```
    encoded_tensor = torch.tensor(encoded).unsqueeze(0)
    return encoded_tensor

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0)
    return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

.unsqueeze(0) adds the batch dimension

Removes batch dimension

Using this code, the `model` generates the following text:

```
Output text:
 Every effort moves you rentingetic wasn? refres RexMeCHicular stren
```

Clearly, the model isn't yet producing coherent text because it hasn't undergone training. To define what makes text "coherent" or "high quality," we have to implement a numerical method to evaluate the generated content. This approach will enable us to monitor and enhance the model's performance throughout its training process.

Next, we will calculate a *loss metric* for the generated outputs. This loss serves as a progress and success indicator of the training progress. Furthermore, in later chapters, when we fine-tune our LLM, we will review additional methodologies for assessing model quality.

5.1.2 Calculating the text generation loss

Next, let's explore techniques for numerically assessing text quality generated during training by calculating a *text generation loss*. We will go over this topic step by step with a practical example to make the concepts clear and applicable, beginning with a short recap of how the data is loaded and how the text is generated via the `generate_text_simple` function.

Figure 5.4 illustrates the overall flow from input text to LLM-generated text using a five-step procedure. This text-generation process shows what the `generate_text_simple` function does internally. We need to perform these same initial steps before we can compute a loss that measures the generated text quality later in this section.

Figure 5.4 outlines the text generation process with a small seven-token vocabulary to fit this image on a single page. However, our `GPTModel` works with a much larger

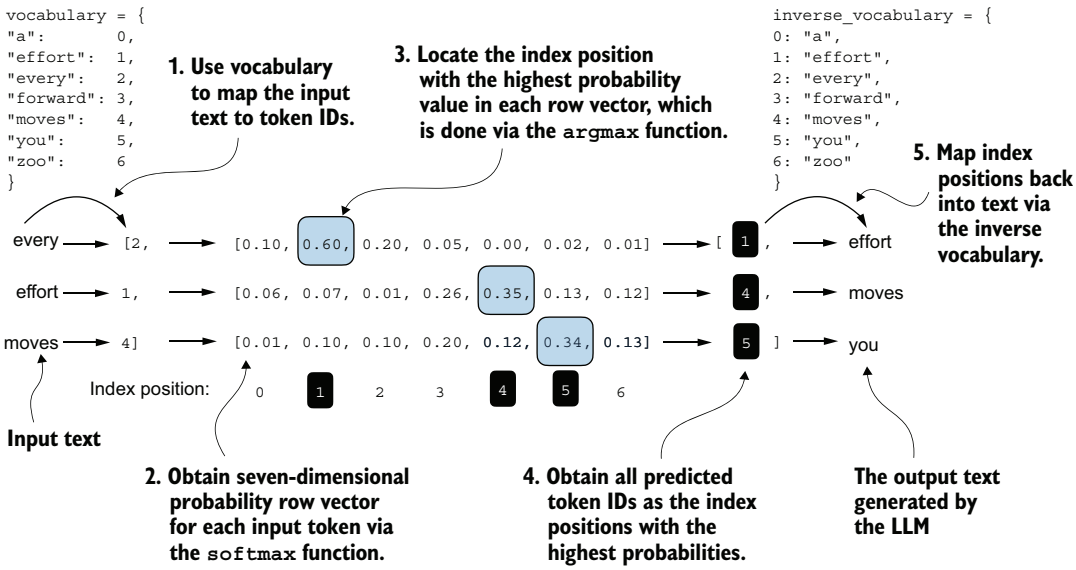

Figure 5.4 For each of the three input tokens, shown on the left, we compute a vector containing probability scores corresponding to each token in the vocabulary. The index position of the highest probability score in each vector represents the most likely next token ID. These token IDs associated with the highest probability scores are selected and mapped back into a text that represents the text generated by the model.

vocabulary consisting of 50,257 words; hence, the token IDs in the following code will range from 0 to 50,256 rather than 0 to 6.

Also, figure 5.4 only shows a single text example (`"every effort moves"`) for simplicity. In the following hands-on code example that implements the steps in the figure, we will work with two input examples for the GPT model (`"every effort moves"` and `"I really like"`).

Consider these two input examples, which have already been mapped to token IDs (figure 5.4, step 1):

```
inputs = torch.tensor([[16833, 3626, 6100],   # ["every effort moves",
                       [40,    1107, 588]])   #  "I really like"]
```

Matching these inputs, the `targets` contain the token IDs we want the model to produce:

```
targets = torch.tensor([[3626, 6100, 345  ],  # [" effort moves you",
                        [1107, 588, 11311]])  #  " really like chocolate"]
```

Note that the targets are the inputs but shifted one position forward, a concept we covered in chapter 2 during the implementation of the data loader. This shifting strategy is crucial for teaching the model to predict the next token in a sequence.

Now we feed the inputs into the model to calculate logits vectors for the two input examples, each comprising three tokens. Then we apply the `softmax` function to transform these logits into probability scores (`probas`; figure 5.4, step 2):

```
with torch.no_grad():                          <-- Disables gradient tracking
    logits = model(inputs)                         since we are not training yet
probas = torch.softmax(logits, dim=-1)         <-- Probability of each
print(probas.shape)                                token in vocabulary
```

The resulting tensor dimension of the probability score (`probas`) tensor is

```
torch.Size([2, 3, 50257])
```

The first number, 2, corresponds to the two examples (rows) in the inputs, also known as batch size. The second number, 3, corresponds to the number of tokens in each input (row). Finally, the last number corresponds to the embedding dimensionality, which is determined by the vocabulary size. Following the conversion from logits to probabilities via the `softmax` function, the `generate_text_simple` function then converts the resulting probability scores back into text (figure 5.4, steps 3–5).

We can complete steps 3 and 4 by applying the `argmax` function to the probability scores to obtain the corresponding token IDs:

```
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)
```

Given that we have two input batches, each containing three tokens, applying the `argmax` function to the probability scores (figure 5.4, step 3) yields two sets of outputs, each with three predicted token IDs:

```
Token IDs:
 tensor([[[16657],        <-- First batch
          [  339],
          [42826]],
         [[49906],        <-- Second batch
          [29669],
          [41751]]])
```

Finally, step 5 converts the token IDs back into text:

```
print(f"Targets batch 1: {token_ids_to_text(targets[0], tokenizer)}")
print(f"Outputs batch 1:"
      f" {token_ids_to_text(token_ids[0].flatten(), tokenizer)}")
```

When we decode these tokens, we find that these output tokens are quite different from the target tokens we want the model to generate:

```
Targets batch 1:  effort moves you
Outputs batch 1:  Armed heNetflix
```