The print function call returns

```
torch.Size([8, 4, 256])
```

The 8 × 4 × 256–dimensional tensor output shows that each token ID is now embedded as a 256-dimensional vector.

For a GPT model's absolute embedding approach, we just need to create another embedding layer that has the same embedding dimension as the `token_embedding_layer`:

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
pos_embeddings = pos_embedding_layer(torch.arange(context_length))
print(pos_embeddings.shape)
```

The input to the `pos_embeddings` is usually a placeholder vector `torch.arange(context_length)`, which contains a sequence of numbers 0, 1, ..., up to the maximum input length –1. The `context_length` is a variable that represents the supported input size of the LLM. Here, we choose it similar to the maximum length of the input text. In practice, input text can be longer than the supported context length, in which case we have to truncate the text.

The output of the print statement is

```
torch.Size([4, 256])
```

As we can see, the positional embedding tensor consists of four 256-dimensional vectors. We can now add these directly to the token embeddings, where PyTorch will add the 4 × 256–dimensional `pos_embeddings` tensor to each 4 × 256–dimensional token embedding tensor in each of the eight batches:

```
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)
```

The print output is

```
torch.Size([8, 4, 256])
```

The `input_embeddings` we created, as summarized in figure 2.19, are the embedded input examples that can now be processed by the main LLM modules, which we will begin implementing in the next chapter.

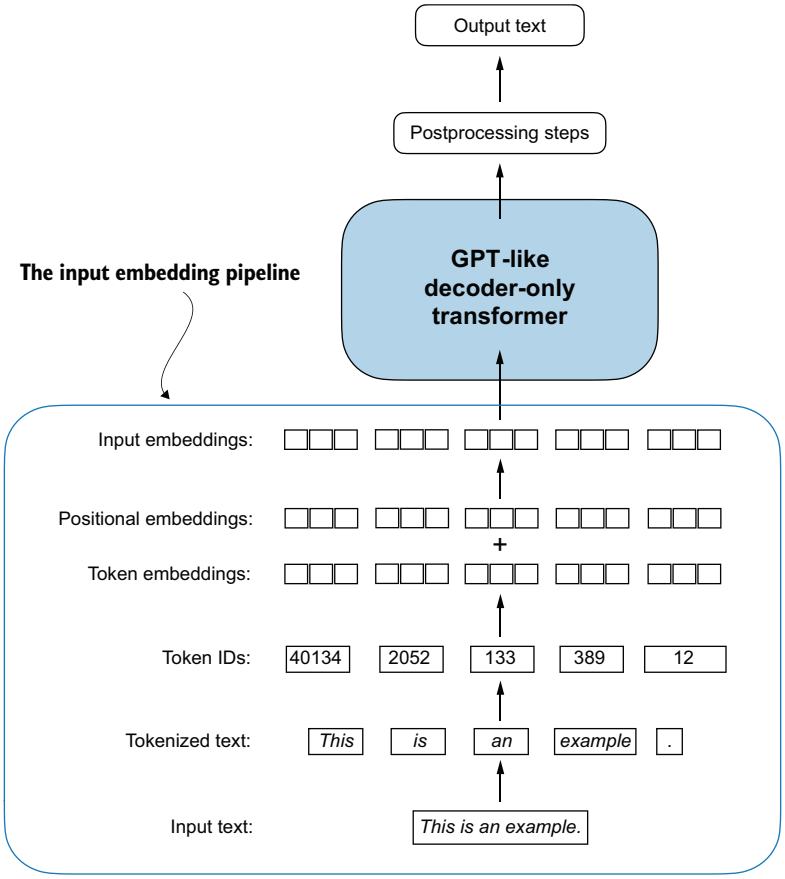

Figure 2.19 As part of the input processing pipeline, input text is first broken up into individual tokens. These tokens are then converted into token IDs using a vocabulary. The token IDs are converted into embedding vectors to which positional embeddings of a similar size are added, resulting in input embeddings that are used as input for the main LLM layers.

Summary

- LLMs require textual data to be converted into numerical vectors, known as embeddings, since they can't process raw text. Embeddings transform discrete data (like words or images) into continuous vector spaces, making them compatible with neural network operations.
- As the first step, raw text is broken into tokens, which can be words or characters. Then, the tokens are converted into integer representations, termed token IDs.
- Special tokens, such as `<|unk|>` and `<|endoftext|>`, can be added to enhance the model's understanding and handle various contexts, such as unknown words or marking the boundary between unrelated texts.

- The byte pair encoding (BPE) tokenizer used for LLMs like GPT-2 and GPT-3 can efficiently handle unknown words by breaking them down into subword units or individual characters.
- We use a sliding window approach on tokenized data to generate input–target pairs for LLM training.
- Embedding layers in PyTorch function as a lookup operation, retrieving vectors corresponding to token IDs. The resulting embedding vectors provide continuous representations of tokens, which is crucial for training deep learning models like LLMs.
- While token embeddings provide consistent vector representations for each token, they lack a sense of the token's position in a sequence. To rectify this, two main types of positional embeddings exist: absolute and relative. OpenAI's GPT models utilize absolute positional embeddings, which are added to the token embedding vectors and are optimized during the model training.