While we have seen how to use the LLM to generate a single token, in the next section, we will put these concepts to action and implement a function that applies this concept sequentially to generate coherent output text.

2.7 Coding a minimal text generation function

The previous section explained a single iteration in the basic, sequential text generation process in LLMs. In this section, building on that concept, we will implement a text generation function that uses the pre-trained LLM to generate coherent text following a user prompt, as illustrated in Figure 2.13 in the chapter overview.

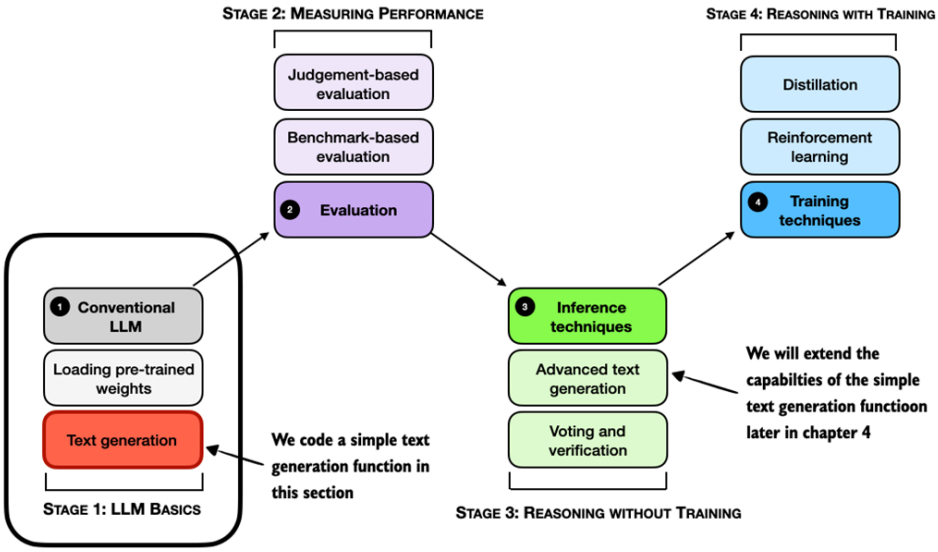

Figure 2.13 An overview of the four key stages in developing a reasoning model in this book. In this section we implement a text generation function for the pre-trained LLM.

This text generation function, mentioned in figure 2.13, works by first converting the input prompt into token IDs that the model can process. The model then predicts the next most likely token, appends it to the sequence, and reprocesses the extended sequence to generate the next token. This iterative process continues until a stopping condition is met, and the generated token IDs are then decoded back into text.

Figure 2.14 shows this process step by step, with both the generated token IDs and their corresponding text at each stage. (This figure is similar to figure 2.10 shown at the beginning of the previous section, except it shows the generated token ID alongside their text representation.)

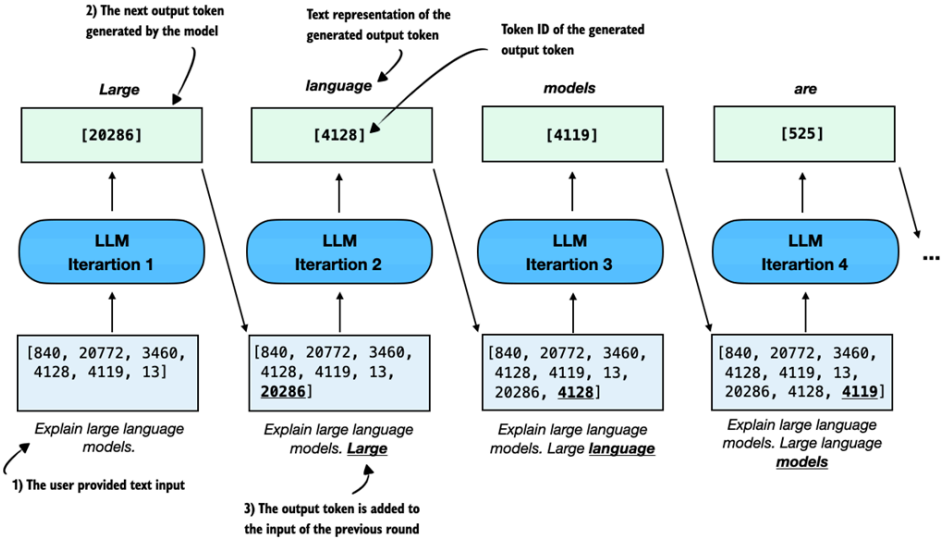

Figure 2.14 An illustration of sequential (autoregressive) text generation in large language models (LLMs), with token IDs shown explicitly. At each iteration, the model generates the next token based on the original input and all previously generated tokens. The predicted token is added to the sequence in both its textual and token ID form.

The `generate_text_basic` function in listing 2.1 below implements the sequential text generation process (figure 2.14) using the `argmax` function introduced in the previous section:

Listing 2.1 A basic text generation function

```
@torch.inference_mode()                                          #A
def generate_text_basic(
    model,
    token_ids,
    max_new_tokens,
    eos_token_id=None
):
    input_length = token_ids.shape[1]
    model.eval()                                                 #B

    for _ in range(max_new_tokens):
        out = model(token_ids)[:, -1]                            #C
        next_token = torch.argmax(out, dim=-1, keepdim=True)

        if (eos_token_id is not None                             #D
                and torch.all(next_token == eos_token_id)):
            break

        token_ids = torch.cat(                                   #E
            [token_ids, next_token], dim=1)                      #E
    return token_ids[:, input_length:]                           #F
```

#A Disable gradient tracking for speed and memory efficiency
#B Switch model to evaluation mode to enable deterministic behavior (best practice)
#C Get the scores of the last token
#D Stop if all sequences in the batch have generated EOS
#E Append the newly predicted token to the sequence
#F Return only the generated tokens (excluding the original input)

In essence, the `generate_text_basic` function listing 2.1 applies the `argmax`-based token ID extraction via a for-loop for a user-specified number of iterations (`max_new_tokens`). It returns the generated token IDs, similar to what's shown in figure 2.14, which we can then convert back into text.

Let's use the function to generate a 100-token response to a simple `"Explain large language models in a single sentence."` prompt to make sure that the `Qwen3Model` and `generate_text_basic` function work (we get to the reasoning task examples in later chapters).

Please note that the following code will be slow and can take 1-3 minutes to complete, depending on your computer (we will speed it up in later sections):

```
prompt = "Explain large language models in a single sentence."
input_token_ids_tensor = torch.tensor(
    tokenizer.encode(prompt),
    device=device                                                    #A
    ).unsqueeze(0)

max_new_tokens = 100                                                 #B
output_token_ids = generate_text_basic(
    model=model,
    token_ids=input_token_ids_tensor,
    max_new_tokens=max_new_tokens,
)
output_text = tokenizer.decode(
    output_token_ids.squeeze(0).tolist()                             #C
)
print(output_text)
```

#A Transfer the input token IDs onto the same device (CPU, GPU) where the model is located
#B Let the model generate up to 100 new tokens
#C Convert output token IDs from PyTorch tensor to Python list

The generated output text is as follows:

```
Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.<|endoftext|>Human language
is a complex and dynamic system that has evolved over millions of
years to enable effective communication and social interaction. It is
composed of a vast array of symbols, including letters, numbers, and
words, which are used to convey meaning and express thoughts and
ideas. The evolution of language has
```

Note that the output above was generated on a CPU. Depending on the device (e.g., CPU versus GPU), the exact wording may vary slightly due to differences in floating-point behavior on different hardware.

As we can see based on the output above, the model follows the instruction quite well by producing a single, clear sentence in response to the prompt. However, it continues generating additional, off-topic text after the special token `<|endoftext|>`. This token is used during training to mark the end of a document and separate different samples.

TIP The leading whitespace in `" Large"` (the first output word) appears because the model continued the text based on the input prompt but we sliced off the original prompt with `token_ids[:, input_length:]` in the return line in listing 2.1. If this leading whitespace bothers you, you can remove it via `token_ids[:, input_length:].lstrip()` or `output_text.lstrip()`.

When using the model for inference (that is, generating outputs based on input), we typically want it to stop as soon as it produces the special token `<|endoftext|>`. This token is represented by the ID 151643, which we can confirm using:

```
print(tokenizer.encode("<|endoftext|>"))
```

For convenience, this token ID is also saved via the `tokenizer.eos_token_id` attribute. We can pass this ID to the `generate_text_basic` function to signal when generation should stop:

```
output_token_ids_tensor = generate_text_basic(
    model=model,
    token_ids=input_token_ids,
    max_new_tokens=max_new_tokens,
    eos_token_id=tokenizer.eos_token_id                          #A
)
output_text = tokenizer.decode(
    output_token_ids_tensor.squeeze(0).tolist()
)
print(output_text)
```

#A Pass end-of-sequence (eos) token ID

The output looks like this:

```
Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.
```

If we compare the response to the previous response, we can see that the text generation stopped once the end-of-sequence token was encountered.

You may have noticed that generating the response is relatively slow and might take several seconds up to multiple minutes, depending on the hardware.

EXERCISE 2.2: STREAMING TOKEN GENERATION

Write a modified version of the `generate_text_basic` function that returns each token as it is generated and prints it, which is also known as *streaming* token generation.

The goal of this exercise is to understand how to implement token-by-token text generation, a technique often used in real-time applications like chatbots and interactive assistants.

Tip 1: Use `yield` instead of `return` to turn the function into a generator.

Tip 2: Then, outside the function, decode each token using a tokenizer and print it as it's generated (`for token in generate_text_basic_stream(...):...`) to simulate streaming output.

Before we wrap up and learn how to speed up this function substantially, let's implement a simple utility function that measures the runtime of the text generation process:

```
def generate_stats(output_token_ids, tokenizer, start_time, end_time):
    total_time = end_time - start_time
    print(f"Time: {total_time:.2f} sec")
    print(f"{int(output_token_ids.numel() / total_time)} tokens/sec")

    if torch.cuda.is_available():
        max_mem_bytes = torch.cuda.max_memory_allocated()
        max_mem_gb = max_mem_bytes / (1024 ** 3)
        print(f"Max memory allocated: {max_mem_gb:.2f} GB")

    output_text = tokenizer.decode(output_token_ids.squeeze(0).tolist())
    print(f"\n{output_text}")
```

The `generate_stats` function above will calculate the total runtime, given a start and end time stamp, the generation speed in terms of tokens per second (tokens/sec), and the GPU memory used. Note that the GPU memory usage is currently only computed for CUDA-supported GPUs, as PyTorch lacks similar utility functions for CPUs and Apple Silicon GPUs.

To apply the `generate_stats` function, we obtain a `start_time` and `end_time` stamp immediately before and after running the `generate_text_basic` function via Python's `time` module:

```
import time

start_time = time.time()
output_token_ids_tensor = generate_text_basic(
    model=model,
    token_ids=input_token_ids_tensor,
    max_new_tokens=max_new_tokens,
    eos_token_id=tokenizer.eos_token_id
)
end_time = time.time()
generate_stats(output_token_ids_tensor, tokenizer, start_time, end_time)
```

The output, on a Mac Mini M4 CPU, is as follows:

```
Time: 7.94 sec
5 tokens/sec
 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.
```

At 5 tokens per second, the generation speed is relatively slow. In the next section, we will implement a caching technique that speeds up the generation process 5-6 fold.

> **TEXT GENERATION AND INFERENCE TERMINOLOGY**
>
> When reading LLM literature or software documentation, you will inevitably stumble upon the term *inference*, often used in place of *text generation*. In this context, inference comes from neural network jargon and refers to using a trained model to make predictions, such as generating the next tokens from a prompt. This is different from inference in statistics, which typically means drawing conclusions about a population from data. So when we call the *generate_text_basic* function, we may be performing inference in the neural network sense.

2.8 Faster inference via KV caching

So now that we have a basic text generation function in place, we can turn our attention to what happens when we actually run it in practice. As you may have noticed, the text generation in the previous section can be a bit slow. That slowdown points us to a key concern: performance during inference.

When running inference with LLMs, which in this context means generating text from a prompt, runtime performance (efficiency) quickly becomes important, especially for long sequences. While the code in this book emphasizes clarity over speed, real-world systems often use engineering tricks to make inference more efficient.

In the remaining two sections, we will cover two fundamental techniques, KV caching and model compilation, as shown in the overview in figure 2.15, to speed up the text generation.

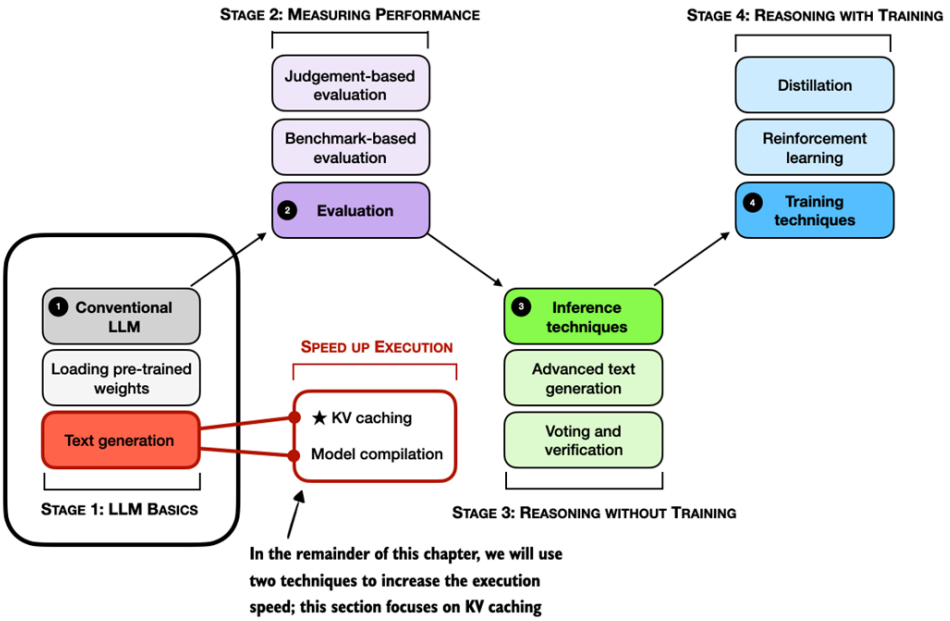

Figure 2.15 An overview of the four key stages in developing a reasoning model in this book. This section builds on pre-trained LLM and the basic text generation function we coded earlier and applies KV caching to speed up execution.

As shown in figure 2.15, One engineering trick that increases the text generation speed is *KV caching,* where KV refers to the keys and values used in the model's attention mechanism. If you are not familiar with these terms, that's okay. The key idea is that we can cache certain intermediate values and reuse them at each step of text generation, as shown in figure 2.16, which helps speed up inference.

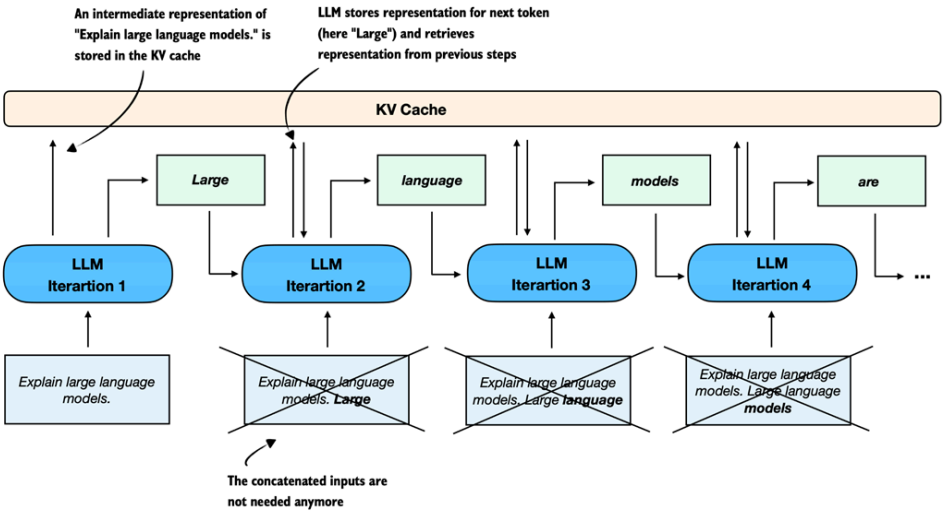

Figure 2.16 Illustration of how a KV cache improves efficiency during autoregressive text generation. Instead of reprocessing the entire input sequence at each step, the KV cache stores intermediate representations so that the LLM can reuse them to generate the next token. This eliminates the need to concatenate the generated token with prior inputs in each subsequent iteration.

The key idea of KV caching, as shown in figure 2.16, is to store intermediate values computed in each iteration in a cache. Previously, each new token generated by the network was concatenated to the entire input sequence and fed back into the model repeatedly (indicated by crossed-out boxes in the diagram). This approach was inefficient because all tokens, except the newly generated one, remain identical in subsequent iterations. By using a KV cache, we avoid redundant computation and instead directly retrieve stored intermediate representations.

As mentioned earlier, the non-reasoning focused LLM details like KV caching, which we used to improve the token generation speed, are outside the scope of this book, and they are not required for the topics covered later in this book. However, interested readers can find more information on the mechanics of KV caching in my freely available article: *Understanding and Coding the KV Cache in LLMs from Scratch* (https://magazine.sebastianraschka.com/p/coding-the-kv-cache-in-llms).

Below is a modified version of the `generate_text_basic` function that incorporates a KV cache, which is almost identical to the basic text generation function in listing 2.1, except for the KV cache-related change highlighted via the comments:

Listing 2.2 A basic text generation function with KV cache

```
from reasoning_from_scratch.qwen3 import KVCache

@torch.inference_mode()
def generate_text_basic_cache(
    model,
    token_ids,
    max_new_tokens,
    eos_token_id=None
):

    input_length = token_ids.shape[1]
    model.eval()
    cache = KVCache(n_layers=model.cfg["n_layers"])               #A
    model.reset_kv_cache()
    out = model(token_ids, cache=cache)[:, -1]                    #B

    for _ in range(max_new_tokens):
        next_token = torch.argmax(out, dim=-1, keepdim=True)

        if (eos_token_id is not None
                and torch.all(next_token == eos_token_id)):
            break

        token_ids = torch.cat([token_ids, next_token], dim=1)
        out = model(next_token, cache=cache)[:, -1]               #C

    return token_ids[:, input_length:]
```

#A Initialize the KV cache
#B In the first round, the whole input is provided to the model as before
#C Consequent iterations only feed the next_token to the input

The `generate_text_basic_cache` function in listing 2.2 differs only slightly from the `generate_text_basic` function in listing 2.1. The main difference is the introduction of a `KVCache` object.

During the first iteration, the model is given the full input token sequence as before, using `model(token_ids, cache=cache)`. Behind the scenes, the KV cache stores intermediate values for all these input tokens.

In the following iterations, we no longer need to pass the entire sequence. Instead, we only provide the `next_token` to the model using `model(next_token, cache=cache)`. The model then retrieves the necessary context from the previously stored KV cache.

Let's time this function to see whether it provides any performance benefits:

```
start_time = time.time()
output_token_ids_tensor = generate_text_basic_cache(
    model=model,
    token_ids=input_token_ids_tensor,
    max_new_tokens=max_new_tokens,
    eos_token_id=tokenizer.eos_token_id,
)
end_time = time.time()
generate_stats(output_token_ids_tensor, tokenizer, start_time, end_time)
```

The output is:

```
Time: 1.40 sec
29 tokens/sec

 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.
```

As we can see, this approach is significantly faster, generating 29 tokens per second compared to just 5 tokens per second previously (measured on a Mac Mini M4 CPU).

Importantly, we also see that the generated text is the same as before, which is an important sanity check to ensure that the KV cache is implemented and used correctly.

In the next section, we will learn about another technique we can use to further improve the generation speed, which will come in handy when we evaluate the model in the upcoming chapters. Faster generation allows us to run more evaluations in less time and makes it easier to compare different models or settings efficiently.

2.9 Faster inference via PyTorch model compilation

In the previous section, we covered KV caching as a technique to improve runtime efficiency as shown in the overview in figure 2.17.

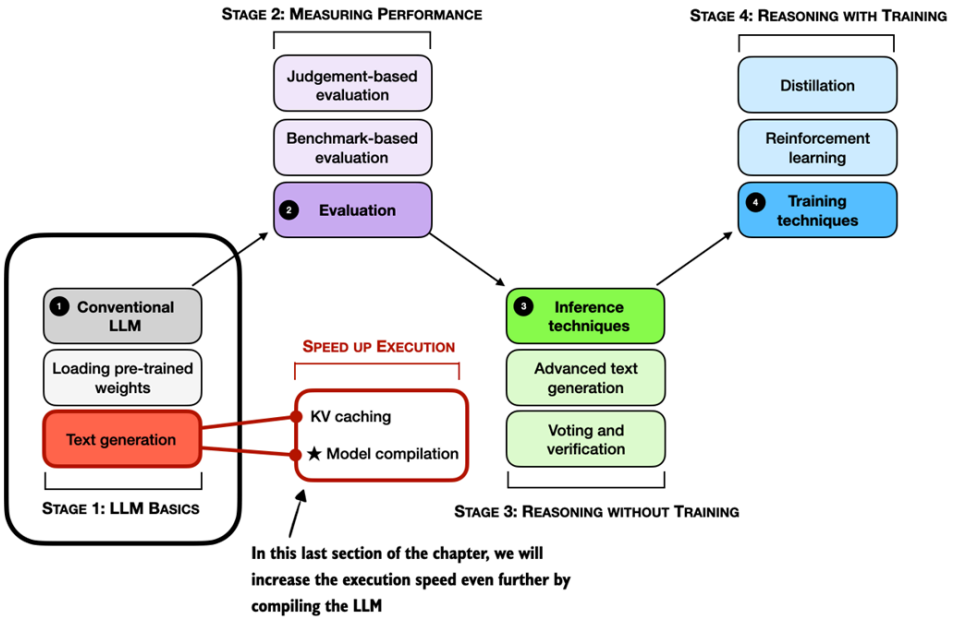

Figure 2.17 An overview of the four key stages in developing a reasoning model in this book. This section builds on pre-trained LLM and the basic text generation function we coded earlier, including KV caching, and adds model compilation to speed up the execution speed even further.

As shown in figure 2.17, in this remaining section of this chapter, we will apply another technique that can substantially speed up model inference: model compilation using `torch.compile`. This feature allows the model to be compiled ahead of time, which reduces overhead and improves runtime performance during text generation.

At the time of writing, however, `torch.compile` is not well supported on MPS devices (Apple Silicon GPUs). Attempting to use it on such hardware will result in an `InductorError` for the `Qwen3Model`.

To maintain compatibility across devices, we check the hardware type and apply `torch.compile` only when it is supported:

```
if device.type == "mps":
    print("`torch.compile` is not supported"
          f" for the {model.__class__.__name__} model"
          " on MPS (Apple Silicon) as of this writing."
    )
    model_compiled = model                                          #A
else:
    model_compiled = torch.compile(model)
```

#A Skip compilation on Apple Silicon GPUs

We assign the compiled model to a new variable so that the code in this chapter continues to function properly.

It is worth noting that the first execution using the compiled model may be slower than usual due to the initial compilation and optimization steps. To better measure the performance improvement, we will repeat the text generation process multiple times.

To begin, we will test this using the non-cached version of the generation function. The code is similar to what we used before except that we run it three times in a row. The code execution may take a few minutes to finish, depending on the system:

```
for i in range(3):                                                    #A
    start_time = time.time()
    output_token_ids_tensor = generate_text_basic(
        model=model_compiled,
        token_ids=input_token_ids_tensor,
        max_new_tokens=max_new_tokens,
        eos_token_id=tokenizer.eos_token_id
    )
    end_time = time.time()

    if i == 0:                                                        #B
        print("Warm-up run")                                          #B
    else:
        print(f"Timed run {i}:")
    generate_stats(output_token_ids_tensor, tokenizer, start_time, end_time)

    print(f"\n{30*'-'}\n")
```

#A We run the token generation three times
#B The first run is labeled as "Warm-up run"

The output is as follows:

```
Warm-up run
Time: 11.68 sec
3 tokens/sec

 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.

------------------------------
```

```
Timed run 1:
Time: 6.78 sec
6 tokens/sec
Output text:

 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.

------------------------------

Timed run 2:
Time: 6.80 sec
6 tokens/sec
Output text:

 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.

------------------------------
```

As we can see from the results above, the compiled model achieves a slight improvement in speed, with around 6 tokens per second compared to the previous 5 tokens per second.

Next, let's see how the KV cache version performs in comparison, using the same code as before except for swapping `generate_text_basic` with `generate_text_basic_cache`:

```
for i in range(3):
    start_time = time.time()
    output_token_ids_tensor = generate_text_basic_cache(
        model=model_compiled,
        token_ids=input_token_ids_tensor,
        max_new_tokens=max_new_tokens,
        eos_token_id=tokenizer.eos_token_id
    )
    end_time = time.time()

    if i == 0:
        print("Warm-up run")
        generate_stats(
        output_token_ids_tensor, tokenizer, start_time, end_time
    )
    else:
        print(f"Timed run {i}:")
        generate_stats(output_token_ids, tokenizer, start_time, end_time)

    print(f"\n{30*'-'}\n")
```

The output is as follows:

```
Warm-up run
Time: 8.07 sec
5 tokens/sec

 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.

------------------------------

Timed run 1:
Time: 0.60 sec
68 tokens/sec
Output text:

 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.
```

```
------------------------------

Timed run 2:
Time: 0.60 sec
68 tokens/sec
Output text:

 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.

------------------------------
```

As we can see based on the outputs above, the model generation speed improved from 29 tokens per second for the uncompiled model with KV cache to 68 tokens per second when the same model is compiled (on a Mac Mini M4 CPU), which is more than a 2-fold speed-up.

EXERCISE 2.3: RERUN CODE ON NON-CPU DEVICES

If you have access to a GPU, rerun the code in this chapter on a GPU device and compare the runtimes to the CPU runtimes.

In case you are curious, how the different model configurations compare on an Apple Silicon GPU and a high-end NVIDIA GPU, see table 2.1.

Table 2.1 Token generation speeds and GPU memory usage for different model configurations on different hardware

Mode	Hardware	Tokens/sec	GPU memory
Regular	Mac Mini M4 CPU	5	-
Regular compiled	Mac Mini M4 CPU	6	-
KV cache	Mac Mini M4 CPU	28	-
KV cache compiled	Mac Mini M4 CPU	68	-
Regular	Mac Mini M4 GPU	17	-
Regular compiled	Mac Mini M4 GPU	InductorError	-
KV cache	Mac Mini M4 GPU	18	-
KV cache compiled	Mac Mini M4 GPU	InductorError	-
Regular	NVIDIA H100 GPU	51	1.55 GB
Regular compiled	NVIDIA H100 GPU	164	1.81 GB
KV cache	NVIDIA H100 GPU	48	1.52 GB
KV cache compiled	NVIDIA H100 GPU	141	1.81 GB

As shown in the table above, the NVIDIA GPU delivers the best performance, which is expected. However, the CPU also performs remarkably well when using both a KV cache and a compiled model. It's worth noting that this is a relatively small model, and I optimized the KV-cache implementation for CPUs to ensure accessibility for most readers. With a larger model or GPU-optimized code, the performance gap in favor of the NVIDIA GPU would likely be more pronounced.

Also, keep in mind that performance can vary with longer input sequences since the cost of the LLM-internal attention mechanism scales quadratically with the input length.

All examples were run using a single prompt (i.e., a batch size of 1). For readers interested in how performance scales with multiple inputs, batched inference is discussed in appendix E.

2.10 Summary

- Using LLMs to generate text involves multiple key steps:
  - Setting up the coding environment to run LLM code and install necessary dependencies.
  - Loading a pre-trained base LLM (such as Qwen3 0.6B), which will be extended with reasoning capabilities in later chapters.
  - Initializing and using a tokenizer, which converts text input into token IDs and decodes output back to human-readable form.

- Text generation in LLMs follows a sequential (autoregressive) process, where the model generates one token at a time by predicting the next most likely token.
- The speed and efficiency of text generation can be improved through:
  - KV caching, which stores intermediate states to avoid recomputing previously encountered input tokens at each step.
  - Model compilation using torch.compile, which optimizes runtime performance.

- This chapter lays the technical foundation for reasoning capabilities in upcoming chapters by implementing a functional, efficient text generation pipeline using a pre-trained base LLM.

Appendix A. References and further reading

A.1 Chapter 1

A.1.1 References

The announcement article of OpenAI's o1 model, which is regarded as the first LLM-based reasoning model:

- Introducing OpenAI o1-preview, https://openai.com/index/introducing-openai-o1-preview/

DeepSeek-R1 is the first open-source reasoning model that was accompanied by a comprehensive technical report, which was the first to show that reasoning emerges from reinforcement learning with verifiable rewards (a topic covered in more detail in chapter 5):

- DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, https://arxiv.org/abs/2501.12948

OpenAI CEO's comment on the reasoning ("chain-of-thought") capabilities of future models:

- "[...] We will next ship GPT-4.5, the model we called Orion internally, as our last non-chain-of-thought model. [...]", https://x.com/sama/status/1889755723078443244

A research paper by AI researchers at Apple finding that reasoning models are sophisticated (but very capable) pattern matchers:

- The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity, https://machinelearning.apple.com/research/illusion-of-thinking

An in-depth book and guide on implementing and training large language models step-by-step:

- Build a Large Language Model (From Scratch),http://mng.bz/orYv

A.1.2 Further Reading

An introduction to how DeepSeek-R1 works, providing insights into the foundations of reasoning in LLMs:

- Understanding Reasoning LLMs, https://magazine.sebastianraschka.com/p/understanding-reasoning-llms

A.2 Chapter 2

A.2.1 References

Official installation page for the uv Python package and project manager:

- Installing uv, https://docs.astral.sh/uv/getting-started/installation/

Cloud compute platforms with GPU support:

- Lightning AI, https://lightning.ai/
- Google Colab, https://colab.research.google.com/

Qwen3 resources with additional benchmark performance and comparison to other models:

- Blog post, https://qwenlm.github.io/blog/qwen3/
- Technical report, https://arxiv.org/abs/2505.09388

A.2.2 Further Reading

A PyTorch tutorial for readers who are new to PyTorch or would like a refresher:

- PyTorch in One Hour: From Tensors to Training Neural Networks on Multiple GPUs tutorial, https://sebastianraschka.com/teaching/pytorch-1h

Additional resources on tokenization:

- Build a Large Language Model (from Scratch) chapter 2, https://mng.bz/M96o
- Implementing A Byte Pair Encoding (BPE) Tokenizer From Scratch, https://sebastianraschka.com/blog/2025/bpe-from-scratch.html

Appendix B. Exercise solutions

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

B.1 Chapter 2

EXERCISE 2.1

You can use a prompt similar to "Hello, Ardwarklethyrx. Haus und Garten.", which contains a made-up word (`"Ardwarklethyrx"`) and three words in a non-English language (German):

```
"Haus und Garten":
prompt = "Hello, Ardwarklethyrx. Haus und Garten."
input_token_ids_list = tokenizer.encode(prompt)
for i in input_token_ids_list:
    print(f"{[i]} --> {tokenizer.decode([i])}")
```

The output is:

```
[9707] --> Hello
[11] --> ,
[1644] -->  Ar
[29406] --> dw
[838] --> ark
[273] --> le
[339] --> th
[10920] --> yr
[87] --> x
[13] --> .
[47375] -->  Haus
[2030] -->  und
[93912] -->  Garten
[13] --> .
```

As we can see, unknown words are broken into smaller pieces of subwords or even single tokens; this allows the tokenizer and LLM to handle any input.

German words are not broken down into characters or even subwords here, suggesting that the tokenizer has seen German texts during training. This also suggests that the LLM was likely trained on German texts, too, and should be able to handle certain non-English languages well.

EXERCISE 2.2

The updated `generate_text_basic` function looks like as follows:

```
@torch.inference_mode()
def generate_text_basic_stream(
    model,
    token_ids,
    max_new_tokens,
    eos_token_id=None
):

    # input_length = token_ids.shape[1]                          #A

    model.eval()
    for _ in range(max_new_tokens):
        out = model(token_ids)[:, -1]
        next_token = torch.argmax(out, dim=-1, keepdim=True)

        if (eos_token_id is not None
                and torch.all(next_token == eos_token_id)):
```

```
            break

        yield next_token                                          #B

        token_ids = torch.cat([token_ids, next_token], dim=1)
    # return token_ids[:, input_length:]                          #C



for token in generate_text_basic_stream(
    model=model,
    token_ids=input_token_ids_tensor,
    max_new_tokens=50,
    eos_token_id=tokenizer.eos_token_id
):
    token_id = token.squeeze(0).tolist()
    print(
        tokenizer.decode(token_id),
        end="",
        flush=True
    )
```

#A The input_length is no longer needed
#B We now yield each token as it's generated
#C Since we use yield, we no longer need the return statement

This prints the following text:

```
 Large language models are artificial intelligence systems that can
understand, generate, and process human language, enabling them to
perform a wide range of tasks, from answering questions to writing
articles, and even creating creative content.
```

Note that the text above is identical to the text generated by the `generate_text_basic` function when using the same prompt, as expected. However, if you execute the code on your computer, you should see each word being generated on the fly now.

Similarly, we can modify the KV cache variant (`generate_text_basic_cache`) as follows to add streaming support:

```
from reasoning_from_scratch.qwen3 import KVCache

@torch.inference_mode()
def generate_text_basic_stream_cache(
    model,
    token_ids,
    max_new_tokens,
    eos_token_id=None
):
    # input_length = token_ids.shape[1]                          #A
    model.eval()
    cache = KVCache(n_layers=model.cfg["n_layers"])
    model.reset_kv_cache()

    out = model(token_ids, cache=cache)[:, -1]
    for _ in range(max_new_tokens):
        next_token = torch.argmax(out, dim=-1, keepdim=True)

        if (eos_token_id is not None
                and torch.all(next_token == eos_token_id)):
            break

        yield next_token                                         #B
        # token_ids = torch.cat([token_ids, next_token], dim=1)
        out = model(next_token, cache=cache)[:, -1]

    # return token_ids[:, input_length:]                         #C
```

#A The input_length is no longer needed

#B We now yield each token as it's generated

#C Since we use yield, we no longer need the return statement

EXERCISE 2.3

You can simply delete the line `device = torch.device("cpu")` in section 2.5, and then rerun the rest of the code in chapter 2 as is. Reference numbers for the hardware I tried the code on are provided in table 2.1 at the end of chapter 2.