4/01/20 Title Deep learning

MLP Basics - Chapter 1

I) ● Why do we need non-linear?

→ Complex decision boundary

→ Single neuron can only compute a linear function even if non linearity activation is used. We need nets to get complete non linear decision boundary

→ Bias Term tells us how active neuron is w/o input

→ Loss function is the important one to optimize that helps to actually learn

→ Universal approximator tells us network has capacity to learn any function but doesn't tells us about how to learn them?

II) Hard tail Activation :- Piece wise linear → Take max of linear fns at that space.

● → w/o activation we just get linear units, all multiple linear units collapse to single layer neuron. But using linear units is used in model compression or matrix decomposition (SVD or PCA) → $1000 \times 1000 \Rightarrow 1000 \times 100 + 100 \times 1000$

III) Bias need :- Some systems need to output some defined value as default state. To keep deep default state of n/w. It is can be treated as another input hence could be moved

IV) MLP capacity :- Empirical Tuning, while compressing output to lower dim we tend to remove unnecessary variance like noise and include variance inherent to data's true nature

V) Softmax :- Represent probability distribution, exponential makes sure o/p is always +ve, Normalization applied to add up to one

→ It is like sigmoid with multiple classes. This is most like soft argmax rather than softmax. To get softmax we must multiply them with final output

VI) Loss function :- Maximum likelyhood :- Learn a model under which data is more likely : Maximize $P(Y=i|z)$. Use logarithm to avoid underflow/overflow issues. $\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$ (ith o/p)

→ Due to presence of log, softmax works well for -ve log likelyhood

Disclaimer :- Since its a student notes, there can be many errors. Please don't use notes as is

4/07/20 — Deep learning

MLP Basics — Chapter 1

I) Why do we need non-linear?

→ Complex decision boundary

→ Single neuron can only compute a linear function even if non linearity activation is used. Use neural nets to get complete non linear decision boundary

→ Bias Term Tells us how active neuron is w/o input

→ Loss function is the important one to optimize that helps to actually learn

→ Universal approximator Tells us network has capacity to learn any function but doesn't Tells us about how to learn them?

II) Max out Activation :- Piece wise linear → Take max of linear fnc at that space.

→ W/o activation we just get linear units, all multiple linear units collapses to single layer/neuron. But using linear units is used in model compression or matrix decomposition → SVD or PCA $1000 \times 1000 \Rightarrow 1000 \times 100 + 100 \times 1000$

III) Bias need :- Some systems need to output some defined value as default state. To keep default state of n/w. It is can be treated as another input hence could be moved

IV) MLP capacity : Empirical Tuning, while compressing output to lower dim we tend to remove unnecessary variance like noise and include variance inherent to data's true nature

V) Softmax : Represent probability distribution, exponential makes sure o/p is always +ve, Normalization applied to add up to one.

→ It is like sigmoid with multiple classes. This is most like soft argmax rather than softmax. To get softmax we must multiply them with final output

VI) Loss function :- Maximum likelyhood :- Learn a model under which data is more likely :- Maximize $P(Y=i|z)$. Use logarithm to avoid underflow/overflow issues. $\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$

ith o/p

→ Due to presence of log, softmax works well for -ve log likelyhood

→ Loss function like MSE doesn't work well with softmax as softmax squishes data, the gradients near 1 or 0 becomes less & less hence learning cannot happen

Which loss function To use?

i) Multi class classification :- cross entropy / negative log likelyhood + softmax

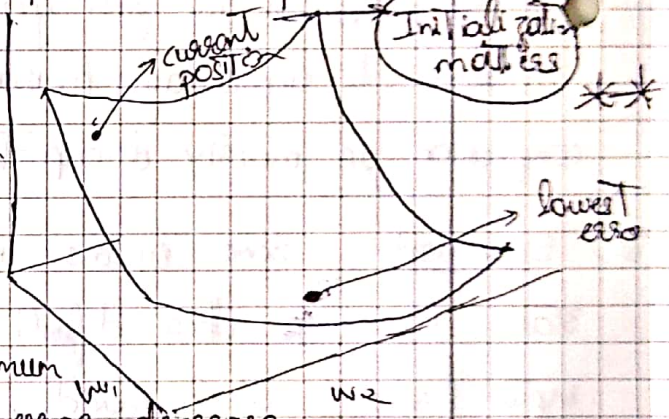
ii) Mean square error :- Linear o/p layer ← used for regression problem like predicting stock

Learning by Gradient descent :- Why do we need To compute gradients?

→ Because we don't have the info of network parameters v/s error plot beforehand, so that we can visually handpick best parameters

→ Knowing This before hand we need To plot for every combo of $w1, w2$ v/s error curve

→ For a network with millions of parameters This is not possible



→ Hence find gradient That Tells us maximum change of error. Traverse in direction where error decreases

** Problem :- If network has lots of parameters Then it becomes expensive To compute derivatives directly. Grad descent is a learning algorithm we use To optimize our network parameters.

→ Use backprop approach To help calculate gradients efficiently

grad descent v/s Backprop

$w_i = w_i - \alpha \Delta w_i$ → gradient descent algorithm

$\Delta w_i = \Delta w_j \cdot \Delta w_k \cdot \Delta o_n$ → backpropagation algorithm

→ Backprop is chain rule backwards. Use computation graph To do This in software

Types of Training

a) Batch Training :- Smooth descent, but can take really long for huge batch

→ Also complete data may not be fitting memory

→ The experience / loss of previous examples can help in learning next examples

b) Stochastic learning : faster learning, but can be a bit jumpy and can miss minima or stuck in local minima. Even noise (unnecessary) in data is captured. Cannot take advantage of GPU

c) Trade off - Minibatch Training :

Chapter-2

CNN - Block 1

Motivation : Recognize objects, audio independent from its position on image/spectrogram

Idea : Use shared weights → Reusing same weights applied at different positions

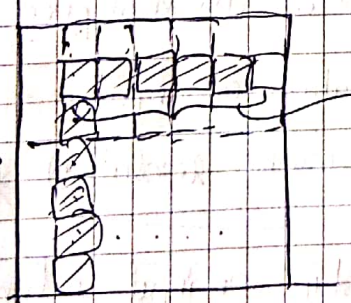
Implementation : Same filter kernel applied throughout image/spectrogram.

Activation map : Output after convolving filter with input. Same weights are applied throughout different places at image

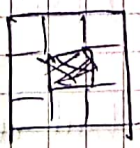
How it is learnt? Weights & Biases are learnt through gradient descent + backprop.

→ Each time filter is applied on different position of image we may receive different gradient for same parameter

→ So take all these gradients, average them up and then update the param in total per image



→ same cell in filter } Recieves 25 gradients as filter moves throughout image



→ Average all 25 of them & update

→ To apply convolution one must mostly do apply on linear space

Example : spectrogram [Frequency (log scale) v/s Time (linear scale)]

Properties of CNN

a) Local connectivity : Unlike dense fully connected n/w where every neuron is connected to all neurons from previous layer. If receptive field is not complete we say

For convolution n/w receptive field

To increase receptive field of CNN as we go deeper

b) Shared weights :- Unlike FC The same filter/param is used To apply To all position, Hence weights are shared

c) Translation equivariance :- (No rotation / scale / shear just shifting.) (invariant ✗)

→ if we move image To another position, convolution can work with almost same filters w/o much change $f(g(x)) = g(f(x))$

Filters and Activation :- Each filter when applied results in a single o/p channel

Multichannel computation :- Convolution only on axis That has semantics hence X, Y

This means we cannot convolve across channels as There is (no ordering) ✗

RGB = GRB = BGR = BRG

Filter output size :- Activation size decreases as convolved.

Padding :- To ensure output activation have desired weight as input.

a) Valid padding
Don't want To add info if it's not There

b) Same padding
Simplicity giving edge pixels ~~more~~ equal priority

c) Full adding
Convolution To text processing → Paying equal attention To all words

→ odd sized filter makes sense as center pixel can look at all local pixel

Striding :- Controls how often we apply convolution on input. How much conv you apply before moving again. (Downscale / Decrease resolution) ✱✱

Dialation :- We skip values like stride but also skip on input when applying
→ Improves receptive field w/o having filter nature huge.
→ Bigger filters are more specific whilst smaller filters are more general

Wavenet :- 16kHz per second is audio, Nothing much interesting happens at millisecond scale. Need a filter That has receptive field of 16k samples
→ We cannot use a big filter, hence use stacked small filters with dialation
→ Really (less parameters To train), and (resolution is not reduced) ✱✱

Pooling :- Differs from dialation and striding. Aim of pooling is not downsampling but it's a sideeffect of pooling when used along with strides.

→ In dialation, increase in receptive field but not reducing Total resolution but it's not The case in pooling

→ Approximate invariance To small Translations when combined with maxout

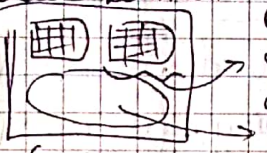
→ No trainable parameters

→ But loses some info about location of pixel, helpful to detect whether obj exists and not where?

→ Learn invariance wrt not big small rotation Translation

Even with slight rotation translation maxout can always take max pixel

Strong prior :- Conv filters are local, hence have a bias. This doesn't work well with Training Tasks where we need To pay attention arbitrarily as local connectivity might not help. This can cause overfitting, underfitting



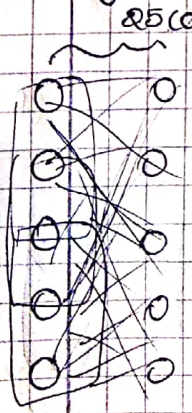
filter is local hence strong prior only to certain positions (non-zero values mostly) (all zeros no priors)

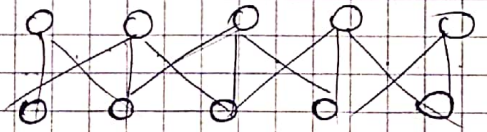
LeNet :- Input → Two Conv layers → Subsampling + bias → Convolution → subsampling + bias → output (26 × 1)

→ Bias parameter inside subsampler/pooling

→ pooling inside subsampler (strides = 2)

→ No flattening and dense layer at end but used a direct convolution. It's degraged as we use 4×4 filter alfabet characters spanning 2 channels. Eats up entire subsample feature map





How can we encourage learning complex filters?
encourage filter sparsity

→ Pooling is a cheap and fixed operation, That controls amount of invariance To input Translation

→ Striding is a costly operation compared To pooling which controls (non-linear subsampling rate) by Amount of skipping

CNN- block 2 (Important architectures only)

Alexnet

→ Distributed on Two GPUs, Two streams → One looking at Top part and another at bottom part of image. **

→ 11×11 filter size right at start → doesn't help in generalization can only learn specific patterns. Skipping lot of relevant info

→ stride of 4 → can lead to too much downsampling **

→ Cross linking of activations between Two streams of two gpus → lowers speed ** But important as one needs to have info on complete image to take decision

→ Usage of ReLU to boost Training

ZF net

→ Remove or keep filter size to 7×7 * to retain more info

→ How did they find it? They used deconvolutional layer ** to view input from activation. and also unpooling for maxpooling

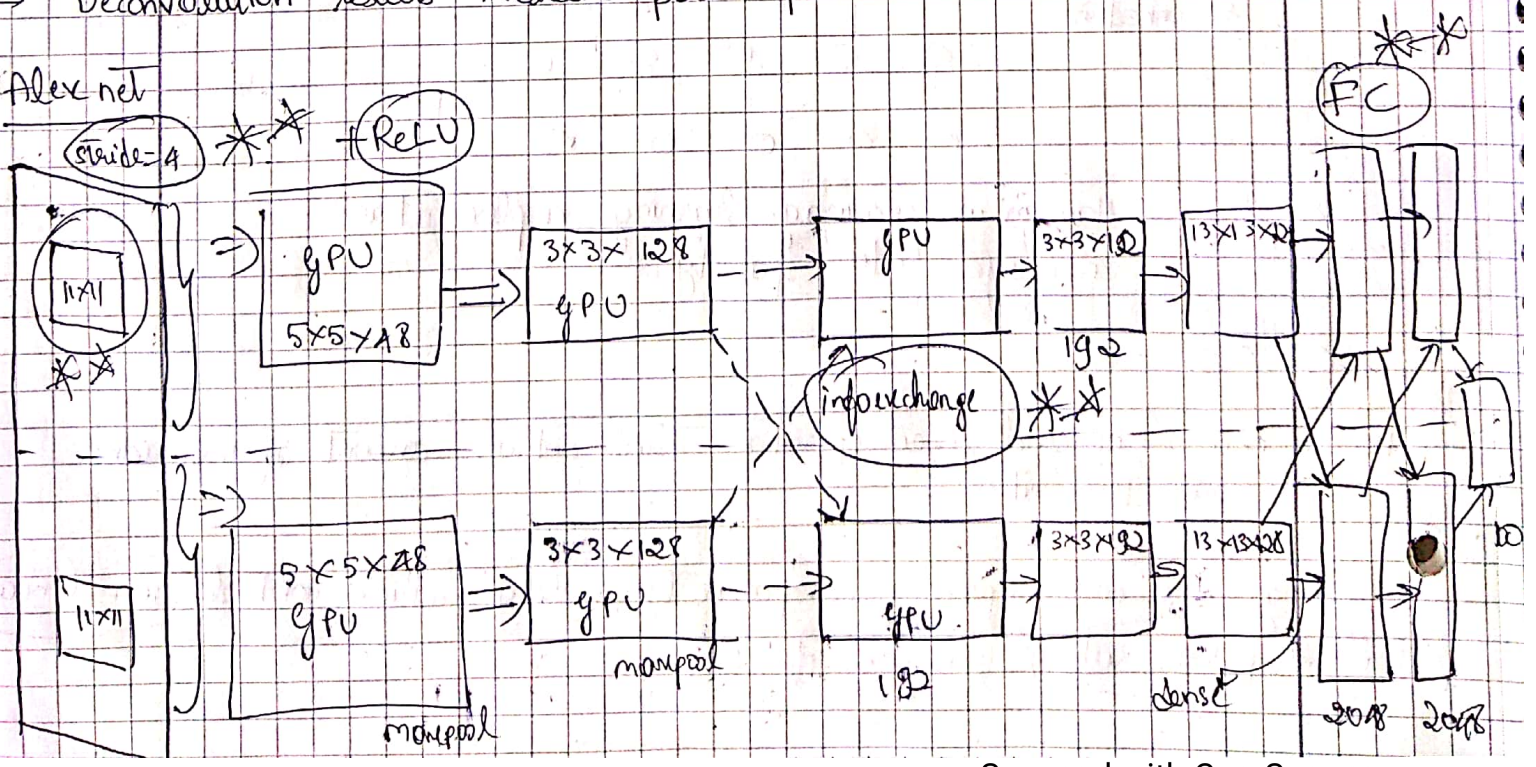
→ Using unpooling is difficult as maxpooling leads to loss of info, hence save the info of which position max element occured & use it. Other values are set to zero

→ Can see what do individual layers detect and recognize?

→ First effort to know how/what convnet learns

→ Deconvolution results in checker board patterns

Alex net



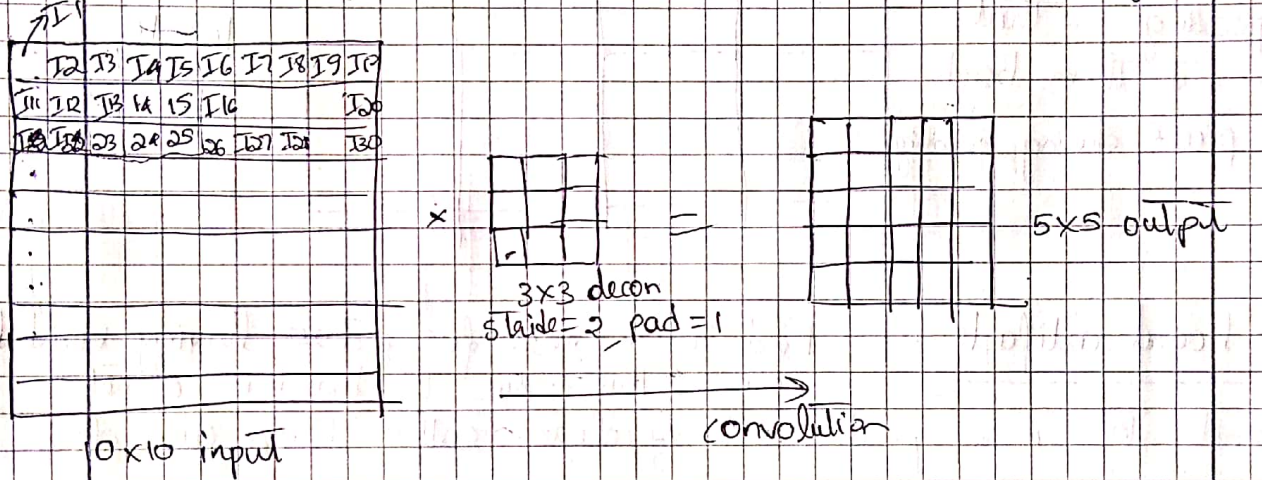
** Alexnet

→ Remember about 11x11 filter with 4 stride, Exchange of info b/w Two GPU, Max Pooling and ReLU

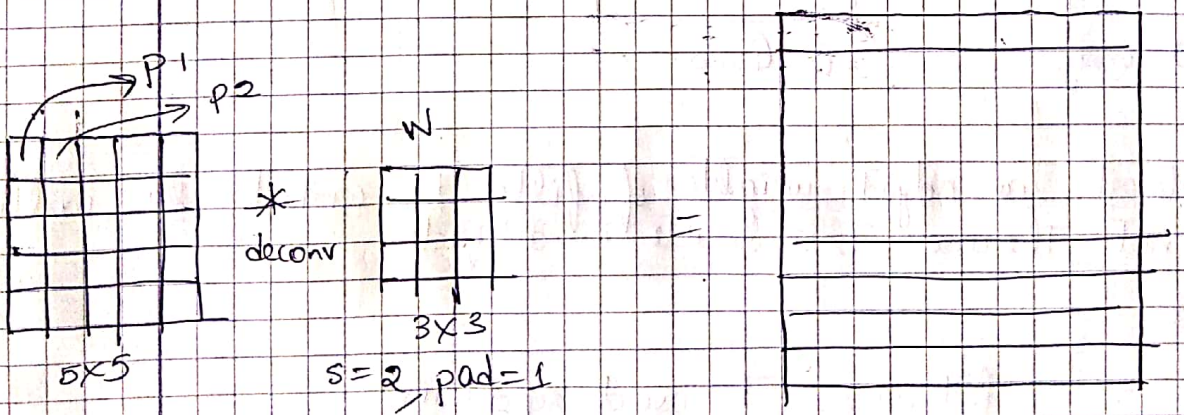
- ZFnet → Remember using 7x7 filters with stride = 2 to retain those info that was getting lost in alexnet. Deconvolution / unpooling layers were first introduced To visualize feature maps.

⊛ How does deconvolution work and would it result in info loss?

No, it doesn't end with information loss. It can be approximately reversed as convolution is an operation based on system of linear equations; ReLU too helps in this as there cannot be -ve pixel range.



formula $\left(\frac{n+2p-f}{s}\right)+1$ $= \text{floor}\left(\frac{10+2\times1-3}{2}\right)+1 = 5\times5$



$p1 = W * (I_1 \dots I_{23}) = w_1I_1 + w_2I_2 + \dots + w_9I_{23}$

$p2 = W * (I_3 \dots$

$w_1I_1 + w_2I_2 + w_3I_3 + w_4I_{11} + w_5I_{12} + w_6I_{13} + w_7I_{21} + w_8I_{22} + w_9I_{23}$

commonterms

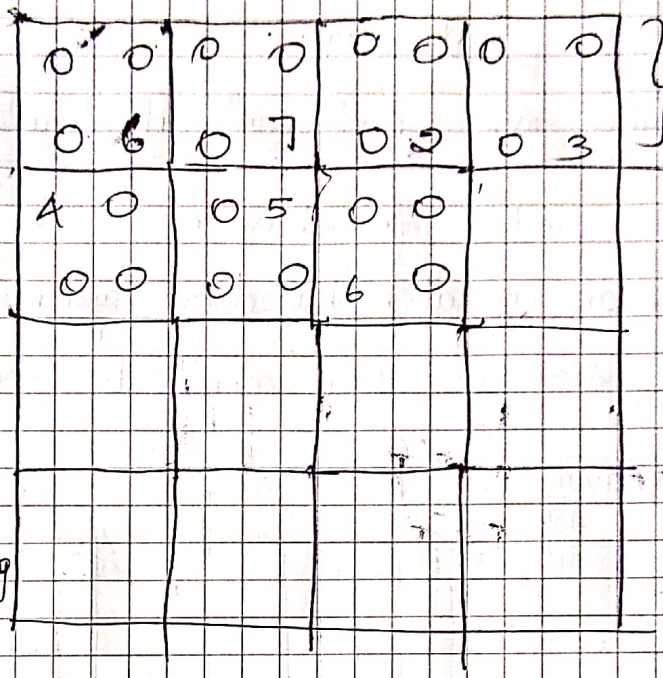
$p2 = w_1I_3 + w_2I_4 + w_3I_5 + w_4I_{13} + w_5I_{14} + \dots + w_7I_{23} + w_8I_{24} + w_9I_{25}$

→ As we can see that we have a system of linear eqn which can be solved to finally get $(I_1, I_2 \ldots I_{100})$ back

Unpooling : Causes info loss as same as pooling

2×2 pool

unpool →

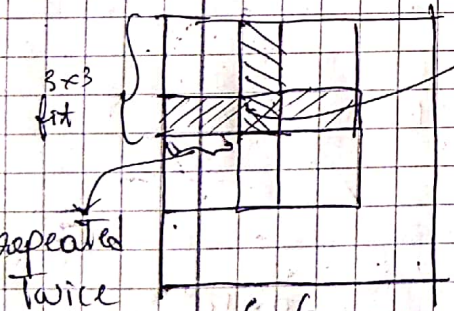


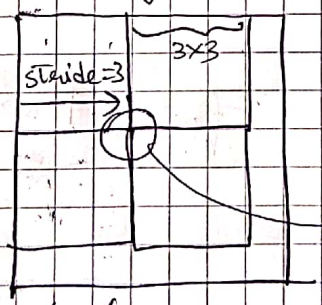
fill o There as 0 (info loss)

** → We would have saved position of max elements during pooling so that we can put them back To same place during unpooling

Checker board artifacts : Tend To be more for pictures having vivid bright colors. This is due To bias term of net.

Bias tend to more produce avg color rather than rare colors





stride size = 3 filter size = 3

no repetition

*** → Although network can adjust weights of filter to combat this multi pass (passes) this is often not the case. This results in artifacts

Solution : * *

i) Make stride size = filter size To avoid repetition

ii) Better upsampling Technique

U-net :- High level image → low res image → features → low res image
↓
high res image

→ Often times used for segmentation, where we predict each pixel or group of pixels. Basically finds segmentation mask

→ Pooling switches are not saved during up sampling as data at same levels are different

→ Activations at same level are concatenated This what gives segmentation

→ At the end have 2 channels → one neuron per pixel per channel (one for foreground and another for background)

→ What is a special architecture used for segmentation. semantics.

Object detection :- Put bounding box around required objects & classify them

Semantic segmentation :- Paint a specific class or classes pixel with single color ignore the rest (group of people)

Instance segmentation :- Paint individual person with different pixel color & ignore rest of classes

What different in this architecture? Before they were using sliding window techniques which are slower To perform semantic segmentation. This architecture changes the Tasks and Problems that paper is solving

→ Very few training examples in biomedical training

→ Usage of Data augmentation To create a generic model

→ Overlap tile strategy

→ Need for per pixel classification.

Idea

→ Downsample To extract features and remove unnecessary variance.

→ Upsample To get precise locations for these features in high res layers

→ Concatenating downsampled feature maps with corresponding upsampling layer To assemble a precise output

→ Overlap Tile strategy :- To predict pixels at border region, missing context is extrapolated by mirroring

→ Give more weight To background pixels in loss fnc That helps separating objects of same class close To each other.

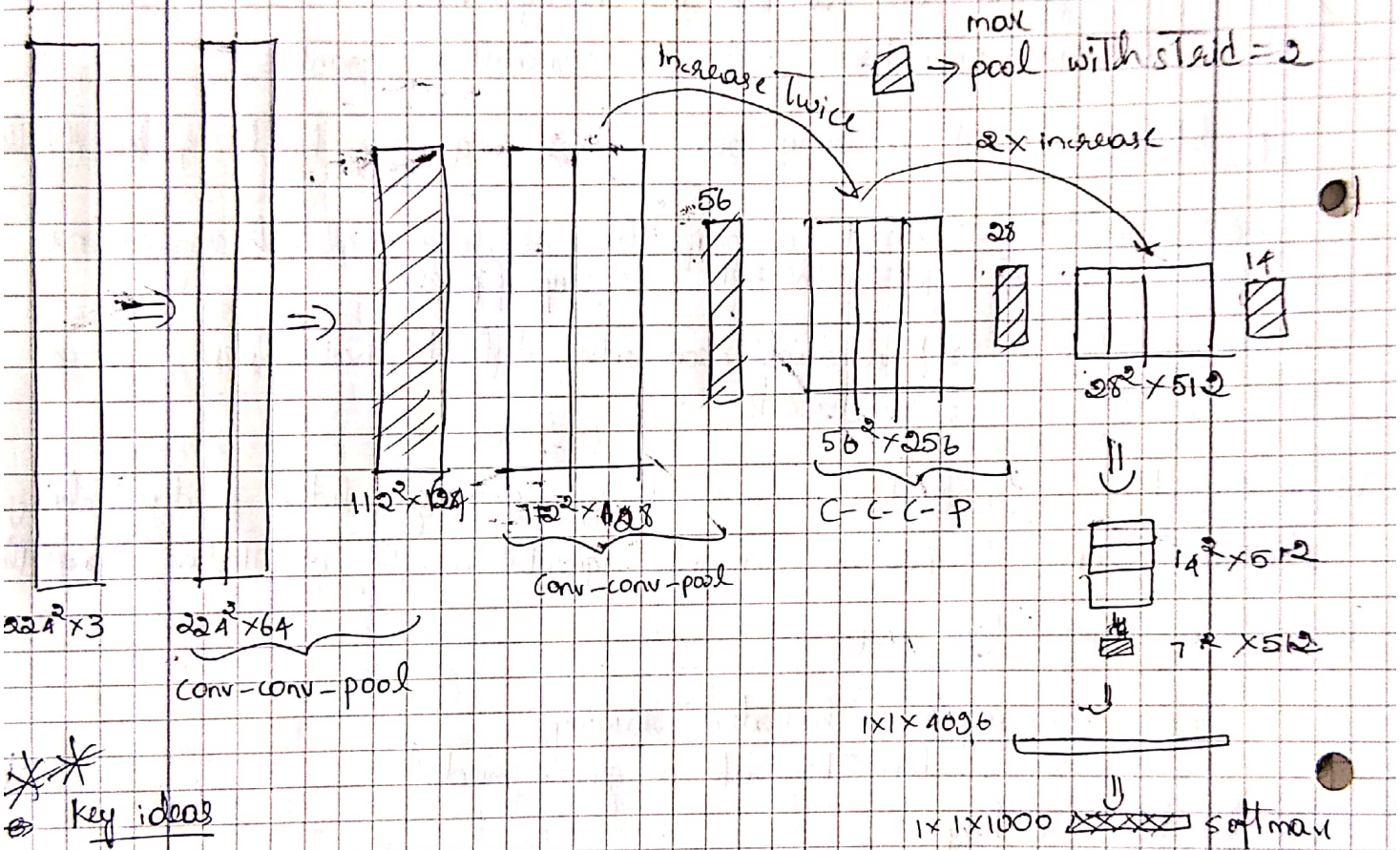
Probabilistic U-net for ambigous images

i) Image data annotated by medical students and inexperienced professionals differently

ii) Need for model which can handle borderline stuff

VGG NET (VGG-16) (2-M ; 2-M ; 3-M ; 3-M ; 3-M)

Design principle : Simplicity & depth



** Key ideas

→ Simple and elegant design

** Usage of more filters with lower dim size of 3x3 instead of 5x5 or 7x7 as neural network can essentially use Two 3x3 or Three 3x3 to makeup for 5x5 or 7x7 filters. This makes filters more discriminative

→ Usage of 1x1 convolution in some configurations to increase non-linearity

→ Elegant design by usage of (same padding) and double channel size after each maxpool downsampling

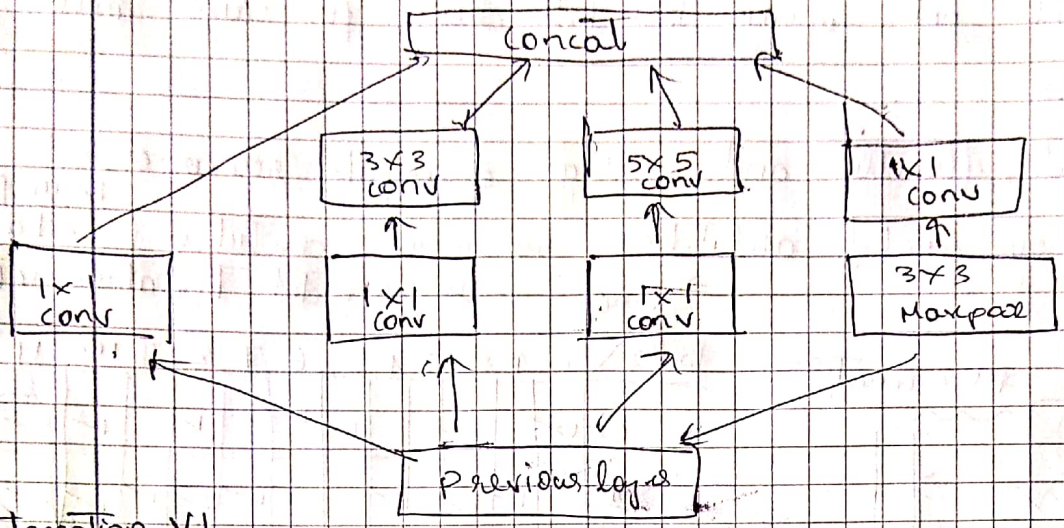
(same padding) → This preserves resolution and info among edge pixels

** → (lesser kernel size leads to less params even if no of filters are more)

Inception Module :- Apply convolutions of diffent sizes in ||el and concatenat

→ Due to concatenation, filters/channels may become too large a number.

→ Use 1×1 convolution as dimensional reduction to reduce no of channels



Inception V1

Googlenet :- 9 inception modules, lesser params in alexnet, use average pool at last layer instead of FC layer to reduce params

Inception v2 :- Adding batch norm. In intermediate layers more changes are present

→ What comes into hidden layer is a feature map of previous layer which are constantly changing. All inputs except i/p image changes. Hence learning is much harder. (Ex:- Input has pixel range of 0-255 but layer1 can have 0-500 & layer2 b/w 0-350 at each iteration. This may change)

→ Normalize intermediate inputs maps to make learning faster as feature maps will be in defined range

→ Use shifting factor to shift mean and to change variance use scaling factor

$$y_i = \gamma \hat{x}_i + \beta \quad (**) \qquad \hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad (\beta)$$

ϵ → small delta to avoid $\sigma_B = 0$

Inception v3 :-

→ Avoid representational bottlencks in early part of network

→ Higher dimensional representations are easier to process locally within n/w

→ Spatial aggregation can be done over lower dimensional embeddings w/o loss of representational power

→ Balance width and depth of n/w

Inception network : General Idea & problems it solving

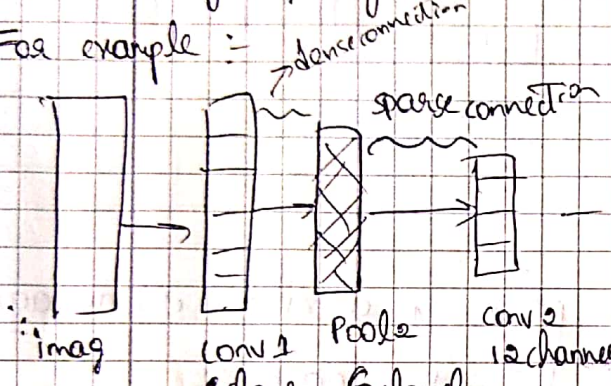
Problems it Tackling :- Inception net - Version 1

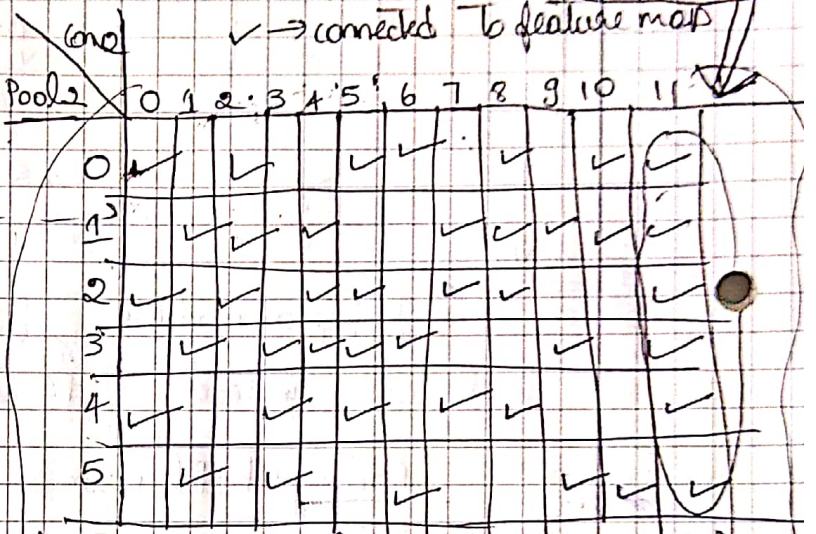
→ Network with too much capacity is prone to overfit (like vgg). Hence bigger n/w might not always be better choice for cases with less training data

→ Parameters not used efficiently and wastage of compute resources

Ideas :- Using sparsely connected architectures even inside convolution (See Lecun paper / See Lenet paper)

For example :-





** Motivation :-

→ As we can see that not every output of pool2 is utilized by conv2 and selection is mostly random (3, 4, all 6)

Why ? This helps to break symetry, if everything is connected to everything then features learnt by conv2 filters may look the same & this is not useful.

→ Breaking this symetry helps learn complement any patterns see for example (0, 6) and (1, 6) one may lear horizontal and another vertical patch

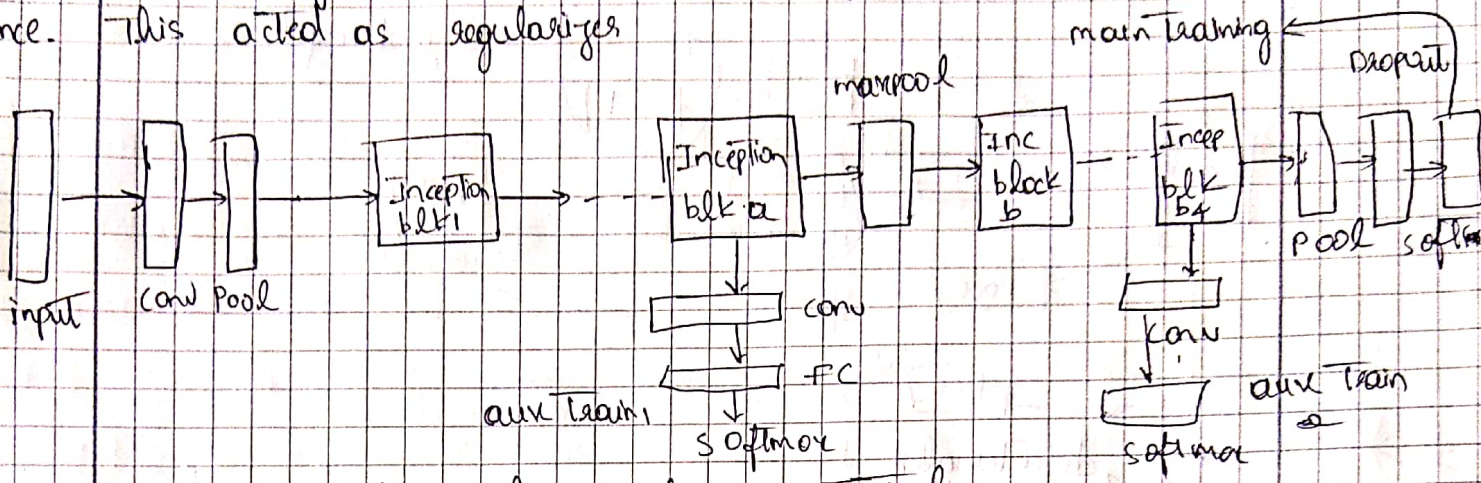
→ We can also combine all maps of previous pool layer and use it in one combo of (11) This provides an overview

Problems with this approach :- GPUs and Tensor libraries works well and optimized for dense matrix operations

Solutions in inception n/w :- Use combination of filters of multiple size and concatenate their output to mimic this sparse connectivity. This also takes care of hardware utilization problems as all operations are done on dense stuff.

→ Usage of 1×1 convolutions so that the paramsize won't blow up when feature maps are passed from one inception block to another. They also ** add additional non-linearity through ReLU

- **Training Inception net** :- After stacking inception modules they also had a couple of auxiliary outputs opened for training. This was removed during inference. This acted as regularizer



- → 9 Inception modules with 100 layers in total

Inception net V2 :-

→ 12x fewer parameters than alexnet

→ Use average pool at the end instead of an fully connected (FC) layer to keep no of total parameters down

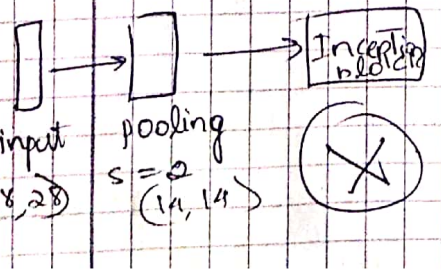
Inception net - V2 :- Solving the problems during training by using Batch normalization. This makes input of intermediate layers to be in pre-expected range of near (0, 1)

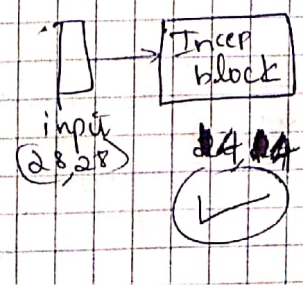
Inception net - V3 :- Problems it is solving

- No concrete explanations on general design rules of inception net - V1 was provided in earlier papers. In this we discuss on these principles and see how can we customize them to our needs

Design principles

1) Avoid representational bottlenecks in early layers :- Simply put it don't do too much pooling or any other info reduction where spatial info (28 x 28) is suddenly decreased or halved (14 x 14) in early layers. Give n/w time to learn filters on high dim structure and decrease gently.





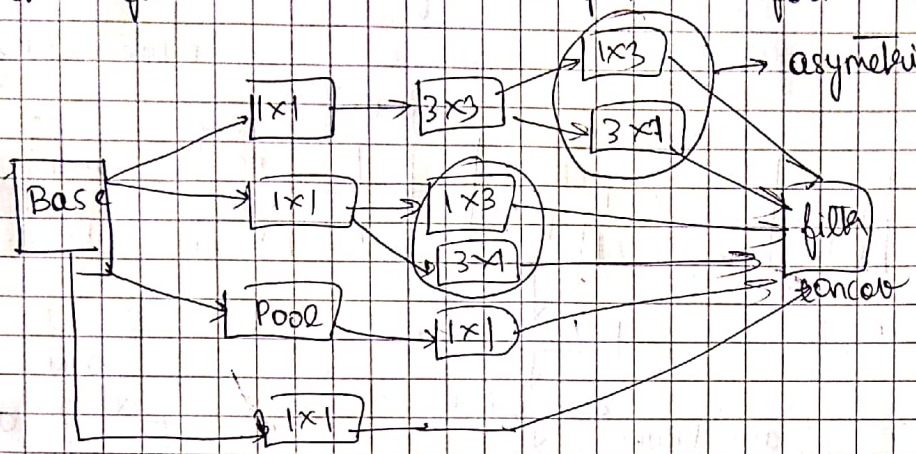
Representation reduced from stride of inception block where filters are captured and not pooling layer

2) Higher dimensional representations are easier to process locally within n/w :-

→ Simply put Make or process any spatial related info inside inception block and not outside ↗ n×n → (1×n)(n×1)

→ Use asymetric filters to make computation faster and eat less memory

**



**

→ Capturing more disentangled / diverse activations using diffrent filter structure leads to more discriminative featuremaps and distinct :- Helpful to add variety & learn

2×1 → 1×3 → detects and outputs feature map that is horizontal
1×2 → 3×1 → — // — — — — vertical

3×3 → 1×3 → 3×1 } Already 3×3 would have subsampled and lowered resolution so applying another set of filters would provide blurred image lower res featuremaps that may be helpful in some cases

3) Spatial aggregation can be done ~~with~~ over lower dim embeddings w/o much loss in representation power

→ Simply put using Two (3×3) filters instead of one (5×5) would not learn the process

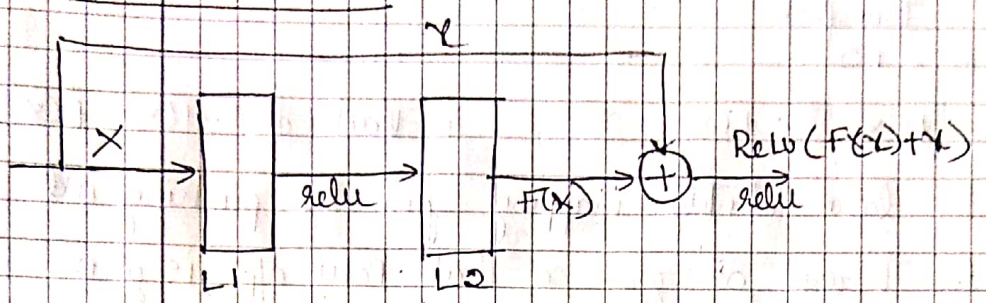
→ To take a step further apply general factorization and factorize 3×3 → (1×3) → (3×1) this reduces parameters drastically and doesn't impact learning

5×5 ⟹ 3×3 → 3×3

3×3 ⟹ 1×3 → 3×1

A) Balance depth and width of network : This is simply number of inception blocks, filter sizes and concatenated maps etc like hyper parameters should be choosen wisely

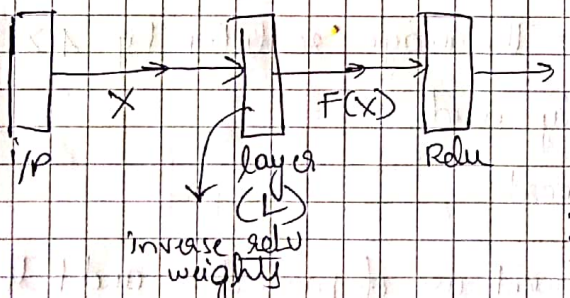
Residual networks



Problems it Tackling :- Surprisingly deeper networks that are too deep resulted in higher training error than shallow networks ? This was because in extreme deep networks, some interim layers may not learn anything new, but try to pass the info it got from previous layer un adulterated.

→ Here lies the problem, once non-linearity is applied in layer, it has to learn weights such that it reverses the non-linearity as we just need to pass the signal and not learn anything. This process of adjusting weights are difficult

For example :- lets say i want my final output to be just x i.e; g(x) = x



now i am trying to make x = g(x)

⇒ x = ReLU[F(x)]

⇒ F should be inverse relu

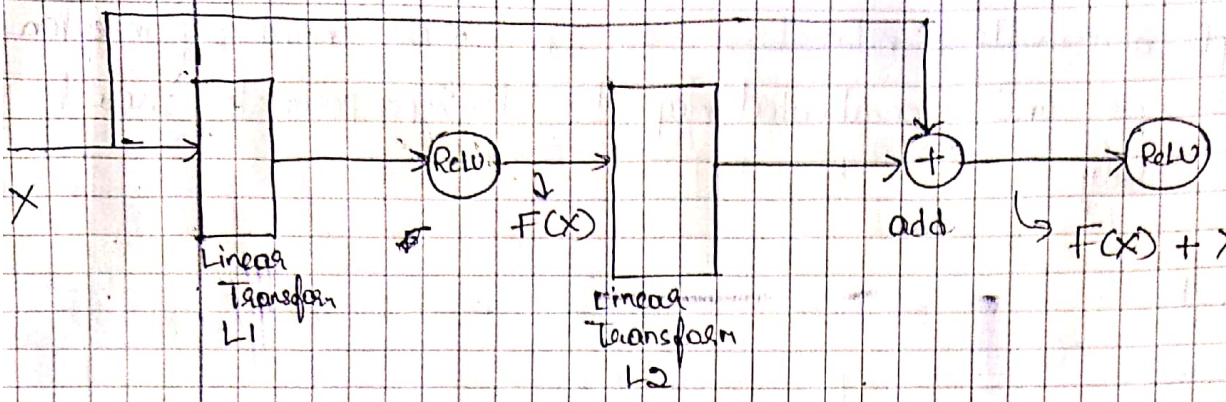
→ if layer (L) learns the weights such that it can inverse future ReLU operation then signal x would be passed as is w/o much changes

$$\Rightarrow x = \text{ReLU}[F(x)] = \text{ReLU}[\text{ReLU}^{-1}(x)] = x$$

→ Often in reality this is difficult and we need different structure

Solution by ResNet :- Reformulate the problem, instead of learning weights to make inverse relu, learn weights to approach a constant value so learning during these identity phases could be better.

→ Also provide an option to learn other complex fnc instead of identity when necessary



→ Learn residual function $H(X) = F(X) - X$ instead of $H(X) = F(X)$

This way, it is easier to learn identity mapping just by pushing all weights of L1 close to zero or zero so that ReLU o/p is zero hence final o/p is just X. In case more complex fnc needs to be learnt then learn different non-zero weights in residual layer "L1"

** → The above is easily realized by providing a short ckt connection from input to addition layer. As a sideeffect, this also improves backprop as there is no challenge/obstacle for gradients to travel in the short ckt path, hence even initial layers can be trained easily with less gradient vanishing problem

Typical practical Resnet

→ 152 layer deep

→ First two layers almost decrease the image resolution by 4X in total

→ 3×3 conv + 64 filters are normally used

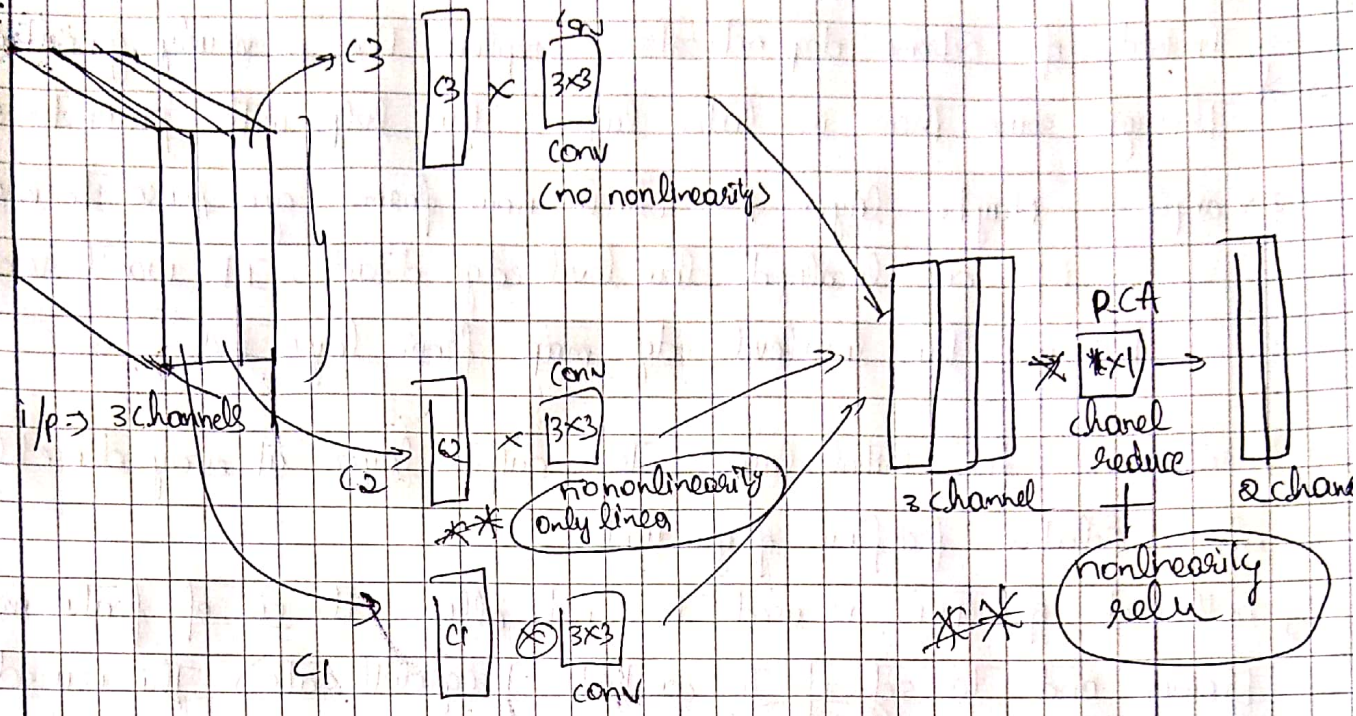
→ Can be combined with inception network

Exception network : Much more efficient use of parameters and better performance than Inception-V3. Extreme version of inception network

Principle: In inception net we tried to somewhat decouple them by using multiple filters in 1 layer and concatenating all output. Here we take this to extreme and do channel by channel convolution using separate kernels of same size

→ ReLU is All the output of these operations are concatenated and reduced by (1×1) PCA type of operation. (Later we apply ReLU activation)

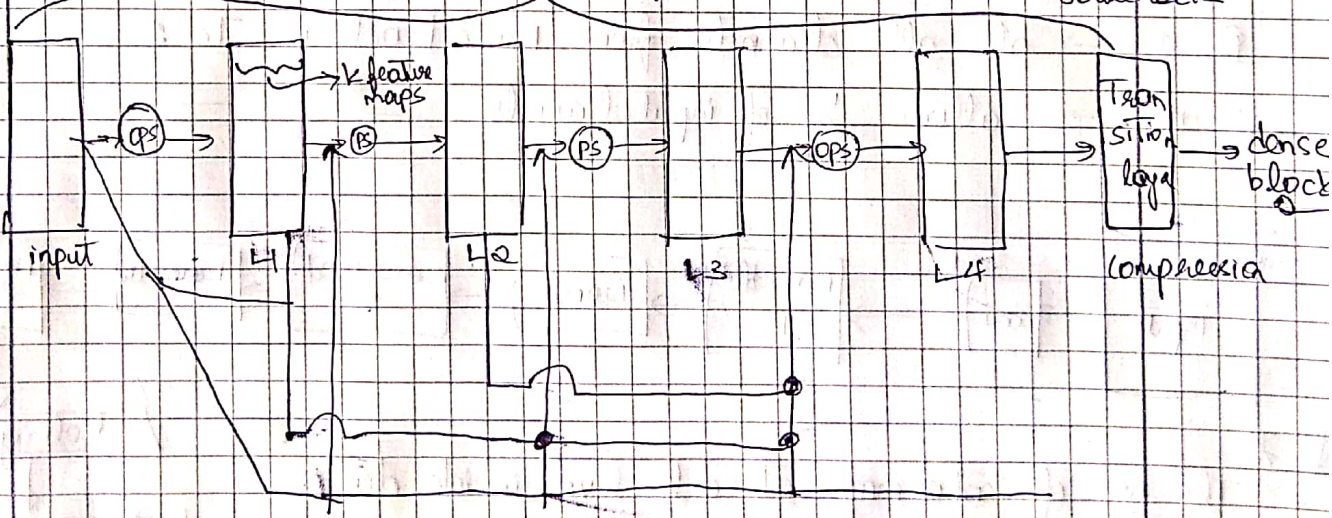
And not in between doing depth wise convolution



Densenets ; ~~Prob~~

Problems To tackling :- Nothing serious in particular, but more optimization of network capacity and feature reuse promotion and increase info flow w/o much adulteration.

Solutions : Dense block $l <= 4$ (growth rate) ops = convolution + ReLU + bottleneck



→ No of direct connections = $\frac{L*(L+1)}{2}$ where L is no of layers inside dense block

→ Input To l^{th} layer = $X_l = H_l([X_0, X_1 \dots X_{l-1}])$
where each layer produces k feature maps

→ Total feature maps = $k_0 + k*(l-1)$
(k_0: Input maps; k: growth rate; $(l-1)$: no of layer)

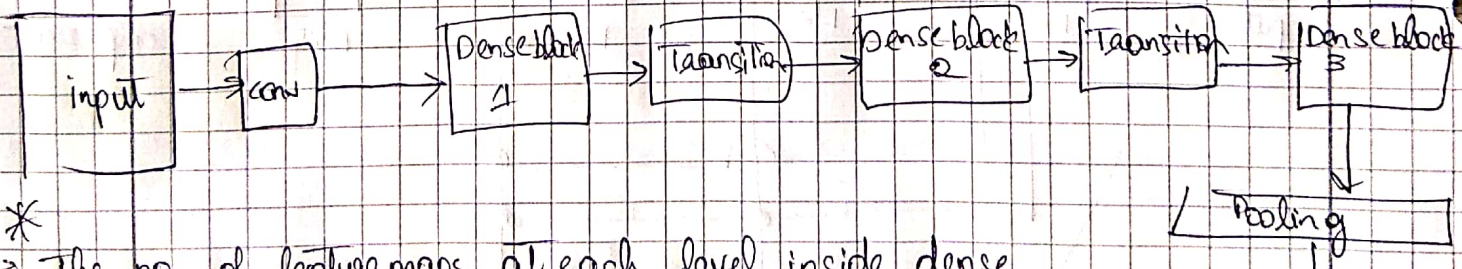
Highlights

→ Instead of extreme deep networks, exploit the previously generated features
** through reuse them in later layer. This helps with parameter efficiency.
Example:- May be layer L3 which now forms eyes/nose etc need some info on localized low level edge details. It won't recreate this but use this low level edge maps from layer L1

→ Direct access for initial layer to output layer at every dense block allows relative free flow of gradients.

→ Bottleneck of (1×1) is used to not allow the no of feature maps to
** blowup and to set it to constant K (growth rate). This is used in
between each layer / inside within dense block

→ Transition layer consist of convolution + Compression + Pooling to control
** amount of info exchanged b/w consecutive dense blocks. Again use dynamic number of (1×1) conv channels using parameter θ
θ = 1 → all all channels from Dense block 1 to 2
θ = 0.5 → allow ½ of input channels — " — —



** → The no of feature maps at each level inside dense block is controlled by growth rate K which is kept less, around typically K = 12. Hence less params are required

RNN - Block 1

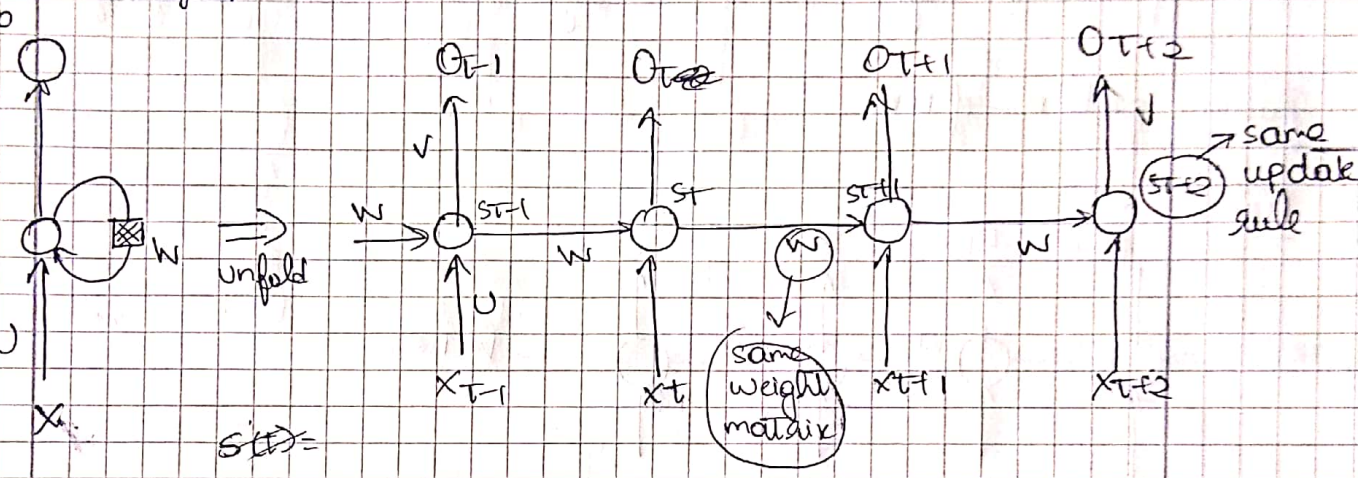
Motivation:- Process sequential data and also data for which order is important

→ Model for which historical states are required

→ Output at current step is a function applied to previous step o/p

→ produced using same update rule applied in previous step

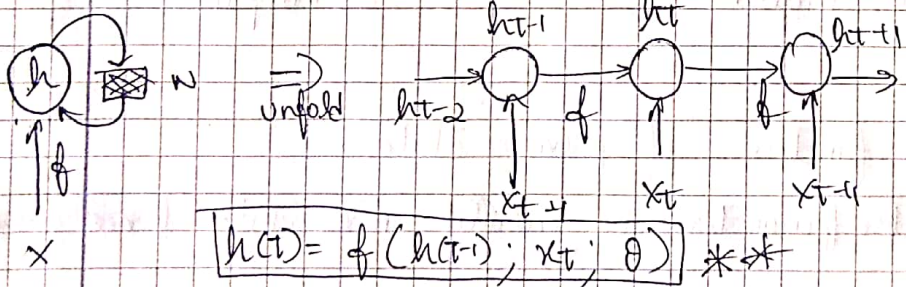
Computational graph



$S(t) =$

→ Regardless of sequence length dimensionality of input remains constant

Back Propagation Through Time



$$h(t) = f(h(t-1); x_t; \theta) \quad ***$$

Advantage

→ gradient computation in unfolded n/w wrt parameters are expensive

Disadvantage

→ No parallelization

→ O(T) where T is history length complexity

1-d Convolution v/s RNN

→ 1-D convolution can be easily used to model Time series data or speech recognition, wherever there is a fixed length analysis is needed

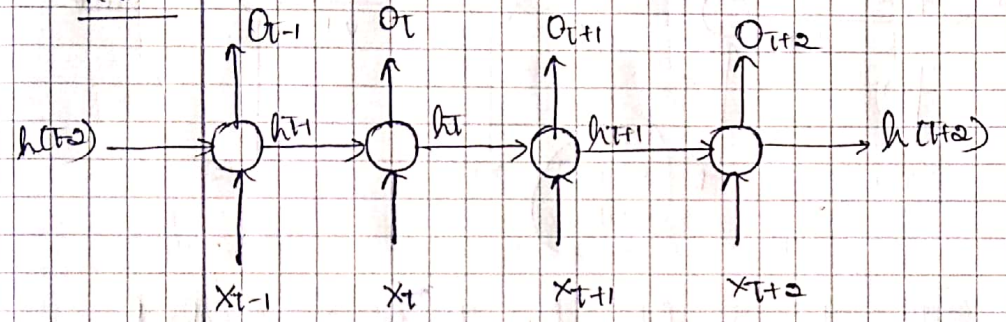
For example:- In Time series, CNN can learn seasonal customer patterns, since customer sales patterns repeat year round (like sales in christmas, summer, easter etc) - Fixed number of filters and fixed length window may be sufficient. Customer behaviour from summer will change in winter hence unlike RNN where all history is used to make decision we can make a quick pattern based matching decision in CNN

similar is the case for audio recording as always, some phonemes tend to occur together. This feature can be position invariant and a patter

Where it doesn't work? Even if we increase receptive field of one-d CNN to include large history by stacking & dialation, the amount of receptive field is still finite Technically, also CNN is a pattern matcher where it tries to reuse pattern filters in any position/order. RNN will still work even if there is no visible pattern or the patterns are extremly unique & non-repititive & might not occur again (like some language modelling tas

Gradient problems of RNN

RNN



Recurrent neural networks :- A class of neural networks that are mainly used to process sequential data. It consist of three states :-

a) Input b) Hidden c) Output

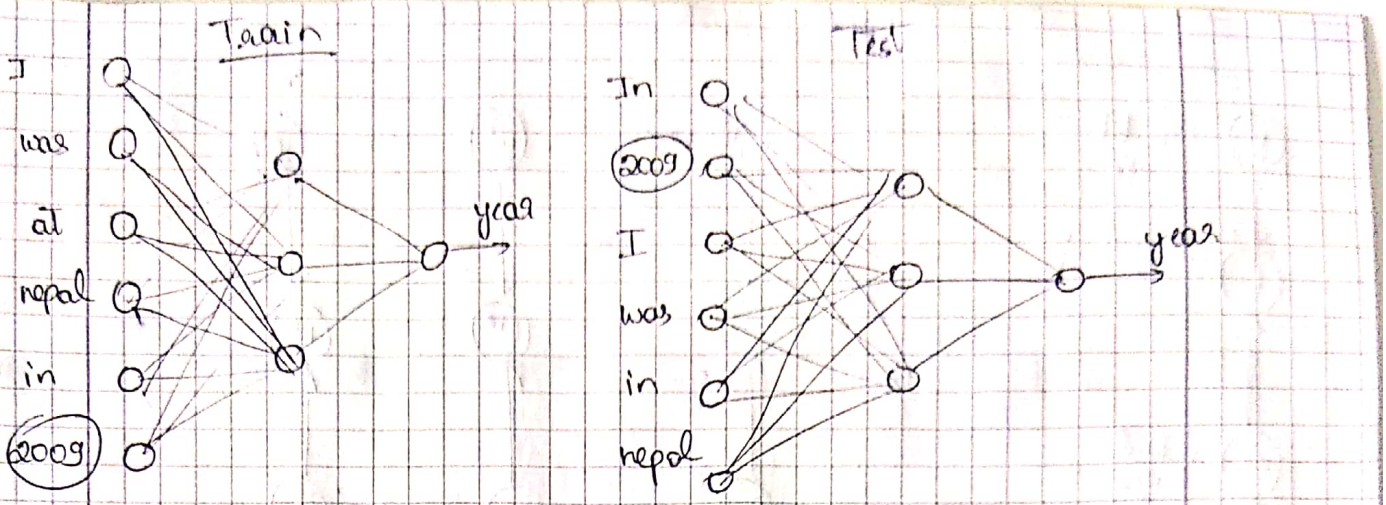
Properties of RNN

a) Current state is a function of previous state
b) We use the same function/parameters to calculate next states, hence parameters are shared
c) Regardless of input size, the hidden state dimensions are fixed and they keep a lossy summary of history upto that current time t
We can also ask to present complete i/p sequence history as it is ingested as with case of auto encodes

Parameter sharing and applying same function 'f' at each time steps

Advantages and Disadvantages of RNN

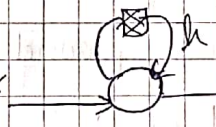
→ With just feed forward n/w, lets say we are trying to extract date



→ The position of the date in sentence matters as weights are learnt separately for each Token and if position changes in Test, it doesn't generalize well as during Train we focussed on last Token.

→ To remedy this we need to Train on all possible combos of sentence so that network figures out all rules. This is a disadvantage

→ RNNs parameter sharing & same function helps so that you don't need To learn separate rules of language applied To diffrent position Every Thing is encapsulated in hidden state which is kind of summary

→ Once only with folded RNN 

There is a dis advantage that the input cannot be defined over constant length of history

For ex:- $h(T) = g(T)(x(1), x(2), \dots x(T-1))$ — T-1 params

$h(T-1) = g(T-1)(x(1), x(2), \dots x(T-2))$ — T-2 inputs

$h(T+1) = g(T)(x(1), x(2), \dots x(T))$ — T params

Problem:- each time $g(T)$ is looking on variable length of inputs

This may lead to $g(T)$ trying To learn a diffrent rule at each time step completely

with onfolding we can refer current state using only its fixed previous states. This allows for more generalization & faster Training

$h(T) = f(h(T-1); x(T); \theta)$

$h(T-1) = f(h(T-2); x(T-1); \theta)$

$h(T+1) = f(h(T); x(T+1); \theta)$

} same one i/p & one hidden state

→ Since we have choosen to represent current state only with respect to some defined previous state and not looking at all inputs at every time, Gradient propagation becomes problematic and expensive. Even forward prop takes more time ($O(\tau)$ τ → length of sequence)

For ex:- compare two cases below

$h(t-1) = g_1(t)\,(x^{(1)}, x^{(2)}, \ldots x^{(t-2)})$

$h(t) = g_2(t)\,(x^{(1)}, x^{(2)}, \ldots x^{(t-1)})$

$h(t+1) = g_3(t)\,(x^{(1)}, x^{(2)}, \ldots x^{(t)})$

Rolled n/w

↓

→ Direct access to inputs, so better gradient propagation

→ No dislike dependency of $g_3(t)$ to $g_1(t)$ or $g_2(t)$, since we would already have complete i/p sequence we can compute all $h(t-1)$, $h(t)$, $h(t+1)$ in ||el

→ Need to learn different g_1, g_2, g_3 for different steps, which's really difficult

$h(t-1) = f(h(t-2), x(t-1), \theta)$

$h(t) = f(h(t-1), x(t), \theta)$

$h(t+1) = f(h(t), x(t+1), \theta)$

unrolled n/w

→ No direct visibility from $h(t)$ to $h(t-2)$ or $h(2)$, should always go through $h(t-1)$

→ Sequential processing of info on both forward and backward pass as $h(t)$ cannot see $h(t-2)$ $h(t-3)$ etc & can only see $h(t-1)$ as per defined model

→ Just need to learn f, θ same is applied at every step

Different gradient problems in RNN

Consider change of error in final layer τ wrt first hidden state

$$\frac{\partial E^{[\tau]}}{\partial h^{[0]}} = \frac{\partial E^{[\tau]}}{\partial h^{[\tau]}} \prod_{t=1}^{\tau} \frac{\partial h^{[t]}}{\partial h^{[t-1]}}$$

τ → depth of layers
E → error
h → hidden state

WKT $h^{[t]} = \tanh(a^{[t]})$

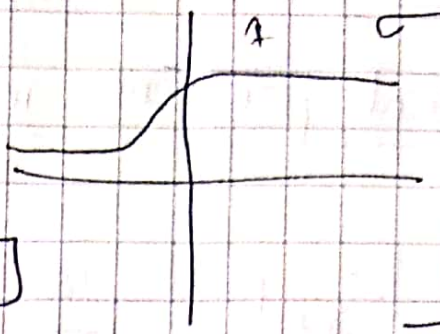
$a^{[t]} = W x^{[t]} + W h^{[t-1]}$

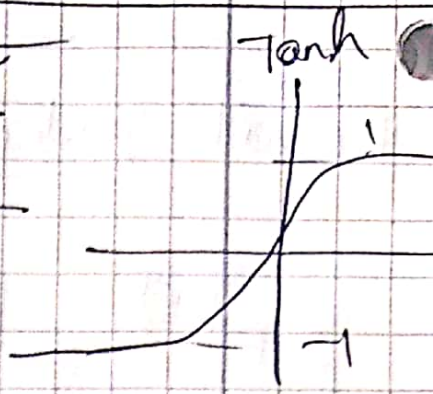
$\frac{\partial \tanh\theta}{\partial \theta} = 1 - \tanh^2\theta$

↓
vanishing gradient

Pre requisite

1) Bounding of derivatives of σ, Tanh





WKT range of $\sigma \cong [0, 1]$ & Tanh $[-1, 1]$

$\frac{d\sigma(x)}{dx}$ range would be $\cong [0, 1/4]$

why? $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$ is max at $\sigma(x) = 1/2$

$$\Rightarrow \left(\frac{d\sigma(x)}{dx}\Big|_{max} = 1/4\right) \quad **$$

IIIly for Tanh $\frac{d\,\tanh x}{dx}\Big|_{max} = 1 \Rightarrow$ range bound $= 1$

why: $\frac{d\,\tanh x}{dx} = 1 - \underbrace{\tanh x^2}_{\text{@ zero}} = 1$

II) Eigenvalue decomposition and Powerlaw

→ Any weight matrix W or U or V can be decomposed into eigen vectors & values

$$W = M \, \text{diag}(\lambda) M^{-1} \quad \text{let } \lambda = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -3 \end{bmatrix} \text{ for example}$$

→ '-3' is dominant eigen vector by magnitude

$$W^2 = M (\text{diag}(\lambda))^2 M^{-1} \; ; \; W^3 = M (\text{diag}(\lambda))^3 M^{-1} \Rightarrow$$

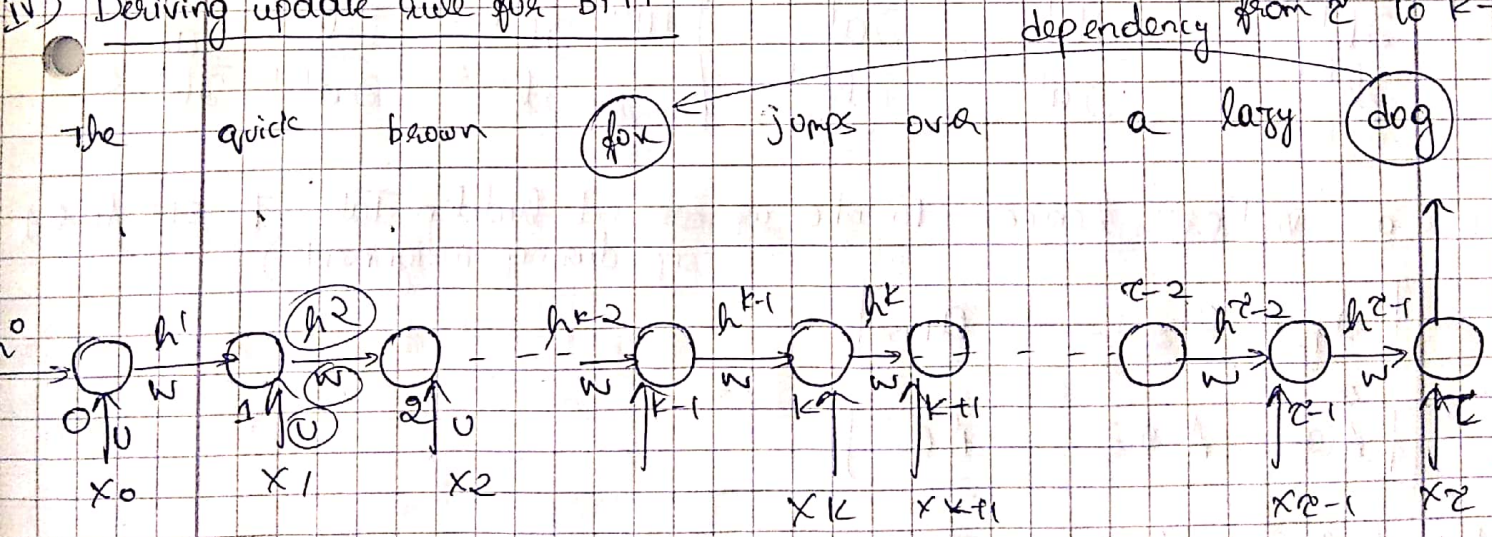
$$W^T = M (\text{diag}(\lambda))^T M^{-1} \Rightarrow \text{Why? WRT eigenvalue act as scaling factor to keep eigen vector same}$$

Hence $W^T \rightleftarrows \lambda^T = \begin{bmatrix} 1^T & 0 & 0 \\ 0 & 2^T & 0 \\ 0 & 0 & -3^T \end{bmatrix}$ $1^T \ll 2^T \lll 3^T$ ⇒ we can ignore 1 & 2 & can get away by only considering principal vector $(3)^T$

III) Chain rule and gradient descent

We always try to (learn parameters) (through loss function), because this is how we learn stuff ⇒ $\frac{\partial \text{Out}}{\partial W}$ or $\frac{\partial \text{Out}}{\partial \text{Bias}}$

IV) Deriving update rule for BPTT



→ lets say Task is to remember animals Then we need to backpropagate from last layer τ to required layer k-1, it can be 0th layer if you want Too.

My requirement : $\frac{\partial Loss}{\partial W} = ?$

→ Apply chain rule ⇒ $\frac{\partial Loss}{\partial h^2} \cdot \frac{\partial h^2}{\partial h^{2-1}} \cdot \frac{\partial h^{2-1}}{\partial h^{2-2}} \cdots \frac{\partial h^k}{\partial h^{k-1}} \cdot \frac{\partial h^{k-1}}{\partial h^{k-2}} \cdot$

$\frac{\partial h^{k-1}}{\partial W} \cdot \frac{\partial h^{k-1}}{\partial K} \cdot \frac{\partial h^{k-1}}{\partial U}$ → discard

h^{k-2} = (constant so ignore) = weight of hidden state

not interested

WKT $h^{k-1} = W h^{k-2} + U x^{k-1}$ **

$\frac{\partial h^{k-1}}{\partial W} = h^{k-2}$ constant

** $\Rightarrow \frac{\partial Loss}{\partial W} = \frac{\partial Loss}{\partial h^2} \cdot \frac{\partial h^2}{\partial h^{2-1}} \cdot \cdots \frac{\partial h^{k+1}}{\partial h^{k-1}} = \frac{\partial Loss}{\partial h^2} \prod_{\tau=k-1}^{\tau=2} \frac{\partial h^\tau}{\partial h^{\tau-1}}$

Consider one step of backprop $\frac{\partial h^\tau}{\partial h^{\tau-1}}$ or $\frac{\partial h^{\tau-1}}{\partial h^{\tau-2}} = ?$

WKT $h^\tau = \text{Activation}(a^\tau) = A(a^\tau)$

$a^\tau = W h^{\tau-1} + U x^\tau$

$\Rightarrow \frac{\partial h^\tau}{\partial h^{\tau-1}} = \frac{\partial h^\tau}{\partial a^\tau} \cdot \frac{\partial a^\tau}{\partial h^{\tau-1}}$ | III by $\frac{\partial h^{\tau-1}}{\partial h^{\tau-2}} = \frac{\partial h^{\tau-1}}{\partial a^{\tau-1}} \cdot \frac{\partial a^{\tau-1}}{\partial h^{\tau-2}}$

activations are vectors hence (Remember we set hidden state to 512/1024 size during assignment)

$a^\tau = [a_{\tau,1}, a_{\tau,2} \cdots a_{\tau,n}]$

III by $h^\tau = [A(a_{\tau,1}), A(a_{\tau,2}) \cdots A(a_{\tau,n})]$

we need to find rate of change of each individual h^τ v/s all vector a^τ

Hence Take Jacobian

For example :-

$$\frac{\partial h_1^T}{\partial a^T} = \left[\frac{\partial A(a_{T1})}{\partial a_{T1}} \quad \underbrace{\frac{\partial A(a_{T2})}{\partial a_{T1}} \quad \frac{\partial A(a_{T3})}{\partial a_{T1}} \quad \cdots \quad \frac{\partial A(a_{Tn})}{\partial a_{T1}}}_{\text{zeros}}\right]$$

$a_{T3} \neq a_{T1}$ (not dependent / related to denominator, so constant numerator)

$$\frac{\partial h_2^T}{\partial a^T} = \left[\underbrace{\frac{\partial A(a_{T1})}{\partial a_{T2}}}_{\text{zero}} \quad \frac{\partial A(a_{T2})}{\partial a_{T2}} \quad \underbrace{\frac{\partial A(a_{T3})}{\partial a_{T2}} \quad \cdots \quad \frac{\partial A(a_{Tn})}{\partial a_{T2}}}_{\text{zero}}\right]$$

$$\Rightarrow \begin{bmatrix} \partial h_1^T/\partial a_T \\ \partial h_2^T/\partial a_T \\ \vdots \\ \partial h_n^T/\partial a_T \end{bmatrix} = \underbrace{\begin{bmatrix} \partial A(a_{T1})/\partial a_{T1} & 0 & 0 & \cdots & 0 \\ 0 & \frac{\partial A(a_{T2})}{\partial a_{T2}} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & & 0 \\ 0 & 0 & 0 & \cdots & \frac{\partial A(a_{Tn})}{\partial a_{Tn}} \end{bmatrix}}_{\text{diagonal matrix}} \rightarrow (2)$$

Multiply for $\frac{\partial a^T}{\partial h^{T-1}} = \cancel{\text{diagonal}} \frac{\partial \{W h^{T-1} + U x^T\}}{\partial h^{T-1}} = W \rightarrow (2)$

$$\Rightarrow \frac{\partial h^T}{\partial h^{T-1}} = \text{diagonal}\{\overset{\text{partial derivative short notation}}{\overbrace{\cancel{A}\, Act'}}(a_T)\}W$$

$$\left\| \frac{\partial h^T}{\partial h^{T-1}} \right\|_{\text{magnitude}} = \| \text{diagonal}\{Act'(a_T)\} W \|$$

from (II) if $\begin{cases} \text{activation } Act = \sigma \text{ then } Act' \approx [0, \frac{1}{4}] = \gamma \\ \text{activation} = \tanh \text{ then } Act' \approx [0, 1] \end{cases}$

$$\Rightarrow \|grad\| = \|\gamma W\| \quad **$$

wkt $\|a.b\| \leq \|a\| \|b\| \Rightarrow \|grad\| \leq \|\gamma\| \|W\|$

Use (1) and replace $||W||$ by its eigen equivalent

$$\Rightarrow ||grad|| = ||\gamma|| \, ||M \lambda M^{-1}|| = \frac{\partial h^T}{\partial h^{T-1}}$$

$$\text{III}^{ly} \quad \frac{\partial h^{T-1}}{\partial h^{T-2}} = ||\gamma|| \, ||M \lambda M^{-1}||$$

$$\Rightarrow \prod_{T=K+1}^{\infty} \frac{\partial h^T}{\partial h^{T-1}} = (\lambda . \gamma)^{T-K} \, ||M M^{-1}||$$

λ → dominant eigen vector

γ → upper bound of activation derivative

if

Vanishing gradient problem

if $\lambda . \gamma < 1$ Then for significantly large $T \ll K$

gradients may vanish

ex:- $(0.4 * 0.9)^{100-10} = (0.36)^{90} \approx 0$

final step, intermediate

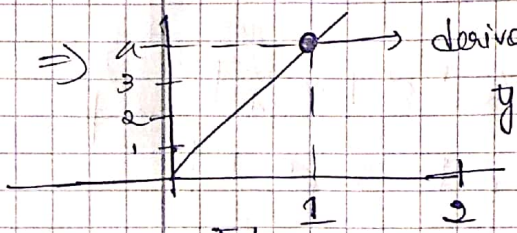
if $\lambda . \gamma > 1$ Then for significantly large $T \gg K$

gradients explode

$(0.4 * 4)^{100-10} = (1.6)^{90} \gg 1$ (1.6^{90})

Why its difficult To use ReLU in RNN? **

ReLU scales linearly for positive quadrant hence its derivative = scaling factor



$\Rightarrow$ derivative = 4

$y = 4x$

$\Rightarrow \frac{\partial y}{\partial x} = 4$

$\Rightarrow$ upper bound $\gamma = [0, 4]$

large number, hence more chances of $(\gamma . \lambda)^{T-K}$ becoming huge.

**

→ Here notice that gradient problem is due to two reasons

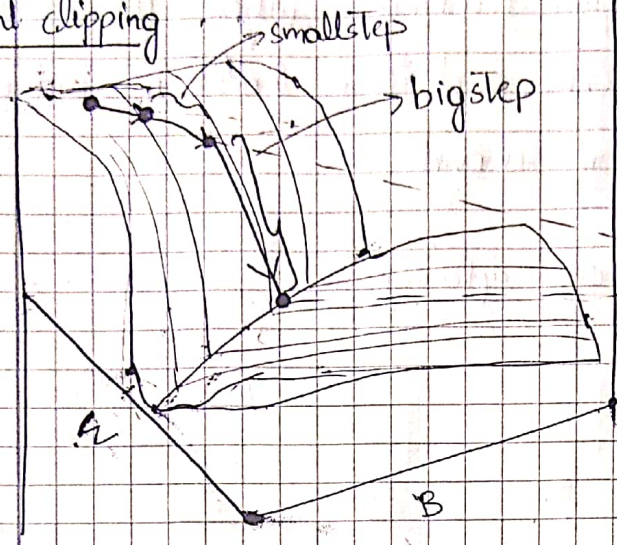
a) choice of activation fnc

b) Repeated multiplication of weight matrix with itself

Remedies

- **Exploding gradient** : Gradient clipping , activation function, no of layers

Gradient clipping :



Loss $J(w,b)$ → Due to big diff loss decreased suddenly
→ caused large gradients and caused parameters to jump out of space during gradient descent

$W_i = W_i - \alpha (\text{large } \nabla W_i)$

new weights after a gradient update (out of space)

→ Makes learning unstable and we lose out on our efforts we put for optimization earlier

---

Vanishing gradient : Activation function, gating, initialization

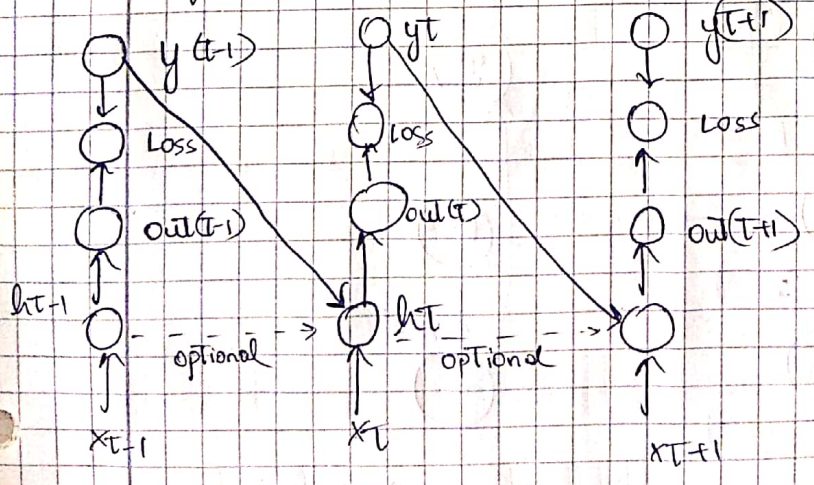
$W_i = W_i - \alpha \nabla W_i$ ∇W_i is too small hence no learning takes place

→ Better to have vanishing than exploding gradient as in the latter optimization is not possible and for vanishing gradient its painfully slow.

$$h^{(t)} = \theta^{T} \theta\, h(0)$$ (T: Transpose) → eigen value

---

Teacher forcing : Arises from MLE criterion. Recieve expected output as one of the input to hidden state



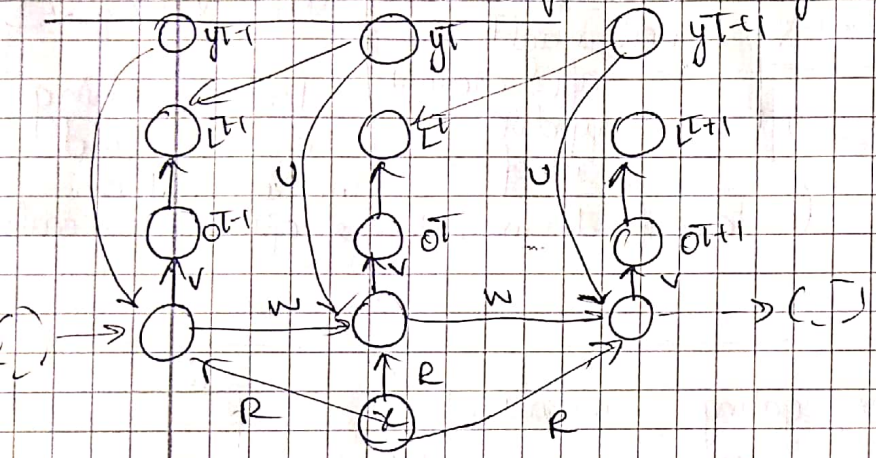
→ if no hidden-hidden connection Then BPTT not required, if connection exists Then use BPTT To Train
→ Works well if "out" is high dim & rich
→ Allows arbitrary distribution over y given sequence of x of same length

Disadvantages :- Teacher forcing was trained using expected outputs directly, hence it hasn't learnt much of useful things in its memory state.

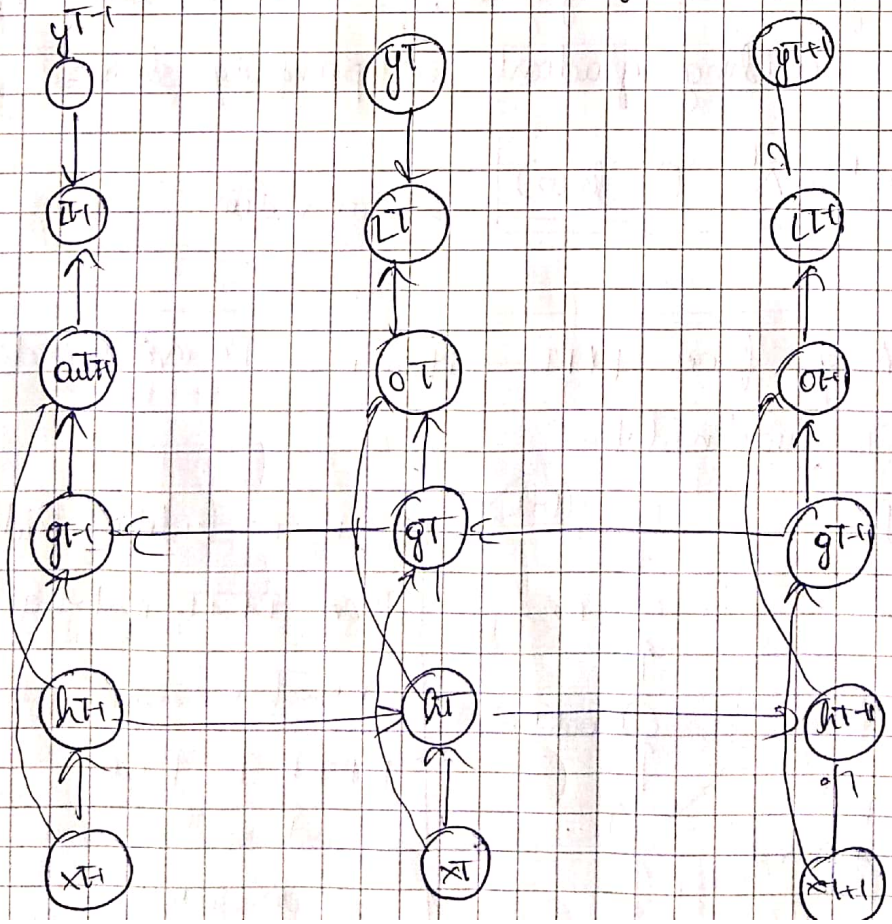
→ Hence during test it doesn't generalize well on unseen instances.

Remedy :- Use a type of curriculum learning, where the network is Teacher forced for first few epochs and then learnt in normal RNN way with its o/p → hidden connection removed

Fixed context vector To sequence :- Image captioning



Bidirectional RNN :- Hand writing recognition, seq-seq models



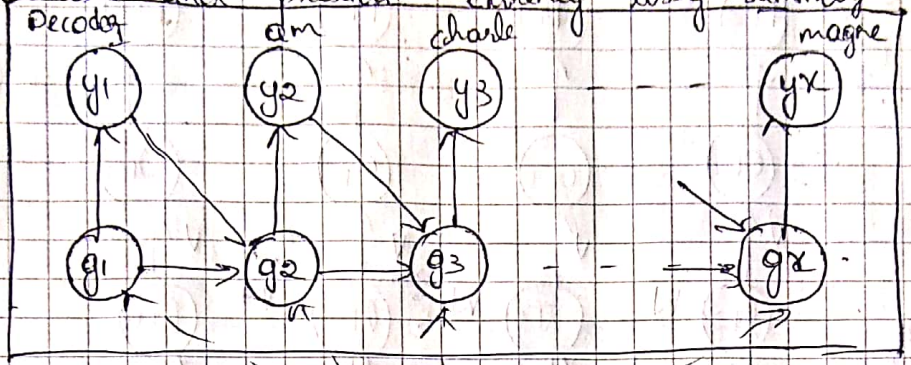
Take into account both past & future info → helpful in predicting middle word of sentence

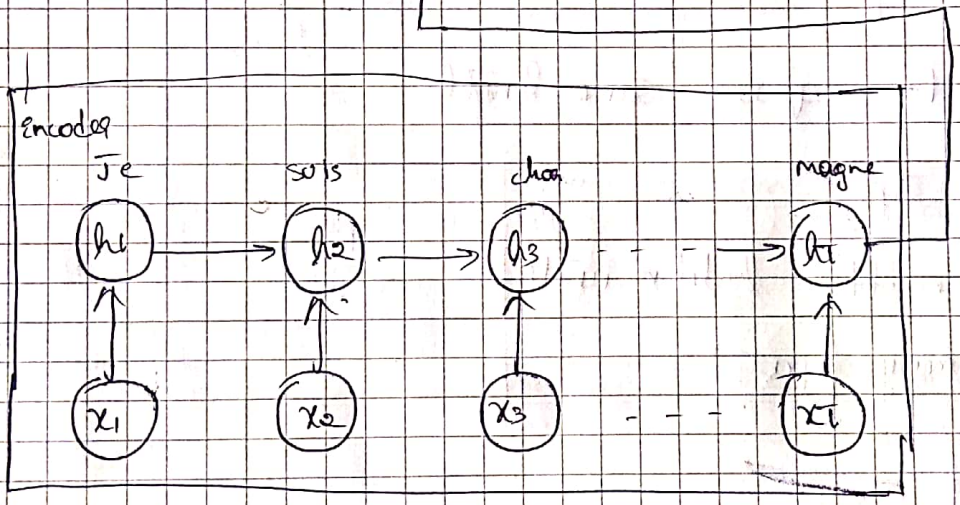
Encoder-Decoder : Machine language Translation

→ Variable length i/p and o/p

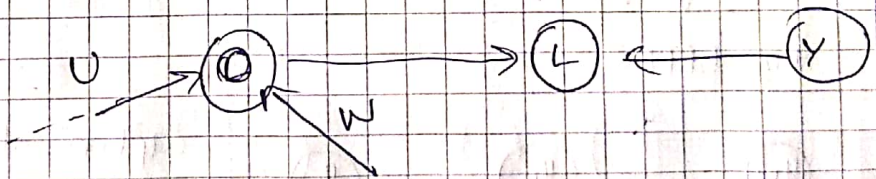
→ Problems faced in (context vector c) as it cannot capture long range dependences and presents extremely lossy summary. Also gradient needs to flow through context vector slows down learning

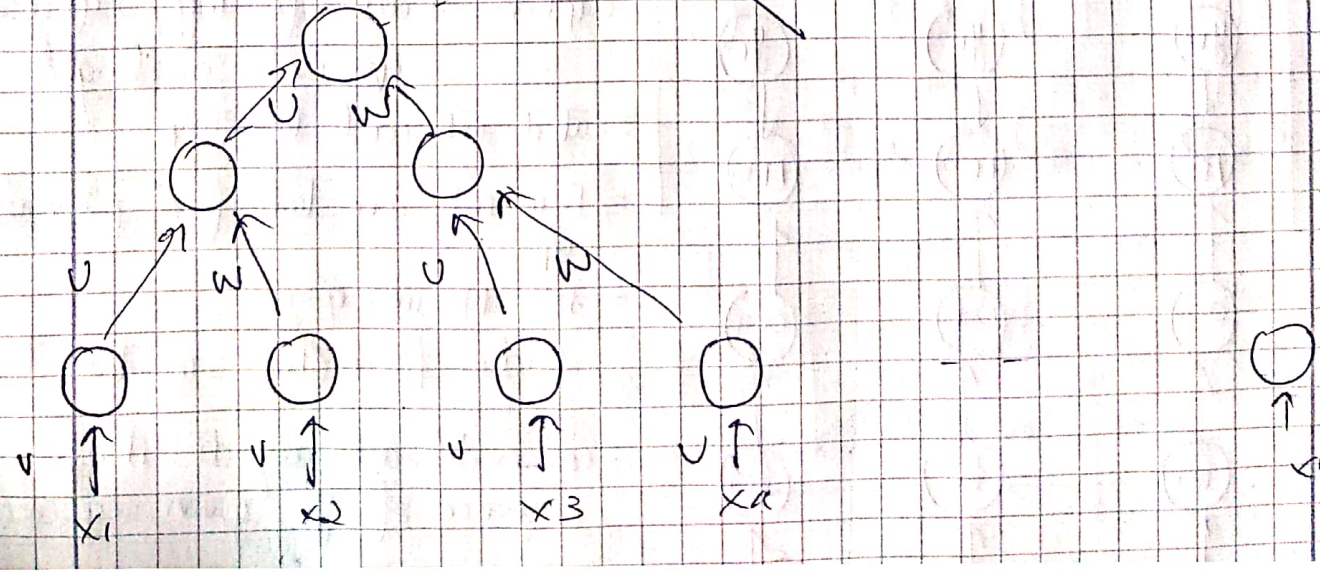
bottleneck





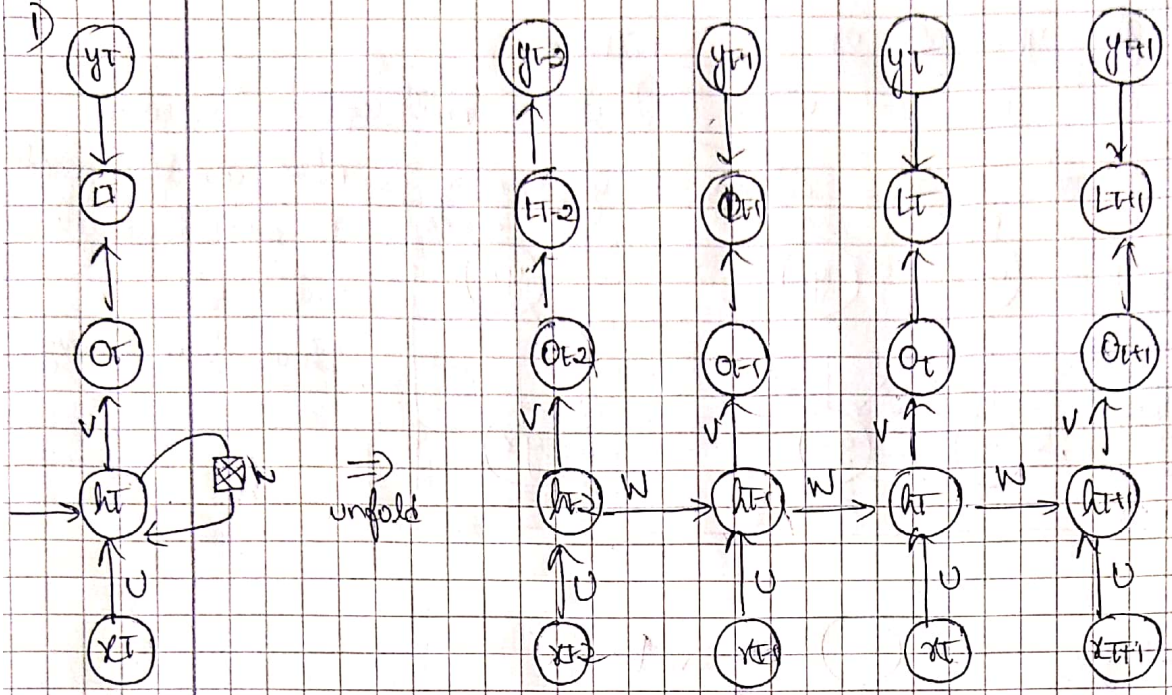
Recursive Tree :- Sentiment analysis, Sentence parser





Match the following :- Architectures with captions

1)



Caption :- Sequence to sequence same length

→ output at each time step

→ Recurrent connection b/w hidden units

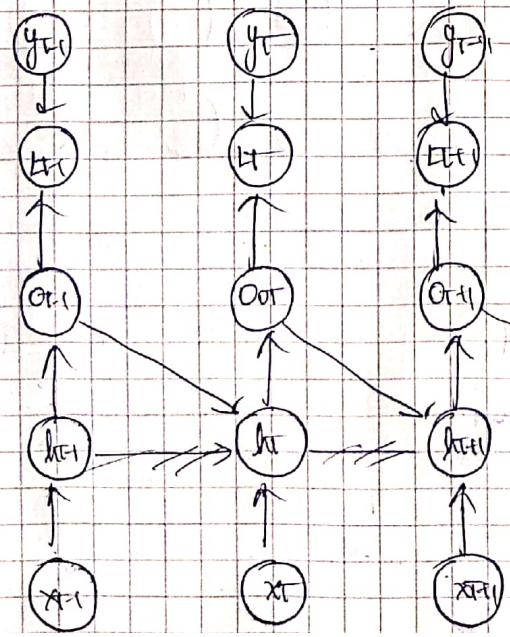
→ Only Relevent summary of past

→ Sequential Training

→ Can compute any fnc computable by Turing m/c (UTC)

→ Trainable by BPTT

2)



Caption :- Teacher forcing / Sequence To sequence same length

→ output at each time step

→ Recurrent connection from previous o/p

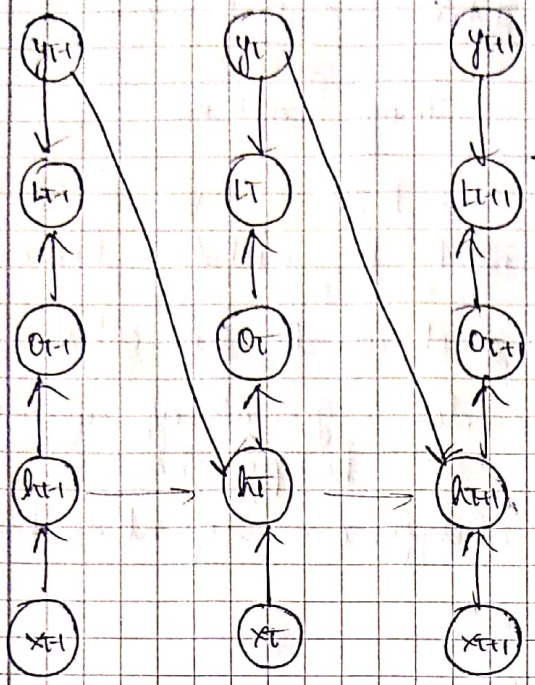
→ ~~Rate~~ No summary

→ Teacher force but sequential Training

→ Can model arbitrary distribution over sequences of y given sequence of x

→ lacks important info from past unless o is high dim & rich

3)



Caption: Teacher forced / seq-seq same length

→ Output at each timestep

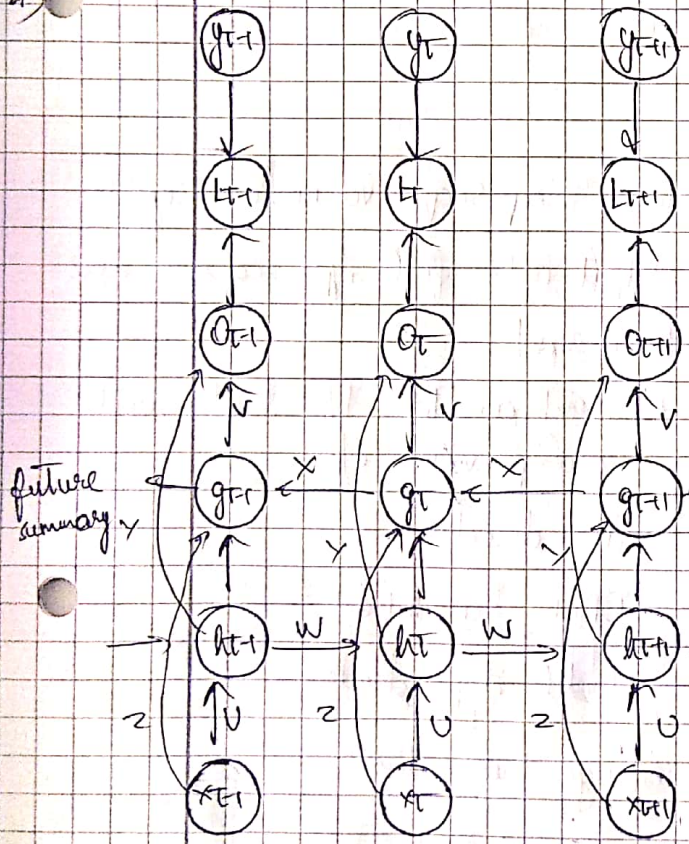
→ Recurrent connection from both hidden & previous o/p

→ Summary from past

→ Parallelized Training / Teacher forced

→ Can model arbitrary distribution over sequence of y given sequences of x

4)



Caption: Seq-seq bidirectional same length

→ Output at each time step

→ Recurrent connection b/wn hidden units

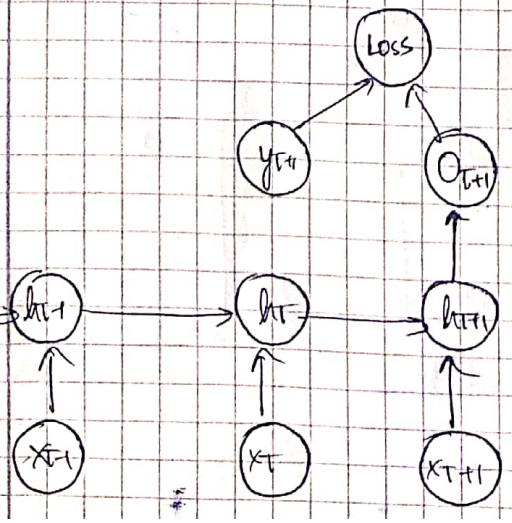
→ Both past and future summary

→ Sequential BPTT Training

→ Compute any function computable by Turing m/c

past summary

5)



Caption: Complex structure To fixed size vector

Text summarization

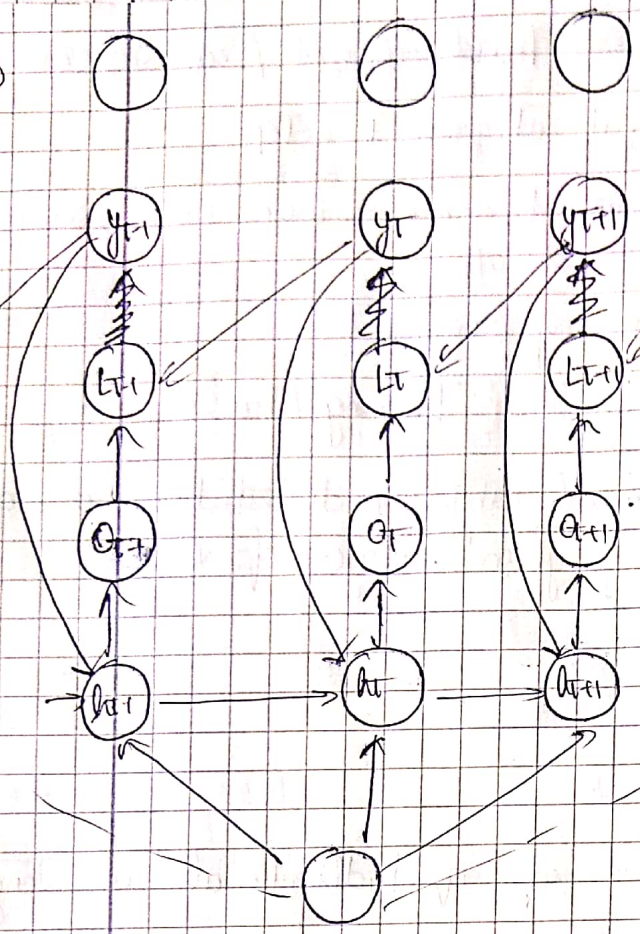
→ Output after all i/p has been read

→ Recurrent connection b/wn hidden units

→ Only past summary

→ Sequential Training BPTT

→ Mode Encoder: generate content vector



Caption :- Content fixed vector to sequence

Image caption

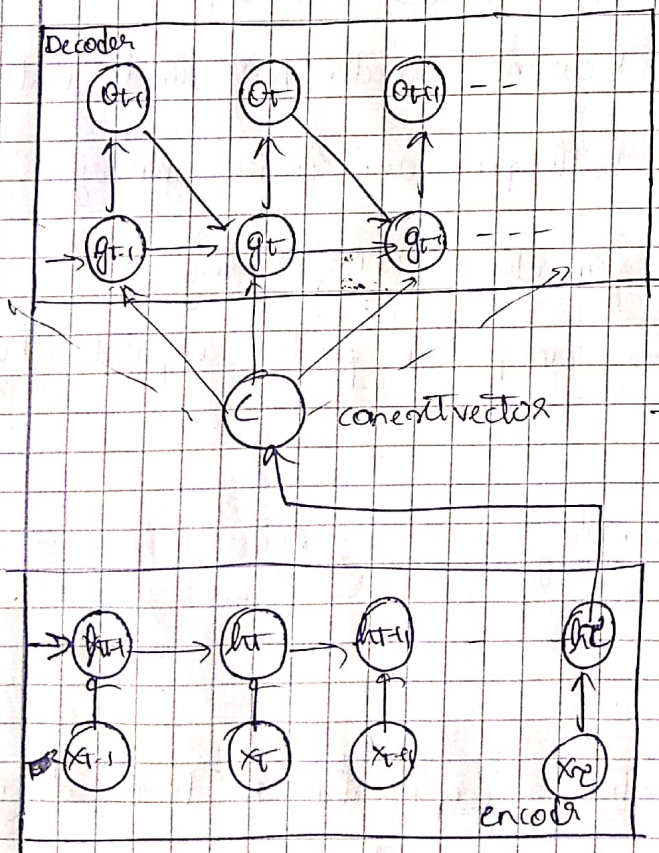
→ Input x serves as constant context to initializ hidden state

→ Recurrent connection from output + hidden

→ Decoder :- Generate o/p sequence from fix input state

→ Teacher force + sequential Training

→ UAC



Caption :- Seq seq variable length

→ Output after full i/p sequence has been read

→ Recurrent connections b/w hidden units & from previous o/p

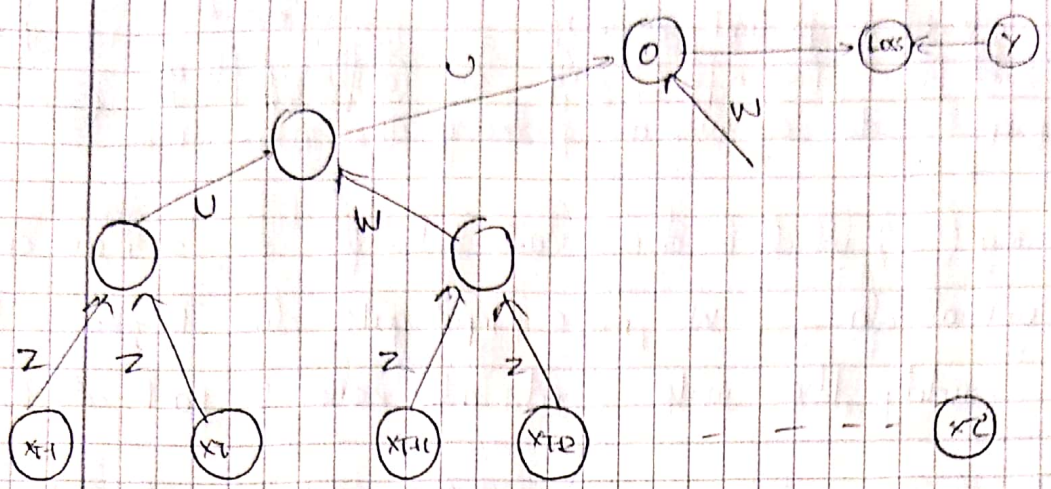
→ encoder-decoder

→ BPTT sequential

→ Model dependencies

Language Translation

8)



Caption:- Seq - fixed size vector ⇒ Sentiment analysis / Sentence Parse

→ Compute graph structured as deep tree

→ BPTT sequential Training

→ Encoder Type architecture

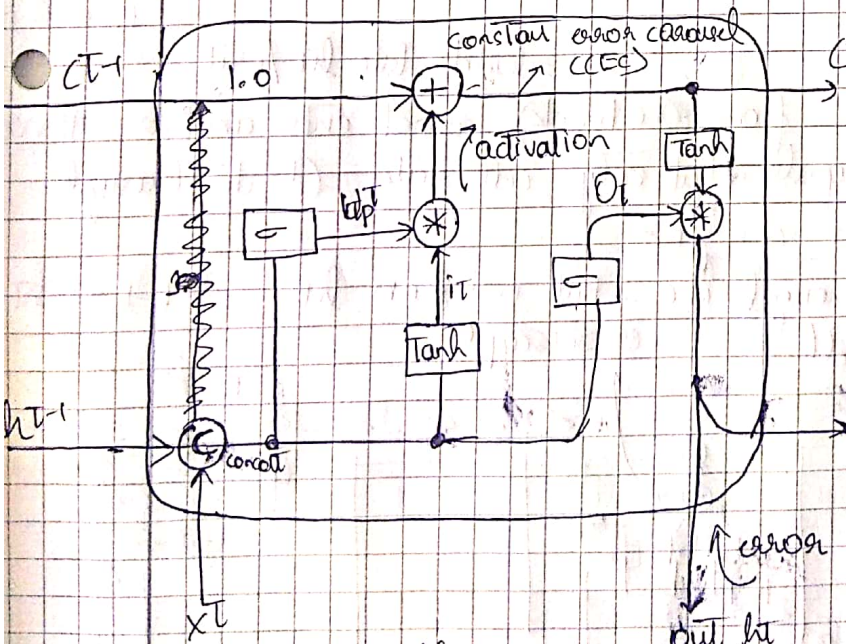
→ Output after complete sequence X has been read.

~~LSTM and~~ o

Gated Recurrent Neural Nets

LSTM - Long Short Term Memory

Basic 1997 LSTM cell (w/o forget gate)



LSTM cell

(Basic unit of hidden layer of LSTM n/w)

→ NO forget gate

⇒ ~~No~~ Idea: Instead of utilizing gradient at every step or make updates at every step, allow another parallel path-way w/o much obstructions where error can flow back and we can add our new current activation info only if required

→ Control the info flow through gates

** → Don't use any heavy transformation on error path to make it flow almost constant

How does it solve vanishing gradient problem?

irrelevant ← "the quick brown fox jumps over a lazy dog" ← irrelevant

□ → relevant

⊗ → irrelevant

During forward prop :- Lets say we are processing the above string

→ While processing "a" which is an irrelevant token for me, as i am only extracting "animals" class. We make input gate close to zero so that this doesn't muddy the waters and add noise to past cell history.

→ Since past cell history C_{t-1} is on parallel path even if $\tilde{C}_t * I_t$ is zero C_t won't become zero. I_t would still remain $C_t = C_{t-1}$ *

$$C_t = C_{t-1} * 1.0 + \text{input gate} * \tilde{C}_t$$

(no forget gate)

→ Input gate protects cell state from being distorted by noise like token "a"

During backward prop :-

→ Errors path through LSTM cell is through output gate and this protects it.

Example :- Propagate error from dog → quick. (Only update near time steps that has (dog, lazy, fox, quick) ignore the rest.

→ Once the error escapes through a valid (relevant) LSTM output gate, at lets say token "dog" position, Our thinking is, it can ride CEC and reach "lazy", "fox" and "quick"

constant error carousel / parallel path

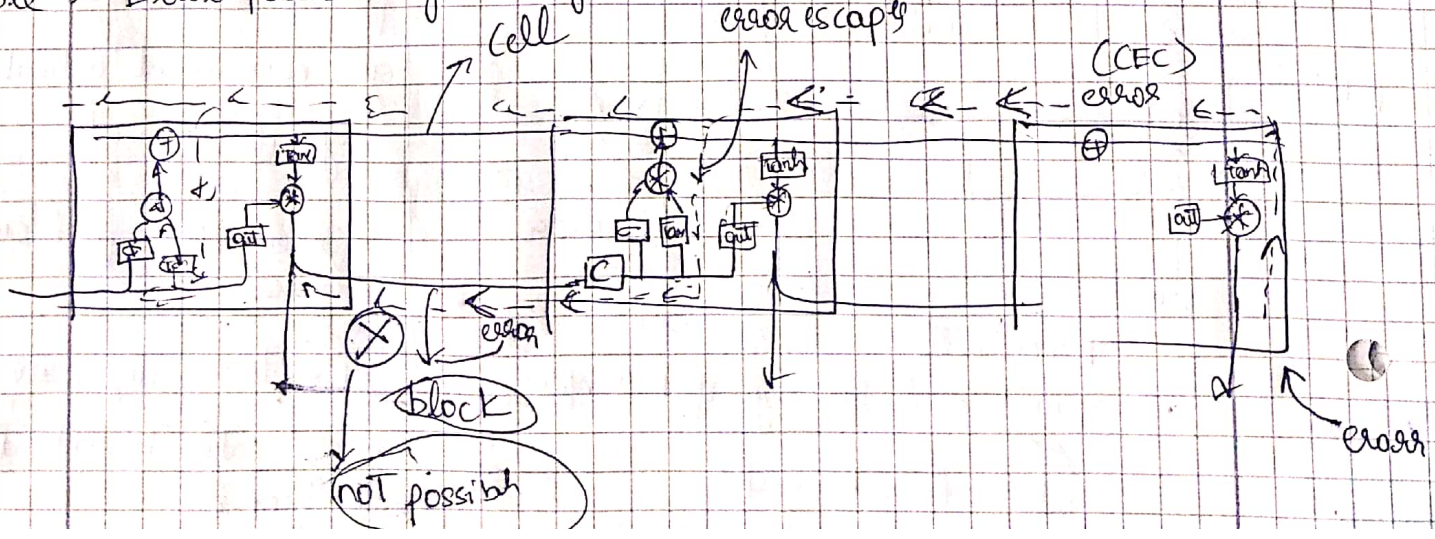
* * *

→ Due to Truncated BPTT error from one block cannot enter another, Hence blocks don't exchange error signals but only cell states C_t do through CEC.

hidden state h_t / Out

$C_t \rightarrow C_{t-1} \rightarrow C_{t-2}$

Example :- Error passes only through CEC. But not from $h_t \rightarrow h_{t-1} \rightarrow h_{t-2}$

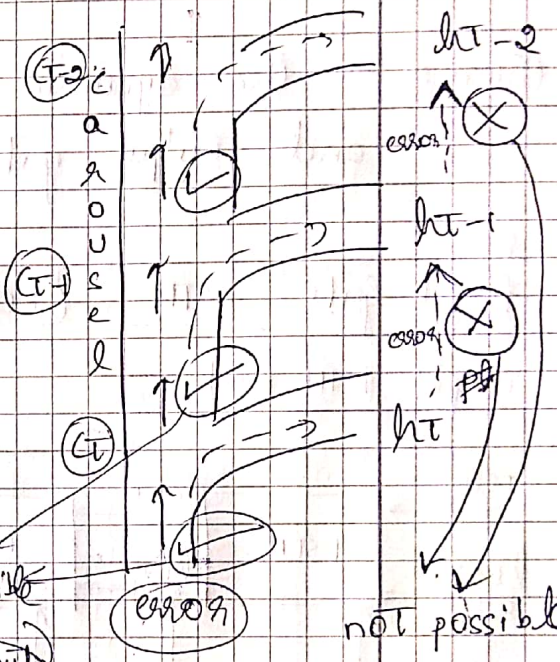


→ First error signal arrives at memory cell output which gets scaled by output gate O_g and nonlinearity squasher tanh. Then it enters Carousel CEC

→ It can only escape and update weights of other cell only if their input gates allow so (if they are relevant for us).

→ Let's say they escape through input gate and again scales by tanh and updates weights and h_T (block/output).

→ Now this residual error from h_T cannot travel to h_{T-1} due to truncated back prop; but error can still flow through cell state C_T to C_{T-1} as there is no obstruction



→ Problems **

→ LSTM cannot work where one block h_t exclusively save others like h_{T-1}, h_{T-2} etc as all error are dependent on error propagated from final output to carousel rather than blocks

** → This also means that info in cell state gets built up linearly over time

Ex: $C_T = 1.0 * C_{T-1} + I_g * g(\text{NetInput})$

$Y_2 = X_1 + mX_2$

$Y_3 = 1.0 * Y_2 + mX_3 = X_1 + mX_2 + mX_3$

$Y_4 = Y_3 + mX_4 = X_1 + mX_2 + mX_3 + mX_4$

** → Since we use squashing function tanh, over time all the inputs outputs gets dangerously close to +1 or -1 as magnitude of C_T grows linearly (if C_T becomes 100 or -100 then its tanh value would be close to ±1.)

→ This causes saturation and usually gradients don't backpropagate well enough near these saturation points. Hence we need to methodically forget some past stuff so that we move away from

saturation region of Tanh. This can be done if Cell state is not allowed to blowup linearly. (How to do this?)

Option 1 :- Use constant annealing factor to forget stuff of past and make cell C_t not saturable

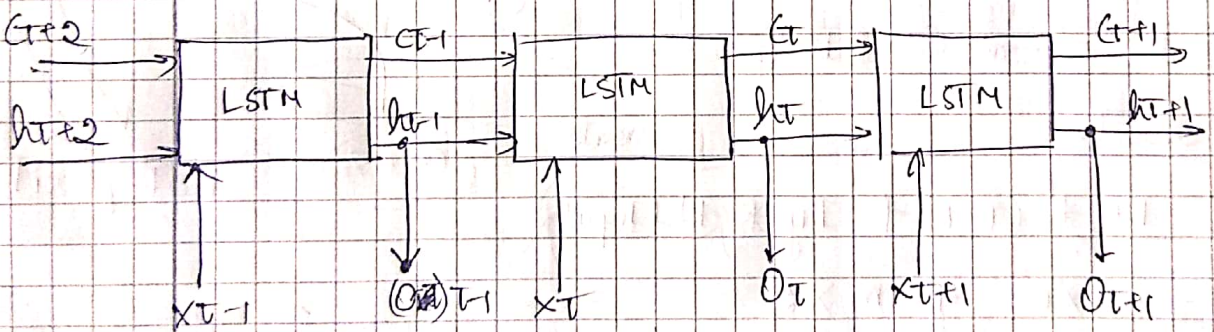
$$C_t = \underbrace{0.9}_{\text{annealer (non-learnable / constant)}} * C_{t-1} + I_g * Net$$

Problems :- We should not forget relevant tokens / history / info but only forget irrelevant ones. Having a constant annealer for every LSTM cell makes every cell forget some stuff of history whether it is relevant or not

Option 2 :- Use dynamic forget gate which is similar to input and output gate & learnable

Forget gate LSTM (2003 - Gers Phillip)

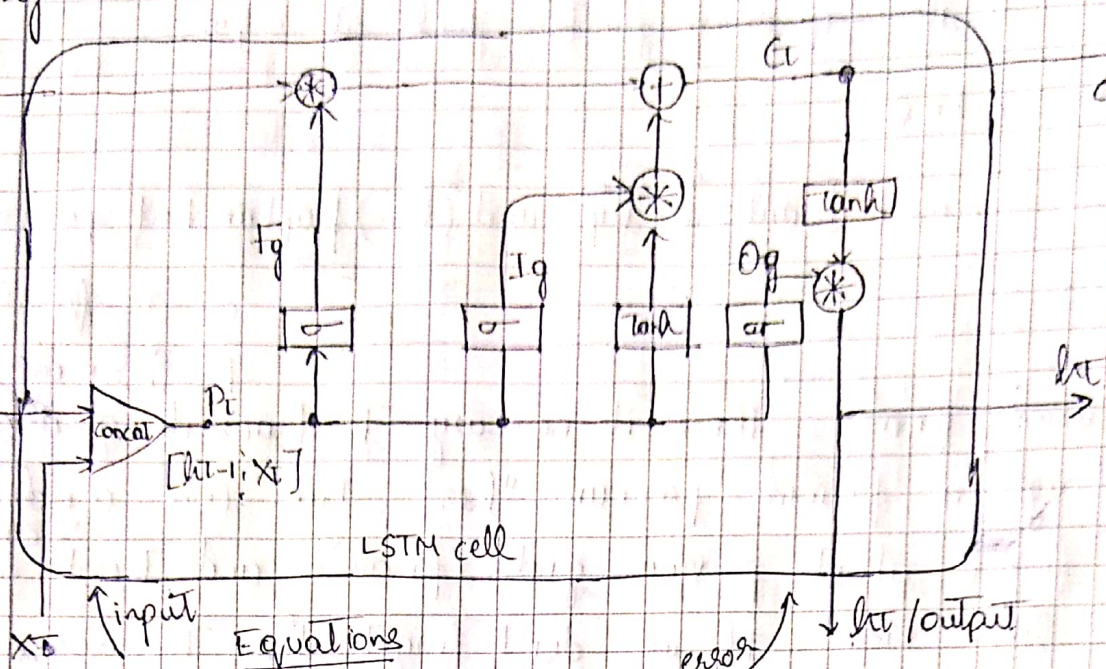
Complete LSTM blocks



→ Along with hidden state we have a cell state and cell state pathway forms constant carousel of errors to prevent errors from blowing / dieying

→ Here, actually cell state carries past summary and history, whilst hidden state acts as output and also links to next state. Lets look inside a cell

History



Equations

$P_t = \text{concat}\,[h_{t-1}, x_t]$ — (1)

$F_{gate} = \sigma\,(W_F P_t + b_F)$ — (2)

$In_{gate} = \sigma\,(W_I P_t + b_I)$ — (3)

$Out_{gate} = \sigma\,(W_o P_t + b_o)$ — (4)

$\tilde{C}_t = \tanh\,(W_c P_t + b_c)$ — (5)
($\tilde{C}_t$: new cell update value)

$C_t = F_{gate} * C_{t-1} + In_{gate} * \tilde{C}_t$ — (6)
(C_t: updated cell state)

$O_t = \tanh\,(C_t)$ → (7)

$h_t = Out_{gate} * O_t$ → (8)

Explanation

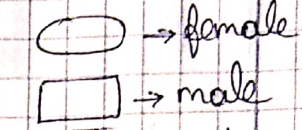
→ We concatenate both h_{t-1} & x_t else we may require two weight matrix to train instead of one for every gate.

Forget gate : Learn the gate values, which are from sigmoid distribution to selectively allow only parts of history to move forward. Remove unwanted part of history.

→ This also helps to keep linear sum C_t within certain limits so that it doesn't saturate squashing function tanh at output.

Example:- Gender Tracking

Black pearl is the most beautiful ship, and her captain Jack Sparrow is a king of pirates.



→ WKT ships are usually female. Hence it is necessary to track it's gender or confirm it though a feminine pronoun "her". But once a male noun is introduced we need to forget about previous gender (female) and track for male noun/pronoun like "king". If we don't forget then this may add irrelevant history and noise thereby diluting our prediction process.

Example:- female female male female

Queen Anne's Revenge is a fearsome ship and Captain Teach is her Captain, Captain Teach is dad of his first mate angelica

female

→ Without forget gate we have transition from female (ship) → male (captain) → female (angelica) we end up doing wrong prediction & may predict all as female

Input gate + Updatedable Cell state value
i_{gate} $(\tilde{C}_t)$

a) $\tilde{C}_t$: Updatable cell state values calculates the new values that it should output based on previous output h_{t-1} & current input X_t

b) Input gate :- Controls the flow of new updates to cell stream, w/o which we may add unwanted activations/noise and screw up cellstate having past history.

Example:- From previous gender tracking example

Black pearl is the most beautiful ship
relevant irrelevant relevant activation

→ Adding activations of "is" the most" for my last makes previous cell state "Black pearl" noisy. It is like shouting my own feel. Don't update cell state for the activations of these tokens using Input gate.

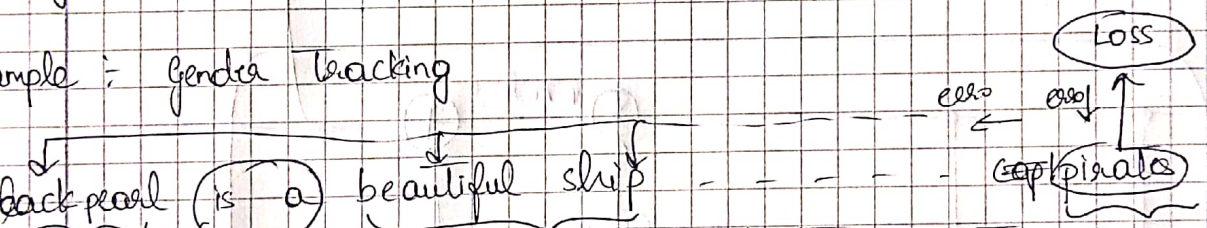
Og Output gate ~~and~~ / New cell state / Output state h_t
C_t

a) ⇒ Armed with the information of what to forget about past and what relevant info we must add for current step we update existing cell state using this combo

$$C_t = \underbrace{F_{gate} * C_{t-1}}_{\text{what to forget about past}} + \underbrace{I_{gate} * \tilde{C}_t}_{\text{what relevant current info should be added}}$$

b) ⇒ Output gate : Output gate helps in shielding against irrelevant errors during back prop. This also helps to output current steps data (h_t)

→ Example : Gender tracking



error updates should happen only at relevant time steps

irrelevant

in forward prop

** → "is a" did not add any gender info ∴, hence useless to be update during back prop. This also shields against unnecessary updates and error can travel much longer.

c) Output state : The output of current step LSTM cell based on current cell state and output gate.

Why Tanh ? Tanh is centered around zero & squashed b/w $[-1, 1]$ hence its gradient is a bit stronger than sigmoid.

why sigmoid ingates ? Gates are similar to boolean logic (0, 1), Sigmoid fits this description.

Other LSTM variants

Peephole :- Include cell state and hidden states from previous step with current i/p x_t to calculate all gates & updates

Gated Recurrent Units :- Simpler design, with a single gate for both update and forget, No separate cell state, CEC unchanged / carousel

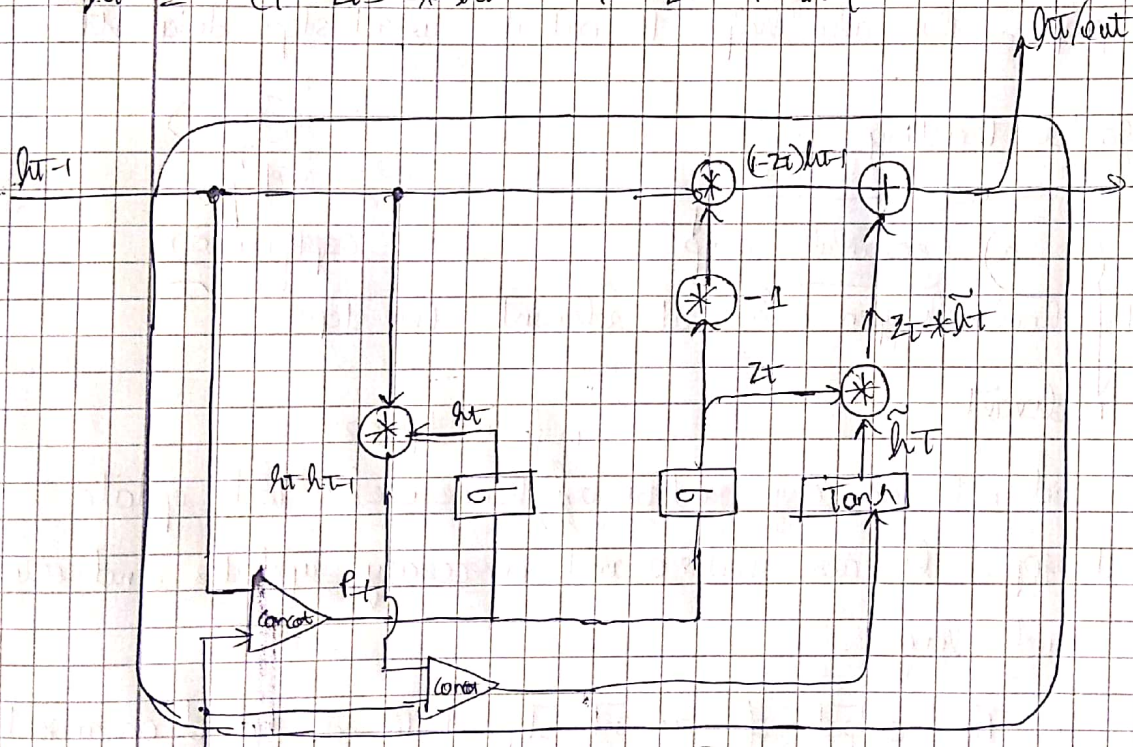
$P_t = \text{concat}\ [x_t, h_{t-1}]$

(update) $z_t = \sigma\ (W_z\ P_t + b_z)$

(reset) $r_t = \sigma\ (W_r\ P_t + b_r)$

$\tilde{h}_t = \tanh\ (W_h\ [r_t * h_{t-1}; x_t] + b_h)$

$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$



How to initialize gates?

Forget gates :- Initialize to near one, so that system can gradually learn what to forget

Input gates :- Initialize near zero, so system can increase weights for relevant steps later

Output gate :- Initialize near — — — — — relevant steps later