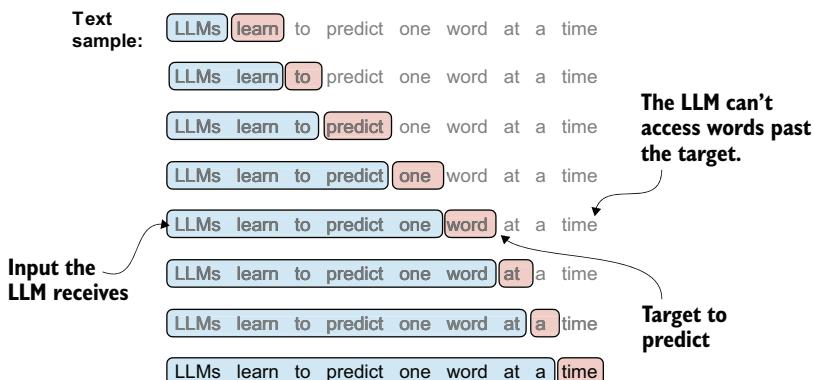


words like “define,” “depend,” “made,” and “hidden.” The merges are determined by a frequency cutoff.

## 2.6 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. What do these input–target pairs look like? As we already learned, LLMs are pretrained by predicting the next word in a text, as depicted in figure 2.12.



**Figure 2.12** 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.

Let’s implement a data loader that fetches the input–target pairs in figure 2.12 from the training dataset using a sliding window approach. 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))
```

Executing this code will return 5145, the total number of tokens in the training set, after applying the BPE tokenizer.

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 next-word prediction task is to create two variables, `x` and `y`, where `x` contains the input tokens and `y` contains the targets, which are the inputs shifted by 1:

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

**The context size determines how many tokens are included in the input.**

Running the previous code prints the following output:

```
x: [290, 4920, 2241, 287]
y: [4920, 2241, 287, 257]
```

By processing the inputs along with the targets, which are the inputs shifted by one position, we can create the next-word prediction tasks (see figure 2.12), as follows:

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

The code prints

```
[290] ----> 4920
[290, 4920] ----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287] ----> 257
```

Everything left of the arrow (---->) refers to the input an LLM would receive, and the token ID on the right side of the arrow represents the target token ID that the LLM is supposed to predict. 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]))
```

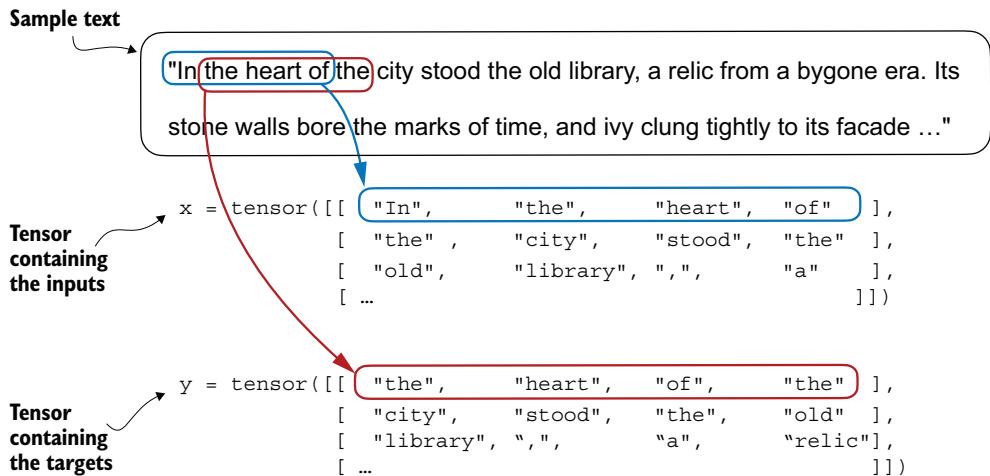
The following outputs show how the input and outputs look in text format:

```
and ----> established
and established ----> himself
and established himself ----> in
and established himself in ----> a
```

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 the

inputs and targets as PyTorch tensors, which can be thought of as multidimensional arrays. In particular, we are interested in returning two tensors: an input tensor containing the text that the LLM sees and a target tensor that includes the targets for the LLM to predict, as depicted in figure 2.13. While the figure shows the tokens in string format for illustration purposes, the code implementation will operate on token IDs directly since the `encode` method of the BPE tokenizer performs both tokenization and conversion into token IDs as a single step.



**Figure 2.13** 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.

**NOTE** For the efficient data loader implementation, we will use PyTorch’s built-in `Dataset` and `DataLoader` classes. For additional information and guidance on installing PyTorch, please see section A.2.1.3 in appendix A.

The code for the dataset class is shown in the following listing.

#### Listing 2.5 A dataset for batched inputs and targets

```
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) ← Tokenizes the entire text
```