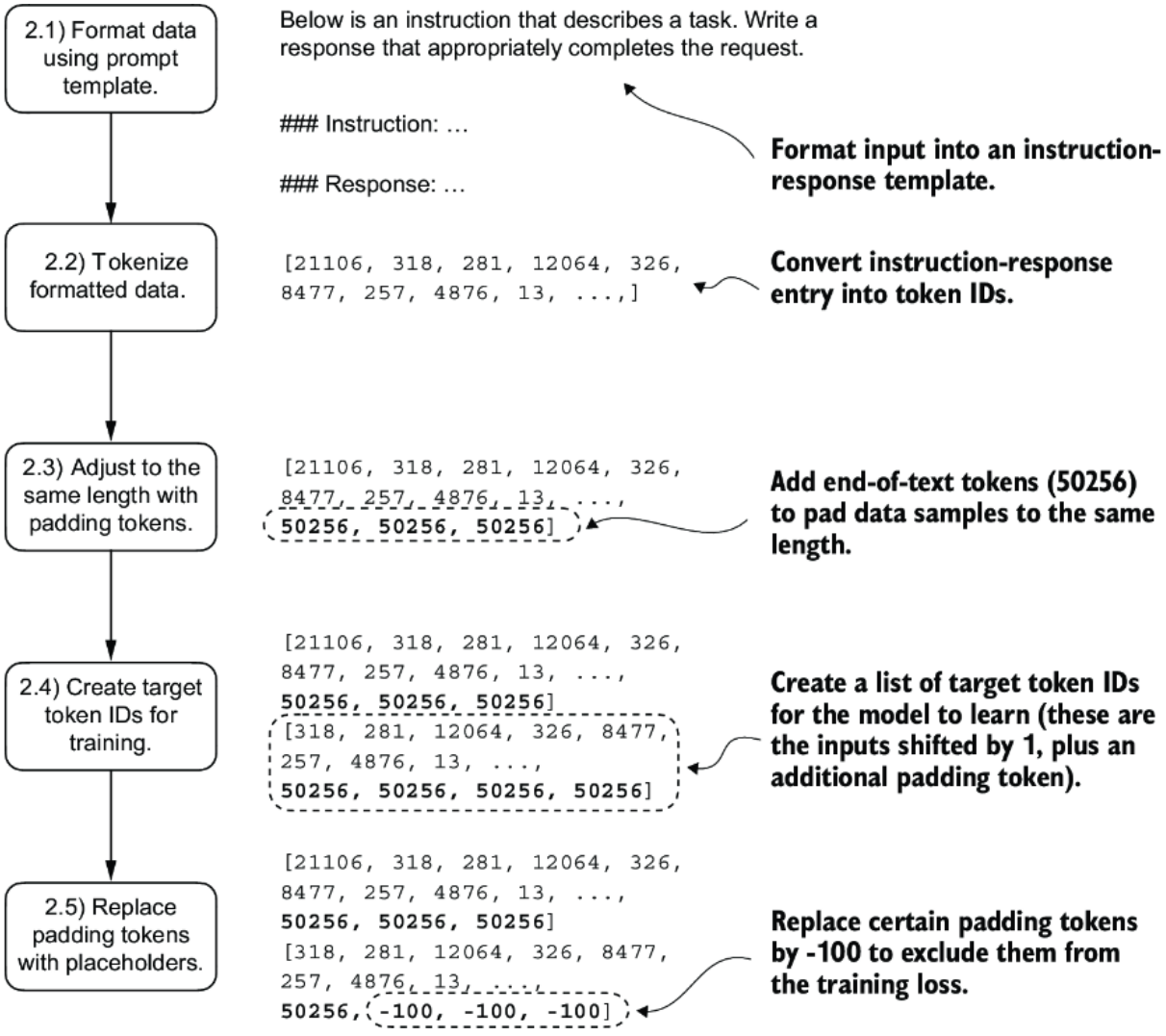
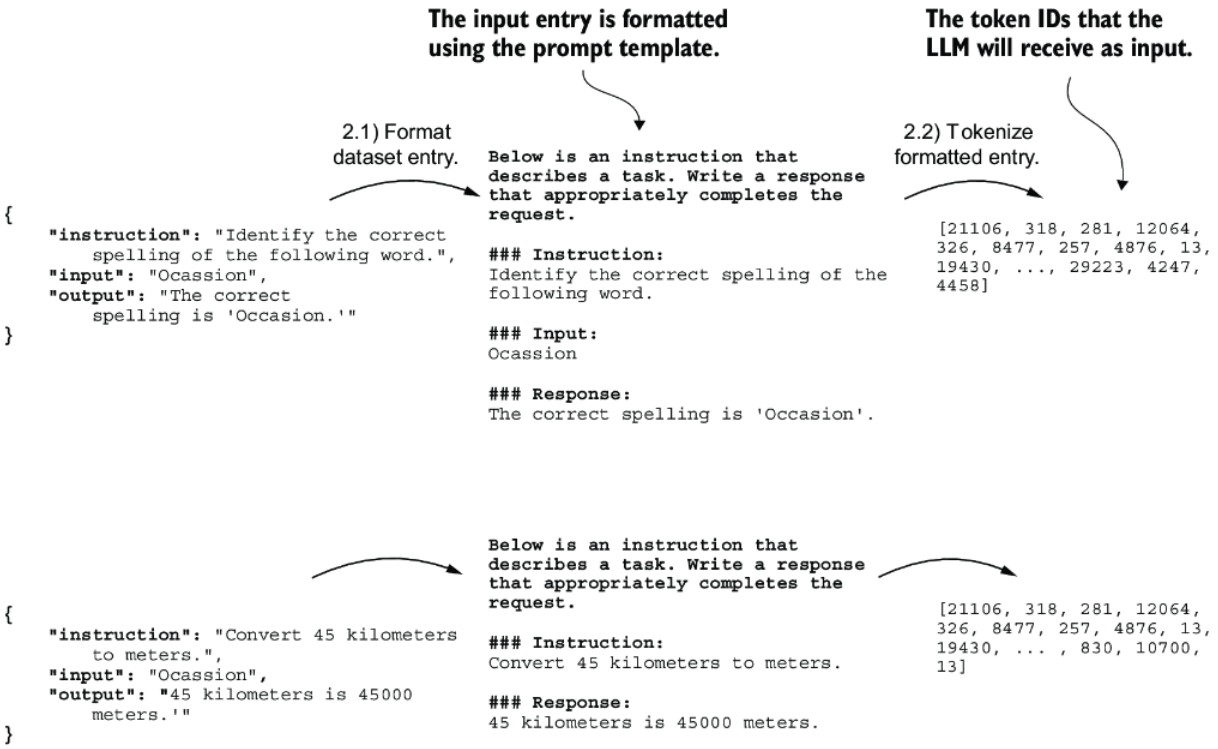
chapter7

The five substeps involved in implementing the batching process: (2.1) applying the prompt template, (2.2) using tokenization from previous chapters, (2.3) adding padding tokens, (2.4) creating target token IDs, and (2.5) replacing `-100` placeholder tokens to mask padding tokens in the loss function.



Let's tackle the *batching process* in several steps, including coding the custom collate function, as illustrated in figure 7.6. First, to implement steps 2.1 and 2.2, we code an `InstructionDataset` class that applies `format_input` and *pretokenizes* all inputs in the dataset, similar to the `SpamDataset` in chapter 6. This two-step process, detailed in figure 7.7, is implemented in the `__init__` constructor method of the `InstructionDataset`.

The first two steps involved in implementing the batching process. Entries are first formatted using a specific prompt template (2.1) and then tokenized (2.2), resulting in a sequence of token IDs that the model can process.



we want to accelerate training by collecting multiple training examples in a batch, which necessitates padding all inputs to a similar length. As with classification fine-tuning, we use the `<|endoftext|>` token as a padding token.

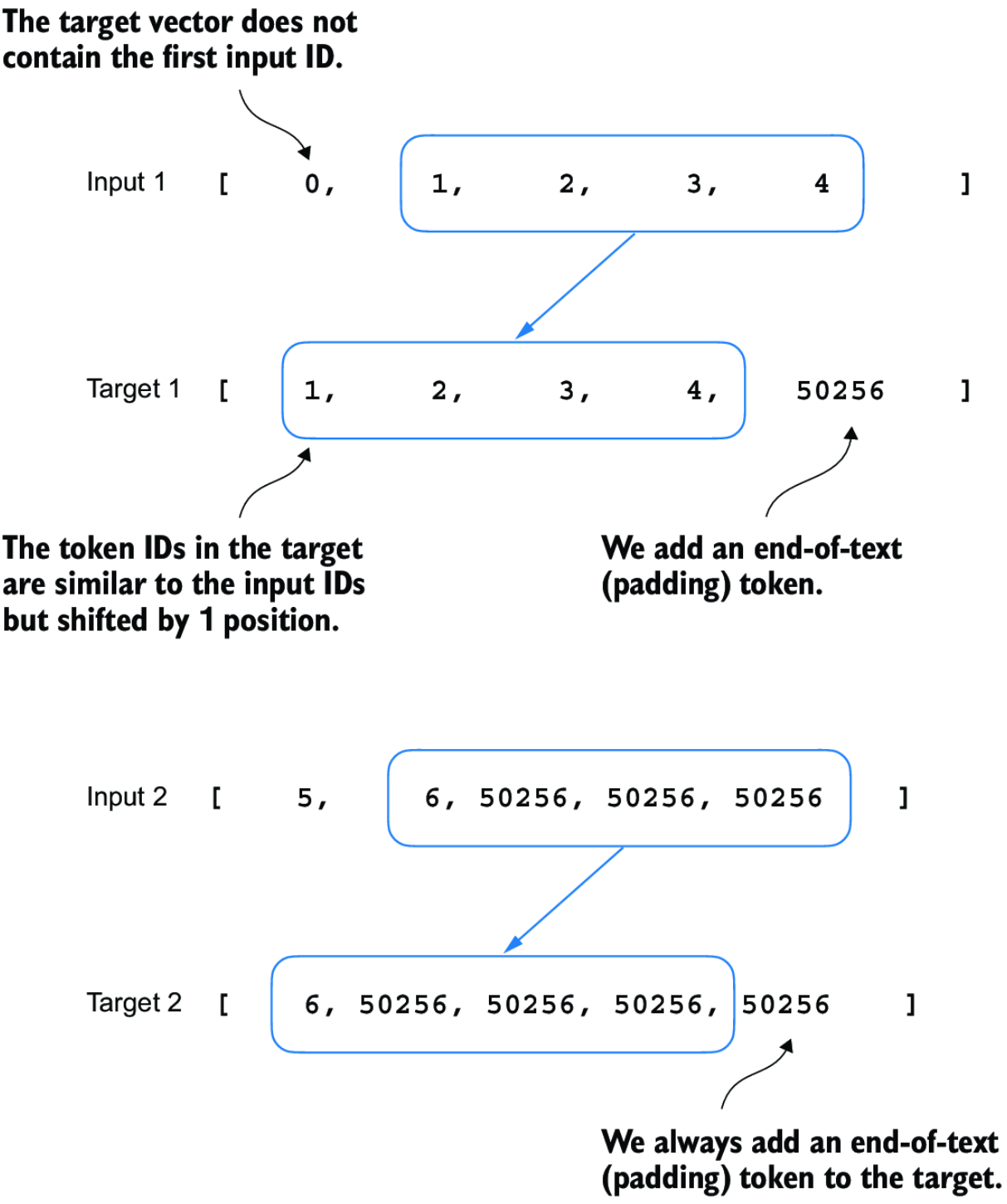
Instead of appending the `<|endoftext|>` tokens to the text inputs, we can append the token ID corresponding to `<|endoftext|>` to the pretokenized inputs directly. We can use the tokenizer's `.encode` method on an `<|endoftext|>` token to remind us which token ID we should use:

Moving on to step 2.3 of the process (see figure 7.6), we adopt a more sophisticated approach by developing a custom collate function that we can pass to the data loader. This custom collate function pads the training examples in each batch to the same length while allowing different batches to have different lengths, as demonstrated in figure 7.8. This approach minimizes unnecessary padding by only extending sequences to match the longest one in each batch, not the whole dataset.

focusing on step 2.4, the creation of target token IDs. This step is essential as it enables the model to learn and predict the tokens it needs to generate.

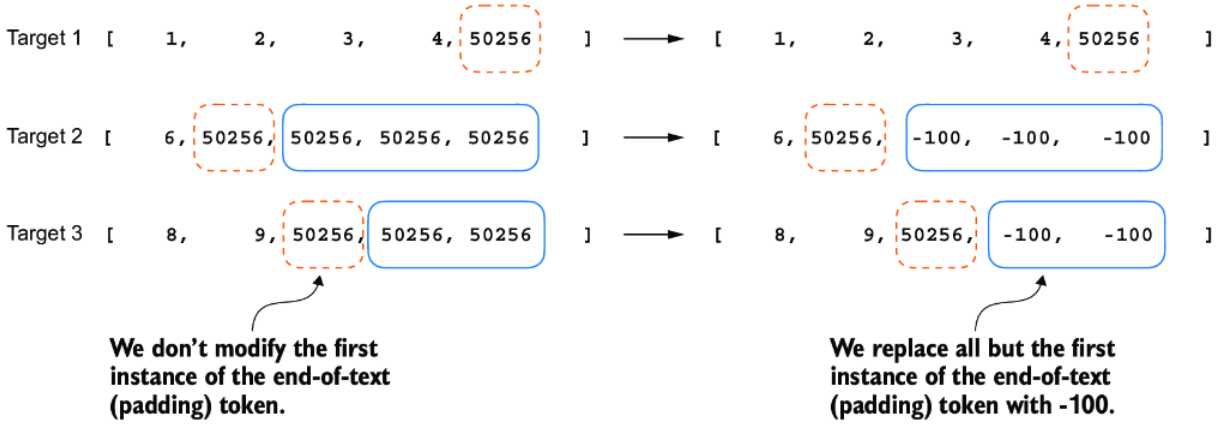
Similar to the process we used to pretrain an LLM, the target token IDs match the input token IDs but are shifted one position to the right. This setup, as shown in figure 7.10, allows the LLM to learn how to predict the next token in a sequence.

The input and target token alignment used in the instruction fine-tuning process of an LLM. For each input sequence, the corresponding target sequence is created by shifting the token IDs one position to the right, omitting the first token of the input, and appending an end-of-text token.



In the next step, we assign a `-100` placeholder value to all padding tokens, as highlighted in figure 7.11. This special value allows us to exclude these padding tokens from contributing to the training loss calculation, ensuring that only meaningful data influences model learning. We will discuss this process in more detail after we implement this modification. (When fine-tuning for classification, we did not have to worry about this since we only trained the model based on the last output token.)

However, note that we retain one end-of-text token, ID `50256`, in the target list, as depicted in figure 7.12. Retaining it allows the LLM to learn when to generate an end-of-text token in response to instructions, which we use as an indicator that the generated response is complete.



In the following listing, we modify our custom collate function to replace tokens with ID `50256` with `-100` in the target lists. Additionally, we introduce an `allowed_max_length` parameter to optionally limit the length of the samples. This adjustment will be useful if you plan to work with your own datasets that exceed the 1,024-token context size supported by the GPT-2 model.

As of this writing, researchers are divided on whether masking the instructions is universally beneficial during instruction fine-tuning. For instance, the 2024 paper by Shi et al., "Instruction Tuning With Loss Over Instructions" (https://arxiv.org/abs/2405.14394), demonstrated that not masking the instructions benefits the LLM performance (see appendix B for more details). Here, we will not apply masking and leave it as an optional exercise for interested readers.

Creating data loaders for an instruction dataset

We have completed several stages to implement an `InstructionDataset` class and a `custom_collate_fn` function for the instruction dataset. As shown in figure 7.14, we are ready to reap the fruits of our labor by simply plugging both `InstructionDataset` objects and the `custom_collate_fn` function into PyTorch data loaders. These loaders will automatically shuffle and organize the batches for the LLM instruction fine-tuning process.

Next, to reuse the chosen device setting in `custom_collate_fn` when we plug it into the PyTorch `DataLoader` class, we use the `partial` function from Python's `functools` standard library to create a new version of the function

with the device argument prefilled. Additionally, we set the `allowed_max_length` to `1024`, which truncates the data to the maximum context length supported by the GPT-2 model, which we will fine-tune later:

Next, we can set up the data loaders as we did previously, but this time, we will use our custom collate function for the batching process.