The content of the example entry is

```
Example entry:
 {'instruction': 'Identify the correct spelling of the following word.',
  'input': 'Ocassion', 'output': "The correct spelling is 'Occasion.'"}
```

As we can see, the example entries are Python dictionary objects containing an `'instruction'`, `'input'`, and `'output'`. Let's take a look at another example:

```
print("Another example entry:\n", data[999])
```

Based on the contents of this entry, the `'input'` field may occasionally be empty:

```
Another example entry:
 {'instruction': "What is an antonym of 'complicated'?",
  'input': '',
  'output': "An antonym of 'complicated' is 'simple'."}
```

Instruction fine-tuning involves training a model on a dataset where the input-output pairs, like those we extracted from the JSON file, are explicitly provided. There are various methods to format these entries for LLMs. Figure 7.4 illustrates two different

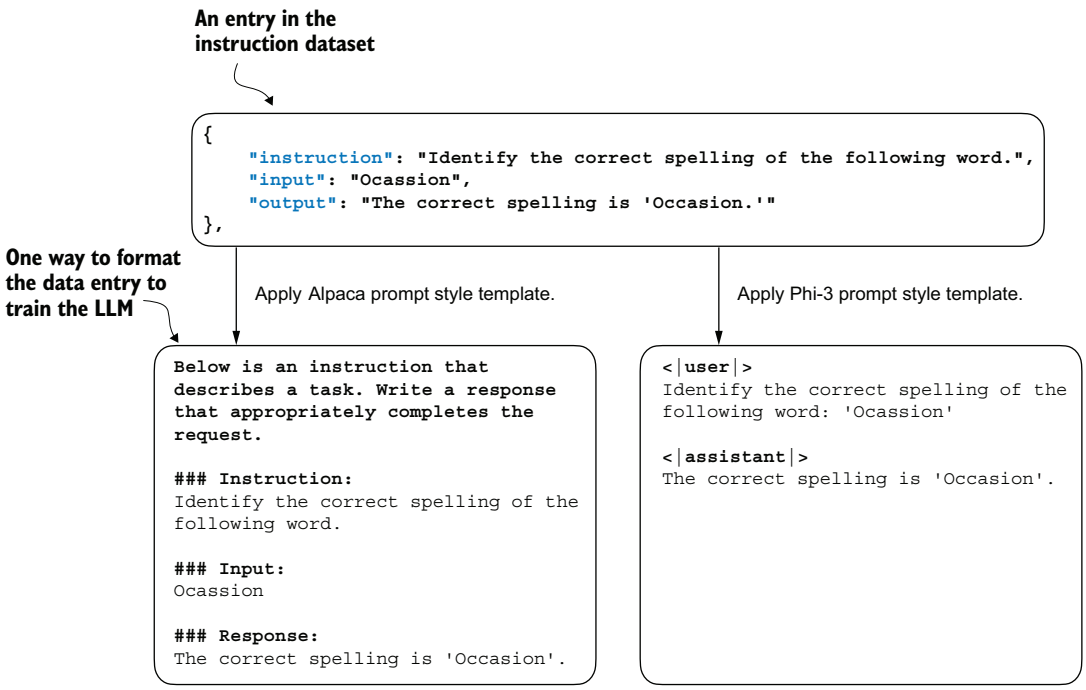


Figure 7.4 Comparison of prompt styles for instruction fine-tuning in LLMs. The Alpaca style (left) uses a structured format with defined sections for instruction, input, and response, while the Phi-3 style (right) employs a simpler format with designated <|user|> and <|assistant|> tokens.

example formats, often referred to as *prompt styles*, used in the training of notable LLMs such as Alpaca and Phi-3.

Alpaca was one of the early LLMs to publicly detail its instruction fine-tuning process. Phi-3, developed by Microsoft, is included to demonstrate the diversity in prompt styles. The rest of this chapter uses the Alpaca prompt style since it is one of the most popular ones, largely because it helped define the original approach to fine-tuning.

Exercise 7.1 Changing prompt styles

After fine-tuning the model with the Alpaca prompt style, try the Phi-3 prompt style shown in figure 7.4 and observe whether it affects the response quality of the model.

Let's define a `format_input` function that we can use to convert the entries in the `data` list into the Alpaca-style input format.

Listing 7.2 Implementing the prompt formatting function

```
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

This `format_input` function takes a dictionary `entry` as input and constructs a formatted string. Let's test it to dataset entry `data[50]`, which we looked at earlier:

```
model_input = format_input(data[50])
desired_response = f"\n\n### Response:\n{data[50]['output']}"
print(model_input + desired_response)
```

The formatted input looks like as follows:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
Identify the correct spelling of the following word.

### Input:
Ocassion

### Response:
The correct spelling is 'Occasion.'
```

Note that the `format_input` skips the optional `### Input:` section if the `'input'` field is empty, which we can test out by applying the `format_input` function to entry `data[999]` that we inspected earlier:

```
model_input = format_input(data[999])
desired_response = f"\n\n### Response:\n{data[999]['output']}"
print(model_input + desired_response)
```

The output shows that entries with an empty `'input'` field don't contain an `### Input:` section in the formatted input:

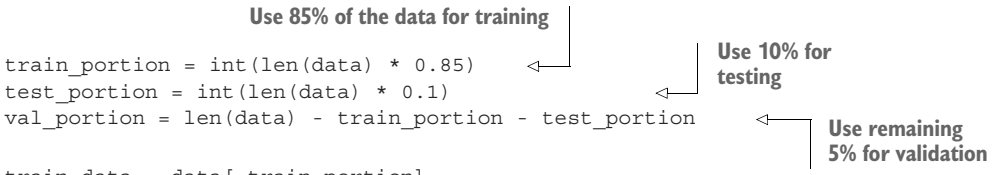
```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
What is an antonym of 'complicated'?

### Response:
An antonym of 'complicated' is 'simple'.
```

Before we move on to setting up the PyTorch data loaders in the next section, let's divide the dataset into training, validation, and test sets analogous to what we have done with the spam classification dataset in the previous chapter. The following listing shows how we calculate the portions.

Listing 7.3 Partitioning the dataset




```
train_portion = int(len(data) * 0.85)
test_portion = int(len(data) * 0.1)
val_portion = len(data) - train_portion - test_portion

train_data = data[:train_portion]
test_data = data[train_portion:train_portion + test_portion]
val_data = data[train_portion + test_portion:]

print("Training set length:", len(train_data))
print("Validation set length:", len(val_data))
print("Test set length:", len(test_data))
```

This partitioning results in the following dataset sizes:

```
Training set length: 935
Validation set length: 55
Test set length: 110
```