how fine-tuning large language models with new factual information affects their ability to use preexisting knowledge, revealing that models learn new facts more slowly and their introduction during fine-tuning increases the model's tendency to generate incorrect information:

- "Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?" (2024) by Gekhman, https://arxiv.org/abs/2405.05904

Preference fine-tuning is an optional step after instruction fine-tuning to align the LLM more closely with human preferences. The following articles by the author provide more information about this process:

- "LLM Training: RLHF and Its Alternatives," https://mng.bz/ZVPm
- "Tips for LLM Pretraining and Evaluating Reward Models," https://mng.bz/RNXj

Appendix A

While appendix A should be sufficient to get you up to speed, if you are looking for more comprehensive introductions to deep learning, I recommend the following books:

- *Machine Learning with PyTorch and Scikit-Learn* (2022) by Sebastian Raschka, Hayden Liu, and Vahid Mirjalili. ISBN 978-1801819312
- *Deep Learning with PyTorch* (2021) by Eli Stevens, Luca Antiga, and Thomas Viehmann. ISBN 978-1617295263

For a more thorough introduction to the concepts of tensors, readers can find a 15-minute video tutorial that I recorded:

- "Lecture 4.1: Tensors in Deep Learning," https://www.youtube.com/watch?v=JXfDlgrfOBY

If you want to learn more about model evaluation in machine learning, I recommend my article

- "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning" (2018) by Sebastian Raschka, https://arxiv.org/abs/1811.12808

For readers who are interested in a refresher or gentle introduction to calculus, I've written a chapter on calculus that is freely available on my website:

- "Introduction to Calculus," by Sebastian Raschka, https://mng.bz/WEyW

Why does PyTorch not call `optimizer.zero_grad()` automatically for us in the background? In some instances, it may be desirable to accumulate the gradients, and PyTorch will leave this as an option for us. If you want to learn more about gradient accumulation, please see the following article:

- "Finetuning Large Language Models on a Single GPU Using Gradient Accumulation" by Sebastian Raschka, https://mng.bz/8wPD

This appendix covers DDP, which is a popular approach for training deep learning models across multiple GPUs. For more advanced use cases where a single model doesn't fit onto the GPU, you may also consider PyTorch's Fully Sharded Data Parallel (FSDP) method, which performs distributed data parallelism and distributes large layers across different GPUs. For more information, see this overview with further links to the API documentation:

- "Introducing PyTorch Fully Sharded Data Parallel (FSDP) API," https://mng.bz/EZJR

appendix C
Exercise solutions

The complete code examples for the exercises' answers can be found in the supplementary GitHub repository at https://github.com/rasbt/LLMs-from-scratch.

Chapter 2

Exercise 2.1

You can obtain the individual token IDs by prompting the encoder with one string at a time:

```
print(tokenizer.encode("Ak"))
print(tokenizer.encode("w"))
# ...
```

This prints

```
[33901]
[86]
# ...
```

You can then use the following code to assemble the original string:

```
print(tokenizer.decode([33901, 86, 343, 86, 220, 959]))
```

This returns

```
'Akwirw ier'
```