Deep Learning

Activation function

① Linear function

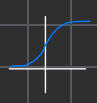
↳ two classes seperated by a line

↙ fails over complex data

② Non-linear function (5)

(i) Sigmoid

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

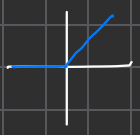


(ii) tanh

$\tanh(x)$

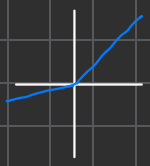
(iii) Relu

$\max(0, x)$



(iv) Leaky Relu

$\max(0.1x, x)$



↳ when data gets more complicated we start adding more layers.

↙ but with increasing number of layers the layers stop learning, and in classification, the boundary no longer changes.

↘ Vanishing Gradient problem

↳ during back propagation the gradients gets smaller and smaller and vanishes.

and as $w' = w - \lambda \frac{\partial}{\partial x}$ → turns out to be 0

Hence, No Learning

Remedy ⇝ Use activation function that does not squeezes information

↙ ReLU

↘ Again after a while the boundary does not changes again.

↙ Dying ReLU Problem

blocks information < 0.

↙ Leaky ReLU / ELU

Activation function of Outputs

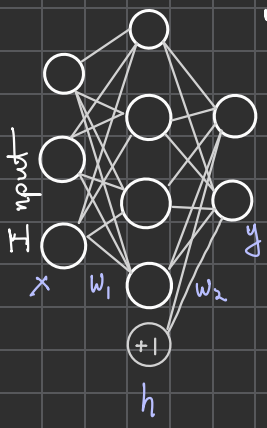
for classification → Softmax function → # neurons = # classes

$$\text{Softmax}(x) = \frac{e^{z}}{\sum_{2} e^{z}}$$ → Output probabilities of classes

for regression
↳ Don't need an Activation function. → based on the type of output we want, we can choose the activation function

---

In depth → Why linear function produces a straight line?



$y = w_2 h + b_2$

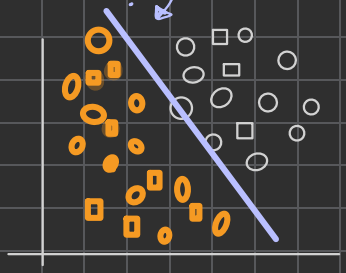
$h = w_1 x + b_1$

$y = w_2 (w_1 x + b_1) + b_2$

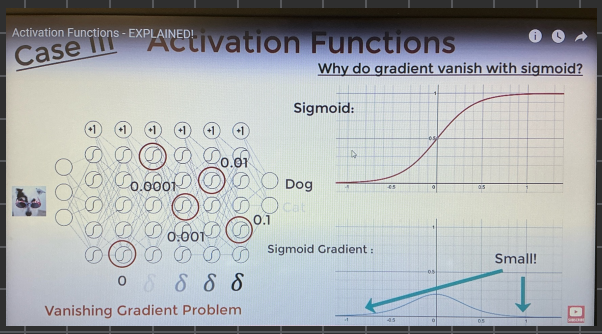
$y = w_1 w_2 x + w_2 b_1 + b_2$

U x B

$y = Ux + B$ → straight line



② Why Sigmoid causes vanishing gradient?



with larger true/-ve values, Sigmoid gradient tends to 0. causing it to minimize.

and with backpropogation, with multiplication it goes further lower → making it 0.

due to squeezing nature of Sigmoid function. [takes a number and squeezes it to a range]

One more problem that comes in with increasing # layers. → Exploding gradient

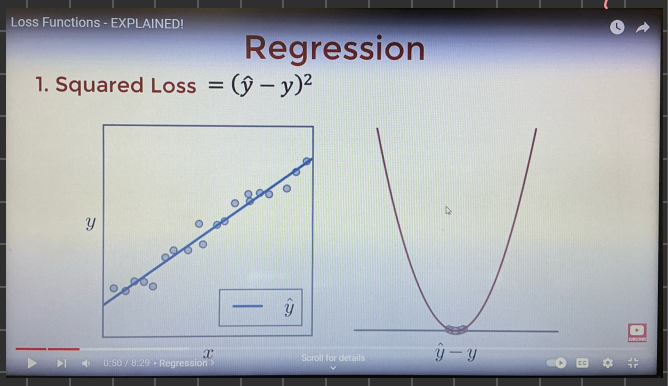
when $\frac{\partial L}{\partial x}$ → goes very large, we can not converge and hence we don't reach to minimal loss value → underfitting

Loss functions

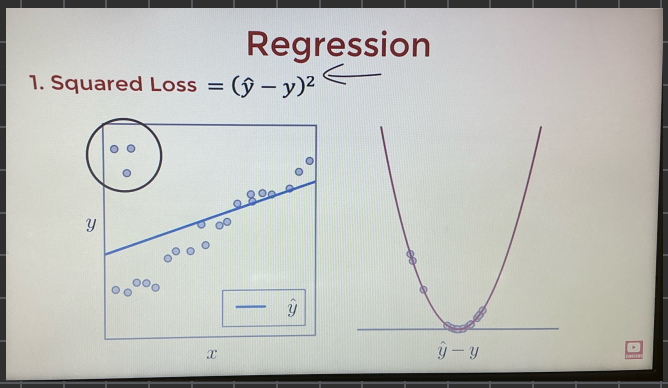
① Regression

the difference is squared or absolute so, that +ve / -ve differences do not cancel out.

but with outlier

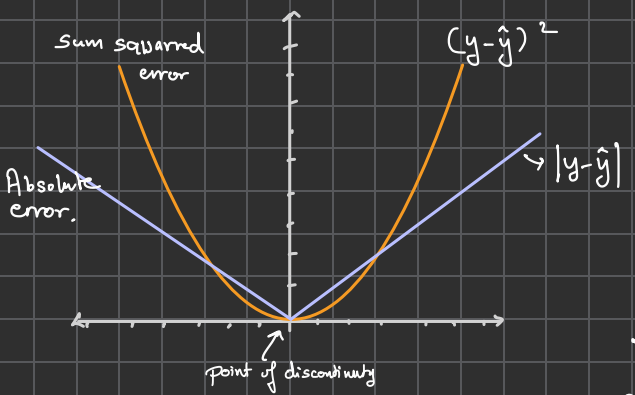


this performs bad, as the model wants to fit in the outliers



thus Absolute error $|\hat{y} - y|$ → this might lead to poor predictions

often used in Support vector Regression



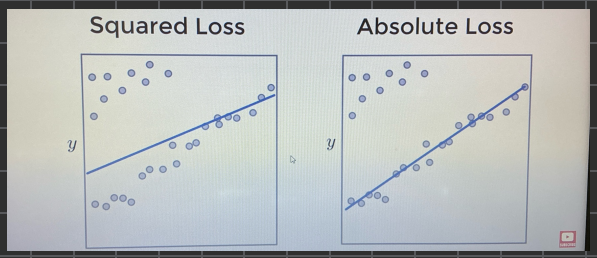
mean absolute error is not optimized through gradient descent

↳ Sub gradients

- one of them considers the outliers
- other one completely ignores them.

↓

both resulting in poor predictions



Classification

↳ classes (probability distribution)

A	0.08
B	0.74 → max → Hence result
C	0.07
D	0.11

we compare it with the ground truth, and how we compare → depends on the loss we use

① Cross-entropy loss

we must first understand KL divergence

↓

distance between two probability distributions

$$D_{KL}(P||Q) = \sum P(i) \log \frac{P(i)}{Q(i)}$$

or

$$D_{KL}(P||Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx$$

In depth

Coin 1

0.5	Heads
0.5	Tails

Coin 2

0.95	Heads
0.05	Tails

These two coins are very different, but How different are they? quantitatively

$$\frac{P(\text{observation} \mid \text{real coin})}{P(\text{observation} \mid \text{coin 2})}$$

True Coin

P_1	Heads
P_2	Tails

Coin 2

q_1	Heads
q_2	Tails

trials → H H T H H T H H H T H T

$P_1 \cdot P_1 \cdot P_2 \cdot P_1 \cdot P_1 \cdot P_2 \cdot P_1 \cdot P_1 \cdot P_1 \cdot P_2 \cdot P_1 \cdot P_2 \longrightarrow P_1^{\#H} P_2^{\#T}$

$q_1 \cdot q_1 \cdot q_2 \cdot q_1 \cdot q_1 \cdot q_2 \cdot q_1 \cdot q_1 \cdot q_1 \cdot q_2 \cdot q_1 \cdot q_2 \longrightarrow q_1^{\#H} q_2^{\#T}$

$$\frac{P(\text{observation} \mid \text{real coin})}{P(\text{observation} \mid \text{coin 2})} = \frac{p_1^{\#H}\, p_2^{\#T}}{q_1^{\#H}\, q_2^{\#T}}$$

↓

Normalizing it based on sample size

$$= \left[\frac{p_1^{H}\, p_2^{T}}{q_1^{H}\, q_2^{T}}\right]^{1/N}$$

↙

$$\log\left[\frac{p_1^{H}\, p_2^{T}}{q_1^{H}\, q_2^{T}}\right]^{1/N} \rightarrow \frac{1}{N}\log\left[\frac{p_1^{H}\, p_2^{T}}{q_1^{H}\, q_2^{T}}\right]$$

$$= \frac{1}{N}\log p_1^{H} + \frac{1}{N}\log p_2^{T} - \frac{1}{N}\log q_1^{H} - \frac{1}{N}\log q_2^{T}$$

$$= \frac{H}{N}\log p_1 + \frac{T}{N}\log p_2 - \frac{H}{N}\log q_1 - \frac{T}{N}\log q_2$$

↓

considering sample size tending $\rightarrow \infty$

we assume $\rightarrow p_1$

$$= p_1 \log p_1 + p_2 \log p_2 - p_1 \log q_1 - p_2 \log q_2$$

$$= p_1 \log \frac{p_1}{q_1} + p_2 \log \frac{p_2}{q_2}$$

$$\boxed{= \sum_x p \log \frac{p}{q}}$$

input → x | predicted class distribution → $P(y|x_i;\theta)$ | Actual class distribution = $P^*(y|x_i)$

Are these close?

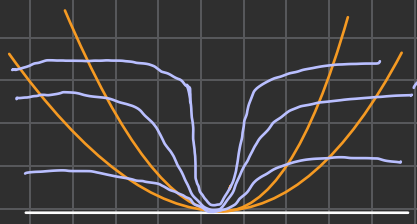
↙

KL divergence

$$D_{KL}\left(P^*(y|x_i) \,||\, P(y|x_i;\theta)\right) = \sum P(i) \log \frac{P(i)}{Q(i)}$$

② Hinge Loss ← used in support Vector machine

③ for Regression → Adaptive loss



Weight initialization

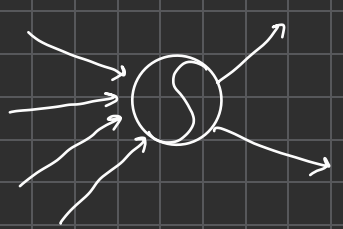
↳ to keep in mind

① weight should be small

② should have good variance, each neuron must carry an information within itself.

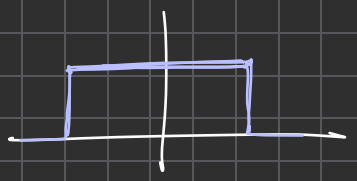
different neurons will learn different aspects of input.

randomly initialization



① Uniform initialization

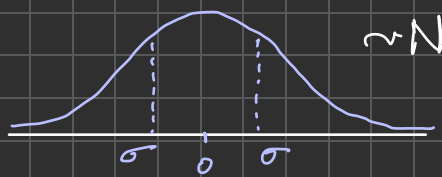
$$\hookrightarrow \left[\frac{-1}{\sqrt{\text{fan in}}}, \frac{1}{\sqrt{\text{fan in}}}\right]$$



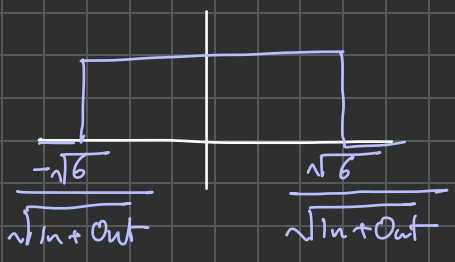
② Xavier / glarot initialization

Normal

$$\sim N\left(0, \frac{2}{\text{fan in} + \text{fan Out}}\right)$$



→ Uniform → $\sim U\left[\frac{-\sqrt{6}}{\sqrt{\text{fan in} + \text{fan Out}}}, +\right]$



③ He-init {often used with ReLU}

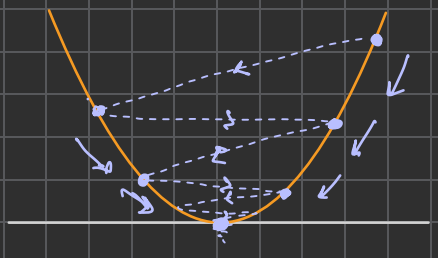
Normal

$$\sim N\left(0, \sqrt{\frac{2}{\text{fan in}}}\right)$$

Uniform

$$\sim U\left[-\sqrt{\frac{6}{\text{fan in}}}, \sqrt{\frac{6}{\text{fan in}}}\right]$$

Optimizers → gradient descent



for every epoch:

$$w' = w - \alpha \frac{\partial L}{\partial x}$$

↓

weight is only updated once after seeing the entire dataset. ↓

and the weight just hovers at its optimal value, rather than actually reaching there.

↓

update the parameters more frequently.

eg: SGD

but SGD is very noisy, as it gets influenced by every single sample

↓

Mini batch GD

Another way to handle this is by introducing **Momentum**

instead of taking just one current gradient, it also takes in account the previous update to smooth out the oscillations

↓

faster convergence + Less Oscillations

$$w_{t+1} = w_t - v_t$$

$$v_t = \beta v_{t-1} + \eta \nabla w_t$$

but:

- Too much momentum can overshoot the minima
- As the optimizer gets closer to minima, the gradient become smaller, ⟶ leading to slower convergence., and it can still overshoot.

↓

SGD + Momentum + Acceleration

Nesterov Accelerated gradient → Instead of updating, solely based on current gradient,

NAG computes the gradient at the predicted future, (where the momentum might take it.) → controlling momentum

$$w_{new} = w_{old} - v_{new}$$

$$v_{new} = \beta v_{old} + \eta \nabla (w_{old} - \beta \cdot v_{old})$$

Momentum

past velocity

$$w_{t+1} = w_t - (\beta v_{t-1} + \eta \nabla w_t)$$

gradient at the moment

earlier,

$$w_{t+1} = w_t - \eta \nabla w_t$$

NAG →

Next step depends upon

History of Velocity + gradient at that position

Seq → (1) → (2) →

for momentum you go sequential

the only difference with NAG is → we apply momentum and then calculate gradient for the next step point.

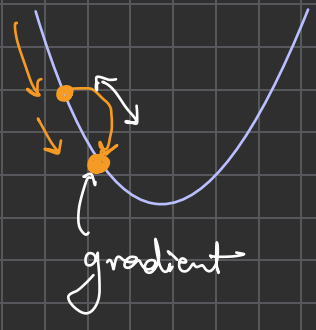
Hence, NAG performs better.

Look ahead → la

$$w_{la} = w_t - \beta v_{t-1}$$

$$v_t = \beta v_{t-1} + \eta \nabla w_{la}$$

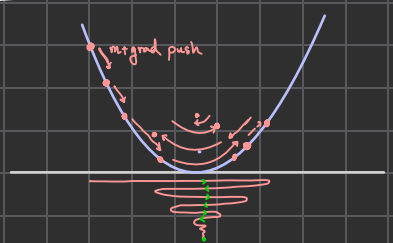
$$w_{t+1} = w_t - v_t$$

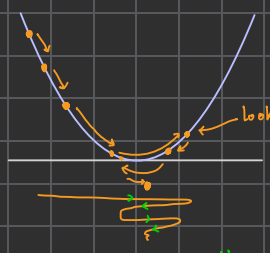


geometric intuition

momentum → Acceleration → NAG

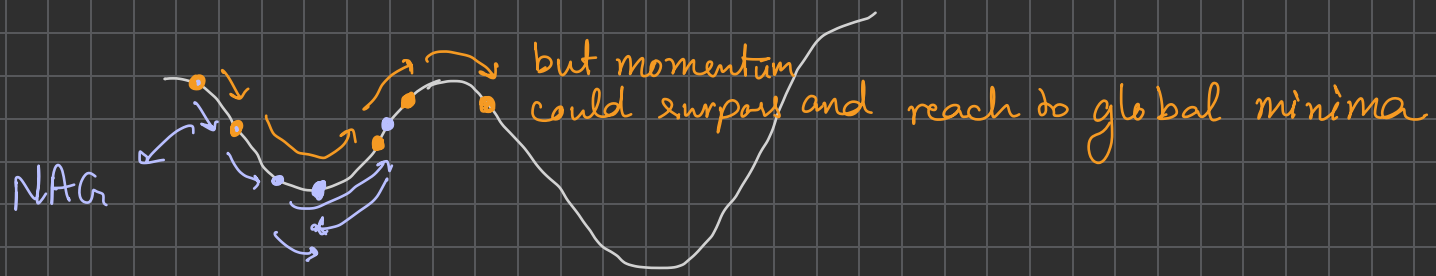
Momentum





Hence, decreasing # oscillations

but the problem with NAG, because it dampens the oscillations might, saddle point at local minima



Adagrad → very similar to SGD, but with adaptive gradient.

SGD → $w_t = w_{t-1} - \eta \, \Delta g_t$

Adagrad → $w_t = w_{t-1} - \eta' \, \Delta g_t$

↳ different learning rate at each weight

$$\eta' = \frac{\eta}{\sqrt{G_t + \epsilon}}$$

↳ sum of squared of all prev. gradients

$$\sum_{i=1}^{t-1} g_i^2$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{\sum g_i^2 + \epsilon}} \cdot \Delta L$$

but if $\sum g_i^2$ → goes very high

↓

Slow convergence

↙

RMS prop

(Root Mean Square Propagation)

RMS Prop → It only considers recent gradients through exponentially decaying average, allowing it to make more reasonable learning rate throughout training.

$$\eta_t = \frac{\eta_{t-1}}{\sqrt{W_{avg} + \epsilon}} \qquad W_{avg} = \alpha\, W_{avg_{(t-1)}} + (1-\alpha)\sum\left(\frac{\partial L}{\partial w}\right)^2$$

Adam Optimizer → It combines two ideas

① keep track of average of gradients → [Direction & Speed] → to capture the direction in which params are moving.

② keep track of squared average → [Step Size] → to adapt the learning rate for each parameters.

① First moving average [Momentum] : okay, I have been moving down hill for a while, let's keep going in this direction

$$w_{new} = 0.9\, w_{old} + 0.1 \text{ current gradient}$$

② Second moving average [learning rate adjustment] : If i am on steep hill → smaller steps, flat → larger steps

$$v_{new} = 0.999\, v_{old} + 0.001 \cdot [\text{Current gradient}]^2$$

③ Adjust the bias → initially because the averages are just starting they might be incorrect (too small). Thus needs to be corrected.

→ ④ $$w_{new} = w_{old} - \frac{\text{momentum}}{\sqrt{\text{adjusted step size} + \epsilon}}$$

Batch Normalization

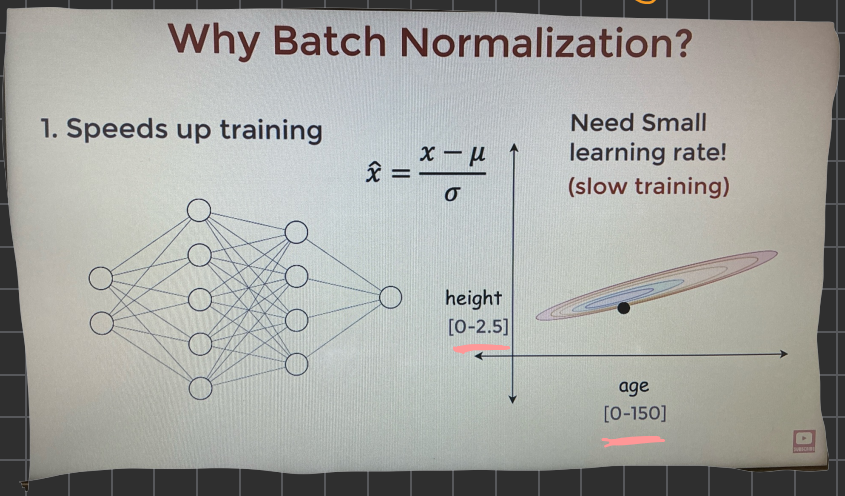
Normalizing input speeds up the training process.

↓

then why not normalize activation layer.

Why batch normalization?

① Speed up training

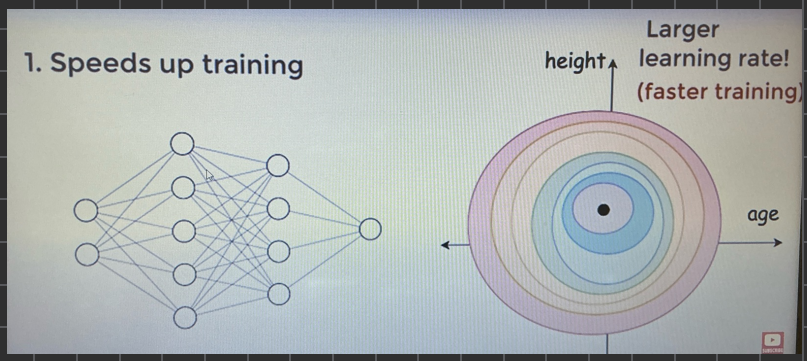


Small variation in Height can greatly change the loss.

↓

and Hence, we would need small learning rate, so that we don't shoot the minimum

& with normalizing



with normalizing we can take larger learning rate thus reaching minimum faster.. that said, we can also use adam.

② Allow suboptimal starts → freedom with initial weights, as with batch normalization initial weights are less important.

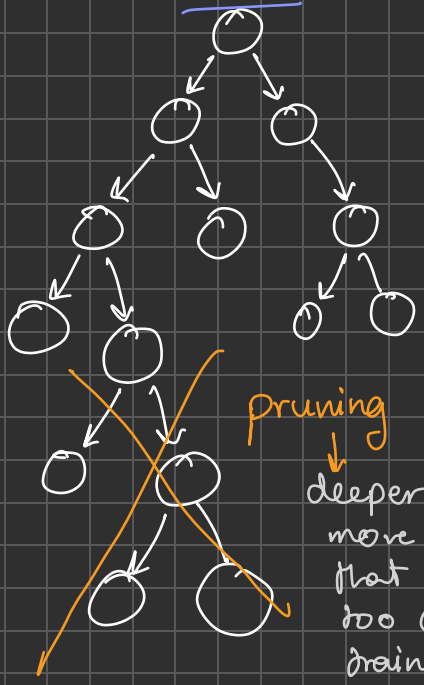
③ little bit → acts as regularizers

Regularization → technique to reduce overfitting

Linear Regression / GLM

$$\sum_i (y-\tilde{y})^2 + \lambda \sum_i \beta^2$$

Decision Tree

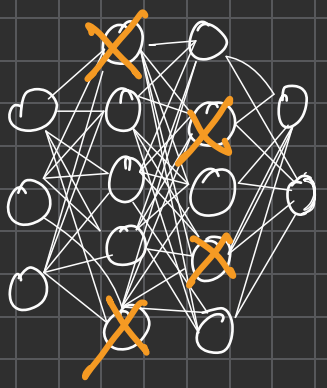


pruning

↓

deeper the tree more chance that it will be too catered to training data

Neural Network



dropouts

2 ways to regularize

↳ (1) Constraining the model → constraining the flexibility of model or fewer degrees of freedom. → to handle overfitting

↳ Dropout | L_1/L_2 Reg | Early stopping

Convolutional Neural Network

Ann is sufficiently good for image classification of handwriten note However, if there is a shift or complexity like a current picture ANN by itself would not be sufficiently handle it.

Image Size → 1920 x 1080 x 3 ~→ 6 million → 4 million

1st layer 2nd layer

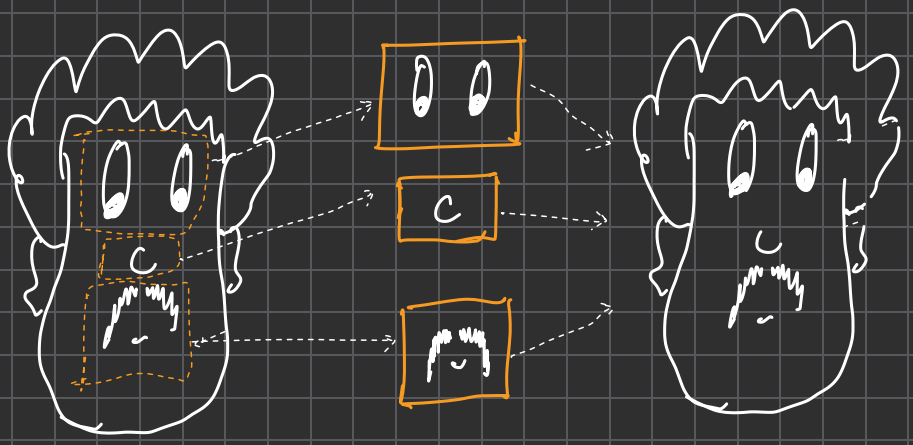
many weights.

↙

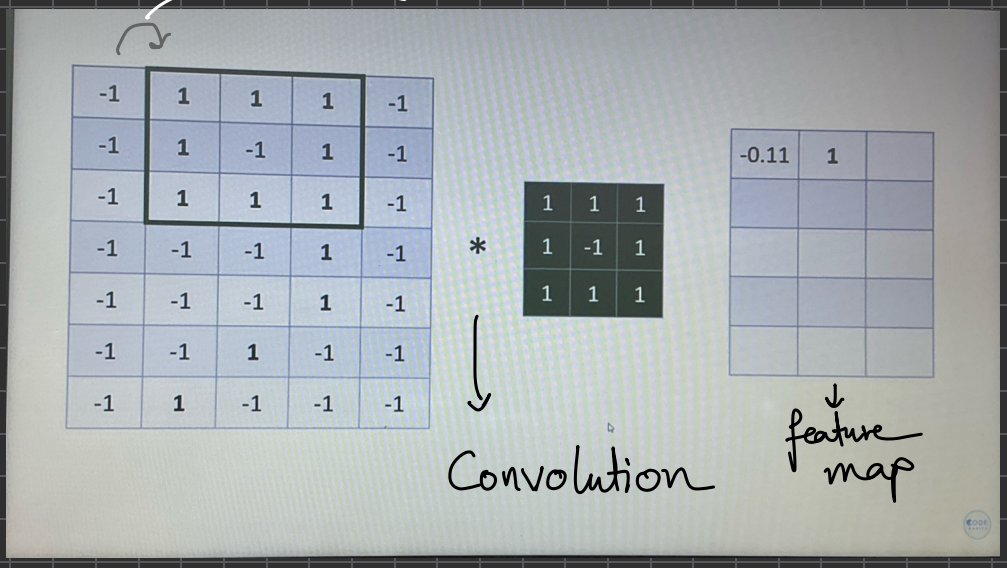
• Too much computation.

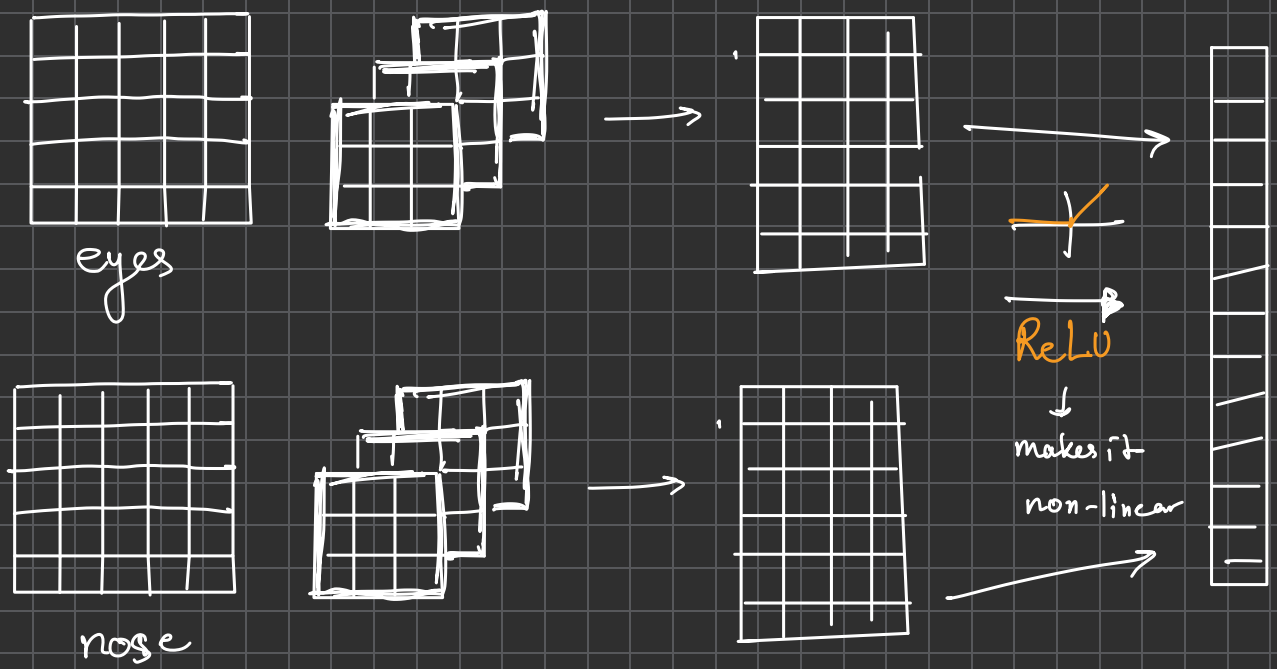
↙

CNN



convolution / filter operation





But the problem of computation is still there

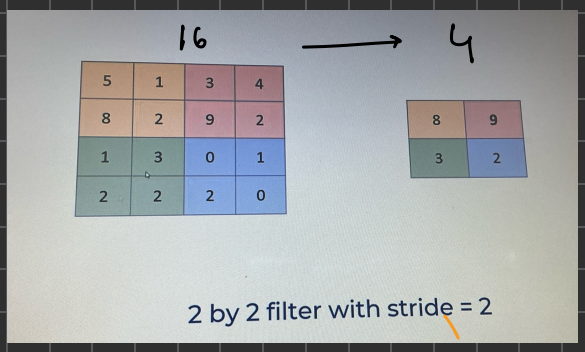
↓

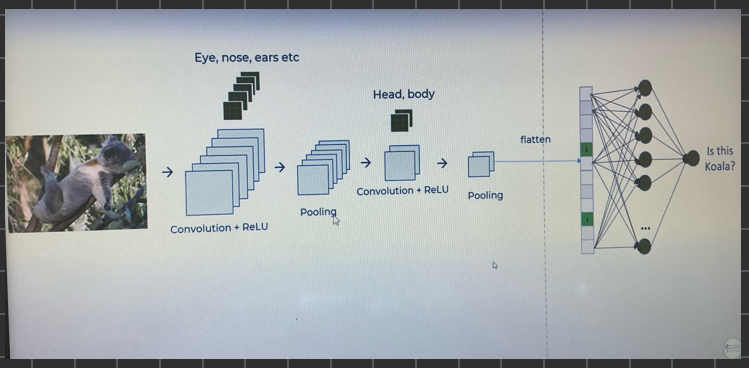
Pooling → to reduce the dimension

↓

Max pooling

Complete CNN





Convolution Layer

- Connection sparcity → reduces overfitting
- Conv + Pooling → location invariant feature detection

Relu

- reduces non-linearity
- faster to compute

Pooling

- reduces dims and hence computation
- reduces overfitting

rotation and thickness can not be handled

↓

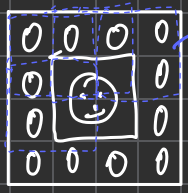
augment training samples → manually

The idea is the weights of the kernel, will be randomly initialized and with forward and backward propagation, iter after iteration, these weights will keep on getting updated. → thus training the model.

Convolution → element wise multiplication + Addition
↓ to capture important element/info from image,
↓ thus, multiple kernels, each carrying essence of individual information

Because, we don't want to loose information at the image border.

↓ padding →



① Maintain Output size
② Prevent information loss.

strides → with small steps we capture minute details, while with larger steps, we might miss out on minute details. However, it totally depends on the Scenario.

Pooling → by summarizing the context / pixelate / downscaling the image, it solves for → location invariance
scale invariance
rotation invariance

Image Net → {dataset}

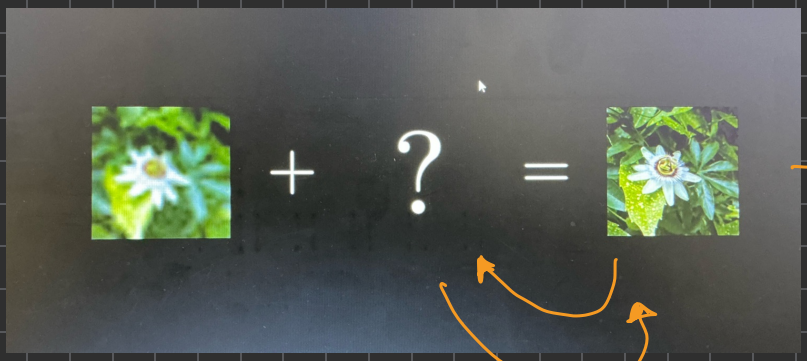
ConV
Single layer model → Le net

Multi layer model → Alex Net → VGG

Conv 2D → solves for images

Conv 1D → time-series data

Conv 3D → image + time series data

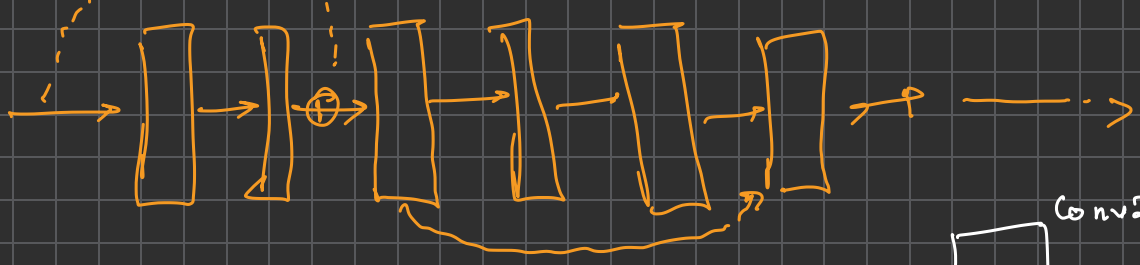


blurred + residual = Upscaled Image

there was a problem on vanishing gradient,

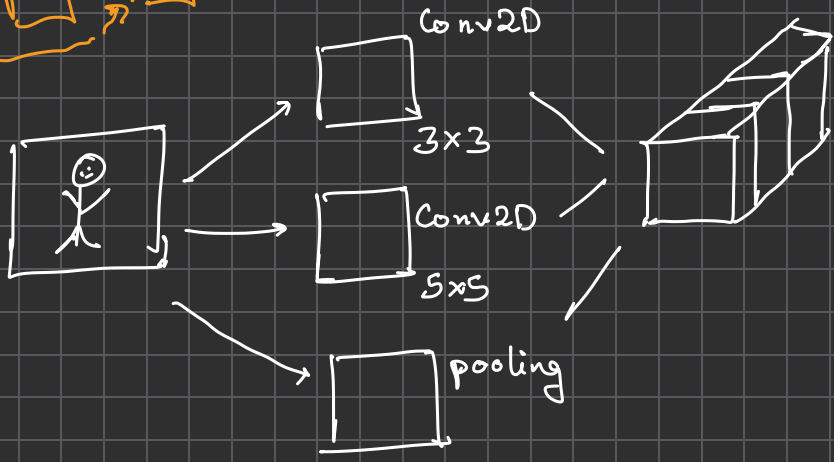
↓

ResNet → the concept is ; it will skip connection thus handling bias mess.

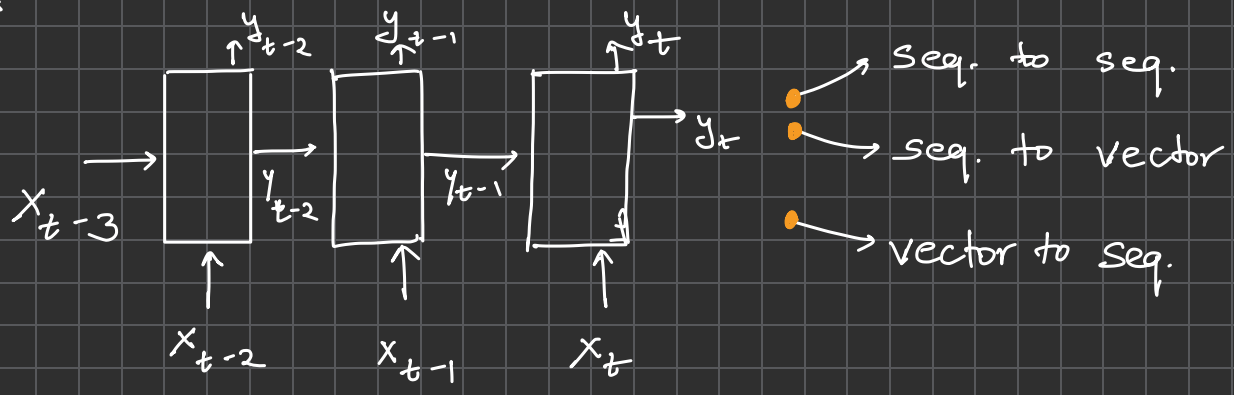


→ ResNet50 → pretrained model
50 residual block
↳ each 3/4 CNN

Inception Network →



RNN → Recurrent Neural Network



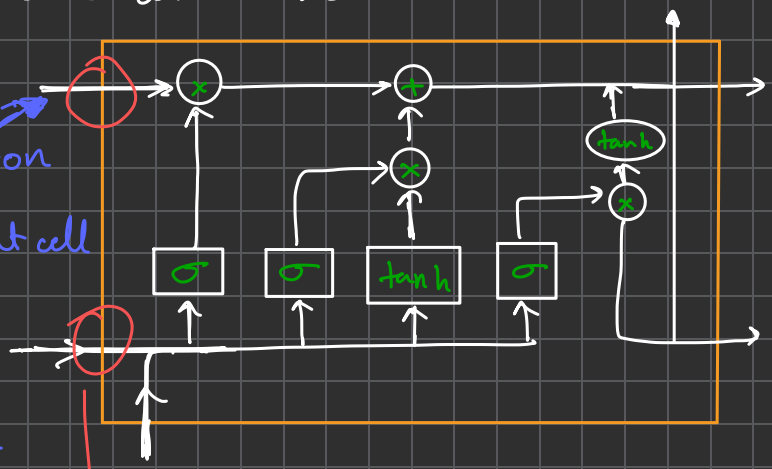
Deals with sequential data

Disadvantages → slow to train

→ fail for long seq. → due to vanishing gradient

LSTM { Long Short term memory }

Instead of neurons → replace with LSTM cell



Past information to skip lot of processing of current cell

↓

Helping memory to be retained for longer sequences.

RNNs are slow but LSTMs are even slower, because of complexity. → and the input data needs to be passed sequentially. → Not taking advantage of GPUs at all

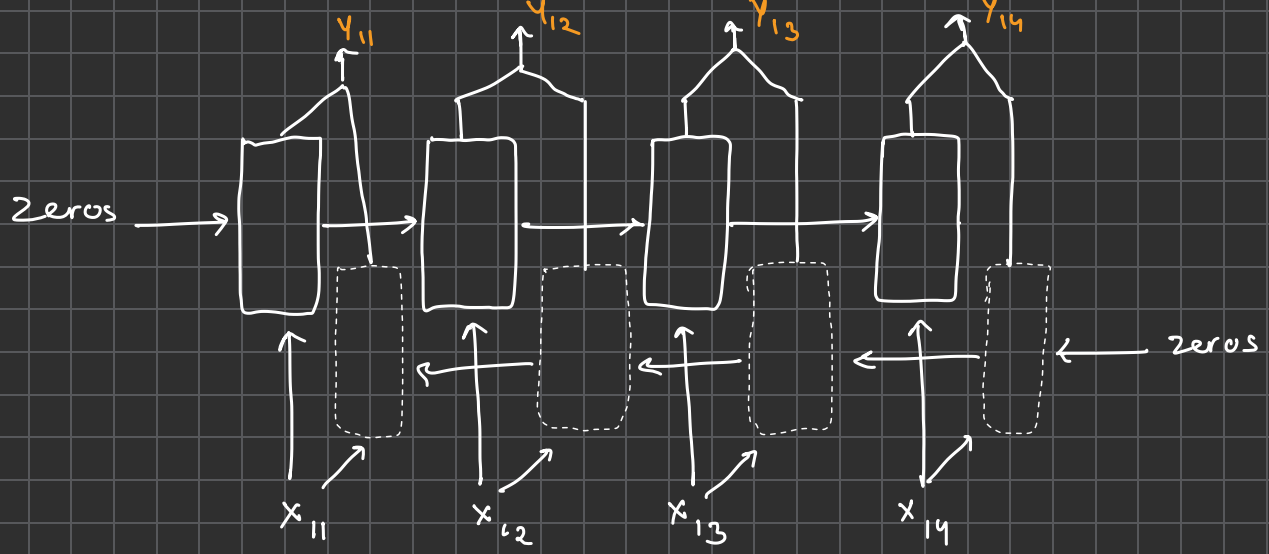
As an alternative we also formed bidirectional RNN.

as in a sentence, x_1 x_2 x_3 x_4 x_5

y_1 y_2 y_3 y_4 y_5

y_3 could be dependent on x_5.

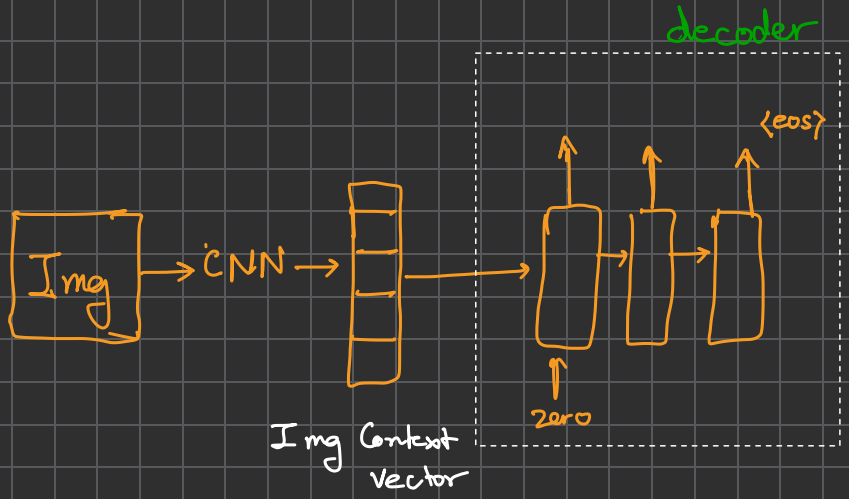
Thus, bidirectional RNN



However, for a longer seq. of text, this wasn't a good idea either.

Encoder - Decoder models

① Image Captioning

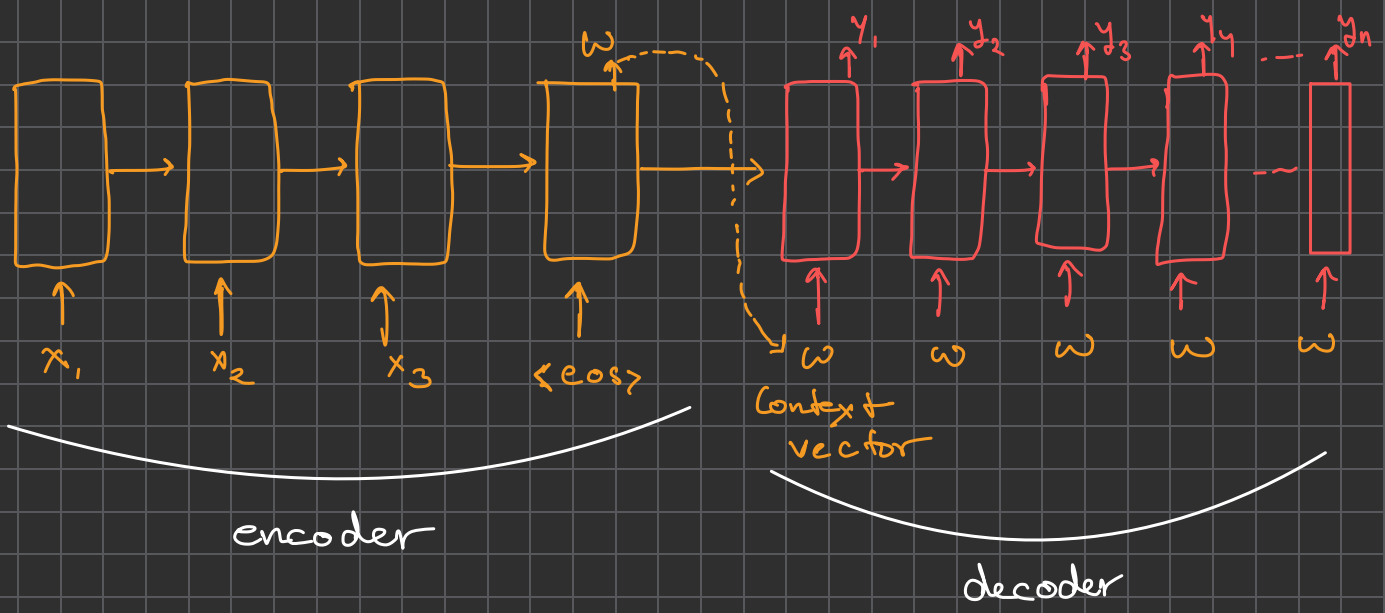


② Seq. to Seq. model → translation

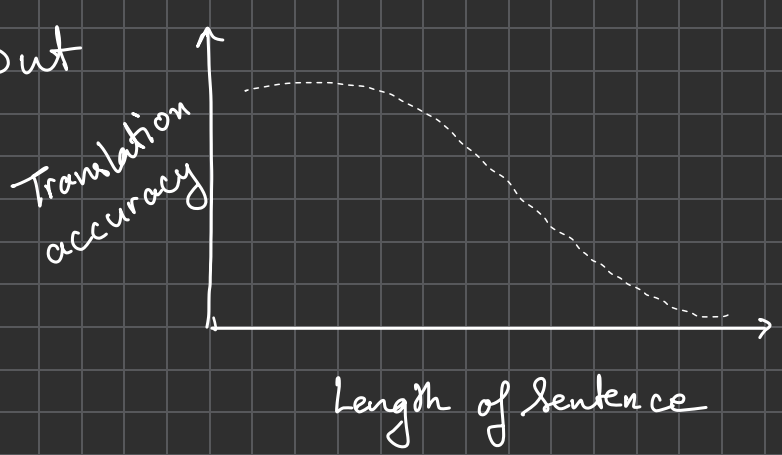
email - Auto reply

code - errors

discussion

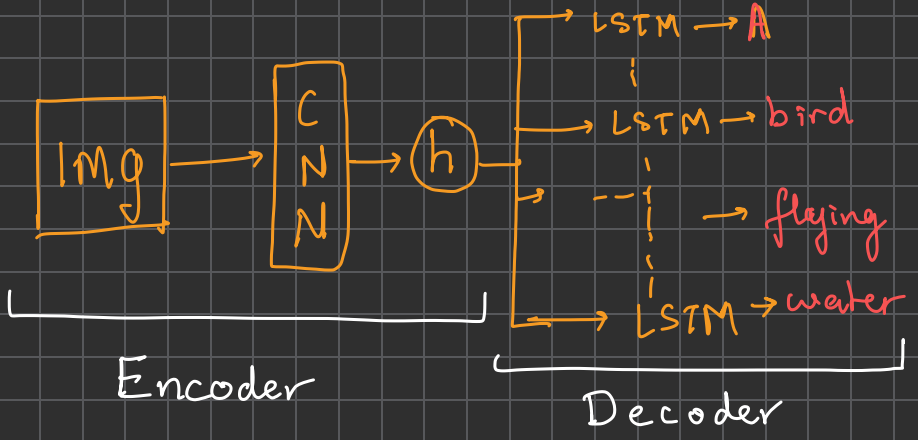


but



for a simple Encoder-Decoder

Attention Mechanism → Giving attention to specific eg: word in a sentence. Just like how we humans do.

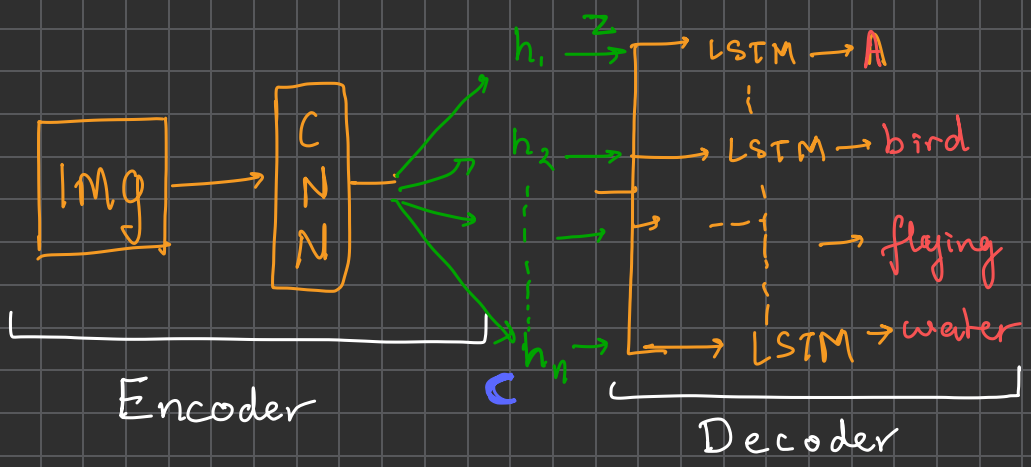


Problem with this approach is for generating caption LSTM looks at entire image representation 'h' every time

↙

to solve this

↙



h → Attention model

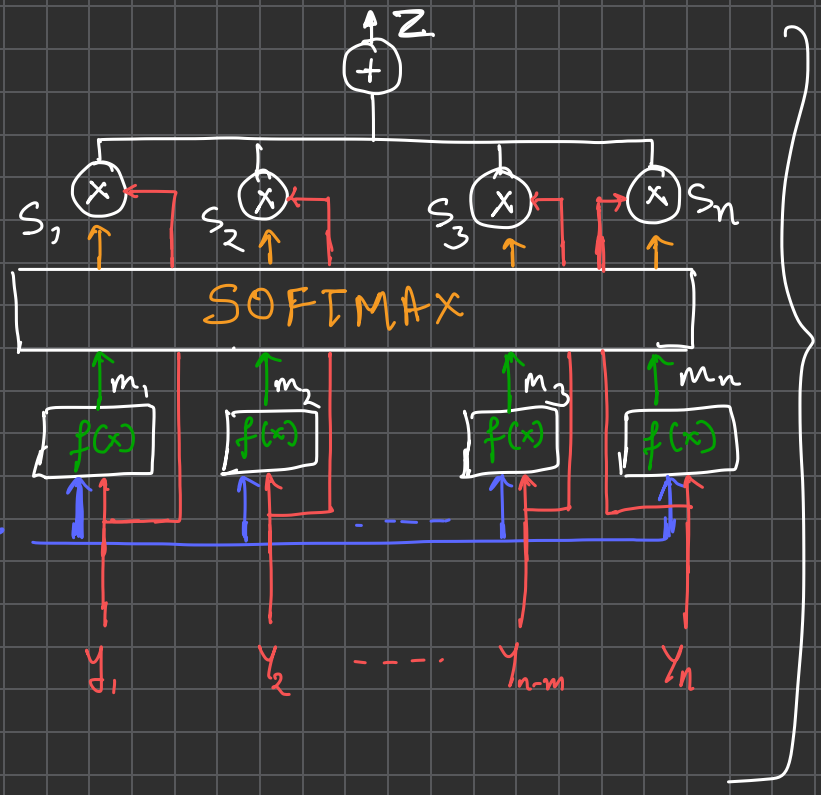
z → weighted Arithmetic mean

How does it actually decide the region to focus?

$$E(x) = \sum_n P(x = x_n) \cdot x_n$$ → inner product of Actual values and their probabilities [Arithmatic Mean]

?

How are these determined? → by context C.



Attention Unit

$m_i = f(y_i W_{y_i} + C W_c)$

tanh → Higher/Lower values will return closest to 1 → giving more attention to certain details.

yC → Simplest form. → the more similar → Higher is product → hence, more relevant information has more weightage

$$s_i = \text{softmax}(m) = \frac{e^{m_i}}{\sum_n e^{m_n}}$$

→ $z = \sum s_n y_n$ {inner product}

Input Representation → Embedding space

↓

each word representing some meaning.

↓

Such that in vector space, words with similar meanings are closer to each other.

We can either pre-train these, or use existing trained models → Glove.

I → [0·3, 0·2, ---- 0·9]

am → [0·42, 0·16, ---- 0·24]

happy → [0·29, 0·34, ---- 0·18]

→ but each word can have different interpretation based on the sentence and position

↓

Positional encoding

↓

defining order of words

I → [0·3, 0·2, ---- 0·9]

am → [0·42, 0·16, ---- 0·24]

happy → [0·29, 0·34, ---- 0·18]

+

[0·3, 0·2, ---- 0·19]

[0·24, 0·18, ---- 0·27]

[0·15, 0·16, ---- 0·94]

↓ Contextual embedding vectors

I → [0·16, ---- 0·24] am → [0·14, 0·13, ---- 0·25] happy → [---]

↓

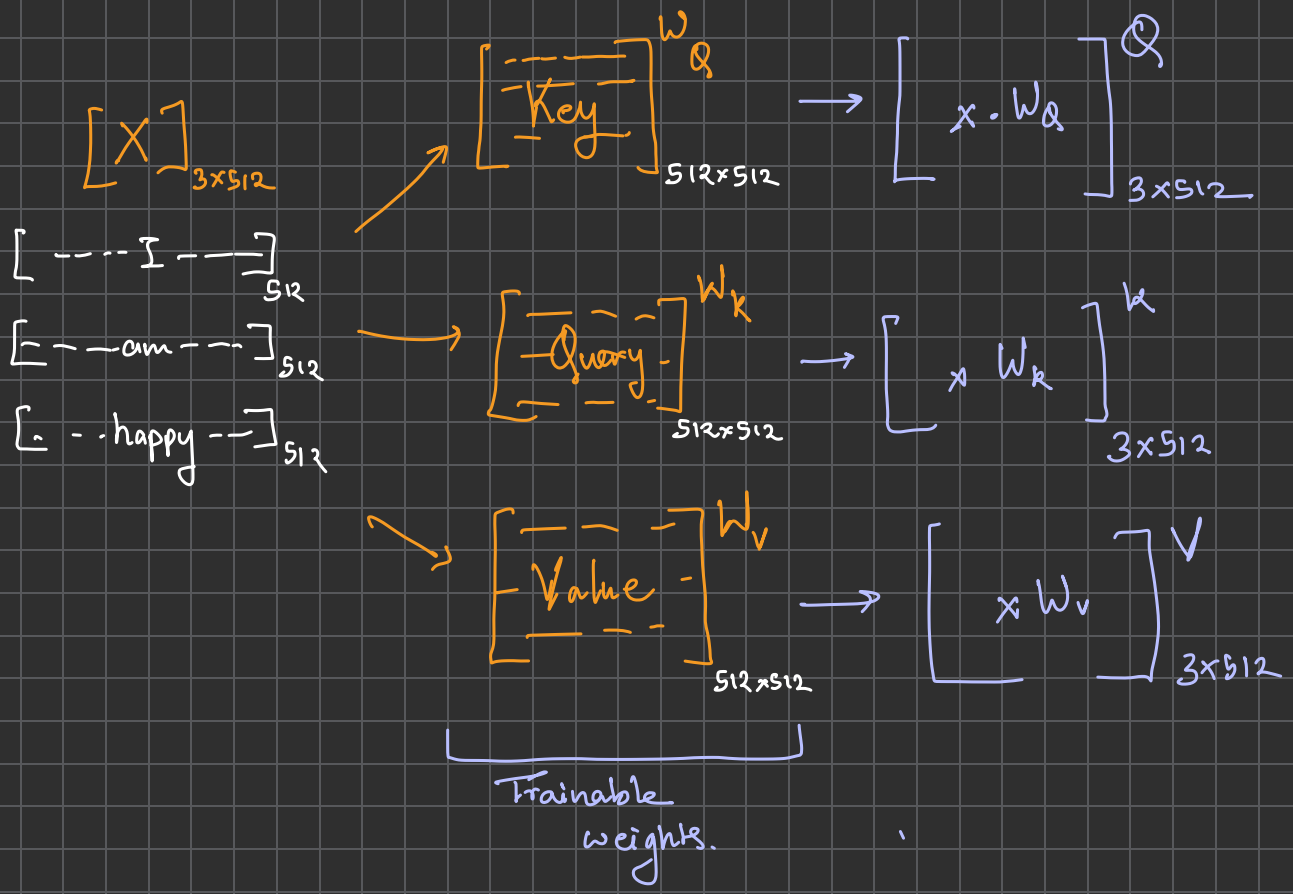
Encoder Self attention Layer

I → [0·16, ---- 0·24]

am → [0·14, 0·13, ---- 0·25]

happy → [---]

} X = [0·16, ---- 0·24 ; 0·14, ---- 0·25 ; ----] 3 × 512



attention score → attention score is given by → $\text{Attention}(Q, k) = k^T Q$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix}_{3\times 512} \times \begin{bmatrix} | \\ Q \\ | \end{bmatrix} = \begin{bmatrix} \cdot \\ A \\ \cdot \end{bmatrix}_{3\times 3}$$

Attention matrix

multiply because

more similar → Higher product → more attention.

To stabilize the gradients we scale by square root of dims. $\sqrt{d_k}$

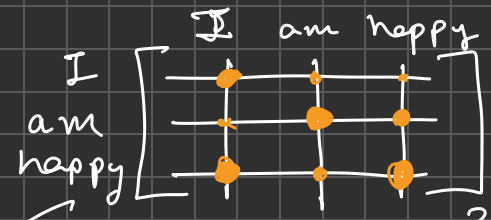
$$\begin{bmatrix} A_{11}/\sqrt{3} & \cdots \\ \vdots & \end{bmatrix} \rightarrow \begin{bmatrix} A/\sqrt{d_k} \end{bmatrix}$$

Now we sending to a softmax function to get normalized scores.

$$\text{Softmax}(A) = \frac{e^{A_{ij}}}{\sum_{j=1}^{3} e^{A_{ij}}}$$

↓

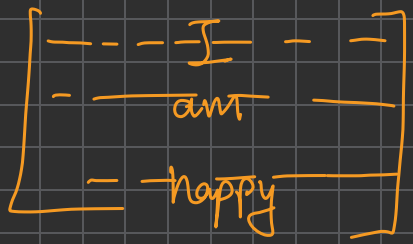
Normalized Attention Matrix



3x3

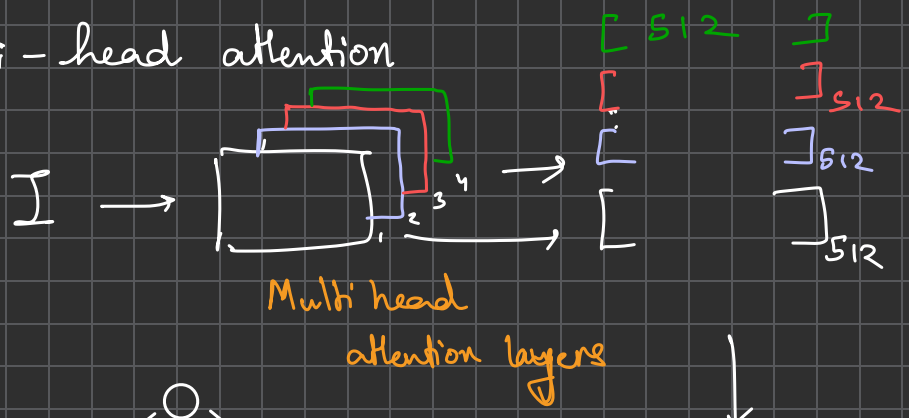
Now we weight it by value

$A \times V \rightarrow$

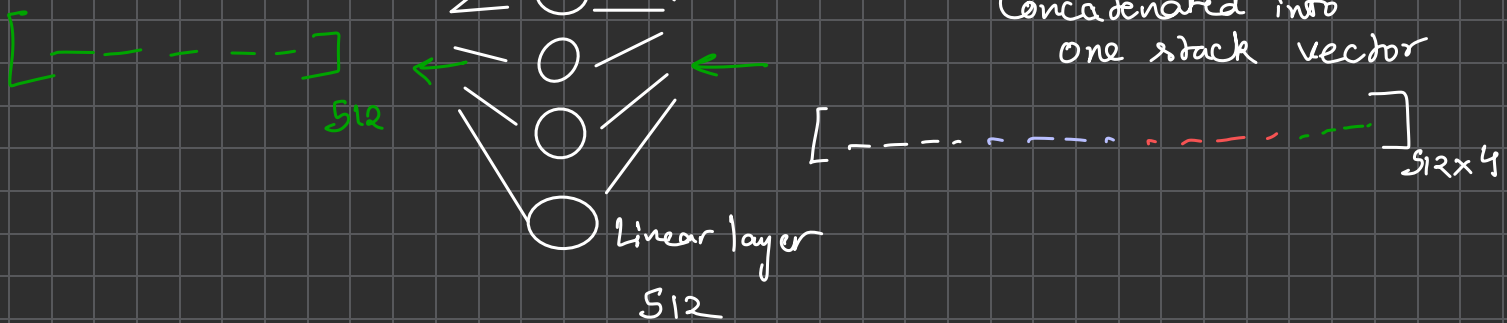


3x512

For a multi-head attention



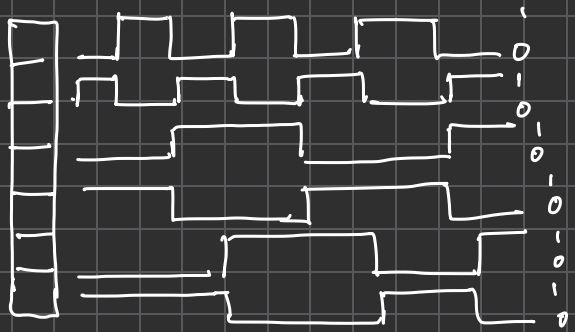
Concadenated into one stack vector



Positional encoding?

word embedding will carry semantic vector representation, but every word based on the position, can have different interpretat and therefore position embedding.

The idea is we add position encoding as a vector of shape same as embedding layer, so that output remains of same shape



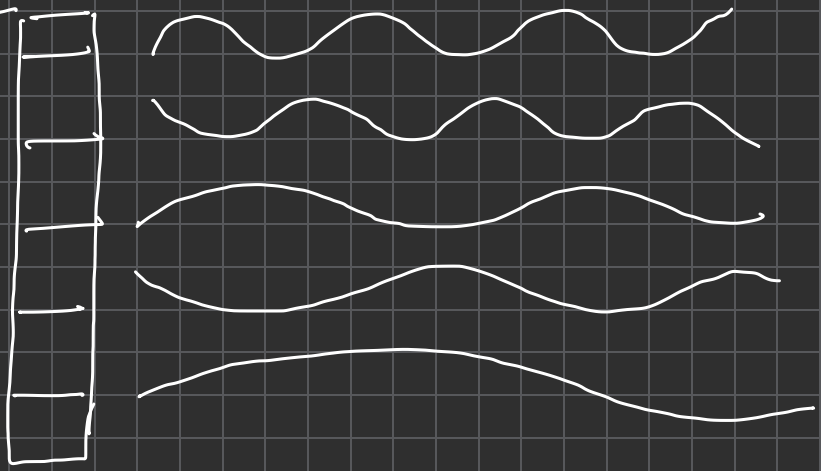
we could have gone ahead with binary values with different frequency for positional encoding.

↙ vector.

but because for a high-dim vector

↙ or for a larger seq.

we go ahead with smothened and rotatory trignometric calculations., ie



thus,
① periodic
② limited [-1, +1]

for even values → $PE(pos, 2i) = Sin\left(\frac{Pos}{1000^{2i/d\_model}}\right)$

for odd values → $PE(pos, 2i+1) = Cos\left(\frac{Pos}{1000^{2i/d\_model}}\right)$

i → index dim | pos → position of word | d_model → embedding len

Self Attention → capture inter-relation

Seq → 6 , d_model = 512

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

$$\text{Softmax}\left(\frac{Q_{6\times512} \times K^T_{512\times6}}{\sqrt{512}}\right) = 6\times6 \quad (\Sigma = 1)$$

$$\times V_{6\times512}$$

→ Attention 6×512 } Self Attention matrix

Multi-head Attention

make 3 copies

Input 6×512 (Seq × d_model)

K, Q, V $(6\times512) \times W \; (512\times512) = 6\times512$

W_K, W_Q, W_V : d_model × d_

K′, Q′, V′ : Seq × d_model

Split them by d_model dim → to capture different aspect of each word

d_model → $d_K = d_m / h$ (# heads)

K′ → $K_1, K_2, K_3, \ldots, K_n$ } Seq

Q′ → $Q_1, Q_2, Q_3, \ldots, Q_n$

V′ → $V_1, V_2, V_3, \ldots, V_n$

$$\text{Attention}(K, Q, V) = \text{Softmax}\left(\frac{K_1^T Q_1}{\sqrt{d_{k_1}}}\right)V_1$$

Seq. { HEAD 1, HEAD 2, HEAD 3, ..., HEAD n

$d_v = d_k$

d_model

→ H × W^0

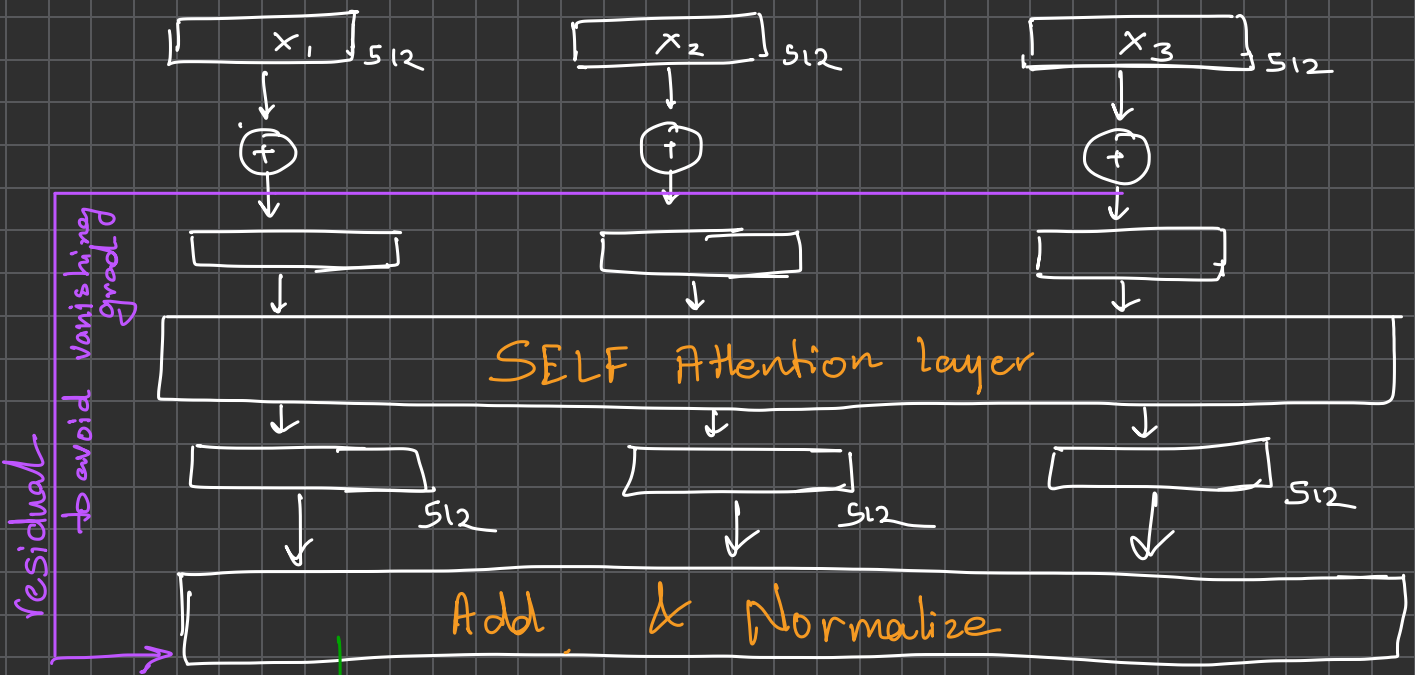
Seq, h×d_v × h×d_v, d_m

or

Seq, d_m

Concat $(H_1, H_2, H_3 \ldots)$ ↓

Multihead Attention



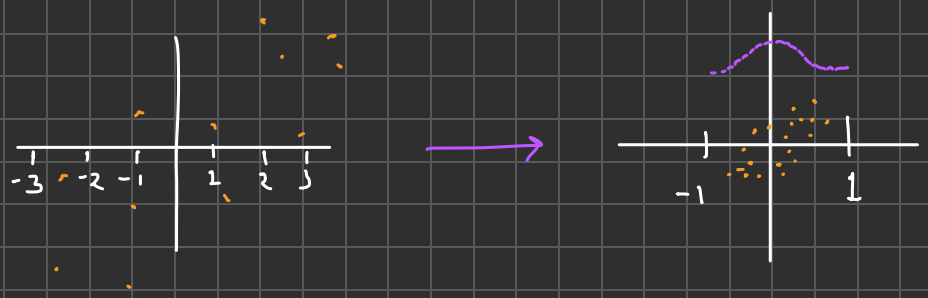
Solves for Internal Covariance Shift

Stabilize training

$$x' = f\left[W_1^T x + b_1\right]$$

$$y = \gamma\left[\frac{x' - \mu}{\sigma}\right] + \beta$$

} Normalization across feature dims.



Input Embedding X → 3×512

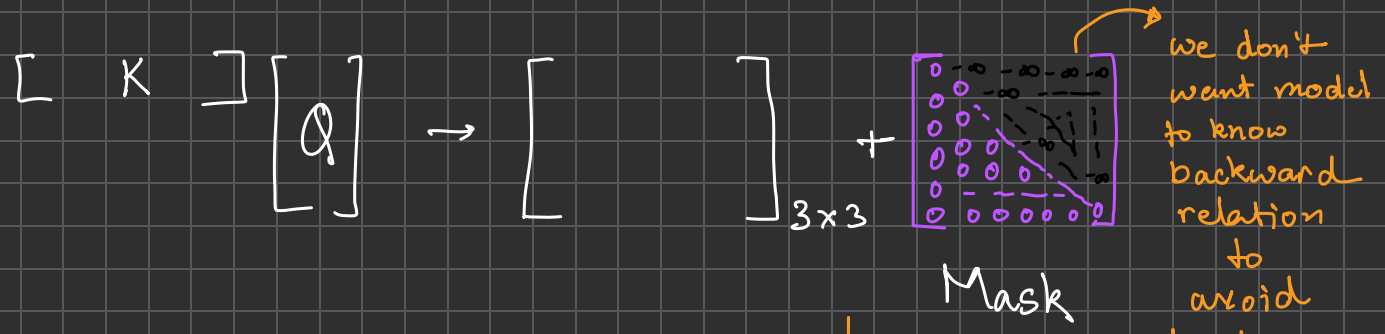
Attention Output → 3×512

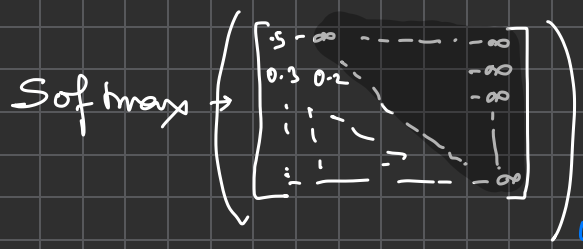
$+$ → 3×512

↙ then Normalize

Decoder

generates → <Start> → <SOS>

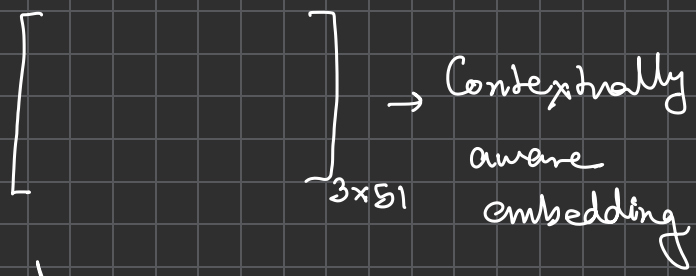




Why -∞?

↓

Softmax → will make them 0.



Multi head → concatenate & Linear layer

Layer Normalization

↓

[3x512]