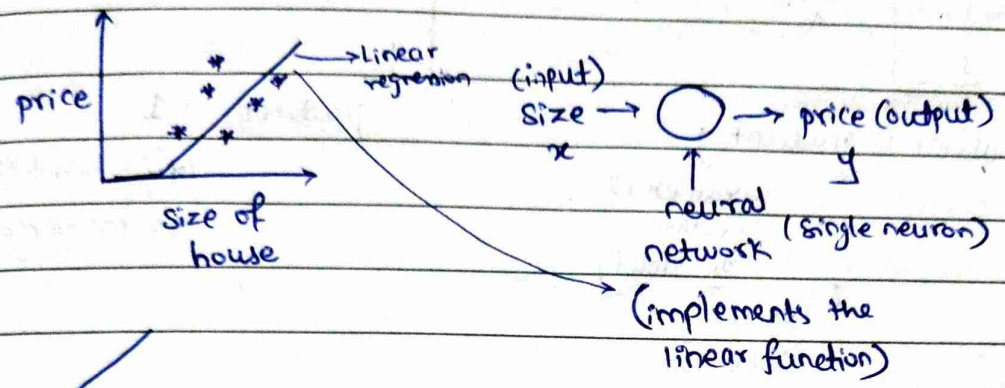
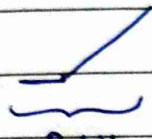
* Deep Learning And Neural Networks *

L=1

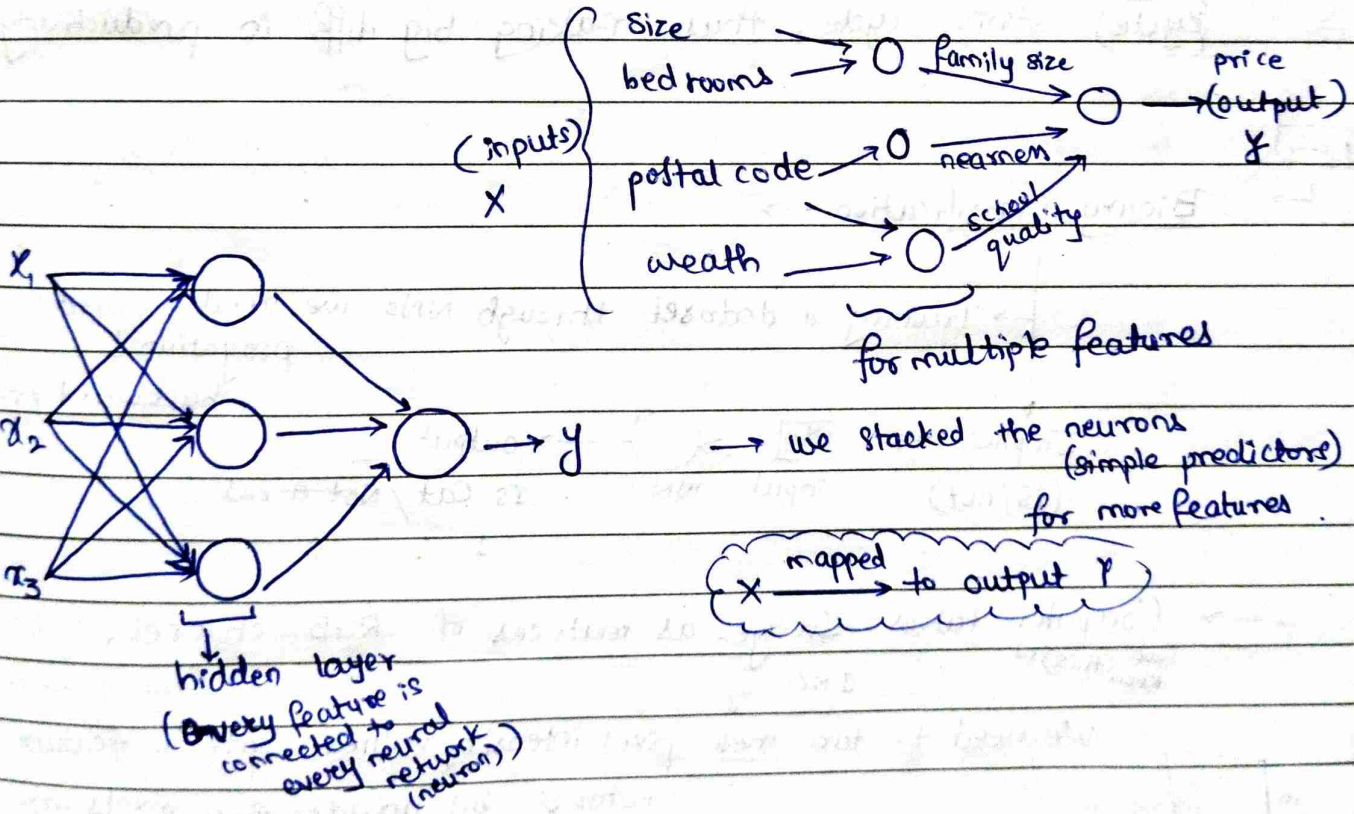
→ Neural Networks ⇒

↳ Let take an ex. of Housing price Prediction





↳ Rectified Linear Unit → Multiple parameters also included having non-linear fun⁻



→ we stacked the neurons (simple predictors) for more features.

X mapped to output Y

L=2 →

- Structured Data → We can have databases having data in form of rows & columns, well defined features.
- Unstructured Data → mainly Audio files, images or texts
  ↳ NNs made it easy to recognise unstructured data.

L-3)

↳ For High performance : 1) We need a neural network that is large enough to take the advantage of large amounts of data,

2) We also need a lot of data for the training.

→ Also algorithms making the computation faster.

⇓

Sigmoid fun^n → to → ReLU fun^n (faster)

parameters change very slowly → learning becomes slow (computations)

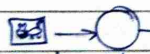
most lon likely to converge !

→ We need faster algorithms, as training N.N.s is basically a loopy structure and if training takes longer time to does (cycle) this cycle, thus making big diff. in productivity.

L-4

↳ Binary Classification →

↳ Training a dataset through NNs we need forward progation & backward propga.

Implies →  → output

(is/not) input NN

Is Cat / Not a cat

1 0

→ Computer takes Images as matrices of RGB channel,

(we know this)

We need to turn these pixel intensity values into a feature vector by unfolding these pixels into an input X → X: dimensions = m×m×3

vector $n_x = m \times m \times 3$

→ A single training sample is denoted by (x, y) ← pair

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

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

↓ no of training samples (examples)

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

columns

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

↓

$n_x \times m$ dimensional matrix

X.shape = (n_x, m) ✓

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

$Y \in \mathbb{R}^{1 \times m}$ ← 1×m dimensional matrix

Y.shape = $(1, m)$

L-5 → Logistic Algorithm ← Used when the output label y is 0/1.

Regression ↳ Binary classifi^n problem

⟹ Given : X , $\hat{y} = P(y=1|x)$

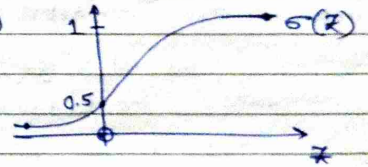
↓ feature input vector → probability of y being 1

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

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

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

Sigmoid function to be applied to ensure range of $\hat{y}$ b/w (0 → 1)



$\sigma(z) = \frac{1}{1+e^{-z}}$ ↳ If z → large(+ve) $\sigma(z) \approx 1$

↳ If z → large(-ve) $\sigma(z) \approx 0$

L-6 ⟹ Logistic regression cost fun^n -

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

↓ transpose

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

↓ predicted ↳ correct sample

→ Loss error fun^n : measures how large the error is b/w the predicted value of output $\hat{y}$ & the the freue value y (provided)

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

↳ not convex (optimized)

$$\mathcal{L}(\hat{y}, y) = -(y \log\hat{y} + (1-y)(\log(1-\hat{y}_2))$$

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

↓ we need to minimise this error $\quad \boxed{0 \le \hat{y} \le 1}$

If $(y=1)$: $L(\hat{y}, y) = -\log\hat{y}$ ← want $\log\hat{y}\uparrow$, want $\hat{y}\uparrow$ i.e. $\hat{y}=1$

↳ $\hat{y}=1$

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

$\hat{y}=0$

→ Cost fun^n → measures how well the param. w, b predict the training set

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

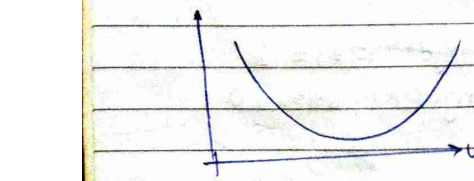
$$= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$$

→ Loss fun^n is applied to single training examples & cost fun^n acts like cost of parameters.

↳ We need to minimize Cost fun^n (Overall)

To find w, b

L-7 → Gradient Descent →



Repeat {

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

"dw" ← write implementation we take

}

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

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

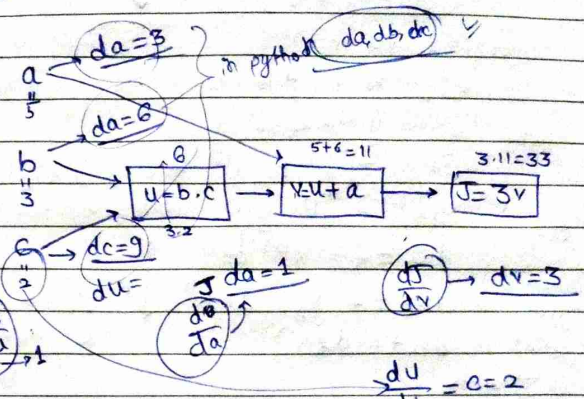
L-8 Computation Graph →

Let → $J(a,b,c) = 3(a+b.c)$

$u = b.c$

$v = a+b.c$

$J = 3v$



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

$\frac{dJ}{da} = \frac{dJ}{dv}\cdot\frac{dv}{da} = 3.1 = 3$

$\frac{dJ}{db} = \frac{dJ}{du}\cdot\frac{du}{db} = 3.2 = 6$

$\frac{dJ}{dc} = \frac{dJ}{du}\cdot\frac{du}{dc} = 3.3 = 9$

$\frac{du}{db} = c = 2$

L-9

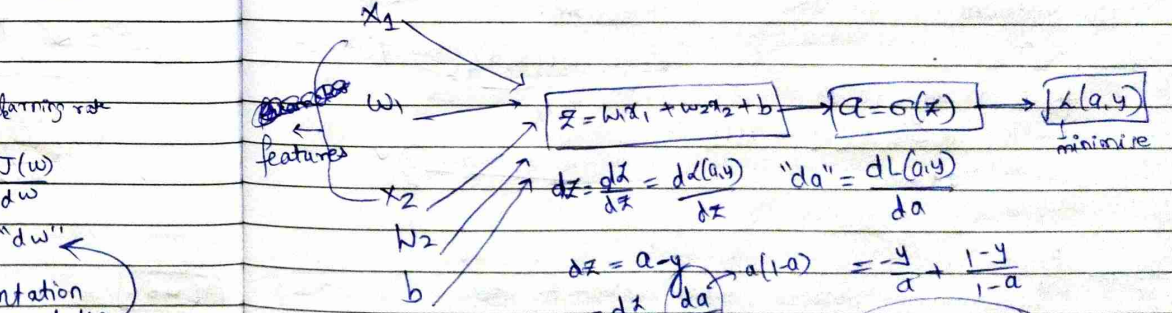
→ $\frac{dJ}{db} = \frac{dJ}{du}\cdot\frac{du}{db} = 6$

L-10 →

$z = w^T x + b$

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

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



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

$dz = a - y$

$= \frac{d\mathcal{L}}{da}\cdot\frac{da}{dz}$ $\quad a(1-a)$ $= -\frac{y}{a} + \frac{1-y}{1-a}$

$dw_1 = w_1 - \alpha.dw_1$

$w_2 = w_2 - \alpha.dw_2$

$dw_1 = "dw_1" = x_1 dz$

$dw_2 = "dw_2" = x_2 dz$

$db = dz$

L-11 Logistic regression on m examples →

↳ Actual algorithm

→ Let initialise :-

One Single Step :-

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

for i = 1 to m → explicit for loop

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

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

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

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

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

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

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

for loop for multiple features i.e. $dw_1, dw_2, \ldots dw_n$ → n=2 → two features

$J/=m$

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

But explicit for loops reduces efficiency.

$dw_1 = \frac{dJ}{dw_1}$

One Step of gradient descent → $w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

L-12 ⇒ Vectorisation →

→ We need to compute : $z = w^T x + b$

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

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

Non-vectorised :

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

Vectorized :

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

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

$w^T x$

much faster 😊 ✓

L-13 ⇒ PTR :-

↳ Wherever possible, avoid explicit for-loops.

Numpy → build-in functions use ✓

⇒ dw = np.zeroes (n_x, 1) instead of dw_1 & $dw_2 = 0$

L-14 ⇒ Vectorising Logistic Regression →

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

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

→

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \ldots & x^{(m)} \end{bmatrix} \quad (n_x, m) \quad \mathbb{R}^{n_x \times m}$$

$$w^T X = \begin{bmatrix} \text{---} \end{bmatrix}_{w^T} \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \ldots & x^{(m)} \end{bmatrix}$$

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

$Z = np.dot(w^T, X) + b$ → real no. converted to $1 * m$ → row vector

Z calculated in one-line. Broadcasting in python.

L-15

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

$dz = \begin{bmatrix} dz^{(1)} & dz^{(2)} & \ldots & dz^{(m)} \end{bmatrix}$

$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \ldots & a^{(m)} \end{bmatrix} \quad Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \ldots & y^{(m)} \end{bmatrix}$

$dz = A - Y = \begin{bmatrix} a^{(1)} - y^{(1)} & a^{(2)} - y^{(2)} & a^{(3)} - y^{(3)} & \ldots \end{bmatrix}$

Vectorisation in logistic Regression

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

For i in range 1,m :

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

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

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

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

loop for more features:

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

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

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

→ Efficient (vector value replaced) → $dw += x^{(i)} * dz^{(i)}$

$Z = W^T X + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

compute for all the values of (i)

$dw = \frac{1}{m} X dZ^T$

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

$w = w - \alpha dw$ (learning rate)

$b = b - \alpha db$

Just step for gradient descent

A for loop will be used for multiple steps.

axis=0 → vertically

L-16 → Broadcasting example →

eg 1: $\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100$ (python takes it as numeral constant) $\rightarrow \begin{bmatrix}100\\100\\100\\100\end{bmatrix} \Rightarrow \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$

(4,1) vector

eg 2: $\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}100&200&300\end{bmatrix} \Rightarrow \begin{bmatrix}101&202&303\\404&505&606\end{bmatrix}$

$\begin{bmatrix}100&200&300\\100&200&300\end{bmatrix}$

(m,n) vector

(1,n) vector → python copies this vector m times to make it (m,n) vector

L-17 Recap →

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

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

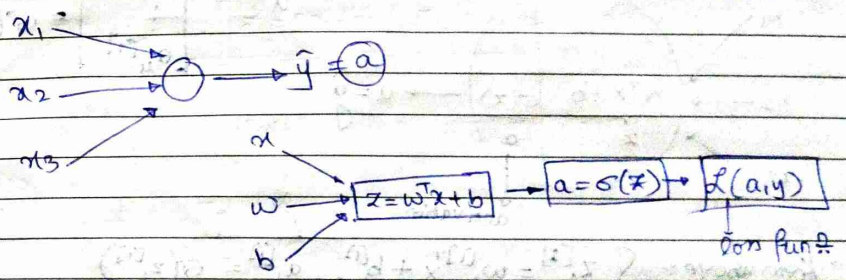
$y=1$: $P(y|x) = \hat{y}^{(1)} (1-\hat{y}^{(0)}) = \hat{y}$

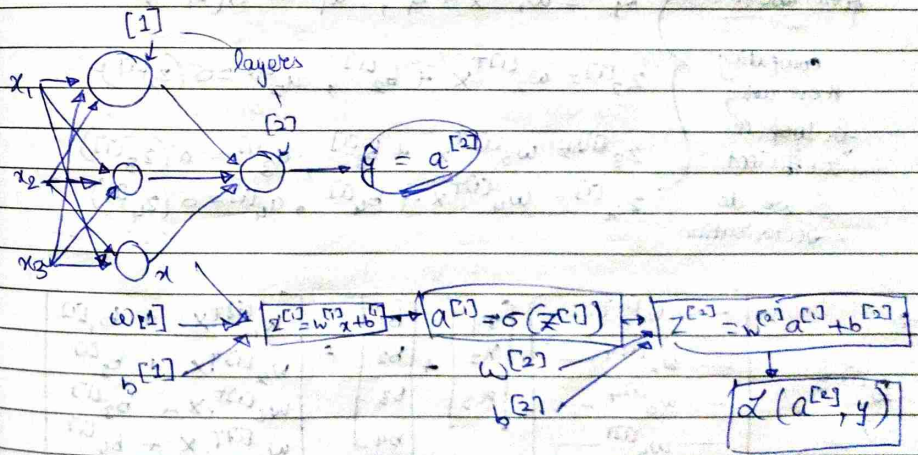
$\log P(y|x) = \log \hat{y}^y (1-\hat{y})^{(1-y)}$

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

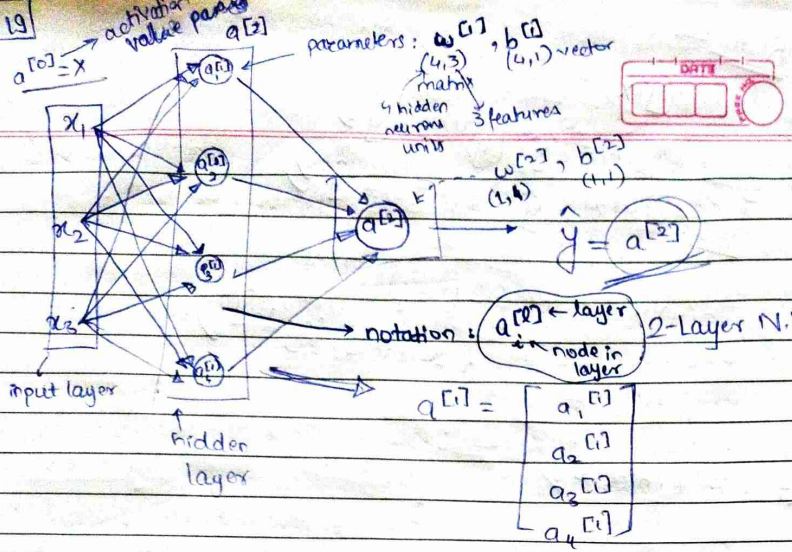
* Cost on m' exples →

L-18 → What is Neural Network →

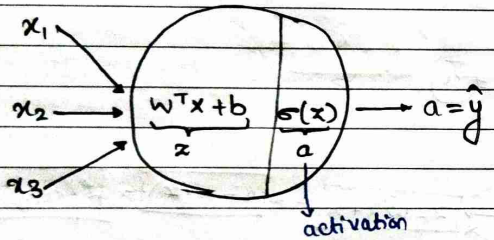




L-19



→ Representation →



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

→ From above :-

computing these using for loop is inefficient ↓ so we do vectorisation

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

→

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

$W^{[1]}$ ✓ $\quad b^{[1]}$ ✓

→ Given input x : $a^{[0]}$

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

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

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

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

L-20 Vectorisation across multiple variables/examples →

→ Non-vectorised

for i = 1 to m; → inefficient

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

i^{th} training sample

Vectorisation →

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

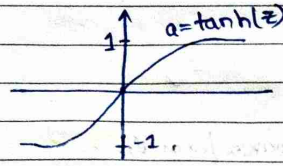
$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

Implementation →

⊙ Forward Propagation examples =

L-21 Activation Functions →
↳ Sigmoid

↳ tanh(z) is always superior & better than sigmoid
↳ tan-hyperbolic ↳ ranging ∈ [-1, 1)
↳ gets mean value '0'
↓
which is quite good

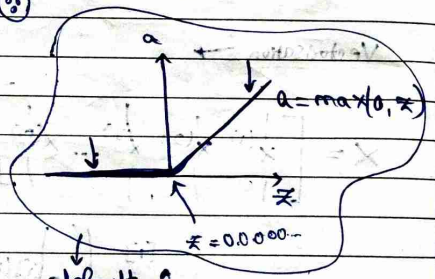


⇒ tanh(z) function for hidden layer.

⇒ Sigmoid σ(z) function for output layer. (0 & 1) (Binary class.)

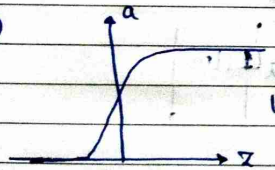
→ A common disadvantage in both these fun^n s is that, when z is very large / small, the derivative approaches zero, → gradient descent can become smaller 😵

ReLU → choice!!!



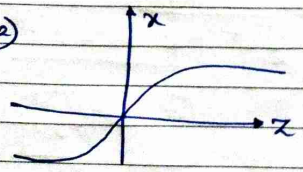
default choice

*Pros & Cons of activation fun^n →

1)


Use only for Binary classif^n (Output layer)

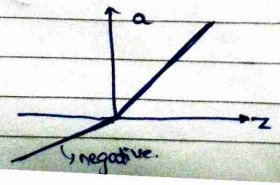
sigmoid : $a = \frac{1}{1+e^{-z}}$

2)


Can be Used in hidden layers

3) ReLU :


most efficient

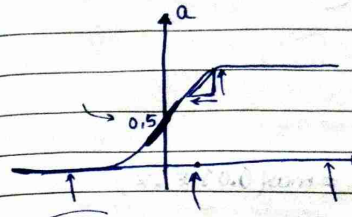
4)


Leaky ReLU (better than ReLU)

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

L-22 → When we need to implement back propagation, we need to compute derivative of the activation fun^n

1) Sigmoid Activation fun^n :-



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

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

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

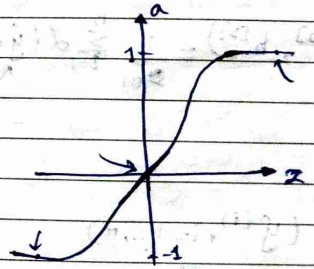
$= g(z)(1 - g(z))$

$z = 10$, $g(z) \approx 1$
$\frac{d}{dz} g(z) \approx 1(1-1) \approx 0$

$z = -10$, $g(z) \approx 0$
$\frac{d}{dz} g(z) \approx 0(1-0) \approx 0$

$z = 0$, $g(z) \approx 0.5$
$\frac{d}{dz} g(z) \approx 0.5(0.5) = 0.25$

2) Tanh activation fun^n →

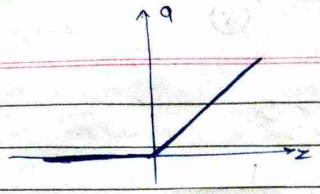


$g(z) = \tanh(z)$
$= \frac{e^z - e^{-z}}{e^{(z)} + e^{-z}}$

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

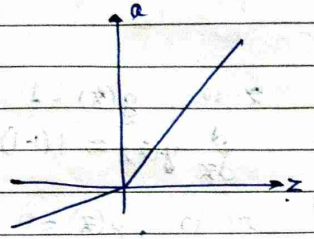
- $z = 10$, $\tanh(z) \approx 1$
  ↳ $g'(z) = 1 - (1)^2 = 0$
- $z = -10$, $\tanh(z) = -1$
  ↳ $g'(z) = 1 - (-1)^2 = 0$
- $z = 0$, $\tanh(z) = 0$
  ↳ $g'(z) = 1$

3) ReLU & Leaky ReLU →



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

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



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

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

L-23 Gradient Descent for N.N.

parameters : $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$

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

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

Gradient descent:

Repeat {

→ compute predicts $(\hat{y}^{(i)}, i = 1, .., m)$

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

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

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

* Forward Propagation →

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

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

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

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

* Back propagation →

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

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

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

"To not output funny 1-D array"

$dz^{[1]} = \underbrace{w^{[2]T} dz^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]'}(z^{[1]})}_{\text{produa } (n^{[1]}, m)}$

* (element-wise)

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

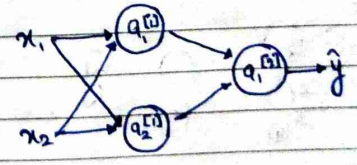
$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims = True)$

"horizontal"

L-24

↳ Random Initialisation -

→ Initialising $w=0$, $b=0$ can be used won't work as multiple iterations gives exact value so no good/outcome of training the model (N.N.)



Randomly →

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

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

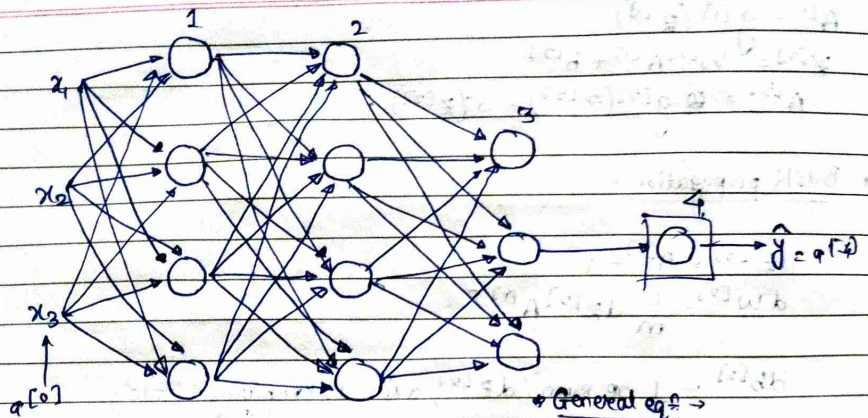
↳ Kay to initialise to 0

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

$b^{[2]} = 00$

may be better constants than 0.01 (but relatively small no.)

L-25 Forward Propagation in Deep Network →



* General eq^n →

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

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

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

* Vectorized → $X = A^{[0]}$

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

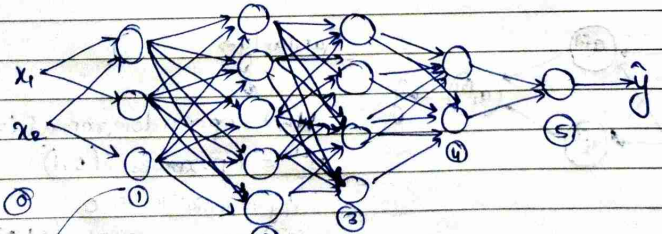
For layer 2: $a^{[2]} = g^{[2]}(z^{[2]})$ $A^{[1]} = g^{[1]}(z^{[1]})$

$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$ $z^{[2]} = w^{[2]} A^{[1]} + b^{[2]}$

$a^{[4]} = \sigma(z^{[4]})$ $A^{[2]} = g^{[2]}(z^{[2]})$

→ Stacking ↑

L-26 → Dimensions for parameters $w^{[l]}$ & $b^{[l]}$ →



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

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

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

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

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

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

$(5,1)$ $(5,3)$ $(3,1)$ $(5,1)$

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

$w^{[4]} : (2,4)$

⇒ $w^{[l]} : (n^{[l]}, n^{[l-1]})$

→ $b^{[l]} : (n^{[l]}, 1)$

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

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

↳ As in training samples

$z^{[1]} \rightarrow : (n^{[1]}, m)$

↳ for b → python broadcasting

→ Deep Representation → simple fun^s like

→ Early layers can detect the edges of the image, and composing them together in later layers for learning more complex fun^n

→ Extracting features ✓

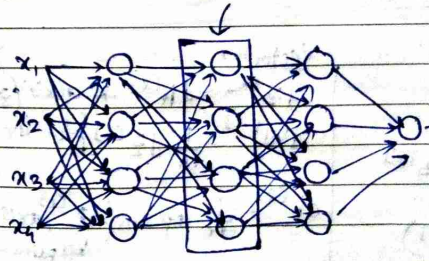
→ When 'not more' hidden layers are needed, then we need to have exponentially large no. of hidden units.

↳ i.e. (limited)

$\sim 2^{(n-1)}$

$O(2^n)$

L-27



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

→ forward: Input $a^{[l-1]}$, output $a^{[l]}$ (also cache $z^{[l]}$)

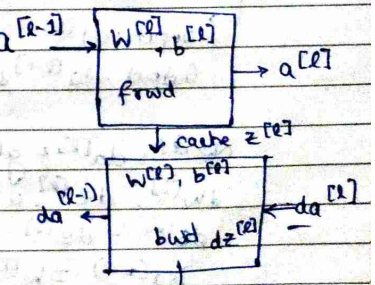
$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

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

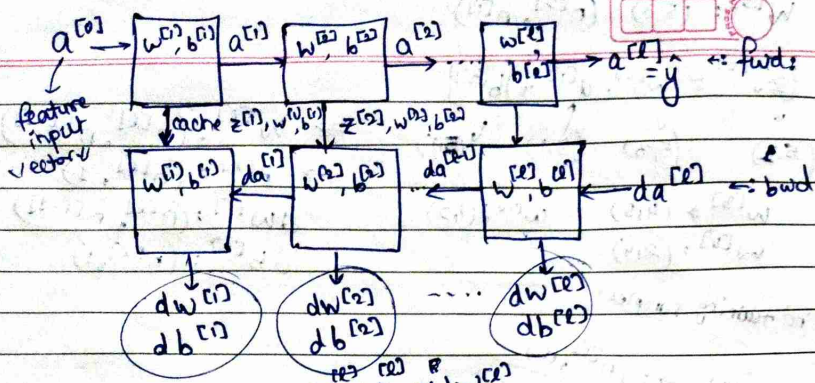
← backward: Input $da^{[l]}$, output $da^{[l-1]}$

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

Layer l: →



→ Basic Mecha. →



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

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

One-iteration of gradient descent ✓✓

* Forward Propag^{n} → for layer 'l'

↳ Input: $a^{[l-1]}$

output: $a^{[l]}$, cache ($z^{[l]}$)

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

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

• Vectorisⁿ →

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

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

* Backward Propag^{n} → for layer 'l'

↳ Input: $da^{[l]}$

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

- $dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$
- $dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$
- $db^{[l]} = dz^{[l]}$
- $da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$
- $dz^{[l]} = w^{[l+1]T} \cdot dz^{[l+1]} * g^{[l]\prime}(z^{[l]})$

Vectorisⁿ →

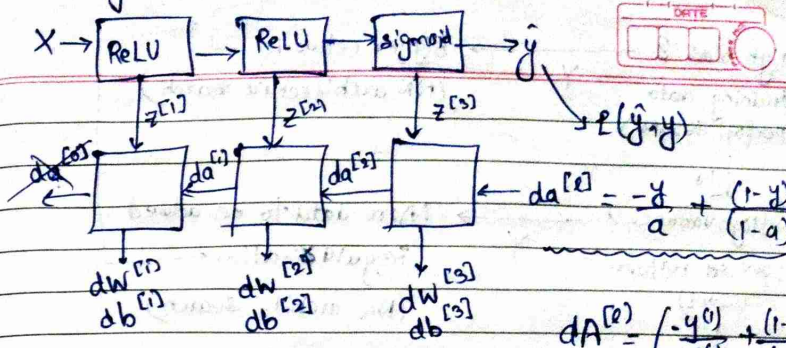
$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$

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

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

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

→ Summary:



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

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

L=28 →

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

→ Hyperparameters: (control W & b)

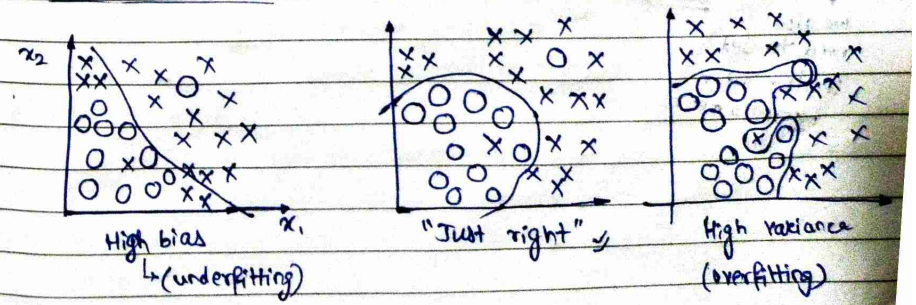
- Learning rate (α)
- # iterations (# : no. of)
- #hidden layers
- #hidden units: $n^{[1]}, n^{[2]}, \ldots$
- Choice of activⁿ funⁿ

Course=2 →

→ Machine learning model → Training set
→ Hold-out / Cross-Validⁿ / Development (dev) set
→ Test set

→ Make sure dev and test set come from same distributing

↔ Bias & Variance:



→ Basic Recipe:

- High bias? (training data perfor. issues) → Y → Bigger network (NN architecture search)
  - N
- High variance? (dev set perform. issues) → Y → More data to be added / Regularisation (NN archit. search)
  - N
- Done

Regularization

Euclidean Norm: $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2 + \dots + x_n^2}$

Position of $x: (x_1, x_2, x_3, \dots, x_n)$ (basically q: distance from origin)

↳ for logistic regression =

$\min_{w,b} J(w,b) \quad w \in \mathbb{R}^{n_x}, \; b \in \mathbb{R}$

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

omit

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

L_1 regul: $\|w\|_1 \rightarrow \frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$

w → has more parameter so we include it (matrix)

→ λ: Regularization parameter (lambda)

we also need to set it like other hyperparameters.

↳ Neural Network →

$$J(w^{[1]}, w^{[2]}, w^{[3]}, \dots w^{[l]}, b^{[1]}, b^{[2]}, b^{[3]}, \dots b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2$$

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

"Frobenius norm"

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

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

("weight decay")

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

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

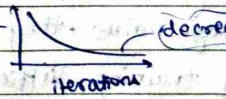
→ Regularization reduces overfitting (can) !!

↳ Large 'λ' = (but can have negative effects on our model if too large).

- If λ is too large, $w^{[l]} \downarrow$ → losing weights ↓ (losing relevant features)

→ To debug the gradient descent →

1) Create a plot of the cost fun. J to the no. of iterations



$$\Rightarrow \quad J(\dots) = \sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_l \|w^{[l]*}\|_F^2$$

Dropout Regularization

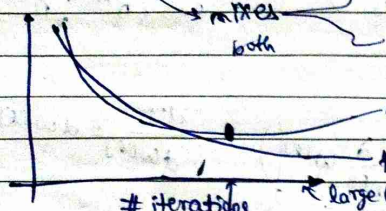
↳ Based on a "specific probability" some nodes from the hidden layers are dropped (eliminated) out & with the few left nodes we train our model. (May seem vague but works!)

→ Data Augmentation: (Regularization)

↳ A few changes in data

↳ If an image is provided, flip it/rotate it, we get new data to train & be efficient.

☆ Early Stopping →



* Orthogonalization:
  - Optimise cost fun J
  - Not overfit

Both

mid-size $||W||_F^2$

* Normalizing training Data → (Done so that we get a symmetric structure easy for gradient descent to take steps).

↳ 1) Subtract mean:

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

$$X := X - \mu$$

Normalise variance:

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

→ The model take a very long time if the derivates or gradients take a very small or large values, they can become exponentially smaller/larger making training difficult. (Vanishing) & (Exploding)

$W^{[l]}$ > Identity matrix (large-very)

$W^{[l]}$ < Identity matrix (very small)

$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$

Large (n) → smaller w_i

no. of input features

→ Variance:

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

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

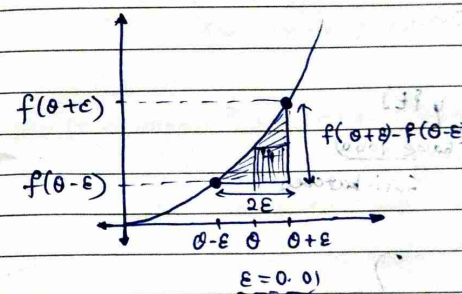
→ for tanh:

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

Xavier's initialisation

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

Derivative/Gradient Computation:



$\varepsilon = 0.01$

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

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

$g(\theta) \approx 3.0001 \approx 3$

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

app. error → 0.0001

→ We take $W^{[1]}, b^{[1]} \dots, W^{[L]}, b^{[L]}$ & reshape it into a big vector 'θ'

(concatenate)

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

→ We take $dW^{[1]}, db^{[1]}, \dots dW^{[L]}, db^{[L]}$ and reshape into a big vector 'dθ'

↳ for each 'i',

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

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

Check, $\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2}$ $\approx 10^{-7}$ or smaller → then great approximation

$\varepsilon = 10^{-7}$

* Tips →
  - → Dont use Grad. Checking in training - to debug
  - → If algo. fails gradcheck, loop at other components to identify the bug
    i.e. if $d\theta_{approx.}$ is very far from $d\theta$, then for diff i, which values give very diff
  - → Remember Regularization
  - → Doesn't work with Dropout

→ If we have a very large nos of training samples we divided then in few groups called (mini-) batches.

$X^{\{t\}}, Y^{\{t\}}$

↳ Mini-batch gradient descent =

→ 1 step of grad. descent using $X^{\{t\}}, Y^{\{t\}}$ (as if we have 1000 mini-batches)

repeat {

for t = 1, ..., 5000 {

→ forward prop. on $X^{\{t\}}$ (vector implementation (1000 examples))

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

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

⋮

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

Now we can process 1000 mini-batches of training sets than large amount together.

→ Compute cost fun. :

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

(for $X^{\{t\}}, Y^{\{t\}}$)

→ backprop. to compute grad. over $J^{\{t\}}$ (using $X^{\{t\}}, Y^{\{t\}}$)

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

}

}

this is 1 pass → "1 epoch"

→ Mini-size batch size must be between (1, m), not too small (↳ If this also doesn't lead to minimum (decrease α)) not too large.

- → If size = 1 → (Stochastic grad. descent)
  - ↓ loses speed up from vectorisation
  - (never hits minimum can lead to wrong direction)
- → If size = m → Batch grad. descent.
  - ↳ Too long time per iteration

• Guidelines →

- → for small training set : Use batch grad. descent. ($m \leq 2000$)
- → for large — | — : Use mini-batch sizes
  - ↳ $m \in (64, 128, 256, 512, 1024)$
  - 2^p → efficient
- → Make sure mini-batches $X^{\{t\}}, Y^{\{t\}}$ fits in CPU/GPU.

→ To implement Exponentially weighted Averages :

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

V_t as approximately avg. over: $\frac{1}{1-\beta}$

→ Gradient descent momentum. →

→ Momentum :

On iteration t :

Compute dW, db on current mini-batch

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

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

velocity

friction (<1) ⇒

$W = W - \alpha V_{dW}$, $b := b - \alpha V_{db}$

make smoother

$(1-\beta)\,dW$ as (θ_t) omitted may be in few cases

Hyper-parameters → α, β $\beta = 0.9$ (pretty well)

→ Root mean Square prop → (β_2 : hyperparameter)
(RMS)

element-wise

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

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

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

delete if ≈ 0

we can have large α without much deviation in our oscillations.

→ Adam Optimisation algorithm →

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

On iteration t :

Compute dw, db using current mini-batch

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

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

combined

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

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

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

Hyperparameters :

→ Very effective ✓

Imp. } α : needs to be tuned often
β_1 : 0.9 (dw)
β_2 = 0.999 (dw^2)
ε : 10^{-8} (recommended)

→ Learning Rate Decay →

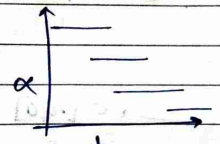
1 epoch = 1 pass

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

→ Other formulas :-

$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0$ → exponentially decay rate.

$\alpha = \frac{k\ (\text{some constant})}{\sqrt{\text{epoch-num}}} \cdot \alpha_0$ or $\frac{k}{\sqrt{t}} \cdot \alpha_0$ (t : mini-batch no.)



discrete stair case

* Tuning Process →

→ Priority for hyperparameters to be tuned :-

1. α → most imp.
2. β [(default) = 0.9]
   $\beta_1 \sim 0.9$ $\beta_2 \sim 0.99$ $\varepsilon \sim 10^{-8}$
3. no. of hidden units
4. mini-batch size
5. no. of layers
6. learning rate decay

→ Choosing random values rather than grid allows us to explore diff. possible values for most imp. hyperparameters.

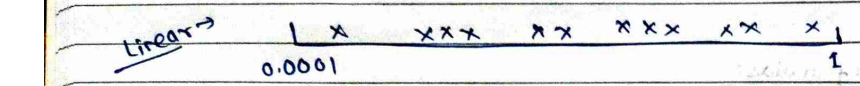
→ In Coarse to fine approach from a bigger area we get a smaller region which have better values and get our hyperparameters (random)

⇒ Logarithmic scale makes more sense & work's better than a linear scale.

→ Appropriate scale for hyperparameters →

Let say $\alpha \in [0.0001, \ldots, 1]$

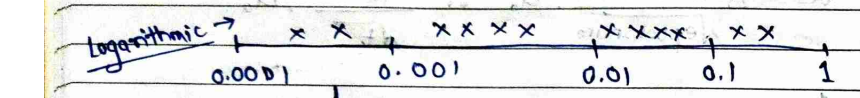
Linear →



0.1 1

not good.

Logarithmic →



Uniform & good (better)

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

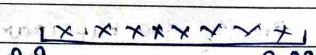
$\alpha = 10^r$ ⟷ $r \in [-10^{-4}, 10^0]$

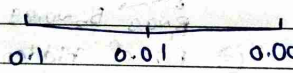
→ for exp. weightage averages :-

$\beta = 0.9$ 0.999

10 ⟷ 1000

$1-\beta = 0.1$ 0.001





10^{-1} ⟷ 10^{-3}

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

$\boxed{\beta = 1 - 10^r}$

* Two approaches :- → 1) Panda approach (babysitting the model)
↳ when u have large models & diff. to train

→ Caviar approach (training models parallelly)
↳ & pick the best one
↳ / enough computers to train models

→ Normalising input features can speed up learning.

→ $\mu = \frac{1}{m} \sum_i x^{(i)}$

$x = x - \mu$ ✓ element wise

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

$x = x / \sigma$

→ But activations are also included, Can we normalise $a^{[2]}$ so as to train $w^{[2]}, b^{[2]}$ faster?

↓

Actually normalisation of $z^{[2]}$ take place ☺!

→ Given some intermediate values in NN. → l^{th} layer ✓

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

$\mu = \frac{1}{m} \sum_i z^{(i)}$ → $z^{[l](i)}$

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

$\beta = 0$, $\gamma = 1$ → original paper recommends

$z^{(i)}_{norma.} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \Rightarrow \tilde{z}^{(i)} = \gamma z^{(i)}_{norma.} + \beta$

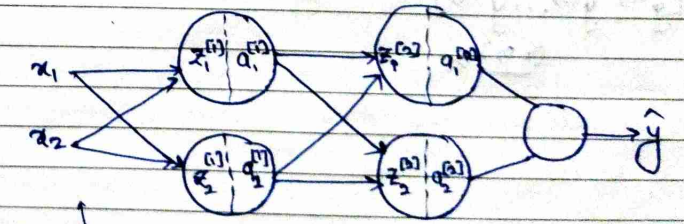
don want mean as 0 and variance

learnable parameters of the model (allows set us mean other than 0/1) variance

$\gamma = \sqrt{\sigma^2 + \varepsilon}$, $\beta = \mu$ } true? : $\tilde{z}^{(i)}_{norma.} = z^{(i)}_{norma.}$ ✓ ☺!!

→ Use $\tilde{z}^{(i)}$ instead of $z^{(i)}$ ✓ ☺!!

* Fitting →



Without norm!

With norm: $X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \ldots = a^{[2]} \leftarrow \tilde{z}^{[2]} \xleftarrow[\text{B.N.}]{\beta^{[2]}, \gamma^{[2]}} z^{[2]} \xleftarrow{w^{[2]}, b^{[2]}}$

* → Softmax →

↳ Suppose we have 4 classes,

$C = 4$

$(4,1)$

Let $z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$ $\quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$

Softmax activ^n fun^n $\rightarrow g^{(l)}(\cdot)$

S.A.F. $\longrightarrow$ $g^{(l)}(z^{[l]}) = \begin{bmatrix} e^5 / \sum_{j=1} t_i \\ e^2 / e^5+e^2+e^{-1}+e^3 \\ e^{-1} / \sum_{j=1} t_i \\ e^3 / \sum_{j=1} t_i \end{bmatrix} = \begin{bmatrix} 0.0842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ "hardmax"

→ Loss function :-

$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ cat $\quad y_2 = 1$

$y_1 = y_3 = y_4 = 0$

$a^{[l]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ $\quad C = 4$

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

$= -y_2 \log \hat{y}_2$

$= -\log \hat{y}_2$

(make $\hat{y}_2 \rightarrow$ large)

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

Zero gradient → Saddle point ✓
(derivative)

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

$(4, m)$