Chapter-wise Important Concepts Of Sebastian Raschka's Build a Large Language Model from Scratch

Parit Kansal

Content

1. Understanding Large Language Models
    1.1. Defining the Modern LLM
    1.2. Applications and Development Lifecycle
        1.2.1. Core Applications
        1.2.2. The LLM Development Lifecycle
    1.3. The Transformer Architecture: The Engine of LLMs
    1.4. A Closer Look at the GPT Architecture
    1.5. A Practical Roadmap for Building an LLM
2. Working with text data
    2.1. Understanding word embeddings
    2.2. Tokenizing text
    2.3. Converting tokens into token IDs
    2.4. Adding special context tokens
    2.5. Byte Pair Encoding
        2.5.1. Training Algorithm
        2.5.2. Encoding Algorithm
        2.5.3. Custom Python Code

1. Understanding Large Language Models

Large Language Models (LLMs) signify a fundamental change from older Natural Language Processing (NLP) models, which were typically rule-based or created for specific tasks like spam detection. Today's LLMs, built on the **transformer architecture** and trained on

extensive text datasets, demonstrate impressive adaptability and a thorough grasp of language, allowing them to perform exceptionally well in many difficult language-related assignments.

1.1 Defining the Modern LLM

LLMs are deep neural networks, typically comprising tens to hundreds of billions of parameters, and built upon the transformer architecture. They are a prominent application of Generative AI (GenAI) due to their text-generation capabilities.

A key distinction from traditional machine learning is the LLM's ability to perform automatic feature extraction. While conventional ML models for NLP often required manual feature engineering (e.g., n-grams, bag-of-words), deep learning architectures like transformers learn hierarchical and contextual features directly from raw text data.

The core training methodology for LLMs is a self-supervised objective, most commonly next-word prediction, which allows the model to learn syntax, semantics, and contextual relationships from massive, unlabeled datasets.

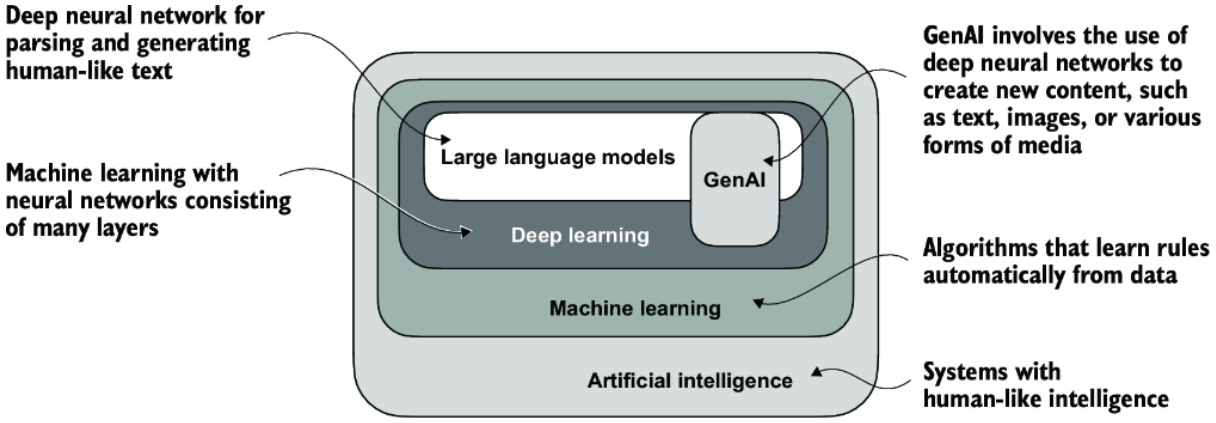


Fig: LLMs are a specific application of deep learning, a branch of machine learning that uses multilayer neural networks. Both machine learning and deep learning focus on enabling computers to learn from data and perform tasks requiring human intelligence.

1.2 Applications and Development Lifecycle

1.2.1 Core Applications

Large Language Models (LLMs) excel at processing and generating unstructured text, making them highly versatile for various applications:

- **Text Creation:** This includes creative writing, generating code, drafting articles, and producing documentation.
- **Natural Language Understanding (NLU):** LLMs can perform sentiment analysis, named entity recognition, and text classification.
- **Content Transformation:** They are capable of summarizing extensive documents and translating between different languages.
- **Conversational Interfaces:** LLMs power chatbots and virtual assistants like ChatGPT and Google Gemini.

- **Specialized Knowledge Retrieval:** They enhance search and knowledge retrieval in fields such as finance, law, and medicine for domain-specific Q&A.

1.2.2 The LLM Development Lifecycle

The creation and deployment of a Large Language Model (LLM) involves a two-step process:

1. **Pretraining:** This initial, computationally intensive phase involves training the model on a vast and diverse collection of unlabeled text data, such as Common Crawl and Wikipedia. The training is self-supervised, often with the goal of predicting the next token in a sequence. The result of this stage is a **foundation model** (or base model), which possesses extensive linguistic knowledge and exhibits emergent abilities like few-shot learning.
2. **Fine-tuning:** Following pretraining, the foundation model is tailored for specific tasks using a smaller, labeled dataset. This specialization process is more resource-efficient than pretraining. Common fine-tuning approaches include:
    - **Instruction Fine-tuning:** Training with instruction-response pairs (e.g., question-answer datasets) to develop conversational agents.
    - **Classification Fine-tuning:** Training on labeled examples (e.g., distinguishing spam from non-spam) to construct specialized classifiers.

Developing custom LLMs offers several benefits over using general-purpose models, including superior performance in specific domains, enhanced data privacy and sovereignty, reduced inference latency, and greater control over the model's architecture.

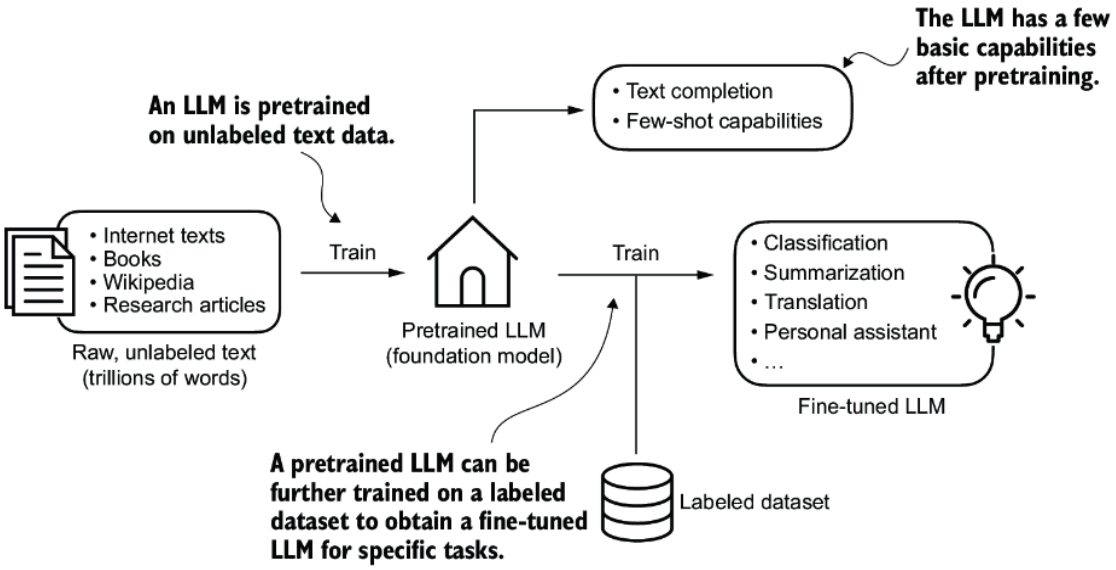


Fig: Pretraining an LLM involves next-word prediction on large text datasets. A pretrained LLM can then be fine-tuned using a smaller labeled dataset.

1.3 The Transformer Architecture: The Engine of LLMs

The transformer architecture, introduced in "Attention Is All You Need" (2017), serves as the bedrock for most cutting-edge Large Language Models (LLMs). Its core innovation lies in the **self-attention mechanism**, which enables the model to dynamically assess the significance

of different words within an input sequence, thereby effectively capturing long-range dependencies.

The original transformer architecture comprises an **encoder** and a **decoder**, giving rise to two distinct families of models:

- **Encoder-Only Models (e.g., BERT):** These models process the entire input text bidirectionally to create rich contextual representations. They excel at Natural Language Understanding (NLU) tasks such as sentiment analysis, text classification, and named entity recognition. BERT is commonly pretrained using a **masked language modeling (MLM)** objective.

- **Decoder-Only Models (e.g., GPT):** These models are **autoregressive**, generating text one token at a time from left to right. This architecture is optimized for Natural Language Generation (NLG) tasks including text completion, translation, and summarization. Their proficiency in **zero-shot** and **few-shot learning** is a result of the extensive knowledge acquired during pretraining.

Note: *While the majority of modern LLMs are built upon the transformer architecture, the terms are not interchangeable. Transformers also find applications in other fields like computer vision, and some LLMs explore alternative architectures (e.g., RNN-based) to enhance computational efficiency.*

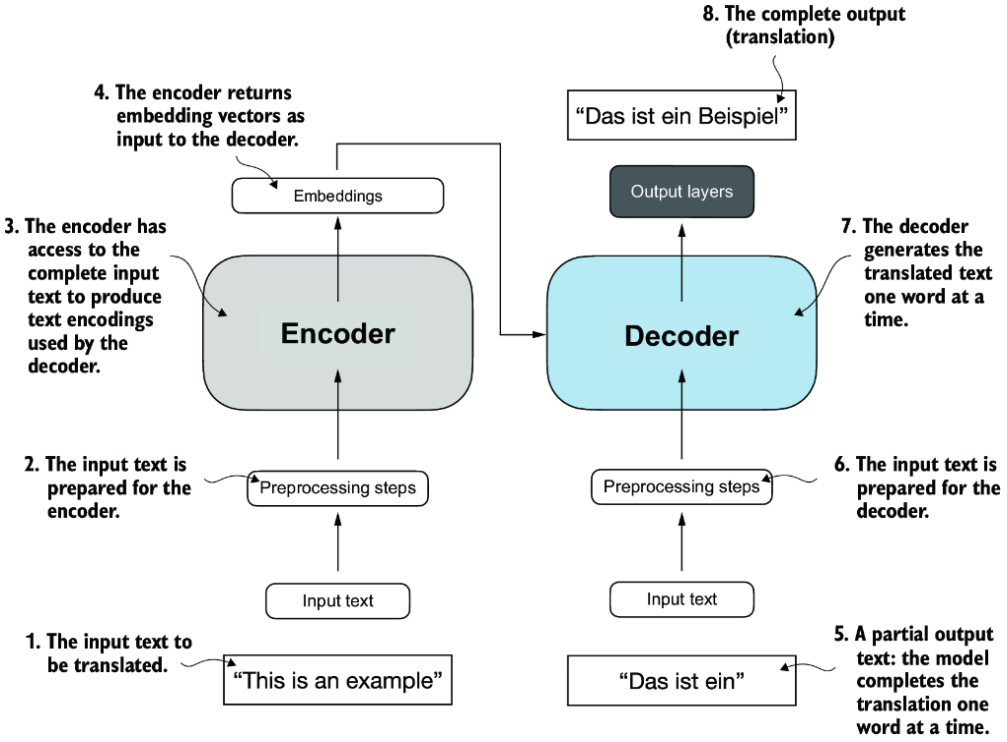


Fig: Simplified transformer for translation with (a) an encoder converting input text into embeddings, and (b) a decoder generating the translation word by word. The figure shows the final step, where "Beispiel" is produced from "This is an example" and the partial output "Das ist ein."

1.4 A Closer Look at the GPT Architecture

The Generative Pretrained Transformer (GPT) series showcases the effectiveness of large-scale decoder-only models. Their key characteristics in terms of architecture and training are:

- **Architecture:** These models employ a **decoder-only** transformer architecture. For instance, GPT-3 features 96 layers and encompasses 175 billion parameters.
- **Training Objective:** GPT models are exclusively pretrained on a **next-word prediction** task. This self-supervised approach allows them to be trained on vast amounts of unlabeled web-scale data.
- **Autoregressive Nature:** Text generation is performed sequentially, with each predicted token being dependent on the sequence of previously generated tokens.

The remarkable scale of the model and training data contributes to **emergent abilities** that were not explicitly trained for. These include capabilities like translation and in-context learning, which arise from exposure to diverse multilingual and multi-format data. As an example, GPT-3 was trained on approximately 300 billion tokens from sources such as CommonCrawl, WebText2, and Wikipedia, with an estimated computational cost of $4.6 million.

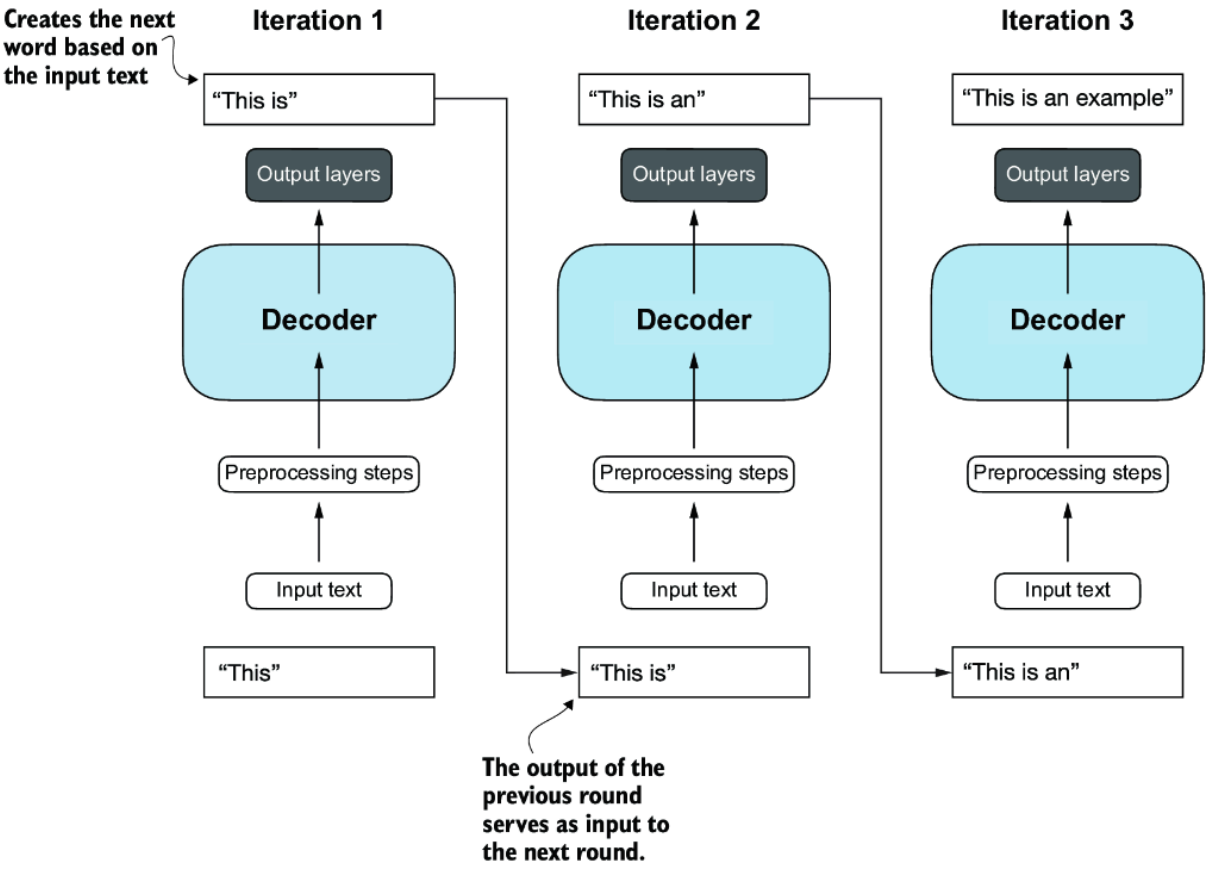


Fig: The GPT architecture employs only the decoder portion of the original transformer. It is designed for unidirectional, left-to-right processing, making it well suited for text generation and next-word prediction tasks to generate text in an iterative fashion, one word at a time.

1.5 A Practical Roadmap for Building an LLM

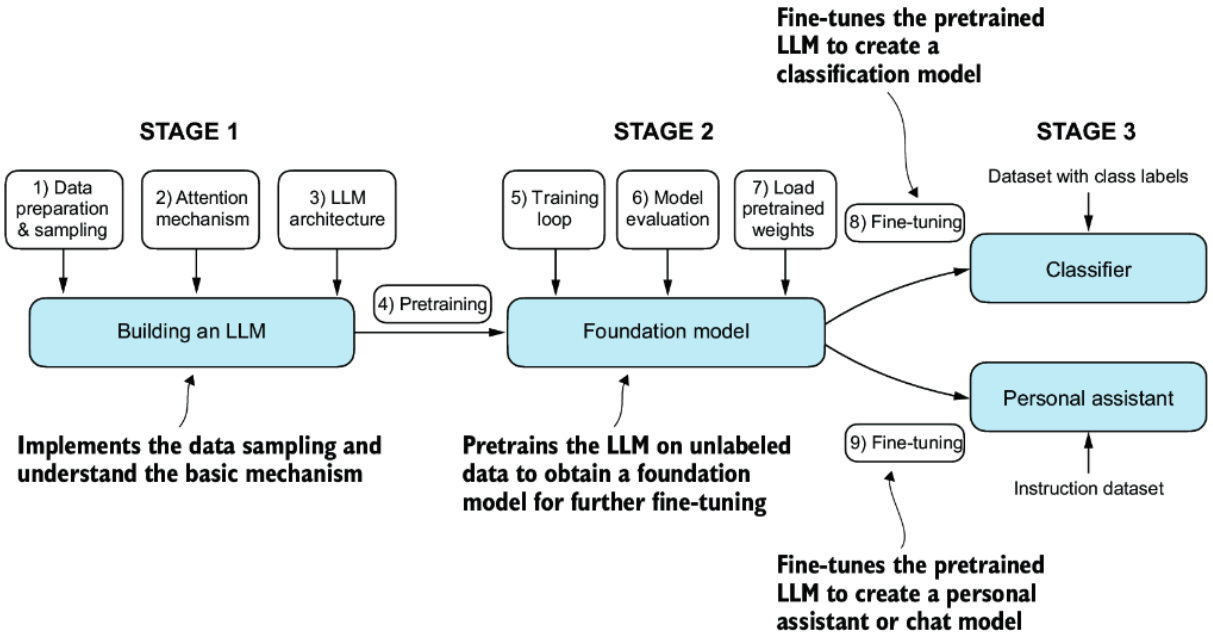


Fig: The three main stages of coding an LLM are implementing the LLM architecture and data preparation process (stage 1), pretraining an LLM to create a foundation model (stage 2), and fine-tuning the foundation model to become a personal assistant or text classifier (stage 3).

2 Working with text data

This section focuses on preparing input text for training Large Language Models (LLMs). We will explore the process of splitting text into word and subword tokens, which are then converted into vector representations for the LLM. The discussion will also cover advanced tokenization methods, such as byte pair encoding, used in well-known LLMs like GPT. Finally, we will implement a sampling and data-loading strategy to generate the necessary input-output pairs for LLM training.

2.1 Understanding word embeddings

Converting data into a vector format is known as embedding. Essentially, an embedding transforms discrete objects such as words, images, or entire documents into points within a continuous vector space. The main goal of this process is to make non-numeric data understandable to neural networks.

While word embeddings are the most common form, text can also be embedded at the sentence, paragraph, or document level. Sentence and paragraph embeddings are particularly useful for retrieval-augmented generation, a technique that combines text generation with information retrieval from external knowledge bases.

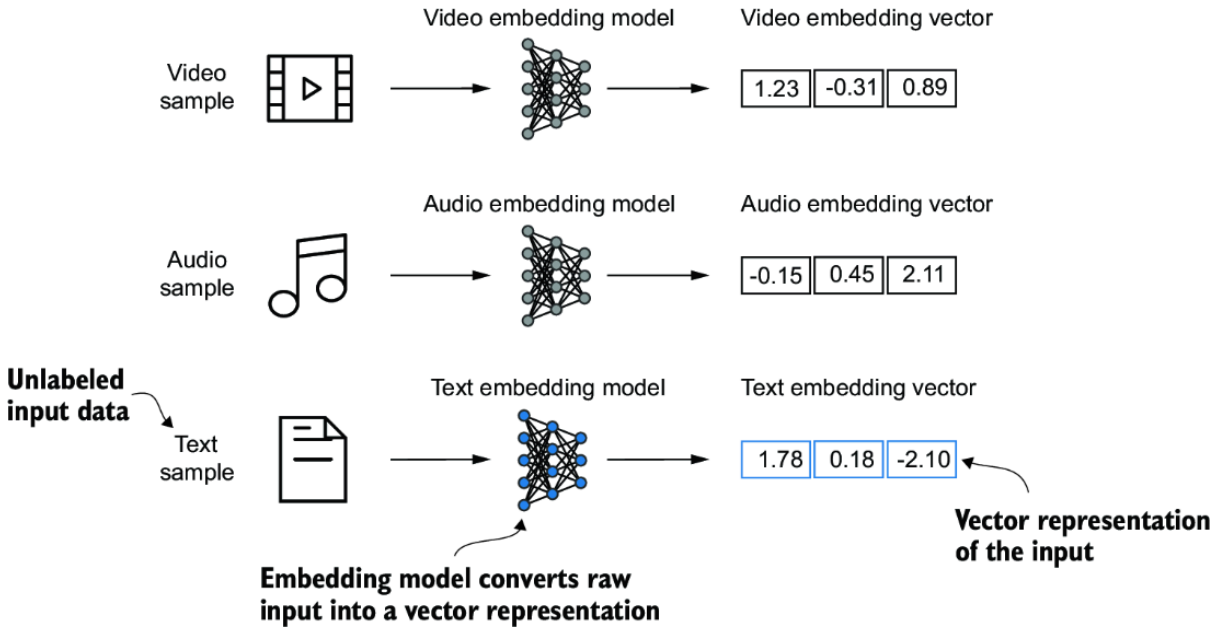


Fig: Deep learning models cannot process data formats like video, audio, and text in their raw form. Thus, we use an embedding model to transform this raw data into a dense vector representation that deep learning architectures can easily understand and process. Specifically, this figure illustrates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

Word2Vec is a notable method that creates embeddings by predicting a word's context or vice versa. This approach results in words with similar meanings being positioned closely together in the embedding space. Embeddings can have varying dimensions, from a few to thousands; higher dimensions capture more detail but are computationally more intensive.

Large Language Models (LLMs) typically develop their own embeddings during training, making them more effective and specific to their tasks than pre-trained Word2Vec vectors. However, high-dimensional embeddings are difficult to visualize. For instance, GPT-2 uses 768-dimensional embeddings, while the largest GPT-3 model uses 12,288 dimensions.

2.2 Tokenizing text

Before creating embeddings, text must be **split into tokens**—words or special characters. We can load the text in Python:

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

Tokenization can be done with Python's **re** library. A basic split on whitespace (`re.split(r'(\s)', text)`) separates words but keeps punctuation attached. By extending the regex (`re.split(r'([,.:;?_!"()\']|--|\s)', text)`), we can split out punctuation and special characters.

```
text = "Hello, world. Is this-- a test?"
tokens = re.split(r'([,.:;?_!"()\']|--|\s)', text)
tokens = [t.strip() for t in tokens if t.strip()]
print(tokens)
```

This scheme separates words, punctuation, and handles special cases like `--`.

***Note**: Whether to keep whitespaces depends on the task. For simplicity, we remove them here, but in some cases (e.g., Python code), they are important. Later, we will use prebuilt tokenizers.*

2.3 Converting tokens into token IDs

Once we have split text into tokens, the next step is to represent them numerically. Neural networks cannot directly process words or characters—they require numbers. To achieve this, we first build a **vocabulary** that maps each unique token to a unique integer ID.

Building a Vocabulary

We begin by extracting all unique tokens from our preprocessed text:

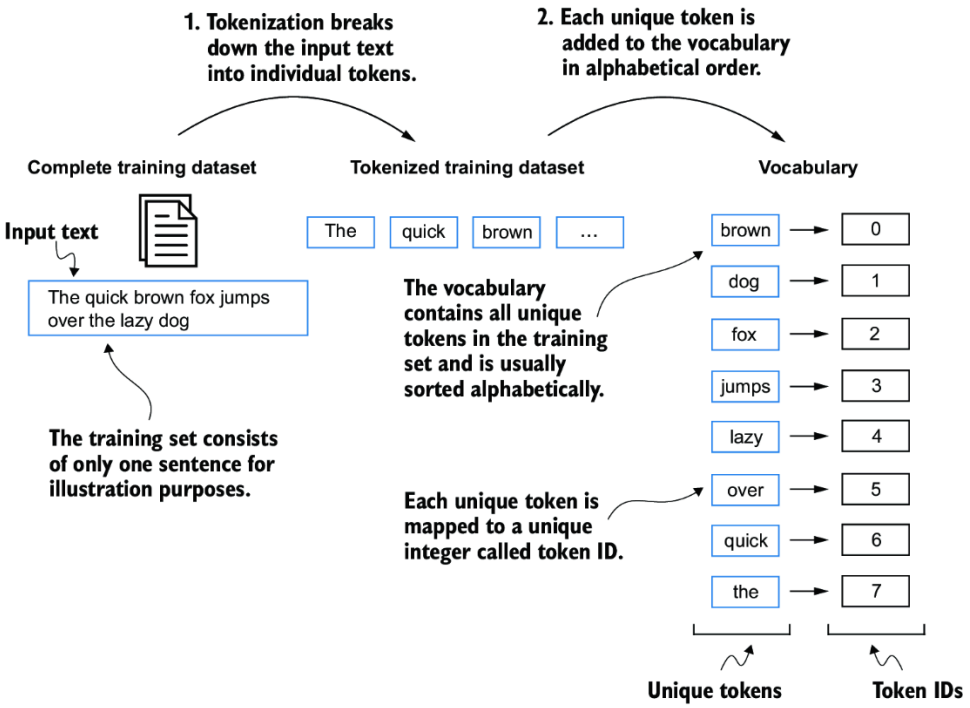
```
all_words = sorted(set(preprocessed))   # Get unique tokens in sorted order
vocab_size = len(all_words)             # Size of the vocabulary
```

Here, `all_words` contains all distinct tokens, and `vocab_size` tells us how many unique tokens exist in our dataset.

Next, we assign each token a unique integer:

```
vocab = {token: integer for integer, token in enumerate(all_words)}
```

***Note**: This vocabulary is fixed. If you try to encode a word that is not in `vocab`, the tokenizer will raise an error (`KeyError`). This is because the tokenizer does not yet handle out-of-vocabulary (OOV) words.*



Implementing a Simple Tokenizer

To automate the conversion between text and IDs, we define a tokenizer class:

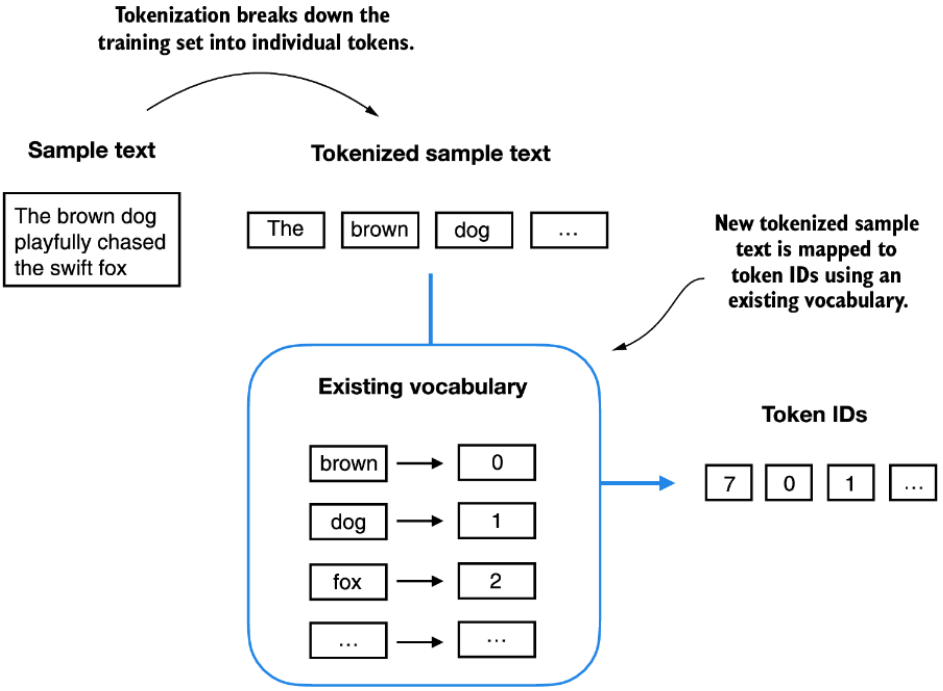
```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        # Mapping from token → id
        self.str_to_int = vocab
        # Mapping from id → token
        self.int_to_str = {i: s for s, i in vocab.items()}

    def encode(self, text):
        # Split text into tokens (words + punctuation)
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        # Remove empty strings
        preprocessed = [item.strip() for item in preprocessed if item.strip()]
        # Convert tokens into integer IDs
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        # Convert IDs back into tokens
        text = " ".join([self.int_to_str[i] for i in ids])
        # Remove extra spaces before punctuation
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
        return text
```

This class provides two methods:

- **encode**: Converts raw text into a list of token IDs.
- **decode**: Converts a list of token IDs back into readable text.



Example Usage

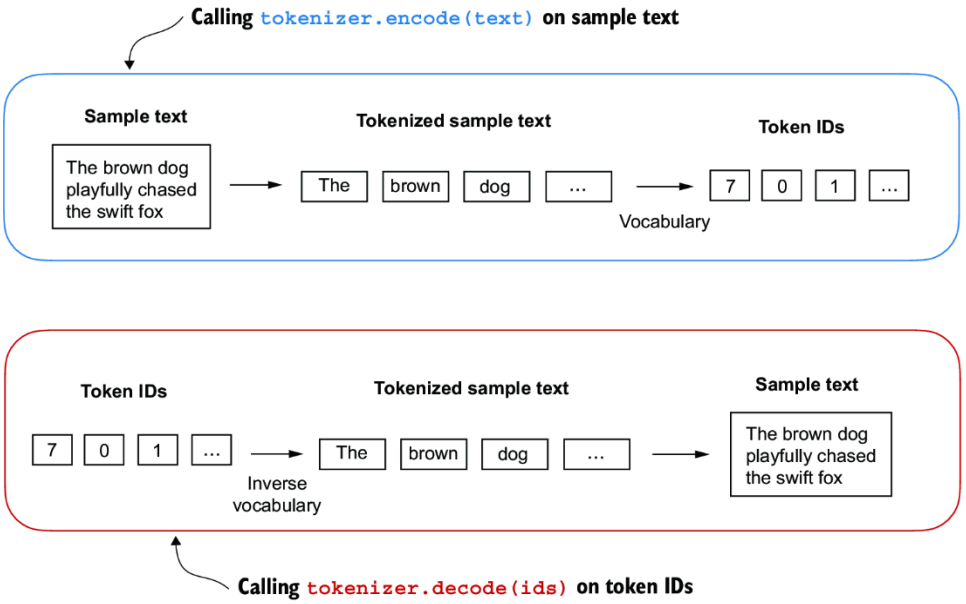
Let's apply our tokenizer to a short piece of text:

```
tokenizer = SimpleTokenizerV1(vocab)

text = """It's the last he painted, you know"""

# Encode text into token IDs
ids = tokenizer.encode(text)
print(ids)

# Decode IDs back into text
decoded = tokenizer.decode(ids)
print(decoded)
```



2.4 Adding special context tokens

When building tokenizers, we must handle unknown words and mark text boundaries. To achieve this, we add two special tokens:

- **<|unk|>** — Represents words not in the vocabulary.
- **<|endoftext|>** — Marks the boundary between independent text sources.

These tokens help the model distinguish unrelated texts and handle new words gracefully.

Adding special tokens to the vocabulary:

```
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()))
```

Example output (last 5 entries):

```
('younger', 1127)
('your', 1128)
('yourself', 1129)
('<|endoftext|>', 1130)
('<|unk|>', 1131)
```

Tokenizer implementation (SimpleTokenizerV2):

```
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
```

Example usage:

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

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

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

Additional Special Tokens

Depending on the model, researchers sometimes introduce other tokens:

- **[BOS] (beginning of sequence):** Marks where a text starts.
- **[EOS] (end of sequence):** Marks where a text ends, similar to `<|endoftext|>`.
- **[PAD] (padding):** Used to make all sequences in a batch the same length.

However, GPT models use only `<|endoftext|>`. This token serves both as an **end marker** and as **padding** when necessary. For out-of-vocabulary handling, GPT avoids `<|unk|>` by using **byte pair encoding (BPE)**, which splits words into smaller subword units—a topic we will explore in the next section.

2.5 Byte Pair Encoding

The BPE algorithm is split into two distinct phases: training, where merge rules are learned from a corpus, and encoding, where those rules are applied to new text.

2.5.1 Training Algorithm

The goal of training is to learn an ordered set of merge rules from a text corpus.

```
FUNCTION Train(corpus, num_merges):
    // Step 1: Initialize vocabulary
    // Each word is split into characters, with a special end-of-word symbol.
    // e.g., "cat" -> "c a t </w>"
    // The vocabulary stores the frequency of each such sequence.
    vocab = initialize_vocabulary_from_corpus(corpus)

    // This will store the learned merge rules in order.
    merges = new empty_ordered_list()

    // Step 2: Iteratively learn merge rules
    LOOP i FROM 1 TO num_merges:
        // Count frequencies of all adjacent symbol pairs in the current vocabulary
        pair_counts = get_pair_statistics(vocab)

        // If there are no more pairs to merge, stop early
        IF pair_counts is empty THEN
            BREAK LOOP
        END IF

        // Find the most frequent pair
        best_pair = find_most_frequent_pair(pair_counts)

        // Merge this pair into a new token
        new_token = merge_symbols(best_pair)

        // Apply this merge to the entire vocabulary for the next iteration
        vocab = apply_merge_to_vocab(vocab, best_pair, new_token)

        // Save the learned rule. The order is important!
```

```
        ADD (best_pair -> new_token) TO merges

    RETURN merges
END FUNCTION
```

2.5.2 Encoding Algorithm

The goal of encoding is to tokenize a new string using the previously learned merge rules.

```
FUNCTION Encode(text, merges):
    // This will store the final list of tokens for the entire text
    output_tokens = new empty_list()

    words = split_text_into_words(text)

    FOR EACH word IN words:
        // Pre-process the word just like in training
        symbols = split_word_into_characters(word) + end_of_word_symbol

        // Apply every learned merge rule in the exact order they were learned
        FOR EACH rule (pair -> new_token) IN merges:
            symbols = replace_all_occurrences(symbols, pair, new_token)
        END FOR

        // Add the resulting sequence of tokens to our final list
        ADD symbols TO output_tokens
    END FOR

    RETURN output_tokens
END FUNCTION
```

2.5.3 Custom Python Code

This is a self-contained Python implementation based on the pseudocode above. It includes the tokenizer class and an example of how to train and use it.

```python
def get_stats(vocab):
    """Count frequency of adjacent symbol pairs in the vocabulary."""
    pairs = {}
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols) - 1):
            key = (symbols[i], symbols[i + 1])
            pairs[key] = pairs.get(key, 0) + freq
    return pairs

def merge_vocab(pair, vocab):
    """Replace occurrences of the pair 'a b' with 'ab' """
    bigram = ' '.join(pair)
    merged = ''.join(pair)
    new_vocab = {}
    for word, freq in vocab.items():
        new_word = word.replace(bigram, merged)
        new_vocab[new_word] = freq
    return new_vocab

class BpeTokenizer:
    def __init__(self):
```

```
        # merges is an ordered dict-like mapping (pair_tuple) -> merged_string
        self.merges = {}
        self.vocab = []

    def train(self, text, num_merges):
        """Train on a whitespace-separated corpus and learn `num_merges`

        merges"""
        # Format training vocab as: characters separated by spaces + end-of-word

        token </w>
        vocab = {}
        for word in text.split():
            key = ' '.join(list(word)) + ' </w>'
            vocab[key] = vocab.get(key, 0) + 1

        # Track initial character-level vocabulary (informational)
        self.vocab = sorted(set(''.join(vocab.keys())))

        print("*" * 135)
        for i in range(num_merges):
            pairs = get_stats(vocab)
            if not pairs:
                break
            best = max(pairs, key=pairs.get)
            self.merges[best] = ''.join(best)
            vocab = merge_vocab(best, vocab)

        # Add learned subwords to final vocab list (optional)
        self.vocab.extend(self.merges.values())

    def encode(self, text):
        """Encode new text using learned merges (applies merges in learned

        order).

        """
        tokens = []
        for word in text.split():
            w = ' '.join(list(word)) + ' </w>'
            # Apply merges in the same order they were learned
            print(w, end = "")
            for pair, merged in self.merges.items():
                temp = w
                w = w.replace(' '.join(pair), merged)
                if temp != w:
                  print(" -> ", w, end="")
            # split back into tokens (merged tokens have no spaces)
            print()
            print("*"*100)
            tokens.extend([t for t in w.split(' ') if t])
        return tokens



num_merges_to_learn = 500
tok = BpeTokenizer()
tok.train(raw_text, num_merges_to_learn)



print("\nLearned merges:", tok.merges)
print("Final vocab size (approx):", len(tok.vocab))
```

```
print("\nEncode:", tok.encode("""It's the last he painted, you know,
       Mrs. Gisburn said with pardonable pride...""" ))
```

2.5.4 Python Code Using tiktoken

```
!pip install tiktoken
```

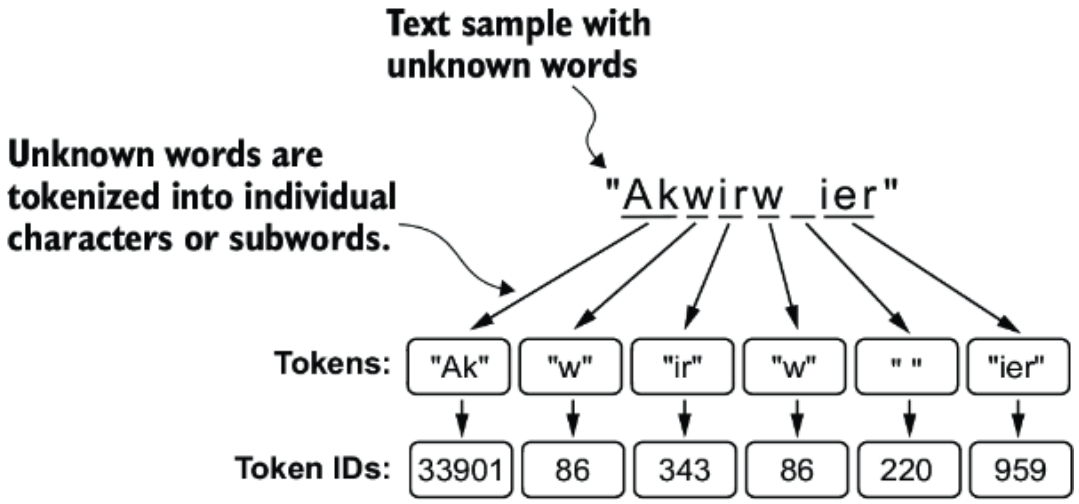
```
from importlib.metadata import version
import tiktoken
```

```
print("tiktoken version:", version("tiktoken"))
tokenizer = tiktoken.get_encoding("gpt2")
```

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

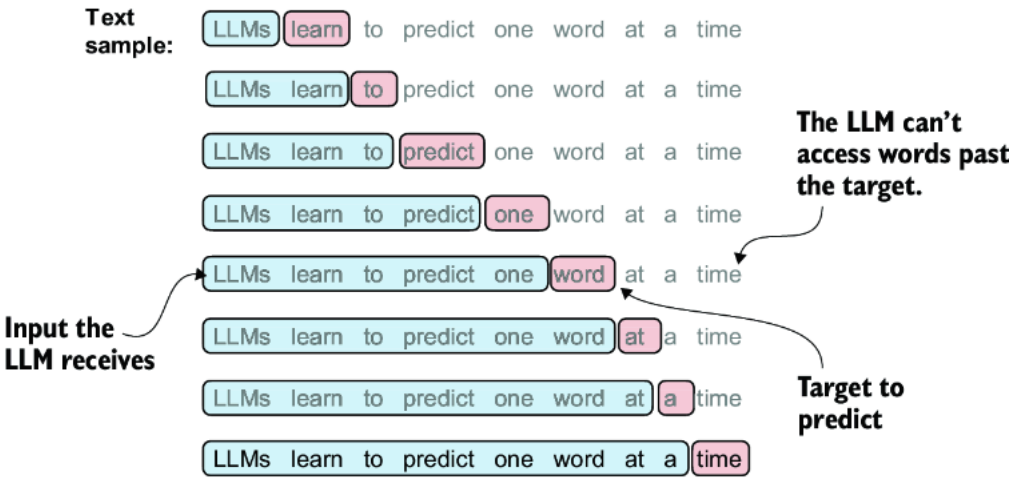
```
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
```

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

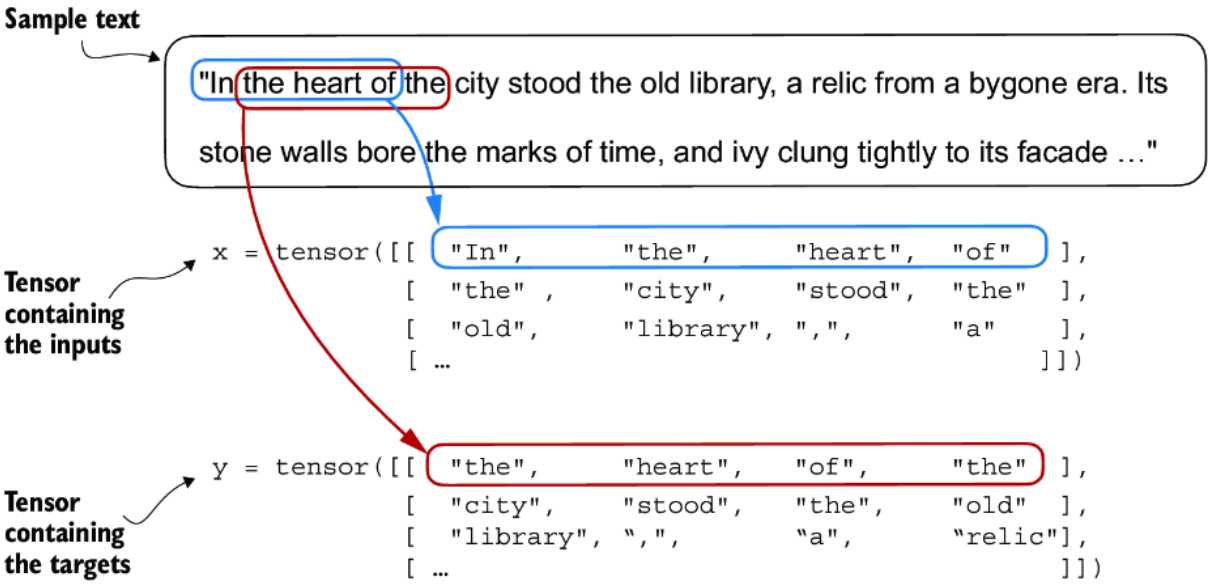


2.6 Data sampling with a sliding window

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.



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.




```
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]



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



with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

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)
```

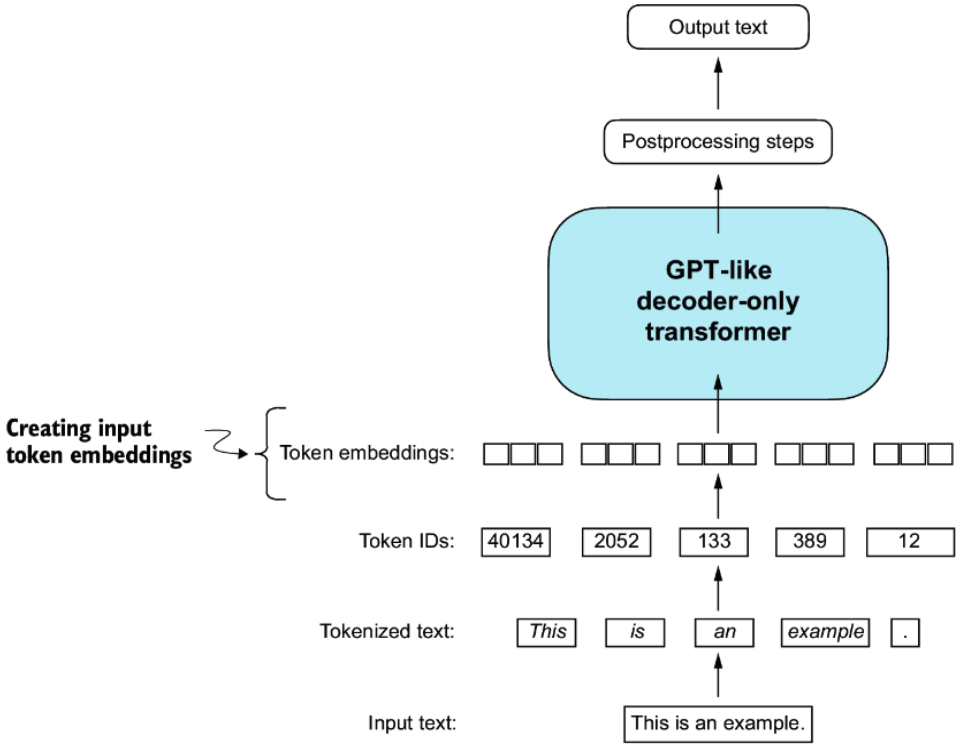
Note: *We use stride = max_length to utilize the dataset fully (we don't skip a single word). This avoids any overlap between the batches since more overlap could lead to increased overfitting.*

2.7 Creating token embeddings

The final step in preparing input text for LLM training is to convert token IDs into embedding vectors. To begin, we initialize the embedding weights with random values—this provides the starting point from which the model learns meaningful representations during training.

Conceptually, an embedding layer is equivalent to performing one-hot encoding of the tokens, followed by a matrix multiplication in a fully connected layer. However, instead of explicitly creating large, sparse one-hot vectors, the embedding layer directly performs an efficient lookup operation. This approach is computationally faster and uses less memory.

Since the embedding layer is simply a learnable weight matrix, it can be optimized through backpropagation just like any other neural network layer.




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

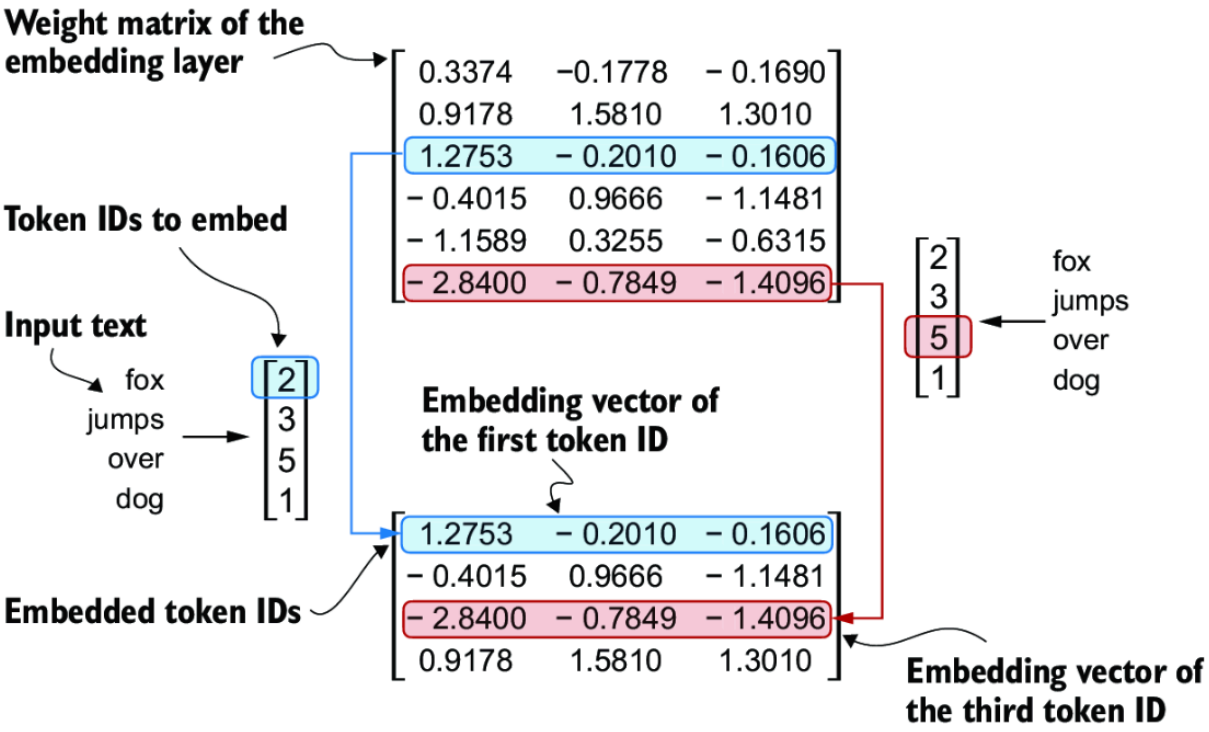


Fig: Embedding layers perform a lookup operation, retrieving the embedding vector corresponding to the token ID from the embedding layer's weight matrix. For instance, the embedding vector of the token ID 5 is the sixth row of the embedding layer weight matrix (it is the sixth instead of the fifth row because Python starts counting at 0).