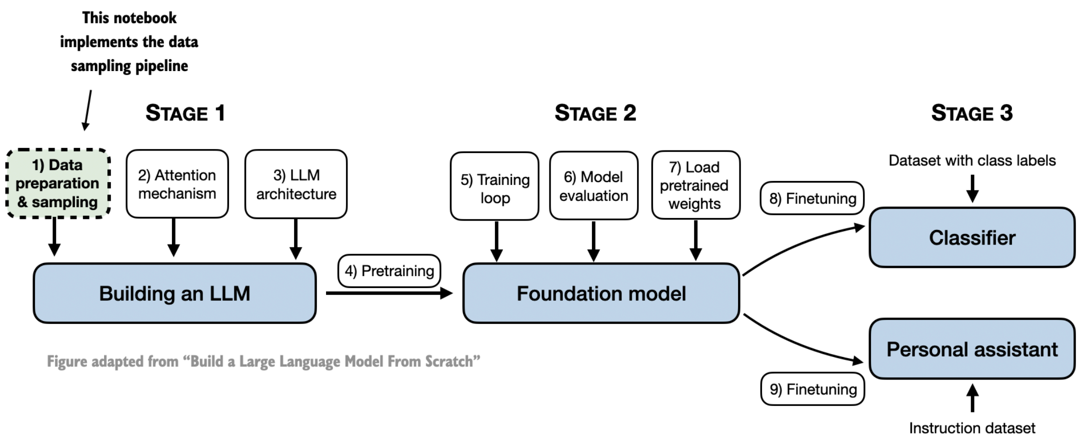
2. Understanding LLM Input Data

In [1]:
```
from importlib.metadata import version


print("torch version:", version("torch"))
print("tiktoken version:", version("tiktoken"))
```

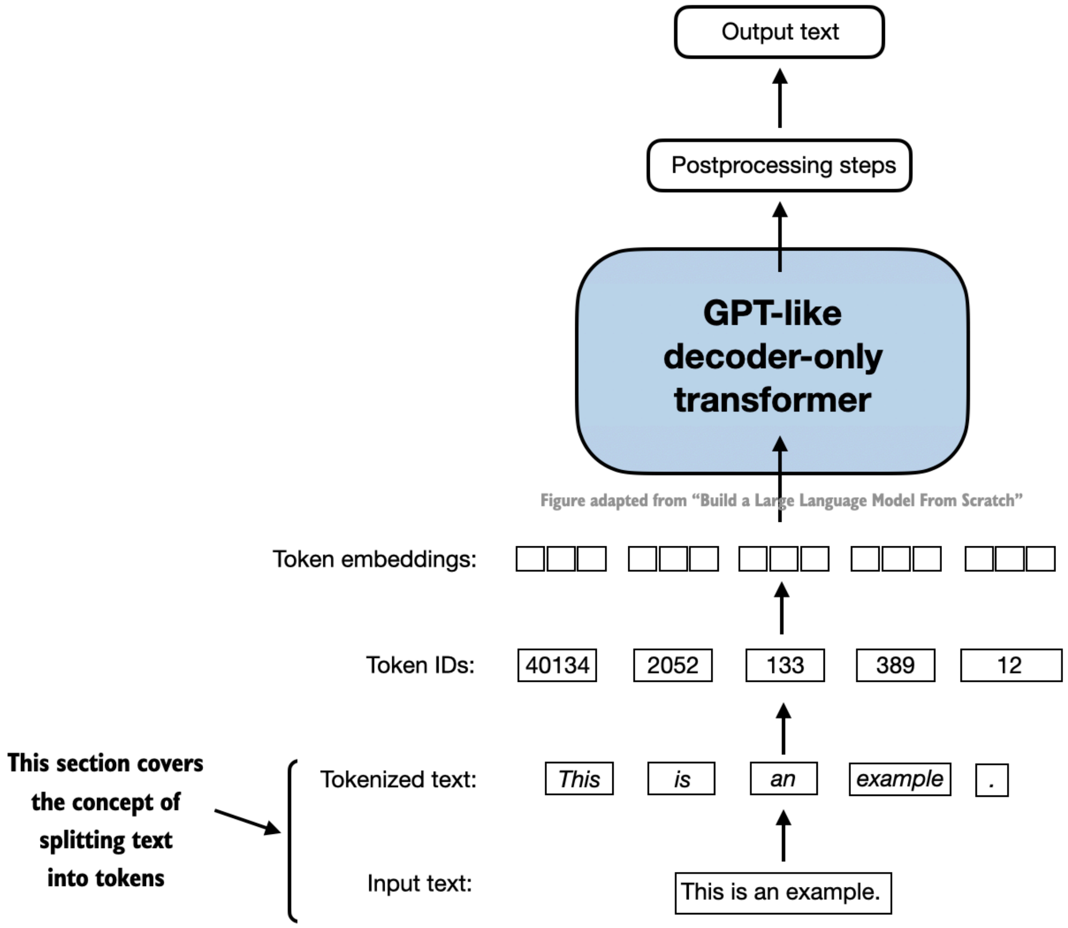
```
torch version: 2.8.0
tiktoken version: 0.11.0
```

- This notebook provides a brief overview of the data preparation and sampling procedures to get input data "ready" for an LLM
- Understanding what the input data looks like is a great first step towards understanding how LLMs work



2.1 Tokenizing text

- In this section, we tokenize text, which means breaking text into smaller units, such as individual words and punctuation characters



- Load raw text we want to work with
- The Verdict by Edith Wharton is a public domain short story

In [2]:
```
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])
```

```
Total number of character: 20479
I HAD always thought Jack Gisburn rather a cheap genius--though a good fel
low enough--so it was no
```

- The goal is to tokenize and embed this text for an LLM
- Let's develop a simple tokenizer based on some simple sample text that we can then later apply to the text above

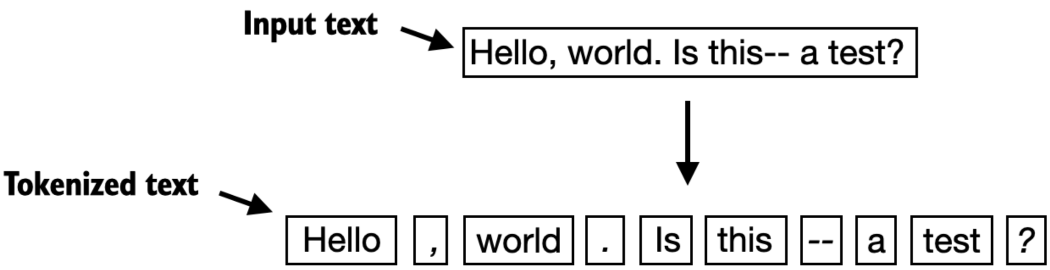


Figure adapted from “Build a Large Language Model From Scratch”

- The following regular expression will split on whitespaces and punctuation

In [3]:
```
import re

preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', raw_text)
print(preprocessed[:30])
```

```
['I', ' ', 'HAD', ' ', 'always', ' ', 'thought', ' ', 'Jack', ' ', 'Gisbur
n', ' ', 'rather', ' ', 'a', ' ', 'cheap', ' ', 'genius', '--', 'though',
' ', 'a', ' ', 'good', ' ', 'fellow', ' ', 'enough', '--']
```

In [4]:
```
print("Number of tokens:", len(preprocessed))
```

```
Number of tokens: 9235
```

In [5]:
```
len(set(preprocessed))
```

Out[5]:
```
1133
```

In [6]:
```
sorted(set(preprocessed))[:30]
```

```
Out[6]: ['',
 '\n',
 ' ',
 '!',
 '"',
 "'",
 '(',
 ')',
 ',',
 '--',
 '.',
 ':',
 ';',
 '?',
 'A',
 'Ah',
 'Among',
 'And',
 'Are',
 'Arrt',
 'As',
 'At',
 'Be',
 'Begin',
 'Burlington',
 'But',
 'By',
 'Carlo',
 'Chicago',
 'Claude']
```

2.2 Converting tokens into token IDs

- Next, we convert the text tokens into token IDs that we can process via embedding layers later
- For this we first need to build a vocabulary

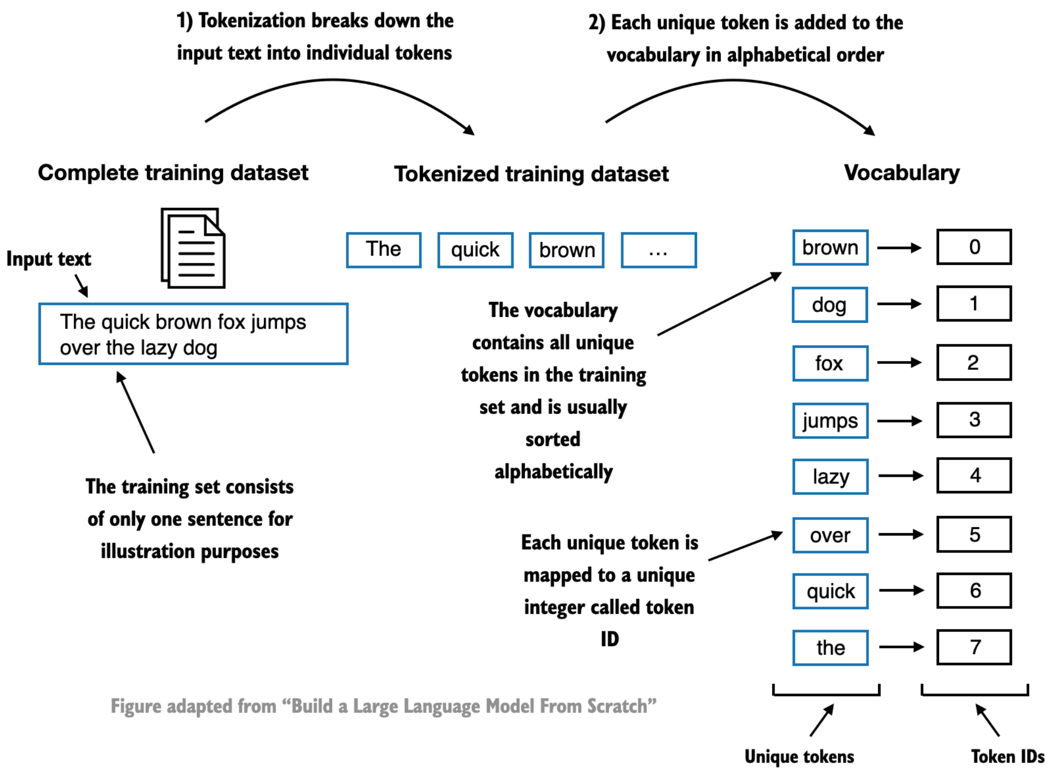


Figure adapted from "Build a Large Language Model From Scratch"

- The vocabulary contains the unique words in the input text

In [7]:
```
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)

print(vocab_size)
```

```
1133
```

In [8]:
```
for i in enumerate(all_words[:30]):
    print(i)
```

```
(0, '')
(1, '\n')
(2, ' ')
(3, '!')
(4, '"')
(5, "'")
(6, '(')
(7, ')')
(8, ',')
(9, '--')
(10, '.')
(11, ':')
(12, ';')
(13, '?')
(14, 'A')
(15, 'Ah')
(16, 'Among')
(17, 'And')
(18, 'Are')
(19, 'Arrt')
(20, 'As')
(21, 'At')
(22, 'Be')
(23, 'Begin')
(24, 'Burlington')
(25, 'But')
(26, 'By')
(27, 'Carlo')
(28, 'Chicago')
(29, 'Claude')
```

In [9]:

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

for i, item in enumerate(vocab.items()):
    print(item)
    if i > 28:
        break
```

```
('', 0)
('\n', 1)
(' ', 2)
('!', 3)
('"', 4)
("'", 5)
('(', 6)
(')', 7)
(',', 8)
('--', 9)
('.', 10)
(':', 11)
(';', 12)
('?', 13)
('A', 14)
('Ah', 15)
('Among', 16)
('And', 17)
('Are', 18)
('Arrt', 19)
('As', 20)
('At', 21)
('Be', 22)
('Begin', 23)
('Burlington', 24)
('But', 25)
('By', 26)
('Carlo', 27)
('Chicago', 28)
('Claude', 29)
```

- Below, we illustrate the tokenization of a short sample text using a small vocabulary:

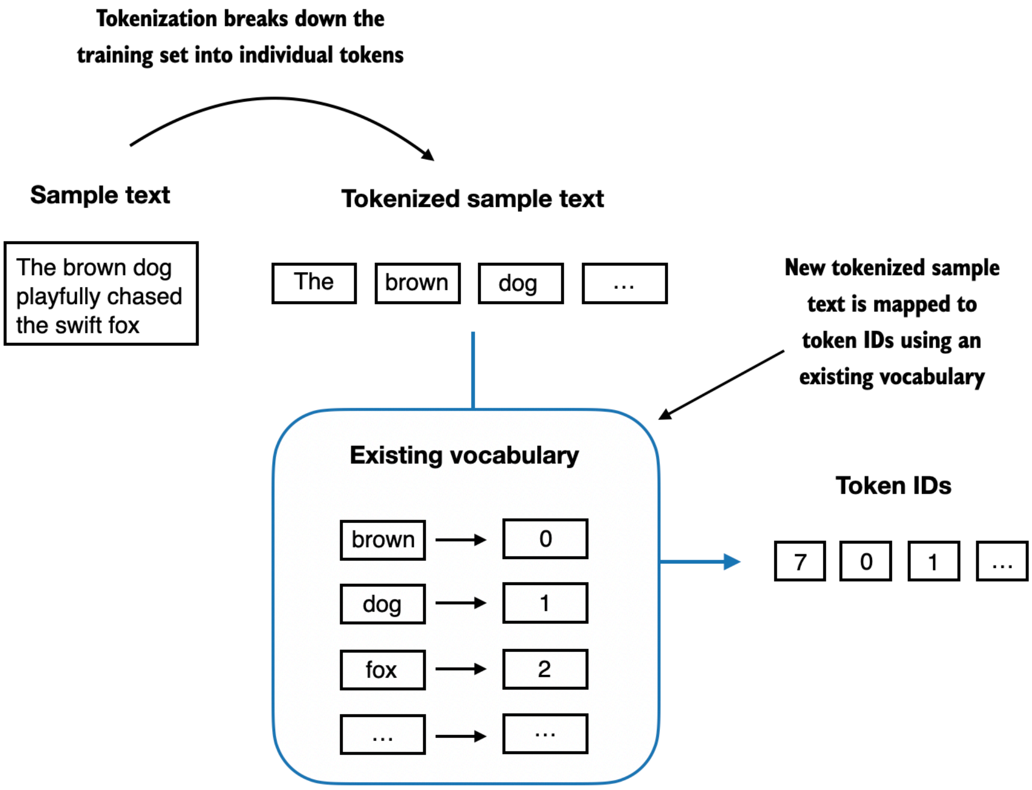


Figure adapted from "Build a Large Language Model From Scratch"

- Let's now put it all together into a tokenizer class

In [10]:
```python
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])
        # Replace spaces before the specified punctuations
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
        return text
```

- The `encode` function turns text into token IDs
- The `decode` function turns token IDs back into text

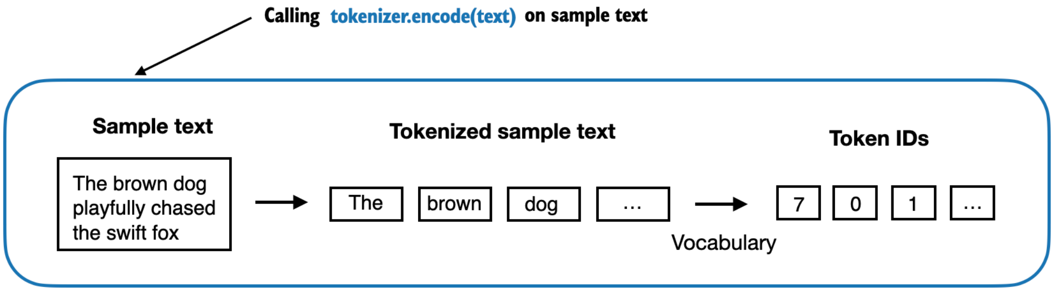
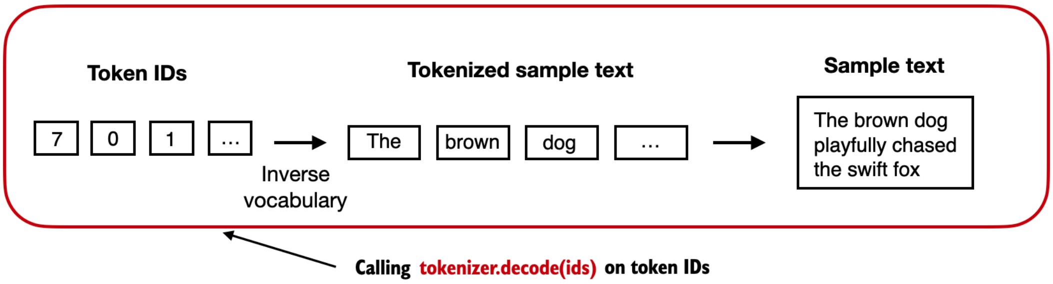


Figure adapted from "Build a Large Language Model From Scratch"



- We can use the tokenizer to encode (that is, tokenize) texts into integers
- These integers can then be embedded (later) as input of/for the LLM

In [11]:
```python
tokenizer = SimpleTokenizerV1(vocab)

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

```
[4, 59, 5, 853, 991, 605, 536, 749, 8, 1129, 599, 8, 4, 70, 10, 41, 854, 1
111, 757, 796, 10]
```

- We can decode the integers back into text

In [12]:
```python
tokenizer.decode(ids)
```

Out[12]:
```
'" It\' s the last he painted, you know," Mrs. Gisburn said with pardona
ble pride.'
```

In [13]:
```python
tokenizer.decode(tokenizer.encode(text))
```

Out[13]:
```
'" It\' s the last he painted, you know," Mrs. Gisburn said with pardona
ble pride.'
```

2.3 BytePair encoding

- GPT-2 used BytePair encoding (BPE) as its tokenizer
- it allows the model to break down words that aren't in its predefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words
- For instance, if GPT-2's vocabulary doesn't have the word "unfamiliarword," it might tokenize it as ["unfam", "iliar", "word"] or some other subword breakdown, depending on its trained BPE merges
- The original BPE tokenizer can be found here: https://github.com/openai/gpt-2/blob/master/src/encoder.py
- In this lecture, we are using the BPE tokenizer from OpenAI's open-source tiktoken library, which implements its core algorithms in Rust to improve computational performance
- (Based on an analysis here, I found that `tiktoken` is approx. 3x faster than the original tokenizer and 6x faster than an equivalent tokenizer in Hugging Face)

In [14]:
```python
# pip install tiktoken
```

In [15]:
```python
import importlib
import tiktoken

print("tiktoken version:", importlib.metadata.version("tiktoken"))
```

```
tiktoken version: 0.11.0
```

In [16]:
```python
tokenizer = tiktoken.get_encoding("gpt2")
```

In [17]:
```python
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, 88
12, 2114, 1659, 617, 34680, 27271, 13]
```

In [18]:
```
strings = tokenizer.decode(integers)

print(strings)
```

```
Hello, do you like tea? <|endoftext|> In the sunlit terracesof someunknown
Place.
```

In [19]:
```
# gpt 2 tokenizer can handle unknown words
tokenizer.encode("qwertyuiopasdfghjklzxcvbnm")
```

Out[19]:
```
[80, 15448, 774, 9019, 404, 292, 7568, 456, 73, 41582, 89, 25306, 85, 93
74, 76]
```

In [20]:
```
print(f"{tokenizer.decode([80])} {tokenizer.decode([15448])} {tokenizer.d
```

```
q wer ty ui
```

- BPE tokenizers break down unknown words into subwords and individual characters:

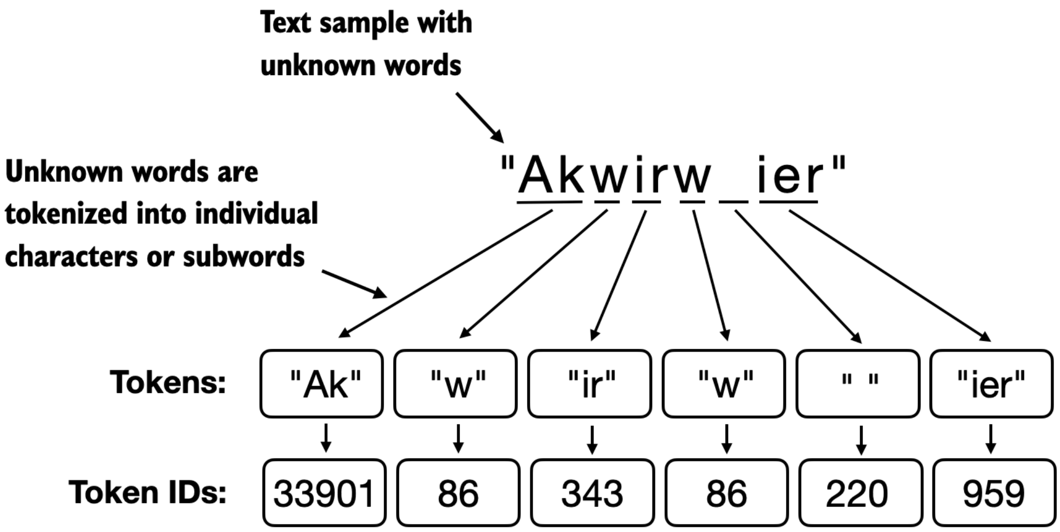


Figure adapted from "Build a Large Language Model From Scratch"

In [21]:
```
tokenizer.encode("Akwirw ier", allowed_special={"<|endoftext|>"})
```

Out[21]:
```
[33901, 86, 343, 86, 220, 959]
```

2.4 Data sampling with a sliding window

- Above, we took care of the tokenization (converting text into word tokens represented as token ID numbers)
- Now, let's talk about how we create the data loading for LLMs
- We train LLMs to generate one word at a time, so we want to prepare the training data accordingly where the next word in a sequence represents the target to predict

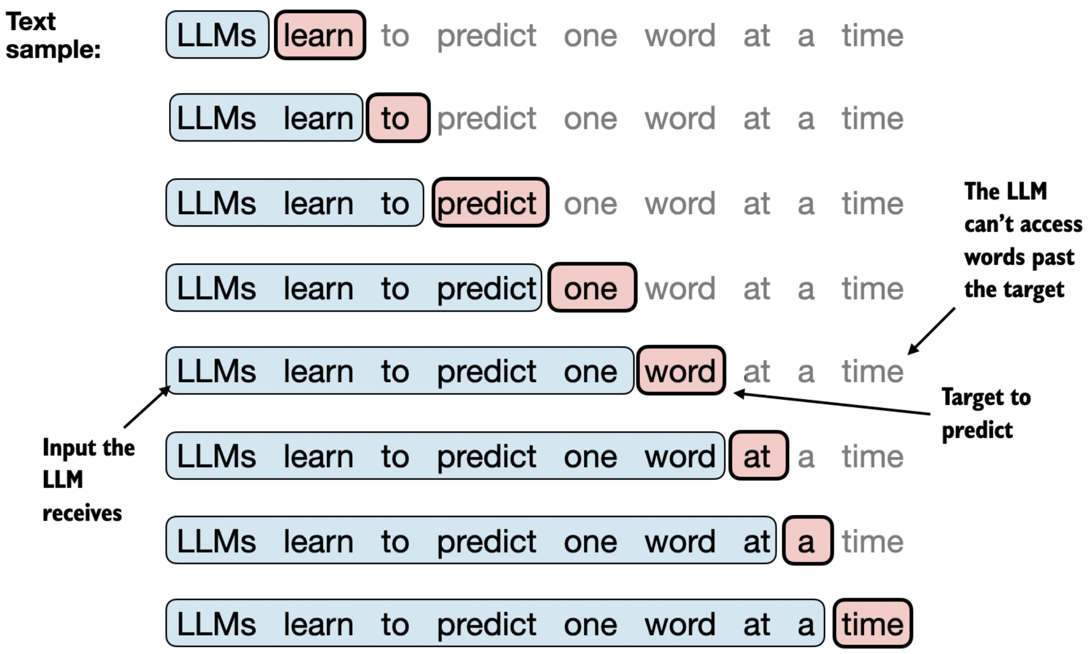


Figure adapted from "Build a Large Language Model From Scratch"

- For this, we use a sliding window approach, changing the position by +1:

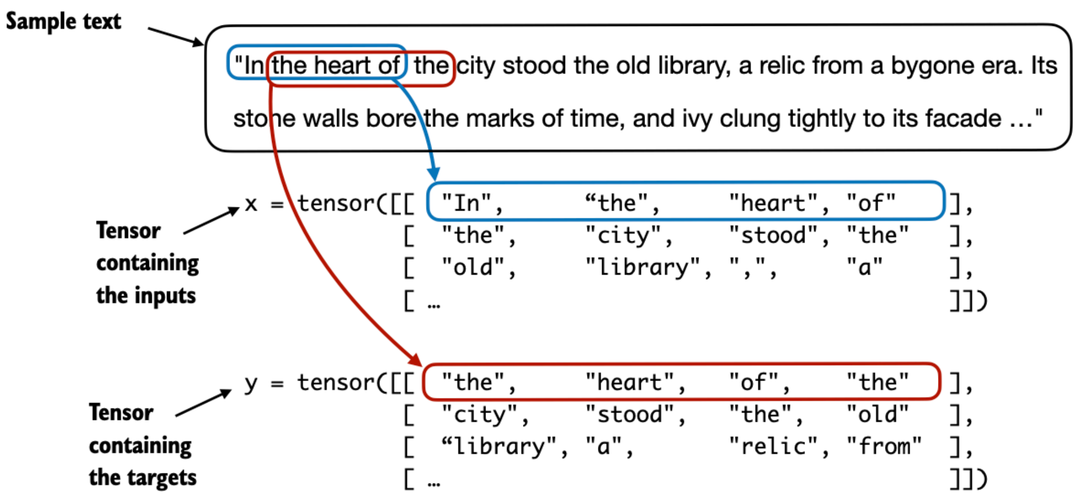


Figure adapted from "Build a Large Language Model From Scratch"

- Note that in practice it's best to set the stride equal to the context length so that we don't have overlaps between the inputs (the targets are still shifted by +1 always)

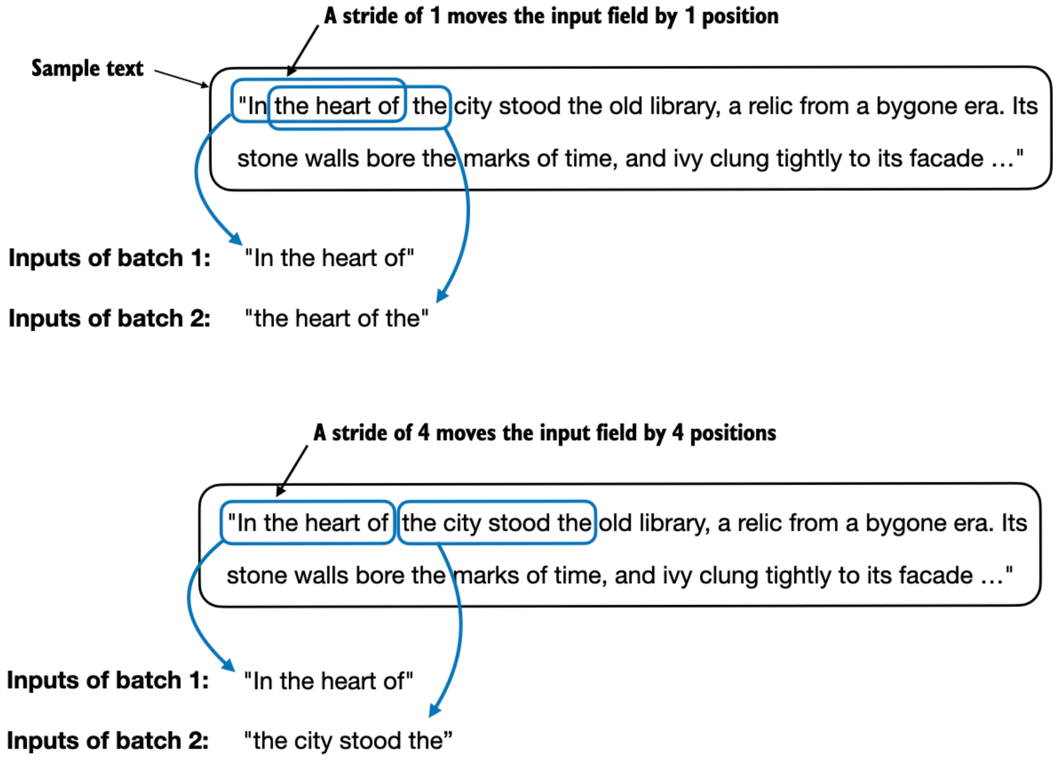


Figure adapted from "Build a Large Language Model From Scratch"

In [22]:
```
from supplementary import create_dataloader_v1


dataloader = create_dataloader_v1(raw_text, batch_size=8, max_length=4, s

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("Input Size: ", inputs.size())
print("\nTargets:\n", targets)
print("Target Size: ", targets.size())
```

```
Inputs:
 tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]])
Input Size:  torch.Size([8, 4])

Targets:
 tensor([[  367,  2885,  1464,  1807],
        [ 3619,   402,   271, 10899],
        [ 2138,   257,  7026, 15632],
        [  438,  2016,   257,   922],
        [ 5891,  1576,   438,   568],
        [  340,   373,   645,  1049],
        [ 5975,   284,   502,   284],
        [ 3285,   326,    11,   287]])
Target Size:  torch.Size([8, 4])
```

NEXT: Coding an LLM architecture