🔥 Can I create a LLM from scratch?

Let's try!
Source of study 📚 Build a Large Language Model - Sebastian Raschka

- My Github repository

⚠️ Knowledge Debt

- Improving the understanding of Cross entropy for loss function

Visao Geral

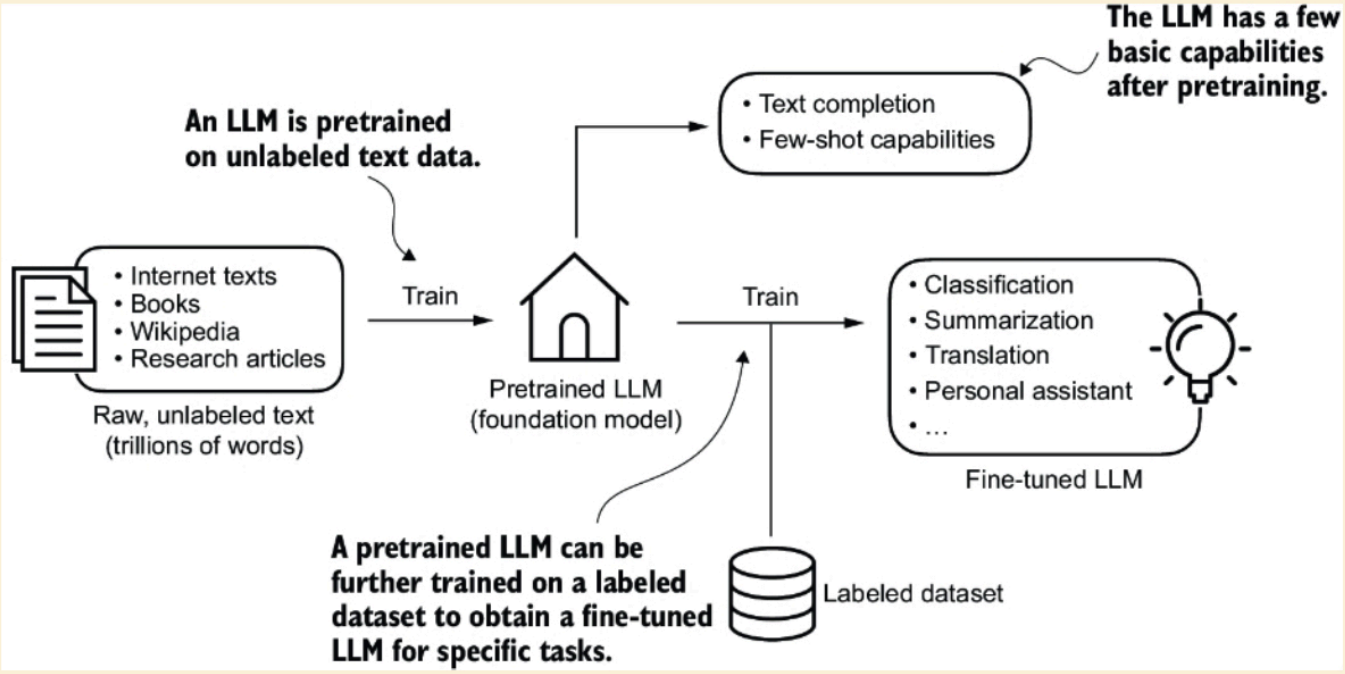
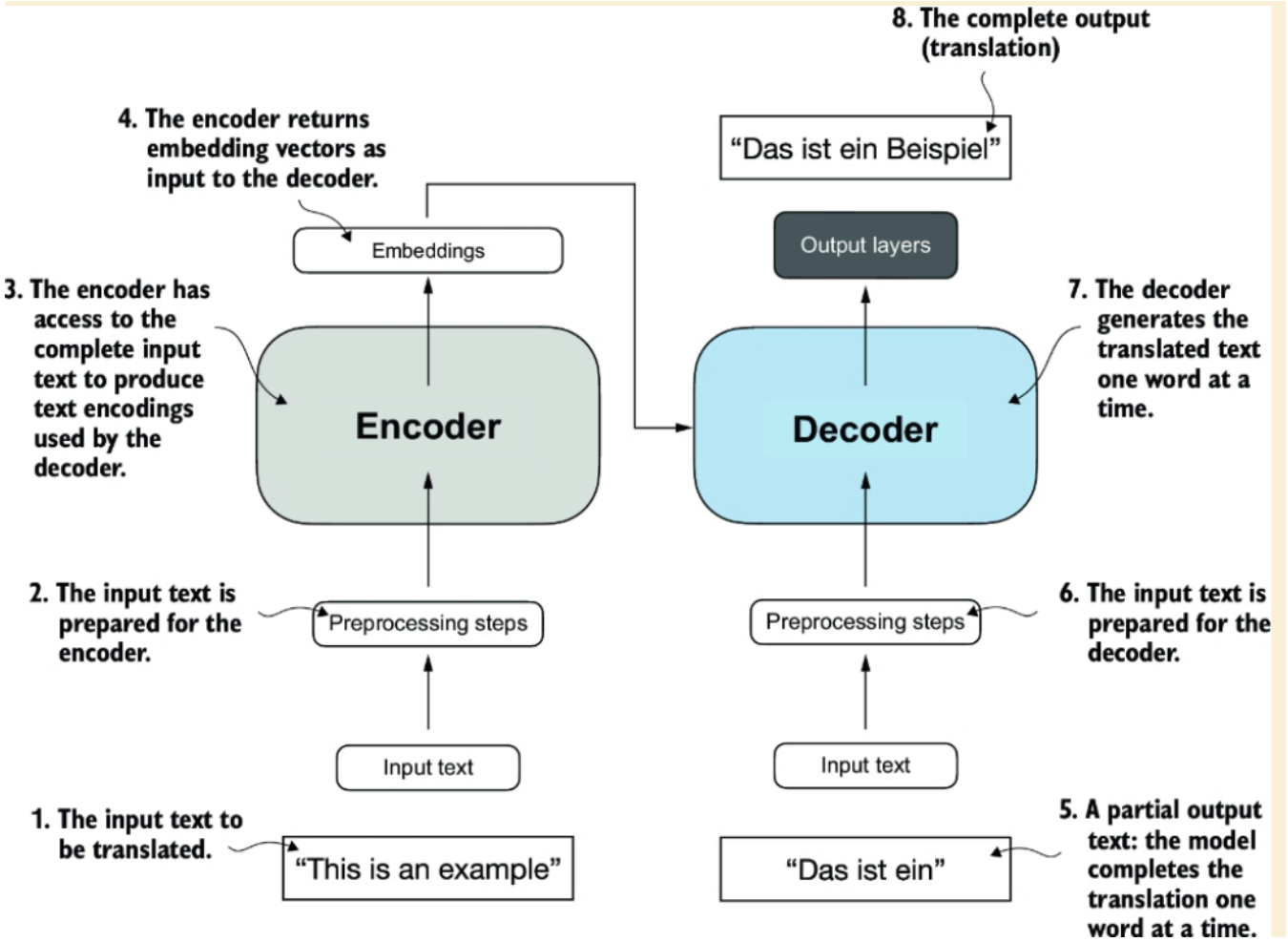


Figure 1.3 Pretraining an LLM involves next-word prediction on large text datasets. A pretrained LLM can then be fine-tuned using a smaller labeled dataset.

Riiiiight lets get this started!
Reference Github Project

Most LLMs rely on Transformers architecture for Deep Learning

Transformer Architecture



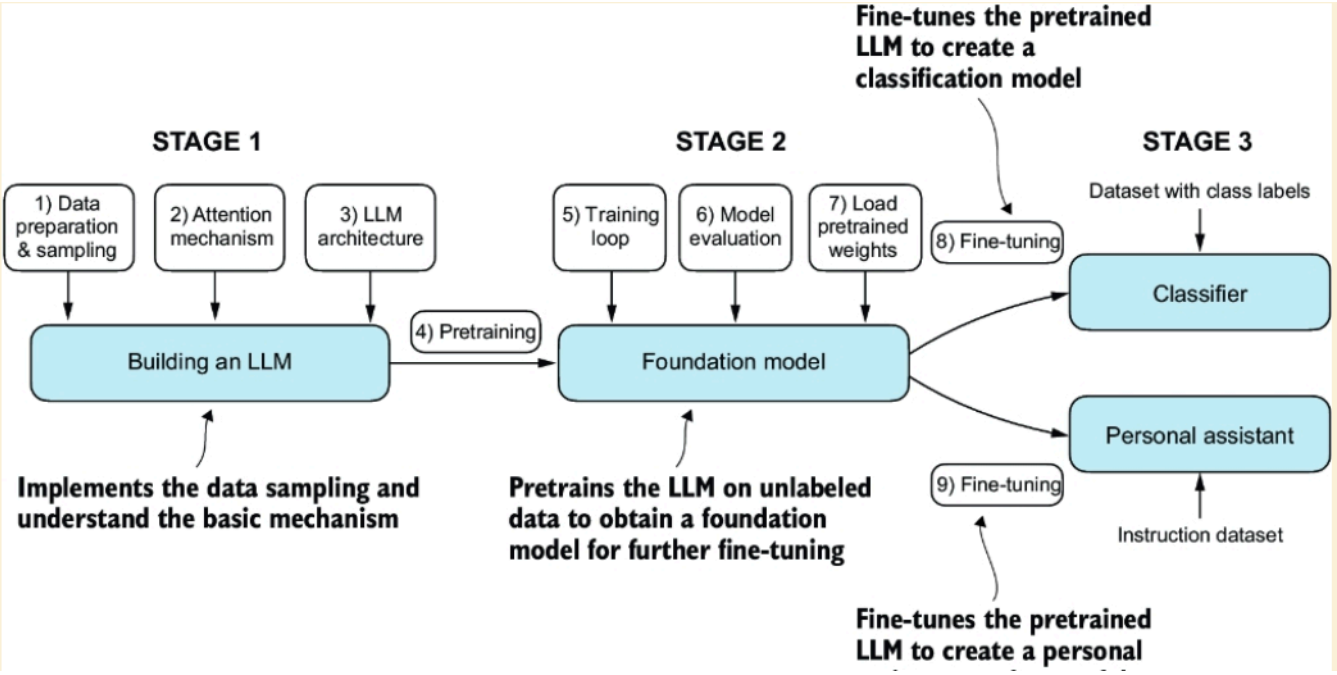
Possible Language Dataset: ~~𝕏~~ Dolma, a open source corpus of Three Trillion tokens for LLM pretraining

Pretrained LLMs can be used to generate text not present in their training data - we will use that

From 📚 Build a Large Language Model - Sebastian Raschka

> GPT tools, like ChatGPT, are called autoregressive models - incorporating their previous outputs as inputs for future predictions. Each new word is chosen based on the sequence that precedes it, which improves the coherence of the resulting text

Let's build! - Expected Work here



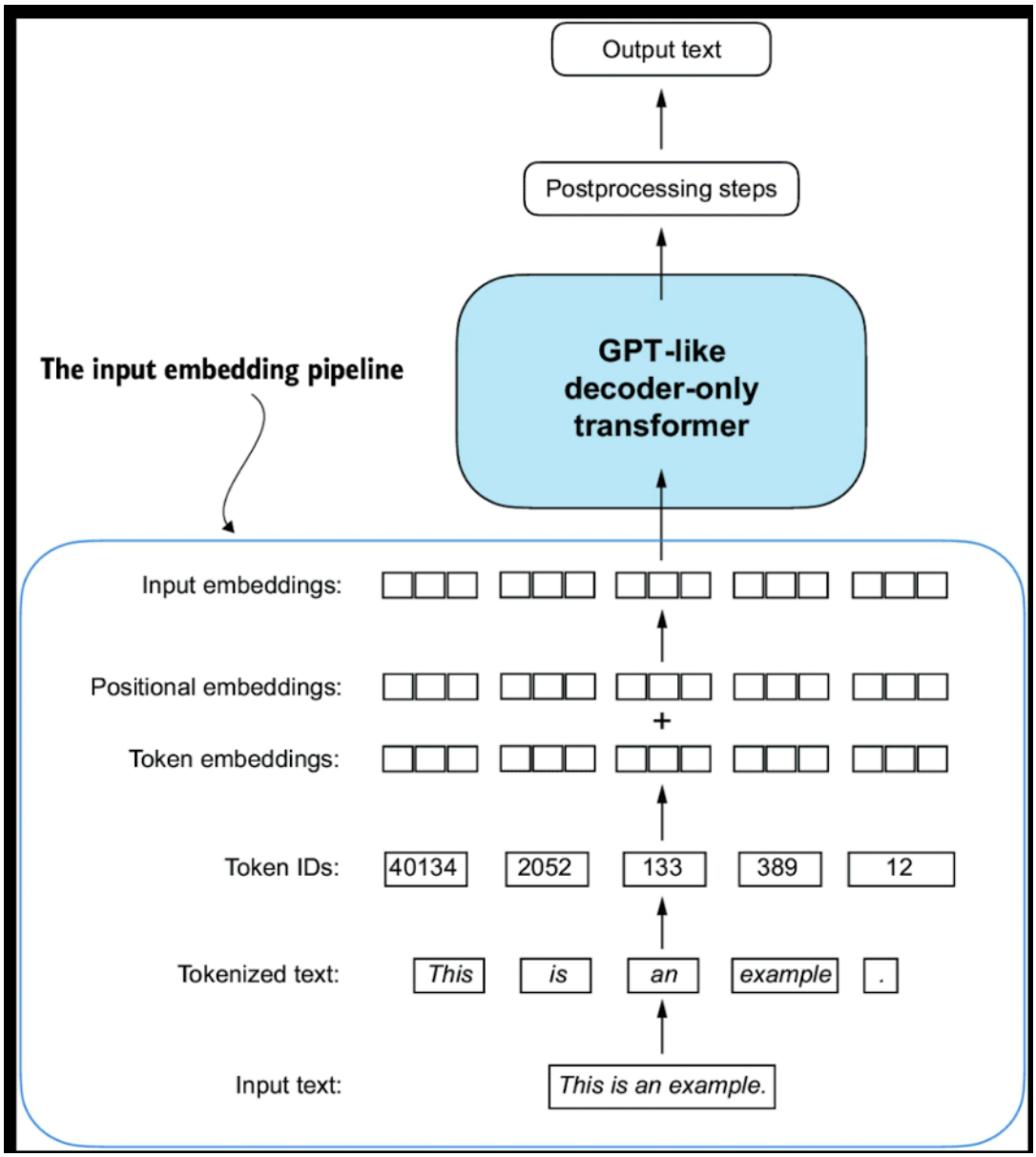
Tokenizing and Embedding

_The input Processing pipeline

The way of split input text into individual tokens, a required preprocessing step for creating embedding for an LLM. It can be words, punctuation, special characters, etc

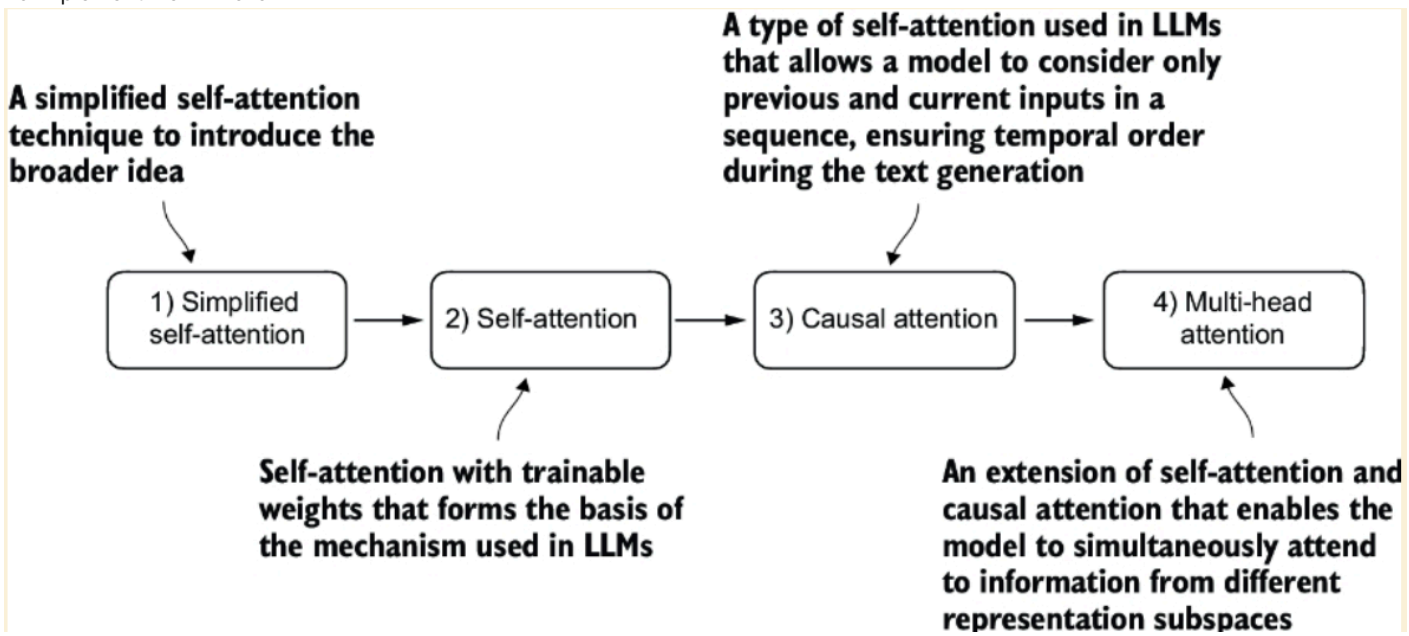
First step is to determine the tokens and create a dictionary of IDs (the way to train the model with categorical data - text) - one option is using Byte Pair Encoding (BPE), used on early versions of ChatGPT

Generating embeddings in order to have information for training the Artificial Neural Networks



Next step:

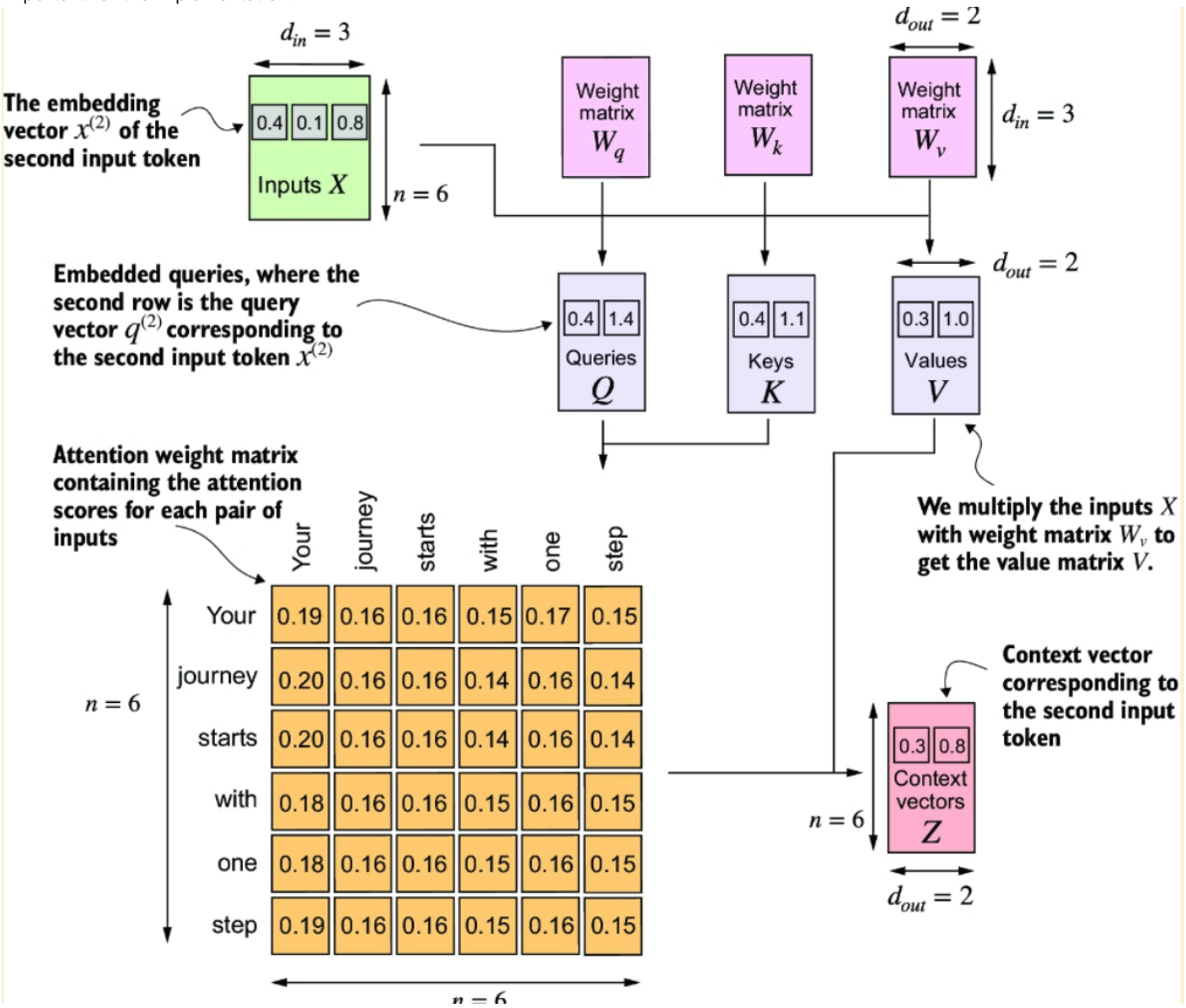
To implement we will follow



Attention Mechanism

Attention Mechanism is used to account for the context and influence of each word based on the relationship with the other words.

Important for the implementation:



Simpler explanation in code

PYTHON

```python
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))

    def forward(self, x):
        keys = x @ self.W_key
        queries = x @ self.W_query
        values = x @ self.W_value
        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1] ** 0.5, dim=-1
        )
        context_vec = attn_weights @ values
        return context_vec
```

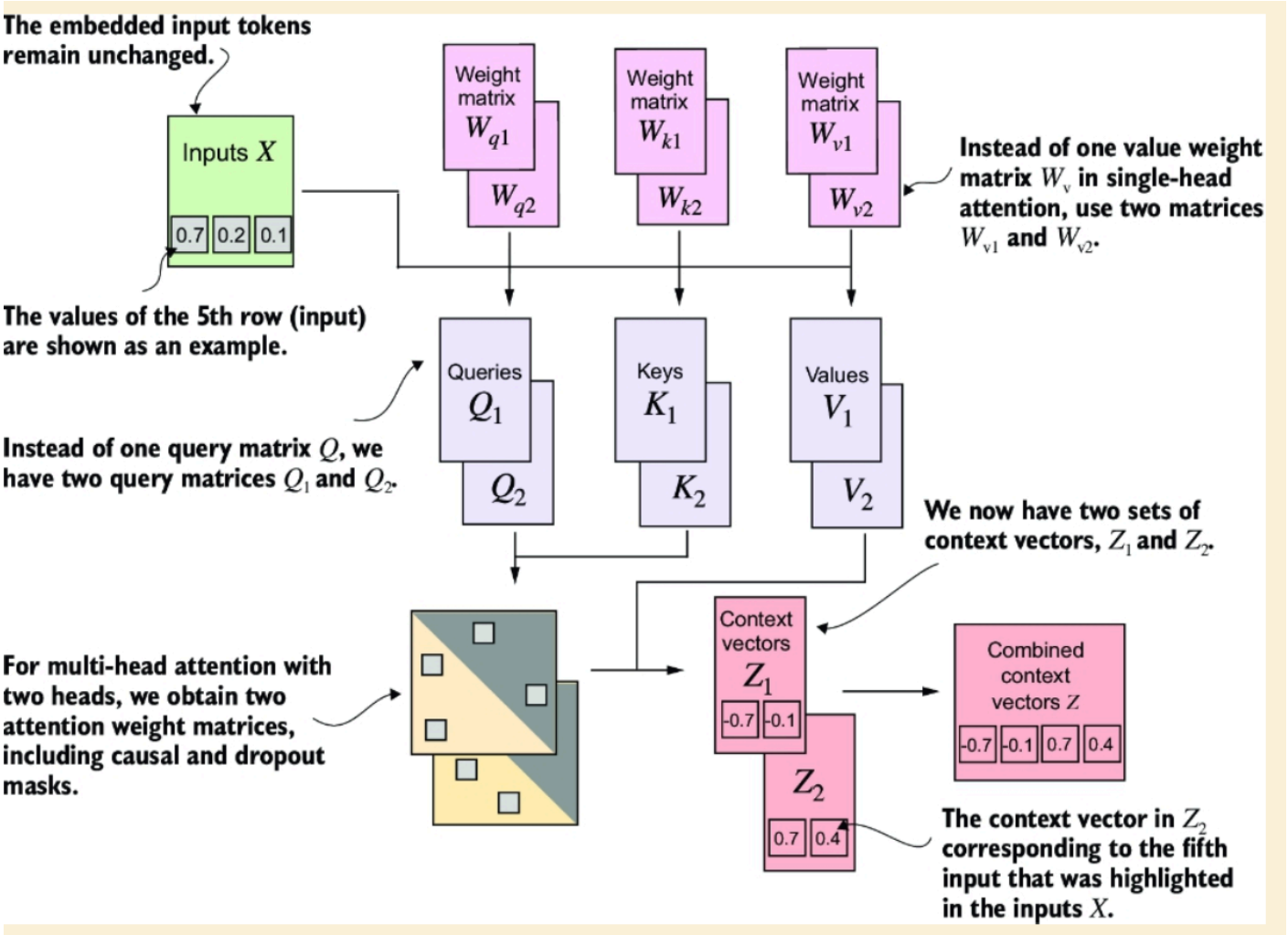
In order to train the ML we need to give it only the previous tokens from the part we wan to predict, reduciong the possibility of overfitting. This is also called *Masked Attention*

This means that for each token we mask out the future tokens, only maintaining the previous tokens. The process is:

- calculate the attention scores. - softmax
- apply a diagonal mask (tril) and multiply the attn scores
- renormalize the attention score based on the new diagonalized data

Them we go to implement the *Multihead attention mechanism*. The idea is that a single Casual Attention Module is an specific 'head' of attention. This is important aspect of this work, and it is what allowed for chatGPT to work; almost infinite scalability

The idea is to stack heads on top of each other



GPT Model

We will create a model architecture inspired on GPT2

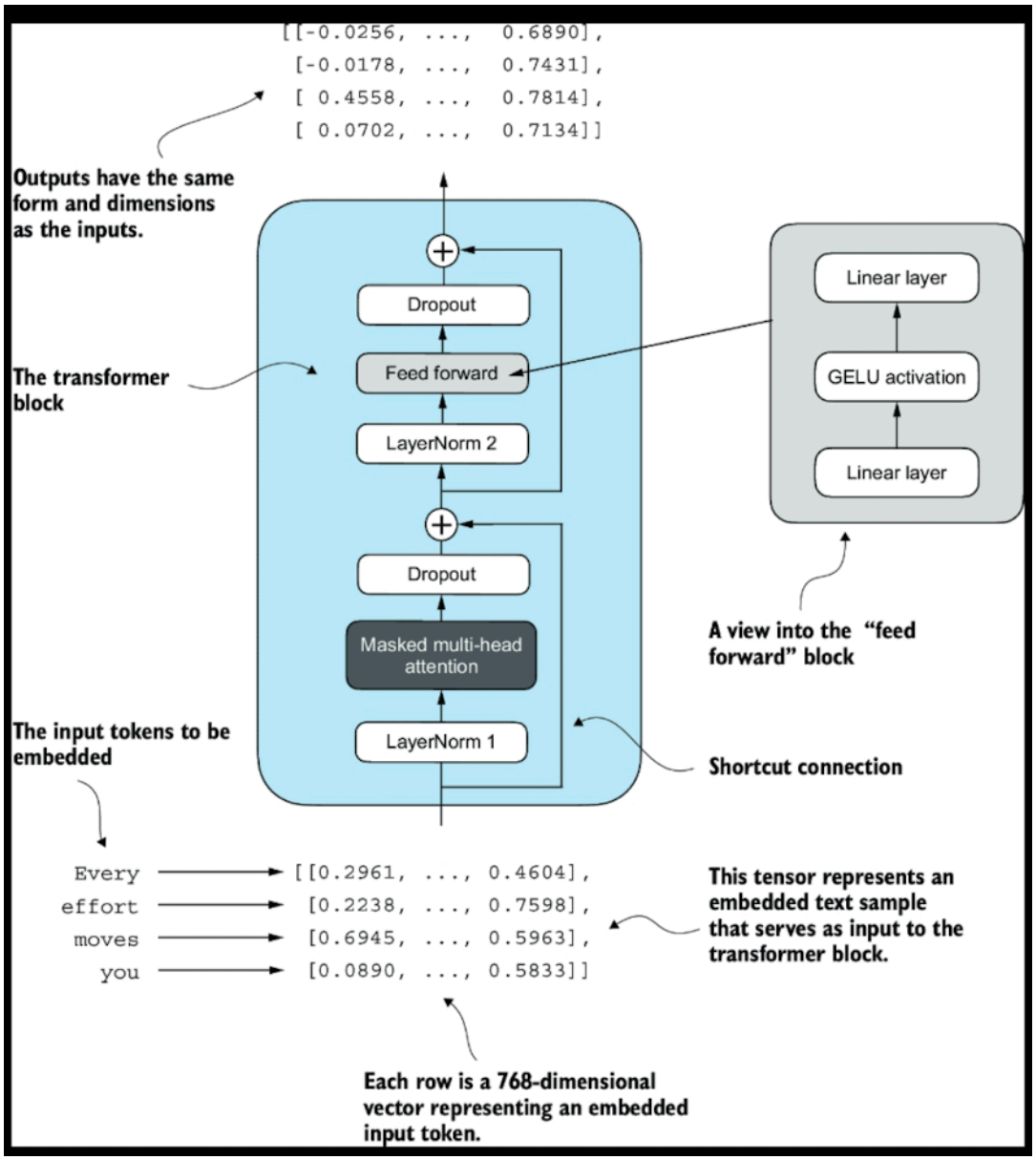
We can understand a Neural Network as a matrix connected to an activation function for each node

Generate a normalization layer.

A necessary asset to understand how facts is stored on a LLM: 3Blue1Brown – How might LLMs store facts DL7

The process for each token is to align their representation to an specific function that encodes the ifnormation for when one of the neuron is activate, resulting in the information being stored in the network

Integrating the transformer arch with the rest (embeddings, attention, etc), creating the GPT Model Architecture



The result is a matrix

Text Generation

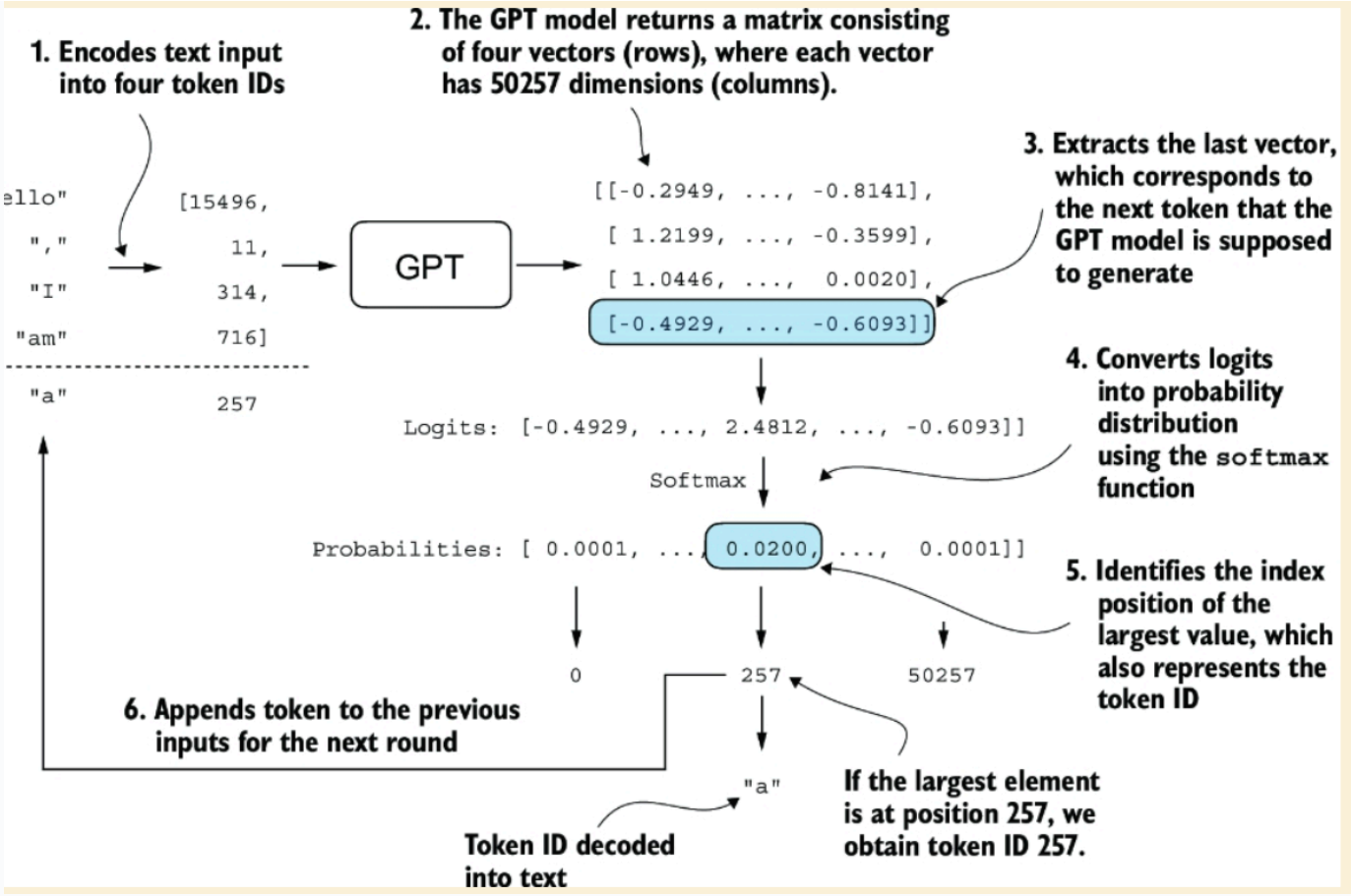
After implementing it, we go for Text Generation:

Steps from phrase as input:

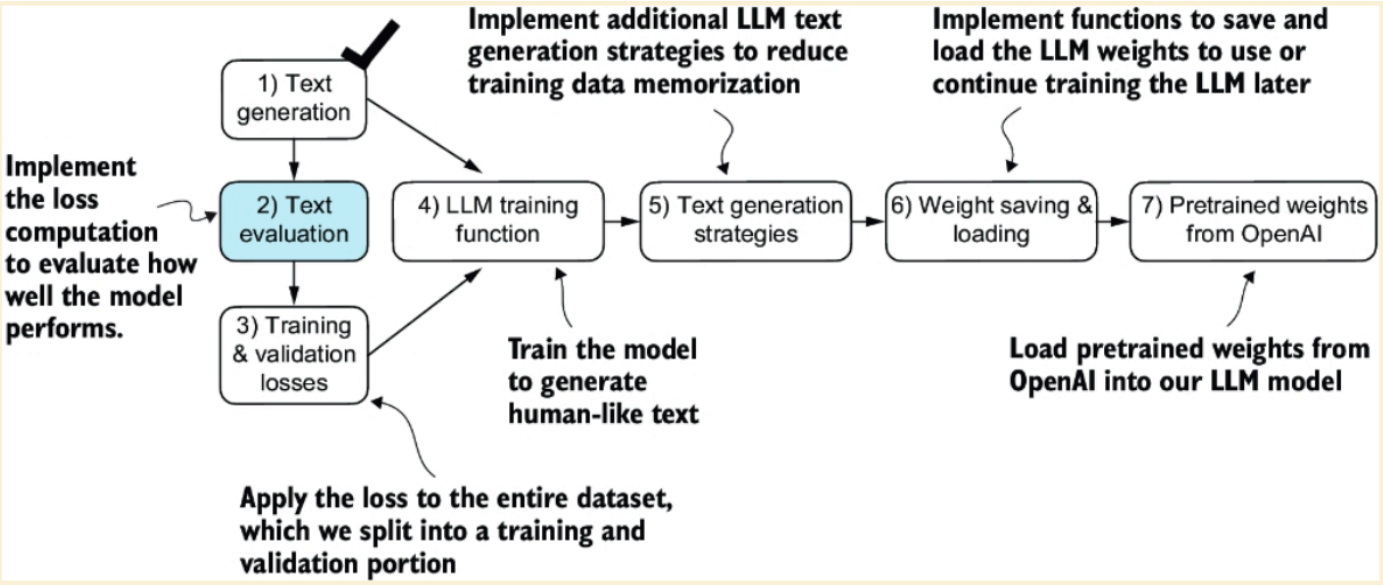
- tokenize input
- generate embeddings and positional
- Transformers block (GPT)
- receives a matrix and get last vector
- converts logits (vectors) to prob distribution using softmax
- highest item is the tokenId to get word
- Bone

Text Generation on a LLM

from; 📚 Build a Large Language Model - Sebastian Raschka
The mechanics of text generation in a GPT model by showing a single iteration in the token generation process. The process begins by encoding the input text into token IDs, which are then fed into the GPT model. The outputs of the model are then converted back into text and appended to the original input text.



Train the model



Calculate the quality of text generation first

- we do that by finding the probability distribution for the generation loss
- We measure loss by calculatiing how "far" the generated tokens are from the correct predictions.
- WE train the model to increase the probability that the softmax funcion will return the expected tokenIds
- We can use the Cross Entropy Loss, as a popular measurement in Machine Learning that tells us the difference between two probability distributions.
- Another measure is Perplexity, measuring how well a prob distribution using the model matches the actual distribution of the words in the dataset. *"How unsure the model is to predict the next token"*

Divide Training and Validation Data Sets

Actions, split data into trainign and validation datasets .

- Tokenize the datasets
- Divide datasets into "chunks" os user specified sizes
- shuffle the rows and organize batches
- this will be fed to the training process

Actually Training the model

From book:

> A typical training loop for training deep neural networks in PyTorch consists of numerous steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update the model weights so that the training set loss is minimized.

2025-04-22 - It worked!

It generated text, although still overfitting giving the dataset size it was trained on. But no worries... we will look into way to prevent that

Decoding Strategies (text generation)

Model Memorization is a problem, meaning that the model is not generating properly the content. This means that if you use the same input, it will always generate the same output, given the fact that the model was trained using a small data sample

Strategies to improve how we generate text are *Top-k sampling* and *temperature scaling*, that add variety to the generation process

- temperature scaling
  - adds probabilistic selection to next token, reducing the greedy decoding (generate text used argmax and always returns the highest prob)
- top-k sampling
  - choose the *k* first elements on the prob distribution and attribute everything else -inf to be zero on the softmax

Saving and loading model

Done. Saving and loading is easy.
can be seen here:

- https://github.com/victorhg/llm-from-scratch/blob/main/ch05-model-save-load.ipynb

Fine Tuning for Classification

Fine tuning is a way to specify the model we pre trained to execute target task, as classification - like finding spam, non spam.

Two main options for fine tuning, all related to the Classification Problems:

Instruction Fine-tuning

- broader range of tasks when compared with classification
- can be used for models that need more flexibility
- harder to train, demands larger resources (computational and data)

Classification Fine-tuning

- constrained into predicting classes it has encautered during training (spam)
- cannot say anything else about the text
- generally easier to develop (given its restriction nature)
- can achieve great results on specific tasks

The stages of Fine Tuning

Stage 1 *preparing the dataset*:

- Download/find a dataset and formating the data
- Batching the dataset

- Create Dataloaders

Stage 2 *fine-tuning the llm*:

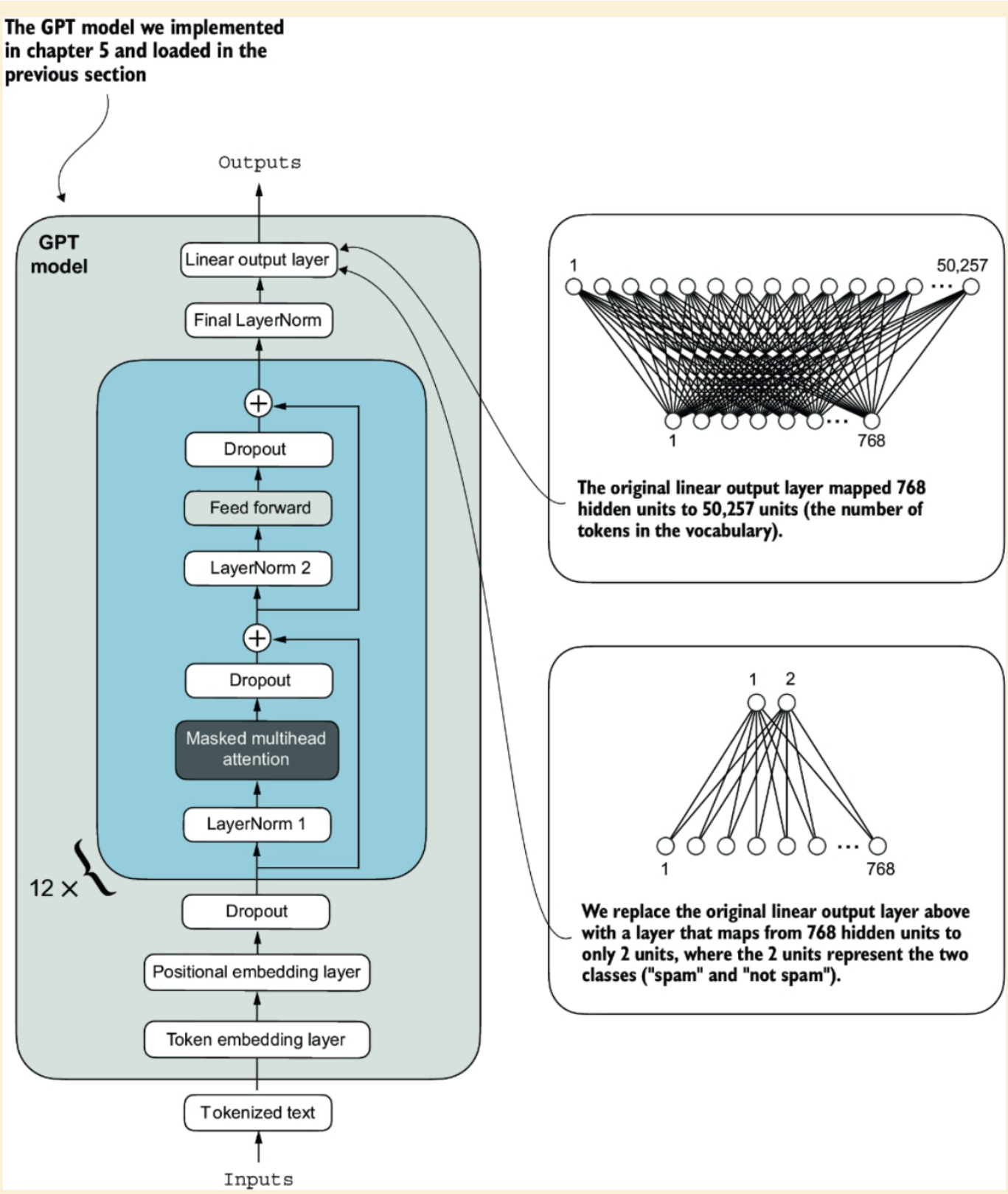
- Load pretrained LLM
- Execute the fine tuning Instruct or Classification
- Inspect the modeling loss

Stage 3 *evaluation*:

- Extracting responses
- Qualitative evaluation
- Scoring responses

Adding a classification Head

To add a classification capacity on our model, we need to map the output layer (originally with 50k Vocabulary) to only two classes for detecting spam *yes* or _no



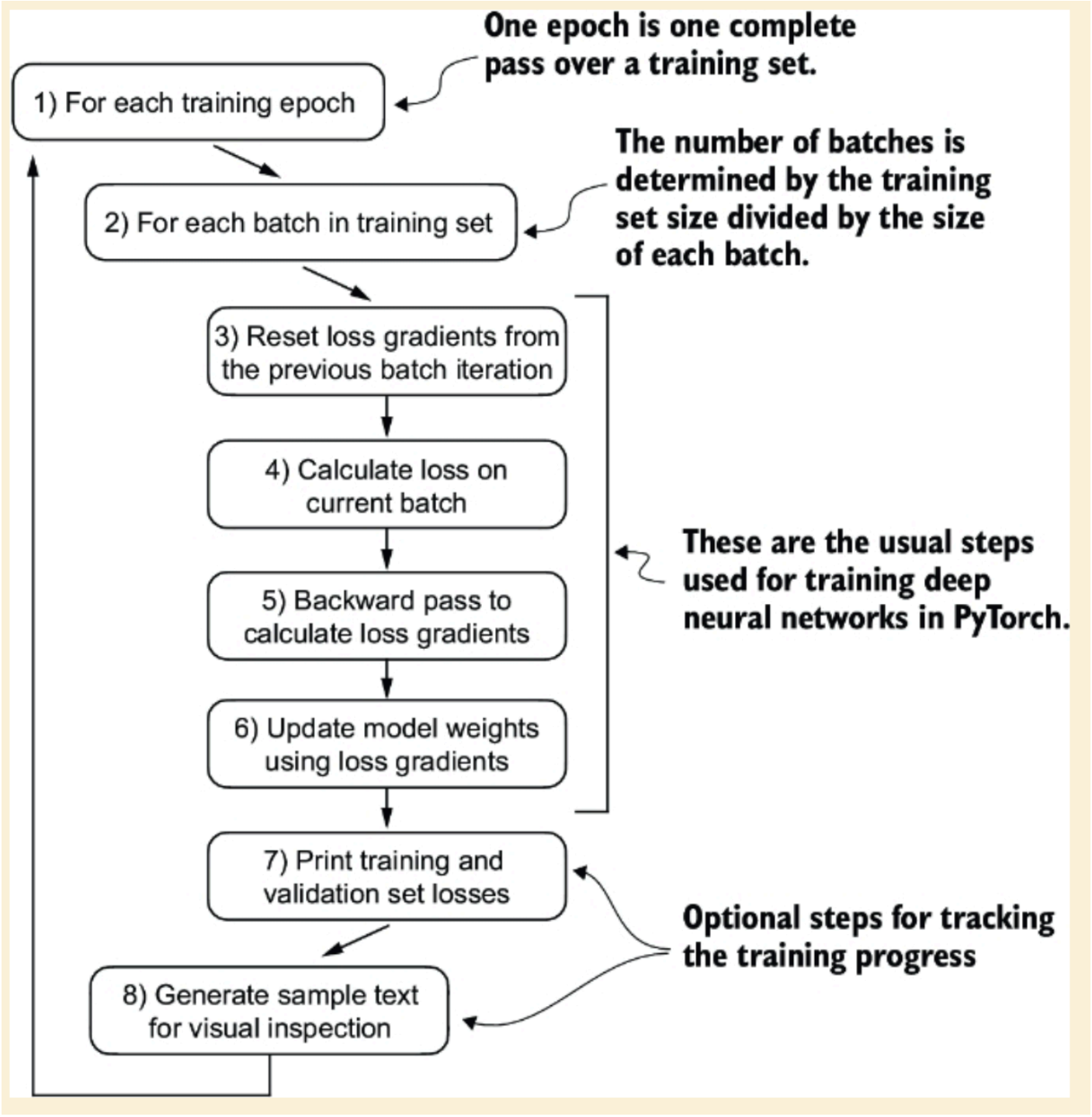
With a pre trained model we can only fine tune only selected layers - given the first few stages already defined language structures to analyze text, and we only need to change the output layer

This changes the funcioning of our model. Instead of returning a logit representation with 50k dimensions, it will return a 2 dimension array.

We create utility function to determine accuracy and loss of our prediction:

- accuracy is the proportion of correct predictions done by our model for multiple batches

- loss is the measure of missed predictions during definition



A typical training loop for training deep neural networks in PyTorch consists of several steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update the model weights to minimize the training set loss.

Fine Tuning for Instructions

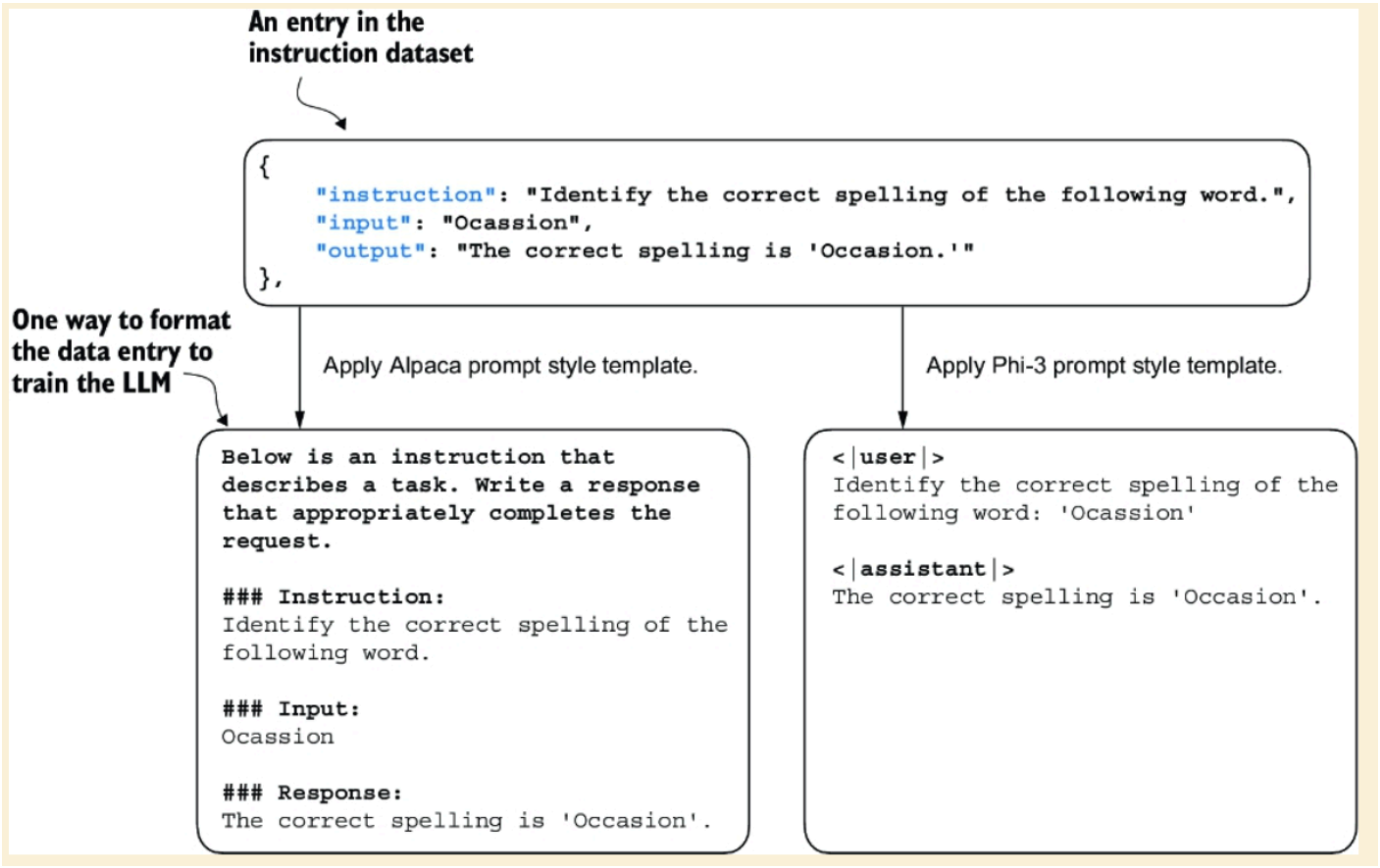
LLMs are created for text completion, and usually struggle with instructions, and they need to be fine tuned to understand and execute commands.

The dataset need to have instructions determining what to do, so that the LLM can proceed

- *Instruction*
  - "Convert 45 km into meters"
- *Goal Response*

    - "45 km is 4500 meters"
- *Instruction*
    - "Edit the following sentence to remove passive voice: 'The song was composed by the artist'"
- *Goal Response*
    - "The Artist composed the song"

Dataset Preparation for instruction, Alpaca and Phi-3 styles



In order to first create the instruction batches we need a different process:

Batching process for instruction fine tuning (different process):

- Apply the prompt template
- Tokenize text
- Padding tokens for normalization
- Create target token IDs
- Replacing -100 tokens to mask padding in the loss function

Applying the prompt template:

The template is determined as a text organization that simulates the followin

```
Bellow is an instruction that describes a task. Write a response that appropriety completes the task

### Instructuion:
Description of what the goal is set to do

### Input:
Input information to be evaluated

### Response:
The correct expected answer for the instruction
```

We run all the fine tuning process again and tested the results using ollama!