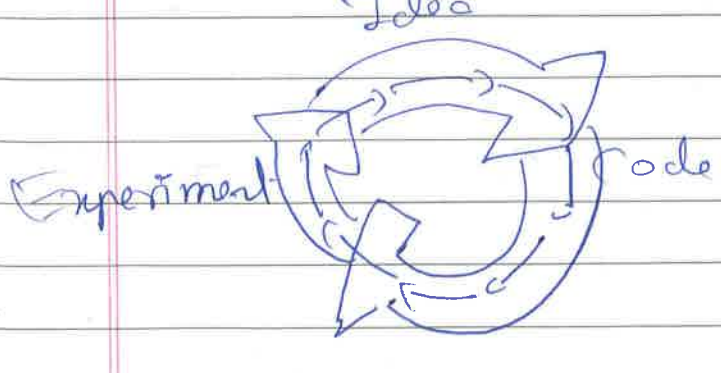
→ Parameters Vs Hyperparameters.

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

Hyperparameters :- learning rate α, number of iterations
Hidden layer L
Hidden units $n^{[1]}, n^{[2]} \ldots$
Choice of activation function.

⇒ Hyperparameters determine the value of parameters.

⇒ Applied deep learning is a very empirical process.



It is not very accurate to say that deep learning is like human brain. We know very less about neurons & human brain to make this analogy.

=> "Cache" is used to pass variables computed during forward propagation to the corresponding backward propagation step. It contains useful values for backward propagation to compute derivatives.

Eg. Like $Z^{[2]}$ which is computed in forward propagation is used to compute

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

→ Some notes from the python code upon L-layer neural networks:-

Specialization 2 Week 1

PAGE No.
DATE / /

→ Train / dev / test / sets

Data [| |]

(Training set) (test)

- hold-out cross validation set
- Development set
- "dev" set

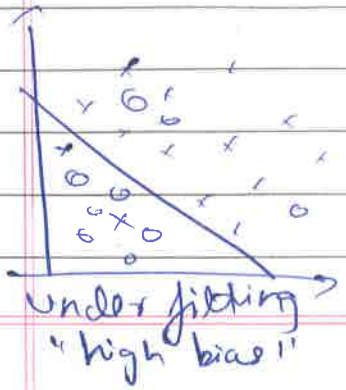
⇒ Earlier we used to divide the data as :-
60 % / 20 % / 20 %.

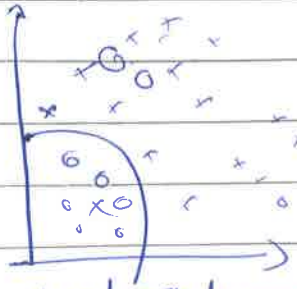
⇒ But now in the age of Big data, for example when we have [redacted],000,000 (1 million samples) then even 10,000 samples for (dev set)/(test set) would be enough
thus :- 98 % / 1 % / 1 %.

⇒ Make sure that "dev" & "test" set are captured from the same distribution.

⇒ Not having a test set might be okay. (Only dev set)

→ Bias / Variance





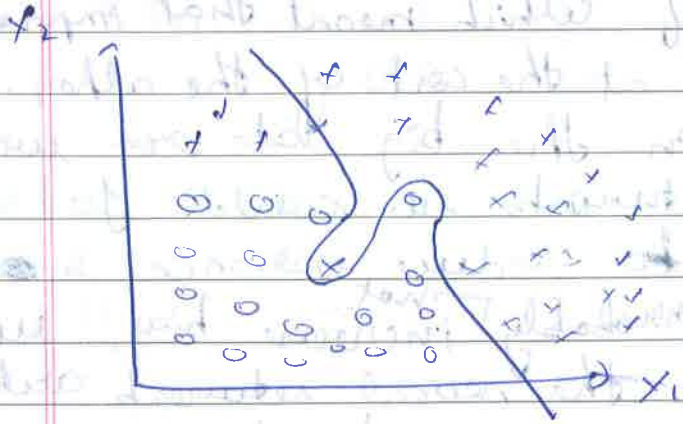
Eg Cat classification

Train set error :	1 %	15 %	15 %	0.5 %
Dev set error :	11 %	16 %	30 %	1 %
	(high variance)	(high bias)	(high bias & high variance)	(low bias & low variance)

=> This is assuming that Bayes error ≈ 0 % & train & Dev set are taken from the same distribution.

=> If Bayes error ≈ 15 % (This means that even humans are 15 % time wrong while classifying cat image probably because the image is too blur. then this would not be high bias but would be suitable

=> High bias & high variance



-> Basic recipe for machine learning

High bias? (training data performance)

(Yes) -> high (we dont have high bias)

low (we have high bias)

(No)

-> bigger network
-> Train longer (increase number of iterations)
-> Search for better neural network architecture

No.

High variance? ——Yes.——→ (More data) (Regularization) (get better neural network architecture)

(dev set performance: good → you don't have high variance; bad → you have high variance)

=> If you could collect more data then you could reduce variance but not necessary you could considerably reduce bias

=> Earlier there was concept of bias / variance tradeoff. Which meant that improving one would come at the cost of the other.

But now in the big data era with rise of neural networks we could feed in a lot of data to reduce variance & that would most probably not increase bias, we could also improve the neural network architecture. Thus the concept of bias / variance tradeoff may not strictly apply.

→ Regularization. (to prevent overfitting)

=> logistic regression:-

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

("regularization parameter")

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

L2 regularization.

L1 regularization:-

here we add $\frac{\lambda}{2m} \sum_{i=1}^{n_x} |w| = \frac{\lambda}{2m} \|w\|_1$

If we use L1 regularization then w will be sparse, it means that w would have a lot of zeroes. That would save memory.

=> Neural network:-

$$J(w^{[1]}, b^{[1]}, \ldots w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

"Forbenius norm"

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

[illegible]

{p. 70}

=> Earlier we used to do:-

$dw^{[l]} = (\text{from backpropagation}) + \frac{\lambda}{m} w^{[l]}$

=> Earlier before regularization we used to do:-

$dw^{[l]} = (\text{the value we got from backpropagation})$

where $dw^{[l]} = \frac{\partial J}{\partial w^{[l]}}$

now we will do:-

$dw^{[l]} = (\text{from backpropagation}) + \frac{\lambda}{m} w^{[l]}$

Then

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

$\Rightarrow w^{[l]} := w^{[l]} - \alpha \left((\text{from backpropagation}) + \frac{\lambda}{m} w^{[l]} \right)$

$\Rightarrow w^{[l]} := w^{[l]} \left(1 - \frac{\alpha \lambda}{m} \right) - \alpha [(\text{from backpropagation})]$

↳ this is +ve

this would be less than 1

then this is called weighted decay

-> How does regularization prevent overfitting

$J(w^{[l]}, b^{[l]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{n} \| w^{[l]} \|_F^2$

=> Some raw as λ ↑, $w^{[l]}$ ↓, it is also possible that some of the values of $w^{[l]}$ would tend to 0.

Thus then our answer would be more linear.

=> If λ ↑ thus $w^{[l]}$ ↓ then :-
$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$ would be low.

If we are $g(z) = \tan^{-1}(z)$

as z would be small thus we would get $g(z)$ to be more linear, thus our final ans would tend be more linear. This is because the ans of every layer would tend to be more linear.

This would give us a boolean easy

=> $J(\ldots) = \sum l(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum \|w^{[l]}\|_F^2$

Dropout regularization

Illustrating dropout on layer $l = 3$

$d3 =$ np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob

$a3 =$ np.multiply(a3, d3)

$a3 = a3 /$ keep_prob ; (we do this so that the result of the cost will have the same expected value as without dropout.

this was "inverted dropout"

⇒ making predictions at test time.

$a^{[0]} = X$

No drop out:-

$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$

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

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

$a^{[2]} = \ldots$

$\hat{y}$

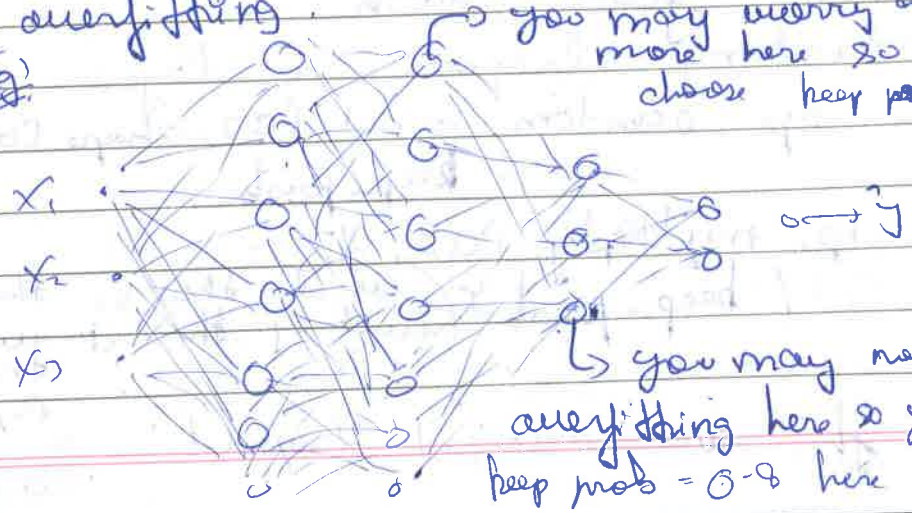
In dropout regularization you randomly drop some nodes for each iteration.

→ Why does drop-out work.

Intuition:- Can't rely on any one feature, so have to spread out weight.

If you are worrying about overfitting [redacted] on a layer more, then [redacted] decrease keep-prob on them and increase keep-prob on those which you [redacted] think won't pose a problem in overfitting.

Eg:

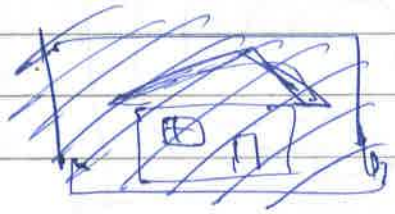


Here cost function J is not very much properly defined.

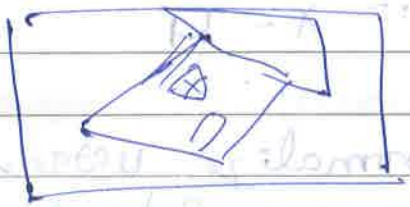
→ Other regularization method.

=> Data augmentation.
If you have this picture

and you need more data and you dont have then you could do this:-



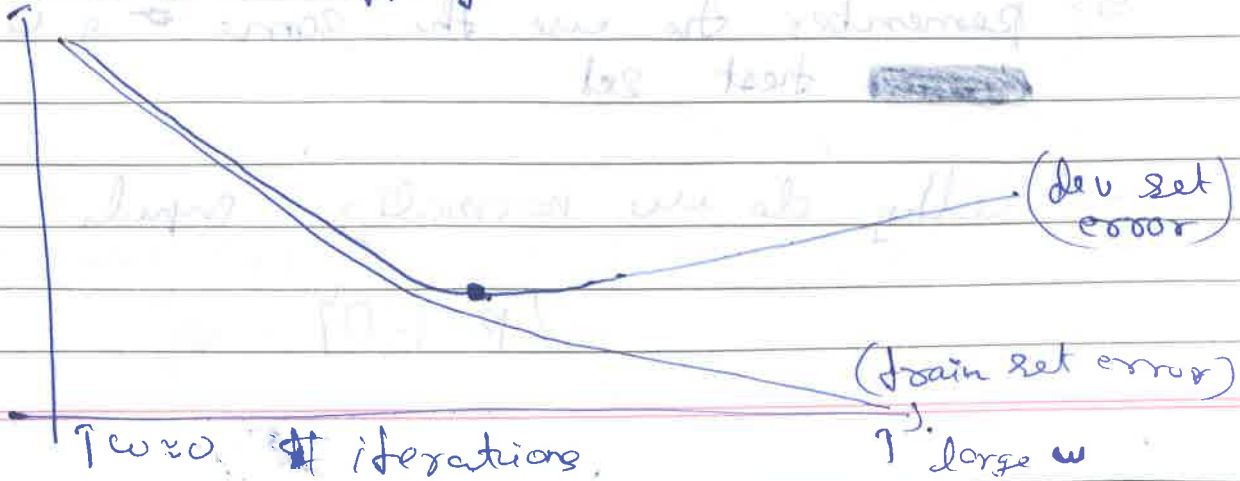
(Flipped sideways)



(Cropped & rotated)

This would not give very much new information but still give a little information and would help a little in regularization.

=> Early stopping.



=> Orthogonalization.
- This means that at one time focus only on one task.
So lets look at these tasks:
(i) Optimize cost function J
(ii) Do not overfit.
So first task is to concentrate on just decreasing J & other is to prevent overfitting.

→ Normalizing inputs.
(i) Subtract mean:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$

(ii) normalize variance
Variance = (standard deviation)^2 = σ^2

$$\Rightarrow \sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})^{**}2$$

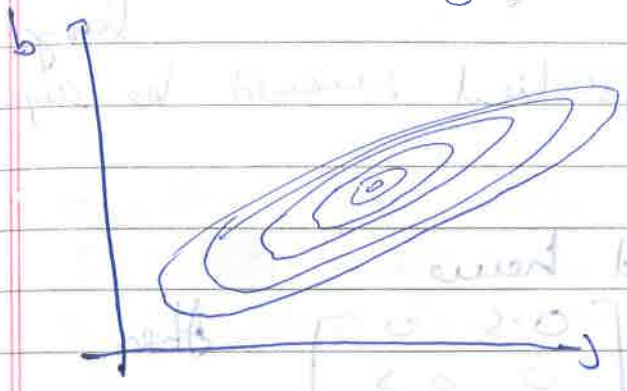
$$x := x / \sigma$$

=> Remember to use the same σ & μ on the test set

=> Why do we normalize inputs:-

{p. 7.0}

(i) Un-normalized :-

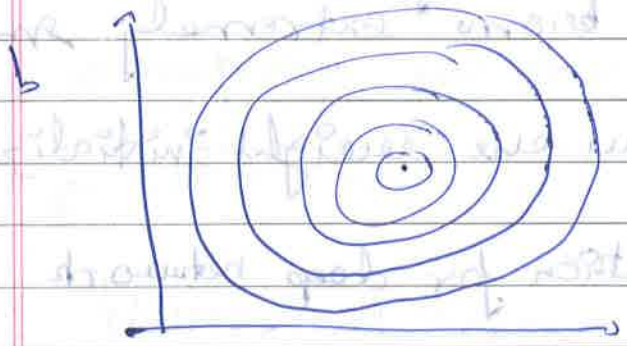


x_1 : 1 - 1000

x_2 : 0 - 1

(graph of Cost function)

(ii) normalized



x_1 : -1 : 1

x_2 : 0 : 1

(graph of Cost function)

If we normalize we would be able to train our model faster.

→ Vanishing / Exploding gradient

Eg)

$g(z) = z$ $b^{[e]} = 0$

$\hat{y} = W^{[l]} W^{[l-1]} W^{[l-2]} \cdots W^{[3]} W^{[2]} W^{[1]} X$

assuming that all $W^{[e]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$

then $\hat{y} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{l-1} X = 1.5^{l-1} X$

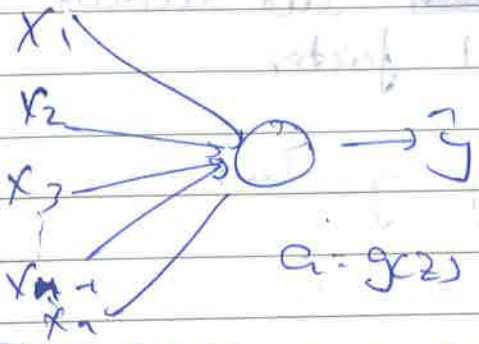
now here thus $\hat{y}$ would be exponentially large,
and even the gradient would be exponentially large

and if we would have
$W^{[2]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ then
$\hat{y}$ would be extremely small & even gradient would become extremely small

To solve this we use weight initialization

→ weight initialization for deep network

⇒ single neuron example



$z = w_1x_1 + w_2x_2 + w_3x_3 \cdots w_nx_n$

now if $n \to$ is large then we would want w_i to be small so that z doesn't become very large.

thus we sometimes define:

$\text{Var}(w_i) = \frac{2}{n}$

$w^{[2]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[2]-1}}\right)$

we prefer two are
$Var(w_i) = \frac{2}{n}$ when we have $g^{(L)}(z) = Relu(z)$

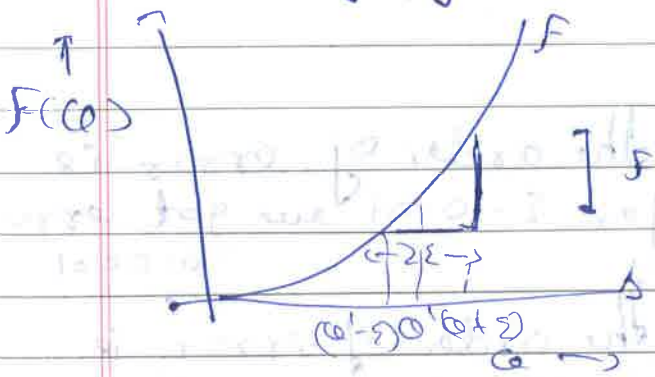
in other cases we could use other variance.
eg. [redacted] when we have tanh as our activation function then we could use $\sqrt{\frac{1}{n^{[l-1]}}}$ [redacted]

This is called Xavier Initialization.

sometimes we also use:
$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

→ Numerical approximation of gradients

⇒ Checking your derivatives computation.



let $F(\theta) = \theta^3$

$F(\theta+\varepsilon) - F(\theta-\varepsilon)$

$\varepsilon = 0.01$
$\theta - \varepsilon = 0.99$
$\theta = 1$
$\theta + \varepsilon = 1.01$

(i) $\frac{F(\theta+\varepsilon) - F(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$

⇒ $\frac{(1.01)^3 - (0.99)^3}{2 \cdot (0.01)} = 3.0001$

$g(\theta) = 3\theta^2 = 3$

approximation error :- 0.0001

(ii) Now trying :

$$\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$$

$$= \frac{(1.01)^3 - 1}{0.01} = 3.0301$$

error : 0.03

Then we would prefer to use

$$\left(\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}\right)$$ on gradient over

$$\left(\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}\right)$$

→ In this case the order of error is (ε^2) ⇒ for $\varepsilon = 0.01$ we got error = 0.0001

→ In this case the order of error is (ε) ⇒ for $\varepsilon = 0.1$ we get error = 0.03

→ Gradient checking

⇒ Take $W^{[1]}, b^{[1]}, \ldots W^{[L]}, b^{[L]}$ and reshape into a big vector θ.

$J(W^{[1]}, b^{[1]}, \ldots W^{[L]}, b^{[L]}) = J(\theta)$

=> Take $dw^{[1]}, db^{[1]} \ldots dw^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

$J(dw^{[1]}, db^{[1]} \ldots dw^{[L]}, db^{[L]}) = J(d\theta)$

=> Gradient check.

for each i:

$$d\theta_{approx}^{(i)} = \frac{J(\theta_1 + \varepsilon, \theta_2 + \varepsilon \ldots \theta_i + \varepsilon \ldots) - J(\theta_1 - \varepsilon \ldots \theta_i - \varepsilon)}{2\varepsilon}$$

use $\varepsilon = 10^{-7}$

its right ↑

Check $\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx 10^{-7}$ — (✓)

$\approx 10^{-5}$ — (check)

$\approx 10^{-3}$ — (X)

↓ something is wrong.

--> Gradient checking implementation notes:-

=> Dont use in training - only to debug.

=> If algorithm fails grad check, look at components to try to identify bug.

=> Remember regularization.

=> Doesn't work with dropout.

=> Run at random initialization; perhaps again after some time.

~~Regularization is explained.~~

→ python code for L2 regularization

let there be a 3 layer neural network with

cost = (older cost) + ((λ/2m) * (np.sum(np.square(w1)) + np.sum(np.square(w2)) + np.sum(np.square(w3)))

In backpropagation we make some changes only for dw_1, dw_2 & dw_3 terms:-

$dw_3 = 1/m (np.dot(dz_3, A2.T)) + ((\lambda/m) * w_3)$

$dw_2 = 1/m (np.dot(dz_2, A1.T)) + (\lambda/m * w_2)$

$dw_1 = 1/m (np.dot(dz_1, X.T) + (\lambda/m * w_1))$

PAGE No.

DATE : / /

week 2 (specialization 2)

→ mini-Batch gradient descent.

⇒ Batch vs mini batch gradient descent

Vectorization allows you to efficiently compute on m examples.

make mini-batches (suppose) from your main training set. Eg: You have $m = 5,000,000$ training examples. So you make mini batches with one mini-batch having 1000 training examples.

Then we have 5000 mini-batches.

$$X_{(n_x, m)} = [\underbrace{x^{(1)} x^{(2)} x^{(3)} \cdots x^{(1000)}}_{X^{\{1\}}_{(n_x, 1000)}} \underbrace{x^{1001} \cdots x^{2000}}_{X^{\{2\}}_{(n_x, 1000)}} \cdots \cdots \underbrace{\cdots x^{(m)}}_{X^{\{5000\}}_{(n_x, 1000)}}]$$

$$Y_{(1, m)} = [\underbrace{y^{(1)} y^{(2)} y^{(3)} \cdots y^{(1000)}}_{Y^{\{1\}}_{(1, 1000)}} \underbrace{y^{(1001)} \cdots y^{(2000)}}_{Y^{\{2\}}_{(1, 1000)}} \cdots \cdots \underbrace{\cdots y^{(m)}}_{Y^{\{5000\}}_{(1, 1000)}}]$$

⇒ mini-batch gradient descent

Repeat {

for $t = 1, \ldots, 5000$

{ forward prop on $X^{\{t\}}$

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

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

$$\vdots$$

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

Compute cost $J = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{1}{2 \cdot 1000} \sum \|w^{[l]}\|_F^2$

Backprop to compute gradients $J^{\{t\}}$ (using $X^{\{t\}}, Y^{\{t\}}$)

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

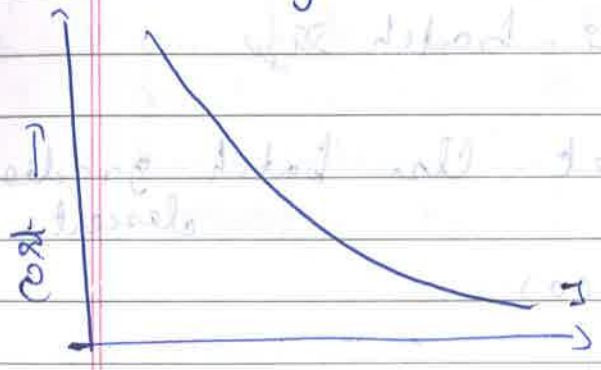
$b^{[l]} := b^{[l]} - \alpha \cdot db^{[l]}$

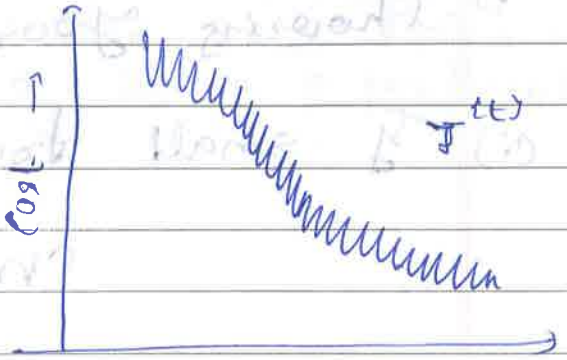
} This is 1 epoch "1 pass through training set"

}

→ Understanding mini batch gradient descent.

Batch gradient descent.





Plot $J^{\{t\}}$ computed using $X^{\{t\}}, Y^{\{t\}}$

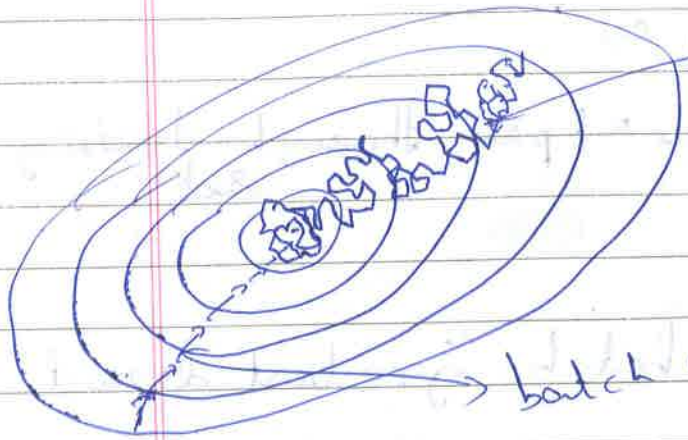
⇒ Choosing your mini-batch size

If mini batch size = m: Batch gradient descent $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

If mini batch size = 1: Every example is its own (Stochastic gradient descent

$(X^{\{1\}}, Y^{\{1\}}) = (X^{(1)}, Y^{(1)}) \ldots (X^{\{m\}}, Y^{\{m\}})$

In practise: choose between 1 & m.



=> Choosing your mini-batch size.

(i) If small train set: Use batch gradient descent.
(m ≤ 2000)

(ii) Typical mini-batch sizes:
→ 64, 128, 256, 512
2^6 2^7 2^8 2^9

Try to choose the size as a power of 2. This would make the code run faster as the memory systems in computer is designed in this way.

(Batch gradient descent) — (In between mini batch size) — (Stochastic gradient descent)

→ too long per iteration

→ fastest learning
→ vectorization
→ makes progress without processing entire training set

→ lose your speed up from vectorization.

→ Exponentially weighted averages

Eg) Temperature in London

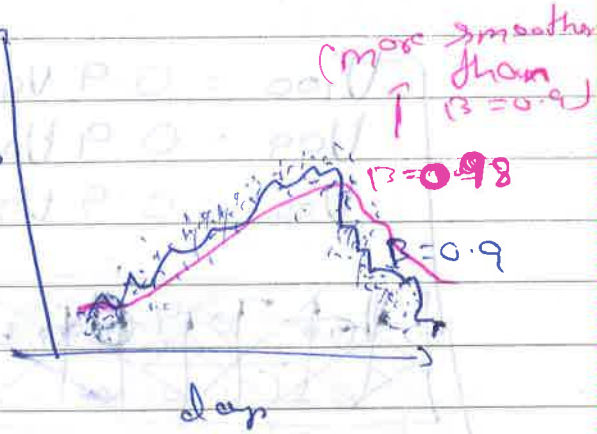
$\theta_1 = 40°F \quad 4°C$

$\theta_2 = 49°F \quad 9°C$

$\theta_3 = 45°F$ ⋮

$\theta_{180} = 60°F \quad 15°C$

$\theta_{181} = 56°F$ ⋮



now lets say to fit a line over this.
let V be the value of line at a particular temperature

$V = f(\text{days})$

$V_0 = 0$

$V_1 = (0.9) V_0 + \theta_1 (0.1)$

$V_2 = (0.9) V_1 + \theta_2 (0.1)$

$V_t = (0.9)(V_{t-1}) + \theta_t (0.1)$ → average

(V_t is approximately average over $\approx \frac{1}{1-0.9}$ days)

then

let $V_t = \beta \cdot (V_{t-1}) + \theta_t (1-\beta)$

(V_t is approximately average over $\frac{1}{1-\beta}$ days)

so if $\beta = 0.9$

then approximately average over 10 days

If $\beta = 0.98$ then approximately over 50 days.

-> Understanding exponentially weighted averages.

$V_t = \beta \cdot V_{t-1} + (1-\beta)\theta_t$

$V_{100} = 0.9\, V_{99} + (0.1)\cdot\theta_{100}$

$V_{99} = 0.9\, V_{98} + (0.1)\cdot\theta_{99}$

$V_{98} = 0.9\, V_{97} + (0.1)\theta_{98}$

$V_{100} = (0.1)\theta_{100} + 0.9((0.1)\theta_{99} + 0.9((0.1)\theta_{98} + 0.9(\ldots$

$\Rightarrow V_{100} = (0.1)\theta_{100} + (0.1)(0.9)\theta_{99} + (0.1)(0.9)^2\theta_{98} + (0.1)(0.9)^3\theta_{97} + (0.1)(0.9)^4\theta_{96}$

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$

$(1-\varepsilon)^{1/\varepsilon} \approx \frac{1}{e}$

-> Implementing exponentially weighted average

$V_0 = 0$

$V_1 = \beta \cdot V_0 + (1-\beta)\cdot\theta_1$

$V_2 = \beta \cdot V_1 + (1-\beta)\cdot\theta_2$

$V_3 = \beta \cdot V_2 + (1-\beta)\cdot\theta_3$

[p. 7.0]

$V_0 = 0$

Repeat {

get next θ_t

$V_\theta := \beta \cdot V_\theta + (1-\beta) \cdot \theta_t$

}

→ Bias correction in exponentially corrected averages.

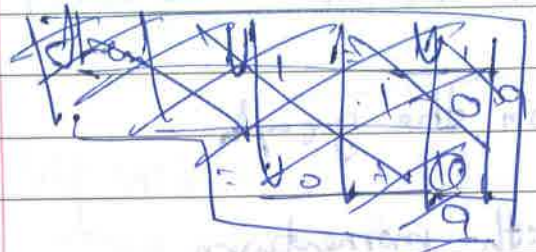
⇒ Bias correction.

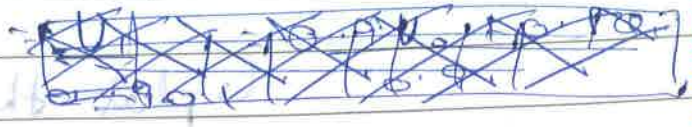
$$V^t := \frac{V^t}{1-\beta^t}$$

See the thing is that initially we initialize $V_0 := 0$ so when we calculate $V_1 := 0.9 V_0 + 0.1\theta_1$ then V_1 would be much lower than actual value? So what we could do is that we could write $V^t := \frac{V^t}{1-\beta^t}$

And this would happen for quite a few initial values.

Also for example $\beta = 0.9$





$$V_1 = \frac{0.9 V_0}{1-(0.9)^1} + \frac{0.1\theta_1}{0.1} = 0.9 V_0 + 0.1\theta_1$$

$V_0 = 0$

$\Rightarrow V_1 = 0.9 V_0 + 0.1\theta_1 = 0.1\theta_1$

$\Rightarrow V_2 = \frac{0.9(V_1) + 0.1\theta_2}{1-(0.9)^2} = \frac{0.9\theta_1 + 0.1\theta_2}{1-0.81}$

$\Rightarrow V_2 = \frac{0.9\theta_1}{0.19}$

let $\beta = 0.98$

$V_0 = 0$

$V_1 = (0.98)(V_0) + (0.02)(\theta_1)$

$\frac{V_1}{1-(0.98)^1} = \frac{(0.98)(V_0) + (0.02)(\theta_1)}{1-0.98}$

$\Rightarrow \frac{0.98(V_0) + (0.02)(\theta_1)}{0.02}$

$\Rightarrow 49V_0 + \theta_1$ → use this value to plot graph

now $V_2 = (0.98)(V_1) + (0.02)\theta_2$

$\Rightarrow V_2 = (0.98)((0.98)V_0 + (0.02)\theta_1) + (0.02)\theta_2$

$\Rightarrow V_2 = (0.98)^2 V_0 + (0.98)(0.02)\theta_1 + (0.02)\theta_2$

$\frac{V_2}{1-(0.98)^2} = \frac{V_2}{0.0396} = \frac{(0.98)^2 V_0}{(0.0369)} + \frac{(0.98)(0.02)\theta_1}{(0.0396)} + \frac{(0.02)\theta_2}{(0.0396)}$

$= 24.252 V_0 + 0.4949\theta_1 + 0.5050\theta_2$

↳ plot this on the graph

→ Gradient descent with momentum.

We do so to fasten the process we want :-

(↕ slower learning rate in this direction)

(↔ larger learning rate in this direction)

On iteration t :→

Compute dw, db on current mini-batch:

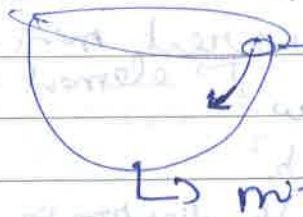
$V_{dw} = \beta \cdot V_{dw} + (1-\beta)\, dw$

$V_{db} = \beta \cdot V_{db} + (1-\beta) \cdot db$

$w := w - \alpha \cdot V_{dw}$

$b := b - \alpha \cdot V_{db}$

→ you can understand this by the analogy of :-



→ ball that is going towards the minima.

↳ minima

so for this ball we can see the equation as:

$V_{dw} = \beta \cdot V_{dw} + (1-\beta) \cdot dw$

↳ Velocity ↳ acceleration

↳ friction

just to get a better intuition

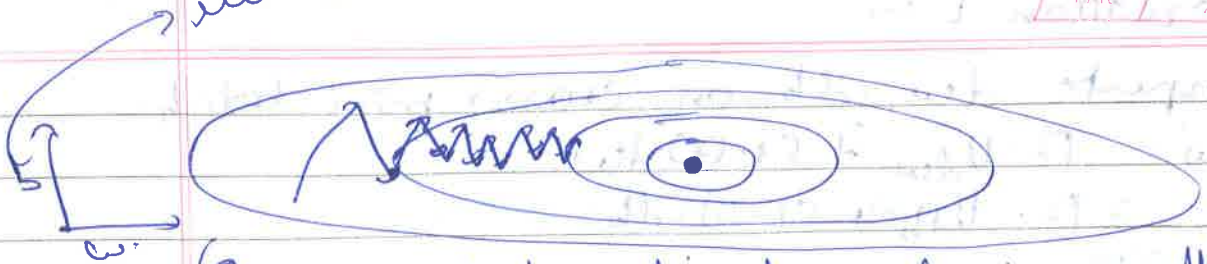
⇒ We initialize $V_{dw} = 0$ & $V_{db} = 0$ & we have hyperparameters α & β.

⇒ Often we don't implement bias correction, because we think that it is not that necessary here

→ RMS prop.

{P.T.O}

we want to reduce the disturbance here



(↕ we want to slow down learning in this direction)

(↔ we want to speed up learning in this direction)

On iteration t:

compute dw, db on current mini-batch

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2$ → element wise → (Relat

$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2$ → (Relat

(Previous value of S_{dw})

(here b means + the the direction of the joining coefficients missing)

$w = w - \alpha \cdot \frac{dw}{\sqrt{S_{dw} + \varepsilon}}$ $\quad b = b - \alpha \cdot \frac{db}{\sqrt{S_{db} + \varepsilon}}$

(previous value of S_{db})

(as this is relatively small, updates of w would be relatively larger

(as this is relatively large updates of b would be relatively smaller

we add this small number to ensure that the denominator doesn't become 0 ever.

→ Adam optimization algorithm

Initialization:

$V_{dw} = 0, \quad S_{dw} = 0, \quad V_{db} = 0, \quad S_{db} = 0$

{p. 7.0}

On iteration t:

Compute dw, db using current mini-batch:

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$ } "momentum" β_1

$V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

$S_{dw} = \beta_2 \cdot S_{dw} + (1-\beta_2) dw^2$ } "RMSprop" β_2

$S_{db} = \beta_2 \cdot S_{db} + (1-\beta_2) db$

$V_{dw}^{corrected} = \frac{V_{dw}}{(1-\beta_1^t)}$, $V_{db}^{corrected} = \frac{V_{db}}{(1-\beta_1^t)}$

(fuely small) (ely large)

$S_{dw}^{corrected} = \frac{S_{dw}}{(1-\beta_2^t)}$, $S_{db}^{corrected} = \frac{S_{db}}{(1-\beta_2^t)}$

$$w := w - \frac{\alpha \cdot V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon}$$

$$b := b - \frac{\alpha \cdot V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

⇒ Hyperparameters choice:

α: needs to be tuned

β_1 : 0.9 (dw)

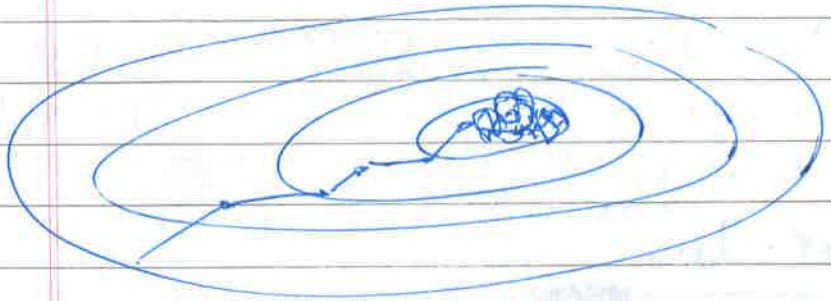
β_2 : 0.999 (dw^2)

ε : 10^{-8}

Adam = Adaptive + Moment estimation

→ Learning rate decay.

If we gradually reduce learning rate alpha then in the initial stages when our learning rate is still high our learning will be high but as we reach our minima we could reduce our learning rate so that our J value would just not keep buggering around minima like :-



⇒ 1 epoch :- 1 iteration / 1 pass through the data.

$$\alpha = \frac{1}{1 + (\text{decay rate}) * (\text{epoch number})}$$

so if $\alpha_0 = 2$ & decay rate = 1

then :-

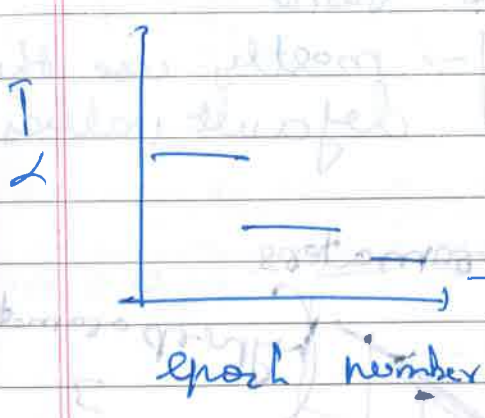
Epoch no.	α
1	0.1
2	0.067
3	0.05
4	0.04
.	.
.	.

⇒ Other learning rate decay methods:-

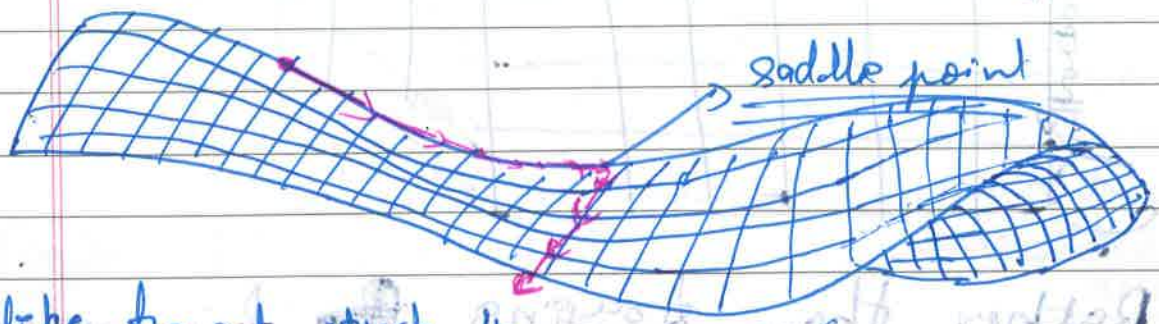
$\alpha = (0.95)^{\text{epoch-number}} \cdot (\alpha_0)$

↳ exponential decay.

$\alpha = \frac{k \cdot \alpha_0}{\sqrt{\text{epoch-number}}}$



→ The problem of local optima in neural network.



Unlike to get stuck in a bad local optima. This is because there are so many dimensions that it would ultimately go down & it is going down after being stuck at a saddle point for some time.

But these saddle points in plateaus can make learning very slow.

(Assume that this is a graph of J value across various parameters.)

Specialization 2 (week 3)

PAGE No.
DATE / /

→ Tuning hyperparameters

α → most imp to tune.
β (for momentum) } → Second most important to tune
hidden units
mini-batch size
layers } → Third most imp to tune
learning rate decay
β1, β2, ε } → mostly use these default values.
↳ 0.9 ↳ 0.999 ↳ 10^-8

⇒ Ways to tune hyperparameters.

⇒ Try random values

Rather than trying hyperparameters values in ascending values systematically try random values of hyperparameters.

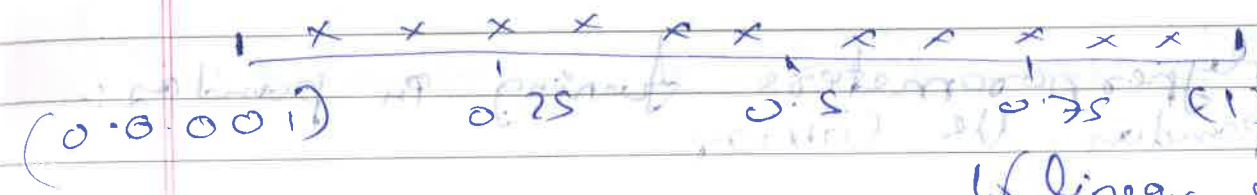
⇒ Coarse to fine

if this dot suits best then zoom in this region & try values closer to that.

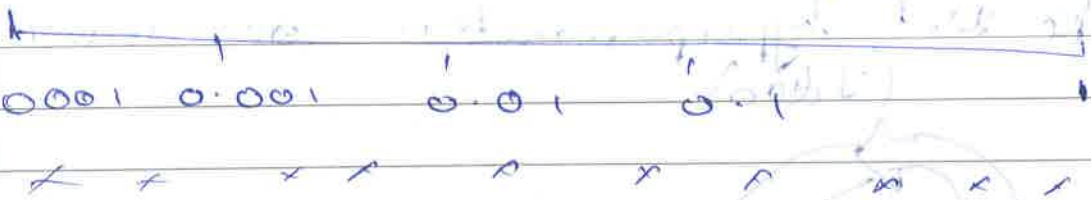
→ Using an appropriate scale to pick hyperparameters.

Eg:) $\alpha = 0.0001$ to 1

instead of searching on a linear scale between 0.0001 to 1 we would prefer to search α on a logarithmic scale.



(linear scale).



→ for this we write as:

$r = -4 * np.random.rand()$

$\alpha = 10^r$

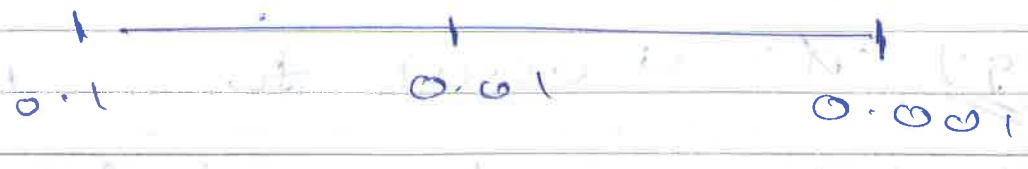
Eg:) β → used to calculate exponentially weighted averages.

$\beta = 0.9 \ldots 0.999$

Here also we ~~want~~ don't want to search on a linear scale.
So we will use logarithmic scale on $(1-\beta)$

This is because results are very sensitive to change in the value of β when it is close to one.

$1 - 17 = 0.1 \quad \ldots \quad 0.001$



$(10^{-1}) \qquad (10^{-3})$

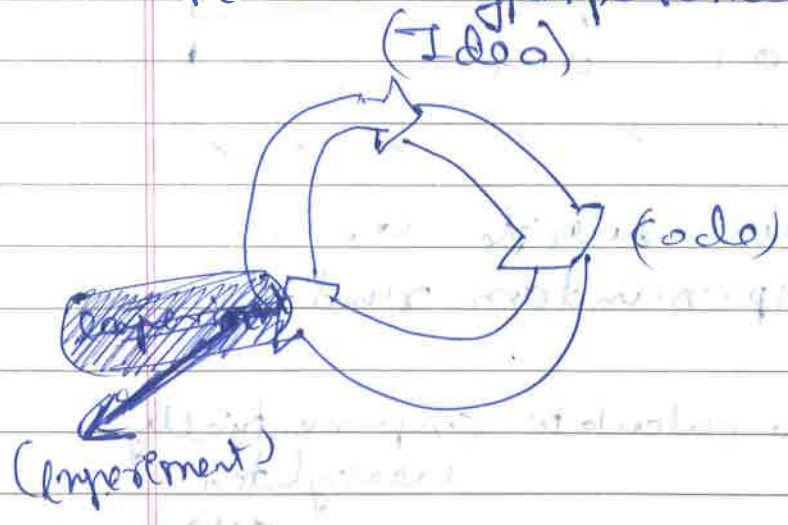
or $r \in [-3, -1]$

$1 - 17 = 10^{r}$

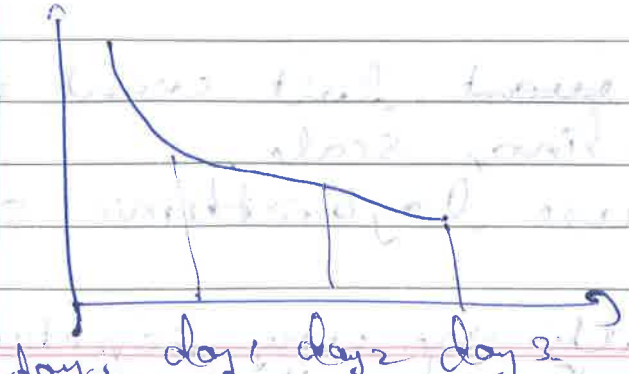
$17 = 1 - 10^{r}$

→ Hyperparameters tuning in pandas:- pandas Vs Caviar.

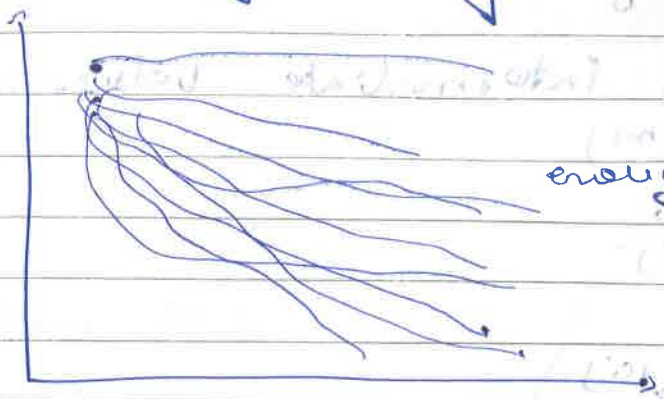
⇒ Re-test hyperparameters ~~or~~ occasionally



⇒ Either we could babysit one model (Pandas approach)
(when we do not have much computation power)



- Training many models in parallel. (Caviar approach)



(When we have enough computers then we do this)

→ Batch normalization.

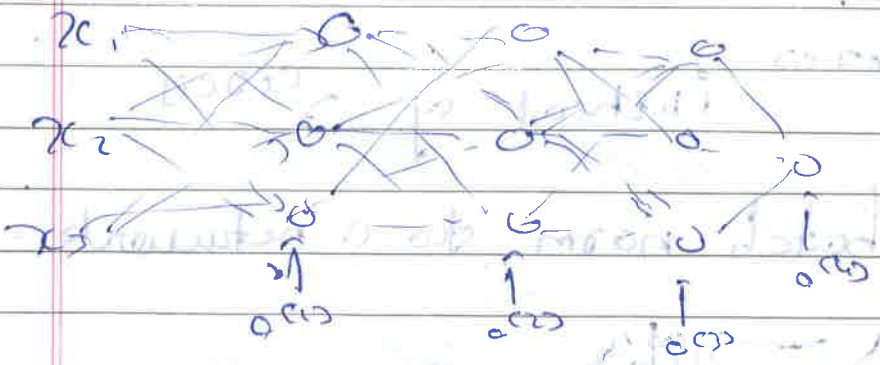
⇒ normalizing inputs to speed up learning.

$\mu = \frac{1}{m} \sum_i x^{(i)}$

$X = X - \mu$ → element wise.

$\sigma^2 = \frac{1}{m} \sum_i X^{(i)2}$

$X = \frac{X}{\sigma}$] → After doing this we have a mean of 0 and standard deviation of 1.



If we normalize x_1, x_2 & x_3 then it would become faster to calculate w_1 & b_1. Similarly if we could normalize $z^{[2]}$ then it would become faster to calculate w_2 & b_2. This is what batch norm does.

Implementing batch norm.

Given some intermediate values in N.N

$z^{(1)}, \ldots, z^{(m)}$

$\hookrightarrow z^{[l](i)}$

$$\mu = \frac{1}{m} \sum z^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum (z_{(i)} - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

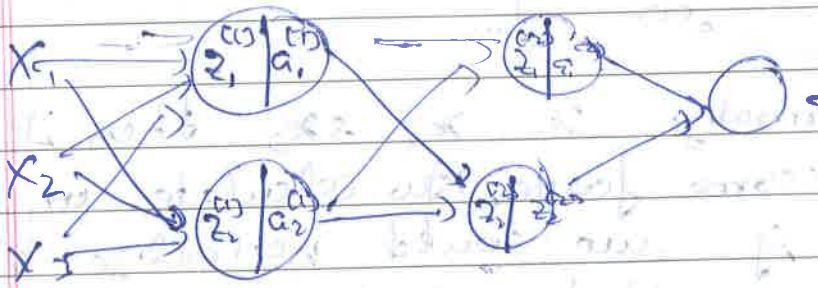
↳ we add this so that denom. inator never become 0.

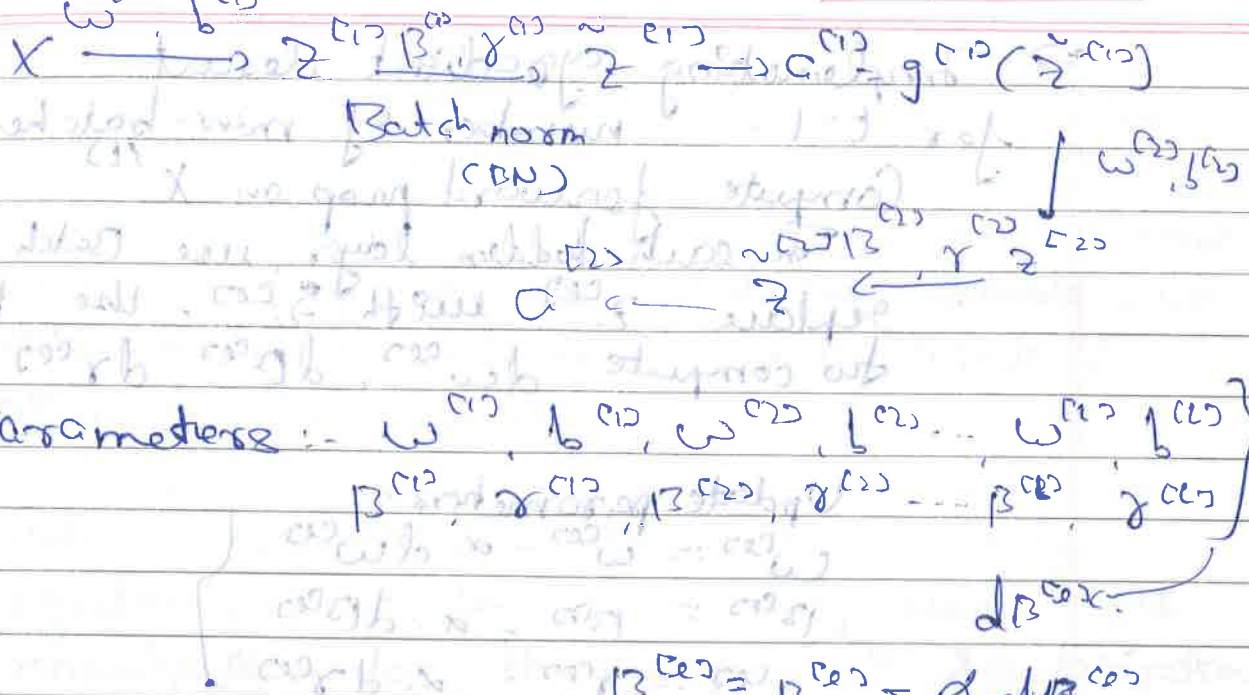
$$\tilde{z}^{[l](i)} = \gamma z^{(i)}_{norm} + \beta$$

where γ & β are learning parameters of the model.

use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$

→ Adding batch norm to a network :-





Parameters :- $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \ldots W^{[L]}, b^{[L]}$
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]} \ldots \beta^{[L]}, \gamma^{[L]}$

$d\beta^{[2]}$

$\beta^{[2]} = \beta^{[2]} - \alpha \cdot d\beta^{[2]}$

⇒ Working with mini batches

$X^{\{1\}} \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \rightarrow g^{[1]}(\tilde{Z}^{[1]}) = a^{[1]} \rightarrow \ldots$

$X^{\{2\}} \rightarrow$

$X^{\{3\}} \rightarrow$

$X \rightarrow$

After batch norm

$Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$ → Here cancel this term. This is because anyway we are going to subtract mean & this will cancel out

$Z^{[l]}_{norm} = \frac{Z^{[l]} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$

$\tilde{Z}^{[l]} = \gamma^{[l]} Z^{[l]}_{norm} + \beta^{[l]}$

=> Implementing gradient descent

for t=1 ... number of mini-batches
Compute forward prop on $X^{\{t\}}$
In each hidden layer, use Batch norm to replace $z^{[l]}$ with $\tilde{z}^{[l]}$. Use backprop to compute $dw^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$, $db^{[l]}$ (not needed)

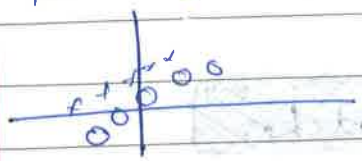
Update parameters:-

$$w^{[l]} := w^{[l]} - \alpha \, dw^{[l]}$$
$$\beta^{[l]} := \beta^{[l]} - \alpha \, d\beta^{[l]}$$
$$\gamma^{[l]} := \gamma^{[l]} - \alpha \, d\gamma^{[l]}$$

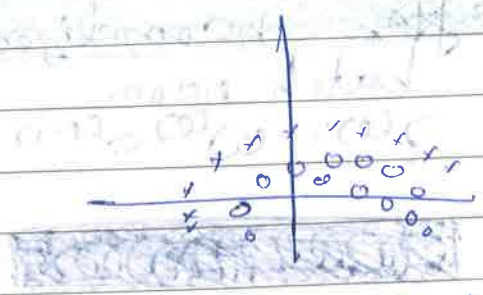
- This works with momentum, RMS prop, Adam.

→ Why does batch norm work

[a] we train a model on these data:



and we try our model on data:



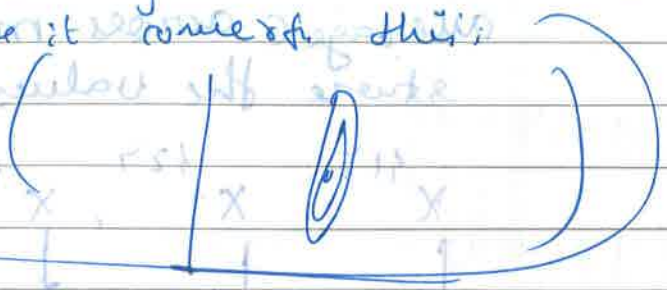
So our model may not be able to fit so well. This problem is called covariate shift.

↳ this means that if we train a model on $X \to Y$ & we change the distribution of X then we have to retrain our model.

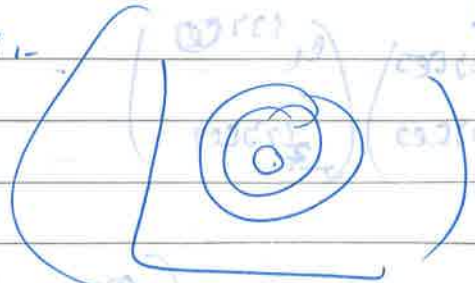
The basic idea behind batch normalization is to limit covariate shift by normalizing the activations of each layer (transforming the inputs to be mean 0 and unit variance). This, supposedly, allows each layer to learn on a more stable distributions of inputs, and would thus accelerate the training of the network.

Batch normalization also induces regularization but we rarely use batch normalization for the purpose of regularization.

→ Actually some recent papers state that limiting the co-variate shift is not the primary reason, probably it isn't even an important reason why batch normalization works.
You can I think safely say that batch normalization works because it converts this:



→ to this:



→ thus making gradient descent s learning faster.

→ Batch norm at test time.

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

Now during training we can calculate μ & σ as we have a lot of examples or some examples [while] training. But this might not be case while testing.

Then we do this while testing :-
estimate μ, σ^2 using exponentially weighted average (across mini batches)
store the values of μ & σ^2 for all [mini] batches

$X^{\{1\}}$, $X^{\{2\}}$, $X^{\{3\}}$, ...

$\mu^{\{1\}}[e]$, $\sigma^{2\{1\}}[e]$; $\mu^{\{2\}}[e]$, $\sigma^{2\{2\}}[e]$; $\mu^{\{3\}}[e]$, $\sigma^{2\{3\}}[e]$ ....

→ μ

→ σ^2

(exponentially weighted averages of stored μ's

(exponentially weighted averages of stored (σ^2)'s)

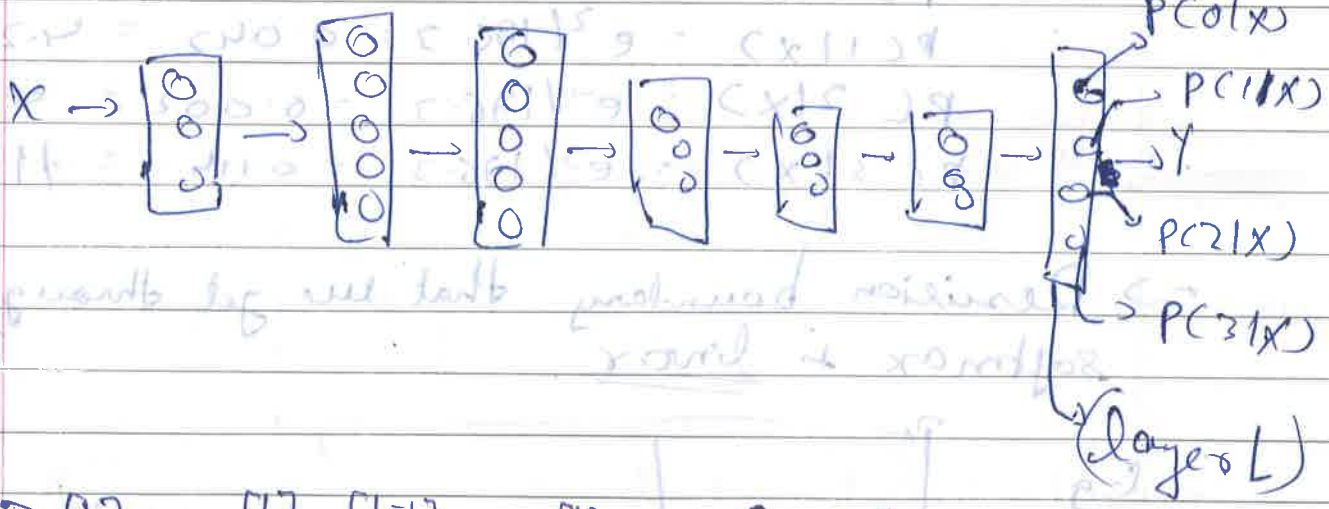
And then calculate

$$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\tilde{z} = \gamma z_{norm} + \beta$$

→ Softmax regression:
(Generalization of logistic regression so that we can classify multiply classes instead of just binary classes)

Eg 0 → none of the above
1 → cat
2 → [redacted] dog.
3 → chick



$$z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]} \quad (4,1)$$

Activation function:

$$t = e^{z^{[L]}}$$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i} \qquad a^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

Eg $z^{(i)} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$

$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$

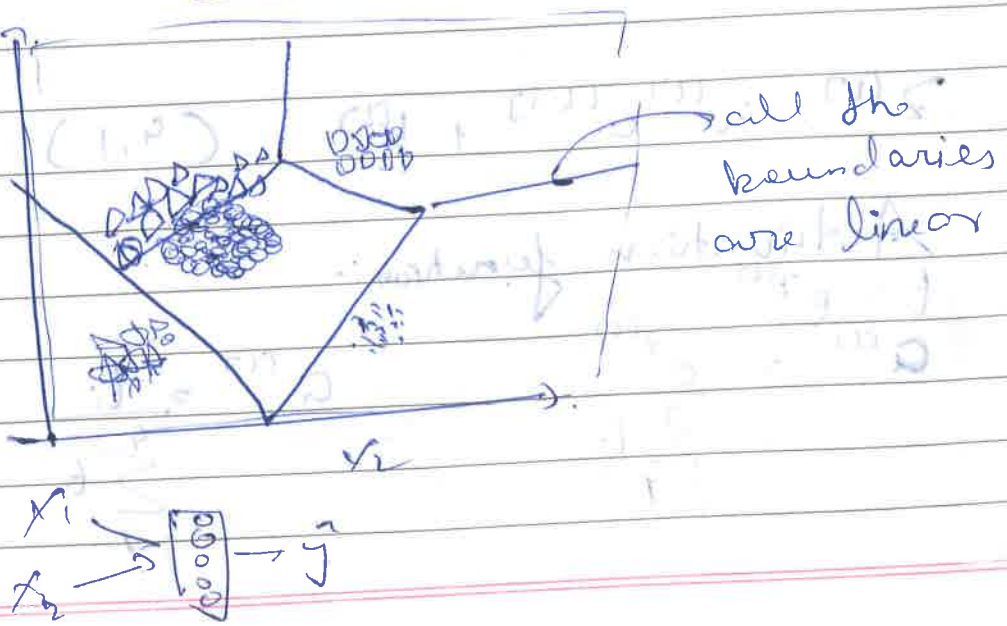
$\sum_{i=1}^{4} t_i = 176.3$

$a^{(i)} = \frac{t}{176.3}$

Thus $P(0|x) = e^5/176.3 = 0.842 = 84.2\%$
$P(1|x) = e^2/176.3 = 0.042 = 4.2\%$
$P(2|x) = e^{-1}/176.3 = 0.002 = 0.2\%$
$P(3|x) = e^3/176.3 = 0.114 = 11.4\%$

=> Decision boundary that we get through softmax is linear.

Eg



→ Training softmax classifier

⇒ Softmax regression is the generalization of logistic regression to C classes.

⇒ hardmax is just telling (1/0)'s softmax is telling the probabilities.

Eg. $\begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$ $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

softmax hardmax

→ This means that if we reduce softmax regression to just two classes it converts to logistic regression.

Eg: $\begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix}$ so as both the probabilities add up to one so we don't need to calculate both of them, even calculating one of them would suffice. So while doing so it is reduced to logistic regression. Don't go into the detailed math of it. It just reduces to that.

{p. 7.0}

=> loss function:-

$$L(\hat{y}, y) = -\sum_{j=1}^{n_y} (y_j)(\log(\hat{y}_j))$$

so in this example:

$$y^{(1)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad a^{[L](1)} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

$$L(\hat{y}, y) = -[(0)(\log 0.3) + (1)(\log 0.2) + (0)(\log 0.1) + (0)(\log 0.4)]$$

$$= -\log(0.2)$$

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

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

now as we want $L(\hat{y}, y)$ to be small & $\hat{y}$ lies from 0 to 1, so we would want $\hat{y}$ to be as close to 1 as possible to minimise loss function.

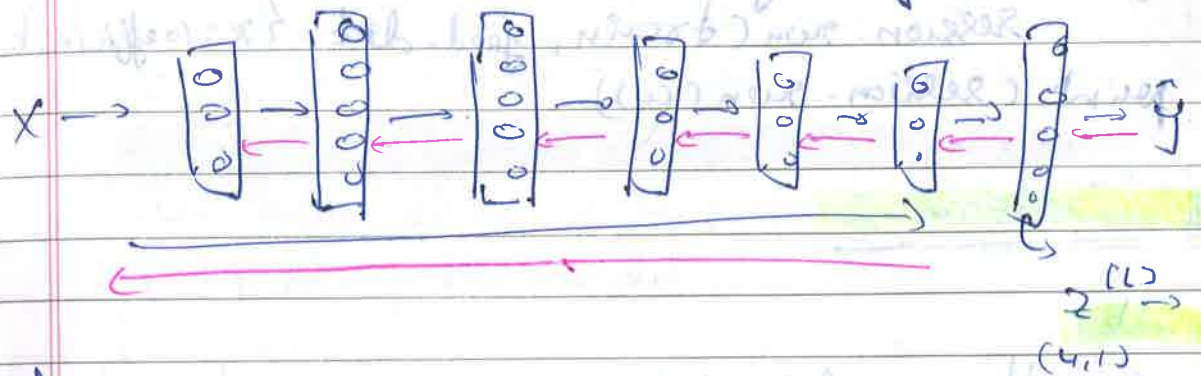
$$Y = [y^{(1)}, y^{(2)} \ldots y^{(m)}]$$

$$= \begin{bmatrix} 0 & 0 & \\ 1 & 0 & \cdots \\ 0 & 1 & \\ 0 & 0 & \end{bmatrix}_{(4,m)}$$

$\hat{y} = [\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(m)}]$

$= \begin{bmatrix} 0.7 & \cdots & \\ 0.2 & \cdots & \\ 0.1 & \cdots & \\ 0.1 & \cdots & \end{bmatrix}_{(4,m)}$

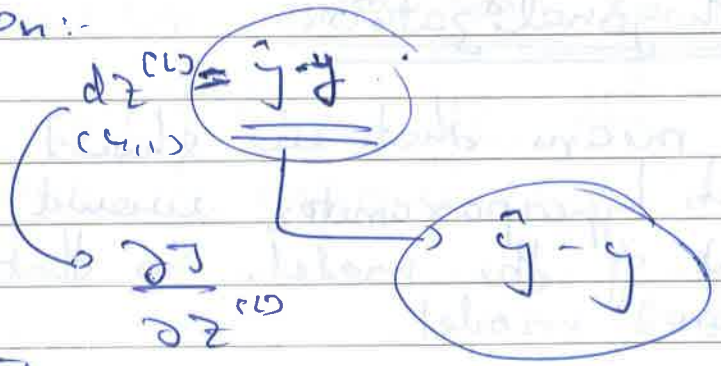
⇒ Gradient descent with Softmax :-



$z^{[1]} \to a^{[1]} = \hat{y}$
$(4,1)$

$L(\hat{y}, y)$

Backpropogation :-



→ Tensor Flow

Code example :-

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0], dtype = tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
# (w-5)**2

train = tf.train.GradientDescentOptimizer(0.01).minimize(cos
```

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

} with tf.Session() as sess:
session.run(init)
print(sess.run(w))

for i in range(1000):
session.run(train, feed_dict = {x : coefficients})
print(session.run(w))

Specialization 3

Week 1.

=> Orthogonalization

This means that we should know that which hyperparameter would effect which aspect of the model. So that we can make a good model.

=> Chain of assumptions in ML
-> Fit training set well on cost function
(bigger network, Adam optimization ...)
↓
Fit dev set well on cost function
(Regularization, Bigger training set)
↓
Fit test set well on cost function
(Bigger dev set)

↓.

Performs [redacted] well in real world (change dev set or cost function)

Try the not use "Early stopping" [redacted] because this would effect both fitting on training set and dev set, so this is against orthogonalization as it states that it is better [redacted] if we tune those hyperparameters that tune only one part of the model.

→ Single number evaluation metric

=> Precision :- (% of examples that are actually cats over the examples which are classified as cats)

=> Recall :- (% of examples that are actually cats over the examples which are actually cats)

=> F1 score :- Harmonic mean of precision and recall :- $\left(\frac{2}{\frac{1}{P}+\frac{1}{R}}\right) \Rightarrow \left(\frac{2PR}{P+R}\right)$

→ Satisficing and optimizing metric

Classifier	Accuracy	Running time
A	90%	80ms
B	92%	95ms
C	95%	1,500ms

Optimizing → Satisficing

We could do:-

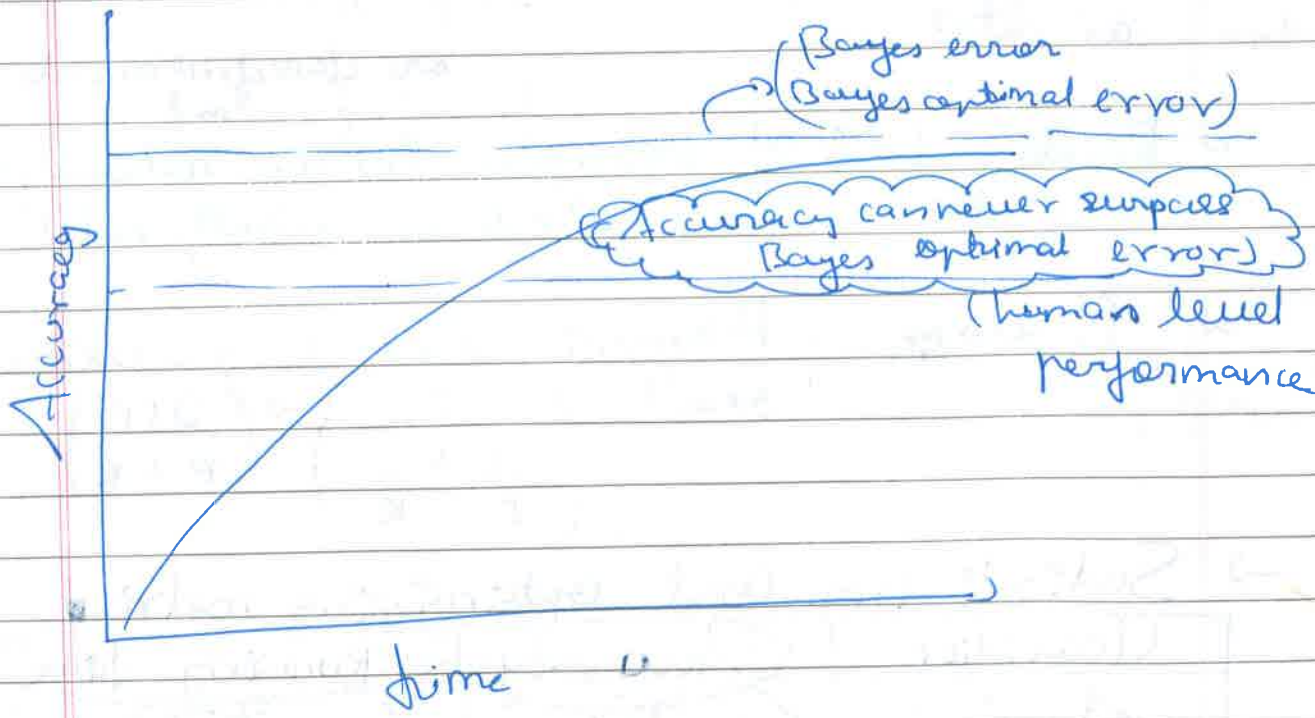
~~Optimizing metric~~

satisficing metrics:- That we would accept the model if it satisfies this threshold minimum condition.

We could in this case choose that the satisficing condition in Running time is that it should be less than 100ms.

-> Train / dev / test distributions.
(Dev set) & (test set) should come from the same distribution

-> Comparing to human-level performance



If the accuracy of the model is below human level performance then there is a scope of improvement.

→ Avoidable bias.

Eg1) Recognizing cat images

Human error	1%.
Training error	8%.
Dev error	10%.

(Focus on bias reduction)

Eg2) Recognizing cat images (very low resolution)

Human error	7.5%.
Training error	8%.
Dev error	10%

(Focus on variance reduction)

→ avoidable bias :- 7%.

→ avoidable bias :- 0.5%.

→ Human level error

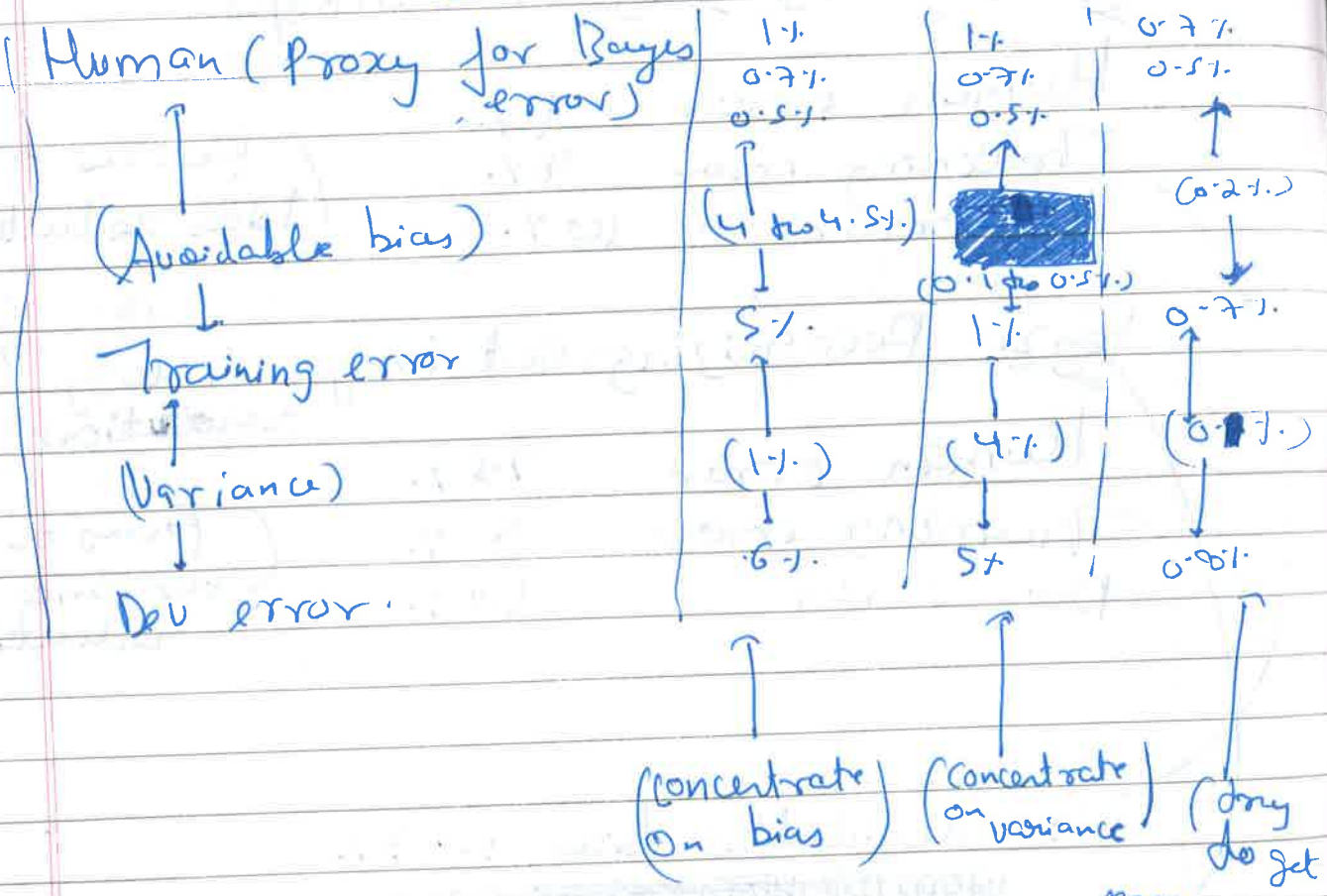
⇒ medical image classification example :-

(a) Typical human - 3%.
(b) Typical doctor - 1%.
(c) Experienced doctor - 0.7%.
(d) Team of experienced doctor - 0.5%.

Now Bayes error ≤ 0.5%.

We might consider human level error to be 0.5% to check the model.

=> Error analysis example.



If we have an estimate on human level error then we can estimate on avoidable bias, variance & hence make decision that on which thing should we work on.

and use at proxy for bayes level error.

→ Surpassing human level performance

Eg1

	Error (Eg 1)	Error (Eg 2)
Team of humans	0.5 %	0.5 %
One human	1 %	1 %
training error	0.6 %	0.3 %
Dev error	0.8 %	0.4 %

Eg where ML significantly surpasses human-level performance :-

→ Online advertisement
→ product recommendation
→ logistics (predicting transit time)
→ loan approvals.

in some cases ML has also surpassed human level performance :-

these cases are :-

→ Medical diagnosis (Skin cancer .. etc)
→ Speech recognition
→ computer vision

→

Week 2

→ Carrying out error analysis

=> Example that you are building a model that would recognize cat's to tell whether that there is a cat in the picture.

So now after building the initial model we could analyze the [redacted] errors

Ideas for cat detection:-

1) Fix pictures of dogs being recognized as cats
2) Fix great cats (lions, panthers, etc.) being misrecognized
3) Improve performance on blurry images.
4) misclassifying instagram filters as cats

Images	Dog	Great cats	Blurry	Instagram	Comments
1	✓			✓	Pitbull
2			✓	✓	[redacted]
3		✓	✓		
⋮	⋮	⋮	⋮	⋮	
% of total	8%	43%	61%	12%	

→ A table like this is made on error images so that we can create categories or solve problems to make our model better.

→ Cleaning up incorrectly labelled data

This is the dataset that is wrong, in which the data that we are inputting into the model for training are incorrectly

It is more important to correct incorrectly labelled data in dev/test set as compared to training set.

labelled

Sometimes it is fine to train ML models on mislabelled data when the error is random.

The problem arises when the input that feed in is incorrectly labelled systematically. Eg. when all white dogs are incorrectly labelled as cats

So we make a table on all error images labelled data:-

Image	Dog	Great cat	Blurry	Incorrectly labelled	Comments
98					
99				✓	Labeller missed cat in the background
100		✓			
⋮ Rows in 200	⋮	⋮	⋮	⋮	⋮
% of total	8%	43%	61%	6%	

Here overall dev set error :- 10%
Errors due to incorrect labels :- 0.6%
Errors due to other causes :- 9.4%

So there is not much need to concentrate on incorrectly labelled data now.

But suppose you have an overall dev set error of 10%. Then there is need to correct the incorrectly labelled data.

=> Correcting incorrect dev/test set examples

- Apply same process to your dev and test sets to make sure they continue to come from the same distribution.

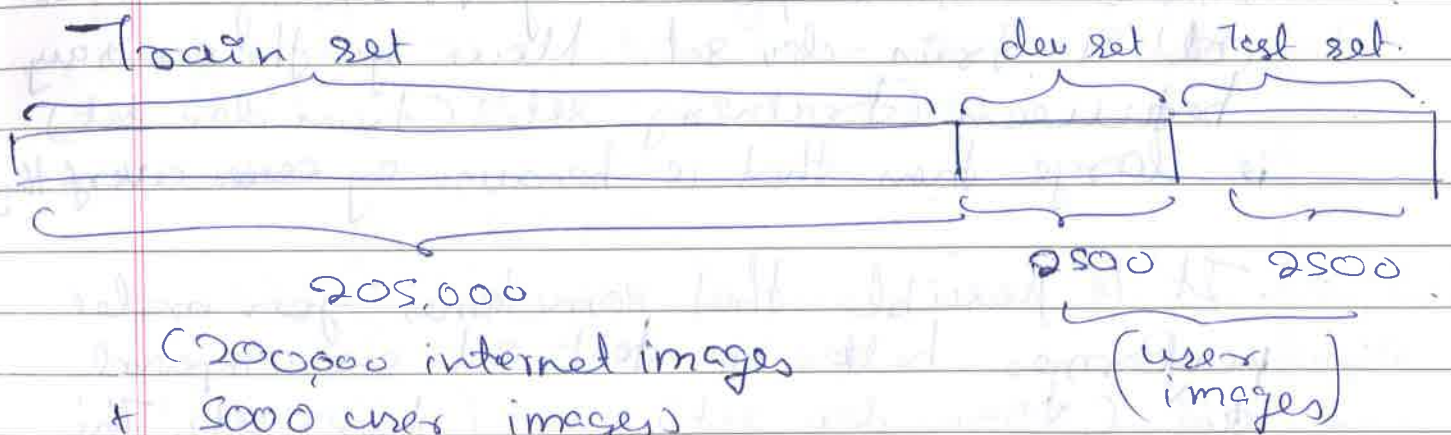
- Consider examining examples your algorithm got right as well as ones it got wrong.

=> It is important that we make sure (dev) & (test) set are from the same distribution even though training set is not so.

-> A tip:-
=> Set up dev/test set and metric
=> Build initial system quickly
=> Use Bias / Variance analysis and error analysis to prioritize next steps.
Guideline:- Build your first system quickly, then iterate

-> Training & testing on different distributions
=> Let's say that you have to make a cat recognizer model.
So for testing users would input data (images of cat) clicked from their phone. But for training you have 200,000 images of [redacted] cat taken from the internet (professionally clicked with much

better quality). You do a survey and ask people to click cat photos and send it to you, like this you get 10,000 images. Now these image come from the same distribution as the test set but the dilleme is that you have 200,000 images of cats that dont come from the same distribution as the test set so the most preferable option should be the following:



→ Bias & Variance with mismatched distribution.

Human level
(avoidable bias)
Training set error
(variance)
Train - dev set error
(data mismatch)
Dev error
(degree of overfitting is coming from same distribution)
Test error

=> Some times as we have seen the distribution of dev set & training set might be fairly different so if we train our model on training set and then test it on dev set, and if we get low accuracy on dev set then we may not know that the lower accuracy is due to overfitting or data mismatch.

To solve this problem we randomly remove a small portion of training set and call it train-dev set. Now if the accuracy between (training set) & (train-dev set) is large then that is because of ~~error~~ overfitting.

=> It is possible that sometimes your model performes better on test set as compared to (train-dev set) and (train set). This might be because test examples may be fairly simple.

-> Generating artificial data.

=> Eg you need to generate artificial data of people in car for a rear view mirror speech recognition model. And you have 10,000 hours of normal talking data of people in quiete environment & 1 hour of car noise data.

↳ (just)

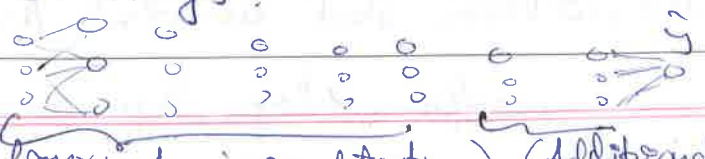
What you could do is that shuffle that 1 hour of just car noise data and overlap with 10,000 hours of quiete data to generate artificial data.

This would pose a problem though. Our model may overfit to that 1 hour of car sounds. For normal human ears, all of 10,000 hour of car noise is unique and 1 hour of car noise overlapped 10,000 times might sound the same but the computer may overfit to just that 1 hour of car noise.

→ Transfer learning

Transfer learning is when we use one machine learning model to solve another problem. For example you built a model for image detection. Now you want to build a model for radiology X-ray detection. Now the basis of radiology & image recognition is same, like boundary recognition, shape recognition. So it would make sense to train radiology model on image recognition model after adding some additional neural layers on the existing neural network.

And also this would make sense when we have just 100-1000 images for radiology detection but 10,000,000 examples for image detection because we can't train a radiology model solely on the basis of 100 images.



(Initial layers for image detection) (Additional layers for radiology detection)

5) When does transfer learning make sense:

(i) Task A & task B have the same input X

(ii) You have more data on task A as compared to task B

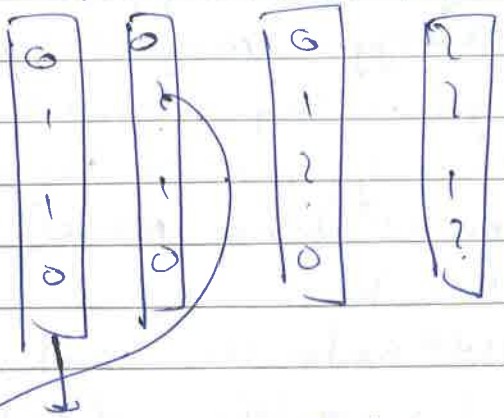
(iii) Low level features from A could be helpful for learning B.

→ Multi-task learning.

Suppose we have to make a model that would give us an output like:

- 0 → (0/1) for traffic light
- 0 → (0/1) for stop sign
- 0 → (0/1) for humans on road
- 0 → (0/1) for trucks on road.

so this would mean:



(no traffic light, there is stop signal, there are humans and there is no truck)

→ (This would mean that we don't know weather there is a ~~traffic~~ stop sign or not)

$$\text{loss} : \underset{(4,1)}{\hat{y}^{(i)}} \rightarrow \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L(\hat{y}_j^{(i)}, y_j^{(i)})$$

$$\downarrow$$

$$-\left(y_j^{(i)} \log \hat{y}_j^{(i)} - (1-y_j^{(i)}) \log(1-\hat{y}_j^{(i)})\right)$$

So the other way to solve this problem could have been to make 4 different neural network models for 4 of these tasks. But it is better to make a single neural network for multiple such tasks because the initial layers for all of these tasks would be same and we could combine all the dataset for these 4 tasks for a single dataset for our neural network. So our combined neural network would have a much larger dataset than the datasets available.

(This is not cut (read it also))

Currently multi-tasking is being used in computer vision.

Still today transfer learning is used more than multi-task learning.

→ End-to-End deep learning.

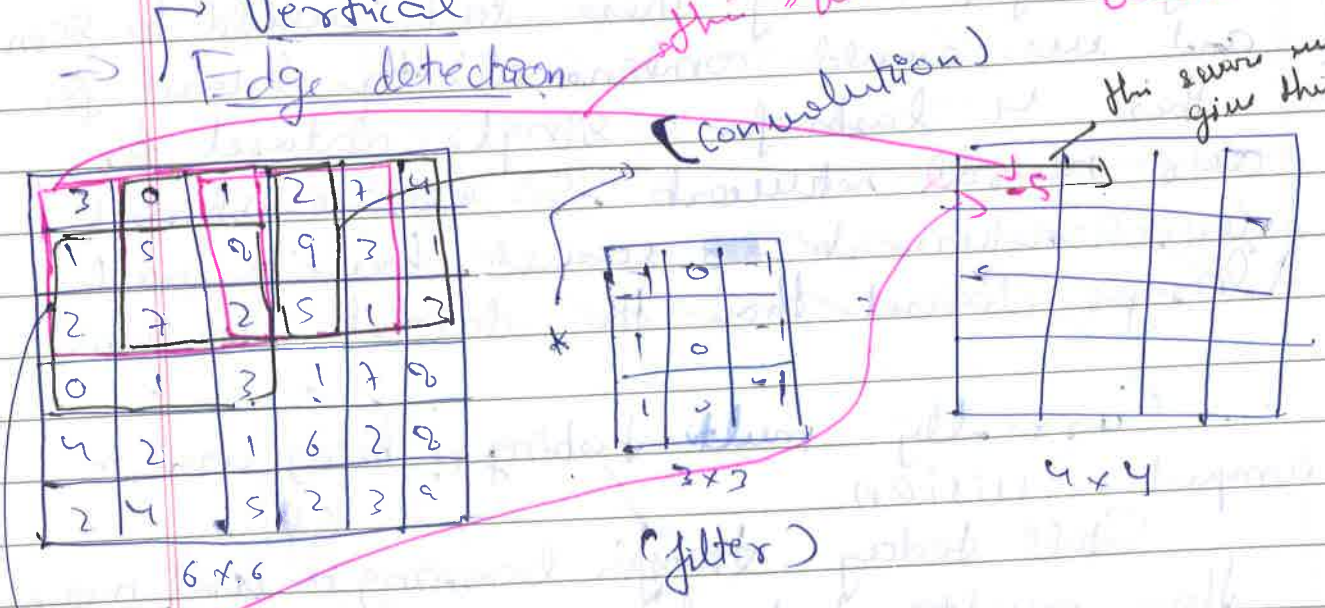
Initially we used to break the learning process into different parts. For example if for a voice recognition system we would first make a model to recognize phonems (sounds like "a", "b", "c"... etc) then further on.

Because of this the algorithm many times does not have a free hand to go about thus restricting its performance. But for this (end to end) deep learning we need large amounts of data. If we have less data then it would be better to manually guide the model at steps.

Specialization 4 week 1

Convolution neural networks

→ Vertical Edge detection



looking at the first (3×3) square

$$\Rightarrow (3)(1) + (1)(1) + (2)(1) + (0)(0) + (5)(0) + (0)(7) + (1)(1) + (8)(-1) + (2)(-1)$$

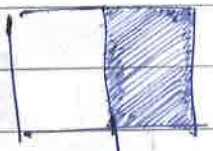
$$\Rightarrow 3 + 1 + 2 - 1 - 8 - 2$$

$$= -5$$

Eg:

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6



edge in the picture

*

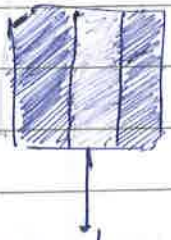
1	0	-1
1	0	-1
1	0	-1

6 × 6

=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

4 × 4



we have detected the output edge.

Eg: now if we change our initial matrix to:-

0	0	0	10	10
0	0	0	10	10
0	0	0	10	10
0	0	0	10	10
0	0	0	10	10

(6 × 6)

* Convolution (3 × 3)

0	-30	-30	0
0	-30	-30	0
0	-30	-30	0
0	-30	-30	0

we will have -30 instead of +30.

→ horizontal edge detector:-

1	1	1
0	0	0
-1	-1	-1

PAGE No.
DATE / /

Eg:

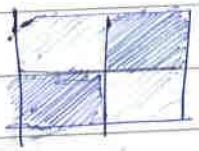
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10

$*$

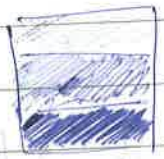
1	1	1
0	0	0
-1	-1	-1

$=$

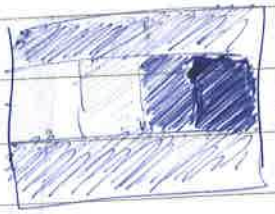
0	0	0	0
30	10	-10	-30
30	10	-10	-30
0	0	0	0



$*$



$=$



⇒ Now we could choose different filters instead of

1	0	-1
1	0	-1
1	0	-1

.

Some of the other popular filters are:

1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
-3	0	-3

Schon filter

} → Vertical

1	2	1
0	0	0
-1	-2	-1

3	10	3
0	0	0
-3	-10	-3

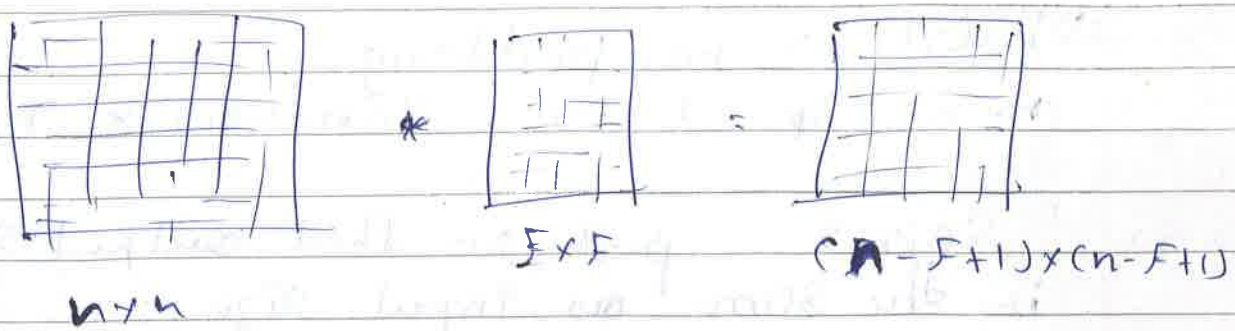
} → horizontal

⇒ We could also learn these as parameters.

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

do not only detect vertical/horizontal edges but do also detect edges at a certain angle. Eg. 75° edges.

→ Padding.

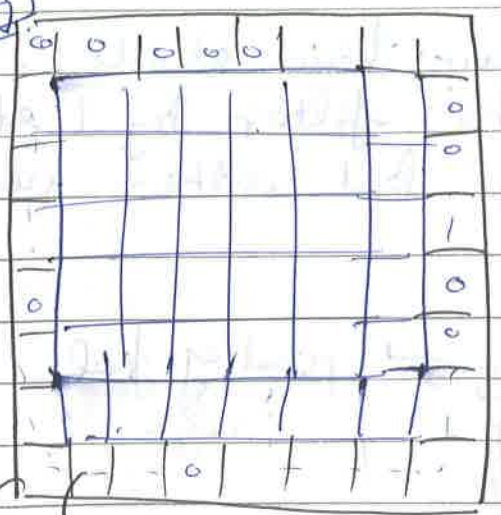


⇒ The disadvantage of this is that the size of the image (pixels) is reduced from (n×n) to (n-F+1)×(n-F+1)

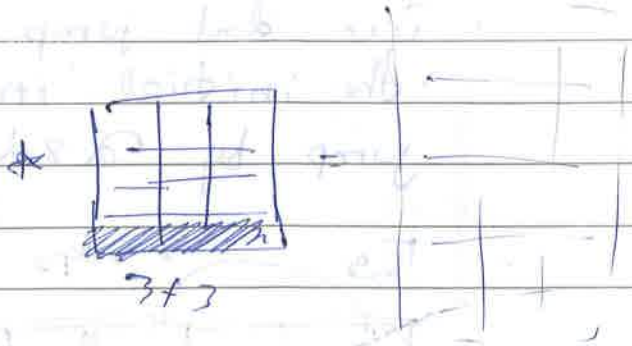
⇒ One more disadvantage is that the pixels at the edge give relatively less info than pixels in the middle. This is because pixels in the middle are used more to determine the output image.

To solve this we introduce padding. Padding is adding layers on the border of initial image.

Eg)



Padded layer



(n-F+1+2P)×(n-F+1+2P)

P = padding = 1

Conventionally we are padding with 0

=> Valid and same convolutions.

=> "Valid" → no padding.

$(n \times n) * (f \times f) = (n-f+1) \times (n-f+1)$

=> "Same": pad so that output size is the same as input size.

$(n \times n) * (f \times f) = (n-f+1+2p) \times (n-f+1+2p)$

we need:

$n = n-f+1+2p$

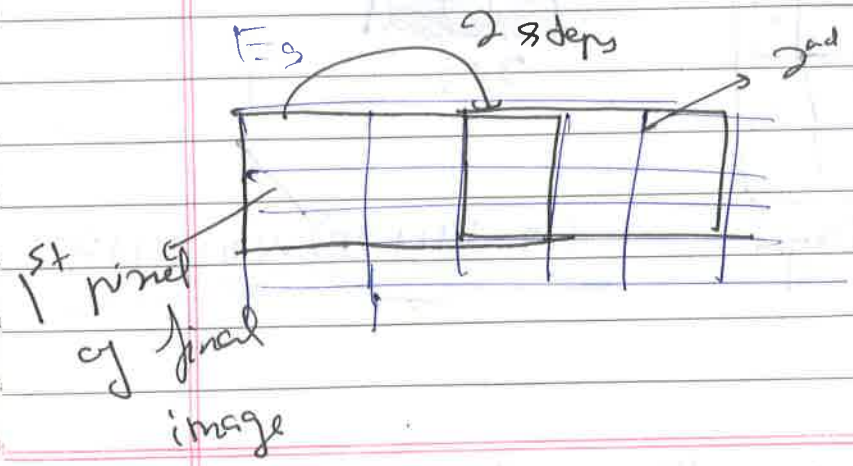
=> $2p = f-1 \Rightarrow p = \frac{f-1}{2}$

f is almost always odd. We rarely see f as even.

→ Strided convolution.

Example when we have stride = 2 then we don't jump the filter by 1 step on the initial image but rather we would jump by 2 steps

Eg:



Let s be the number of strides then:

→ $(n \times n) * (F \times F) = \left[\frac{n + 2p - F + 1}{s}\right] \times \left[\frac{n + 2p - F + 1}{s}\right]$

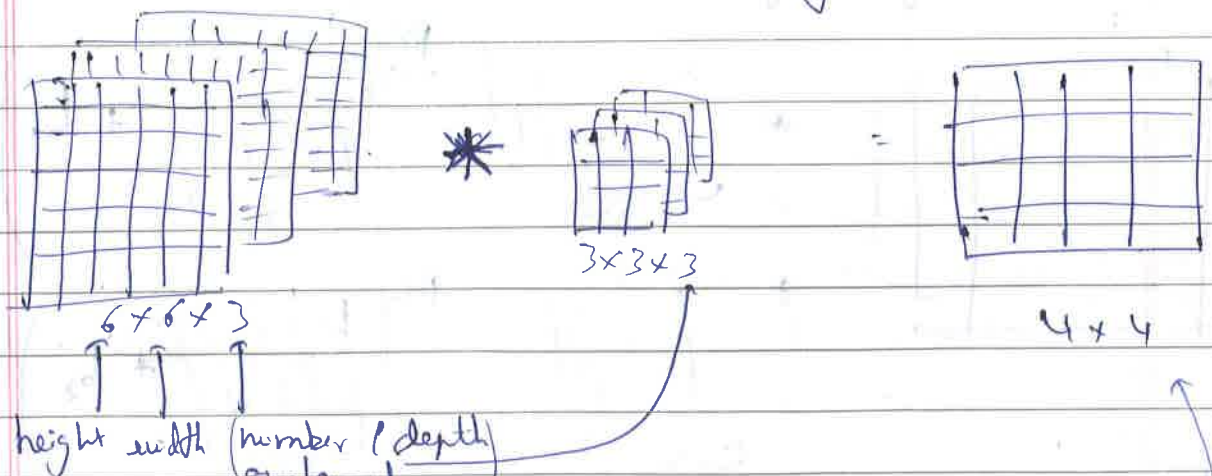
(∵ G.I.F)

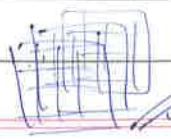
Note:- What we did, what we call convolution is called cross-correlation in mathematics. But we call it convolution in deep learning. Convolution in mathematics is little different. it is basically rotating the filter 180° clockwise and then applying it. 

(or you could just flip it vertically then horizontally)

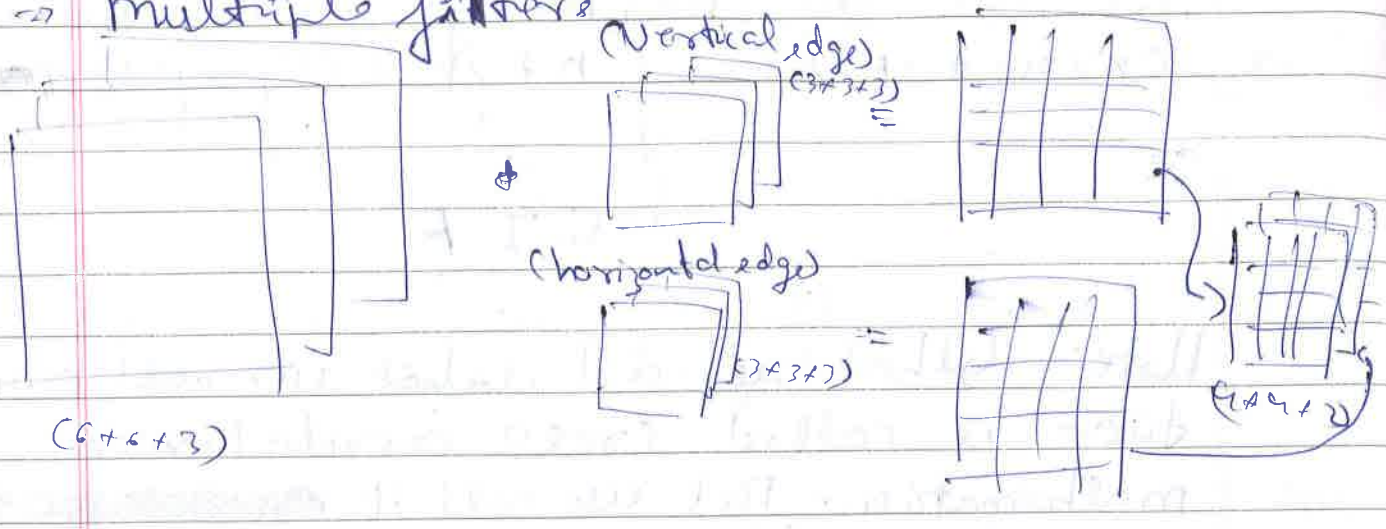
→ Convolutions over volumes.

→ Convolutions on RGB image.





→ multiple filters

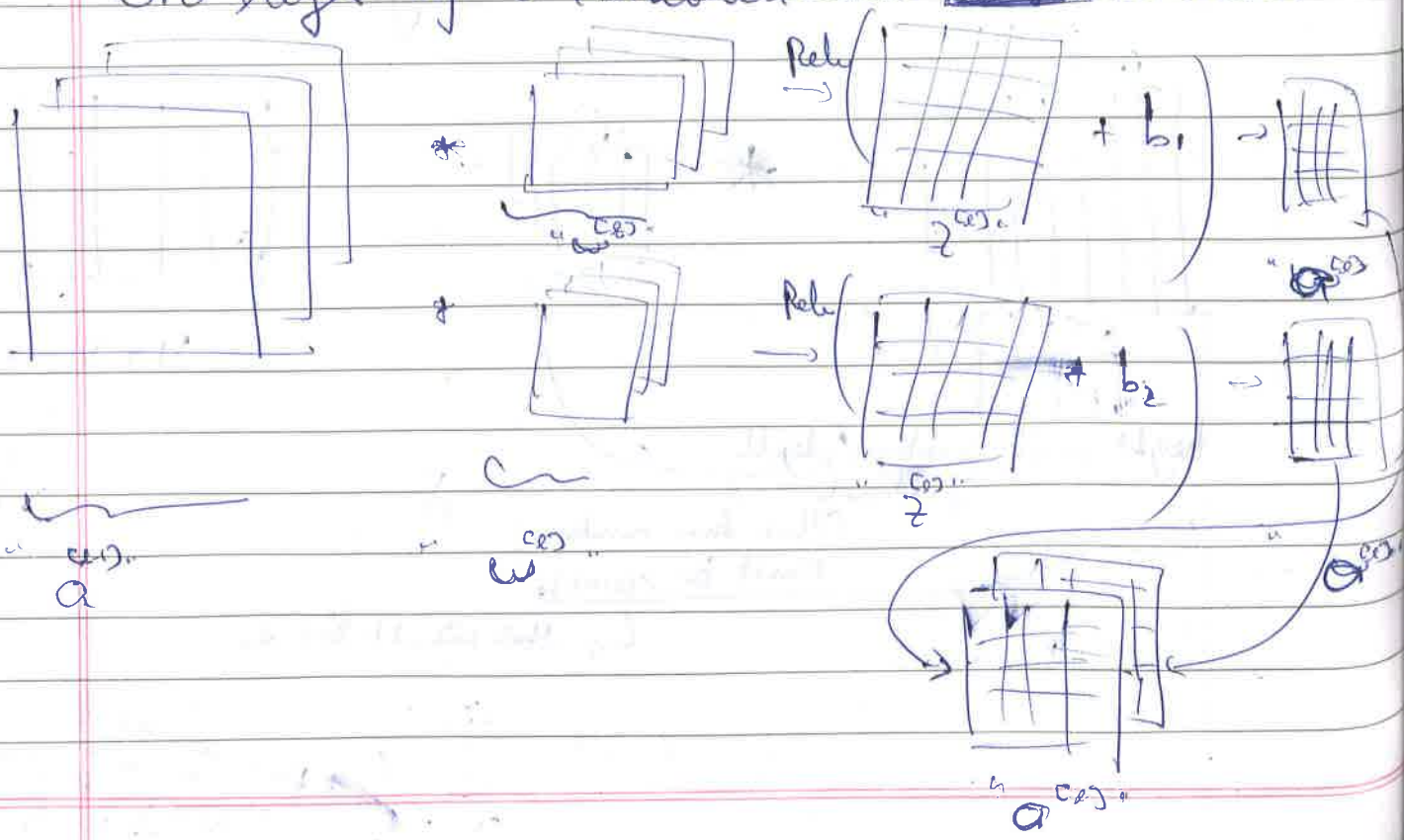


Summary

$(n \times n \times n_c)$ (number of channels) $* (f \times f \times n_c) \rightarrow (n-f+1) \times (n-f+1) \times n_c'$

n_c' : number of filters

→ One layer of a convolutional network.



~~So what the total number of parameters in one layer are:~~

So the total number of parameters when we have 10 filters are:-

$10 \times (3 \cdot (F \cdot F) + 1) \Rightarrow 30F^2 + 10$

let $F = 3$

thus $\Rightarrow 30(9) + 10 = 280$

So now the total number of parameters in on layer is 280 even if we have an image that has a lot of pixels

$\Rightarrow$ Summary of notation:-

If l is a convolution layer:-

$f^{[l]}$ = filter size

$p^{[l]}$ = padding

$s^{[l]}$ = stride

$n_c^{[l]}$ = number of filters

(Input:- $n_H^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$; height, width)

(output:- $n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$)

(each filter is:- $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$)

(Activations:- $a^{[l]} \rightarrow n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$)

($A^{[l]} \rightarrow m \times n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$)

$$n_H^{[l]} = \left[\frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right]$$

$$n_w^{[l]} = \left[\frac{n_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right]$$

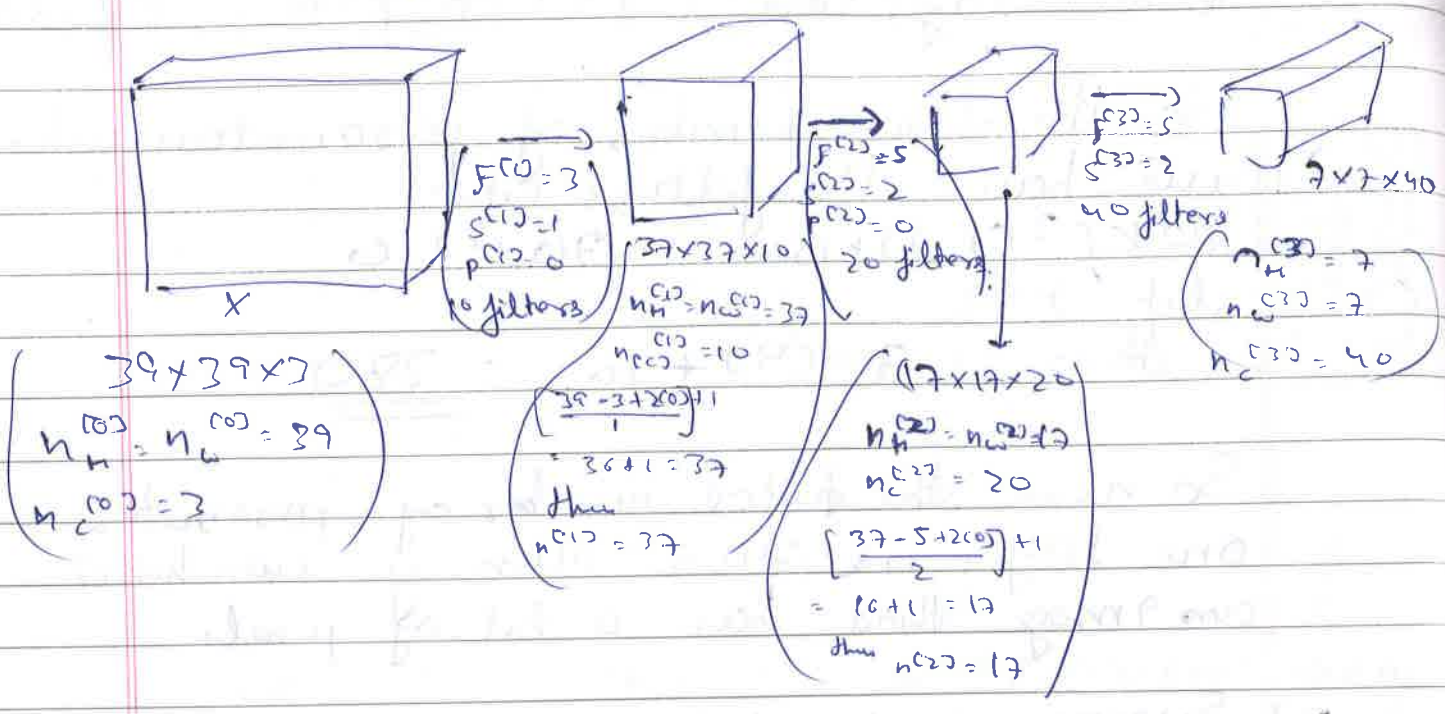
weights: $n_c^{[l]} \times f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

($n_c^{[l]}$: number of filters; $n_c^{[l-1]}$: no. of channels in the previous layer)

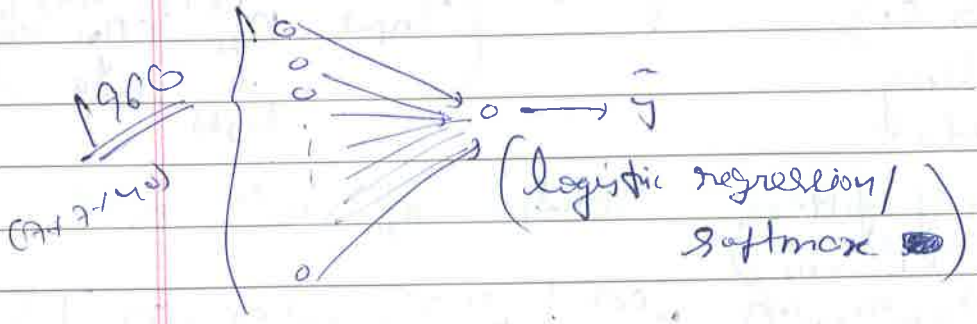
bias: $n_c^{[l]}$ it would be more convenient if we write it as: $(1, 1, 1, n_c^{[l]})$

$\rightarrow$ b, the constant which is added.

⇒ Example of convolution network

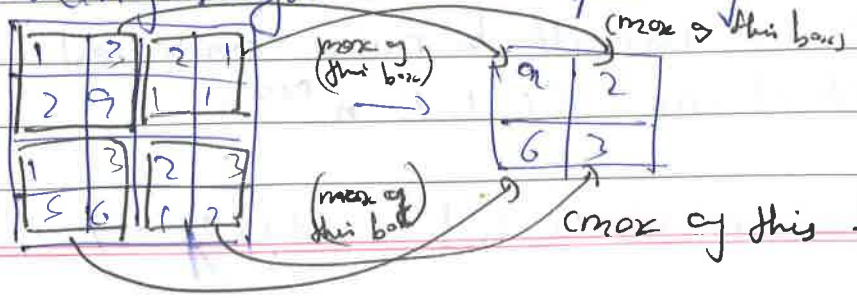


now total units we have are:- $7 \times 7 \times 40 = 1960$



⇒ types of layers in a convolutional network
- Convolution (CONV)
- Pooling (POOL)
- Fully connected (FC)

⇒ Pooling layer : max pooling



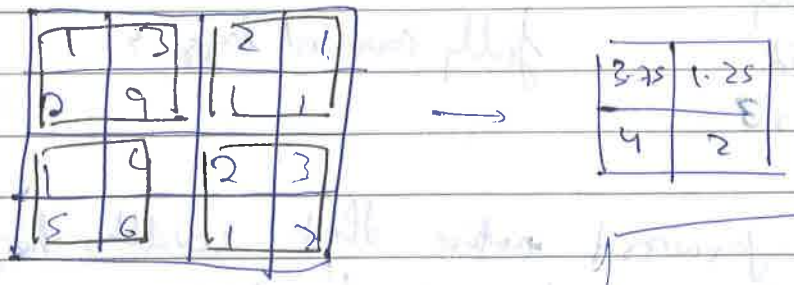
You can use pooling layer to reduce n_H, n_W but not n_C

hyper parameters:- $f = 2$ (boxes are 2*2)
$s = 2$ (stride is 2)

There are no parameters to learn in max pooling. Thus no gradient descent would be applied on any parameters of max pooling as there are none.

The intuition behind max pooling is that we would be able to detect in which quadrant or in which part of the image we have a particular feature. For example an eye would have very high pixels. So in the ~~follow~~ left above example we had 9 in the upper ~~right~~ quadrant, thus we may infer ~~that~~ that we have a cat's eye in the upper left quadrant.

⇒ average pooling
Here instead of taking the max we take the average of all the numbers in that box.

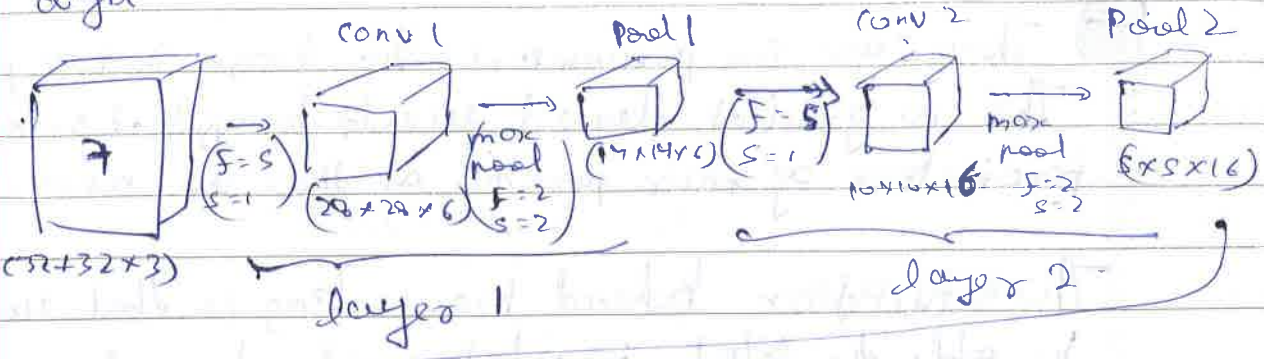


when there is no padding:-
$n_H = \left[\frac{n_{H(prev)} - f}{stride}\right] + 1$
$n_W = \left[\frac{n_{W(prev)} - f}{stride}\right] + 1$
$n_C = n_{C\,prev.}$
when there is padding add p as usual

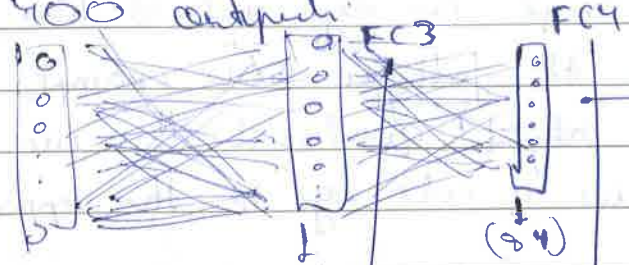
⇒ Hyperparameters:-
f: filter size
s: stride
max or average pooling
Usually we don't do padding here
Input:- $n_H \times n_W \times n_C$
Output:- $\left[\frac{n_H - f}{s} + 1\right] \times \left[\frac{n_W - f}{s} + 1\right] \times n_C$

max pooling also performs as a noise suppressant. It discards the noisy activations altogether and performs de-noising along with dimensionality reduction on the other hand average pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that max pooling performs

→ CNN example. Let build a neural network to identify a digit



Now we would flatten our output to 400 output



fully connected layer 3

fully connected layer 4

As we move forward notice that width, height would decrease & number of channels increases.

→ Convolution helps us in parameter sharing:
A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.

- Max pooling better than average pooling.

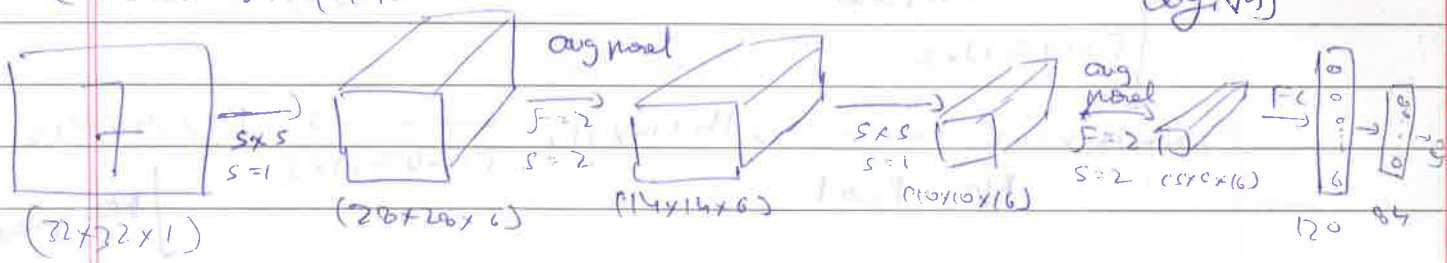
=> Sparsity of connections:- In each layer, each output value depends only on a small number of inputs

=> Convolution neural network are less prone to regularization.

=> CNN are immune to translation invariance. This means even if some pixels are shifted then also it would give good results.

Week 2 Specialization 4 (Case studies)

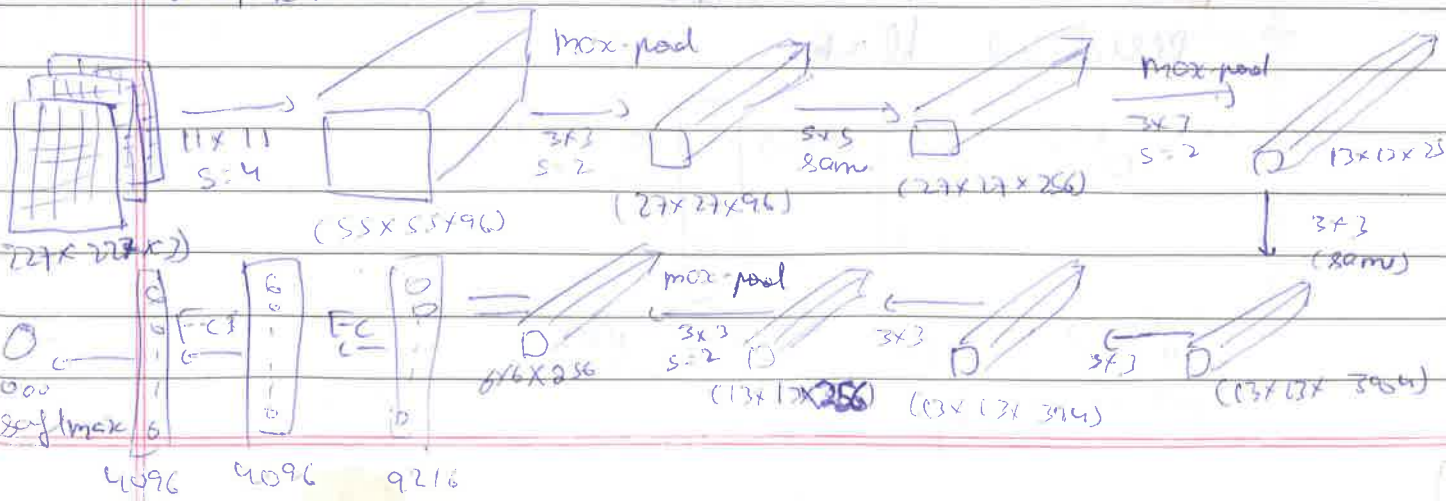
→ LeNet-5 (used to recognize handwritten digits)
(made in 1990's)



This has around 60k parameters, but networks used today have millions of parameters.

Now as we move forward $n_H, n_W \downarrow$ & $n_c \uparrow$

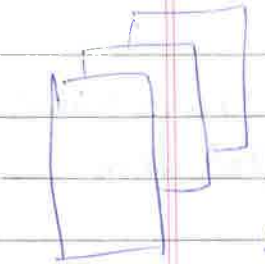
→ AlexNet



this is much bigger then Le Net-5, it has around 60 million parameters.

Relu activation function was used here.

→ VGG-16 (here Conv means 3×3 filter, s:1, padding=1 and Max-pool = 2×2, s=2)



224×224×3

[Conv 64] ×2 → 2 layers of Conv 64

224×224×64 → Max-pool → 112×112×64

[Conv 128] ×2

112×112×128 → Max-pool → 56×56×128

[Conv 256] ×3

56×56×256 → Max-pool → 28×28×256

[Conv 512] ×3

28×28×512 → Max-Pool → 14×14×512

[Conv 512] ×3

14×14×512 → Max-pool → 7×7×512

FC 4096 → FC 4096 → FC 1000 → Softmax

This has around 138 million parameters.

→ Residual Networks (Res Nets)

⇒ Residual block

$a^{[l]} \rightarrow a^{[l+1]} \rightarrow a^{[l+2]}$

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

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

Resnets prevent exploding or vanishing gradients

PAGE NO.
DATE / /

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

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

$a^{[l]} \rightarrow$ linear $\rightarrow$ Relu $\xrightarrow{a^{[l+1]}}$ linear $\rightarrow$ Relu $\rightarrow a^{[l+2]}$

this is the main path.

Now in residual block we will do:-

"shoot cut" / skip connection

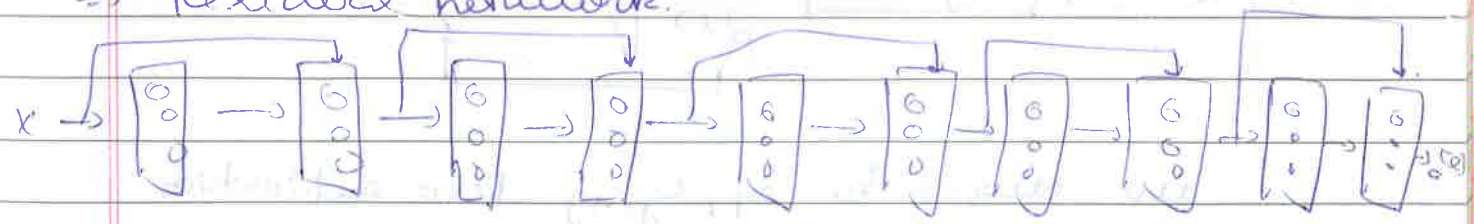
$a^{[l]} \rightarrow$ linear $\rightarrow$ Relu $\xrightarrow{a^{[l+1]}}$ linear $\rightarrow$ Relu $\rightarrow a^{[l+2]}$

"main path"

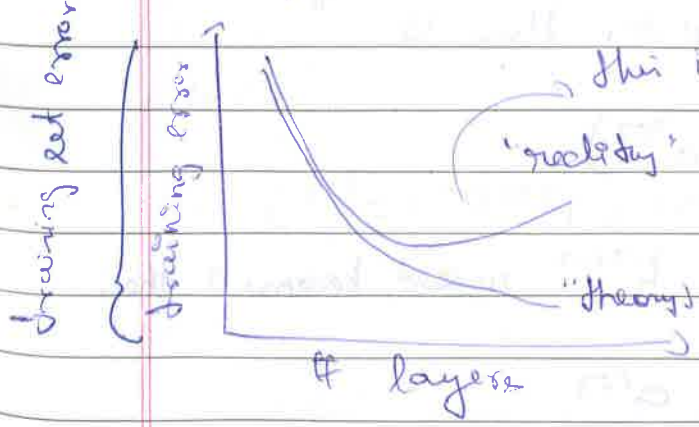
then $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

Using residual blocks help us to make much deeper neural networks

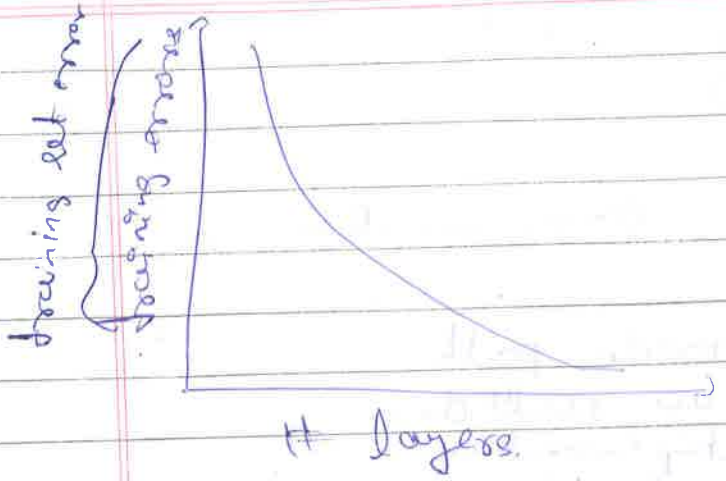
=> Residual network.



In plain neural network

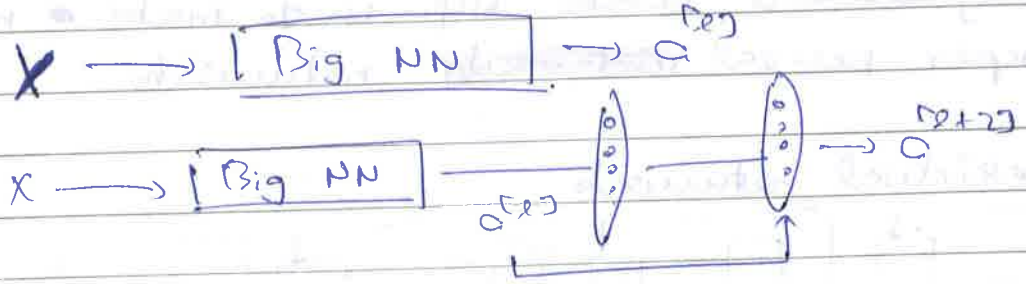


→ In residual networks.



Thus if we use residual networks then on increasing the ~~number~~ number of layers, the error keeps decreasing.

→ Why ResNet work



we would be applying Relu activation function thus $a \geq 0$

b a would always be greater than 0.

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$
$$= g(w^{[l+2]} a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

now if $w^{[l+2]}$ & $b^{[l+2]}$ would become 0 then we would have

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

this is because $a^{[l]}$ is +ve & Relu on a +ve number is that number itself.

as we do
$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

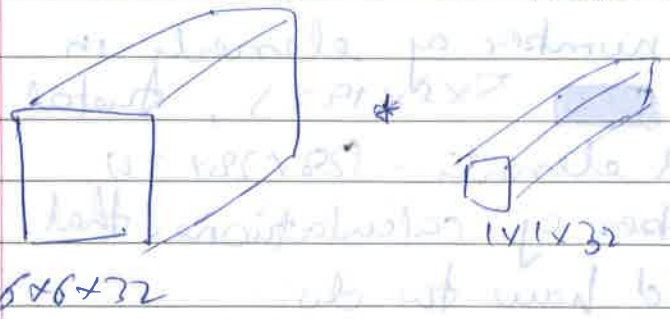
↳ thus both should have same dimensions.

Hence many times for resNets we use "same" convolution.
If they are not of same dimensions then we do:
$Wa^{[l]}$ where W is a matrix such that $W a^{[l]}$ dimensions would become equal to $z^{[l+2]}$

→ Networks in networks and 1×1 convolutions

⇒ [2 | 7; 3 | 4] * [2] = [4 | 14; 6 | 8]

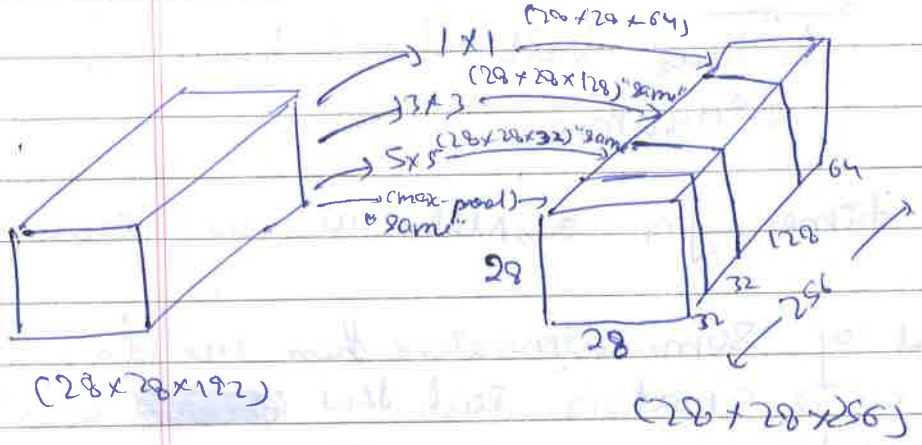
now it is not making much sense here (in 2-D) because we are just multiplying each and every element by 2.
But it will make sense in 3-D like this:-



This idea of 1×1 convolution is called network in network.

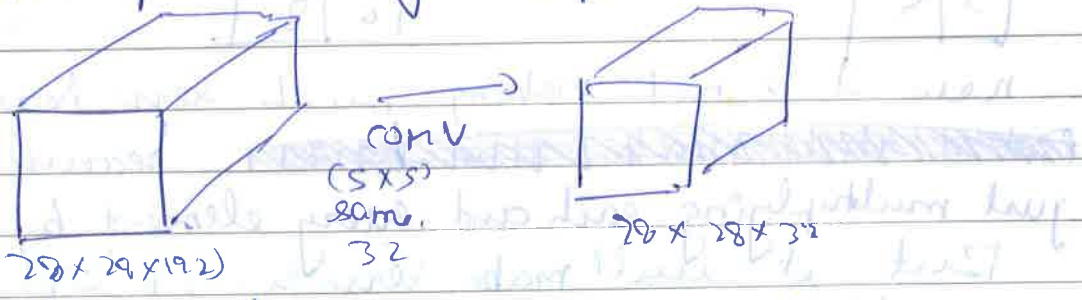
⇒ Using 1×1 convolutions.
If you want to convert 28×28×192 to 28×28×32 then you can use 32 (1×1) filters.

→ Inception network motivation.



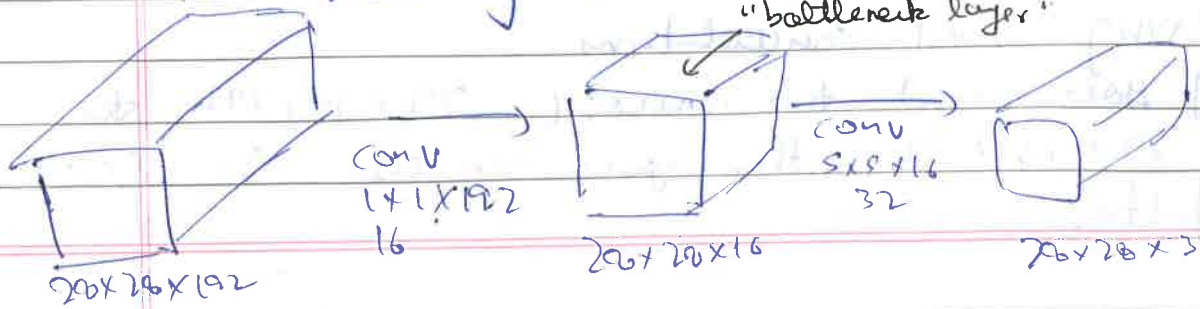
Here we are using different types of filters and then concating them.

→ The problem of computational cost.



Here total number of elements in one filter :- (5×5×192), total number of output elements :- (28×28×32)
Thus total number of calculations that computer would have to do:
28×28×32×5×5×192 ≈ 120 million

But now if we use 1×1 convolution.



So here the total number of computations would be:-

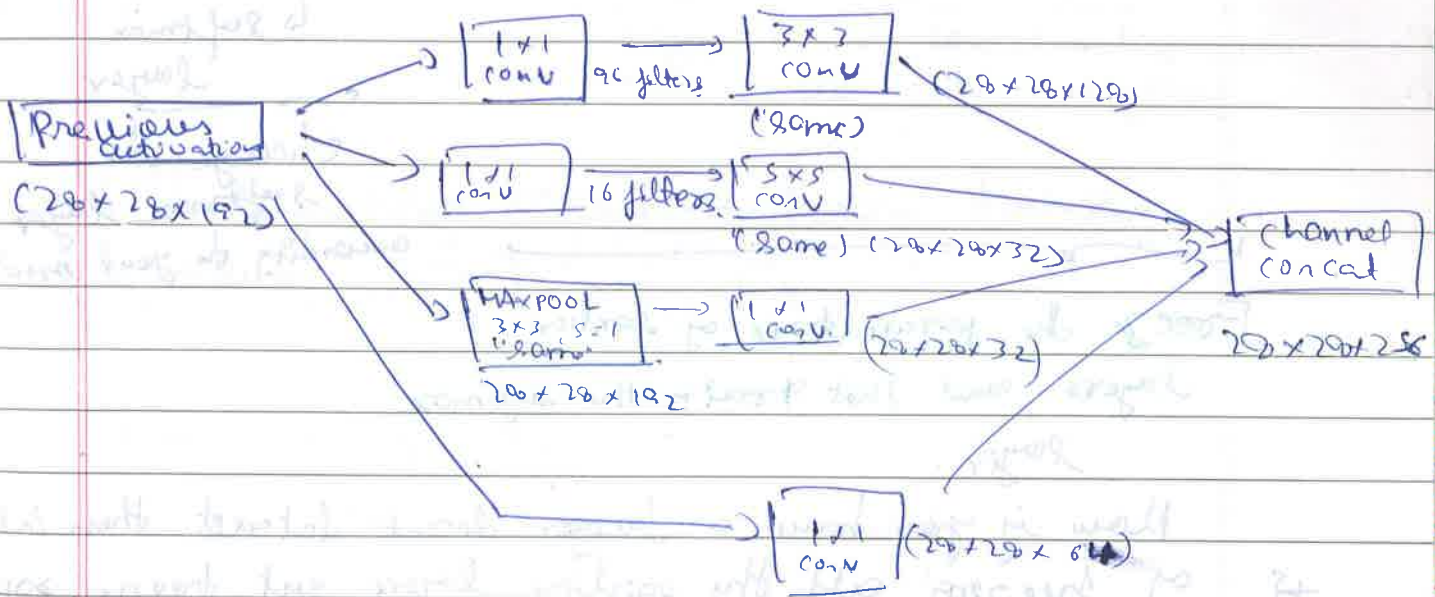
$(28 \times 28 \times 16 \times 192) + (28 \times 28 \times 32 \times 5 \times 5 \times 16)$

2.4 million ← 10.0 million

12.4 million

Thus the computation cost is reduced by around 10 times

→ Inception network



This is an inception block

→ Open source implementation of ConvNet

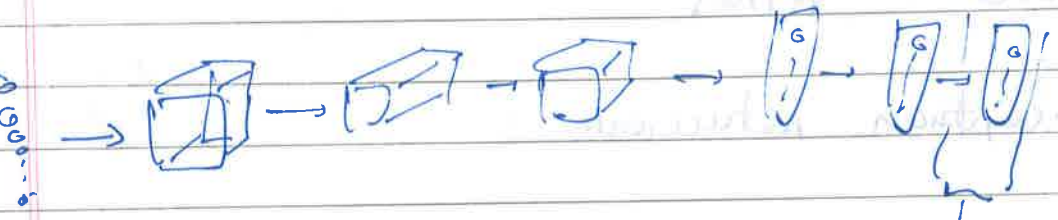
You could use ResNets already trained available on GitHub.

→ Transfer learning

Transfer learning is something that you must almost always do.

Let's say that you want to build a network that would classify a cat as → Tigger (Cat 1) and Misty (Cat 2) and Neither. and you don't have a very large dataset.

So you could use an open source trained neural network :



Change the softmax layer according to your model

Freeze the parameters of earlier layers and just train the softmax layer.

Now if you have a ~~large~~ decent dataset then instead of freezing all the earlier layers just freeze some of them & train the parameters of remaining layers along with the softmax layers. Or you could add your own layers instead of using some of the layers of open source network.

If you have large enough dataset then you can train the entire network on your dataset and not freeze any of the layers.

If training weights

7LpQ8UgR
letter

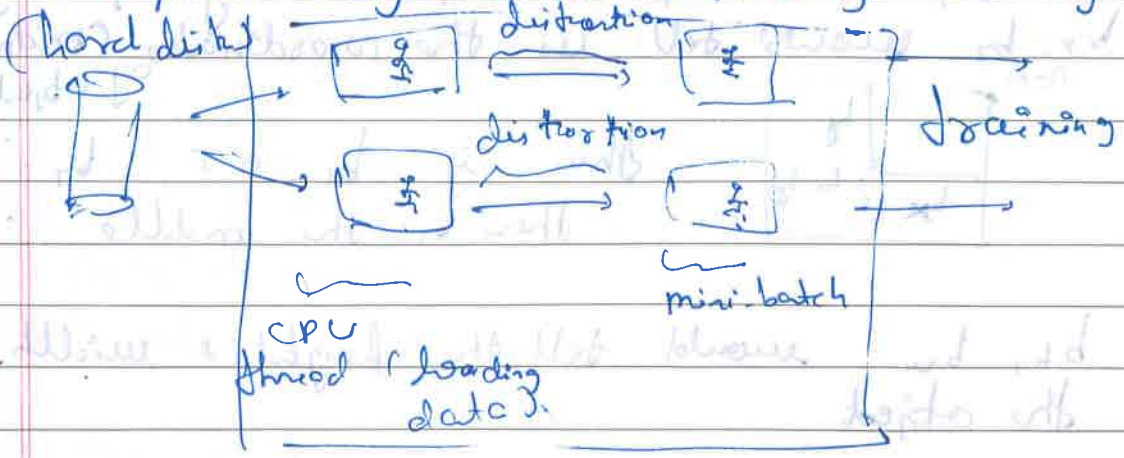
→ Data Augmentation.

(i) mirroring (using mirror image of object)
(ii) Random cropping
(iii) Rotation
(iv) Shearing
(v) local wraping
(vi) Color shifting (adding some distortion to the RGB values).

Eg. +20 -20 +20
R G B → this would make the image more purple.

To immune the model to change in illumination we could use PCA colour augmentation.
↳ you could search online when need to use color shifting.

(vii) Implementing distortions during training.

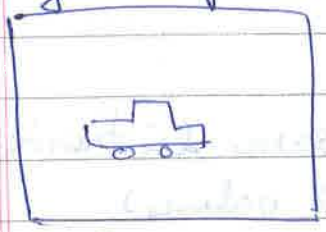


→ Ensembling.
) Train several networks independently and average their outputs
→ Multi-crop at test time
) Run classifier on multiple versions of test images & average results.
→ This is used in competitions but not in production systems for real use.

specialization 4 week 3

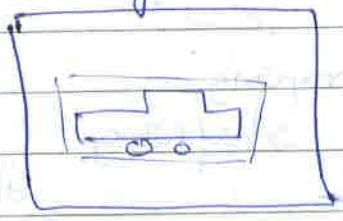
→ Object localization

(Image classification)



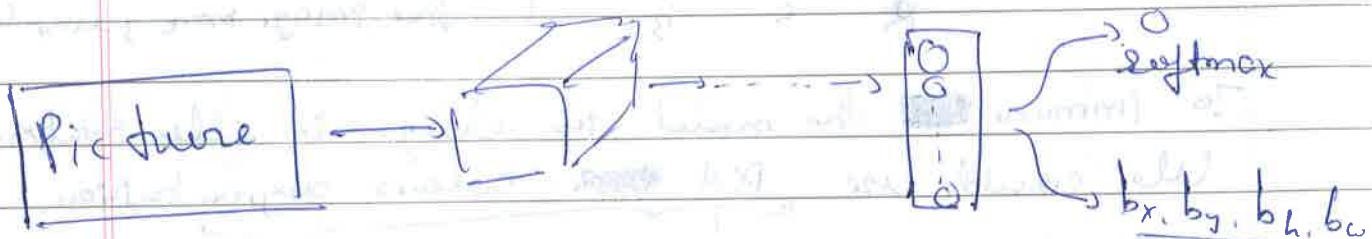
"Car"

((lassification and localization)



"Car"

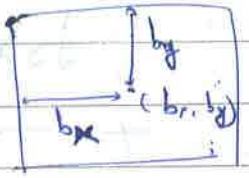
(then we draw a boundary around the object also)



=> Here softmax layer would tell us the classification the object (car / pedestrian / motorcycle / background)

=> b_x, b_y would tell us the co-ordinates (midpoint of object)

(0,0)



then if $b_x = 0.5$, $b_y = 0.5$, then in the middle.

=> b_h, b_w would tell the height & width of the object.

=> $y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$ → this would tell weather there is an object or not (1 → there is an object, 0 → there is no object)

Eg. When there is a car:-

$y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ ? \end{bmatrix} \rightarrow \begin{matrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_8 \end{matrix}$

$L(\hat{y}, y) = (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \dots + (\hat{y}_8 - y_8)^2$

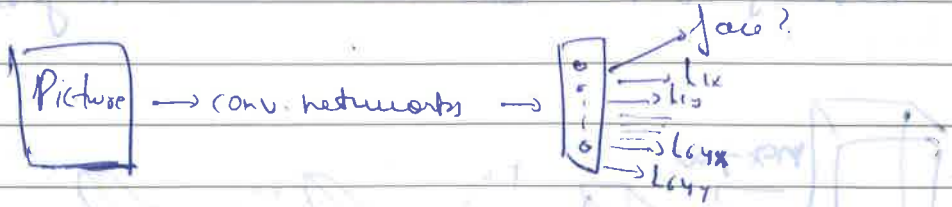
Eg. when there is no object:

$y = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix} \rightarrow \begin{matrix} y_1 \\ y_2 \\ | \\ | \\ | \\ | \\ y_8 \end{matrix}$

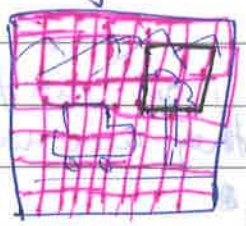
$L(\hat{y}, y) = (\hat{y}_1 - y_1)^2$

→ landmark detection.

For example we are making a face detection neural network. And for some reason we want to detect 64 points (landmarks) on the face. (Eg like the edges of face, edges of eyes, etc).



→ Object detection.



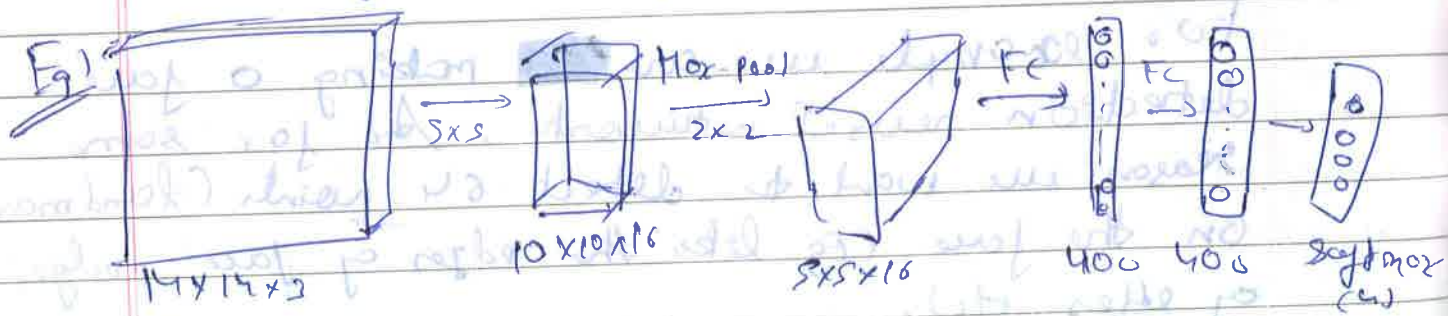
Suppose we have to detect weather there is a car and also show the boundaries of the car then we could take a neural network that given a cropped image of car tell weather it is a car or not.

We crop out different part of the images one by one with a particular stride. And then pass into that neural network. Then it tells us that weather it is a car or not.

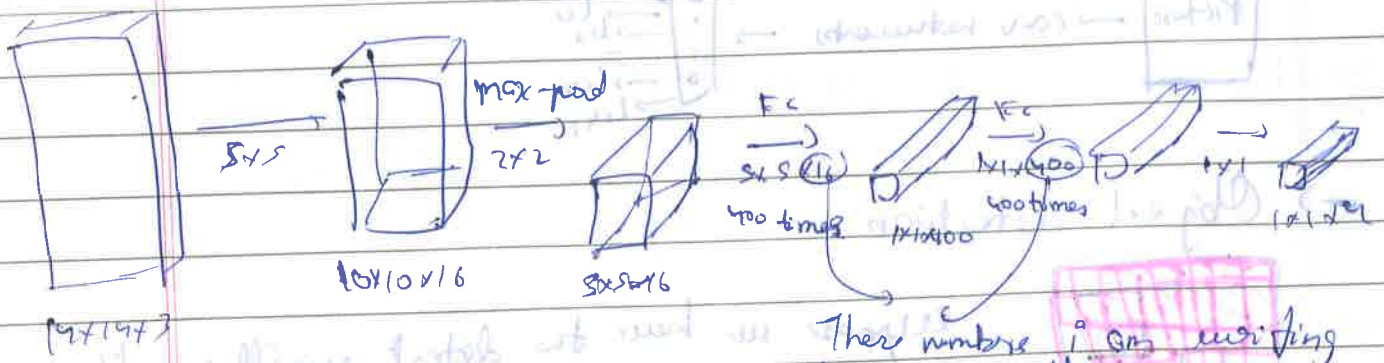
We can also change the size of the part that we were cropping out each time, like initially we were choosing squares of (2*2) only, now we will try squares of (3*3) only.

Assume the image is of (20*20) only.

→ Turning FC layers into convolutional layers.



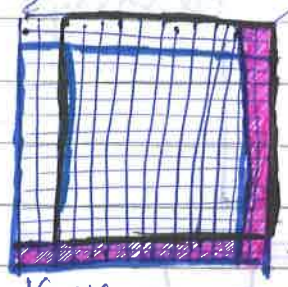
Now turning it into convnet layers.



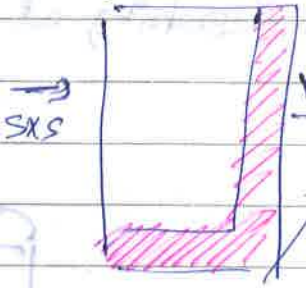
These numbers I am writing by myself, they were not in the video.

PAGE No.
DATE / /

Convolution implementation of sliding windows

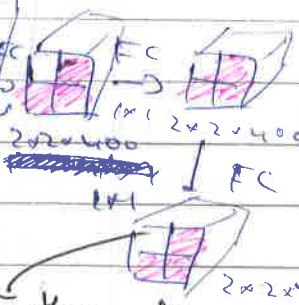


16×16×3



12×12×16

6×6×16



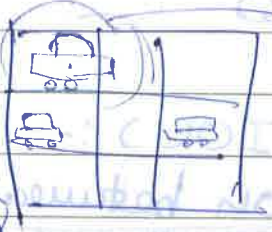
Here we have divided the image int. 4 windows

weather there is a car in the first part or not.

The would tell us weather there is a car in the four windows (for each).

Bounding box prediction

YOLO algorithm

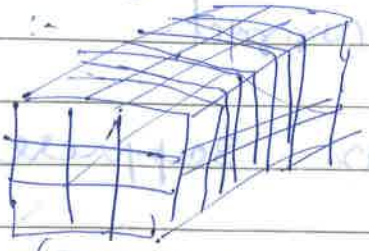


100×100 pixel picture
(we divid into 9 part(s).

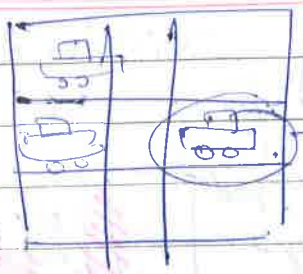
→ this object would be assigned to the first (0,0) part not to the second (0,1) part because the center of this object is in the first part

→ for one part : $\begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$

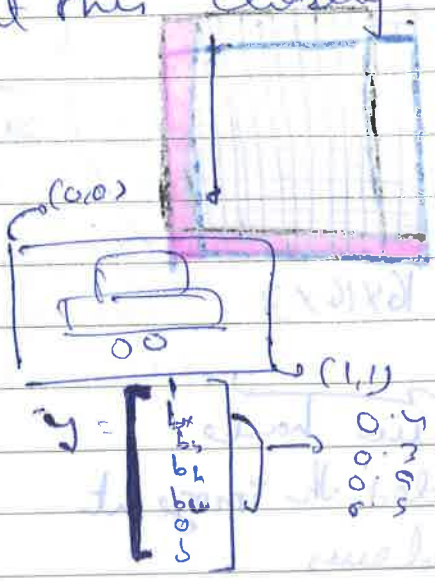
Then output :



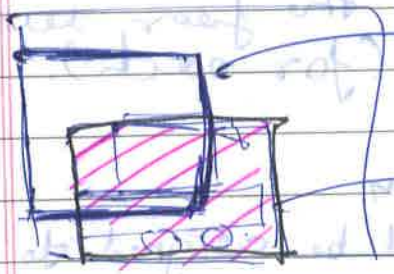
(3×3×8)



→ looking at this closely



→ Intersection over union



→ box 1 (th box when we detect)

→ box 2 (where car actually is)

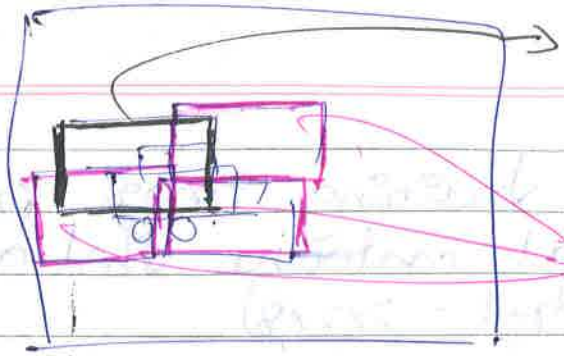
→ Intersection over union (IOU) :-

$$= \frac{\text{size of intersection between box1 \& box2}}{\text{size of union of box1 \& box2}}$$

many times we use IOU to evaluate object localization.

If IOU ≥ 0.5 then we usually consider correct.

→ Non-Max suppression.



→ this has maximum P_c. let say for this P_c is 0.8.

→ there remaining boxes which are very much overlapping with the selected black box would be discarded. (Here they are not being overlapped very much)

or we could do:-

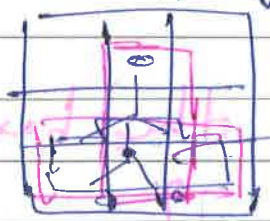
Discard all boxes with $P_c \leq 0.6$

while there are any remaining boxes:-

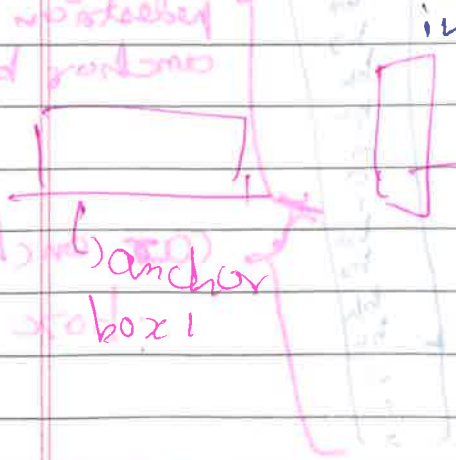
- Pick out the box with the largest P_c output as a prediction
- Discard any remaining box with IOU ≥ 0.5 with the box output in the previous step.

→ Anchor boxes.

⇒ overlapping objects.



→ Here the midpoint of both the pedestrian & the car have midpoint in this box.

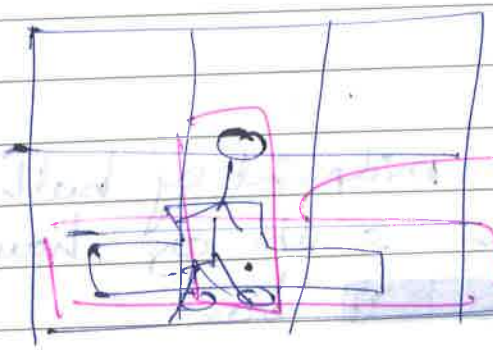


previously :-
Each object in training image is assigned to grid cell that contains that object's midpoint. (output : 3×3×8)

with two anchor boxes :-
Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IOU. (output : 3×3×16)

y = [p_c b_x b_y b_h b_w c_1 c_2 c_3] → Anchor box 1
[p_c b_x ... c_3] → Anchor box 2

then for this image :



→ for the box :-

y = [1 b_x b_y b_h b_w 0 0 1] pedestrian anchor box
[b_x b_y b_h b_w 0 1 0] car anchor box

PAGE No.
DATE / /

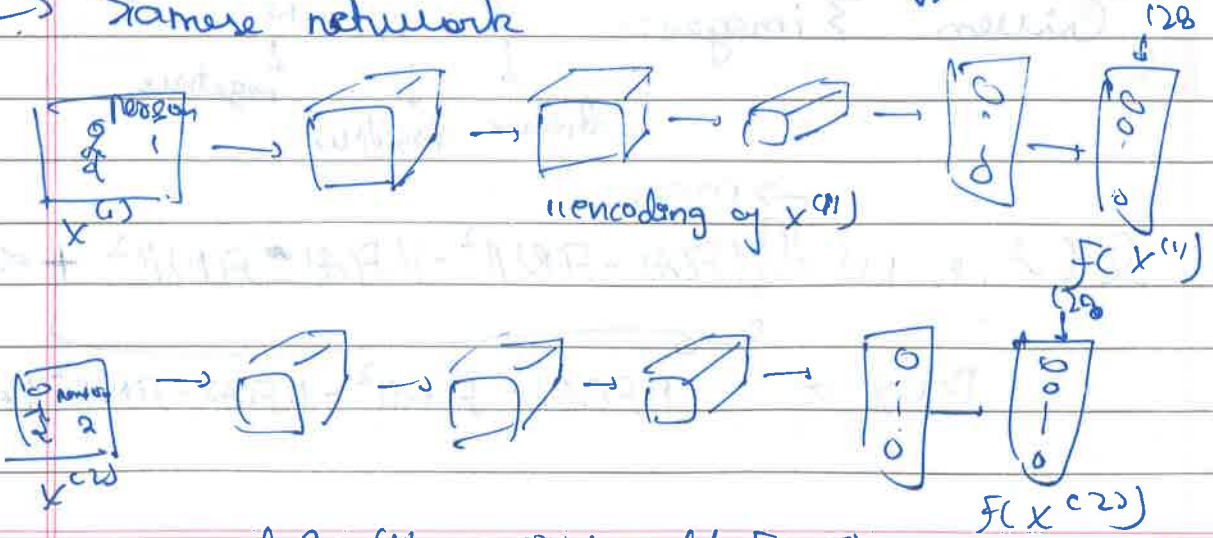
Specialization 4. week 4.

→ Verification
- Input image, name / ID
- Output whether the input image is that of the claimed person.

→ Recognition
- Has a database of k persons.
- Get an input image.
- Output ID if the image is any of the k persons (or "not recognized")

→ One shot learning
Many time we have just one or two images of a particular person. And we have to learn our model to recognize that person with just that image.
So we use a "similarity" function.
d(img1, img2) = degree of difference between images.
If $d(img1, img2) \leq \tau$ "same"
$> \tau$ "different"
} Verification

→ Siamese network



$$d(x^{(1)}, x^{(2)}) = \| f(x^{(1)}) - f(x^{(2)}) \|_2^2$$

Parameters of NN define an encoding $f(x^{(i)})$

learn parameters so that:

If $x^{(i)}, x^{(j)}$ are the same person,

$\|f(x^{(i)}) - f(x^{(j)})\|^2$ is small

If $x^{(i)}, x^{(j)}$ are different persons,

$\|f(x^{(i)}) - f(x^{(j)})\|^2$ is large

→ Triplet loss

What we want is:

$\|f(A) - f(P)\|^2 + \alpha \leq \|f(A) - f(N)\|^2$ ← margin

$d(A, P)$ — $d(A, N)$

Anchor image (initial image of a person) — positive image (image of the same person) — negative image (image of the another person)

$\Rightarrow \|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha \leq 0$

margin

$\Rightarrow$ loss function

Given 3 images: A, P, N

Anchor, Positive, negative.

$L(A, P, N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$

max of ($\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha$, 0)

$$J = \sum_{i=1}^{m} \ell(A^{(i)}, P^{(i)}, N^{(i)})$$

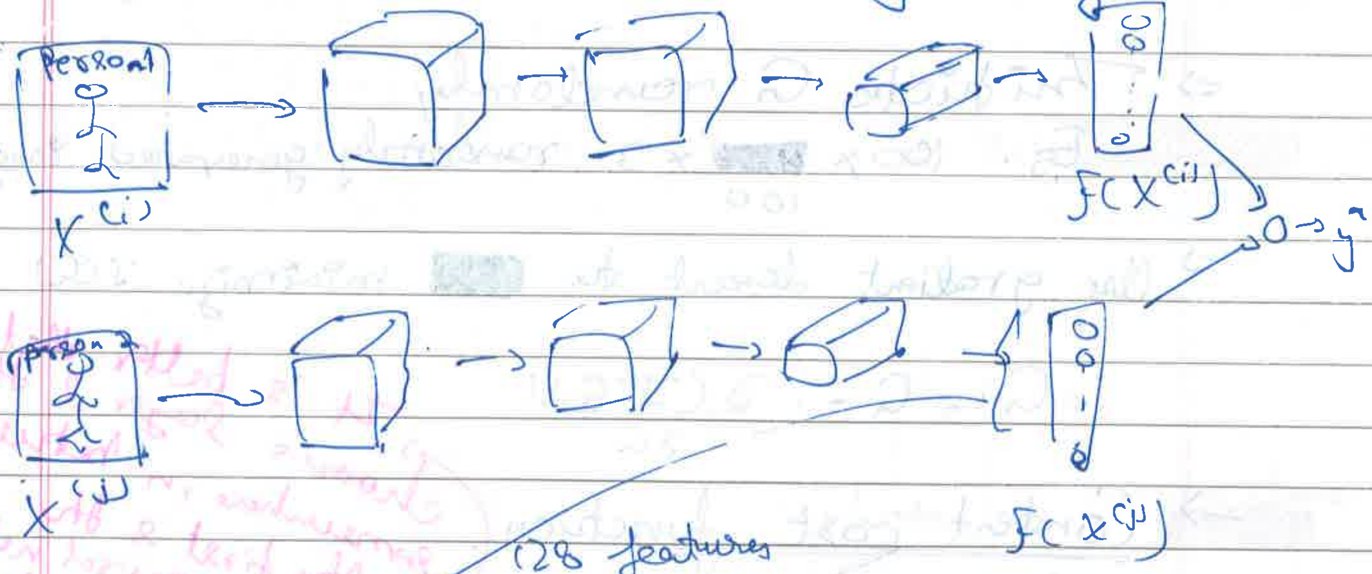
(cost function)

If you have training set of 10k [redacted] pictures of 1k people then we have to select triplets (A, P, N)

⇒ Now if we choose A, P, N randomly then $d(A,P) + \alpha \le d(A,N)$ is easily satisfied.

Then choose triplets that are "hard" to train on.

⇒ Face recognition and binary classification.



(Explaining this Siamese network architecture the learner & here, neural networks have different input images, but have exactly same parameters)

$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_k \left| f(x^{(i)})_k - f(x^{(j)})_k \right| + b\right)$$

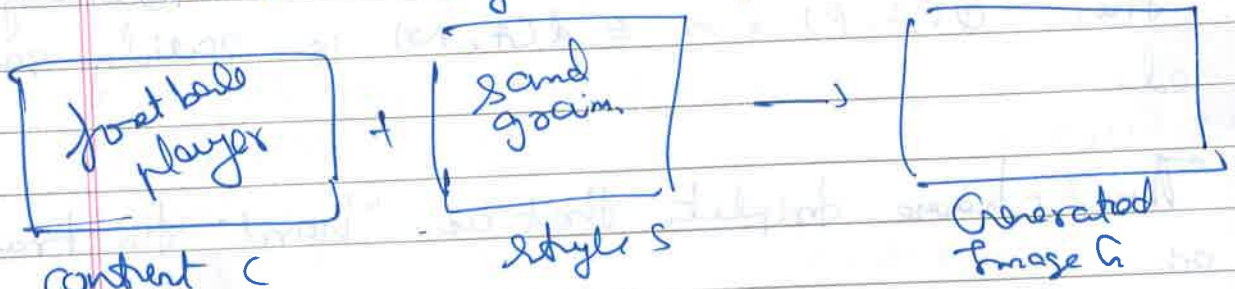
or we could use

$$\left\{ \frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k} \right\}$$

→ So here we are using a pre-trained ConvNet (ImageNet network) and then building on top of it. The idea of using a network trained on a different task and then applying it to a new task is called transfer learning which we are using here.

→ Neural style transfer
This adds style to content image. What ang may do does in editing is getting background of sand dark on a football player would give a granular texture.

→ Neural style transfer cost function



content C
style S

$$J(G) = \alpha \cdot J_{content}(C, G) + \beta \cdot J_{style}(S, G)$$

=> Initiate G randomly.
Eg. 100 × 100 × 3 randomly generated image.

=> Use gradient descent to minimize J(G)

$$G := G - \frac{\partial (J(G))}{\partial G}$$

→ Content cost function

$$J(G) = \alpha \cdot J_{content}(C, G) + \beta \cdot J_{style}(S, G)$$

- Say you use hidden layer l to compute content cost
- Use pre-trained ConvNet (Eg. VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the image.

It is better that you choose layer l that is somewhere in between the first & the last neural network layer.

edges and simple textures. The deeper layers tend to detect higher level features such as more complex textures and object classes. We want the generated image "G" to have similar content as input image C, then we should ~~st~~ choose a layer somewhere in middle.

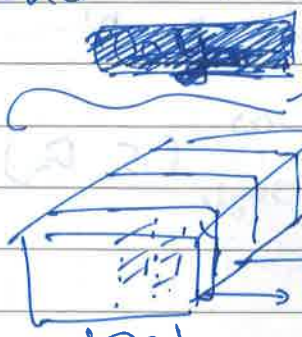
) If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content.

$$J_{content}(C, G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

$$J_{content}(C,G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{(all entries)}} (a^{(C)} - a^{(G)})^2$$

→ Style cost function.

Say that you are using layer l's activation to measure "style". Define style as correlation between activations across channels.



↳ how much are these units in the [redacted] channels are correlated to one another.

⇒ Style matrix

let $a^{[l]}_{i,j,k}$ = activation at (i, j, k).

i indicates height, j indicates width, k indicates channel

$G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{ijk} \, a^{[l](S)}_{ijk'}$$

for style image

→ for style image

In deeper layers of convnet, each channel corresponds to a feature detector. The style matrix $G^{[l]}$ measures the degree to which the activations of different feature detectors in layer l vary or correlate together with each other.

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} G^{[l](G)}_{ijk} \cdot G^{[l](G)}_{ijk'}$$

→ generated image → generated image.

This is "Gram matrix"

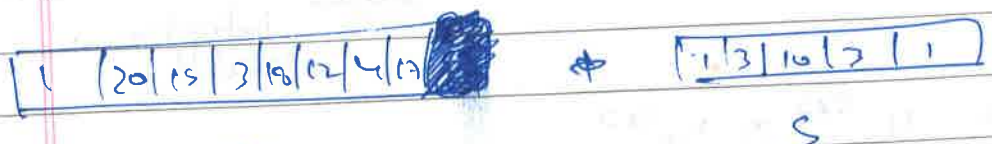
$$J^{[l]}_{style}(S, G) = \frac{1}{(2 n_H^{[l]} n_W^{[l]} n_C^{[l]})} \cdot \sum_{k} \sum_{k'} (G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'})$$

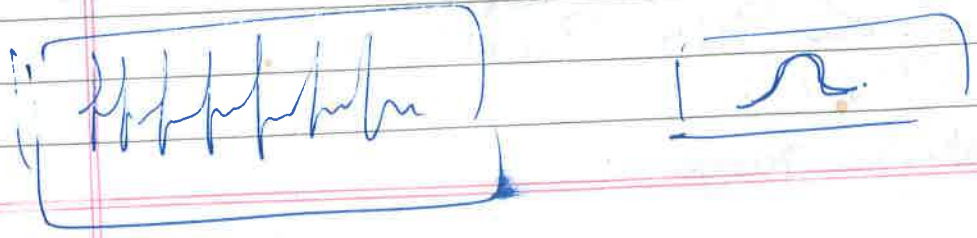
$$J_{style}(S, G) = \sum_{l} \lambda^{[l]} J^{[l]}_{style}(S, G)$$

$$J(G) = \alpha \cdot J_{content}(C, G) + \beta \cdot J_{style}(S, G)$$

→ 1D & 3D generalization.

⇒ for 1-D :-





using 16 different filters

$(8 \times 1) * (5 \times 1) = (4 \times 16)$

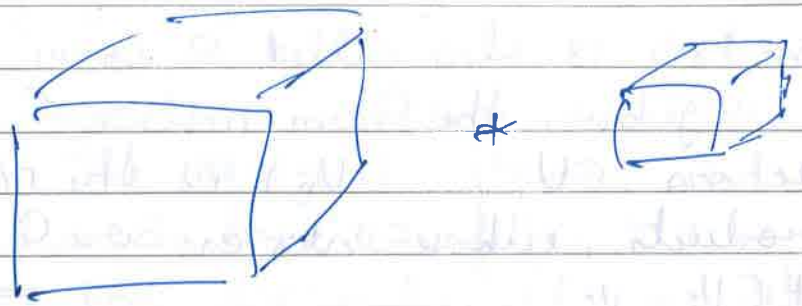
$*$

(5×16)

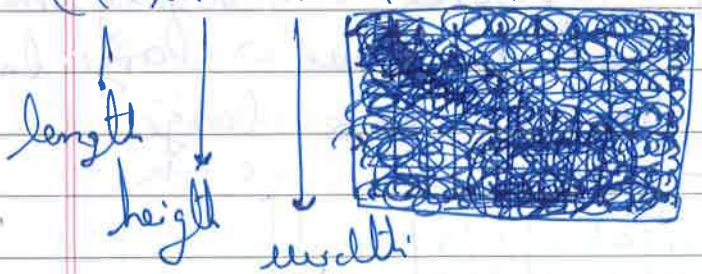
(8×16)

⇒ for 3-D

2.



$(14 \times 14 \times 14 \times 1) * (5 \times 5 \times 5 \times 16) = (10 \times 10 \times 10 \times 16)$



⇒ Understanding neural style transfer a little more

——→ eg. VGG network

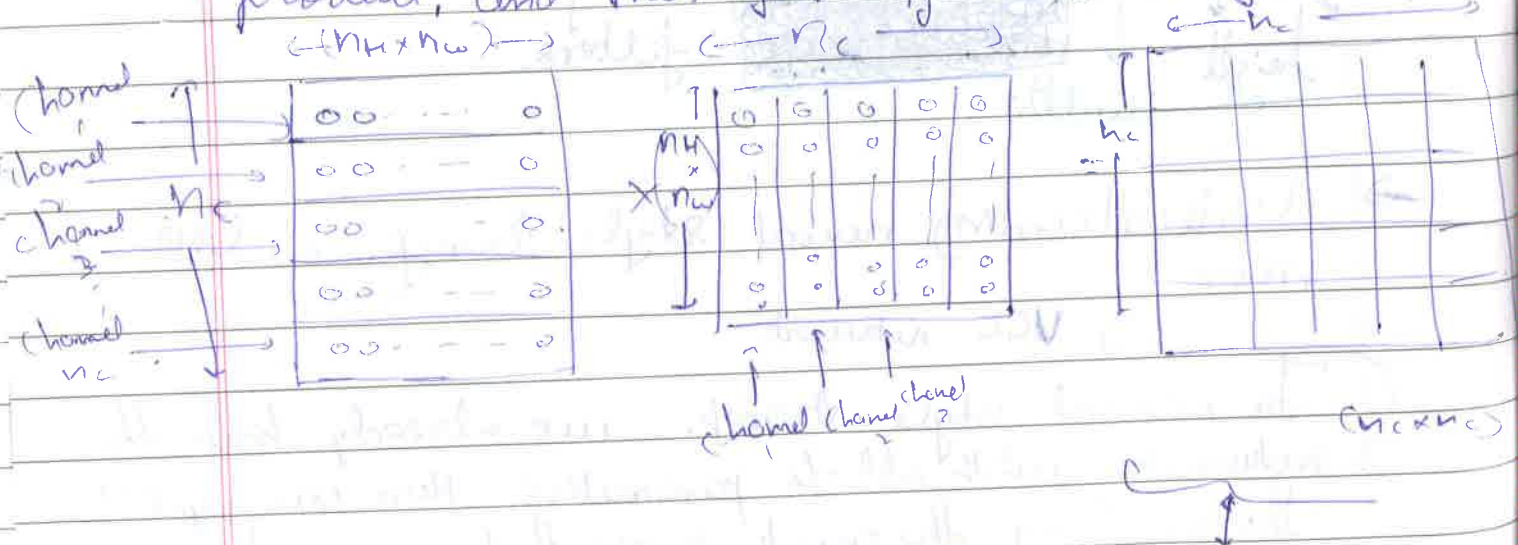
In neural style transfer we already have the network with all its parameters. Here we update the pixels of the input image that we put into the network rather than the parameters of that network. Earlier as we used to have hyperparameters that used to determine how the network parameters would be updated, how we have the hyperparameter that would determine how the

pixel values would be updated.

Content cost function ($J_{content}(C, G)$):-

$$J_{content}(C, G) = \left(\frac{1}{4 \times n_H \times n_w \times n_c}\right) \sum_{\text{all entries}} (a^{(C)} - a^{(G)})^2$$

⇒ (i) The style matrix is also called a "Gram matrix".
(ii) In linear algebra, the Gram matrix G of a set of vectors $(v_1, \ldots, v_n)$ is the matrix of dot products, whose entries are $G_{ij} = v_i^T v_j$.
$= np.dot(v_i, v_j)$.
(iii) In other words, G_{ij} compares how similar v_i is to v_j: If they are highly similar, you would expect them to have a large dot product, and thus for G_{ij} to be large.



So in this matrix we would have all combinations of $n_c^i \times n_c^j$ where n_c^i are all activations of a particular channel of a network and n_c^j are all activations of another particular channel of network G that

a = [-4, -3, -2, -1, 0, 1, 2, 3, 4]

np.linalg.norm(a) = 7.745966

$= \sqrt{(-4)^2 + (-3)^2 + (-2)^2 + (-1)^2 + (0)^2 + (1)^2 + (2)^2 + (3)^2 + (4)^2}$

$= 7.745966$

$G_{(gram)_{i,j}}$: correlation

The value $G_{(gram)_{i,j}}$ measures how similar the activations of filter i are to the activations of filter j.

⇒ $G_{(gram)_{i,j}}$:- prevalence of patterns or textures.

(i) The diagonal element $G_{(gram)_{i,i}}$ measures how "active" a filter i is.

(ii) For example, suppose filter i is detecting vertical textures in the image. Then $G_{(gram)_{i,i}}$ measures how common vertical textures are in the image as a whole.

(iii) If $G_{(gram)_{i,i}}$ is large, this means that the image has a lot of vertical textures

⇒ $$J^{[l]}_{style}(S, G) = \frac{1}{4 \cdot n_c^2 \cdot n_h^2 \cdot n_w^2} \sum_{i=1}^{n_c} \sum_{j=1}^{n_c} \left(G^{(S)}_{(gram)_{i,j}} - G^{(G)}_{(gram)_{i,j}}\right)^2$$

For a particular hidden layer l of the network.

$G^{(S)}_{gram}$:- Gram matrix of "style" image

$G^{(G)}_{gram}$:- Gram matrix of "generated" image.

⇒ $$J_{style}(S, G) = \sum_{l} \lambda^{[l]} \cdot J^{[l]}_{style}(S, G)$$

{p. 70}

$$J(G) = \alpha \cdot J_{content}(C, G) + \beta \cdot J_{style}(S, G)$$

→ Image recognition.

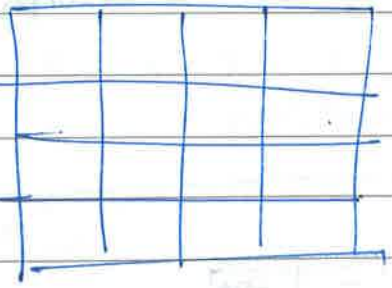
We repeat the image verification technique with all the k images (of employees). We cross check the image of the face of the employee entering with the first image. Compute that cost and assign that as min-cost. Now we cross check with 2nd image in the dataset. Compute cost, if the cost is less than min-cost then we would assign that cost to min-cost. We would repeat this for all the k pictures in database. Now if min-cost is less than the threshold value then we will allow that employee to pass.

→ Anchor boxes are chosen by exploring the training data to choose reasonable height/width ratios that represent the different classes.

→ Some more insight on anchor boxes :-

When using Anchor boxes, you can evaluate all object predictions at once. Anchor boxes eliminate the need to scan an image with a sliding window that computes a seperate prediction at every potential position. An object that uses anchor boxes can process an entire image at once, making real-time object detection systems possible.

The position of an anchor box is determined by mapping the location of the network output back to the input image. The process is replicated for every network output. The result produces a set of tiled anchor boxes across the entire image. Each anchor box represents a specific prediction of a class



(4 x 4)

Input image.

Let say that we define two anchor boxes. Then the output will have 2 anchor boxes per tile. Then we will have (2 x 4 x 4 = 32) anchor boxes in the output

before we eliminate them through IOU & probability.

If the center / midpoint of an object falls into a grid cell, that grid is responsible for detecting that object

→ Understanding Anchor boxes with YOLO.

⇒ Let assume that we have an image with dimensions :- (m, 608, 608, 3)

⇒ Anchor boxes are chosen exploring the training data to choose reasonable height / width ratios that represent the different classes

⇒ For this problem, we will choose 5 anchor boxes (to cover the 80 classes).

⇒ The YOLO architecture is :-

Image (m, 608, 608, 3) → Deep CNN → Encoding (m, 19, 19, 5, 85)

(examples) (dimension) (5 anchor boxes)

For each anchor box :-

| p_c | b_x | b_y | b_h | b_w | c_1 | c_2 | ... | c_{80} |

p_c → Probability that there is an object

c_1 → probability that the object is of class c_1

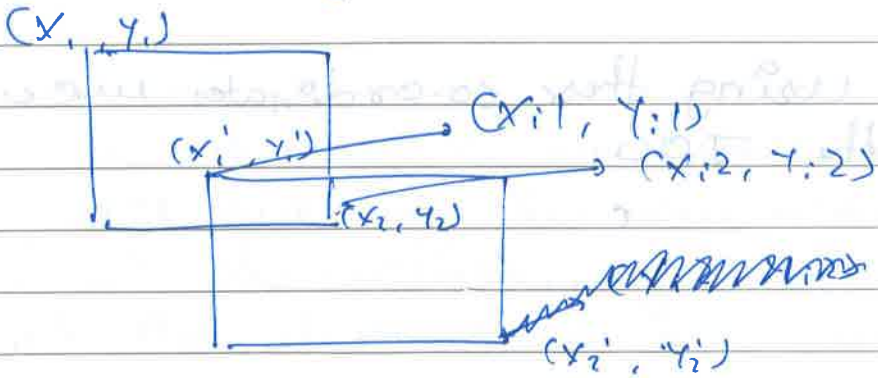
thus class scores :-

$$\text{score} := p_c * \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{80} \end{pmatrix} = \begin{pmatrix} p_c \cdot c_1 \\ p_c \cdot c_2 \\ p_c \cdot c_3 \\ \vdots \\ p_c \cdot c_{80} \end{pmatrix}$$

If we get $P_c \cdot C_{80} = 0.6$, then there is 60% probability that an object of class 80 exists, if we get $P_c \cdot C_{37} = 0.77$, then there is a probability of 77% that object of class 37 exists.

⇒ Then we will choose the object of maximum probability and we make a bounding box for that. Thus in total we will have (19 × 19 × 5 = 1805) anchor boxes and then we will eliminate many of them using non-max suppression.

→ Implementing IOU :-



Upper left corner of intersection box :-

x_{i1} : maximum of x_1 and x_2 ; most of x_1 and x_2 as you go down in the value of x

Upper left corner of intersection box :-

(i) x_{i1} :- rightmost of (x_1) & (x_1'), thus max of (x_1) and (x_1')

(ii) y_{i1} :- lowermost of (y_1) & (y_1'), then max of (y_1) & (y_1') { as we go down, the value of y increases }

bottom right corner of intersection box :-

(i) X_{i2} :- leftmost of (X_2 & X_2') ; thus the minimum of X_2 & X_2'.

(ii) Y_{i2} :- Uppermost of (Y_2 & Y_2') ; thus the minimum of Y_2 & Y_2'.

The height of the iteration box would be :-

$Y_{i2} - Y_{i1}$

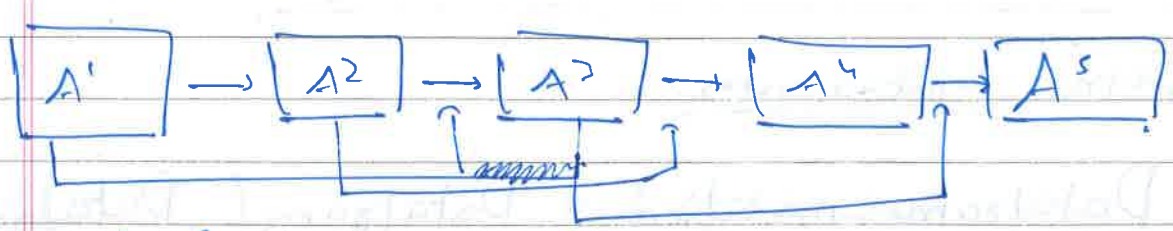
The length of the intersection box would be :-

$X_{i2} - X_{i1}$

If any of them is -Ve then there is no intersection.

Now using these co-ordinates we could calculate the IOU.

→ Resnets



$A^1 = g(w^1 A^0 + b^1)$

$A^2 = g(w^2 A^1 + b^2)$

$A^3 = g(w^3 A^2 + b^3) + A^1$

$A^4 = g(w^4 A^3 + b^4) + A^2$

$A^5 = g(w^5 A^4 + b^5) + A^3$

Assuming g is relu function

↳ $g(w^5 [g(w^4 A^3 + b^4) + A^2] + b^5) + g(w^3 A^2 + b^3) + A^1$

So using Resnet we would be able to expand in such way thus this would keep into account the activations of previous layers such that it would not lead to vanishing or exploding gradients.

Some notes from the API project.

⇒ Some syntax notes:-

(i) Dataframe_modified = Dataframe[Dataframe["parameter"] == value]
This would return us a dataframe with all the values of the selected parameter == value

(ii) df.drop(["col-1", "col-2", "col-3"], axis = 1)
This would drop col-1, col-2 & col-3 from the dataframe

(iii) df.reset_index(drop = True, inplace = True)

(iv) df.dropna(axis = 0, how = 'any', thresh = 2)

- if axis = 0, then it would drop rows and if axis = 1 then it would drop columns
- if any NA value is present it would drop if it were how = "all" then all the values should be NA for it to be dropped
- for example if I set thresh = 3 then that row/column would be dropped if there were atleast 2 NA values in it

df.dropna(subset = ["col-1", "col-2"], inplace = True)

This would state that we are to look into these columns when we would be looking for NA values to drop

→ df.set_index(list, inplace=True)

(v) df.isna() ; df["parameter"].isna()

(vi) df.notna() → opposite of isna()

(vii) df["parameter"].isna().sum()

↳ This would give the total number of samples that have missing value for that parameter

(viii) df[df.parameter == value].count()
This would return the number of samples that has that parameter = value

(ix) df.sort_values(by = ["param_1", "param_2"])

(x) df_shuffled = df.sample(len(df))
df_shuffled.reset_index(drop = True, inplace = True)
or df_shuffled = df.sample(frac = 1)

(xi) df.loc[3] ; df.iloc[3]

(xii) pd.crosstab(df["model"], df['doors'])

Door / model	3	4	5
BMW	0	0	1
Honda	0	3	2
Nissan	1	2	0

(xiii) df.groupby(["make"]).mean()

make	odometer (km)	doors
BMW	—	—
Honda	—	—
Nissan	—	—

(xiv) df["Price"] = df["price"].str.replace('[\$\,\.]', '').astype(int)

(xv) Suppose we are dealing with a classification problem with 2 results → (0 & 1). We need to find how many 1's & 0's are there in the output.
So we use :-
df["result"].value_counts()
df["result"].value_counts(normalize=True)
↳ This would give the distributions in percentage.

(xvi) Ctrl + S → Save
Ctrl + Z → Undo.

(xvii) plt.style.use("seaborn")
(xviii) car_names = pd.Series(["BMW", "WAGONR", "SUZUKI"])
car_colors = pd.Series(["Blue", "Black", "White"])
df = pd.DataFrame({"Car-names": car_names, "Car-colours": car_colours})
df["ready to use"] = True
df["Quantity"] = 4
df.rename(columns={"fruits": "my_fruits"}, inplace=True)
df["Car-names"] = df["Car_names"].str.lower()
df.columns = new_column_names → this is a good way to change the names of columns in dataframe
some basic operations on dataframe

or you could do df = pd.DataFrame(data={"Car-names": car_names, "Car-colour": car_colours})

·) Press (Shift + Tab) to get info about that syntax
·) press (Tab + →) to complete the word

DATE / /

(xxi) df["odometer"].fillna(df["odometer"].mean(), inplace=True)

(xxii) animals = pd.Series(["cat", "dog", "lion"], index=[0, 9, 8])

(xxiii) You can convert a 2-D numpy array to Pandas dataframe. Index & column names would take values of 0, 1, 2...

(xxiv) np.ones((2, 3)) → [[1,1,1],[1,1,1]]
np.zeros((1, 3)) → [[0,0,0]]

(xxv) np.random.randint(0, 10, (5, 3))
- 0 → inclusive
- 10 → exclusive
- (5, 3) → dimension of array returned

Or we could have written this :-
np.random.randint(0, 10, size=(5, 3))
np.random.random((5,3)) } → return an array of dimensions (5, 3) which has random numbers in it from 0 (inclusive) to 1 (exclusive)
np.random.rand(5,3)

(xxvi) np.unique(random_array) or random_array.unique()
→ this would return all the unique elements

(xxvii) np.square(a) ; np.sum(a)
np.sqrt(a) ; np.mod(a, 2) → a % 2
np.exp(a) ; np.log(a) ; np.min(a) ;
np.max(a) ; np.var(a) ; np.mean(a) ;
np.std(a)

%timeit → use this for python datasets

(xvi) %timeit sum(a)

%timeit np.sum(a)

↳ use this for numpy datasets

(xvii) np.sort(array)

np.argsort(array) → sort indexes

np.argmin(array)

np.argmax(,array)

np.argmax(array, axis=1)

↳ returns the index with maximum argument in rows.

np.argmin(array, axis=0)

returns the index with minimum argument in columns.

(xviii) df.aggregate("min")

this would return the minimum value of each column

df.aggregate("min", axis=1)

would return the minimum value of each row, by default axis=0

df.aggregate(["min", "max", "mean"])

this would return the min, max & mean values of all columns.

df.aggregate({"column-1": ["sum", "min"], "column-2": ["sum", "max"]})

(xxix) Suppose that we have a dataframe like:-

	A	B
0	4	9
1	4	9
2	4	9

then we do:- df.apply(np.sqrt)

so we get:-

	A	B
0	2.0	3.0
1	2.0	3.0
2	2.0	3.0

(xxx) df.apply(lambda x: [1, 2], axis=1)

0	[1, 2]
1	[1, 2]
2	[1, 2]

now:-

df.apply(lambda x: [1,2], axis=1, result_type='expand')

	0	1
0	1	2
1	1	2
2	1	2

(xxxi) df.replace(to_replace=0, value=df.mean(), inplace=True)

(xxxii) cols = df.columns.tolist()

cols = cols[-1:] + cols[:-1]

df = df[cols]

{p-7.0}

(~~xxii~~) np. where (condition, x, y)

parameters :-

(condition :-
where True yield x otherwise y)

(x :- the value which will be yielded if condition is true

(y :- the value which will be yielded if condition is false.

Eg1

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>> np.where(a<5, a, 10*a)
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

→ condition

→ if condition is true this will be yielded (x)

→ if condition is not true then this will be yielded (y)

If we just give the conditions then only those elements would be returned which fulfill that condition.

Eg2

```
>> np.where(a<5)
array([0, 1, 2, 3, 4])
```

(~~xxiv~~) Plotting :-

[p. 7. 0]

fig, (ax1, ax2, ax3, ax4, ax5)
= plt.subplots(nrows = 5, ncols = 1,
figsize = (16, 32))

ax1.plot(df.index, df.param-1, color = "red",
label = "Param1 Vs Index", marker
= "o")

ax1.plot(df.index, df.result, color = "blue",
label = "Param-2 Vs index", marker = "o")

[redacted]

ax1.set_xticks(ticks = np.arange(0, 40, 1))
ax1.set_yticks(ticks = np.arange(0, 1, 0.05))

~~ax1.fill_between(df.index, df.param-1, df.param-2, where = df.param-2 > df.param-1)~~

ax1.fill_between(df.index, df.param-1, df.param-2, where = (df.param-2 >= df.param-1), color = "red", interpolate = True, alpha = 0.25)

ax1.fill_between(df.index, df.param-1, df.param-2, where = (df.param-2 < df.param-1), color = "blue", interpolate = True, alpha = 0.25)

ax1.axhline(y = 0.5, color = "yellow", label = "yellow line", linewidth = 1.6, linestyle = "--")

ax1.axvline(x = 0.5, color = "black", label = "black line", linewidth = 1.6, linestyle = "-.")

ax1.legend()

ax1. set_title (" ..... ");

→ Iteratively appending rows to a dataframe can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

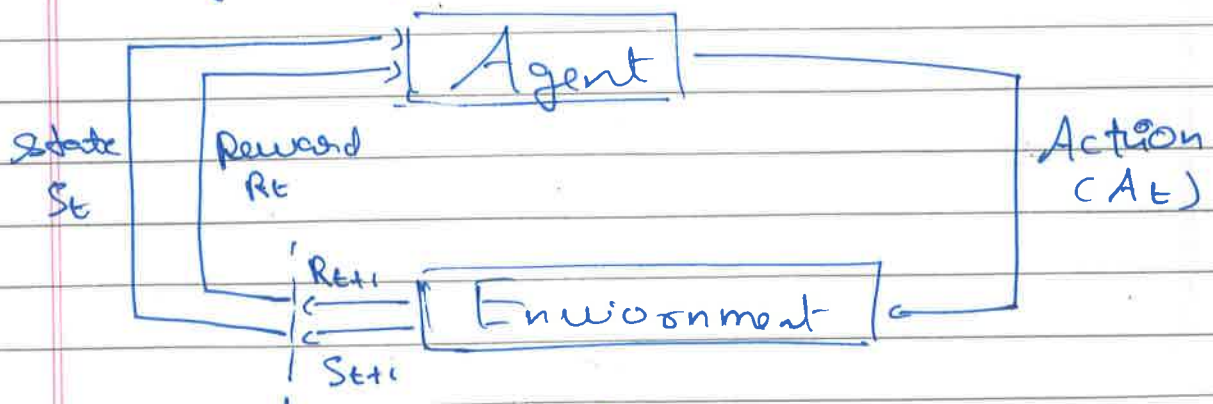
→ DataFrame.sample(frac = 0.5)

→

→ **Reinforcement learning.**

Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Reinforcement learning is one of the three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

Here we have a decision maker called an agent, that interacts with the environment it is placed in. These interactions occur sequentially over time. At each step, the agent will get some representation of the environment's state. Given this representation, the agent selects an action to take. The environment is then transitioned into a new state, and then agent is given a reward as a consequence of the previous action.



↳ end of cycle

→ break

let [redacted] down the diagram into steps:

(i) At time t, the environment is in state S_t

(ii) The agent observes the current state and selects action A_t.

(iii) The environment transitions to state S_{t+1} and grants the agent reward R_{t+1}.

(iv) The process then starts over for the next time step, $t+1$.

- Note, $t+1$ is no longer in the future but is now the present. When we cross the dotted line on the bottom left, the diagram shows $t+1$ transforming into the current time step t so that S_{t+1} are [redacted] now S_t and R_t

This looping process was MDP (Markov decision process)

R programming

PAGE No.
DATE / /

→ function_as_argument (func, a, b){
func (a, b) }

→ sentence ← function (...) {
args = list (...)
a = args [1]
b = args [2]
c = args [3]
paste (a, b, c, collapse = " ")
}

→ unclass (Sys. time ())
this would return the number of seconds passed since 01-01-1970

→ unclass (Sys. Date ())
number of days passed since 01-01-1970

→ unclass (as. Date (" 1969-01-01 "))
↳ -365

→ time is stored in two forms :-

- POSIXct (stored as number of seconds since 1970-01-01)
- POSIXlt (stored as a list of seconds, minutes, hours, etc)

→ str (unclass (as. POSIXlt (Sys. time ())

t ← as. POSIXlt (Sys. time ())

→ To get a specific attributes.
t$min

→ weekdays (t)
↳ would return todays DAY

→ months (t)
↳ would return todays month

→ quarters (t)
↳ would return the current quarter

→ t3 ← "October 17, 1986 08:24"
t4 ← strptime (t3, "%B %d, %Y %H:%M")
↳ There is space here

→ difftime (Sys.time(), t1, units = "days")

→ .head (data_frame)

→ dim (data_frame)
"would return the number of rows and column in data frame"

→ (ls_list ← lapply (dataframe, class)
↳ list of input
↳ function we want to apply on the list of inputs without parenthesis.

→ Creating Custom functions in R
Eg 1)

"%mult_add_one%" <- function(a, b) {
(a * b) + 1 }

% mult_add_one % 5 would return
21.

-> viewing() (this would bring up documentation about the dataset.) (Just try viewing() and nothing else).

-> vapply() is a safer and faster version for sapply. Faster because it does faster computation on larger datasets and safer because here we can explicitly mention in which format we expect the output. And if we do not get in that format then an error will be thrown.
Eg.
vapply(flags, unique, numeric(1))
Here this should show error because we expect to get numeric output of vector length 1 but instead we get all unique values as output. So it throws error.

but if we do :-
vapply(flags, class, character(1))
this wouldn't show any error.

{P.T.O}

Prodyum, Phemon, Arthur, Param, Jatin

PAGE NO.
DATE / /

→ We can use tapply() to split our data into groups based on value of some variable, and then apply some function to each group.

Eg.
tapply (flags$population, flags$red, mean)

0 1
55.2 76.4

→ On an average 55.2 million people live in countries without red flag and 76.4 million people live in countries with red flag.

Eg:
tapply (flags$population, flags$landmass, summary)

For each and every landmass the following will be returned:-

(min 1st Qu. Median Mean 3rd Qu. Max.)

→ When we apply :- lapply ( ) the results may not be very tidy so we apply sapply(), to simplify the results. lapply() gives output as a list while sapply simplifies the output to a vector, matrix or dataframe.

Eg) sapply (ingredients, ranges)

	dulsi	haldi	salt	sugar
min values	0	0	1	2
max values	4	5	7	9

→ class :- matrix

lapply (ingredients, ranges)

$dulsi
[1] 0 4

$haldi
[1] 0 5

$salt
[1] 1 7

$sugar
[1] 2 9

class :- list
dim :- Null

columns from 11 to 17

→ flag_colors ← flags[, 11:17]

= T, all rows

→ lapply(unique_vals, function(element) element^2)

Here we are applying a custom function on the data unique_vals which return the second element of the data. So if data is a list of vectors so this will return the second element of each vectors.

Recurrent neural networks.

Used in:- Speech recognition, music generation, sentiment classification, DNA sequence analysis, Machine translation, Video activity recognition, Name entity recognition.

Eg: Name recognition:

Harry Potter and Hermione Granger invented a new spell.

$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ $x^{<4>}$ $x^{<5>}$ $x^{<6>}$ $x^{<7>}$ $x^{<8>}$ $x^{<9>}$

→ y: 1 1 0 1 1 0 0 0 0

$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ $y^{<4>}$ ——— $y^{<9>}$

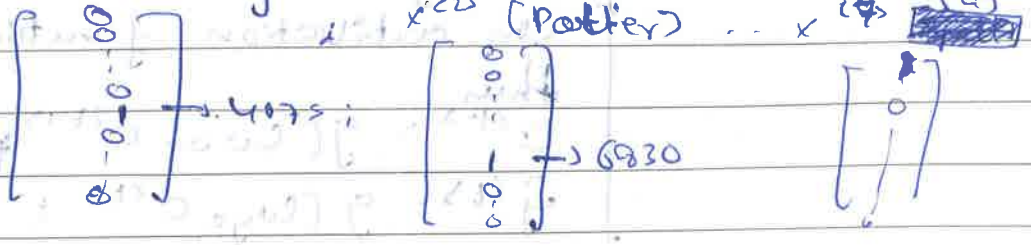
$T_x = 9$ (number of inputs)

$T_y = 9$ (number of outputs)

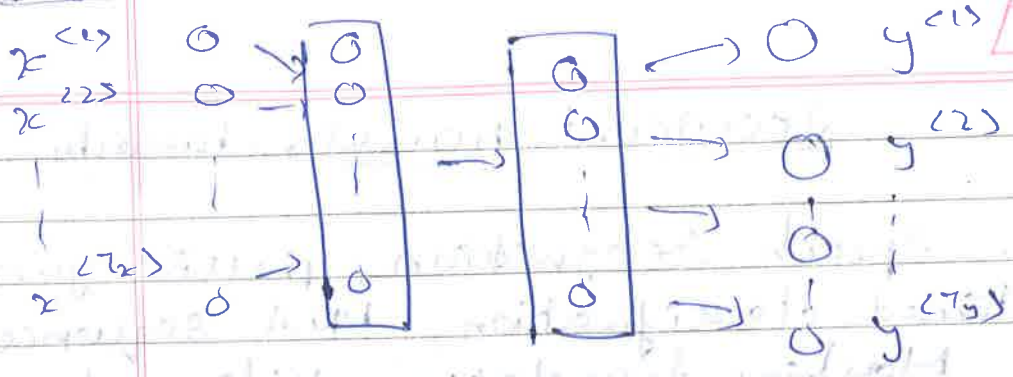
Vocabulary

a	1
aaron	2
:	:
and	367
:	:
Harry	4075
:	:
Potter	6830
:	:
Zulu	10,000

Then $x^{<1>}$ = Harry could be represented in one-hot-encoding as:- $x^{<2>}$ (Potter) ... $x^{<9>}$ (a)



Why not a standard network.

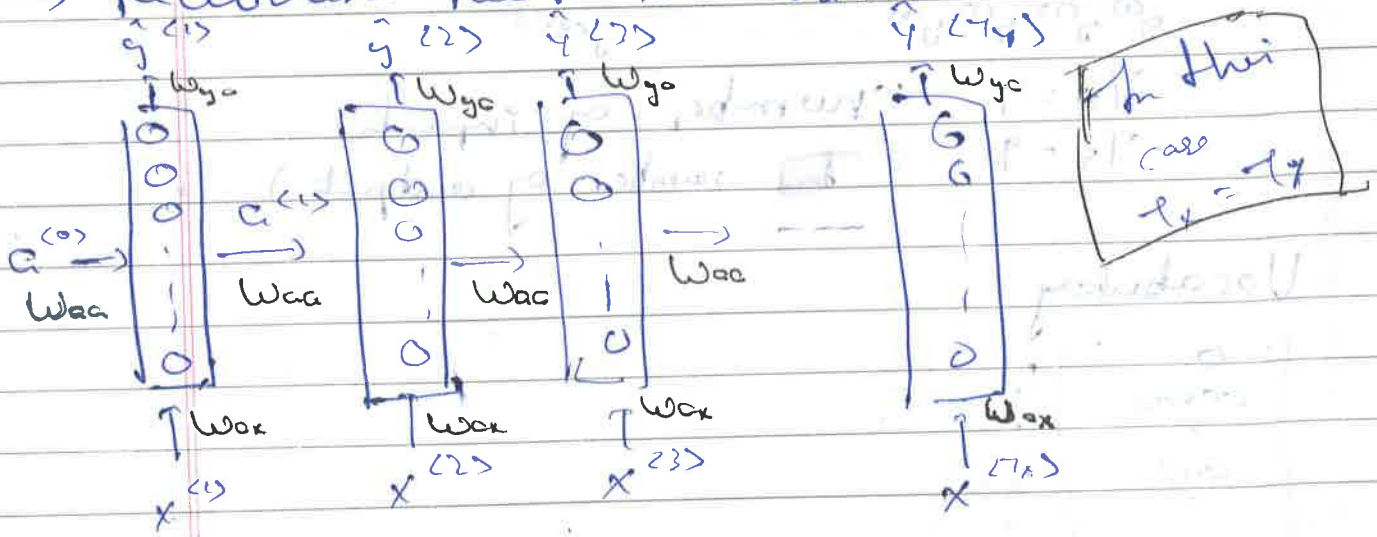


Problems:-

-> (Inputs, outputs can be of different lengths in different examples)

-> (Doesn't share features learned across different positions of text)

-> Recurrent neural network.



-> Forward propagation

$a^{\langle 0 \rangle} = \vec{0}$

$a^{\langle 1 \rangle} = g(W_{aa} a^{\langle 0 \rangle} + W_{ax} x^{\langle 1 \rangle} + b_a)$

$\hat{y}^{\langle 1 \rangle} = g(W_{ya} a^{\langle 1 \rangle} + b_y)$

mostly we use Tanh or relu function as an activation function in RNN's

thus

$a^{\langle t \rangle} = g(W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle} + b_a)$

$\hat{y}^{\langle t \rangle} = g(W_{ya} a^{\langle t \rangle} + b_y)$

$$a^{\langle t \rangle} = g(W_a [a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_a)$$

$$W_a = [W_{aa} \mid W_{ax}] \quad \updownarrow 100$$

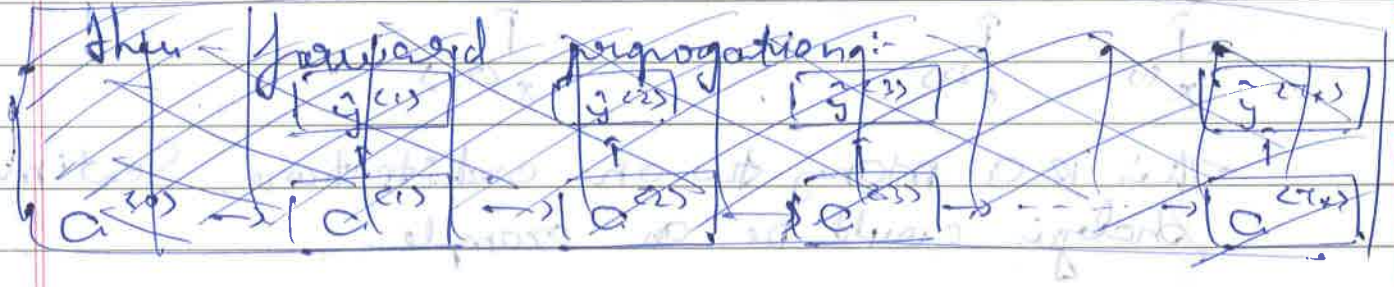
$\leftarrow 100 \rightarrow \quad \leftarrow 10{,}000 \rightarrow$

$$= W_a \quad \updownarrow (100, 10{,}000)$$

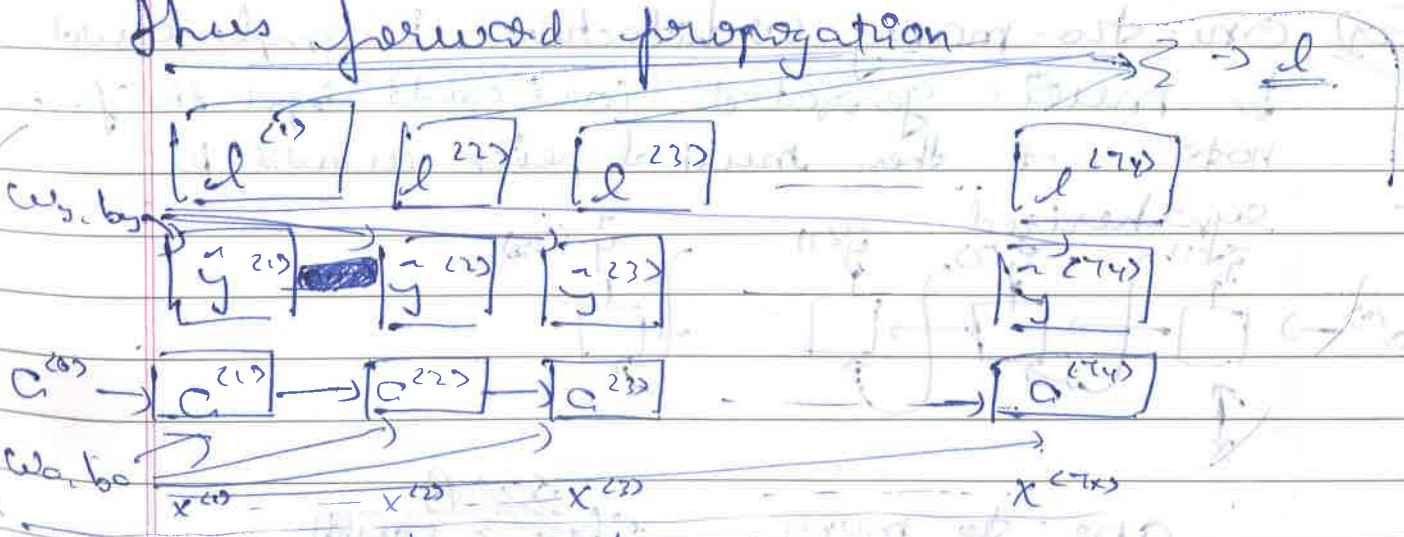
$$[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] = \begin{bmatrix} a^{\langle t-1 \rangle} & \updownarrow 100 \\ x^{\langle t \rangle} & \updownarrow 10{,}000 \end{bmatrix} \quad 10100$$

$$[W_{aa}, W_{ax}] \begin{bmatrix} a^{\langle t-1 \rangle} \\ x^{\langle t \rangle} \end{bmatrix} = W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle}$$

Thus forward propagation:-



Thus forward propagation



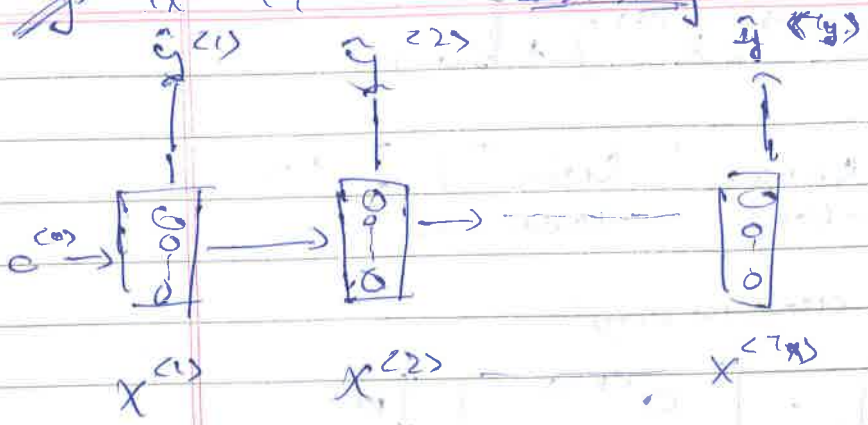
$$\mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}) = -y^{\langle t \rangle} \log \hat{y}^{\langle t \rangle} - (1 - y^{\langle t \rangle}) \log(1 - \hat{y}^{\langle t \rangle})$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle})$$

→ Example of RNN architectures.

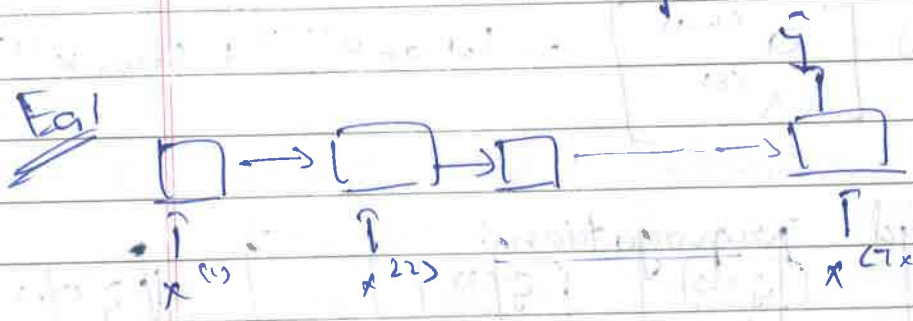
PAGE No.
DATE / /

Eg) $T_x = T_y$



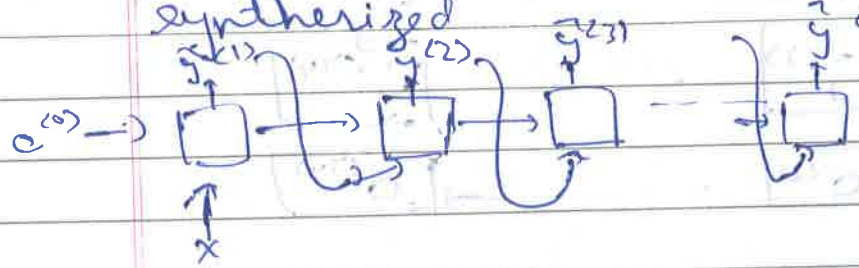
- This is an example of many-to-many architecture.

Eg!



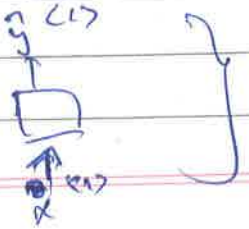
- This is a many-to-one architecture. Sentiment Analysis could be an example.

Eg! one-to-many architecture: Example could be music generated. You could input the first note ... etc then musical peice would be synthesized



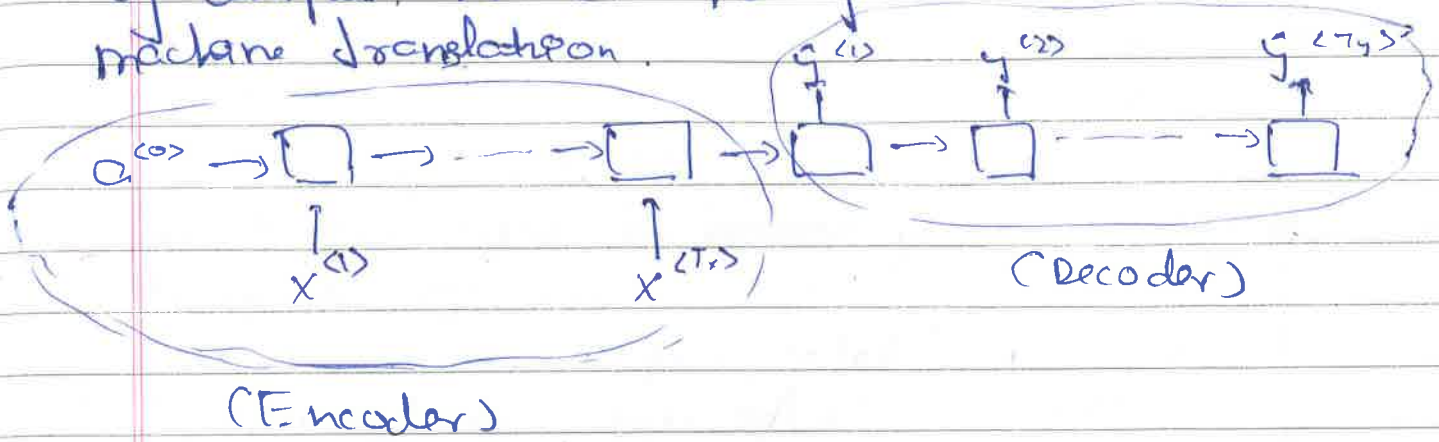
one-to-many.

Eg! We could also have one-to-one



→ This is just a generic neural network, not specifically RNN.

Eg: example of many-to-many architecture with the number of input not equal to the number of outputs. An example of this could be machine translation.



→ Autoencoders.
Unsupervised neural networks

→ What is language modelling.

Speech recognition:-

The apple and pair salad

The apple and pear salad.

Now [redacted] language model will tell which of the two should be the one when both of them sound exactly the same.

So it will return the [redacted] probability of the sentence being the right one.

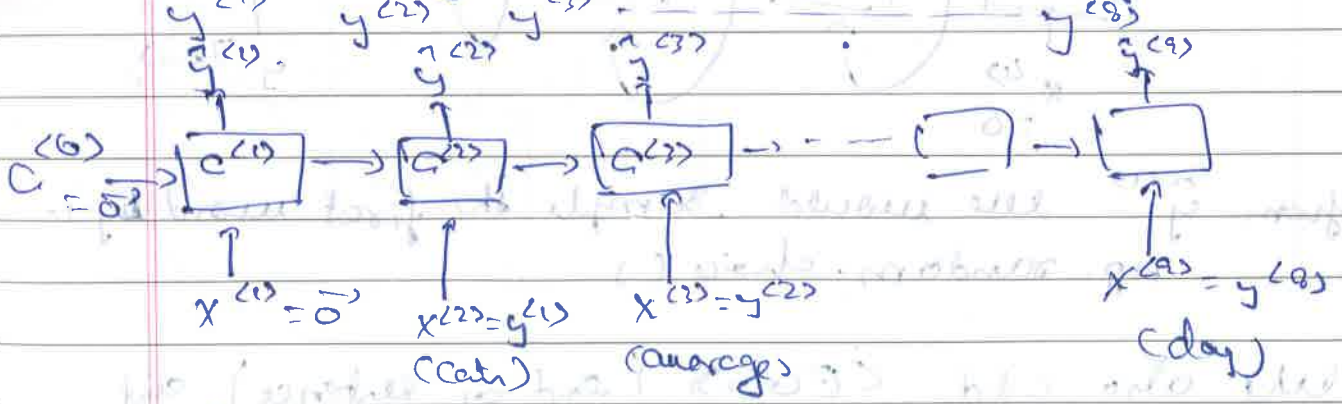
$P(\text{The apple and pair salad}) = 3.2 \times 10^{-13}$

$P(\text{The apple and pear salad}) = 5.7 \times 10^{-10}$

Training set :- large corpus of English text.

Tokenize.

(Cats average 15 hours of sleep a day.



$\hat{y}^{<t>}$ would be Softmax vector telling the probability of each and every word of the vocabulary.

we add <EOS> (end of sentences) and <UNK> (unknown) token in our vocabulary also

↳ for a word which is not in our vocabulary.

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}^{<t>}$$

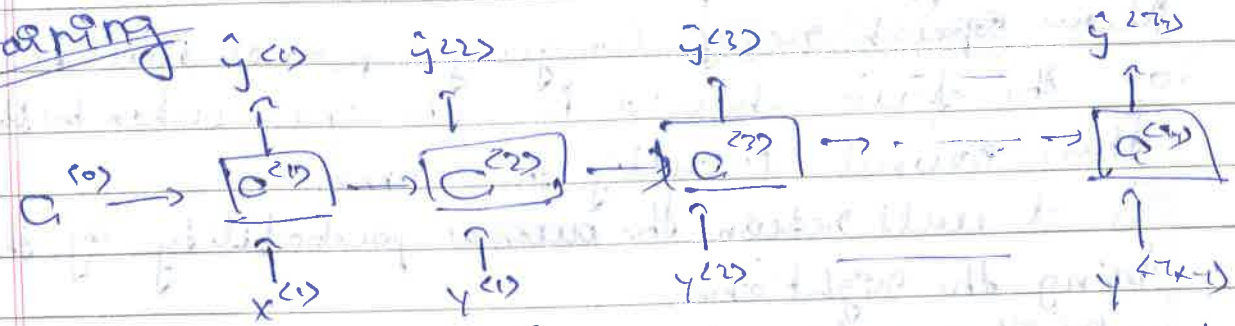
$$\mathcal{L} = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

Then the probability that the given sentence is $P(y^{<1>}, y^{<2>}, y^{<3>})$ is :-

$$P(y^{<1>}) \cdot P(y^{<2>} | y^{<1>}) \cdot P(y^{<3>} | y^{<1>}, y^{<2>})$$

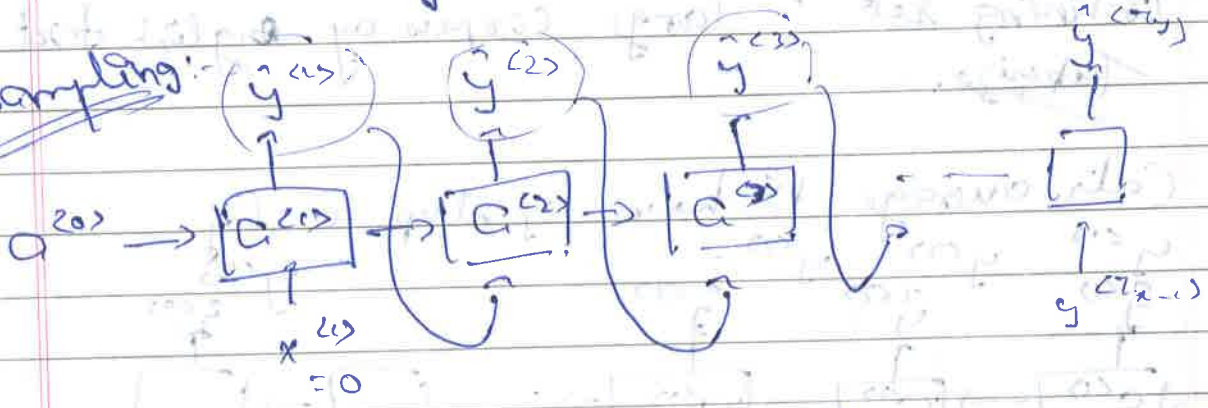
Sampling a sequence from a trained RNN.

Training



here each and every $\hat{y}$ would be a softmax vector giving the probability of each and every word from the vocabulary.

Sampling:-



from $\hat{y}^{<1>}$ we would sample the first word by :-

np.random.choice()

We also add <EOS> {end of sentence} and <UNK> {unknown} tokens in our vocabulary too.

→ Till now we were looking at word-level language model. Now let's look at character-level language model.

Vocabulary = [a, b, c ... z, ., ,, ', i, A, ... Z]

Here each and every character is given as output.

→ Vanishing gradients with RNN's

Eg. The [cat] which . . . . . [was] full.

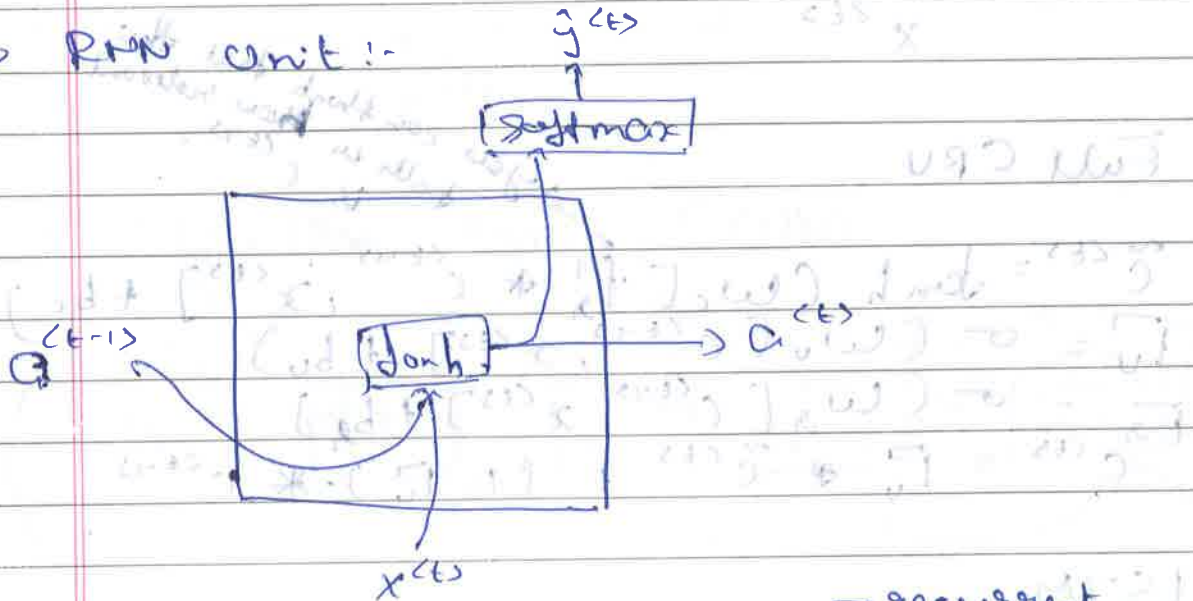
↳ if singular cat then was

The [cats] which . . . . . [were] full

↳ if plural cats then were

These words much earlier in the sentence can have effect on words much farther in the sentence. Thus RNN must also have long range dependencies. Vanishing gradient is a big problem in RNN

→ RNN unit :-



→ GRU (simplified) (Gated recurrent Unit)

(= memory cell. For this example we will consider $a^{\langle t \rangle}$ and $c^{\langle t \rangle}$ same

$$\tilde{c}^{\langle t \rangle} = \tanh(W_c[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$$

Update Gate $\Gamma_u = \sigma(W_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$

↳ "update"

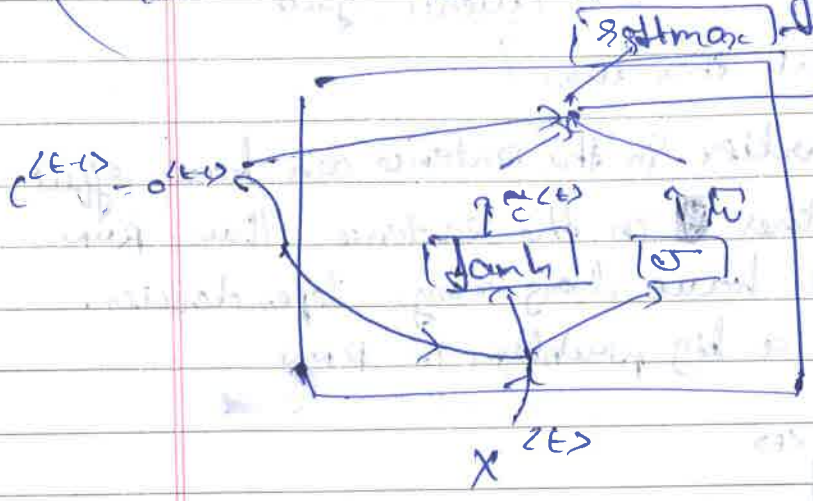
$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle}$$

element wise

$\Gamma_u = 1 \quad \Gamma_u = 0 \quad \Gamma_u = 0 \quad \cdots \quad \Gamma_u = 1$

$c^{\langle t \rangle} = 1$

The cat, which already ate ... , was full.



=> Full GRU

you can think that this tells us how relevant $c^{\langle t-1 \rangle}$ is.

$\tilde{c}^{\langle t \rangle} = \tanh(W_c[\Gamma_r * c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$

$\Gamma_u = \sigma(W_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$

$\Gamma_r = \sigma(W_r[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_r)$

$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle}$

-> LSTM

$\tilde{c}^{\langle t \rangle} = \tanh(W_c[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$

update gate $\Gamma_u = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$

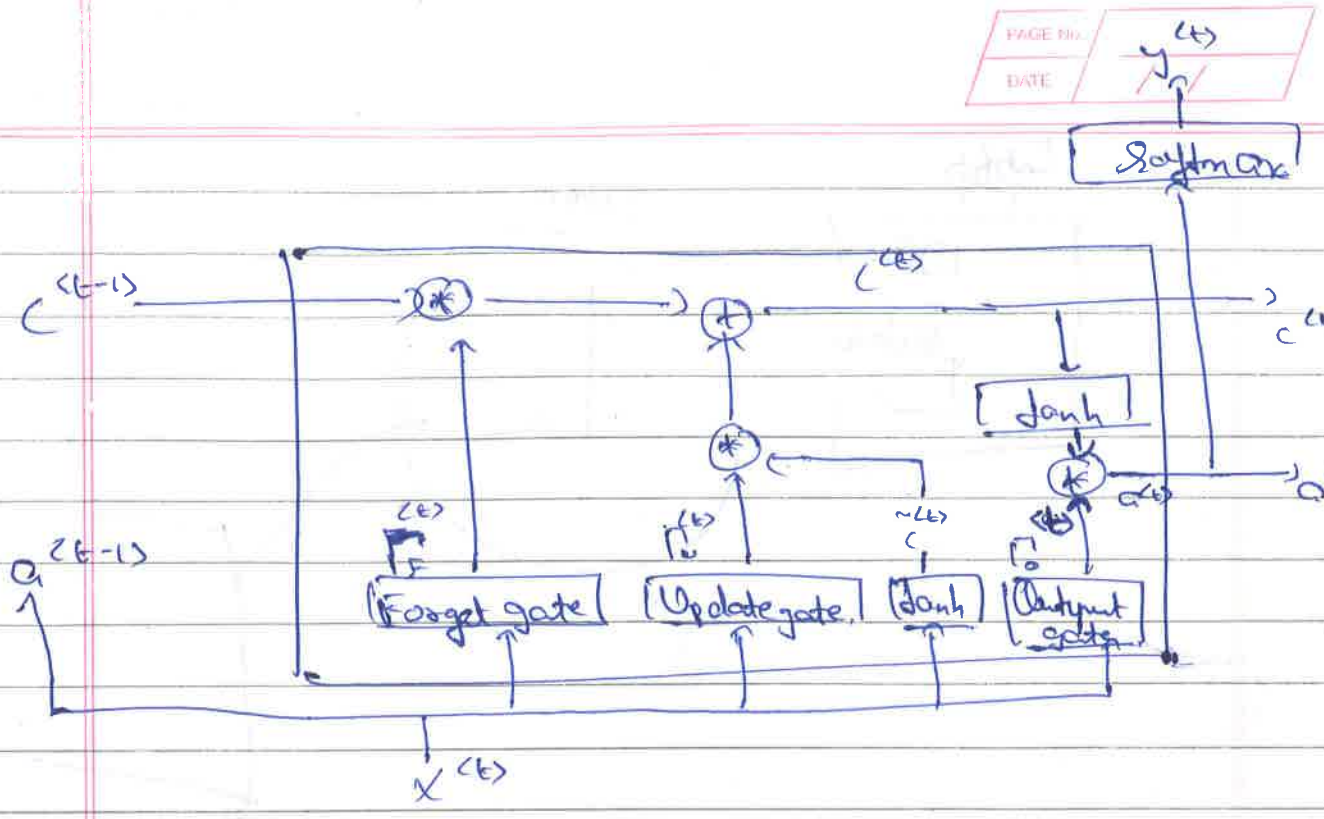
forget gate $\Gamma_f = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$

output gate $\Gamma_o = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$

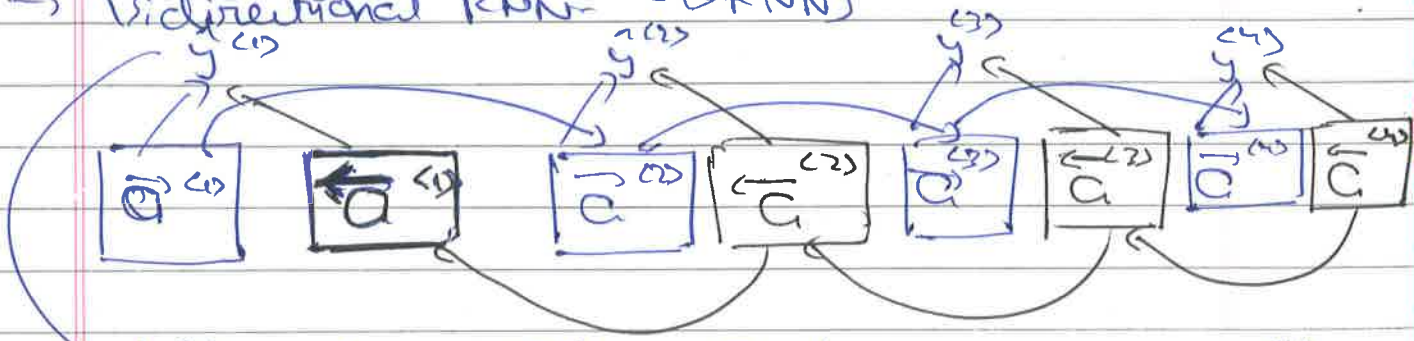
$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + \Gamma_f * c^{\langle t-1 \rangle}$

$a^{\langle t \rangle} = \Gamma_o * \tanh c^{\langle t \rangle}$

{P.T.O}

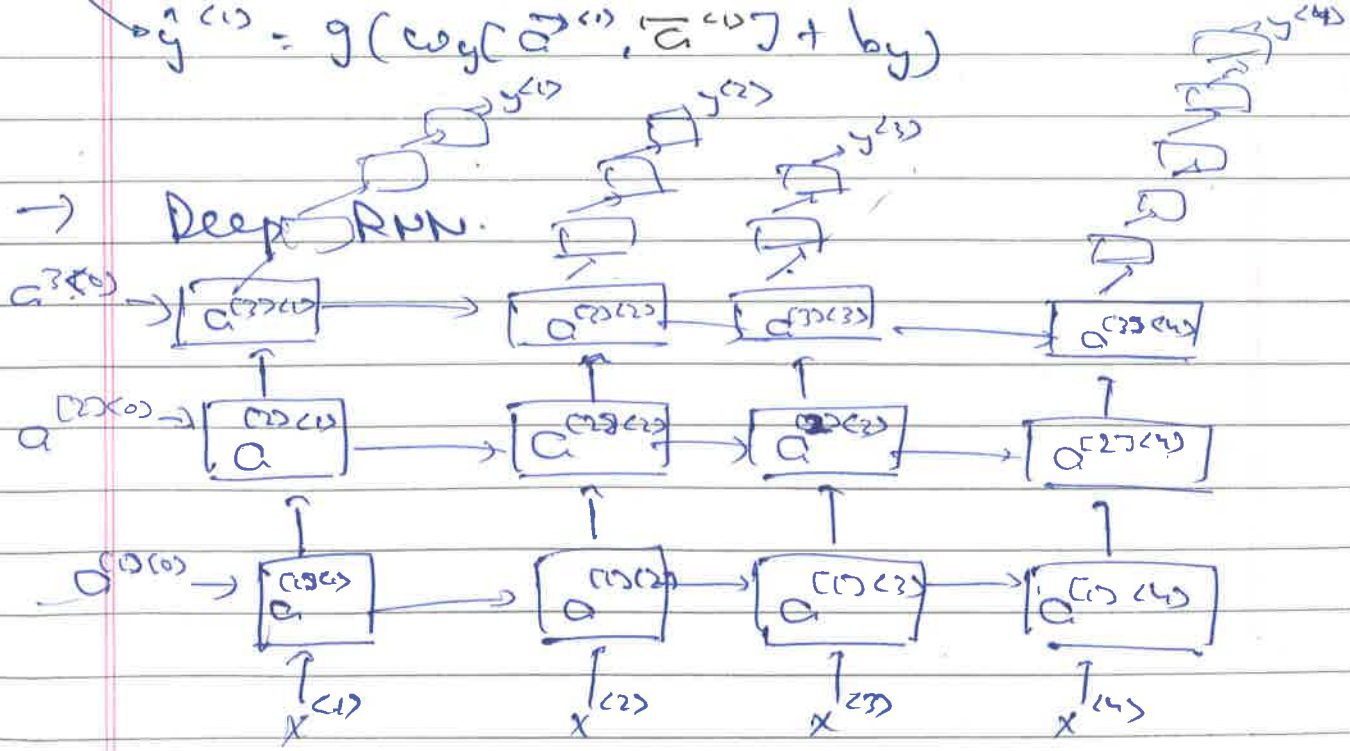


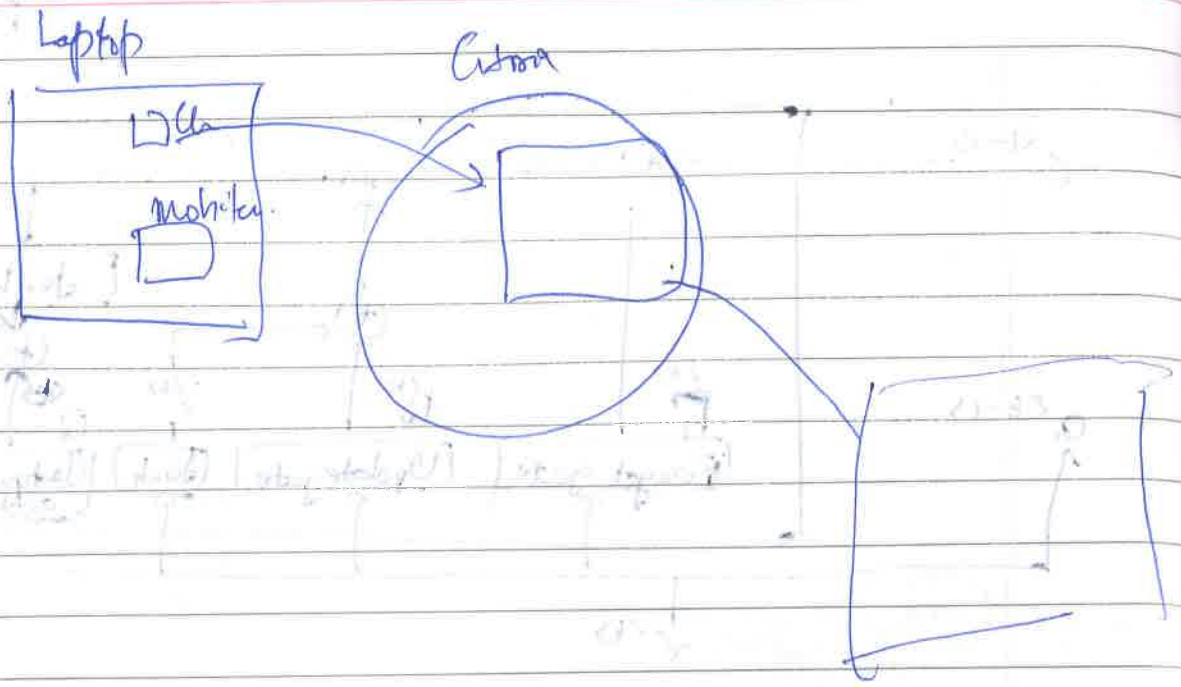
→ Bidirectional RNN (BRNN)



$$\hat{y}^{\langle 1 \rangle} = g(w_y[\overrightarrow{a}^{\langle 1 \rangle}, \overleftarrow{a}^{\langle 1 \rangle}] + b_y)$$

→ Deep RNN.




Laptop
Mobile
Cisco

→ why was nohup
← how do you install
node

bash

nohup ~~[illegible]~~ jupyter notebook &

cat nohup.out

ps -ef | grep jupyter

kill -9 982 989 31901

cat nohub.out

> nohub.out → would empty.

bash ~~script.py~~ script.sh

new session → ~~[illegible]~~ [illegible] →

[illegible] adm

du -sh ~~*~~

sudo apt ~~[illegible]~~ update

vi config

nohup.out

xpose → nohup ny

~~[illegible]~~ nohup.sh jupyter notebook.py

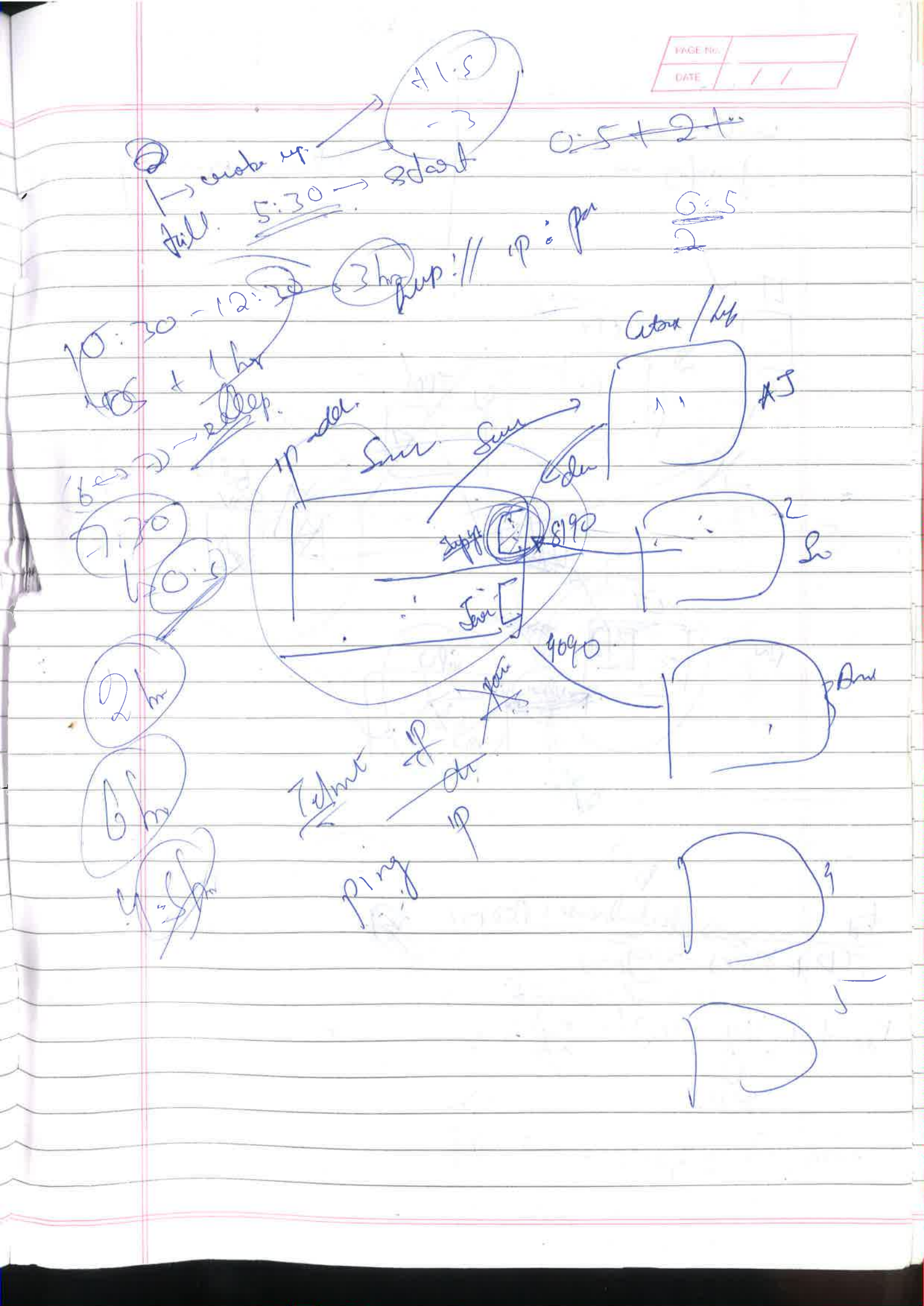
export

[illegible]

[illegible] [illegible]

conda

02279644444



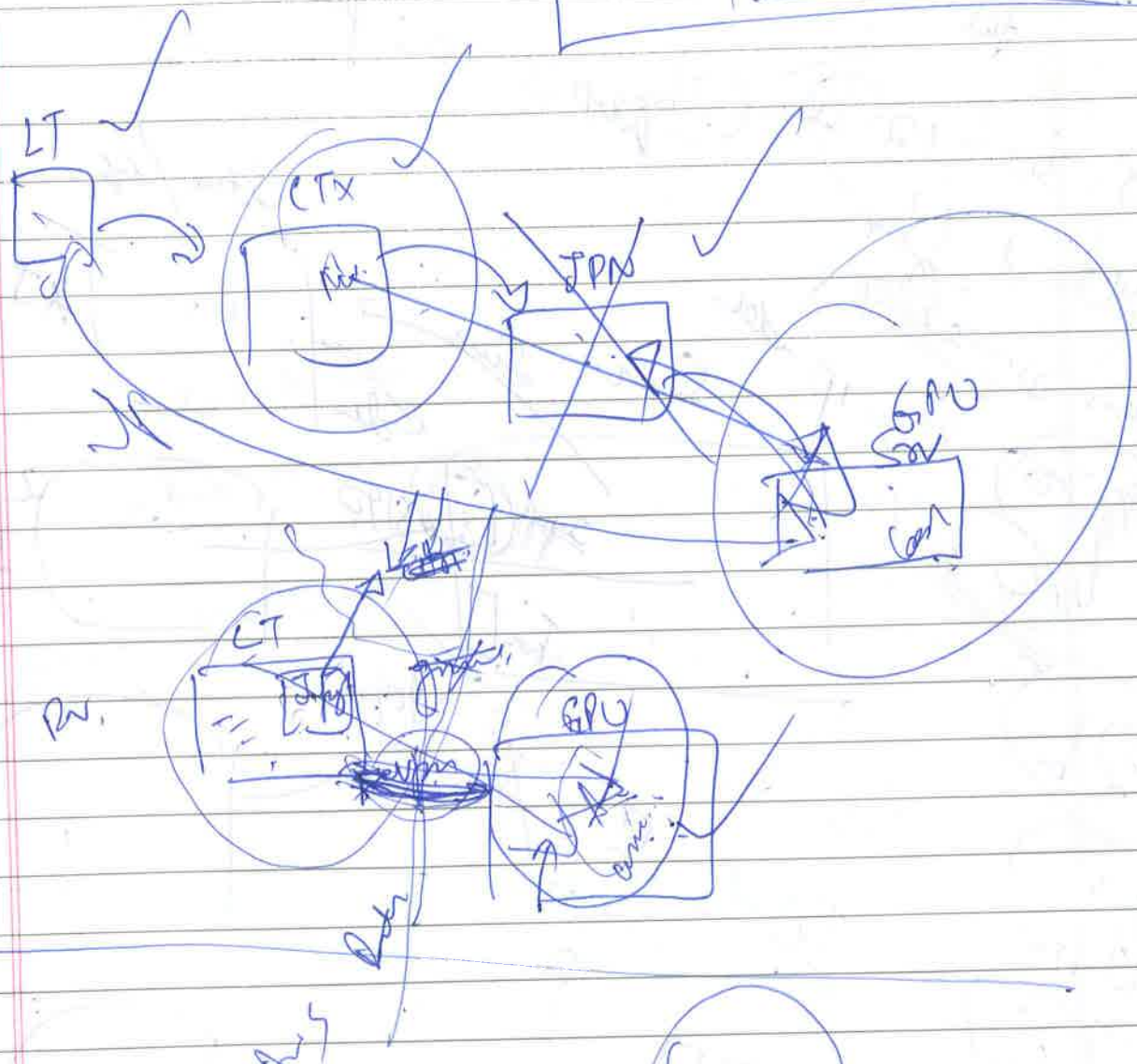
bd

Prog $m_1 \rightarrow$

10.140.102.207

whoami

ifconfig -a



database (Param)

$ID + f(x) = g(x)$

Randomly generate

$P_{ds} + (ID + f(x)) \rightarrow P_{ds}$

$g(x)$

$ID + f(x) \rightarrow h(x)$

$P_{ds} + h(x)$ $\Rightarrow$ $\phi \cdot P_{new}$

PAGE NO.
DATE / /

People

Central Bank

Firm.

Country

1 → 1 / Cr

F 1 C

Country

Govt

Central Bank

Firms / Banks.

~~1000~~ 2000 2

Tax

2%

Expense > Tax

500

23

People

1900 → Certificate → 5% yield → 100 → 5

5

5

5

15 year

200

15 200

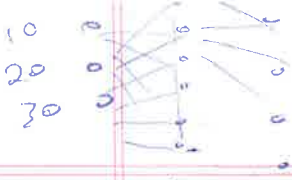
100

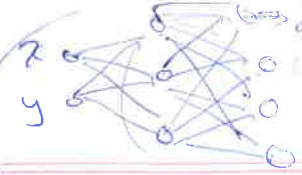
200 → 1 $

1$ → 400

200

2.5 Cr.



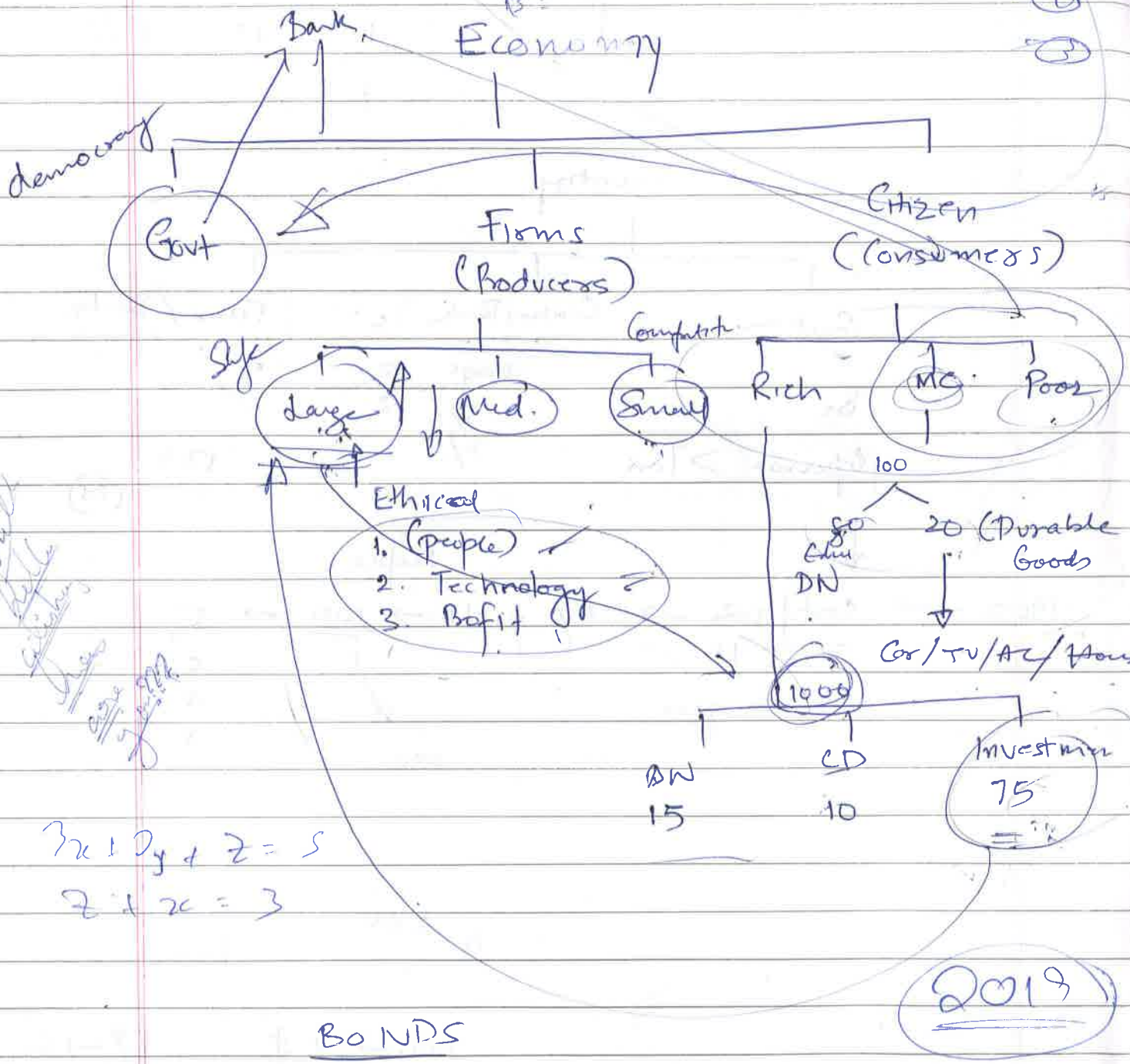


PAGE No.
DATE / /

α : $w_1.(x_{cr} + y_{w_2}) + \ldots$ -(1)

β : -(2)

-(3)



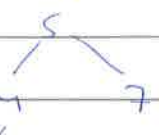
$3x + 2y + z = 5$

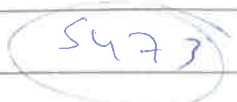
$z + x = 3$

2019

BONDS

Virtual environment





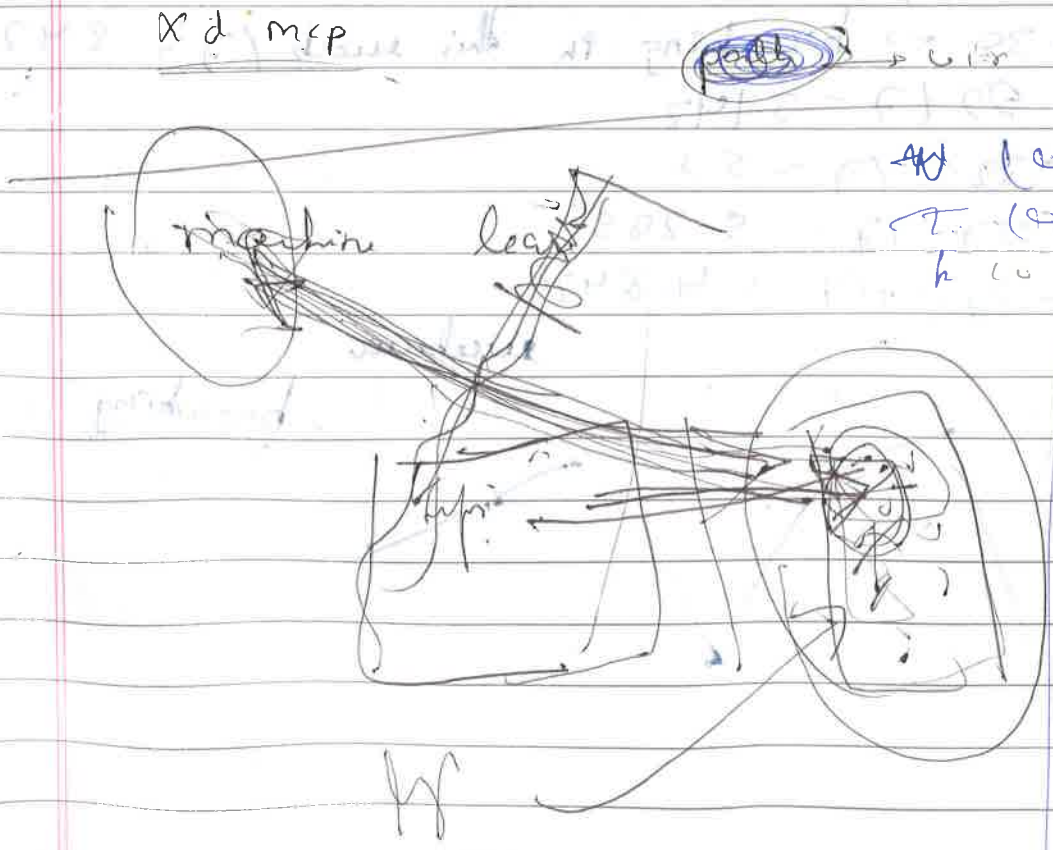
→ If we need to reach 9hr/day for the month of June :- 13.5 hr

→ If we need to reach 8hr/day overall then we need to study 10 hr 20 min everyday till the end of July.

→ If we need to reach 7hr/day overall then we need to study 12.5 hr everyday till the end of July.

subaterm → sessions → SFTP →

X d mcp



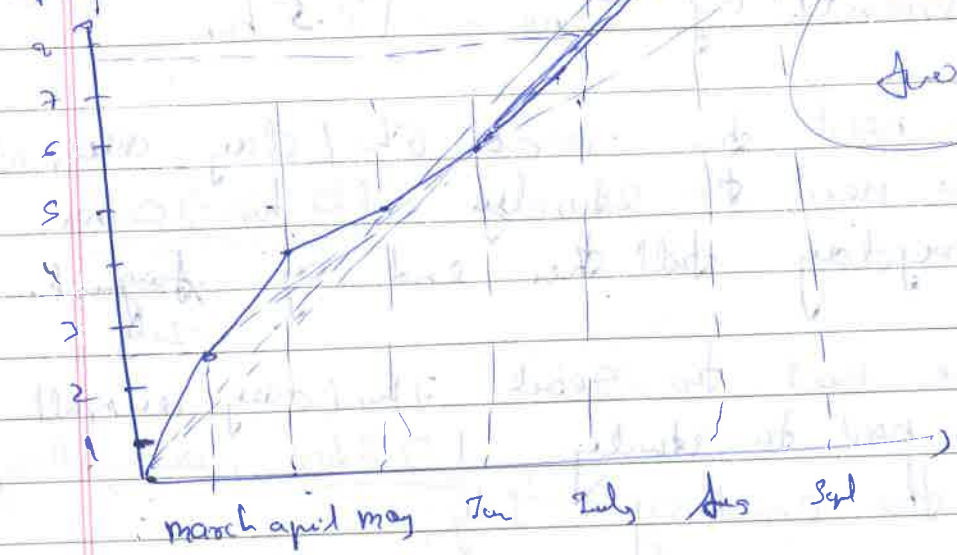
Mysadm

yetit@gmail.com

PAGE No.
DATE / /

not authorized
due scan this cale

monthwise :-



From may weekwise :-

week 1 - 19.5 /7 = 2.785

week 2 - 45 /7 = 6.428

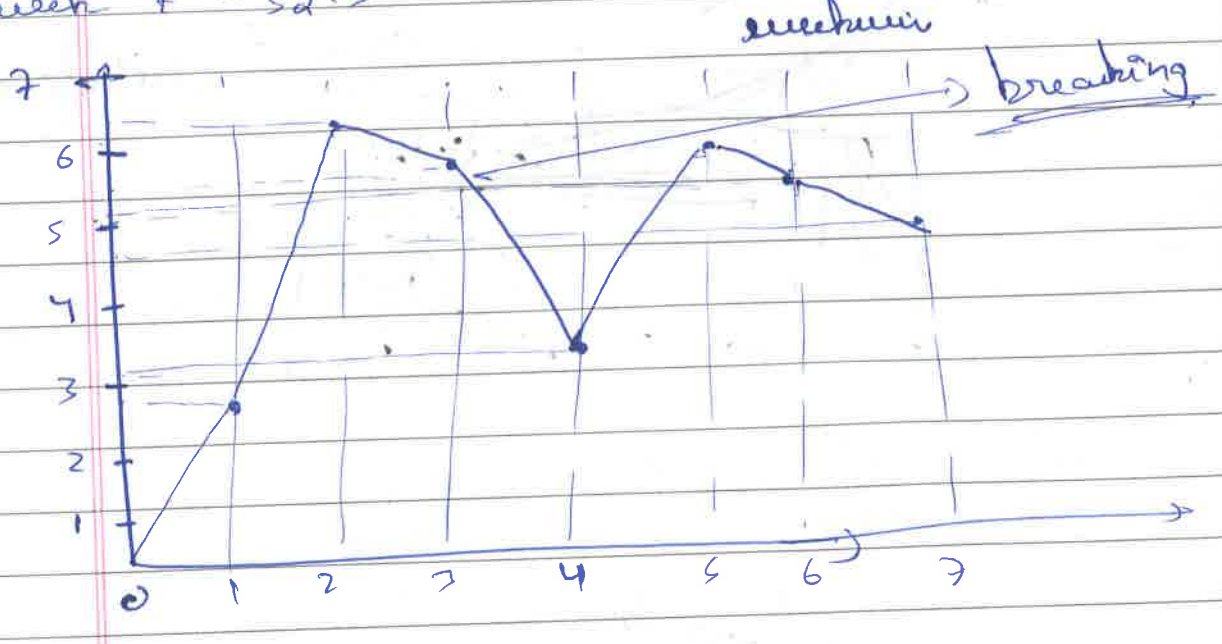
week 3 - 38 → breaking in this week /7 = 5.428

week 4 - 22 /7 = 3.142

week 5 - 38.5 /7 = 5.5

week 6 - 37 /7 = 5.285

week 7 - 32.5 /7 = 4.642



financial crises terms, frozen cell, ... vocabulary 89. 76 / 88

9 hr 8:20 → dw[1] instead of dw[2] (2 terms) 40 (57)

(ii) So this is my theory on zero initialization of w & b. I just wanted to confirm it:-

→ If we use relu activation function [redacted] then the [redacted] w & b values will be 0

→ If we use sigmoid activation function then all of the nodes of each layer would have the same value ↳ along with the w & b values of each layer

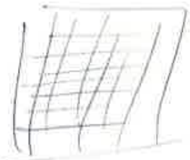
↳ and will be updated simultaneously with the same values. Thus we could then compress the entire neural network to a single equation. (Eg. equation of logistic regression)

---

linear_cache ⇒ (A, W, b) → $A^{[0]}$

cache ⇒ (linear_cache, activation_cache)

→ visualize weights corresponding to a node as a picture ideally with ((x)×(y)) pixels.



→ In non-CNN neural networks the nodes of layers not necessarily detect edges / curves / combination of edges because even though we input a totally random image we still get an output that the image is this particular ~~img~~ object with high probability.

normal initialization → randn