DL- 06/03

Matrix calculus

dependent ← $Y = f(X)$ → independent

↳ can be scalar/ vector/ matrix

Let $y = f(x, y, z)$

$$\nabla f = \frac{\partial f}{\partial x}\hat{x} + \frac{\partial f}{\partial y}\hat{y} + \frac{\partial f}{\partial z}\hat{z} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{bmatrix}^T$$

① Derivative of a vector wrt scalar,

Let $y \in \mathbb{R}^m$ wrt scalar $x \in \mathbb{R}$

$$y = [y_1 \; y_2 \cdots y_m]^T$$

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} & \frac{\partial y_2}{\partial x} \cdots \frac{\partial y_m}{\partial x} \end{bmatrix}^T$$

② Derivative of a scalar wrt vector*

scalar $y \in \mathbb{R}$ wrt vector $x \in \mathbb{R}^m$

$$x = [x_1 \; x_2 \cdots x_m]^T$$

$$\frac{\partial y}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_m} \end{bmatrix}$$

(3) Derivative of a vector wrt vector*

Let $y \in \mathbb{R}^m$ & $x \in \mathbb{R}^n$

$$y = [y_1 \; y_2 \cdots y_m]^T \qquad x = [x_1 \; x_2 \cdots x_n]^T$$

$$\frac{\partial y}{\partial x} = (\text{Jacobian matrix}) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

(4) Derivative of a matrix by a scalar

Let $Y \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}$ is scalar

$$Y = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix} = \{y_{ij}\}_{m \times n}$$

$$\frac{\partial Y}{\partial x} = \begin{bmatrix} \partial y_{11}/\partial x & \partial x_{12}/\partial x & \cdots & \partial y_{1n}/\partial x \\ \partial y_{21}/\partial x & \partial y_{22}/\partial x & \cdots & \partial y_{2n}/\partial x \\ \partial y_{m1}/\partial x & \partial y_{m2}/\partial x & \cdots & \partial y_{mn}/\partial x \end{bmatrix}$$

* Numerator layout : Follow the dimension of numerator. If denominator has a dimension, the overall derivative is transposed, with dim. of denominator.

(5) Derivative of a scalar wrt matrix*

$y \in \mathbb{R}$ is scalar & $X \in \mathbb{R}^{m \times n}$ is a matrix

$$\frac{\partial y}{\partial X} = \begin{bmatrix} \frac{\partial y}{\partial x_{11}} & \frac{\partial y}{\partial x_{21}} & \cdots & \frac{\partial y}{\partial x_{m1}} \\ \frac{\partial y}{\partial x_{12}} & \frac{\partial y}{\partial x_{22}} & \cdots & \frac{\partial y}{\partial x_{m2}} \\ \vdots & \vdots & & \vdots \\ \frac{\partial y}{\partial x_{1n}} & \frac{\partial y}{\partial x_{2n}} & \cdots & \frac{\partial y}{\partial x_{mn}} \end{bmatrix}$$

Identities

(1) $y = Ax \Rightarrow \frac{\partial y}{\partial z} = A \frac{\partial x}{\partial z}$

(2) $\alpha = y^T x = [x_1 y_1 + x_2 y_2 + \ldots + x_n y_n]$

$$\frac{\partial \alpha}{\partial z} = \left[\left(x_1 \frac{\partial y_1}{\partial z_1} + y_1 \frac{\partial x_1}{\partial z_1} \right) + \left(x_2 \frac{\partial y_2}{\partial z_2} + y_2 \frac{\partial x_2}{\partial z_2} \right) + \ldots \right.$$

$$= \left(x_1 \frac{\partial y_1}{\partial z_1} + x_2 \frac{\partial y_2}{\partial z_2} + \ldots \right) + \left(y_1 \frac{\partial x_1}{\partial z_1} + y_2 \frac{\partial x_2}{\partial z_2} + \ldots \right)$$

$$\Rightarrow \frac{\partial \alpha}{\partial z} = x^T \frac{\partial y}{\partial z} + y^T \frac{\partial x}{\partial z}$$

Gradient Descent

Loss fun^n $J(f(x^{(i)};\theta), y^{(i)})$

$\theta^{(t)}$

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon \nabla_\theta J$$

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} J(f(x^{(i)}, \theta), y^{(i)})$$

Cons - computation time ↑↑
↳ for each update, all points in dataset are to be considered & corresponding losses are added

Sol^n

Stochastic GD
↳ select 1 instance randomly & use it for updating θ, let $x^{(2)}$ be the random instance

$$\mathcal{L} = \frac{1}{N} J(f(x^{(2)}, \theta), y^{(2)})$$

Cons
↳ each iteration, $\mathcal{L}$ is highly fluctuating leading to slow convergence

Sol^n → Mini-Batching

↳ select m instances (sample) instead of whole N

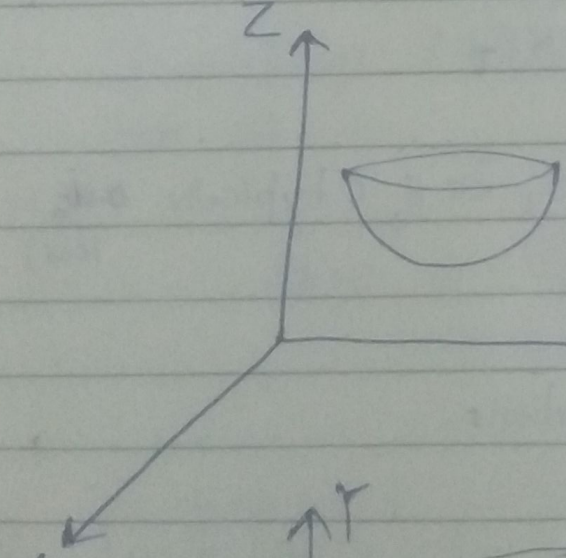
$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} J(f(x^{(i)}, \theta), y^{(i)})$$

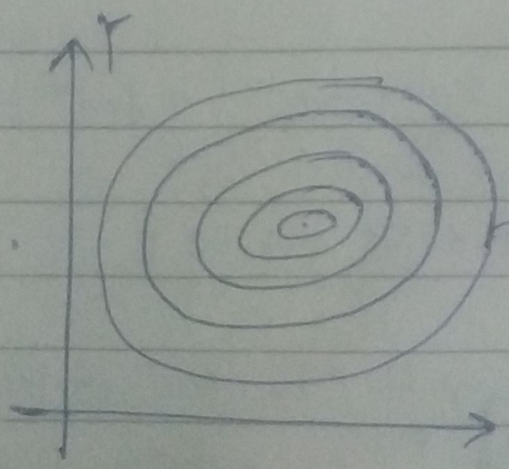
Mini-batch = $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$

ε — bottleneck

↳ ε ↑ or const. overshooting the minima

↳ ε ↓ slow convergence





Level surface Diagram (LSD)

each pt. on this curve has ~~same~~ same values of z

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau \quad ; \quad \alpha = k/\tau$$

DL - 09/03

Mini-Batch GD $\quad \hat{g} = \frac{1}{m} \sum_{i=1}^{m} J(f(x^{(i)}, \theta), y^{(i)})$

$$\theta^{(t+1)} = \theta^{(t)} - \varepsilon \hat{g}_t$$

Sufficient cond^n on ε for the convergence of GD,

$$\begin{cases} \sum_{k=1}^{\infty} \varepsilon_k = \infty \\ \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty \end{cases} \quad \&$$

→ ε_k should be fractional ; k is arbitrary

Common practice, — use an adaptive (decaying) value of ε

$$\varepsilon_k = (1-\alpha)\,\varepsilon_0 + \alpha \varepsilon_\tau$$

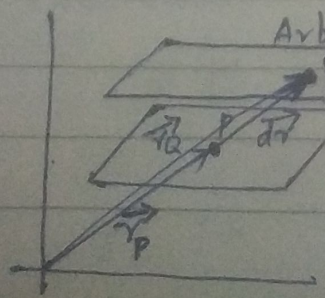
where $\alpha = k/\tau$, $\varepsilon_\tau \ll \varepsilon_0$ (typically $0.01\varepsilon_0$ or less)

k = nö of iteration
τ = total nö of iterations

Let $\phi(x,y,z)$ be a scalar func^n (a func^n with scalar output)

$$\vec{\nabla} = \partial/\partial x\,\hat{i} + \partial/\partial y\,\hat{j} + \partial/\partial z\,\hat{k}$$

$$\vec{\nabla}\phi = \frac{\partial \phi}{\partial x}\hat{i} + \frac{\partial \phi}{\partial y}\hat{j} + \frac{\partial \phi}{\partial z}\hat{k}$$



shifting position of P by $\vec{dr}$, we get Q in another level surface. So surface is $\phi + d\phi$

Let $\overrightarrow{OP} = \vec{r}$, $\overrightarrow{OQ} = \vec{r} + d\vec{r}$

Then shifting $\vec{r}$ to $\vec{r} + d\vec{r}$ shifts the level surface from ϕ to $d\phi$.

$d\phi$ = change in level surface

$$d\phi = \frac{\partial \phi}{\partial x} dx + \frac{\partial \phi}{\partial y} dy + \frac{\partial \phi}{\partial z} dz$$

$$\Rightarrow d\phi = \left(\frac{\partial \phi}{\partial x} \hat{i} + \frac{\partial \phi}{\partial y} \hat{j} + \frac{\partial \phi}{\partial z} \hat{k} \right) \cdot (dx \hat{i} + dy \hat{j} + dz \hat{k})$$

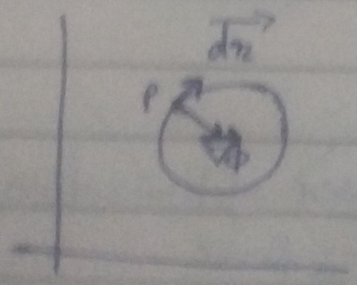
$$\Rightarrow \boxed{d\phi = \vec{\nabla}\phi \cdot d\vec{r}}$$

If we change P to Q s.t Q is on same LS as P, then $d\phi = 0$.

$$d\phi = 0 \Rightarrow \vec{\nabla}\phi \cdot d\vec{r} = 0$$

$\Rightarrow \vec{\nabla}\phi$ & $d\vec{r}$ are $\perp$ar to each other

Thus, gradient at P is $\perp$ar to $d\vec{r}$ (tangent to LS at P)



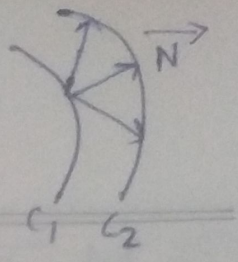
$\vec{\nabla}\phi$ points in a direction normal to the LS.

$$\boxed{\vec{\nabla}\phi = |\vec{\nabla}\phi| \cdot \hat{\nabla}\phi}$$

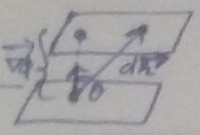
$$\hat{\nabla}\phi = \hat{N}$$

↳ unit vector

θ = angle b/w $\vec{\nabla\phi}$ & $\vec{dx}$



(normal)
min^m dist b/w the 2 LS = $\vec{dN} = \underline{dx\cos\theta}$ ←



$$d\phi = \vec{\nabla\phi} \cdot \vec{dx}$$

$$\Rightarrow d\phi = |\vec{\nabla\phi}| \cdot |\vec{dx}| \cdot \cos\theta^*$$

$$\Rightarrow d\phi = |\vec{\nabla\phi}| \cdot \vec{dN} \Rightarrow \boxed{\frac{d\phi}{\vec{dN}} = |\vec{\nabla\phi}|}$$

$\vec{dN}$ is the normal vector

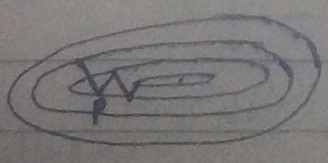
Gradient is the direction of max^m change, ie, the direction in which ϕ changes most rapidly

For minimization, one must come in direction opposite to gradient for steepest descent. That's why we use $-\hat{g}_t$ in GD, with ε for its own reasons.

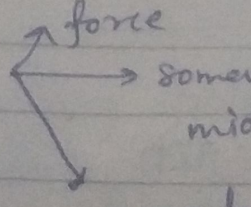
Challenges to GD

- choice of proper ε
  - $\varepsilon\downarrow$ slow convergence, many iterations
  - $\varepsilon\uparrow$ overshooting minima, oscillations
- predefined ε for all directions
- saddle points – slope (+ve) in 1 direction & (-ve) in 1 direction

Handling oscillations → momentum-based optimization techniques

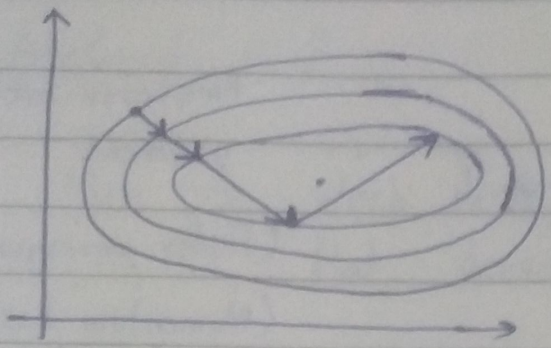


at P,



* In DL, random initialisation is seldom. Pre-trained set of parameters is used instead. However, in theory (here), it may be considered random.

DL-10/03



Challenge - oscillating convergence

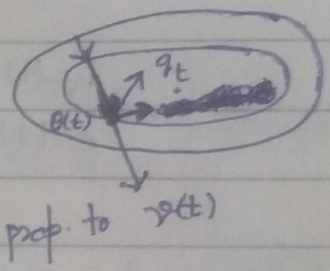
$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \varepsilon g_t$$

Sol^n - momentum-based technique

Assume that there is a momentum given by $v(t)$,

$$v(t) \leftarrow \alpha v(t-1) - \varepsilon g_t$$

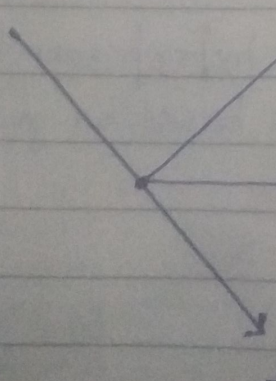
$$\theta^{(t+1)} \leftarrow \theta^{(t)} + v(t)$$



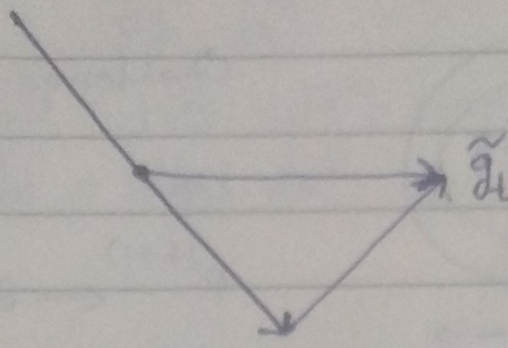
Momentum-based optimizers handle oscillations better.

$\theta^{(0)} \leftarrow$ randomly initialized parameter *

Nesterov acc^n gradient



Simple Momentum-based optimization



Nesterov accl^n gradient

(Faster convergence)

(Momentum based)

(intermediate set of parameters, θ & $\tilde{g}_t$)

$$\tilde{\theta}(t) = \tilde{\theta}(t-1) + \alpha v(t-1)$$

$$\tilde{g}_t = \frac{1}{m} \sum_{i=1}^{m} J(f(x^{(i)}, \tilde{\theta}), y^{(i)})$$

$$v(t) = \alpha v(t-1) - \varepsilon \tilde{g}_t$$

$$\theta(t+1) = \theta(t) + v(t)$$

Challenges to momentum-based optimizers

↳ hyperparameters α & ε determine learning rate

↳ uses same values of α & ε across all dimensions

↳ better if tuned a/c to gradient

Sol^n - adaptive hyperparameter settings

↳ AdaGrad

↳ tries to adapt hyperparameter values across dimensions based on gradient

* element-wise product ($\odot$)

let $g_\tau \in \mathbb{R}^m$ then $g_\tau \odot g_\tau = \begin{bmatrix} g_1^2 \\ g_2^2 \\ g_m^2 \end{bmatrix}$ & so on.

AdaGrad

Idea → steep descent ⇒ low values for hyperparameters

flat descent ⇒ high " " "

$$g_t = \frac{1}{m} \nabla_\theta J(f(X, \theta^{(t)}), Y)$$

$$r_t = \sum_{\tau=1}^{t} g_\tau \odot g_\tau$$

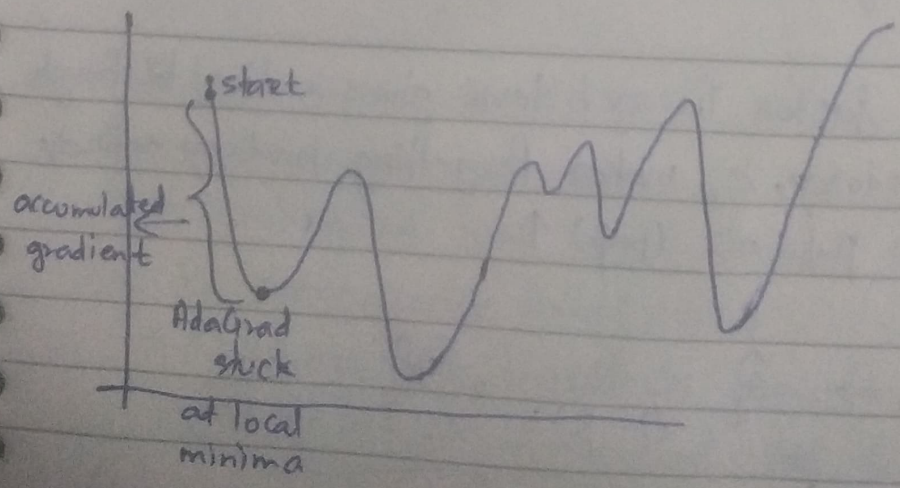
$\theta^{(t+1)} = \theta^{(t)} - \eta g_t$ is now changed to,

$g_t = \begin{bmatrix} g^{(1)} \\ g^{(2)} \\ g^{(m)} \end{bmatrix}$ changes to

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta g_t}{\sqrt{\varepsilon I + r_t}}$$

$$g_t = \begin{bmatrix} g^{(1)}/\sqrt{\varepsilon + r_1} \\ g^{(2)}/\sqrt{\varepsilon + r_2} \\ \vdots \\ g^{(m)}/\sqrt{\varepsilon + r_m} \end{bmatrix}$$

Challenge → gets stuck in local minima as it accumulates gradient over time



Improvement - discard the history from the extreme past

⇓

RMS Prop

RMS Prop

$$g_t = \frac{1}{m} \sum \nabla_\theta J(f(X, \theta^{(t)}), Y)$$

$$r_{t+1} = \beta r_t + (1-\beta) s_t \quad ; r_0 = 0$$

$$s_t = \sum_{\tau=1}^{t} g_\tau \circ g_\tau \qquad \theta^{(t+1)} = \theta^{(t)} - \frac{\eta g_t}{\sqrt{\epsilon I + r_t}}$$

$$r_1 = (1-\beta) s_0 \,, \quad r_2 = \beta(1-\beta) s_0 + (1-\beta) s_1$$
$$r_3 = \beta^2(1-\beta) s_0 + \beta(1-\beta) s_1 + (1-\beta) s_0$$

If $\beta \downarrow$ priority is given to the s_t values which are closer to current time, ie, history gets priority lessened over time. All game is in powers of β getting small & small as β is small.

The $(1-\beta)$ factor in each term gives a bias, though, to the history, ie, makes forgetting history entirely difficult. $\beta \downarrow \Rightarrow (1-\beta) \uparrow$

Correction → $\hat{s}_t = \frac{s_t}{(1-\beta)}$, $\hat{r}_t = \frac{r_t}{(1-\beta)}$

Idea behind AdaM (Adaptive Moment) Technique

AdaM optimizer

$$s_t = \beta_1 s_{t-1} + (1-\beta_1) g_t$$

$$r_t = \beta_2 r_{t-1} + (1-\beta_2) g_t \cdot g_t$$

Corrections

$$\hat{s}_t = \frac{s_t}{(1-\beta_1)}$$

$$\hat{r}_t = \frac{r_t}{(1-\beta_2)}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta \hat{s}_t}{\sqrt{\epsilon I + \hat{r}_t}}$$

DL - 16/03

Batch Normalization

↳ not an optimization technique, booster of optimization process indeed (like, catalyst)

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} J(f(x^{(i)}, \theta), y^{(i)})$$

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \nabla_\theta J$$

Consider mini-batch $B = \{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$

- → mini-batch GD
- → compromise b/w computation complexity (GD) & convergence time (SGD)
- → challenge - mini-batch may n't be a representative of actual set of inputs

Internal Covariance shift

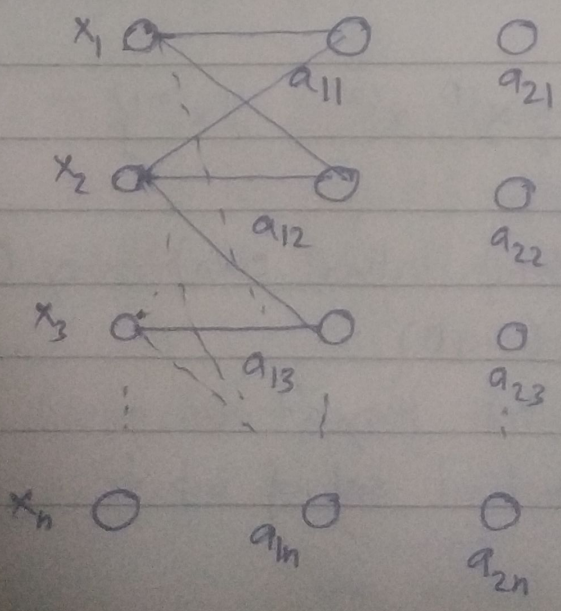
- when the distribution of input features varies across mini-batches
  - (in other words) mini batches fail to represent actual input effectively
- may lead to slow convergence

Soln – (proposed by Ioffe & Szegedy)

- normalization may lead to stable & representative mini-batches
- normalization at each layer of NN may help in better propagation of info

Batch Normalization (BN)

- would help if the pre-activations at each layer are normalized
- ensured by standardizing the pre-activations at each unit



$$\hat{a}_{ik} = \frac{a_{ik} - E[a_{ik}]}{SE[a_{ik}]}$$

E → expected value

SE → standard error

a_{ik} → BN → $z_{ik} = \text{activation func}^n(a_{ik})$

* local pixels may convey more meaningful info
** objects in motion appear different from those at rest
*** an object is built up from smaller sub-objects or compositions
**** stride is the no of steps CM is strided on input matrix

a_{ik} has been shifted and scaled which needs to be rescaled & retransformed for getting an output.

$$y^{(k)} = \gamma^{(k)} \hat{a}_{ik} + \beta_k^{(k)}$$

$\gamma^{(k)}$ & $\beta^{(k)}$ need to be learnt & aren't hyperparameters.

DL - 20/03

Image properties
- Locality*
- Stationarity**
- Hierarchy***

CM ≡ kernel

CNN
- convolution layer

NXN input matrix, FXF predefined convolution matrix (CM)
It reduces NXN ⟶ (N-F+1) X (N-F+1) for 1 stride. If stride=s, $\left(\frac{N-F}{s}+1\right) \times \left(\frac{N-F}{s}+1\right)$ is the resultant matrix

$$\cos\theta = \frac{\vec{u}\cdot\vec{v}}{|\vec{u}|\cdot|\vec{v}|}$$

$\vec{u}$ & $\vec{v}$ are similar when $\cos\theta \uparrow$ or $\theta \to 0$

⇓

Underlying principle of cosine similarity

further insights paper ref.,
Group equivariance in CNNs

So the convolution operation which is a dot product of kernel with the image, basically tries to identify parts of image similar to kernel. Multiple kernels can be used. Trained kernels (Learnt) are better than predefined ones.

o to i

The no of units in layer i affecting a unit in layer i+1 is the receptive field of unit in layer (i+1).

Due to sparse connectivity in CNN, receptive field goes vulnerable. To accommodate, the network must be deep, otherwise info will be lost.

Invariance

↳ If I is an object, g(I) is translation of I, then f is said to be translationally invariant if $f(g(I)) = f(I)$

Equivariance*

↳ If I is an object, g(I) is translational of I, then f is said to be translationally equivariant if $f(g(I)) = g(f(I))$

Convolution is partially invariant as well as partially equivariant.

DL-23/03

Equivariance - shift in object position in image doesn't affect the result

Invariance - slight perturbations in image doesn't affect the result

ReLU is an effective activation func^n as tanh & sigmoid tend to have saturated gradient values for larger values.

Pooling → dimensionality red^n
↳ additionally to maintain invariance property

Maxpool vs Avgpool → maxpool better
↳ better invariance

Pooling comes after convolution as convolution weighs the important parts of the image. If pooling is done earlier, it would just identify the more intense parts of the image which may not be significant at all.

Padding → borders are under-represented in convolution
↳ to maintain actual dimensions

Let p be the padding size. Stride value = s, input size n and filter size f.
output size = $(n+2p-f+1) \times (n+2p-f+1)$

If output size is to be same as input size,

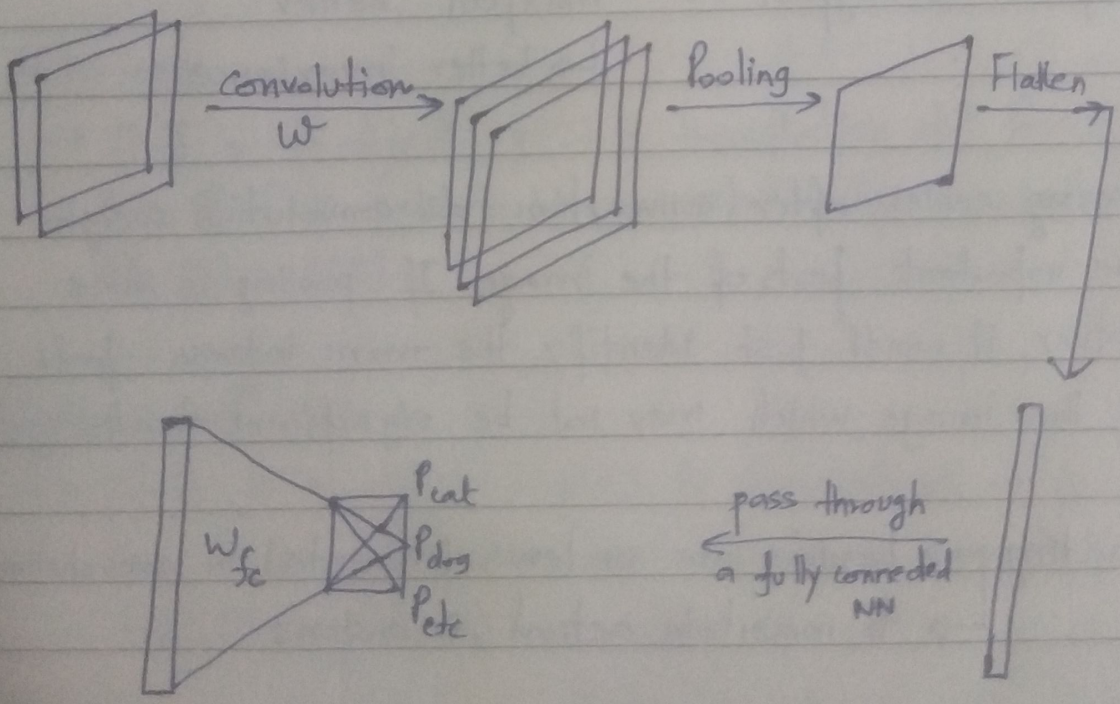
$$n + 2p - f + 1 = n$$

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

With stride = s,

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

$$\Rightarrow \boxed{p = \frac{ns - (n+s) + f}{2}}$$



$$\mathcal{L} = \max(x_1, x_2) \rightarrow \text{(maxpool)}$$

$$\frac{\partial \mathcal{L}}{\partial x_1} = \begin{cases} 1 & \text{if } x_1 > x_2 \\ 0 & \text{otherwise} \end{cases}$$

DL - 24/03

AlexNet Architecture

Input : 227 x 227 x 3

Conv. 1 : 96 filters of dimension 11 x 11 x 3, stride = 4

Output size : $\frac{227 - 11}{4} + 1 = 55 \times 55 \times 96$

Parameters : (11 x 11 x 3) x 96

Max pool : 3x3 filters of stride 2

Output size : $\frac{55-3}{2} + 1 = 27 \times 27 \times 96$

Parameters : 0

Normalization : ------ $(x-H)/\sigma$

Conv. 2 : 256 filters of dimension 5x5x96, stride=1, Padding = 2

Output size : $\frac{27-5+4}{1} + 1 =$ 27 x 27 x 256

Parameters : (5 x 5 x 96) x 256

Max pool : 3x3 filters of stride 2

output size : $\frac{27-3}{2} + 1 = 13 \times 13 \times 256$

Parameters : 0

Normalization : ----- $(x-H)/\sigma$

Conv. 3 : 384 filters of dimension 3x3x384, stride=1, pad=1

Output size : 13+2-3+1 = 13 x 13 x 384

Parameters : (3 x 3 x 256) x 384

Conv. 4 : 384 filters of dimension 3x3x384, stride=1, pad=1

Output size : 13 x 13 x 384

Parameters : (3 x 3 x 384) x 384

* 1000 classes of ~~dataset~~ images in dataset

Conv. 5 : 256 filters of dimension $3 \times 3 \times 384$, pad = 1, stride = 1
Output size : $13 \times 13 \times 256$
Parameters : $(3 \times 3 \times 384) \times 256$
Max pool : 3×3 filters of stride 2
Output size : $\frac{13-3}{2} + 1 = 6 \times 6 \times 256$

Parameters : 0
Fully-Connected Layer 1 : 4096 neurons
Output size : 4096 Parameters : 9216×4096
Fully-Connected Layer 2 : 4096 neurons
output size : 4096 Parameters : 4096×4096
Fully-Connected Layer 3 : 1000 neurons
Output size : 1000* Parameters : 4096×1000

Total 60M parameters, no of parameters ↑↑
ReLU activation is used after every convolution.

Challenges - performance

Improvement - VggNet
- → made the network deeper to boost accuracy & performance (kernel size↓)
- → parameters learned at each layer is less, but as depth ↑ total no of parameters ↑ (Parameters = 138M)
- → receptive field ↑ for ~~deeper layer~~ whole network, helps extracting complex features

* Perception & observation aren't complete. We perceive an image at different scales at once & try to augment our observations to learn what the image tries to convey.

DL - 25/03

Complexity of algorithms like VggNet & AlexNet is measured by no of Floating-Point operations (FLOPs).

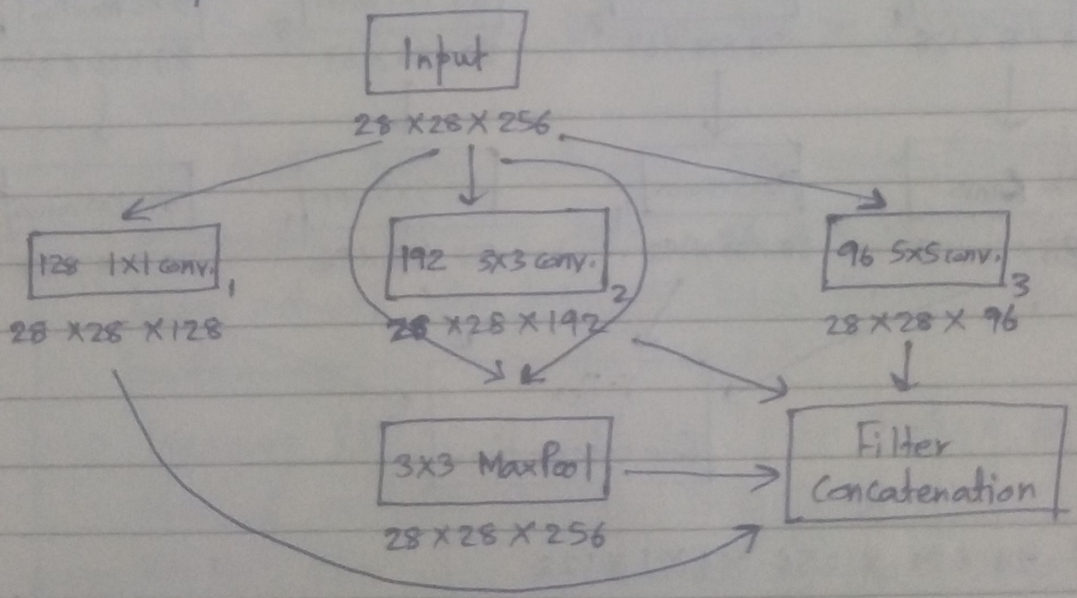
VggNet FLOPs ↑

GoogleNet objective → to reduce FLOPs & achieve comparable performance as that of VggNet

↓

based on the idea of Inception

Naive Inception*



Output size: 28 X 28 X (128 + 192 + 96 + 256)

= 28 X 28 X 672

Considering only MULTIPLICATIONS, total no of operations; (MULTs) (FLOPs)

$28 \times 28 \times 256 \times 1 \times 1 \times 128$ for conv. 1

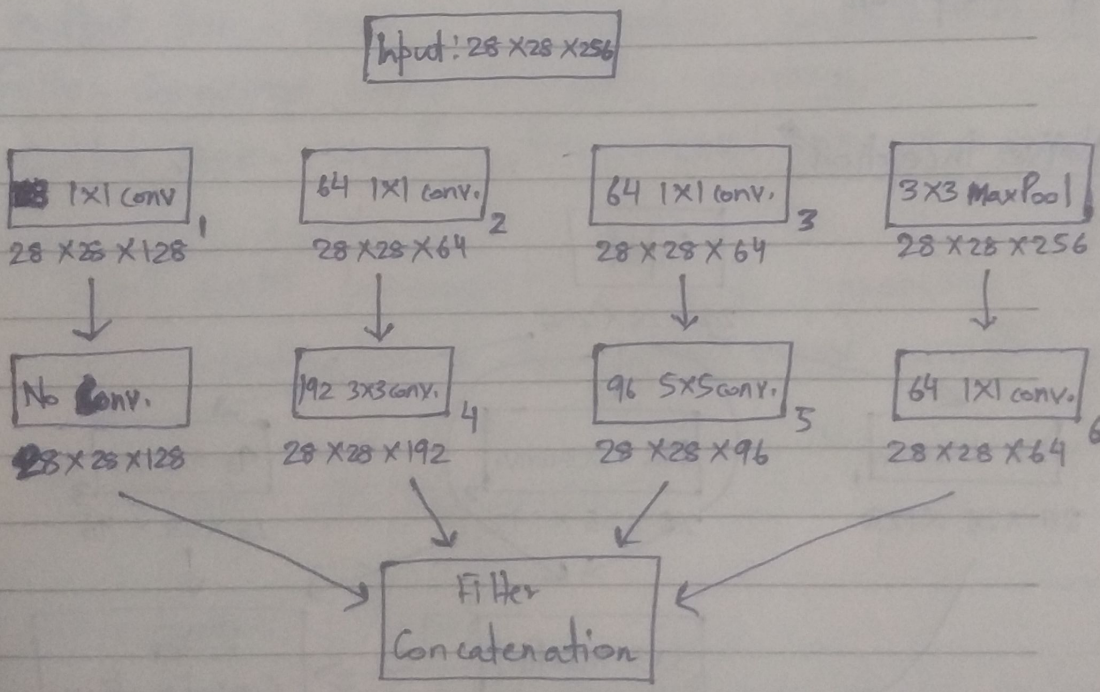
$28 \times 28 \times 256 \times 3 \times 3 \times 192$ for conv. 2

$28 \times 28 \times 256 \times 5 \times 5 \times 96$ for conv. 3

Max Pool doesn't require significant no of MULTs.

Total FLOPs $\approx$ 854 M which is very high.

GoogleNet introduced a bottleneck layer to naive inception to reduce FLOPs.



C_1) $28 \times 28 \times 256 \times 1 \times 1 \times 128$

C_2) $28 \times 28 \times 256 \times 1 \times 1 \times 64$

C_3) $28 \times 28 \times 256 \times 1 \times 1 \times 64$

C_4) $28 \times 28 \times 64 \times 3 \times 3 \times 192$

C_5) $28 \times 28 \times 64 \times 5 \times 5 \times 96$

C_6) $28 \times 28 \times 256 \times 1 \times 1 \times 64$

Total FLOPs $\approx$ 358 M

* paper on ResNet by Microsoft scientists, reference

** ResNet-1202 had error % of 7.93, more than ResNet-110 due to overfitting.

GoogleNet

- tries to visualise an image the same way a human does
- reduces no of FLOPs than naive inception using bottleneck layer

ResNet *

- assumption of a deeper network performing better is contradicted
- 56-layer network performed worse than a 20-layer network experimentally & it was not due to overfitting as training error was also worse for 56-layer network
- conclusion was that problem lies in initialisation values
- also that deeper networks are uncomfortable in learning identity func^n, as if in the 56-layer network, the last 36 learn $f(x)=x$, the performance of both would match
- learning $F(x)=0$ is feasible than $F(x)=x$, idea of residual learning introduced
- ResNet idea was implemented & the results were now as expected. ResNet-34 performed better than ResNet-18 on both validation & generalization
- ResNet performed better & error went down with increasing depth, ResNet-110 is min^m with 1.7M parameters & 6.43% error **

* "I went to IIT in year 2017" & "In 2017, I went to IIT" are same but CNN would fail in this model to identify them similar.

DL - 27/03

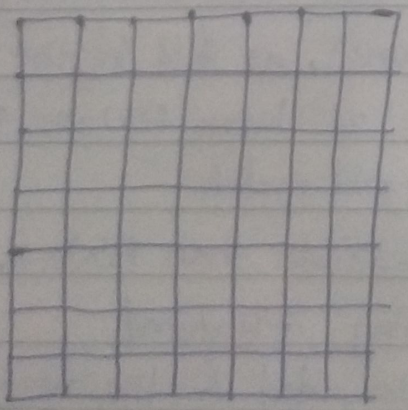
Recurrent NN (RNN)

Text comprehension,

Model → convert each word to a fixed-length vector & concatenate to form an image, apply CNN on the image for perception

I →
went → "
to → "
IIT → "
in → "
year → "
2017 → "

Vector conversion follows a certain ruleset & depends on many factors including the nearby words conveying the context.

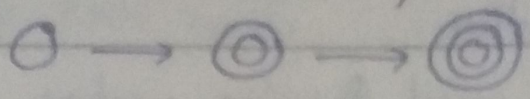


→ apply CNN

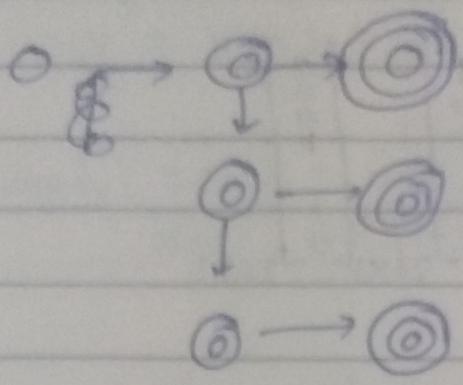
Problem

- → Texts can be indefinitely long arbitrarily
- → there may be non-local strong dependencies
- → grammatically & orientationally different texts may convey same meaning *

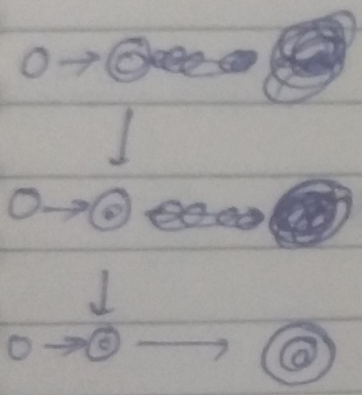
Common architectures,



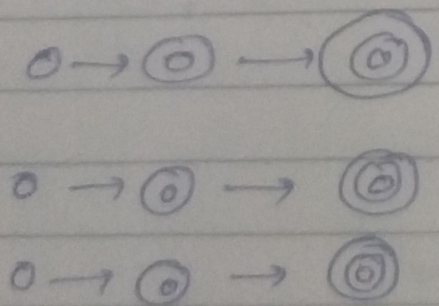
Vector conversion of word



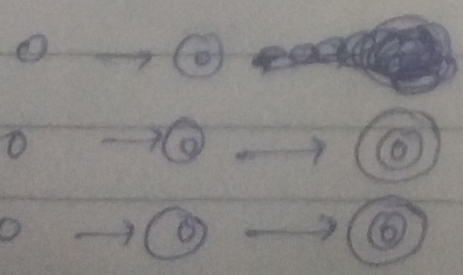
Text generation



sentiment classification



speech to text



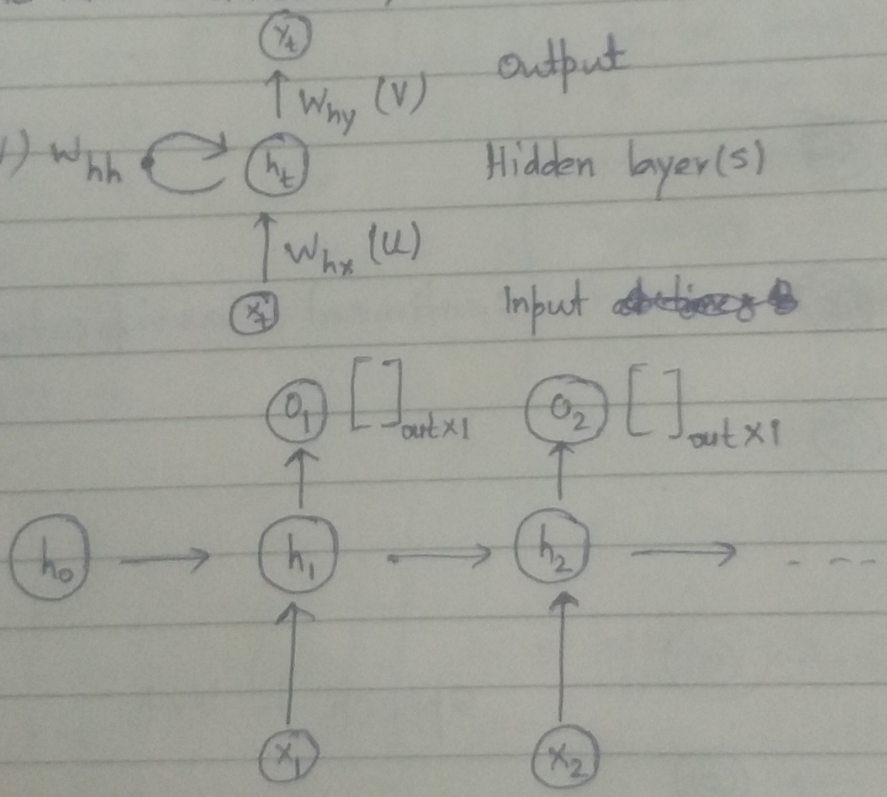
Machine translation

One-hot encoding / representation

↳ representing a word using vectors s.t only 1 of the elements is 1 & rest 0

ex. bad → $\begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix}_{26\times1}$ $\begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}_{26\times1}$ $\begin{bmatrix}0\\0\\0\\1\\\vdots\\0\end{bmatrix}_{26\times1}$

Basic RNN architecture,



$$E_t = \frac{1}{2}\sum_{k=1}^{out}(\hat{y}_k - y_k)^2 \quad \text{for 1 layer}$$

$$E = \frac{1}{2}\sum_{t=1}^{\tau}\sum_{k=1}^{out}(\hat{y}_{tk} - y_{tk})^2$$

Basic RNN equ^{n}s,

$$h_t = Ux_t + W\phi(h_{t-1})$$
$$y_t = V\phi(h_t)$$

Gradient, $\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdot \cdots \frac{\partial h_{k+1}}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}$$

DL - 30/03

$$\mathcal{L} = ab \Rightarrow \frac{\partial \mathcal{L}}{\partial z} = a^T \frac{\partial b}{\partial z} + b^T \frac{\partial a}{\partial z}$$

$$h_i = Ux_t + W\phi(h_{i-1})$$
$$\Rightarrow \frac{\partial h_i}{\partial h_{i-1}} = W^T \frac{\partial \phi(h_{i-1})}{\partial h_{i-1}}$$

$$\phi(h_{i-1}) = \begin{bmatrix} \phi_{h_{i-1}}^{(1)} \\ \phi_{h_{i-1}}^{(2)} \\ \vdots \end{bmatrix}$$

$$h_i = \begin{bmatrix} h_i^{(1)} \\ h_i^{(2)} \\ \vdots \\ h_i^{(k)} \end{bmatrix}$$

$$\frac{\partial \phi(h_{i-1})}{\partial h_{i-1}} = \begin{bmatrix} \frac{\partial \phi_{h_{i-1}}^{(1)}}{\partial h_{i-1}^{(1)}} & \frac{\partial \phi_{h_{i-1}}^{(1)}}{\partial h_{i-1}^{(2)}} & \cdots \\ \frac{\partial \phi_{h_{i-1}}^{(2)}}{\partial h_{i-1}^{(1)}} & \frac{\partial \phi_{h_{i-1}}^{(2)}}{\partial h_{i-1}^{(2)}} & \cdots \\ \frac{\partial \phi_{h_{i-1}}^{(3)}}{\partial h_{i-1}^{(1)}} & \cdots & \cdots \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \phi'^{(1)}_{h_{i-1}} & 0 & \cdots \\ 0 & \phi'^{(2)}_{h_{i-1}} & \cdots \\ 0 & 0 & \cdots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

So, $\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} W^T \text{diag}[\phi'(h_{i-1})]$

If $\frac{\partial h_i}{\partial h_{i-1}} < 1 \quad \forall i, \frac{\partial h_t}{\partial h_k} \to 0$.

↳ Problem of Vanishing Gradients
- ↳ RNN stops learning at a point
- ↳ ~~How we~~ weights aren't updated, non-optimal weights remain

If $\frac{\partial h_i}{\partial h_{i-1}} > 1 \quad \forall i, \frac{\partial h_t}{\partial h_k} \to \infty$.

↳ Problem of Exploding Gradients
- ↳ RNN weights updated drastically
- ↳ can be handled by clipping gradients by a threshold, called gradient clipping

Sol^n to Vanishing Gradients
↳ do back propagation over smaller nō of steps
↳ problem – error is also propagated.

$$h_t = \underset{m\times 1}{Ux_t} + \underset{m\times 1}{W\phi(h_{t-1})}$$

new input $I = \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \in \mathbb{R}^{2m}$ $\quad W_a^{\phi} = \begin{bmatrix} & W & U & \end{bmatrix}$

$2m \times 1$ $\qquad\qquad m \times 2m$

ϕ → activation func^n

$$h_t = \phi(W_a [h_{t-1}, x_t] + b_a)$$

b_a ↓ bias

Applications of RNN – Language Generation and Sequence Models

Let $y_1, y_2, \cdots, y_n$ be the words of a language
($n \rightarrow 10^6$ or even 10^{80} for rich language like English)

Goal of a language model

- ~~key to find~~ derive the probability of a given sequence of words $P(y_1, y_2, \cdots, y_k)$
- helpful in tasks like speech recognition

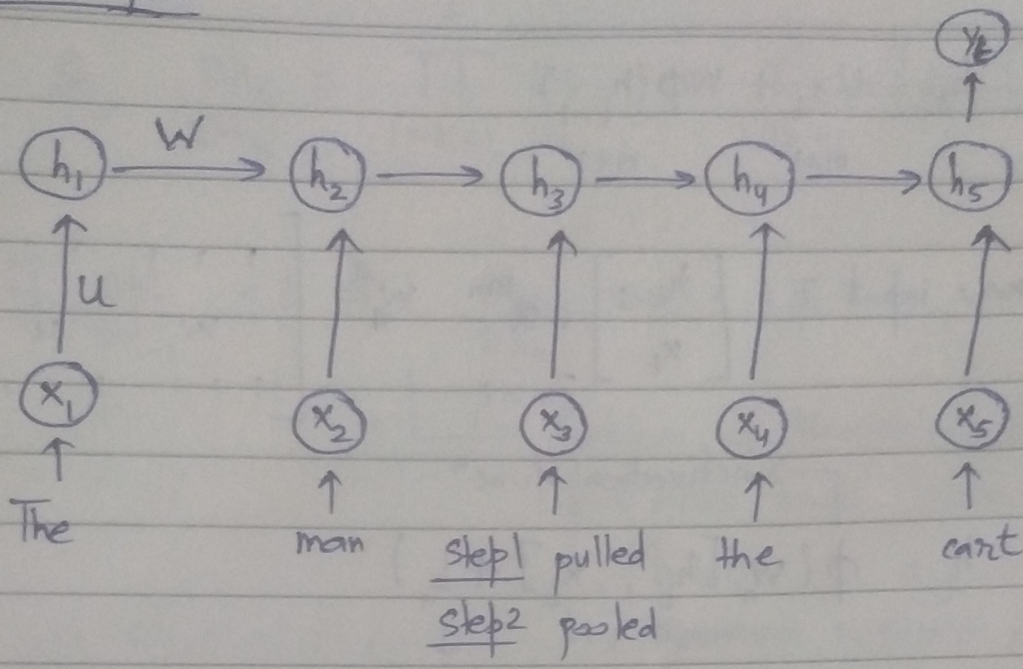
Ex. "The man _____ a cart." (Pooled / pulled)

Find P(The man pooled a cart) & P(The man pulled a cart)

$\Downarrow$ $P(E_1)$ $\qquad\qquad$ $\Downarrow$ $P(E_2)$

The model must establish $P(E_1) < P(E_2)$.

DL - 31/03



Step1
Step2 pooled

y_t → prob. score, whether input sentence is correct
Input with higher prob. score is used to fill up the blank.

Training
- use a set of sentences
- known sentences can be altered by changing/removing words so that incorrect sentences are also in the set so as to make the model learn

Model

$$h_t = \sigma_h(Ux_t + Wh_{t-1} + b_h)$$
$$\hat{y}_t = \sigma_y(Vh_t + b_y)$$

b_h, b_y → bias
σ_h, σ_y → activation function

Improvement
- use extra hidden layers

$$\mathcal{L} = y_t \log \hat{y}_t + (1-y_t)\log(1-\hat{y}_t)$$

Problem extended Now the position of missing word is also unknown.

Ex. The man the cart.

↳ To identify the appropriate position & appropriate word.

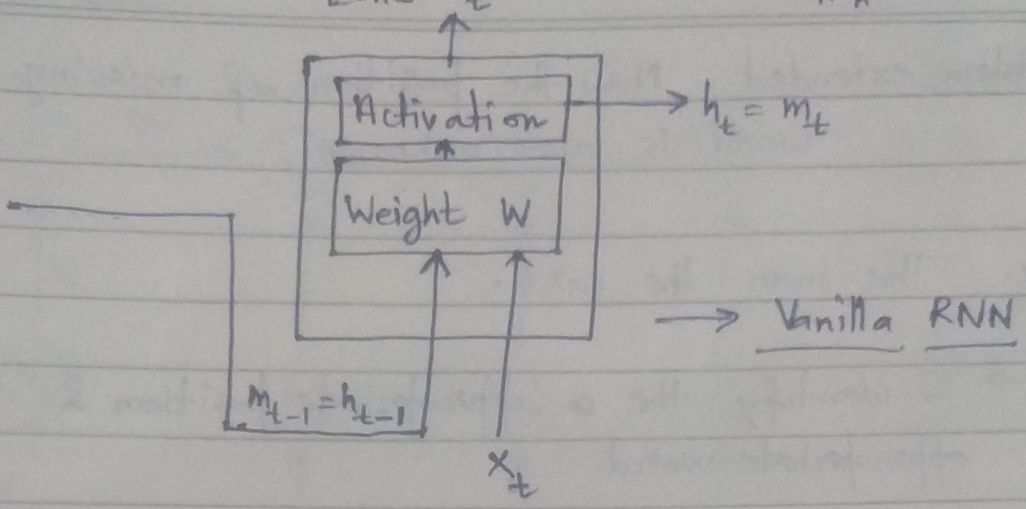
Approach → Pick random words from corpus, put at each position in the sentence & check probability scores. 1 word, 5 positions in ex.

Problem
- → Sentences can be arbitrarily long, so the problem of vanishing gradients may come up
- → carrying info over large portions of text is difficult as knowledge will fade away due to vanishing gradients (Memory problem)

Soln → Gated Recurrent Unit (GRU)
- → maintains a memory cell $m_t = h_t$ that ~~maintains~~ contains relevant info like whether noun encountered was singular/plural & so on
- → at every step, GRU needs to decide whether ~~copy~~ m_t needs to be replaced by an alternate/candidate value $\tilde{m}_t$

* $\odot$ → Hadamard product

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad a \odot b = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_n b_n \end{bmatrix}$$

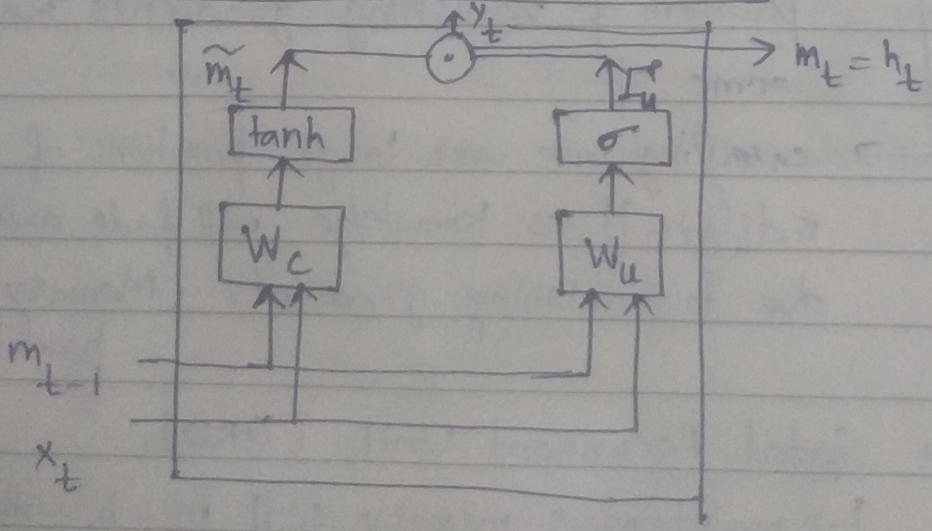


→ Vanilla RNN

$$\tilde{m}_t = \tanh(W_c[m_{t-1}, x_t] + b_m)$$

To make the decision, an update gate $\Gamma_u = \sigma(W_u[m_{t-1}, x_t] + b_u)$ is used.

$$m_t = \Gamma_u \odot \tilde{m}_t + (1 - \Gamma_u) \odot m_{t-1}$$ *

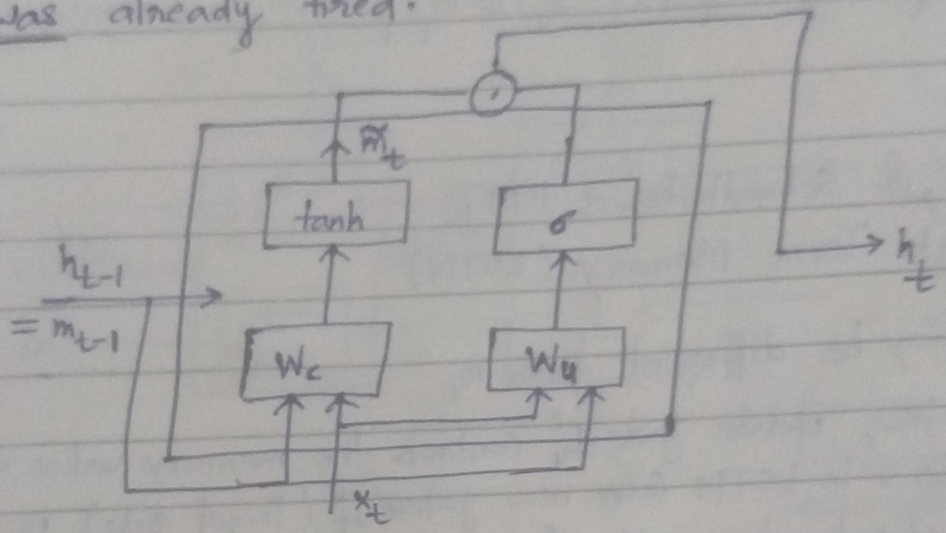


Simplified GRU

DL - 06/04

In arbitrary-long sentence, GRU or memory retainment over a long sequence is necessary for noun-verb agreement

Ex:
The man who pulled the cart when the king arrived was already tired.



$$\tilde{m}_t = \tanh(W_c [m_{t-1}, x_t] + b_m)$$

$$\Gamma_u = \sigma(W_u [m_{t-1}, x_t] + b_u)$$

$$m_t = \Gamma_u \odot \tilde{m}_t + (1 - \Gamma_u) \odot m_{t-1}$$

→ even if $\Gamma_u \to 0$ due to vanishing gradients, m_t won't be 0 due to this. So memory retainment is improved.

A full GRU tries to further improve over simplified GRU. The idea is to consider only data relevant to assigned task by using a relevance gate.

* GRU architecture ref.,

Cho et al. 2014, on the properties of Neural Machine Translation, encoder & decoder approaches

Full GRU*

$$\tilde{m}_t = \tanh(W_c[\Gamma_r \odot m_{t-1}, x_t] + b_m)$$
$$\Gamma_u = \sigma(W_u[m_{t-1}, x_t] + b_u) \quad \text{(Update Gate)}$$
$$\Gamma_r = \sigma(W_r[m_{t-1}, x_t] + b_r) \quad \text{(Relevance Gate)}$$
$$m_t = \Gamma_u \odot \tilde{m}_t + (1 - \Gamma_u) \odot m_{t-1}$$

Hochreiter & Schmidhuber – 1997

Long Short Term Memory (LSTM)

- h_t may be different from m_t
- In GRU, update gate Γ_u controls the memory value m_t, whereas in LSTM new gates Γ_f (forget gate) is introduced along with Γ_o (output gate)
- Γ_f controls m_t & Γ_o controls h_t

$$\Gamma_f = \sigma(W_f[h_{t-1}, x_t] + b_f]$$
$$m_t = \Gamma_u \odot \tilde{m}_t + \Gamma_f \odot m_{t-1}$$
$$h_t = \Gamma_o \odot m_t$$
$$\Gamma_o = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$\tilde{m}_t = \tanh(W_m[h_{t-1}, x_t] + b_m) \quad (W_m = W_c \text{ in GRU})$$
$$\Gamma_u = \sigma(W_u[h_{t-1}, x_t] + b_u)$$

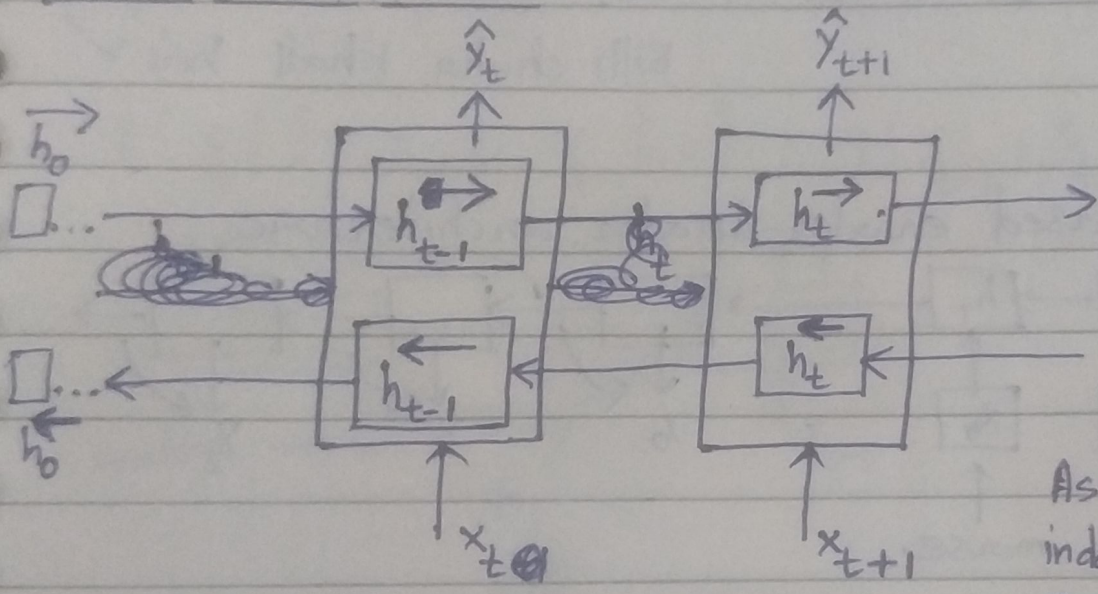
{ He said, "Teddy bears are on sale."
  He said, "Teddy Roosevelt was a great president" }

→ significance of looking forward in a sentence

Understanding forward parts is also important as illustrated.

↳ Bidirectional networks!

Bi-directional RNN



Forward & backward can be done independently or in parallel.

As they may be independent, GRU can also be used here.

$$h_t = [\overrightarrow{h_t} \quad \overleftarrow{h_t}]$$

$$h_t = \sigma(W_h [h_{t-1}, x_t] + b_{h_t})$$

$E[(y - \hat{y})^2]$ is to be minimized.

* Ref. Neural Machine Translation by jointly learning to align & translate
— Bahdanau et al., ICLR 2015

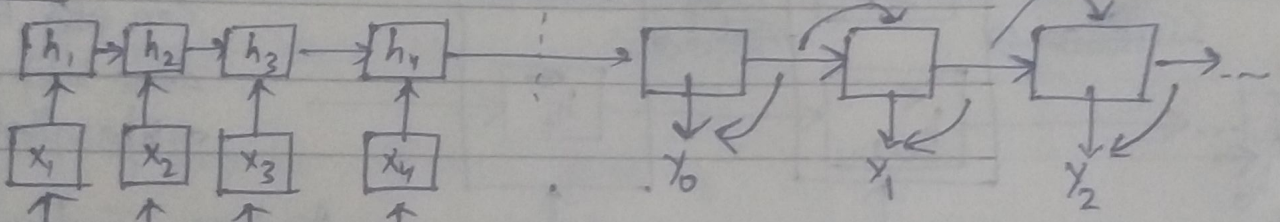
DL - 10/04

Attention & Transformers *

Word to word translation ~~is~~ maybe meaningless.

Ex. The cat eats mouse. ⟶ Wo billi khati chuha ✗
Billi chuha khati hai ✓

Soln - RNN-based encoder-decoder architecture



Problem — Sentences can be arbitrarily long. Capturing the context of the whole sentence in a lower dimension embedding may produce inefficient results.

Soln — Create a context vector. Feed it to the decoder so as the decoder can choose the best possible output given the input sentence & the context.

RNN-based encoder

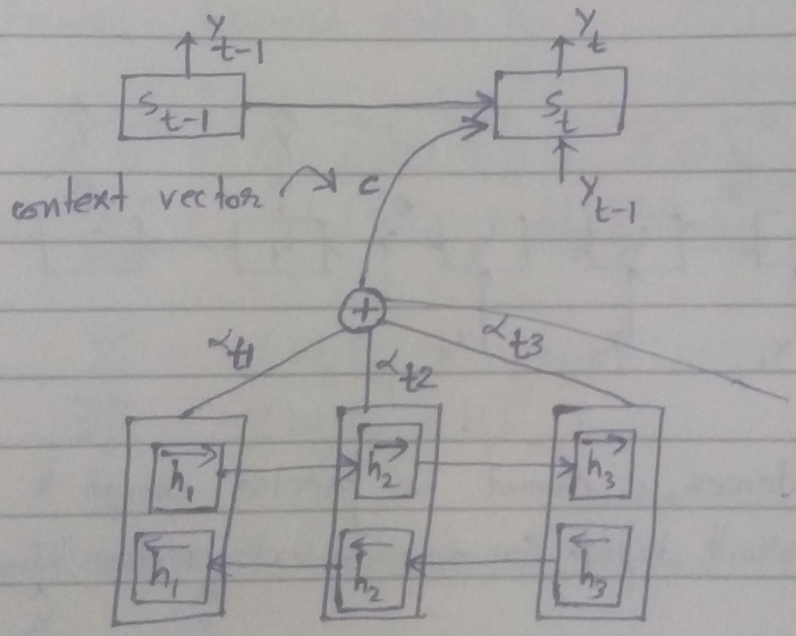
$x = (x_1, x_2, \ldots, x_{T_x})$ Encoder

$h_t = f(h_{t-1}, x_t)$

$c = g(h_1, h_2, \ldots, h_{T_x})$ Context vector

$p(y) = \prod_{t=1}^{T} p(y_t \mid \{y_1, y_2, \ldots, y_{t-1}\}, c)$ Decoder

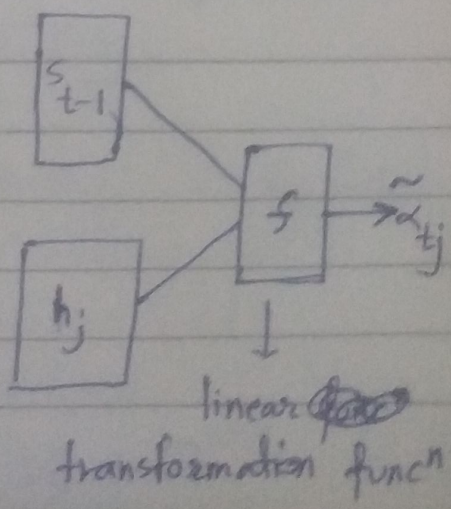
$p(y_t \mid \{y_1, y_2, \ldots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c)$



$$\alpha_{tj} = \frac{\exp(\tilde{\alpha}_{tj})}{\sum_{j'} \exp(\tilde{\alpha}_{tj'})}$$

$$\tilde{\alpha}_{tj} = f(s^{(t-1)}, h^{(j)})$$

$$c^{(t)} = \sum_{j=1}^{T_x} \alpha_{tj} h_j$$



Learnable parameters are weights,

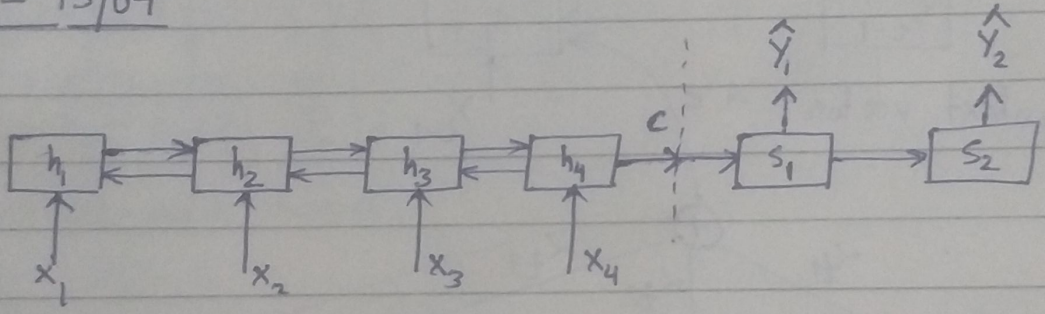
(1) Weights used in f on linearly transforming $s^{(t-1)}$ & h_j

(2) Weights used to generate $\overrightarrow{h_t}$ & $\overleftarrow{h_t}$ from given input

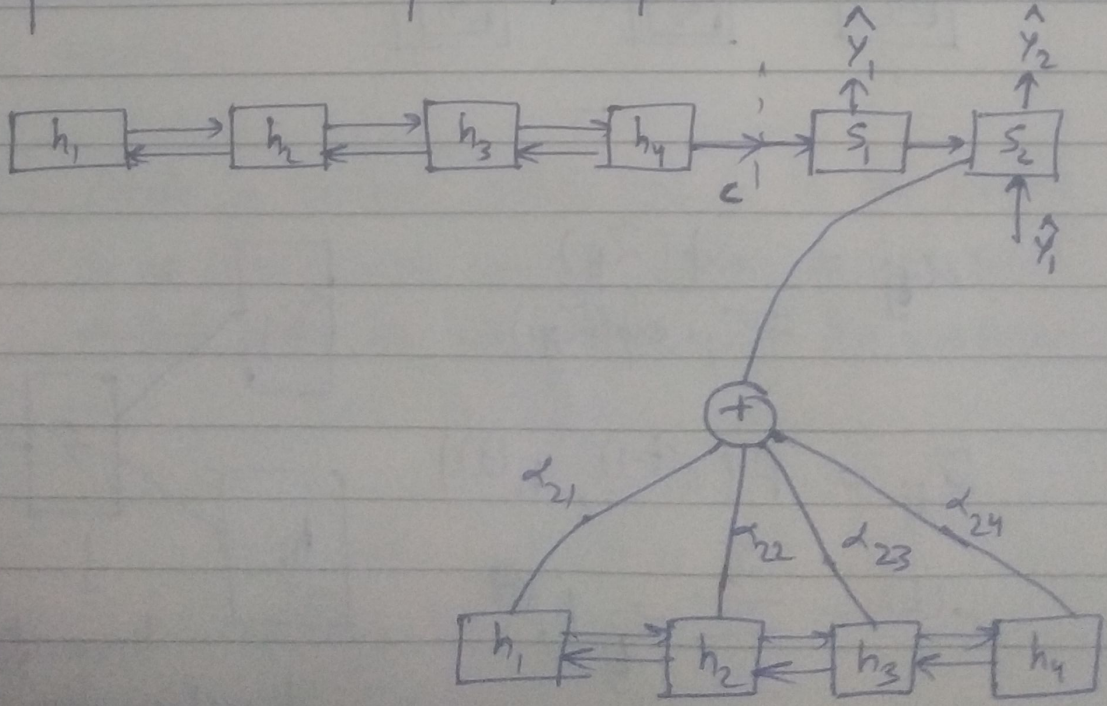
$y_1\ y_2\ y_3 \dots y_n \qquad P(\hat{y}_1)\, P(\hat{y}_2) \dots P(\hat{y}_n)$

$\alpha = \prod_{y_i \in T_y} P(\hat{y}_i) \rightarrow$ optimization by maximising the likelihood

DL - 13/04



For long sentences, c cannot be precise enough to represent overall input. So, ~~that~~ context vector introduced



* z_i for encoder & decoder aren't the same

** $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$ | $e_{ij} = v_a^T \tanh(w_a s_{i-1} + U_a h_j)$

Encoder :- ~~Bidirectional~~ Bidirectional GRU

Source sentence → word tokens → 1 of k vectors

Construct a vocabulary of K_x words.

1 of K → one-hot encoding over K_x words

Input in $X = \{x_1, x_2, \cdots, x_{T_x}\}$ where $x_i \in \mathbb{R}^{K_x}$

(1 of k vector)

Translation output $Y = \{y_1, y_2, \cdots, y_{T_y}\}$ where $y_i \in \mathbb{R}^{K_y}$

Encoder: forward states $\overrightarrow{h_i}$

$$\overrightarrow{h_i} = (1 - \overrightarrow{z_i}) \odot \overrightarrow{h_{i-1}} + \overrightarrow{z_i} \odot \overrightarrow{h_i}$$

$$\underline{\overrightarrow{h_i}} = \tanh(W E_{x_i} + U[\overrightarrow{r_i} \odot \overrightarrow{h_{i-1}}]) \quad (E_{x_i}: \text{embedding of } x_i)$$

$$\overrightarrow{z_i} = \sigma(W_z E_{x_i} + U_z \overrightarrow{h_{i-1}}) \quad \text{(Update Gate)}$$

$$\overrightarrow{r_i} = \sigma(W_r E_{x_i} + U_r \overrightarrow{h_{i-1}}) \quad \text{(Relevance gate) or Reset}$$

Same eqn for backward states.

~~Dec~~ Decoder :- GRU

W, U, C are weights

$$s_i = (1 - z_i^d) \odot s_{i-1} + z_i^d \odot \underline{s_{i-1}} \quad *$$

$$\underline{s_i} = \tanh(W E_{y_{i-1}} + U[r_i \odot s_{i-1}] + C c_i) \quad **$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

~~Eqn~~ $z_i^d = \sigma(W_z E_{y_{i-1}} + U_z s_{i-1} + C_z c_i)$

$r_i^d = \sigma(W_r E_{y_{i-1}} + U_r s_{i-1} + C_r c_i)$

$s_0 = \tanh(W_s \overleftarrow{h_1})$ $\quad W_s \in \mathbb{R}^{n \times n}$

Probability of output,

$$P(y_i \mid s_i, y_{i-1}, c_i) \propto \exp(y_i^T W_o t_i)$$

$$t_i \sim U_o s_{i-1} + V_o E y_{i-1} + C_o c_i$$

→ maximize likelihood to minimize loss func^n & learn the weights

Performance Measures in Machine Translation

↳ Bleu score

Consider 2 sentences,

(1) The ball is blue (Predicted)
(2) The ball has a blue color (Translated)

Attempt – consider the overlap of words appearing in predicted sentence & translated sentence

Failure – "The ball the ball" overlaps 100 % with (2) but is very wrong

Sol^n: Tokenise into N-grams, find precision

* If all weights are $w_n = 1/N$, then

$$P_{gavg} = \exp\left(\sum_{n=1}^{N} \log P_n^{1/N}\right) = \exp\left(\log(P_1 P_2 \cdots P_n)^{1/N}\right)$$

$$\Rightarrow \boxed{P_{gavg} = (P_1 P_2 \cdots P_n)^{1/N}}$$

For (1)

1-gram : "The", "ball", "is", "blue"

2-gram : "The ball", "ball is", "is blue"

3-gram : "The ball is", "ball is blue"

$$\text{Precision} = \frac{\#\text{ Correct predicted words}}{\text{total predicted words}}$$

In (1), & (2), precision = 3/4

But there can be repetitions, "the the the ball" or multiple target sentences.

So, use clipped precision, if a word appears >1 times in predicted sentence, or if it is there in multiple sentences (target), count its value as 1 only.

To find bleu score,

find precision for 1-gram - P_1

" " " 2-gram - P_2

" " " 3-gram - P_3

- - - - - - -

" " " n-gram - P_n

(GAP)

Find geometric avg precision score, P_{gavg},

$$P_{gavg} = \exp\left(\sum_{n=1}^{N} W_n \log P_n\right)$$

* W_n = weight for n-gram precision

r = reference text length

find brevity penalty, (BP)

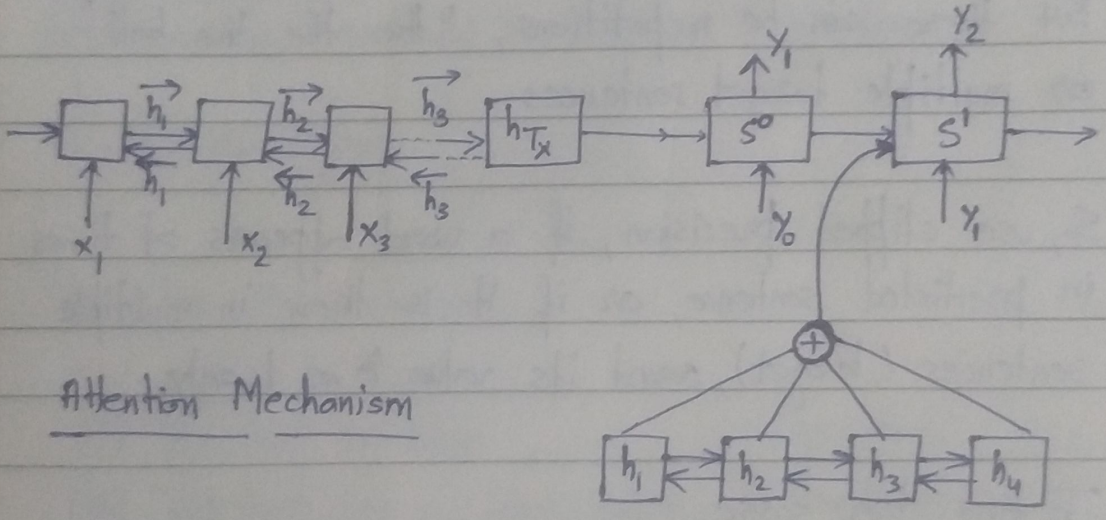
(penalises short translations)

$$BP = \begin{cases} 1 & ; \text{ if length} > r^* \\ e^{(1 - r/len)} & ; \text{ else} \end{cases}$$

len → predicted length

Bleu Score = GAP · BP

DL - 15/04



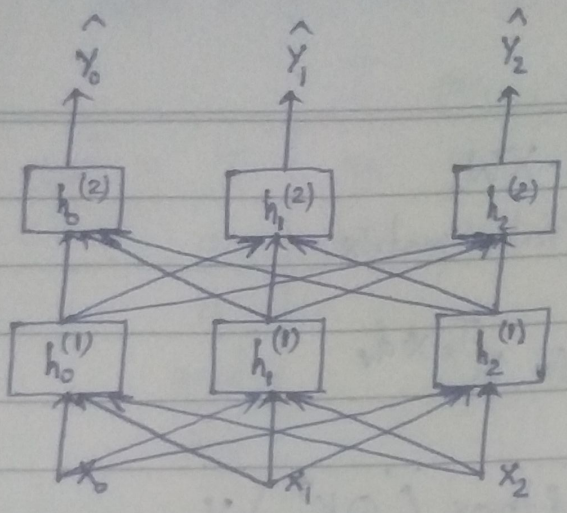
Attention Mechanism

For arbitrarily long sentences, there are recurrence units in the model which tend to slow down the training significantly.

Improvement — Attention is all you need
— Vaswani et al., NIPS 2017

Sequential RNN-based architecture can't be parallelized to optimize computation complexity. Transformations introduced.

Intuition for Transformer architecture



Attention of any unit is upon all units in previous layer

Advantage Sequential dependency & recurrences have been eliminated. Parallelization can be done to optimize computational complexity.

Ex. Given x_0, x_1 & x_2, $h_0^{(1)}, h_1^{(1)}$ & $h_2^{(1)}$ can be evaluated ~~fast~~ in parallel, & so on for other layers.

Consider a word sequence, "I am a good boy."

$$X = \begin{matrix} \text{I} \\ \text{am} \\ \text{a} \\ \text{good} \\ \text{boy} \end{matrix} \begin{bmatrix} .75 & .66 & .21 & \cdots \\ .5 & .2 & .34 & \cdots \\ .6 & .4 & .32 & \cdots \\ .1 & .25 & .35 & \cdots \\ .5 & .26 & .5 & \cdots \end{bmatrix} \in \mathbb{R}^{T_x \times d_k}, \; T_x = 5$$

transformation follows some rule

$q_i = W_q x_i \; ; \; W_q \in \mathbb{R}^{d_k \times d_k}$

$Q = XW_q \in \mathbb{R}^{T_x \times d_k}$

↳ Query matrix formed by a transformation of X by W_q, a set of learnable parameters, similar for K & V

$K = XW_k$ & $V = XW_v$

↳ Key matrix ↳ Value matrix

$K \in \mathbb{R}^{T_x \times d_k}$ $V \in \mathbb{R}^{T_x \times d_v}$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$Q = XW_q =$

I	.65	.62 ...
am	.31	.67 ...
a	.45	.47 ...
good	.32	.62 ...
boy	.47	.68 ...

= Some linear transformation of X

$QK^T =$

	I	am	a	good	boy

Q

- when evaluating QK^T, ↑ row of "I" is multiplied by K^T column of "I", so result is some kind of a pairwise attention of "I" over "I", same for other words.

Also since words are aligned vertically in Q & horizontally in K^T, a pairwise dependency of each word upon another is also evaluated.

* d_k may also be same as d_v. The difference just helps in a smooth generalisation.

QK^T =

	I	am	a	good	boy
I	.75	.45	.36	...	
am	.86	.86	.75	...	
a					
good					
boy					

$\nexists$ as $Q \in \mathbb{R}^{T_x \times d_k}$ & $K^T \in \mathbb{R}^{d_k \times T_x}$

$QK^T \in \mathbb{R}^{T_x \times T_x}$

as d_k is dimension of Q & K^T, it ~~doe~~ does a good job at scaling the values in QK^T. Also known as scaled normalization.

The softmax over $\frac{QK^T}{\sqrt{d_k}}$ now changes the matrix into a probability distribution. In other words, in the row of "I", the column of word with highest value (probability) is the most relevant in deciding "I" & so on. Also known as scaled dot product normalization.

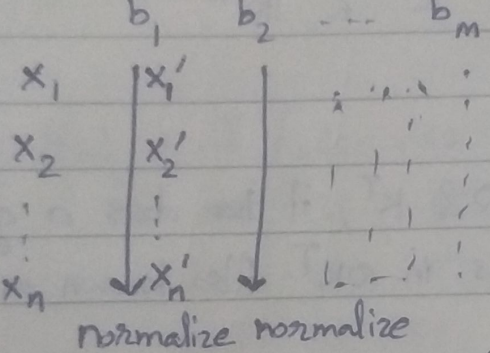
Now when multiplied by V, it finally gives how much attention is to be paid to a word in deciding another word.

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V = \begin{matrix} I \\ am \\ a \\ good \\ boy \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ - & - & - & - \\ \cdots & & & \end{bmatrix} \begin{bmatrix} .5 & .7 & \cdots \\ .8 & .6 & \cdots \\ - & - & - & - \\ \cdots & & & \end{bmatrix}$$

(W_q, W_k, W_v)

↳ In multi-head attention, use different values for these 3 learnable parameters

In multi-head attention, different matrices are obtained which are then normalized after adding over entire input set.



$$x_i' = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

↳ Layer-wise batch normalization

As these are separate batches, the whole values are evenly distributed.

DL - 20/04

Ex: I am a good boy.

$$X = \begin{matrix} I \\ am \\ a \\ good \\ boy \end{matrix} \begin{bmatrix} .7 & .5 & \cdots \\ .2 & .8 & \cdots \\ & & \\ & & \\ & & \end{bmatrix}$$

$q_i = W_q x_i$, $Q = XW_q$

$k_i = W_k x_i$, $K = XW_k$

$v_i = W_v x_i$, $V = XW_v$

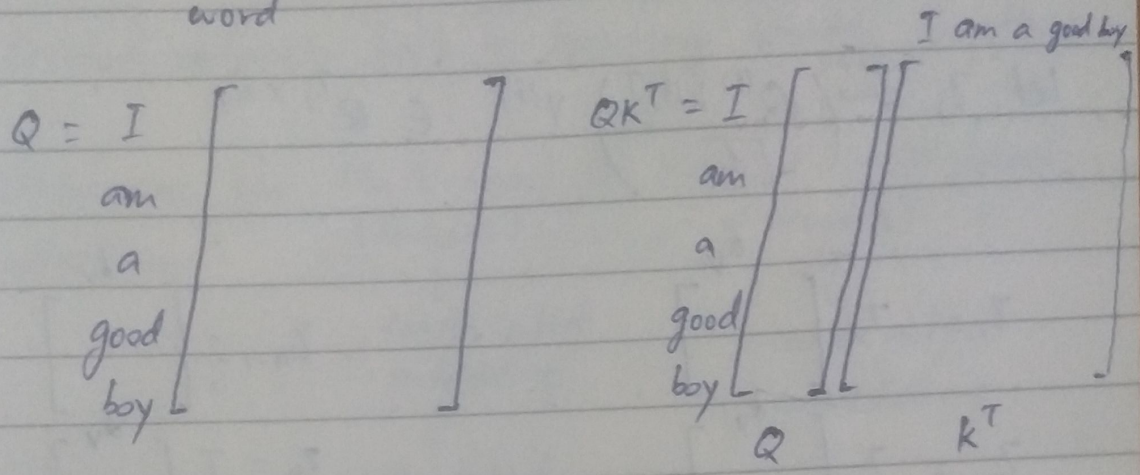
$$\text{Self-attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

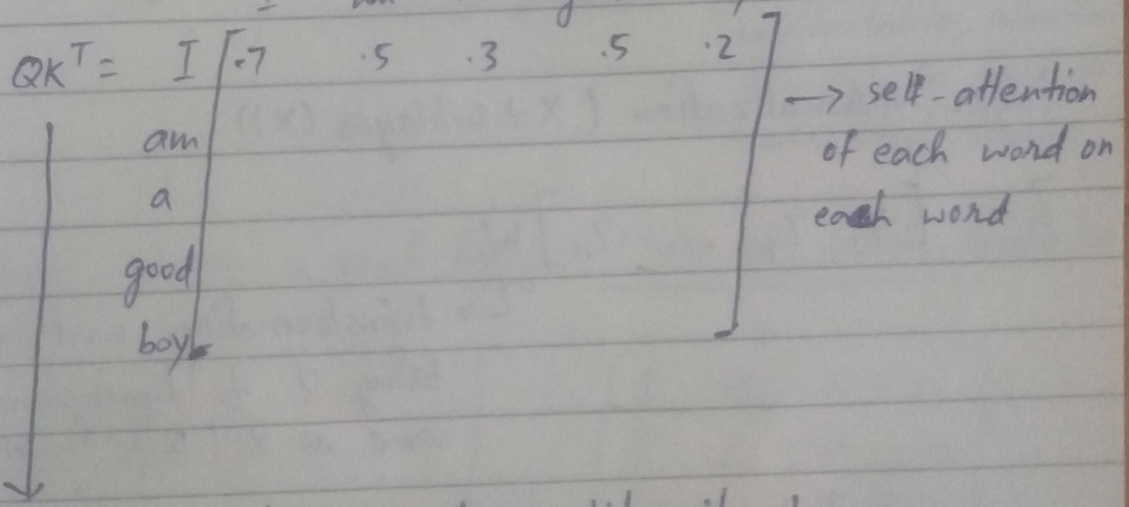
$X \in \mathbb{R}^{T_x \times d_{model}}$ $\quad W_q \in \mathbb{R}^{d_{model} \times d_k}$

$XW_q = Q \in \mathbb{R}^{T_x \times d_k}$

$QK^T \in \mathbb{R}^{T_x \times T_x}$

↳ kind-of self-attention for each word on each word





Problem :- values can be arbitrarily large

↳ loss func^n doesn't change much wrt weights

↳ gradient doesn't change much

↳ inefficient learning

Solution - scale ($\sqrt{d_k}$) & apply softmax

↳ gets converted to probability scores

↳ "how important words are wrt some word"

* $d_v = d_k$ in paper. For general case, they may be taken to be different.

Multi-head attention, there are different sets of weights of W_q, W_k & W_v which ~~are~~ can be stacked or concatenated.

Let $Z_i = \text{softmax}\left(\frac{Q^{(i)} K^{(i)T}}{\sqrt{d_k}}\right) V^{(i)} \in \mathbb{R}^{T_x \times d_v}$

$$Z_1 = T_x \begin{bmatrix} & d_v & \\ & & \end{bmatrix} \quad \cdots \quad Z_{n-1} = T_x \begin{bmatrix} & d_v & \\ & & \end{bmatrix}$$

$$Z_2 = T_x \begin{bmatrix} & d_v & \\ & & \end{bmatrix} \quad \cdots \quad Z_n = T_x \begin{bmatrix} & d_v & \\ & & \end{bmatrix}$$

Layer Normalization (X + sublayer (X))

$$Z = [Z_1 \; Z_2 \; \cdots \; Z_n] W_o$$

↳ transformation matrix to bring Z to dimensions same as X ($\mathbb{R}^{T_x \times d_{model}}$)

I_1 I_2 ... I_n

x_1
x_2
x_3
⋮
x_m

Column-wise Normalization

↓

Difference of mean across batches is normalized

↓

Layer wise Normalization (LN)

Row-wise Normalization

↓

Batch Normalization (BN)

LN $\mu = \frac{\sum x_i}{f}$ $x'_j = \frac{x_i - \mu}{SE[x_i]}$

Residual networks help in learning identity functions.

Difference b/w Multi-head & Masked multi-head attention,

(Encoder) → takes input embedding → whole input sentence available

(Decoder) → takes output embedding → whole input unavailable, attention can only be paid to generated words

Stage I am a good boy → Main ek achcha bachcha hun

$$\begin{matrix} \text{Main} \\ \text{ek} \\ \text{achcha} \end{matrix} \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix} + \begin{bmatrix} 1 & -\infty & -\infty & -\infty \\ 1 & 1 & -\infty & -\infty \\ 1 & 1 & 1 & -\infty \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

Mask

Idea is to deplete attention of non-output words to $-\infty$ so that softmax is 0, this is masked multi-head attention. Attention / Importance of a word wrt look-ahead word gets to 0.

Positional Encoding.

- Position of a word in a sentence is important, or I am a good boy ≡ Am a I boy good
- Positional info ≡ feeder of words to the model

$PE_{(pos, 2i)} \in \mathbb{R}^{d_{model}}$

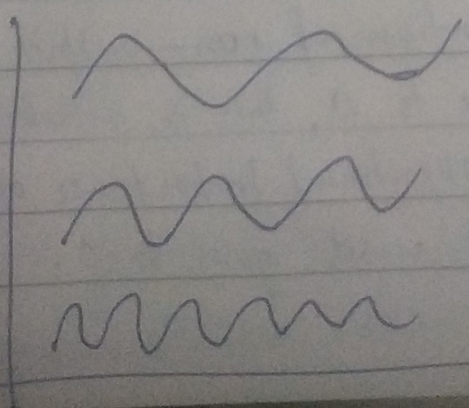
pos: position; $2i$: for even. For odd is $(2i+1)$

For I in "I am a good boy", pos = 1

$$PE_{(pos, 2i+1)} = \cos\left(pos / 10000^{2i/d_{model}}\right)$$
$$PE_{(pos, 2i)} = \sin\left(pos / 10000^{2i/d_{model}}\right)$$

$(2i, 2i+1) \longrightarrow (1, d_{model})$

$$PE_{(1)} \text{ for I} = \begin{bmatrix} PE_{(1,1)} \\ PE_{(1,2)} \\ \vdots \\ PE_{(1, d_{model})} \end{bmatrix}$$



→ sin, cos wavy curves encode positional info into the model.

* Ref. Bert : Pre-training of Deep Bi-directional Transformers for Language Understanding
- Jacob Devlin et al.

DL - 21/04

BERT : Bi-directional Encoder Representations using Transformers*

1) Masked Word Prediction
2) Next line Prediction

Autoencoders — slides

PCA aims to vanish variance along a certain no of dimensions so that the dependency on ~~that~~ those dimensions of the output ~~one~~ is eradicated. Thus, PCA reduces the dimensionality.