Large Language Models

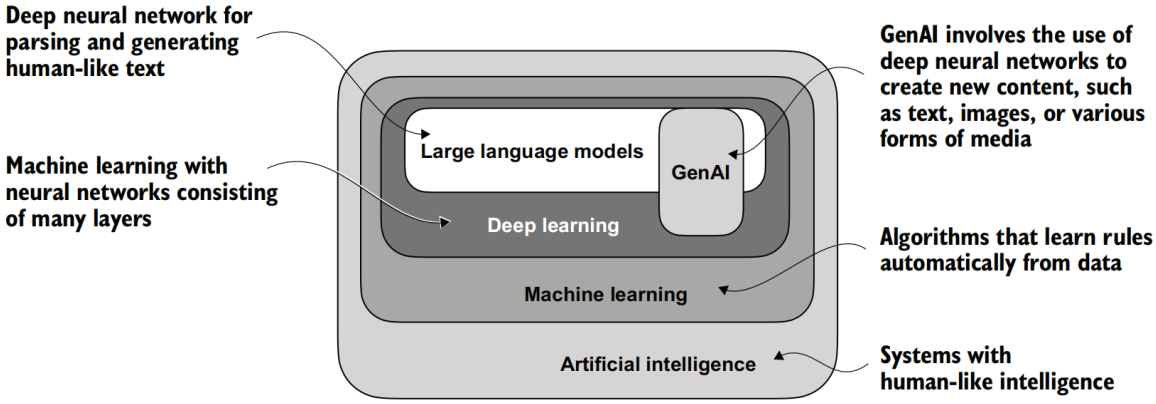
1. Understanding large language models

(1) want is llm?

An LLM is a neural network designed to understand, generate, and respond to human like text. These models are deep neural networks trained on massive amounts of text data, sometimes encompassing large portions of the entire publicly available text on the internet.

The “large” in “large language model” refers to both the model’s size in terms of parameters and the immense dataset on which it’s trained.

When we say language models “understand,” we mean that they can process and generate text in ways that appear coherent and contextually relevant,

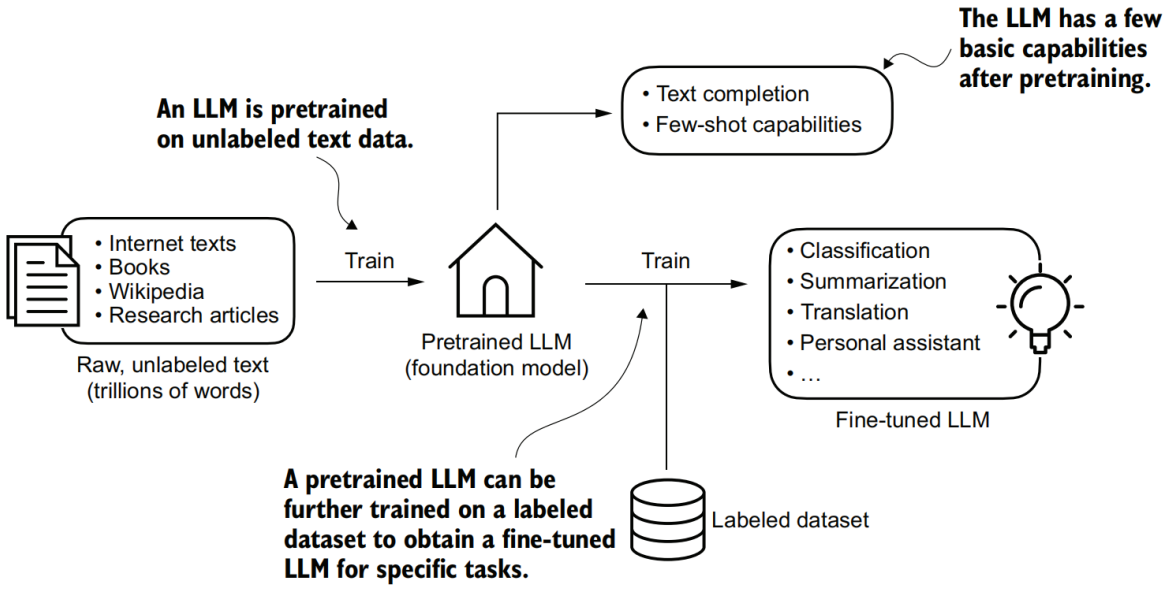


(2) Stages of building and using LLMs

The general process of creating an LLM includes pretraining and fine-tuning. The“pre” in “pretraining” refers to the initial phase where a model like an LLM is trained on a large, diverse dataset to develop a broad understanding of language.

In this phase, LLMs use self-supervised learning, where the model generates its own labels from the input data.

This pretrained model then serves as a foundational resource that can be further refined through fine-tuning, a process where the model is specifically trained on a narrower dataset that is more specific to particular tasks or domains.



(3) fine tuning

After obtaining a pretrained LLM from training on large text datasets, where the LLM is trained to predict the next word in the text, we can further train the LLM on labeled data, also known as *fine-tuning*.

The two most popular categories of fine-tuning LLMs are *instruction fine-tuning* and *classification fine-tuning*.

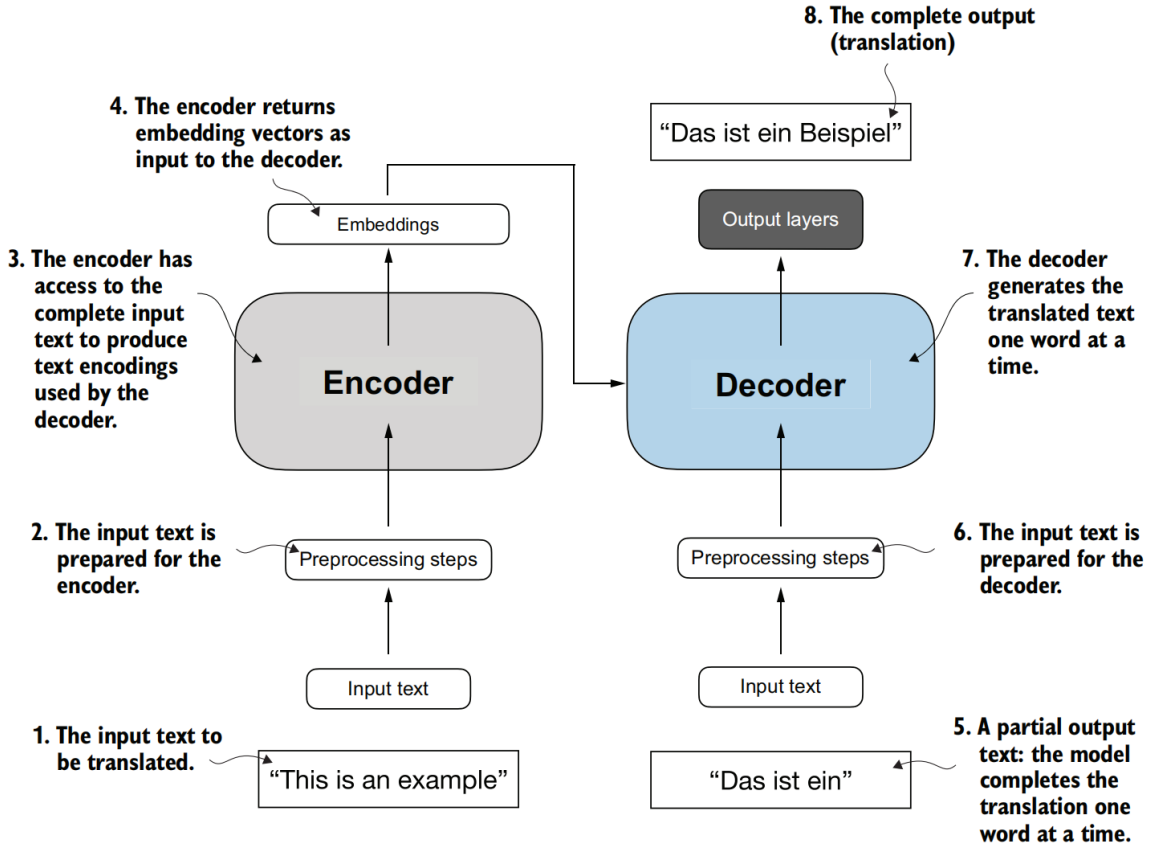
- In instruction fine-tuning, the labeled dataset consists of instruction and answer pairs, such as a query to translate a text accompanied by the correctly translated text.
- In classification fine-tuning, the labeled dataset consists of texts and associated class labels—for example, emails associated with "spam" and "not spam" labels.

(4) intro to transformer

Most modern LLMs rely on the *transformer* architecture, which is a deep neural network architecture introduced in the 2017 paper "Attention Is All You Need" (https://arxiv.org/abs/1706.03762).

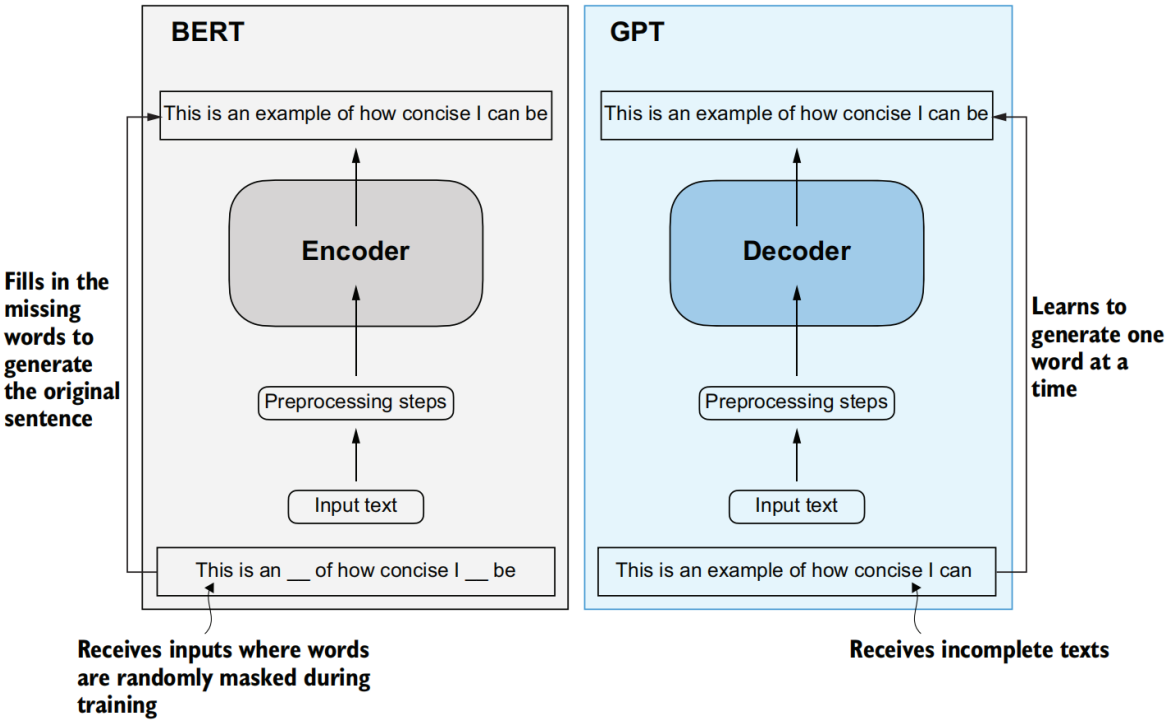
The transformer architecture consists of two submodules: an encoder and a decoder.

- The encoder module processes the input text and encodes it into a series of numerical representations or vectors that capture the contextual information of the input.
- Then, the decoder module takes these encoded vectors and generates the output text. In a translation task, for example, the encoder would encode the text from the source language into vectors, and the decoder would decode these vectors to generate text in the target language.
- Both the encoder and decoder consist of many layers connected by a so-called self-attention mechanism.



Later variants of the transformer architecture, such as BERT (short for **bidirectional encoder representations from transformers**) and the various GPT models (short for **generative pretrained transformers**), built on this concept to adapt this architecture for different tasks.

- BERT: built upon the original transformer's encoder submodule and specialize in masked word prediction, where the model predicts masked or hidden words in a given sentence.
- GPT: focuses on the decoder portion of the original transformer architecture and is designed for tasks that require generating texts.



zero-shot and few-shot learning

- Zero-shot learning refers to the ability to generalize to completely unseen tasks without any prior specific examples.
- Few-shot learning involves learning from a minimal number of examples the user provides as input

(5) large dataset

Token is a unit of text that a model reads and the number of tokens in a dataset is roughly equivalent to the number of words and punctuation characters in the text.

GPT-3 dataset details

The proportions column in the table sums up to 100% of the sampled data, adjusted for rounding errors. Although the subsets in the Number of Tokens column total 499 billion, the model was trained on only 300 billion tokens. The authors of the GPT-3 paper did not specify why the model was not trained on all 499 billion tokens.

For context, consider the size of the CommonCrawl dataset, which alone consists of 410 billion tokens and requires about 570 GB of storage. In comparison, later iterations of models like GPT-3, such as Meta's LLaMA, have expanded their training scope to include additional data sources like Arxiv research papers (92 GB) and StackExchange's code-related Q&As (78 GB).

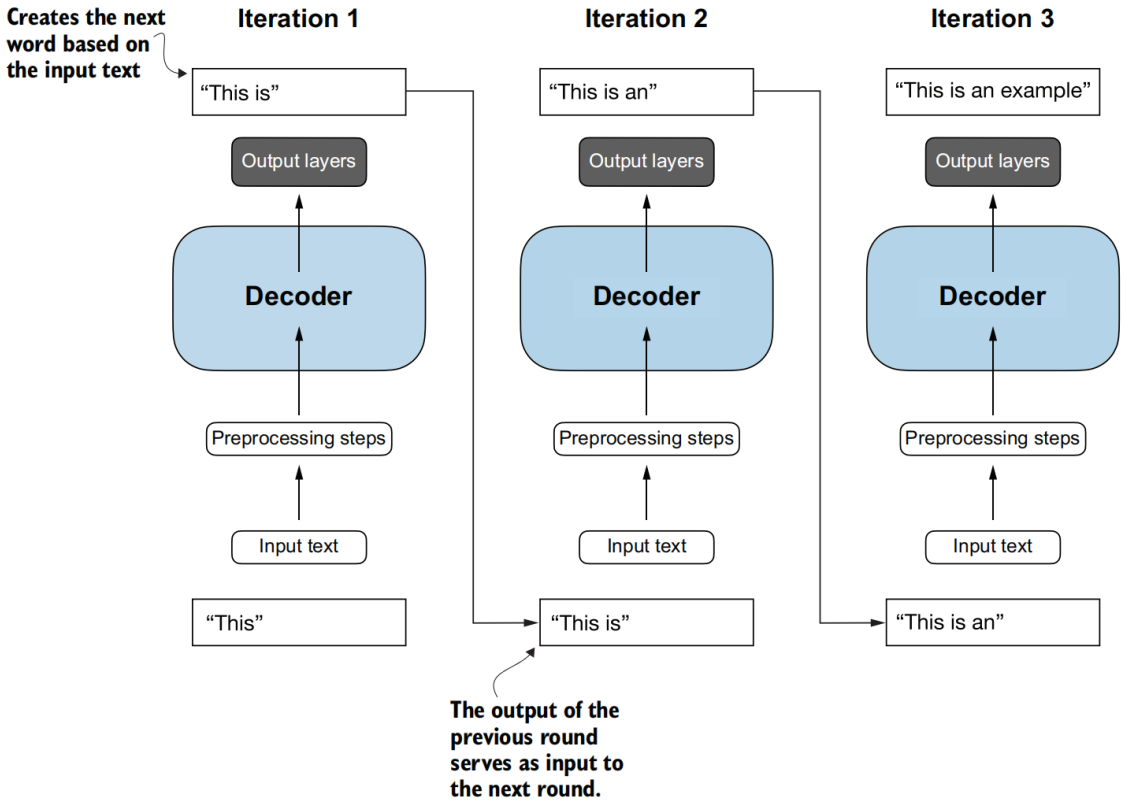
The authors of the GPT-3 paper did not share the training dataset, but a comparable dataset that is publicly available is **Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research** by Soldaini et al. 2024 (https://arxiv.org/abs/2402.00159).

(6) GPT architecture

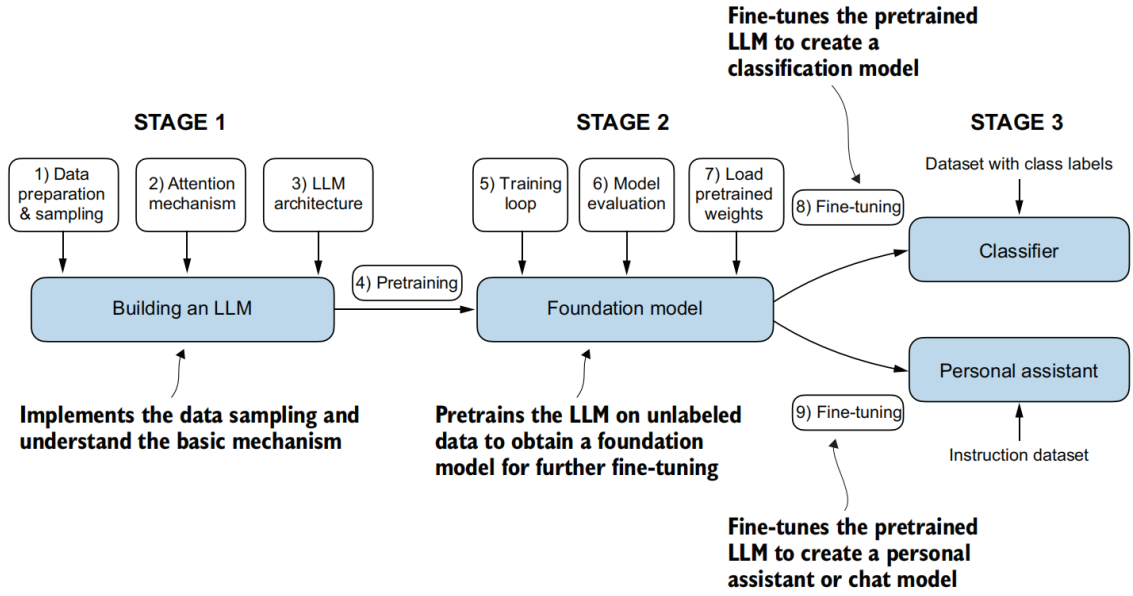
GPT was originally introduced in the paper "Improving Language Understanding by Generative Pre-Training" (https://mng.bz/x2qg) by Radford et al. from OpenAI.

The original model offered in ChatGPT was created by fine-tuning GPT-3 on a large instruction dataset using a method from OpenAI's InstructGPT paper (https://arxiv.org/abs/2203.02155).

The next-word prediction task is a form of self-supervised learning, which is a form of self-labeling. This means that we don't need to collect labels for the training data explicitly but can use the structure of the data itself: we can use the next word in a sentence or document as the label that the model is supposed to predict.



(7) build LLMs

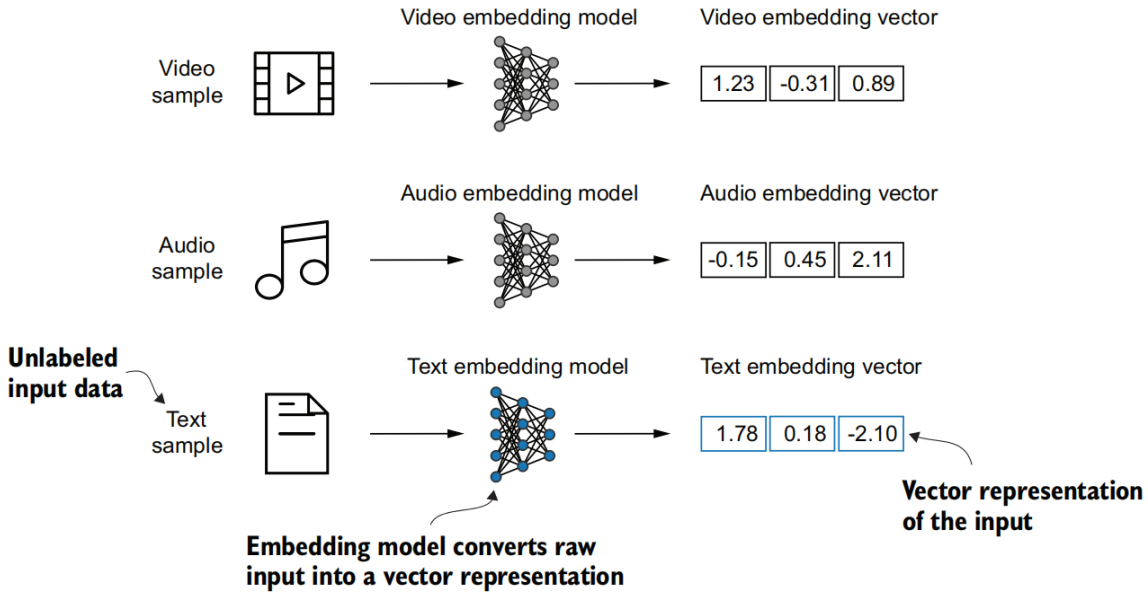


2. Working with text data

(1) word embedding

Deep neural network models, including LLMs, cannot process raw text directly. Since text is categorical, it isn't compatible with the mathematical operations used to implement and train neural networks. Therefore, we need a way to represent words as continuous-valued vectors.

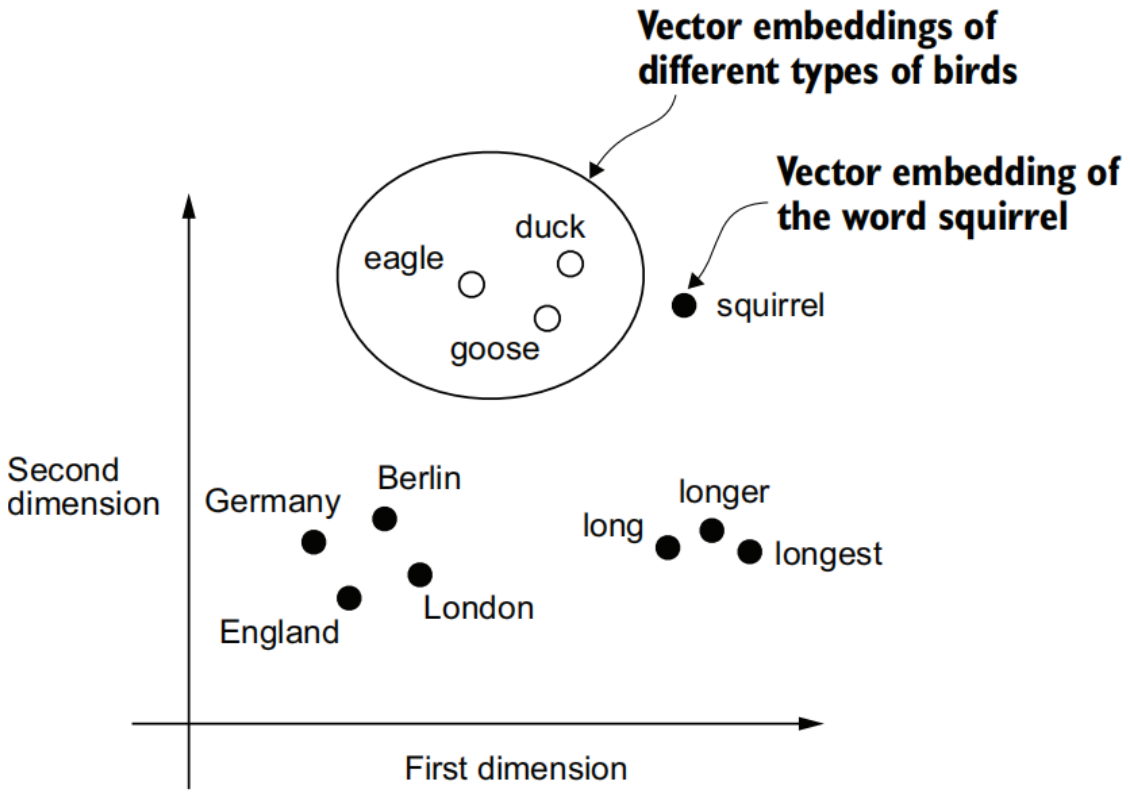
The concept of converting data into a vector format is often referred to as ***embedding***. Using a specific neural network layer or another pretrained neural network model, we can embed different data types—for example, video, audio, and text.



At its core, an embedding is a mapping from discrete objects, such as words, images, or even entire documents, to points in a continuous vector space.

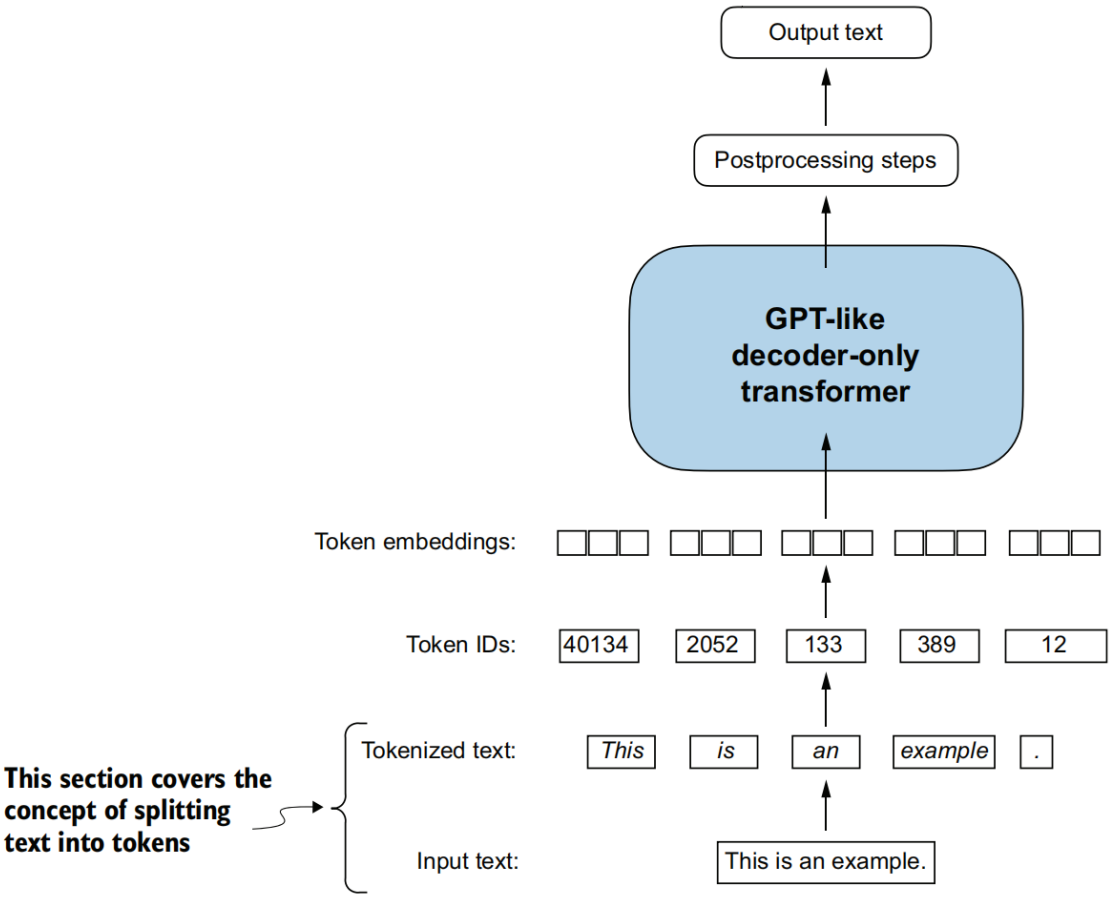
- Example: **Word2Vec** approach

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.



(2) Tokenizing text

- tokenization process



The text we will tokenize for LLM training is “The Verdict,” a short story by Edith Wharton. The text is available on Wiki source at https://en.wikisource.org/wiki/The_Verdict.

download code

```
import urllib.request
url = ("https://raw.githubusercontent.com/rasbt/"
       "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
       "the-verdict.txt")
file_path = "the-verdict.txt"
urllib.request.urlretrieve(url, file_path)
```

print for look

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

split the word

```
import re
text = "Hello, world. This, is a test."
result = re.split(r'(\s)', text)
print(result)
```

This simple tokenization scheme mostly works for separating the example text into individual words; however, some words are still connected to punctuation characters that we want to have as separate list entries.

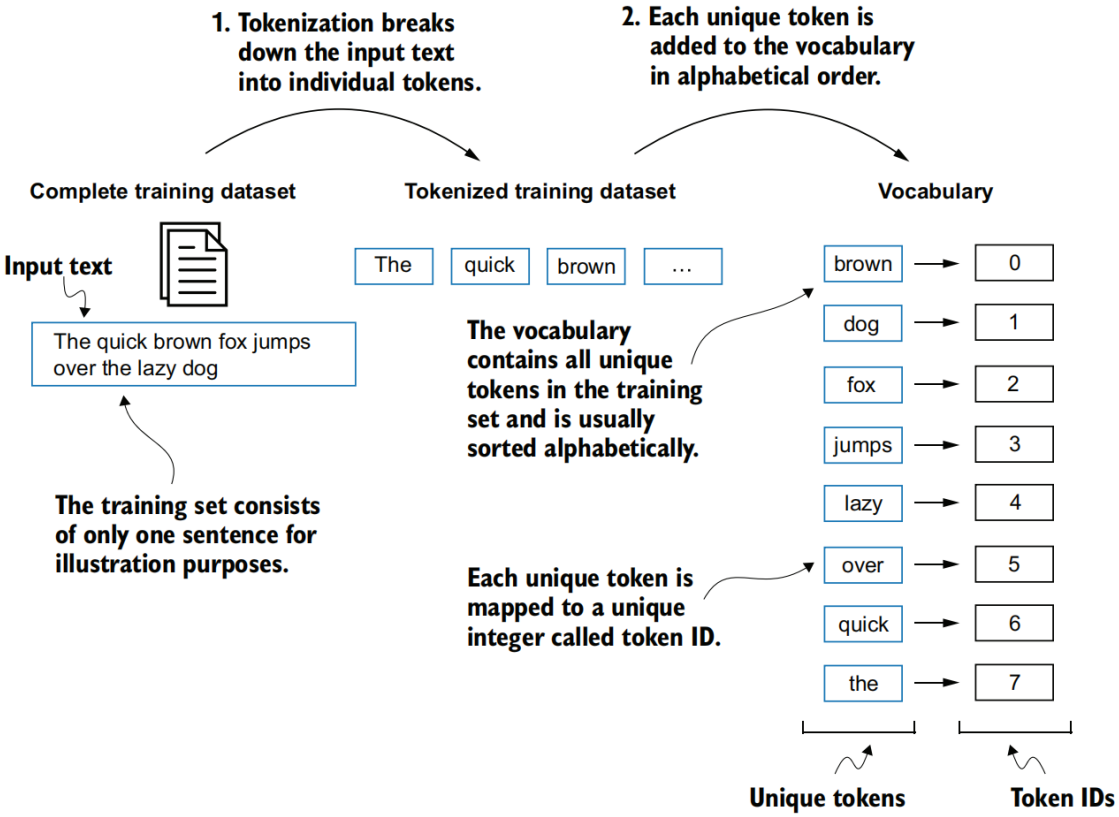
```
result = re.split(r'([,.]|\s)', text)
result = [item for item in result if item.strip()]
print(result)
```

Now that we have a basic tokenizer working

When developing a simple tokenizer, whether we should encode whitespaces as separate characters or just remove them depends on our application and its requirements. Removing whitespaces reduces the memory and computing requirements.

(3) Converting tokens into token IDs

Next, let's convert these tokens from a Python string to an integer representation to produce the token IDs. This conversion is an intermediate step before converting the token IDs into embedding vectors.




```
preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if item.strip()]
print(len(preprocessed))

# check length
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)
print(vocab_size)
```

create the vocabulary

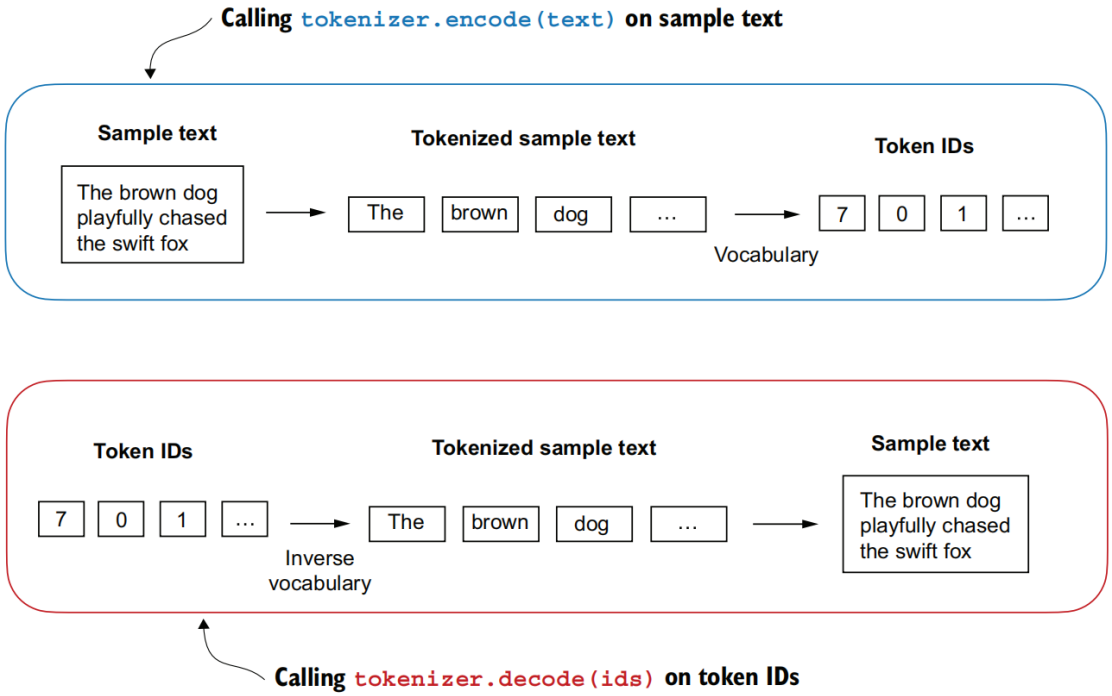
```
vocab = {token:integer for integer,token in enumerate(all_words)}
for i, item in enumerate(vocab.items()):
    print(item)
    if i >= 50:
        break
```

create the complete tokenizer

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i:s for s,i in vocab.items()}

        def encode(self, text):
            preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
            preprocessed = [
                item.strip() for item in preprocessed if item.strip()
            ]
            ids = [self.str_to_int[s] for s in preprocessed]
            return ids

        def decode(self, ids):
            text = " ".join([self.int_to_str[i] for i in ids])

            text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
            return text
```

you can encode from encoder and reverse id to text by decoder

```
tokenizer = SimpleTokenizerV1(vocab)
text = """It's the last he painted, you know,"
 Mrs. Gisburn said with pardonable pride."""
ids = tokenizer.encode(text)
print(ids)

# converse
print(tokenizer.decode(ids))
```



now, if you try

```
text = "Hello, do you like tea?"
print(tokenizer.encode(text))
```

you will get error as there are words not in vocabulary

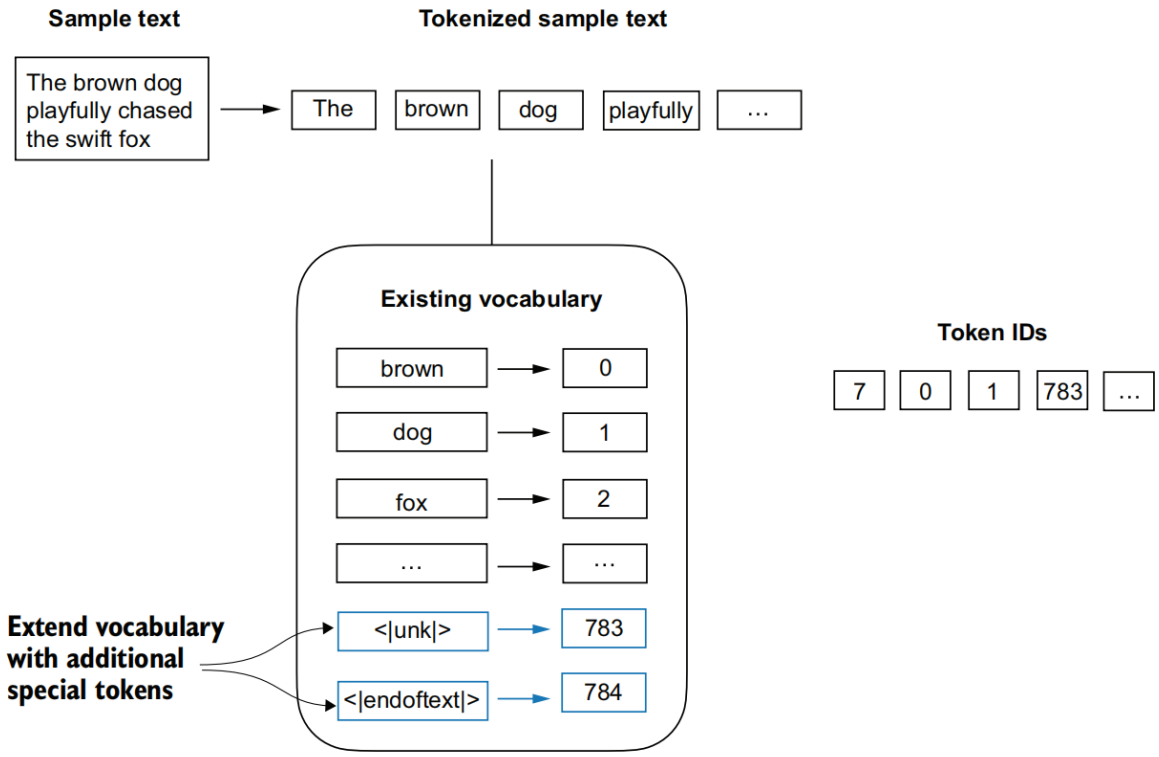
we will solve it nextly

(4) Adding special context tokens

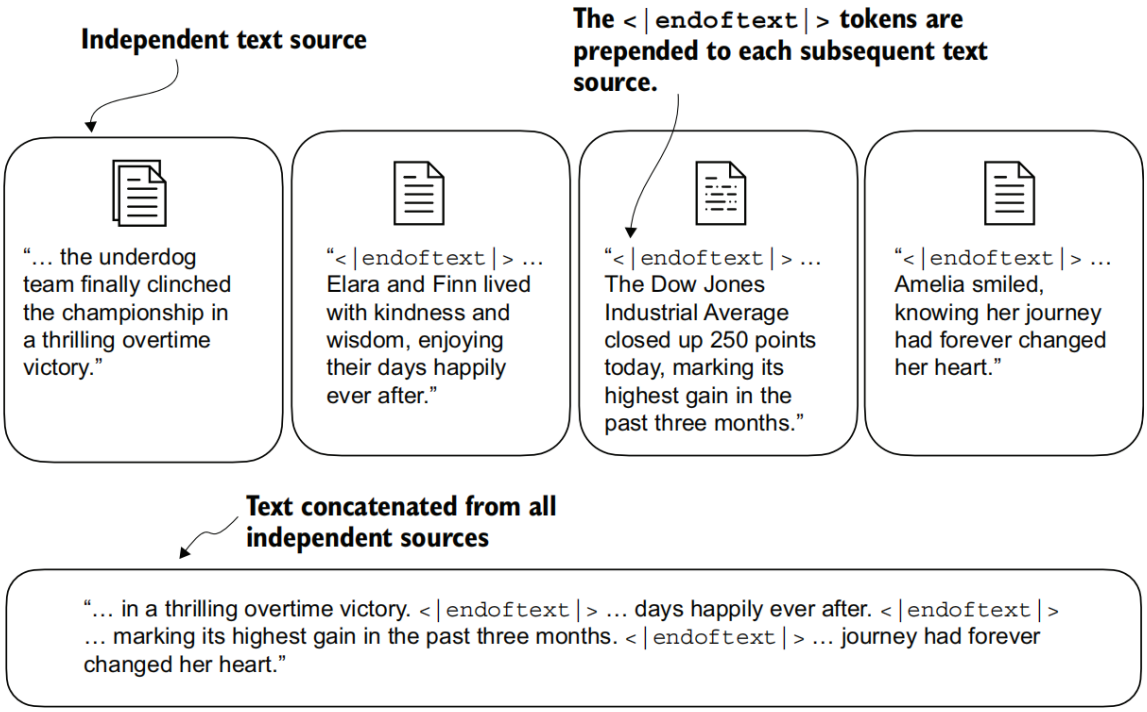
We need to modify the tokenizer to handle unknown words. We also need to address the usage and addition of special context tokens that can enhance a model's understanding of context or other relevant information in the text.

we will modify the vocabulary and tokenizer, `SimpleTokenizerV2`, to sup

port two new tokens, `<|unk|>` and `<|endoftext|>`



We can modify the tokenizer to use an `<|unk|>` token if it encounters a word that is not part of the vocabulary. Furthermore, we add a token `<|endoftext|>` between unrelated texts.




```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token:integer for integer,token in enumerate(all_tokens)}
print(len(vocab.items()))
```

after add new tokens, we can get

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
```

```
        self.int_to_str = { i:s for s,i in vocab.items()}

        def encode(self, text):
            preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)
            preprocessed = [
                item.strip() for item in preprocessed if item.strip()
            ]
            preprocessed = [item if item in self.str_to_int
                            else "<|unk|>" for item in preprocessed]
            ids = [self.str_to_int[s] for s in preprocessed]
            return ids

        def decode(self, ids):
            text = " ".join([self.int_to_str[i] for i in ids])
            text = re.sub(r'\s+([,.:;?!"()\'])', r'\1', text)
            return text
```

let's try simple text

```
text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."
text = " <|endoftext|> ".join((text1, text2))
print(text)
```

then, encode and decode our text

```
tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.encode(text))

# decode
print(tokenizer.decode(tokenizer.encode(text)))
```

(5) Byte pair encoding

Let's look at a more sophisticated tokenization scheme based on a concept called byte pair encoding (BPE). The BPE tokenizer was used to train LLMs such as GPT-2, GPT-3, and the original model used in ChatGPT.

Implementing BPE can be relatively complicated, we will use an existing Python open source library called *tiktoken* (https://github.com/openai/tiktoken)

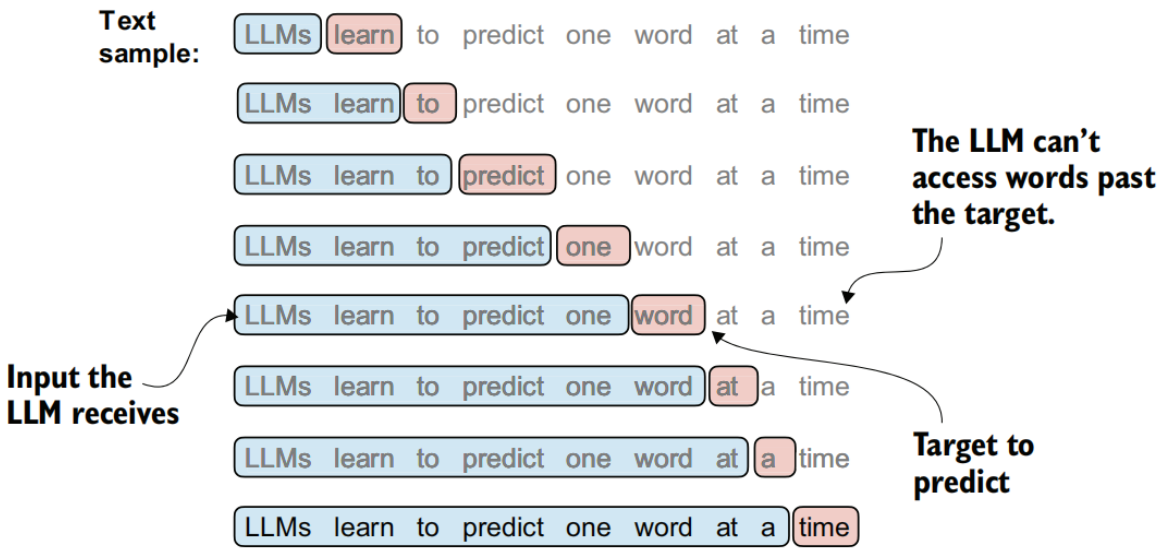
```
pip install tiktoken
```

```
from importlib.metadata import version
import tiktoken
print("tiktoken version:", version("tiktoken"))
tokenizer = tiktoken.get_encoding("gpt2")

# try encode with bpe
text = (
 "Hello, do you like tea? <|endoftext|> In the sunlit terraces"
 "of someunknownPlace."
)
```

```
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})  # add
split word
print(integers)

# decode
strings = tokenizer.decode(integers)
print(strings)
```

(6) Data sampling with a sliding window

The next step in creating the embeddings for the LLM is to generate the input–target pairs required for training an LLM.



To get started, we will tokenize the whole "The Verdict" short story using the BPE tokenizer

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

Next, we remove the first 50 tokens from the dataset for demonstration purposes, as it results in a slightly more interesting text passage in the next steps:

```
enc_sample = enc_text[50:]
```

One of the easiest and most intuitive ways to create the input–target pairs for the nextword prediction task is to create two variables, x and y, where x contains the input tokens and y contains the targets,

```
context_size = 4
x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y: {y}")
```

By processing the inputs along with the targets, which are the inputs shifted by one position, we can create the next-word prediction tasks

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

Let's repeat the previous code but convert the token IDs into text:

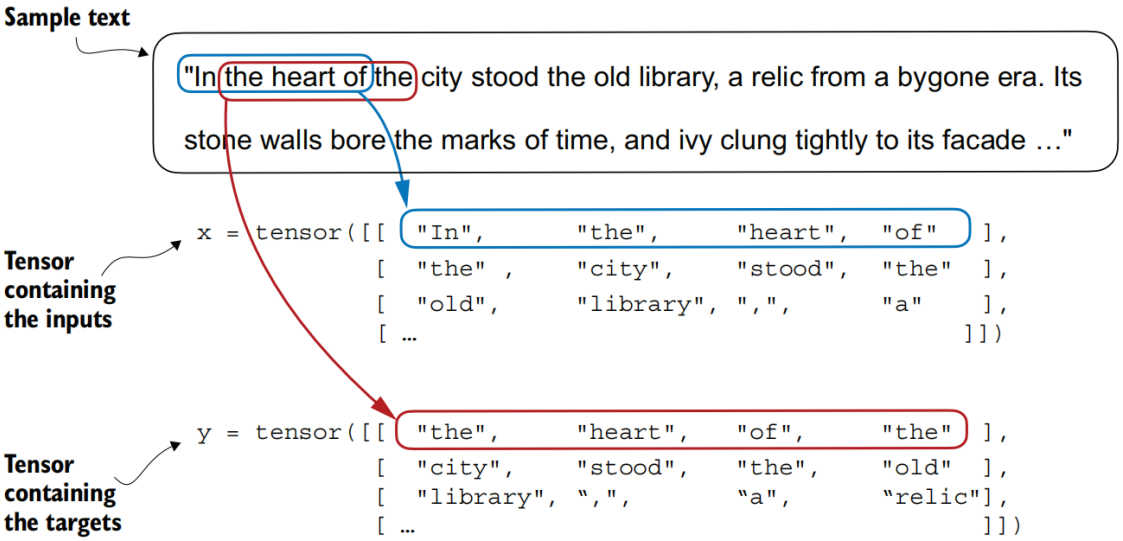
```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

We've now created the input–target pairs that we can use for LLM training.

There's only one more task before we can turn the tokens into embeddings: implementing an efficient data loader that iterates over the input dataset and returns theinputs and targets as PyTorch tensors, which can be thought of as multidimensional arrays.

we are interested in returning two tensors

- an input tensor containing the text that the LLM sees
- a target tensor that includes the targets for the LLM to predict




```
import torch
from torch.utils.data import Dataset, DataLoader
class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []
        token_ids = tokenizer.encode(txt)
        for i in range(0, len(token_ids) - max_length, stride):
            input_chunk = token_ids[i:i + max_length]
            target_chunk = token_ids[i + 1: i + max_length + 1]
            self.input_ids.append(torch.tensor(input_chunk))
            self.target_ids.append(torch.tensor(target_chunk))
    def __len__(self):
        return len(self.input_ids)
    def __getitem__(self, idx):
```

```
        return self.input_ids[idx], self.target_ids[idx]
```

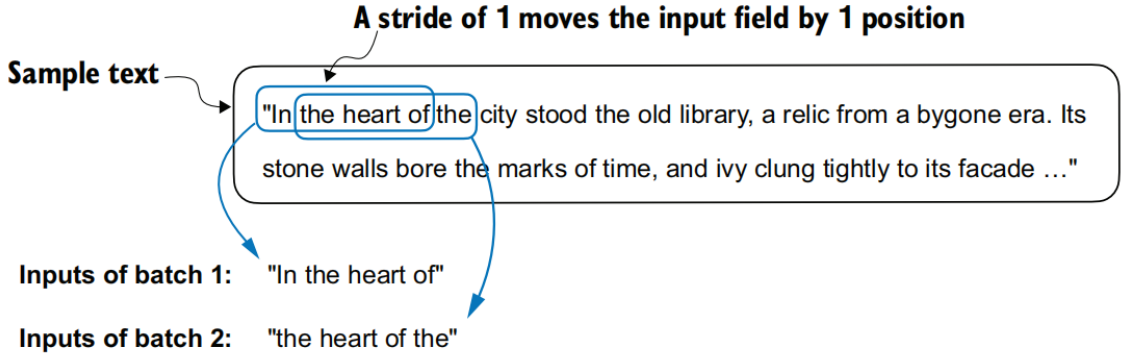
The `GPTDatasetV1` class is based on the PyTorch Dataset class and defines how individual rows are fetched from the dataset, where each row consists of a number of token IDs (based on a `max_length`) assigned to an `input_chunk` tensor. The `target_ chunk` tensor contains the corresponding targets.

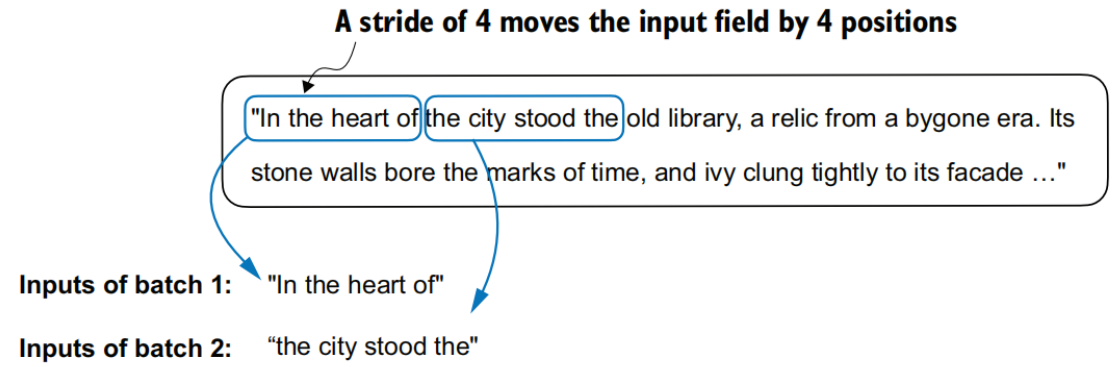
The following code uses the `GPTDatasetV1` to load the inputs in batches via a PyTorch `DataLoader`.

```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True,
                         num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last,
        num_workers=num_workers
    )
    return dataloader
```

Let's test the `dataloader` with a batch size of 1 for an LLM with a context size of 4

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, stride=1, shuffle=False)
data_iter = iter(dataloader)
first_batch = next(data_iter)
print(first_batch)
```





Let's look briefly at how we can use the data loader to sample with a batch size greater than 1

```
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=4, stride=4,
    shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

(7) Creating token embeddings

The last step in preparing the input text for LLM training is to convert the token IDs into embedding vectors

Let's see how the token ID to embedding vector conversion works with a hands-on example.

```
input_ids = torch.tensor([2, 3, 5, 1])
vocab_size = 6
output_dim = 3

torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

Now, let's apply it to a token ID to obtain the embedding vector

```
print(embedding_layer(torch.tensor([3])))
```

We've seen how to convert a single token ID into a three-dimensional embedding vector. Let's now apply that to all four input IDs

```
print(embedding_layer(input_ids))
```

Each row in this output matrix is obtained via a lookup operation from the embed

ding weight matrix

3. Coding attention mechanisms

4. Implementing a GPT model from scratch to generate text

5. Pretraining on unlabeled data

6. Fine-tuning for classification

7. Fine-tuning to follow instructions