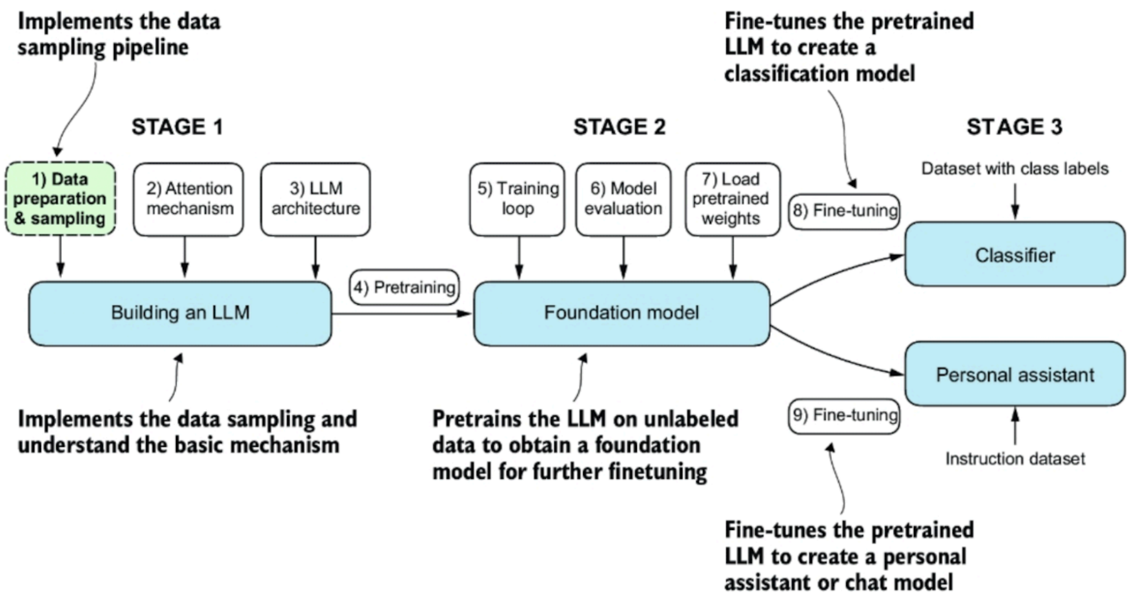
Ch02-Working With Text Data

This chapter has notes from book (Raschka, Sebastian. Build a Large Language Model (From Scratch) (Function). Kindle Edition.) and code will go together :

Video Reference :https://www.youtube.com/watch?v=341Rb8fJxY0&list=PLTKMiZHVd_2IIEsoJrWACkIxLRdfMIw11&index=6

LLMs are pretrained on large amount data. In this chapter, we will do coding for the step1 of the stage 1 i.e **Data preparation and Sampling**


Implements the data sampling pipeline
STAGE 1
1) Data preparation & sampling
2) Attention mechanism
3) LLM architecture
Building an LLM
4) Pretraining
Implements the data sampling and understand the basic mechanism
STAGE 2
5) Training loop
6) Model evaluation
7) Load pretrained weights
Foundation model
Pretrains the LLM on unlabeled data to obtain a foundation model for further finetuning
Fine-tunes the pretrained LLM to create a classification model
8) Fine-tuning
STAGE 3
Dataset with class labels
Classifier
Personal assistant
9) Fine-tuning
Instruction dataset
Fine-tunes the pretrained LLM to create a personal assistant or chat model

We will learn in this chapter about how we can prepare the text data for LLMs. involves splitting text into the individual words and subwords tokens, which can be encoded into the vector representations for the LLMs.

We will also learn:

- Advance tokenisation scheme byte pair encoding.
- Samplings and data loading strategies.

2.1 Understanding the Word Embedding

- Deep neural networks can not process the raw text data as it categorical. Deep Neural networks need numbers so that we can perform the mathematical operations on it.

- Therefore we need to represent the words into the **continous-valued vectors**.

💡 The concept of converting data into the vectors is often referred as ***embeddings***.

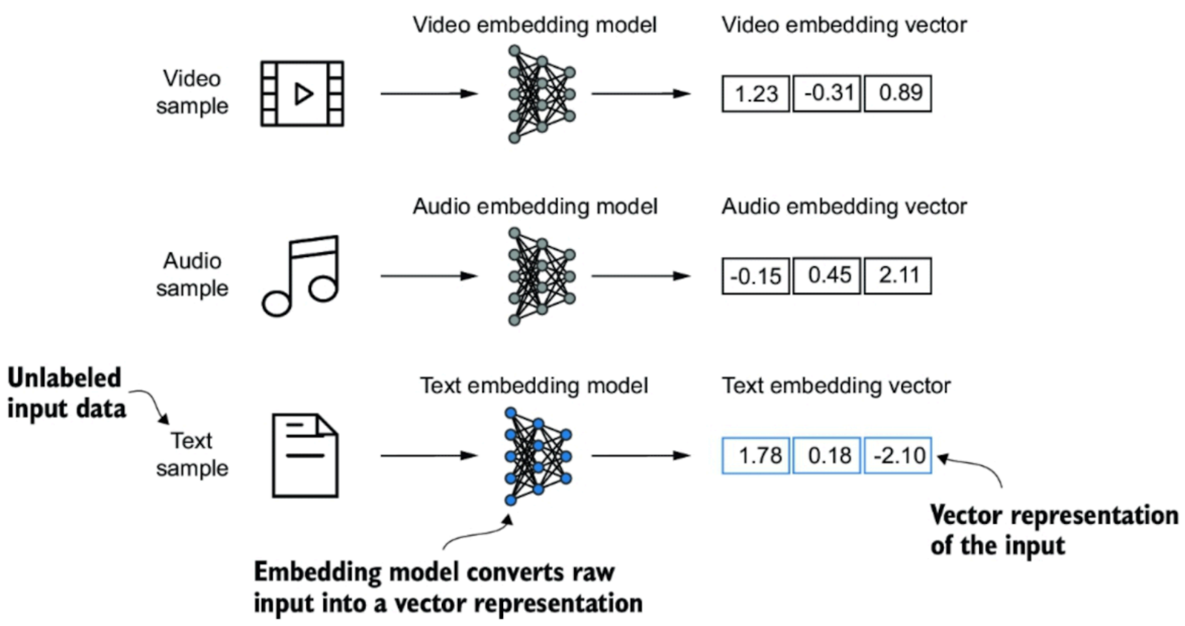


Figure 2.2 Deep learning models cannot process data formats like video, audio, and text in their raw form. Thus, we use an embedding model to transform this raw data into a dense vector representation that deep learning architectures can easily understand and process. Specifically, this figure illustrates the process of converting raw data into a three-dimensional numerical vector.

💡 Different data formats require distinct embedding models. For example, an embedding model designed for text would not be suitable for embedding audio or video data.

- Word embedding are most common embeddings. Sentence or paragraph embeddings are popular choices for *retrieval augmented generation*.

> Retrieval-augmented generation combines generation (like producing text) with retrieval (like searching an external knowledge base) to pull relevant information when generating text.

We will focus on word embeddings now.

Several algorithms and frameworks are available for word embedding. One of earlier and most popular approach is *Word2Vec*.

Word2Vec :

Word2Vec trained neural network architecture to generate word embeddings by predicting the context of a word given the target word or vice versa. The main idea behind Word2Vec is that words that appear in similar contexts tend to have similar meanings. Consequently, when projected into two-dimensional word embeddings for visualization purposes, similar terms are clustered together, as shown in figure 2.3.

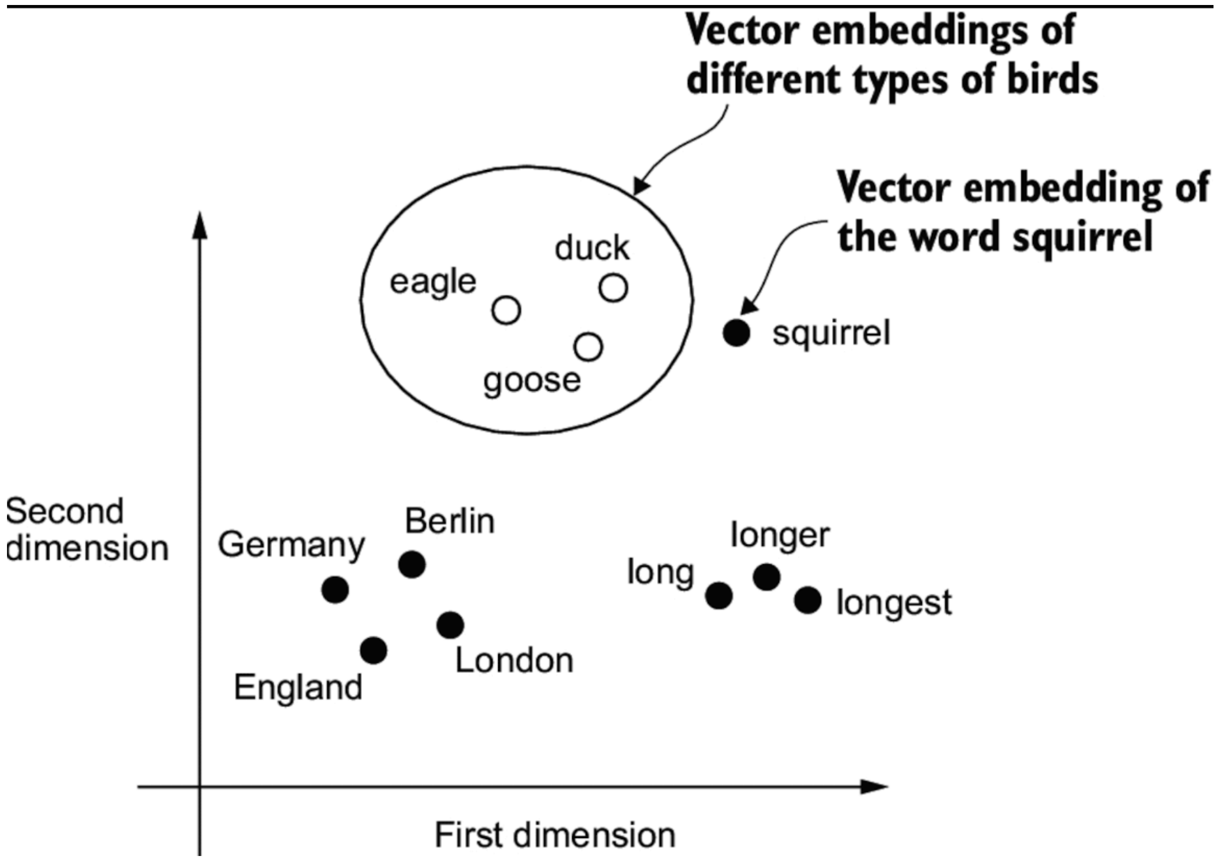


Figure 2.3 If word embeddings are two-dimensional, we can plot them in a two-dimensional scatterplot for visualization purposes as shown here. When using

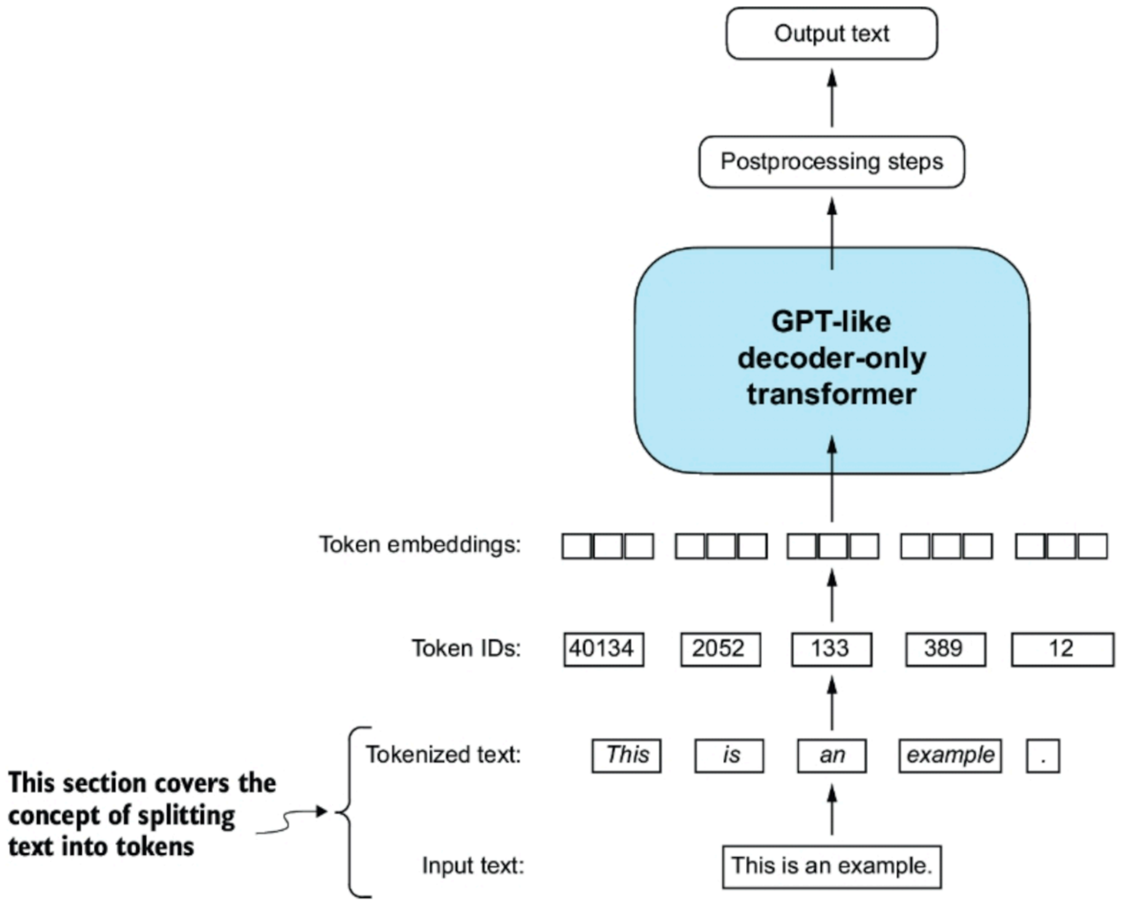
word embedding techniques, such as Word2Vec, words corresponding to similar concepts often appear close to each other in the embedding space. For instance, different types of birds appear closer to each other in the embedding space than in countries and cities.

💡 Word embeddings can have varying dimensions, from one to thousands. A higher dimensionality might capture more nuanced relationships but at the cost of computational efficiency.

- We can use pretrained models such as Word2Vec to generate embeddings for machine learning models, LLMs commonly produce their own embeddings that are part of the input layer and are updated during training.
- The advantages of optimizing the embeddings as a part of LLMs training is instead of of using the Word2Vec is embeddings are optimized for particular tasks and data at hand. We will implement such embedding layers in this chapter later.
- In GPT-2 and GPT-3, the embedding size often referred as dimensionality of model hidden states.
- The smallest **GPT-2 models** (117M and 125M parameters) use an embedding size of **768 dimensions** to provide concrete examples. The **largest GPT-3 model** (175B parameters) uses an embedding size of **12,288 dimensions**.

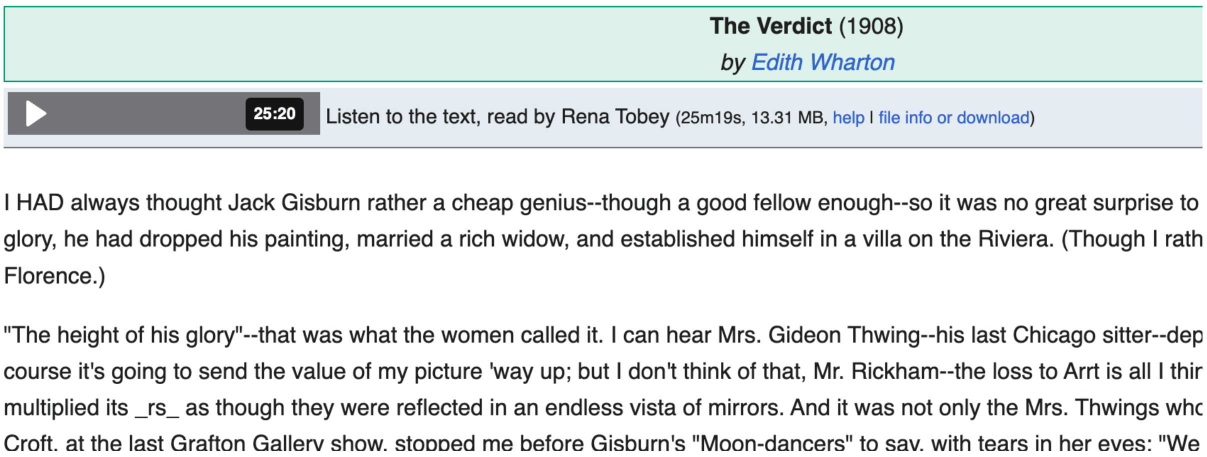
Now we will discuss : **Text → Word → token → embeddings**

2.2 Tokenizing the text



The text we will tokenize for LLM training is “**The Verdict**,” a short story by Edith Wharton, which has been released into the public domain and is thus permitted to be used for LLM training tasks. The text is available on Wikisource at https://en.wikisource.org/wiki/The_Verdict,

- Copy paste the text from the above link into : "the-verdict.txt" or directly download from book github: https://mng.bz/Adng.



The Verdict (1908)
by Edith Wharton

25:20 Listen to the text, read by Rena Tobey (25m19s, 13.31 MB, help I file info or download)

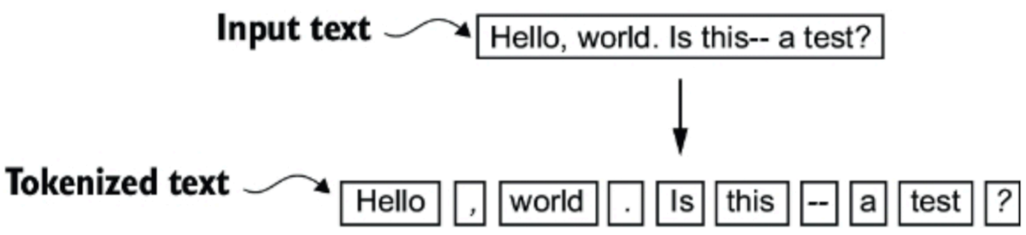
I HAD always thought Jack Gisburn rather a cheap genius--though a good fellow enough--so it was no great surprise to glory, he had dropped his painting, married a rich widow, and established himself in a villa on the Riviera. (Though I rath Florence.)

"The height of his glory"--that was what the women called it. I can hear Mrs. Gideon Thwing--his last Chicago sitter--dep course it's going to send the value of my picture 'way up; but I don't think of that, Mr. Rickham--the loss to Arrt is all I thir multiplied its _rs_ as though they were reflected in an endless vista of mirrors. And it was not only the Mrs. Thwings whc Croft, at the last Grafton Gallerv show, stopped me before Gisburn's "Moon-dancers" to sav, with tears in her eves: "We

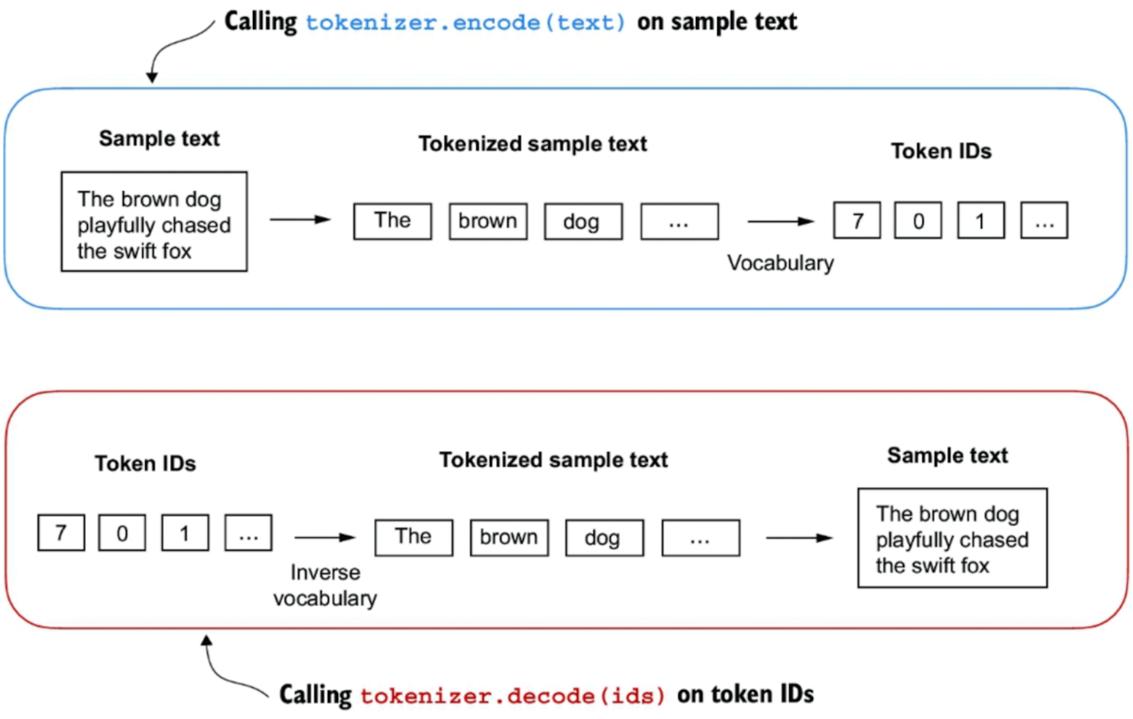
- A good app to visualize the tokenization :https://tiktokenizer.vercel.app/

Follow the code now Pyare-ch02.ipynb.

Notes : We did this using the regular expression (import re)



2.3 Converting the tokens into the token IDs.

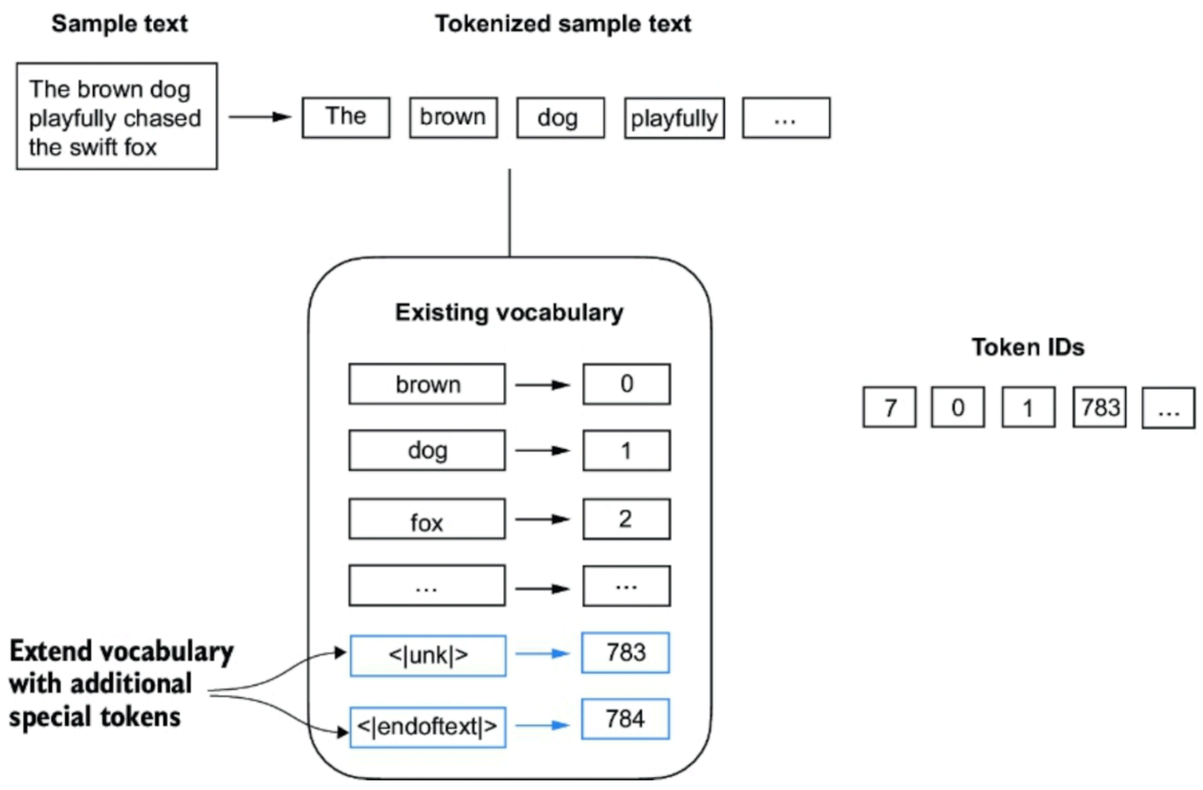

Calling tokenizer.encode(text) on sample text
Sample text
The brown dog playfully chased the swift fox
Tokenized sample text
The
brown
dog
...
Vocabulary
Token IDs
7
0
1
...
Token IDs
7
0
1
...
Inverse vocabulary
Tokenized sample text
The
brown
dog
...
Sample text
The brown dog playfully chased the swift fox
Calling tokenizer.decode(ids) on token IDs

```
[24]: text = "Hello! do you like green tea"
      print(tokenizer.encode(text))
```

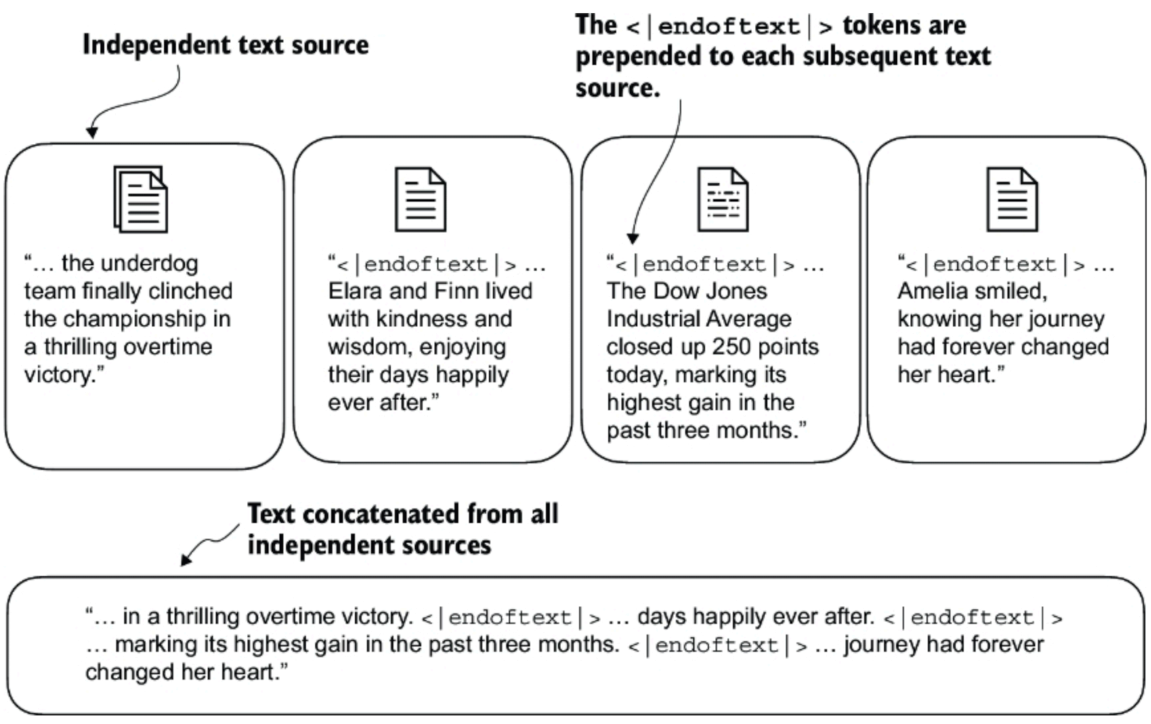
```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[24], line 2
      1 text = "Hello! do you like green tea"
----> 2 print(tokenizer.encode(text))

Cell In[18], line 12, in SimpleTokenizerV1.encode(self, text)
     10 # removing whitespaces
     11 preprocessed = [item.strip() for item in preprocessed if item.strip()]
---> 12 ids = [self.str_to_int[s] for s in preprocessed]
     13 return ids

Cell In[18], line 12, in <listcomp>(.0)
     10 # removing whitespaces
     11 preprocessed = [item.strip() for item in preprocessed if item.strip()]
---> 12 ids = [self.str_to_int[s] for s in preprocessed]
     13 return ids

KeyError: 'Hello'
```

The above error came due to the Hello word is not in the vocabulary. So this highlight the need of the large and diverse vocabulary when dealing with the LLMs.

2.4 Adding Special context tokens



💡 We need to modify the tokenizer to handle the unknown words. we will modify the existing vocabulary and add 2 new tokens as `<|unk|>` : for unknown words and $< |endoftext| >$


Independent text source
The <|endoftext|> tokens are prepended to each subsequent text source.
"… the underdog team finally clinched the championship in a thrilling overtime victory."
"<|endoftext|> … Elara and Finn lived with kindness and wisdom, enjoying their days happily ever after."
"<|endoftext|> … The Dow Jones Industrial Average closed up 250 points today, marking its highest gain in the past three months."
"<|endoftext|> … Amelia smiled, knowing her journey had forever changed her heart."
Text concatenated from all independent sources
"… in a thrilling overtime victory. <|endoftext|> … days happily ever after. <|endoftext|> … marking its highest gain in the past three months. <|endoftext|> … journey had forever changed her heart."

When working with multiple independent text source, we add <|endoftext|> tokens between these texts. These <|endoftext|> tokens act as markers, signaling the start or end of a particular segment, allowing for more effective processing and understanding by the LLM.

We will make now `SimpleTokenizerV2`

Next: → Follow code section 2.4

LLM researcher also use some of other special tokens as :

- [BOS] (beginning of sequence) —This token marks the start of a text. It signifies to the LLM where a piece of content begins.
- [EOS] (end of sequence) —This token is positioned at the end of a text and is especially useful when concatenating multiple unrelated texts, similar to <|endoftext|>. For instance, when combining two different Wikipedia articles or books, the [EOS] token indicates where one ends and the next begins.
- [PAD] (padding) —When training LLMs with batch sizes larger than one, the batch might contain texts of varying lengths. To ensure all texts have the same length, the shorter texts are extended or "padded" using the [PAD] token, up to the length of the longest text in the batch.

ChatGPT donnot use these special tokens except <|endoftext|>

- Moreover, the tokenizer used for GPT models also doesn't use an <|unk|> token for out-of-vocabulary words. Instead, GPT models use a byte pair encoding tokenizer, which breaks words down into subword units, which we will discuss next.

2.5 Byte Pair Encoding

From scratch you can use : https://github.com/rasbt/LLMs-from-scratch/blob/main/ch02/05_bpe-from-scratch/bpe-from-scratch.ipynb

But this optional. Look at it if requied, For now focus on use.

We will use python open source library called tiktoken : (https://github.com/openai/tiktoken), It is efficient as core implemented in rust and later wrapped in python.

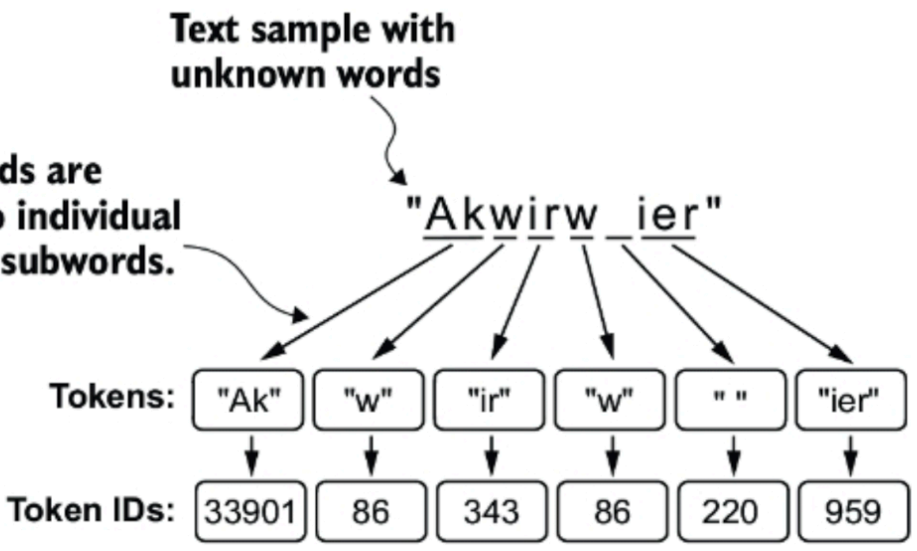
This encoding was used in ChatGPT, GPT-2 and GPT-3

💡 ChatGPT, has a total vocabulary size of 50,257, with <|endoftext|> being assigned the largest token ID i.e 50,256.

```
text = (
    "Hello, do you like tea? <|endoftext|> In the sunlit terraces"
     "of someunknownPlace."
)
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)

[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250,
 8812, 2114, 286, 617, 34680, 27271, 13]
```

```
# decoding
strings = tokenizer.decode(integers)
print(strings)
Hello, do you like tea? <|endoftext|> In the sunlit terracesof someunknownPla
```

BPE is able to handle the unknown words without using the token `<|unk|>` .It breaks down the unknown token into sub words unit or even in characters.


Text sample with unknown words
Unknown words are tokenized into individual characters or subwords.
"Akwirw ier"
Tokens:
"Ak"
"w"
"ir"
"w"
" "
"ier"
Token IDs:
33901
86
343
86
220
959

this ensure that LLM can process any data provided to it.

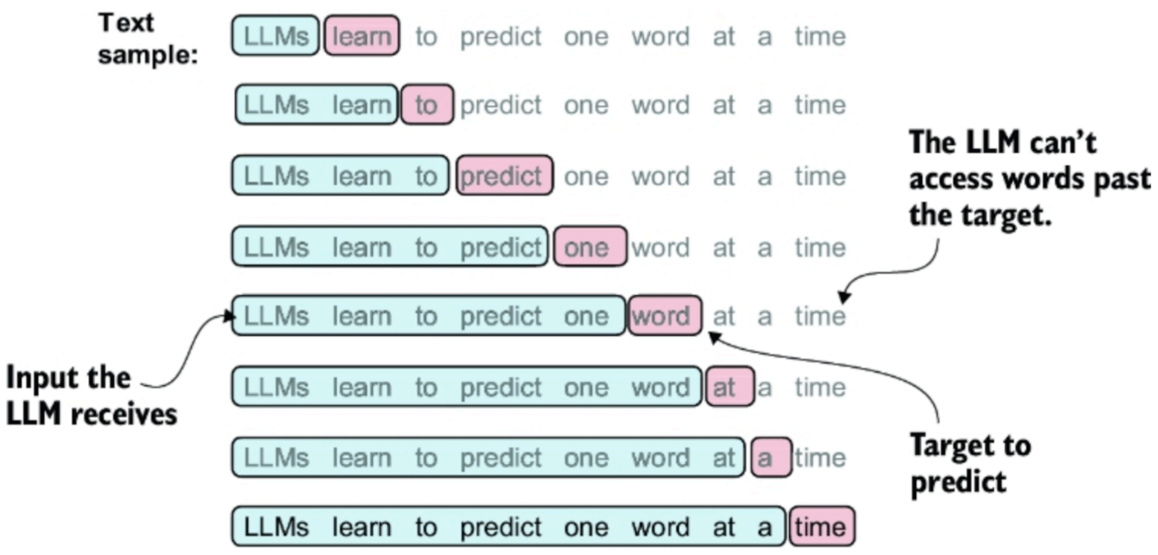
❓ **Exercise 2.1** Byte pair encoding of unknown words Try the BPE tokenizer from the tiktoken library on the unknown words “Akwirw ier” and print the individual token IDs. Then, call the decode function on each of the resulting integers in this list to reproduce the mapping shown in figure 2.11. Lastly, call the decode method on the token IDs to check whether it can reconstruct the original input, “Akwirw ier.”

```
tokenizer = tiktoken.get_encoding("gpt2")
text = ("Akwirw ier")
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
[33901, 86, 343, 86, 220, 959]
strings = tokenizer.decode(integers)
print(strings)
Akwirw ier
```

> ## BPE:

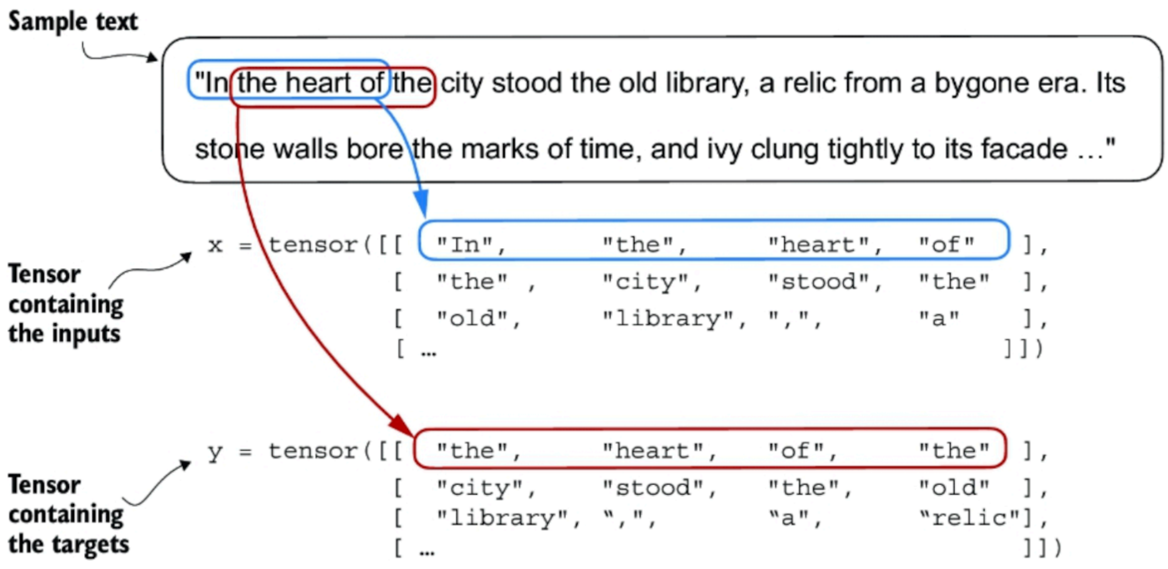
> It builds its vocabulary by iteratively merging frequent characters into subwords and frequent subwords into words. For example, BPE starts with adding all individual single characters to its vocabulary (“a,” “b,” etc.). In the next stage, it merges character combinations that frequently occur together into subwords. For example, “d” and “e” may be merged into the subword “de,” which is common in many English words like “define,” “depend,” “made,” and “hidden.” The merges are determined by a frequency cutoff.

2.6 Data Sampling with a sliding window


Text sample:
LLMs learn to predict one word at a time
The LLM can't access words past the target.
Input the LLM receives
Target to predict

Given a text sample, extract input blocks as subsamples that serve as input to the LLM, and the LLM’s prediction task during training is to predict the next word that follows the input block. During training, we mask out all words that are past the target. Note that the text shown in this figure must undergo tokenization before the LLM can process it; however, this figure omits the tokenization step for clarity.

Next step is efficient data loader , which convert text into the 2 tensor as input and target as shown below:



"In the heart of the city stood the old library, a relic from a bygone era. Its stone walls bore the marks of time, and ivy clung tightly to its facade …"

```
x = tensor([[ "In",       "the",      "heart",  "of"    ],
            [ "the" ,     "city",     "stood",  "the"   ],
            [ "old",      "library",  ",",      "a"     ],
            [ ...                                      ]])

y = tensor([[ "the",      "heart",    "of",     "the"   ],
            [ "city",     "stood",    "the",    "old"   ],
            [ "library",  ",",        "a",      "relic"],
            [ ...                                        ]])
```

To implement efficient data loaders, we collect the inputs in a tensor, x, where each row represents one input context. A second tensor, y, contains the corresponding prediction targets (next words), which are created by shifting the input by one position.

Apendix A Introduction to Pytorch (Basics which you require)

Introduction to Pytorch

next `pyare-ch02.ipynb`