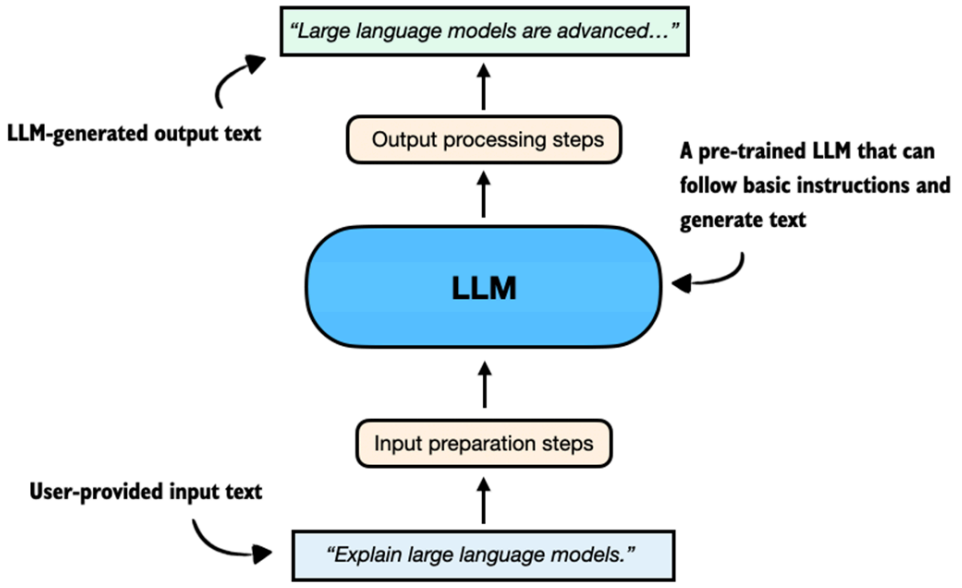

Figure 2.2 An overview depicting an LLM generating a response (output text) given a user query (input text)

Figure 2.2 summarizes the components of an LLM text generation pipeline, and we will discuss and implement these steps in more detail later in this chapter.

> **NOTE** By convention, diagrams involving neural networks such as LLMs are drawn and read vertically from bottom (inputs) to top (outputs). Arrows indicate the flow of information upward through the model.

If you have not coded an LLM or used LLMs programmatically before, this chapter will teach you how the *text generation* process works. However, in this chapter, we will not go deep into the internals of an LLM, such as the attention mechanism and other architecture components; this is the topic of my other book, *Build a Large Language Model (from Scratch)*. Note that understanding these internals are not required for this book, and, if you are curious, you can learn about them after you finish reading this book.

Before we begin implementing the components shown in figure 2.2, including input preparation, loading the LLM, and generating text, we first need to set up our coding environment. This is the focus of the next section.

2.2 Setting up the coding environment

This section provides instructions and recommendations for setting up your Python coding environment to follow along with the examples in this book. I recommend reading this section in its entirety before deciding which way is for you.

If you are reading this book, you have probably coded in Python before. In this case, the simplest way to install dependencies, if you already have a Python environment set up (with Python 3.10 or newer), is to use Python's package installer (`pip`) in your terminal.

If you have downloaded the code from the publisher's website, use the `requirements.txt` file to install the required Python libraries used throughout this book:

```
pip install -r requirements.txt
```

Alternatively, to install the required packages directly without downloading the `requirements.txt` file, use:

```
pip install -r https://raw.githubusercontent.com/\
rasbt/reasoning-from-scratch/refs/heads/main/requirements.txt
```

PYTHON PACKAGES USED IN THIS CHAPTER

If you prefer to install only the packages used in this chapter, you can do this with the following command:

```
pip install torch>=2.7.1 tokenizers>=0.21.2 reasoning-from-scratch
```

- `torch` refers to PyTorch, a widely used deep learning library that provides tools for building and training neural networks.
- `tokenizers` is a library that provides efficient tokenization algorithms, used to prepare input data for LLMs.
- `reasoning-from-scratch` is a custom library that I developed for this book. It includes all the code examples implemented throughout the chapters, along with additional utility functions we will be using.

While `pip` is the canonical way to install Python packages, my preferred way to use Python is via the widely recommended `uv` Python package and project manager instead. It comes with its own Python executable, so it's also a great option if you don't have Python installed on your system, yet.

Figure 2.3 outlines the 4-step process from installing uv to getting ready to execute the code in this chapter, which we will cover in the remainder of this section.

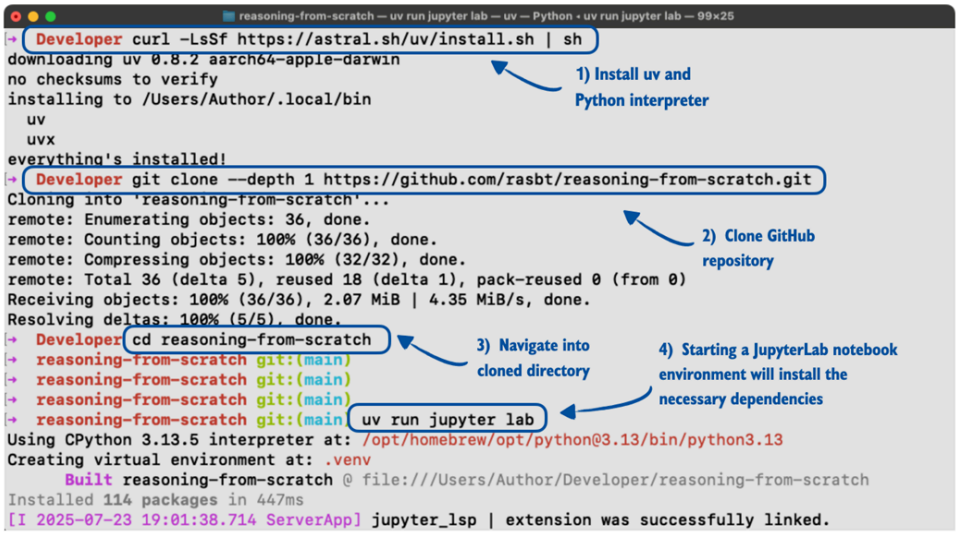

Figure 2.3 Installing and using the `uv` Python package and project manager via the macOS terminal

Note that figure 2.3 steps through the `uv` installation and usage on a macOS terminal, but `uv` is supported by Linux and Windows as well.

1) To install `uv`, run the installation for your OS from the official website: https://docs.astral.sh/uv/getting-started/installation/

2) Next, clone the GitHub repo:

```
git clone --depth 1 https://github.com/rasbt/reasoning-from-scratch.git
```

Here, the `--depth 1` option tells `git` to perform a shallow clone, which means it only downloads the latest version of the code without the full version history. This makes the download faster and uses less space.

If you don't have `git` installed, you can also manually download the source code repository from the publisher's website or by opening this link in your browser: https://github.com/rasbt/reasoning-from-scratch/archive/refs/heads/main.zip (unzip it after downloading).

3) Next, in the terminal, navigate to the `reasoning-from-scratch` folder.

4) Inside the `reasoning-from-scratch` folder, execute:

```
uv run jupyter lab
```

The command above will launch JupyterLab, where you can open a blank Jupyter notebook to type and execute code or open the chapter 2 notebook that contains all the code covered in this chapter.

TIP **Python script files can be executed via** `uv run script-name.py`.

The above `uv run...` command also sets up a local virtual environment (usually inside an invisible `.venv/` folder) and installs all dependencies from the `pyproject.toml` file inside the `reasoning-from-scratch` folder automatically. So, the manual installation of code dependencies via the requirements file is not needed. However, if you plan to install additional packages, you can use the following command:

```
uv add packagename
```

The supplementary code repository contains additional installation instructions and details inside the `ch02` subfolder if needed.

2.3 Understanding hardware needs and recommendations

You may have heard that training LLMs is very expensive. For leading LLM companies, it is not uncommon to spend anywhere between 1-10 million Dollars on the small end and >50 million Dollars on the high end in terms of compute costs to train a new base model LLM before even adding any reasoning techniques.

This high price tag would make the development of an LLM unfeasible for me and most readers. So, we are going to use a relatively small (but capable) pre-trained LLM on top of which we implement reasoning techniques.

Note that this smaller LLM is a scaled-down version that otherwise follows the same architecture as contemporary state-of-the-art models. And the reasoning methods that we will apply are the same as those used by larger LLMs. The difference is that the smaller LLM allows us to explore these methods in a budget-friendly way.

As an analogy, imagine you are curious to learn how cars work. If you are new to cars, as a learning exercise, you probably wouldn't start out building an expensive Ferrari right away. Instead, you would, for example, create a smaller car like a Volkswagen Beetle to start with, which still teaches you a lot about how engines and the transmission work. On the contrary, I would even say that working on a smaller car helps you *better* understand how the engine and transmission work because it gets complicated refinements and other details out of the way.

However, while we will use a relatively small model for these educational purposes in this book, the usage, development, and application of the reasoning techniques are still computationally intensive, and later chapters, such as chapters 5-7, will benefit from using a GPU.

If you followed the previous section, you should have PyTorch installed, which you can use to see if your computer has a PyTorch-supported GPU by executing the following PyTorch code in Python:

```
import torch

print(f"PyTorch version {torch.__version__}")
if torch.cuda.is_available():
    print("CUDA GPU")
elif torch.mps.is_available():
    print("Apple Silicon GPU")
else:
    print("Only CPU")
```

Depending on your machine, the code may return:

```
PyTorch version 2.7.1
Only CPU
```

Don't worry if your machine does not have a GPU to run the code. Chapters 2-4 can be executed in a reasonable time on a CPU.

Depending on the chapter, the code will automatically use an NVIDIA GPU if available, otherwise run on the CPU (or Apple Silicon GPU if recommended for a particular section or chapter). However, I will provide more information in the respective sections and chapters.

Like many other AI researchers who work on and with LLMs daily, I don't have a machine with the necessary GPU hardware to train LLMs at home and use cloud resources instead. If you are looking for cloud provider options, my personal preference is Lightning AI Studio (https://lightning.ai/), due to its ease of use and feature support, as shown in figure 2.4. Alternatively, Google Colab (https://colab.research.google.com/) is another good choice.

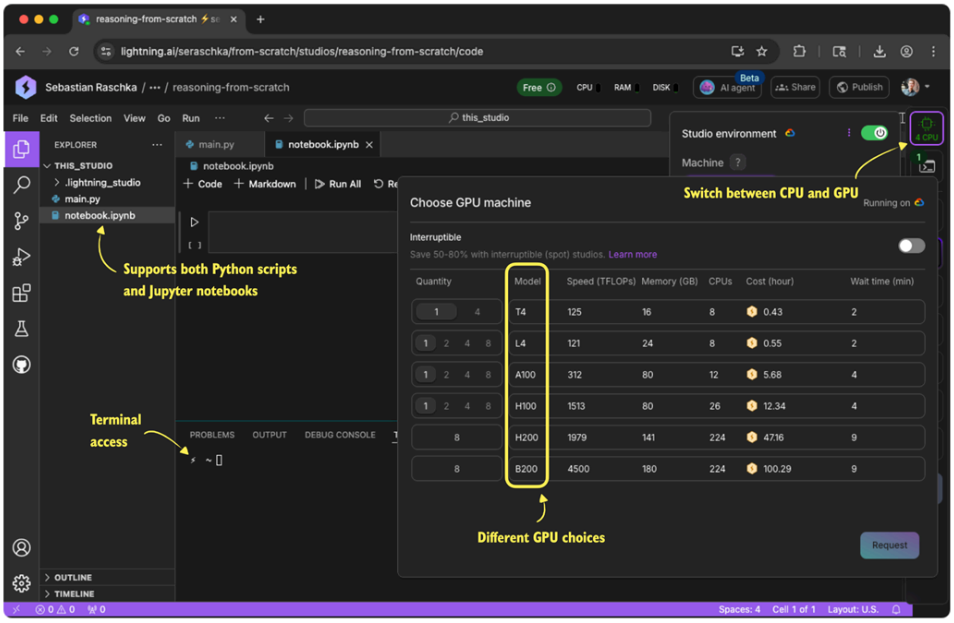

Figure 2.4 An overview of the Lightning AI GPU cloud platform in a web browser. The interface supports Python scripts, Jupyter notebooks, terminal access, and lets users switch between CPU and various GPU types based on their compute needs.

As of this writing, Lightning AI also offers users free compute credits after the sign-up and verification process, which can be used for the different GPU choices shown in figure 2.4. (As mentioned before, a GPU is not needed for this chapter; however, if you want to use a GPU, the L4 GPU is more than sufficient for this chapter.

> **NOTE** For disclosure, I helped build and launch the Lightning AI platform in 2023 and still hold a small stake. I am not sponsored to recommend it and pay for it myself. I use it because I simply find it the most convenient. It supports multiple types of GPUs, allows easy switching between them and back to CPU to save costs, and lets me pause or resume environments without redoing the setup.

The supplementary code repository contains additional GPU platform recommendations inside the `ch02` subfolder if needed.

USING PYTORCH

In this section, we imported and used the PyTorch library, which is currently the most widely used general-purpose library. We will use it throughout this book to run and train LLMs, including the reasoning methods we will develop. If you are new to PyTorch, to get the most out of this book, I recommend reading through my *PyTorch in One Hour: From Tensors to Training Neural Networks on Multiple GPUs* tutorial, which is freely available on my website at https://sebastianraschka.com/teaching/pytorch-1h.

2.4 Preparing input texts for LLMs

In this section, we explore how to use a tokenizer to process input and output text for an LLM, as shown in figure 2.5, which expands on the input and output preparation steps shown earlier in figure 2.2 to provide a more detailed view of the tokenization pipeline.

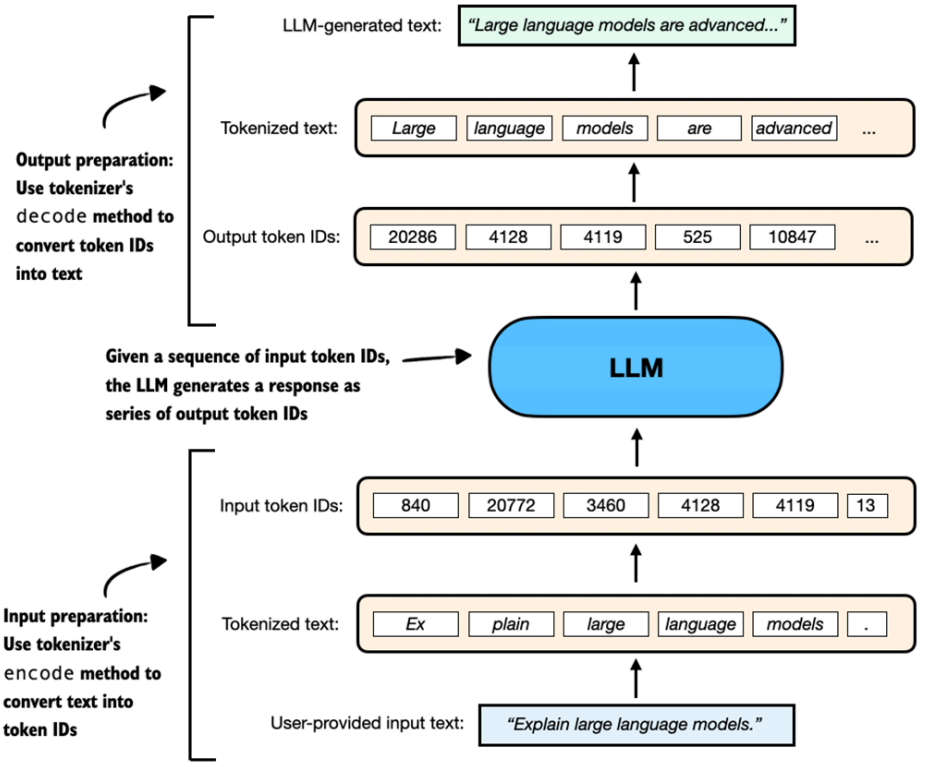

Figure 2.5 A simplified illustration of how an LLM receives input data and generates output. The user-provided text is tokenized into IDs using the tokenizer's `encode` method, which are then processed by the LLM to generate output token IDs. These are decoded back into human-readable text using the tokenizer's `decode` method.

To see how this works in practice, we will begin by loading a tokenizer from this book's `reasoning-from-scratch` Python package, which should have been installed according to the instructions in section 2.2.

To download the tokenizer files (corresponding to the *Qwen3* base LLM, which we will introduce in the next section), run:

```
from reasoning_from_scratch.qwen3 import download_qwen3_small
download_qwen3_small(kind="base", tokenizer_only=True, out_dir="qwen3")
```

This will display a progress bar similar to:

```
tokenizer-base.json: 100% (6 MiB / 6 MiB)
```

The command downloads the `tokenizer-base.json` file (approximately 6 megabytes in size) and saves it in a `qwen3` subdirectory.

Now, we can load the tokenizer settings from the tokenizer file into the `Qwen3Tokenizer`:

```
from pathlib import Path
from reasoning_from_scratch.qwen3 import Qwen3Tokenizer

tokenizer_file_path = Path("qwen3") / "tokenizer-base.json"
tokenizer = Qwen3Tokenizer(tokenizer_file_path=tokenizer_file_path)
```

Since we have not loaded the LLM yet (the central component shown in figure 2.5), we will first do a simpler dry run using just the tokenizer. Specifically, we will do a tokenization round-trip, that is, we will encode a text into *token IDs* and then decode those IDs back into text, as illustrated in figure 2.6.

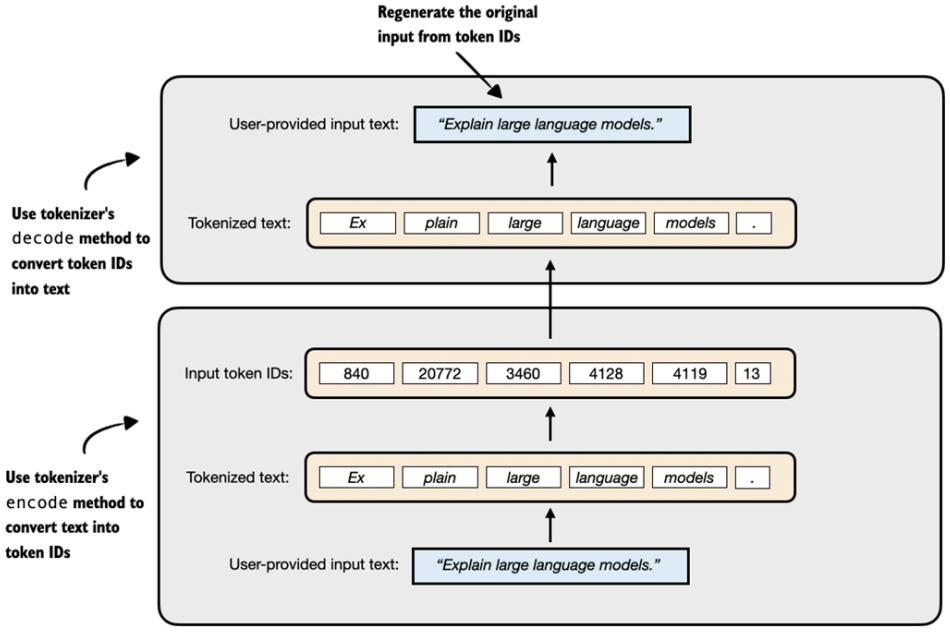

Figure 2.6 A demonstration of the round-trip tokenization process using a tokenizer. The user-provided input text is first converted into token IDs using the `encode` method, and then accurately reconstructed back into the original text using the `decode` method.

The following code snippet implements the encoding process shown at the bottom of figure 2.6:

```
prompt = "Explain large language models."
input_token_ids_list = tokenizer.encode(prompt)
```

And the following code implements the decoding process, converting the token IDs back into text, shown at the top of figure 2.6:

```
text = tokenizer.decode(input_token_ids_list)
print(text)
```

Based on the printed results, we can see that the tokenizer reconstructed the original input prompt from the token IDs:

```
'Explain large language models.'
```

Before we move on to the LLM, let's take a look at the token IDs that were generated by the `encode` method. The following code prints each token ID and its corresponding decoded string to help illustrate how the tokenizer works:

```
for i in input_token_ids_list:
    print(f"{[i]} --> {tokenizer.decode([i])}")
```

The output is as follows:

```
840 --> Ex
20772 --> plain
3460 -->  large
4128 -->  language
4119 -->  models
13 --> .
```

As shown in the output, the original text is split into six token IDs. Each token represents a word or subword, depending on how the tokenizer segments the input.

For example, "Explain" was split into two separate tokens, "Ex" and "plain". This is because the tokenizer algorithm uses a subword-based method based on *Byte Pair Encoding* (*BPE*). BPE can represent both common and rare words using a mix of full words and subword units. Spaces are also often included in tokens (for example, " large"), which helps the LLM detect word boundaries.

The `Qwen3Tokenizer` has a *vocabulary* of about 151,000 tokens, which is considered relatively large as of this writing (for comparison, the early GPT-2 has a vocabulary size of approximately 50,000 tokens, and Llama 3 has a vocabulary size of approximately 128,000 tokens).

A larger vocabulary in a language model increases its size and computational cost for each individually generated token, but it also allows more words to be represented as single tokens rather than being split into subword components. This is beneficial because splitting a word (like breaking "Explain" into "Ex" and "plain") results in more input tokens. More tokens lead to longer input sequences, which increases processing time and resource usage. For instance, doubling the number of tokens can roughly double the computational cost of running the model as it needs to generate more tokens to complete the response.

Unfortunately, a detailed coverage and from-scratch implementation of a tokenizer is outside the scope of this book. However, interested readers can find additional resources, including my from-scratch implementation, in the further resources and reading sections in appendix A.

EXERCISE 2.1: ENCODING UNKNOWN WORDS

Experiment with the tokenizer to see if and how it handles unknown words. For this, get creative and make up words that don't exist. Also, if you speak multiple languages, try to encode words in a different language than English.

2.5 Loading pre-trained models

In the previous section, we loaded and familiarized ourselves with the tokenizer that prepares the input data for an LLM and converts LLM outputs back into a human-readable text representation. In this section, we will load the LLM itself, as shown in the overview in figure 2.7.

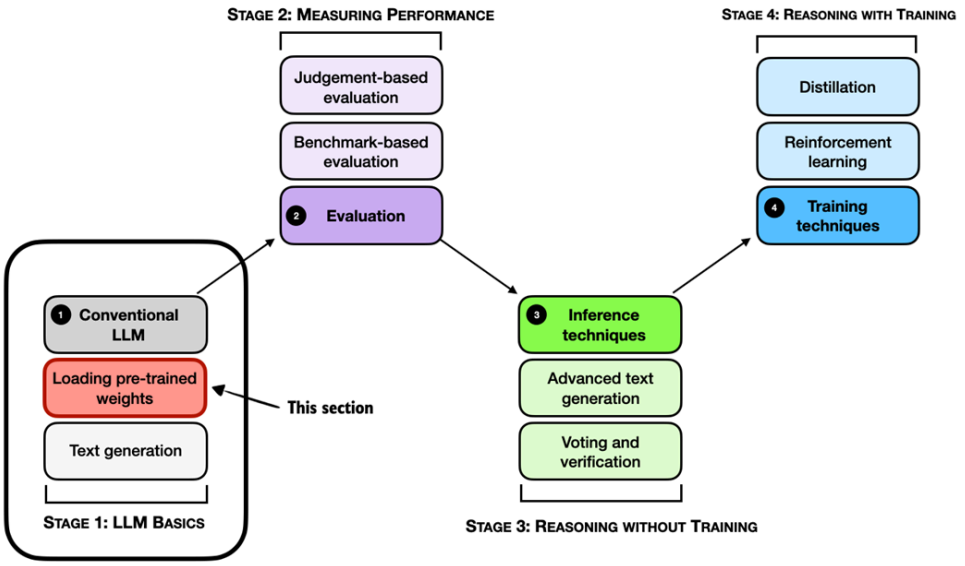

Figure 2.7 An overview of the four key stages in developing a reasoning model in this book. This section focuses on loading pre-trained LLM in Stage 1.

As mentioned in the previous section, this book uses Qwen3 0.6B as a pre-trained base model. In this section, we load its pre-trained weights, as shown in figure 2.7. The "0.6B" in the model name indicates that the model contains approximately 0.6 billion weight parameters.

Why Qwen3? After carefully evaluating several open-weight base models, I chose Qwen3 0.6B for the following reasons:

- For this book, we want a small yet capable open-weight model that can run on consumer hardware.
- The larger variants of the Qwen3 model family are, as of this writing, the leading open-weight models in terms of modeling performance.
- Qwen3 0.6B is more memory-efficient compared to Llama 3 1B and OLMo 2 1B.
- Qwen3 offers both a base model (the focus of our reasoning model development) and an official reasoning variant that we can use as a reference for evaluation purposes.

NOTE The canonical spelling of "Qwen3" does not include whitespace, whereas "Llama 3" does.

In line with the spirit of building things "from scratch," this book uses a custom reimplementation of Qwen3 that I wrote in pure PyTorch, which is entirely independent of external LLM libraries. The emphasis of this reimplementation is on code readability and tweakability, in case you want to modify it later for your own experiments. Despite being built from scratch, this implementation remains fully compatible with the original pre-trained Qwen3 model weights.

However, this book does not cover the Qwen3 code implementation in depth. This topic alone would fill an entire separate book, similar to my other book, *Build A Large Language Model (From Scratch)*. Instead, this *Build A Reasoning Model (From Scratch)* book specifically focuses on implementing reasoning methods on top of a base model, in this case, Qwen3.

> **NOTE** This reimplemented Qwen3 LLM runs entirely locally, just like any other neural network implemented in PyTorch. There are no server-side components or external API calls involved. All model usage happens on your own machine, and your data stays on your device. If you are concerned about privacy, the setup we are using ensures full control over both the LLM inputs and outputs.

For those interested in additional details about Qwen3, as well as the model code, please see appendix C.

Before we load the model, we can specify the device we are going to use, namely, CPU or GPU. The following code will select the best-available device automatically:

```
def get_device():
    if torch.cuda.is_available():
        device = torch.device("cuda")
        print("Using NVIDIA CUDA GPU")
    elif torch.backends.mps.is_available():
        device = torch.device("mps")
        print("Using Apple Silicon GPU (MPS)")
    else:
        device = torch.device("cpu")
        print("Using CPU")
    return device

device = get_device()
```

While GPUs generally provide substantial speed and performance improvements, it can be helpful to initially run the remaining code in this chapter using the CPU for compatibility and debugging purposes. You can temporarily override the automatic selection by explicitly setting:

```
device = torch.device("cpu")
```

After finishing the chapter and verifying the code works properly on the CPU, remove or comment out the manual override and rerun the code. If your system has a GPU, you should then observe improved performance.

> **NOTE** The code in the remainder of this chapter was executed on a Mac Mini with an Apple M4 CPU. Performance comparisons with the Apple Silicon M4 GPU and the NVIDIA H100 GPU are included at the end of the chapter.

However, before we load the model and put it onto the selected `device`, we first need to download the weights for Qwen3 0.6B. These files are required to initialize the pre-trained model correctly:

```
download_qwen3_small(kind="base", tokenizer_only=False, out_dir="qwen3")
```

The output is as follows:

```
qwen3-0.6B-base.pth: 100% (1433 MiB / 1433 MiB)
✓ qwen3/tokenizer-base.json already up-to-date
```

(There is a checkmark in front of the tokenizer because we already downloaded it in the previous section.)

After downloading the model weights via the previous step, we can now instantiate a `Qwen3Model` class into which we load the pre-trained weights via PyTorch's `load_state_dict` method:

```
from reasoning_from_scratch.qwen3 import Qwen3Model, QWEN_CONFIG_06_B

model_file = Path("qwen3") / "qwen3-0.6B-base.pth"
model = Qwen3Model(QWEN_CONFIG_06_B)                                #A
model.load_state_dict(torch.load(model_file))                       #B
model.to(device)                                                    #C
```

#A Instantiate a Qwen3 model with random weights as placeholders
#B Load the pre-trained weights into the model
#C Transfer the model to the designated device (e.g., "cuda")

Note that if the device setting is `"cpu"`, the `model.to(device)` operation will be skipped because the model already sits in CPU memory by default.

After executing the code above, you should see the following output:

```
Qwen3Model(
  (tok_emb): Embedding(151936, 1024)
  (trf_blocks): ModuleList(
    (0-27): 28 x TransformerBlock(
      (att): GroupedQueryAttention(
        (W_query): Linear(in_features=1024, out_features=2048, bias=False)
        (W_key): Linear(in_features=1024, out_features=1024, bias=False)
        (W_value): Linear(in_features=1024, out_features=1024, bias=False)
        (out_proj): Linear(in_features=2048, out_features=1024, bias=False)
        (q_norm): RMSNorm()
        (k_norm): RMSNorm()
      )
      (ff): FeedForward(
        (fc1): Linear(in_features=1024, out_features=3072, bias=False)
        (fc2): Linear(in_features=1024, out_features=3072, bias=False)
        (fc3): Linear(in_features=3072, out_features=1024, bias=False)
      )
      (norm1): RMSNorm()
      (norm2): RMSNorm()
    )
  )
  (final_norm): RMSNorm()
  (out_head): Linear(in_features=1024, out_features=151936, bias=False)
)
```

This output is a summary of the Qwen3 0.6B base model architecture, as printed by PyTorch. It highlights the model's core components: an embedding layer, a stack of 28 transformer blocks, and a final linear projection head. Each transformer block includes a grouped-query attention mechanism and a multi-layer feedforward network, along with normalization layers throughout.

These components are also illustrated visually in figure 2.8 for readers familiar with LLM architectures. However, a detailed understanding of this architecture is not required for this book. Since we are not modifying the base model itself, but rather building reasoning methods on top of it, you can safely treat the architecture as a black box for now. However, interested readers can optionally find more information on these components in appendix C.

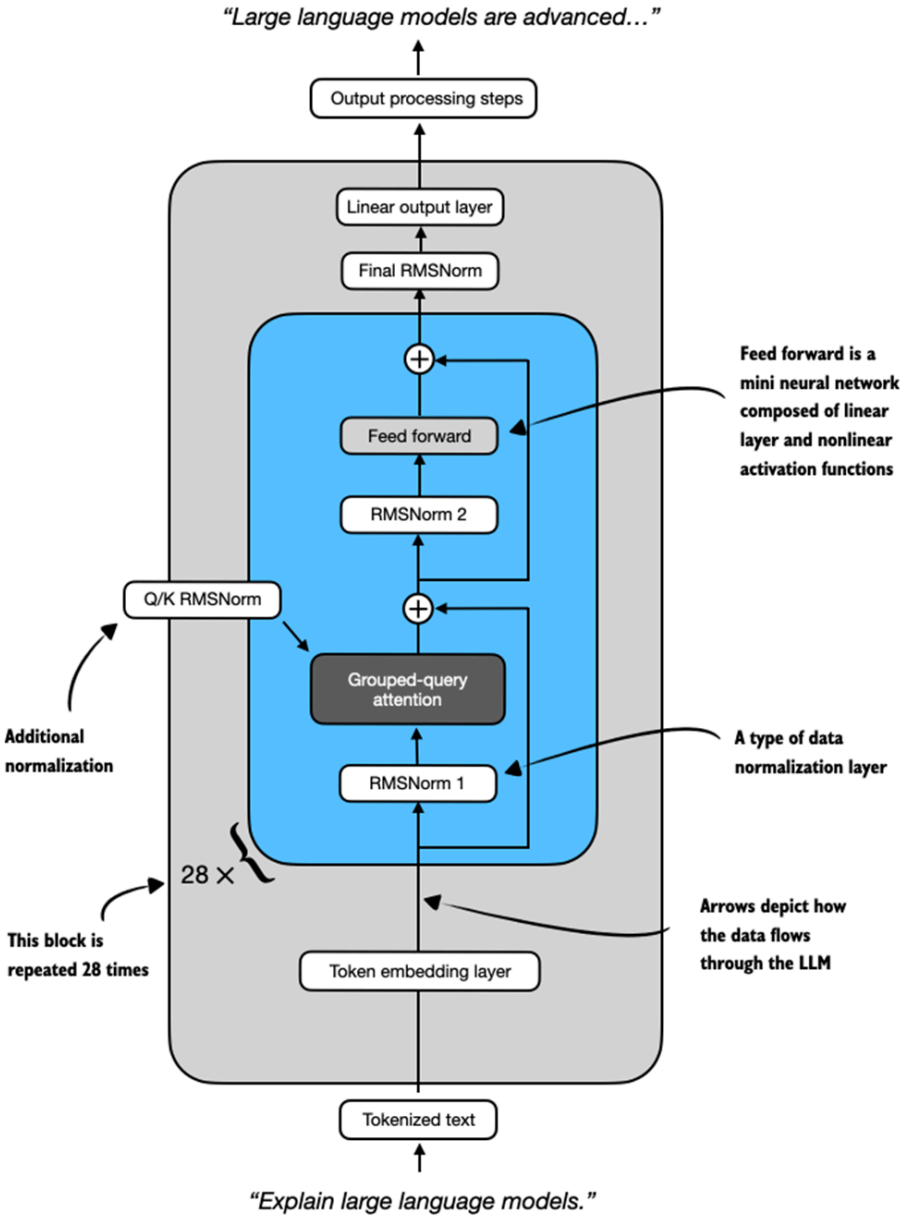

Figure 2.8 Overview of the Qwen3 0.6B model architecture. Input text is tokenized and passed through an embedding layer, followed by 28 repeated transformer blocks. Each block contains grouped-query attention, feedforward layers, and RMS normalization. The model ends with a final normalization and linear output layer. Arrows show the data flow through the model.

The key takeaway from this section is that we have now loaded a pre-trained model, with its architecture shown in figure 2.8, that should be capable of generating coherent text. In the next section, we will code a text generation function that feeds tokenized data into the model and returns the response in a human-readable format.

2.6 Understanding the sequential LLM text generation process

After loading a pre-trained LLM, our goal is to write a function that leverages the LLM to generate text. This function forms the foundation for reasoning-improving methods that we will implement later in the book, as shown in figure 2.9.

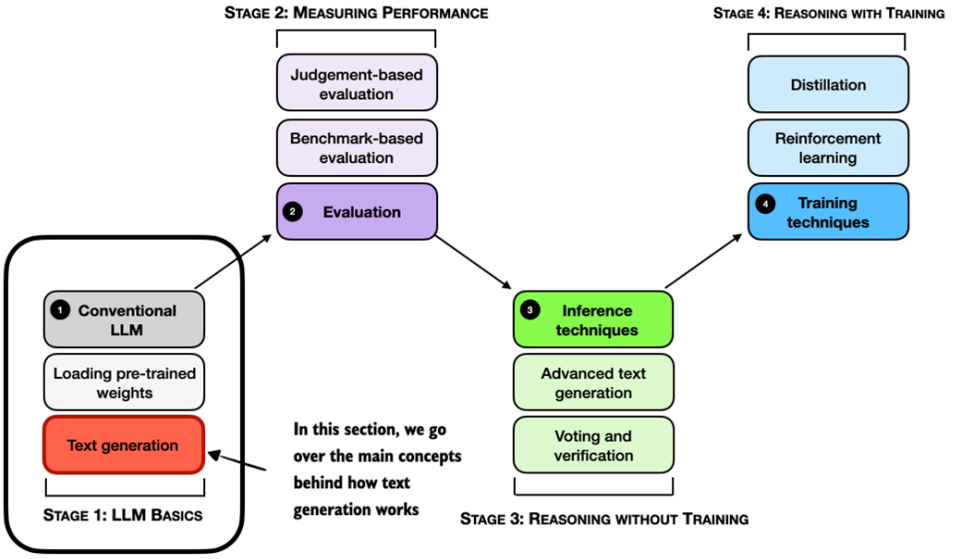

Figure 2.9 An overview of the four key stages in developing a reasoning model in this book. This section explains the main concept behind text generation in LLMs, which allows us to implement a text generation function for using the pre-trained LLM in the remainder of this chapter.

However, before we get to implement this text generation function that we will use in this and upcoming chapters (as shown in figure 2.9), let's go over the basic concepts behind text generation in LLMs.

You may already know that text generation in LLMs is a sequential process where LLMs generate one word at a time. This is often also called *autoregressive* text generation and is shown in figure 2.10

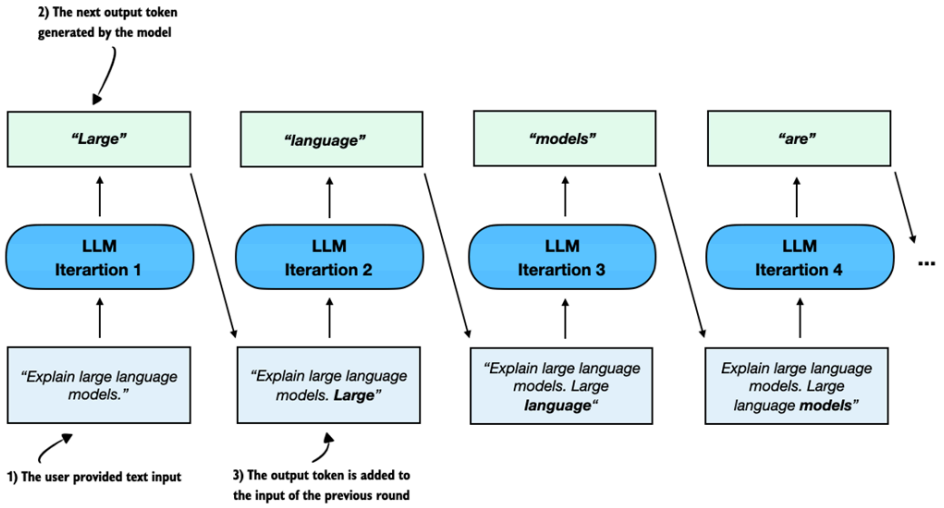

Figure 2.10 An illustration of the sequential (autoregressive) text generation in LLMs. At each iteration, the model generates the next token based on the input and previously generated tokens, which are cumulatively fed back into the model to produce coherent output.

Note that the sequential text generation process shown in figure 2.10 is a broad overview. The figure shows one generated output token (top row) at each step, when feeding it with an input prompt. This is done for simplicity to explain the main concept behind LLM-based text generation.

Now, if we look at one of these iterations more closely, an LLM generates one output token for each input token. This means that if we have six input tokens, the LLM returns six output tokens, as illustrated in figure 2.11. However, it is important to note that we only care about the last generated token in each iteration.

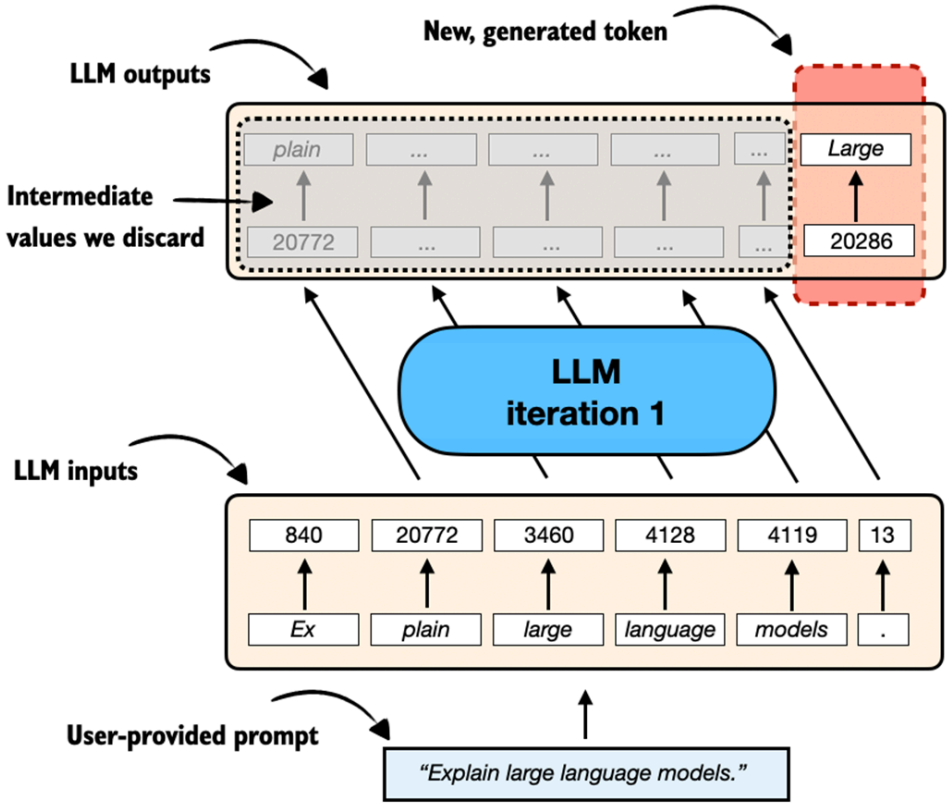

Figure 2.11 A closer look at a single iteration of the autoregressive text generation process. The LLM generates an output sequence that mirrors the input but is shifted one position to the right. At each iteration, the model predicts the next token in the sequence, The LLM effectively learns to continue the input prompt one token at a time.

Before implementing a text-generation function that uses the concept shown in figure 2.11 for each iteration to implement the autoregressive text generation process shown in figure 2.10, let's take a look at a code example to illustrate figure 2.11 further by reusing the "Explain large language models." example prompt from section 2.4:

```
prompt = "Explain large language models."
input_token_ids_list = tokenizer.encode(prompt)
print(f"Number of input tokens: {len(input_token_ids_list)}")

input_tensor = torch.tensor(input_token_ids_list)                #A
input_tensor_fmt = input_tensor.unsqueeze(0)                     #B

output_tensor = model(input_tensor_fmt)                          #C
output_tensor_fmt = output_tensor.squeeze(0)                     #D
print(f"Formatted Output tensor shape: {output_tensor_fmt.shape}")
```

#A Convert Python list into PyTorch tensor
#B Add an additional dimension
#C Generate the output
#D Remove the extra dimension

SQUEEZING AND UNSQUEEZING TENSORS

The `.squeeze()` and `.unsqueeze()` operations in PyTorch are used to change the shape of a tensor by removing or adding dimensions of size 1. This is often useful for reshaping a tensor to match what a model expects. For example, a model might expect input tensors with two dimensions (e.g., rows and columns) so it can process batches of inputs (see appendix E). But if the input is just a row vector, we can use `.unsqueeze(0)` to add an extra dimension and make it compatible:

```
example = torch.tensor([1, 2, 3])
print(example)
print(example.unsqueeze(0))
```

This returns:

```
tensor([1, 2, 3])
tensor([[1, 2, 3]])
```

Here, `.unsqueeze(0)` adds a new dimension at position 0, turning a 1D tensor into a 2D tensor with shape `(1, 3)`. Conversely, `.squeeze(0)` removes a dimension of size 1 from position 0:

```
example = torch.tensor([[1, 2, 3]])
print(example)
print(example.squeeze(0))
```

This returns:

```
tensor([[1, 2, 3]])
tensor([1, 2, 3])
```

This is useful when you want to remove extra dimensions that are not needed.

The output from the previous code example is follows:

```
Number of input tokens: 6
Formatted Output tensor shape: torch.Size([6, 151936])
```

As we can see, we feed six input tokens into the model, which returns a 6×151,936-dimensional matrix. The 6 in this matrix corresponds to the six input tokens. The second dimension, 151,936, corresponds to the vocabulary size that the model supports. For instance, each of the six tokens is represented by a vector with 151,936 values. We can think of the values in these vectors as scores for each possible word in the vocabulary, where the highest score corresponds to the most likely word or subword (in the 151,936-entry vocabulary) to be chosen as the generated token.

So, to get the next generated word, we extract the last row of this 6×151,936-dimensional matrix, find the token ID corresponding to the largest score value in this row, and convert this token ID back into text via the tokenizer, as illustrated in figure 2.12.

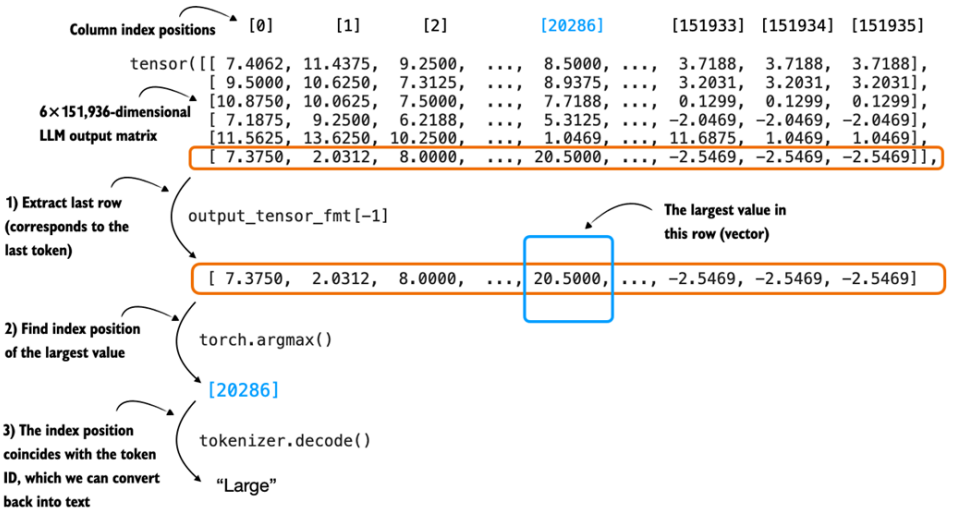

Figure 2.12 A closer look at how the raw scores output by an LLM, in a single text generation iteration, are converted into a token ID and its corresponding text representation.

Let's see how we can convert the LLM output matrix into the generated text token (shown in figure 2.12) in code:

Note that LLMs are trained with a next-word prediction task, and as shown in figure 2.11, we are only interested in the last token, which we can obtain via the `[-1]` index:

```
last_token = output_tensor_fmt[-1].detach()
print(last_token)
```

Here, `.detach()` separates the tensor from the part of the system that tracks how the model learns. In simple terms, it lets us take the last token from the model's output and use it for the next step without keeping extra information we don't need during generation. This saves memory and can make things run faster.

This prints the 151,936 values corresponding to the last token:

```
tensor([ 7.3750,  2.0312,  8.0000,  ..., -2.5469, -2.5469, -2.5469],
       dtype=torch.bfloat16)
```

Then, we can use the argmax function to obtain the position with the largest value score (value) in this tensor:

```
print(last_token.argmax(dim=-1, keepdim=True))
```

The result is:

```
tensor([20286])
```

This returned integer value is the position of the largest value in this vector, and it also corresponds to the token ID of the generated token (`last_token`), which we can translate back into text via the tokenizer:

```
print(tokenizer.decode([20286]))
```

This prints the generated token:

```
 Large
```

SQUEEZING AND UNSQUEEZING TENSORS

The `.squeeze()` and `.unsqueeze()` operations in PyTorch are used to change the shape of a tensor by removing or adding dimensions of size 1. This is

MAX VERSUS ARGMAX

The `torch.max()` and `torch.argmax()` functions in PyTorch are used to find the largest value in a tensor and the index of that value. For example:

```
example = torch.tensor([-2, 1, 3, 1])
print(torch.max(example))
print(torch.argmax(example))
```

This returns:

```
tensor(3)
tensor(2)
```

The maximum value is 3, and it first appears at index 2.

We can also use `keepdim=True` with `torch.argmax()` to keep the output shape consistent by retaining the reduced dimension:

```
print(torch.argmax(example, keepdim=True))
```

This returns:

```
tensor([2])
```

Here, `keepdim=True` keeps the result as a 1D tensor with the same number of dimensions as the input, which can be helpful for keeping the shape required by the tokenizer and for concatenation later on in our text generation function.

To recap, figure 2.10 illustrated the iterative (autoregressive) text generation process in an LLM. Then, figure 2.11 zooms in on one of the iterations in this process. Figure 2.12 then further zoomed into this one iteration and shows how the score matrix (output by an LLM), gets converted into a token ID (and its corresponding text representation).