Building GPT-2 From Scratch

NOTES Covering all topics

Guided by: Raj Abhijit Dandekar Sir (Linkedin)
Notes by: Prasad Chavan (Linkedin)

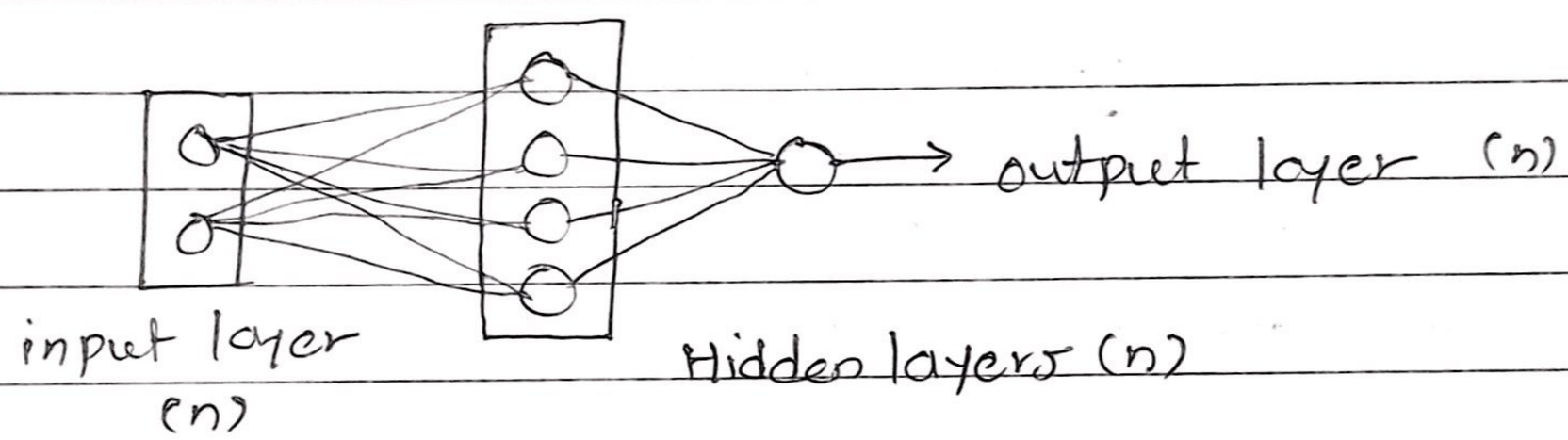
Building LLM's from Scratch

* llm intro - large language Models.
Neural networks designed to understand, generate & respond to human like text.

⇓

Deep NN's trained on massive amount of textual data :)

NN looks like this :-



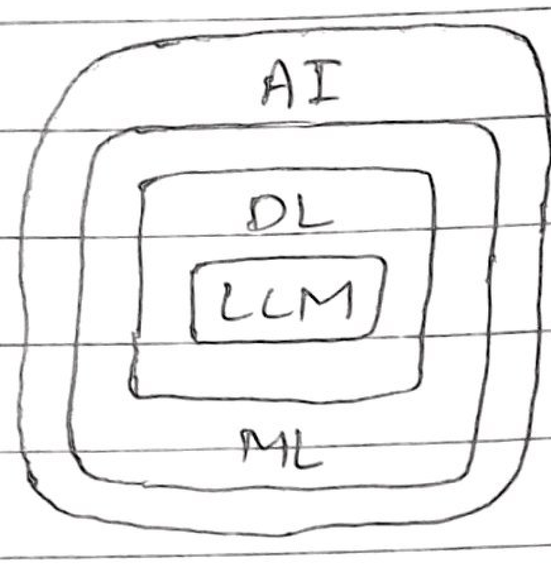
Breakdown of LLM word :-

LARGE — Models having millions/billions of parameters

LANGUAGE MODELS — These models do wide range of NLP or any tasks like : Text generation, classific^n, ~~image segmenta^n~~, etc....

- llm can do wide range of nlp tasks.
- Transformer makes llm more enhanced.
- lets learn in detail about ~~basic~~ basics.

* llm vs GenAi vs DL vs ML



AI ↓
ML ↓
DL ↓
LLM F

GenAi = DL + LLM

* Applica^n of llm's

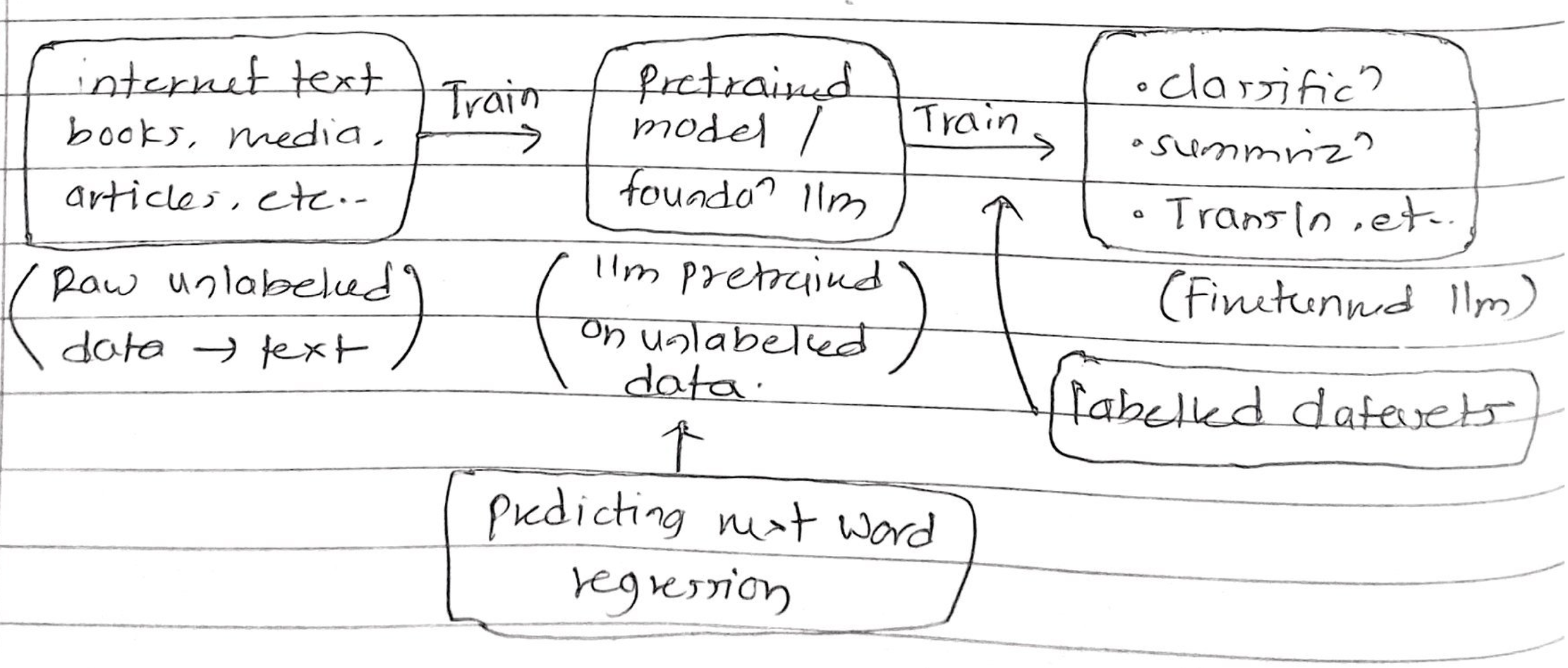
- content creation
- chatbot assistants
- Machine transla^n
- sentiment analysis
- etc...

* Stages of building llm's :-

Creating LLM = Pretraining + Finetuning

Training on a large diverse datasets

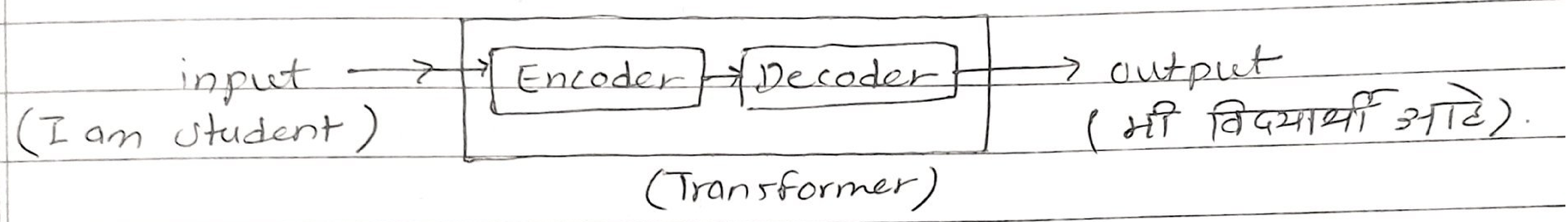
Refinement by training on narrowed datasets, specific to particular task.



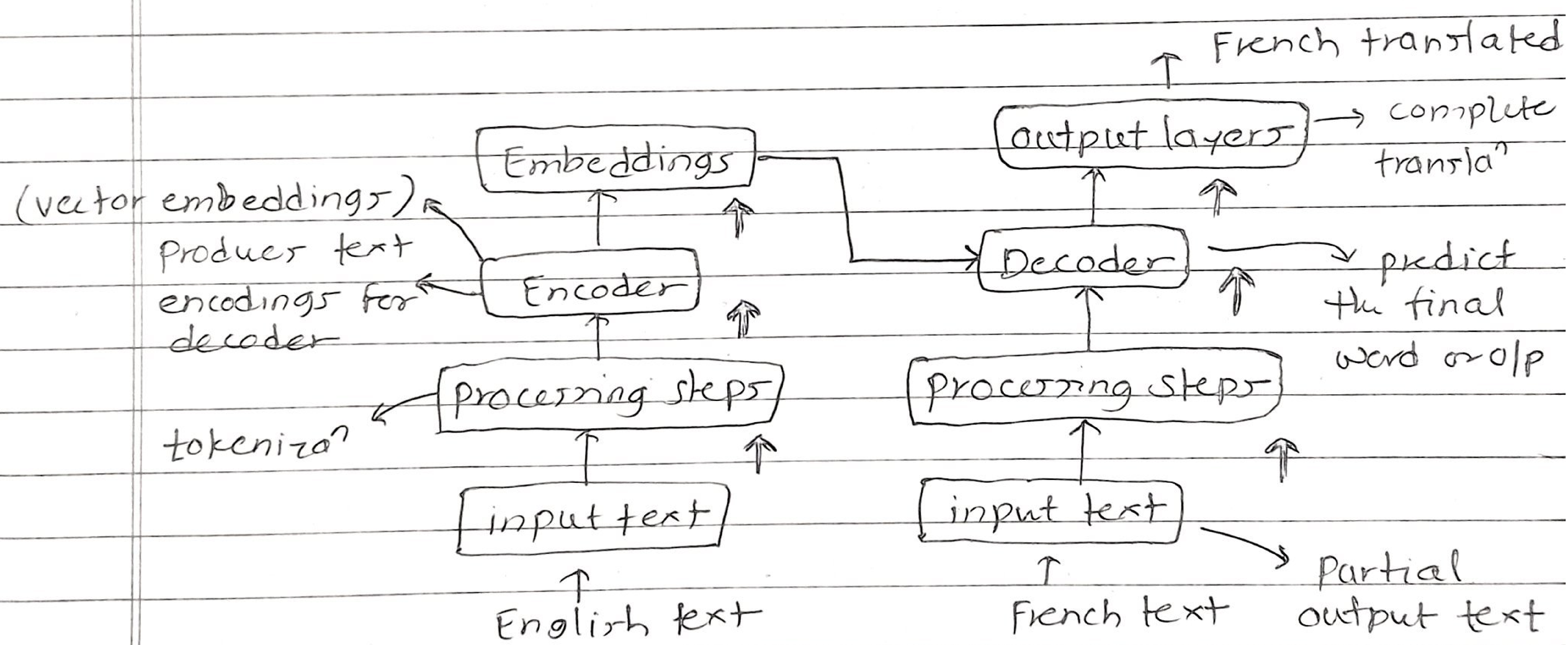
* Finetuning
  - instruc^{n}s finetuning — e.g. summr^{n}, etc.
  - Finetuning for classific^{n} tasks — spam/ham.

* Transformers :-

- Most of llm rely on transformer architecture on paper Atten^{n} is all you need.



- initially transformer was introduced for the translation task.



- Encoder - Generate vector embeddings (semantic meanin)
- Decoder - predict the next seq of words or word.

(Simplified Transformers Architecture)

Page No.
Date

* key point of transformers :-

Self Attention :

- Allows model to weigh importance of different words / tokens relative to each other.
- Enables model to capture long range dependancies.
- e.g. The animal didnt cross street cuz it was tired.

here the it word has attention scores as follows :- it : Animal = 0.9
it : Street = 0.1

* Variations of transformer architecture -

→ Bert - Bidirectional encoder representations
→ Gpt - generative pretraind transformers.

Bert

Predicts hidden words in given sentence.
↓
looks sentence from both left n right side.
e.g. Sentiment analysis. etc...
↓
has only encoder

Gpt

Generates a new word.
↓
single side left → right.
e.g. text genera^n etc.
↓
has only decoder.

How Gpt works

* Transformers vs LLM's

- Not all transformers are llm
- transformers can also be used for computer vision
- Not all llm's are transformers
- llm's can be based on Rnn's or convolu^{nal} cnn's architecture as well.

* Gpt (Generative pretrained transformers)

- gpt takes input sequence and uses attention mechanism to predict next work or token in seq.
- Versions -
  - GPT - 1 : 2018
  - GPT - 2 : 2019
  - GPT - 3 : 2020
  - GPT - 3.5 : 2022
  - GPT - 4 : 2023
- limitations :
  - Hullications - may generate false or mislead info
  - Bias - can reflet bias present in training data.
  - context limit - can lose tracks of info
  - expensive computation power.
- applica^{ns} :
  - chatbot dev
  - content genera^{n}
  - programming assistant

Page No.
Date

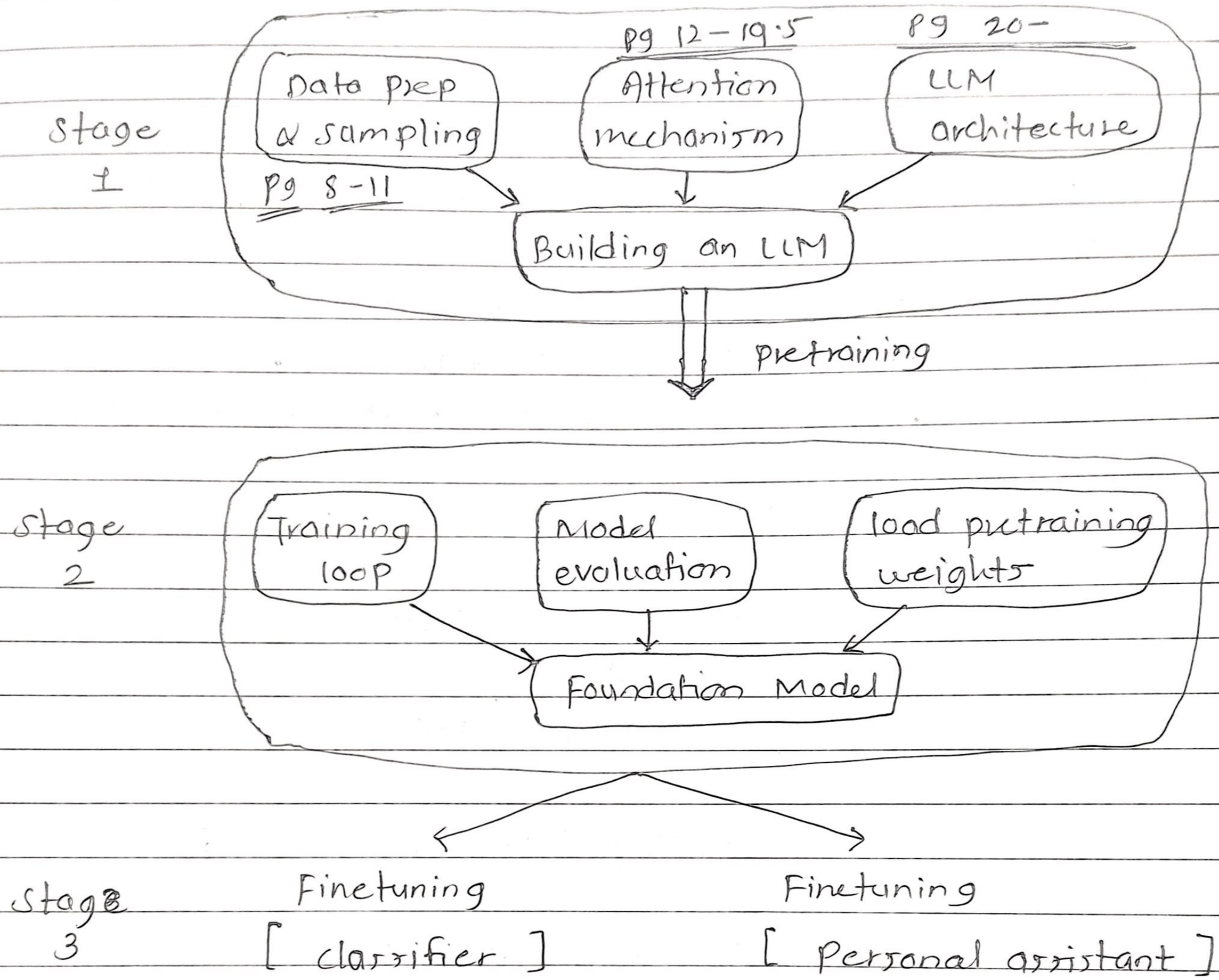
* Zero shot vs One shot vs Few shot

① Zero shot :
Ability to generalize completely unseen tasks without any prior specific egs.
Translate en to hi : ⇒ prompt
Boy ⇒ .... ⇒ input

② One shot :
in addi^n to task desc the model sees a single example of task.
Translate en to hi : ⇒ task desc
I ⇒ मैं : e.g.
Boy ⇒ ..... : prompt.

③ Few shot :
in add^n to task desc the model sees a set of examples of task.
Translate en to hi : ⇒ task desc
I ⇒ मैं } e.gs
Boy ⇒ लड़का
girl ⇒ ..... ⇒ prompt.

Stages of building LLM



Coding LLM tokenizer

Page No.
Date

* Coding the tokenizer from scratch

Q How do you prepare input text for training llm
→

(a) splitting txt into individual word & subword
(b) convert tokens into token idr.
(c) encode token idr to vector representation.

for eg. "I am a boy"

↓

I am a boy

↓

529	63	9	948

⇒ step 1 : Creating tokens from txt file

⇒ step 2 : Creating token idr

⇒ step 3 : defining tokenizer class & coding encode, decode functions

⇒ Handling ~~to~~ special context tokens like for unknown words, endoftext

⇒ step 4 : defining enhanced version of tokenizer

Byte - Pair encoding

* Byte - Pair encoding ---- Used in Gpt's.

tokenization algos

- word based → Problem: what to do with out of vocab words. diff mean of similar words
- sub word based → Byte Pair ....
- character based → Problem: meaning of word is lost. also tokenized seq is much larger than visible txt

* Sub word based tokenization :

Rule 1 : dont split freq used words into smaller subwords

Rule 2 : Split rare words into smaller meaningful subwords.

e.g. 'boy' should'nt be split but
'boys' should be split as 'boy' & 's'.

Advantages :

- Subword splitting helps model learn that diff words with same root word are similar in meaning
- it also helps model to identify sufixes, prefixes & used in syntactic situations
  e.g. ionization, formation ⇒ 'ation' ⇒ suffix.

* BPE algo - Byte pair encoding algo

Most common pairs of consecutive bytes of data is replaced with byte that doesn't occured in data.

Working of algo =>

eg. " aaabdaaac "

⇓ replace 'aa' with 'z'

'zabdzac'

⇓ za → Y

'ybdyc' ... compressed data.

Practical eg. in case of llm :-

- lets consider dataset :

{ 'old' : 7 , 'older' : 3 , 'finest' : 2 , 'lowest' : 1 }

- Preprocessing - adding end token </w> at end of each word

{ 'old</w>' : 7 , 'older</w>' : 3 , ... }

- generate freq table :

Num	token	freq
1	</w>	13
2	o	11
3	l	11
⋮	⋮	⋮

- looks for paining & generates table again ...

Creating I/p-target pairs.

- step to be done before doing vector embeddings.

e.g.s [ ] → input [ ] → target to predict

[I] [am] a boy. → predicting next word.

[I am] [a] boy. → llm's cant access words part the target.

[I am a] [boy.]

input llm receives.

predicted words or o/p.

Data loader

Token Embeddings

* Token embeddings / vector embeddings

Q what are token emb ? & why we need ?

→

first of all using the simple assigning of token ids we get following rslt:

'cat' → 34
'dog' → 12
'kitten' → -20
'puppy' → 96

Here cat - kitten , dog - puppy are semantically related but Here meaning isnt captured due to distance in betn them.

· Semantically similar words should have similar vector embeddings.

e.g.

	"dog"	"cat"	"Apple"	"banana"	
has_tail	20	15	1	1	
is_eatable	2	1	22	38	--- token
has_legs	19	18	0.5	2	embeddings
make_sound	38	32	.9	3	

· features & having high values if having that features like shown above so that semantic meaning is captured.

· we can train NN to create vector embeddings.

Positional Encodings

- in the embedding layer the same token ID gets mapped to the same vector representation regardless of where the token ID is positioned in input sequence.
- its helpful to inject additional position info to the LLM.
- 2 types of positional Embeddings

Absolute

For each posⁿ in i/p seq. a unique embedding is added to the tokens embeddings to convey its exact location

the positional vectors have same dimenⁿ as token embeddings.

suitable when fixed order of tokens is crucial. such as seq generaⁿ.

GPT uses, openAI uses this.

Relative

distance in betⁿ tokens. model learns relationship in terms of 'how far apart' rather than at which exact position.

Adv: Model can generalize better to seq of varying lengths, even if it hasn't seen such lens during training.

suitable for tasks like language modelling over long seqs. where same phrase can appear in diff parts of seq.

{* This positional & token emb are optimized during training process.}

Page No.
Date

* Data processing pipeline

① Tokenization
- word based
- subword based (BPE)
- character based

② Token embeddings — conv tokenids → vec

③ Positional embeddings — encode info abt ps

④ Input embeddings = Token embeddings + Positional Embeddings

Attention Mechanism

- Atten^n mechanism is fundamental comp in modern deep learning specially in NLP
- e.g. Animal didn't crossed street cuz <u>it</u> was tired

here the underlined word <u>it</u> pays more atten^n to animal.

- **Types of Attentions**

Simplified Self attention	Self attention	Causal attention	Multi head attention
↓	↓	↓	↓
Simplified to introduce the broad idea	Self att^n with trainable weights that form basis of mechanism used in llm.	type of self att^n used in llm that allow model to only consider prev & curr ips in seq.	exte^n of self att^n & causal att^n that enables model to simultaneous attend to info from diff repr.

- for e.g lets understand with transla^n eg.

मी मुलगा आहे.
↓ ↓ ↓
I boy am

So here translation process requires <u>contextual understanding</u> & <u>grammer alignment</u>.

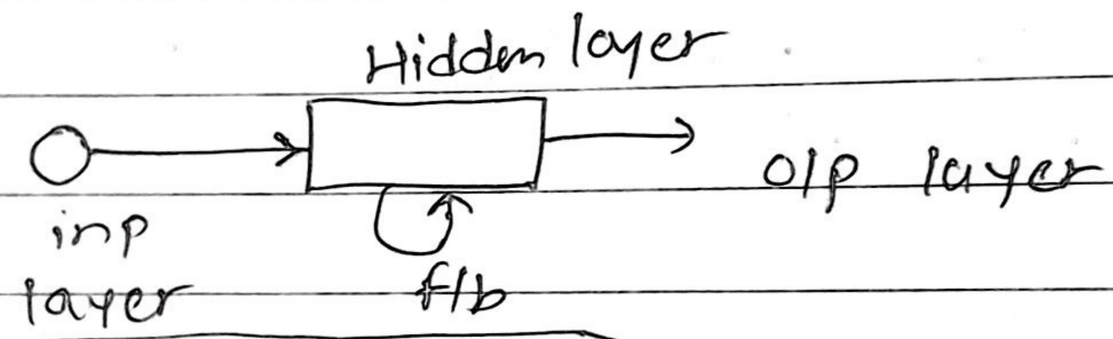
- to address this issue that we cant translate text word by word, its common to use a NN with 2 submodules

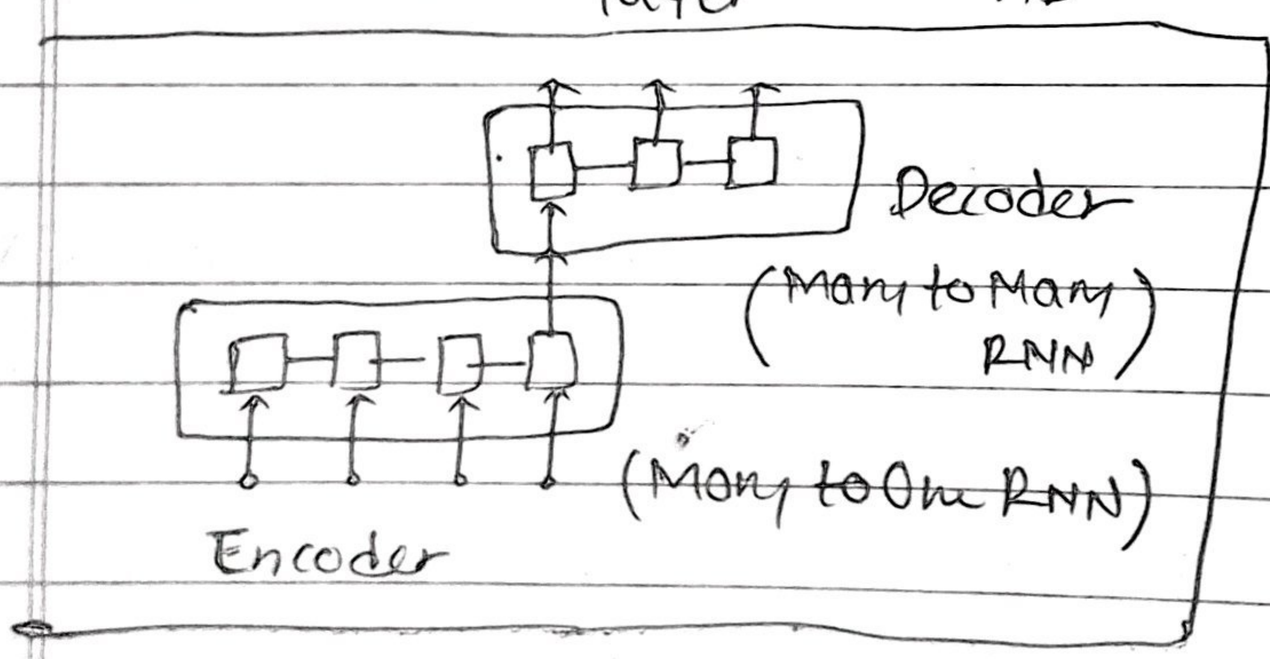
Encoder → Reads & process text

Decoder → o/p the text.

- Before transformers, RNN's were most popular encoder decoders architecture for language translations.

- RNN :





in RNN's the decoder dont have access to previous hidden states so ~~there need for~~ like we provide entire only one vector to decoder by processing in Many to One RNN or encoder. So it have only last vector's access. it may leads in losing info.

* Bahdanau att^n mechanism for RNN's.

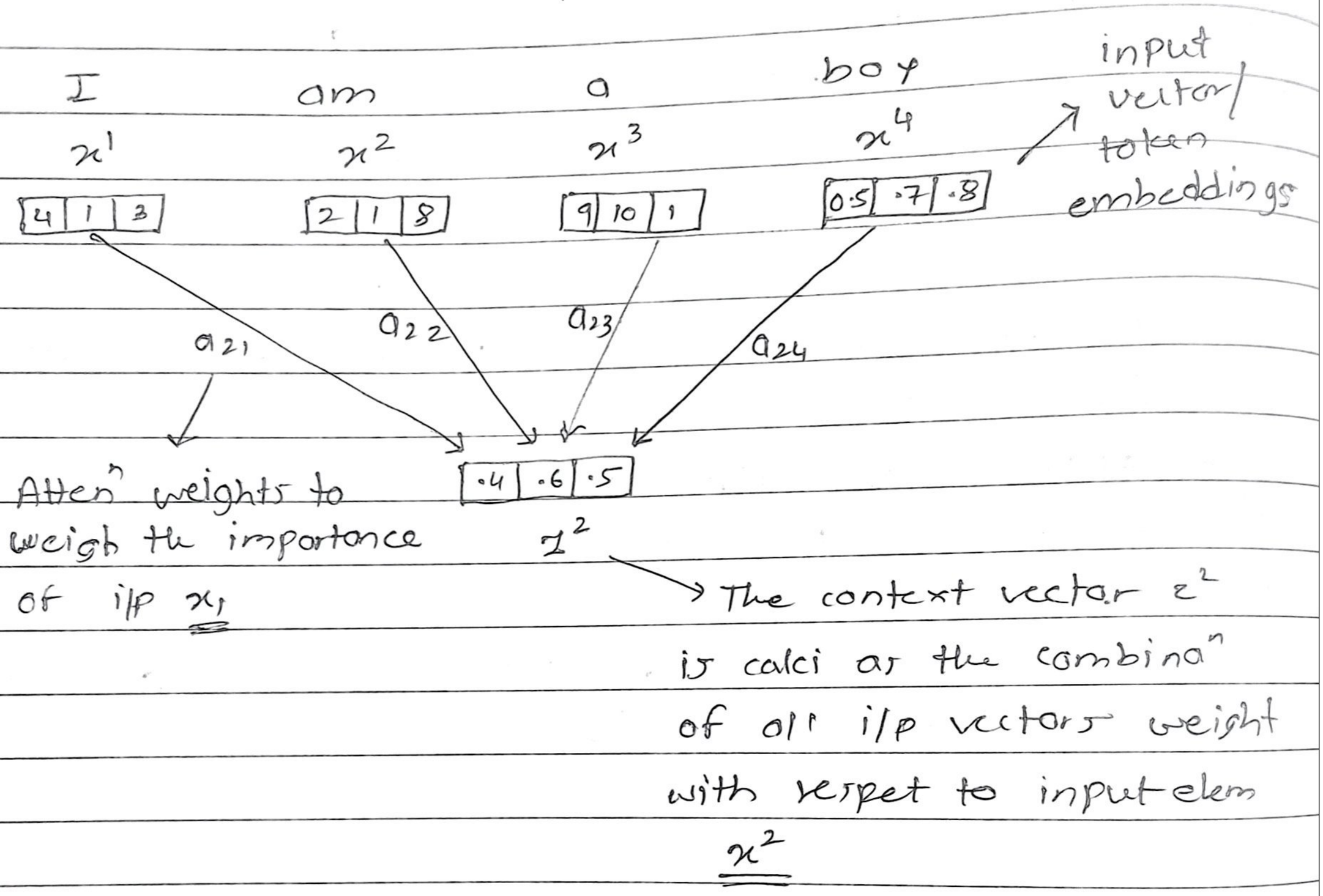
↓

modifies encoder decoder rnn such that the decoder can selectively access diff parts of input seq at each decoding step.

- using an att^n mechanism, text generating decoders part of network can access all i/p tokens selectively.
- this means that some i/p tokens are more important than others for generating a given output token.
- this importance is determined by so called "Attention weights".

IMP

* Implementaⁿ of simplified self attention :
(Non trainable weights)



- We are looking for score of x^2 & the corresponding context vector is z^2.
- z^2 is an embedding which contains info about x^2 & all other input elems x^1 to x^n ...

* Key concepts :

- Query - repr elem we're focusing on
- Key - captures relevant features to match against query
- Value - contains actual info wanna aggregate using attⁿ.

- intermidiate attention score are calculated betn the query token & the input token.
& for this attn score ⇒ dot prod of query token & input token

- in context of self attention dot product determines the extent to which elements of seq attend to one another → (dot prod ∝ similarity & attention score)
(direct proportional)

* Preudo code:

```
q = inputs[1]
attn_score_x2 = torch.empty(....)
for each inputs :
    attn_score_x2[i] = torch.dot(xi , q)
```

* Now for Normalizing them
Normalized o/p (attn_weights) = Attn_score each / sum(Attn_scores)

Here the sum of attn_weights alway equal to one.
- its best practice to use softmax for normalizing the attn_scores.
- in softmax : $\left[\frac{e^{x1}}{sum}, \frac{e^{x2}}{sum}, \dots \frac{e^{xn}}{sum} \right]$

* last step calculating context vectors.

context vector (z^2) = embedded inp tokens vectors (x_i) * attention weights corresponding to (x_i)

* Implementa^n of Self Att^n (Trainable weights)

- Unlike non-trainable weights self att^n the trainable weight self att^n allow model to learn meaningful repr & relations bet^n elems which improve performance.

lets assume, W_Q, W_K, W_V => learnable weight matrices

so →

Query (Q) => xW_Q -- x = input matrix

key (K) => xW_K

value (V) => xW_V

& Trainable vs Non-trainable

	aspect	Non-trainable	Trainable
1)	Q, K, V	same as i/p	projected using the learnable weights
2)	Training req.	NO	Yes
3)	computa^n cost	low	high

- Self attention mechanism uses trainabe wts. so its more preferrable than simplified self attn as it captures more info from data.

- Self attn mechanism also called as "Scaled-dot product".

- Every query has att^n score with all keys

* Steps in self attn^n calculation

① initialize input_ids

② input, output dimensions -- same in gpt's

③ initialize weight matrices Wq, Wk & Wv by populating random values, will optimize using back propagation

④ calculate actual keys, queries & values matrices
keys = inputs @ Wk
queries = inputs @ Wq
values = inputs @ Wv.

⑤ Calculate attn scores
attn_scores = query @ keys

⑥ Taking scaled product with dividing by sq root of embedding dimensions of key. why →
- Reason 1 : for stability in learning
since softmax is sensitive to magnitude of i/ps when i/ps are large the diff bet^n each expo value becomes peaky & o/p becomes peaky.
- Reason 2 : lowering variance. reduce overfitting

⑦ Normalize using softmax

attn_wts = torch.softmax(attn_scores / dk **0.5, dim=-1)

d_k **0.5 => sqroot scaling
dim=-1 => for controling which axis of i/p tensor the func^n should apply for normalizing.

* Causal attention

- Also known as Marked attention, special form of self attention.
- it restricts model to only consider previous current inputs in a seq, when processing any given token.
- This in contrast to self attn mechanism which allows access to entire input seq at once
- when computing attn score, the causal attention mechanism ensures that the model only factors in tokens that occur at or before current token in sequence.

e.g

	I	am	boy
I	0.19	0.16	0.17
am	0.20	0.21	0.19
boy	0.15	0.18	0.16

→

	I	am	boy
I	1.0		
am	0.55	0.4	
boy	0.38	0.2	0.4

→ Marked out future tokens (zeros)

- in this we mask out weights above the diagonal & we normalize the non marked attn weights.

* Workflow -

initial attn scores (unnormalized) $\xrightarrow[\text{softmax}]{\text{apply}}$ Attn weights $\xrightarrow[0\text{'s}]{\text{mask with}}$ Masked attn scores

This process leads to data leakage

Masked attn wts ← normalize rows (unmasked)

so more efficient way for it is -

(Attention scores)

→ (upper triangular infinity mask)

→ (softmax)

DROPOUT

* Dropout layer :

- dropout is dl technique where randomly selected hidden layer units are ignored during training
- prevents overfitting & improves generalizaⁿ performance
- in GPT dropout are applied in 2 specific areas as - after calci attn scores
  - after applying attⁿ wts to value vec
- randomly switches weight on basic of given probability 0.5, 0.3, etc..

* For constructing causal attⁿ compact class having self attn & dropouts →

① initialize random wts (W_k, W_q, W_v)
② calculat queries, keys & values metrix
③ get the attn scores (queries @ keys.T)
④ Upper triangular infinity marking
⑤ softmax (scaling by sq root)
⑥ Attⁿ weights
⑦ multiply by values for context vector matrix.

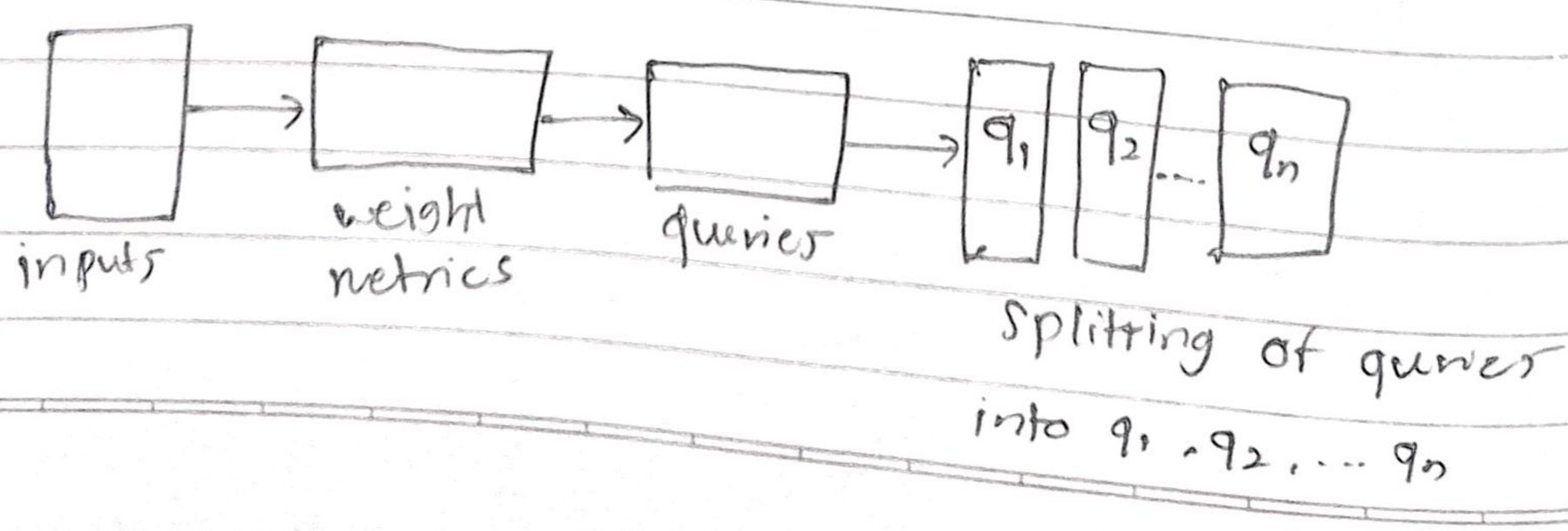
Multihead attention (GPt's)

* Multihead attentions

- Many of causal attentions stacked together to work leads to multihead attention.
- Stacking multiple self attention heads together.
- implementing multi head attⁿ involves creating multiple instances of self attn mechanism. each with its own weights & then combining their o/ps.
- this becomes more extensive in computaⁿ in case of these as multiple heads are given like 96 (in case of gpt) so the efficient way to do this as -

* implementaⁿ of multi head attention with weights split. :-

- instead of initializing multiple weight metrices we will initialize only single weight matrix & then perform the matrix operaⁿ on it.
- this initialized matrix has bigger dimensions than the simple.
- After will split matrix.



- Step 1 start with input lets assume -

b, num_tokens, d_in = (1, 3, 6) repr in 6 dimn

batchsize tokens

x = torch.tensor([[[1.0, 2.0, 3.0, 4.0, 5.0, 6.0],
[6.0, 5.0, 4.0, 3.0, 2.0, 1.0],
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]]])

- Step 2 decide d_out & num_heads

d_out = 6, num_heads = 2

head_dim = $\frac{d_{out}}{num_{heads}}$ = 3

- in gpt based models d_in & d_out must same.

- Step 3 initialize trainable wt matrices for key, query & value (Wk, Wq, Wv)

dimensions = d_in × d_out ⇒ (6×6) matrices

∴ Wk = [ ]$_{6\times6}$ Wq, Wv = [ ]$_{6\times6}$

- complete random initialization.

- Step 4 calculate key, queries & values by ~~doing~~ ~~do~~ doing - (inp * Wk, inp * Wq, inp * Wv)

so, keys, queries, values = (1 × 3 × 6) matrix

b ↙ ↓ num_tokens → d_out.

Page No.
Date

- (Step-5) Unroll last dimension of keys, queries & values to include num_heads & head_dim

head_dim = d_out / num_heads = 6/2 = 3

so transformation like ⇒
hau (1,3,6) ⇒ (1,3,2,3)
(b,numtok,d_out) ⇒ (b,numtok, numhead, headdim)

k,q,v
kshaped ~~qun~~ matrices –

tensor([[[[[. . . . .]] token
[ ]]. 2nd head for tok 1
1st head for tok 1

[[ ].
[ ]].
⋮
[[ ],
[ ]]]])

- (Step-6) group matrices by num_heads.

(b, numtok, numhead, headdim)
(b, numhead, numtok, headdim)

so (1,3,2,3) → (1,2,3,3)
by taking transpose like .T(1,2) ..
for index 1,2. – e.g. ([[[[[ ],
[ ].
[ ]],
⋮
]]]]) 3 tokens

- Step 7 Finding attn scores

Attn_score = Queries @ keys.T (2, 3)

Here we have to exchange index 2 & 3 dimensions like -

(b, num_tok, num_head, num_dim) -- Queries

(b, num_tok, num_dim, num_head) -- keys

attn scores

attn_scores matrix = (1, 2, 3, 3)
(b, num_heads, num_tok, num_tok)

- Step 8 Finding attn weights | Marking

- first marking attn scores to implement causal attⁿ by replacing "-inf"
- divide by $\sqrt{\text{head-dim}} = \sqrt{\text{d-out}/\text{num-heads}} = \sqrt{6/2} = \sqrt{3}$
- Scaling varience by sq root for avoiding the gradient vanish prob.
- Apply softmax function.
- we can implement dropout

- Step 9 getting context vector

context_vec = Attⁿ weights * Values

(1, 2, 3, 3) (b, num_head, num_tok, num_tok) (1, 2, 3, 3)

(....) (b, num_heads, num_tok, head_dim) (1, 2, 3, 3)

- Now we have to merge the dimensions

so -

contextvec = b, num_heads, num_tok, ~~num_dim~~ he

merge

so final context vec will be [ ] 3×d_out.

- Step 10 Reformat context vector

(original dimensions) -- T(1,2) 2 -- (1,3,2,

↳ b, num_tok, numheads, head_dim

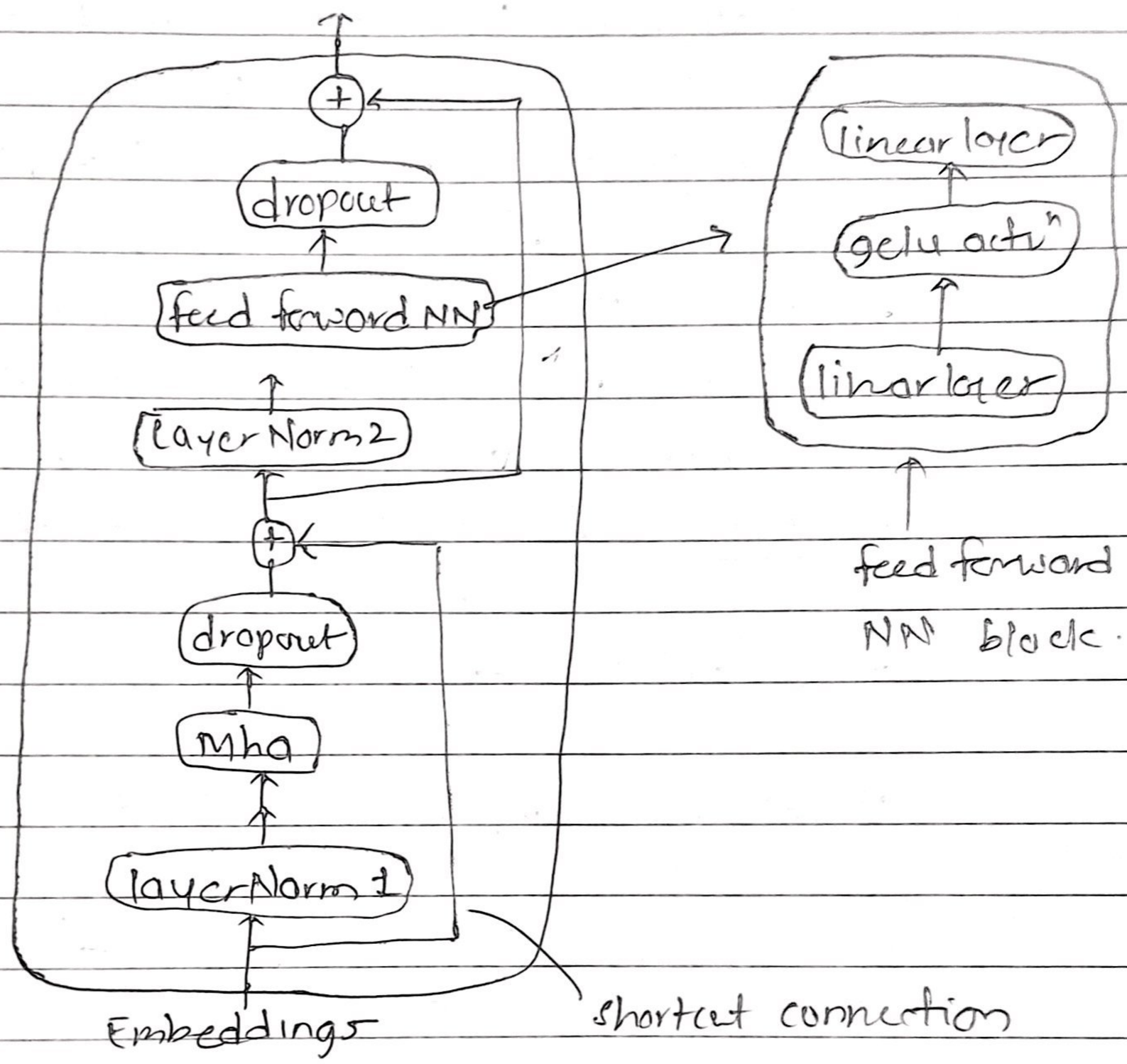
- Step 11 combine heads rslt

so final dim will be -

(b, num_tok, d_out) ⇒ (1, 3, 6)

LLM architecture Bird Eye View

* Transformer block -



for this we are going to learn & train gpt-2 from scratch so the config is - -

* GPT_Config_124M = {

vocab size = 50257
context len = 1024
emb dim = 768
n heads = 12 ... Multi head attn
n layers = 12 ... 12 transformer modules
drop rate = 0.1 ... dropout
qkv bias = False ... query key value bias
}

Page No.
Date

* Normalization layer :

1) Training deep NN's with many layers can be challenging due to -
   - gradient vanishing / Exploding

2) leads to unstable training dynamics
3) layer norm improves stability and efficiency of NN training.
4) Main idea - Adjust ops of NN to have mean zero & varience of one. speeds up convergence.

Here for input seq → we do ⇒

$$\text{Normalized} = \left[\frac{x_1 - \mu}{\sqrt{var}} \cdot \frac{x_2 - \mu}{\sqrt{var}} \cdots\right] = [\cdots]$$

where $\text{mean}(\mu) = \frac{x_1 + x_2 + \cdots x_n}{n}$

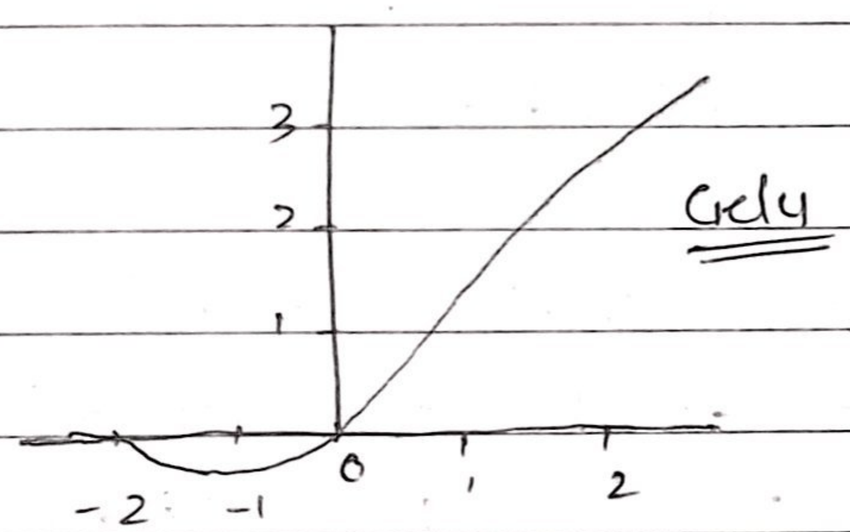
$$\text{varience} = var = \frac{1}{n}\left[(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots n\right]$$

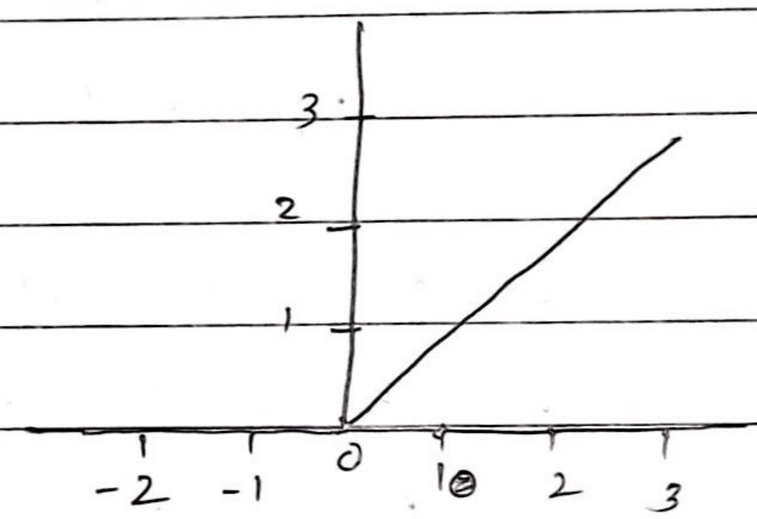
* GELU Activation function : Y feed forward NN :

Gelu = $x * \phi(x)$ → cumulative distribu^n func^n of std gaussian distribu^n.

$$Gelu(x) = 0.5 \cdot x \cdot (1 + \tanh[\sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3]))$$

plot : gelu vs Relu

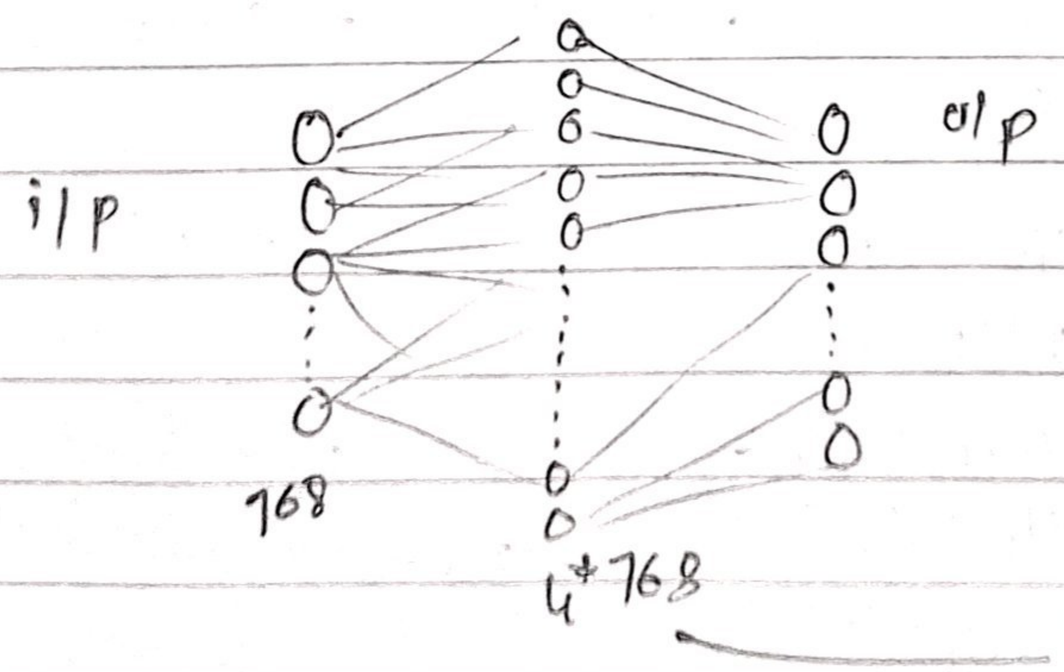




Advantages -

- No problem of dead neuron
- work better with llm

FF NN : in gpts inp neurons are 768 & the hidden layer neurons are 4*768 & o/p as 768



⟹ expansion 768 → 4*768
compression 4*768 → 768
makes rich enhanced.

hidden layer (Gelu)

feed forward →

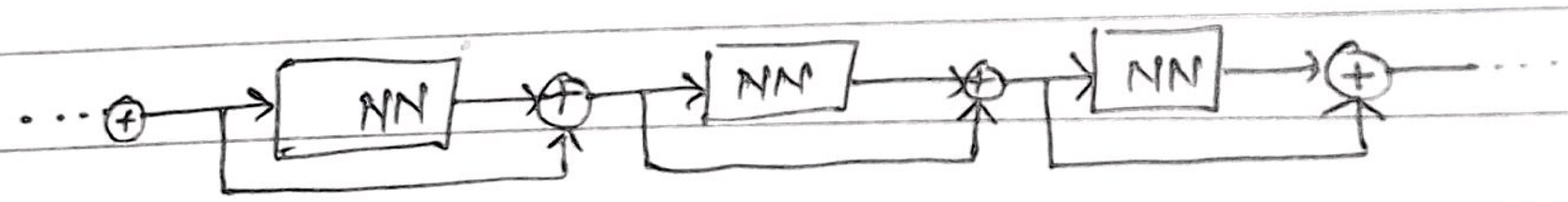
* Shortcut connections -

- Also known as skip or residual connections.
- shortcut connc^{ns} 1st proposed in com vision projects to solve prob of vanishing gradient.

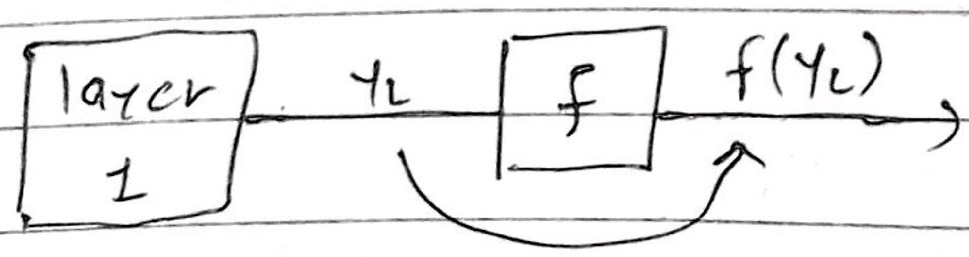
gradient becomes progressively smaller as they propogate backward → making difficult to train earlier layers

due to vanishing gradient the -
$W_{new} \approx W_{old}$ -- updated weights

- to solve this we use shortcut connc^{ns}
- shortcut connections create an alternative path for gradient to flow by skipping one or more layers.
- done by adding output of one layer to output of preceddding layer.



Mathematical intitution :



$f(y_L) + y_L = y_{L+1}$

o/p of 1st layer y_L & 2nd layer is $f(y_L)$

so $y_{L+1} = f(y_L) + y_L$

$$\frac{dL}{dy_L} = \frac{dL}{dy_{L+1}} \left(\frac{dy_{L+1}}{dy_L} \right) + 1 \cdots$$

loss updation wt

keeps gradient flowing thru Network.