Generative AI with LLMs | Andrew NG | Coursera

what is a LLm capable of? basically, they are "next word generators"

so they can do any kind of question answering, summerizing + Entity extraction

augmenting LLMs with connected APIs.

- LLMs are distinguished by their "parameter size" (E.g. 10M (parameters), 2B (parameter))

Larger LLMs: more conceptual understanding → best for complicated & multi-purpose

Smaller LLMs: Less " " but faster → " " focused tasks

- Before the Transformers (structure used in LLMs and latest GenAI), we used RNNs for next word prediction.

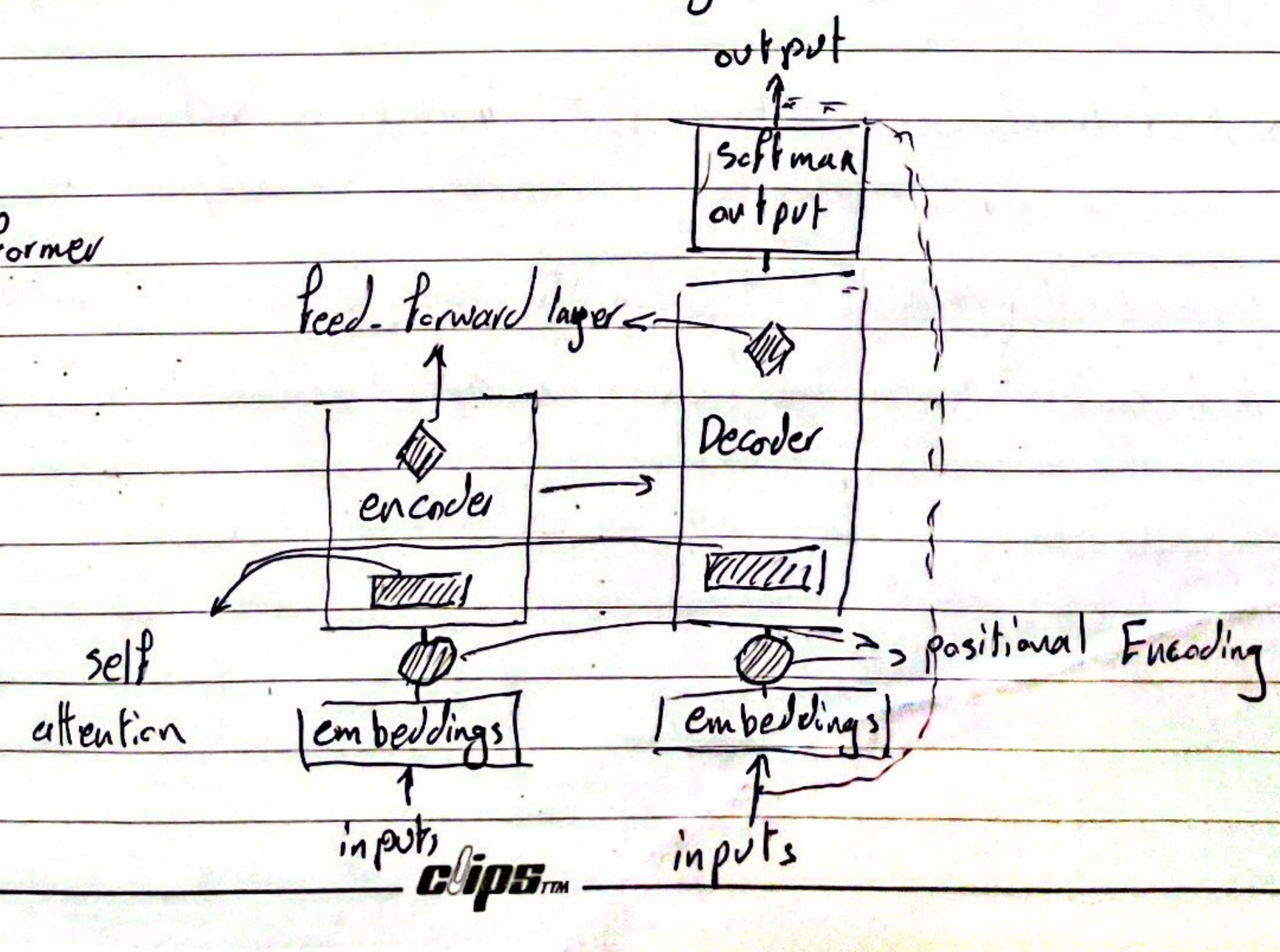
what we lacked?
- memory: longer sentences couldn't be analyzed
- multi-meaning words in sentence
- two/multiple sided sentences

→ solved them using Transformer

But how? the transformers use "self attention": a vector, containing relation of every word/token in a sentence (or even a whole text), called "embedding vector"

input size of an LLm is based on it's size of embeddings (vector)
(max)

structure of a transformer



let's break down some of it's parts:

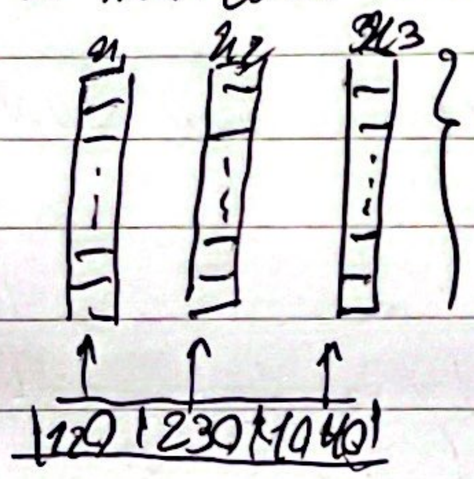
① every input, gets tokenized, each word being represented as a unique ID, based on dictionary in use: the teacher taught the ↓ "Tokenizer" Token IDs [120 | 290 | 1040 | 931]

* use the same tokenizer for both training and generation

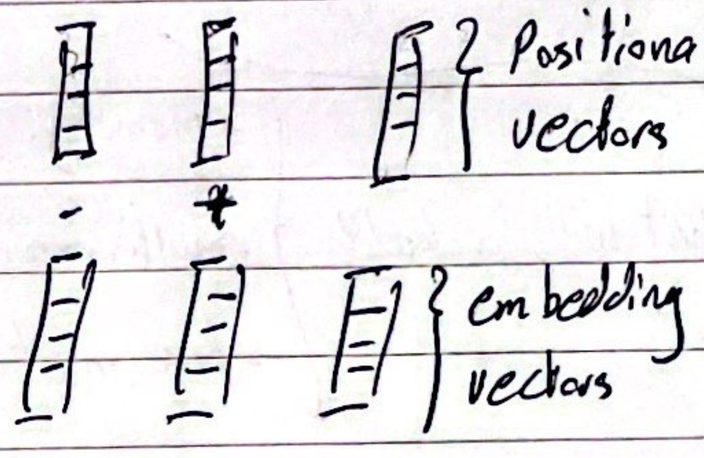
② Token ID vector gets passed to a (learnable) embedding layer.

trainable

so that: each token ID → high dimentional vector → represents the connection between words, etc.



③ Add positional encodings → to preserve the word order



④ feed the resulting vector (embed + positional) into a self attention layer.

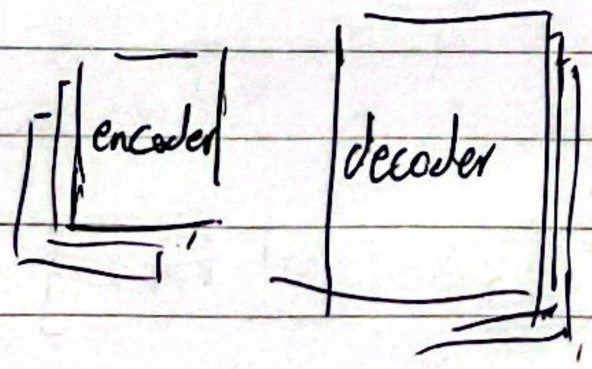
→ which learns to better capture contextual relation between input words

the self attention weights learnt during training, helps the model to capture the relation between each word, with any other word in the input

↳ (and the importance of, hence, "attention")

* in most of transformer architechures, the "multi-headed self-attention" is used. meaning that there are parallel self-attention layers, which are independently learning the importance of each encoding (embed + positional) in the sentence/text. (12 ≃ 100 number of multi-heads)

* in fact, each attention head, learns an aspect of language (people, name, rhyme, ...)

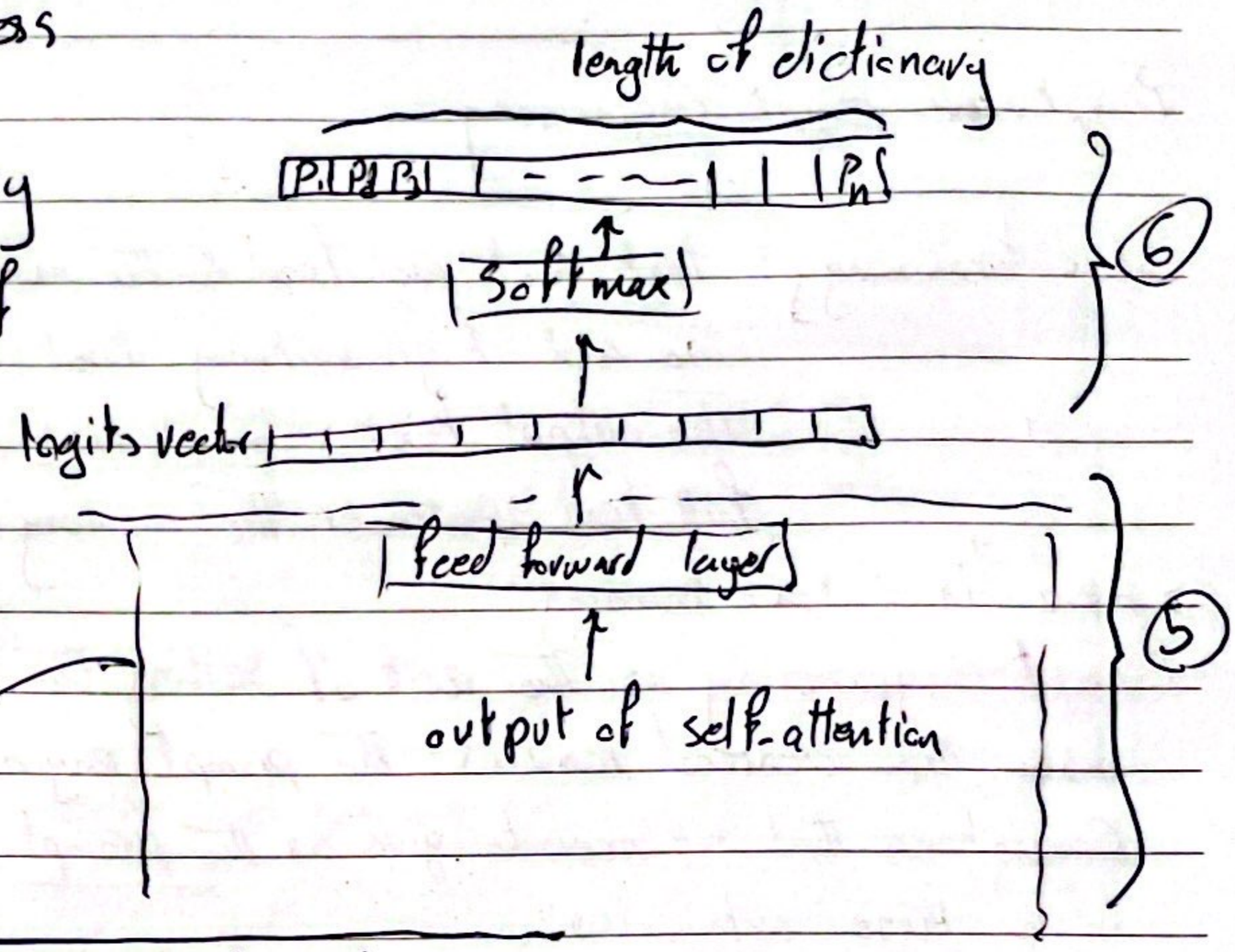


(multi-headed self-attention)

(5) after the attention weights have applied to the data, it will be fed to a feed-forward network, and it's output is a vector of logits, with each value, proportional to the probability score, for each and every token for the tokenization dictionary

(6) The we can feed it to a softmax layer, outputting a vector with length of dictionary, containing prob.-score for each token.
The highest score, in most cases, is the next word that should be used to start/
Continue the word generation process

* there are a number of methods, to vary the final selection, from the output of softmax layer



Generating text with transformers

So, how do transformers work after the training phase?

A) Path of encoder:

Input → Tokenizer → embedding + positional → self-attention → f.f. layer → inserted in the decoder for influence on generation

(attention vectors)

Parallel steps (because multi-head)

B) Path of decoder

Input → Tokenizer → embed + positional → self-attention → inserted vector → f.f. → softmax → generate a token

(from encoder)

This loop continues as long as the tokens finish

other achitectures of transformers:
(seq-to-vec)

① seq.-to-seq encoder-only → used for classification tasks (like sentimental analysis) like BERT

② Enc-dec models → tasks such as text generation → BART and T5

③ decoder-only models → can be generalized to most tasks → GPT - BLOOM - Claude...

Prompt and prompt engineering

some terminology: text that we feed to the model: Prompt — often limited size
the act of generating text: Inference
the output text: Completion
full text space or the memory to use for prompt: Context window

↳ Aka. "in Context learning"

Prompt engineering is the act of setting the prompt, so that the model gives the user the desired results. The prompt engineering, devides by "how many sample Completions that we need to give as the prompt, to have the desired Completion" in to these catagories:

Zero-shot inference: just give the prompt, without any samples.

one-shot " give a single Completed example to the model as prompt.

few-shot " more than 1 sample " " " " " .

in Context learning: The process of training/fine-tuning model using prompt engineering
e.g. zero-shot, —

Prompt: Give the sentimental analysis result:
"The movie was ~~bad~~ good"
result: [ ]

Context window → model → "positive" → Completion

} Zero-shot prompting

Give the sentimental ---- :
"The movie was ~~bad~~ good"
result: Positive
Give the ---------- :
"The movie was nice"
result: [ ]

→ model → "positive"

} one shot prompting

★ In most cases, the ~~best~~ model's completion won't improve above 5/6 shots. start with zero-shot, end with 5/6 shots. if it wasn't satisfying → fine-tune or go for another model

★ larger models (parameter-wise) can easily work with zero-shot prompting.

★ smaller models " may need one or few-shot prompting to give the perfect completion

Generative Configuration

there are some hyper parameters (non-trainable) variables, that can alter and influence the inference phase:

1) Max new tokens: the maximum number of tokens, that can be generated by the transformer. (it's not a hard cap) - (model can end the completion with <eos> token before the max new token value

2) Sample top K: the output of softmax layer, is a vector containing proba values for each token in the dictionary, to be used as the next token.

To pick the next token from this vector, we can use two methods:

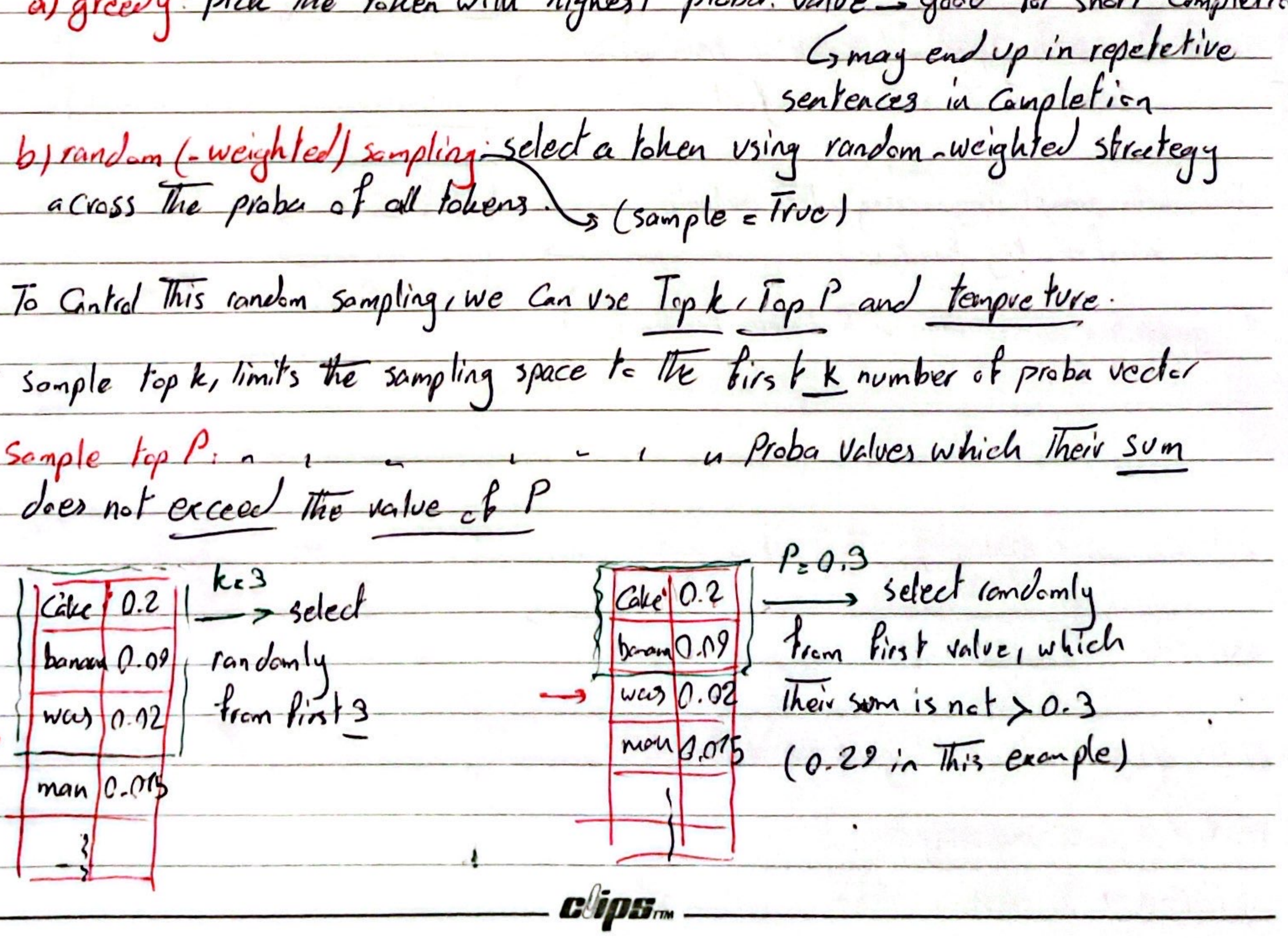
a) greedy: pick the token with highest proba. value → good for short completion
↳ may end up in repetetive sentences in completion

b) random (-weighted) sampling: select a token using random-weighted strategy across the proba of all tokens → (sample = True)

To Control this random sampling, we can use Top k, Top P and temperature.

↳ sample top k, limits the sampling space to the first k number of proba vector

3) Sample top P: n , - , - , - , u Proba values which their sum does not exceed the value of P

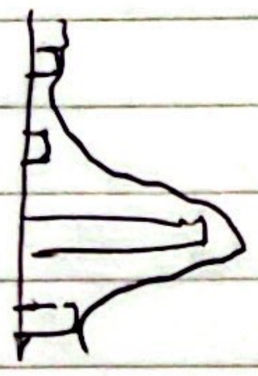


* temp : 1 → default Softmax output

4) Temperature: Directly a floating softmax output, The lower value (≃0) decrease variability (strongly peaked proba distribution)

Cooler temprature (<1)

Cake	0.002
banana	0.01
Nord	0.49
men	0.1



strongly peaked proba distribution

<=> less randomness

higher temp (>1)

Cake	0.3
banana	0.2
word	0.5
man	0.1

more smooth distribution

=> more randomness

lower top k higher top P (≃1) higher temp => more random generation

Generative AI life cycle of project

- most important part of project →

- choose from existing pretraind models or train your own. →
  based on the scope of project

- start with prompt engineering, then evaluate.
  if not satisfied, try fine-tune, ... →

- The generation must align with human feedback

- optimize and deploy for The certain application / use-case →

- LLMs are prone to inventing / hullucination
  => bad at math / reasoning / ...
  optimize it to overcome those