

```
[ 1049,  5975,   284,   502],
[ 284,  3285,   326,    11]])

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]])
```

Note that we increase the stride to 4 to utilize the data set 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 last step in preparing the input text for LLM training is to convert the token IDs into embedding vectors, as shown in figure 2.15. As a preliminary step, we must initialize

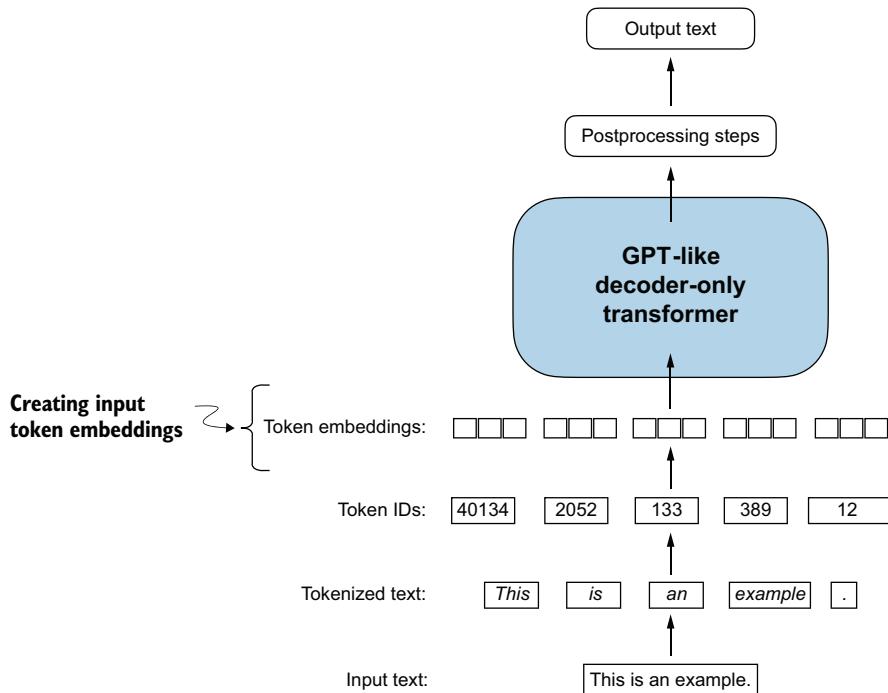


Figure 2.15 Preparation involves tokenizing text, converting text tokens to token IDs, and converting token IDs into embedding vectors. Here, we consider the previously created token IDs to create the token embedding vectors.

these embedding weights with random values. This initialization serves as the starting point for the LLM’s learning process. In chapter 5, we will optimize the embedding weights as part of the LLM training.

A continuous vector representation, or embedding, is necessary since GPT-like LLMs are deep neural networks trained with the backpropagation algorithm.

NOTE If you are unfamiliar with how neural networks are trained with backpropagation, please read section A.4 in appendix A.

Let’s see how the token ID to embedding vector conversion works with a hands-on example. Suppose we have the following four input tokens with IDs 2, 3, 5, and 1:

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

For the sake of simplicity, suppose we have a small vocabulary of only 6 words (instead of the 50,257 words in the BPE tokenizer vocabulary), and we want to create embeddings of size 3 (in GPT-3, the embedding size is 12,288 dimensions):

```
vocab_size = 6
output_dim = 3
```

Using the `vocab_size` and `output_dim`, we can instantiate an embedding layer in PyTorch, setting the random seed to 123 for reproducibility purposes:

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

The print statement prints the embedding layer’s underlying weight matrix:

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
       [ 0.9178,  1.5810,  1.3010],
       [ 1.2753, -0.2010, -0.1606],
       [-0.4015,  0.9666, -1.1481],
       [-1.1589,  0.3255, -0.6315],
       [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

The weight matrix of the embedding layer contains small, random values. These values are optimized during LLM training as part of the LLM optimization itself. Moreover, we can see that the weight matrix has six rows and three columns. There is one row for each of the six possible tokens in the vocabulary, and there is one column for each of the three embedding dimensions.

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

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

The returned embedding vector is

```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0>)
```

If we compare the embedding vector for token ID 3 to the previous embedding matrix, we see that it is identical to the fourth row (Python starts with a zero index, so it's the row corresponding to index 3). In other words, the embedding layer is essentially a lookup operation that retrieves rows from the embedding layer's weight matrix via a token ID.

NOTE For those who are familiar with one-hot encoding, the embedding layer approach described here is essentially just a more efficient way of implementing one-hot encoding followed by matrix multiplication in a fully connected layer, which is illustrated in the supplementary code on GitHub at <https://mng.bz/ZEB5>. Because the embedding layer is just a more efficient implementation equivalent to the one-hot encoding and matrix-multiplication approach, it can be seen as a neural network layer that can be optimized via backpropagation.

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 (`torch.tensor([2, 3, 5, 1])`):

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

The print output reveals that this results in a 4×3 matrix:

```
tensor([[ 1.2753, -0.2010, -0.1606],
       [-0.4015,  0.9666, -1.1481],
       [-2.8400, -0.7849, -1.4096],
       [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0>)
```

Each row in this output matrix is obtained via a lookup operation from the embedding weight matrix, as illustrated in figure 2.16.

Having now created embedding vectors from token IDs, next we'll add a small modification to these embedding vectors to encode positional information about a token within a text.

2.8 Encoding word positions

In principle, token embeddings are a suitable input for an LLM. However, a minor shortcoming of LLMs is that their self-attention mechanism (see chapter 3) doesn't have a notion of position or order for the tokens within a sequence. The way the previously introduced embedding layer works is that the same token ID always gets mapped to the same vector representation, regardless of where the token ID is positioned in the input sequence, as shown in figure 2.17.