Page No. 101
Date 9 | 5 | 21

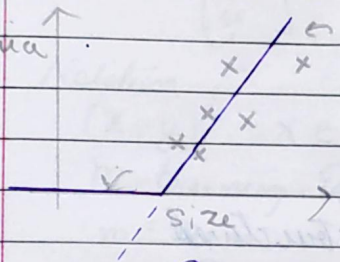
DEEP LEARNING - Course 1

↳ AI is the new electricity and will bring about an equally big transformation.

* What we'll learn in this Specialization?

1. Neural Network & Deep learning
2. Improving Deep Neural Networks : Hyperparameter tuning, Regularization and optimization
3. Structure your Machine learning project
4. Convolutional Neural Networks [CNN]
5. Natural Language processing - Building sequence models (RNN, LSTM)

⇒ House Price Prediction :

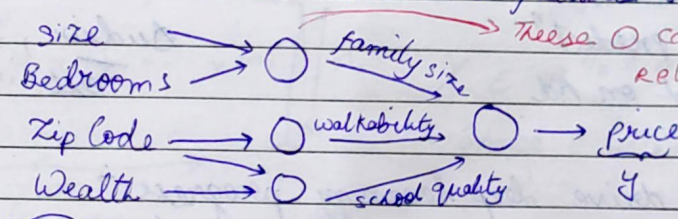


as Price can't be negative
→ ReLU } Rectified Linear Unit.

(x) Size → ○ → Price (y)
↳ "Neuron"

There can be more features too

Ex

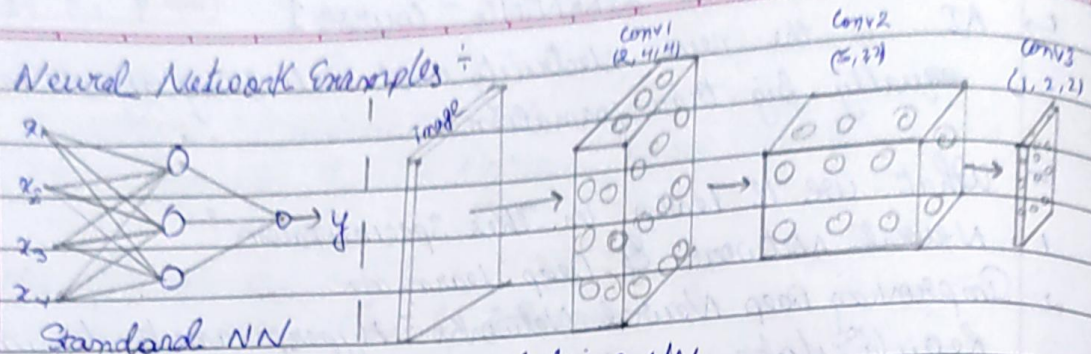


x

⇒ Supervised learning :

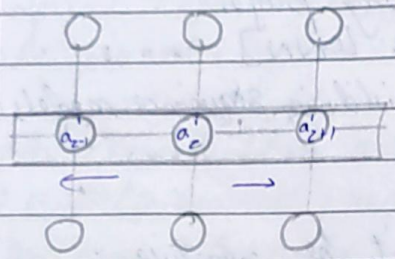
Input (x)	Output (y)	Application	
Home features	Price	Real Estate	} standard NN
Ad, user info	Click on Ad (0,1)	Online Advertising	
Image	Object (1,...,1000)	Photo Tagging	} CNN
Audio	Text transcript	Speech Recognition	} RNN
English	Chinese	Machine Translation	
Image, Radar info	Position of other cars	Autonomous Driving	} Custom / Hybrid

⇒ Neural Network Examples :



Standard NN

Convolutional NN



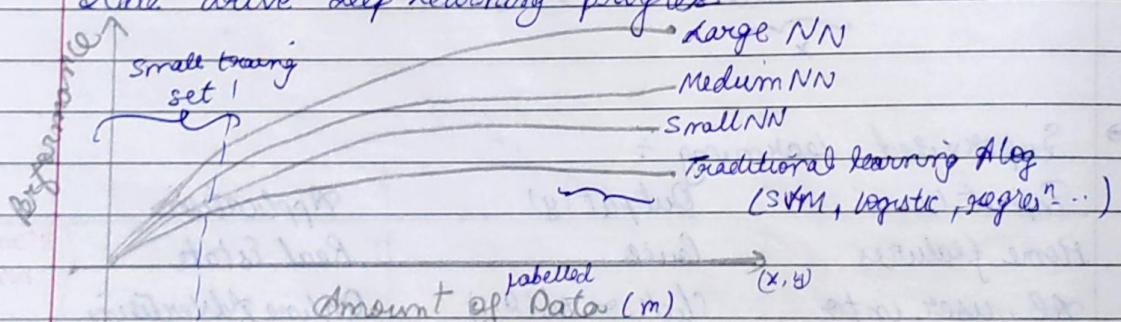
Recurrent NN

⇒ Supervised learning :

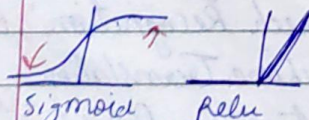
Structured Data
Price predict"
Clicked on Ad

Unstructured
Audio, Image,
Text

⇒ Scale drive deep learning progress :



Algorithms : To run faster Relu is better than Sigmoid?



In Sigmoid slope is close to zero at end so learning becomes very slow. When GD is used the parameters change very slowly ; when we use Relu gradient = 1, so GD runs faster than sigmoid. Computationaly deep

This cycle can be done in seconds, minutes, days or even weeks.

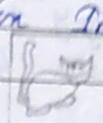
Idea → Experiment → Code

Week - 2

Basics of Neural Network

Binary Classification

Ex Image is a cat (1) or not (0)



→ This image can be represented in Blue, Green, Red

$\therefore x = \begin{bmatrix} 255 \\ 134 \\ \vdots \\ 255 \\ 134 \\ \vdots \\ 45 \end{bmatrix}$ $64 \times 64 \times 3 = 12288$

$n = n_x = 12288$ $\quad x \to y\ (0/1)$

Notation :-

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

m training Example $\{(x^{(1)}, y^{(1)}), \cdots (x^{(m)}, y^{(m)})\}$

$m = m_{train}$; m_{test} ⇒ test examples

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

m

$X.shape = (n_x, m)$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} \quad ; \quad Y \in \mathbb{R}^{1 \times m}$$

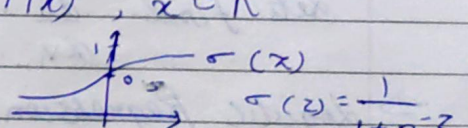
$Y.shape = (1, m)$

⇒ Logistic Regression :-

Given x, we want $\hat{y} = P(y=1|x)$; $x \in \mathbb{R}^{n_x}$

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

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



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

If z is large $\sigma(z) \approx 1$; If z is small $\sigma(z) \approx 0$; We do use $x_0 = 1$, $x \in \mathbb{R}^{n_x+1}$

$\hat{y} = \sigma(\theta^T x)$ & $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$ $\theta_0 \to b$, $\theta_1 \cdots \theta_{n_x} \to w$

⇒ Cost function :

Loss (error) function : $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

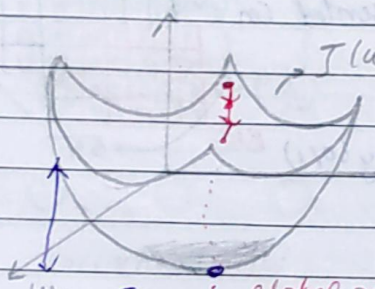
Applied to single training example

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

If $y=1$ $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ← As small as possible i.e $\hat{y}$ as big

If $y=0$ $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ ← $\hat{y}$ must be big

Cost fn : $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})$

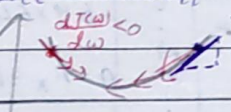


↳ global optimum

↳ It can be multidimensn

For graph purpose we are taking it as a single vectore here

Gradient Descent :



repeat {

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

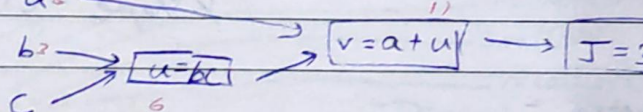
$b := b - \alpha \frac{\partial J(w,b)}{\partial b}$ }

⇒ Computational graph :

Say $J(a, b, c) = 3(a + bc) = 33$ $u = bc$ $a = 5$

Now how to compute $v = a + u$ $b = 3$

using 3 steps $J = 3v$ $c = 2$



Lets find $\frac{dJ}{dv} = ? \; 3$; $\frac{dJ}{da} = 3$

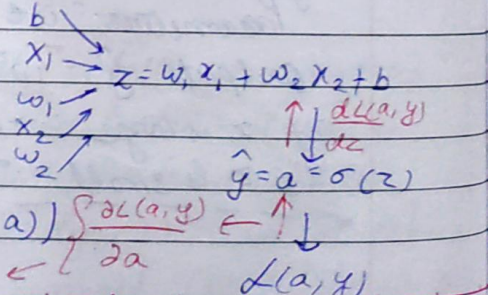
⇒ Logistic Regression :

$z = w^T x + b$

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

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

$\frac{\partial \mathcal{L}(a,y)}{\partial a} \leftarrow \frac{-y}{a} + \frac{1-y}{1-a}$



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

$\frac{dL}{dz} = \frac{dL}{da} \cdot \frac{da}{dz} \rightarrow a(1-a) \rightarrow$ Sigmoid derivative

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

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

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

These are "Backpropagation" steps to go back

$\frac{dL}{dw_1} = x_1 dz \; ; \; dw_2 = x_2 dz \; , \; db = dz$

⇒ Gradient descent for m training example:

$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(a^{(i)}, y)$ $\quad dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

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

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

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

⇒ Logistic Regression on m examples:

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

First for loop

For $i = 1$ to m:

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

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

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

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

Second for loop (though we only chose 2 features) — for $n = 2$:

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

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

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

$J /= m \; , \; dw_1 /= m \; ; \; dw_2 /= m \; ; \; db /= m \; ;$

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

★ Vectorization :

What is Vectorization ?

$z = w^T x + b$

$w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $x, w \in \mathbb{R}^{n_x}$

Non-Vectorized

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

Vectorized

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

($w^T x$)

★ How much fast vectorized version is
Create your own Jupyter Notebook

CPU, GPU → SIMD - single instruct^n multiple data

⇒ Neural Network programming guidelines :

Whenever possible, avoid explict for-loops.

$u = Av$

$u_i = \sum_j A_{ij} v_j$

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

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

Vectorized way faster

Ex We need exponential operation on each and every element

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

For-loops

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

Vectorized

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

⇒ Logistic regression derivatives :

$J = 0$ ~~$dw_1 = 0$; $dw_2 = 0$~~ , $db = 0$ $dw = np.zeros((6_x, 1))$

one for loop — for i in range(m):

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

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

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

Second for loop & REMOVED & vectorized method take its place

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

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

~~$dw_2 \mathrel{+}= x_2^{(i)} \cdot dz^{(i)}$~~ $\Big\}$ n_x $dw \mathrel{+}= x^{(i)} dz^{(i)}$

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

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

$dw /= m$

⇒ Vectorizing Logistic Regression :

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

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

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}; \quad X \in \mathbb{R}^{n_x \times m} \qquad w^T \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

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

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

$Z = np.dot(w^T, X) + b \rightarrow (1, 1)$

$1 \times m$ "Broadcasting"

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

So here we didn't use any for loops for our model

⇒ Vectorizing Logistic Regression's Gradient computation :

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

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

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

$Y = [y^{(1)} \cdots y^{(m)}]$

We still have this for loop left

$dw = 0$ | $db = 0$
$dw += x^{(1)} dz^{(1)}$ | $db += dz^{(1)}$
$dw += x^{(2)} dz^{(2)}$ | $db += dz^{(2)}$
⋮ | $db += dz^{(m)}$
$dw /= m$ | $db /= m$

In one line

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

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

$$= \frac{1}{m} \left[x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)} \right] \rightarrow n \times 1$$

⇒ Vectorized Implementation: for iter in range(1000):

We have to use one for loop for iteration

$Z = w^T X + b = np.dot(w.T, X) + b$ ↓ Can't replace it
$A = \sigma(Z)$
$dZ = A - Y$
$dw = \frac{1}{m} X dZ^T$
$db = \frac{1}{m} np.sum(dZ)$
$w := w - \alpha\, dw$
$b := b - \alpha\, db$

⇒ Broadcasting in python:

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

	Apples	Beef	Eggs	Potatoes
Carbs	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	0.8
Fat	1.8	135.0	99.0	0.9

(3, 4) = A

Calculate % of carbs from Carb, Protein, Fat?

59 cal

$\therefore \frac{56}{59} \approx 95\%$ of carb

Let's do it without a for-loop?

cal0 = A.sum(axis = 0) → for column axis = 0

percentage = 100 * A / (cal0.reshape(1, 4)) → Not necessary to call

* check for Broadcasting in Jupyter Notebook.

(3,4) / (1,4) Matrix

Ex $\begin{bmatrix}1\\2\\3\end{bmatrix} + 100 \rightarrow \begin{bmatrix}100\\100\\100\end{bmatrix} = \begin{bmatrix}101\\102\\103\end{bmatrix}$

Python Broadcasted (1,1) to (3,1) Matrix

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

(m,n) (2,3) Python Broadcasted (1,3) to (2,3) Matrix

Ex $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$

(2,3) (2,1) → (2,3)

⇒ General Principle :

(m, n) $\pm \times /$ (1, n) ⇝ (m, n) Broadcasted to

Matrix (m, 1) ⇝ (m, n)

Ex $(m,1)\begin{bmatrix}1\\2\\3\end{bmatrix} + \mathbb{R}\,100 = \begin{bmatrix}101\\102\\103\end{bmatrix}$

[1 2 3] + 100 = [101 102 103]

Matlab/Octave : bsx fun } Broadcasting in Matlab/octave

⇒ Python/numpy vectors :

✗ a = np.random.randn(5)
a.shape = (5,) → This is a "rank 1 array" } Don't use rank 1 arrays

✓ a = np.random.rand(5,1) } use this column vector
a.shape = (5,1)
assert(a.shape == (5,1)) } To check for shape
a = a.reshape((5,1)) } you can reshape rank 1 array to colum vector

★ Vectors/Matrix in Python & Reshaping

⇒ Logistic Regression Cost function :

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

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

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

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

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

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

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

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

$= -\mathcal{L}(\hat{y}, y)$ ↓ minimize cost fn

⇒ Cost on m examples :

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

$\log P(\dots) = \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)})$ } Maximum Likelihood estimation

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

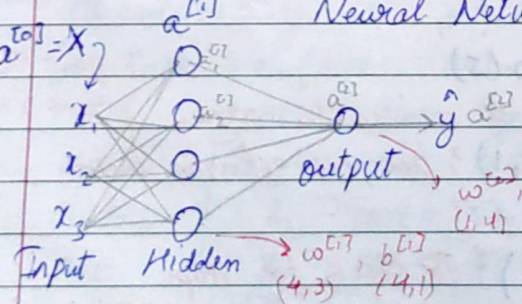
$= -\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

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

(Minimize)

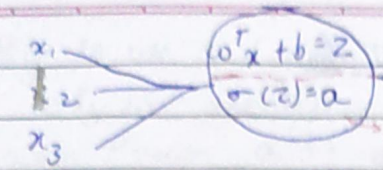
Week - 3

Neural Network overview



This is a 2 layer NN (Hidden & Output)

We don't count input layer.



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

$a_i^{[l]}$ ← layer, → node in layer

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

Instead of using for-loop we will be vectorising these equations

$$\underbrace{\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}}_{z^{[1]}} = \underbrace{\begin{bmatrix} \cdots & w_1^{[1]T} & \cdots \\ \cdots & w_2^{[1]T} & \cdots \\ \cdots & w_3^{[1]T} & \cdots \\ \cdots & w_4^{[1]T} & \cdots \end{bmatrix}}_{W^{[1]}\ (4,3)} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{(3,1)} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ \vdots \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]}\ (4,1)} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ \vdots \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix}$$

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

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

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

⇒ Vectorizing across multiple examples :

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

for various x cases

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

$a^{[2](i)}$ → example i, ↳ layer 2

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

for $i = 1$ to m,

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

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

$$A^{[1]} = \begin{bmatrix} a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$
(Hidden unit-number, → training Example)

So our Vectorized Algo will be :-

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

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

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

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

⇒ Justificat^n for vectorized implementation :-

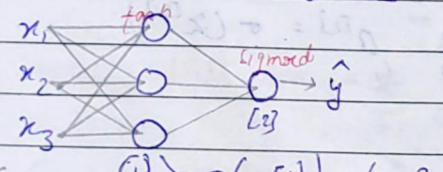
$z^{[1](1)} = W^{[1]T} x^{(1)} + b^{[1]}$; $z^{[1](2)} = W^{[1]T} x^{(2)} + b^{[1]}$ ....

$W^{[1]} = \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix}$; $W^{[1]} x^{(1)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$; $W^{[1]} x^{(2)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$ ....

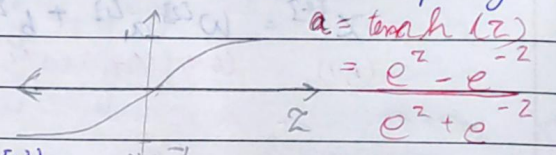
$Z^{[1]} = W^{[1]} \begin{bmatrix} | & | \\ x^{(1)} & x^{(2)} \cdots \\ | & | \end{bmatrix} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} = \begin{bmatrix} | & | \\ z^{[1](1)} & z^{[1](2)} \cdots \\ | & | \end{bmatrix}$

We can add $b^{[1]}$ via broadcasting $+b^{[1]} \quad +b^{[1]} \cdots$

⇒ Activation Functions :- There can be other f^n except sigmoid



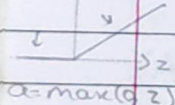
$g(z^{[1]}) = \tanh(z^{[1]})$



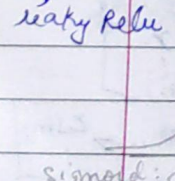
$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

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

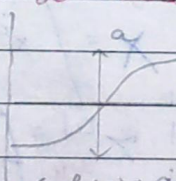
$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$ & $a^{[2]} = \sigma(z^{[2]})$ $g(z^{[2]})$



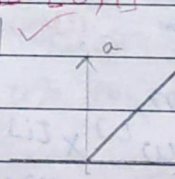
We mostly use tanh ; just for final node of binary classificat^n we use sigmoid [0,1] whereas tanh [-1,1] ✗



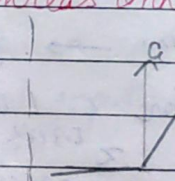
sigmoid : $a = \frac{1}{1+e^{-z}}$



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



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



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

⇒ Why do we need Non-Linear Activation function ?

Why we are using f^n like sigmoid, tanh, or ReLu

We know $a^{[1]} = g^{[1]}(z^{[1]})$ instead we will write

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

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

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

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

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

If we use linear f^n we are eventually getting a linear f^n as our output. This funct^n is only useful when we predict house price. $g(z) = z$ and that too at the output layer not in between. We can also use ReLu for housing price prediction.

⇒ Derivatives of Activation funct^n :

i) Sigmoid f^n $f(z) = \frac{1}{1+e^{-z}}$ $f'(z) = \frac{0 + (+e^{-z})}{(1+e^{-z})^2}$

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

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

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

ii) $g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ $g'(z) = \frac{(e^z+e^z)(e^z+e^{-z}) - (e^z-e^{-z})(e^z-e^{-z})}{(e^z+e^{-z})^2}$

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

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

If $z = 10$, $\tanh \approx 1$; $g'(z) \approx 0$

$z = -10$; $\tanh \approx -1$; $g'(z) \approx 0$

$z = 0$; $\tanh \approx 0$; $g'(z) = 1$

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

iii) Relu & Leaky Relu :

$g(z) = \max(0, z)$ $\qquad$ $g(z) = \max(0.01z, z)$

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

⇒ Gradient Descent for Neural Network :

Parameters : $w^{[1]}$ $b^{[1]}$ $w^{[2]}$ $b^{[2]}$; $n_x = n^{[0]}, n^{[1]}, n^{[2]} = 1$

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

Cost function : $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$ ($\hat{y} \hookrightarrow a^{[2]}$)

Gradient descent : Repeat {

Compute Predictions $(\hat{y}^{(i)}, i = 1 \cdots m)$

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

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

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

; for $w^{[2]}$ & $b^{[2]}$ }

⇒ Formulas for computing derivatives :

Forward propogation

↳ $z^{[1]} = w^{[1]} X + b^{[1]}$

↳ $A^{[1]} = g^{[1]}(z^{[1]})$

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

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

Backpropogation Step

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

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

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

$(n^{[2]}, 1)$ avoids rank 1 array

↳ $dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$

$(n^{[1]}, m)$ ↳ element wise × $(n^{[1]}, m)$

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

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

$(n^{[1]}, 1)$ or we can keep reshape too

⇒ Backpropagation Intuition :

Computing Gradient :

Logistic Regression

$x, w, b \rightarrow z = w^T x + b \rightarrow a = \sigma(z) \rightarrow \mathcal{L}(a, y)$

$dz = a - y \qquad da = \frac{d}{da}\left[\mathcal{L}(a,y) = -y \log a - (1-y)\log(1-a)\right]$

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

$dw = dz \cdot x \qquad dz = da \cdot g'(z) \qquad g'(z) = \sigma(z)$

$db = dz$

$\because \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \rightarrow \frac{\partial}{\partial z} g(z) = g'(z)$

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

⇒ Neural Network gradient : $w^{[2]}, b^{[2]} \rightarrow dw^{[2]}, db^{[2]}$

$x \rightarrow w^{[1]}, b^{[1]} \rightarrow z^{[1]} = w^{[1]} x + b^{[1]} \rightarrow a^{[1]} = \sigma(z^{[1]}) \rightarrow z^{[2]} = w^{[2]} a^{[1]} + b^{[2]} \rightarrow a^{[2]} = \sigma(z^{[2]}) \rightarrow \mathcal{L}(a^{[2]}, y)$

$dz^{[1]} \leftarrow da^{[1]} \leftarrow dz^{[2]} \leftarrow da^{[2]} \leftarrow$

$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]}) \qquad dz^{[2]} = a^{[2]} - y$

$\begin{cases} dw^{[2]} = dz^{[2]} \cdot a^{[1]T} \rightarrow dw = dz \cdot x^T \\ db^{[2]} = dz^{[2]} \end{cases}$ Extra Transpose only

$x_1, x_2, x_3 \rightarrow \hat{y} \qquad n^{[2]} = 1 \qquad n_x = n_0^{[0]} \qquad n^{[1]}$

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

$z^{[2]}, dz^{[2]} = (n^{[2]}, 1)$ i.e. $(1,1)$ here

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

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

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

Similar

$\begin{cases} dw^{[1]} = dz^{[1]} \cdot x^T \\ db^{[1]} = dz^{[1]} \end{cases}$

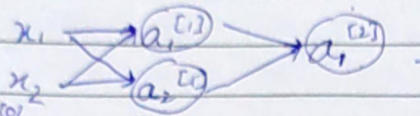
⇒ Vectorized Implementation :

All the same formula

→ These 6 are the vectorized formulas.

$\frac{1}{m}$ appears just because $J(\hat{y}, y) = \frac{1}{m} \sum \ldots$

⇒ Random initialization :- What happens if you initialize weights to zero ?



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

$$\begin{cases} w^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{cases}$$ We have initialize it with zeros.

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

$dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$ same $\cdots$ $w^{[1]} = w^{[1]} - \alpha dw$ & $w^{[1]} = \begin{bmatrix} \cdots \\ \cdots \end{bmatrix}$ same

This will make all nodes of hidden layer identical & they will compute same thing again & again.

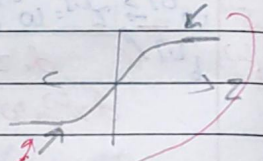
Soln must randomly initialize the weights.

$$\begin{cases} w^{[1]} = np.random.randn((2,2)) * 0.01 \\ b^{[1]} = np.zero((2,1)) \end{cases}$$ To have small values. No problem with b it can be 0

Same for $w^{[2]}$ & $b^{[2]}$

If w is too large $z^{[1]} = (w^{[1]} x + b^{[1]})$ large

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

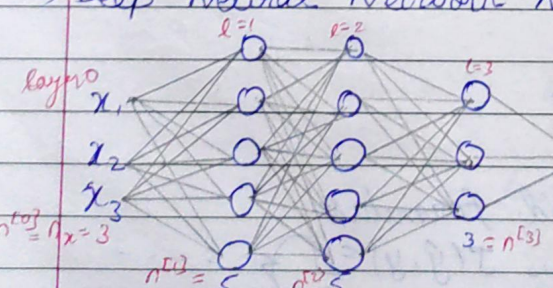


then activation function have to start from far away activation points which will slow down gradient descent & we keep small weights in increase speed of GD.

Week - 4

* Deep L-layer Neural Network

⇒ Deep Neural Network Notation :-



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

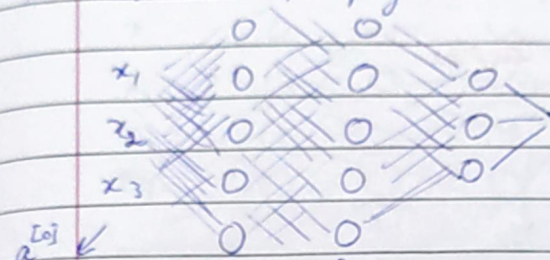
$a^{[l]} = g^{[l]}(z^{[l]})$

$w^{[l]}$ = weights for $z^{[l]}$

$b^{[l]}$ = bias for $z^{[l]}$

L = 4 [no. of layers] , $n^{[l]}$ = no. of units in layer l

⇒ Forward propogation in a deep network :



$z^{[1]} = w^{[1]} x + b^{[1]}$
$a^{[1]} = g^{[1]}(z^{[1]})$
$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$
$a^{[2]} = g^{[2]}(z^{[2]})$
$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$
$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$

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

⇒ Vectorized :

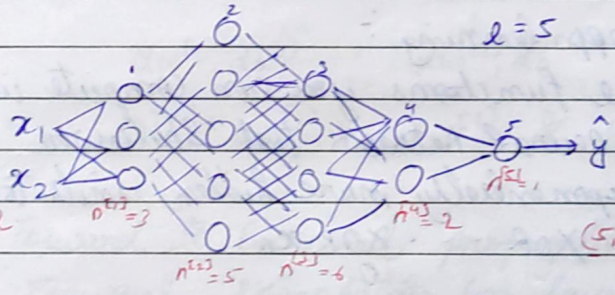
$[z^{[1](1)} z^{[1](2)} \dots] = Z^{[1]} = w^{[1]} X + b^{[1]}$; $A^{[0]} = X$ | for $l = 1 \dots 4$

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

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

⇒ Getting your matrix Dimensions right :

Parameters $w^{[l]}$ & $b^{[l]}$



$z^{[1]} = w^{[1]} x + b^{[1]} \rightarrow (3,1)$; $(z,1)$ or $(n^{[0]},1)$

$(3,1)$ or $(n^{[1]},1) \rightarrow (n^{[1]}, n^{[0]})$

dim $w^{[2]} = (5,3)$ or $(n^{[2]}, n^{[1]})$

$w z^{[2]} = w^{[2]} a^{[1]} + b^{[2]} \rightarrow (5,1)$

$(5,1) = (5,3) \cdot (3,1)$; w will be $(5,3)$

$w^{[3]} = (4,5)$
$w^{[4]} = (2,4)$
$w^{[5]} = (1,2)$

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

⇒ When using back-propogat^n dim w = dim $dw = (n^{[l]}, n^{[l-1]})$

⇒ When using back-propogat^n dim b = dim $db = (n^{[l]}, 1)$

⇒ Vectorized : $Z^{[l]} = W^{[l]} X + b^{[l]}$

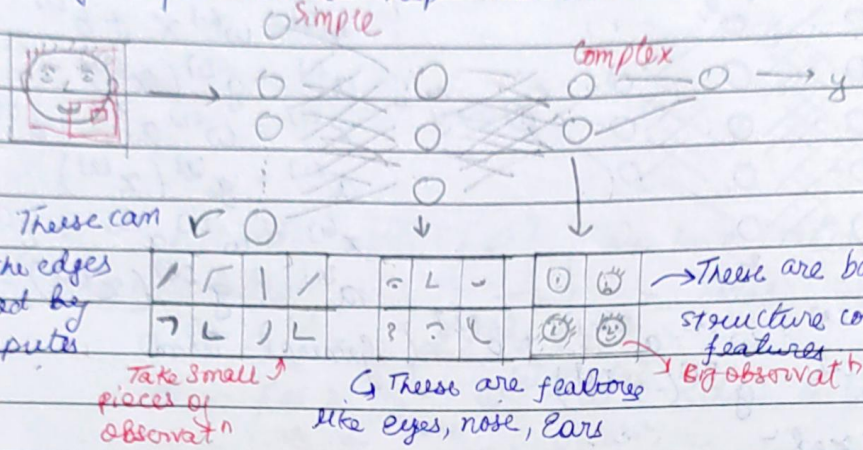
$[z^{[1](1)} z^{[1](2)} \dots]$

$(n^{[l]}, m)$ $(n^{[l]}, n^{[l-1]})$ $(n^{[0]}, m)$ $(n^{[l]}, 1)$ $\xrightarrow{\text{Broad cast}}$ $(n^{[l]}, m)$

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

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

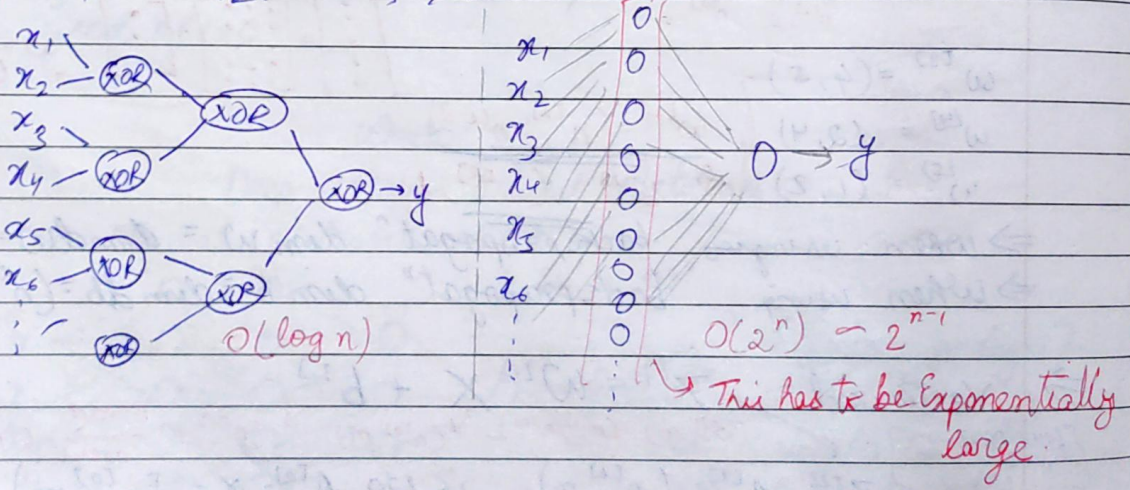
⇒ Why deep-neural Representations :-



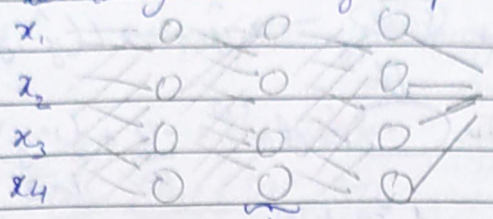
Ex for Audio detection :-
first layer might catch audability of sound - high or low
second can be phonems :- cat
Later may be learn to recognize word & Audio
finally recognise sentences & sounds / phrases

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

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

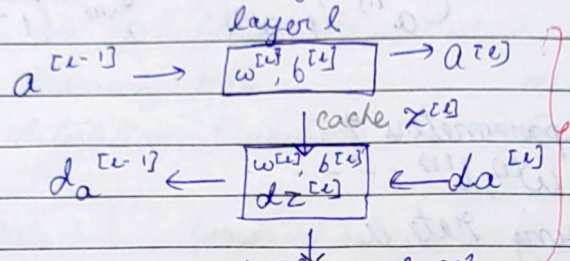


⇒ Building blocks of Deep Neural Network :

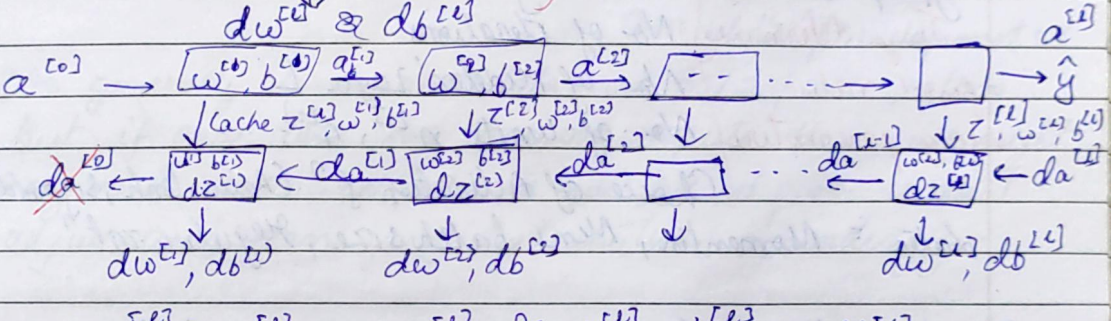


Layer l : $w^{[l]}, b^{[l]}$
↳ Forward : Input : $a^{[l-1]}$
Output : $a^{[l]}$
$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$ Cache $z^{[l]}$
$a^{[l]} = g^{[l]}(z^{[l]})$

↳ Backward propogat^n Input : $da^{[l]}$ & Output : $da^{[l-1]}, dw^{[l]}, db^{[l]}$



⇒



$$w^{[l]} = w^{[l]} - \alpha dw^{[l]} \quad \& \quad b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

⇒ Forward & Backward propogation :

i) Forward propogation for layer l
↳ Input $a^{[l-1]}$ → $w^{[l]}, b^{[l]}$
↳ Output $a^{[l]}$, cache $(z^{[l]})$

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

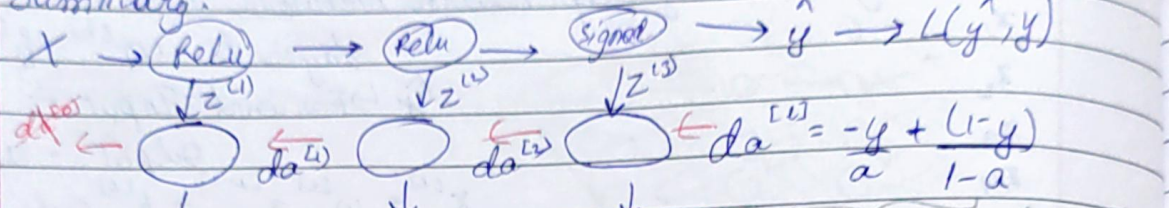
Vectorized
$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
$A^{[l]} = g^{[l]}(Z^{[l]})$

ii) Backward propogat^n for layer l :
↳ Input : $da^{[l]}$ & Output $da^{[l-1]}, dw^{[l]}, db^{[l]}$

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

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

⇒ Summary:



From Logistic Regression

We can also write: $dA^{[2]} = \left(\frac{-y^{(1)}}{a^{(1)}} + \frac{(1-y^{(1)})}{(1-a^{(1)})} \cdots \frac{y^{(m)}}{a^{(m)}} + \frac{(1-y^{(m)})}{(1-a^{(m)})}\right)$

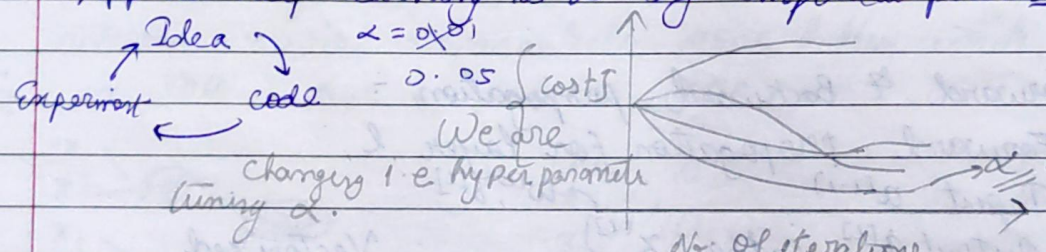
⇒ Parameters V/s Hyperparameters ?:

Parameter : $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \ldots$

Hyperparameters : (They determine final value using parameters)
- learning rate α
- No of iterations
- No. of hidden layer L
- No. of units $n^{[1]}, n^{[2]} \ldots$
- Choice of Activation f^n (Relu, tanh, sigmoid)

Later : Momentum, Mini batch size, Regularization ...

⇒ Applied deep learning is a very empirical process :



Ex Deep learning used in Vision, Speech, NLP, Ad, Search, Recommendation.

DEEP LEARNING : Course - 2

Improving Deep Neural Networks : Hyperparameter Tuning, Regularization and Optimization

Week - 1

★ Train / dev / test sets :

Applied ML is a highly iterative process.

↳ No. of layers
↳ No. of hidden units
↳ Learning rate
↳ Activation function
⋮

Idea → Code → Experiment

NLP, Vision, Speech, Ads, Search, Security, Logistics ...

Data :	Training Set	Dev	Test

Cross validation / Development set

We generally divide our data in 60/20/20% ratios. But if our data has 1,000,000 inputs then we might only need 10,000 inputs as our dev set. So u must adjust and plan your distribution accordingly.

⇒ Mismatched train / test distribution :

Training set : Cat pictures from webpage

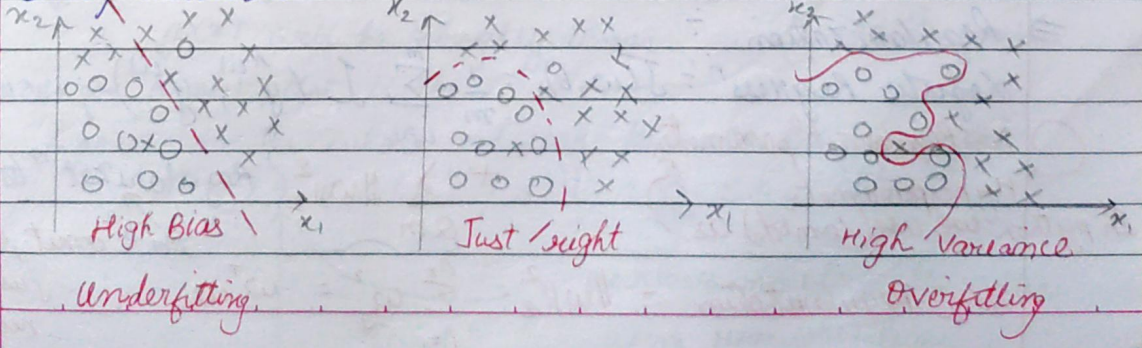
Dev / test sets : Cat pictures from users using your app

↳ Make sure dev & test come from same distribut^n.

Not having a test set might be okay (only dev set)

Train / dev → test

⇒ Bias / Variance :



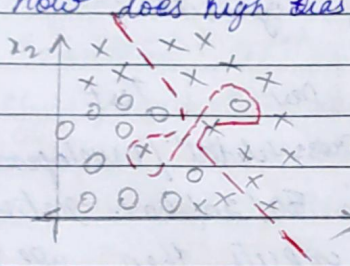
⇒ Bias and Variance :-

Cat Classification $y = 1/0$

If train set error has	1%	15%	15%	0.5%
Dev set error has	11%	16%	30%	1%
	High Variance	High Bias	High Bias & Variance	Low Bias & Low Variance

Acc to humans we can easily classify it as cat or non-cat with 0% error. So in comparisⁿ 15% & 16% gives us high bias and vice versa.

⇒ How does high bias & high variance graph look like :-



High Bias & High Variance.

In high Dimensional Data there can be some

⇒ Basic "recipe" for machine learning :-

High Bias? (Training Data performance) → Bigger Network (Train longer) [Diff NN architecture]

↓

High Variance? (Dev Set performance) → More Data / Regularization (NN architecture search)

↳ In early day we didn't had tools to improve any of bias or variance but with the help of deep learning we can individually improve bias or variance.

⇒ Regularization :-

Logistic Regressⁿ : $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$; $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

$+ \frac{\lambda}{2m}\|w\|_2^2$ } Regularizatⁿ term ; $+\frac{\lambda}{2m}b^2$ omit, as b is just one number

$\lambda \Rightarrow$ Regularizatⁿ parameter ↳ Hyperparameter

In python we use (lambd) as

L2 Regularization : $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

↳ regularization $\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w| = \frac{\lambda}{2m}\|w\|_1$

↳ Not used much as w will be sparse and it will compress our data as it has a lot of zeros

⇒ Neural Network :

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

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{ij}^{[l]})^2 \quad \text{as } w : (n^{[l-1]}, n^{[l]})$$

↳ Frobenius Norm $\|\cdot\|_2^2 \rightarrow \|\cdot\|_F^2$

$dw = (\text{from backpropogation}) + \frac{\lambda}{m}w^{[l]}$ $\quad \frac{\partial J}{\partial w^{[l]}} = dw^{[l]}$

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

↳ L_2 regularization is also called as weight decay.

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

$w^{[l]} := w^{[l]} - \alpha\frac{\lambda}{m}w^{[l]} - \alpha\,(\text{from backprop})$

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

we are subtracting $\frac{\alpha\lambda}{m}$ from $w^{[l]}$ so it got its name as weight decay.

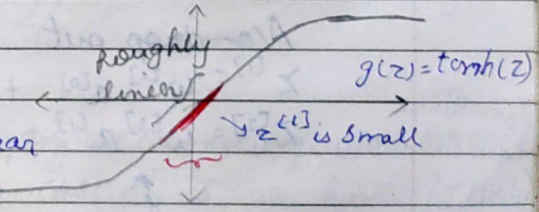
⇒ How does regularization prevent overfitting :

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

If $\lambda \uparrow$ then $w^{[l]} \downarrow$

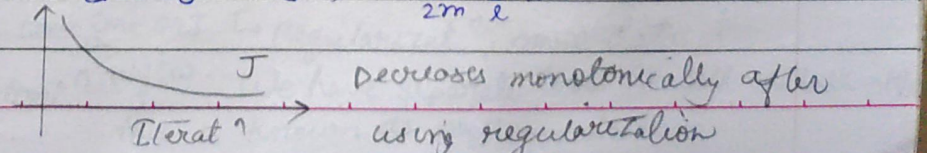
$\downarrow z^{[l]} \downarrow = w^{[l]} \cdot a^{[l-1]} + b^{[l]}$

$\therefore g(z)$ will be roughly linear



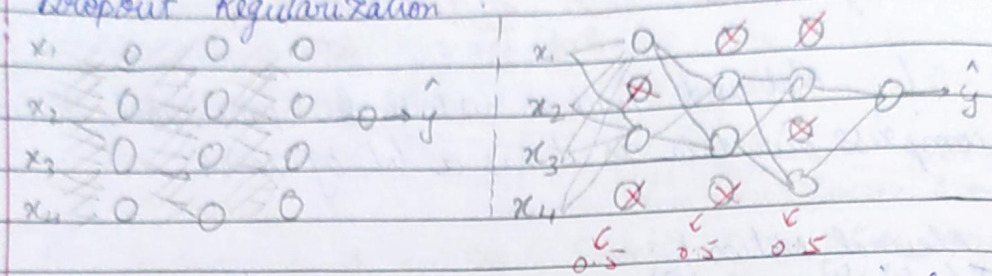
As we know, that if the f^n is linear so the whole network becomes linear

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



Page No.
Date 16 | 5 | 21

⇒ Dropout Regularization :



We drop say (0.5) no. of nodes from a given layer and train a comparative small Network.

⇒ Implementing dropout ("Inverted dropout")

Illustrate with layer $l = 3$

$d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep\text{-}prob$

Say $keep\text{-}prob = 0.8$; 0.2 chance of eliminating any hidden unit.

$a3 = np.multiply(a3, d3)$ i.e $a3 *= d3$ [element wise]

Inverted dropout ensure Expected value Of a_3 remains the same

$a3 \;/= keep\text{-}prob$

Say we have 50 units ∴ ~ 10 units shutt off i.e they are zero

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

↳ Reduced by 20% ; 20% of $a^{[3]} = 0$

then we $/= 0.8$ to not change value of $z^{[4]}$

it will bump back up roughly by 20% that we need

⇒ Making prediction at test time :

$a^{[0]} = X$

No drop out

$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$; $a^{[1]} = g^{[0]}(z^{[1]})$

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

$\vdots$

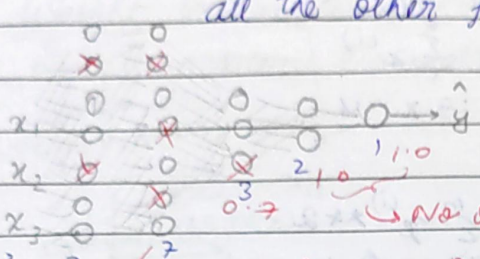
$\hat{y}$

⇒ Understanding dropout : Why does drop-out work?

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

Shrinking squared norm of the weights

Due to drop out we don't give any particular feature emphasis, we distribute it amongst all the other features



for $w^{[1]} = (3,7)$; $w^{[2]} = (7,7)$

$w^{[3]} = (7,3)$; $w^{[4]} = (3,2)$ ...

$\therefore w^{[2]}$ is biggest weight matrix

↳ No overfitting and no dropout

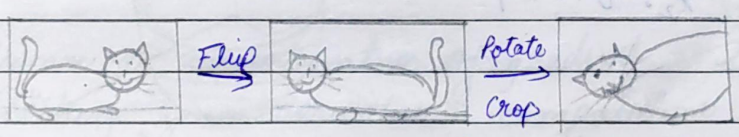
↳ We will keep keep-prob relatively low say 0.5

In computer-vision we often use keep-prob.

Downside of dropout = $J()$ is not well defined in every iteration we are randomly calling out bunch of nodes.

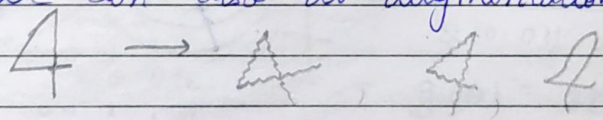
⇒ Other regularization methods :

Data augmentation :



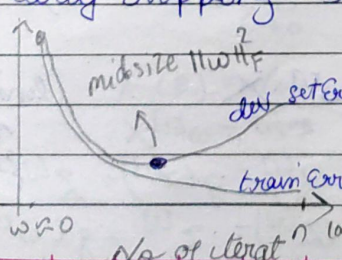
using them we can add more data to our training set in an inexpensive way and reduce overfitting

We can also do augmentations on digits :



These are also digit forms.

⇒ Early Stopping : Dues to Early stopping we can no longe work ① & ② simultaneously ✗



① ⇒ Optimize cost f^n J we have $\{J(w,b)$

↳ G.D, Momentum, Adam, RMSprop $\}$

② ⇒ Not overfit :

↳ Regularizat^n, more data $\}$

We have separate tools for both the task in ML this is known as orthogonalisation.

So instead of early stopping we can use L_2 regularization but for that we have to try a lot of different values of λ. which can be computationally expensive

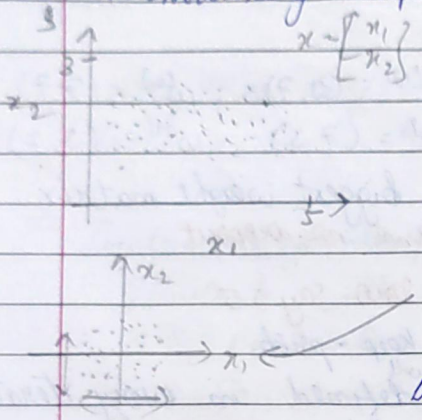
⇒ Normalizing inputs :

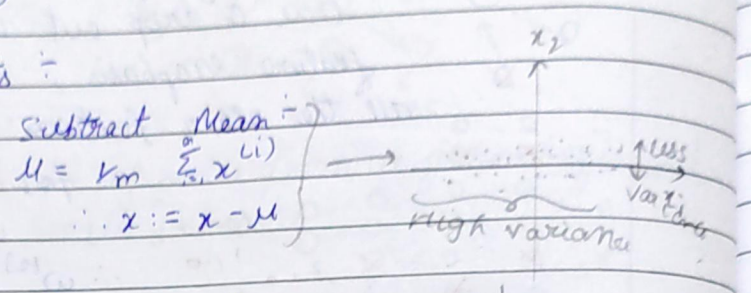
$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

Subtract Mean :-

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

$\therefore x := x - \mu$





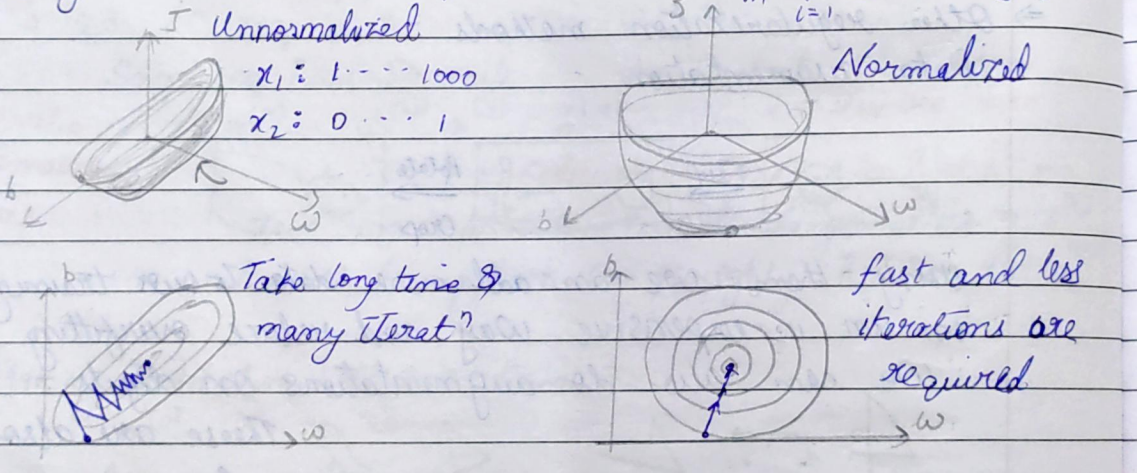
$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} ** 2$ ← element wise

$x /= \sigma^2$

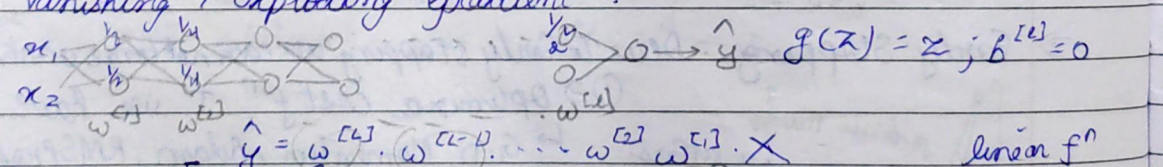
Normalize variance

Use same σ^2 & μ to normalize test set.

⇒ Why to normalize inputs ? $J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$



⇒ Vanishing / Exploding gradient :



$\hat{y} = w^{[L]} \cdot w^{[L-1]} \cdot \cdots \cdot w^{[2]} w^{[1]} \cdot X$

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

linear fn

Say $w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$

$\hat{y} = w^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X$

$1.5^{L-1} X$

$z^{[1]} = w^{[1]} X$

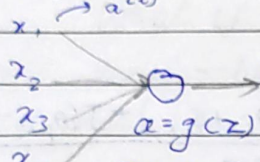
$a^{[1]} = g(z^{[1]}) = z^{[1]}$

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

If we take 0.5 instead of 1.5
we have $0.5^L x$

for a deep Network: $\begin{cases} w^{[l]} \geq 1 ; & \text{activat}^n \text{ can explode} \\ w^{[l]} \leq 1 ; & \text{activation will decrease exponential} \end{cases}$

⇒ Weight initialization for deep networks:
Single neuron example



$z = w_1 x_1 + w_2 x_2 \cdots w_n x_n + b^0$

Larger $n \rightarrow$ Smaller w_i

Variance $(w_i) = 1/n$

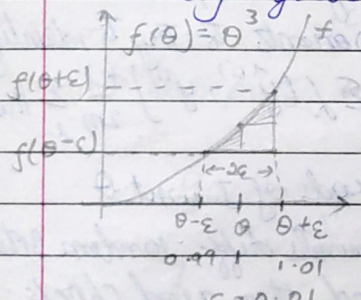
$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

↳ For ReLu activat^n (g) we see that:
We take variance as $2/n$
This is according to working $g^{[l]}(z) = ReLu(z)$

↳ for Tanh we use $1/n$

Xavier initializat^n ⇒ $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

⇒ Numerical approximation of gradient:
Checking your derivative computation



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

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

$g(\theta) = 3\theta^2 = 3$ at $\theta = 1$

∴ Approx Error $= 0.0001$

$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ → $O(\varepsilon^2)$, 0.0001 } More Accurate

$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$ → $O(\varepsilon)$, 0.01 } Less Accurate

Page No.
Date

⇒ Gradient Checking :- For a neural network

↳ Take $w^{[1]}, b^{[1]} \cdots w^{[L]}, b^{[L]}$ and reshape into a big vector θ and then concatenate

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

↳ Take $dw^{[1]}, db^{[1]}, \cdots dw^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$ and then concatenate

Is $d\theta$ the gradient of $J(\theta)$

$J(\theta) = J(\theta_1, \theta_2 \cdots)$

for each i :

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

$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$ | $d\theta_{approx} \approx d\theta$ same dimensions

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$; $\varepsilon = 10^{-7}$

↳ If this value $\approx 10^{-7}$ or smaller : great.

10^{-5} → OK OK ; 10^{-3} → Wrong

⇒ Gradient checking for implementing notes :-

↳ Don't use in training - only to debug [$d\theta_{approx}[i]$ $d\theta$]

↳ If algo fails grad check, look at components to try to identify bug.

↳ Remember regularization $J(\theta) = \frac{1}{m} \sum \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_l \|w^{[l]}\|_F^2$

$d\theta$ = grad of J wrt θ

↳ Doesn't work with dropout. [eliminates diff random sets]

or else keep-probs = 1.0 and then grad check

↳ Run at random initialization ; perhaps again after some training. $w, b \approx 0$

Page No. 128
Date 18 | 5 | 21

Week - 2

* Optimization Algorithms :-

⇒ Mini - batch gradient Descent :-

Batch vs mini batch gradient descent.

Vectorization allows you to efficiently compute on m examples.

$X = [x^{(1)}\ x^{(2)}\ \cdots\ x^{(m)}] \rightarrow (n_x, m)$

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

What if $m = 5,000,000$?

mini - batches of 1,000 each $[x^{(1)} \cdots x^{(1000)} \mid x^{(1001)} \cdots x^{(2000)} \mid \cdots x^{(m)}]$

$X^{\{1\}}$ $X^{\{2\}}$ $X^{\{5000\}}$

Same for y

Mini-batch t : $X^{\{t\}}, Y^{\{t\}}$

$X^{\{t\}}, Y^{\{t\}}$ for mini batches

$X^{\{t\}} \rightarrow (n_x, 1000)$ & $Y^{\{t\}} \rightarrow (1, 1000)$

⇒ Mini - batch GD :-

for $t = 1, \cdots, 5000$ → 1 Step of GD using $X^{\{t\}}$ & $Y^{\{t\}}$ (as if $m = 1000$)

Forward prop on $X^{\{t\}}$ — Vectorized implementation (1000 examples)

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

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

$A^{[L]} = g^{[L]}(Z^{[L]})$ → for $X^{\{t\}}, Y^{\{t\}}$

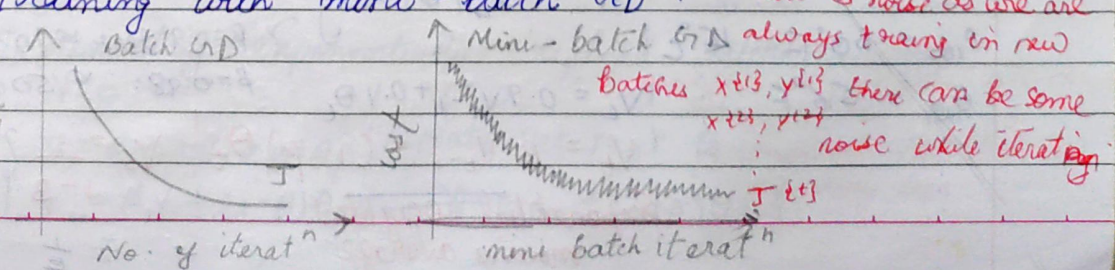
Compute cost $J = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \times 1000} \sum_{l} \|w^{[l]}\|_F^2$

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

$w^{[l]} := w^{[l]} - \alpha dw^{[l]}$; $b^{[l]} := b^{[l]} - \alpha db^{[l]}$

This is also called as "1 epoch" one pass through the training set

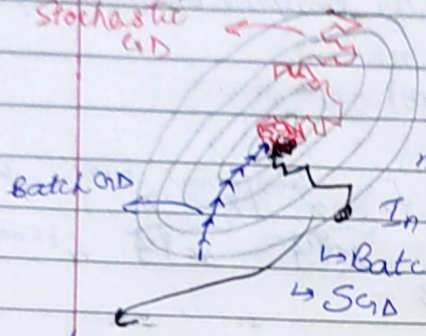
⇒ Training with mini - batch GD :- There is noise as we are always training in new batches $X^{\{t\}}, Y^{\{t\}}$ there can be some noise while iterating.



⇒ Choosing your mini-batch size:

↳ If mini-batch size = m : Batch GD $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

↳ If mini-batch size = 1 : Stochastic GD. Every example is its own mini batch $(X^{\{1\}}, Y^{\{1\}}) = (X^{(1)}, Y^{(1)}) \ldots$



SGD wont even converge always oscillate and wander around the region around the minimum.

In practice mini-batch is in b/w 1 & m

↳ Batch GD → mini-batch size = m Too long per iteration

↳ SGD → mini-batch size = 1 Loose speedup from vectorization

↳ In between (mini-batch size (not too small not too big)

This gives us fastest learning

i) we do get vectorization

ii) Make progress without preceding entire train set.

⇒ Choosing your mini-batch size

If small training set : Use batch GD ($m \leq 2000$)

Typical mini-batch size:

64, 128, 256, 512 i.e power of 2

Make sure mini-batch fits in CPU/GPU memory $X^{\{t\}}, Y^{\{t\}}$

⇒ Exponentially weighted averages:

Temperature in London

$\theta_1 = 40°F$ $4°C$

$\theta_2 = 49°F$ $9°C$

$\theta_3 = 45°F$:

:

$\theta_{180} = 60°F$ $15°C$

$\theta_{181} = 56°F$:

:

Now; $V_0 = 0$

$V_1 = 0.9 V_0 + 0.1 \theta_1$

$V_2 = 0.9 V_1 + 0.1 \theta_2$

$V_3 = 0.9 V_2 + 0.1 \theta_3$

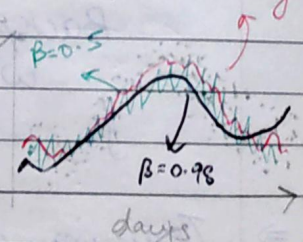
:

$V_t = 0.9 V_{t-1} + 0.1 \theta_t$

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

Exponentially weighted moving average

graph for 0.9 weighted average



$\beta = 0.9$ here ≈ 10 days temp

$\beta = 0.98$: ≈ 50 days temp

$V_t \approx \frac{1}{1-\beta}$ for $\beta = 0.9$, $\beta = 0.98$

$= \frac{1}{e}$

V_t as approximately average over $\approx \frac{1}{1-\beta}$ days temperature.

For $\beta = 0.98$ we are getting a smoother curve as we taking average of last 50 terms
for $\beta = 0.5$: $\approx$ 2 days

⇒ Understanding Exponentially Weighted averages :

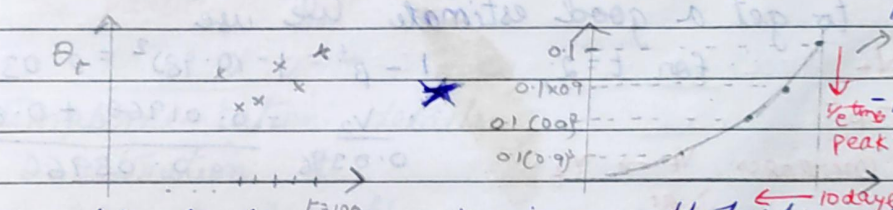
$V_t = \beta V_{t-1} + (1-\beta)\theta_t$

$V_{100} = 0.9 V_{99} + 0.1 \theta_{100}$
$V_{99} = 0.9 V_{98} + 0.1 \theta_{99}$
$V_{98} = 0.9 V_{97} + 0.1 \theta_{98}$
...

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

$\therefore V_{100} = 0.1\theta_{100} + 0.1 \times 0.9 \theta_{99} + 0.1 \times (0.9)^2 \theta_{98} + 0.1(0.9)^3 \theta_{97} \cdots$

So what we are actually doing here



We do the element wise multiplication of the two graph points

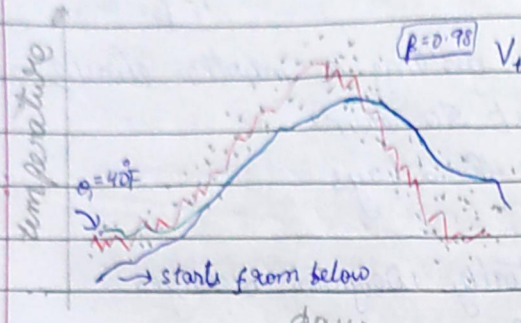
Some algebric & calculus insights $0.9^{10} \approx 0.35 \approx 1/e$
We know $(1-\epsilon)^{1/\epsilon} = 1/e$; for $\epsilon = 0.1$
So we can say that around 10 days the peak will decay be $1/e$ times.
For different β we have diff parameters
Ex. $\beta = 0.98$; $0.98^{50} \approx 1/e$; here it will take 50 days.
$\epsilon = 1 - \beta$

⇒ Implementing exponentially weighted averages :

$V = 0$
$V_i := \beta V + (1-\beta)\theta_1$
$V := \beta V + (1-\beta)\theta_2$
:

$V_\theta = 0$ ← only 1 variable we just substitute the other $V_1, V_2, V_3 \cdots$ So memory allocation is very less
Repeat { get next θ_t
$V_\theta := \beta V_\theta + (1-\beta)\theta_t$ }

⇒ Bias correction in exponentially weighted average :



$\beta = 0.98$ $V_t = \beta V_{t-1} + (1-\beta)\theta_t$

$V_0 = 0$

$V_1 = 0.98\, V_0 + 0.02\theta_1$ — ①

if $\theta_1 = 40°F$ ∴ Using ① $V_1 = 8°F$ as $V_0 = 0$ so we get a lower value.

On using Bias correction we get this blue curve and it starts from below ↙

$V_2 = 0.98\, V_1 + 0.02\theta_2$

$V_2 = 0.98 \times 0.02 \times \theta_1 + 0.02\theta_2$

$V_2 = 0.0196\theta_1 + 0.02\theta_2$ [∵ V_2 is also small]

It also isn't a very good estimate.

So to get a good estimate we use

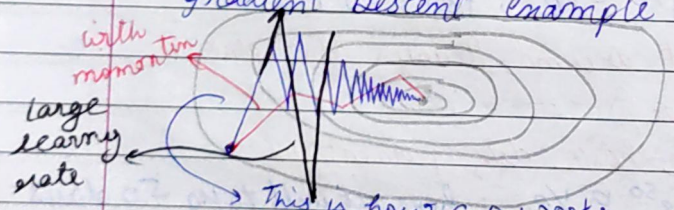
$\frac{V_t}{1-\beta^t}$ ∴ for $t = 2$, $1 - \beta^t = 1 - (0.98)^2 = 0.0396$

∴ $\frac{V_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

As t increases $\frac{V_t}{1-\beta^t} \to V_t$

⇒ Gradient Descent with momentum :

Gradient Descent example



we must keep small learning rate or else we might overbound.

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

What we want is that ?

↕ slow learning ↔ fast learning

So we have

Momentum :

On iteration t :

Compute dw, db on current mini batch

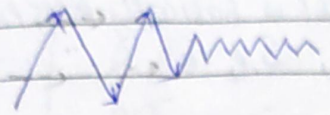
$V_{dw} = \beta V_{dw} + (1-\beta)\, dw$

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

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

$\beta := \beta - \alpha V_{db}$

With GD we had so on taking average for vertical we have (+)&(-) direct" so they cancel each other and we will get slow learning in ↕ vertical direct" and that is what we want



⇒ for horizontal we have all vectors in same direct" this will increase its speed and will reach minimum faster than batch GD.

⇒ Ex for a hemispherical bowl countor plot

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

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

↳ velocity

strict"

This is a basic analogy in understanding the particular case

⇒ Implementation details :

On iteration t :

Compute dw, db on the current mini-batch

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

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

Some cases we might see $V_{dw} = \beta V_{dw} + dw$ but its wrong

$(1-\beta)$ stabilises our outcome

$w = w - \alpha V_{dw}$; $b = b - \alpha V_{db}$

Hyperparameter : α, β $\beta = 0.9$ → avg over last ≈ 10 gradients

People dont often use bias correct" i.e $\frac{V_t}{1-\beta^t}$ as after few iterat" the GD becomes normal

∴ we can also use large learning rate in RMS prop

⇒ RMSProp :

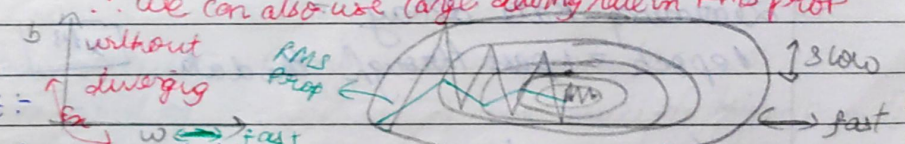
On iterat" t :

Compute dw, db on mini batch

$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$ → small → element wise

$S_{db} = \beta S_{db} + (1-\beta) db^2$ → large

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



∴ dw^2 will be small ∴ S_{dw} ↕

↑ $w = w - \alpha \frac{dw}{\sqrt{S_{dw}}}$ ↕

↓ $b = b - \alpha \frac{db}{\sqrt{S_{db}}}$ ↑

$\varepsilon \approx 10^{-8}$ so that if $S_{dw} = 0$ ∴ we dont get error

Page No.
Date

⇒ Adam optimization Algorithm : It is basically RMSProp + Momentum

$V_{dw} = 0$; $S_{dw} = 0$, $V_{db} = 0$, $S_{db} = 0$

On iteration t:

Compute dw, db using current mini-batch

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

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

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

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

$$V_{dw}^{corrected} = \frac{V_{dw}}{1-\beta_1^t} \quad ; \quad V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$$

$$S_{dw}^{correct} = \frac{S_{dw}}{1-\beta_2^t} \quad ; \quad S_{dw}^{correct} = \frac{S_{dw}}{1-\beta_2^t}$$

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \quad \& \quad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

⇒ Hyperparameters choice :

α : needs to be tune

β_1 : 0.9 (dw)

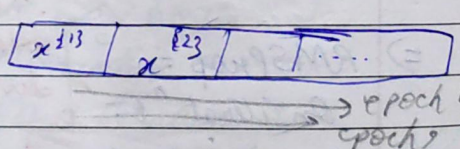
β_2 : 0.999 (dw^2)

ε : 10^{-8}

Adam : Adaptive moment estimation.

⇒ Learning Rate decay :

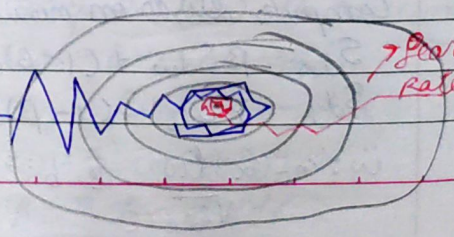
1 epoch = 1 pass through the data



$$\alpha = \frac{1}{1 + \text{decayrate} * \text{epochnum}} \times \alpha_0$$

with this decay we can closly approx to our minimum & get good result

Big learning rate converging is not close to minimum



Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮

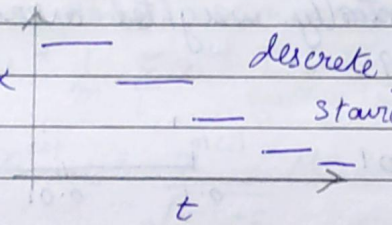
for $\alpha_0 = 0.2$
decay rate = 1
α decreases

⇒ Other learning rate decay methods :-

different formulas

$\alpha = 0.95^{\text{epoch num}} \alpha_0$ ← Exponential decay
↳ It can be any number less than 1

$\alpha = \frac{k}{\sqrt{\text{epoch num}}} \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$



discrete staircase or By manually decreasing α

Week - 3

* Hyperparameter tuning

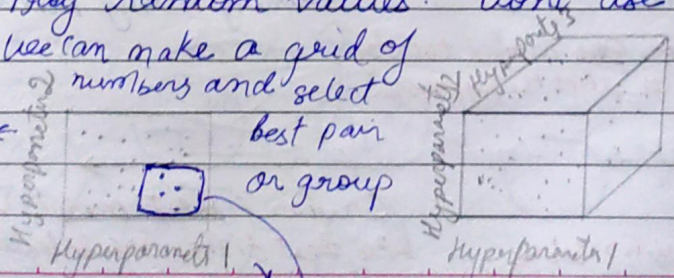
⇒ Tuning process :-

We have hyperparameters like α, β, β_1, β_2, ε, layers, hidden units, mini-batch size, learning rate decay

○ → most important
○ → they are important too

⇒ Try random values :- Don't use a grid

We can make a grid of numbers and select best pair or group



For multiple hyperparameters we can use 3-D graph and select the best fit.

↳ Here we now take smaller square and sample more points.

⇒ Using an appropriate scale to pick Hyperparameters :

Picking Hyperparamters at random :

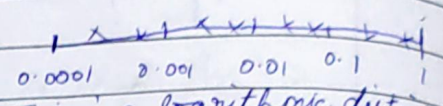
$n^{[l]}$ = 50, ... 100

No. of layers L : 2 to 4 or 2, 3, 4

Appropriate scale for hyperparameters :

α = 0.0001, ... ; 1

we can randomly search from



This is a logarithmic distribution

∴ Python code :

$r = -4 * np.random.rand()$ ∈ $r \in [-4, 0]$

$\alpha = 10^r$ ∈ $\alpha \in [10^{-4}, 1]$

∴ $a = \log_{10} 0.0001 = -4$

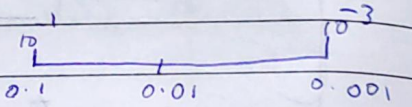
$b = \log_{10} 1 = 0$ ∴ $r \in [a, b]$ i.e $[-4, 0]$

⇒ Hyperparameter for exponentially weighted averages :

β = 0.9, ... 0.999

↳ last 10 avg ↳ last 1000 avg

$1 - \beta$ = 0.1, ... 0.001



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

$1 - \beta = 10^r$

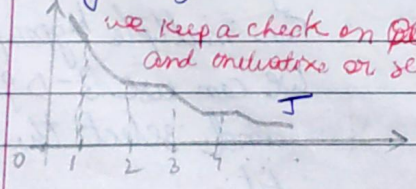
∴ $\beta = 1 - 10^r$

∴ We must use a log scale not a linear scale.

⇒ Hyperparameter tuning in practice : Pandas V/s Caviar.

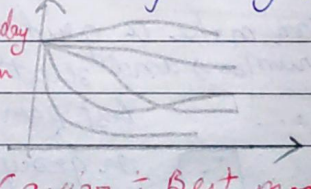
We keep on changing hyperparameters as intuitions do get stale and re-evaluate your model occassionally

Babysitting one model



Panda : only have 1-2 babies and it takes care of it

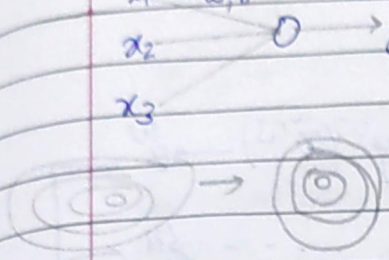
Training many models in parallel



Caviar : Best model is selected like best fishes are selected as there are millions of eggs.

Batch normalization :

Normalizing activations in a network :
Normalizing inputs to speed up learning :



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

$$X := X - \mu$$

$$\sigma^2 = \frac{1}{m} \sum_i x^{(i)2} \rightarrow \text{element wise}$$

$$X := X / \sigma^2$$

Can we normalize $a^{[2]}$ so as to train $w^{[3]}, b^{[3]}$ faster
Normalize $z^{[2]}$

Implementing Batch Norm :

Given some intermediate values in Neural Net $(z^{(1)} \cdots z^{(m)})$

$\therefore \mu = \frac{1}{m} \sum_i z^{(i)}$; $\sigma^2 = \frac{1}{m} \sum_i (z_i - \mu)^2$ $\quad z^{[l](i)} \leftarrow$

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

If $\gamma = \sqrt{\sigma^2 + \varepsilon}$ & $\beta = \mu$ then $\tilde{z}^{(i)} = z^{(i)}$ } If for diff components

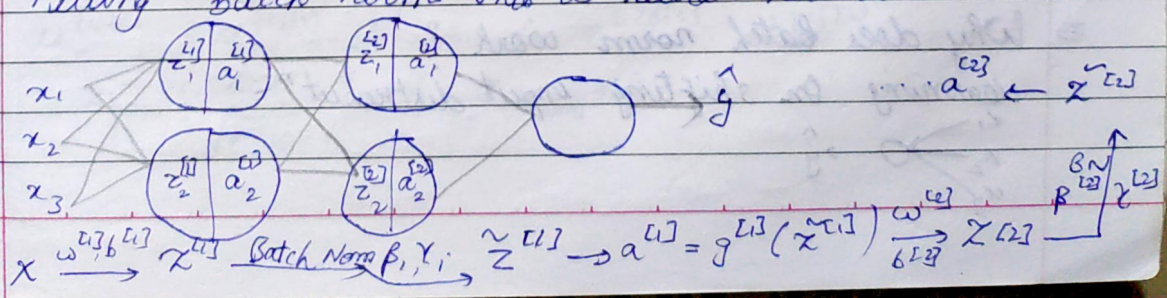
$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$ so that we don't always get variance = 1 & mean = 0

learnable params of model

So we can $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$

Because we use $\sigma^2 = 1$ & $\mu = 0$ for our data
So our data will be on a linear fit
so we must choose different values of σ^2 & μ to adjust our model.

Fitting batch norm into a neural network :



Page No.
Date 21 | 5 | 21

Parameters : $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \ldots w^{[L]}, b^{[L]}$ $\} db^{[l]}$
$\beta^{[1]}, \gamma^{[1]}, \ldots \ldots \beta^{[L]}, \gamma^{[L]}$ $\} \beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$

This β & Adam wala β is different.

We can use batch normalizat^n with one line of code
i.e tf.nn.batch-normalization.

⇒ Working with mini-batches :

$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\beta^{[1]}, \gamma^{[1]}]{BN} \tilde{z}^{[1]} \rightarrow g^{[1]}(\tilde{z}^{[1]}) \rightarrow a^{[1]} \ldots\ldots z^{[2]}$

$X^{\{2\}} \rightarrow z^{[1]} \rightarrow \tilde{z}^{[1]} \ldots$

$\vdots$

Parameters : $w^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$; $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

$\beta^{[l]}$ gets eliminated coz we do normalizat^n on $z^{[l]}$ take its mean so $b^{[l]}$ doesn't have significant role. So we take $b^{[l]} = 0$ $\quad z^{[l]}_{norm}$; $\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$

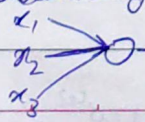
⇒ Implementing gradient descent :

for $t = 1 : \ldots$ num Mini batch
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 $dw^{[l]}, db^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$
Update parameters : $w^{[l]} := w^{[l]} - \alpha dw^{[l]}$
$\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$
$\gamma^{[l]} := \gamma^{[l]} - \alpha d\gamma^{[l]}$

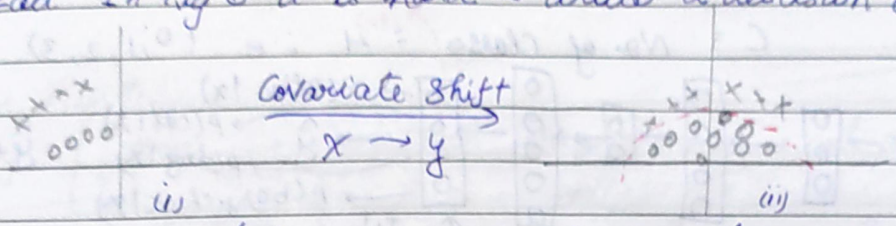
Works with momentum ; RMSprop, Adam.

⇒ Why does batch norm work ?

Learning on shifting input distribut^n :



When we classify and train our model to identify cat and non-cat; what if we only provide computer with black cat images and then ask for classification. Our model will fail. In Fig (i) it is hard to draw a decision boundary



So what does batch norm do; when operating with different batches we might get different results but mean and variance will always be the same on plotting it on graph.

⇒ Batch norm as regularization :

↳ Each mini-batch is scaled by the mean/variance computed on just that mini-batch.

↳ This adds some noise to the values $z^{[l]}$ within that mini batch. So similar to dropout; it adds some noise to each hidden layer's activations.

↳ This has a slight regularization effect.

⇒ Batch norm at test time :

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

m is no. of example in mini-batch

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \quad ; \quad \tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta$$

μ, σ^2 : estimate using exponential weighted average (across mini-batch)

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

↓ ↓

$\mu^{\{1\}[l]}$ $\mu^{\{2\}[l]}$ & Same for $\sigma^{2\{1\}[l]}$, $\sigma^{2\{2\}[l]}$ ...

$$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}} \quad ; \quad \tilde{z} = \gamma z_{norm} + \beta$$

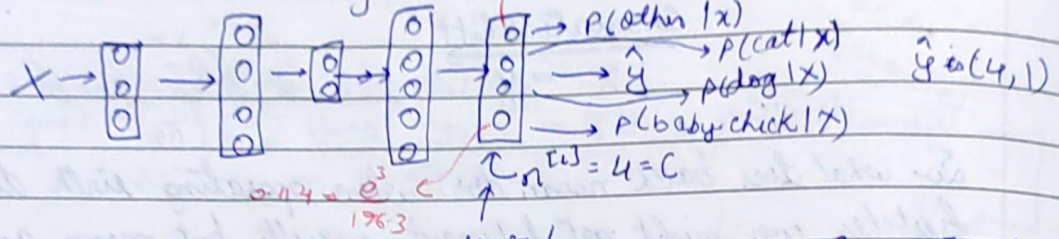
Page No.
Date

Multi Class classification :

Softmax Regression :

Recognizing cats, dogs and baby chicks ; other
1 2 3 0

C = No. of classes = 4 i.e (0, 1, 2, 3)



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

Activation function : Softmax fⁿ

$t = e^{(z^{[L]})}$ (4,1)

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

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

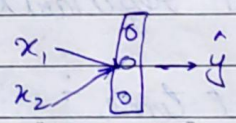
$\sum_{i=1}^{4} t_i = 176.3$

$a^{[L]} = \frac{t}{176.3}$

$a^{[L]} = \begin{bmatrix} 0.84 \\ e^2/176.3 \\ e^{-1}/176.3 \\ 0.114 \end{bmatrix}$ (4,1)

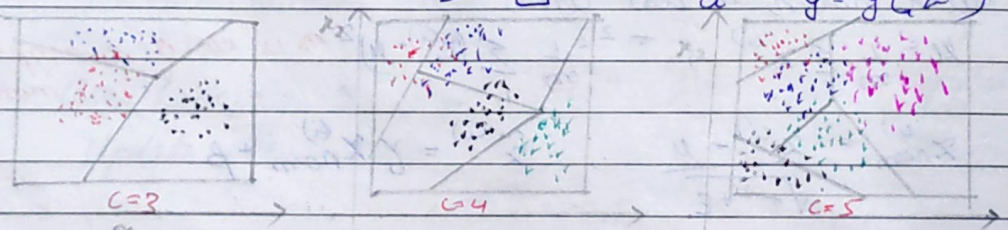
$\therefore a^{[L]} = g^{[L]}(z^{[L]})$ → (4,1)

⇒ Softmax Example :



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

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



⇒ Training a Softmax classifier : C=4

hard max

$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$; $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$; $g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/(e^5+e^2+e^{-1}+e^3) \\ e^{-1}/(e^5+e^2+e^{-1}+e^3) \\ e^3/(e^5+e^2+e^{-1}+e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$ $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

$g^{[L]}(\cdot)$ → Softmax Activation fⁿ

Hardmax

⇒ Softmax regression generalizes logistic Regression to C classes
If C = 2, softmax reduces to logistic Regression.

$a^{[4]} = \begin{bmatrix} 0.842 \\ 0.158 \end{bmatrix}$ } Single output

⇒ Loss Function :

$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ cat $\quad \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} = a^{[L]}$ So our model is not performing well.

$y_1 = y_3 = y_4 = 0$, $y_2 = 1$

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j \Rightarrow -y_2 \log \hat{y}_2 = -\log \hat{y}_2$$

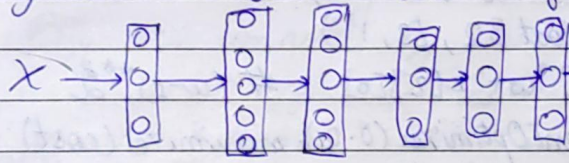
∴ To reduce cost, make $\hat{y}_2$ as big as possible.

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

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & \cdots \\ 1 & 0 & \cdots & \cdots \\ 0 & 1 & \cdots & \cdots \\ 0 & 0 & \cdots & \cdots \end{bmatrix} \rightarrow (4, m)$$

$$\hat{Y} = \begin{bmatrix} \hat{y}^{(1)} & \cdots & \cdots & y^{(m)} \end{bmatrix} = \begin{bmatrix} 0.3 & \cdots & \cdots & \cdots \\ 0.2 & \cdots & \cdots & \cdots \\ 0.1 & \cdots & \cdots & \cdots \\ 0.4 & \cdots & \cdots & \cdots \end{bmatrix} \rightarrow (4, m)$$

⇒ Gradient Descent with softmax :

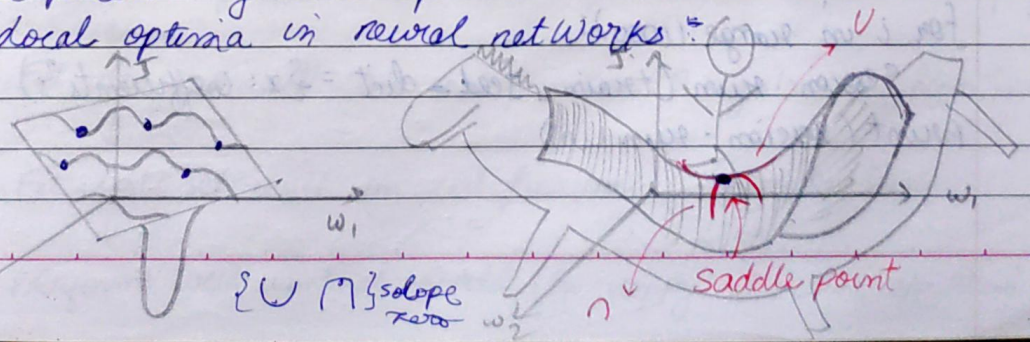


$(4,1) \rightarrow z^{[4]} \rightarrow a^{[4]} = \hat{y} \rightarrow \mathcal{L}(\hat{y}, y)$

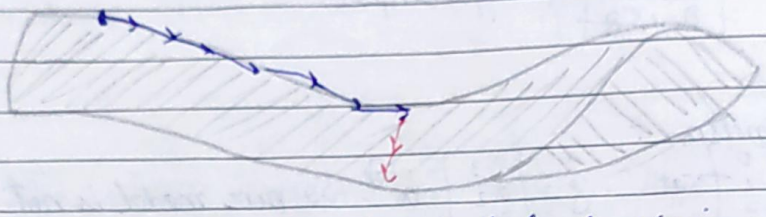
Back propogation : $dz^{[4]}_{(4,1)} = \hat{y} - y \rightarrow \frac{\partial J}{\partial z^{[4]}}$

⇒ The problem of local optima :

Local optima in neural networks :



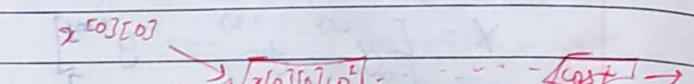
⇒ Problem of plateau :



↳ Unlikely to get stuck in a bad local optima
↳ Plateaus can make learning slow.

⇒ Tensorflow :
Motivating problem
$J_{cost}(w) = w^2 - 10w + 25 = (w-5)^2$ at $w=5$ minimum

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

Also written as

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

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

DEEP LEARNING : Course-3

Introduction to ML Strategy

Week-1

Structuring ML Projects

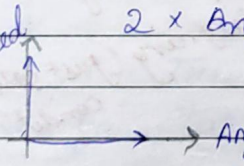
* Ideas :
  - Collect more data
  - Collect more diverse training set
  - Train algo longer with Gradient Descent
  - Try Adam instead of Gradient Descent
  - Try bigger network
  - Try smaller network
  - Try dropout
  - Add L_2 regularization
  - Network Architecture
    - Activation functions
    - No of hidden units
    - ....

⇒ Orthogonalization :

Car Tuning Example :

We have Steering, Acceleration, Breaking what if they are connected by this relation

$0.3 \times \text{Angle} - 0.8\,\text{Speed}$ } But instead we can have
$2 \times \text{Angle} + 0.9\,\text{Speed}$ } 2 different features
i.e Angle and Speed



⇒ Chain of Assumption in ML :

Fit training set well on cost function (≈ Human level performance)
↓
Fit dev set well on cost function ≈ Regularizat^n, Bigger Training Set
↓
Fit test set well on cost function ≈ Bigger Dev Set & cost function
↓
Performs well in real World (Ex : Happy cat picture app users)

⇒ Single number evaluation metric :

F_1 Score	Classifier	Precision (Of the examples recognised as cat, what % are actually cats?)	Recall (Of all the images really are cats, what % were correctly recognised by your classifier?)
92.4%	A	95%	90%
91.0%	B	98%	85%

F_1 score = $\frac{2PR}{P+R}$ Harmonic mean of P & R

Dev set + Single real number evaluat^n metric → Speeds up & Improve this iterative process.

⇒ Another example : What if we are training a lot of models on different parameters (Example of train error) :

Algorithm	US	China	India	Avergee
A	3%	7%	5%	5%
B	5%	6%	4%	5%
C	2%	3%	4%	3%
D	5%	8%	2%	5%

We can check the performance by taking average.

⇒ Satisficing and optmizing metrics :

Another cat classification example : → optimizing (Max)

Classifier	Accuracy	Running Time → Satisficing (just satisfy a condition)
A	90%	80ms
B	92%	95ms ≤ 100ms
C	95%	1500ms

Say cost = accuracy − 0.5 × Running time.

So we maximize accuracy & subject to running time ≤ 100ms

N metrics : 1 Optimizing
N − 1 Satisficing

⇒ Wake words / Trigger words :

Alexa, OK Google, Hey Siri, nihaobaidu (你好百度)

Accuracy & false positive : Maximize Accuracy such that ≤ 1 false pos in every 24 hours.

⇒ Train / dev / test distributions :

Cat classification dev / test sets

US, UK, Europe → Dev
India, China, Aus → Test

Wrong method

You must randomly shuffle all the data and then divide it into dev / test sets

If we chose this idea then we are particularly aiming at this target to gain accuracy But what if target is shifted from the original space our model will fail to obtain high level accuracy.

⇒ Guideline :

same distribution

chose a dev set and test set to reflect data you expect to get in the future and consider important To do well on

⇒ Size of dev / test sets :

Old way of splitting data :

70 % | 30 %

Train | Test

Good for 100, 1000, 10000 Examples

60 % | 20 % | 20 %

Train | Dev | Test

98 % | 1 % | 1 %

Train | Dev | Test

for 1,000,000 Examples

↳ Set your test set to big enough to give high confidence in the overall performance of your system

⇒ When to change Dev / Test sets and metrics :

Cat dataset Example :

Metric : Classification Error

× Algo A : 3 % Error → It shows some pornographic images

✓ Algo B : 5 % Error → Not showing any pornographic image

Metric + Dev → Prefers A || But You / users → Prefer B

Page No.
Date

Error $\Rightarrow \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} [\{ y_{pred}^{(i)} \neq y^{(i)} \}] \cdot w^{(i)}$

↳ predicted value (0/1)

$w^{(i)} = \begin{cases} 1 & \text{if } x^{(i)} \text{ is non-porn} \\ 100 & \text{if } x^{(i)} \text{ is porn} \end{cases}$

So now our algo on iterating through pornographic images it will have a huge error.

⇒ Orthogonalization for cat pictures : Anti-porn

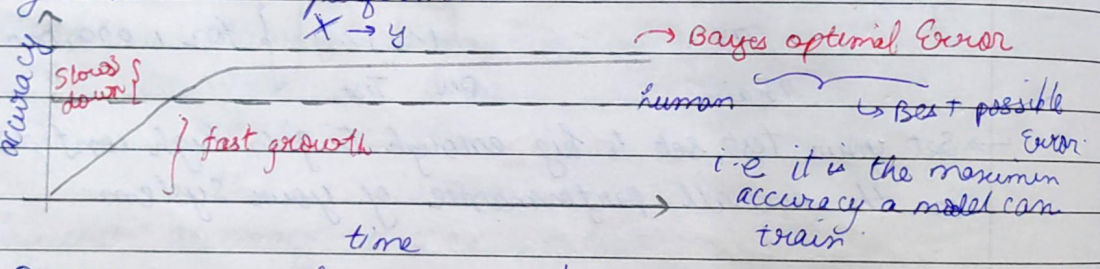
↳ So far we've only discussed how to define a metric to evaluate classifiers. ≈ Place the target

↳ Worry separately about how to do well on this metric.
≈ It is a separate prob how to shoot the target.

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

↳ If your algo/model is doing well on your metric + dev/test set does not correspond to doing well on your application, change your metric and/or dev/test set.
As user images can be blurry, rotated.

⇒ Why human level performance :



Or cat images which are very blurry.

Or there can be some example say (audio) which is very noisy and can't be transcripted.

↳ As humans are also good at classifying cat images or audio so the gap is very short.

⇒ Why compare to human level performance:
Humans are quite good at a lot of tasks. So as long as ML is worse than humans you can:
↳ Get labeled data from humans. (X, y)
↳ Gain insight from manual error analysis:
Why did a person get this right?
↳ Better analysis of bias / variance.

⇒ Avoidable Bias:
Cat classificat^n:

Bayes → Humans	1%	7.5%
Training Error	8%	8%
Dev Error	10%	10%
	Focus on Bias	Focus on Variance

0.5% ↕ Avoidable Bias, we can't go below Bayes error
2% ↕ Variance

Human level error as a proxy for Bayes Error

⇒ Understanding human-level performance:
Human level error as a proxy for Bayes Error
Medical image classification Example
Suppose:
a) Typical human ... 3% error
b) Typical doctor ... 1% Error
c) Experienced doctor ... 0.7% Error
d) Team of Exp. doc ... 0.5% Error
∴ What is human-level error?
Bayes error ≤ 0.5%

⇒ Error Analysis Example:

Human (proxy for Bayes Error)	1% / 0.7% / 0.5%	1% / 0.7% / 0.5%
↕ Avoidable bias	4-4.5%	0.1%
Training Error	5%	1%
↕ Variance	1%	4%
Dev. Error	6%	5%
	Train a bigger Network	Bias / Variance: Regularization?

0.5%
0.2% Avoidable Bias
0.7%
0.1%
0.8%

⇒ Surpassing human level performance:

Team of humans	0.5%	Avoidable Bias = 0.1%
One human	1%	
Training Error	0.6%	Variance 0.2%
Dev Error	0.8%	

Is Avoidable Bias = 0.1%.
0.5%. 0.2%.
1%. 0.3%.
0.3% As 0.3% might
0.4%. be an overfit
So incomplete data

⇒ Problems where ML significantly surpasses human level performance:
↳ Online Advertising [Click on ad] } Lots of data
↳ Product Recommendations
↳ Logistics (predicting transit time)
↳ Loan approvals
↳ These are structured data; humans are good at Natural perception problems like Computer vision, audio, video not at structured data.

↳ Some image recognition } Computers have surpassed humans
↳ Some speech recognition
↳ Some medical. Eg. ECG

⇒ Improving your model performance:
The two fundamental assumptions of supervised learning
↳ You can fit the training set pretty well ⇔ Avoidable Bias
↳ The training set performance generalizes pretty well to the dev/test set ⇔ Variance

⇒ Reducing (avoidable) bias and variance:
Human-level
↑ Avoidable Bias ↓ → Train bigger model
Train longer / better optimization Algo like Adam, RMSProp
NN architecture / hyperparameter search (RNN, CNN)

Training Error
↕ Variance → More data
Regularization (L2, dropout, data augmentation)
Dev Error. NN architecture / hyperparameter search

Week - 2

* Error Analysis :

Carrying out error analysis :

⇒ Should you try to make your cat classifier do better on dogs?

Error Analysis

↳ Get ~100 mislabeled dev set examples.

↳ Count up how many dogs. "Ceiling Method"

If our model is showing 10% of Error and if we have 5/100 dogs and we manually classify them But then too our error will drop down to 9.5% from 10% error.

But if there are 50/100 dogs then our error will drop down to 5% from 10%.

⇒ Evaluate multiple ideas in parallel :

↳ Fix pictures of dogs being recognized as cats

↳ Improve performance on blurry images

↳ Fix great cats (Lions, panthers, etc) being misrecognized

Incorrectly labeled	Image	Dog	Great Cats	Blurry	Instagram	Comments
✓	1	✓			✓	Pitbull
	2			✓	✓	
✓	3		✓	✓		Rainy day at zoo
6%	% total	8%	43%	61%	12%	

⇒ Cleaning up Incorrectly labeled data :

x → cat dog Cat cat Mat dog dog

y → 1 0 1 1 0 1 0

DL Algorithms are quite robust to random errors in the training set.

For classifying these incorrect labels we can add anew column of Incorrectly labelled data.

⇒ Now the question arises that wether we should fix these 6% labels or not. Before that we look at these 3 cases:

i) Overall dev set error - - - 2% - 10%. So instead of solving
Errors due to incorrect labels - - - 0.6% - 0.6%. error we can
Errors due to other causes - - - 1.4% - 9.4%. focus on other problems

↳ Fix up dev set errors here

⇒ 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 distribut^n.

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

↳ Train and dev/test data may now come from slightly different distributions

⇒ Build your first system quickly then iterate:

↳ Noisy Background
Café Noise, Car Noise.

↳ Accented speech

↳ Far from microphone

↳ Young children's speech

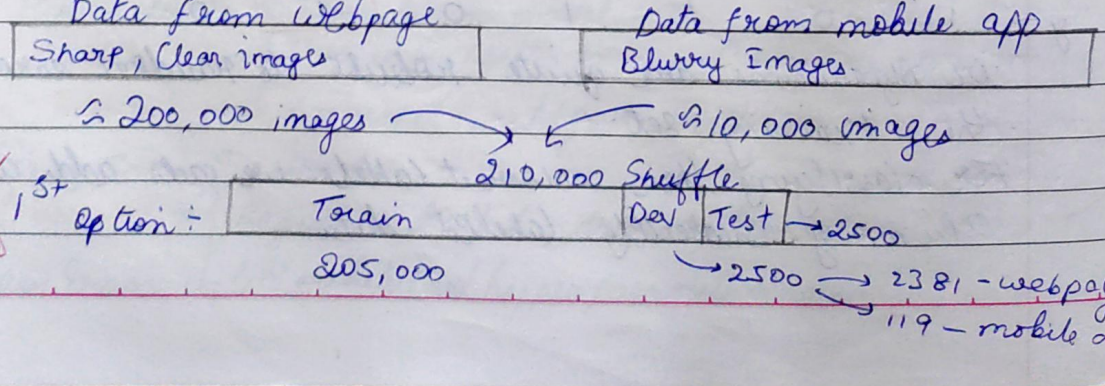
↳ Stuttering (uh, ah, hmm ..)

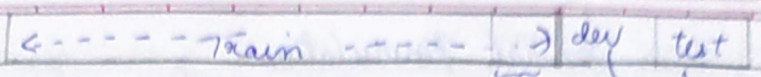
. . .

- Set up dev/test set and metric
- Build initial system quickly
- Use Bias/Variance analysis & Error analysis to prioritize next steps.

⇒ Training and Testing on Different Distributions:

Cat app example



2nd option :- 

web = 200,000 app = 5000
Both app 2500 each

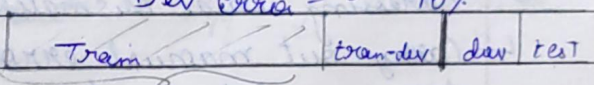
⇒ Speech Recognition example :-
Ex :- Speech oriented rearview mirror

Training	Dev/Test
Purchased data	Speech activated rearview mirror
Smart speaker control	↳ 20,000
Voice keyboard	Train / dev / Test
..... ↳ 500,000	510K / 5K / 5K

↳ 10K

⇒ Cat classifier example :-
Assume humans get 0% error
Training error - - 1%
Dev error - - - 10%

Traing Error - - 1%	Variance problem
Train-dev error - - 9%	
Dev Error - - - 10%	



NN saw this

Training-dev set : Same distribut^n as training set but not used for training

Training Error - - 1%	Human 0%	10% ↕ Bias	10% ↑ Avoidable Bias
	Avoidable Bias		↕ Variance
Train-Dev Error - - 1.5%	↕ data mismatch	11%	11%
Dev error - - 10%		12%	20% ↕ Data mismatch
		Bias	Bias + data-mismatch

⇒ Bias / Variance on mismatched training and dev/test sets :-

Human level - - - 4%	↕ avoidable Bias	4%
Training set error - - 7%	↕ Variance	7% } Training set evaluat^n
Train-dev set error - 10%	↕ Data mismatch	10% }
Dev - error - - - 12%	↕ degree of overfitting to dev set	6% } Dev/test set evaluat^n
Test - Error - - - 12%		6%

⇒ More general formulation :

	General Speech Recognition	Rearview mirror Speech data
Human level	"Human level" say (4%)	6% ↑ Avoidable Bias
Error on Examples NN has trained on	"Training error" (7%)	6% ↑ Variance
Error on Ex's NN has not trained on	"Train-dev error" 10%	"Dev/test error" 6%

data mismatch

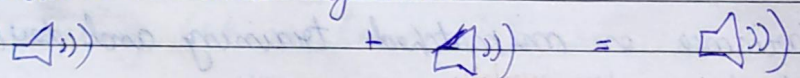
⇒ Addressing Data Mismatch :

↳ Carry out manual error analysis to try to understand difference between training and dev/test sets.
Ex: noisy - car noise ; mismatching street numbers.

↳ Make training data more similar, or collect more data similar to dev/test sets.
Ex : Simulate noisy in-car data

⇒ Artificial data synthesis :

It has all the attributes



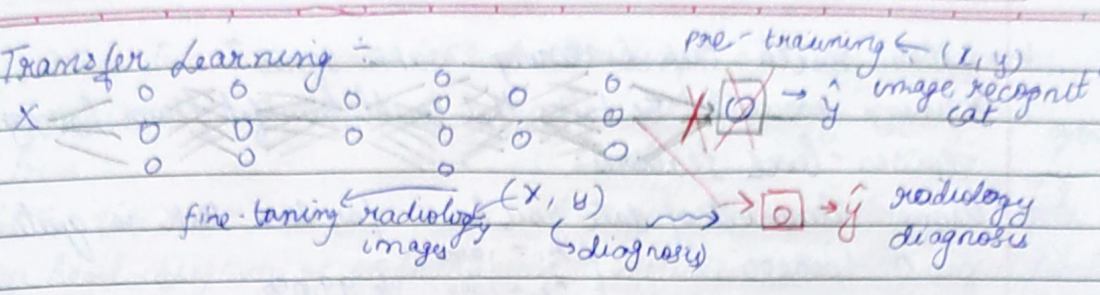
"The quick brown fox jumps over the lazy dog"
↳ 10,000 hours

Car noise
↳ 1 hour of car noise

Synthesized in-car audio
↳ ⊙ Set of all audio in car.

There is a chance that 1 hour of car noise can overfit the data so one ~~must~~ can get 10,000 hours of car noise data.

⇒ Transfer Learning :



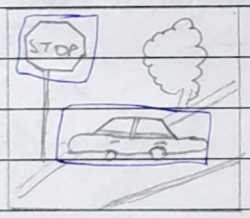
↳ When does transfer learning make sense?

Transfers from A → B

i) Task A and B have the same input x [like image, audio]
ii) You have a lot more data for Task A than task B.
iii) Low level features from A could be helpful for learning B.

⇒ Multi-task learning :

Simplified autonomous driving example :



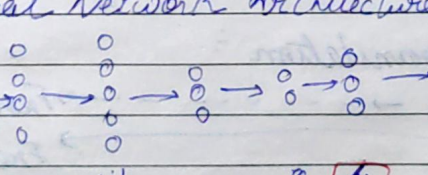
$x^{(i)}$

$y^{(i)}$ $(4,1)$

Pedestrian 0
Cars 1
Stop signs 1
Traffic light 0

$Y = [y^{(1)} \; y^{(2)} \; \cdots \; y^{(m)}]$ $(4, m)$

⇒ Neural Network Architecture :



Multi task learning

$$Y = \begin{bmatrix} 1 & 1 & 0 & ? \\ 0 & 1 & \cdots & 1 \\ ? & ? & 1 & ? \\ ? & ? & 0 & ? \end{bmatrix}$$

Loss $\hat{y}^{(i)}$ $(4,1)$

$$\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{4} \mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$$

↳ usual length.

sum only over values of j with 0/1 label.

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

Unlike softmax regression : here one image can have multiple labels like cars, pedestrian ...

Page No.
Date

↳ When multi-task learning make sense :

i) Training on a set of tasks that could benefit from having shared lower-level features.

ii) Using : Amount of data you have for each task is quite similar.

A 1,000,000 : $A_1 = 1000$, $A_2 = 1000$, ..., $A_{100} = 1000$ } 99,000

B 1000

iii) Can train a big enough neural network to do well on all the tasks.

⇒ End-to-End Deep learning :

Speech recognition example :- Traditn pipeline approach for small data

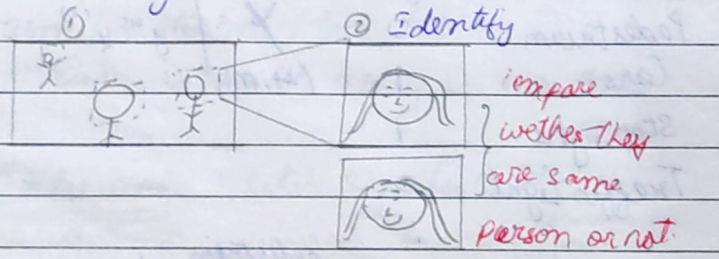
30,000 hrs → X → MFCC → features → ML → Phonemes → "C a t" → words → transcript

1,00,000 hrs audio

audio → transcript } DL approach

1,000,000 hours

⇒ Face Recognition :-



⇒ Examples like machine translation :

English → text analysis → - - - - - - - → French

English → French

⇒ Estimating child age from X-ray :-

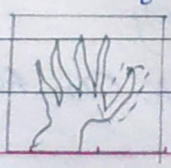


Image → ① bones → ② age

Image → age

⇒ Wether to use end-to-end learning:
Pros and cons of end to end deep learning
Pros:
↳ Let the data speak $x \rightarrow y$
↳ Less hand-designing of components needed

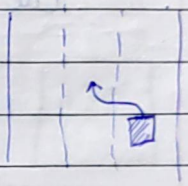
Cons
↳ May need large amount of data.
↳ Excludes potentially useful hand-designed components

input $x \rightarrow y$ output

⇒ Applying end-to-end deep learning:
Key question: Do you have sufficient data to learn a funct^n of the complexity needed to map x to y?

⇒ Autonomous car driving:

Image, radar $\xrightarrow{DL}$ Cars, pedestrians $\xrightarrow{\text{motor planning}}$ route $\xrightarrow{\text{control}}$ steering



↳ Use DL to learn individual component
↳ Carefully choose $X \rightarrow Y$ depends on what tasks you can get data for.

image ——————————→ steering

Page No.
Date 7 | 6 | 21

DEEP LEARNING :- Course - 4

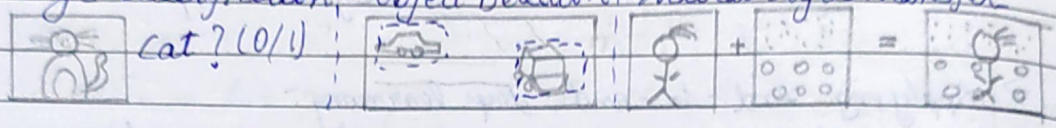
Convolutional Neural Networks

Week 1

★ Foundations of Convolutional Neural Network :-

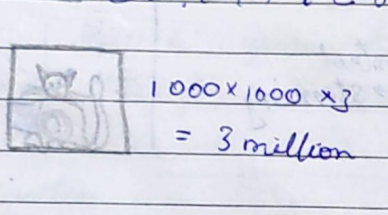
Computer Vision :-

Image classification | Object Detection | Neural Style Transfer



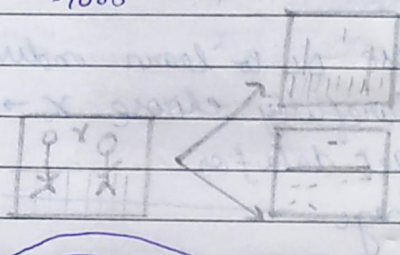
⇒ Deep learning on large images :-

Cat? (0/1) i.e. 64×64×3 = 12288 Low res image



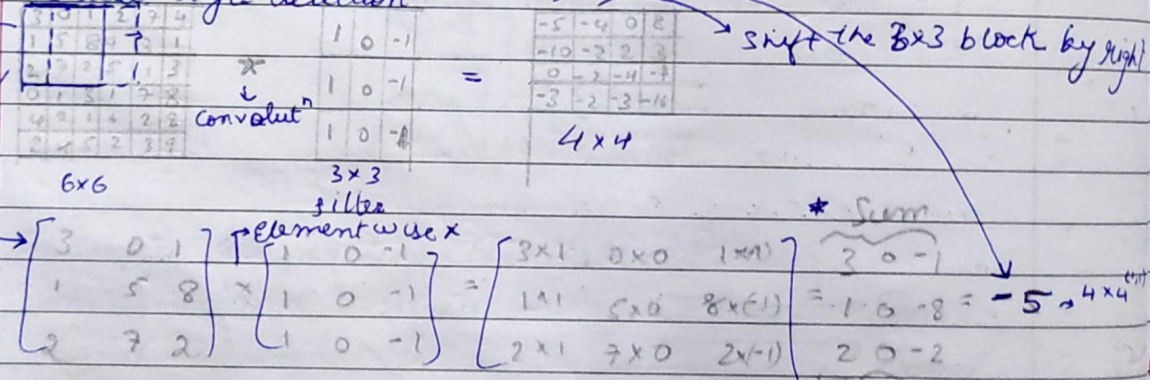
⇒ Edge Detection Example :-

Computer Vision Problem :-



★ How it works

⇒ Vertical Edge detection :-

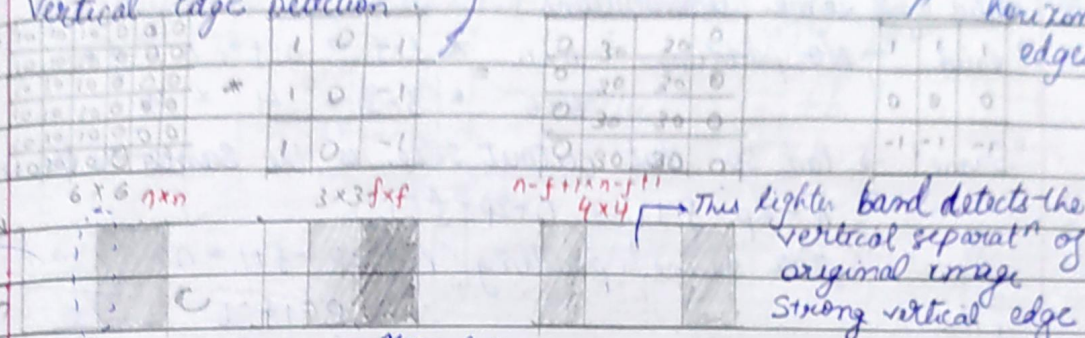


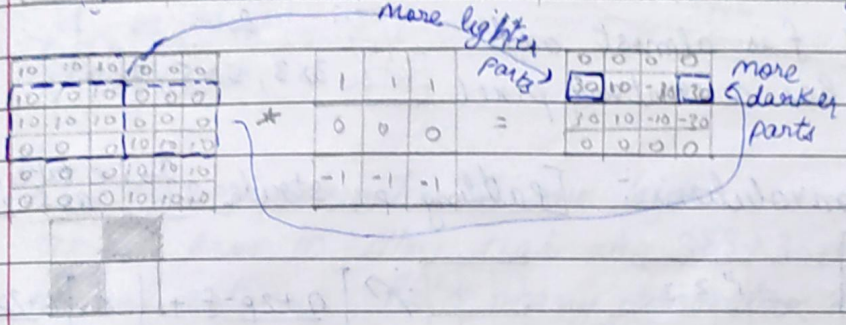
python : conv-forward

tensorflow : tf.nn.conv2d

Keras : Conv2D

⇒ Vertical Edge detection :





⇒ Filters :

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

Sobel filter

→ Advantage : It puts a little bit more weight to the central row, the central pixel and it makes it maybe a little bit more robust.

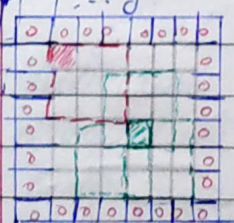
3	0	-3
10	0	-10
3	0	-3

Scharr filter

With these filter we use it for edge detection. But computer has a better understanding of selecting the 9 numbers and it often more robust than hand-picked 9 numbers (i.e 3×3 matrix)

for p=2 two layers ⇒

Now with Padding 6×6 image has changed to 8×8 Pad it with 0 P=padding P=1



Padding : Why we need to used Padding i.e adding extra column of blank pixels :

i) Shrinking output our picture is originally of 6×6 pixels but shrinks to 4×4 and this may continue as we train high Deep layer neural net.

ii) Throwing away info from edge : As we only travel once through first all so we may lack imp info from that pixels. Now the final Image Size will be $n+2p-f+1 \times n+2p-f+1$

$6+2-3+1 \times 6+2-3+1$

6×6

⇒ Valid and Same convolutions :

"Valid" → No padding : $n \times n \;*\; f \times f \rightarrow n-f+1 \times n-f+1$

$6 \times 6 \;*\; 3 \times 3 \rightarrow 4 \times 4$ Size

"Same" → Pad so that output size is the same as the input.

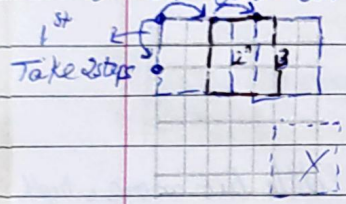
$n+2p-f+1 \times n+2p-f+1$

∴ For "same" padding $n+2p-f+1 = n$

$\therefore p = \frac{f-1}{2}$

↳ In CV f is almost odd as it has a central pixel  3×3, 5×5, 7×7 ...

⇒ Strided Convolutions : [padding p, stride s] → i.e take 2 jumps as s=2



7×7 ($n \times n$) * 3×3 ($f \times f$) = 3×3

(say s=2)

$\left[\frac{n+2p-f+1}{s}\right] \times \left[\frac{n+2p-f+1}{s}\right]$

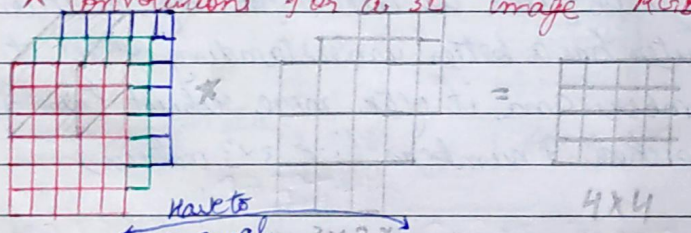
$\frac{7+0-3+1}{2} = 3 \times 3$

floor [ ]

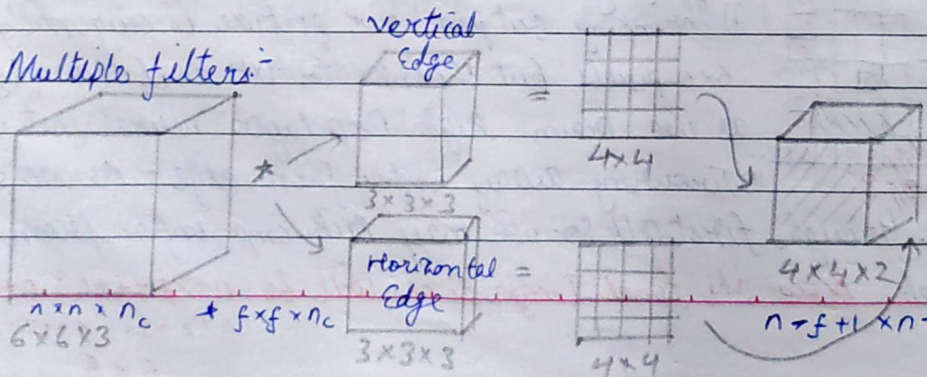
floor [7.1] = 7

⇒ Convolutions of RGB image :

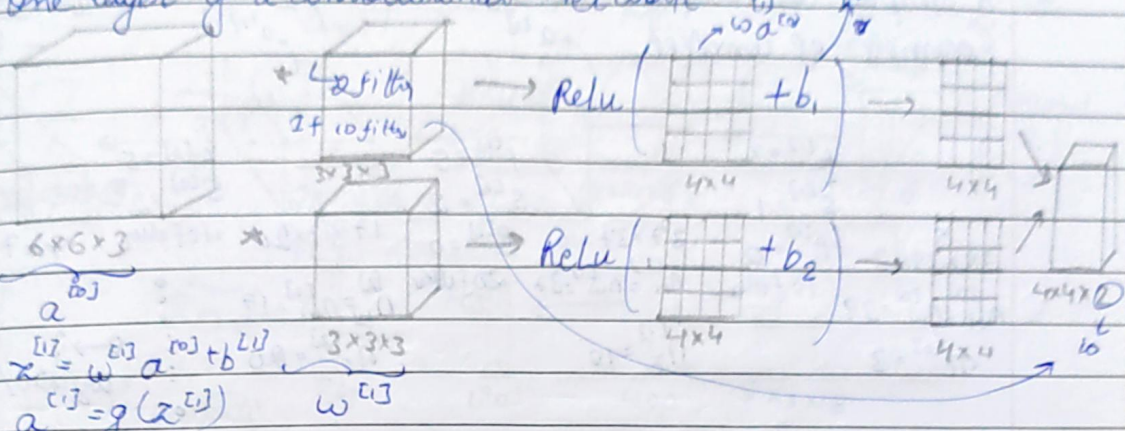
★ Convolutions for a 3D-image RGB images.



⇒ Multiple filters :

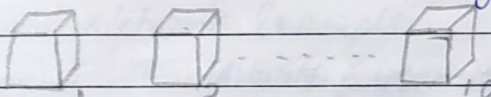


⇒ One layer of a Convolutional Network :



⇒ Number of parameters in one layer :

Q) If you have 10 filters that are 3×3×3 in one layer of a neural network, how many parameters does that layer have?



i.e 27 parameters + bias = 28

∴ 28 × 10 = 280 parameters.

⇒ Summary of notation :

If layer l is a convolution layer :

$f^{[l]}$ = filter size Input : $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$

$p^{[l]}$ = padding

$s^{[l]}$ = stride Output : $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

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

Each filter is : $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$

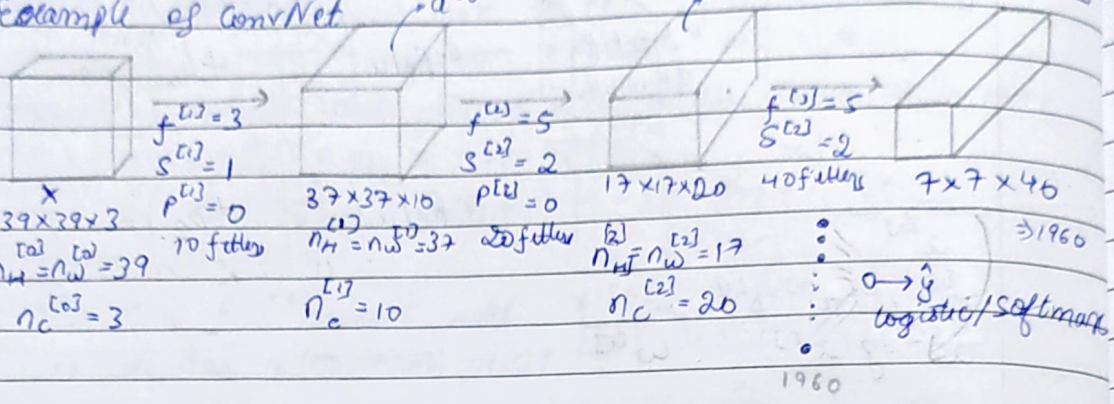
Activations : $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$ $A^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

Weights : $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]} \rightarrow$ no. of filters in layer l

Bias : $n_C^{[l]} \rightarrow (1, 1, 1, n_C^{[l]})$

⇒ A simple convolution network example : Formula ⇒ $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$

Example of ConvNet



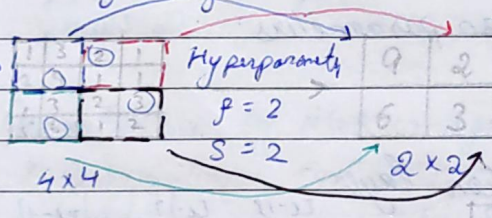
⇒ Types of layer in a convolutional network :

↳ Convolution (CONV)

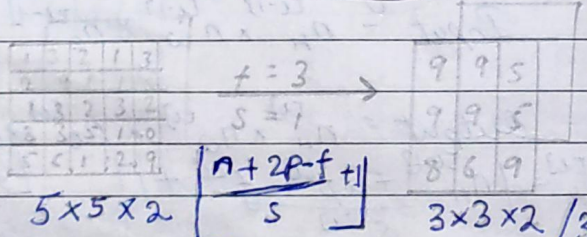
↳ Pooling (POOL)

↳ Fully Connected (FC)

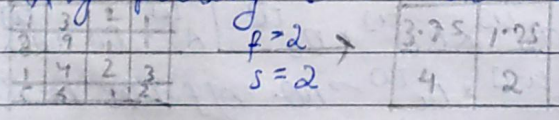
⇒ Pooling layers : Max pooling

↳ 

Take maximum number from each 2×2 bracket.

↳ 

↳ Avg pooling (Not used much)



⇒ Summary of pooling : Hyperparameters $n_H \times n_W \times n_c$

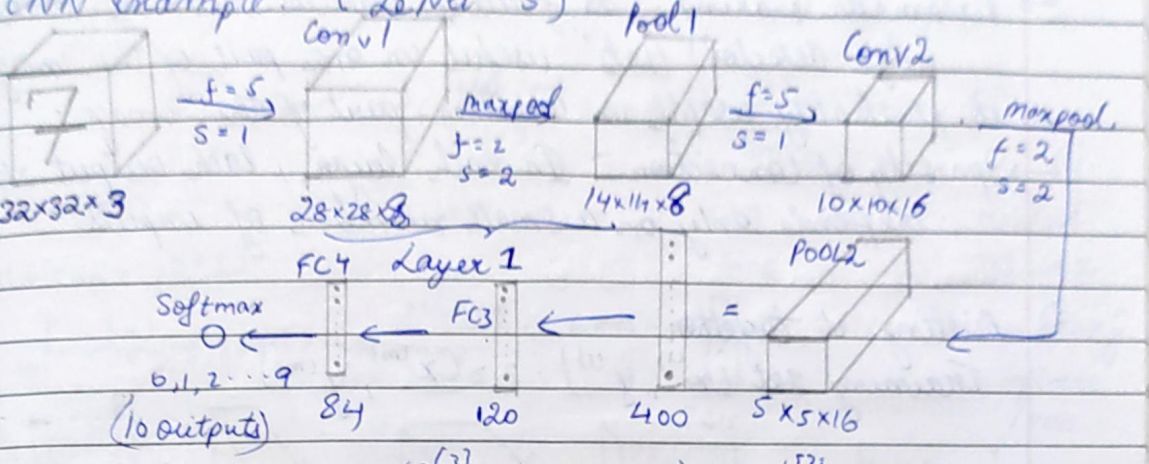
↳ f → filter size ; Say f=2, s=2 / f=3, s=2 ↓

↳ s → Stride ↳ p → Padding (p=0) often $\left\lfloor \frac{n_H - f + 1}{s} \right\rfloor \times \left\lfloor \frac{n_W - f}{s} + 1 \right\rfloor$

↳ Max or average pooling $\times n_c$

No parameters to learn !!! Fixed funct^n

⇒ CNN Example : (LeNet-5)



$w^{[3]} = (120, 400)$ & $b^{[3]} = (120)$

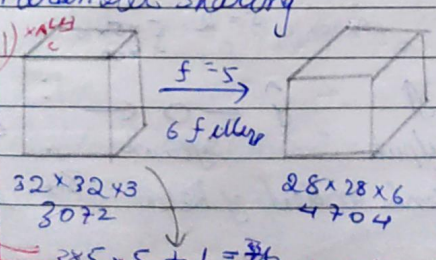
As we go deeper in the neural network n_H & n_W will decrease & Number of channels will increase.

⇒ Neural Network Example :

	Activation Shape	Activation Size	# parameters
Input	(32,32,3)	3072 $a^{[0]}$	0
Conv1 (f=5,s=1)	(28,28,8)	6272	608
Pool 1	(14,14,8)	1568	0
Conv2 (f=5,s=1)	(10,10,16)	1600	3216
Pool 2	(5,5,16)	400	0
FC3	(120,1)	120	48120
FC4	(84,1)	84	10164
Softmax	(10,1)	10	850

⇒ Why convolutions : Why convolution layers have advantage over fully connected layers (FC)

i) Parameter Sharing



ii) Sparsity of connection

So if we create a neural network of $3072 \times 4704 \approx 14M$ it is huge

So we can see why conv has few features

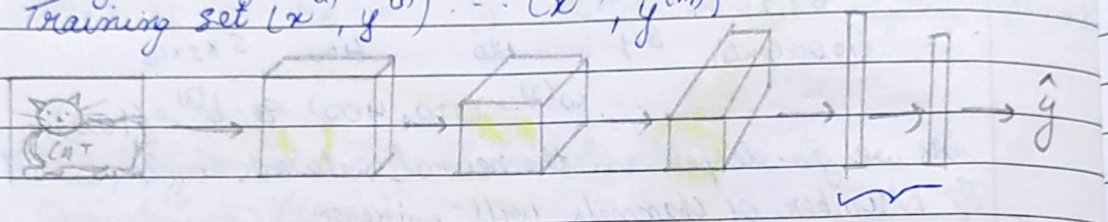
Page No.
Date 16 | 6 | 21

↳ Parameter sharing :- A feature detector (such as vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.

↳ Sparsity of connection :- In each layer, each output value depends only on a small number of inputs.

⇒ Putting it together :-

Training set $(x^{(1)}, y^{(1)}) \cdots (x^{(m)}, y^{(m)})$



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

Use gradient descent to optimize parameters to reduce J

Week - 2

* Case Studies :- Outline

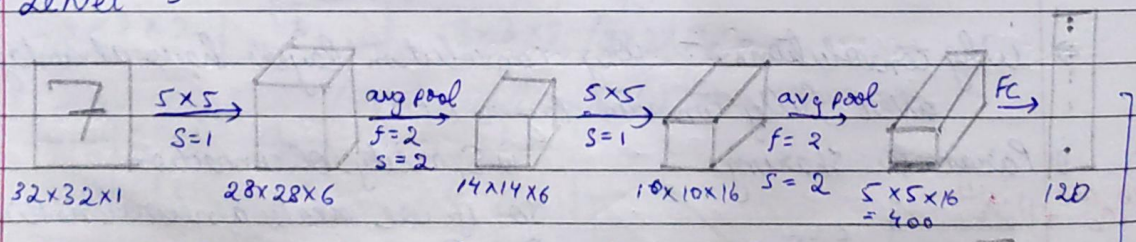
⇒ Classic Networks :- ⇒ Res Net (152) ⇒ Inception

↳ Le Net - 5

↳ Alex Net

↳ VGG

⇒ Le Net - 5

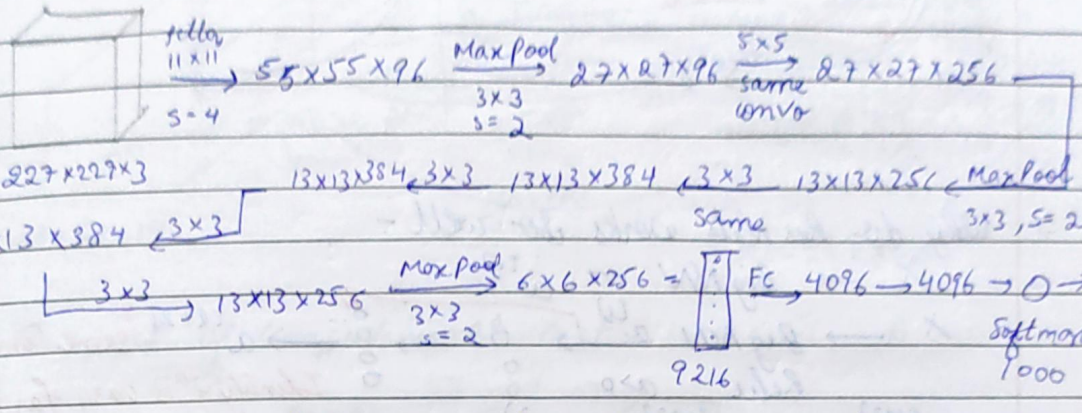


from left to right (n_H, n_W)↓ & (n_c)↑

Conv pool conv pool fc fc output

In early days they used sigmoid/tanh instead of Relu

⇒ Alex Net :-

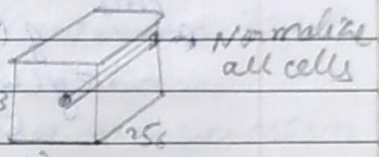


↳ It is similar to LeNet but much bigger

↳ We use Relu

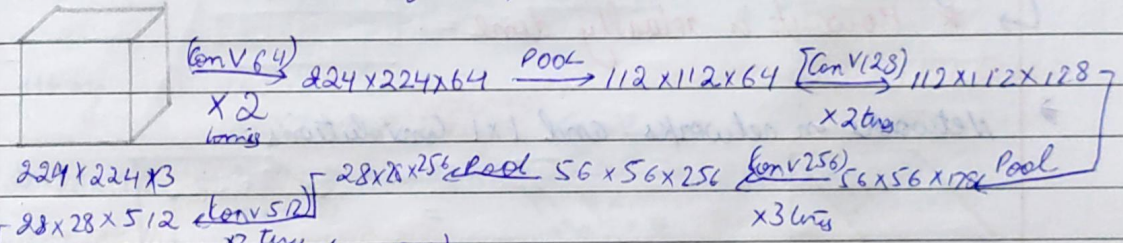
↳ Multiple GPU's are used

↳ Local Response Normalization (LRN) Not used Nowadays.



⇒ VGG - 16 : I+ means 16 layers which has weight

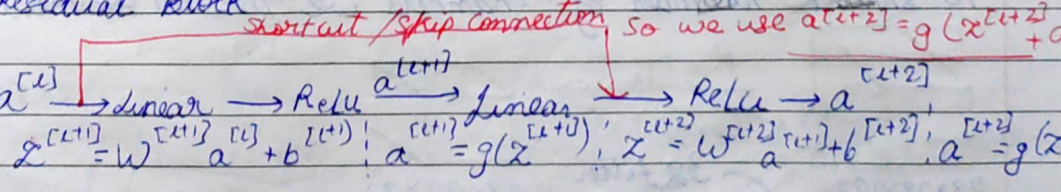
Conv = 3×3 filter, s=1, same conv & Max pool = 2×2, s=2



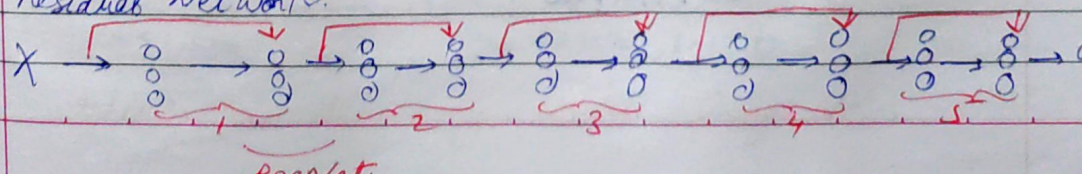
⇒ 138 M parameters

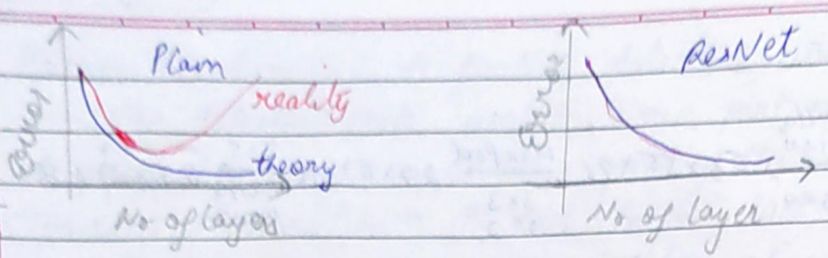
⇒ Residual Networks (Res Nets) : $a^{[l]} \rightarrow a^{[l+1]} \rightarrow a^{[l+2]}$

Residual Block



⇒ Residual Network :-





⇒ Why do ResNets works so well :-

$X \rightarrow$ Big NN $\rightarrow a^{[l]}$

$X \rightarrow$ Big NN $\rightarrow a^{[l]} \rightarrow \rightarrow \rightarrow a^{[l+2]}$ "same Convo"

Relu $a \geq 0$

Identity f^n is easy for residual block to learn

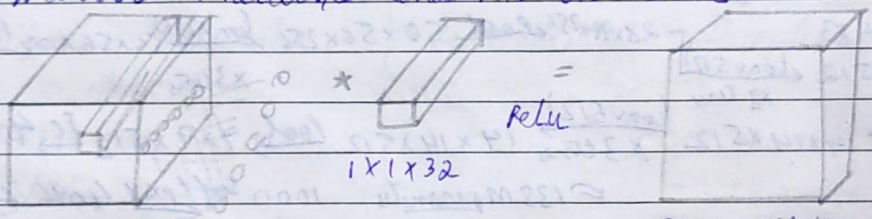
$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$
$$= g(w^{[l+2]} a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

↳ If $w^{[l+2]} = 0$ & $b^{[l+2]} = 0$ $\quad a^{[l]} = g(a^{[l]})$
$= a^{[l]}$ [Relu]

⇒ Why do residual network work :

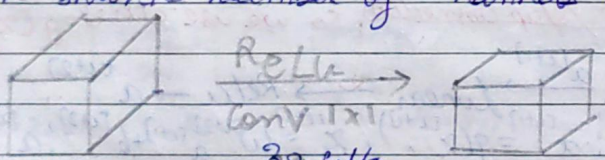
↳ ★ How it is actually done

⇒ Networks in networks and 1×1 Convolutions :



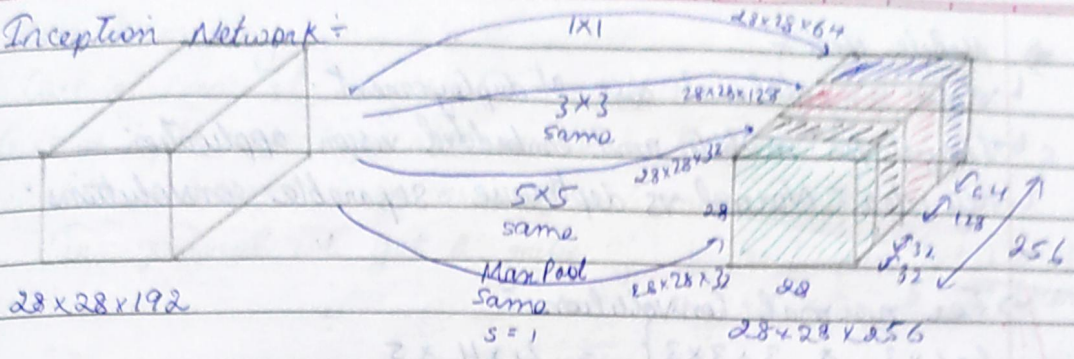
6×6×32 Network in Network 6×6× ~~32~~ No of fetters

⇒ To shrink number of channels we use 1×1 convolutions :

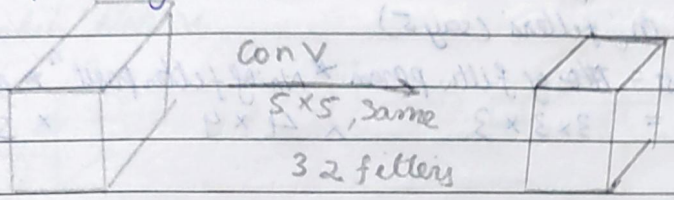


28×28×192 28×28×32

⇒ Inception Network :



⇒ The prob. of computational cost :



Here, we have 32 filters of 5×5×192.

Total 28×28×32×5×5×192 ≈ 120M

So we can see it computationally expensive. We will use 1×1 convolution and it will reduce it by 1/10th.

⇒ Using 1×1 convolutions :

as channels shrunk from 192 to 16

Conv (1×1, 16) → Bottleneck layer → Conv (5×5, 32)

28×28×192 → 28×28×16 → 28×28×32

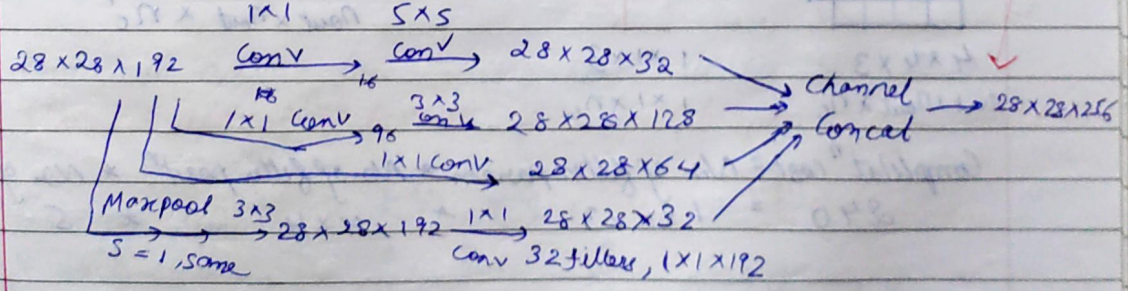
No of filters: 16 (1×1×192) | filter size: 5×5×16

28×28×16×1×1×192 = 2.4M

28×28×32×5×5×16 = 10M

∴ Total = 12.4M ∴ It dropped from 120M to 12M

⇒ Inception module :



* How does it actually looks : Google Net

⇒ Mobile Nets :

↳ Low computational cost at deployement.

↳ Useful for mobile and embedded vision applications.

↳ Key idea : Normal vs depthwise - separable convolutions.

⇒ For normal Convolutions :

$6 \times 6 \times 3$ * $3 \times 3 \times 3$ } = $4 \times 4 \times 5$

$n \times n \times n_c$ $\quad$ $n \times n \times n_c$ $\quad$ $n_{out} \times n_{out} \times n_c'$

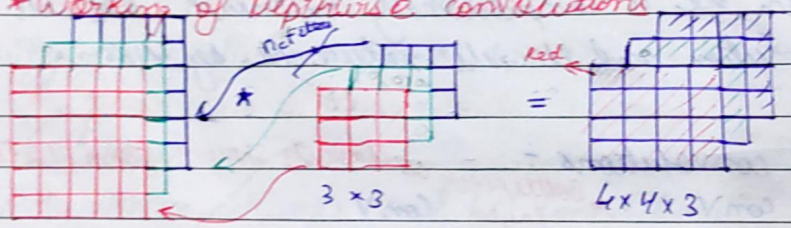
n_c' filters (say 5)

Computational cost = #No of filter param * No of filter posit^n * no of filters

2160 = $3 \times 3 \times 3$ $\times$ 4×4 $\times$ 5

⇒ i) Depthwise Separable Convolutions :

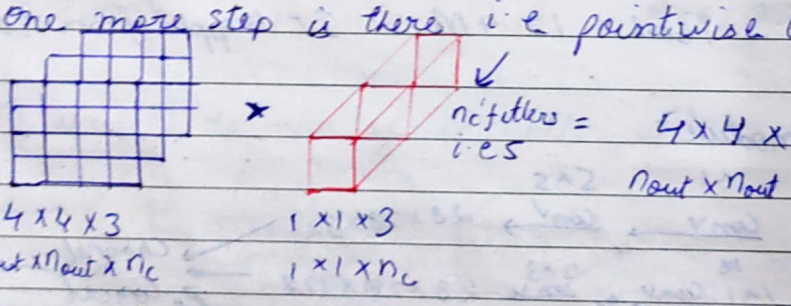
* Working of Depthwise convolutions



Computational Cost = No. of filter param * No of filter posit^n * No of filters

432 = 3×3 $\times$ 4×4 $\times$ 3

ii) One more step is there i.e pointwise convolution



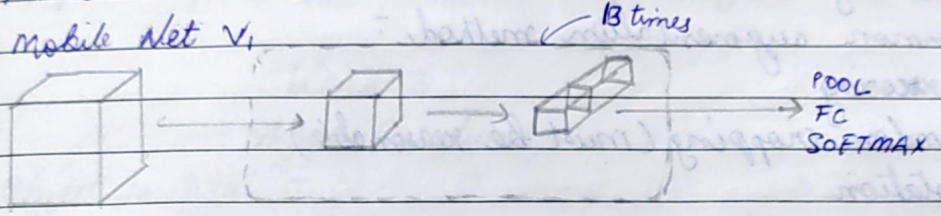
Compl$^{at^n}$ cost = No of filter param * No of filter posit^n * No of filters

240 = $1 \times 1 \times 3$ * 4×4 * 5

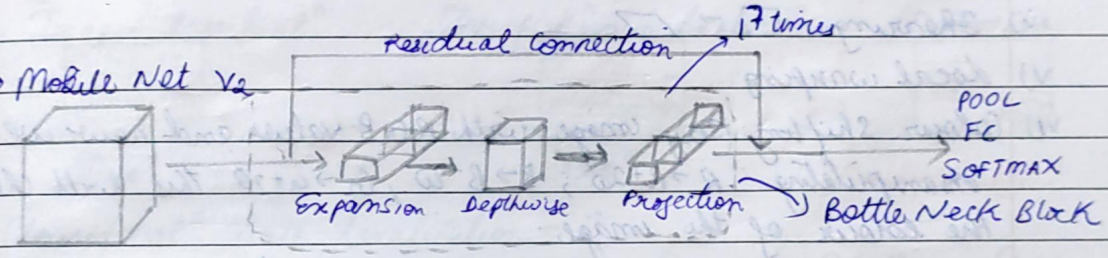
⇒ Cost Summary :
Cost of Normal Convolution = 2160
Cost of depth wise separable convolution = 432 + 240 = 672
∴ Normal conv was around 31% more expensive
In general we get a ratio of $\frac{1}{n_c} + \frac{1}{f^2}$ i.e $\frac{1}{5} + \frac{1}{3^2}$ ≅ 31% (Here)

⇒ Mobile Net Architecture :

↳ Mobile Net V1



↳ Mobile Net V2

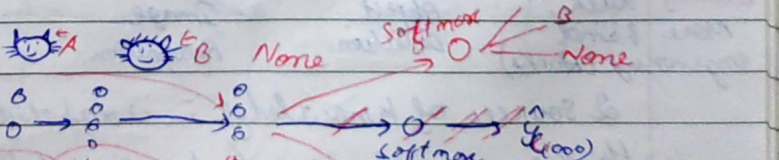


* How does bottleneck work

⇒ Effcient Net :
* How Efficient Network works
↳ We cana use High Resolution, make a deeper architecture, make the layers wider. So which one is best suited or you can also do compound scalling i.e simultaneously performing 2 tasks.

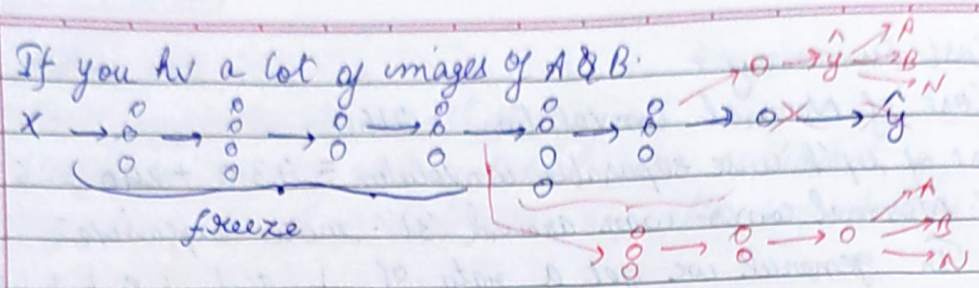
* Practical advice for using conv-net :
* Practical applicat^n of ResNets using github open source via CMD

⇒ Transfer learning :



If u hv less pics of A & B. freeze = 1

⇒ If you hv a lot of images of A & B.



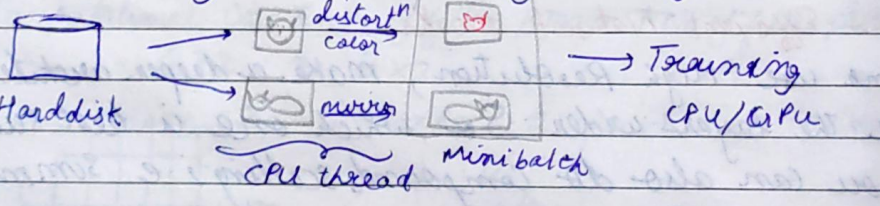
If u Rv sufficient data u can train your own Nets.

⇒ Data Augmentation :
Common augmentation method :

i) Mirroring
ii) Random cropping (must be reasonable)
iii) Rotation
iv) Shearing □ → ▱
v) Local warping
vi) Colour shifting (say image with R G B values and now we manipulating R → R+20 ; B → B-20 , G → G+20 this will change the colour of the image.

↳ Alex Net Paper for colour augmentation "PCA colour augmentation".

⇒ Implementing distortions during training :



⇒ State of Computer Vision :
Data V/S hand engineering

Transfer learning — More hand engineering (Hacks) — little data — Object Detection — Image Recognition — Speech Recognition — Lots of data — Simple algo — Less hand-Engineering

2 sources of Knowledge : i) Labeled data (x, y)
ii) Hand engineer features / network architecture / other components.

⇒ Tips for doing well on Benchmarks/winning competition :-

i) Ensembling : Train several 3-15 networks independently and average their outputs ($\hat{y}$)

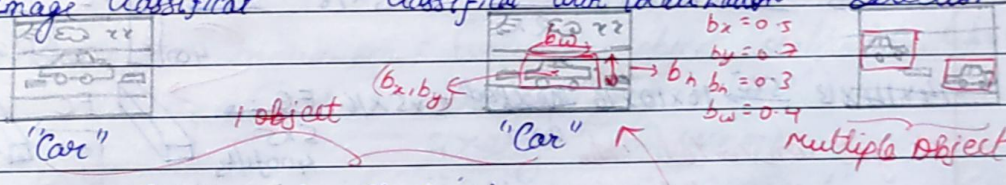
iii) Multi crop at test time : Run classifier on multiple versions of test images and average results.

iii) Use open source code : Use architectures of networks published in the literature / Use open source implementation if possible / Use pretrained models and fine-tune on your dataset

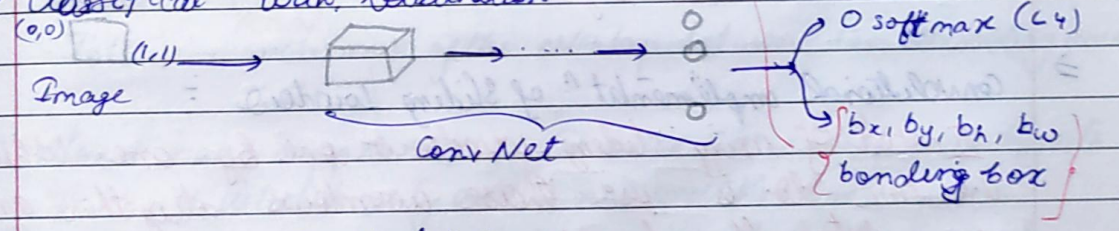
Week-3

Object Detection

★ Object localizat^n :-



⇒ Classificat^n with localization :-



⇒ Defining the target label y :-

1. pedestrian
2. car
3. Motorcycle
4. Background

Need to output b_x, b_y, b_h, b_w class label (1-4)

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$$ → is there any object

So for an image with car

With Car

$$y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

No car

$$y = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$ No one cares

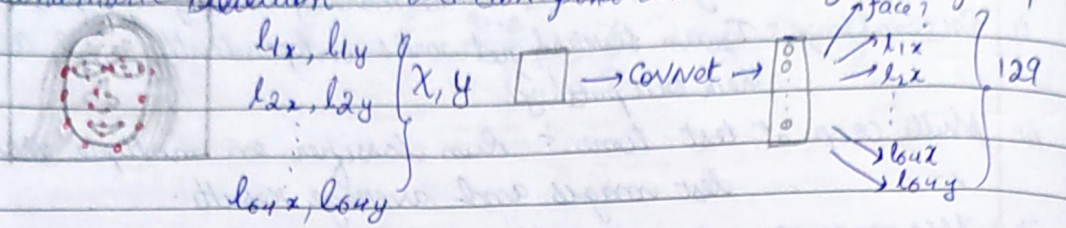
↳ loss f^n : $\mathcal{L}(\hat{y}, y)$

i) $(\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \cdots + (\hat{y}_8 - y_8)^2$ if $y = 1$

ii) $(\hat{y}_1 - y_1)^2$ if $y = 0$

Page No.
Date

⇒ Landmark Detection : We can find landmarks of nose, eyes, lips



⇒ Object Detection :

Car detection example : We use Sliding window detection

★ Sliding window detecting working

⇒ Convolutions involving Sliding windows :

Turning FC layer into convolutional layers

↳ 14×14×3 → (5×5) → 10×10×16 → (max pool 2×2) → 5×5×16 → FC → 400 → FC → 400 → y → Softmax(4)

↳ 14×14×3 → (5×5) → 10×10×16 → (max pool 2×2) → 5×5×16 → (FC 5×5, 400 filters) → 1×1×400 → (FC 1×1) → 1×1×400 → (1×1, 4 filters) → 1×1×4

FC got converted into a convolutional layer

⇒ Convolutional implementation of Sliding windows :

Instead of using sliding window one by one, what we can do is use these windows altogether and then detect the object

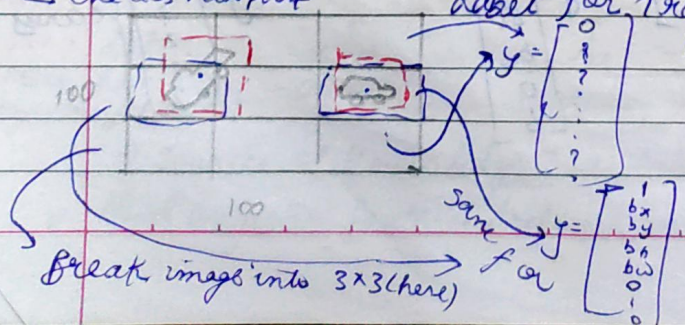
★ How it is implemented :

★ Real life Example

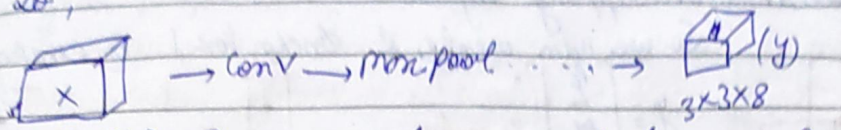
→ Object Detection : Bounding box predictions

YOLO Algorithm (You only look once)

". " checks mid point

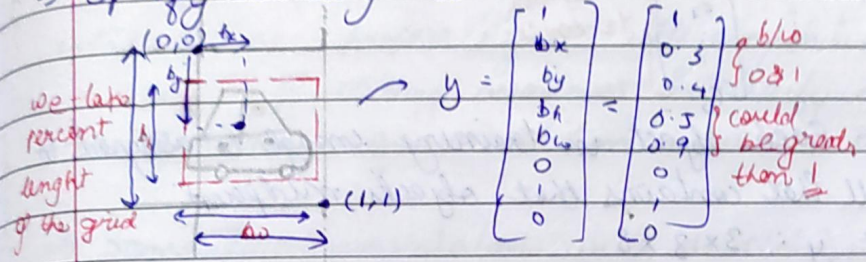


So;



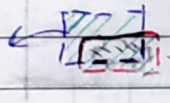
So in this algo we get bounding box of the object. We might use a grid of 19 by 19 instead of 3 by 3.

⇒ Specify bounding boxes :



⇒ Intersection over union : Evaluating object localization (IoU)

$$IoU = \frac{\text{size of } \square \text{ (intersect)}}{\text{size of } \square \text{ (union)}}$$

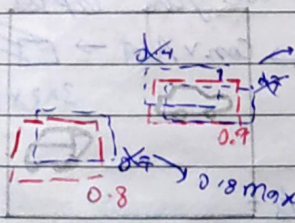


if IOU ≥ 0.5 "correct"

IoU is a measure of the overlap between two bounding boxes

⇒ Non-max suppression : It make sures that our objects is detected only once, not multiple times.

Here first we will check the max Pc & then we will check if IoU is greater than threshold we will reject the other boxes.



So what are the steps :

i) Firstly discard all boxes with $P_c \leq 0.6$

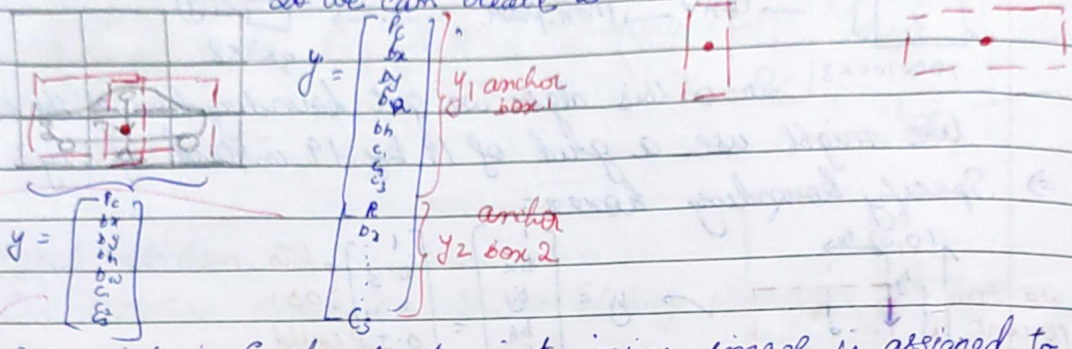
ii) While there are any remaining boxes

↳ Pick out the box with the largest P_c, output as the prediction.

↳ Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step.

Page No.
Date 29 | 7 | 21

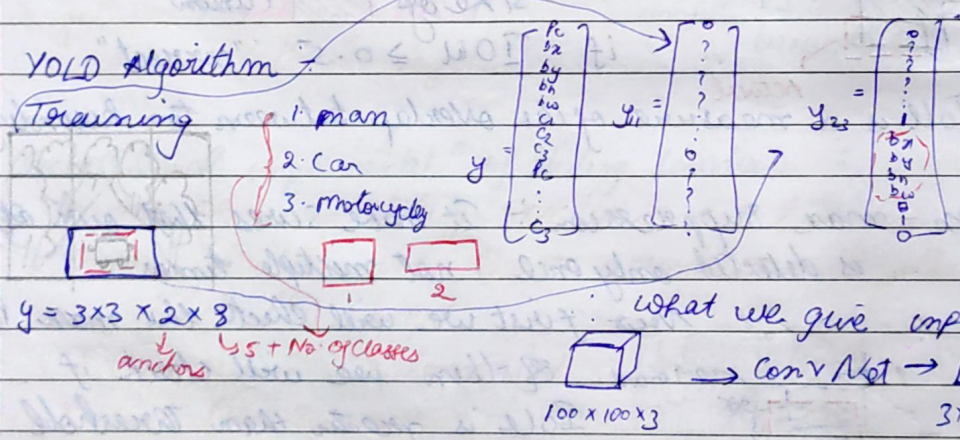
⇒ Anchor Boxes : Overlapping object.
So we can create 2 Anchor box 1 & anchor box 2



↳ Previously : Each object in training image is assigned to grid cell that contains that object's midpoint
Output y : 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 (grid cell, anchor box)
Output y = 3 × 3 × 16 or 3 × 3 × 2 × 8

⇒ YOLO Algorithm :
Training



What we give input

⇒ Making prediction :
Outputting the non-max suppressed outputs :
↳ For each grid cell, get 2 predicted bounding boxes.
↳ Get rid of low probability predictions.
↳ For each class (pedestrian, car, motorcycle) use non-max suppression to generate final prediction.

⇒ Region proposal : R-CNN

* There are many boxes where nothing is to be found, so a R-CNN segmentation algorithm is used and then objects are predicted.

Faster algorithms :

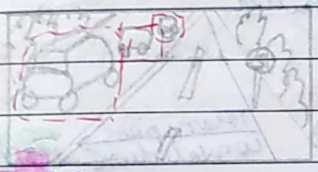
↦ R-CNN : Propose regions. Classify proposed regions one at a time. Output label + bounding box.

↦ Fast R-CNN : propose regions. Use convolution implementation of sliding windows to classify all the proposed regions.

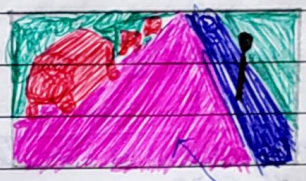
↦ Faster R-CNN : Use convolutional network to propose regions.

⇒ Semantic segmentation with U-Net :

Object detection v/s Semantic Segmentation.



Object Detection

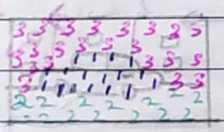


Semantic Segmentation find each and every pixel what it represents.

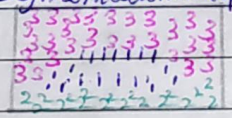
For self driving cars instead of detecting road it is better to (say road)

* Other applications like chest X-ray & Brain MRI

⇒ Per pixel class label :



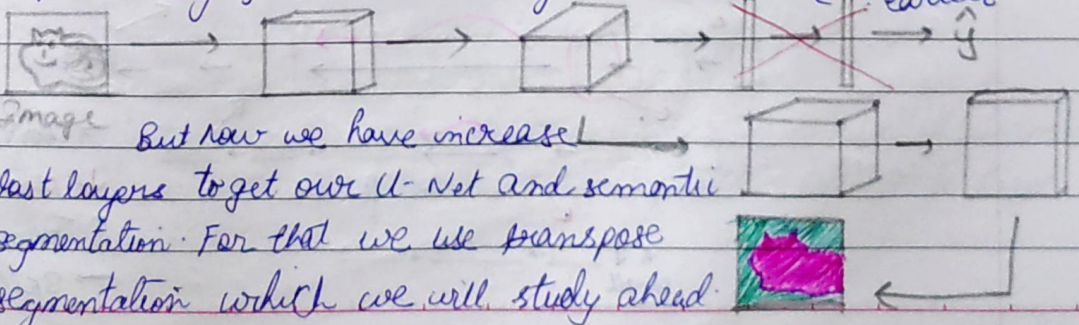
Segmentation Map 1. car



2. Road

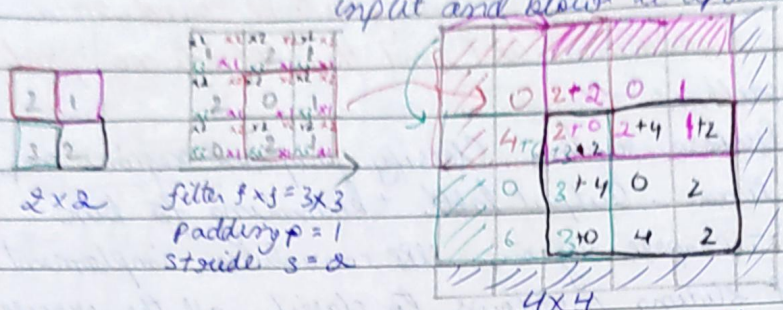
3. Building

⇒ Deep learning for Semantic Segmentation :



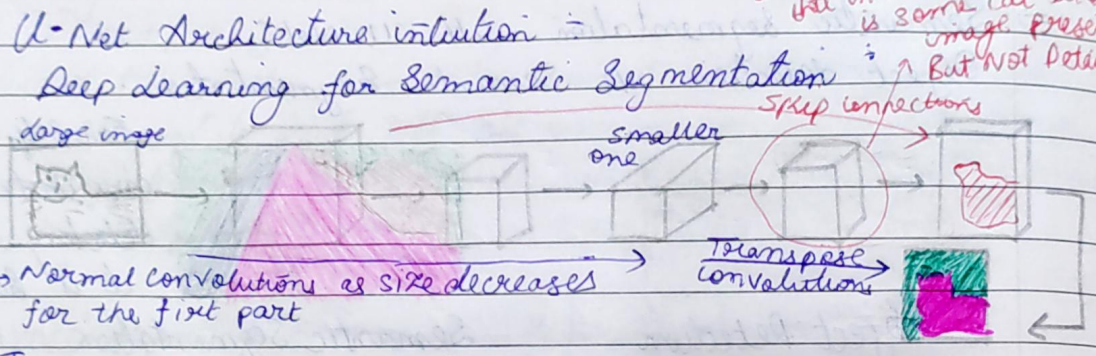
But now we have increased last layers to get our U-Net and semantic segmentation. For that we use transpose segmentation which we will study ahead.

⇒ Transpose Convolution :- Basically how do you take 2×2 input and blow it upto 4×4 input.



⇒ U-Net Architecture intution :-

Deep learning for Semantic Segmentation

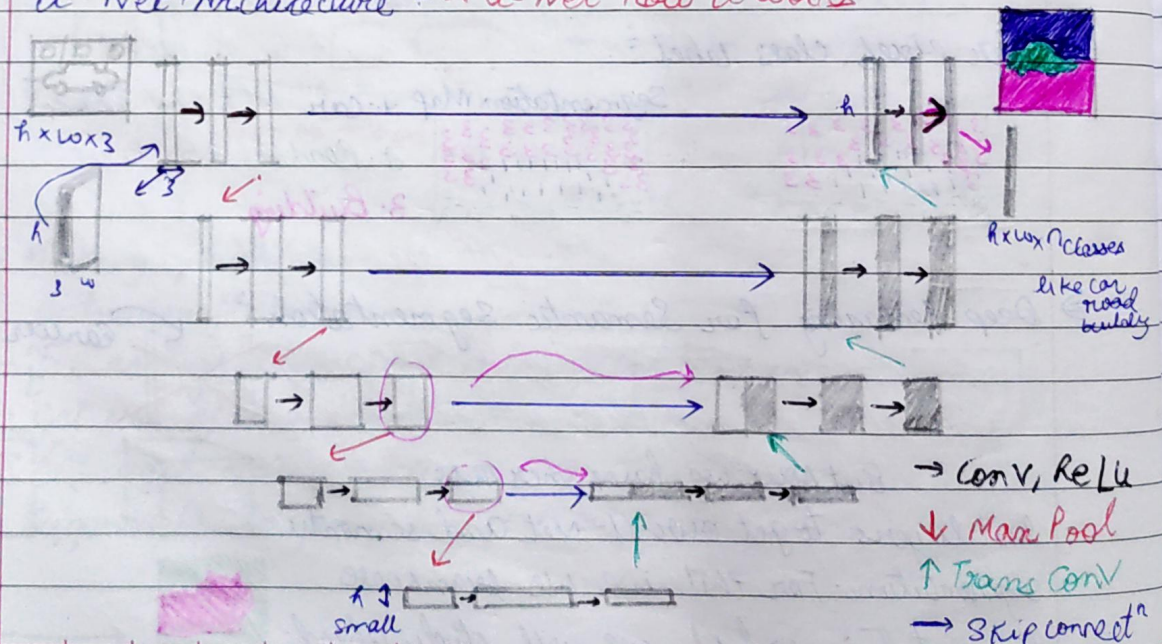


↳ Normal convolutions as size decreases for the first part

↳ In smaller one we have lots of spatial information

In this we can get an idea that in lower right - there is some cat like image present But not details

⇒ U-Net Architecture :- ☆ U-Net How it works



Week-4

Special Applications: Face recognition & Neutral Style Transfer

* Face Recognition :

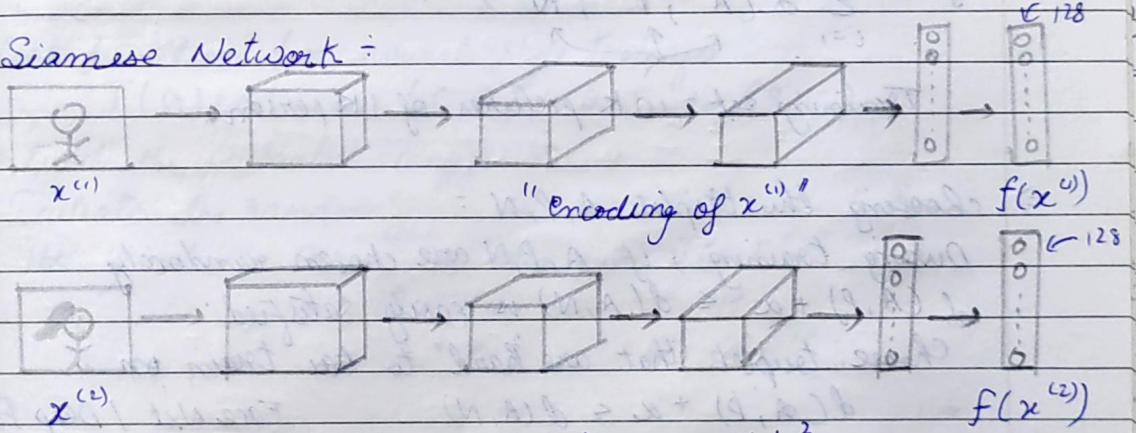
*Face recognition system = Baidu system face recognition

Face Verification 1:1
↳ Input image, name/ID
↳ Output whatever the input image is that of the claimed person

Face recognition 1:K
↳ 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 : we often have less data for recognit"/verificn
So we use dt concept of "similarity" function
d(img1, img2) = degree of difference between images
If d(image1, image2) ≤ τ "same" } Verification
> τ "different" }

⇒ Siamese Network :



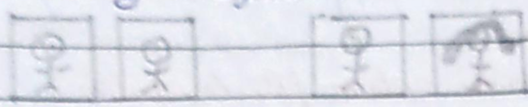
$$d(x^{(1)}, x^{(2)}) = \| f(x^{(1)}) - f(x^{(2)}) \|_2^2$$

⇒ Goal of learning :
Parameters of NN define an encoding $f(x^{(i)})$ → 128
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 the different person $\| f(x^{(i)}) - f(x^{(j)}) \|^2$ is large

Page No.
Date

⇒ Face Recognition : Triplet Loss → As we need 3 images → Anchor → Positive → Negative

Learning Objective



Anchor (A) Positive (P) Anchor (A) Negative (N)

(say 0.2)

we want $\|f(A) - f(P)\|^2 + \alpha \le \|f(A) - f(N)\|^2$ Not big enough

$d(A,P) = \alpha$ $d(A,N) = \alpha$ (0.5)

$\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 \le 0 - \alpha$ we keep alpha to prevent $0 - 0 \le 0$ Redundant

$+\alpha$

So our network will give reasonable output

⇒ Loss function :

Given 3 images A, P, N :

$\mathcal{L}(A, P, N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$

≤ 0 or > 0

$J = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$

Training set : 10K pictures of 1K person

⇒ choosing the triplets A, P, N :

During training, if A, P, N are chosen randomly

$d(A,P) + \alpha \le d(A,N)$ is easily satisfied.

Choose triplet that are "hard" to train on

$d(A,P) + \alpha \le d(A,N)$ Face Net / Deep Face

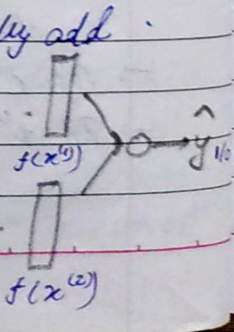
$d(A,P) \approx d(A,N)$

↓ ↑

⇒ Face verification and Binary classification :

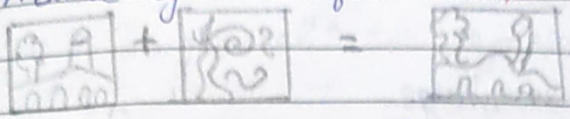
Using a siamese network of 2 person and finally add a single dense layer for binary classification :

$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_k |f(x^{(i)})_k - f(x^{(j)})_k| + b\right)$



$\frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k} = \chi^2$

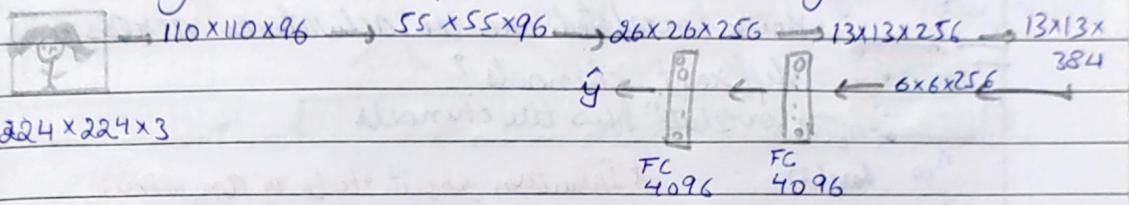
* Neural Style Transfer: * How neural style Transfer works



Content (C) Style (S) Generated Image (G)

⇒ What are deep ConvNets learning:

Visualizing what a deep Network is learning:



Pick a unit in layer 1. Find the nine image patches that maximize the units activation. Repeat for other units

* How to visualize deep layers

⇒ Cost function for Neural Style Transfer:

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

↳ Find the generated image G:

i) Initiate G randomly ; G: 100×100×3

ii) Use gradient Descent to minimize J(G)

$$G := G - \frac{\partial}{\partial G} J(G)$$ * How it works

⇒ content cost function:

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

↳ Say you use hidden layer l to compute content cost.

↳ Use pre-trained ConvNet (Eg; VGG Network)

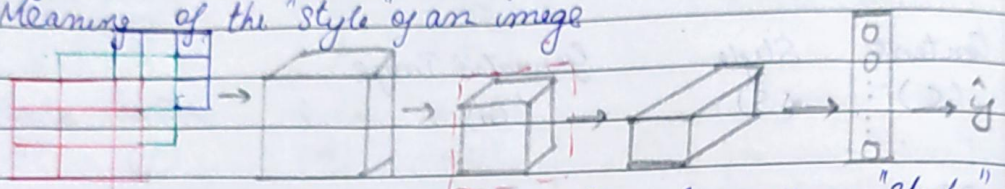
↳ Let $a^{[l](C)}$ & $a^{[l](G)}$ be the activation of layer l on the images.

↳ If $a^{[l](C)}$ & $a^{[l](G)}$ are similar both images have similar content.

$$J_{content}(C, G) = \frac{1}{2} || a^{[l](C)} - a^{[l](G)} ||^2$$

⇒ Style cost function :-

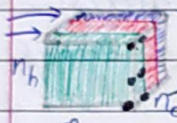
Meaning of the "style" of an image



say you are using layer l's activation to measure "style".

Define style as correlation between activations across channels.

How correlated are the activations across different channels?



correlat^n B/w the channels

here (3)

it can be 5, 7, 5 ...

★ Intuition about style of an image

⇒ Style matrix :-

Let $a^{[l]}_{i,j,k}$ = activation at (i,j,k). $G_r^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$ (H, W, C)

$\therefore G^{[l]}_{kk'}$ → It compares how related images are in channel k & k' $\quad k = 1, \dots, n_c^{[l]}$

$$G_r^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_w^{[l]}} a_{ijk}^{[l](S)} \cdot a_{ijk'}^{[l](S)}$$

$$G_r^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_w^{[l]}} a_{ijk}^{[l](G)} \cdot a_{ijk'}^{[l](G)}$$

} "Gram matrix" (G_r)

β; $J^{[l]}_{style}(S, G) = \| G_r^{[l](S)} - G_r^{[l](G)} \|_F^2$ ← Frobenius norm

$$= \frac{1}{(2 n_w^{[l]} n_H^{[l]} n_c^{[l]})^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'} \right)^2$$

(does not matter much)

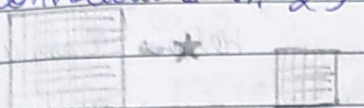
$$\therefore J_{style}(S, G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S, G)$$

$$J(G) = \alpha J_{content}(G, G) + \beta J_{style}(S, G)$$

G

⇒ 1D and 3D ÷ Convolutional Networks
1D & 3D generalization of Models.

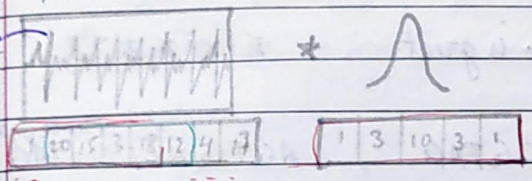
↳ Convolutions in 2D & 1D:



→ 10×10×16 / 10×10×16 * 5×5×16 ⇒ 6×6×32

This is for a 2D convolution

2D input image 14×14 | 2D filter 5×5

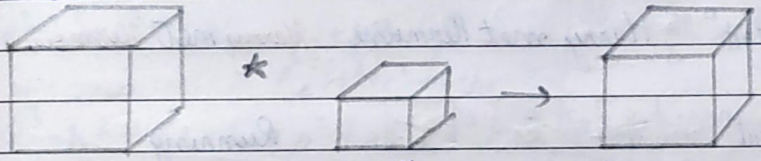


14 * 5 → 10 (if 16 filters are used ↑)
or 14×1 * 5×1 → 10×16
10×16 * 5×16 → 6*32

↳ 14 (say) | 5

* 3-D Brain CT Scan

↳ 3D convolution :



3D Volume 14×14×14×1 | 3D filter 5×5×5×1 | 3D 10×10×10×16 (say 16 filters were used)

↓ no. of channel

10×10×10×16 * 5×5×5×16 → 6×6×6×32
(32 filters)

Brain CT, MRI Scan, Movies slices of it comes under 3D convolutions.