Neural networks and Deep learning Course 1

Andrew ng

DATE

ReLu
Rectified Linear Unit

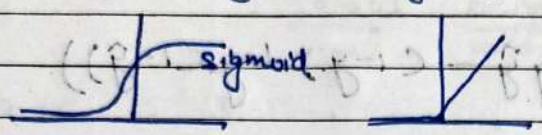
What is neural network

size → family size
#bedrooms →

Supervised learning with neural network

Input (X)	Output (y)	Application
Structured data		Unstructured Data

Why is deep learning taking off



Data
Computation
Algorithms

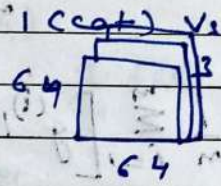
Introduction, Basics of Neural Network programming, One hidden layer Neural networks, Deep neural networks.

Introduction to neural network. (Basics of Neural network)

Binary classification :- (forward and Backward pass)

Logistic Regression 1 (cat) vs 0 (non cat)

Blue
Green
Red



$64 \times 64 \times 3 = 12288$

$n = n_x = 12288$

$x \longrightarrow y$

Notations

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

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

$m = m_{train}$ $\quad m_{test}$ = # test examples

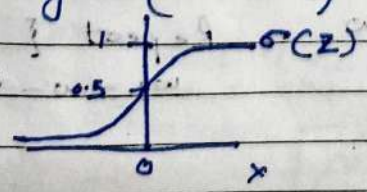
Logistic regression :-

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

$x \in \mathbb{R}^{n_x}$ $\quad 0 \le \hat{y} \le 1$

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

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



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

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

if z is large $-z$ $\sigma(z) \approx \frac{1}{1+\text{Bigno.}}$

≈ 0

Logistic Regression cost function

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

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

$z^i = w^T x^{(i)} + b$ $\quad x^{(i)}$ $\quad$ ith example

$y^{(i)}$

$z^{(i)}$

Loss error function:

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

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

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

, want $\hat{y}$ large

If $y = 0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ ← want $\log 1-\hat{y}$ —

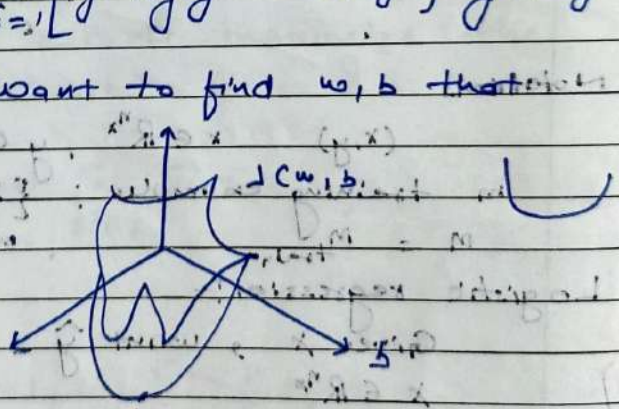
want $\hat{y}$ small

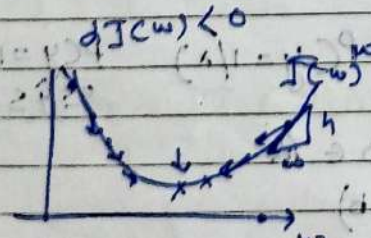
Cost function : $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}^{(i)}$

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

Gradient descent: want to find w, b that minimize $J(w, b)$

Global optimal





Repeat {

$w := w - \alpha \frac{dJ(w)}{dw}$ ← update or change

learning rate

}

$\frac{dJ(w)}{dw} = ?$

represent the slope of curve function

$w = w - \alpha \, dw$

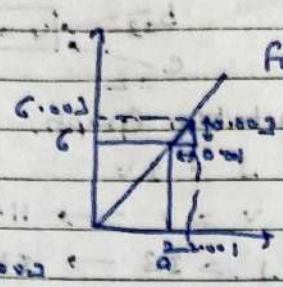
$J(w,b)$ $\quad w := w - \alpha \boxed{\frac{\partial J(w,b)}{dw}}$ $\quad \partial$ partial derivative

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

DATE

∂J $\quad dw$

Derivatives:-



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

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

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

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

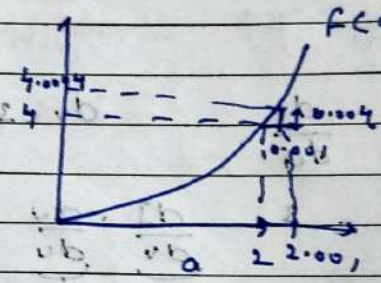
$a = 5.001$, $f(a) = 15.003$

$\frac{0.003}{0.001}$ $\quad \frac{\text{height}}{\text{width}}$

slope at $a = 5$ is 3

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

More derivative examples



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

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

(4.004004)

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

$a = 5$ $\quad f(a) = 25$

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

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

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

$f(a) = a^2$ $\quad \frac{d}{da} f(a) = 2a$ $\quad a = 2$ $\quad f(a) = 4$

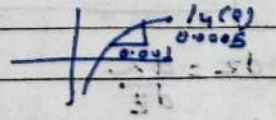
$a = 2.001$ $\quad f(a) = 4.004$

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

$a = 3.001$ $\quad f(a) \approx 8.012$

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

$\ln(a)$



$a = 2$ $\quad f(a) = 0.6931$

$a = 2.001$

$f(a) = 0.69365$

≈ 0.0005

$\frac{d}{da} (a) = \frac{1}{2}$

Computation Graph

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

$u = bc$

$v = a+u$

$J = 3v$

$\frac{dv}{da}$

DATE

$\frac{dJ}{dv} = 11$

$a = 3$, $b = 3$, $c = 2$ → $u = bc$ (6) → $v = a+u$ (11) → $J = 3v$ (33)

Derivatives with a computation Graph.

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

$J = 3v$

$v = 1 \longrightarrow 11.001$

$J = 33 \longrightarrow 33.003$

$f(a) = 3a$

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

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

$a \rightarrow v \rightarrow J$

$a = 5 \Rightarrow 5.001$

$v = 1 \longrightarrow 11.001$

$J = 33 \longrightarrow 33.003$

$\boxed{J = 3v}$

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

$\frac{dv}{da} = 1$

$\frac{dJ}{da}$ $\quad dv = 3 \quad \frac{dJ}{dv}$

$\frac{dJ}{du} = 3 = \underbrace{\frac{dJ}{dv}}_{3} \cdot \underbrace{\frac{dv}{du}}_{1}$

$u = 6 \longrightarrow 6.001$

$v = 1 \longrightarrow 11.001$

$J = 33 \longrightarrow 33.003$

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

$= 3 \quad = 2$

$b = 3 \longrightarrow 3.001$

$u = b \cdot c = 6 \longrightarrow 6.002 \quad c = 2$

$J = 33.006$

$v = 11.002$

$\frac{dJ}{dc} = \frac{dJ}{du} \cdot \frac{du}{dc} = 9$

$J = 3v$

Gradient descent

$z = w^T x + b$

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

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

x_1, w_1, x_2, w_2, b → $z = \boxed{w_1 x_1 + w_2 + b_2 + b}$ → $\boxed{\hat{y} = a = \sigma(z)}$ → $\boxed{\mathcal{L}(a,y)}$

$dz = \frac{d\mathcal{L}}{dz}$

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

$\frac{\partial h}{\partial w_1} = dw_1 = x_1 dz$

$dw_2 = x_2 dz$, $db = dz = \frac{d\mathcal{L}(a,y)}{dz}$

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

$a(1-a)$

$dw = a - y$

$= \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$

$w_1 = w_1 - \alpha \, dw_1$

$w_2 = w_2 - \alpha \, dw_2$

$b = b - \alpha \, db$

(2)

LATE

$y^{(m)}$

$y^{(2)} \ldots]$

Gradient descent on m examples.

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

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

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

$dw_1, dw_2, db^{(1)}$

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

$$\frac{\partial}{\partial w} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})}_{dw_1^{(i)} = (x^{(i)}, y^{(i)})}$$

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

for $i = 1$ to m

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

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

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

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

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

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

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

$J /= m$

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

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

$w_1 = w_1 - \alpha \, dw_1$

$w_2 = w_2 - \alpha \, dw_2$

$b = b - \alpha \, db$

<u>Vectorization</u>

$z = w^T x + b$

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

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

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

non vectorization

$z = 0$

for i in seq $(n - x)$

$z \mathrel{+}= w[i] * x[i]$

$z \mathrel{+}= b$

Vectorized

$$z = \underbrace{np.dot(w, x)}_{w^T x} + b$$

non vectorized version took more time

GPU
CPU } SIMD

DATE

More vectorization examples:

whenever possible, avoid explicity loops

$u = Av$

$u_i = \sum_i A_i v_i$

$u = np.zeros(0,1)$

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

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

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

import numpy as np

$u = np.exp(v)$

$np.log(v)$

$np.abs(v)$

$np.maximum(v, 0)$

$v**2$ $\quad 1/v$

Vectorizing Logistic Regression

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

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

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

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

$$w^T \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \end{bmatrix}$$

$$z \quad \boxed{w^T x^1 + b} \quad \boxed{w^T x^2 + b} \cdots \boxed{w^T x^m + b}$$

$1 \times m$

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

$(1,1)$ $\mathbb{R}$

"BROADCASTING"

Vectorizing Logistic Regression's Gradient Computation

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

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

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

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

$dw = 0$	$db = 0$	$db = \frac{1}{m}\sum_{i=1}^{m} dz^{(i)}$
$dw += x^{(1)} dz^{(1)}$	$db += dz^{(1)}$	$= \frac{1}{m}\, np.sum(dz)$
$dw += x^{(2)} dz^{(2)}$	$db += dz^{(2)}$	$dw = \frac{1}{m} X dz^T$
$\vdots$	$\vdots$	$= \frac{1}{m}\begin{bmatrix} \vert & & \vert \\ x^{(1)} & \ldots & x^{(m)} \\ \vert & & \vert \end{bmatrix}\begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$
$dw /= m$	$db += dz^{(m)}$	$= \frac{1}{m}[x^{(1)} dz^{(1)} + \ldots x^{(m)} dz^{(m)}]$
	$db /= m$	$= n \times 1$

<u>Broadcasting in Python</u>

cal = A.sum(axis=0)

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

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

$$\underset{(m,n)}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}} + \underset{(1,n) \rightsquigarrow (m,n)}{\begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix}} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

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

<u>General principle</u>: $\pm \quad (1,n) \rightsquigarrow (m,n)$

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

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

$$[1, 2, 3] + 100 = [101 \quad 102 \quad 103]$$

Matlab/octave bsxfun

A note on Python / numpy vectors

Explanation of Logistic Regression's cost function

DATE

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

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

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

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

$$p(y|x) = \underbrace{\hat{y}^{y}}_{y} \underbrace{(1-\hat{y})^{(1-y)}}_{(1-\hat{y})^0 = 1} \leftarrow$$

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

If $y=0$ $P(y|x) = \underbrace{\hat{y}^0}_{=1} \times (1-\hat{y})^{\overset{=1}{(1-y)}} = 1 \times (1-\hat{y}) = 1-\hat{y}$

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

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

Cost of m examples:

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

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

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

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

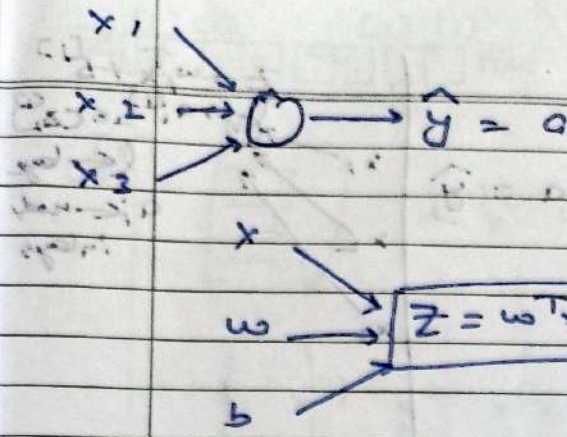
cost (minimize) : $J(w,b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$

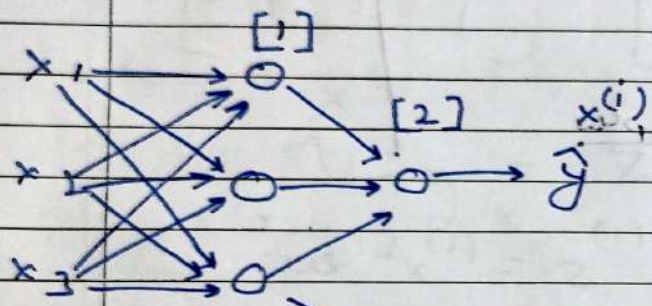
PAGE

classmate

Neural network overview

DATE

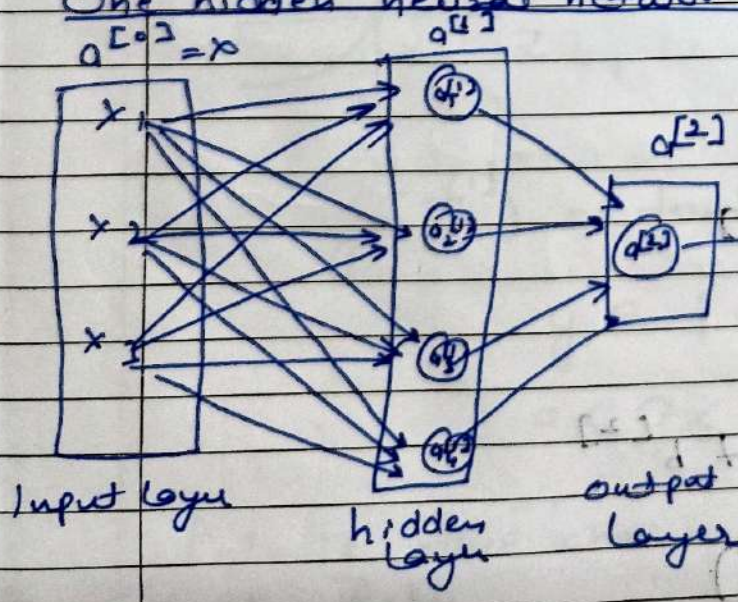




$w^{[2]} a^{[1]} + b^{[2]}$ → $a^{[2]} = \sigma(z^{[2]})$ → $L(a^{[2]}, y)$

$dz^{[2]}$ $da^{[2]}$

One hidden neural network



$a^{[0]} = x$

2 Layer Neural network

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

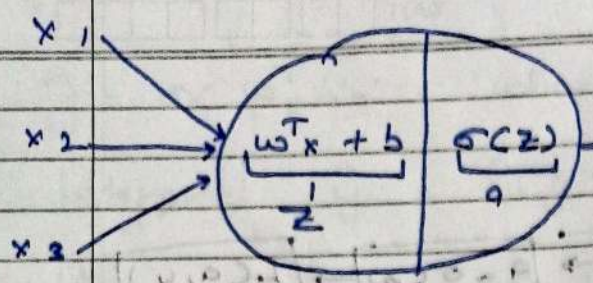
$\hat{y} = a$

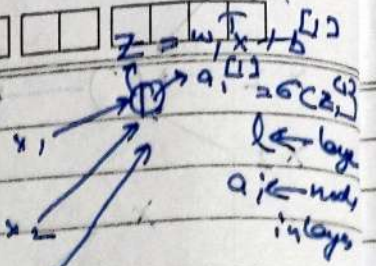
$a^{[1]} =$

PAGE

classmate

Computing Neural network output





$z = w^T x + b$
$a = \sigma(z)$

Vectorizing multiple examples

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

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

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

$\vdots$

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

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

for $i = 1$ to m

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

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

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

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

Explanation for vectorized implementation:

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

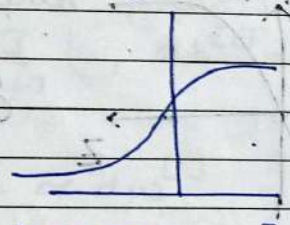
$$w^{[1]} = \begin{bmatrix} \text{—} \\ \text{—} \\ \text{—} \\ \text{—} \end{bmatrix} \quad w^{[1]}x^{(1)} = \begin{bmatrix} \vdots \end{bmatrix} \quad w^{[1]}x^{(2)} = \begin{bmatrix} \vdots \end{bmatrix} \quad w^{[1]}x^{(3)} = \begin{bmatrix} \vdots \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix}}_{X} \cdot \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = Z^{[1]}$$

$$\therefore w^{[1]}x^{(1)} = z^{(1)(1)}$$

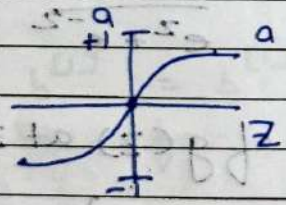
Activation functions:

sigmoid function



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

b/w 0 and 1



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

tan h is better than sigmoid function

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

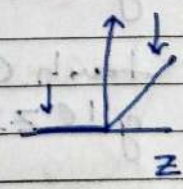
if z is very large or low then slope is very less

$$y \in \{0, 1\}$$

$$0 \le \hat{y} \le 1$$

ReLu is other choice

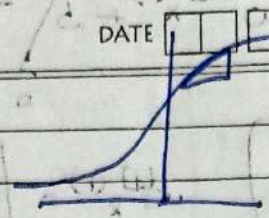
Leaky ReLu



Why non Linear activation function:-
Derivatives of activation function

DATE

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



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

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

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

$$= a(1-a)$$

$z = 10$ $g(z) \approx 1$

$\frac{d}{dz} g(z) \approx 1(1-1) \approx 0$

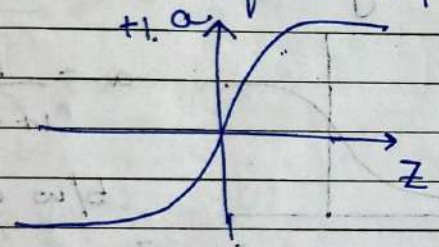
$z = -10$, $g(z) \approx 0$

$\frac{d}{dz} g(z) \approx 0.$

$z = 0$, $g(z) = \frac{1}{2}$

$\frac{d}{dz} g(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$

Tanh activation function.



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

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

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

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

$z = 10$ $\tanh(z) \approx 1$

$g'(z) \approx 0$

$z = -10$ $\tanh(z) \approx -1$

$g'(z) \approx 0$

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

$g'(z) = 1$

Relu and Leaky ReLU

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

$g'(z) = 0 \quad \text{if } z < 0$

$\quad\quad\quad 1 \quad \text{if } z > 0$

DATE

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

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

Gradient descent for one Neural network

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

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

Gradient descent

Repeat $\quad$ compute predicts $(\hat{y}^{(i)}, i = 1, \dots m)$

$$dw^{[1]} = \frac{dJ}{dw^{[1]}}, \quad db^{[1]} = \frac{dJ}{db^{[1]}}, \dots$$

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

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

Formulas for computing derivatives.

Back propagation intuition:

Random Initialization:

what happen if you initialize weights to zero?

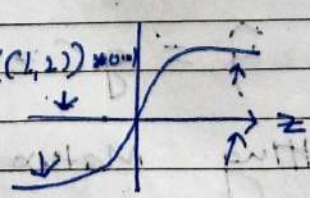
Random initialization:

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

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

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

$b^{[2]} = 0$



Deep L-Layers Neural network
4 Layer hidden

DATE

L (no. of layers) (0, 1, 2, 3...)

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

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

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

Forward propagation in a Deep Neural network

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

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

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

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

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

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

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

Vectorized

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

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

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

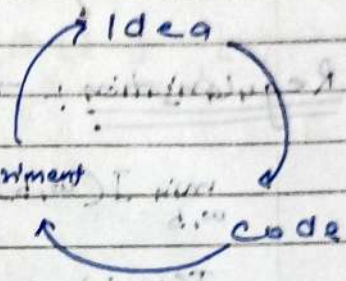
$$A^{[2]} = g^{[2]}[Z^{[2]}]$$

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

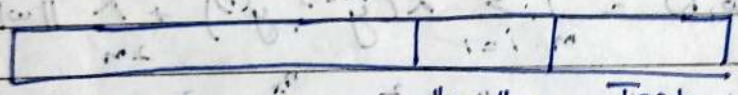
Getting Matrix dimensions Right

Why deep representation?
Building blocks of deep neural networks:
Forward and backward propagation
Parameters Vs Hyperparameters.
What does this have to do with the brain?

IMPROVING Deep neural network
Train / Dev / Test series
Applied ML is highly iterative process
Layers
hidden layers
learning rates
activation functions



Data



Training set "dev" set Test

Previous era 70 / 30 %. 60 / 20 / 20 %

Bigdata 100 - 1000 - 10000 98 / 1 / 1 %
99.5 / .4 / .1 %

Mismatched train / test distribution

Bias / Variance

	high bias	just right	high variance
Train set error	1%	15%	15%
Dev set error	11%	16%	30%
	high variance	high bias	high bias high variance

0.5 "
1%
Low bias
Low variance

optimal error ≈ 0%.
15 % Blur images

Basic Recipe of Machine Learning:

High bias? $\xrightarrow{Y}$ Bigger network
Train layer
(NN architecture search)

training data

N

High variance?

dev set $\xrightarrow{Y}$ More data
Regularization
(NN architecture search)

N

Done

"Bias variance tradeoff"

$\downarrow \uparrow$ $\downarrow \uparrow$

Regularization: to prevent overfitting

Logistic regression

$$\min_{w,b} J(w,b)$$

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

L_2 regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L_1 regularization $\frac{\lambda}{m}\sum_{i=1}^{n_x} |w| = \frac{\lambda}{2m}\|w\|_1$

w will be sparse

λ = regularization parameter

Neural network

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

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

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

"fobenius norm"

$\|\cdot\|_2^2$ $\|\cdot\|_F^2$

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

classmate

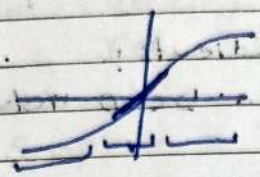
" weight decay"

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

DATE

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

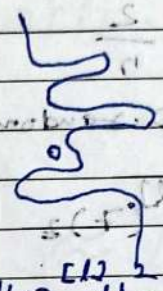
Why regularization reduces overfitting?



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

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

Every layer $\approx$ linear



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

Drop regularization:

Implementing dropout ("Inverted dropout")

Understanding dropout:

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

shrink weights L_2

Other regularization methods

Early stopping

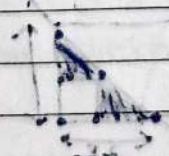
optimize cost function

Gradient

Not overfit

Regularization

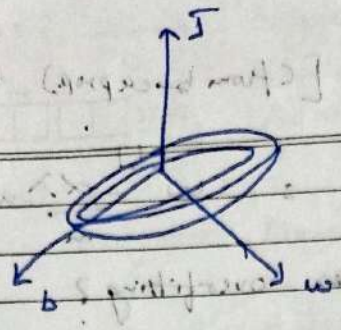
Orthogonalization



Normalizing Input:

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

$$x := x - \mu \qquad x /= \sigma^2$$



Vanishing / exploding gradients:
weight initialization in a Deep network

Larger $n \longrightarrow$ Smaller w_i

$Var(w_i) = \cancel{\frac{1}{n}} \quad \frac{2}{n}$

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

ReLu $\quad g^{[l]}(z) = Relu(z)$

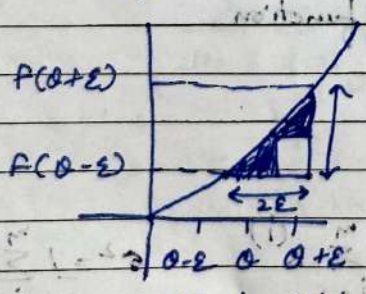
$w^{[l]}$ other variance

tanh activation function

tanh: $\sqrt{\frac{1}{n^{[l-1]}}}$

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

Numerical approximations of Gradients:



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

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

$\varepsilon = 0.01$

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

approx : 0.0001

Gradient checking

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

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

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

Gradient checking (Grad check)

$J(\theta) = J(\theta_1, \theta_2, \theta_3)$

for each i

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

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

$$d\theta_{approx} \overset{?}{\approx} d\theta$$

check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx 10^{-7}$ — great!

10^{-5}, 10^{-3} — worry

$\varepsilon = 10^{-7}$

Gradient checking Implementation notes

- Donot use in training — Only to debug
- If algorithm fails grad check, look at components to try to identify debug

$db^{[l]}$ $dw^{[l]}$

- Remember regularization
- Doesn't work with dropout keep prob = 1.0
- Run at random initialization, perhaps again after some training.

$w, b \approx 0$

Optimization Algorithm: Mini-batch gradient descent

classmate

Batch vs mini-batch gradient descent

Vectorization allows us to efficiently computer m examples.

$x^{(i)}, z^{[l]}, X^{[t]}, Y^{[t]}$

for $t = 1, \ldots 5000$

~~t step~~

1 step of gradient descent using $X^{[t]}, Y^{[t]}$

forward prop on $X^{[t]}$

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

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

$$\vdots$$

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

Vectorized independent 1000 examples

compute cost $J = \frac{1}{1000} \sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \| W^{[l]} \|_F^2$

Back prop to compute gradients

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

1 epoch

Choosing your mini-batch size

If mini batch size $= m$: Batch gradient descent $m \leq 2000$

If mini batch size $= 1$ stochastic gradient descent

(64, 128, 256, 512)
2^6 2^7 2^8 2^9

Make sure minibatch fit in CPU/GPU memory

$X^{[t]}, Y^{[t]}$

Exponentially weighted averages.

Understanding Exponentially weighted averages

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$

Bias correction:-

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$

$\frac{v_t}{1-\beta^t}$

$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

Gradient descent with momentum:-

Momentum

On iteration t

Compute dw, db on current mini-batch

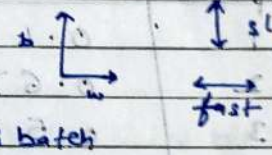
$V_{dw} = \beta V_{dw} + (1-\beta)\boxed{dw}$ → Acceleration

$V_{db} = \beta V_{dw} + (1-\beta)\boxed{db}$ "$V_\theta = \beta V_{\theta T} + (1-\beta)\theta_t$"

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

Limitation Velocity

RMSprop on iteration t:



Compute dw, db on current mini batch

$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$

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

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

Adam optimization algorithm:-

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

On iteration t:

Compute dw, db using mini-batch

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

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

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

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

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

classmate PAGE

~~Hyperma~~ Hyperparameters choice :

α : needs to be tune

β_1 : 0.9 $\quad dw$

β_2 : 0.999 $\quad dw^2$

ε : 10^{-8}

Adam Adaptive momentum estimation

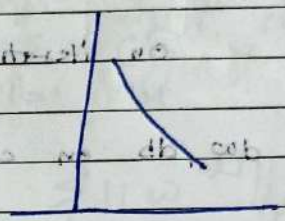
Learning rate decay : slowly reduce

1 pass = 1 pass through data

$x^{(1)}$ $x^{(2)}$

$$\alpha = \frac{1}{1 + \text{decay rate} * \text{epoch-num}} \alpha_0$$

Epoch	α
1	0.4
2	0.67
3	0.5
4	0.4
⋮	⋮



other learning rate decay methods

$\alpha = 0.95^{\text{epoch-num}} \alpha_0$ — exponential decay

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

Measured decay

Hyperparameter tuning

↑ α

β

$\beta_1, \beta_2, \varepsilon$

layers

hidden units

Learning rate decay

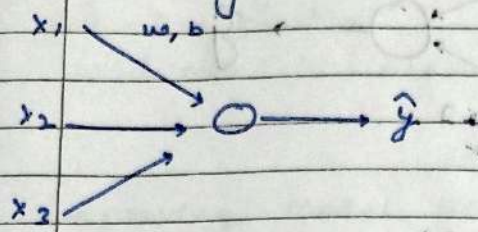
mini batch size

Baby sitting one model ; Training many models in parallel

How much computation approaches you have

DATE

Normalizing activations in a network

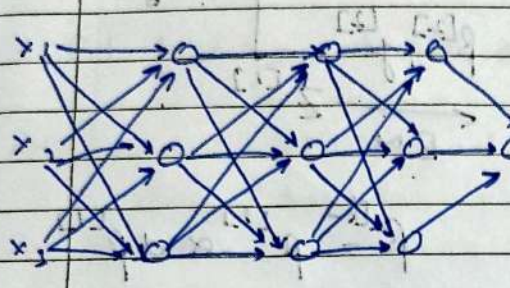


$$\mu = \frac{1}{m} \sum_i x^{(i)}$$

$$x = x - \mu$$

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

$$x = x / \sigma^2$$



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

Normalize $z^{[2]}$

Implementing Batch Norm

Given some intermediate values in NN

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

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

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

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

if $\gamma = \sqrt{\sigma^2 + \varepsilon}$

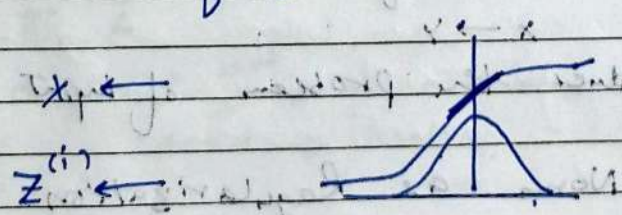
$\beta = \mu$

then $\tilde{z}^{(i)} = z^{(i)}$

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

Learning parameter of Model

Use $\tilde{z}^{(i)}$ if $z^{[l](i)}$

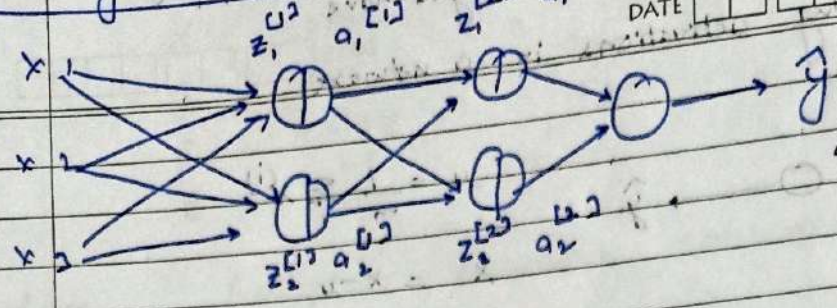


PAGE

classmate

Fitting Batch Norm into Neural networks

DATE



$$X \xrightarrow{w^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[\text{Batch norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{Z}^{[1]})$$

$$\xrightarrow{w^{[2]}, b^{[2]}} Z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{Z}^{[2]} \longrightarrow a^{[2]}$$

$d\beta^{[1]}$ $\qquad \beta^{[1]} = \beta^{[1]} - \alpha\, d\beta^{[1]}$

If using batch-normalization
working with mini batches

$$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \longrightarrow g \; Z^{[1]} \; a^{[1]} \xrightarrow{w^{[2]}, b^{[2]}} Z^{[2]} \longrightarrow \cdots$$

$$X^{\{2\}} \longrightarrow Z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \longrightarrow \cdots$$

$$X^{\{3\}} \longrightarrow \cdots$$

Why does Batch norm work?

covariance shift

$x \rightarrow y$

Reduces the problem of input value changes

Batch Norm as Regularization

one mini batch at a time

PAGE

classmate

Batch norm at test ~~time~~ time

μ, σ^2, Z^{i}_{norm}, $\tilde{Z}^{(i)}$

DATE

Softmax Regression:-

Activation function

hard max Loss function

Improving Model performance

Orthogonalization TV tuning example car

speed Steering

Accelerator

Brake

angle

Fit training set well on cost function

Fit dev set well on cost function Big hidden set Regularization

Fit test set Bigger dev set

Bigger network

Performance well in real world Adam

Change dev set or cost function.

$\boxed{\text{avoid early stopping}}$

Single number evaluation metric

classifier Precision Recall F1 score

F1 score = Average of P & R

$\left(\frac{2}{\frac{1}{P}+\frac{1}{R}}\right)$ 'Harmonic mean'

Dev set + single real number evaluation matrix

speed up iteration

Satisfying and optimizing metrics

PAGE

classmate

Cost = accuracy − 0.5 × running time

Maximize accuracy

running time ≤ 100ms

N metrics: 1 optimizing

N−1 ~~satisfying~~ satisfying

Trigger words

Alexa, ok Google, Hey Siri

accuracy

~~#~~false positive

maximize accuracy

st ≤ 1 false +ve

Train / Dev / test set Distributions

cat classification dev / test set

Guideline: choose 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 and test sets

Old way of splitting data

70%	30%
Train	Test

60%	20%	20%
Train	Dev	Test

98%	1%	1%
Train		

1,000,000

Size of test set

→ set your test set to be big enough to give high confidence in the overall performance of your system.

when to change dev / test state

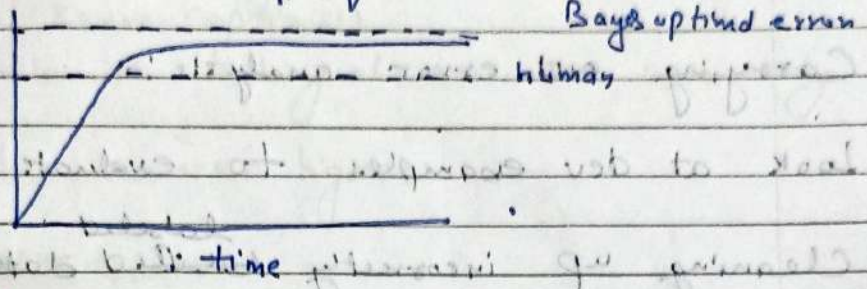
$$\text{Error} = \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathbb{1}\{ y_{pred}^{(i)} \neq y^{(i)} \}$$

pred value (0/1)

$w^{(i)} = 1$ if $x^{(i)}$ is non porn

10 if $x^{(i)}$ is porn

Why human level performance!



Avoidable bias

Understanding human level performance

Human-level error as a proxy for Bayes error

0%		
	Human-level error	
Bias		Avoidable bias
	Training error	
		variance
	Dev error	

Surpassing human level performance!

Team human	0.5%	0.5% 0.4% 0.2% 0.3%
One human	1%	
Training error	0.6%	0.3%
Dev error	0.8%	0.4%

Improving your model performance

Two fundamental assumption of supervised learning

1. You can fit the training set pretty well
2. The training set performance generalizes pretty well to the dev / test set

Reducing (avoidable) bias and variance

Human level — Train bigger model
Train longer / better optimization algo
— momentum, RMSprop, Adam

Train error — NN architecture / hyperparameter search

Dev error — More data
Regularization
l_2, dropout, data augmentation

Carrying out error analysis :-

Look at dev examples to evaluate ideas

Cleaning up incorrectly labeled data

Training set — Incorrectly labeled

Overall dev set error	10%
Errors due to incorrect labels	0.6%
Errors due to other causes	9.4%

Correcting incorrect dev / test set eg.
Apply same process to your dev and test sets to make sure they continue to come from the same distribution.

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

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

Probability → classical theory approch
→ statistical approch

$P(A) = \frac{n(3,6)}{n(1,2,3,4)}$

Classical approch

① Random experiment :- more than one outcome
- No prediction.

② Sample space

2 coins [HH, TT, HT, HH]
3 coins HHH HHT HTH HTT, THH, THT, TTH, TTT)

Total outcomes for single dyes = 6
Total out comes for two diec = 6 × 6 = 36
$S = \{x, y\} : x, y \in \{1, 2, 3, 4, 5, 6\}$

{(1,1), (1,2), (1,3) ... (1,6) (2,1) ... (2,6), (3,1) ... (3,6)
(4,1), ... (4,2) ... (4,6), (5,1) ... (5,6) (6,1) ... (6,6)}

Event :- A subset of sample space is called event

{HH HT TH TT}

A = A head apair {HT, TH}

B = Atleast a head appear = {HT, TH, HH}

C = At most 2 tail appear

Data preparation ~~of~~ for machine learning :-

Data collection → Data preprocessing → Data Transformation

Data Augumentation :-

Data Labeling

snorker

Data preprocessing :-

formatting, Data cleaning

Treatment for missing value

Mean, median, mode

Global constant

Binnig :- sort in increasing order

Bin size (3)

Bin mean, Bin median, bin boundary

clustering, Regression, Sampling

Data transformation

Normalization - Min-max normalization

Z-score normalization

$$v' = \frac{v - mean_A}{std\ Dev_A}$$

Decimal scaling $v' = \frac{v}{10^j}$

Aggregation, Decomposition

Data visulization

Activation functions Linear activation function

non Linear activation function

Linear → Activation function → non Linear

Types of Activation function

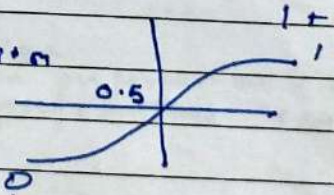
(1) Thershold / Binary ~~set~~ step / Heavy side function

(2) Sigmoid activation function

$$f(x) = \frac{1}{1+e^{-x}}$$

range b/w 0 and one

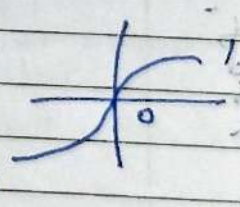
- used in logistic regression and neural network



(3) Tanh Hyperbolic tangent function

~~bette~~ better than sigmoid

- range b/w -1 to 1



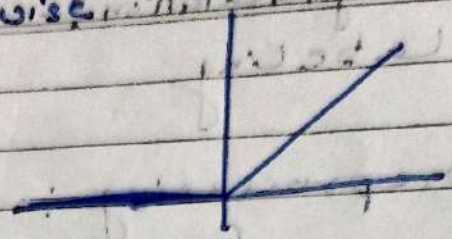
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Relu | (Rectified Linear unit activation function

used in CNN.

It gives an output x if x is +ve and zero otherwise.

DATE



Half rectified

- create dead neuron

Leaky Relu

0.01 to α

Terms use in Artificial neural network

weights / parameter - strength between connection between nodes, Multilayer artificial neural network

Activation function L-18

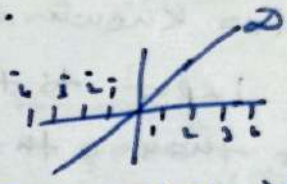
- Activation function is used to get the output of a node.
- It convert a input signal of a node in ANN to an o/p signal
- Activation function is used to maps the resulting values in b/w 0 to 1 or −1 to 1 depending upon the type of function used.

Activation function ⟶ Linear activation function
non linear activation function

Linear activation function :- eg linear or identity activation function. The output of the function will not be confined b/w any range

- NN could not be able to learn.

$f(x) = x$



Non Linear activation function :- It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output

- It make the network more powerful and add ability to it to learn something complex and complicated from data

eg logistic / sigmoid, tanh, Relu. etc

- The activation function help the network use the important information and supress the irrelevant information

A.f decides whether a neuron should be activated or not by calculating the weight sum and further added bias to it.

It introduces non-linearity into output of a neuron.

- If we donot apply activation function then the output signal would be simply linear function (1 degree polynomial)

* non linear function - degree more than one

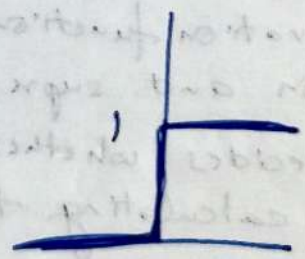
- Linear function is easy to solve but they are limited in ~~capacity~~ their complexity & have less power.

- Without activation function our model cannot learn and model complicated data such as image video, audio, speed etc.
- We need ANN to learn and represent almost anything and any arbitrary complex function that map an input to output.

Types of Activation function

1. Threshhold Activation function (Binary step function)
   - A Binary step function is a threshold based activation function
   - Also known as Heaviside function
- If the input to the binary step activation function is greater than a threshold, then the neuron is activated else it is ~~deca~~ deactivated
- Binary step activation function are useful for binary classification scheme
- The activation function is useful when the input pattern can only belong to one or two groups, ~~that~~ is binary classification

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



<u>Limitation</u> This function will not be useful where there are multiple classes in the target variable

2. Sigmoid (Logistic function) It is one of the most widely used non linear activation function
   - It transfer the value b/w the range 0 and 1

$$f(x) = \frac{1}{1+e^{-x}}$$

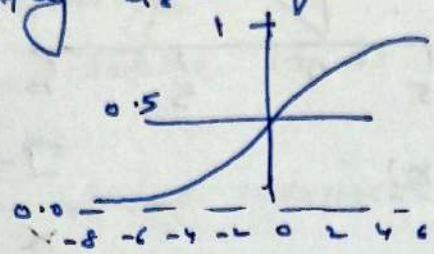
$x \to -\infty \Rightarrow f(x) \to 0$

$x \to \infty \quad f(x) \to 1$

$x = 0 \quad f(x) = 0.5$

A sigmoid function is a mathematical function having S-shaped curve or sigmoid curve which ranges b/w 0 and 1

- It is used for models where we need to predict the probability as an output
- It is mostly used for ~~binary~~ multiclass classification



* The output is not zero centered.

- It is used in major machine learning algorithms such as logistic regression and neural networks

$$\text{probability of event } (x) = \frac{1}{1+e^{-x}}$$

It converts a value x into a probability of something happening

such as, the probability p of rainfall given the weather condition.

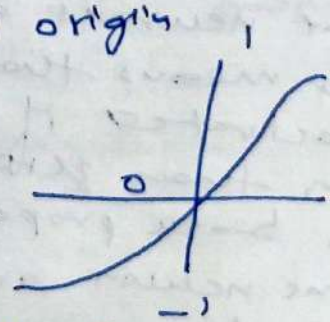
If $P(x) = 0.5$, the output is marked as true

If $P < 0.5$ output is false

3. <u>Tanh - hyperbolic target function</u>

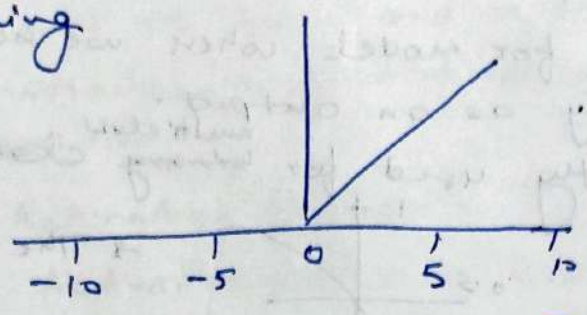
- It is similar to sigmoid but better in Performance
- It is also a nonlinear activation function
- The function range b/w $(-1, 1)$
- It is symmetric around the origin

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



- The advantage is that the -ve input will be mapped strongly -ve & zero input will be mapped near zero in the tanh graph
- tanh used classification b/w two classes

4. Relu: Mostly used activation function since it is used in almost all the CNN or deep learning



$f(x) = \max(0, x)$

or

$f(x) = x, \; x \geq 0$

$= 0, \; x < 0$

It gives an output x if x is +ve and 0 otherwise

- It range b/w 0 to ∞
- It is half rectified

$f(x) = x$ when x is above or equal to zero

- Limitation :- All the -ve values become zero immediately which ↓s the ability of the model to fit or train from the data property
- It is less computationally expensive than tanh & sigmoid because it involve simple mathematical operation
- The main advantage of using ReLu function over other activation function is that it does not activate all the neurons at the same time
- This means that the ~~neu~~ neurons will only be deactivated if the output of the linear transformation is less than zero.
- In back propagation process, the weight & biases for some neuron are not not updated. This can create dead neuron which never get activated
- This is taken care by the Leaky Relu function

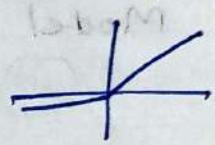
5. Leaky Relu – It is an improved version of Relu function

– for Relu function the gradient is 0 for $x<0$, which would deactivate the neurons in that region. leaky Relu is defined to address this problem

– Instead of defining the Relu function as a negative values of x, we define it as an extremely small linear component of x

– The mathematical expression for leaky Relu function is

$$f(x) = 0.01x, \quad x<0$$



$$= x, \quad x \geq 0$$

By making the small modification, the gradient of the left side of graph comes out to be a nonzero value

– so there would no layer encounter dead neuron in that region.

L-23

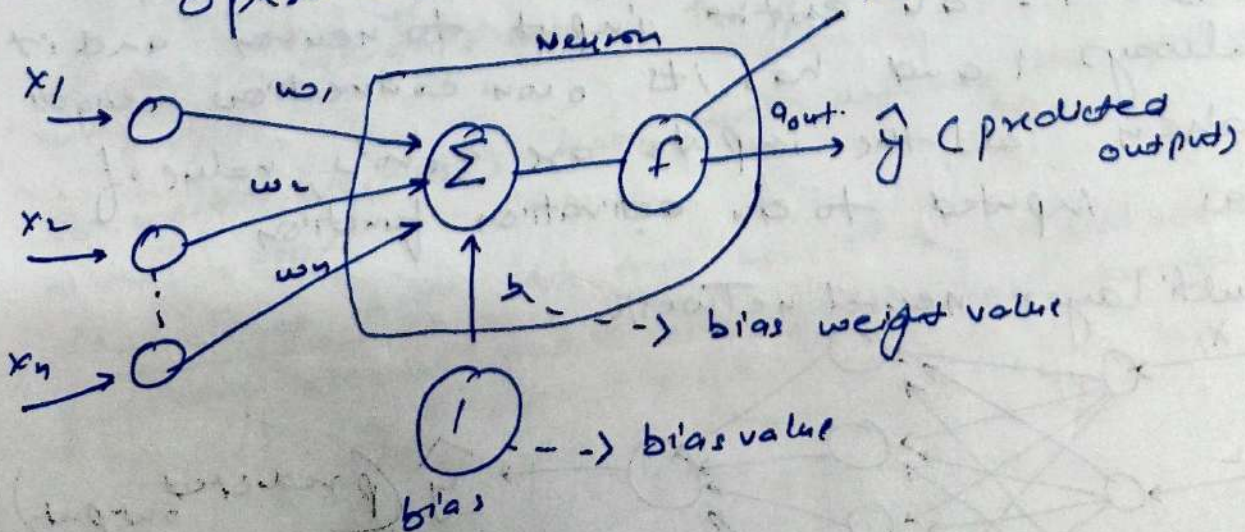
Terms used in Artificial Neural network

Neuron / Node

– It's basic unit of a neural network

– It gets certain number of input & a bias value

Operation at one neuron of a neural n/w



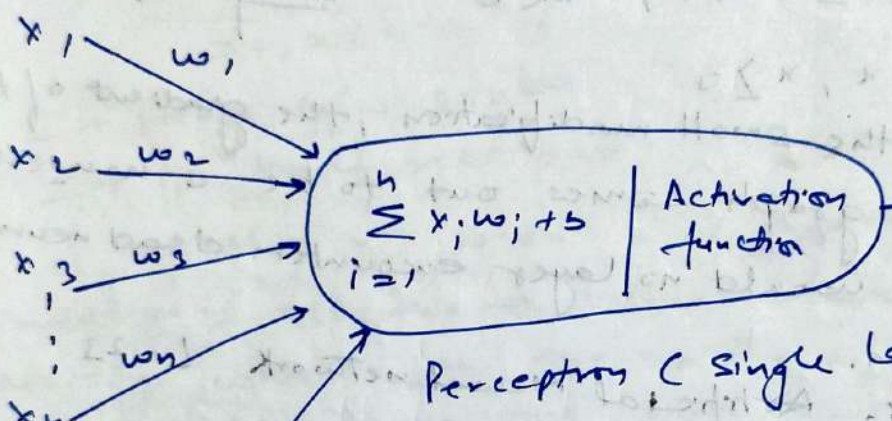
$$Z = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n + b$$

$$Z = \sum_{i=1}^{n} x_i w_i + b$$

$$a_{out} = f(Z) = \frac{1}{1+e^{-Z}}$$ (for sigmoid activation fun)

predicted output $\boxed{\hat{y} = \frac{1}{1+e^{-Z}}}$

Model of ANN



Perceptron (single layer ANN

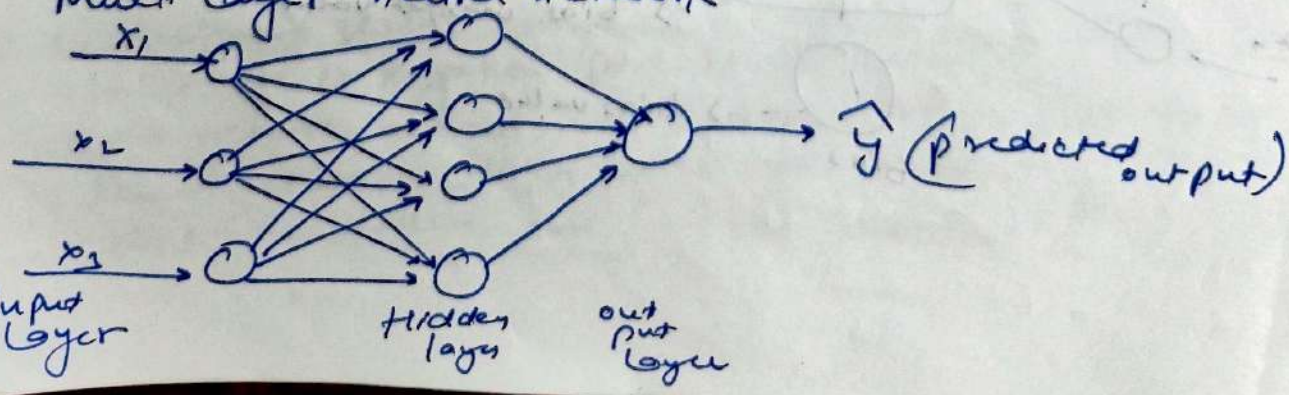
$x_1, x_2, \ldots x_n \rightarrow$ input

$w_1, w_2, \ldots w_n \rightarrow$ weight of connection

$b \rightarrow$ bias

Bias is an extra input to neuron and it is always 1 and has its own connection weight when all the inputs are zero's, value of bias inputed to an activation function

Multi layer neural network



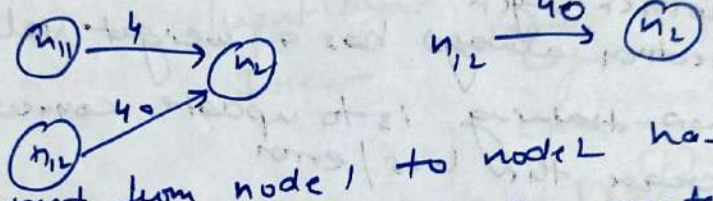
* There may be single or multiple output
* There can be multiple no. of hidden layers

<u>Connection</u>

- It connects one neuron in one layer to another neurons in another layer
- A connection always has a weight value associated with it
- Goal of ANN training is to update connections weight values to decrease the less / error

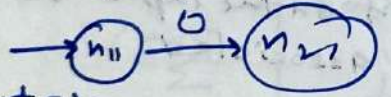
<u>Weights / parameters</u>

A weight represent the strength of connection b/w nodes / neurons



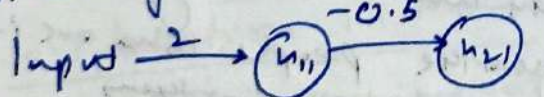
If the weight from node 1 to node 2 has greater magn
- Thus, it means that neuron 1 has greater influence over neuron 2

A weight brings down the importance of the input value



Input = 2 $\quad 2 \times 0 = 0$

- weights near zero means changing this input will not change the output



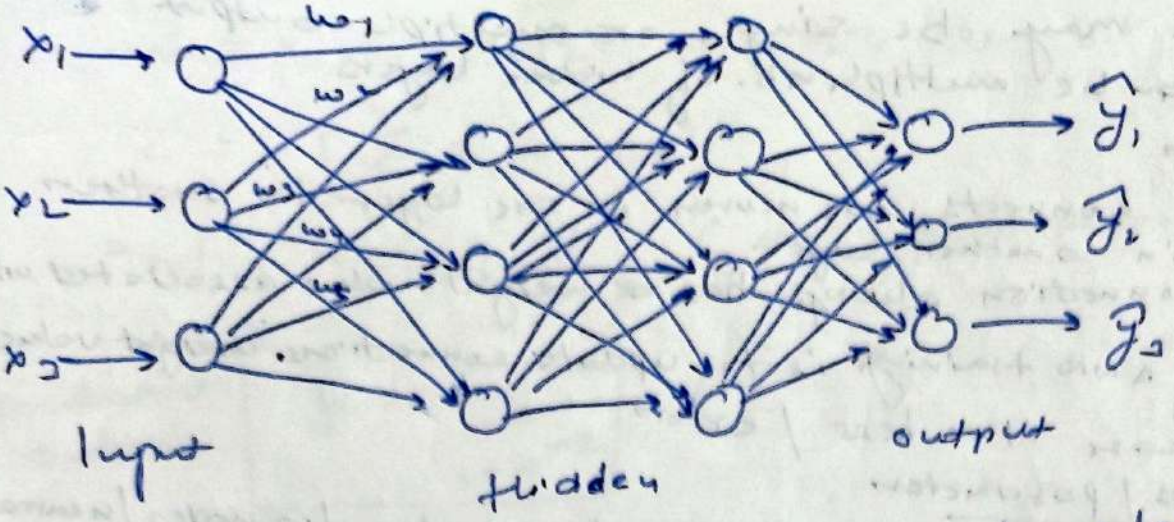
$2 \times (-0.5) = -1$

negative weights means ↑ing this input will ↓ the output.

So weight decides how much influence the input will have on the output

<u>Multilayer ANN</u> An Ann consist of artificial neurons or processing element and is organized in three interconnected layers

1. Input layer (single)
2. Hidden layer (one or more)
3. Output layer (single)



Connection :- It connects one neuron in one layer to another neuron in another layer

A connection always has a weight value associated with it

Goal of ANN training is to update connection weight values to decrease the loss/error

Input Layer

- First layer in the NN.
- This contain the neurons which receive input from the outside world and passes them on to the next layer
- It doesnot apply any operation on the input value and has no weights and biases values associated

Hidden Layer These neurons are in b/w input & output layer

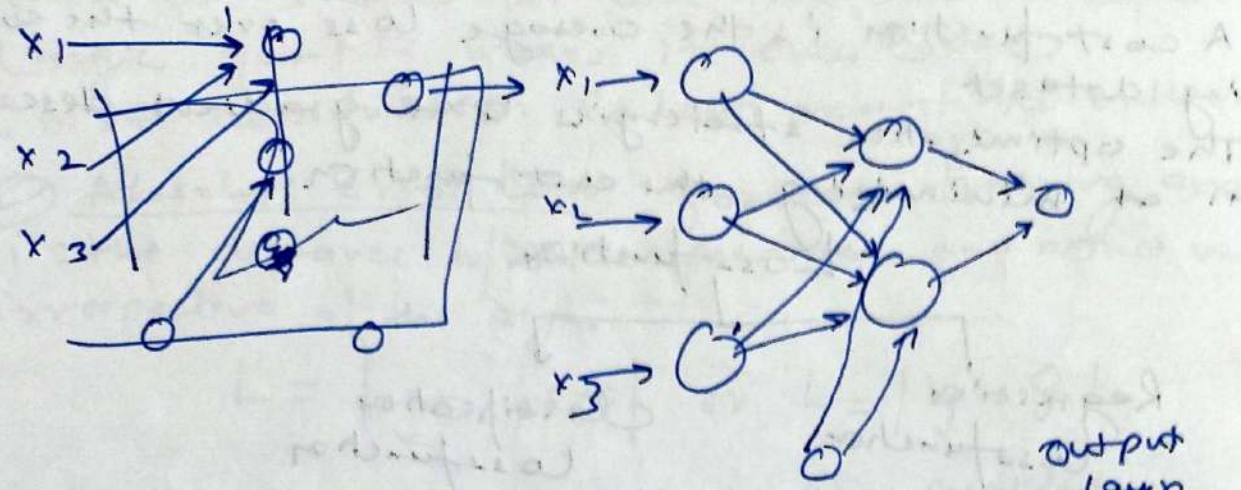
- The job of hidden layer is to transform the input into some thing that output unit can use in some way
- These neurons are hidden from the people who are interfacing with the system & acts as a blackbox for them
- On increasing the hidden layer with neurons, the system computation and processing power can be increased but the training process of system gets more complex at the same time
- Hidden layer neurons can send information to output layer

Output Layer It is the last layer in the NN. & it receives input from the last hidden layer

- with this layer we can get desired number of values & in a desired range

Most ANN are fully connected that means each hidden neuron is fully connected to every neuron in the previous layer (input) & to the next layer (output layer)

Representation of weights and ~~ANN~~ bias in ANN



Input layer 3 neurons Hidden layer 2 neurons output layer 1 neurons

$$Z_1^{[1]} = x_1 w_{1,1} + x_2 w_{1,2} + x_3 w_{1,3} + 1 \times b_1^{[1]}$$

$$a_1^{[1]} = \sigma(Z_1^{[1]}) = \cancel{1+e^{-}} \; \frac{1}{1-e^{-Z_1^{[1]}}}$$ for sigmoid activation function

Linear Σ | non linear activation function

No. of parameters in ANN

Loss function in Machine

Loss function is a method of evaluating how well machine learning algorithm models given datasets

If the prediction of models are wrong, loss function will output a higher number and if predictions are good, it will out put a lower number

To improve the output (predictions) of machine learning algorithm, loss function help us

- Machine learns by means of loss function
- with the help of some optimization function like gradient descent, loss function learn to reduce the error in prediction

- A loss function / error function is for a single training example
- A cost function is the average loss over the entire training dataset
- The optimization strategies like gradient descent aim at minimizing the cost function

Loss function

- Regression loss function
- Classification loss function

<u>Regression</u>: It deals with predicting a continuous value for example future salary of an employee future price of house etc

<u>classification</u> :- Refers to assigning an object into one or more than one classes

Regression Loss function

<u>Squared error loss function</u> :- (L2 loss) / Quadratic loss

<u>Squared error loss</u> for each training example is the square of the difference b/w the actual and the predicted values

$$L = (y - f(x))^2 \qquad y \rightarrow \text{actual value}$$

$$\text{or } L = (y - \hat{y})^2 \qquad \hat{y} \rightarrow \text{predicted value}$$

* Easier to solve.

In regression machine learning algorithm, our goal is to minimize this map, which will provide us with the best line that goes through all the points

- Due to squaring prediction which are far away from actual values are penalized heavily in comparing to less deviated predictions
- on

But if our data is prone to many outliers then we should not use MSE because squaring a large quantity makes it even large.
- MSE cost function is less robust to outliers

(2) <u>Absolute error loss</u> for each training example is the distance b/w the predicted and actual values, irrespective of the sign

$$L = |y - f(x)| \quad \text{or} \quad L = |y - \hat{y}|$$

y = actual, $\hat{y}$ = predicted.

The corresponding cost function is the mean of the absolute errors (MAE)

$$MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n} \qquad n = \text{no. of observed samples}$$

MAE is measured as the average of sum of absolute difference b/w predicted and absolute observations.

<u>Advantage</u> Unlike MSE, MAE is more robust to outliers since it doesnot make use of square

<u>Disadvantage</u> Handling the absolute or modulus operator in mathematical equations is not easy

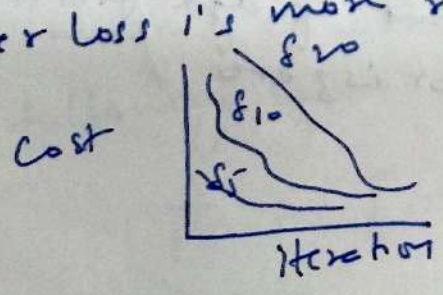
(3) <u>Huber loss</u> - It combine the best properties of MSE and MAE.

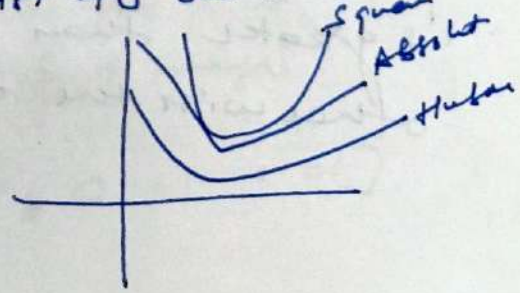
It is quadratic for smaller errors & is linear otherwise

It is identified by its delta parameter

$$L_\delta = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \le \delta \\ \delta(y - f(x)) - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

Huber loss is more robust to outliers than MSE





Huber loss is more robust to outlier than MSE
Drawback:- we might need to train hyperparameters delta which is an iteration process

Classification loss function

1. Hinge loss / Multi class SVM loss -
- It is popular with SVM.
- These are used for training the classifier
- Binary classification, loss function

Loss function

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

$$= \sum_{j = y_i} \max(0, s_j - s_{y_i} + 1)$$

	cat	car	frog
cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
loss	2.9	0	12.9

$5.1 - 3.2 + 1 = 2.9$

$-1.7 - 3.2 + 1 = -3.9 \approx 0$

In the sum above, we compare the score of right category against those for the wrong category
- This is done by subtracting the wrong class value & the right class value if the score difference is greater than some fudge factor (1) otherwise the error is zero

In simple terms, the score of correct category should be greater than sum of scores of all incorrect categories by some safety margin (usually one)

Cross entropy lossfunction

Animal image	cat	dog	frog
Label.	[1 0 0]	[0 1 0]	[0 0 1]

Probability distribution of each image

- each image is labeled with one corresponding animal using the one hot encoding.
- Let we designed a machine learning model that classifies those images

Initial prediction

first it classify the first image as

$$\hat{P} = [0.4 \quad 0.3 \quad 0.3]$$

It means model says that the first image is

40% cat | 30% frog
30% dog

But this estimation is not very precise about what animal the first image has.

Label for cat $(P) = [1 \; 0 \; 0]$

$$\text{cross entropy} = -(1 \times \log 0.4 + 0 \times \log 0.3 + 0 \times \log 0.3)$$

$$= -\log 0.4$$

$$\approx 0.916 \; (>0 - \text{error/loss})$$

when model is well trained, let it predict the following value for first image

$$\hat{P} = [0.98 \quad 0.01 \quad 0.01]$$

$\therefore$ cross entropy $= -(1 \times \log 0.98 + 0 \times \log 0.01 + 0 \times \log 0.01)$

$= -\log 0.98$

≈ 0.02 (> 0 — error low

Comparision to first prediction of image the cross entropy for 2nd prediction is much lower

That means the cross entropy goes down as the machine learning model prediction gets more and more accurate

Cross entropy (for perfect prediction.)

<u>Binary cross entropy lossfunction</u> It is cross entropy with two classes. It is used for binary classification

formula for binary cross entropy

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

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

$$\boxed{L = -\sum_{i=1}^{2} P_i \log \hat{P}_i}$$

cat — P — 60%

Dog — (1-P) — 40%

loss function

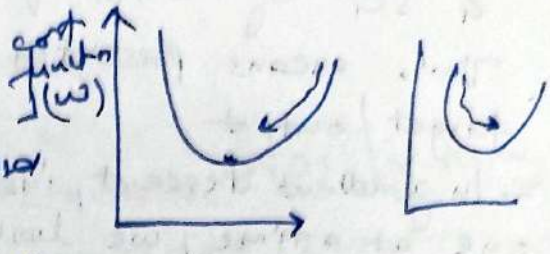
For multi class classification

$$L = -\sum_{i=1}^{m} P_i \log \hat{P}_i$$

Usually 1st a sigmoid activation function is applied to get the predicted output in probabilities & then cross entropy loss is calculated.

Gradient descent

slope → act of moving downward



- Gradient descent is one of the most popular and widely used optimization algorithm used for training machine learning models
- The optimizer is a search technique, which is used to update weight in the model
- for a given machine learning model with parameters (weights & biases) and a cost function (used to evaluate the model) * coefficient (parameters) – Linear Regression

The aim of our learning problem (ANN) is to find a good set of weights for our model which minimizes the cost function.

So Gradient descent is an optimization algorithms that finds the optimal weights that reduces prediction error (cost function)

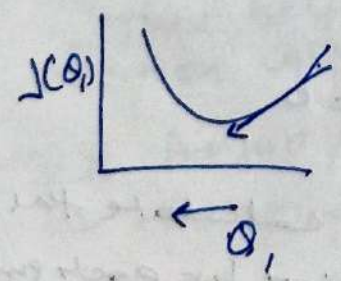
Let cost function is mean of squared Error

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

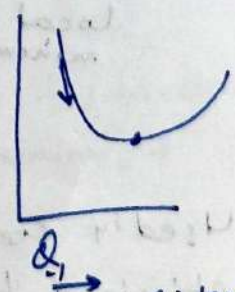
y_i – actual o/p value

$$J(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$\hat{y}_i$ – predicted o/p value



$$\theta_1 = \theta_1 - \eta \frac{\partial}{\partial \theta_1} J(\theta_1)$$



Gradient descent is an optimization algorithm (iteration) used to minimize cost function by iteratively moving in the direction of steepest descent as defined by the -ve of the gradient

During training of Neural network our aim is to

a set of values for w such that $(y_i - \hat{y}_i)^2$ is small
This means prediction of learning model will close to the target / output

- In gradient descent, instead of group through every weight one at a time, we look at the angle of the cost function line
- Gradient descent works well when we have a convex curve

(updated weight) $w' = w - \eta \frac{\partial J}{\partial w}$ η = learning rate

when $\frac{\partial J}{\partial w} = -ve$

$w' = w + \eta \frac{\partial J}{\partial w}$

$\frac{\partial J}{\partial w} = +ve$

$w' = w - \eta \frac{\partial J}{\partial w}$

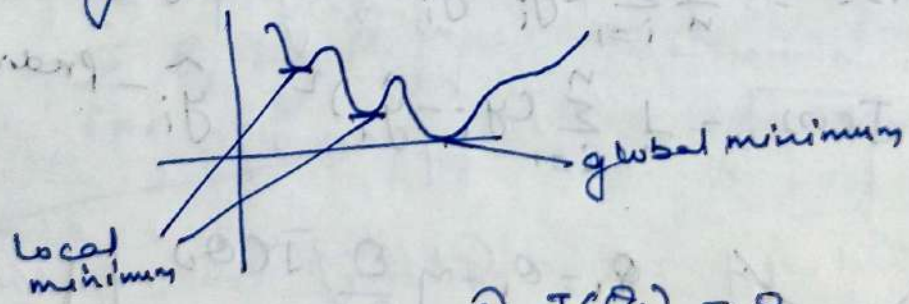
- Gradient descent only works for problem which have a well defined convex optimization problem.

* convex optimization problem may also consists of many minimized points.

Global minimum — lowest point

Local minima — other point

In gradient descent our aim is to find global minimum while avoiding local minima



$$\frac{\partial}{\partial \theta_1} J(\theta_1) = 0$$

- Used in complex, cpus
- optimized for performing vector addition & multiplication

we calculate the gradient for each small mini batch of training data

Mini batch

Gradient descent

full batch:
- entire training data whole data is used at once to compute the gradient.
- slow algorithm calculates gradient by differentiation of cost function.

stochastic (SGD) (single training data)
A sample data is used to compute the gradient.
- fast algorithm

* Gradient descent only work for problems which have a well defined convex optimization problem

Pseudocode / Algorithm for GD

Aim:- It is used to minimize a cost function $J(w)$ parametrized by model parameters w

step1 Initialize the weights w randomly

step2 Calculate Error (sse) G of cost function w.r.t. w

step3 Calculate the gradient i.e change in sse when the weight w are changed by a very small value from their original randomly initialized value
- This helps us to move the value of w in the direction where sse is minimized

step4 Adjust/update the weights with the gradients to reach the optimal values where sse is minimized

$$w = w - \eta \frac{\partial J}{\partial w}$$

where η is learning rate

$$\text{Gradient } G = \frac{\partial J}{\partial w}$$

Step 5 Use the new weights for prediction & to calculate the new SSE

Repeat step 1 & 4 till further adjustment to weight doesn't significantly reduce the error. (ie) several passes are performed over the training data before the termination criteria is met.

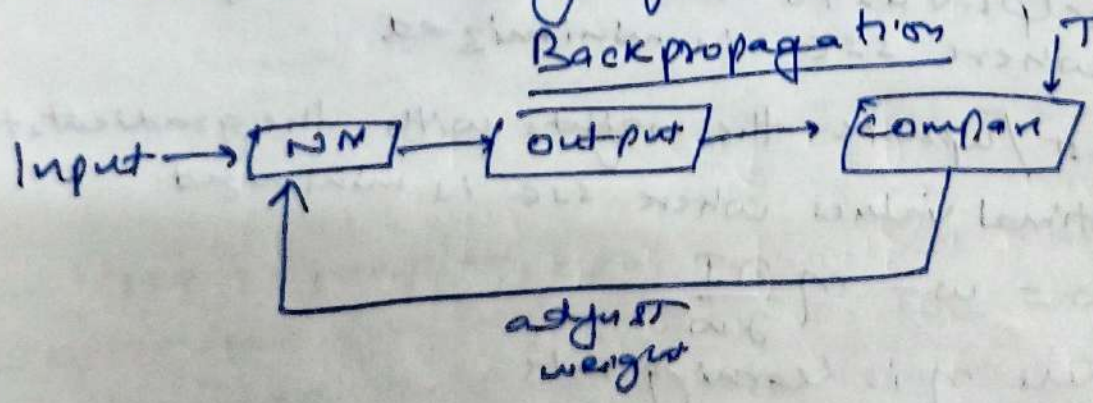
Each pass is called an epoch

Learning rate determines the size of steps we take to reach a minimum

- This parameter is called as hyperparameters in ANN since high values of η may overshoot the minimum & very low value will reach minimum very slowly. →

- we usually start with a small value of learning such as 0.1, 0.01 or 0.001 & change it based on whether the cost function is reduced very slowly → increase learning rate or if exploding / being erratic → ↓ce learning rate

1. If η is too small, gradient descent can be slow
2. If η is too large, gradient descent can overshoot minimum. It may fail to converge or even diverge

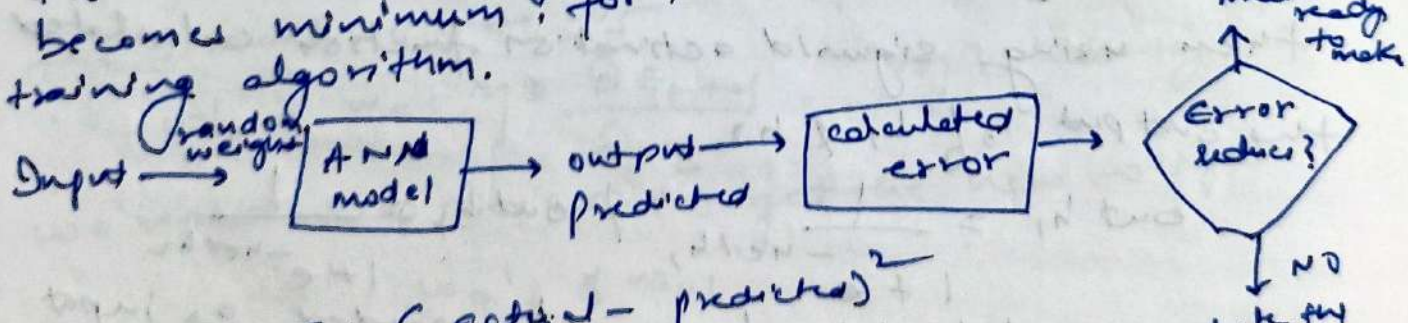


The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary input to outputs

while designing a neural network in the begining we initialize weights with some random values, it's not necessary that whatever weight values we have selected will be correct it may happen that with these weights

<u>Backpropagation (supervised algorithm)</u> It is the training or learning algorithm. It is standard method of training A N N.

- It learns by example. If you submit to the algorithm the example of what you want the network to do, it changes the network weights so that it can produced desired output for a particular input on finishing the training
- These networks are used for simple pattern Recognition & mapping tasks
- our model output is very much different than our actual output that means the error value is high
- To reduce the error, we need to somehow explain the model to change the weights such that errors becomes minimum. for this we use backpogation training algorithm.



$$\sum \frac{1}{2} (\text{actual} - \text{predicted})^2$$

Back propagation algorithm uses gradient descent technique to minimize the value of error function

- Here we are trying to get the values of weight such that error becomes minimum
- we have to find out whether we need to ↑ or ↓ the value of weight. If by ↑ing value of

of weight. If by ↑ing value of weights error reduce, we keep on ↑ing the values otherwise we decrease the weight values to get minimum error

— keep going on right direction we might reach a pt where further update of weight value will ↑ the error. At that time we stop and that is the final weight value

<u>steps involve in backpropagation</u>

Initialize ~~some~~ weights to some random value, also predict value of b_1 & b_2 bias.

<u>step 1, forward propagation</u> The input from a training set are passed through the network

we find the ~~total~~ net input to each hidden layer neuron h_1 & h_2 using following ~~fat~~ formulae

$$net\,h_1 = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net\,h_2 = w_3 * i_1 + w_4 * i_2 + b_2 * 1$$

then using sigmoid activation function calculated the output of h_1 & h_2

$$out\,h_1 = \frac{1}{1+e^{-net\,h_1}} \quad \& \quad out\,h_2 = \frac{1}{1+e^{-net\,h_2}}$$

— The output of hidden layer forwarded as input to output layer, again using above formulas we calculated the output of output neuron o_1 & o_2

$$net\,o_1 = w_5 * out\,h_1 + w_6 * out\,h_2 + b_2 * 1$$

$$net\,o_2 = w_7 * out\,h_1 + w_8 * out\,h_2 + b_2 * 1$$

<u>step 2</u> Calculating the ~~total~~ error

As we are working with a training set, the correct output is known

so we calculated the error for each output neuron o_1 & o_2 using the squared error function

$$E_{o_1} = \frac{1}{2}(target_{o_1} - out_{o_1})^2$$

$$E_{o_2} = \frac{1}{2}(target_{o_2} - out_{o_2})^2$$

The total error for the neural network is the sum of these errors

$$E_{total} = E_{o_1} + E_{o_2}$$

<u>step 3</u> <u>Backward Propagation</u> The objective of backpropagation is to change the weights & bias for the neurons, inorder to bring the errorfunction to a minimum

Now we will propagate backwards

first update weight w_5

first we have to calculated how much a change in w_5 affect the total error

calculated $\frac{\partial E_{total}}{\partial w_5}$ – the partial derivative of E_{total} w.r.t w_5

To use the error we then subtract this value from the current weight w_5

$$w_5' = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} \qquad \eta = \text{learning rate}$$

we repeat this process to get the new weight

w_6', w_7', & w_8'

chain rule

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial o_1} * \frac{\partial o_1}{\partial o_2} * \frac{\partial o_2}{\partial w_5}$$

Next we will continue backward to calculated new values of w_1, w_2, w_3 & w_4 as

$$w_1' = w_1 - \eta * \frac{\partial total}{\partial w_1}, \quad w_2' = w_2 - \eta * \frac{\partial E_{total}}{\partial w_2}$$

$$w_3' = w_3 - \eta * \frac{\partial total}{\partial w_3} \qquad w_4' = w_4 - \eta * \frac{\partial E_{total}}{\partial w_4}$$

Finally we have update all of our weights
After that we will again propagate forward & calculated the output. Again we will calculated the error
- If the error is minimum we will stop else we will will again propagate backward & update the values
- The process will keep on repeating untill error becomes minimum
- At the end of this process, the model is ready to make prediction for unknown input data
- New data can be fed to the model, a forward propagation is performed & the model generates its prediction.

<u>Vanishing gradient problem - L 37</u>

<u>Weight initialization</u> first step for building a neural network is the initialization of parameters (weights) if correct initialization is done then optimization will be achived in the least time otherwise converging to a minima will be impossible using optimization technique like gradient descent

Initially we assign random values to weight working with a deep network it can lead to 2 problems

Let we use sigmoid activation function to produce output

Incase of deep network using sigmoid activation function if weights are small then the gradient will be vanishing small (the gradient of the cost with respect to the weights are too small)

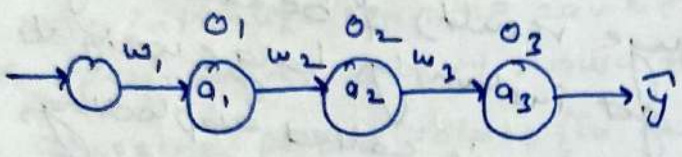
During the backpropagation stage, the error is calculated & gradient values are determined. The gradient are sent back to hidden layers & the weights are update accordingly

The process of gradient determination & scaling

it back to the next hidden layer is continued until the input layer is reached

- The gradient becomes smaller & smaller it reaches the bottom of the network. Therefore, the weights of the initial layers will either update very slowly or remain the same. In other words, the initial layers of the network wont learn efficiently hence it can lead to overall inaccuracy of the whole network

- The simplest solution is to use other activation function such as Relu which does not cause small derivative



$$\frac{\partial I}{\partial w} = \frac{\partial I}{\partial o_3} * \frac{\partial o_3}{\partial o_2} * \frac{\partial o_2}{\partial o_1} * \frac{\partial o_1}{w_1} \quad \text{— (1)}$$

$$\frac{\partial o_3}{\partial o_2} = \frac{\partial \text{ sigmoid } (o_2)}{\partial o_2} * w_3$$

$$= \frac{1}{4} * \frac{1}{3} * \frac{1}{12} \quad \text{(Less value)}$$

$$\frac{\partial o_2}{\partial o_1} = \frac{\partial \text{ sigmoid } (o_1)}{\partial o_1} * w_2 \simeq \text{Lesser value}$$

* wts of neural network can usually b/w -1 and 1

- output of derivative of sigmoid function lies b/w 0 and $\frac{1}{4}$

In equ (1) we multiply values b/w 0 & 1 so this gradient $\left(\frac{dI}{dw_1}\right)$ will be smaller

Vanishing Gradient problem :- As we move backward in the ANN gradient becomes smaller in every layer and it becomes tiny in the early layers (first layer). This is called vanishing gradient problem

As input layer is farthest from the output layer, its derivatives will be the layers expression (chain rule) Hence it contain more sigmoid derivatives and it will have smallest derivative, this makes lower layer slowest to train

Problem

① Because of small step changes ANN may converge at a local minimum instead of global minimum

② If early layer are not accurate then later layers build on this inaccuracy & the entire neural network gets ~~converted~~ corrupted

<u>Exploding gradient problem</u> If the gradients were bigger than one (that mean multiplying number greater than 1 always gives huge results) & because of this some layers may get instantly large weights & the algorithm diverges. This is called exploding gradient problem

— At an extreme, the values of weights can become so large as to overflow & result is NaN values

Exploding gradient can result in an unstable network that at best cannot learn from the learning data & at worst results in NaN weight values that can not longer be updated

<u>weight initialization and its techniques</u> L 38

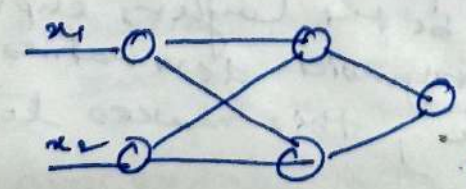
There are a large no. of weight ~~initialization~~ in a neural network leading to a large search space for minima of the loss function

— Weights must be ~~effectively~~ randomly initialized to prevent convergence to false minima

— If all the weights are initialized with 0, in ANN ~~training~~ training, it does not make model better than a linear model

— Zero initialization is not successful in classification

— Assigning random value to weight is better than just zero assignment



$Z^{[1]} = x_1 w_{11} + x_2 w_{12} + b$

$Z_1^{[1]} = Z_2^{[1]} \Rightarrow a_1^{[1]} = a_2^{[1]}$

$Z_2^{[1]} = x_1 w_{21} + x_2 w_{22} + b$

MDP

↓
Policy evaluation
↓
Value iteration

① If weights are initialized with very high values
- exploding gradient problem

② If weights are initialized with very less values
- vanishing gradient problem

- The initialization step can be critical to the model's ultimate performance and it require the right method
- Choosing proper values for initialization is necessary for efficient training
- weights should be random and different from each other often

iteration → forward
→ cost
→ backward
→ update

to prevent the gradients of the network's activations from vanishing or exploding
- The mean of activation should be zero ($\mu = 0$)
- The variance of the activation should stay the same across every layer

$$Var(a^{[l-1]}) = var(a^{[l]})$$

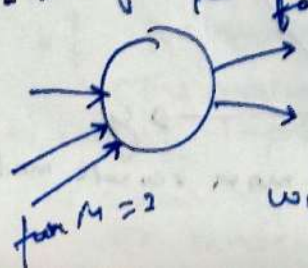
Different weight initialization techniques

① He initialization :- It is useful technique for weight initialization in neural network which uses Relu activation function

for Relu activation function, the best weight initialization strategy is to initialize the weights randomly but with following variance

$$\sigma^2 = \frac{2}{N}$$ which is initialization

N - no. of input features



$$\sigma = \sqrt{\frac{2}{N}} \text{ or } \sqrt{\frac{2}{fan_{in}}}$$

$w_{ij} \sim N(0, \sigma)$ N — Normal distribution

② Xavier Initialization or Gorot Initialization

- It work with tanh activation function
- It's same as He initialization but in this method 2 is replaced with 1

N - No. of input features $\quad \sigma^2 = \frac{1}{N} \Rightarrow \sigma = \sqrt{\frac{1}{N}}$

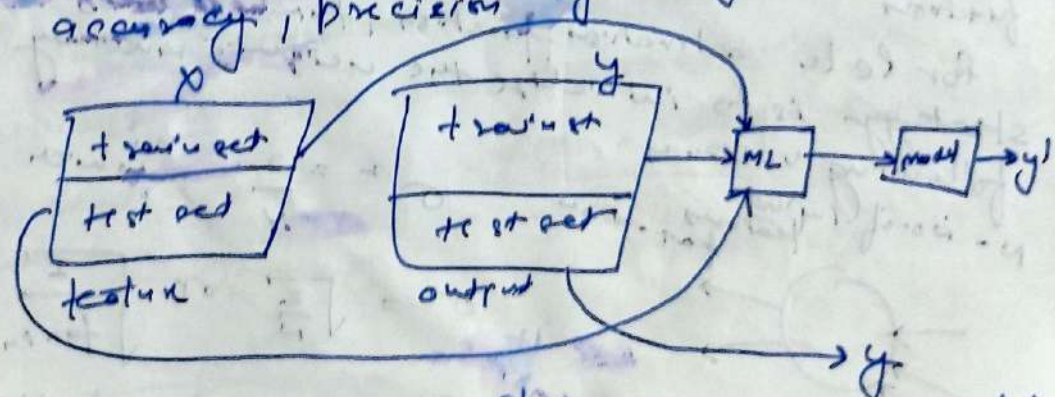
or $\sigma = \sqrt{\frac{1}{fan\ in}}$ $\quad w_{ij} \approx N(0, \sigma)$

N = normal distribution

- The above methods serve as good starting points for initialization and reduces the chances of vanishing or exploding gradients
- They set the weights neither too much ~~large~~ bigger than one nor too much less than 1
- The help avoid slow convergence, also ensuring that we don't keep oscillating off the minima

<u>Training and testing in ML</u> L-40

- In ML, we basically try to create a model to predict on the test data
- Training data is used to fit the model & testing data is used to test the data
- The model generated are to predicted the results unknown which is named as test set.
- Dataset is divided into train and test set in order to check accuracy, precision by training and testing on it

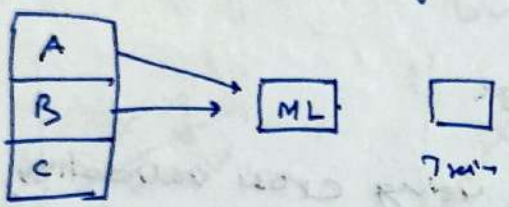


$y - y' \approx$ Less → more accurate model

- In addition to training and test data, a third set of observation called a validation or hold out set is required
- The validation set is used to tune variables called hyper parameters which control how the model is learned.
- It is common to allocate 50% or more to training set
  25% to the test set
  25% to the validation set

An algorithm trained on a large collection of noisy, irrelevant or incorrectly labeled data will not perform better than an algorithm trained on a small set of data that is more representative of problem in the real world.

- → Cross validation can be used to train and validate an algorithm on the same data, so that ~~time~~ the algorithm is trained and evaluated on all of the data



A, C → train
B – test

IInd iteration
B, C training
A – test

Cross validation:- provide a more accurate estimate of the model's performance than testing a single partition of data

Dataset → Training set :- A set of examples used for learning of neural network
→ validation set :- A set of examples used to tune parameters of NN, e.g. no. of hidden layers in NN, e.g. learning rate
→ Test set :- A set of examples used to assess the performance of NN

Overfitting and underfitting problem in ML [4]

Overfitting is a phenomenon where model learns the detail & noise in the training data to an extent that it negatively impacts the performance of the model on new data

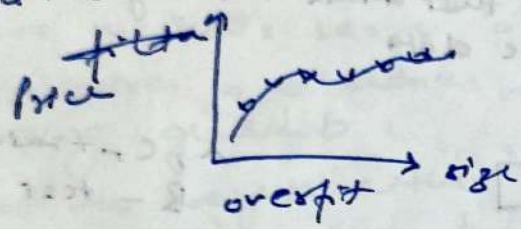
The main goal of ML models is to give correct output to set of inputs that it has never seen before

The cause of poor performance in ML is either overfitting or underfitting the data

Overfitting in ML is a problem in which one model does not generalize well from our training data to unseen data

- A model that has learned the noise instead of the signal is considered overfit because it fits the training dataset but has poor fit with new dataset

- A ~~Mt~~ machine model is said to be overfitted when we train it with a lot of data. When a model gets trained with so much of data, it starts learning from the noise ~~of data~~ & inaccurate data entries in our data set & build an unrealistic ~~data~~ model



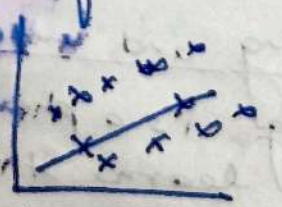
Overfitting can be avoided by using cross validation, early stopping, pruning, & regularization

Underfitting refers to a model that can neither model the training data nor generalize to new data

It occurs when that model does not fit the data well enough

It usually happens when we have less data to build an accurate model

- Underfitting can be avoided by using more data and also reducing the features by feature selection

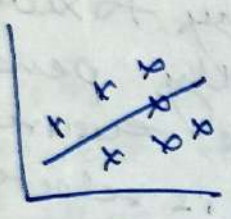


Regularization L 1/2 to make thing regular or acceptable

when a ML model the training data well but fails to perform well on the testing data i.e was not able to predict test data. It is called overfitting and this situation can be dealt with regularization in ML.

- Preventing overfitting is necessary to improve the performance of our machine learning model
- Regularization adds a regularization order to prevent the coefficients to fit so perfectly to overfit
- Regularization is a method for automatically penalizing extra features

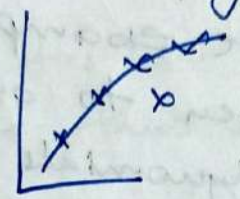
Linear regression



$\theta + \theta_1 x$

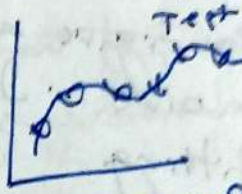
High bias
underfitting

↑ Train error
↑ test error



$\theta + \theta_1 x + \theta_2 x^2$

Just right



$\theta + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

High variance

$J(\theta) \approx 0 - TD$

fail to generalize new examples due to high features

To reduce overfitting

(1) Reduce number of features manually or by model selection algorithm

Disadv. some important features may thrown out

(2) Regularization → keep all features, but reduce magnitude or values of parameters θ_j

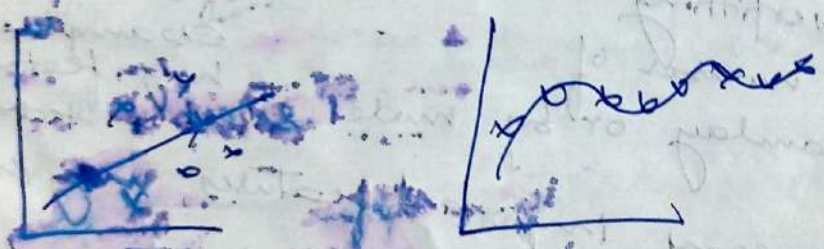
- Regularization works well when we have a lot of features & each of which contributes a bit to predicting y (target output)
- Regularization regularizes or shrinks the coefficient estimate toward zero

- Regularization are techniques used to reduce error by fitting a function approximately on the given training set & avoid overfitting

- Regularization is a technique used for tuning the machine learning model by adding an additional penalty term in the error function

- The additional term (penalty) ~~contains~~ controls the ~~error~~ excessively fluctuating function such the coefficient don't take extreme values.

- The ~~figure~~ algorithm on the right (fig) is fitting the noise. In above example a way to reduce overfitting is then to artificially penalize higher degree polynomials. This ensure that a higher degree polynomial is selected only if it reduces the error significantly compared to simple model, to overcome the penalty This is regularization



Types of Regularization

1). L1 Regularization or Lasso Regularization used when data is large

2) L2 Regularization or Ridge Regularization

L1 Regularization It adds a penalty to the error function & the penalty is the sum of the absolute values of weights

$$\text{Min}\left(\underbrace{\sum_{i=1}^{n}(y_i - w_i x_i)^2}_{\text{Lossfunction}} + \underbrace{\rho\sum_{i=1}^{n}|w_i|}_{\text{penalty}}\right)$$

when ρ is tuning parameter which decides how much we want to penalize the model.

$$\text{costfunction} = \Sigma(y_i - \hat{y}_i)^2 + \alpha\Sigma|\beta_i|$$

$$y_i = \beta_0 + \beta_1 x_1 + \dots \beta_p x_p$$

Lasso – Least ~~Absolute~~ shrinkage and selection operator

high $\beta \to 0$

- Lasso shrinks the less important features coefficient to zero, thus removing some features altogether
- Lasso works well for feature selection in case we have a huge number of features
- Lasso adds absolute value of magnitude of coefficient as penalty term to the loss function.
- Cross validation works well with small set of features but for large set of features this technique is beneficial
- Lasso makes some features with zero weights that means these features are essentially ignored completely in the model
- So only a subset of most important features are left with non zero weights. It also makes the model easier to interpret
- As the value of ρ/α increases it reduces the value of coefficients, hence avoiding overfitting but after certain value, the

Model start loosing important properties of the data (underfitting problem)
Therefore the value of λ/α should be carefully selected by using some hyper parameter tuning techniques

(29) L2 Regularization or Ridge Regularization

It also adds a penalty to error function
But in L2, the penalty is the sum of the squared values of weights

$$\text{Min}\left(\sum_{i=1}^{n}(y_i - w_i x_i)^2 + \lambda \sum_{i=1}^{n}(w_i)^2\right)$$

where λ is the tuning parameter which decides how much we want to penalize the model

Ridge Regression adds squared magnitude of coefficient or penalty term to the loss function

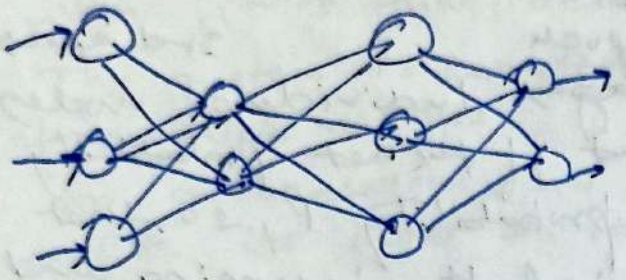
$$\text{cost function} = \sum (y_i - \hat{y}_i)^2 + \alpha \sum \beta_j^2$$

If $\alpha = 0$ – no penalty – regression without regularization

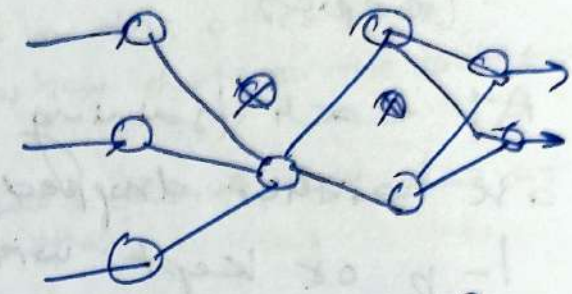
Ridge regression enforces the β coefficient to be lower but it does not enforce them to zero. That it will not remove irrelevant features but rather minimize their impact on trained model

Dropout – L45 Method to avoid overfitting

(1) Reduced no. of features (2) Add more training data, (3) Regularization (4) Dropout

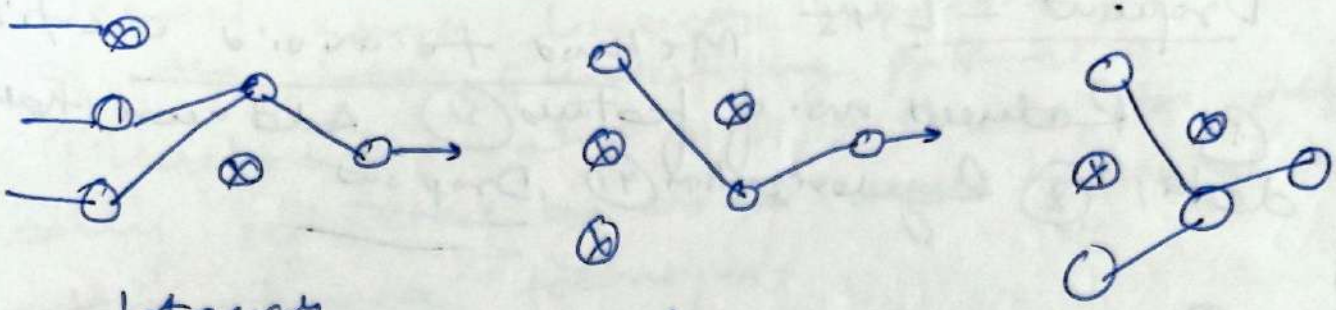


original NN



Dropout NN

- Randomly drop units (neurons) along with their connections from the neural network during training
- Dropout is a technique in which some nodes of the network are temporarily deactivated. This technique is applied in training phase to reduce overfitting ~~problems~~ effects.
- The basic idea behind dropout is to dropout nodes so that the network can concentrate on other features
- In dropout approach we randomly choose a certain no. of nodes from the input & the hidden layers, which remain active & turn off the other nodes of these layers. After this we can train a part of our learn set with this network
- The next ~~step~~ step consists in activating all the nodes again & randomly choose other nodes
- It is also possible to train the whole training set with randomly created dropout ~~network~~ networks.

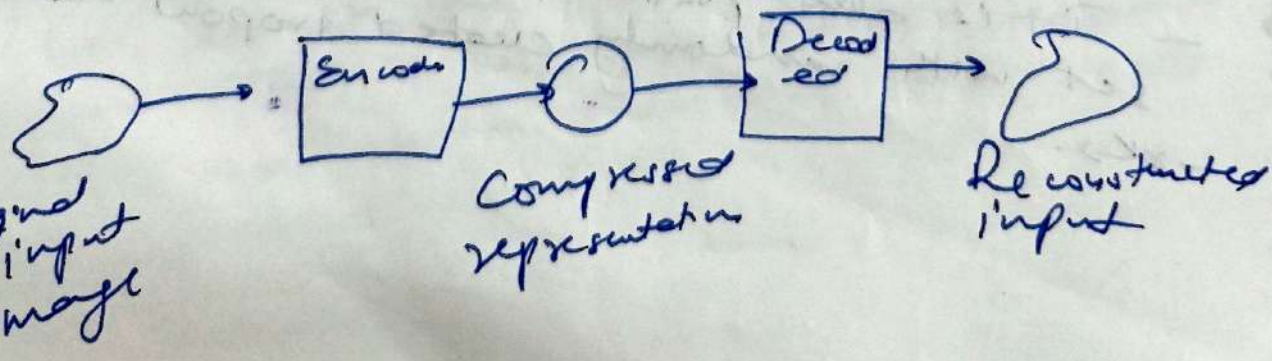


1st epoch 2nd epoch 3rd epoch

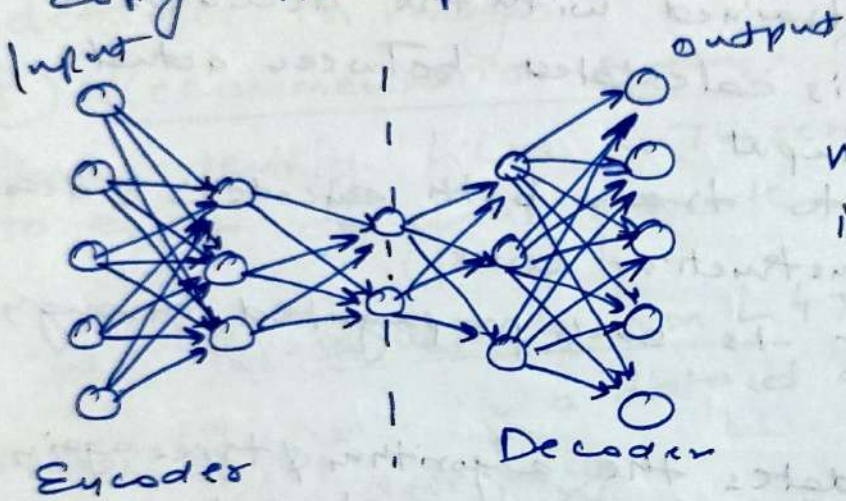
At each training stages individual nodes are either dropped out of the net probability 1-p or kept with probability p, so that a reduced network is left, incoming and outgoing edges to a dropped-out node are also removed (p=0.5 works well.)

- Dropout is an approch to regularize neural network which help reducing interdependent learning amongst the neurons.
- using dropout approch the network become less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization & is less likely to overfit the training data.
- Dropout roughly doubles the number of iteration required to converge. However training time for each epoch is less

Auto encoders L 46



- An autoencoder is a neural network that tries to reconstruct its input
- Autoencoders are unsupervised deep learning algorithm that reduce the no. of dimensions in the data to encode it
- Once the data has been encoded through the algorithm, it is then decoded on the other side
- The system is completed when data on the both sides of encoding matches
- An autoencoder is a neural network that learn to copy its input to its output



no. of neurons in Input is equal to no. of output layer

- Let us have only a set of unlabled training examples $\{x^{(1)}, x^{(2)}, x^{3}, \ldots\}$
- An autoencoder neural network is an unsupervised learning algorithm that applies backpropagation setting the target values to be equal to the input i.e it uses

$$y^{(i)} = x^{(i)}$$

- This autoencoder tries to learn a function

$$f_{w,b}(x) \approx x$$

An autoencoder nn. consists of a input layer an output layer & one or more hidden layers connecting them where the output layer has the same no. of neurons as the input layers

An auto encoder consist of two parts:-

① Encoder:- An encoder is a network that takes the input & output a feature map / vector. These features vector hold the information, that represents the input

② Decoder:- The decoder is a network usually the same network structure as encoder but in opposite orientation, it takes the feature vector from the encoder & gives the best closest match to the actual input or intended output

The encoders are trained with the decoders.

- The loss function is calculated between actual and reconstructed input
- The optimizers try to train both encoders & decoder to lower the reconstruction loss
- finally the error is back propagated through the system

The autoencoder updates the algorithm & tries again forward & back until - solution is found. Once the validation on each side matches, the cycle stops

we will design a neural network architecture such that we impose a bottleneck in the network which forces a compressed knowledge representation of original input

① (1, 0, 0, 1, 0) → (x, y) → (1, 0, 0, 1, 0)

Input — hidden layer — output

② 10010 → 50 neurons → 100 neurons

= 100 neurons

Application of ~~Autoencoder~~:

① Data storage :- The encoding process allows to compress down large quantities of data. So it has big benifits for data storage

② Feature detection :- The process used to encode the data identifies features of the data that can be used to identify it

This list of features is used in multiple systems to understand the data like in convolutional neural detection in images

③ Recommendation system :- These are the systems that identify films or TV series you are likely to enjoy on your favorite streaming services.

~~Eg~~ Batch normalization L47

Normalization :- In a dataset all the features (column) may not be in some range.

Ex price of house — Age of house

5000000 — 25 yrs

- Neural network takes lots of time to train for these kinds of dataset.

In simple machine learning algorithms like linear regression, the input is normalized before training to make them into single distribution

Normalization is to convert the distribution of all input to have mean = 0 & standard deviation = 1. So most of the value lie b/w -1 & 1

Batch! As we know that ML algorithm require training dataset (group of examples) based on which it can learn. Data needs to be in numerical form so algorithm can understand it

	Feature 1	Feature 2	Feature n	Vector y
Sample 1 :-	[number,	number ...	number]	[expected Result]
Sample 2 :-	[number,	number ...	number]	[exp result]
⋮				⋮
Sample m	[number,	number ...	number]	[expected result]

<u>Stochastic gradient descent</u> when only one sample per iteration is used to update model parameters this is called as stochastic gradient descent

<u>Mini Batch Gradient descent</u> :- when more than one sample per iteration but less than all available samples are used to update model parameters this technique is called as Mini Batch gradient descent

<u>Batch gradient Descent</u> :- when all available samples are used to update model parameters this techniques is called Batch Gradient Descent

<u>Batch</u>: It is a no. of samples fed to NN during single iteration

- Training very deep NN that standizes for each mini batch
- This has the effect of stabilizing the learning process & dramatically reducing the number of training epoch required to train deep network

As we know that NN learn the problem using backpropagation algorithms.

- During back propagation of errors to the weights biases, each layer is trying to correct its for the error made up during the forward propagation

<u>Batch normalization</u> is a technique for training very deep nn that standardizes the input to a layer of each mini batch

- This has the effect of stabilizing the learning process & dramatically reducing the number of training epochs required to train deep network

- As we know that nn learn the problem using Backpropagation algorithm

- During backpropagation of errors to the weights & biases, each layer is trying to correct itself for the error made up during the forward propag-tion

- But every single layer acts seperately to correct itself for error made up

- for example the 2nd layer adjust its weights & biases to correct for the output, but due to this readjustment, the output of 2nd layer (input of 3rd layers) is changed for some initial input so the 3rd layer has to learn from scratch to produce the correct outputs for the same data.

- This creates a problem because 3rd layer learn after 2nd finished, 4th start learning after 3rd layer

- If deep nn 1000 layer deep; this it would really take lots of time to train

- This is happen because due to changes in weight of previous layers, the distribution of input value for

current layer changes & forcing it to learn for new input distribution.

To overcome from this problem, Batch normalization adds normalization layer b/w each layers

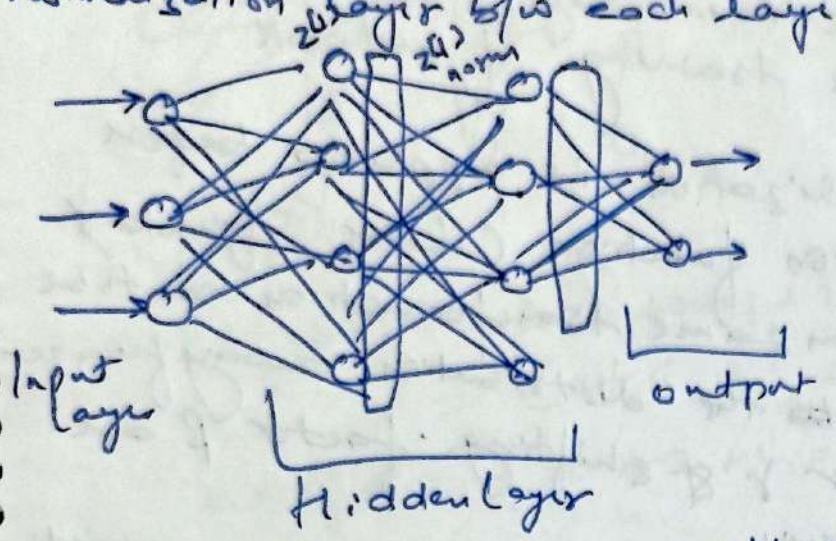


fig Batch normalization

- Due to this normalization layers b/w each fully connected layers, the range of input distribution of each layer stays the same, no matter the changes in previous layer

$$z^n_{norm} = \frac{z^n - \text{mean(batch)}}{\sqrt{\text{variance(batch)}}}$$

Adding new training parameters

$$\hat{z}^n = \gamma^n z^n_{norm} + \beta^n$$

$$a^n_{norm} = g(\hat{z}^n)$$

Randomly sampled batches

Hidden layer 1 input layer → Linear → Batch normalization → Activation

$a^1_{norm} = g(z^1_n)$

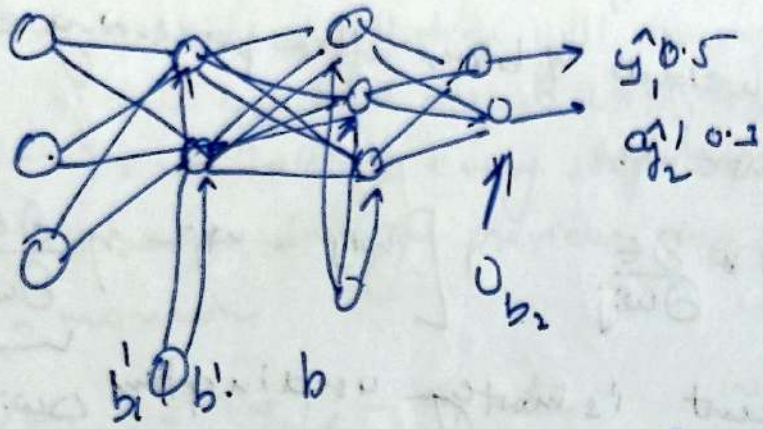
Hidden layer n Activation → Linear → Batch normalization → output layer

Normalization ~~being~~ brings all the input centered around 0 so there is not much change in each layer input so there is no need for layers to wait for the previous layers to learn, hence it fastens up the training of network

- If Batch normalization is applied before applying activation function (like sigmoid & tanh, Relu) then same transformation has to be done to move ~~to~~ the distribution away from zero
- A scaling factor γ & shifting factor β are used to do this
- ~~As~~ training progresses, along with weights γ and β also learn through back propagation so as to improve accuracy of nn
- If batch normalization is applied after activation function then set $\gamma = 1$ & $\beta = 0$

<u>Momentum</u> L48 Momentum is a simple technique that ~~app~~ often improve both training speed & accuracy of neural network

- Training a nn is the process of finding values for the weights & biases so that for get set of input values the computed output value closely match the known target values



no. of wts. $= 3\times2 + 2\times3 + 3\times2$
$= 6 + 6 + 6 = 18$

no. of biases $= 2+3+2 = 7$

stochastic gradient descent optimizer

The figure shows the each step of SGD

The line not pointed directly towards the central (global minima) because the gradient estimates in SGD are noisy, due to small sample size. So, the gradient steps are noisy even if they are correct on average

- So SGD wastes to much time swinging back & forth along the direction in parallel with x_2 axis which advancing too slowly along the direction of the x_1 axis

- Momentum: is a simple technique that often improves both training speed & accuracy of neural network

Training a NN is the process of finding values for the weights & biases so that for a get set of input values the computed output values closely match the known target values

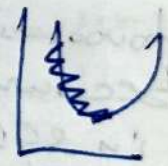
Back propagation Algorithm is used to train most of the nn.

- update all the weights & biases after processing single training item, we use

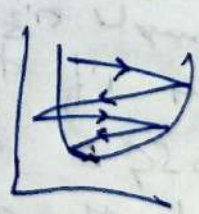
$$\Delta w_{ij} = \eta * \frac{\partial E}{\partial w_{ij}} \qquad \left[w_{ij}' = w_{ij} - \eta \underbrace{\frac{\partial E}{\partial w_{ij}}}_{\Delta w_{ij}} \right.$$

- Gradient descent is mostly used in optim
- -zation algorithm in ML
- But one of the problem with gradient descent is that it oscillates too much on vertical axis & this oscillation slows down the convergence to local minimum

we want less oscillation on vertical axis & faster movement ~~forward~~ toward the global minimum, to make faster learning & convergence to the global minimum



- Gradient descent with momentum is an optimization algorithm which helps to achieve this



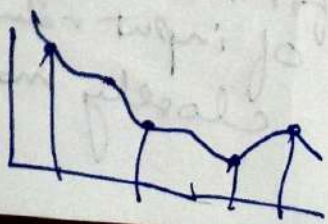
$$0 \le \beta \le 1$$

Now $\Delta w_{ij} = \eta * \frac{\partial E}{\partial w_{ij}} + (\beta * \Delta w_{ij}^{t-1})$

wt. increment — learning rate — wt gradient — momentum factor — wt increment previous iteration

Here another hyperparameter is used

- The derivative represents acceleration whereas momentum ~~terms~~ terms are velocity



By adding $\beta \times \Delta w_{ij}^{t-1}$ it smooth out gradient descent & the vertical oscillation will average out to zero beca-use of this gradient descent will to converge quickly as it is not taking every step depends of previous one. It is rather taking previous steps into account to gain momentum

$$w_{new} = w_{old} - \eta \times \frac{d\varepsilon}{\partial w_{old}}$$ without momentum

$$w_{new} = w_{old} - \left[\eta \times \frac{d\varepsilon}{\partial w_{old}} + \beta \times w_{t-1} \right]$$ with momentum

$\beta = 0.9$ mostly used value

reduce noise

- The idea of momentum-based optimizers is to remember the previous gradient from recent optimization steps & to use them to help to be a better job of choosing the direction to move next where the objective cost function can be reduced fastest on a given example

Tuning hyper parameters

Three dimensional space :

Dot and cross product

$$\vec{a}\cdot\vec{b} = |\vec{a}|\,|\vec{b}|\cos\theta = x_1x_2 + y_1y_2 + z_1z_2$$

$$\vec{a}\times\vec{b} = \begin{bmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{bmatrix} = |\vec{a}|\,|\vec{b}|\sin\theta\,\vec{n}$$

where $\vec{n}$ is the unit vector $\perp$ to the vectors $\vec{a}$ and $\vec{b}$

$\cancel{|\vec{a}||\vec{b}| = 0}$ $\quad \vec{a}\cdot\vec{b} = 0$ $\quad$ [orthonormal $\perp$ and length each vector is one]

Coordinate transformation

R_{f_M} is the representation of M w.r.t f

Proof

Type of optimizer

Gradient

BGD

SGD

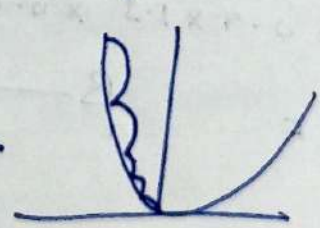
min Batch GD

Computation

Challenges

1. Learning rate

$$w_n = w_0 - \eta\frac{\partial L}{\partial w_0}$$



developers.google.com

2. Learning rate scheduling

→ predifined

(3) 9 weight 1 bias — η is same both the direction

10-D Restriction

4) Local minima

5) Saddle point $\frac{\partial L}{\partial w} = 0$

1) Momentum 2) Adagrad 3) NAG 4) RMS prop 5) Adam

<u>Exponentially weighted moving average</u> (EWMA)

to find trend of time series data.

- time series forcasting
- ~~single~~ signal process
- $\frac{DL}{\theta}$

- weightage decrease as time decay

D1	25
D2	13
D3	17
D4	31
D5	43

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

$$\beta \rightarrow 0 < \beta < 1$$

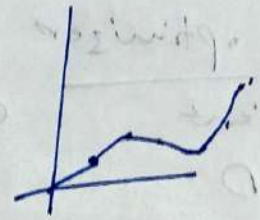
$$\beta = 0.9$$

$$V_0 = \theta_0$$

$$V_1 = 0.9 \times V_0 + 0.1 \times 13$$

$$= 1.3$$

$$V_2 = 0.9 \times 1.3 \times 0.1 \times 17$$

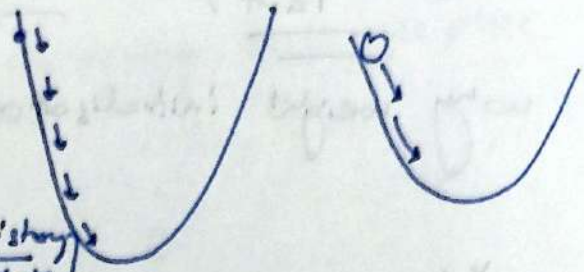


$$\beta \rightarrow \left[\frac{1}{(1-\beta)}\right] \rightarrow \beta = 0.9$$

$$\frac{1}{1-0.9} = \frac{1}{0.1} = 10.$$

```
X1 = df['mean temp'].ewm(alpha=0.9).mean()
```

SGD with momentum



Momentum optimizer

m v

v = previous [history / update]

Advantage — speed

Mathematics

$V \longrightarrow$ Velocity

$w_{t+1} = w_t = \eta * w_t$ $\qquad w_{t+1} = w_t - v_t$

$v_t = \beta * v_{t-1} + \eta \nabla w_t$

history of past velocity is used as momentum

when $\beta = 0$ then momentum behave as SGD.

NAG ~~A~~ Nestr Accelerate Gradient

historical velocity + Gradient at that point

set different learning rate

big $\rightarrow$ lr $\rightarrow$ small

lr $\rightarrow$ small $\rightarrow$ gradient $\rightarrow$ big

$$\boxed{w_{t+1} = w_t - \frac{\eta \nabla w_t}{\sqrt{v_t + \varepsilon}}} \qquad \boxed{v_t = v_{t-1} + (\nabla w_t)^2}$$

$\frac{\partial L}{\partial w}$

ε = small number

$w_{t+1} = w_t - \eta \Delta w_t$ $\qquad v_t = 0$

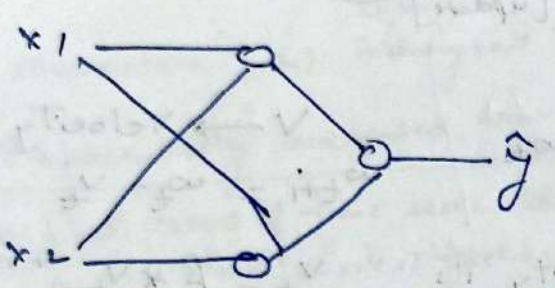
$v_t =$

$$\boxed{w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} \nabla w_t}$$

Weight initialization techniques campus x

Part 1

why weight initialization is important?



1. Initialize the parameters
2. choose an optimization algorithm

Sigmoid A.f.

wrong not initialization

wrong weight initialization

case 1: Zero initialization

$$a_{11} = \max(0, z_{11}) = 0$$

$$z_{11} = w'_{11} x_1 + w'_{21} x_2 + b_{11} = 0$$

$$a_{12} = \max(0, z_{12}) = 0$$

$$z_{12} = w'_{12} x_1 + w'_{22} x_2 + b_{12} = 0$$

If $w = 0$, $b = 0$ then above equ. become 0

$a_{11} = a_{12} \Rightarrow$ equal and 0

$$w'_{11} = w'_{11} - \eta_1 \frac{\partial L}{\partial w'_{11}}$$ no update

no training will take place

tanh $$a_{11} = \frac{e^{z_{11}} - e^{-z_{11}}}{e^{z_{11}} + e^{-z_{11}}} = 0$$

$a_{12} = 0.5$

$a_{11} = 0.5$

$$\frac{\partial L}{\partial w'_{11}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_{11}} \frac{\partial a_{11}}{\partial z_{11}} \frac{\partial z_{11}}{\partial w_1} \times 1$$

$$\frac{\partial L}{\partial w'_{12}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_{12}} \frac{\partial a_{12}}{\partial z_{12}} \frac{\partial z_{12}}{\partial w'_{12}} \times 1$$

$$\frac{\partial L}{\partial w'_{21}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_{11}} \frac{\partial a_{11}}{\partial z_{11}} \frac{\partial z_{11}}{\partial w'_{21}} \times 0$$

$$\frac{\partial L}{\partial w'_{22}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_{12}} \frac{\partial a_{12}}{\partial z_{12}} \frac{\partial z_{12}}{\partial w'_{22}} \times 0$$

non linearity not capture
— as a single node

$w = 0.5 \quad b = 0.5$

$a_{11} = \max(0, z_{11})$ → non zero

$z_{11} = w'_{11} x_1 + w'_{21} x_2 + b_{11} \neq 0$

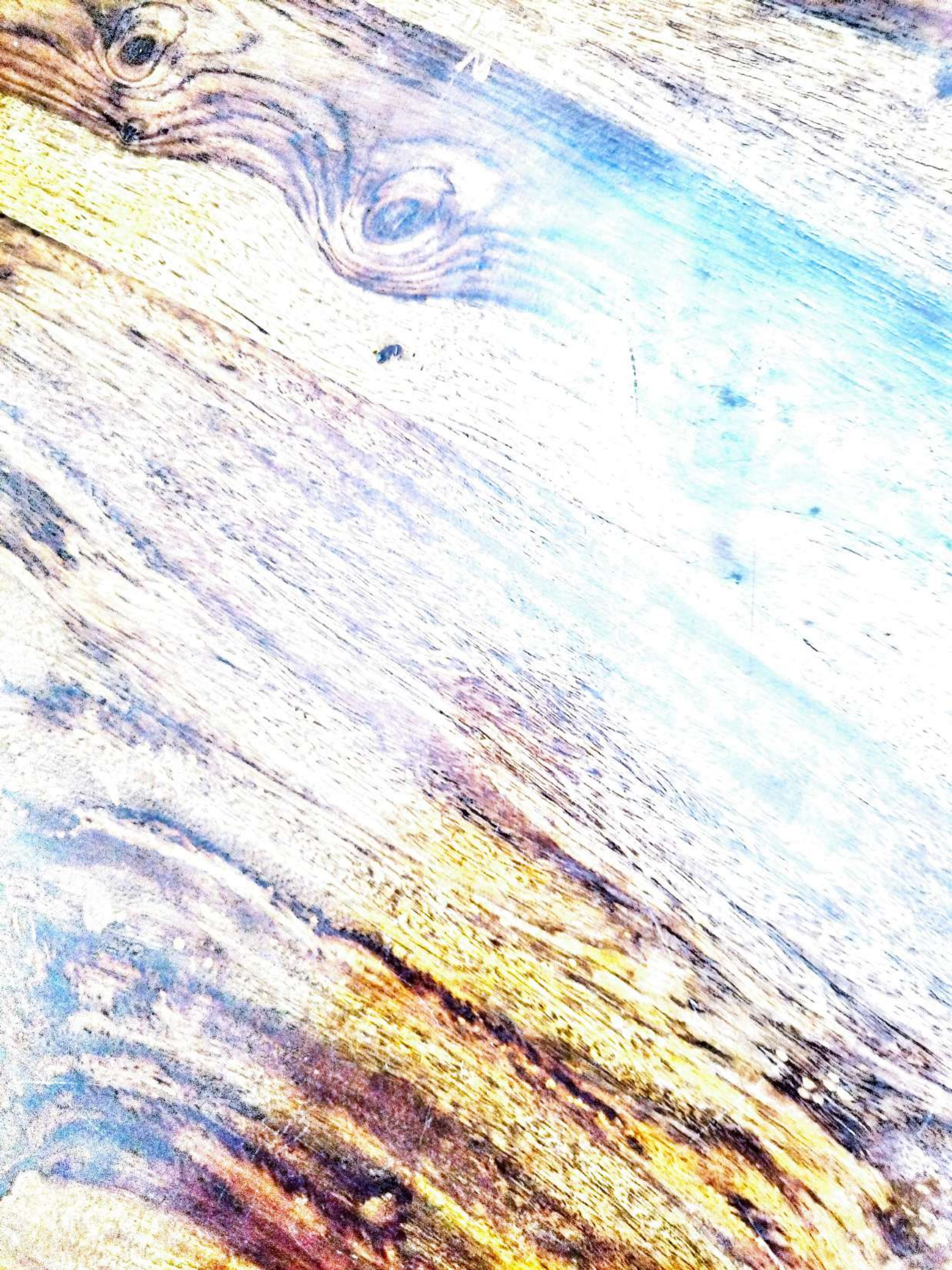
$a_{12} = \max(0, z_{12}) \neq 0$

$z_{12} = w'_{12} x_1 + w'_{22} x_2 + b_{12} \neq 0$

$z_{12} = z_{11} \qquad a_{11} = a_{12}$

$$\frac{\partial L}{\partial w'_{12}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial a_{12}} \cdot \frac{\partial a_{12}}{\partial z_{12}} \cdot x_1$$

— not non zero constant

— not zero

— non zero ~~con~~ random value.