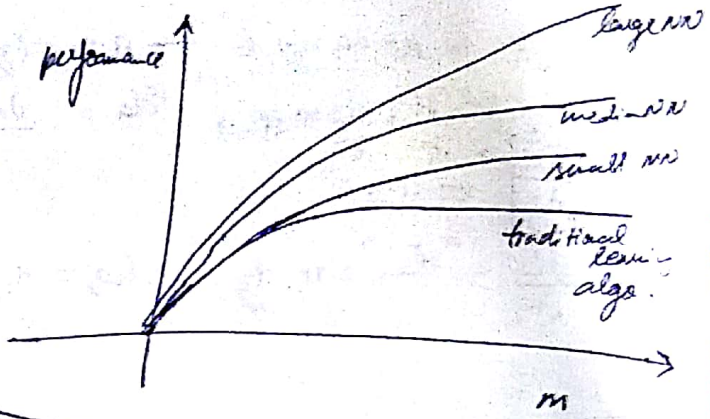
Deep Learning.

Neutral Networks

input output
(x, y) Supervised learning

- Standard neural network (SNN)
- image → convolutia neural network (CNN)
- sequence (eg audio, language) → recurring neural network (RNN)
- hybrid. neural network (combination)

Structured data – table, tabular form
Unstructured data – image, audio, Text



m → training set (labelled data).

Binary Classificatia n_x (0 or 1)

Notatia: (x, y) $x \in \mathbb{R}^n$ $y \in \{0, 1\}$ m_{train}, m_{test}.

$X = \mathbb{R}^{n \times m}$ $Y = \mathbb{R}^{1 \times m}$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \begin{matrix} \uparrow \\ n_x \\ \downarrow \end{matrix} \quad Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$
$$\leftarrow m \rightarrow$$

Logistic Regression.

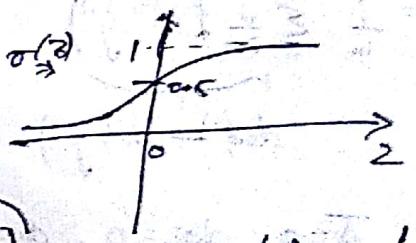
Given x want $\hat{y} = P(y=1|x)$ } Output.

$x \in \mathbb{R}^{n_x}$ ⇒ matrix. $(n_x \times 1)$ $\hat{y} \in [0, 1]$ }

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

matrix $(m \times 1)$

we want $\hat{y}^{(i)} \approx y^{(i)}$



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

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

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

so $y=1$ → we want $\hat{y}$ large
& $y=0$ → we want $\hat{y}$ small

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

↑ for single example.

for large (so average)

Gradient descent ⇒ find w, b that minimise $J(w, b)$.

(Take any random point, move towards optimum, finally reach.)

Global optimum (Convex so works)

Repeat {

$w := w - \alpha \frac{d J(w)}{dw}$ ⇒ written as "dw" from now.

}

to $w := w - \alpha dw$.

to for $J(w,b)$. dw

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

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

---

x_1, w_1, x_2, w_2, b

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

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

"da" $= \frac{\partial L(a,y)}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$

$\frac{\partial L}{\partial z} = a - y$

$\frac{\partial L}{\partial w_1} =$ "dw_1" $= x_1 \cdot dz$ $dw_2 = x_2 \cdot dz$ $db = dz$

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

Logistic regression on m examples.

$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$ $a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$

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

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

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

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

For $i = 1$ to m

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

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

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

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

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

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

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

$n = 2$ (if n more, more loops)

$J /= m$

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

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$.

* (2) for loops

(we have to reduce for loops)

(* Vectorization used to reduce them)

Vectorization

$z = w^T x + b$ (np = numpy)

Non-vectorized

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

vectorized (very fast)

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

$w^T x$

w, x — array.

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

$u = np.exp(v)$

np.log(v)

np.abs(v)

np.maximum(0, v)

v**2 (square)

1/v (inverse of each)

np.softmax(): row normalize karta hai direct. e^x lekar jaata.

Thus previous one, the second for loop can also be removed.

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

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

$(1 \times n_x)(n_x \times m)$

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

Contains all $z^{(1)} \cdots z^{(m)}$ — transpose — (1,m) (1,1)

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

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

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

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

so $dZ = A - Y$

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

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

New logistic regression:

$Z = w^T X + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

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

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

$w := w - \alpha dw$

$b := b - \alpha db$

Now put a for loop for (suppose 1000 iterations) to reach the min. point

Then will be needed.

Broadcasting in Python:

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

cal = A.sum(axis = 0) → [6 15 24] sum.

percentage = 100 * A/(cal.reshape(1,3)) — Broadcasting — this is not needed as it is originally (1,4) only.

each row wise % calculate.

so percentage = 100 * A/cal — (3,3) / (1,3) — division

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

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

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

Broadcasting.

so applicable for + − * /.

a = np.random.randn(5) → rank 1 array. → Don't use. (5,)

use ⇒ a = np.random.randn(5,1) → Column vector. use reshape to convert always.

Tip: X.reshape(X.shape[0], -1).T ⇐ (a,b,c,d) → (b*c*d, a) — transpose

b kr baad c aur baad d. Size ho jayega b*c*d.

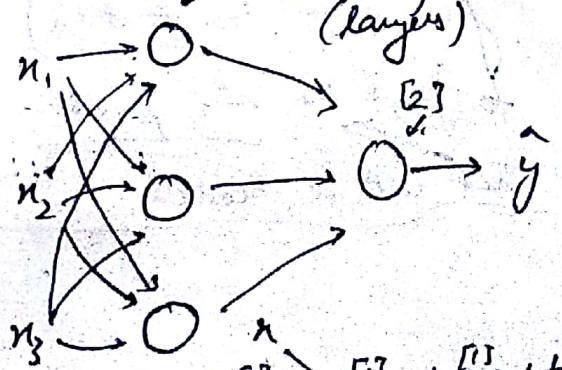
Neural Network.

$x_1, x_2, x_3 \rightarrow \bigcirc \rightarrow \hat{y} = a$

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

da dz

[1] (square bracket) (layers)



[2]

$da^{[2]}$

$\mathcal{L}(a^{[2]}, y) \leftarrow a^{[2]} = \sigma(z^{[2]})$

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

$dW^{[1]}, db^{[1]}$ $W^{[2]}, b^{[2]}$ $dz^{[2]}$

One hidden layer.

2 layer NN.

$z = w^T x + b$

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

hidden + output

linear activation is considered as a single layer. Sigmoid or ReLU.

$a^{[0]} = x$ — $a^{[1]}$ — $w^{[1]}$ $(4,3)$, $b^{[1]}$ $(4,1)$ — $w^{[2]}$ $(1,4)$, $b^{[2]}$ $(1,1)$

x_1, x_2, x_3 — $a_1^{[1]}$, $a_2^{[1]}$, $a_3^{[1]}$, $a_4^{[1]}$ — $a^{[2]} \rightarrow \hat{y} = a^{[2]}$

input layer — hidden layer — output layer

layer 0 — layer 1 — layer 2.

$a^{[l]}$ ← layer

a_i ← node in layer.

$a_i^{[l]} = \sigma(z_i^{[l]}) = \sigma(w_i^{[l]T} x + b_i^{[l]})$

~~$w^{[1]} x + b^{[1]}$~~

$x = a^{[0]}$

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

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

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

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

Now loop mein yeh sab data.

for i to m

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

multiple examples ⇒ (

layer 2

$a^{[2](i)}$ → example i

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

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

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

⇒ How not to use loop

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

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

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

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

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

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

hidden units. — corresponds to different training examples. — different nodes in neural network.

Same for Z, A, X.

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

(so w, b for different set of training examples do remain same.)

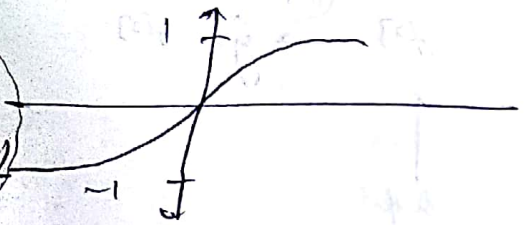
Basically, $z^{[1](1)} = w^{[1]} x^{(1)} + b^{[1]}$; $z^{[1](2)} = w^{[1]} x^{(2)} + b^{[1]}$, $z^{[1](3)} = w^{[1]} x^{(3)} + b^{[1]}$

Activation functions

Example: sigmoid $(\sigma(x))$ $\frac{1}{1+e^{-z}}$

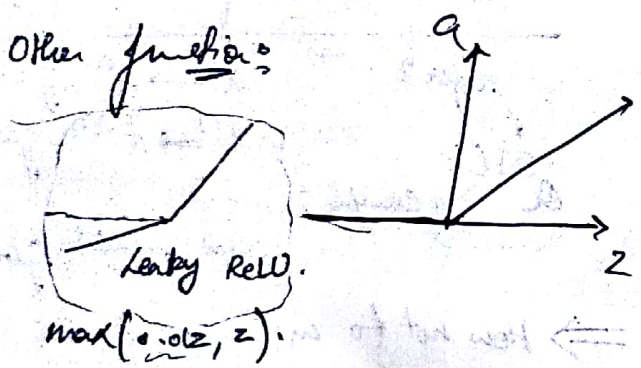
$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

when $y \in (0,1)$ so $\hat{y} \in (0,1)$ we can use sigmoid. (binary classification)



★ instead of sigmoid if we use $g(z)$ it works better.

Other functions:



$a = \max(0, z)$

ReLU (Rectified Linear Unit)

Max people use if. (as $g(z)$ mean +1, -1 slope is very less)

★ $y \in \mathbb{R}$, then we can use linear function other than that we will use non-linear only.

Non-linear Activation function is a critical part.

$a = g(z) = \frac{1}{1+e^{-z}} \Rightarrow g'(z) = g(z)(1-g(z)) = a(1-a)$ [sigmoid]

$a = g(z) = \tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}} \Rightarrow g'(z) = 1 - (\tanh(z))^2 = 1 - a^2$ [tanh]

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

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

Gradient descent for neural networks.

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

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

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

Gradient descent:

Repeat {
Compute predicts ($\hat{y}^{(i)}$, $i = 1, \dots m$)
$dw^{[1]} = \frac{\partial J}{\partial w^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, ...
$w^{[1]} := w^{[1]} - \alpha dw^{[1]}$
$b^{[1]} := b^{[1]} - \alpha db^{[1]}$
}

Summary

Forward propagation:

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

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

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

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

Backward propagation:

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

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

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

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

rank 1 array na aae isliye. ($n^{[2]}$, 1) ✗.

derivative

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

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

↳ ($n^{[1]}$, 1)

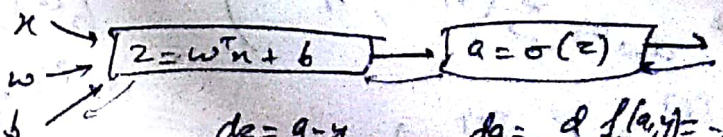
6 imp for back propagation

Backpropagation

$w^{[2]} \to n^{[2]}, n^{[1]}$

$z^{[2]}, dz^{[2]} \to (n^{[2]}, 1)$

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



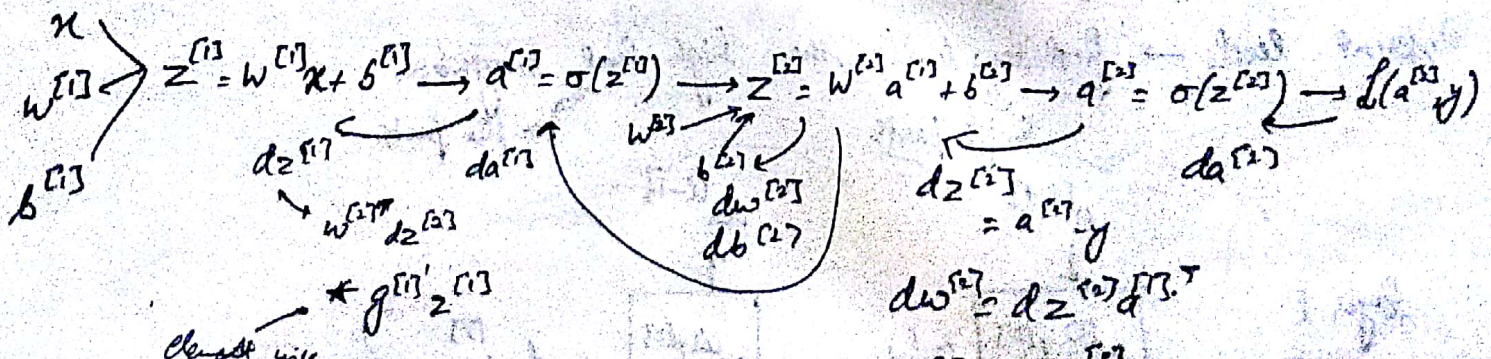
$dz = a - y$ $\left\{\frac{\partial L}{\partial z}\right\}$

$da = \frac{d}{da} \mathcal{L}(a,y) = -y \log a - (1-y) \log(1-a)$

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

$dw = dz \cdot x$

$db = dz$



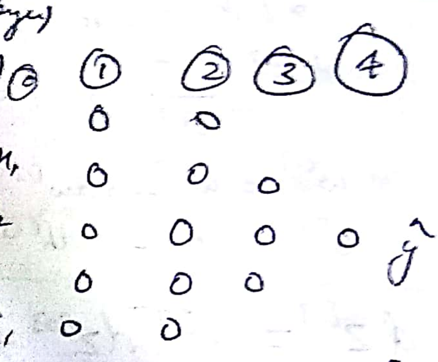
Initialization

w ko random, b ko 0 chalega.
but dono ko agr 0 de dia to saare
same aane lagenge → problem

$w^{[1]} = np.random.randn((2,2)) * 0.01$
$b^{[1]} = np.zeros((2,1))$ — w ko chota rakhe ke lie
$w^{[2]} =$
$b^{[2]} = 0$

Deep Neural Network.
→ Lot of hidden layers

logistic regression.
(0 hidden layers)
(1 layer)



4 layer NN → 3 hidden layers.

$L = 4$ (# layers) $n^{[l]}$ = # units in layer l.
so $n^{[1]} = 5$; $n^{[2]} = 5$; $n^{[3]} = 3$; $n^{[4]} = n^{[L]} = 1$
also $n^{[0]} = n_x = 3$

$a^{[l]}$ = activations in layer l := $g^{[l]}(z^{[l]})$ $w^{[l]}$ = weights for $z^{[l]}$

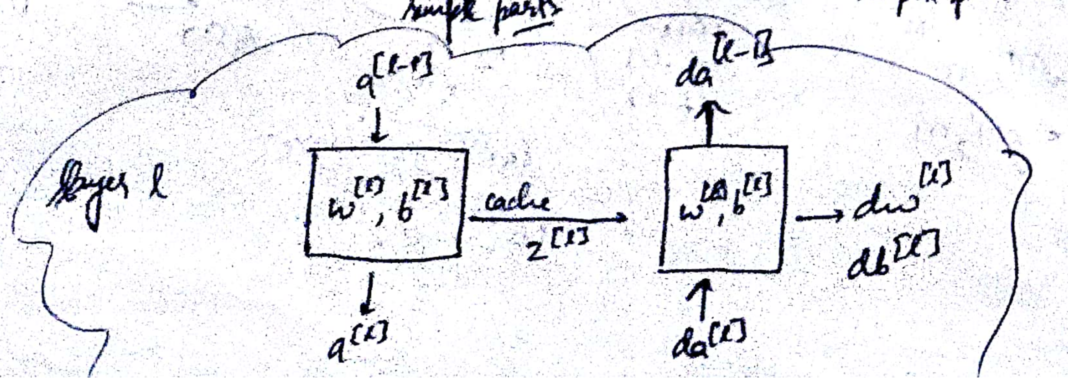
basically; $Z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$; $a^{[l]} = g^{[l]}(z^{[l]})$; $a^{[4]} = \hat{y}$.
and a → A, etc. for vectorically.
and alag alag layers ke lie for loop use krna padega.

Dimensions: $W^{[l]} \Rightarrow (n^{[l]}, n^{[l-1]})$; $dw^{[l]}$ ↗
$b^{[l]} \Rightarrow (n^{[l]}, 1)$; $db^{[l]}$ ↗ (vaise broadcasting se vaise bhi ho jaegi)

$A^{[l]} \to (n^{[l]}, m)$ — same as Z.

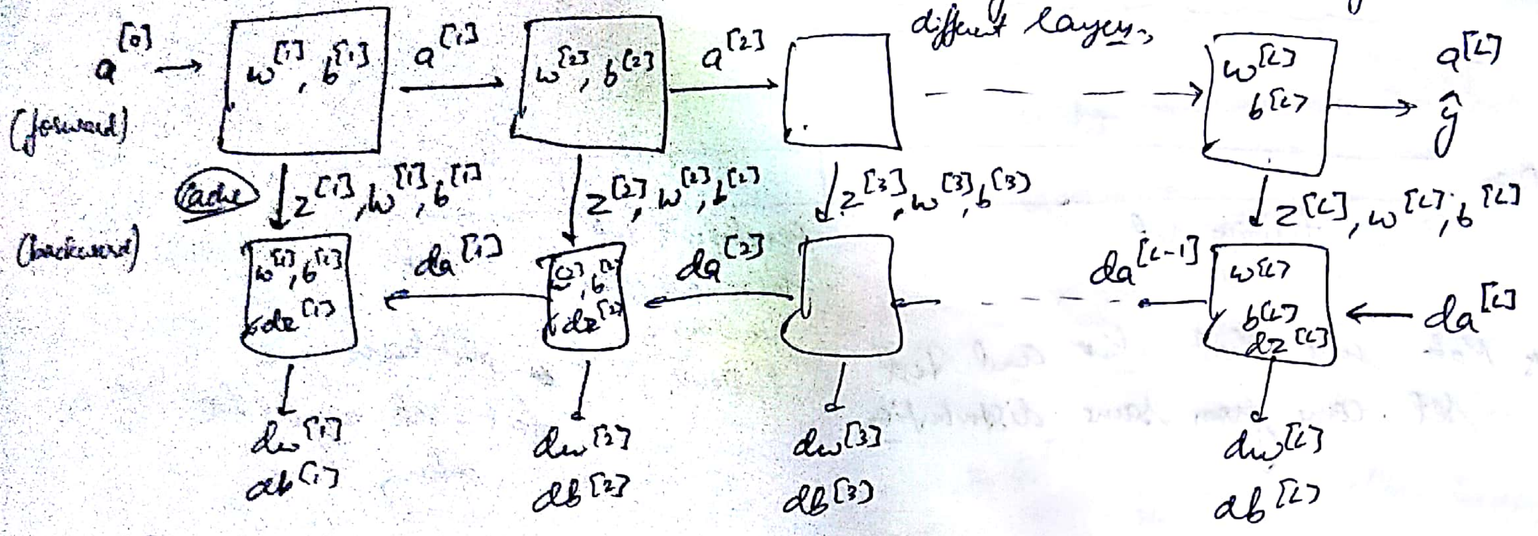
(OR) $b^{[l]} \Rightarrow (n^{[l]}, m)$

Different levels → first low level, then high level recognition
(simple parts) (complex parts.)



Scanned by CamScanner

In each layer, there is forward propagation step & backward propagation step
diffent activation functions also possible for diffent layers.



$dz^{[l]}$ likhna ho toh maan
le $w^{[l]}, b^{[l]}$ bhi aata hai.

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

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

FORWARD

→ Input $a^{[l-1]}$ $\quad w^{[l]}, b^{[l]}$

→ Output $a^{[l]}$, cache $z^{[l]}$

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

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

Backward

→ Input $da^{[l]}$

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

element wise product.

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

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

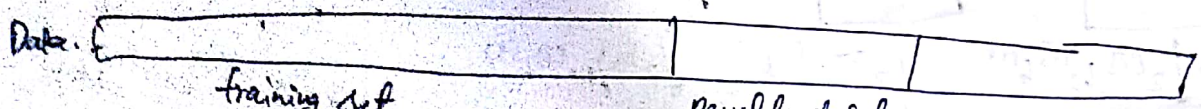
$$db^{[l]} = \frac{1}{m} \text{np.sum}(dZ^{[l]}, \text{axis}=1, \text{keepdims}=\text{True})$$

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

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

Hyperparameter: α, # iterations, # L, # $n^{[1]}, n^{[2]} \ldots$, Choice of activation f^n, etc.
(we have to adjust them ourselves and check the performance).

IMPROVING DEEP NEURAL NETWORKS.

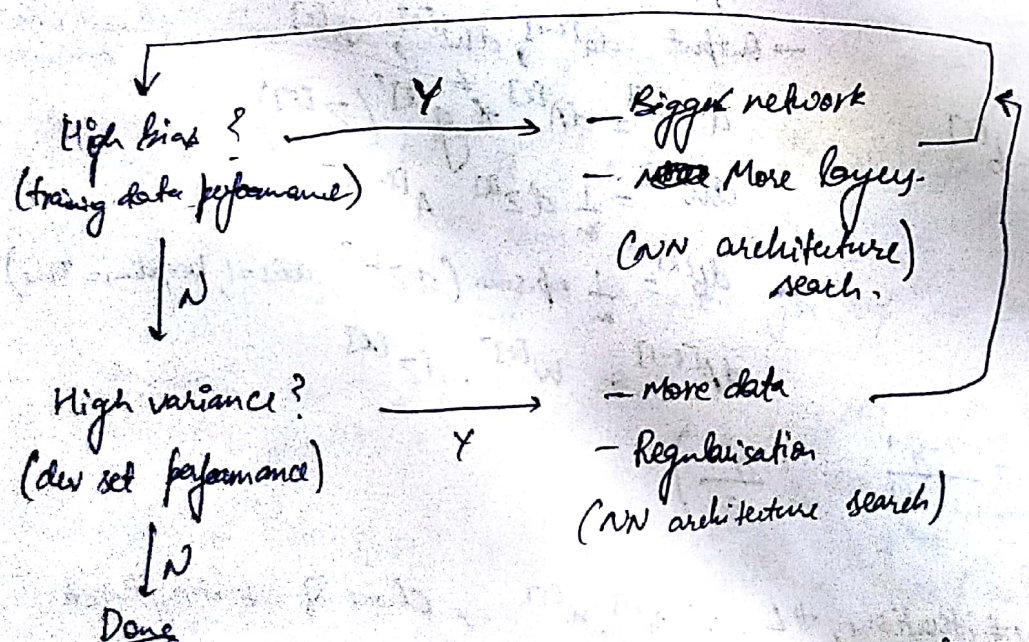


⚠ Make sure dev and test set come from same distribution.

Not having a test set might be okay.

underfitting → high bias
overfitting → high variance
→ higher dev set error.
(train set error ≈ dev set error = relatively high.)

high bias + high variance → high train set error + much higher dev set error.



Don't use if not overfitting.

Regularization ⇒ (prevents overfitting) ⚠

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

regularization parameter.

$\Rightarrow$ usually omit. (big doesn't have much effect)

λ also a hyperparameter.

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

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

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

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

(Frobenius norm)

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

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

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

(1) Dropout regularization: Thode nodes ko hata dete hai.

(2) Inverted dropout. (keep-prob = 0.8)

d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep-prob. → prob. so 80% of being 1, 20% of being 0

(d3 → dropout vector for layer 3) → also works for vectorisation

and a3 = np.multiply(a3, d3) # jo chote the unko 0 kr diya

a3 /= keep-prob ## scaling it up. → To keep expected value to remain same.

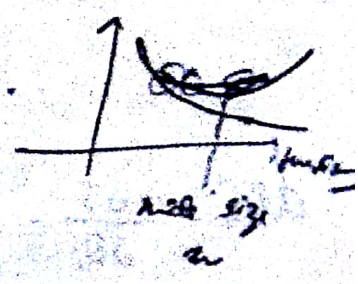
To zero-out some hidden units.

Keep-prob = 1.0 → keeping every unit (no dropout). → always for input

keep-prob can vary for every layer.

(2) Data augmentation: same image ko invert, distort for more input

(3) Early stopping: plot training error, dev set error also.



not good. ⟹ Prefer to use L_2 than this. ⇓ better.

Normalising training set : Basically variance in x,y equal to 1 & horne ke lie.

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

$X := X - \mu$

Use same μ, σ^2 to normalise test set.

normalise variance $\Rightarrow \sigma^2 = \frac{1}{m}\sum_{i=1}^{m} X^{(i)} **2$ ← element wise

$X /= \sigma^2$

used so that features come on almost similar scales.
Cost function changes easier and in better way.

Vanishing/exploding gradients : In case of very deep neural networks $L \geq 100$.
The gradients can be very large or very small, thus causing a problem.

Isko control krne ke lie, there are various research papers. like jb. we do random initialisation then multiply by $\sqrt{\frac{2}{n^{[l-1]}}}$ for $w^{[l]}$ when using ReLU function or multiply by $\sqrt{\frac{1}{n^{[l-1]}}}$ or $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$ for $w^{[l]}$ when using tanh, etc.

for derivative calculation: use $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ ← much more accurate.
rather than $\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$.

$d\theta[i] = \frac{dJ}{} = \frac{J(\theta_1, \theta_2, \ldots, \theta_i+\varepsilon, \ldots) - J(\theta_1, \theta_2, \ldots, \theta_i-\varepsilon, \ldots)}{}$

Gradient Checking implementation notes:

1. Don't use in training - only to debug $dw_{approx}[i] \leftrightarrow dw[i]$
2. If algo fails grad check, look at the components to identify bug
3. Remember regularisation $\frac{\lambda}{2m}\sum_l \|w^{[l]}\|_F^2$
4. Doesn't work with dropout. → keep prob = 1.0
5. Run at random initialisation; perhaps again after some training.

Mini-batch gradient descent

Mini-Batches

If training set: $X^{\{1\}}, X^{\{2\}}, \ldots$ $Y^{\{1\}}, Y^{\{2\}}$

(when very large training set, make mini batches). $X^{\{t\}}, Y^{\{t\}}$

$X^{(1)} X^{(2)} \ldots X^{(k)}$

Suppose $k = 1000$, then mini batch no. also $t \in 1 \to 5000$

for $t = 1, \ldots, 5000$ {

Forward prop. on $X^{\{t\}}$.

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[k]} = g^{[k]}(Z^{[k]})$$

} Vectorial implementation.

This is doing → 1-epoch → single pass through the training set.

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

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

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

}

repeat if for 2nd epoch

Choosing your mini-batch size =>

Stochastic gradient descent → size = 1. → might never converge, etc.

Batch gradient descent → size = m (Too long per iteration)

Small train set ($m \leq 2000$): use batch gradient descent.

Typical mini-batch sizes ⇒ 64, 128, 256, 512, 1024
2^6 2^7 2^8 2^9 2^{10}

Make sure mini-batch fit in CPU/GPU memory.

Exponentially weighted (moving) averages.

Initialise $v_0 = 0$

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

θ_t we know for individual days.

$\beta \simeq 0.9$ → average over 10 day

$\beta \simeq 0.98$ → average over 50 day.

$\beta \simeq 0.5$ → average over 2 day → noisy. ✗

Intuitively, because $v_0 = 0$, subsequent causes problems.

Initial phase of learning we'll bias correction from padega.

Gradient descent with momentum

(To hume different learning rates chahiye hote hai) for horizontal & vertical direction

always work better.

Momentum:

$v_{dw} = v_{db} = 0$

On iteration t:

Compute dw, db on current mini batch

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

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

Consider it like friction velocity acceleration.

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

$b := b - \alpha v_{db}$

$v_{dw}^{corrected} = \frac{v_{dw}}{1-\beta^t}$

$v_{db}^{corrected} = \frac{v_{db}}{1-\beta^t}$

hyperparameter ⇒ β, α

RMSprop (Root mean square prop)

On iteration t:

Compute dw, db on current mini batch.

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

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

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

ε can also be 10^{-8}. Just to make sure 0 na ho jae.

$S_{dw}^{corrected} = \frac{S_{dw}}{1-\beta^t}$

$S_{db}^{corrected} = \frac{S_{db}}{1-\beta^t}$

Adam optimization algorithm

α → needs to be tune.

β_1 → 0.9 (dw)
β_2 → 0.999 (dw^2)
} Not much to be tuned

ε → 10^{-8} (does not matter much)

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

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 β_1

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$; $S_{db} = \beta_2 S_{db} + (1-\beta_2) db$ ← "RMS prop" β_2

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

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

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

Adam: Adaptive moment estimation

Learning rate decay: Slowly decaying α to get better result.

So can put $\alpha = \frac{1}{1 + (\text{decay-rate}) * \text{epoch-num}} \alpha_0$ ← initial learning rate. formulae can vary

$\alpha = 0.95^{\text{epoch-num}} \alpha_0$; $\alpha = \frac{k}{\sqrt{\text{epoch-n}}} \alpha_0$ OR discrete also possible

OR manual decay also possible.

★ Plateaus make learning slow.

Training approaches: ① Baby sitting one model (Pandas)
② Training many models in ||r (Caviar)

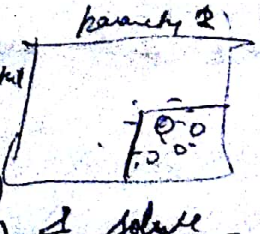
Hyper parameters: α, β, $(\beta_1, \beta_2, \varepsilon)$, #layers, # hidden units, learning rate decay, mini-batch size.
(most imp) (less (least imp).

Try ~~her~~ random values for better, then choose a regia where better hyperparameters, reduce the size so that area, the more points then.

Scaling for hyperparameters: can use log scale also.

$r = \log_{10}(\alpha)$ (eg. for α) and get value, $r \in [-4, 0]$

$\beta \in (0.9, 0.999)$
$1-\beta \in (0.1, 0.001)$ ← solve



Batch Normalisation : It normalises $z^{[l]}$, $z^{[l]}$ also for fast. OR $a^{[l]}$, $a^{[l]}$.

BN

$\mu = \frac{1}{m}\sum_i z^{(i)}$ — considered to be part of a layer. $z^{(1)}, \ldots z^{(n)}$ $\underbrace{\quad}_{z^{[l](i)}}$

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

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

$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$ — learnable parameters of model; different mean, momentum

Thus normalising the hidden layers, not only input layer.

Now we use $\tilde{z}^{(i)}$ instead of $z^{(i)}$

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

Now new param : $w^{[l]}$, ~~$b^{[l]}$~~, $\beta^{[l]}$, $\gamma^{[l]}$ (Gamma, γ)

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

$\Rightarrow z^{[l]} = w^{[l]} a^{[l-1]}$ (can be eliminated, can be considered in $\beta^{[l]}$)

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

This also works with various other optimization algo.

★ Batch norm reduces the problem of input-value changing. It keeps mean and variance same.

★ Batch norm regularizes. (Adds noise to each hidden layer's activation). (slight regularisation effect). ★ using bigger mini-batch size ⇒ reduces regularisation effect

★ μ, σ^2 pe running average like vahna padega.

Jab hum mini-batches banaenge tabe, μ, σ^2 will be different for different mini-batches. So while compiling, take average of μ, σ^2 and use that. for $z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ & $\tilde{z} = \gamma z_{norm} + \beta$

Softmax Regression

Recognizing different objects (3 objects + others) — 1 2 3, 0

Suppose $C = \#$ classes $= 4$ so $\hat{y} \to (4,1)$

Activation function for softmax layer: $t = e^{(z^{[L]})}$ (element-wise)

$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$$ → kind of probability

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1} t_i} \quad a^{[L]} = g^{[L]}(z^{[L]})$$

($\sum t_i$ ⇒ sum of all element)

→ Boundary to separate different classes is linear.

→ hard-max → highest wala → 1, baaki sab → 0.

Softmax regression generalises logistic regression to C classes.

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

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \; (4,1)$$

$= -\log \hat{y}_2$ so $\hat{y}_2$ → make it larger for less $\mathcal{L}$ (Loss) ($\hat{y}_2$ → prob.)

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

Now $Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}] = \begin{bmatrix} 0 & 0 & 1 & \\ 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}$

Back prop: $dz^{[L]} = \hat{y} - y$ & so on. ($(4,1)$)

Tensorflow:

```
tf.variable(   ,   )
tf.placeholder(   ,   )
tf.train.GradientDescentOptimizer(α).minimise(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init).
```

* cost ka eqⁿ banana padta hai, but w.r.t. (w), (b)

Example of of tensorflow code Cost fn => $w^2 - 20w + 25$

```
import numpy as np
import tensorflow as tf
coefficients = np.array([[1], [-20], [25]])
w = tf.Variable([0], dtype = tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]      # w^2 - 20w + 25
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

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

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

# it manages back-propagation as well.
```

Convolution Neural Network

Vertical edge detection : convolution, filter.

for vertical edges.

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Suppose $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$ to multiply this & add all the elements.

python : conv_forward

tensorflow : tf.nn.conv2d

$6\times6 * 3\times3 \rightarrow 4\times4$

(Grayscale image)

$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$ → for horizontal edges.

Sobel filter : $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ Scharr filter : $\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$

Image padding : 6×6 image is converted to 8×8 image. using padding.

It is used because the features & corner elements are usually lost, so to prevent it.

p = padding $\quad 6\times6 \rightarrow 8\times8 \rightarrow$ padding = 1.

$n\times n \quad f\times f$.

Valid convolution : no padding

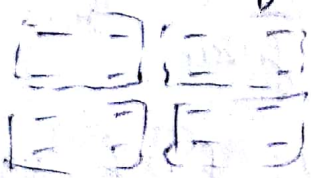
Same convolution : pad so that the output image is of same size as the input image.

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

f is usually odd.

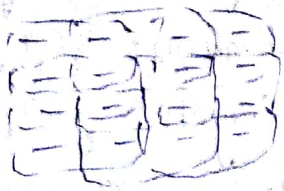
Strided convolution :

stride = 2 : instead of convolution just jumping by 1 step. it jumps by 2 steps.



so stride = 2

$7\times7 * 3\times3 \rightarrow 3\times3$. rather than



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

padding p stride s

$\frac{n+2p-f}{s}$ is not an integer then

Convolutions on RGB images

6×6×3 → # channels.
(height, width)

3×3×3

4×4

To detect vertical edges only in red channel ⇒

R: 1 0 -1 / 1 0 -1 / 1 0 -1
G: 0 0 0 / 0 0 0 / 0 0 0
B: 0 0 0 / 0 0 0 / 0 0 0

* ago we use two different filters, we can stack them up after doing both individually, final will be 4×4×2.

2 filters.

6×6×3 → * 3×3×3 → ReLU(4×4 + b_1) → 4×4

→ * 3×3×3 → ReLU(4×4 + b_2) → 4×4

3×3×3 = $w^{[1]}$; ReLU = g

4×4×2

1 layer of convolution network

Summary

If layer l is a convolution layer:

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

$p^{[l]}$ = padding

$s^{[l]}$ = stride

$n_c^{[l]}$ = no. of filters

Input : $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$

Output : $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

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

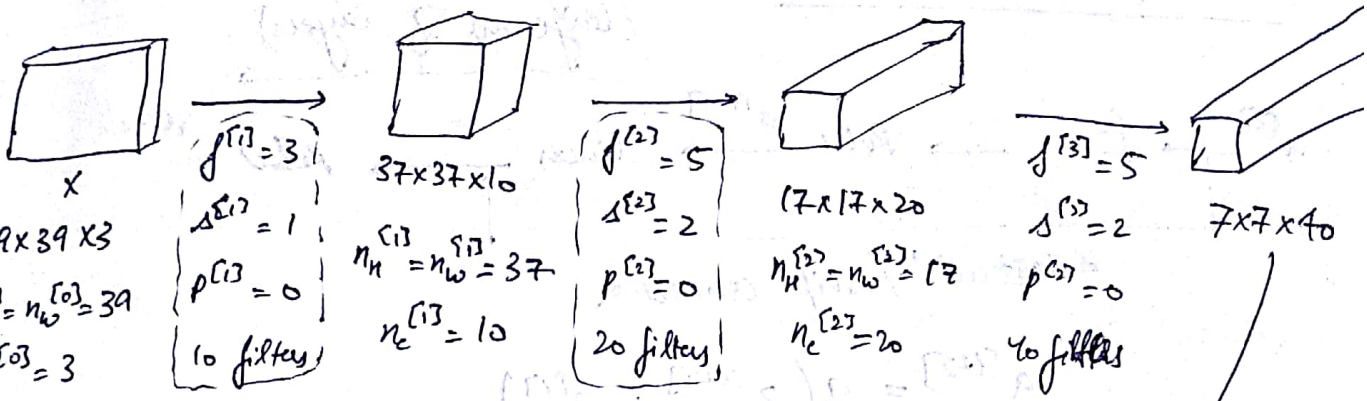
Each filter is : $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

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

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

bias : $n_c^{[l]} \rightarrow (1,1,1,n_c^{[l]})$ ← # filters in layer l

Example of a convolution Neural Network



$$\frac{n+2p-f}{s}+1$$

1960

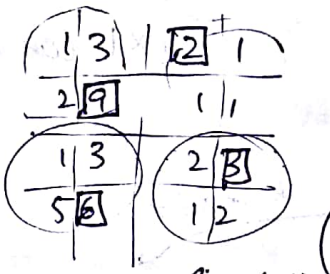
itne ka finally vectors banaye, is to be captured.

Types of layers in a convolution network:

- Convolution (CONV) → less parameters
- Pooling (POOL) → no parameters
- Fully connected (FC) → lots of parameters.

Pooling layers:- Max Pooling

→ Have no parameters.



→ 9 2 / 6 3

f, s can change also acc. to exam → This is correct

not correct.

f=2, s=2 → fixed hyperparameters

* Kuchh adjust nhi karna padta (w, b) etc.

Average Pooling: instead of max, we take average.

Pooling: No parameters to learn! Fixed function → parameters once set is set.

individually koi weight nhi lai jaiye.

* Conv 1 + Pool 1 ko usually mila dete hai and make them as a single layer.

⇒ Fully connected layers (jaisa pehle kiye the w, b wala)

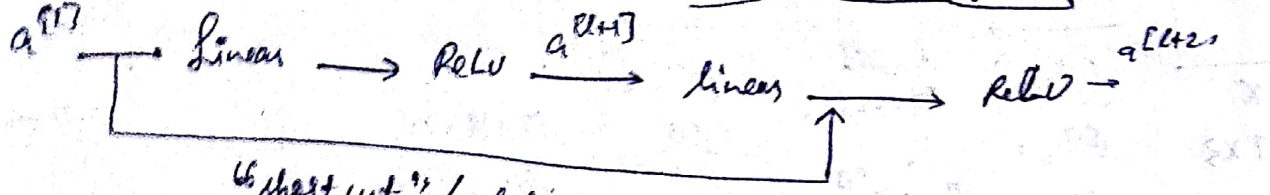
suppose finally 400 units aa rhi hai

so, then 400 → 120 → 84.

They are connected (fully connected layers)

to feed into softmax

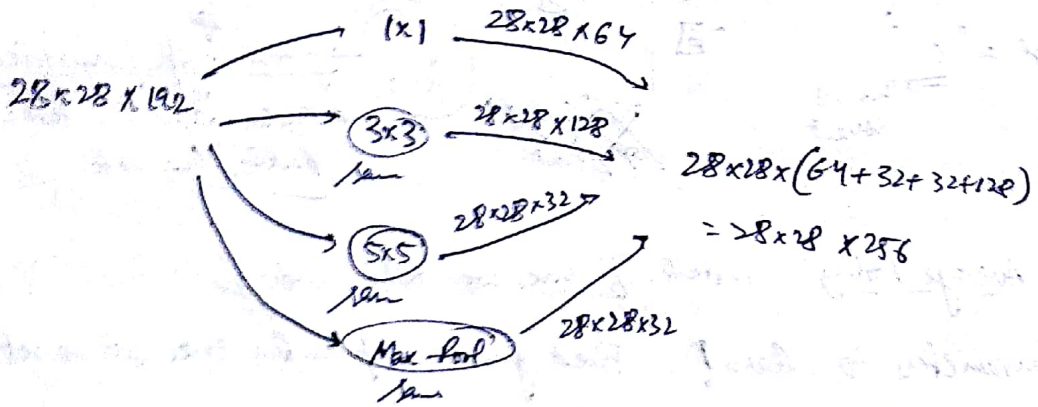
ResNet (Residual Network) :: To train very-deep layered networks. (Large no of layers)



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

So we add short-cut connection to plain-network we continuously get decreased training error with increase in no. of layers.

Inception Network : 1×1 convolutions.



$28 \times 28 \times 192 \xrightarrow[5\times5,\ \text{same}]{\text{CONV}} 28 \times 28 \times 32$ (32)

No computation $= 28 \times 28 \times 32 \times 5 \times 5 \times 192$

$= 120\text{M}$

Significantly reduced

Using 1×1 convolution

$28 \times 28 \times 192 \xrightarrow[1\times1,\ 16,\ 1\times1\times192]{\text{CONV}} 28 \times 28 \times 16 \xrightarrow[5\times5,\ 32,\ 5\times5\times16]{\text{CONV}} 28 \times 28 \times 32$

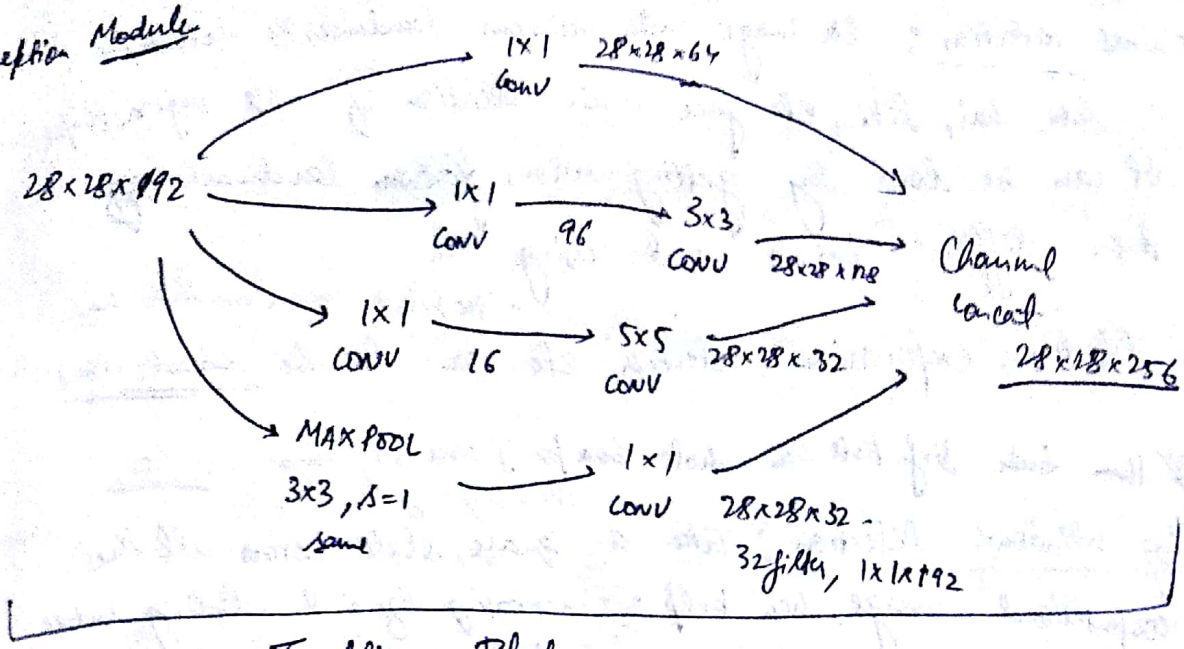
bottleneck layer

$28 \times 28 \times 16 \times 192 \times 1 \times 1 = 2.4\text{M}$

$28 \times 28 \times 32 \times 5 \times 5 \times 16 = 10\text{M}$

Total $\Rightarrow$ 12.4M

Inception Module



Inception Block.

Inception Network: Inception block repeated multiple times.

★ When various Inception blocks are joined, from the channel concat, output is taken for softmax layer and check if it is not overfitting

★ A regularising effect.

★ When making a project, open source se download karlo and pretrained model hai, softmax layer change karlo. (freeze layers) few or all depending on problem statement.

★ Save the parameters to disk.

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

p_c → Is there any object?

b_x, b_y → Image ke centre co-ordinates.

b_h, b_w → height, width of image.

c_1, c_2, c_3 → Probability of 1, 2 & 3.

1, 2, 3 → 1

4 → 0

1, 2, 3 → objects (car, pedestrian, motorcycle, etc).

4 → only background

if $p_c = 0$, then other elements are don't care.

$$L(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \cdots + (\hat{y}_8 - y_8)^2 & \text{if } y_1 = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 0 \end{cases}$$

★ Object ke around Bx box.

Landmark detection: Ek image mein various landmarks detect kr leta hai, like ek face mein location of nose vagera, shape. It can be done by getting various ~~features~~ landmarks ~~eg~~ i.e. different points and using them.

Pose detection, expression detection, etc can also be ~~used~~ done. (→ Snapchat vagera use karte hai)

Hum train sirf karte hai chote box pr jisme sirf ~~...~~ car ho. (Boundary kaafi kam)

Sliding Windows Detection: Take a square, slide across all the image, then keep a increasing size and sliding until the object is detected.

High computational cost

Convolution implementation: To reduce computation of sliding windows.

5x5x16 →(FC) 400 →(FC) 400 → y softmax (4)

instead use → 5x5x16 →(FC, 5x5x16, 400 filters) 1x1x400 →(FC, 1x1x400, 400 filters) 1x1x400 →(1x1) 1x1x4

Convolution ⇒

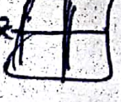
Now, 14x14x3 →(5x5) 10x10x16 →(MAX-POOL 2x2) 5x5x16 →(FC 5x5) 1x1x400 →(FC 1x1) 1x1x400 →(FC 1x1) 1x1x4

so we use this to detect the output of sliding windows. By,

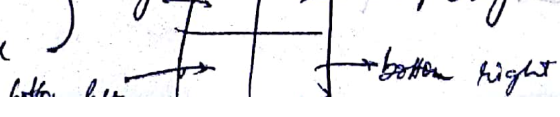
Suppose

16x16x3 →(5x5) 12x12x16 →(MAX-POOL 2x2) 6x6x16 →(FC 5x5) 2x2x400 →(FC 1x1) 2x2x400 →(FC 1x1) 2x2x4

we get this kind of box ⟹ 

We used a 14x14x3 box to run over this image and window was slided by 2 pixels each time. So what we ~~get~~ would get is  4 values for 4 portions.

Those four values are



Bounding Box Prediction

YOLO: 3×3 ya 19×19 type ka main image ko divide karo, then the object belongs to that box where the center of the image lies.

↳ very fast so real time ke liye bhi sahi hai.

now for y ⇒ b_h, b_w should signify the total size of the box.

Intersection over Union (IOU) = $\frac{\text{size of intersection}}{\text{size of union}}$ — of predicted and what actually is.

boxes.

We consider prediction is correct if IOU ≥ 0.5.

Non-max suppression: Possibility when image is divided into larger no. of boxes, more boxes may see that center is there.

Thus multiple detection.

↳ see the one with highest prob. & mateling and the boxes with high IOU with this box get suppressed. Next we find next higher prob. in the remaining & follow.

P_c → prob. Before the above step, we also discard the boxes with $p_c \leq 0.6$

Anchor boxes: To detect multiple type of objects / objects located in same frame

⇒ Predefine two shapes / two anchor boxes (horizontal, vertical)

adding put 16 variables. (8 for anchor box 1, 8 for anchor box 2).

What we do here is calculate which anchor box has higher IOU for that part and we use that anchor box for that part.

Algo not good for

① 2 anchor box, 3 objects

② 2 objects, same type of anchor box.

③ Have have midpoint for objects.

YOLO ALGO:

image divided finally in 3x3.

1- [illegible]
2- [illegible]
3- [illegible]

y → 2 anchor boxes

y is 3x3x2 x 8

9 3x3 grid, # anchors, 5 + ~~# anchors~~ # classes

* * More anchor boxes can also be used acc to our need.

Region Proposals (R-CNN): Rather than running CNN on all windows you just select few windows and run on those.

Segmentation algo is used for selecting the region for running → It finds few areas and runs the informant there.

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

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

Faster R-CNN: Use convolution network to propose regions.

YOLO → YOU ONLY LOOK ONCE

Face recognition

We have only 1 image, not sufficient to train a neural network.

We use similarity function. $\Rightarrow d(img1, img2)$ = degree of difference between images.

if $d(img1, img2) \leq \tau$ "same" } verification
$> \tau$ "different" }

We we have to find d.

This can be used when a new face joins and network have to learn acc. to it.

Siamese Network: Feed the two images into the system of neural network and calculate the matrix of those two images. We then calculate

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

(f is obtained acc. to the network.)

$f(x^{(i)})$ if $x^{(i)}, x^{(j)}$ are the same person, $\| f(x^{(i)}) - f(x^{(j)}) \|^2$ is small & vice-versa

Triplet loss

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

$$\text{so} \quad \underbrace{\| f(A) - f(P) \|^2}_{d(A,P)} + \alpha \leq \underbrace{\| f(A) - f(N) \|^2}_{d(A,N)}$$

($\alpha \geq 0.2$)

margin parameter $\Rightarrow$ needed bcz it is possible the network forms such that $f(A) - f(B)$ always or $f(A) = 0$ always.

To prevent that α is used.

Loss fn

$$L(A,P,N) = \max\left(\underbrace{\| f(A) - f(P) \|^2 - \| f(A) - f(N) \|^2}_{<0 \text{ so problem and loss there, calculate.}} + \alpha, 0 \right)$$

Training set: 10K pictures of 1K people is to be used.

Training: if A, P, N are chosen randomly, the eqn is easily satisfied.

Thus we have to choose A, P, N which are hard to train.

Choose such that $d(A,P) \simeq d(A,N)$ and thus hard to predict

Neural style transfer : Image C (content) to be painted in style (S) to obtain G (generated image)

Deeper layer calculate more complex part of the image

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

cost?

Process
1. Initiate G randomly
2. Use gradient descent to minimise $J(G)$ $G := G - \frac{\partial}{\partial G} J(G)$

Content Cost function ($J_{content}(C,G)$)

- A hidden layer l is used to compute content cost. (l should neither be too shallow nor deep layer)
- Use pre-trained ConvNet
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on images
- If $a^{[l](C)}$ and $a^{[l][G]}$ are similar, both images have similar content

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

Style Cost function ($J_{style}(S,G)$)

Style matrix — Layer l

def $a^{[l]}_{i,j,k}$ = activation at (i,j,k). (i: height, j: width, k: layer). $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$ $G^{[l]}_{kk'}$ $k = 1, 2, \ldots, n_c^{[l]}$

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_h^{[l]}} \sum_{j=1}^{n_w^{[l]}} a^{[l](S)}_{ijk} \, a^{[l](S)}_{ijk'} \quad \{\text{image S ke liye}\}$$

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_h^{[l]}} \sum_{j=1}^{n_w^{[l]}} a^{[l](G)}_{ijk} \, a^{[l](G)}_{ijk'} \quad \{\text{image G ke liye}\}$$

Overall $J_{style}(S,G)$ will be sum. i.e. $\sum_l \lambda^{[l]} J^{[l]}_{style}(S,G)$

$$J^{[l]}_{style}(S,G) = \| G^{[l](S)} - G^{[l](G)} \|_F^2$$

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

Sequence Models

⇒ Recurrent Neural Networks

T_x - length of input
T_y - length of output.

$X^{(i)<t>}$ — t training element : like multiple words of a sentence. (Toh Kaunsa word)
$x^{<1>}$ $x^{<2>}$ ...
i training example.
$T_x^{(i)}$ → input length of i^{th} training example.
$T_y^{(i)}$ — output length of i^{th} traning example.

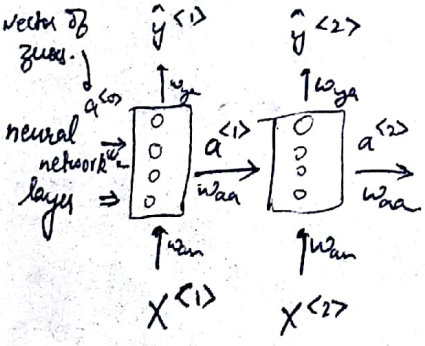
Use a dictionary {Vocabulary} → take fixed amount of words, index those words.

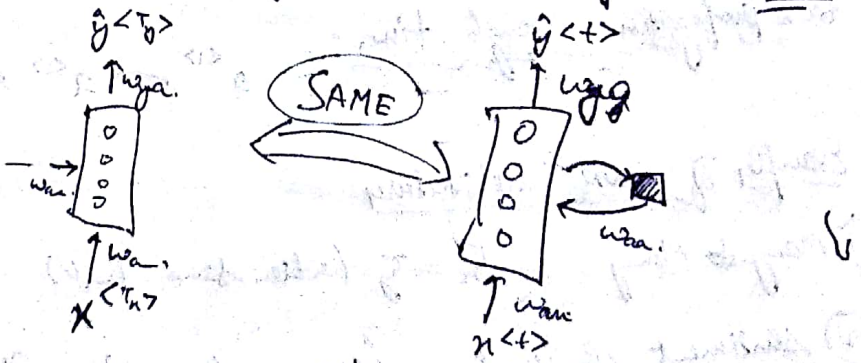
Now for each word, make a vector of 0,1 of the size of dictionary.
(where the word matches the dict, rest all 0. (Thus only 1 one in each vector)

Like for 'a' ⇒ $\begin{bmatrix}1\\0\\\vdots\\0\end{bmatrix}$ the dict ⇒ $\begin{bmatrix}a\\aaron\\\vdots\\zulu\end{bmatrix}$

* If the word is not in dict., make a fake token like <UNK>

⇒ Not the standard network as different layers does not share features learned





BRNN (Bidirectional RNN) : To use all the words in the sentence.

$$a^{<1>} = g_1(w_{aa}\, a^{<0>} + w_{ax}\, x^{<1>} + b_a) \quad \leftarrow \text{tanh/ReLU}$$

$$\hat{y}^{<1>} = g_2(w_{ya}\, a^{<1>} + b_y) \quad \leftarrow \text{sigmoid.}$$

w_{ax} → uses x type entity to compute a type entity

$$a^{<t>} = g_1(w_{aa}\, a^{<t-1>} + w_{ax}\, x^{<t-1>} + b_a) \; ; \; \hat{y}^{<t>} = g_2(w_{ya}\, a^{<t-1>} + b_y)$$

Simplified notation

$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a) \; ; \; \hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$$

$W_a = [W_{aa} \mid W_{ax}]$

Now if a is (100) dimensional.
so $W_{aa} \to (100, 100)$

Thus $W_a \Rightarrow (100, 10100)$

and if x is (10,000) dimensional
so $W_{ax} \to (100, 10,000)$

also $[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$ ↕ 100 / ↕ 10,000 → 10100

Back propagation.

Here we will be using loss fn: $\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log \hat{y}^{<t>} - (1-y^{<t>}) \log (1-\hat{y}^{<t>})$

Now total loss, $\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$

Back propagation through time:

$a^{<1>} \leftarrow a^{<2>} \leftarrow \cdots \leftarrow a^{<T_x>}$

Examples of RNN architectures

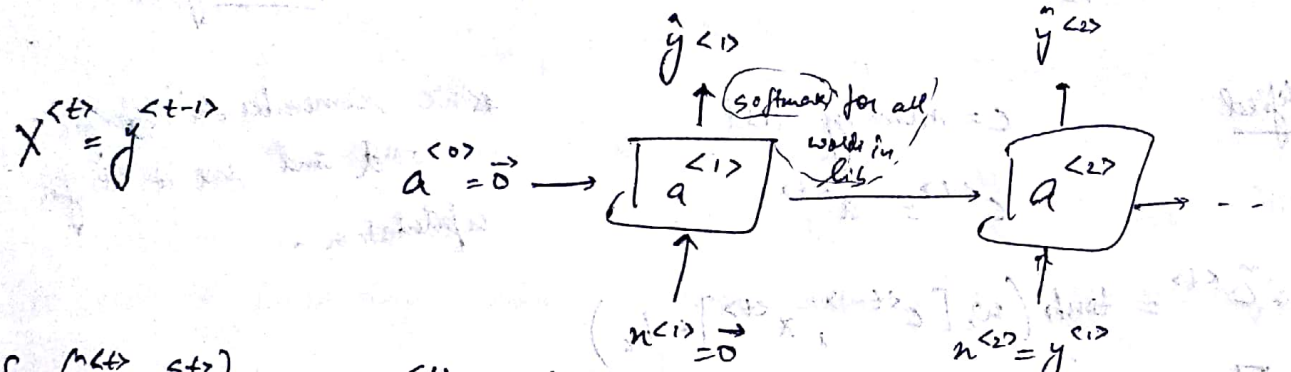
1. Many-to-Many $T_x = T_y$ (when same length)
2. Sentiment classification - (many-to one) x = text, y = 0/1
3. One-to-One → basic vala
4. One-to-many (eg. music generation)
5. ~~Word~~ Many-to-many - different lengths

Language Modelling : Determines the probability of being that sentence & all the similar sounding sentences.

~~Train~~ **Training set** : large corpus of english text.

~~Tokenise~~

* Tokenise the words, end the sentence with <EOS> token. ← end of sentence.
* <UNK> token for words not in library.



$$L(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

$$L = \sum_i L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

Thus it is like,

$$P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>}) \cdot P(y^{<2>}|y^{<1>}) \cdot P(y^{<3>}|y^{<1>}, y^{<2>})$$

↑ Prob. of this three word sentence.

Sampling

Character-level language model : Every character is in vocab. and the sentence is divided on the basis of every character and not word.

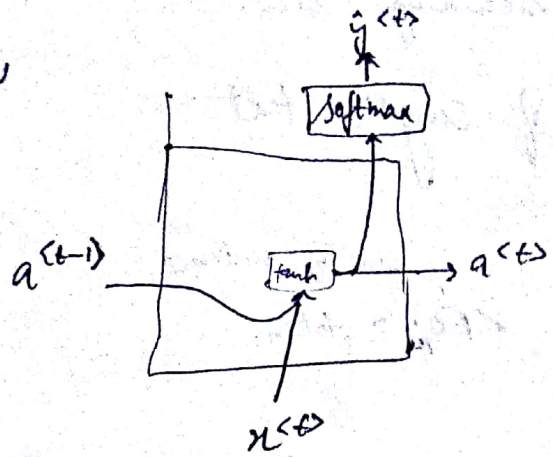
* No prob. of UNK.
* Very computationally expensive.

Vanishing gradients : Kabhi kabhi, because of very initial words, the later word in the sentence is effected.

Exploding gradients : Apply gradient clipping.

Gated Recurrent Unit:

RNN



$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$

(g: tanh / ReLU)

The cat, ... , was full

The cats, . . . , were full.

Simplified

c = memory cell

$c^{<t>} = a^{<t>}$

* we remember the specific word and use it later for updatation.

$$\tilde{c}^{<t>} = \tanh(w_c[c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(w_u[c^{<t-1>}, x^{<t>}] + b_u)$$

(Γ_u: update; σ: sigmoid)

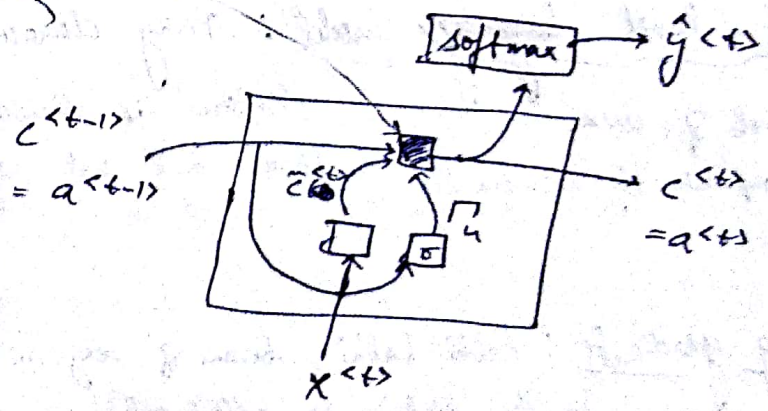
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

$\Gamma_u = 1$ $\Gamma_u = 0$ $\Gamma_u = 0$ . . . . 1. ← To check whether to update Γ_u or not.

$c^{<t>} = 1$

The cat, which already ate . . . . , was full

This simplified version of GRU unit is



Now full GRU

$$\tilde{c}^{<t>} = \tanh(w_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

Another model ↓ LSTM

LSTM → Long Short Term Memory.

$$\tilde{c}^{<t>} = \tanh\left(w_c[a^{<t-1>}, x^{<t>}] + b_c\right)$$

$$\Gamma_u = \sigma\left(w_u[a^{<t-1>}, x^{<t>}] + b_u\right) \quad \text{(update)}$$

$$\Gamma_f = \sigma\left(w_f[a^{<t-1>}, x^{<t>}] + b_f\right) \quad \text{(forget)}$$

$$\Gamma_o = \sigma\left(w_o[a^{<t-1>}, x^{<t>}] + b_o\right) \quad \text{(output)}$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$

$a^{<t>} \neq c^{<t>}$

peephole connection

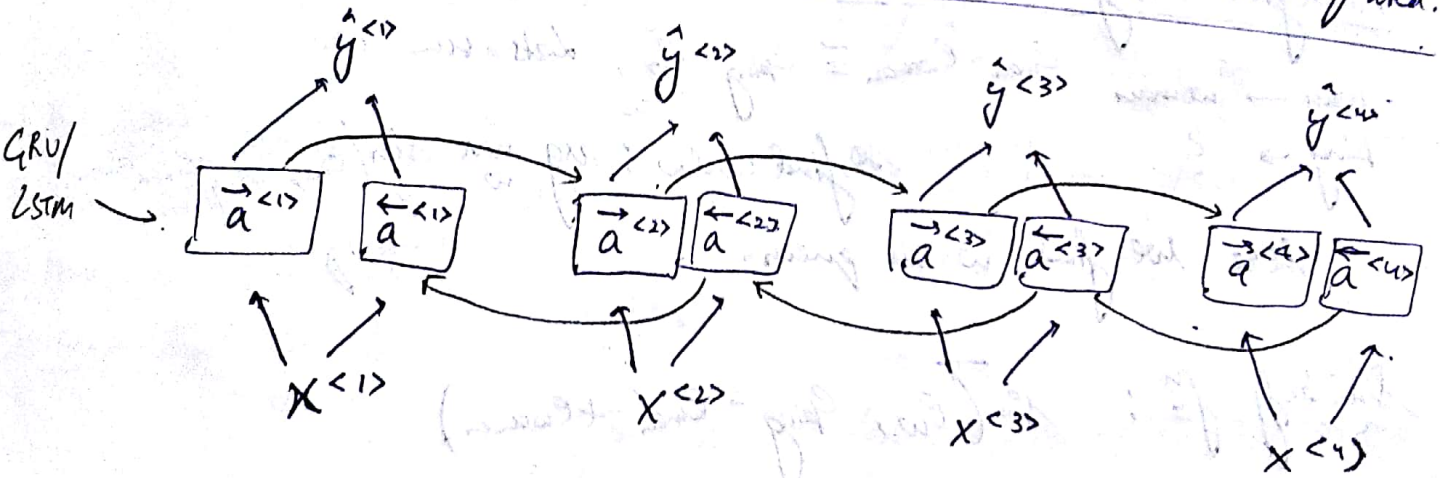
⇓

Sometime people put $c^{<t-1>}$ also with $a^{<t-1>}$ and $x^{<t>}$, by it also effects.

GRU easier to build bigger models whereas LSTM is more flexible. People prefer to use LSTM but GRU is simpler.

Bi-directional RNN : (BRNN)

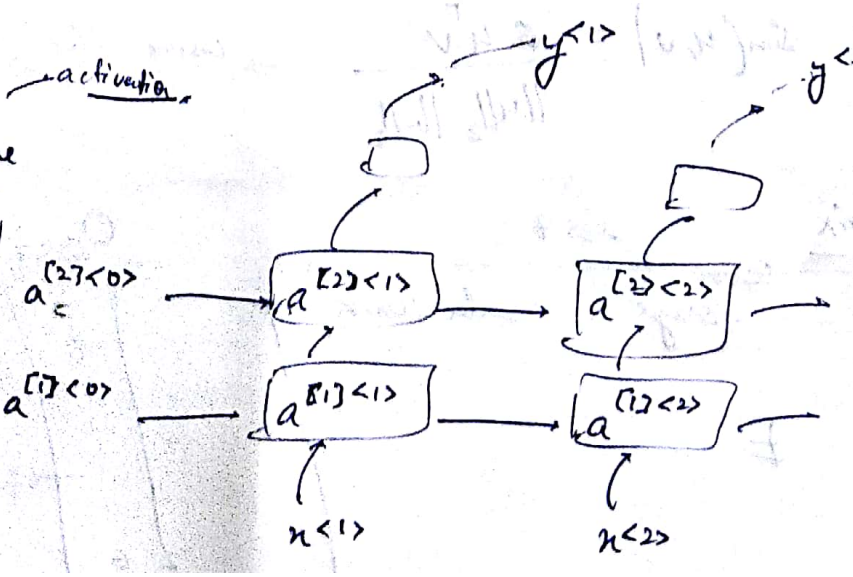
⇒ BRNN with LSTM is also commonly used.



$a^{[l]<t>}$ — layer, time — activation

time – no. of words wala.

layer – kai words ko predict karne ki layer



1-hot representation: But problem here is that words that might be linked to each other are also placed very far apart.

$= O_{5791}$ Featurized representation : word embedding

Words are given different types of features. Like Gender, Royal, Age, Food, ... Suppose there are 300 features so table consists of those features.

300

5791 position

man → e_{5791} position = 5791

★ Transfer learning & word embeddings

① Learn word embeddings from large text corpus (1-100B words) (or download pre-trained embedding online)

② Transfer embedding to new task with smaller training set (say, 100K words)

③ Optional: Continue to finetune the word embeddings with new data.

t-SNE algo. 300D → embedding → 2D

Analogies using word vectors

$e_{man} - e_{woman} \approx e_{king} - e_?$, Lets assume w.

man → woman

king → ?

e_w

so, find word w : $\arg\max_w \text{sim}(e_w, e_{king} - e_{man} + e_{woman})$

similarity

here we find w as queen.

Similarity fn : $\text{sim}(e_w, e_{king} - e_{man} + e_{woman})$

Cosine similarity: $\text{sim}(u, v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$ ⇒ cosine b/w u, v.

Maximise of.

Embedding matrix

10,000 words

a aaron ... orange ... zulu <UNK>

6257

O_{6257}

10K

300

E

E. O_{6257}

(10K, 1)

(300, 10K)

this is embedding

e e e e e

w

Context O_c →

Now this

P mea

Scanned by CamScanner

E. O_{6257} → $e_{(300,1)}$ → e_{orange}

$(6257, 1)$

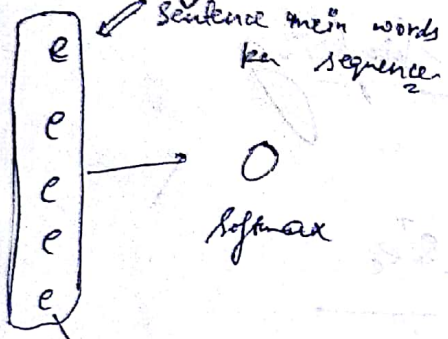
$(300, 10k)$ $(10k, 1)$

(E). (one hot vector)

$E \cdot O_j = e_j$

= embedding for word j

This is kinda inefficient, so use specialised fn to look up an embedding.



Sentence mein words ka sequence

Some we put them according to various contexts.

There can be various contexts. like.

- last 4 words
- 4 words on left & right
- last 1 word
- Nearby 1 word.

θ_t = parameter associated with output t.

"orange" "juice"

Context c → Target t.

$O_c \rightarrow E \rightarrow e_c \rightarrow O \rightarrow \hat{y}$

$e_c = E O_c$ softmax

Softmax: $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$

$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{10,000} y_i \log \hat{y}_i$

$y = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ ← 4834 type ka.

Now this softmax is slow bcz computing

P mean calculating all. For that purpose,

Tree can be used:

① Hierarchical softmax —

& so on.

common words on the top whereas less common words deep inside the tree.

② Negative Sampling — . .

context 'c' must not be random, like (of, key, a, ..) when used, e_c is to be updated again and again, calculated ~~[illegible]~~ it takes large time.

New learning problem

$k = 5-20$ smaller dataset

$k = 2-5$ larger dataset

context	word	target?
orange	juice	1
orange	king	0
orange	book	0
orange	of	0
orange	the	0
q	q	q
c	t	y

$k = 4$ words taken randomly

k: 1 negative to positive ratio examples.

$$P(y=1 \mid c,t) = \sigma(\theta_t^T e_c)$$

Negative Sampling

$$P(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$$

Instead of calculating for 10,000 words we randomly take words and calculate for just $(1+k)$ words. [$(k+1)$ binary classification problems.]

This is 10,000 binary classification problem.

Selecting negative examples: $P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10,000} f(w_j)^{3/4}}$ f → frequency of that word.

GloVe (global vectors for word representation)

X_{ij} = # times i appears in context of j (i = t, j = c)

$X_{ij} = X_{ji}$ → if in range of 10 words

$X_{ij} \neq X_{ji}$ if context like : before, ...

θ_i, e_j are symmetric

$$\text{minimize} \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij}) \left(\theta_i^T e_j + b_i + b_j' - \log X_{ij}\right)^2$$

weighting term

$f(X_{ij}) = 0$ if $X_{ij} = 0$ ($0 \log 0 = 0$)

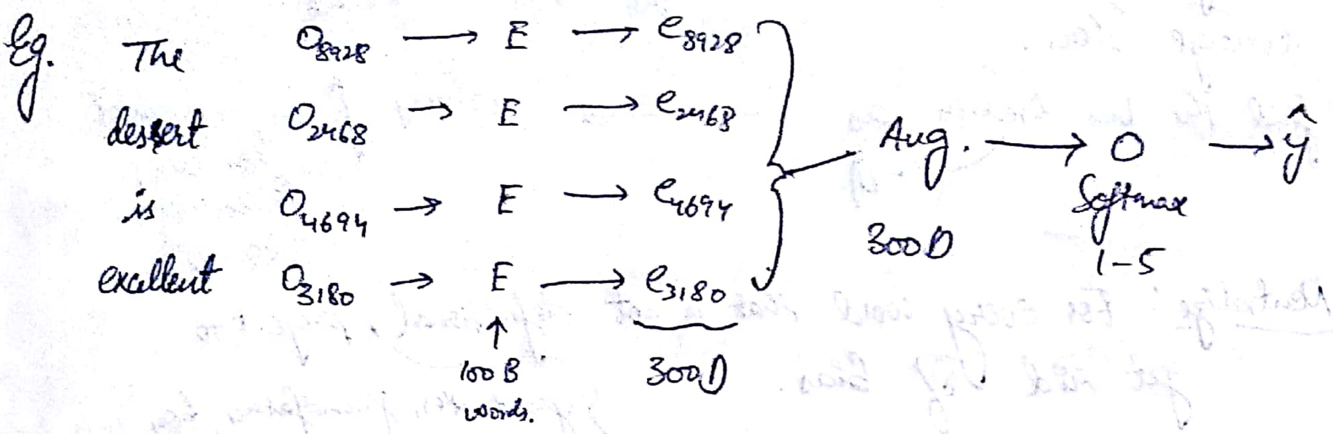
$e_w^{(final)} = \frac{e_w + \theta_w}{2}$

$$\text{minimize} \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij}) \left(\theta_i^T e_j + b_i - b_j' - \log X_{ij} \right)^2$$

Sentiment Classification

(review in words → stars mein convert krna). Data type.

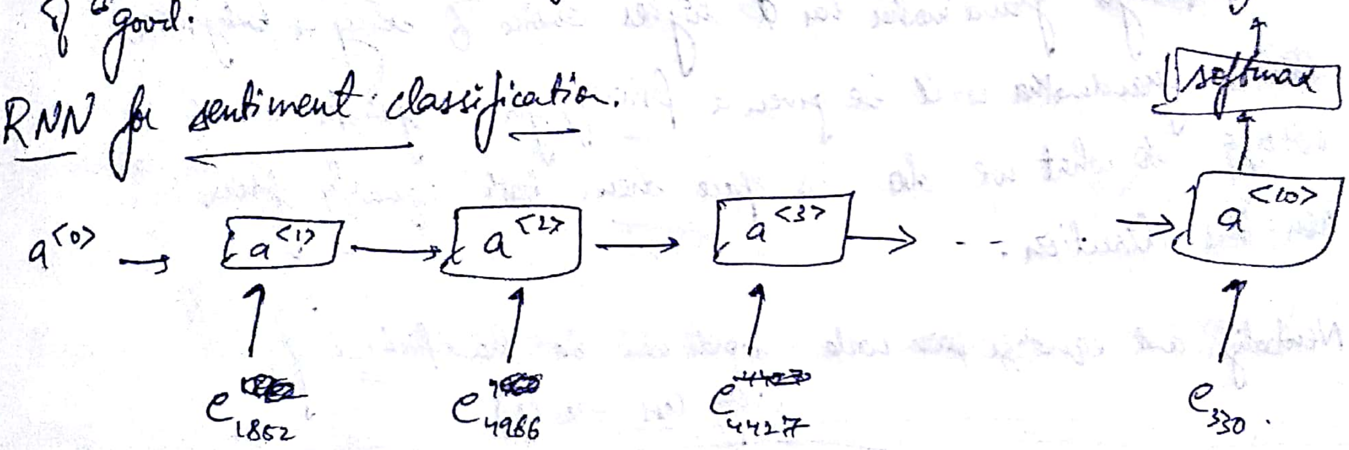
Eg.



Now, "completely lacking in good taste, good service, and good ambience."

→ Here just taking an average will lead in wrong results, bcz too many of "good".

RNN for sentiment classification.



many-to-one RNN.

Debiasing Word Embeddings

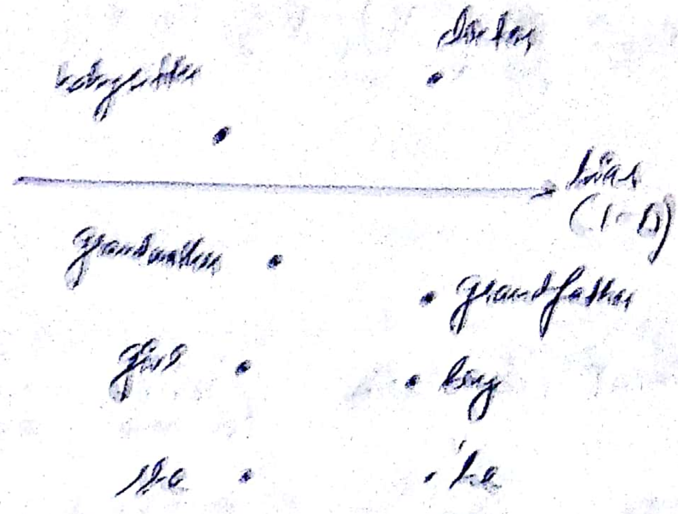
Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model. which is wrong like

Father : Doctor as Mother : Nurse. X.

So we have to debias.

Scanned by CamScanner

Debiasing Bias



① Identify bias direction

average {
$e_{he} - e_{she}$
$e_{male} - e_{female}$
few pairs of these and average them.

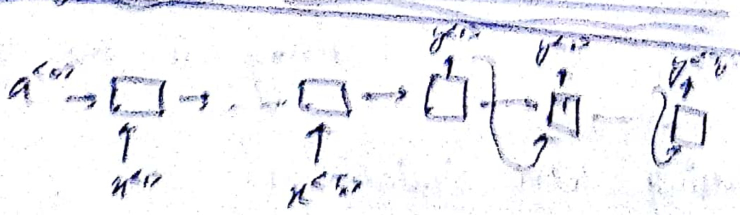
we find the bias direction as ⟶ (1-D), whereas ↑ is unrelated (non-bias direction) (299-D)

② Neutralize: For every word that is not definitional, project to get rid of bias.
{grandmother, grandfather, boy, girl, etc.}

③ Equalize pairs: Eg. grandmother and grandfather with babysitter. although grandmother has a higher chance of being a babysitter, ~~so~~ grandmother will be given a priority for babysitter which is wrong. So what we do is space them both equally from non-bias direction.

[Neutralizing and equalizing pair words words can be handpicked. (very less words)]

Sequence to Sequence Model

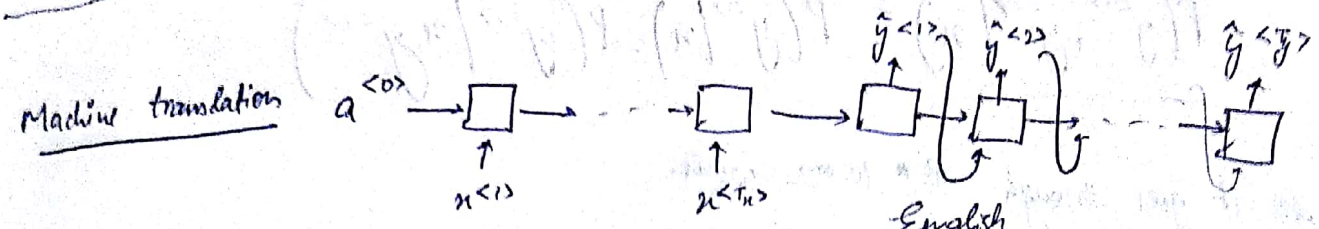


① Translation (French to English)

② Image Captioning

Machine translation as building a conditional language model

Machine translation



"conditional language model"

$$P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$$

English → French.

Finding the most likely translation

$$\arg\max_{y^{<1>}, \ldots, y^{<T_y>}} P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$$

+ Randomly nhi lene hai words.

Greedy Search

Best $\hat{y}^{<1>}$ word choose krlo, then best $\hat{y}^{<2>}$ word choose krlo, and so on. But not a good approach.

$\Rightarrow P(\hat{y}^{<1>} \mid x)$ aa similarly to obtain $P(\hat{y}^{<1>}, \hat{y}^{<2>}, \hat{y}^{<3>}, \ldots, \hat{y}^{<T_y>} \mid x)$

but ho skta has that we choose initial words that are better fitting but sentence that forms is not good.

Approximate Search

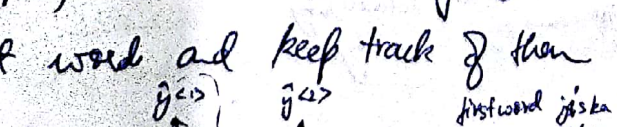
$$\arg\max_{y} P(\hat{y}^{<1>}, \hat{y}^{<2>}, \ldots, \hat{y}^{<T_y>} \mid x).$$

We can't choose the words randomly, so we use various types of search algorithms for that purpose.

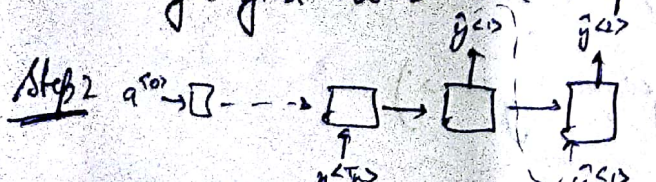
Beam Search

B = beam width ⇒ no. of possibilities considered by beam search at a time.

Step 1 $P(y^{<1>} \mid x)$ is used to search for (B) most likely words that could fit for first word and keep track of them



Step 2



first word jiska track rakha jaa rha hai

$P(y^{<1>} \mid x, \text{"in"})$. Then calculate $P(y^{<1>}, y^{<2>} \mid x)$

Calculate all B possibilities.

$$P(y^{<1>}, y^{<2>} | x) = P(y^{<1>}|x) \cdot P(y^{<2>}|x, y^{<1>})$$

so it goes through $B * 10,000$ words.

After this, we select B such combinations of 1st and 2nd words combined.

Also, only B copies of network are to be used.

Step 3: 3rd words $P(y^{<3>}|x, y^{<2>}, y^{<1>})$

and it keeps going.

To be terminated by <EOS>

finally delhi is highest $P(y^{<1>}, y^{<2>} \dots y^{<T_y>}|x)$

$$= P(y^{<1>}|x) \cdot P(y^{<2>}|x, y^{<1>}) \cdot \dots \cdot P(y^{<T_y>}|x, y^{<1>}, y^{<2>}, \dots, y^{<T_y-1>})$$

usually this product is very less.

so instead this is used $\Rightarrow \sum_{y=1}^{T_y} \log(y^{<t>}|x, y^{<1>}, y^{<2>}, \dots, y^{<t-1>})$

Thus $\arg\max_y \sum_{y=1}^{T_y} \log(y^{<t>}|x, y^{<1>}, y^{<2>}, \dots, y^{<t-1>})$

Then, we normalize to some extent, we divide by T_y^{α} $\alpha \leftarrow$ hyperparameter

$\alpha < 1$ (not normalized), $\alpha = 1$ normalized fully

finally, $\arg\max_y \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \dots y^{<t-1>})$

Beam W

1 → 3 → 10 huge gain

Infinite BFS (

Errors

Case 1

Case

Scanned by CamScanner

Beam width selection:

Large B: better result, slower
Small B: worse result, faster.

1 → 100 (huge gain) | 1000 → 5000 (very less gain)

Like BFS (Breadth First search) or DFS (Depth first search), Beam search runs fast but is not guaranteed to find exact max. for $\arg\max_y P(y|x)$.

Error Analysis on beam search.

(increasing B never hurts.)

y^* translation by human. ⇒ very good. $P(y^*|x) > P(\hat{y}|x)$

$\hat{y}$ by algo ⇒ Not that good — to be compared by RNN.

Case 1: RNN predicts $P(y^*|x) > P(\hat{y}|x)$

Beam search chooses $\hat{y}$. But y^* attains higher $P(y|x)$.

→ Thus, Beam search is at fault.

Case 2: $P(y^*|x) \leq P(\hat{y}|x)$

y^* is a better translation than $\hat{y}$. But RNN predicted $P(y^*|x) < P(\hat{y}|x)$.

→ Thus, RNN model is at fault.

For error analysis, for example we go through dev and check which is majorly at fault, Beam or RNN and then optimize that.

Bleu Score (bilingual evaluation understudy)

Human give references as a part of test set and Machine Text Output is seen.

Modified precision: $\frac{\text{Max count of word in any reference}}{\text{Count of word in output}}$ → single word

Count_{clip}

(for pair of words: bigrams) for each pair, calculate Count and Count_{clip} and sum those, then calculate modified precision

$$P_1 = \frac{\sum_{\text{unigrams} \in \hat{y}} \text{Count}_{clip}(\text{unigram})}{\sum_{\text{unigram} \in \hat{y}} \text{Count}(\text{unigram})}$$

P_1 → unigram

$P_1 = P_2 = \dots = P_n = 1.0$ → so equal to one if the reference is machine output

$$P_n = \frac{\sum_{\text{n-gram} \in \hat{y}} \text{Count}_{clip}(\text{n-gram})}{\sum_{\text{n-gram} \in \hat{y}} \text{Count}(\text{n-gram})}$$

P_n → n-gram

Combined Bleu score : $BP \cdot \exp\left(\frac{1}{n}\sum_{k=1}^{n} p_k\right)$

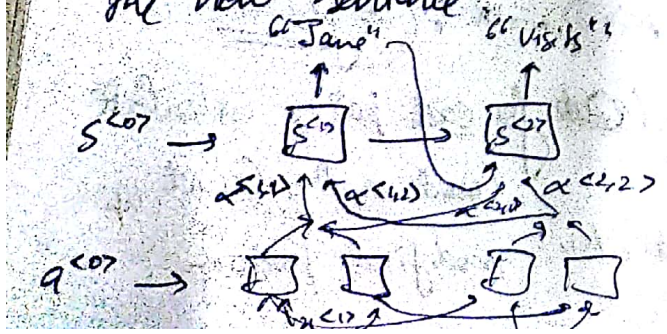
BP = Brevity penalty

$$BP = \begin{cases} 1 & \text{if MT\_output\_length} > \text{reference\_output\_length} \\ \exp(1 - \text{MT\_output\_length}/\text{reference\_output\_length}) & \text{otherwise} \end{cases}$$

Attention Model intuition

Problem of long sequences : Model have to memorise a long sentence, so its blue score drops as it is not that efficient.

Every word is given a certain amount of attention for forming the new sentence

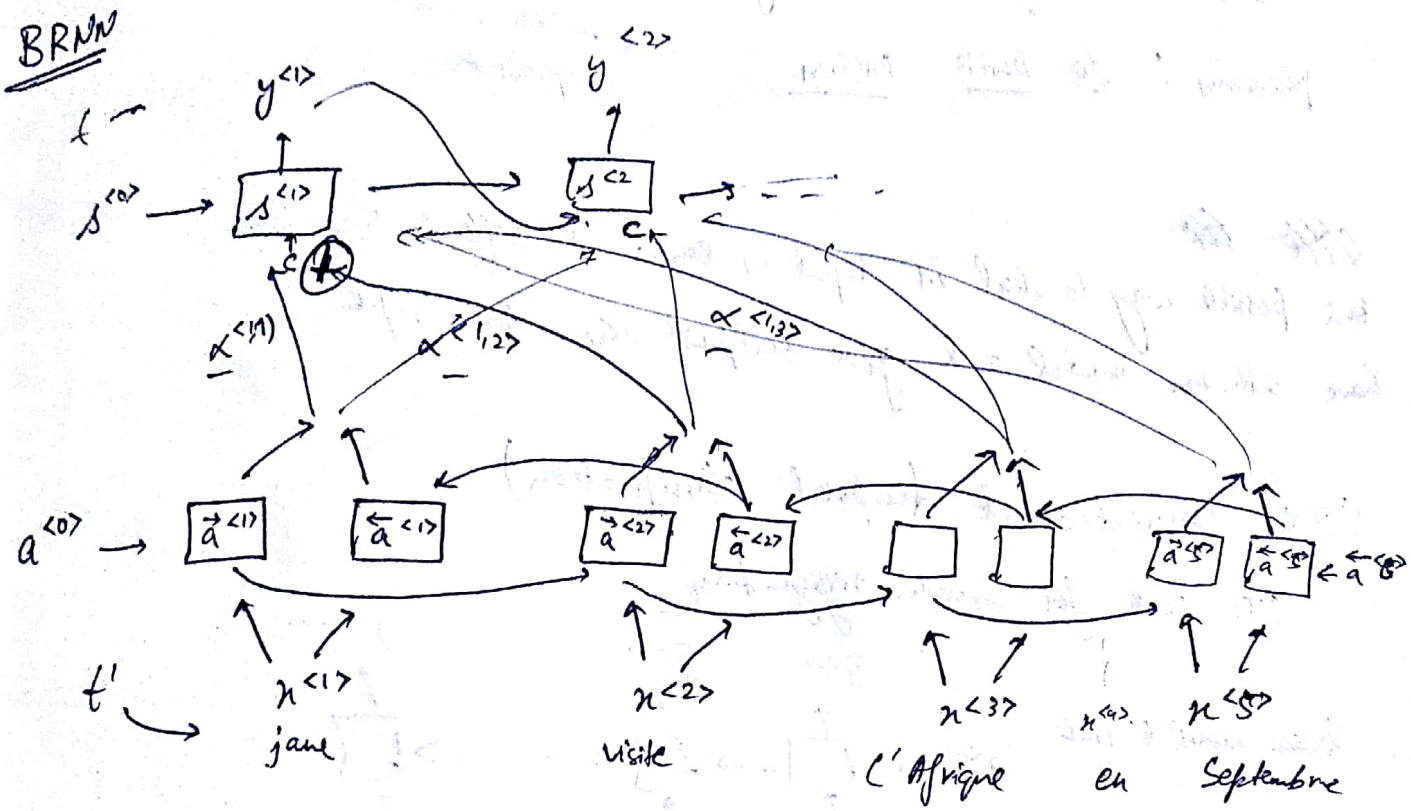


$\alpha^{<1,1>}, \alpha^{<1,2>}, \alpha^{<2,1>}, \alpha^{<2,2>}$ — are the attention given to different words

$\alpha^{<t,t'>}$ are the attention weights

of word bnane ke lie t' ki traf se kitni attention.

BRNN



$a^{<t'>} = (\vec{a}^{<t'>}, \overleftarrow{a}^{<t'>})$

weighted sum of attention weight $\Rightarrow \sum_{t'} \alpha^{<1,t'>} = 1$

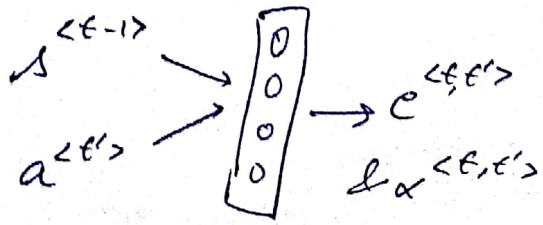
$c^{<1>} = \sum_{t'} \alpha^{<1,t'>} a^{<t'>}$

$\alpha^{<t,t'>}$ = amount of attention $y^{<t>}$ should pay to $a^{<t'>}$

$c^{<2>} = \sum_{t'} \alpha^{<2,t'>} a^{<t'>}$

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$

$\Rightarrow T_x * T_y$ is the amount of attention, so high cost.



Small NN to obtain $e^{<t,t'>}$

phonemes: smallest units of cell.

like "the quick brown fox"

phonemes: de kwik braun ← hand-written representation

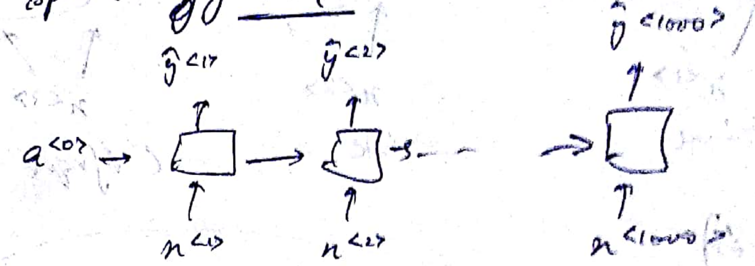
~~CTC cost~~

One possible way to deal is input is audio at different time frame, have attention model and give output the transcript.

CTC (Connectionist temporal classification)

CTC cost for speech recognition

Basic model ki train



possibility that dono ka count along ho.
(Output is way too less)

so output is ttt _ _ h _ _ eee _ _ _ ␣ _ _ _ qqq _

Basic Rule: Collapse repeated characters not separated by "blank".

⇒ the ␣ q

the quick brown fox → 19 characters.

so 19 ko expand karke 1000.

Trigger word detection

Turn on something by saying a work like "Alexa play despacito" 😊

RNN output to be set to 1 just after the trigger word was said, else 0.