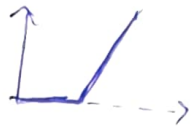
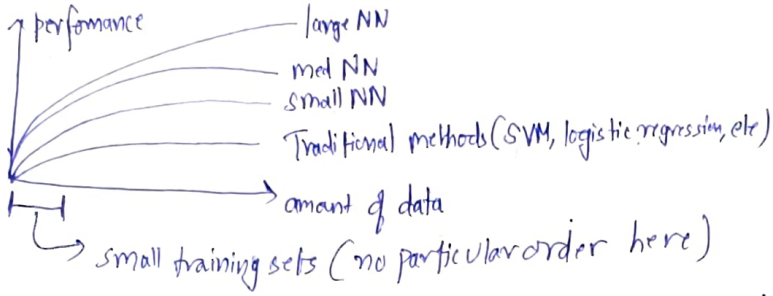
* ReLU :→ Rectified Linear Unit 

↳ Why deep learning taking off → Scale drives deep learning progress



→ Logistic Regression as a Neural Network (week-2 / DLS-1)

↳ Binary classification (Notations)

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

n_x :→ # of features

(eg- for an image ($\in 4 \times \in 4$pi)

↳ $n_x = 64 \times 64 \times 3$

↳ m training examples :→

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

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

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

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

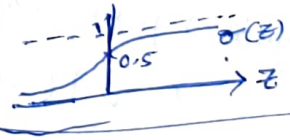
↳ Logistic Regression

↳ given x (feature vector) want $\hat{y} = P(y=1|x)$ (Conditional probability)

↳ Parameters to be used :→ $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

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

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



↳ Logistic regression cost function

* here we don't take squared error as loss function because it turns out to give a non convex optimization problem (multiple local minimas)

↳ Loss function $\boxed{\mathcal{L}(\hat{y}, y) = -[y \log(\hat{y}) + (1-y)\log(1-\hat{y})]}$ → Single training example

↳ Cost function $\boxed{J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}$ → Cost of the parameters

↳ need to minimize it to get best parameters.

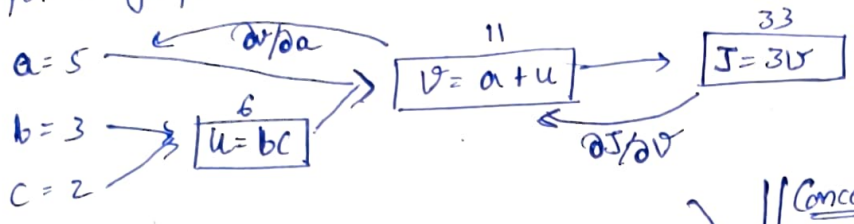
→ Gradient descent

↳ want to find w, b that minimizes $J(w,b)$ $\Big\{ w := w - \alpha \frac{dJ(w)}{dw}$ → represented as "dw"

learning rate

↳ Computing Derivatives using Computation graph

a computation graph is like this :→ (to compute $J = 3(a+bc)$)



→ forward propagation

← backward propagation (to calculate derivatives)

↳ Concept-1 → $\frac{\partial J}{\partial v} \approx \frac{\Delta J}{\Delta v}$ ≈ (how much increase in J due to small increase in v)

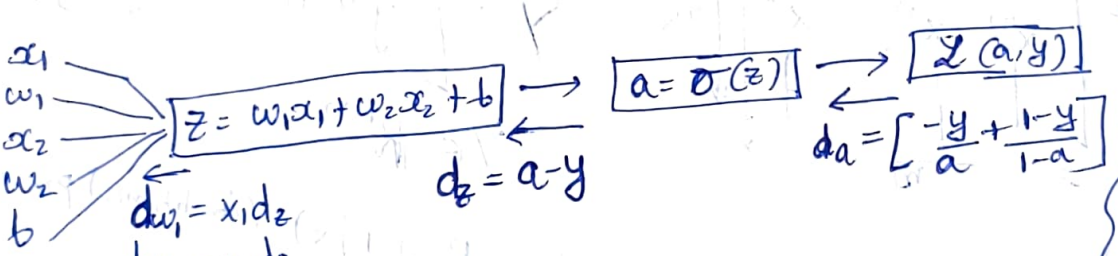
Concept-2 :→ chain rule of derivation

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v} \times \frac{\partial v}{\partial a}$$

↳ Notation :→ repeatedly need to calculate d(J) wrt all the parameters d(var) { J is the final output } → denote :→ $\frac{dJ}{dvar} = dvar$

× Backward propagation for derivatives :→ dvar of the rightmost variables are found first which the helps to find the dvar of left variables

→ Logistic Regression derivatives



↳ Logistic regression on m examples :→ $J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$

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

→ Logistic Regression Implementation

↳ a single step of gradient descent :→

init :→ $J=0$; $dw_1 = 0$; $dw_2 = 0$; $db = 0$

for $i = 1$ to m

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

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

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

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

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

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

$db \mathrel{+}= dz^{(i)}$ ---- [end of for loop]

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

↳ $w_1 := w_1 - \alpha dw_1$; $w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

↳ **Vectorization** :→ Calculations using matrix algebra can be 100x faster in computation than explicit for loops.

↳ **Vectorizing Logistic Regression**

we want to compute :→ $z^{(1)} = w^T x^{(1)} + b \dots z^{(i)} = w^T x^{(i)} + b \dots z^{(m)} = w^T x^{(m)} + b$

$a^{(1)} = \sigma(z^{(1)}) \dots a^{(i)} = \sigma(z^{(i)}) \dots a^{(m)} = \sigma(z^{(m)})$

So, $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$; $Z = [z^{(1)}, z^{(2)}, \dots z^{(m)}]$

$Z = w^T X + [b, b, \dots b] \Rightarrow Z = [w^T x^{(1)} + b, \dots w^T x^{(m)} + b]$

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

↳ **Implementing Logistic Regression by vectorization** :→

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

$A = \sigma(Z)$

$dz = A - Y$

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

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

$w := w - \alpha dw$

$b := b - \alpha db$

} single iteration of gradient descent (it can't be vectorized) ↳ while loop condition based

→ **Python Broadcasting vectors**

$\begin{bmatrix} \quad \end{bmatrix}^{(m,1) \text{ or } (1,m)} + [\#]^{(1,1)} = \begin{bmatrix} \quad \end{bmatrix}^{(m,1)} + \begin{bmatrix} \# \\ \# \\ \# \\ \vdots \end{bmatrix}^{(m,1)}$

$\begin{bmatrix} \quad \end{bmatrix}^{(m,n)} + [\#_1 \ \#_2 \dots]^{(1,n)} = \begin{bmatrix} \quad \end{bmatrix}^{(m,n)} + \begin{bmatrix} \#_1 & \#_2 & \dots \\ \#_1 & \#_2 & \dots \\ \#_1 & \#_2 & \end{bmatrix}^{(m,n)}$

↳ **Why such a cost function of logistic regression** (Maximum likelihood)

$\sigma(z) =$ 

$\hat{y} = \sigma(w^T x + b) = P(y=1|x)$

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

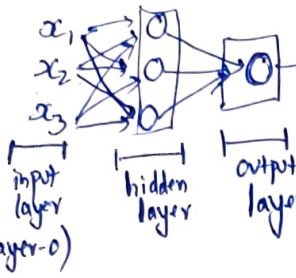
So, $P(y|x) = (\hat{y})^y (1-\hat{y})^{(1-y)}$ → to maximize the probability :→ $\log(P(y|x))$ maximized { $\log(x)$ monotonically increasing

↳ for m-samples :→ IID $\Rightarrow$ $P(y|x) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})$

→ maximize $\log(P(y|x)) = \sum_{i=1}^{m} -\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$ → minimize loss function

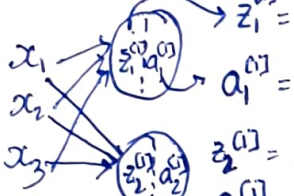
↳ Cost (minimize) $J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

→ Neural Network Representation :→



→ two layer NN (1 hidden layer)
- output from layer 0 :→ $a^{[0]} = x$
- output from layer 1 :→ $a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix}$
- output layer :→ $a^{[2]} = \hat{y}$

Parameter in each layers :→
layer-1 :→ $w^{[1]}$ $(3,3)$ $b^{[1]}$ $(3,1)$ || layer-2 :→ $w^{[2]}$ $(1,3)$; $b^{[2]}$ $(1,1)$



$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$
$a_1^{[1]} = \sigma(z_1^{[1]})$
$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$
$a_2^{[1]} = \sigma(z_2^{[2]})$
and so on

→ Vectorize :→

$$\begin{bmatrix} -w_1^{[1]T}- \\ -w_2^{[1]T}- \\ -w_3^{[1]T}- \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix}$$

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

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

Given input x (1-hidden layer)

$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$

Nt :→ $a^{[0]} = x$

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

for m training examples

Nt :→ super[i] → i^{th} layer ; super(i) → i^{th} example

$x^{(1)} \to a^{[2](1)} = \hat{y}^{(1)}$; $x^{(2)} \to a^{[2](2)} = \hat{y}^{(2)}$; ... $x^{(m)} \to a^{[2](m)} = \hat{y}^{(m)}$

→ take :→ $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$; $Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$

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

Activation Functions :→

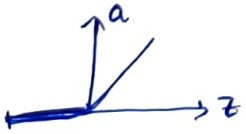
$\sigma(z)$ =  = $\frac{1}{1+e^{-z}}$

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

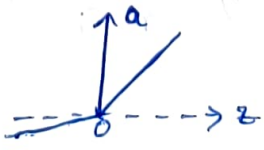
→ almost always $\tanh(z)$ better than $\sigma(z)$
→ as mean of $\tanh(z)$ centered around zero which helps for predictions in the next layers.
→ $\sigma(z)$ only used at output layer when $\hat{y} = \{0, 1\}$

Demerit of $\tanh(z)$
→ ≈ 0 gradient for larger $|z|$ values which may slow down the gradient descent

→ Use ReLU (REctified Linear Unit)
$a = \max(0, z)$



→ also sometimes used :→ Leaky ReLU



$a = \max(0.01z, z)$
→ can be made an hyperparameter

→ Why we need Activation function

↳ if get rid of activation function :- $a^{[1]} = z^{[1]} = W^{[1]}x + b^{[1]}$; $a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

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

So, even if we would have a large number of hidden layers all we are doing here is just calculating a linear function at the end ⟵ no value of having so many parameters in this case.

↳ hence require a non linear activation function

↳ only place to use only linear function :- output layer of a regression problem so that the output $\hat{y} \in (-\infty, \infty)$

→ Derivatives of Activation functions :-

↳ Sigmoid $\left(g(z) = \frac{1}{1+e^{-z}}\right)$:- $\frac{d}{dz}g(z) = g(z)[1 - g(z)] = g'(z)$

↳ tanh $\left(g(z) = \tanh(z)\right)$:- $g'(z) = 1 - (g(z))^2$

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

$g'(z) = \begin{cases} 0 & ; z < 0 \\ 1 & ; z \geq 0 \\ \text{n.d.} & ; z = 0 \end{cases}$

Leaky ReLU : $g(z) = \max(0.01z, z)$

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

→ Gradient Descent for Neural Networks (One-hidden-layer)

Parameters : $W^{[1]}$ $(n^{[1]}, n^{[0]})$, $b^{[1]}$ $(n^{[1]}, 1)$, $W^{[2]}$ $(n^{[2]}, n^{[1]})$, $b^{[2]}$ $(n^{[2]}, 1)$ || $n_x = n^{[0]}$, $n^{[1]}$, $n^{[2]} = 1$

Cost function :- $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}, y)$ || $\hat{y} = a^{[2]}$

NB :- for binary classification loss function is same as logistic one

Gradient descent : Repeat { Compute predicts $(\hat{y}^{(i)}, i = 1, \dots m)$

$dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, ... ;

$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$; $b^{[1]} := b^{[1]} - \alpha db^{[1]}$, ... }

↳ formulas for computing derivatives

↳ Backpropagation :- $dz^{[2]} = A^{[2]} - Y$ || $Y = [y^{(1)}, y^{(2)} \dots y^{(m)}]$

forward

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

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

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

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

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

(as, binary classification $y \in \{0, 1\}$)

$dW^{[2]} = \frac{1}{m} dz^{[2]} (A^{[1]})^T$

$db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, axis = 1, keepdims = True)$ ← $(n^{[2]}, 1)$ and not $(n^{[2]},)$

$dz^{[1]} = \underbrace{W^{[2]T} dz^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]'}(z^{[1]})}_{(n^{[1]}, m)}$ → element wise product

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

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

↳ Initialization of weights

↳ if all the weights are initialized to zero ⇒ all the units of the hidden layer became identical due to symmetrical situation
⇒ they compute the exact same function

↳ $x_1, x_2 \to a_1^{[1]}, a_2^{[1]} \to a^{[2]} \to \hat{y}$ {for, eg}

$W_1^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$; $b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ // $W^{[2]} = [0 \ 0] \Rightarrow a_1^{[1]} = a_2^{[1]} \Rightarrow dz_1^{[1]} = dz_2^{[1]}$

$\Rightarrow dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$; $W^{[1]} := W^{[1]} - \alpha dw$

↳ Hence initialize the parameters Randomly {ok to initialize b to zero}

↳ $W^{[1]} = np.random.randn((2,2)) * 0.01$ → take a small multiplier because don't want to end-up near the edges of a $\sigma(z)$ or $\tanh(z)$ function for faster convergence.

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

and so, on—

$W^{[2]} = np.random.randn((1,2)) * 0.01$ // $b^{[2]} = 0$

↳ Deep Neural Networks ↘ having more than one hidden layer.

$n^{[l]}$ = # of units in layer l // l = # of layers in neural net

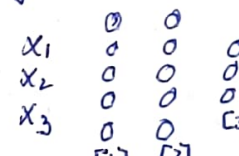
* Forward propagation in a deep network :→

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

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

and so on..

$z^{[4]} = W^{[4]}a^{[3]} + b^{[4]}$; $a^{[4]} = \hat{y} = g^{[4]}(z^{[4]})$



needs to be implemented using for loop only.

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

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

↳ for m-training examples :→

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

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

$Z^{[l]} = [z^{[1]}\ z^{[2]}\ \dots\ z^{[m]}]$

$A^{[l]} = [a^{[1]}\ a^{[2]}\ \dots\ a^{[m]}]$

$A^{[0]} = X = [x^{[1]}\ x^{[2]}\ \dots\ x^{[m]}]$

→ Intuition about Deep Neural Networks

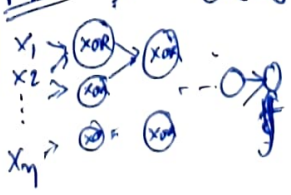
↳ why we need the neural networks to be deep ?

↳ because, the initial layers try to learn the simpler features in the dataset and then the deeper layers tries to compose these simpler features to learn complex features in the dataset.

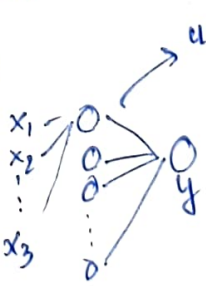
↳ Circuit theory and deep learning :→ There are functions that you can compute with a "small" L-layered deep NN that shallower networks require exponentially more hidden units to compute.

$f = x_1 \text{ (XOR) } x_2 \text{ (XOR) } x_3 \dots \text{ (XOR) } x_n \longrightarrow$

DNN → units required $O(\log(n))$



→ units required $O(2^{n-1})$



→ Building blocks of DNN

Layer l :→

↳ forward: Input $a^{[l-1]}$, output $a^{[l]}$

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

↳ $a^{[l]} = g^{[l]}(z^{[l]})$

↳ Cache $z^{[l]}$

↳ Backward: Input $da^{[l]}$ (Cache $z^{[l]}$) → output $da^{[l-1]}$ ($dW^{[l]}$, $db^{[l]}$)

↳ $dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$ (element wise)

$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]T}$

$db^{[l]} = dz^{[l]}$

$da^{[l-1]} = W^{[l]T} dz^{[l]}$

→ vectorised

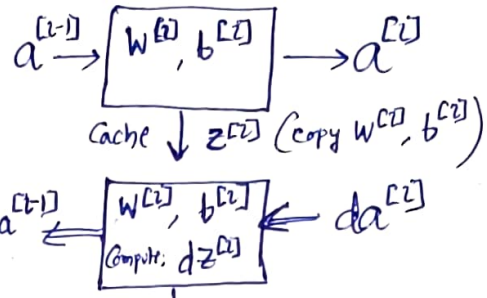
$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$

$dW^{[l]} = \left(\frac{1}{m}\right) dZ^{[l]} A^{[l-1]T}$

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

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

Layer l



initialize backward prop with $da^{[l]} = \frac{d\mathcal{L}(\hat{y}, y)}{d\hat{y}}$ $\left[\hat{y}=a\right]$

↳ eg for binary classification $da^{[l]} = -\frac{y}{a} + \frac{(1-y)}{(1-a)}$

→ Hyperparameters :→ {learning rate α, # iterations, # hidden layers l, # hidden units $n^{[1]}, n^{[2]}, \ldots$, choice of activation function}

↳ these are tunable parameters which control the values of Parameters ($W^{[l]}, b^{[l]}$)

→ Test/Dev/Train Sets

Data: Train set | Dev set | Test set

(Hold out) (Cross validating) (Development set)

train by taking particular set of hyperparameters → test on Dev set → change hyperparameter/model

→ final model tested on test set.

↳ for smaller dataset: good split → 70/30 or 60/20/20

↳ for big data: dev set and test set % going smaller.

↳ Make sure that the dev and test sets come from the same distribution as train set

↳ Not having a test set might be okay :→ train/dev ↳ "test"

↳ okay if you don't want a completely unbiased performance of the algorithm as this might be overfitting on dev set.

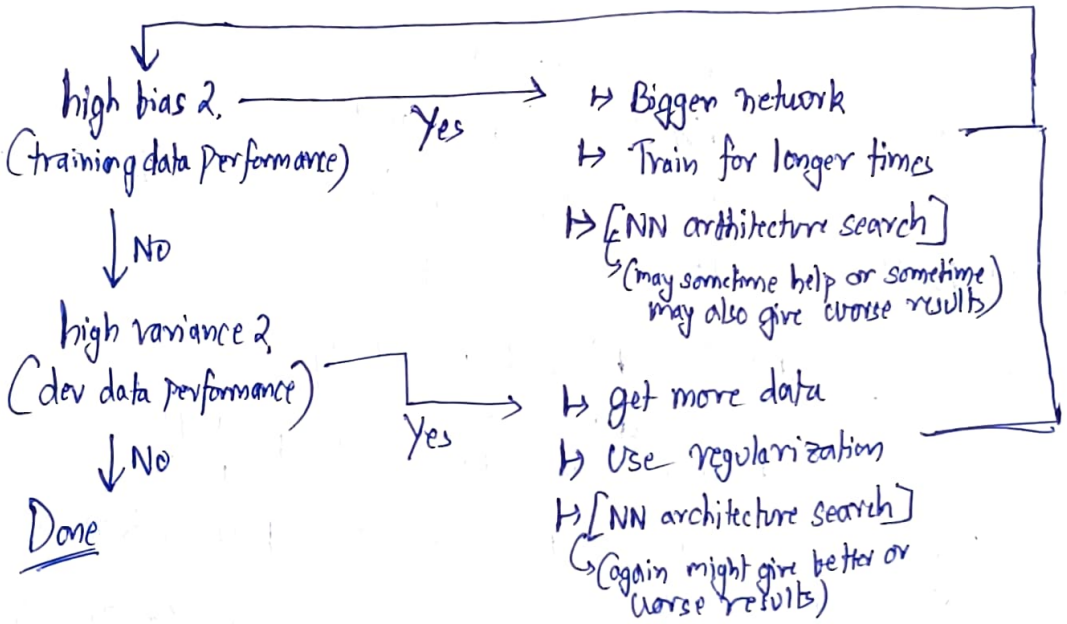
→ Bias and Variance in Deep learning

↳ bias :→ underfitting :→ oversimplification of model
↳ variance :→ overfitting :→ overcomplexity of model

high train and dev set error % ⇒ high bias
low train and high dev set err % ⇒ high variance

↳ Under the assumption { optimal (Bayes) error ≈ 0% → (Some classifier, eg. humans can perfectly classify it. ie for eg no noisy and undistinguishable data like blurry images)
↳ train and dev set from same distribution.

→ Basic Recipe for Machine Learning



★ Bias-variance tradeoff :→ earlier due to lack of tools, reducing one ⇒ increasing the other.
↳ but with availability of larger and larger data and ability to compute bigger and bigger network ⇒ no longer a trade-off
↳ bigger network almost always reduces bias without significantly changing variance (with proper regularization)
↳ bigger dataset almost always reduces variance without significantly changing bias

↳ Regularization in logistic regression (Parameters :→ $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$)
↳ Cost function modified :→ $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}$ (Reg-factor)
↳ L_2 regularization :→ Reg-factor $= \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$
↳ L_1 regularization :→ Reg-factor $= \|w\|_1 = \sum_{j=1}^{n_x} |w_j|$ → w will be sparse here (dimensionality reduction probable (see earlier))
↳ not considering b as $\text{len}(b) = 1 \ll \text{len}(w)$

→ Regularization in Neural Network

↳ Cost function :→ $J(w^{[1]}, b^{[1]}, \dots w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + (\text{reg-term})$

↳ L_2-norm known as Frobenius norm → $\boxed{\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2}$

↳ reg-term = $\frac{1}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$

↳ Change in the update term in gradient descent :→

$dW^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} W^{[l]} \rightarrow W^{[l]} := W^{[l]} - \alpha dW^{[l]}$

↳ hence Frobenius regularization also called weight decay

↳ How does regularization help ? → penalty for larger $W^{[l]}$ values ⇒ $W^{[l]}$ values kept small

⇒ dimensionality reduction of the parameter space

⇒ reduced overfitting.

→ Dropout Regularization

↳ overview :→ Choose randomly some neural nodes and drop them out ⇒ simpler NN.

↳ Inverted Drop-out : let suppose for layer $l=3$, def keep-prob = 0.8

↳ $d3 = np.random.rand(a3.shape[0], a^{[3]}.shape[1]) < \text{keep-prob}$ { $a^{[3]} \in \mathbb{R}^{(n^{[3]}, m)}$

↳ $(n^{[3]}, m)$ boolean matrix

↳ for each training-example, different nodes shut-off.

↳ $a^{[3]} = a^{[3]} * d3$

↳ $a^{[3]} = (a^{[3]} / \text{keep-prob})$ → because $z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$

otherwise will get reduced by 0.2

hence this step keeps expectation($z^{[4]}$) same.

NT

When making predictions at test time

↳ no nodes are dropped out and $\hat{y}$ calculated as usual using all nodes of all layer

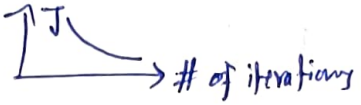
↳ because random dropout of nodes will be incoherent and random outputs increasing noise in the predictions.

↳ also due to the "inverted" dropout step (third step) no need of any normalizing term while making predictions.

↳ Why does drop-out work ? :→ each node can't rely on any one prev-node/feature so have to spread out the weights ⇒ shrink weights ⇒ similar to L_2 regularization.

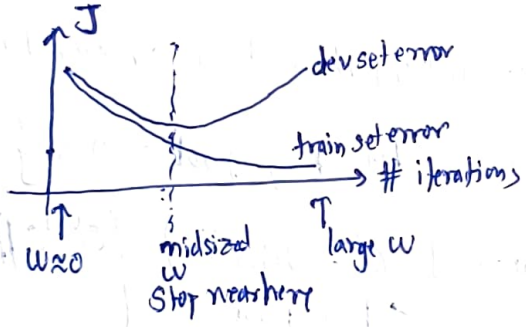
↳ each layer can be given different "keep-probs" values based on the number of nodes in them. ↘ may be smaller layers given larger keep-probs
↳ but increases # hyperparameters.

↳ Downside of Dropout :→ Cost function J is no longer well defined
⇒ Can't get smooth J vs # iterations curve to troubleshoot regularization



→ Data Augmentation :→ generating additional (pseudo) dataset from existing data to simulate having a large dataset
↳ eg :→ flipping cat pictures, adding some kind of noise and rotations in OCR data

↳ Early stopping



One-downside :→ couples the two task of ① optimizing cost function J and ② not overfitting
↳ want to orthogonalize these tasks so as to better find the hyperparameters.

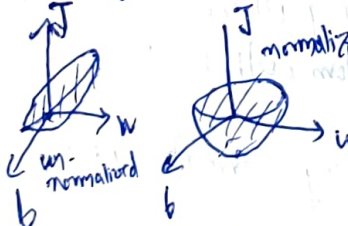
One-upside :→ less computation required compared to L_2 regularisation where it is needed to train again and again for different λ

→ Normalizing training sets :→

$X := \frac{x-\mu}{\sigma}$ || $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$; $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)**2}$

↳ Note :→ use the same μ and σ for both training and test set

↳ Why normalize ? ↘ data features otherwise have different ranges ⇒ W and b parameters have significantly different ranges ⇒ elongated cost-function ⇒ larger time for gradient descent

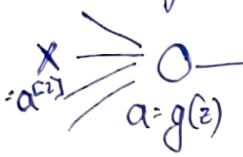


→ Vanishing / Exploding gradients :→ $W^{[l]} > 1$ —very deep NN→ Exploding gradients
$W^{[l]} < 1$ —very deep NN→ Vanishing gradients

as $\hat{y} = W^{[L]} W^{[L-1]} \cdots W^{[1]} x \Rightarrow \{\hat{y} \propto (W^{[l]})^L\}$ → Similar scaling in other scenarios with very large deep Neural Nets.
(for eg. for $g(z) = z$ and $b^{[l]} = 0$) → and all $W^{[l]}$ same

→ One approach to reduce the problem of exploding/vanishing gradients

for a single neuron



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

↳ to keep gradients stable, large n → smaller w_i

↳ $Var(w_i) \propto \frac{1}{n}$

So, initialize weights (for deeper Nets) as

$W^{[l]} = np.random.randn(shape(w^{[l]})) * \sqrt{\frac{2}{n^{[l-1]}}}$ → variance of a normal dist

→ for ReLU

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

→ Xavier initialisation

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

→ Verifying backprop steps through gradient checking using numerical approximation

↳ two point formula to approx gradient: $f'(\theta) \approx \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon}$ // error $= O(\epsilon^2)$

↳ first take $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots W^{[L]}, b^{[L]}$ and reshape it into a big vector θ

↳ take $dW^{[1]}, db^{[1]}, \dots dW^{[L]}, db^{[L]}$ and reshape to vector $d\theta$

↳ now, $d\theta_{approx}^{[i]} \equiv \frac{J(\theta_1, \theta_2, \dots \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots \theta_i - \epsilon, \dots)}{2\epsilon} \approx d\theta^{[i]} = \frac{\partial J}{\partial \theta_i}$

↳ Check: $\epsilon = 10^{-7}$

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

10^{-5} — double check

10^{-3} — worry.

↳ Grad-check only to be used in debug mode not in training mode

↳ when regularization is on, take the reg term in your cost function defination.

↳ Grad-check don't work with dropout so switch-off dropout to check grads

→ Batch gradient descent :→ in which the whole training example is used at once in a single gradient descent step

→ Mini-batch gradient descent :→ the whole training set is divided into mini-batches to be each used in a single gradient descent step.

↳ when # training example is very large ⇒ long time for batch gradient decent ⇒ mini-batch found to be a faster approach

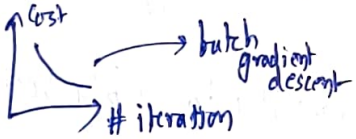
$X = [\underbrace{x^{(1)}, x^{(2)}, \dots x^{(1000)}}_{X^{\{1\}}} | \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}} | \dots]$

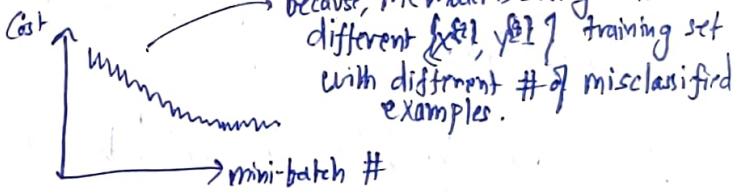
minibatch t :→ $X^{\{t\}}, Y^{\{t\}}$

↳ After doing forward prop and backprop on each minibatch t, a single step of gradient descent is taken.

↳ Epoch :→ After the whole training dataset has passed through forward-backprop once
↳ single step of grad descent in batch grad descent whereas multiple steps for minibatch

→ Cost function with mini-batch gradient descent :→





→ Choosing the minibatch size ↳ Size = m :→ batch grad descent
Size = 1 :→ stochastic gradient descent

stochastic grad descent	mini-batch	batch
↳ lose the speed gained from vectorization	↳ Fastest as ↳ vectorization used ↳ make progress without waiting for whole training set to be processed	↳ too long per iteration
↳ the movement towards minima highly noisy and never converge		

Tips :→
↳ if small training set ⇒ batch gradient descent ($m \leq 2000$)
↳ if bigger then typical mini-batch sizes :→ 64, 128, 256, 512
↳ usually in powers of 2 as it is master b/c the way memory is there in hardware
make sure minibatch size ≤ CPU/GPU memory

→ Exponentially Weighted Moving Averages

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

↳ here V_t is approximately averaging over the θ values of $\left(\frac{1}{1-\beta}\right)$ previous steps

↳ $V_t = (1-\beta)\theta_t + (1-\beta)\beta\theta_{t-1} + (1-\beta)\beta^2\theta_{t-2} + (1-\beta)\beta^3\theta_{t-3} + \dots$

↳ "exponentially decaying weights"

↳ $V_\theta := \beta V + (1-\beta)\theta$ → Advantage :- very little memory usage (only store previous V)

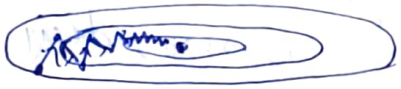
→ Disadvantage :- not as accurate in calculating averages compared to taking means

↳ Bias Correction :- In exponentially weighted moving average, we have a problem in getting the proper values of initial V_t as they are given very small contribution from data points as there are less number of previous points.

→ $V_0 = 0 \;||\; V_1 = \underbrace{(1-\beta)\theta_1}_{\text{small value}} \;||\; V_2 = \underbrace{\beta(1-\beta)\theta_1 + (1-\beta)\theta_2}_{\text{small weights}} \;||\; \dots$

hence add a bias correcting weight :- $\left[\frac{V_t}{1-(\beta)^t}\right]$

→ Disadvantage of gradient descent :-



→ too large a learning rate :- the gradient descent oscillations will diverge

→ too small a learning rate :- although will go in the right direction (overall) but due to many oscillatory steps will take a large number of steps.

↳ Gradient descent with momentum

↳ on iteration t of grad descent :-

dW, db from batch/mini-batch

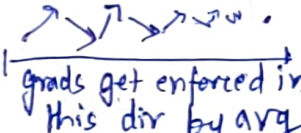
$V_{dw} := \beta V_{dw} + (1-\beta)dW$

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

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

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

→ using the exponentially weighted avg of gradient helps in reducing the grads in the oscillating direction and enforces the grads towards the global minima :-



→ Almost always works better than simple grad descent

→ hyperparameter β :- most usual and best value = 0.9 (avg over 10 grads)

→ RMSProp

↳ on t iteration of grad descent :-

$S_{dW} := \beta_2 S_{dw} + (1-\beta_2)dW^2$

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

$W := W - \alpha \frac{dW}{\sqrt{S_{dW}} + \epsilon}$ || $b := b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$

Logic :- the parameter directions along which the oscillations are happening would have large RMS gradient(s) and hence the parameters in that direction would be updated by a smaller factor

→ can take larger learning rate for faster convergence.

→ to avoid division by zero error $\epsilon \approx 10^{-8}$ used

→ **Adam Optimization Algorithm** (RMSprop + momentum)

Adam: → Adaptive moment estimation

↳ initialize :→ $V_{dw} = S_{dw} = V_{db} = S_{db} = 0$

↳ iteration t :→ get dW, db using mini-batch

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

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

↳ $V_{dw}^{Corrected} := V_{dw}/(1-\beta_1^t)$ // $V_{db}^{Corrected} := V_{db}/(1-\beta_1^t)$ // $S_{dw}^{Corrected} = \frac{S_{dw}}{(1-\beta_2^t)}$ // $S_{db}^{Corrected} = \frac{S_{db}}{(1-\beta_2^t)}$

↳ $W := W - \alpha \frac{V_{dw}^{Corrected}}{\sqrt{S_{dw}^{Corrected}} + \epsilon}$ // $b := b - \alpha \frac{V_{db}^{Corrected}}{\sqrt{S_{dw}^{Corrected}} + \epsilon}$

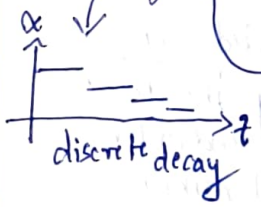
→ **Hyperparameter choice** :→ α : need to tune // $\beta_1 = 0.9$ // $\beta_2 = 0.999$ // $\epsilon = 10^{-8}$ {Recommended}

→ **Learning Rate Decay** → learning rate is reduced with increasing # iterations because as we move closer to the global minima, we want to oscillate in a smaller and smaller neighbourhood of the minima.

↳ decayrate :→ $\alpha = \frac{\alpha_0}{1 + (\text{decay-rate}) * (\text{epoch-num})}$ // hyper-parameters :→ α_0, decay-rate

↳ Other formulae → $\alpha = (0.95)^{\text{epoch\#}}\, \alpha_0$ — exponentially decay

$\alpha = \frac{k}{\sqrt{\text{epoch\#}}}\, \alpha_0$ or $\alpha = \frac{k}{\sqrt{\text{batch\#}}}\, \alpha_0$



{ large training models
Sometimes have manual changes in decay rate
(taking days to complete) }

→ **The problem of local optima** :→ is very-very rare to get a local optima in which all the directions has ∪ (or all have ∩) inflection.

↳ Probability ($\propto 2^{-(\#\text{ of parameters})}$) → usually get saddle points where only some directions have ∪ or ∩

↳ **the problem of plateaux** → can make learning slow as gradient at those places are nearly zero.

algorithms like Adam can speed-up the rate.

→ hyperparameters tuning preferences :→

highly likely to tune ← | highly unlikely to tune →

α (learning rate) > β, # hidden units, mini-batch size, > # layers, learning rate decay > Adam's $\beta_1, \beta_2, \epsilon$

* Trying out hyperparameters values : select points in the hyperparameter space in random
  ↳ don't select hyperparameters using grid search
  ↳ because, random way of using hyperparameter search gives the advantage of having an unique value for each hyperparameter in each trial run.

* Try to implement Coarse to fine approach :→ Search for hps in a broad area first and then search densely in a promising smaller area.

↳ hyperparameter search scales :→ Uniform random search in linear scale :→ # layes, # hidden units
uniform random search in log scale :→ α, β

→ Two approaches for hp tuning ↳ baby sitting one model :→ (when low computational resource)
↳ keep on looking the performance of ongoing model training and nudge the hyperparameters accordingly
↳ Training many models in parallel :→ get the hps which have best and fastest convergence.

→ Normalising inputs to speed-up learning
↳ just like in logistic regression, normalising inputs helps to train w, b faster,
↳ in DL for layer l :→ normalizing $z^{[l-1]}$ can help to train $W^{[l]}$, $b^{[l]}$ faster
→ Batch Normalization

for $z^{[l]}$:→ $\mu = \frac{1}{m}\sum_i z^{(i)}$ // $\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$

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

$\tilde{z}^{(i)}_{norm} = \gamma z^{(i)}_{norm} + \beta$ { β, γ are learnable parameters where γ is new std and β is new mean }

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

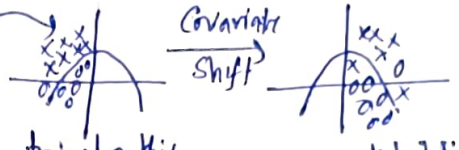
here we use different γ and β which may or maynot be 1 and 0 as we may want to have the mean and std of the dataset to be something which might take the use of not just the middle portion of the activation function

Adds additional set of parameters to the model :→
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots \beta^{[L]}, \gamma^{[L]}$
↳ Batch normalization implemented usually with mini-batches

$\{\beta^{[l]} = \beta^{[l]} - \alpha d\beta^{[l]}\}$ → parameter update

↳ but we no-longer have the set of parameters $b^{[l]}$ as they get cancelled out while taking mean of $z^{[l]}$ to 0 and then to $\beta^{[l]}$ as $\{z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}\}$

↳ Covariate shift :→ related to the need to retrain the model if the distribution of the inputs (x) changes even though the ground truth function is same. Eg-1



Eg-2 → want to identify white-cats when the model to identify cats are trained on darker-colored cats.

Why this a problem in NN

↳ the input distribution of the m^{th} hidden layer gets covariate shifted when W, b of earlier layers are changed during training → batch normalization (BN) helps in stabilizing the inputs to the m^{th} hidden layer by keeping their distribution almost the same → this helps in weakly de-coupling the hidden layers' training as each hidden layer has it's own input distribution which is full-proof

This weak decoupling helps in getting faster training

↳ Batch Norm at test time

↳ BN uses μ and σ^2 of the given minibatch to normalize.

↳ but how to get those values at test time?

↳ We use exponentially weighted averages over mini-batches.

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

$\mu^{\{1\}[l]}$ $\mu^{\{2\}[l]}$ $\mu^{\{3\}[l]}$ ... | $\mu^{[l]}$

$\sigma^{\{1\}[l]}$ $\sigma^{\{2\}[l]}$ $\sigma^{\{2\}[l]}$ .. | $\sigma^{[l]}$

at test-time

$$z^{[l]}_{norm} = \frac{z^{[l]} - \mu^{[l]}}{\sqrt{\sigma^{2[l]} + \epsilon}} \rightarrow \tilde{z}^{[l]} = \gamma z^{[l]}_{norm} + \beta$$

↳ Multiclass-classification in Neural Networks

↳ C = # of classes :→ $n^{[L]}$ = # nodes in output layer = C $\quad a^{[L]} = \hat{y} \in \mathbb{R}^{(C,1)}$

for eg:→ $\hat{y} =$ [ ] → $P(C=0|x)$; [ ] → $P(C=1|x)$; [ ] → $P(C=2|x)$

↳ uses a Softmax activation function in the final softmax layer

↳ $z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$ $\quad (C,1)$

↳ Softmax activation → $t = e^{(z^{[L]})}$ $\quad (C,1)$

↳ $a^{[L]} = \frac{t}{\sum_j t_j}$ $\quad (C,1)$

for eg:→ $z^{[L]} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \Rightarrow a^{[L]} = \frac{1}{(\exp(1)+\exp(2)+\exp(3)+\exp(4))} \begin{bmatrix} \exp(1) \\ \exp(2) \\ \exp(3) \\ \exp(4) \end{bmatrix}$

★ The softmax classifier reduces to logistic regression for the case $C = 2$.

↳ Loss function :→ $\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j \log(\hat{y}_j)$ $\quad$ Cost function $J(w^{[1]}, b^{[1]}, \ldots w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

↳ Backprop : initialization :→ $dz^{[L]} = \hat{y} - y$ $\quad (C,1)$

↳ Orthogonalization of controls in ML

↳ we want to orthogonalize the cause and effect of tuning the various hyperparameters/strategies so as to narrow down our search for better models

↳ Chains of Assumptions in ML

① Fit training set well on cost function :→ tuned by :→ bigger network, changing optimizer, etc

② Fit dev set well on cost function :→ tuned by :→ regularization, bigger train set

③ Fit dev/test set well on cost function :→ tuned by :→ bigger dev set

④ Performs well in real world :→ tuned by :→ change dev set/change cost function.

↳ We tend to avoid strategies like early stopping which can tune more than one assumptions above (eg. early stopping tunes simultaneously both ① and ②)

*→ It is always useful to have a single number evaluation metric which would help to objectively differentiate between different models.
(eg→ use F1 score instead of using both of the precission and recall)

↳ Optimising and Satisficing metric

↳ Optimising: the metric which we want to maximize

↳ Satisficing: the metric which we just want to be above/below certain values

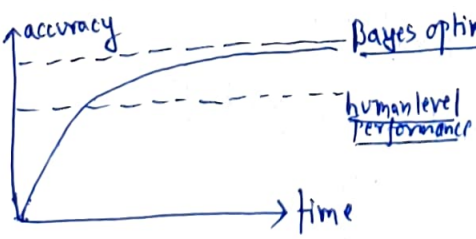
eg: maximizing Accuracy keeping running time < 100ms

↳ When we care about N number of metrics then
↳ take one (most-important one) as an optimizing metric
↳ all other N-1 as satisficing metric

*→ Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on

*→ Choosing the dev set and the evaluation metric determines what target you want to achieve

→ Comparing to human-level performance :→



Bayes optimal error :→ maximum accuracy one can theoretically any function can achieve from mapping X :→ Y
↳ limited by noise in the input

↳ Why compare with human level performance?

↳ Because, so long as ML is performing worse than Human, the model accuracy can be tried to be improved by :-

↳ getting labeled data from humans

↳ better analysis of bias/variance

↳ gain insights from manual error analysis: why person got it right? and machine got it wrong?

↳ human level performance as a proxy for Bayes error

eg:

human error	1%	7.5%
Training error	8%	8%
Dev error	10%	10%
	Focus on bias	Focus on variance

→ Avoidable bias (difference b/w human (≈ Bias) error and training error)

→ variance

↳ You should always consider the best human level performance available out there as a proxy to decide upon bayes error and whether to reduce bias or variance. (Choose the lower error% from team of doctors vs higher error% for one doctor as human level proxy (≈ bayes error) ↳ (example)

→ Carrying out Error analysis

↳ Check the mislabelled data (both false positive and false negative) → Categorise the mislabelled based on what went wrong: (for eg:- blurry image, mislabelling on a special kind of images, etc)

Strategise to figure-out those major issues first. ← try to find-out the major issues (which issue causing the most mislabelling)

*↳ Deep-learning Algorithms are quite robust to random-errors in the training set. But they are not robust to systematic errors.

*↳ If there is a significant amount of incorrectly-labelled data in the dev set which is causing a significant % of total mislabelled cases by the model, then it is worthwhile to go and correct out the incorrectly-labelled dev data (by the humans).

*↳ Build your first system quickly and then iterate :-

↳ Set up dev/test set and metric → build initial system quickly → Use bias/variance analysis and error analysis to prioritize next steps.

→ Scenario :→ Different training and testing distribution

↳ suppose you have a large dataset from one distribution (eg. web scrap) which is different from the distribution where the model is going to be used (eg. user uploaded images). Two approaches here :→ (very little data from the target distribution)

① shuffle the data from both large dataset and target dataset and make the train/dev/test split (all from same distribution)

↳ Problem :→ the dev/test distribution will have a very small fraction of the target dataset and hence model will get more optimised for the large dataset.

② make the test/dev from the target distribution and keep the remaining ~~small~~ part of target distribution along with the large one as the training set

↳ train distribution different than ~~train~~/test/dev

↳ but optimization more towards target distribution

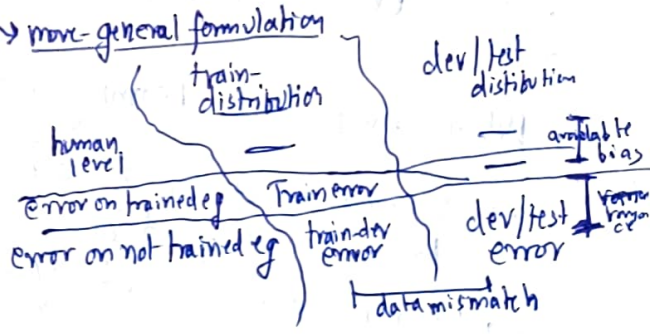
↳ Bias, variance on different training and testing distribution

↳ Train-dev set : Same distribution as training set, but not used for training :

	Bayes error	avoidable bias
Same distribution	Training error	variance
	Train-dev error	data mismatch
Same distribution	dev error	degree of overfitting to the dev set
	test error	

* Note :→ when the dev/test distribution is "easier" then it might sometime happen that train error > dev error.

↳ more general formulation



→ Addressing data Mismatch

↳ carry out manual error analysis to try to understand difference b/w training and dev/test sets

↳ make training data more similar, or collect more data similar to dev/test sets

→ Artificial data synthesis

↳ eg :→ adding noise to data

↳ eg :→ generating images using computer graphics

↳ Risk :→ might overfit to noise/graphical objects.

↳ Transfer Learning

- taking an model trained on one dataset to use it for a different application with fewer data. Eg:- model trained on image recognition transfered to learn radiology diagnosis.
- Makes sense only when the transfer learning is happening from a model with a large amount of data to an application with limited amount of data.
- As the initial basic layers would stay similar in transfer learning, Priority is given to train the final layer(s) (retraining the $W^{[L]}$ $b^{[L]}$ or adding more layers at the end, etc) based on the amount of data available for finetuning

{A} Pre-trained model (on large dataset) —transfer learning (eg:- retraining output layer, adding additional layer at the end)→ Fine-tuning (on smaller dataset) {B}

- Task A and B must have same input
- low level features from A should be helpful for learning B

→ **Multi-task learning** :- using the same NN to do multiple things (eg. classify multiple different things → Presence of car, traffic light, sign, etc)

This case is different from softmax regression

- output layer has node # = number of classifiers we want.
- Loss function $\mathcal{L} := \frac{1}{m} \sum_{i=1}^{m} \sum_{j} \mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$
- as one image has multiple labels here

When multi-task learning makes sense

- Training on a set of tasks that could benefit from having shared lower-level features.
- usually: amount of data you have for each task is quite similar
- can train a big enough neural network to do well on all the tasks!

→ End to End Deep Learning

Pros:
- Let the data speak
- Less hand-designing of components needed

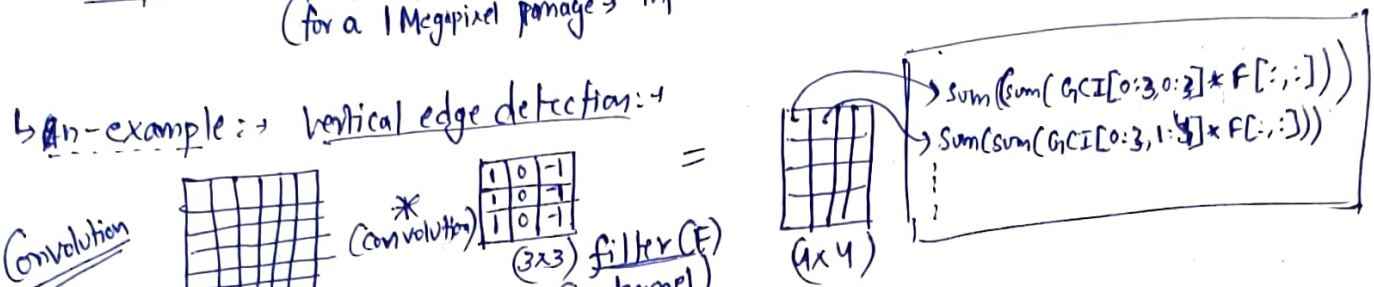
Cons:
- need large amount of data
- excludes potentially useful hand-designed components in case of less data.

Use end-to-end deep learning only when there is enough data to learn the complexity of the problem in hand.

Computer Vision

- one-problem: Very large input parameter and hence input W and b
  (for a 1 Megapixel image → input vector dim = 1000 × 1000 × [rgb] × 3)

- An-example: Vertical edge detection:



- Other vertical edge filters:

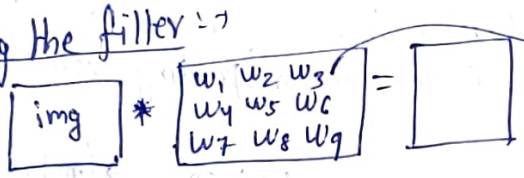
1	0	-1
2	0	-2
1	0	-1

Sobel filter

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

Scharr filter

- learning the filter:



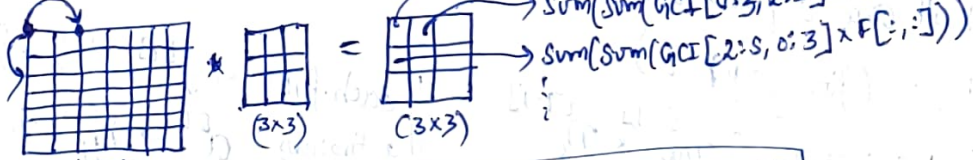
→ learn them as "weights"
→ using NN these filters to detect low level features in images can be learnt easily.

- $(n \times n) * (f, f) = (n-f+1, n-f+1)$
  - every time we apply convolution, the image shrinks,
  - also, the pixels at the edges are used much less (info thrown away)
- Solution: Pad the image: p = padding = 1 → $(n+2, n+2)$
  - $(n+2, n+2) * (3, 3) = (n, n)$

Two types

- "Valid": no-padding
  $(n \times n) * (f, f) = (n-f+1, n-f+1)$
- "Same": same input and output size
  $(n+2p, n+2p) * (f, f) = (n, n)$ → $p = \frac{f-1}{2}$ → by convention: in computer vision f = odd

- Strided-convolution



general (stride = s) (padding = p):

$$(n \times n) * (f \times f) = \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

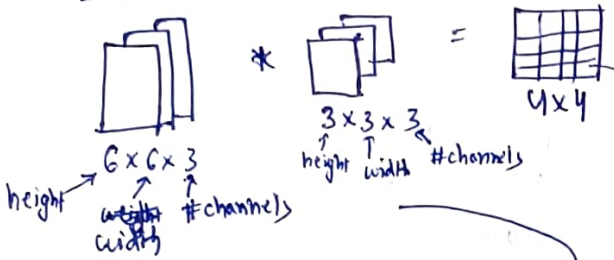
$\lfloor z \rfloor$ = floor of z

↳ Technical note on Cross-correlation and convolution

↳ In reality the "convolution" done in dL is actually cross-correlation, but by convention it is being called convolution

↳ In actual convolution, an additional step of flipping the filter both horizontally and vertically is done before multiplying to preserve associativity ($(a * b) * c = a * (b * c)$)

→ Convolutions on RGB images :→



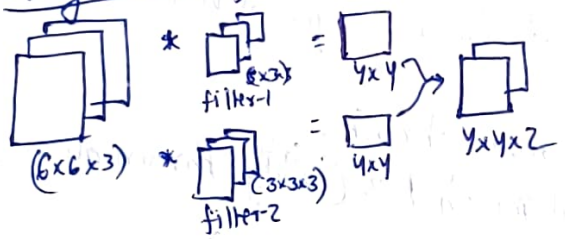
Note :→ (1) # channels of both the input and the filter must be same

(2) the output is a single layer always

(3) each number is the summation of the 27 multiplications (element wise) by sliding the cube over the whole image cube

(4) each channel of the filter detects features in corresponding each channel of input

→ Using multiple-filters at once



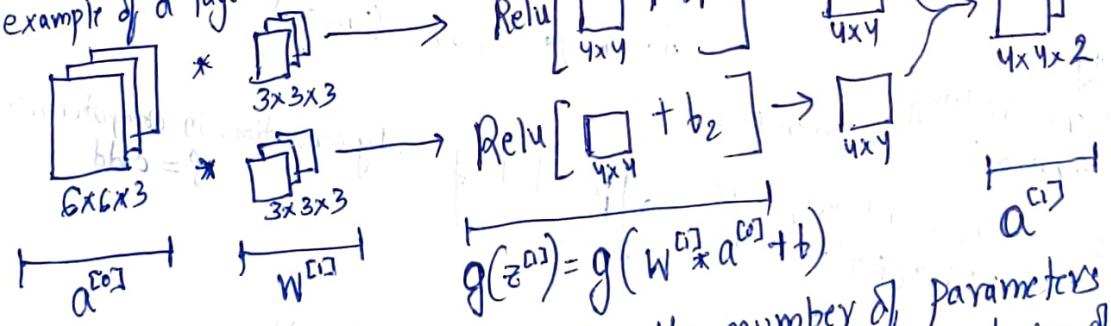
$(n \times n \times n_c) * (f \times f \times n_c) \rightarrow (n-f+1) \times (n-f+1) \times n_c'$

channels (aka depth)

filters

→ A single layer of a Convolutional Network

↳ example of a layer :→



* Note :→ no matter how big the image is, the number of parameters is pretty small and depends on the number of filters and size of filters.

→ Summary of notation :→ (for layer 2)

$f^{[2]}$ → filter size

$p^{[2]}$ → padding

$s^{[2]}$ → Stride

Input :→ $n_H^{[2-1]} \times n_W^{[2-1]} \times n_c^{[2-1]}$

Output :→ $n_H^{[2]} \times n_W^{[2]} \times n_c^{[2]}$

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

each filter is : $f^{[2]} \times f^{[2]} \times n_c^{[2-1]}$

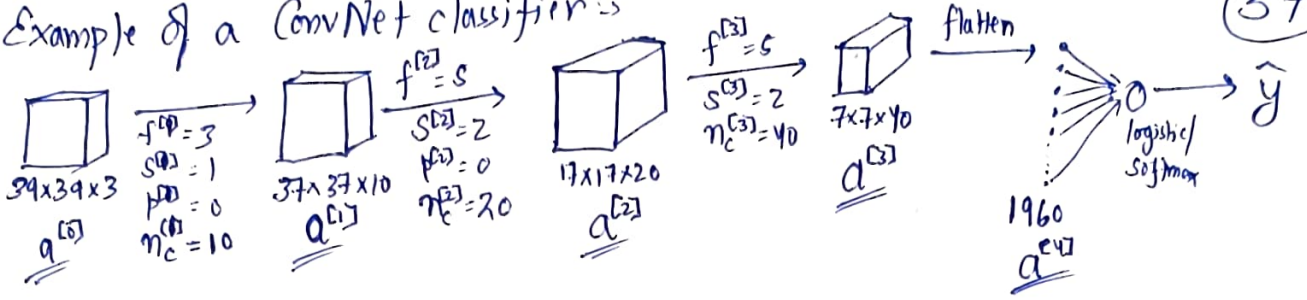
activations : $a^{[2]} \rightarrow n_H^{(2)} \times n_W^{(2)} * n_c^{(2)}$

$(A^{[2]} \rightarrow m \times n_H^{(2)} \times n_W^{[2]} \times n_c^{[2]})$

weights :→ $f^{[2]} \times f^{[2]} \times n_c^{[2-1]} \times n_c^{[2]}$

bias :→ $n_c^{[2]} \rightarrow (1, 1, 1, n_c^{[2]})$

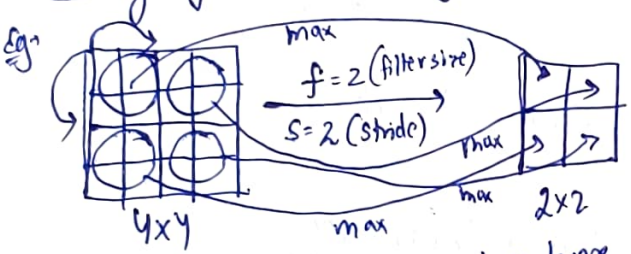
→ Example of a ConvNet classifier :-



→ Types of layers in a convolutional network :-

↳ Convolution (CONV) ↳ Pooling (POOL) ↳ Fully connected (FC)

↳ Pooling layer :- Max pooling



{ take the max value from the sub-matrix on which the filter is currently situated }

→ Intuition :- max pooling helps to preserve the info on the presence/absence of a feature at a particular region of the filter.

→ f, s are the hyperparameters here

→ No parameters to learn here

→ if multiple channels are present, the same computation is done over all the channels independently

→ Input size: $n \times n \times n_c$

→ Output size: $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times n_c$

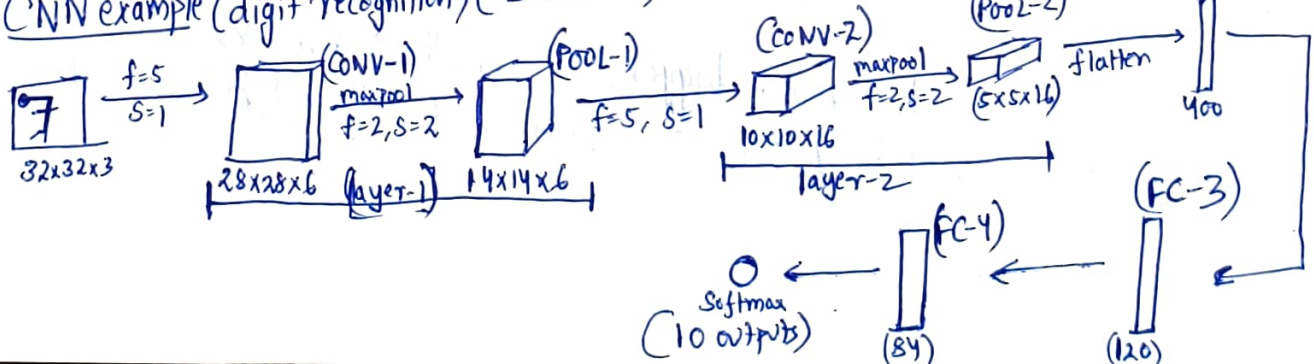
↳ Pooling layer :- Average pooling (not used very often) → instead of taking max, average is taken.

↳ Sometimes used very deep into the neural network to collapse a larger representation (image) to a smaller one.

↳ Usual hyperparameters for Pooling → $f=2, s=2$ or $f=3, s=2$, $p=0$, max pooling

↦ Note :- # of layers in a CNN is usually the # of layers with trainable parameters. So, the pooling layer is considered with the Conv layer as a single layer.

↦ CNN example (digit recognition) (LeNet-5)



→ Fully Connected layers (FC) :→ These are the usual ANN layers in which all input and output nodes are fully interconnected and they use the usual $W^{[l]}$, $b^{[l]}$ parameters.

↳ Note :→ In a usual CNN, as the network progresses, the n_H, n_W ↓ and n_c ↑

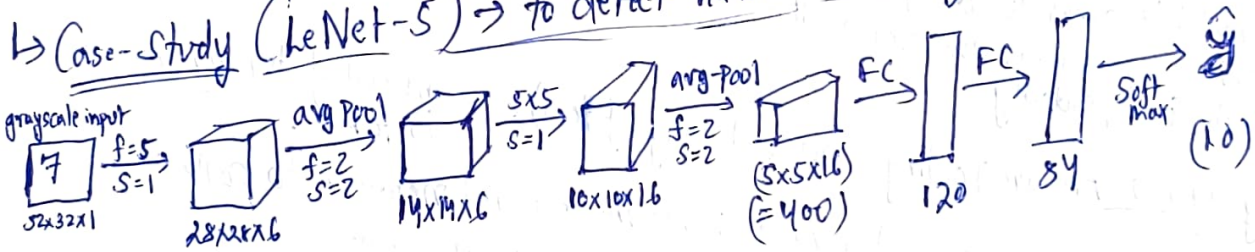
→ Why CNN can get away with smaller number of parameters compared to ANN

↳ Parameter sharing :→ A feature detector that's useful in one part of the image is probably useful in another part of the image.

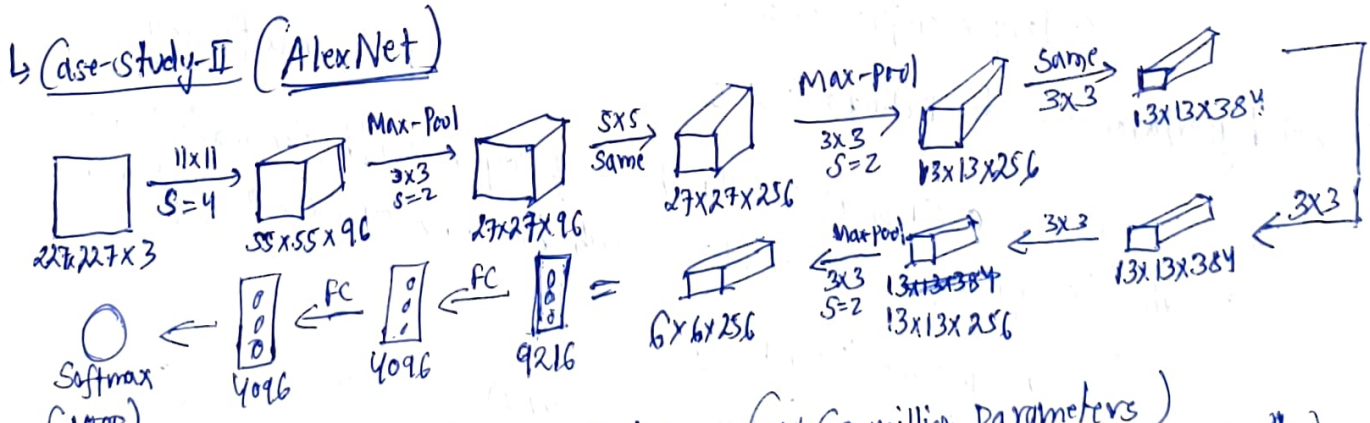
↳ Sparsity of connections :→ in each layer, each output value depends only on a small number of inputs.

↳ CNNs are good at working with translational invariance

↳ Case-Study (LeNet-5) → to detect hand written digits.



↳ Case-Study-II (AlexNet)



→ Similar to LeNet, but much bigger (~60 million parameters)

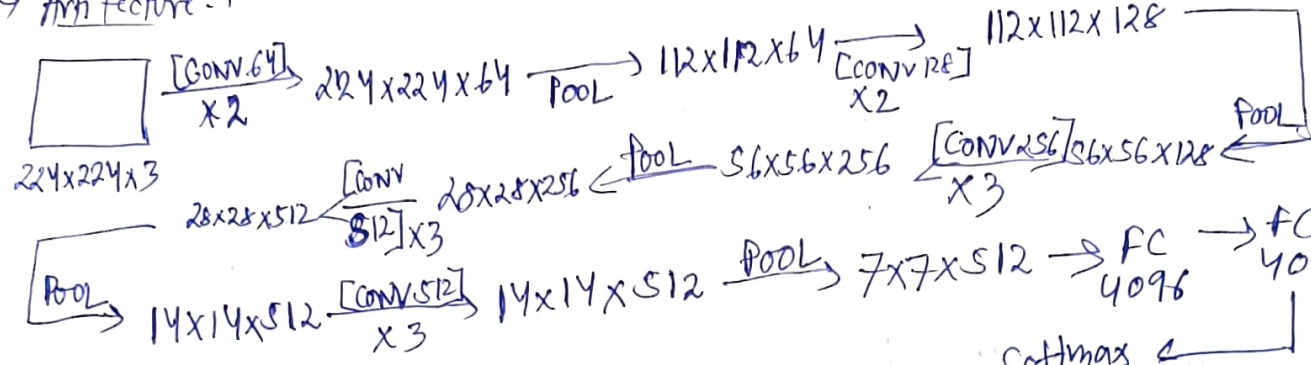
→ Uses ReLU compared to LeNet-5 which used sigmoid/~~ReLU~~ tanh

→ VGG-16 (case-study)

↳ uses the same CONV {3x3 filter, s=1, p=same} and Max-pool {2x2 filter, s=2} layers throughout

↳ eg. [CONV 64] x2 ≡ two conv layers with 64 filters each

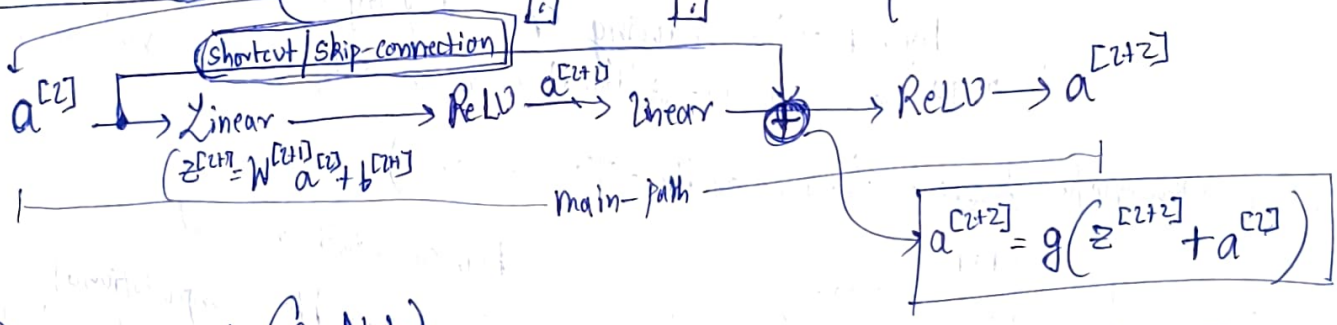
↳ Architecture:



↳ has 128 million parameters in 16 layers with trainable weights.

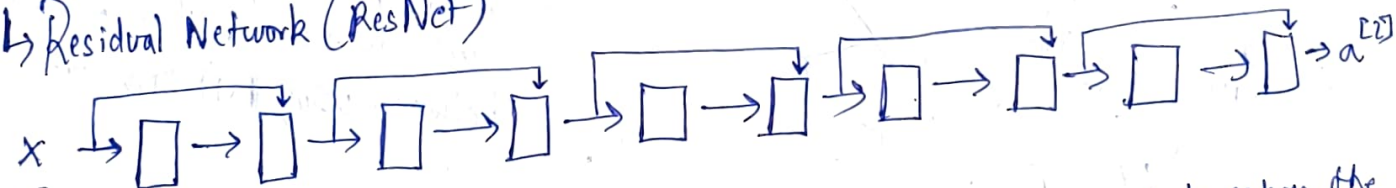
↳ but is simplistic.

↳ Residual block → {for eg:- $a^{[l]} \rightarrow a^{[l+1]} \rightarrow a^{[l+2]}$}

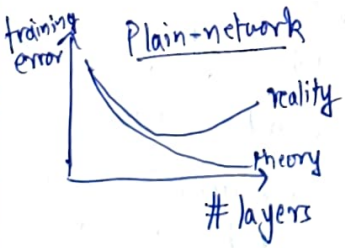


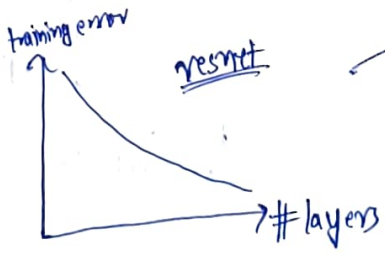
$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

↳ Residual Network (ResNet)



↳ 5-residual blocks





→ ResNet very effectively solves the problem of exploding/vanishing gradients giving the option to have very deep Neural networks.

↳ As the # layers increases, we get the problem of exploding/vanishing gradients

→ Why do residual networks work:?

↳ the residual blocks can easily learn the identity function in case of small W and b values so that the NN doesnot "stop" giving non-zero values.

$$a^{[l+2]} = g(\cancel{+}W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}) \underset{\text{ReLU}}{=} g(a^{[l]}) = a^{[l]}$$

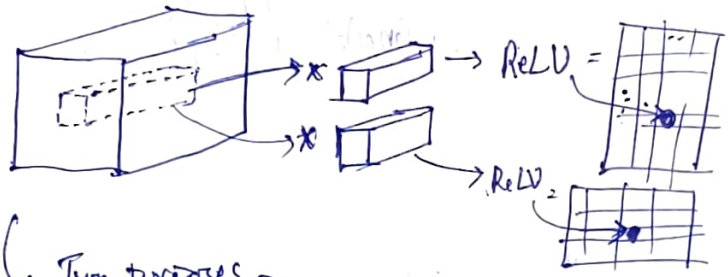
↳ So, this residual block can be effectively "skipped" in these scenarios. without decaying the performance of the whole NN.

Nt → if $\dim(a^{[l+2]}) \neq \dim(a^{[l]})$ in residual layer, then,

$$a^{[l+2]} = g(z^{[l+2]} + W_s a^{[l]}) \quad // \quad W_s \in \mathbb{R}^{\dim(a^{[l+2]}) \times \dim(a^{[l]})}$$

↳ either learnable or to make zero-padding

→ Network in Networks (1×1 convolutions)



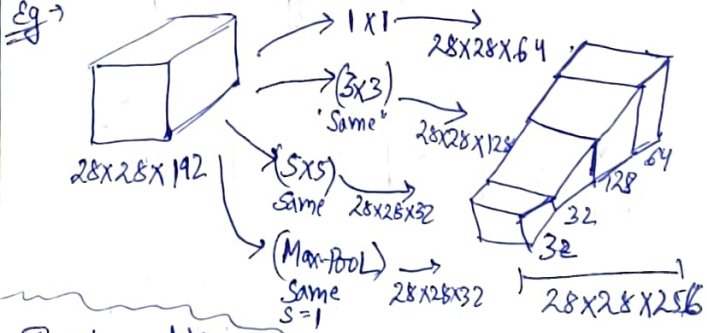
≡ a 1×1 convolution acts like a neural network.

↳ Two purposes → like pool-layers helps in reducing n_H and n_W. 1×1 CONV layer helps in reducing n_c (# of layers)

↳ It does also add additional non-linearity to the network

→ Inception network block (trying out all the filters at once)

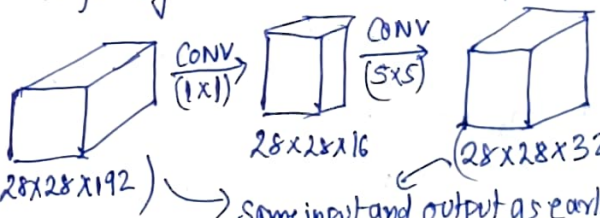
Eg →



↳ high computational cost

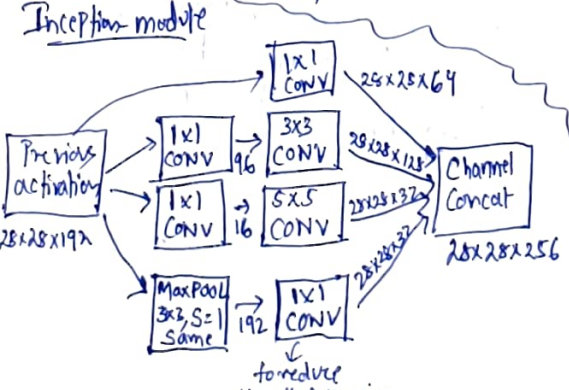
↳ to reduce the computational cost, we introduce a bottleneck layer in between using (1×1) CONV layer

↳ for eg:-



↳ same input and output as earlier

Inception module



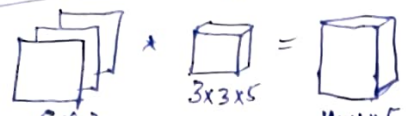
Inception Network → made by stacking up inception modules

↳ later inception modules in this network has an additional branch leading to a softmax to check output from the hidden layers

↳ also called (GoogLeNet). {"we need to go deeper" - from inception movie.}

→ Computational Cost of a normal convolution:-

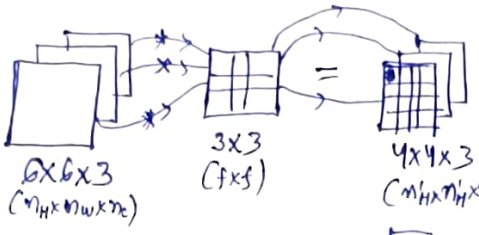
Eg →



Computational cost = # filter params × # filter positions × # of filters

→ Cost = (3x3x3) x (4x4) x 5 = 2160

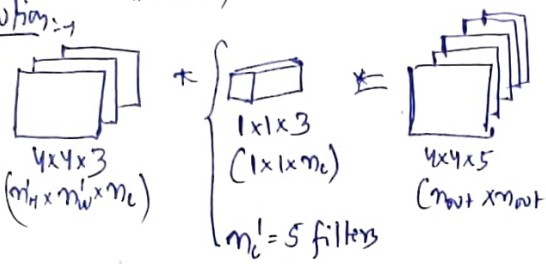
→ Depthwise-Convolution:-



Computational Cost = # filter params × # filter positions × # filter

432 = (3x3) x (4x4) x 3

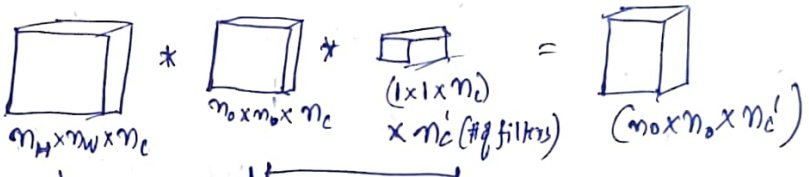
→ Point-wise-Convolution:-



Computational Cost = # filter params × # filter positions × # filter

240 = 1x1x3 x 4x4x5

→ Depthwise-Separable convolutions:-



→ MobileNets

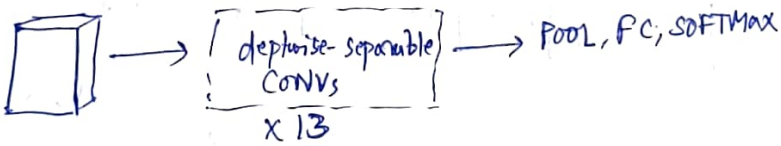
↳ using depthwise-separable convolutions

↳ used to reduce computations →

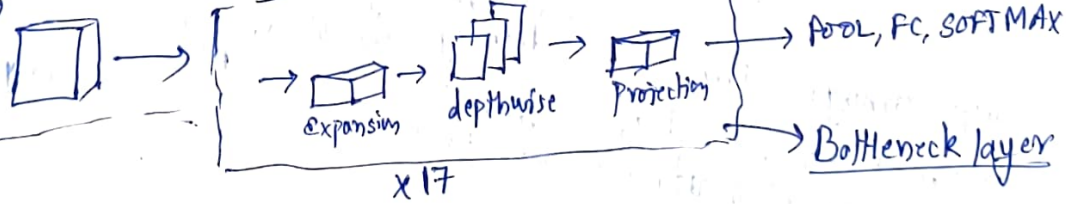
$$\frac{\text{Computation cost [MobileNets]}}{\text{Computation cost [Normal convolution]}} = \frac{1}{n_c'} + \frac{1}{f^2}$$

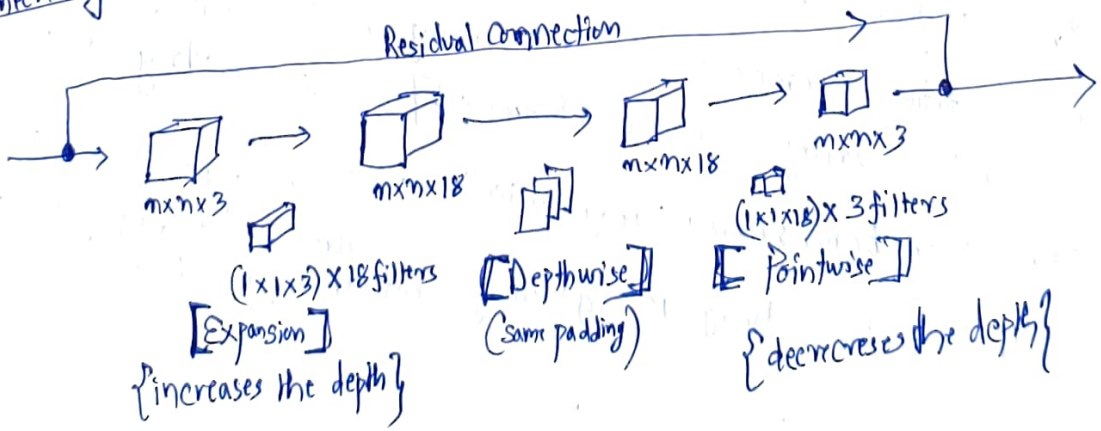
typically ≈ 1/10 as n_c' very large and $f = 3$

→ MobileNet V1 :-



→ MobileNet V2 :-





↳ Why mobilenet v2 bottleneck

↳ it allows to learn a richer (more complex) function as it expands the previous activation, as well as keeping the sizes of the activations small, as it converts back (projects back) to the original size (small) of the activation.

→ Efficient Net :- { how to make a CNN suitable for given computational resources }

↳ Three things to control :- ① r :- resolution of image ② d :- depth of NN ③ w : width of layers

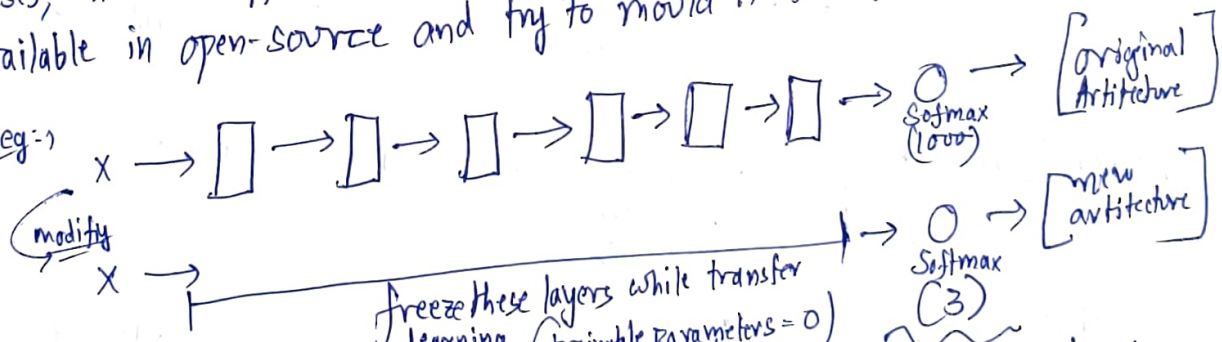
↳ see open-source implementations of efficient-Net.

→ Practical Advices on using ConvNets

↳ try to find open source implementations of the architecture that you are willing to try in sites like github.

↳ Many a times there are very large Neural Networks which would take a lot of time to set up and to do the optimal parameters searches. In these cases, it is helpful to use transfer learning from pre-trained models available in open-source and try to mould it for your application

↳ for eg :-



↳ Transfer learning in CNN

↳ almost always used unless there is a large amount of data and computational resources at your service.

↳ Common data augmentation techniques in CV :- mirroring, random cropping, [rotation, shearing, local warping, -- {less used techniques}], Color shifting, PCA color augmentation

↳ Tips for doing well on benchmarks/winning Competitions :- Ensembling : using different NNs and averaging their outputs

↳ Multi-Crop at testime :- taking (10-crop) images and averaging their outputs

→ Terminology used :-

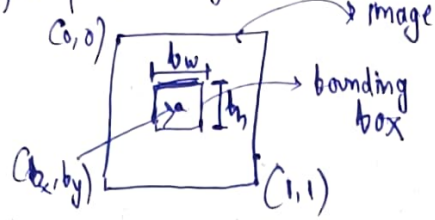
↳ (Image classification) and (Image classification and localization)

↳ both concerning only a single object classification.

↳ (detection) → Concerning multiple objects classification and detection.

↦ Classification with localization

↳ (target) output parameters for localization using bounding boxes :- $\hat{y} = (\text{softmax}, b_x, b_y, b_H, b_w)$ ← class label



↳ Defining the target label for a classification like :- (obj 1, obj 2, ... obj n, no-object) {ie either one of the object present or no object there}

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

p_c → {0: no-object, 1: object there}

$c_1, c_2, \ldots, c_n$ → softmax for which object present.

if $p_c = 0 \Rightarrow y = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ \vdots \\ ? \end{bmatrix}$

one kind of loss function :-

$$\mathcal{L}(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + \cdots + (\hat{y}_n - y_n)^2 & \text{if } y_1 = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 0 \end{cases}$$

→ Landmark detections ~ Landmarks are basically the x,y coordinates that can be taken as labels which would be representing certain key features in an image

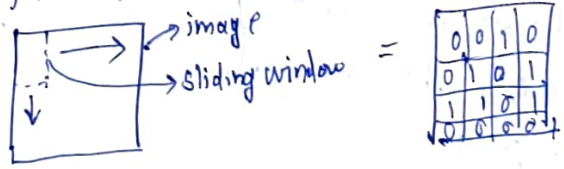
↳ eg :- Corner of eyes, etc.

→ Object detection using sliding windows detection

↳ First build a CNN which is trained on cropped images of single occurrences/non-occurrences of the objects of interest which would classify if the image is of that object or not

↳ Then on the target actual images, slide a window (from top to bottom) which would take that part of image into CNN as see if that is that object or not

↳ after that keep on increasing the window size.

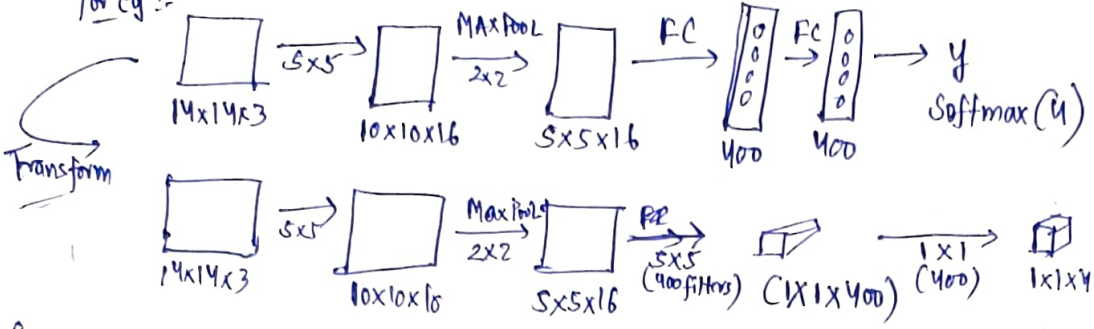


Drawback :-

↳ high computational cost at inference time.

→ Turning FC layers to CONV layers in a CNN

for eg :-

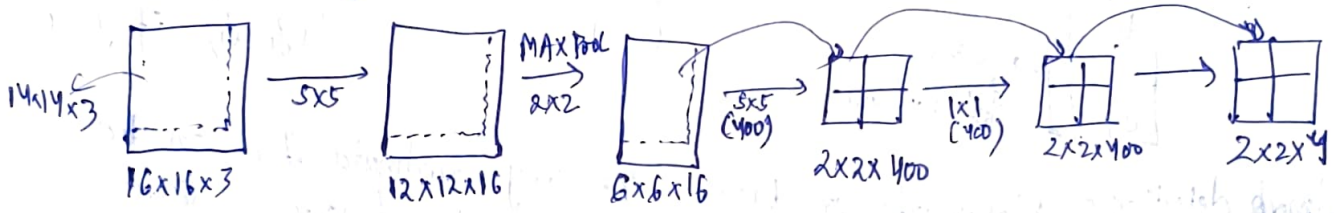


↳ Convolution implementation of sliding window

↳ it is based on the fact that sliding filter calculation shares a lot of similar calculations (as they are being calculated on shared parts of images)

↳ So, it can be done by a single forward propagation of the CNN (using the FC → CONV conversion)

↳ let the sliding filter be of size 14x14x3 on an image of size 16x16x3 (can be of any greater size)

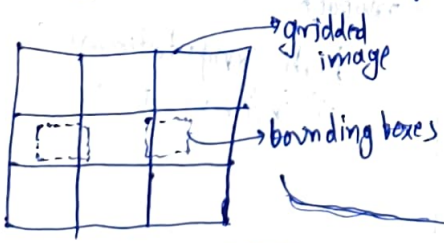


↳ Drawback :→ due to fixed shape and position of the bounding boxes, the algorithm is not so accurate in getting the correct (nearly) positions of them.

↳ YOLO [You Only Look Once] Algorithm.

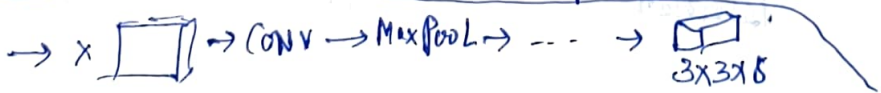
↳ follows the CNN way of splitting the image into 'grids' and each grid is associated with a target vector $\hat{y} = \begin{pmatrix} p_c, b_x, b_y, b_h, b_w, \\ c_1, c_2, \dots c_n \end{pmatrix}$ → {as seen in classification with localization}

↳ Note that it is done by single CNN network which has out layer of (3x3x8) → {for eg for a grid of 3x3 on the image having len($\hat{y}$) = 8}



↳ Assumptions :→ ① the object is associated to the grid box where its center is lying

② the grid sizes are small enough so that no two objects are in the same grid.

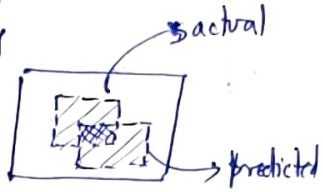


↳ So in YOLO, the bounding boxes are given by the (b_x, b_y, b_h, b_w) labels of each grid box

↳ Note that (b_x, b_y, b_h, b_w) are relative to a particular grid box and they must be transformed according to the whole image.

→ Evaluating Object Localization {Intersection over Union}

$$IoU = \frac{\text{intersection of ground truth and predicted bounding box}}{\text{Union of grand truth and predicted bounding box}}$$



↳ usually $IoU \geq 0.5$ ⇒ correct detection (Convention)

→ Non-max Suppression Algorithm

↳ One problem with YOLO :- multiple grid-boxes might predict multiple different bounding boxes for the same object

↳ Solution: non-max suppression algorithm (for one object detection)

↳ Each output prediction form each grid boxes :- $\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \end{bmatrix}$

↳ discard all boxes with $p_c \leq 0.6$

↳ For remaining boxes → pick the box with the largest p_c

↳ discard any remaining box with $IoU \geq 0.5$ with the box output in the previous step

Note

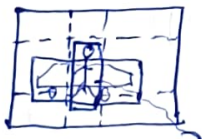
↳ For multi-class Object detection → run the algorithm independently for each class.

→ Anchor boxes → (used when there are more than one objects in the same grid)

↳ pre-define shape of anchor boxes for each grid {eg: ▭ and ▯}

↳ the output label size gets multiplied by the number of anchor boxes.

↳ With two Achor boxes :- each object in training image is assigned to grid cell that contains Object's midpoint and anchor box for the grid cell with highest IOU



eg: $y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \end{bmatrix}$ related to anchor box ▭ / related to anchor box ▯

$y = [1, b_x, b_y, b_h, b_w, 1, 0, 1, b_x, b_y, b_h, b_w, 0, 1]$

Limitations :- ↳ Can't work if # of objects in single grid > # anchor boxes

↳ Can't work if the objects in the same grid have the similar anchor box shape.

→ R-CNN (Region Proposal CNN)

↳ instead of running CNN over the whole image, R-CNN runs CNN only on particular regions of the image where there is high chance of finding the object using segmentation algorithm.

↳ R-CNN :→ Propose regions. classify proposed regions one at a time. output label + bounding box

↳ Fast R-CNN → propose regions, use convolution implementation of sliding windows to classify all the proposed regions.

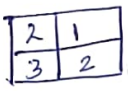
↳ Faster R-CNN :→ use convolutional network to propose regions.

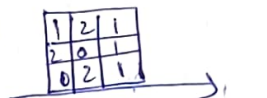
→ Semantic Segmentation → in this each pixel is classified to which class it belongs to. This helps in outlining the objects properly.

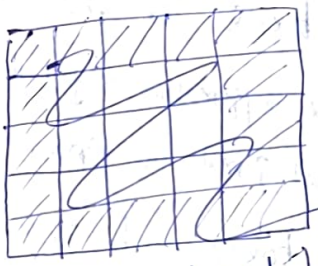
↳ The input and output size is to be same.
⇒ as the CNN goes deeper, the size of the image block needs to be increased and filter size decreased.

Transpose Convolutions

an example

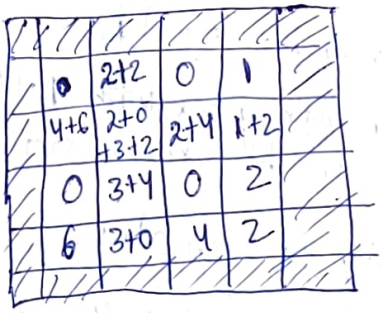




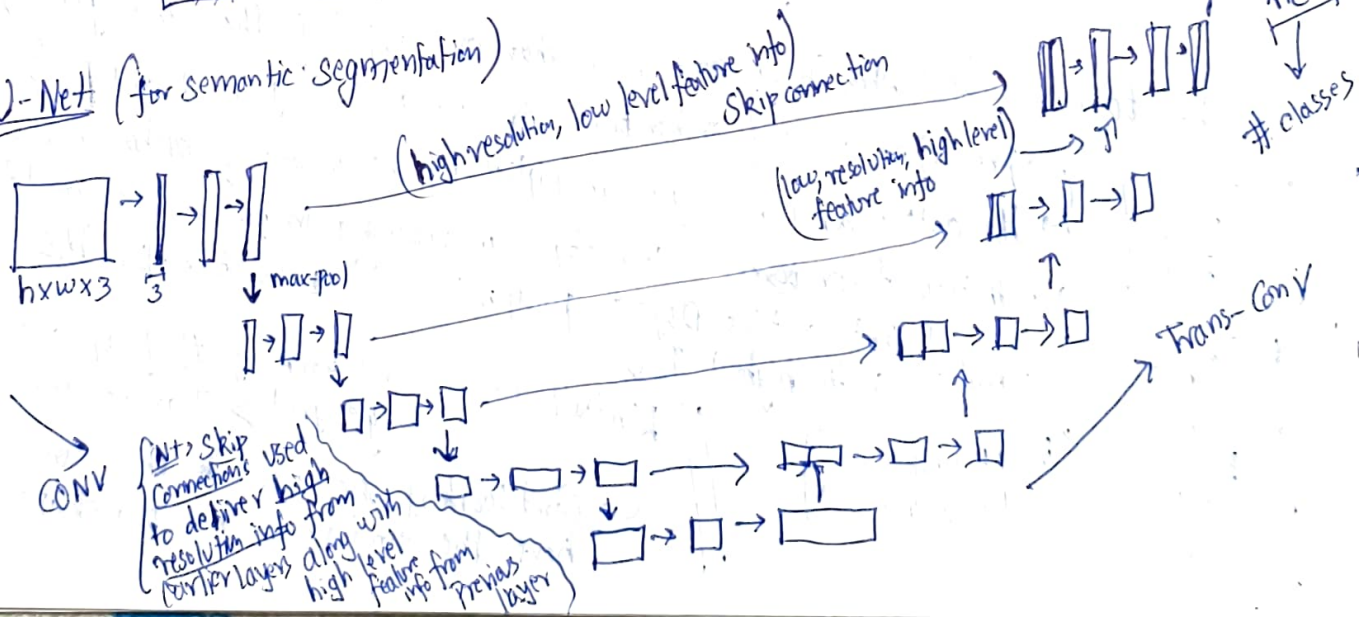


Keep multiplying each value in the input with the filter values and placing the multiplied values on top of the output

In case of overlapped boxes in the output, the values are just added



→ U-Net (for semantic segmentation)



→ Face verification
↳ input image, name/ID
↳ output whether the input image is that of the claimed person

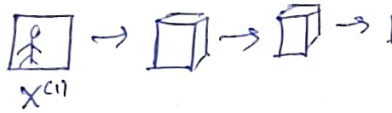
Face recognition
↳ has a database of K persons
↳ get an image as input
↳ output id if the image is any of the K persons.

→ One-shot learning :→ learning from one example

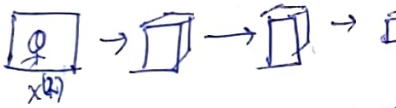
→ Problem with face recognition :→ it has to be one-shot, also can't use a softmax layer as output as each time a new person is added to the database, the model have to be retrained.

↳ Learning a similarity function ↳ $d(img1, img2)$ = degree of difference b/w images
↳ if $d(img1, img2) \leq \tau$:→ "same-person"
$> \tau$:→ "different-person"

→ Siamese network



:→ (encoding of $x^{(1)}$)



:→ (encoding of $x^{(2)}$)

{ Same neural net }

↳ Similarity function :→ $d(x^{(1)}, x^{(2)}) = \| f(x^{(1)}) - f(x^{(2)}) \|_2^2$

↳ Goal of learning in Siamese network :→ learn parameters so that : ↳ $d(x^{(i)}, x^{(j)})$ is small for same person's images
↳ $d(x^{(i)}, x^{(j)})$ is large for different person's images

(Parameters of NN here define an encoding $f(x^{(i)})$)

→ Triplet loss : → three images compared at a time :→
Anchor image (A) → (Compared with) → Positive image (P) { same person example }
↳ (Compared with) → Negative image (N) { different person example }

↳ we want :→ $\| f(A) - f(P) \|^2 - \| f(A) - f(N) \|^2 + \alpha \leq 0$ → margin { to ensure a distance b/w positive different neg diff }

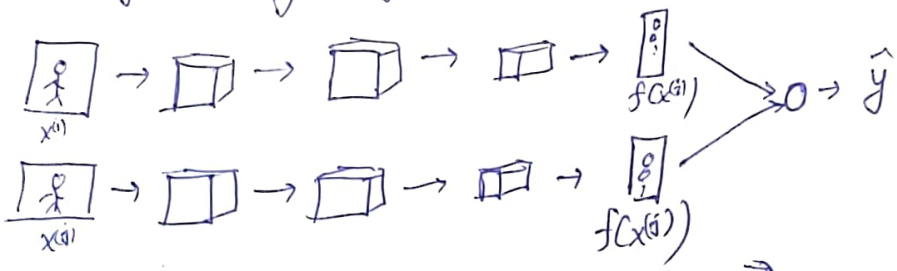
Triplet
↳ Loss function → $\mathcal{L}(A, P, N) = \max(\| f(A) - f(P) \|^2 - \| f(A) - f(N) \|^2 + \alpha, 0)$

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

Note :→ for training with triple loss, we require multiple images of the same person. once the training of the encoder is done, we can try out the one-shot learning.

↳ Choosing the triplets A, P, N :→ if A, P, N are choosen randomly then $d(A,P) + \alpha \leq d(A,N)$ is easily satisfied.
↳ choose triplets that are hard to train.
(take triplets where $d(A,P) \approx d(A,N)$)

→ Face recognition using binary classification

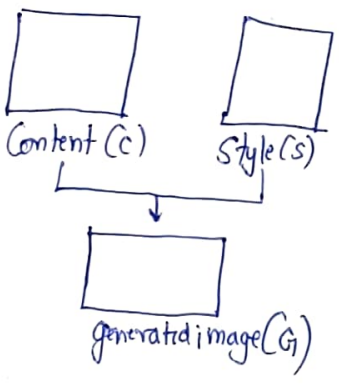


$$\hat{y} = \sigma\left[\sum_{k=1}^{128} w_k \left| f(x^{(i)})_k - f(x^{(j)})_k \right| + b\right]$$

↳ Other variations: χ^2

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

→ Neural style Transfer



Cost function

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

↳ finding the generated image (G)
↳ (1) initialize the G randomly
G : 100 x 100 x 3
↳ (2) use gradient descent to minimize $J(G)$

→ Content cost function
↳ say you use hidden layer l to compute content cost. (usually a layer near the middle)
↳ use pretrained ConvNet to pass the content (C) and generated (G) images.
↳ let $a^{[l](C)}$ and $a^{[l](G)}$ be activations from layer l on C and G images

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

→ Style cost function :-

↳ measure of style for a particular layer l of CNN :- how much correlations there is present between the different channels (having different features) ie which feature appears along with which other features. (susceptibility / not appears)

Style matrix

$$G^{[l]}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l]}_{ijk}\, a^{[l]}_{ijk'}$$

$a^{[l]}_{ijk}$ = activation at (i, j, k) (height, width, channel)

$G^{[l]} \in \mathbb{R}^{n_c^{[l]} \times n_c^{[l]}}$

$G^{[l](S)}_{kk'}$:- for style image // $G^{[l](G)}_{kk'}$:- for generated image

↳ called "gram" matrix in linear algebra.

↳ Style cost function :-

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

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

↳ Gives better results if we use different layers for style cost function :-

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

★ Note → 2D-CONV Nets can be easily generalized to 1D-CONV Nets and 3D-CONV Nets by taking 1-dimensional and 3-dimentional filters respectively.

→ Notations in RNN

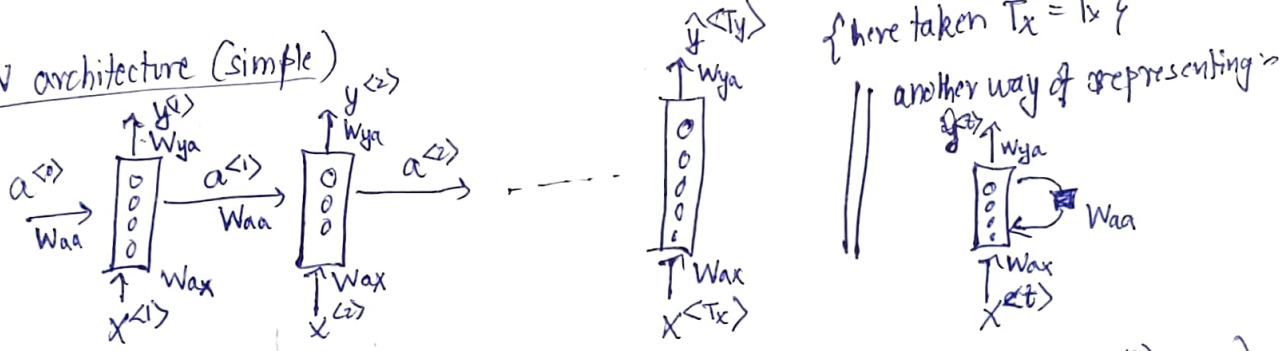
input sequence: X: Harry Potter and Hermione Granger invented a new spell.
(T_x: total length of the X)
$x^{<1>}$ $x^{<2>}$ ---- $x^{<7>}$ ---- $x^{<T_x>}$

output sequence: Y: 1 1 0 1 1 0 0 0 0
$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ --- $y^{<7>}$ --- $y^{<T_x>}$

↳ the words are given as input as a one-hot encoding of a vocabulary of > 10000 words or so. This means each input word is a >10,000 dimensional vector with 1 at only one place and 0 everywhere else.

↳ Why not a standard ANN: → (1) Input and outputs can be of different lengths
(2) doesnot share features learned across different positions of text

→ RNN architecture (simple)



$a^{<0>} = \vec{0}$

$a^{<1>} = g_1(W_{aa} a^{<0>} + W_{ax} x^{<1>} + b_a)$

$\hat{y}^{<1>} = g_2(W_{ya} a^{<1>} + b_y)$

$a^{<t>} = g_1(W_{aa} a^{<t-1>} + W_{ax} x^{<t>} + b_a)$

$\hat{y}^{<t>} = g_2(W_{ya} a^{<t>} + b_y)$

↳ Common choice of activation here: tanh/relu

↳ Simplified RNN notation

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

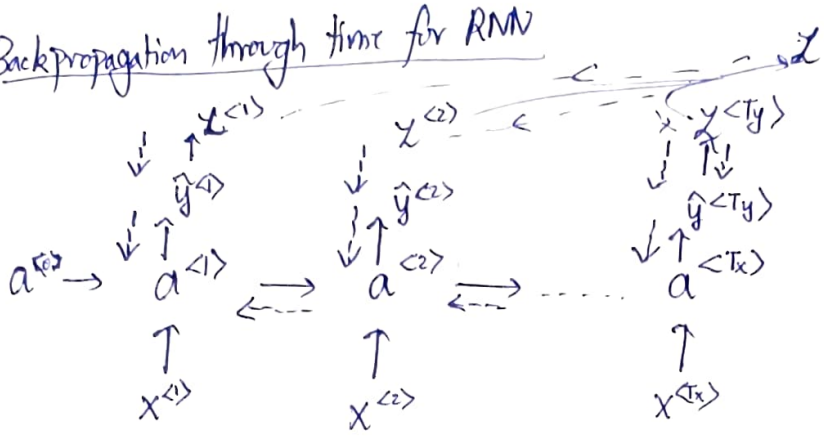
where, $W_a = [W_{aa} | W_{ax}]$

→ Loss at a particular time t

$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log \hat{y}^{<t>} - (1-y^{<t>}) \log(1-\hat{y}^{<t>})$

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

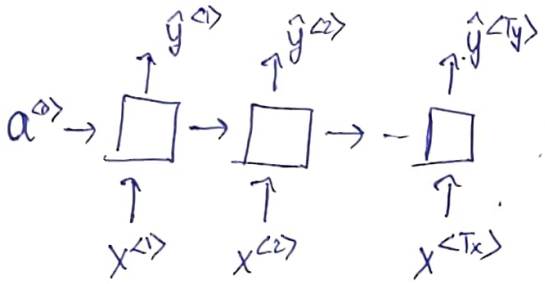
→ Backpropagation through time for RNN



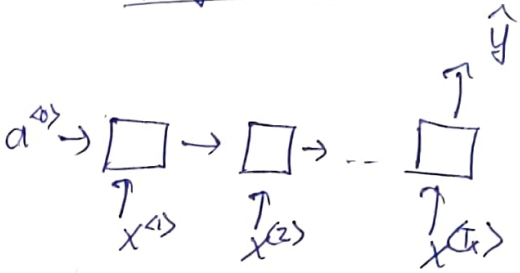
→ Forward prop
--> Backward prop

→ Examples of RNN architecture:-

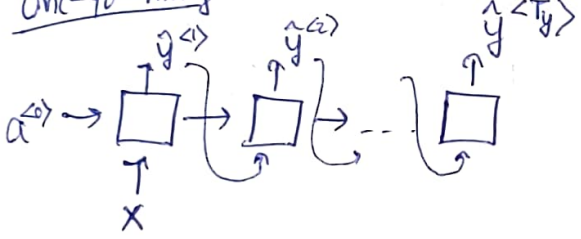
Many to many ($T_x = T_y$)



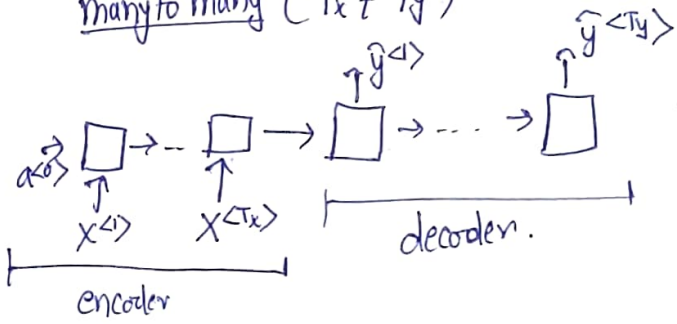
Many-to-one



One-to-many



many to many ($T_x \neq T_y$)



→ Language Modelling → given a sentence, predict its probability of occurence,

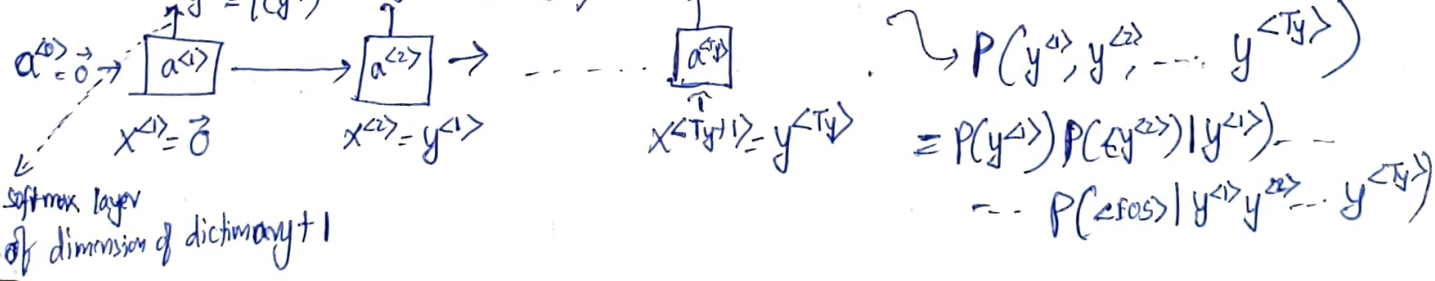
→ $P(\text{Sentence}) = \#$, $P(y^{<1>}, y^{<2>}, \dots y^{<Ty>}) = ?$

→ Before training, first need to tokenize the text.

for eg: Cats ($y^{<1>}$) averages ($y^{<2>}$) 15 hours of sleep a day ($y^{<8>}$). <EOS> ($y^{<9>}$)

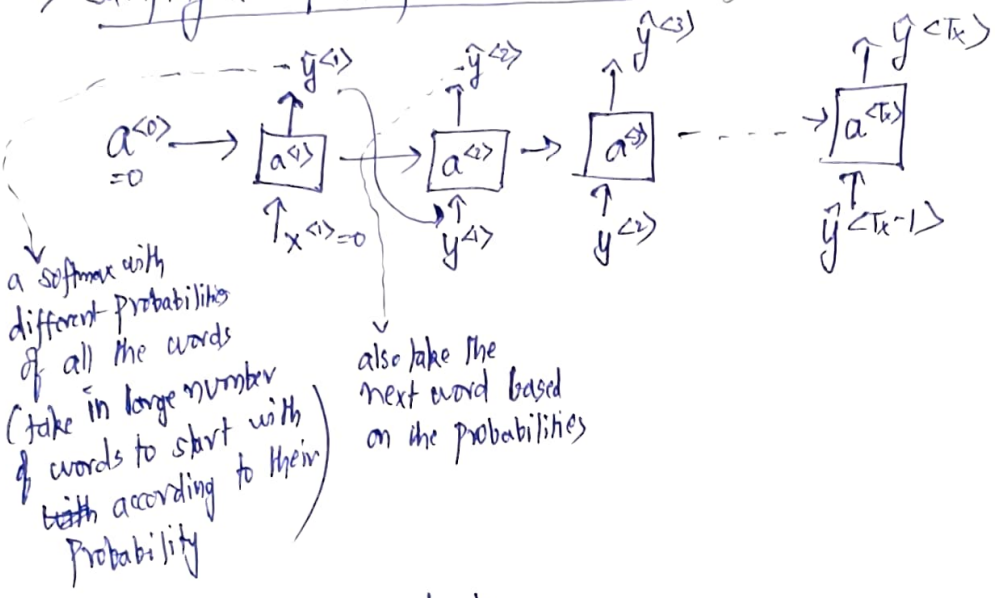
→ taken from a dictionary
→ if word not in dictionary ⇒ use <unk> (unknown) tag

RNN model



→ $P(y^{<1>}, y^{<2>}, \dots y^{<Ty>})$
$= P(y^{<1>}) P(y^{<2>}|y^{<1>}) \dots$
$\dots P(<EOS>|y^{<1>} y^{<2>} \dots y^{<Ty>})$

→ Sampling a sequence from a trained RNN for language model



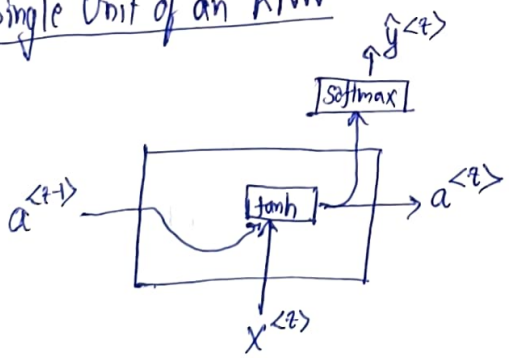
→ Problem of vanishing gradients

for eg. :- The cats, which were ---, were full
The cat, which was ---, was full
} long distance b/w the word (at(3) which directly affects word was/were

↳ Because long sequences in RNN ⇒ vanishing grandients can't be able to affect these far words relationships

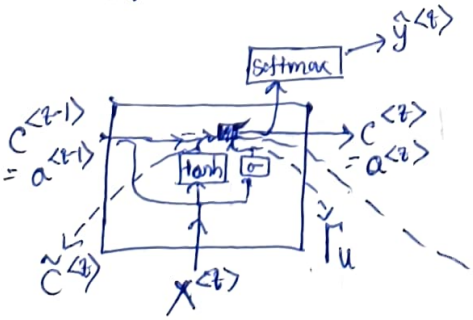
NI ⇒ Problem of Exploding gradients :- easier to handle, can be detected by NaN values of the gradients and can be handled by gradient clipping
↳ upper limit on gradient values

→ Single unit of an RNN



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

→ Simplified GRU (Gated Recurrent Unit)



c = memory cell (only change value when threshold ≈ 1 (gate))
↳ here $c^{<t>} = a^{<t>}$

$\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$ → alternate memory changed value

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

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

↳ So, if Γ_u is kept to be zero (or near to zero), then the value of $c^{<t>}$ is kept across many long distance (in time) to influence the values of farther outputs → helps to address the vanishing gradient problem
↳ Can learn long distance relationships

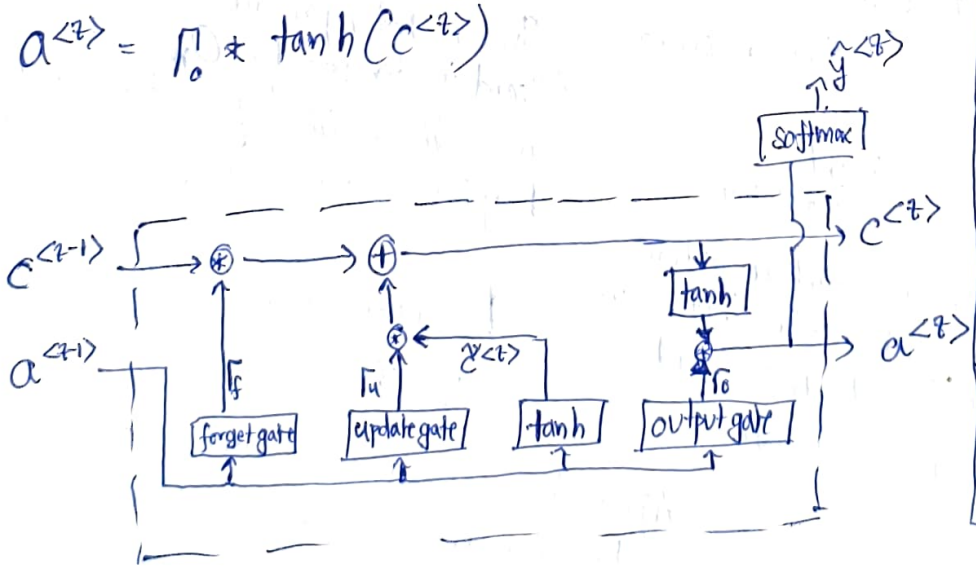
↳ Full GRU

$$\tilde{c}^{<t>} = \tanh\left(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c\right)$$
$$\Gamma_u = \sigma\left(W_c[c^{<t-1>}, x^{<t>}] + b_u\right)$$
$$\Gamma_r = \sigma\left(W_r[c^{<t-1>}, x^{<t>}] + b_r\right)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$$

Why Γ_r ? → relevance gate
↳ to map the relevance of $c^{<t-1>}$ in calculating $c^{<t>}$

→ LSTM (Long-Short Term Memory)

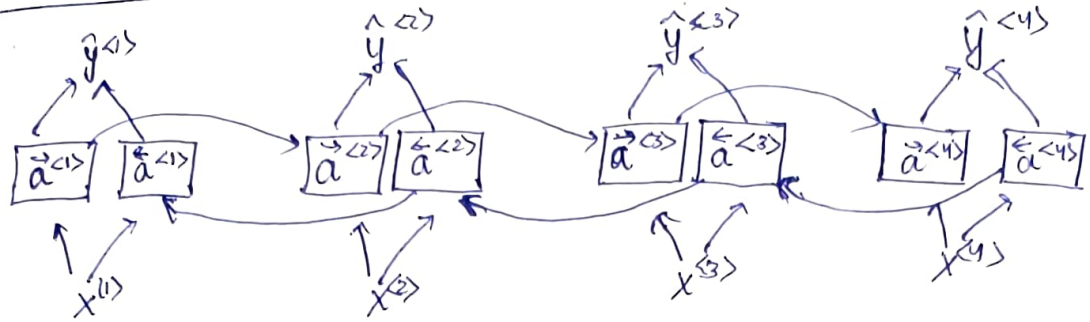
$$\tilde{c}^{<t>} = \tanh\left(W_c[a^{<t-1>}, x^{<t>}] + b_c\right) \quad \{ \text{here } c^{<t>} \neq a^{<t>}$$
$$\Gamma_u = \sigma\left(W_u[a^{<t-1>}, x^{<t>}] + b_u\right) \quad \{ \text{update gate}$$
$$\Gamma_f = \sigma\left(W_f[a^{<t-1>}, x^{<t>}] + b_f\right) \quad \{ \text{forget gate}$$
$$\Gamma_o = \sigma\left(W_o[a^{<t-1>}, x^{<t>}] + b_o\right) \quad \{ \text{output gate}$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh(c^{<t>})$$



Nt→ GRU's are more recent invention than LSTM
Nt→ GRU's simpler unit can allow to have larger RNNs
Nt→ but LSTM's have more power due to model complexity.

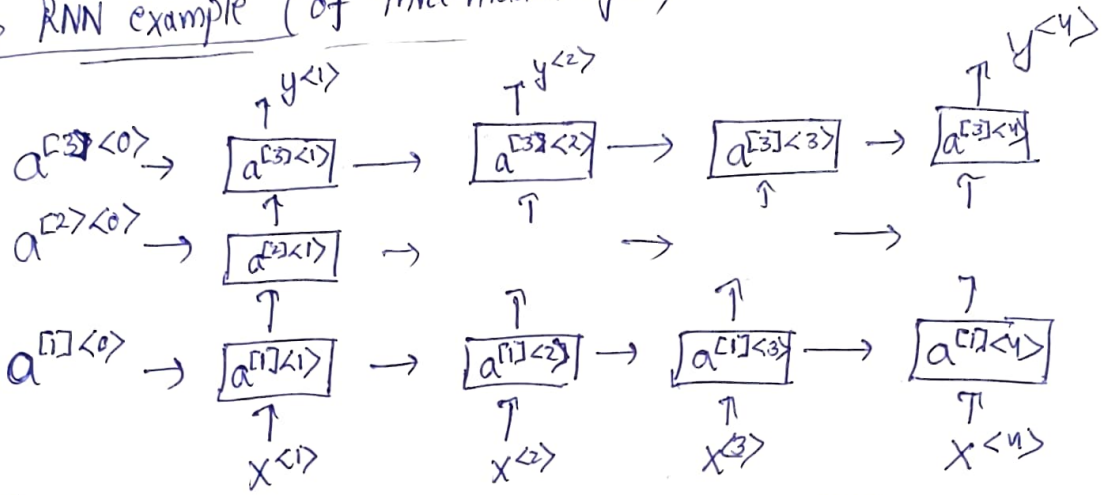
→ Bidirectional RNN

$$\hat{y}^{<t>} = g(W_y[\vec{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$$



Disadvantage: You do need the entire length of data for inference

→ Deep RNN example (of three hidden layers)



for eg → $a^{[2]<3>} = g[W_a^{[2]}[a^{[2]<2>}, a^{[1]<3>}] + b_a^{[2]}]$

NI → We don't see as many deep recurrent layer, but the output of the recurrent layers can be fed to deep ANN layer (not connected temporaly)

↳ In the one-hot representation of the words, all the words are having zero inner product with each other. That means, we donot have any relations between the individual words.

{ In one-hot representation, each word is denoted by O_n where n is the position of vector where 1 appears (zero everywhere else)

↳ Featurized Representation (Word embedding)

↳ each word is associated with a feature vector which gives values for different features related to it the word. for eg. gender, age, cost, etc.

{ Visualize word embeddings using t-SNE

{ Approach :→ Use a large unlabelled text to figure out the word embedding and use a smaller data corpus for the job in hand

→ Transfer learning and word embeddings

↳ learn word embedding from large text corpus. (1-100 B words)
↳ transfer embedding to new task with smaller training set
↳ Optional: continue to finetune the word embeddings with new data.

→ Analogies using word vectors

↳ to find out analogical word to the given word, given a prior analogy.
↳ Eg:- Man is to Woman as King is to ? → Ans:- Queen.
↳ So, here the differences in their corresponding word vectors must be similar to reflect same analogy. ie.

$$e_{man} - e_{woman} \approx e_{king} - e_{?w}$$

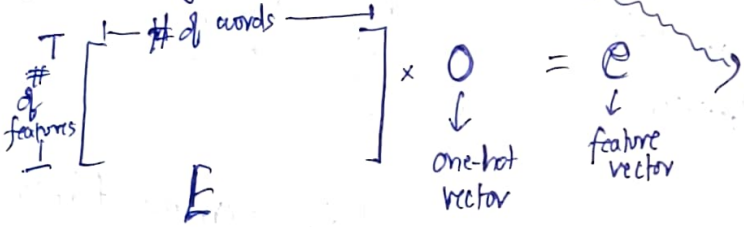
↳ find word w such that $\underset{w}{\text{argmax}} \left[\text{sim}(e_w, e_{king} - e_{man} + e_{woman}) \right]$

↳ Similarity function.

↳ Cosine similarity

$$\text{Sim}(u, v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$$

→ Embedding matrix (E)

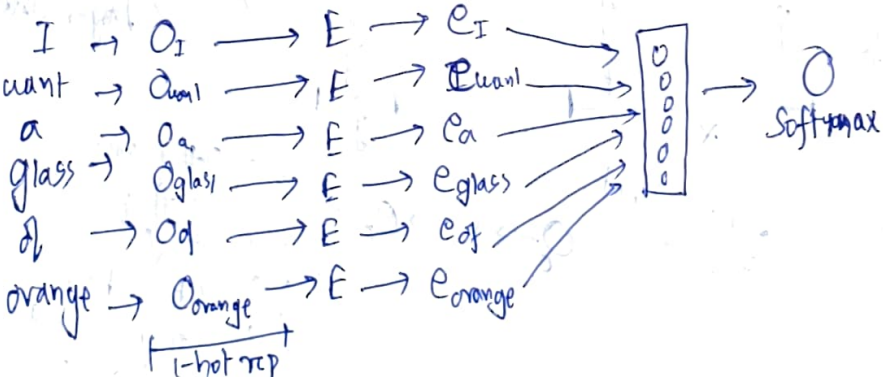


↳ To learn to get word-embeddings

↳ Neural language model to learn word embeddings

↳ learn word embedding by training a neural net to predict the next word
↳ given n- # of prior words.
↳ Eg:- I want a glass of orange juice

↳ here E is the embedding matrix whose parameters need to be learned.



→ Other Context/target pairs

Eg:- I want a glass of orange juice
Context → target

Other context
- four words on left and right
- last one word
- nearby one word.

(Word2Vec)

→ Skip-grams :- {given a context word in a sentence, the problem statement is to predict a randomly chosen Target word in the vicinity (say ±5 words) of the context word}

Model :-

$O_c \rightarrow E \rightarrow e_c \longrightarrow \text{softmax} \longrightarrow \hat{y}$

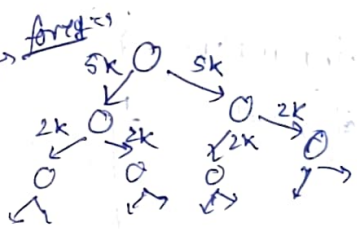
(O_c: 1-hot of context; E: embedding matrix; $\hat{y}$: target word t)

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

θ_t :- parameter associated with output t

loss-function :- $\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{10000} y_i \log(\hat{y}_i)$

→ Problem with Softmax classification
- large computational cost to calculate softmax
- use hierarchial softmax classifier



→ Negative Sampling {Problem statement :- given two words, need to determine if they are context-target words}

→ building the training set :-

	Context	word	target t
original context-target from text	orange	juice	1
random k words from dictionary	orange	king	0
	orange	the	0
	⋮	⋮	⋮
	c	t	y
	x		y

Choosing-k
- k = 5-20 (Smaller dataset)
- k = 2-5 (Larger dataset)

→ We will use binary logistic regression here

i.e. $P(y=1|c,t) = \sigma(\theta_t^T e_c)$ → neural net structure

$O_c \rightarrow E \rightarrow e_c$ → 10000 logistic regressors

- # of logistic regressors = # of words in dic
- but here we will only use k+1 number of regressors

→ Hence, in negative sampling, we are reducing the Softmax having 10000 (# of words in dictionary) different classes to 10000 different binary logistic regressors out of which we only need to train k+1 of them.

↳ Selecting negative samples: two extremes
- uniform random selection from the text corpus. (Cover representation of words like, 'the', 'of')
- uniform random selection from the dictionary (won't have distribution reflective of that in the real world)

a way out (heuristic)

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10000} f(w_j)^{3/4}}$$

$f(w_i)$: frequency of word w_i in the text corpus

→ GloVe (global vectors for word representation)

↳ define a way to get context and target pairs (c, t)

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

↳ GloVe :- take def of (c, t) as based on proximity (target word is in the context of c if it is in its proximity)

⇒ $X_{ij} = X_{ji}$

↳ Model :- need to minimize the function :- (let # of words in dictionary = n = 10000)

$$\sum_{i=1}^{10000} \sum_{j=1}^{10000} f(X_{ij}) \left[\theta_i^T e_j + b_i + b_j' - \log(X_{ij}) \right]^2$$

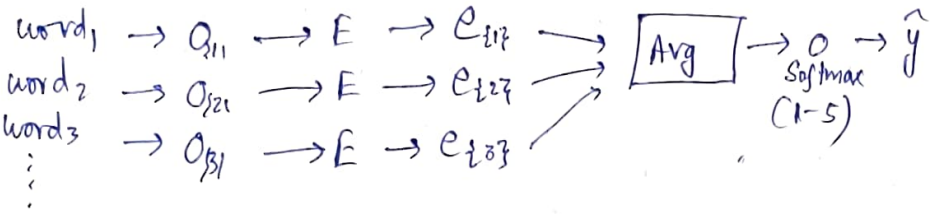
($\theta_i^T e_j$: dot product of target and context word vectors; θ_i: target (t), e_j: context (c), b_i: t, b_j': c)

↳ weighting term
- $f(X_{ij}) = 0$ if $X_{ij} = 0$ (as $\log(0) = n.d.$)
- Can also be taken (heuristically) such that it is avoided to give too much weightage to frequent words like this, is, a, of, etc and too little weightage to rare words.

→ Sentiment classification Problem

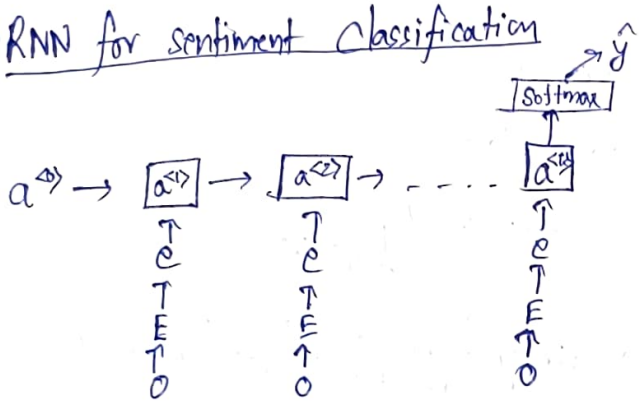
{ given a comment find the associated rating (5 stars/4 stars/...)
↳ data for training might be less here so word embedding might help }

↳ Simple sentiment classification model



{ limitation :- does not preserve word-ordering but preserves only word-meaning. So might not work in cases where ordering is important
(eg. "completely not good" +ve +ve) }

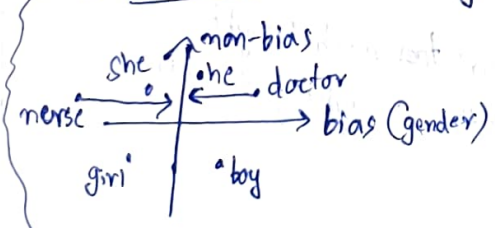
↳ RNN for sentiment classification



↳ Reducing bias in word embeddings { word embeddings can reflect gender, age, etc, biases of the text used to train the model }

↳ ① identify bias direction
eg. for gender bias take { $e_{he} - e_{she}$, $e_{male} - e_{female}$, ... } → Avg → bias direction
(definition based on gender and are separated by gender only)

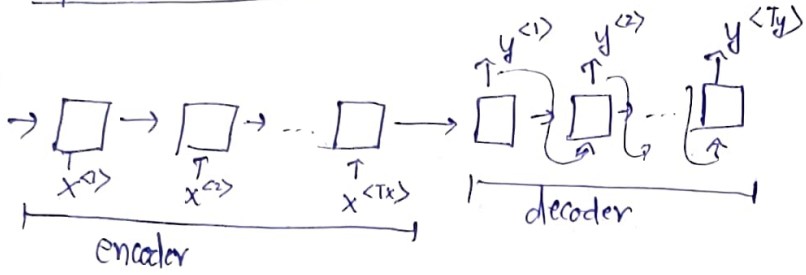
↳ ② Neutralize :- for every word that is not definitional, project to get rid of bias
(for eg. for gender, don't use words like grandpa, grandma and use words like doctors, scientist)



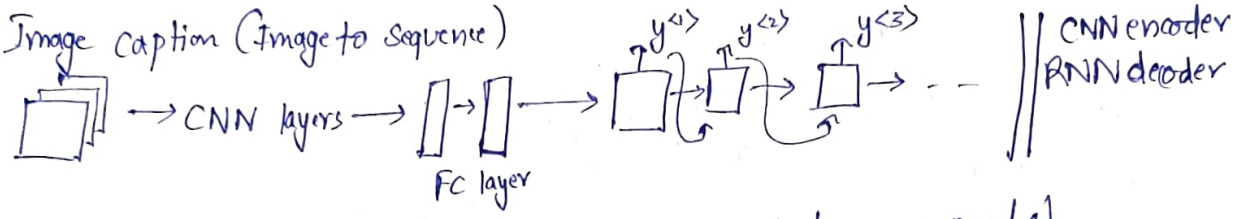
↳ ③ Equalize pairs :- ie make the pairs of words only separated by gender to be equidistant from non-gender words.

* To know which words should be passed through the neutralization step (ie which words are not gender specific) a linear classifier can be used.

→ Sequence to Sequence architechtures



↳ Image caption (Image to sequence)



↳ Machine ~~learning~~ translation as building a conditional language model

$P(y^{<1>} \dots y^{<Ty>} | x^{<1>} \dots x^{<Tx>})$ ← ↳ as we have to find $y^{<1>}, y^{<2>}, \dots y^{<Ty>}$ given $x^{<1>}, x^{<2>}, \dots x^{<Tx>}$

To find the most likely translation we find :→ $\underset{y^{<1>}, \dots y^{<Ty>}}{\arg\max}\ P(y^{<1>}, \dots y^{<Ty>} | x)$

* Why not greedy search ?

↳ because this doesnot optimize the probability of having the whole line towards the correct translation.

↳ because here unlike in language model where we get the most-probable sentence based on only the previous words, we want the most-probable language translation (the sentence as a whole)

→ ie. picking $y^{<i+1>}$ after picking $y^{<i>}$ and not picking all the $y^{<i>}$ simultaneous which ~~minimizes~~ $P(y|x)$

Nt→ greedy search used in language model but not here in machine translation.

Difficulty :→ very large number of probable sentences in the sample space

↳ (# of words in the dictionary) ^ (# of words in sentence)

↳ Beam Search

* Beam width :→ # of possibilities to consider to obtain the next word. (let = b)

↳ choosing 1st word :→ get the top ~~three~~ b # of possibilities based on $P(y^{<1>} | x)$ → # of probabilities to check = # of words in dictionary

↳ choosing 2nd word :→ get the top b # of possibilities based on $P(y^{<1>}, y^{<2>} | x)$ → # of probabilities to check = (# of words in dictionary) × b

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

↳ choosing i^{th} word :→ get the top b # of possibilities based on $P(y^{<1>} \dots y^{<i-1>} | x)$

($P(y^{<1>}, y^{<i>} | x) = P(y^{<1>} \dots y^{<i-1>} | x)\ P(y^{<i>} | x, y^{<1>}, \dots y^{<i-1>})$) → same here.

Nt→ b=1 ⇒ greedy search

→ Normalized log-likelihood objective (length normalization)

↳ Two problems while calculating $\arg\max_y \prod_{z=1}^{T_y} P(y^{<z>}|x, y^{<1>}, \dots y^{<z-1>})$

① as probabilities are pretty small (large # of vocabulary words) multiplying these probabilities can lead to very small #s, smaller than floating point precision of computer
↳ Remedy :→ take log-likelihood.

② as larger sentence ⇒ more # of multiplications ⇒ smaller probabilities
⇒ the algorithm supports shoter lenghts of output even though they might be wrong. ↳ Remedy :→ normalized log-likelihood objective

↳ $\arg\max_y \frac{1}{T_y^{\alpha}} \sum_{z=1}^{T_y} \log P(y^{<z>}|x, y^{<1>}, \dots y^{<z-1>})$ || α → hyper parameter $\in (0,1)$ {soft normalization}

* For beam search, calculate the normalized log-likelihood objective of all the cases found and take the one with the maximum.

→ Error Analysis for beam search

↳ For a machine translation, given x and its human translation y^* and its machine translation $\hat{y}$ how to know which is causing the error? (RNN or Beam search)

↳ Using the RNN, calculate $P(\hat{y}|x)$ and $P(y^*|x)$ {or calculate the corresponding log-likelihood objective}

↳ Case-1 : $P(y^*|x) > P(\hat{y}|x)$:→ Beam search is at fault
↳ beam search chose $\hat{y}$. But y^* attains higher $P(y|x)$

↳ Case-2 : $P(y^*|x) < P(\hat{y}|x)$:→ RNN is at fault

↳ look into all the mislabelled examples and get the fraction of errors due to beam search. If it is in majority, solve this error by increasing beam width.

→ Evaluating Machine Translation (Bleu Score) → (Bilingual Evaluation Understudy)

↳ Multiple different translations can be a good translation for a given text

→ The bleu score is calculated by comparing the machine translation (MT) output with all the reference outputs (more than one) made by humans for a given input.

bleu score on unigrams (one-word)

eg:
Ref-1:→ The cat is on the mat
Ref-2:→ There is a cat on the mat
MT:→ the the the the the the the

precision := $\frac{7}{7}$ = $\frac{\#\text{ of words in MT that appears in either ref}}{\#\text{ of words in MT}}$

(not a good measure)

↳ modified precision := $\frac{2}{7} = \frac{\sum_{w} Count_{clip}(w)}{\#\text{ of words in MT}}$ (summation over all the unique words in MT)

$Count_{clip}(w)$:= max count of word w in any of the reference

bleu score on bigrams (pairs-of-words)

eg:→ Ref:-1 and Ref:-> 2
MT: the cat the cat on the mat.

bigrams	Count	Count clip
the cat	2	1
cat the	1	0
cat on	1	1
on the	1	1
the mat	1	1

modified precision = $\frac{\sum Count_{clip}}{\sum Count}$ = 4/6

Bleu score on n-grams

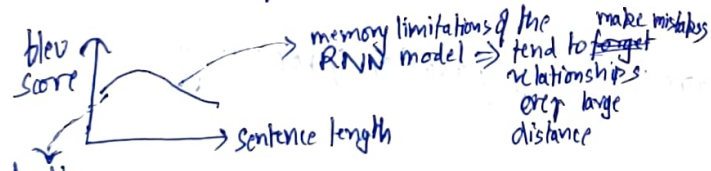
$$P_n = \frac{\sum_{n\text{-grams}\in\hat{y}} Count_{clip}(n\text{-gram})}{\sum_{n\text{-grams}\in\hat{y}} Count(n\text{-gram})}$$

→ Combined Bleu score :- $\boxed{BP \exp\left[\frac{1}{k}\sum_{n=1}^{k} P_n\right]}$

↳ Brevity Penalty (BP) ~~~→ used to penalize shorter length MT output which may tend to have larger bleu score as they might be having all the words already in reference. but the translation overall might be very wrong.

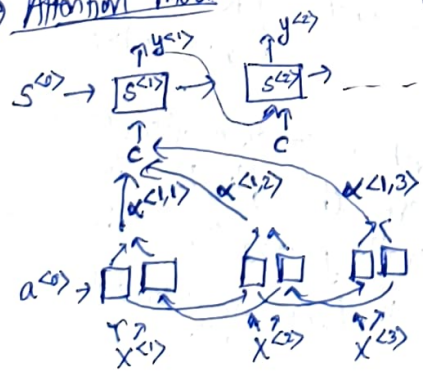
$$= \begin{cases} 1, & MT\_output\_length > ref\_output\_length \\ \exp\left(1 - \frac{MT\_output\_length}{ref\_output\_length}\right) \end{cases}$$

→ The problem of long sequences



→ Solved by attention model

→ Attention model :→ to make the model pay attention to only a part of the input sequence to generate the output.



→ how much attention the word j requires to derive word i in the output → depends on previous $s^{<t-1>}$ and $\vec{a}^{<t>}, \overleftarrow{a}^{<t>} \in a^{<t>}$

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

↳ attention and context :→ $\sum_{t'} \alpha^{<t,t'>} = 1$ (all the attentions are in $[0,1)$)

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

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

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

* Softmax used here to ensure $\alpha^{<t,t'>} \in [0,1)$

→ $e^{<t,t'>}$ can be calculated using a small neural network.

$s^{<t-1>}$, $a^{<t'>}$ → [ ] → $e^{<t,t'>}$

Nt→ Attention models have quadratic time complexity → $O(T_x T_y)$

→ Speech Recognition

↳ $T_x >> T_y$ here, so need to condense the output to the original length

for eg:→ $a^{<0>}$ → $\hat{y}^{<1>}$, $\hat{y}^{<2>}$, ..., $\hat{y}^{<1000>}$; $x^{<1>}$, $x^{<2>}$, ..., $x^{<1000>}$ → audio for "the quick brown fox"

CTC (Connectionist Temporal Classification)

→ $ttt\_h\_eee\_\_\_\sqcup\_\_\_qqq\_\_\_$

↳ space ; ↳ blank

Basic rule :→ Collapse repeated character not separated by blank.

→ Transformers

↳ Sequence to sequence models which are performing parallel processing.

like CNN models.

→ Self-Attention

$A(q,k,v)$ = attention-based vector representation of a word. | q → query, k → key, v → value

Transformer Attention

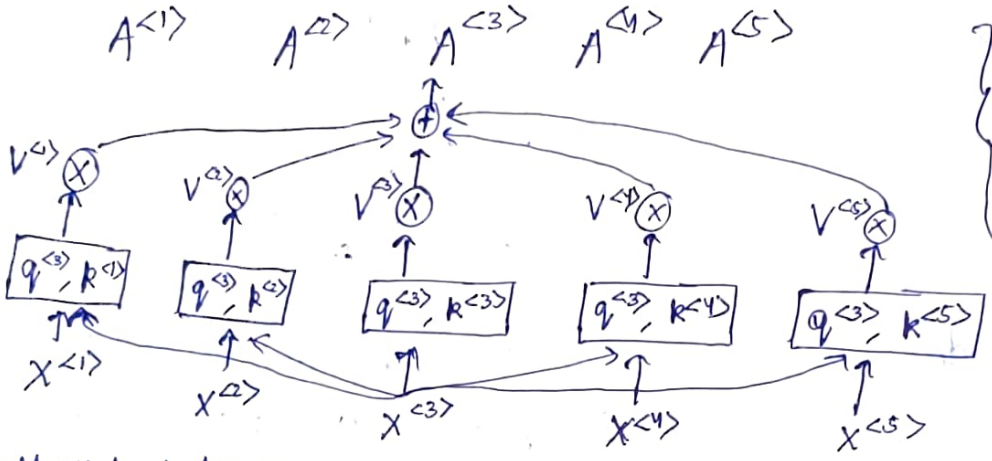
$$A(q,k,v) = \sum_i \frac{\exp(q \cdot k^{<i>})}{\sum_j \exp(q \cdot k^{<j>})} v^{<i>}$$

→ Similar to RNN attention discussed above but here the attention is calculated parallelly for all the words whereas RNN attentions are calculated one after another.

→ here, $q^{<i>} = W^q \cdot x^{<i>}$ → query to enquire about another input
$k^{<i>} = W^k \cdot x^{<i>}$ → key related to the input vector
$v^{<i>} = W^v \cdot x^{<i>}$ → value related to the input vector

} query, key, value similar to database concepts
→ here W^q, W^k, W^v are learnable parameters.

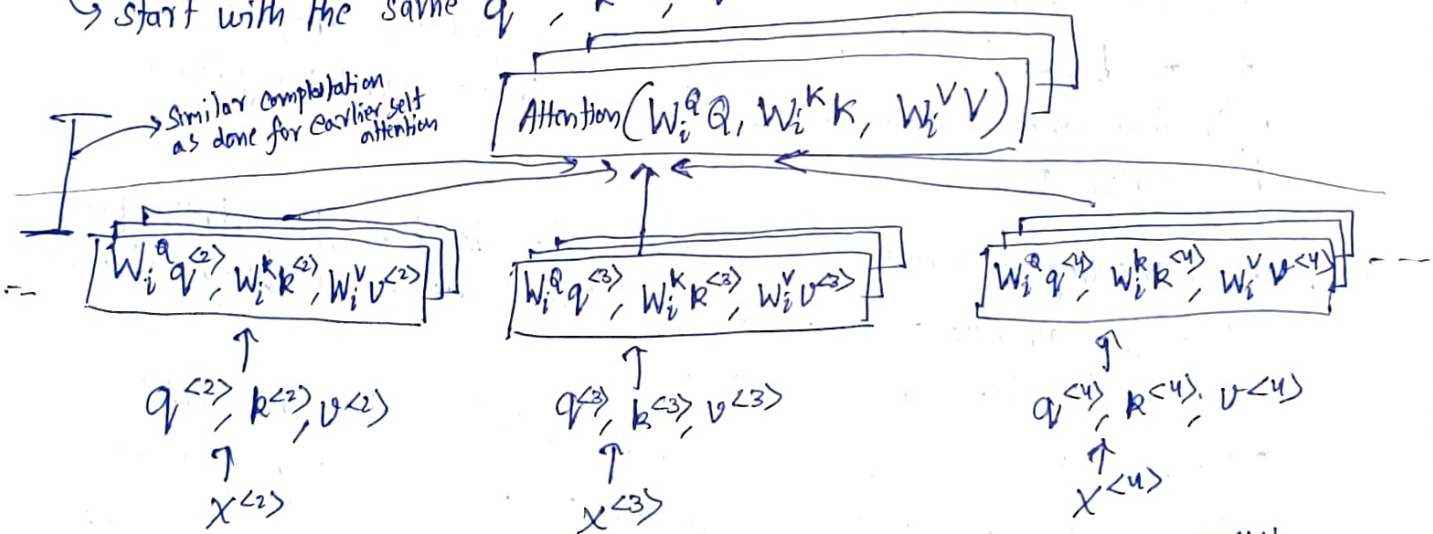
→ Self attention example:-



} gives a more advanced way of representing every word which also has appropriate attention from nearby words in constrast to word embeddings

→ Multi-head Attention

↳ the self-attention calculated earlier was single-head.
↳ let # of heads = h
↳ start with the same $q^{<i>}, k^{<i>}, v^{<i>}$

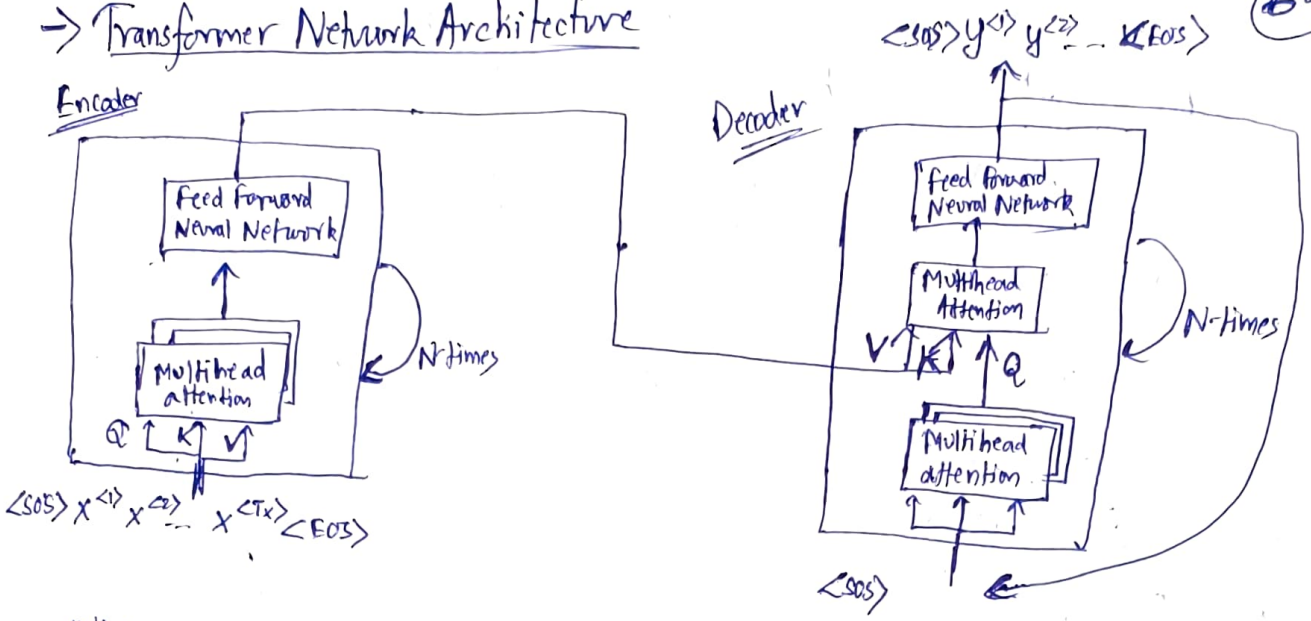


↳ The $\{\{W_i^q, W_i^k, W_i^v\}, i \in [1, h]\}$, each head ($i=i$) try to define a new feature to the self-attention connection { for eg ⇒ if $x^{<3>}$ → Africa
then $W^{Q,K,V}$ → where in africa
$W_2^{Q,K,V}$ → when in africa, etc... }

→ $\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots \text{head}_h) W_o$

$\text{head}_i = \text{Attention}(W_i^q Q, W_i^k K, W_i^v V)$

→ Transformer Network Architecture



NI→ The input to the decoder block starts with <sos> and then the next input is ~~generated~~ taken from the output of the decoder block in the previous run.

NI→ the K and V passed from the encoder contains the info for each input word in context of different features. The Q generated from each of synthesized word till now. So, the 2nd multihead attention in the decoder block tries to query (related to each synthesized output word) information and connections related to each input word using thier key, value pair.

NI→ Positional encoding :→ while calculating the self attention, the relative position of the queried word to the word whose attention is being calculated, is not taken into consideration.

↳ To take positions of the words (relative) into consideration while calculating self attention, a unique positional encoding is added to each $x^{<t>}$, $y^{<t>}$ given as input to both of the encoders and decoder.

↳ let $x^{<t>}$ be d-dimensional $x^{<t>} = [\underset{i=0}{\_}, \underset{i=1}{\_}, \underset{i=2}{\_}, \ldots \underset{i=d-1}{\_}]$

↳ Correspondind d-dimensional positional encoding is {pos = position of the word}

$$PE_{(pos,2i)} = \sin\left[\frac{pos}{(10000)^{2i/d}}\right]$$

$$PE_{(pos,2i+1)} = \cos\left[\frac{pos}{(10000)^{2i/d}}\right]$$