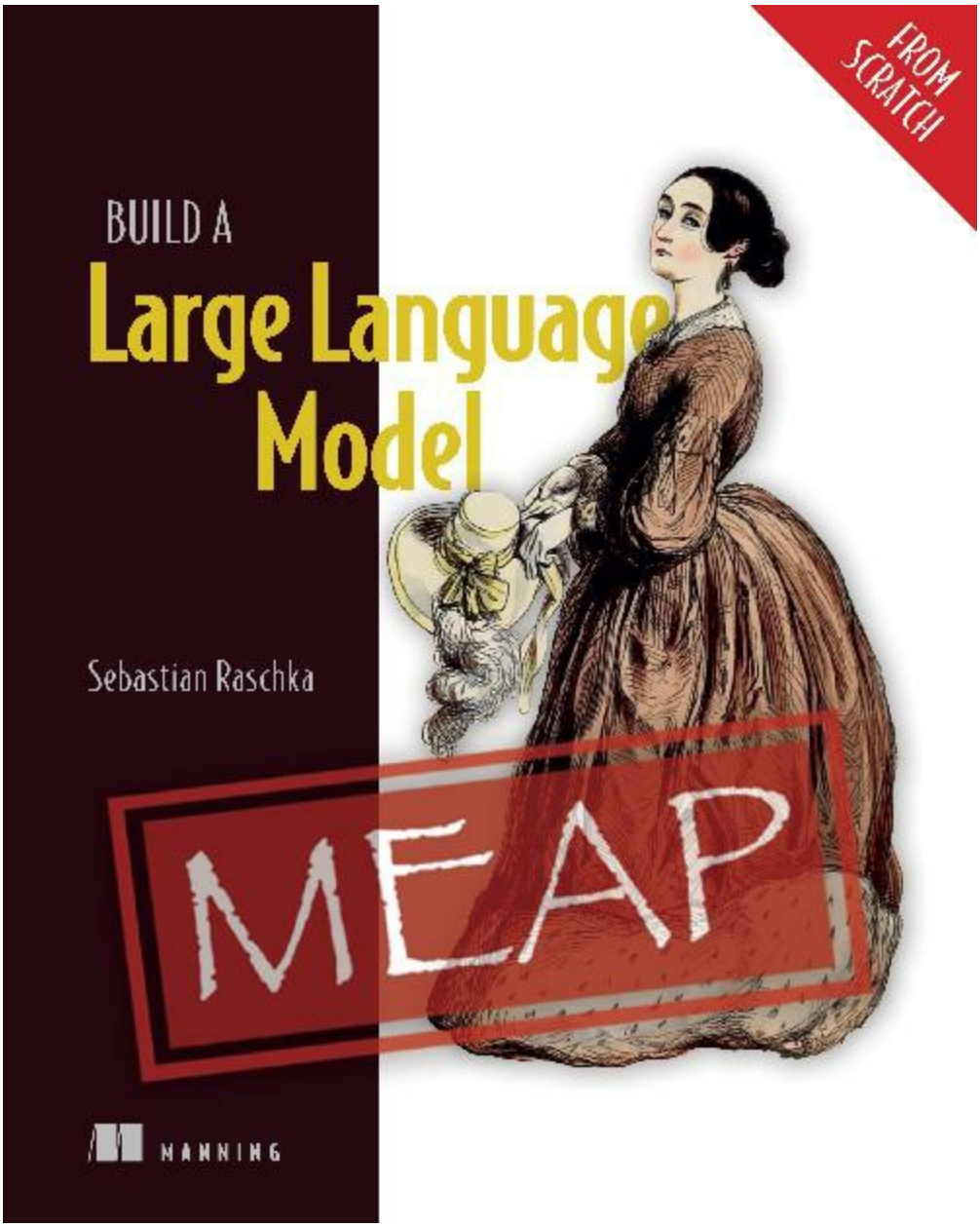
FROM SCRATCH

BUILD A Large Language Model

Sebastian Raschka

MEAP

MANNING


FROM SCRATCH
BUILD A
Large Language Model
Sebastian Raschka
MEAP
MANNING

Build a Large Language Model (From Scratch)

welcome

Thank you for purchasing the MEAP edition of *Build a Large Language Model (From Scratch)*.

In this book, I invite you to embark on an educational journey with me to learn how to build Large Language Models (LLMs) from the ground up. Together, we'll delve deep into the LLM training pipeline, starting from data loading and culminating in finetuning LLMs on custom datasets.

For many years, I've been deeply immersed in the world of deep learning, coding LLMs, and have found great joy in explaining complex concepts thoroughly. This book has been a long-standing idea in my mind, and I'm thrilled to finally have the opportunity to write it and share it with you. Those of you familiar with my work, especially from my blog, have likely seen glimpses of my approach to coding from scratch. This method has resonated well with many readers, and I hope it will be equally effective for you.

I've designed the book to emphasize hands-on learning, primarily using PyTorch and without relying on pre-existing libraries. With this approach, coupled with numerous figures and illustrations, I aim to provide you with a thorough understanding of how LLMs work, their limitations, and customization methods. Moreover, we'll explore commonly used workflows and paradigms in pretraining and fine-tuning LLMs, offering insights into their development and customization.

The book is structured with detailed step-by-step introductions, ensuring no critical detail is overlooked. To gain the most from this book, you should have a background in Python programming. Prior experience in deep learning and a foundational understanding of PyTorch, or familiarity with other deep learning frameworks like TensorFlow, will be beneficial.

I warmly invite you to engage in the liveBook discussion forum for any questions, suggestions, or feedback you might have. Your contributions are immensely valuable and appreciated in enhancing this learning journey.

— Sebastian Raschka

In this book

1 Understanding Large Language Models

This chapter covers

- High-level explanations of the fundamental concepts behind large language models (LLMs)
- Insights into the transformer architecture from which ChatGPT-like LLMs are derived
- A plan for building an LLM from scratch

Large language models (LLMs), such as those offered in OpenAI's ChatGPT, are deep neural network models that have been developed over the past few years. They ushered in a new era for Natural Language Processing (NLP). Before the advent of large language models, traditional methods excelled at categorization tasks such as email spam classification and straightforward pattern recognition that could be captured with handcrafted rules or simpler models. However, they typically underperformed in language tasks that demanded complex understanding and generation abilities, such as parsing detailed instructions, conducting contextual analysis, or creating coherent and contextually appropriate original text. For example, previous generations of language models could not write an email from a list of keywords—a task that is trivial for contemporary LLMs.

LLMs have remarkable capabilities to understand, generate, and interpret human language. However, it's important to clarify that when we say language models "understand," we mean that they can process and generate text in ways that appear coherent and contextually relevant, not that they possess human-like consciousness or comprehension.

Enabled by advancements in deep learning, which is a subset of machine learning and artificial intelligence (AI) focused on neural networks, LLMs are trained on vast quantities of text data. This allows LLMs to capture deeper contextual information and subtleties of human language compared to

previous approaches. As a result, LLMs have significantly improved performance in a wide range of NLP tasks, including text translation, sentiment analysis, question answering, and many more.

Another important distinction between contemporary LLMs and earlier NLP models is that the latter were typically designed for specific tasks; whereas those earlier NLP models excelled in their narrow applications, LLMs demonstrate a broader proficiency across a wide range of NLP tasks.

The success behind LLMs can be attributed to the transformer architecture which underpins many LLMs, and the vast amounts of data LLMs are trained on, allowing them to capture a wide variety of linguistic nuances, contexts, and patterns that would be challenging to manually encode.

This shift towards implementing models based on the transformer architecture and using large training datasets to train LLMs has fundamentally transformed NLP, providing more capable tools for understanding and interacting with human language.

Beginning with this chapter, we set the foundation to accomplish the primary objective of this book: understanding LLMs by implementing a ChatGPT-like LLM based on the transformer architecture step by step in code.

1.1 What is an LLM?

An LLM, a large language model, is a neural network designed to understand, generate, and respond to human-like text. These models are deep neural networks trained on massive amounts of text data, sometimes encompassing large portions of the entire publicly available text on the internet.

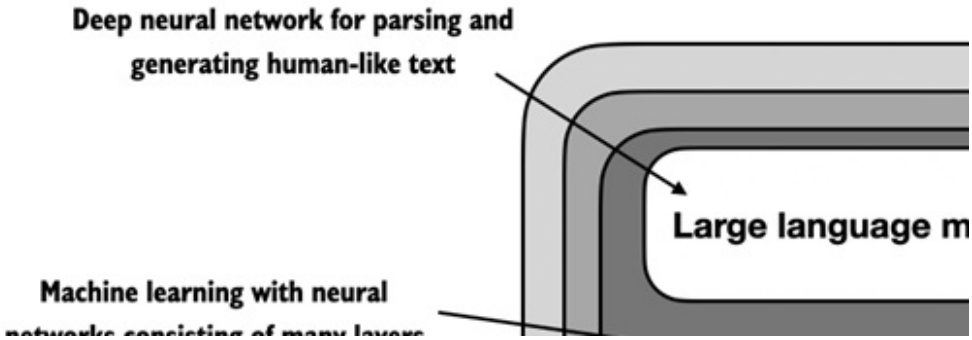
The "large" in large language model refers to both the model's size in terms of parameters and the immense dataset on which it's trained. Models like this often have tens or even hundreds of billions of parameters, which are the adjustable weights in the network that are optimized during training to predict the next word in a sequence. Next-word prediction is sensible because it harnesses the inherent sequential nature of language to train models on

understanding context, structure, and relationships within text. Yet, it is a very simple task and so it is surprising to many researchers that it can produce such capable models. We will discuss and implement the next-word training procedure in later chapters step by step.

LLMs utilize an architecture called the *transformer* (covered in more detail in section 1.4), which allows them to pay selective attention to different parts of the input when making predictions, making them especially adept at handling the nuances and complexities of human language.

Since LLMs are capable of *generating* text, LLMs are also often referred to as a form of generative artificial intelligence (AI), often abbreviated as *generative AI* or *GenAI*. As illustrated in Figure 1.1, AI encompasses the broader field of creating machines that can perform tasks requiring human-like intelligence, including understanding language, recognizing patterns, and making decisions, and includes subfields like machine learning and deep learning.

Figure 1.1 As this hierarchical depiction of the relationship between the different fields suggests, LLMs represent a specific application of deep learning techniques, leveraging their ability to process and generate human-like text. Deep learning is a specialized branch of machine learning that focuses on using multi-layer neural networks. And machine learning and deep learning are fields aimed at implementing algorithms that enable computers to learn from data and perform tasks that typically require human intelligence.



The algorithms used to implement AI are the focus of the field of machine learning. Specifically, machine learning involves the development of algorithms that can learn from and make predictions or decisions based on data without being explicitly programmed. To illustrate this, imagine a spam filter as a practical application of machine learning. Instead of manually

writing rules to identify spam emails, a machine learning algorithm is fed examples of emails labeled as spam and legitimate emails. By minimizing the error in its predictions on a training dataset, the model then learns to recognize patterns and characteristics indicative of spam, enabling it to classify new emails as either spam or legitimate.

As illustrated in Figure 1.1, deep learning is a subset of machine learning that focuses on utilizing neural networks with three or more layers (also called deep neural networks) to model complex patterns and abstractions in data. In contrast to deep learning, traditional machine learning requires manual feature extraction. This means that human experts need to identify and select the most relevant features for the model.

While the field of AI is nowadays dominated by machine learning and deep learning, it also includes other approaches, for example, using rule-based systems, genetic algorithms, expert systems, fuzzy logic, or symbolic reasoning.

Returning to the spam classification example, in traditional machine learning, human experts might manually extract features from email text such as the frequency of certain trigger words ("prize," "win," "free"), the number of exclamation marks, use of all uppercase words, or the presence of suspicious links. This dataset, created based on these expert-defined features, would then be used to train the model. In contrast to traditional machine learning, deep learning does not require manual feature extraction. This means that human experts do not need to identify and select the most relevant features for a deep learning model. (However, in both traditional machine learning and deep learning for spam classification, you still require the collection of labels, such as spam or non-spam, which need to be gathered either by an expert or users.)

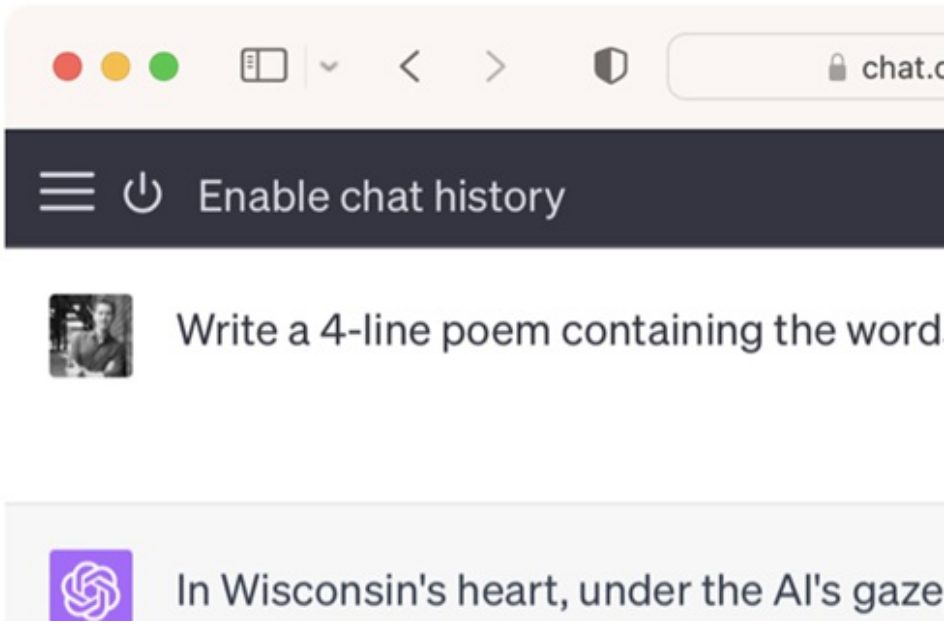
The upcoming sections will cover some of the problems LLMs can solve today, the challenges that LLMs address, and the general LLM architecture, which we will implement in this book.

1.2 Applications of LLMs

Owing to their advanced capabilities to parse and understand unstructured

text data, LLMs have a broad range of applications across various domains. Today, LLMs are employed for machine translation, generation of novel texts (see Figure 1.2), sentiment analysis, text summarization, and many other tasks. LLMs have recently been used for content creation, such as writing fiction, articles, and even computer code.

Figure 1.2 LLM interfaces enable natural language communication between users and AI systems. This screenshot shows ChatGPT writing a poem according to a user's specifications.



LLMs can also power sophisticated chatbots and virtual assistants, such as OpenAI's ChatGPT or Google's Gemini (formerly called Bard), which can answer user queries and augment traditional search engines such as Google Search or Microsoft Bing.

Moreover, LLMs may be used for effective knowledge retrieval from vast volumes of text in specialized areas such as medicine or law. This includes sifting through documents, summarizing lengthy passages, and answering technical questions.

In short, LLMs are invaluable for automating almost any task that involves parsing and generating text. Their applications are virtually endless, and as we continue to innovate and explore new ways to use these models, it's clear

that LLMs have the potential to redefine our relationship with technology, making it more conversational, intuitive, and accessible.

In this book, we will focus on understanding how LLMs work from the ground up, coding an LLM that can generate texts. We will also learn about techniques that allow LLMs to carry out queries, ranging from answering questions to summarizing text, translating text into different languages, and more. In other words, in this book, we will learn how complex LLM assistants such as ChatGPT work by building one step by step.

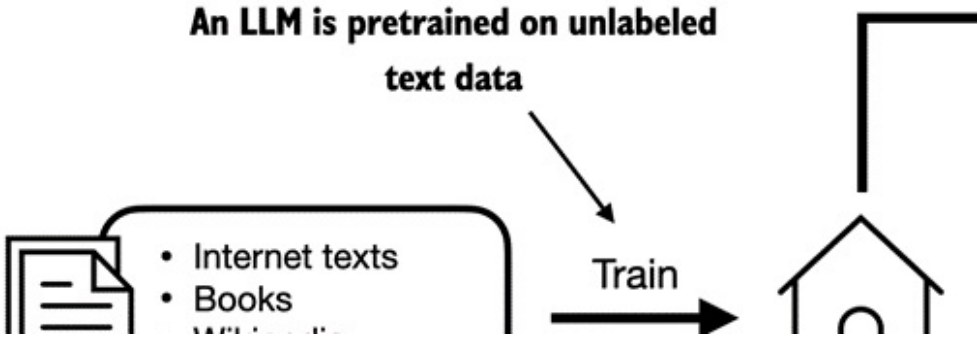
1.3 Stages of building and using LLMs

Why should we build our own LLMs? Coding an LLM from the ground up is an excellent exercise to understand its mechanics and limitations. Also, it equips us with the required knowledge for pretraining or finetuning existing open-source LLM architectures to our own domain-specific datasets or tasks.

Research has shown that when it comes to modeling performance, custom-built LLMs—those tailored for specific tasks or domains—can outperform general-purpose LLMs, such as those provided by ChatGPT, which are designed for a wide array of applications. Examples of this include BloombergGPT, which is specialized for finance, and LLMs that are tailored for medical question answering (please see the *Further Reading and References* section in Appendix B for more details).

The general process of creating an LLM includes pretraining and finetuning. The term "pre" in "pretraining" refers to the initial phase where a model like an LLM is trained on a large, diverse dataset to develop a broad understanding of language. This pretrained model then serves as a foundational resource that can be further refined through finetuning, a process where the model is specifically trained on a narrower dataset that is more specific to particular tasks or domains. This two-stage training approach consisting of pretraining and finetuning is depicted in Figure 1.3.

Figure 1.3 Pretraining an LLM involves next-word prediction on large text datasets. A pretrained LLM can then be finetuned using a smaller labeled dataset.



As illustrated in Figure 1.3, the first step in creating an LLM is to train it in on a large corpus of text data, sometimes referred to as *raw* text. Here, "raw" refers to the fact that this data is just regular text without any labeling information[1]. (Filtering may be applied, such as removing formatting characters or documents in unknown languages.)

This first training stage of an LLM is also known as *pretraining*, creating an initial pretrained LLM, often called a *base* or *foundation model*. A typical example of such a model is the GPT-3 model (the precursor of the original model offered in ChatGPT). This model is capable of text completion, that is, finishing a half-written sentence provided by a user. It also has limited few-shot capabilities, which means it can learn to perform new tasks based on only a few examples instead of needing extensive training data. This is further illustrated in the next section, *Using transformers for different tasks*.

After obtaining a *pretrained* LLM from training on large text datasets, where the LLM is trained to predict the next word in the text, we can further train the LLM on labeled data, also known as *finetuning*.

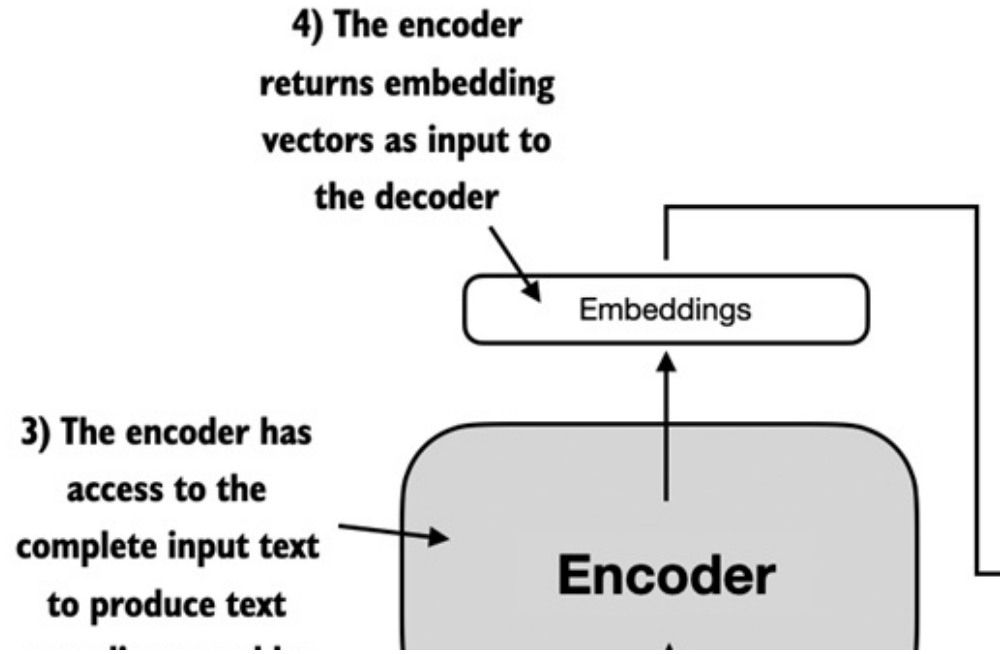
The two most popular categories of finetuning LLMs include *instruction-finetuning* and finetuning for *classification* tasks. In instruction-finetuning, the labeled dataset consists of instruction and answer pairs, such as a query to translate a text accompanied by the correctly translated text. In classification finetuning, the labeled dataset consists of texts and associated class labels, for example, emails associated with *spam* and *non-spam* labels.

In this book, we will cover both code implementations for pretraining and finetuning LLM, and we will delve deeper into the specifics of instruction-finetuning and finetuning for classification later in this book after pretraining a base LLM.

1.4 Using LLMs for different tasks

Most modern LLMs rely on the *transformer* architecture, which is a deep neural network architecture introduced in the 2017 paper *Attention Is All You Need*. To understand LLMs we briefly have to go over the original transformer, which was originally developed for machine translation, translating English texts to German and French. A simplified version of the transformer architecture is depicted in Figure 1.4.

Figure 1.4 A simplified depiction of the original transformer architecture, which is a deep learning model for language translation. The transformer consists of two parts, an encoder that processes the input text and produces an embedding representation (a numerical representation that captures many different factors in different dimensions) of the text that the decoder can use to generate the translated text one word at a time. Note that this figure shows the final stage of the translation process where the decoder has to generate only the final word ("Beispiel"), given the original input text ("This is an example") and a partially translated sentence ("Das ist ein"), to complete the translation.



The transformer architecture depicted in Figure 1.4 consists of two submodules, an encoder and a decoder. The encoder module processes the input text and encodes it into a series of numerical representations or vectors that capture the contextual information of the input. Then, the decoder module takes these encoded vectors and generates the output text from them. In a translation task, for example, the encoder would encode the text from the source language into vectors, and the decoder would decode these vectors to generate text in the target language. Both the encoder and decoder consist of many layers connected by a so-called self-attention mechanism. You may have many questions regarding how the inputs are preprocessed and encoded. These will be addressed in a step-by-step implementation in the subsequent chapters.

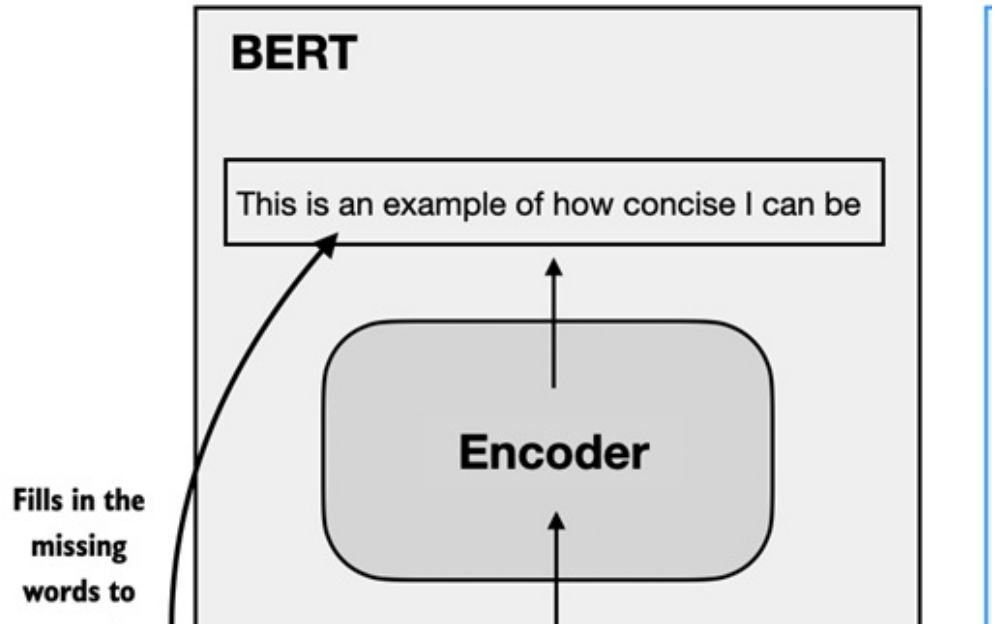
A key component of transformers and LLMs is the self-attention mechanism (not shown), which allows the model to weigh the importance of different words or tokens in a sequence relative to each other. This mechanism enables the model to capture long-range dependencies and contextual relationships within the input data, enhancing its ability to generate coherent and contextually relevant output. However, due to its complexity, we will defer the explanation to chapter 3, where we will discuss and implement it step by

step. Moreover, we will also discuss and implement the data preprocessing steps to create the model inputs in *chapter 2, Working with Text Data*.

Later variants of the transformer architecture, such as the so-called BERT (short for *bidirectional encoder representations from transformers*) and the various GPT models (short for *generative pretrained transformers*), built on this concept to adapt this architecture for different tasks. (References can be found in Appendix B.)

BERT, which is built upon the original transformer's encoder submodule, differs in its training approach from GPT. While GPT is designed for generative tasks, BERT and its variants specialize in masked word prediction, where the model predicts masked or hidden words in a given sentence as illustrated in Figure 1.5. This unique training strategy equips BERT with strengths in text classification tasks, including sentiment prediction and document categorization. As an application of its capabilities, as of this writing, Twitter uses BERT to detect toxic content.

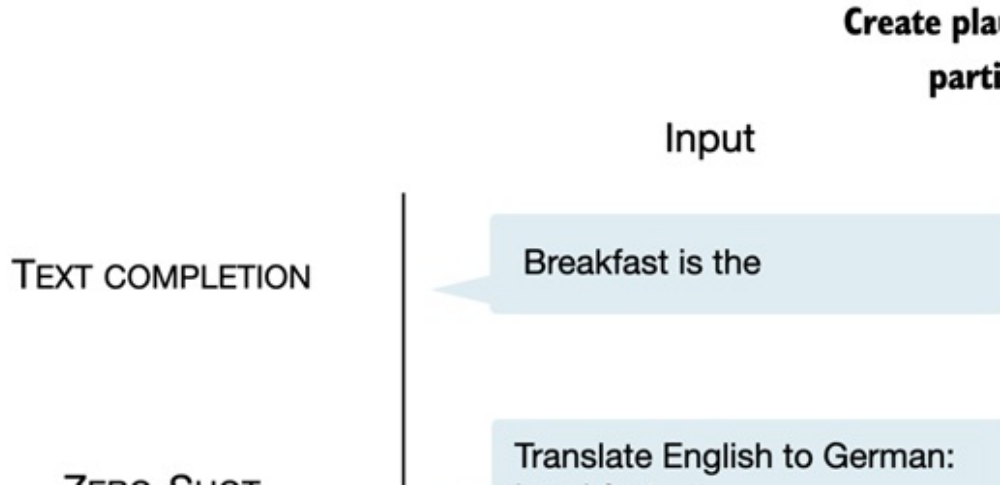
Figure 1.5 A visual representation of the transformer's encoder and decoder submodules. On the left, the encoder segment exemplifies BERT-like LLMs, which focus on masked word prediction and are primarily used for tasks like text classification. On the right, the decoder segment showcases GPT-like LLMs, designed for generative tasks and producing coherent text sequences.



GPT, on the other hand, focuses on the decoder portion of the original transformer architecture and is designed for tasks that require generating texts. This includes machine translation, text summarization, fiction writing, writing computer code, and more. We will discuss the GPT architecture in more detail in the remaining sections of this chapter and implement it from scratch in this book.

GPT models, primarily designed and trained to perform text completion tasks, also show remarkable versatility in their capabilities. These models are adept at executing both zero-shot and few-shot learning tasks. Zero-shot learning refers to the ability to generalize to completely unseen tasks without any prior specific examples. On the other hand, few-shot learning involves learning from a minimal number of examples the user provides as input, as shown in Figure 1.6.

Figure 1.6 In addition to text completion, GPT-like LLMs can solve various tasks based on their inputs without needing retraining, finetuning, or task-specific model architecture changes. Sometimes, it is helpful to provide examples of the target within the input, which is known as a few-shot setting. However, GPT-like LLMs are also capable of carrying out tasks without a specific example, which is called zero-shot setting.



Transformers versus LLMs

Today's LLMs are based on the transformer architecture introduced in the previous section. Hence, transformers and LLMs are terms that are often used synonymously in the literature. However, note that not all transformers are

LLMs since transformers can also be used for computer vision. Also, not all LLMs are transformers, as there are large language models based on recurrent and convolutional architectures. The main motivation behind these alternative approaches is to improve the computational efficiency of LLMs. However, whether these alternative LLM architectures can compete with the capabilities of transformer-based LLMs and whether they are going to be adopted in practice remains to be seen. (Interested readers can find literature references describing these architectures in the *Further Reading* section at the end of this chapter.)

1.5 Utilizing large datasets

The large training datasets for popular GPT- and BERT-like models represent diverse and comprehensive text corpora encompassing billions of words, which include a vast array of topics and natural and computer languages. To provide a concrete example, Table 1.1 summarizes the dataset used for pretraining GPT-3, which served as the base model for the first version of ChatGPT.

Table 1.1 The pretraining dataset of the popular GPT-3 LLM

Dataset name	Dataset description	Number of tokens	Proportion in training data
CommonCrawl (filtered)	Web crawl data	410 billion	60%
WebText2	Web crawl data	19 billion	22%
Books1	Internet-based book corpus	12 billion	8%
Books2	Internet-based book corpus	55 billion	8%
Wikipedia	High-quality text	3 billion	3%

Table 1.1 reports the number of tokens, where a token is a unit of text that a

model reads, and the number of tokens in a dataset is roughly equivalent to the number of words and punctuation characters in the text. We will cover tokenization, the process of converting text into tokens, in more detail in the next chapter.

The main takeaway is that the scale and diversity of this training dataset allows these models to perform well on diverse tasks including language syntax, semantics, and context, and even some requiring general knowledge.

GPT-3 dataset details

In Table 1.1, it's important to note that from each dataset, only a fraction of the data, (amounting to a total of 300 billion tokens) was used in the training process. This sampling approach means that the training didn't encompass every single piece of data available in each dataset. Instead, a selected subset of 300 billion tokens, drawn from all datasets combined, was utilized. Also, while some datasets were not entirely covered in this subset, others might have been included multiple times to reach the total count of 300 billion tokens. The column indicating proportions in the table, when summed up without considering rounding errors, accounts for 100% of this sampled data.

For context, consider the size of the CommonCrawl dataset, which alone consists of 410 billion tokens and requires about 570 GB of storage. In comparison, later iterations of models like GPT-3, such as Meta's LLaMA, have expanded their training scope to include additional data sources like Arxiv research papers (92 GB) and StackExchange's code-related Q&As (78 GB).

The *Wikipedia* corpus consists of *English-language Wikipedia. While the authors of the GPT-3 paper didn't further specify the details,* Books1 is likely a sample from Project Gutenberg (https://www.gutenberg.org/), and *Books*2 is likely from *Libge*n (https://en.wikipedia.org/wiki/Library_Genesis). *CommonCrawl* is a filtered subset of the CommonCrawl database (https://commoncrawl.org/), and *WebText2* is the text of web pages from all outbound Reddit links from posts with 3+ upvotes.

The authors of the GPT-3 paper did not share the training dataset but a comparable dataset that is publicly available is *The Pile*

(https://pile.eleuther.ai/). However, the collection may contain copyrighted works, and the exact usage terms may depend on the intended use case and country. For more information, see the HackerNews discussion at https://news.ycombinator.com/item?id=25607809.

The pretrained nature of these models makes them incredibly versatile for further finetuning on downstream tasks, which is why they are also known as base or foundation models. Pretraining LLMs requires access to significant resources and is very expensive. For example, the GPT-3 pretraining cost is estimated to be $4.6 million in terms of cloud computing credits[2].

The good news is that many pretrained LLMs, available as open-source models, can be used as general purpose tools to write, extract, and edit texts that were not part of the training data. Also, LLMs can be finetuned on specific tasks with relatively smaller datasets, reducing the computational resources needed and improving performance on the specific task.

In this book, we will implement the code for pretraining and use it to pretrain an LLM for educational purposes. All computations will be executable on consumer hardware. After implementing the pretraining code we will learn how to reuse openly available model weights and load them into the architecture we will implement, allowing us to skip the expensive pretraining stage when we finetune LLMs later in this book.

1.6 A closer look at the GPT architecture

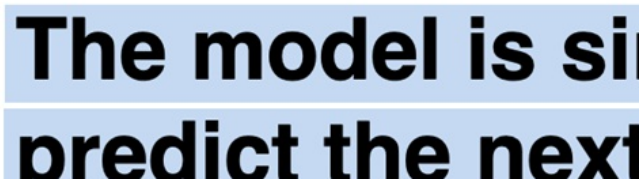
Previously in this chapter, we mentioned the terms GPT-like models, GPT-3, and ChatGPT. Let's now take a closer look at the general GPT architecture. First, GPT stands for ***G***enerative ***P***retrained ***T***ransformer and was originally introduced in the following paper:

- *Improving Language Understanding by Generative Pre-Training* (2018) by *Radford et al.* from OpenAI, http://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

GPT-3 is a scaled-up version of this model that has more parameters and was trained on a larger dataset. And the original model offered in ChatGPT was

created by finetuning GPT-3 on a large instruction dataset using a method from OpenAI's InstructGPT paper, which we will cover in more detail in *chapter 7, Finetuning with Human Feedback To Follow Instructions*. As we have seen earlier in Figure 1.6, these models are competent text completion models and can carry out other tasks such as spelling correction, classification, or language translation. This is actually very remarkable given that GPT models are pretrained on a relatively simple next-word prediction task, as illustrated in Figure 1.7.

Figure 1.7 In the next-word pretraining task for GPT models, the system learns to predict the upcoming word in a sentence by looking at the words that have come before it. This approach helps the model understand how words and phrases typically fit together in language, forming a foundation that can be applied to various other tasks.

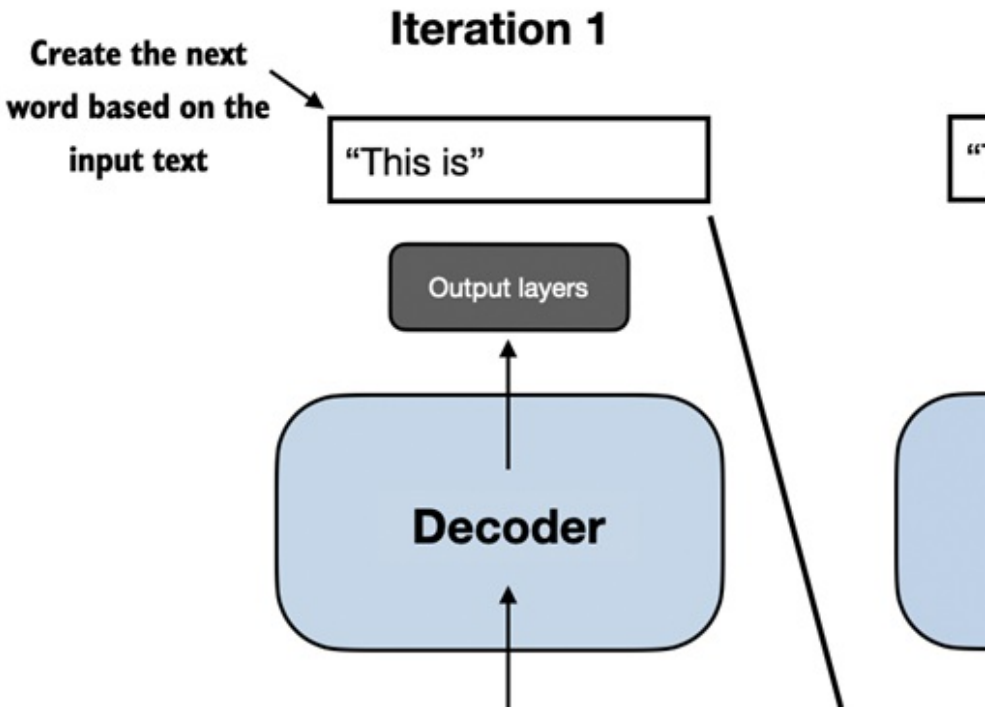


The next-word prediction task is a form of self-supervised learning, which is a form of self-labeling. This means that we don't need to collect labels for the training data explicitly but can leverage the structure of the data itself: we can use the next word in a sentence or document as the label that the model is supposed to predict. Since this next-word prediction task allows us to create labels "on the fly," it is possible to leverage massive unlabeled text datasets to train LLMs as previously discussed in section *1.5, Utilizing large datasets*.

Compared to the original transformer architecture we covered in section 1.4, *Using LLMs for different tasks*, the general GPT architecture is relatively simple. Essentially, it's just the decoder part without the encoder as illustrated in Figure 1.8. Since decoder-style models like GPT generate text by predicting text one word at a time, they are considered a type of *autoregressive* model. Autoregressive models incorporate their previous outputs as inputs for future predictions. Consequently, in GPT, each new word is chosen based on the sequence that precedes it, which improves coherence of the resulting text.

Architectures such as GPT-3 are also significantly larger than the original transformer model. For instance, the original transformer repeated the encoder and decoder blocks six times. GPT-3 has 96 transformer layers and 175 billion parameters in total.

Figure 1.8 The GPT architecture employs only the decoder portion of the original transformer. It is designed for unidirectional, left-to-right processing, making it well-suited for text generation and next-word prediction tasks to generate text in iterative fashion one word at a time.



GPT-3 was introduced in 2020, which, by the standards of deep learning and large language model (LLM) development, is considered a long time ago. However, more recent architectures, such as Meta's Llama models, are still based on the same underlying concepts, introducing only minor modifications. Hence, understanding GPT remains as relevant as ever, and this book focuses on implementing the prominent architecture behind GPT while providing pointers to specific tweaks employed by alternative LLMs.

Lastly, it's interesting to note that although the original transformer model was explicitly designed for language translation, GPT models—despite their larger yet simpler architecture aimed at next-word prediction—are also capable of performing translation tasks. This capability was initially

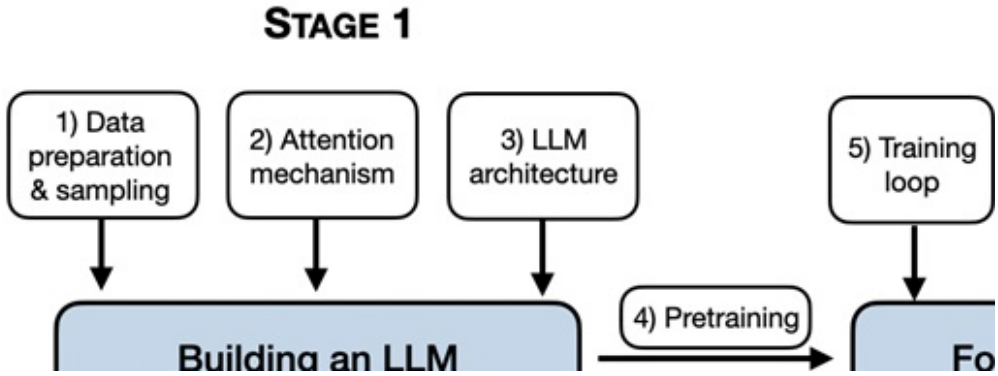
unexpected to researchers, as it emerged from a model primarily trained on a next-word prediction task, which is a task that did not specifically target translation.

The ability to perform tasks that the model wasn't explicitly trained to perform is called an "emergent behavior." This capability isn't explicitly taught during training but emerges as a natural consequence of the model's exposure to vast quantities of multilingual data in diverse contexts. The fact that GPT models can "learn" the translation patterns between languages and perform translation tasks even though they weren't specifically trained for it demonstrates the benefits and capabilities of these large-scale, generative language models. We can perform diverse tasks without using diverse models for each.

1.7 Building a large language model

In this chapter, we laid the groundwork for understanding LLMs. In the remainder of this book, we will be coding one from scratch. We will take the fundamental idea behind GPT as a blueprint and tackle this in three stages, as outlined in Figure 1.9.

Figure 1.9 The stages of building LLMs covered in this book include implementing the LLM architecture and data preparation process, pretraining an LLM to create a foundation model, and finetuning the foundation model to become a personal assistant or text classifier.



First, we will learn about the fundamental data preprocessing steps and code the attention mechanism that is at the heart of every LLM.

Next, in stage 2, we will learn how to code and pretrain a GPT-like LLM capable of generating new texts. And we will also go over the fundamentals of evaluating LLMs, which is essential for developing capable NLP systems.

Note that pretraining a large LLM from scratch is a significant endeavor, demanding thousands to millions of dollars in computing costs for GPT-like models. Therefore, the focus of stage 2 is on implementing training for educational purposes using a small dataset. In addition, the book will also provide code examples for loading openly available model weights.

Finally, in stage 3, we will take a pretrained LLM and finetune it to follow instructions such as answering queries or classifying texts -- the most common tasks in many real-world applications and research.

I hope you are looking forward to embarking on this exciting journey!

1.8 Summary

- LLMs have transformed the field of natural language processing, which previously mostly relied on explicit rule-based systems and simpler statistical methods. The advent of LLMs introduced new deep learning-driven approaches that led to advancements in understanding, generating, and translating human language.
- Modern LLMs are trained in two main steps.
- First, they are pretrained on a large corpus of unlabeled text by using the prediction of the next word in a sentence as a "label."
- Then, they are finetuned on a smaller, labeled target dataset to follow instructions or perform classification tasks.
- LLMs are based on the transformer architecture. The key idea of the transformer architecture is an attention mechanism that gives the LLM selective access to the whole input sequence when generating the output one word at a time.
- The original transformer architecture consists of an encoder for parsing text and a decoder for generating text.
- LLMs for generating text and following instructions, such as GPT-3 and ChatGPT, only implement decoder modules, simplifying the architecture.

- Large datasets consisting of billions of words are essential for pretraining LLMs. In this book, we will implement and train LLMs on small datasets for educational purposes but also see how we can load openly available model weights.
- While the general pretraining task for GPT-like models is to predict the next word in a sentence, these LLMs exhibit "emergent" properties such as capabilities to classify, translate, or summarize texts.
- Once an LLM is pretrained, the resulting foundation model can be finetuned more efficiently for various downstream tasks.
- LLMs finetuned on custom datasets can outperform general LLMs on specific tasks.

[1] Readers with a background in machine learning may note that labeling information is typically required for traditional machine learning models and deep neural networks trained via the conventional supervised learning paradigm. However, this is not the case for the pretraining stage of LLMs. In this phase, LLMs leverage self-supervised learning, where the model generates its own labels from the input data. This concept is covered later in this chapter

[2] *GPT-3, The $4,600,000 Language Model*, https://www.reddit.com/r/MachineLearning/comments/h0jwoz/d_gpt3_the_46

2 Working with Text Data

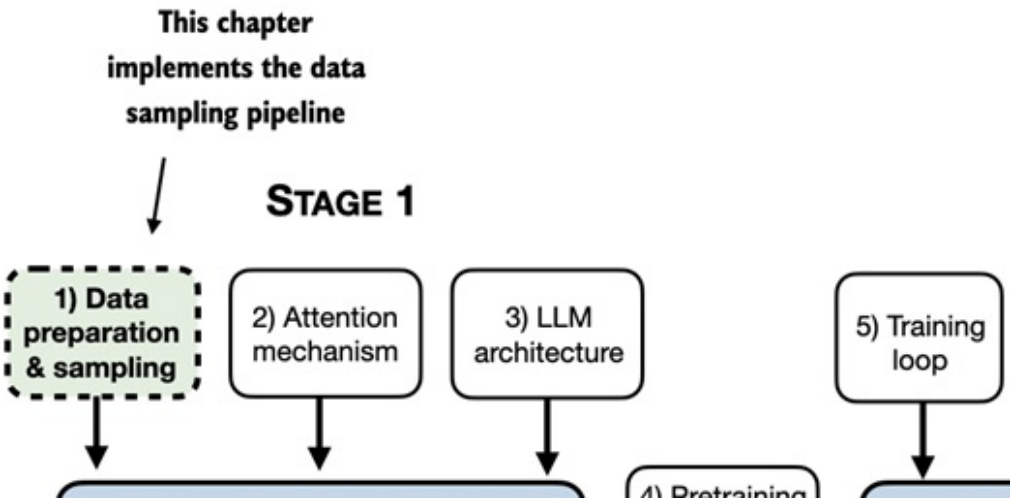
This chapter covers

- Preparing text for large language model training
- Splitting text into word and subword tokens
- Byte pair encoding as a more advanced way of tokenizing text
- Sampling training examples with a sliding window approach
- Converting tokens into vectors that feed into a large language model

In the previous chapter, we delved into the general structure of large language models (LLMs) and learned that they are pretrained on vast amounts of text. Specifically, our focus was on decoder-only LLMs based on the transformer architecture, which underlies the models used in ChatGPT and other popular GPT-like LLMs.

During the pretraining stage, LLMs process text one word at a time. Training LLMs with millions to billions of parameters using a next-word prediction task yields models with impressive capabilities. These models can then be further finetuned to follow general instructions or perform specific target tasks. But before we can implement and train LLMs in the upcoming chapters, we need to prepare the training dataset, which is the focus of this chapter, as illustrated in Figure 2.1

Figure 2.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset, and finetuning it on a labeled dataset. This chapter will explain and code the data preparation and sampling pipeline that provides the LLM with the text data for pretraining.



In this chapter, you'll learn how to prepare input text for training LLMs. This involves splitting text into individual word and subword tokens, which can then be encoded into vector representations for the LLM. You'll also learn about advanced tokenization schemes like byte pair encoding, which is utilized in popular LLMs like GPT. Lastly, we'll implement a sampling and data loading strategy to produce the input-output pairs necessary for training LLMs in subsequent chapters.

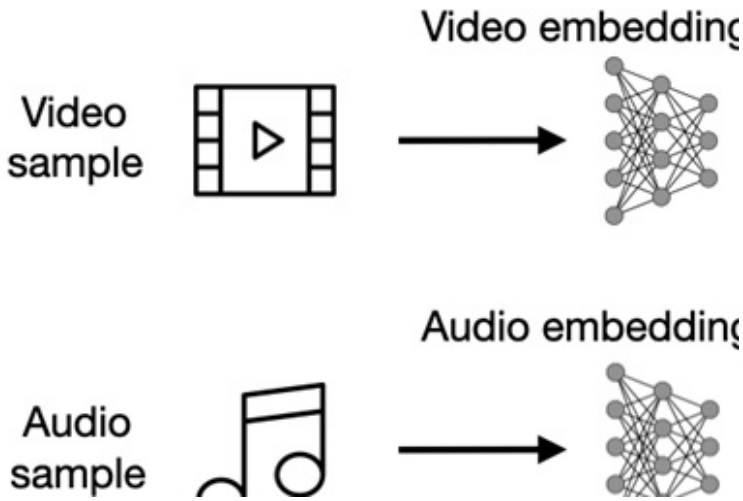
2.1 Understanding word embeddings

Deep neural network models, including LLMs, cannot process raw text directly. Since text is categorical, it isn't compatible with the mathematical operations used to implement and train neural networks. Therefore, we need a way to represent words as continuous-valued vectors. (Readers unfamiliar with vectors and tensors in a computational context can learn more in Appendix A, section A2.2 Understanding tensors.)

The concept of converting data into a vector format is often referred to as *embedding*. Using a specific neural network layer or another pretrained neural network model, we can embed different data types, for example, video, audio, and text, as illustrated in Figure 2.2.

Figure 2.2 Deep learning models cannot process data formats like video, audio, and text in their raw form. Thus, we use an embedding model to transform this raw data into a dense vector representation that deep learning architectures can easily understand and process. Specifically, this figure illustrates the process of converting raw data into a three-dimensional numerical

vector.



As shown in Figure 2.2, we can process various different data formats via embedding models. However, it's important to note that different data formats require distinct embedding models. For example, an embedding model designed for text would not be suitable for embedding audio or video data.

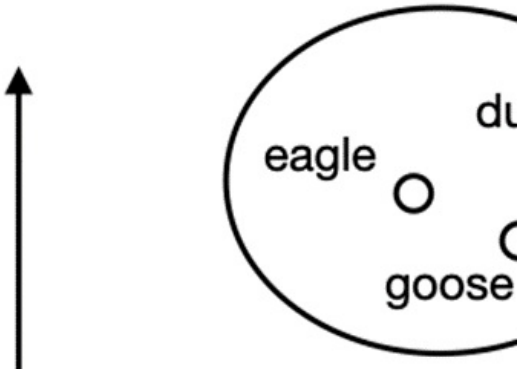
At its core, an embedding is a mapping from discrete objects, such as words, images, or even entire documents, to points in a continuous vector space -- the primary purpose of embeddings is to convert non-numeric data into a format that neural networks can process.

While word embeddings are the most common form of text embedding, there are also embeddings for sentences, paragraphs, or whole documents. Sentence or paragraph embeddings are popular choices for *retrieval-augmented generation.* Retrieval-augmented generation combines generation (like producing text) with retrieval (like searching an external knowledge base) to pull relevant information when generating text, which is a technique that is beyond the scope of this book. Since our goal is to train GPT-like LLMs, which learn to generate text one word at a time, this chapter focuses on word embeddings.

There are several algorithms and frameworks that have been developed to generate word embeddings. One of the earlier and most popular examples is the *Word2Vec* approach. Word2Vec trained neural network architecture to

generate word embeddings by predicting the context of a word given the target word or vice versa. The main idea behind Word2Vec is that words that appear in similar contexts tend to have similar meanings. Consequently, when projected into 2-dimensional word embeddings for visualization purposes, it can be seen that similar terms cluster together, as shown in Figure 2.3.

Figure 2.3 If word embeddings are two-dimensional, we can plot them in a two-dimensional scatterplot for visualization purposes as shown here. When using word embedding techniques, such as Word2Vec, words corresponding to similar concepts often appear close to each other in the embedding space. For instance, different types of birds appear closer to each other in the embedding space compared to countries and cities.



Word embeddings can have varying dimensions, from one to thousands. As shown in Figure 2.3, we can choose two-dimensional word embeddings for visualization purposes. A higher dimensionality might capture more nuanced relationships but at the cost of computational efficiency.

While we can use pretrained models such as Word2Vec to generate embeddings for machine learning models, LLMs commonly produce their own embeddings that are part of the input layer and are updated during training. The advantage of optimizing the embeddings as part of the LLM training instead of using Word2Vec is that the embeddings are optimized to the specific task and data at hand. We will implement such embedding layers later in this chapter. Furthermore, LLMs can also create contextualized output

embeddings, as we discuss in chapter 3.

Unfortunately, high-dimensional embeddings present a challenge for visualization because our sensory perception and common graphical representations are inherently limited to three dimensions or fewer, which is why Figure 2.3 showed two-dimensional embeddings in a two-dimensional scatterplot. However, when working with LLMs, we typically use embeddings with a much higher dimensionality than shown in Figure 2.3. For both GPT-2 and GPT-3, the embedding size (often referred to as the dimensionality of the model's hidden states) varies based on the specific model variant and size. It is a trade-off between performance and efficiency. The smallest GPT-2 models (117M and 125M parameters) use an embedding size of 768 dimensions to provide concrete examples. The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

The upcoming sections in this chapter will walk through the required steps for preparing the embeddings used by an LLM, which include splitting text into words, converting words into tokens, and turning tokens into embedding vectors.

2.2 Tokenizing text

This section covers how we split input text into individual tokens, a required preprocessing step for creating embeddings for an LLM. These tokens are either individual words or special characters, including punctuation characters, as shown in Figure 2.4.

Figure 2.4 A view of the text processing steps covered in this section in the context of an LLM. Here, we split an input text into individual tokens, which are either words or special characters, such as punctuation characters. In upcoming sections, we will convert the text into token IDs and create token embeddings.

The text we will tokenize for LLM training is a short story by Edith Wharton called *The Verdict*, which has been released into the public domain and is thus permitted to be used for LLM training tasks. The text is available on Wikisource at https://en.wikisource.org/wiki/The_Verdict, and you can copy and paste it into a text file, which I copied into a text file `"the-verdict.txt"` to load using Python's standard file reading utilities:

Listing 2.1 Reading in a short story as text sample into Python

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

Alternatively, you can find this `"the-verdict.txt"` file in this book's GitHub repository at https://github.com/rasbt/LLMs-from-scratch/tree/main/ch02/01_main-chapter-code.

The print command prints the total number of characters followed by the first 100 characters of this file for illustration purposes:

```
Total number of character: 20479
```

```
I HAD always thought Jack Gisburn rather a cheap genius--though a
```

Our goal is to tokenize this 20,479-character short story into individual words and special characters that we can then turn into embeddings for LLM training in the upcoming chapters.

Text sample sizes

Note that it's common to process millions of articles and hundreds of thousands of books -- many gigabytes of text -- when working with LLMs. However, for educational purposes, it's sufficient to work with smaller text samples like a single book to illustrate the main ideas behind the text processing steps and to make it possible to run it in reasonable time on consumer hardware.

How can we best split this text to obtain a list of tokens? For this, we go on a small excursion and use Python's regular expression library `re` for illustration purposes. (Note that you don't have to learn or memorize any regular expression syntax since we will transition to a pre-built tokenizer later in this chapter.)

Using some simple example text, we can use the `re.split` command with the following syntax to split a text on whitespace characters:

```
import re
text = "Hello, world. This, is a test."
result = re.split(r'(\s)', text)
print(result)
```

The result is a list of individual words, whitespaces, and punctuation characters:

```
['Hello,', ' ', 'world.', ' ', 'This,', ' ', 'is', ' ', 'a', ' ',
```

Note that the simple tokenization scheme above mostly works for separating the example text into individual words, however, some words are still connected to punctuation characters that we want to have as separate list entries. We also refrain from making all text lowercase because capitalization helps LLMs distinguish between proper nouns and common nouns,

understand sentence structure, and learn to generate text with proper capitalization.

Let's modify the regular expression splits on whitespaces (`\s`) and commas, and periods (`[,.]`):

```
result = re.split(r'([,.]|\s)', text)
print(result)
```

We can see that the words and punctuation characters are now separate list entries just as we wanted:

```
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'This', ',', '', '
```

A small remaining issue is that the list still includes whitespace characters. Optionally, we can remove these redundant characters safely as follows:

```
result = [item for item in result if item.strip()]
print(result)
```

The resulting whitespace-free output looks like as follows:

```
['Hello', ',', 'world', '.', 'This', ',', 'is', 'a', 'test', '.']
```

Removing whitespaces or not

When developing a simple tokenizer, whether we should encode whitespaces as separate characters or just remove them depends on our application and its requirements. Removing whitespaces reduces the memory and computing requirements. However, keeping whitespaces can be useful if we train models that are sensitive to the exact structure of the text (for example, Python code, which is sensitive to indentation and spacing). Here, we remove whitespaces for simplicity and brevity of the tokenized outputs. Later, we will switch to a tokenization scheme that includes whitespaces.

The tokenization scheme we devised above works well on the simple sample text. Let's modify it a bit further so that it can also handle other types of punctuation, such as question marks, quotation marks, and the double-dashes we have seen earlier in the first 100 characters of Edith Wharton's short story, along with additional special characters:

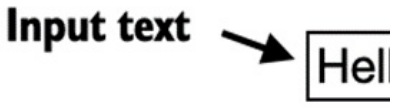
```
text = "Hello, world. Is this-- a test?"
result = re.split(r'([,.:;?_!"()\']|--|\s)', text)
result = [item.strip() for item in result if item.strip()]
print(result)
```

The resulting output is as follows:

```
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?'
```

As we can see based on the results summarized in Figure 2.5, our tokenization scheme can now handle the various special characters in the text successfully.

Figure 2.5 The tokenization scheme we implemented so far splits text into individual words and punctuation characters. In the specific example shown in this figure, the sample text gets split into 10 individual tokens.



Now that we got a basic tokenizer working, let's apply it to Edith Wharton's entire short story:

```
preprocessed = re.split(r'([,.?_!"()\']|--|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if item.str
print(len(preprocessed))
```

The above print statement outputs `4649`, which is the number of tokens in this text (without whitespaces).

Let's print the first 30 tokens for a quick visual check:

```
print(preprocessed[:30])
```

The resulting output shows that our tokenizer appears to be handling the text well since all words and special characters are neatly separated:

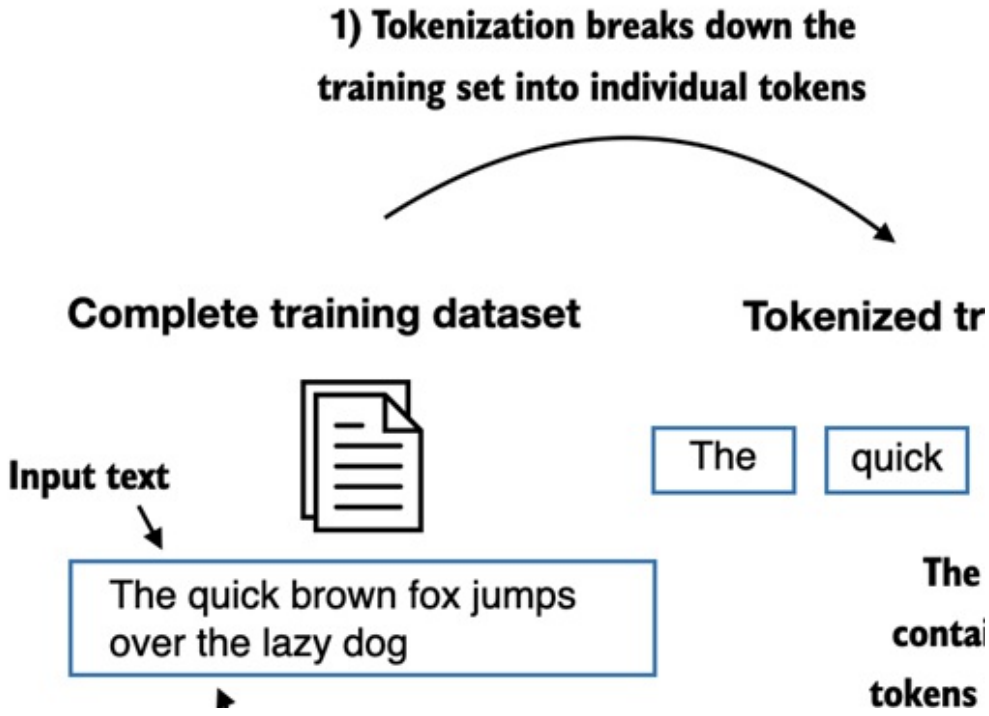
```
['I', 'HAD', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a
```

2.3 Converting tokens into token IDs

In the previous section, we tokenized a short story by Edith Wharton into individual tokens. In this section, we will convert these tokens from a Python string to an integer representation to produce the so-called token IDs. This conversion is an intermediate step before converting the token IDs into embedding vectors.

To map the previously generated tokens into token IDs, we have to build a so-called vocabulary first. This vocabulary defines how we map each unique word and special character to a unique integer, as shown in Figure 2.6.

Figure 2.6 We build a vocabulary by tokenizing the entire text in a training dataset into individual tokens. These individual tokens are then sorted alphabetically, and duplicate tokens are removed. The unique tokens are then aggregated into a vocabulary that defines a mapping from each unique token to a unique integer value. The depicted vocabulary is purposefully small for illustration purposes and contains no punctuation or special characters for simplicity.



In the previous section, we tokenized Edith Wharton's short story and assigned it to a Python variable called `preprocessed`. Let's now create a list of all unique tokens and sort them alphabetically to determine the vocabulary

size:

```
all_words = sorted(list(set(preprocessed)))
vocab_size = len(all_words)
print(vocab_size)
```

After determining that the vocabulary size is 1,159 via the above code, we create the vocabulary and print its first 50 entries for illustration purposes:

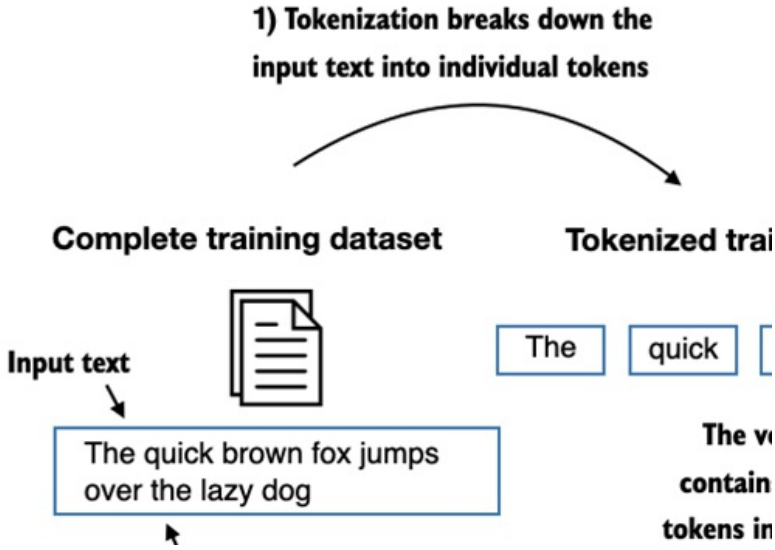
Listing 2.2 Creating a vocabulary

```
vocab = {token:integer for integer,token in enumerate(all_words)}
for i, item in enumerate(vocab.items()):
    print(item)
    if i > 50:
        break
```

```
('!', 0)
('"', 1)
("'", 2)
...
('Has', 49)
('He', 50)
```

As we can see, based on the output above, the dictionary contains individual tokens associated with unique integer labels. Our next goal is to apply this vocabulary to convert new text into token IDs, as illustrated in Figure 2.7.

Figure 2.7 Starting with a new text sample, we tokenize the text and use the vocabulary to convert the text tokens into token IDs. The vocabulary is built from the entire training set and can be applied to the training set itself and any new text samples. The depicted vocabulary contains no punctuation or special characters for simplicity.



Later in this book, when we want to convert the outputs of an LLM from numbers back into text, we also need a way to turn token IDs into text. For this, we can create an inverse version of the vocabulary that maps token IDs back to corresponding text tokens.

Let's implement a complete tokenizer class in Python with an `encode` method that splits text into tokens and carries out the string-to-integer mapping to produce token IDs via the vocabulary. In addition, we implement a `decode` method that carries out the reverse integer-to-string mapping to convert the token IDs back into text.

The code for this tokenizer implementation is as in listing 2.3:

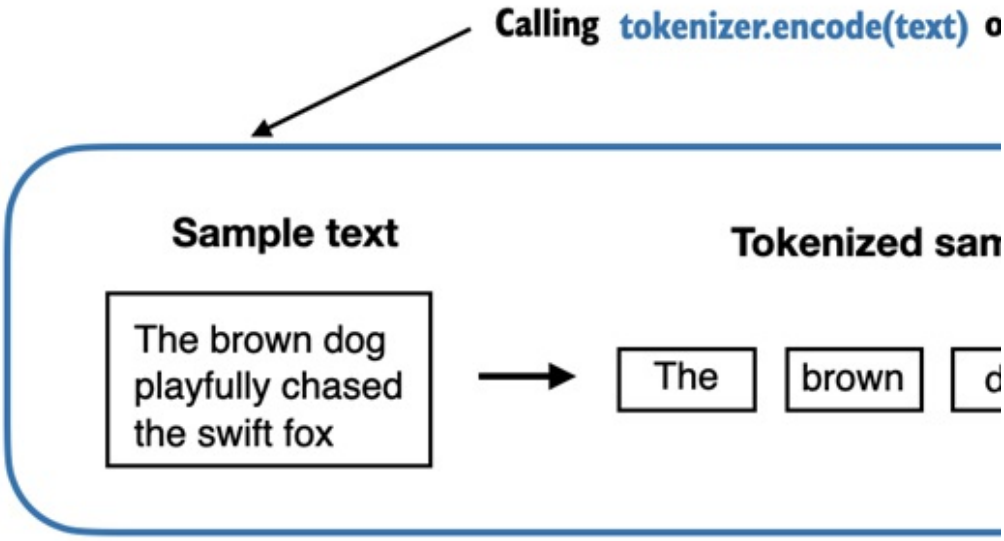
Listing 2.3 Implementing a simple text tokenizer

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab #A
        self.int_to_str = {i:s for s,i in vocab.items()} #B

    def encode(self, text): #C
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids): #D
```

```
        text = " ".join([self.int_to_str[i] for i in ids])

        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text) #E
        return text
```

Using the `SimpleTokenizerV1` Python class above, we can now instantiate new tokenizer objects via an existing vocabulary, which we can then use to encode and decode text, as illustrated in Figure 2.8.

Figure 2.8 Tokenizer implementations share two common methods: an encode method and a decode method. The encode method takes in the sample text, splits it into individual tokens, and converts the tokens into token IDs via the vocabulary. The decode method takes in token IDs, converts them back into text tokens, and concatenates the text tokens into natural text.



Let's instantiate a new tokenizer object from the `SimpleTokenizerV1` class and tokenize a passage from Edith Wharton's short story to try it out in practice:

```
tokenizer = SimpleTokenizerV1(vocab)

text = """It's the last he painted, you know," Mrs. Gisburn said
ids = tokenizer.encode(text)
print(ids)
```

The code above prints the following token IDs:

```
[1, 58, 2, 872, 1013, 615, 541, 763, 5, 1155, 608, 5, 1, 69, 7, 3
```

Next, let's see if we can turn these token IDs back into text using the decode method:

```
print(tokenizer.decode(ids))
```

This outputs the following text:

```
'" It\' s the last he painted, you know," Mrs. Gisburn said with
```

Based on the output above, we can see that the decode method successfully converted the token IDs back into the original text.

So far, so good. We implemented a tokenizer capable of tokenizing and de-tokenizing text based on a snippet from the training set. Let's now apply it to a new text sample that is not contained in the training set:

```
text = "Hello, do you like tea?"
tokenizer.encode(text)
```

Executing the code above will result in the following error:

```
...
KeyError: 'Hello'
```

The problem is that the word "Hello" was not used in the *The Verdict* short story. Hence, it is not contained in the vocabulary. This highlights the need to consider large and diverse training sets to extend the vocabulary when working on LLMs.

In the next section, we will test the tokenizer further on text that contains unknown words, and we will also discuss additional special tokens that can be used to provide further context for an LLM during training.

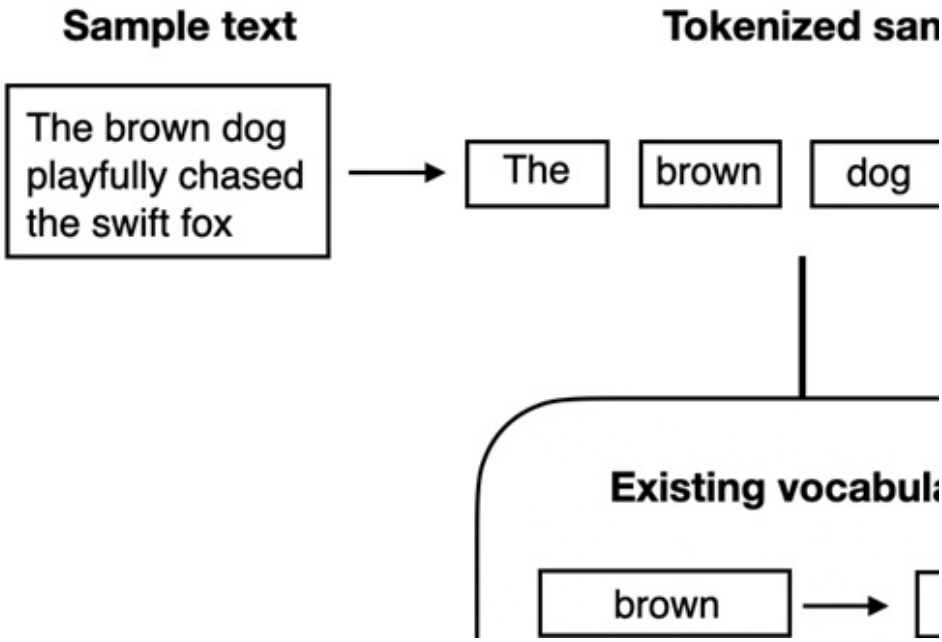
2.4 Adding special context tokens

In the previous section, we implemented a simple tokenizer and applied it to a passage from the training set. In this section, we will modify this tokenizer to handle unknown words.

We will also discuss the usage and addition of special context tokens that can enhance a model's understanding of context or other relevant information in the text. These special tokens can include markers for unknown words and document boundaries, for example.

In particular, we will modify the vocabulary and tokenizer we implemented in the previous section, `SimpleTokenizerV2`, to support two new tokens, `<|unk|>` and `<|endoftext|>`, as illustrated in Figure 2.9.

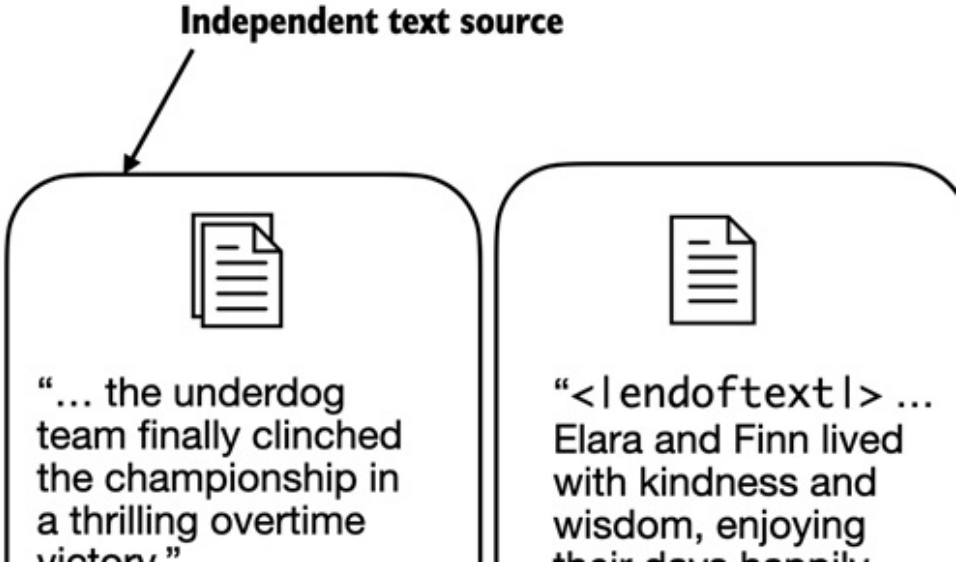
Figure 2.9 We add special tokens to a vocabulary to deal with certain contexts. For instance, we add an <|unk|> token to represent new and unknown words that were not part of the training data and thus not part of the existing vocabulary. Furthermore, we add an <|endoftext|> token that we can use to separate two unrelated text sources.



As shown in Figure 2.9, we can modify the tokenizer to use an `<|unk|>` token if it encounters a word that is not part of the vocabulary. Furthermore, we add a token between unrelated texts. For example, when training GPT-like LLMs on multiple independent documents or books, it is common to insert a token before each document or book that follows a previous text source, as illustrated in Figure 2.10. This helps the LLM understand that, although these text sources are concatenated for training, they are, in fact, unrelated.

Figure 2.10 When working with multiple independent text source, we add <|endoftext|> tokens between these texts. These <|endoftext|> tokens act as markers, signaling the start or end of a

particular segment, allowing for more effective processing and understanding by the LLM.



Let's now modify the vocabulary to include these two special tokens, `<unk>` and `<|endoftext|>`, by adding these to the list of all unique words that we created in the previous section:

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token:integer for integer,token in enumerate(all_tokens)

print(len(vocab.items()))
```

Based on the output of the print statement above, the new vocabulary size is 1161 (the vocabulary size in the previous section was 1159).

As an additional quick check, let's print the last 5 entries of the updated vocabulary:

```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

The code above prints the following:

```
('younger', 1156)
('your', 1157)
('yourself', 1158)
```

```
('<|endoftext|>', 1159)
('<|unk|>', 1160)
```

Based on the code output above, we can confirm that the two new special tokens were indeed successfully incorporated into the vocabulary. Next, we adjust the tokenizer from code listing 2.3 accordingly, as shown in listing 2.4:

Listing 2.4 A simple text tokenizer that handles unknown words

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if
        preprocessed = [item if item in self.str_to_int  #A
                        else "<|unk|>" for item in preprocessed]

        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])

        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text) #B
        return text
```

Compared to the `SimpleTokenizerV1` we implemented in code listing 2.3 in the previous section, the new `SimpleTokenizerV2` replaces unknown words by `<|unk|>` tokens.

Let's now try this new tokenizer out in practice. For this, we will use a simple text sample that we concatenate from two independent and unrelated sentences:

```
text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."
text = " <|endoftext|> ".join((text1, text2))
print(text)
```

The output is as follows:

```
'Hello, do you like tea? <|endoftext|> In the sunlit terraces of
```

Next, let's tokenize the sample text using the `SimpleTokenizerV2` on the vocab we previously created in listing 2.2:

```
tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.encode(text))
```

This prints the following token IDs:

```
[1160, 5, 362, 1155, 642, 1000, 10, 1159, 57, 1013, 981, 1009, 73
```

Above, we can see that the list of token IDs contains 1159 for the <|endoftext|> separator token as well as two 1160 tokens, which are used for unknown words.

Let's de-tokenize the text for a quick sanity check:

```
print(tokenizer.decode(tokenizer.encode(text)))
```

The output is as follows:

```
'<|unk|>, do you like tea? <|endoftext|> In the sunlit terraces o
```

Based on comparing the de-tokenized text above with the original input text, we know that the training dataset, Edith Wharton's short story *The Verdict*, did not contain the words "Hello" and "palace."

So far, we have discussed tokenization as an essential step in processing text as input to LLMs. Depending on the LLM, some researchers also consider additional special tokens such as the following:

- `[BOS]` (beginning of sequence): This token marks the start of a text. It signifies to the LLM where a piece of content begins.
- `[EOS]` (end of sequence): This token is positioned at the end of a text, and is especially useful when concatenating multiple unrelated texts, similar to `<|endoftext|>`. For instance, when combining two different Wikipedia articles or books, the `[EOS]` token indicates where one article ends and the next one begins.
- `[PAD]` (padding): When training LLMs with batch sizes larger than one,

the batch might contain texts of varying lengths. To ensure all texts have the same length, the shorter texts are extended or "padded" using the `[PAD]` token, up to the length of the longest text in the batch.

Note that the tokenizer used for GPT models does not need any of these tokens mentioned above but only uses an `<|endoftext|>` token for simplicity. The `<|endoftext|>` is analogous to the `[EOS]` token mentioned above. Also, `<|endoftext|>` is used for padding as well. However, as we'll explore in subsequent chapters when training on batched inputs, we typically use a mask, meaning we don't attend to padded tokens. Thus, the specific token chosen for padding becomes inconsequential.

Moreover, the tokenizer used for GPT models also doesn't use an `<|unk|>` token for out-of-vocabulary words. Instead, GPT models use a *byte pair encoding* tokenizer, which breaks down words into subword units, which we will discuss in the next section.

2.5 Byte pair encoding

We implemented a simple tokenization scheme in the previous sections for illustration purposes. This section covers a more sophisticated tokenization scheme based on a concept called byte pair encoding (BPE). The BPE tokenizer covered in this section was used to train LLMs such as GPT-2, GPT-3, and the original model used in ChatGPT.

Since implementing BPE can be relatively complicated, we will use an existing Python open-source library called *tiktoken* (https://github.com/openai/tiktoken), which implements the BPE algorithm very efficiently based on source code in Rust. Similar to other Python libraries, we can install the tiktoken library via Python's `pip` installer from the terminal:

```
pip install tiktoken
```

The code in this chapter is based on tiktoken 0.5.1. You can use the following code to check the version you currently have installed:

```
from importlib.metadata import version
```

```
import tiktoken
print("tiktoken version:", version("tiktoken"))
```

Once installed, we can instantiate the BPE tokenizer from tiktoken as follows:

```
tokenizer = tiktoken.get_encoding("gpt2")
```

The usage of this tokenizer is similar to SimpleTokenizerV2 we implemented previously via an `encode` method:

```
text = "Hello, do you like tea? <|endoftext|> In the sunlit terra
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>
print(integers)
```

The code above prints the following token IDs:

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252,
```

We can then convert the token IDs back into text using the decode method, similar to our `SimpleTokenizerV2` earlier:

```
strings = tokenizer.decode(integers)
print(strings)
```

The above code prints the following:

```
'Hello, do you like tea? <|endoftext|> In the sunlit terraces of
```

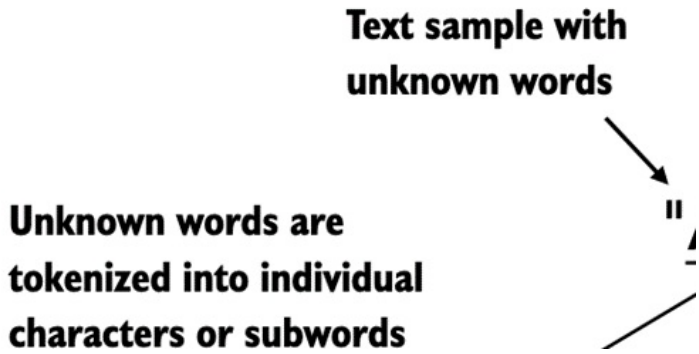
We can make two noteworthy observations based on the token IDs and decoded text above. First, the `<|endoftext|>` token is assigned a relatively large token ID, namely, 50256. In fact, the BPE tokenizer, which was used to train models such as GPT-2, GPT-3, and the original model used in ChatGPT, has a total vocabulary size of 50,257, with `<|endoftext|>` being assigned the largest token ID.

Second, the BPE tokenizer above encodes and decodes unknown words, such as "someunknownPlace" correctly. The BPE tokenizer can handle any unknown word. How does it achieve this without using `<|unk|>` tokens?

The algorithm underlying BPE breaks down words that aren't in its

predefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words. So, thanks to the BPE algorithm, if the tokenizer encounters an unfamiliar word during tokenization, it can represent it as a sequence of subword tokens or characters, as illustrated in Figure 2.11.

Figure 2.11 BPE tokenizers break down unknown words into subwords and individual characters. This way, a BPE tokenizer can parse any word and doesn't need to replace unknown words with special tokens, such as <|unk|>.



As illustrated in Figure 2.11, the ability to break down unknown words into individual characters ensures that the tokenizer, and consequently the LLM that is trained with it, can process any text, even if it contains words that were not present in its training data.

Exercise 2.1 Byte pair encoding of unknown words

Try the BPE tokenizer from the tiktoken library on the unknown words "Akwirw ier" and print the individual token IDs. Then, call the decode function on each of the resulting integers in this list to reproduce the mapping shown in Figure 2.11. Lastly, call the decode method on the token IDs to check whether it can reconstruct the original input, "Akwirw ier".

A detailed discussion and implementation of BPE is out of the scope of this book, but in short, it builds its vocabulary by iteratively merging frequent characters into subwords and frequent subwords into words. For example, BPE starts with adding all individual single characters to its vocabulary ("a", "b", ...). In the next stage, it merges character combinations that frequently occur together into subwords. For example, "d" and "e" may be merged into

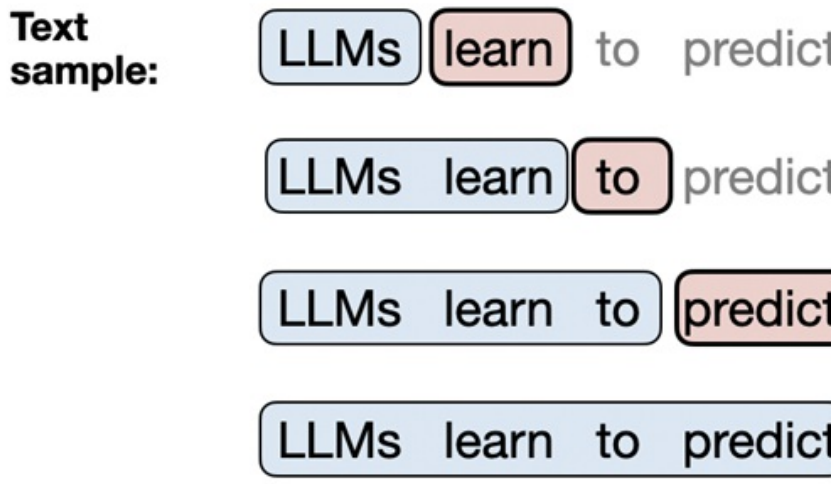
the subword "de," which is common in many English words like "define", "depend", "made", and "hidden". The merges are determined by a frequency cutoff.

2.6 Data sampling with a sliding window

The previous section covered the tokenization steps and conversion from string tokens into integer token IDs in great detail. The next step before we can finally create the embeddings for the LLM is to generate the input-target pairs required for training an LLM.

What do these input-target pairs look like? As we learned in chapter 1, LLMs are pretrained by predicting the next word in a text, as depicted in figure 2.12.

Figure 2.12 Given a text sample, extract input blocks as subsamples that serve as input to the LLM, and the LLM's prediction task during training is to predict the next word that follows the input block. During training, we mask out all words that are past the target. Note that the text shown in this figure would undergo tokenization before the LLM can process it; however, this figure omits the tokenization step for clarity.



In this section we implement a data loader that fetches the input-target pairs depicted in Figure 2.12 from the training dataset using a sliding window approach.

To get started, we will first tokenize the whole The Verdict short story we worked with earlier using the BPE tokenizer introduced in the previous

section:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

Executing the code above will return 5145, the total number of tokens in the training set, after applying the BPE tokenizer.

Next, we remove the first 50 tokens from the dataset for demonstration purposes as it results in a slightly more interesting text passage in the next steps:

```
enc_sample = enc_text[50:]
```

One of the easiest and most intuitive ways to create the input-target pairs for the next-word prediction task is to create two variables, `x` and `y`, where `x` contains the input tokens and `y` contains the targets, which are the inputs shifted by 1:

```
context_size = 4 #A

x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y:      {y}")
```

Running the above code prints the following output:

```
x: [290, 4920, 2241, 287]
y:      [4920, 2241, 287, 257]
```

Processing the inputs along with the targets, which are the inputs shifted by one position, we can then create the next-word prediction tasks depicted earlier in figure 2.12, as follows:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

The code above prints the following:

```
[290] ----> 4920
[290, 4920] ----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287] ----> 257
```

Everything left of the arrow (`---->`) refers to the input an LLM would receive, and the token ID on the right side of the arrow represents the target token ID that the LLM is supposed to predict.

For illustration purposes, let's repeat the previous code but convert the token IDs into text:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([d
```

The following outputs show how the input and outputs look in text format:

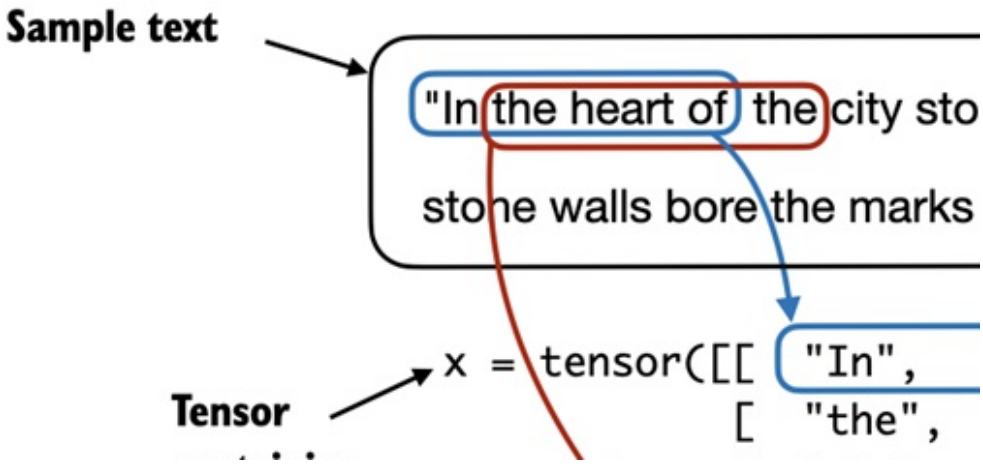
```
 and ---->  established
 and established ---->  himself
 and established himself ---->  in
 and established himself in ---->  a
```

We've now created the input-target pairs that we can turn into use for the LLM training in upcoming chapters.

There's only one more task before we can turn the tokens into embeddings, as we mentioned at the beginning of this chapter: implementing an efficient data loader that iterates over the input dataset and returns the inputs and targets as PyTorch tensors, which can be thought of as multidimensional arrays.

In particular, we are interested in returning two tensors: an input tensor containing the text that the LLM sees and a target tensor that includes the targets for the LLM to predict, as depicted in Figure 2.13.

Figure 2.13 To implement efficient data loaders, we collect the inputs in a tensor, x, where each row represents one input context. A second tensor, y, contains the corresponding prediction targets (next words), which are created by shifting the input by one position.



While Figure 2.13 shows the tokens in string format for illustration purposes, the code implementation will operate on token IDs directly since the `encode` method of the BPE tokenizer performs both tokenization and conversion into token IDs as a single step.

For the efficient data loader implementation, we will use PyTorch's built-in Dataset and DataLoader classes. For additional information and guidance on installing PyTorch, please see section A.1.3, Installing PyTorch, in Appendix A.

The code for the dataset class is shown in code listing 2.5:

Listing 2.5 A dataset for batched inputs and targets

```
import torch
from torch.utils.data import Dataset, DataLoader

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.tokenizer = tokenizer
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt) #A

        for i in range(0, len(token_ids) - max_length, stride): #
            input_chunk = token_ids[i:i + max_length]
            target_chunk = token_ids[i + 1: i + max_length + 1]
            self.input_ids.append(torch.tensor(input_chunk))
            self.target_ids.append(torch.tensor(target_chunk))
```

```
    def __len__(self): #C
        return len(self.input_ids)

    def __getitem__(self, idx): #D
        return self.input_ids[idx], self.target_ids[idx]
```

The `GPTDatasetV1` class in listing 2.5 is based on the PyTorch `Dataset` class and defines how individual rows are fetched from the dataset, where each row consists of a number of token IDs (based on a `max_length`) assigned to an `input_chunk` tensor. The `target_chunk` tensor contains the corresponding targets. I recommend reading on to see how the data returned from this dataset looks like when we combine the dataset with a PyTorch `DataLoader` -- this will bring additional intuition and clarity.

If you are new to the structure of PyTorch `Dataset` classes, such as shown in listing 2.5, please read section *A.6, Setting up efficient data loaders*, in Appendix A, which explains the general structure and usage of PyTorch `Dataset` and `DataLoader` classes.

The following code will use the `GPTDatasetV1` to load the inputs in batches via a PyTorch `DataLoader`:

Listing 2.6 A data loader to generate batches with input-with pairs

```
def create_dataloader_v1(txt, batch_size=4,
        max_length=256, stride=128, shuffle=True, drop_last=True)
    tokenizer = tiktoken.get_encoding("gpt2") #A
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride) #B
    dataloader = DataLoader(
        dataset, batch_size=batch_size, shuffle=shuffle, drop_las
    return dataloader
```

Let's test the `dataloader` with a batch size of 1 for an LLM with a context size of 4 to develop an intuition of how the `GPTDatasetV1` class from listing 2.5 and the `create_dataloader_v1` function from listing 2.6 work together:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, stride=1, shuffle=False
```

```
data_iter = iter(dataloader) #A
first_batch = next(data_iter)
print(first_batch)
```

Executing the preceding code prints the following:

```
[tensor([[  40,  367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1
```

The `first_batch` variable contains two tensors: the first tensor stores the input token IDs, and the second tensor stores the target token IDs. Since the `max_length` is set to 4, each of the two tensors contains 4 token IDs. Note that an input size of 4 is relatively small and only chosen for illustration purposes. It is common to train LLMs with input sizes of at least 256.

To illustrate the meaning of `stride=1`, let's fetch another batch from this dataset:

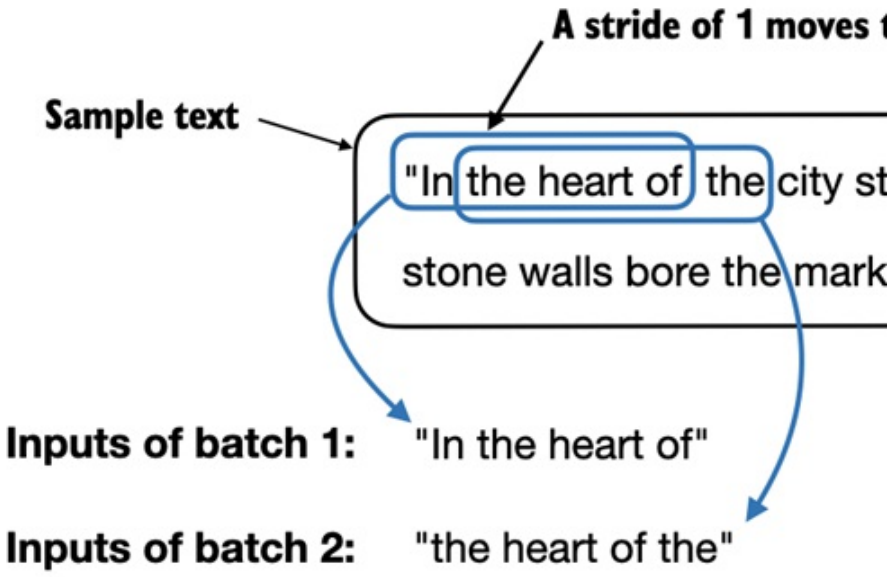
```
second_batch = next(data_iter)
print(second_batch)
```

The second batch has the following contents:

```
[tensor([[ 367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3
```

If we compare the first with the second batch, we can see that the second batch's token IDs are shifted by one position compared to the first batch (for example, the second ID in the first batch's input is 367, which is the first ID of the second batch's input). The `stride` setting dictates the number of positions the inputs shift across batches, emulating a sliding window approach, as demonstrated in Figure 2.14.

Figure 2.14 When creating multiple batches from the input dataset, we slide an input window across the text. If the stride is set to 1, we shift the input window by 1 position when creating the next batch. If we set the stride equal to the input window size, we can prevent overlaps between the batches.



Exercise 2.2 Data loaders with different strides and context sizes

To develop more intuition for how the data loader works, try to run it with different settings such as max_length=2 and stride=2 and max_length=8 and stride=2.

Batch sizes of 1, such as we have sampled from the data loader so far, are useful for illustration purposes. If you have previous experience with deep learning, you may know that small batch sizes require less memory during training but lead to more noisy model updates. Just like in regular deep learning, the batch size is a trade-off and hyperparameter to experiment with when training LLMs.

Before we move on to the two final sections of this chapter that are focused on creating the embedding vectors from the token IDs, let's have a brief look at how we can use the data loader to sample with a batch size greater than 1:

```
dataloader = create_dataloader_v1(raw_text, batch_size=8, max_len

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

This prints the following:

```
Inputs:
 tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]])

Targets:
 tensor([[  367,  2885,  1464,  1807],
        [ 3619,   402,   271, 10899],
        [ 2138,   257,  7026, 15632],
        [  438,  2016,   257,   922],
        [ 5891,  1576,   438,   568],
        [  340,   373,   645,  1049],
        [ 5975,   284,   502,   284],
        [ 3285,   326,    11,   287]])
```

Note that we increase the stride to 4. This is to utilize the data set fully (we don't skip a single word) but also avoid any overlap between the batches, since more overlap could lead to increased overfitting.

In the final two sections of this chapter, we will implement embedding layers that convert the token IDs into continuous vector representations, which serve as input data format for LLMs.

2.7 Creating token embeddings

The last step for preparing the input text for LLM training is to convert the token IDs into embedding vectors, as illustrated in Figure 2.15, which will be the focus of these two last remaining sections of this chapter.

Figure 2.15 Preparing the input text for an LLM involves tokenizing text, converting text tokens to token IDs, and converting token IDs into vector embedding vectors. In this section, we consider the token IDs created in previous sections to create the token embedding vectors.

In addition to the processes outlined in Figure 2.15, it is important to note that we initialize these embedding weights with random values as a preliminary step. This initialization serves as the starting point for the LLM's learning process. We will optimize the embedding weights as part of the LLM training in chapter 5.

A continuous vector representation, or embedding, is necessary since GPT-like LLMs are deep neural networks trained with the backpropagation algorithm. If you are unfamiliar with how neural networks are trained with backpropagation, please read section A.4, *Automatic differentiation made easy*, in Appendix A.

Let's illustrate how the token ID to embedding vector conversion works with a hands-on example. Suppose we have the following four input tokens with IDs 2, 3, 5, and 1:

```
input_ids = torch.tensor([2, 3, 5, 1])
```

For the sake of simplicity and illustration purposes, suppose we have a small vocabulary of only 6 words (instead of the 50,257 words in the BPE tokenizer vocabulary), and we want to create embeddings of size 3 (in GPT-

3, the embedding size is 12,288 dimensions):

```
vocab_size = 6
output_dim = 3
```

Using the `vocab_size` and `output_dim`, we can instantiate an embedding layer in PyTorch, setting the random seed to 123 for reproducibility purposes:

```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

The print statement in the preceding code example prints the embedding layer's underlying weight matrix:

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
        [ 0.9178,  1.5810,  1.3010],
        [ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-1.1589,  0.3255, -0.6315],
        [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

We can see that the weight matrix of the embedding layer contains small, random values. These values are optimized during LLM training as part of the LLM optimization itself, as we will see in upcoming chapters. Moreover, we can see that the weight matrix has six rows and three columns. There is one row for each of the six possible tokens in the vocabulary. And there is one column for each of the three embedding dimensions.

After we instantiated the embedding layer, let's now apply it to a token ID to obtain the embedding vector:

```
print(embedding_layer(torch.tensor([3])))
```

The returned embedding vector is as follows:

```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0
```

If we compare the embedding vector for token ID 3 to the previous embedding matrix, we see that it is identical to the 4th row (Python starts with a zero index, so it's the row corresponding to index 3). In other words,

the embedding layer is essentially a look-up operation that retrieves rows from the embedding layer's weight matrix via a token ID.

Embedding layers versus matrix multiplication

For those who are familiar with one-hot encoding, the embedding layer approach above is essentially just a more efficient way of implementing one-hot encoding followed by matrix multiplication in a fully connected layer, which is illustrated in the supplementary code on GitHub at https://github.com/rasbt/LLMs-from-scratch/tree/main/ch02/03_bonus_embedding-vs-matmul. Because the embedding layer is just a more efficient implementation equivalent to the one-hot encoding and matrix-multiplication approach, it can be seen as a neural network layer that can be optimized via backpropagation.

Previously, we have seen how to convert a single token ID into a three-dimensional embedding vector. Let's now apply that to all four input IDs we defined earlier (`torch.tensor([2, 3, 5, 1])`):

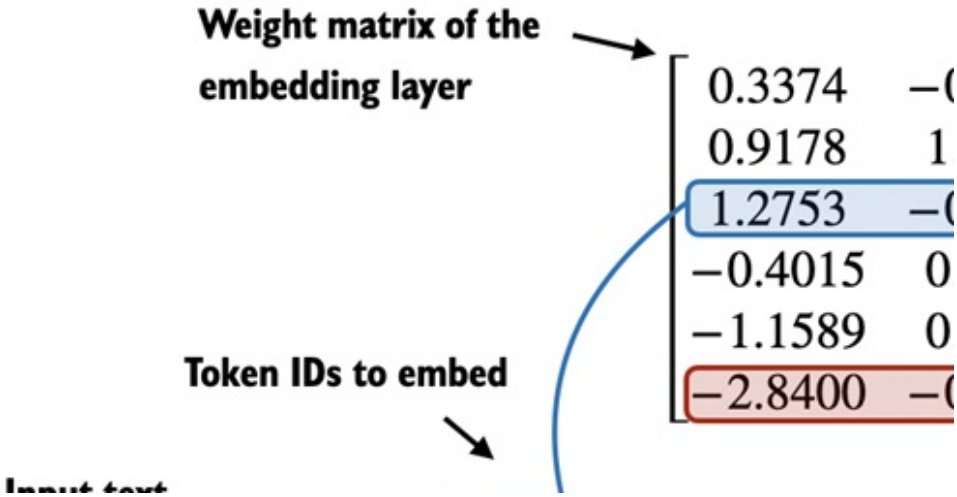
```
print(embedding_layer(input_ids))
```

The print output reveals that this results in a 4x3 matrix:

```
tensor([[ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-2.8400, -0.7849, -1.4096],
        [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0
```

Each row in this output matrix is obtained via a lookup operation from the embedding weight matrix, as illustrated in Figure 2.16.

Figure 2.16 Embedding layers perform a look-up operation, retrieving the embedding vector corresponding to the token ID from the embedding layer's weight matrix. For instance, the embedding vector of the token ID 5 is the sixth row of the embedding layer weight matrix (it is the sixth instead of the fifth row because Python starts counting at 0). For illustration purposes, we assume that the token IDs were produced by the small vocabulary we used in section 2.3.



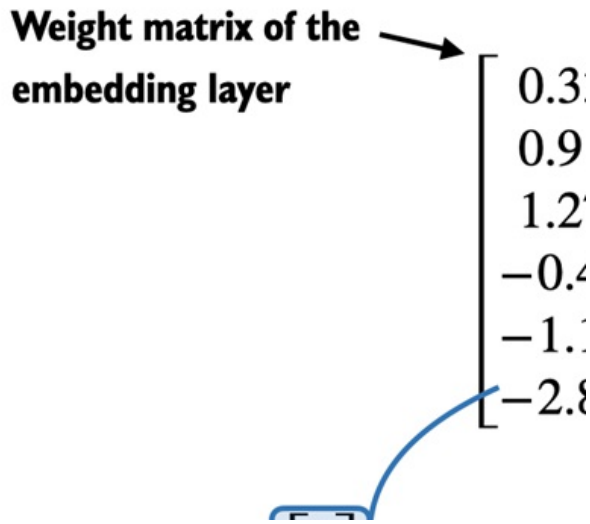
This section covered how we create embedding vectors from token IDs. The next and final section of this chapter will add a small modification to these embedding vectors to encode positional information about a token within a text.

2.8 Encoding word positions

In the previous section, we converted the token IDs into a continuous vector representation, the so-called token embeddings. In principle, this is a suitable input for an LLM. However, a minor shortcoming of LLMs is that their self-attention mechanism, which will be covered in detail in chapter 3, doesn't have a notion of position or order for the tokens within a sequence.

The way the previously introduced embedding layer works is that the same token ID always gets mapped to the same vector representation, regardless of where the token ID is positioned in the input sequence, as illustrated in Figure 2.17.

Figure 2.17 The embedding layer converts a token ID into the same vector representation regardless of where it is located in the input sequence. For example, the token ID 5, whether it's in the first or third position in the token ID input vector, will result in the same embedding vector.

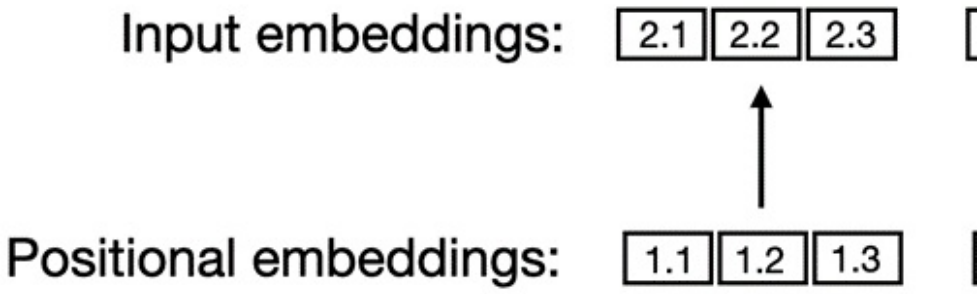


In principle, the deterministic, position-independent embedding of the token ID is good for reproducibility purposes. However, since the self-attention mechanism of LLMs itself is also position-agnostic, it is helpful to inject additional position information into the LLM.

To achieve this, there are two broad categories of position-aware embeddings: relative *positional embeddings* and absolute positional embeddings.

Absolute positional embeddings are directly associated with specific positions in a sequence. For each position in the input sequence, a unique embedding is added to the token's embedding to convey its exact location. For instance, the first token will have a specific positional embedding, the second token another distinct embedding, and so on, as illustrated in Figure 2.18.

Figure 2.18 Positional embeddings are added to the token embedding vector to create the input embeddings for an LLM. The positional vectors have the same dimension as the original token embeddings. The token embeddings are shown with value 1 for simplicity.



Instead of focusing on the absolute position of a token, the emphasis of relative positional embeddings is on the relative position or distance between tokens. This means the model learns the relationships in terms of "how far apart" rather than "at which exact position." The advantage here is that the model can generalize better to sequences of varying lengths, even if it hasn't seen such lengths during training.

Both types of positional embeddings aim to augment the capacity of LLMs to understand the order and relationships between tokens, ensuring more accurate and context-aware predictions. The choice between them often depends on the specific application and the nature of the data being processed.

OpenAI's GPT models use absolute positional embeddings that are optimized during the training process rather than being fixed or predefined like the positional encodings in the original Transformer model. This optimization process is part of the model training itself, which we will implement later in this book. For now, let's create the initial positional embeddings to create the LLM inputs for the upcoming chapters.

Previously, we focused on very small embedding sizes in this chapter for illustration purposes. We now consider more realistic and useful embedding sizes and encode the input tokens into a 256-dimensional vector representation. This is smaller than what the original GPT-3 model used (in GPT-3, the embedding size is 12,288 dimensions) but still reasonable for experimentation. Furthermore, we assume that the token IDs were created by the BPE tokenizer that we implemented earlier, which has a vocabulary size of 50,257:

```
output_dim = 256
vocab_size = 50257
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim
```

Using the `token_embedding_layer` above, if we sample data from the data loader, we embed each token in each batch into a 256-dimensional vector. If we have a batch size of 8 with four tokens each, the result will be an 8 x 4 x 256 tensor.

Let's instantiate the data loader from section 2.6, *Data sampling with a sliding window*, first:

```
max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length, stride=max_len
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Token IDs:\n", inputs)
print("\nInputs shape:\n", inputs.shape)
```

The preceding code prints the following output:

```
Token IDs:
 tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]])

Inputs shape:
 torch.Size([8, 4])
```

As we can see, the token ID tensor is 8x4-dimensional, meaning that the data batch consists of 8 text samples with 4 tokens each.

Let's now use the embedding layer to embed these token IDs into 256-dimensional vectors:

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)
```

The preceding print function call returns the following:

```
torch.Size([8, 4, 256])
```

As we can tell based on the 8x4x256-dimensional tensor output, each token ID is now embedded as a 256-dimensional vector.

For a GPT model's absolute embedding approach, we just need to create another embedding layer that has the same dimension as the `token_embedding_layer`:

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_lengthe, output_
pos_embeddings = pos_embedding_layer(torch.arange(context_length)
print(pos_embeddings.shape)
```

As shown in the preceding code example, the input to the `pos_embeddings` is usually a placeholder vector `torch.arange(context_length)`, which contains a sequence of numbers 0, 1, ..., up to the maximum input length − 1. The `context_length` is a variable that represents the supported input size of the LLM. Here, we choose it similar to the maximum length of the input text. In practice, input text can be longer than the supported context length, in which case we have to truncate the text.

The output of the print statement is as follows:

```
torch.Size([4, 256])
```

As we can see, the positional embedding tensor consists of four 256-dimensional vectors. We can now add these directly to the token embeddings, where PyTorch will add the 4x256-dimensional `pos_embeddings` tensor to each 4x256-dimensional token embedding tensor in each of the 8 batches:

```
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)
```

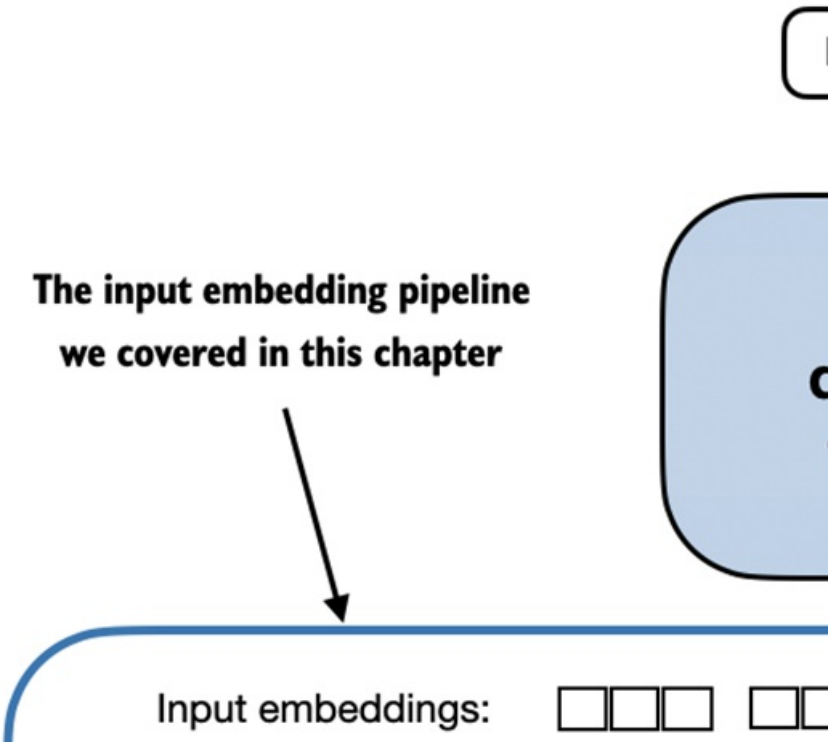
The print output is as follows:

```
torch.Size([8, 4, 256])
```

The `input_embeddings` we created, as summarized in Figure 2.19, are the

embedded input examples that can now be processed by the main LLM modules, which we will begin implementing in chapter 3

Figure 2.19 As part of the input processing pipeline, input text is first broken up into individual tokens. These tokens are then converted into token IDs using a vocabulary. The token IDs are converted into embedding vectors to which positional embeddings of a similar size are added, resulting in input embeddings that are used as input for the main LLM layers.



2.9 Summary

- LLMs require textual data to be converted into numerical vectors, known as embeddings since they can't process raw text. Embeddings transform discrete data (like words or images) into continuous vector spaces, making them compatible with neural network operations.
- As the first step, raw text is broken into tokens, which can be words or

characters. Then, the tokens are converted into integer representations, termed token IDs.

- Special tokens, such as `<|unk|>` and `<|endoftext|>`, can be added to enhance the model's understanding and handle various contexts, such as unknown words or marking the boundary between unrelated texts.
- The byte pair encoding (BPE) tokenizer used for LLMs like GPT-2 and GPT-3 can efficiently handle unknown words by breaking them down into subword units or individual characters.
- We use a sliding window approach on tokenized data to generate input-target pairs for LLM training.
- Embedding layers in PyTorch function as a lookup operation, retrieving vectors corresponding to token IDs. The resulting embedding vectors provide continuous representations of tokens, which is crucial for training deep learning models like LLMs.
- While token embeddings provide consistent vector representations for each token, they lack a sense of the token's position in a sequence. To rectify this, two main types of positional embeddings exist: absolute and relative. OpenAI's GPT models utilize absolute positional embeddings that are added to the token embedding vectors and are optimized during the model training.

3 Coding Attention Mechanisms

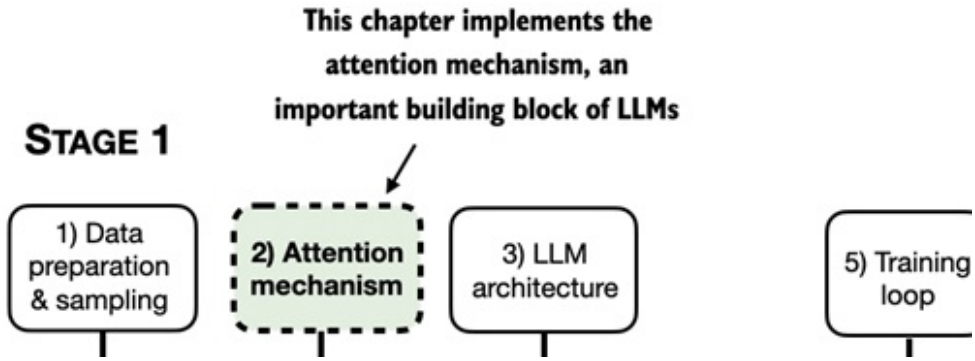
This chapter covers

- Exploring the reasons for using attention mechanisms in neural networks
- Introducing a basic self-attention framework and progressing to an enhanced self-attention mechanism
- Implementing a causal attention module that allows LLMs to generate one token at a time
- Masking randomly selected attention weights with dropout to reduce overfitting
- Stacking multiple causal attention modules into a multi-head attention module

In the previous chapter, you learned how to prepare the input text for training LLMs. This involved splitting text into individual word and subword tokens, which can be encoded into vector representations, the so-called embeddings, for the LLM.

In this chapter, we will now look at an integral part of the LLM architecture itself, attention mechanisms, as illustrated in Figure 3.1.

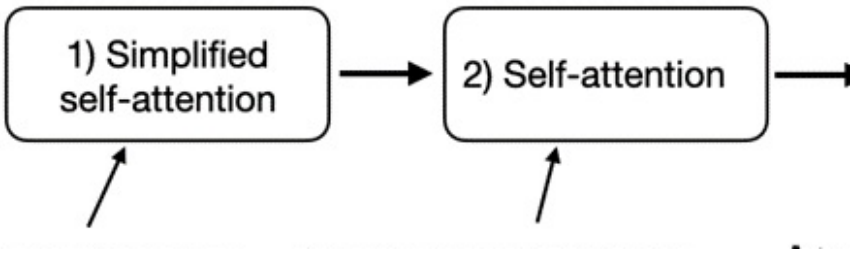
Figure 3.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset, and finetuning it on a labeled dataset. This chapter focuses on attention mechanisms, which are an integral part of an LLM architecture.



Attention mechanisms are a comprehensive topic, which is why we are devoting a whole chapter to it. We will largely look at these attention mechanisms in isolation and focus on them at a mechanistic level. In the next chapter, we will then code the remaining parts of the LLM surrounding the self-attention mechanism to see it in action and to create a model to generate text.

Over the course of this chapter, we will implement four different variants of attention mechanisms, as illustrated in Figure 3.2.

Figure 3.2 The figure depicts different attention mechanisms we will code in this chapter, starting with a simplified version of self-attention before adding the trainable weights. The causal attention mechanism adds a mask to self-attention that allows the LLM to generate one word at a time. Finally, multi-head attention organizes the attention mechanism into multiple heads, allowing the model to capture various aspects of the input data in parallel.



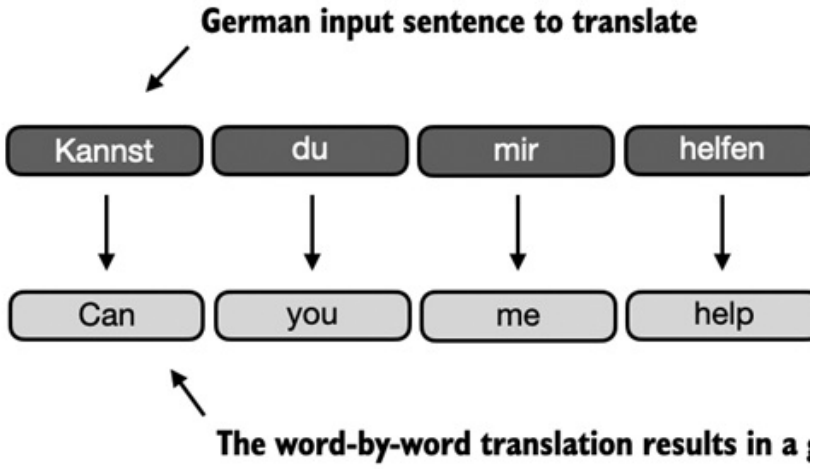
These different attention variants shown in Figure 3.2 build on each other, and the goal is to arrive at a compact and efficient implementation of multi-head attention at the end of this chapter that we can then plug into the LLM architecture we will code in the next chapter.

3.1 The problem with modeling long sequences

Before we dive into the *self-attention* mechanism that is at the heart of LLMs later in this chapter, what is the problem with architectures without attention mechanisms that predate LLMs? Suppose we want to develop a language translation model that translates text from one language into another. As shown in Figure 3.3, we can't simply translate a text word by word due to the grammatical structures in the source and target language.

Figure 3.3 When translating text from one language to another, such as German to English, it's not possible to merely translate word by word. Instead, the translation process requires

contextual understanding and grammar alignment.



To address the issue that we cannot translate text word by word, it is common to use a deep neural network with two submodules, a so-called *encoder* and *decoder*. The job of the encoder is to first read in and process the entire text, and the decoder then produces the translated text.

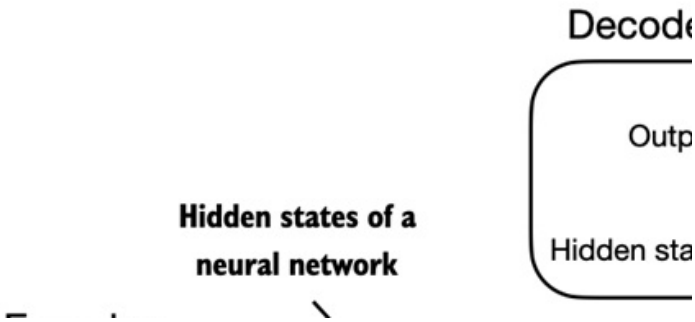
We already briefly discussed encoder-decoder networks when we introduced the transformer architecture in chapter 1 (section 1.4, Using LLMs for different tasks*)*. Before the advent of transformers, *recurrent neural networks* (RNNs) were the most popular encoder-decoder architecture for language translation.

An RNN is a type of neural network where outputs from previous steps are fed as inputs to the current step, making them well-suited for sequential data like text. If you are unfamiliar with RNNs, don't worry, you don't need to know the detailed workings of RNNs to follow this discussion; our focus here is more on the general concept of the encoder-decoder setup.

In an encoder-decoder RNN, the input text is fed into the encoder, which processes it sequentially. The encoder updates its hidden state (the internal values at the hidden layers) at each step, trying to capture the entire meaning of the input sentence in the final hidden state, as illustrated in Figure 3.4. The decoder then takes this final hidden state to start generating the translated

sentence, one word at a time. It also updates its hidden state at each step, which is supposed to carry the context necessary for the next-word prediction.

Figure 3.4 Before the advent of transformer models, encoder-decoder RNNs were a popular choice for machine translation. The encoder takes a sequence of tokens from the source language as input, where a hidden state (an intermediate neural network layer) of the encoder encodes a compressed representation of the entire input sequence. Then, the decoder uses its current hidden state to begin the translation, token by token.



While we don't need to know the inner workings of these encoder-decoder RNNs, the key idea here is that the encoder part processes the entire input text into a hidden state (memory cell). The decoder then takes in this hidden state to produce the output. You can think of this hidden state as an embedding vector, a concept we discussed in chapter 2.

The big issue and limitation of encoder-decoder RNNs is that the RNN can't directly access earlier hidden states from the encoder during the decoding phase. Consequently, it relies solely on the current hidden state, which encapsulates all relevant information. This can lead to a loss of context, especially in complex sentences where dependencies might span long distances.

For readers unfamiliar with RNNs, it is not essential to understand or study this architecture as we will not be using it in this book. The takeaway message of this section is that encoder-decoder RNNs had a shortcoming that motivated the design of attention mechanisms.

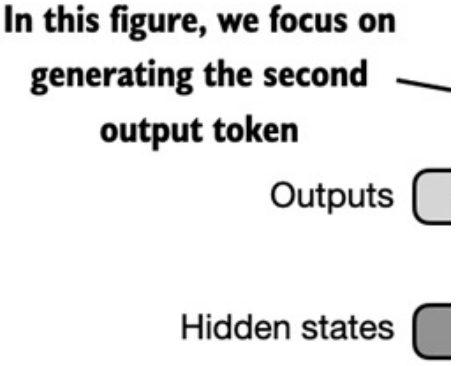
3.2 Capturing data dependencies with attention mechanisms

Before transformer LLMs, it was common to use RNNs for language modeling tasks such as language translation, as mentioned previously. RNNs work fine for translating short sentences but don't work well for longer texts as they don't have direct access to previous words in the input.

One major shortcoming in this approach is that the RNN must remember the entire encoded input in a single hidden state before passing it to the decoder, as illustrated in Figure 3.4 in the previous section.

Hence, researchers developed the so-called *Bahdanau attention* mechanism for RNNs in 2014 (named after the first author of the respective paper), which modifies the encoder-decoder RNN such that the decoder can selectively access different parts of the input sequence at each decoding step as illustrated in Figure 3.5.

Figure 3.5 Using an attention mechanism, the text-generating decoder part of the network can access all input tokens selectively. This means that some input tokens are more important than others for generating a given output token. The importance is determined by the so-called attention weights, which we will compute later. Note that this figure shows the general idea behind attention and does not depict the exact implementation of the Bahdanau mechanism, which is an RNN method outside this book's scope.



Interestingly, only three years later, researchers found that RNN architectures are not required for building deep neural networks for natural language processing and proposed the original *transformer* architecture (discussed in chapter 1) with a self-attention mechanism inspired by the Bahdanau attention mechanism.

Self-attention is a mechanism that allows each position in the input sequence to attend to all positions in the same sequence when computing the representation of a sequence. Self-attention is a key component of contemporary LLMs based on the transformer architecture, such as the GPT series.

This chapter focuses on coding and understanding this self-attention mechanism used in GPT-like models, as illustrated in Figure 3.6. In the next chapter, we will then code the remaining parts of the LLM.

Figure 3.6 Self-attention is a mechanism in transformers that is used to compute more efficient input representations by allowing each position in a sequence to interact with and weigh the importance of all other positions within the same sequence. In this chapter, we will code this self-attention mechanism from the ground up before we code the remaining parts of the GPT-like LLM in the following chapter.

3.3 Attending to different parts of the input with self-attention

We'll now delve into the inner workings of the self-attention mechanism and

learn how to code it from the ground up. Self-attention serves as the cornerstone of every LLM based on the transformer architecture. It's worth noting that this topic may require a lot of focus and attention (no pun intended), but once you grasp its fundamentals, you will have conquered one of the toughest aspects of this book and implementing LLMs in general.

The "self" in self-attention

In self-attention, the "self" refers to the mechanism's ability to compute attention weights by relating different positions within a single input sequence. It assesses and learns the relationships and dependencies between various parts of the input itself, such as words in a sentence or pixels in an image. This is in contrast to traditional attention mechanisms, where the focus is on the relationships between elements of two different sequences, such as in sequence-to-sequence models where the attention might be between an input sequence and an output sequence, such as the example depicted in Figure 3.5.

Since self-attention can appear complex, especially if you are encountering it for the first time, we will begin by introducing a simplified version of self-attention in the next subsection. Afterwards, in section 3.4, we will then implement the self-attention mechanism with trainable weights, which is used in LLMs.

3.3.1 A simple self-attention mechanism without trainable weights

In this section, we implement a simplified variant of self-attention, free from any trainable weights, which is summarized in Figure 3.7. The goal of this section is to illustrate a few key concepts in self-attention before adding trainable weights next in section 3.4.

Figure 3.7 The goal of self-attention is to compute a context vector, for each input element, that combines information from all other input elements. In the example depicted in this figure, we compute the context vector $z^{(2)}$. The importance or contribution of each input element for computing $z^{(2)}$ is determined by the attention weights α_{21} to α_{2T}. When computing $z^{(2)}$, the attention weights are calculated with respect to input element $x^{(2)}$ and all other inputs. The exact

computation of these attention weights is discussed later in this section.

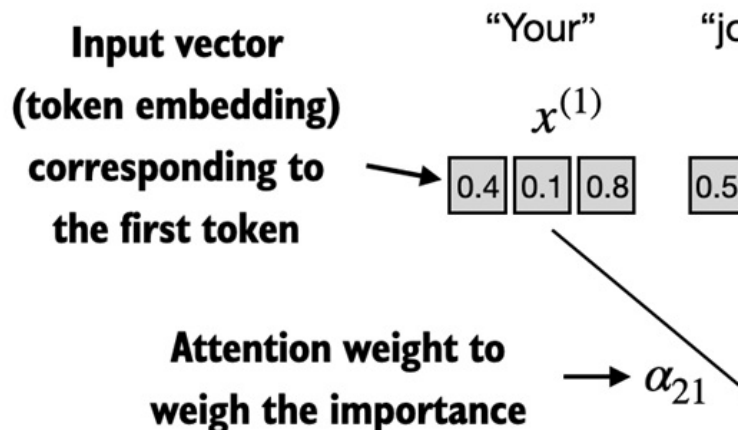


Figure 3.7 shows an input sequence, denoted as x, consisting of T elements represented as $x^{(1)}$ to $x^{(T)}$. This sequence typically represents text, such as a sentence, that has already been transformed into token embeddings, as explained in chapter 2.

For example, consider an input text like *"Your journey starts with one step."* In this case, each element of the sequence, such as $x^{(1)}$, corresponds to a d-dimensional embedding vector representing a specific token, like "Your." In Figure 3.7, these input vectors are shown as 3-dimensional embeddings.

In self-attention, our goal is to calculate context vectors $z^{(i)}$ for each element $x^{(i)}$ in the input sequence. A *context vector* can be interpreted as an enriched embedding vector.

To illustrate this concept, let's focus on the embedding vector of the second input element, $x^{(2)}$ (which corresponds to the token "journey"), and the corresponding context vector, $z^{(2)}$, shown at the bottom of Figure 3.7. This enhanced context vector, $z^{(2)}$, is an embedding that contains information about $x^{(2)}$ and all other input elements $x^{(1)}$ to $x^{(T)}$.

In self-attention, context vectors play a crucial role. Their purpose is to create enriched representations of each element in an input sequence (like a sentence) by incorporating information from all other elements in the sequence, as illustrated in Figure 3.7. This is essential in LLMs, which need

to understand the relationship and relevance of words in a sentence to each other. Later, we will add trainable weights that help an LLM learn to construct these context vectors so that they are relevant for the LLM to generate the next token.

In this section, we implement a simplified self-attention mechanism to compute these weights and the resulting context vector one step at a time.

Consider the following input sentence, which has already been embedded into 3-dimensional vectors as discussed in chapter 2. We choose a small embedding dimension for illustration purposes to ensure it fits on the page without line breaks:

```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

The first step of implementing self-attention is to compute the intermediate values ω, referred to as attention scores, as illustrated in Figure 3.8.

Figure 3.8 The overall goal of this section is to illustrate the computation of the context vector $z^{(2)}$ using the second input sequence, $x^{(2)}$ as a query. This figure shows the first intermediate step, computing the attention scores ω between the query $x^{(2)}$ and all other input elements as a dot product. (Note that the numbers in the figure are truncated to one digit after the decimal point to reduce visual clutter.)

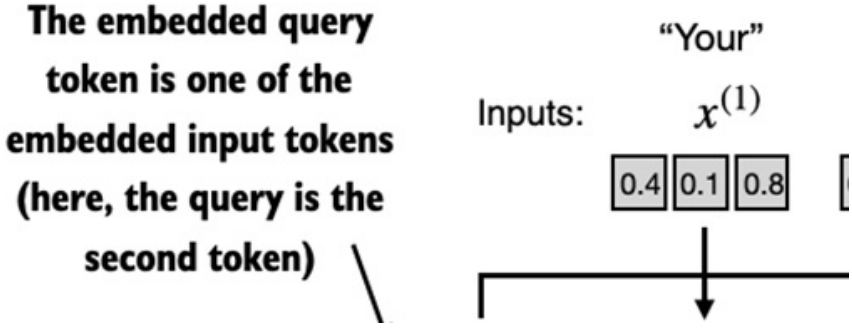


Figure 3.8 illustrates how we calculate the intermediate attention scores between the query token and each input token. We determine these scores by computing the dot product of the query, $x^{(2)}$, with every other input token:

```
query = inputs[1]  #A
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query)
print(attn_scores_2)
```

The computed attention scores are as follows:

```
tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

Understanding dot products

A dot product is essentially just a concise way of multiplying two vectors element-wise and then summing the products, which we can demonstrate as follows:

```
res = 0.

for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]
print(res)
print(torch.dot(inputs[0], query))
```

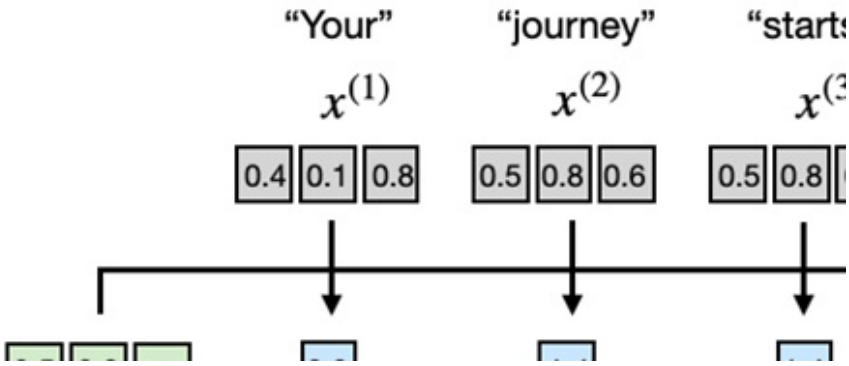
The outputs confirms that the sum of the element-wise multiplication gives the same results as the dot product:

```
tensor(0.9544)
tensor(0.9544)
```

Beyond viewing the dot product operation as a mathematical tool that combines two vectors to yield a scalar value, the dot product is a measure of similarity because it quantifies how much two vectors are aligned: a higher dot product indicates a greater degree of alignment or similarity between the vectors. In the context of self-attention mechanisms, the dot product determines the extent to which elements in a sequence attend to each other: the higher the dot product, the higher the similarity and attention score between two elements.

In the next step, as shown in Figure 3.9, we normalize each of the attention scores that we computed previously.

Figure 3.9 After computing the attention scores ω_{21} to ω_{2T} with respect to the input query $x^{(2)}$, the next step is to obtain the attention weights α_{21} to α_{2T} by normalizing the attention scores.



The main goal behind the normalization shown in Figure 3.9 is to obtain attention weights that sum up to 1. This normalization is a convention that is useful for interpretation and for maintaining training stability in an LLM. Here's a straightforward method for achieving this normalization step:

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print("Attention weights:", attn_weights_2_tmp)
print("Sum:", attn_weights_2_tmp.sum())
```

As the output shows, the attention weights now sum to 1:

```
Attention weights: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077
Sum: tensor(1.0000)
```

In practice, it's more common and advisable to use the softmax function for normalization. This approach is better at managing extreme values and offers more favorable gradient properties during training. Below is a basic implementation of the softmax function for normalizing the attention scores:

```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)

attn_weights_2_naive = softmax_naive(attn_scores_2)
print("Attention weights:", attn_weights_2_naive)
```

```
print("Sum:", attn_weights_2_naive.sum())
```

As the output shows, the softmax function also meets the objective and normalizes the attention weights such that they sum to 1:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082
Sum: tensor(1.)
```

In addition, the softmax function ensures that the attention weights are always positive. This makes the output interpretable as probabilities or relative importance, where higher weights indicate greater importance.

Note that this naive softmax implementation (`softmax_naive`) may encounter numerical instability problems, such as overflow and underflow, when dealing with large or small input values. Therefore, in practice, it's advisable to use the PyTorch implementation of softmax, which has been extensively optimized for performance:

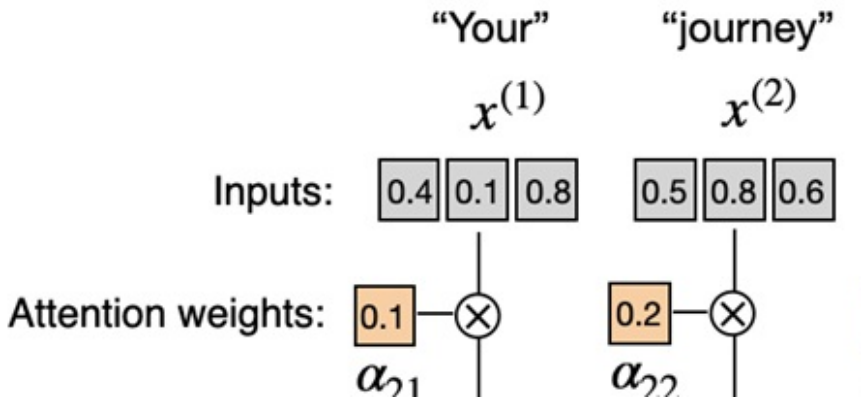
```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
```

In this case, we can see that it yields the same results as our previous `softmax_naive` function:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082
Sum: tensor(1.)
```

Now that we computed the normalized attention weights, we are ready for the final step illustrated in Figure 3.10: calculating the context vector $z^{(2)}$ by multiplying the embedded input tokens, $x^{(i)}$, with the corresponding attention weights and then summing the resulting vectors.

Figure 3.10 The final step, after calculating and normalizing the attention scores to obtain the attention weights for query $x^{(2)}$, is to compute the context vector $z^{(2)}$. This context vector is a combination of all input vectors $x^{(1)}$ *to* $x^{(T)}$ weighted by the attention weights.



The context vector $z^{(2)}$ depicted in Figure 3.10 is calculated as a weighted sum of all input vectors. This involves multiplying each input vector by its corresponding attention weight:

```
query = inputs[1] # 2nd input token is the query
context_vec_2 = torch.zeros(query.shape)
for i,x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

The results of this computation are as follows:

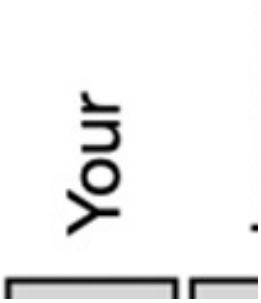
```
tensor([0.4419, 0.6515, 0.5683])
```

In the next section, we will generalize this procedure for computing context vectors to calculate all context vectors simultaneously.

3.3.2 Computing attention weights for all input tokens

In the previous section, we computed attention weights and the context vector for input 2, as shown in the highlighted row in Figure 3.11. Now, we are extending this computation to calculate attention weights and context vectors for all inputs.

Figure 3.11 The highlighted row shows the attention weights for the second input element as a query, as we computed in the previous section. This section generalizes the computation to obtain all other attention weights.



We follow the same three steps as before, as summarized in Figure 3.12, except that we make a few modifications in the code to compute all context vectors instead of only the second context vector, $z^{(2)}$.

Figure 3.12

First, in step 1 as illustrated in Figure 3.12, we add an additional for-loop to compute the dot products for all pairs of inputs.

```
attn_scores = torch.empty(6, 6)
for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        attn_scores[i, j] = torch.dot(x_i, x_j)
print(attn_scores)
```

The resulting attention scores are as follows:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

Each element in the preceding tensor represents an attention score between each pair of inputs, as illustrated in Figure 3.11. Note that the values in

Figure 3.11 are normalized, which is why they differ from the unnormalized attention scores in the preceding tensor. We will take care of the normalization later.

When computing the preceding attention score tensor, we used for-loops in Python. However, for-loops are generally slow, and we can achieve the same results using matrix multiplication:

```
attn_scores = inputs @ inputs.T
print(attn_scores)
```

We can visually confirm that the results are the same as before:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

In step 2, as illustrated in Figure 3.12, we now normalize each row so that the values in each row sum to 1:

```
attn_weights = torch.softmax(attn_scores, dim=1)
print(attn_weights)
```

This returns the following attention weight tensor that matches the values shown in Figure 3.10:

```
tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],
        [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],
        [0.1390, 0.2369, 0.2326, 0.1242, 0.1108, 0.1565],
        [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720],
        [0.1526, 0.1958, 0.1975, 0.1367, 0.1879, 0.1295],
        [0.1385, 0.2184, 0.2128, 0.1420, 0.0988, 0.1896]])
```

Before we move on to step 3, the final step shown in Figure 3.12, let's briefly verify that the rows indeed all sum to 1:

```
row_2_sum = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
print("Row 2 sum:", row_2_sum)
print("All row sums:", attn_weights.sum(dim=1))
```

The result is as follows:

```
Row 2 sum: 1.0
All row sums: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0
```

In the third and last step, we now use these attention weights to compute all context vectors via matrix multiplication:

```
all_context_vecs = attn_weights @ inputs
print(all_context_vecs)
```

In the resulting output tensor, each row contains a 3-dimensional context vector:

```
tensor([[0.4421, 0.5931, 0.5790],
        [0.4419, 0.6515, 0.5683],
        [0.4431, 0.6496, 0.5671],
        [0.4304, 0.6298, 0.5510],
        [0.4671, 0.5910, 0.5266],
        [0.4177, 0.6503, 0.5645]])
```

We can double-check that the code is correct by comparing the 2nd row with the context vector $z^{(2)}$ that we computed previously in section 3.3.1:

```
print("Previous 2nd context vector:", context_vec_2)
```

Based on the result, we can see that the previously calculated `context_vec_2` matches the second row in the previous tensor exactly:

```
Previous 2nd context vector: tensor([0.4419, 0.6515, 0.5683])
```

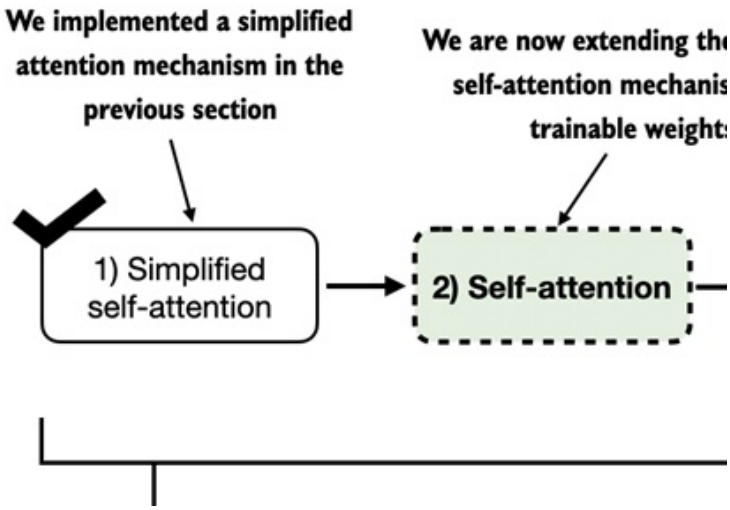
This concludes the code walkthrough of a simple self-attention mechanism. In the next section, we will add trainable weights, enabling the LLM to learn from data and improve its performance on specific tasks.

3.4 Implementing self-attention with trainable weights

In this section, we are implementing the self-attention mechanism that is used in the original transformer architecture, the GPT models, and most other

popular LLMs. This self-attention mechanism is also called *scaled dot-product attention*. Figure 3.13 provides a mental model illustrating how this self-attention mechanism fits into the broader context of implementing an LLM.

Figure 3.13 A mental model illustrating how the self-attention mechanism we code in this section fits into the broader context of this book and chapter. In the previous section, we coded a simplified attention mechanism to understand the basic mechanism behind attention mechanisms. In this section, we add trainable weights to this attention mechanism. In the upcoming sections, we will then extend this self-attention mechanism by adding a causal mask and multiple heads.



As illustrated in Figure 3.13 the self-attention mechanism with trainable weights builds on the previous concepts: we want to compute context vectors as weighted sums over the input vectors specific to a certain input element. As you will see, there are only slight differences compared to the basic self-attention mechanism we coded earlier in section 3.3.

The most notable difference is the introduction of weight matrices that are updated during model training. These trainable weight matrices are crucial so that the model (specifically, the attention module inside the model) can learn to produce "good" context vectors. (Note that we will train the LLM in chapter 5.)

We will tackle this self-attention mechanism in the two subsections. First, we will code it step-by-step as before. Second, we will organize the code into a compact Python class that can be imported into an LLM architecture, which

we will code in chapter 4.

3.4.1 Computing the attention weights step by step

We will implement the self-attention mechanism step by step by introducing the three trainable weight matrices W_q, W_k, and W_v. These three matrices are used to project the embedded input tokens, $x^{(i)}$, into query, key, and value vectors as illustrated in Figure 3.14.

Figure 3.14 In the first step of the self-attention mechanism with trainable weight matrices, we compute query (*q*), key (*k*), and value (*v*) vectors for input elements *x*. Similar to previous sections, we designate the second input, $x^{(2)}$, as the query input. The query vector $q^{(2)}$ is obtained via matrix multiplication between the input $x^{(2)}$ and the weight matrix W_q. Similarly, we obtain the key and value vectors via matrix multiplication involving the weight matrices W_k and W_v.

Earlier in section 3.3.1, we defined the second input element $x^{(2)}$ as the query when we computed the simplified attention weights to compute the context vector $z^{(2)}$. Later, in section 3.3.2, we generalized this to compute all context vectors $z^{(1)}$ ... $z^{(T)}$ for the six-word input sentence *"Your journey starts with one step."*

Similarly, we will start by computing only one context vector, $z^{(2)}$, for illustration purposes. In the next section, we will modify this code to calculate all context vectors.

Let's begin by defining a few variables:

```
x_2 = inputs[1] #A
d_in = inputs.shape[1] #B
d_out = 2 #C
```

Note that in GPT-like models, the input and output dimensions are usually the same, but for illustration purposes, to better follow the computation, we choose different input (`d_in=3`) and output (`d_out=2`) dimensions here.

Next, we initialize the three weight matrices W_q, W_k, and W_v that are shown in Figure 3.14:

```
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_gr
W_key   = torch.nn.Parameter(torch.rand(d_in, d_out), requires_gr
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_gr
```

Note that we are setting `requires_grad=False` to reduce clutter in the outputs for illustration purposes, but if we were to use the weight matrices for model training, we would set `requires_grad=True` to update these matrices during model training.

Next, we compute the query, key, and value vectors as shown earlier in Figure 3.14:

```
query_2 = x_2 @ W_query
key_2 = x_2 @ W_key
value_2 = x_2 @ W_value
print(query_2)
```

As we can see based on the output for the query, this results in a 2-dimensional vector since we set the number of columns of the corresponding weight matrix, via `d_out`, to 2:

```
tensor([0.4306, 1.4551])
```

Weight parameters vs attention weights

Note that in the weight matrices *W*, the term "weight" is short for "weight parameters," the values of a neural network that are optimized during training. This is not to be confused with the attention weights. As we already saw in the previous section, attention weights determine the extent to which a

context vector depends on the different parts of the input, i.e., to what extent the network focuses on different parts of the input.

In summary, weight parameters are the fundamental, learned coefficients that define the network's connections, while attention weights are dynamic, context-specific values.

Even though our temporary goal is to only compute the one context vector, $z^{(2)}$, we still require the key and value vectors for all input elements as they are involved in computing the attention weights with respect to the query $q^{(2)}$, as illustrated in Figure 3.14.

We can obtain all keys and values via matrix multiplication:

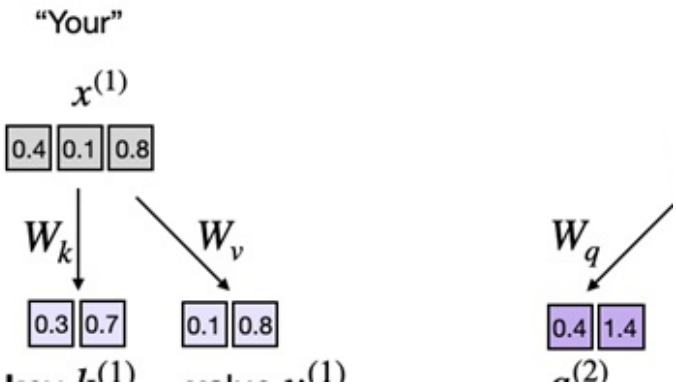
```
keys = inputs @ W_key
values = inputs @ W_value
print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

As we can tell from the outputs, we successfully projected the 6 input tokens from a 3D onto a 2D embedding space:

```
keys.shape: torch.Size([6, 2])
values.shape: torch.Size([6, 2])
```

The second step is now to compute the attention scores, as shown in Figure 3.15.

Figure 3.15 The attention score computation is a dot-product computation similar to what we have used in the simplified self-attention mechanism in section 3.3. The new aspect here is that we are not directly computing the dot-product between the input elements but using the query and key obtained by transforming the inputs via the respective weight matrices.



First, let's compute the attention score ω_{22}:

```
keys_2 = keys[1] #A
attn_score_22 = query_2.dot(keys_2)
print(attn_score_22)
```

The results in the following unnormalized attention score:

```
tensor(1.8524)
```

Again, we can generalize this computation to all attention scores via matrix multiplication:

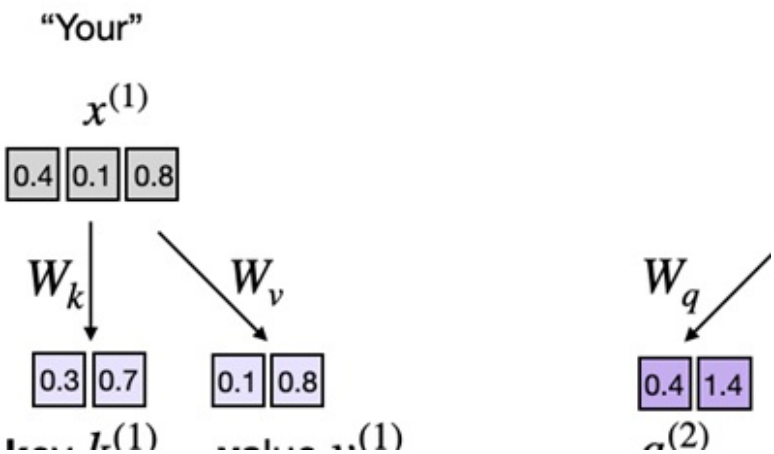
```
attn_scores_2 = query_2 @ keys.T # All attention scores for given
print(attn_scores_2)
```

As we can see, as a quick check, the second element in the output matches `attn_score_22` we computed previously:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

The third step is now going from the attention scores to the attention weights, as illustrated in Figure 3.16.

Figure 3.16 After computing the attention scores *ω*, the next step is to normalize these scores using the softmax function to obtain the attention weights *α*.



Next, as illustrated in Figure 3.16, we compute the attention weights by scaling the attention scores and using the softmax function we used earlier.. The difference to earlier is that we now scale the attention scores by dividing them by the square root of the embedding dimension of the keys, (note that taking the square root is mathematically the same as exponentiating by 0.5):

```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

The resulting attention weights are as follows:

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

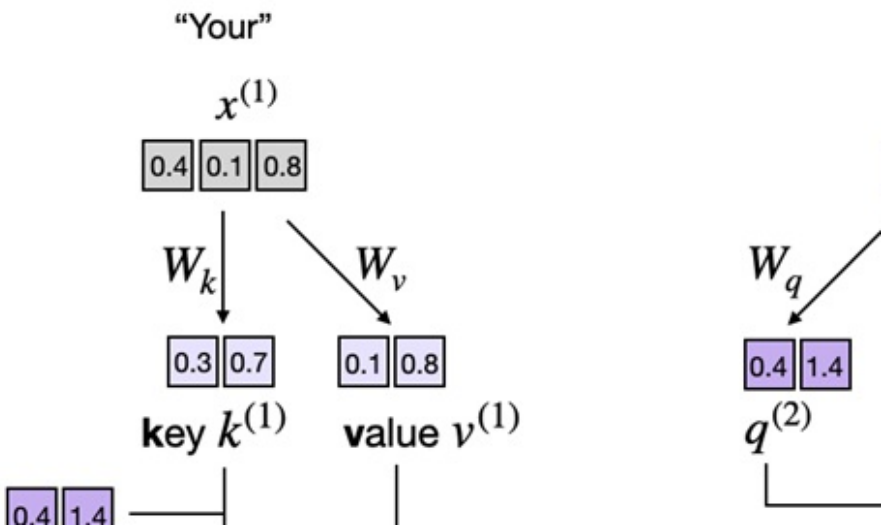
The rationale behind scaled-dot product attention

The reason for the normalization by the embedding dimension size is to improve the training performance by avoiding small gradients. For instance, when scaling up the embedding dimension, which is typically greater than thousand for GPT-like LLMs, large dot products can result in very small gradients during backpropagation due to the softmax function applied to them. As dot products increase, the softmax function behaves more like a step function, resulting in gradients nearing zero. These small gradients can drastically slow down learning or cause training to stagnate.

The scaling by the square root of the embedding dimension is the reason why this self-attention mechanism is also called scaled-dot product attention.

Now, the final step is to compute the context vectors, as illustrated in Figure 3.17.

Figure 3.17 In the final step of the self-attention computation, we compute the context vector by combining all value vectors via the attention weights.



Similar to section 3.3, where we computed the context vector as a weighted sum over the input vectors, we now compute the context vector as a weighted sum over the value vectors. Here, the attention weights serve as a weighting factor that weighs the respective importance of each value vector. Similar to section 3.3, we can use matrix multiplication to obtain the output in one step:

```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

The contents of the resulting vector are as follows:

```
tensor([0.3061, 0.8210])
```

So far, we only computed a single context vector, $z^{(2)}$. In the next section, we will generalize the code to compute all context vectors in the input sequence, $z^{(1)}$ to $z^{(T)}$.

Why query, key, and value?

The terms "key," "query," and "value" in the context of attention mechanisms are borrowed from the domain of information retrieval and databases, where similar concepts are used to store, search, and retrieve information.

A "query" is analogous to a search query in a database. It represents the current item (e.g., a word or token in a sentence) the model focuses on or tries to understand. The query is used to probe the other parts of the input sequence to determine how much attention to pay to them.

The "key" is like a database key used for indexing and searching. In the attention mechanism, each item in the input sequence (e.g., each word in a sentence) has an associated key. These keys are used to match with the query.

The "value" in this context is similar to the value in a key-value pair in a database. It represents the actual content or representation of the input items. Once the model determines which keys (and thus which parts of the input) are most relevant to the query (the current focus item), it retrieves the corresponding values.

3.4.2 Implementing a compact self-attention Python class

In the previous sections, we have gone through a lot of steps to compute the self-attention outputs. This was mainly done for illustration purposes so we could go through one step at a time. In practice, with the LLM implementation in the next chapter in mind, it is helpful to organize this code into a Python class as follows:

Listing 3.1 A compact self-attention class

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))

    def forward(self, x):
        keys = x @ self.W_key
```

```
        queries = x @ self.W_query
        values = x @ self.W_value
        attn_scores = queries @ keys.T # omega
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1)
        context_vec = attn_weights @ values
        return context_vec
```

In this PyTorch code, `SelfAttention_v1` is a class derived from `nn.Module`, which is a fundamental building block of PyTorch models, which provides necessary functionalities for model layer creation and management.

The `__init__` method initializes trainable weight matrices (`W_query`, `W_key`, and `W_value`) for queries, keys, and values, each transforming the input dimension `d_in` to an output dimension `d_out`.

During the forward pass, using the forward method, we compute the attention scores (`attn_scores`) by multiplying queries and keys, normalizing these scores using softmax. Finally, we create a context vector by weighting the values with these normalized attention scores.

We can use this class as follows:

```
torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

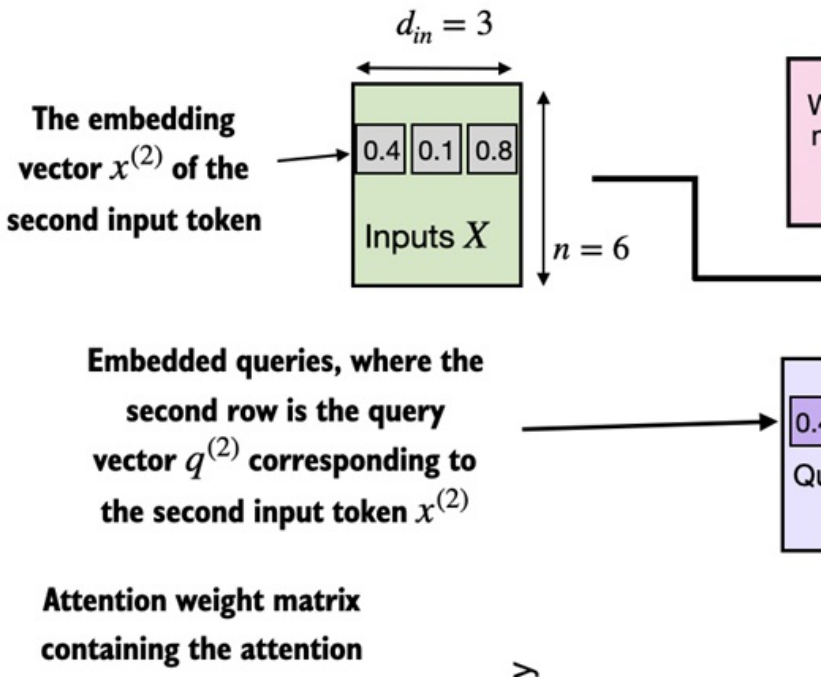
Since `inputs` contains six embedding vectors, this result in a matrix storing the six context vectors:

```
tensor([[0.2996, 0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990, 0.8040]], grad_fn=<MmBackward0>)
```

As a quick check, notice how the second row (`[0.3061, 0.8210]`) matches the contents of `context_vec_2` in the previous section.

Figure 3.18 summarizes the self-attention mechanism we just implemented.

Figure 3.18 In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, Wq, Wk, and Wv. Then, we compute the attention weight matrix based on the resulting queries (Q) and keys (K). Using the attention weights and values (V), we then compute the context vectors (Z). (For visual clarity, we focus on a single input text with *n* tokens in this figure, not a batch of multiple inputs. Consequently, the 3D input tensor is simplified to a 2D matrix in this context. This approach allows for a more straightforward visualization and understanding of the processes involved.)



As shown in Figure 3.18, self-attention involves the trainable weight matrices W_q, W_k, and W_v. These matrices transform input data into queries, keys, and values, which are crucial components of the attention mechanism. As the model is exposed to more data during training, it adjusts these trainable weights, as we will see in upcoming chapters.

We can improve the `SelfAttention_v1` implementation further by utilizing PyTorch's `nn.Linear` layers, which effectively perform matrix multiplication when the bias units are disabled. Additionally, a significant advantage of using `nn.Linear` instead of manually implementing `nn.Parameter(torch.rand(...))` is that `nn.Linear` has an optimized weight initialization scheme, contributing to more stable and effective model training.

Listing 3.2 A self-attention class using PyTorch's Linear layers

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]
        context_vec = attn_weights @ values
        return context_vec
```

You can use the `SelfAttention_v2` similar to `SelfAttention_v1`:

```
torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

The output is:

```
tensor([[-0.0739,  0.0713],
        [-0.0748,  0.0703],
        [-0.0749,  0.0702],
        [-0.0760,  0.0685],
        [-0.0763,  0.0679],
        [-0.0754,  0.0693]], grad_fn=<MmBackward0>)
```

Note that `SelfAttention_v1` and `SelfAttention_v2` give different outputs because they use different initial weights for the weight matrices since `nn.Linear` uses a more sophisticated weight initialization scheme.

Exercise 3.1 Comparing SelfAttention_v1 and SelfAttention_v2

Note that `nn.Linear` in `SelfAttention_v2` uses a different weight initialization scheme as `nn.Parameter(torch.rand(d_in, d_out))` used in `SelfAttention_v1`, which causes both mechanisms to produce different results. To check that both implementations, `SelfAttention_v1` and `SelfAttention_v2`, are otherwise similar, we can transfer the weight

matrices from a `SelfAttention_v2` object to a `SelfAttention_v1`, such that both objects then produce the same results.

Your task is to correctly assign the weights from an instance of `SelfAttention_v2` to an instance of `SelfAttention_v1`. To do this, you need to understand the relationship between the weights in both versions. (Hint: `nn.Linear` stores the weight matrix in a transposed form.) After the assignment, you should observe that both instances produce the same outputs.

In the next section, we will make enhancements to the self-attention mechanism, focusing specifically on incorporating causal and multi-head elements. The causal aspect involves modifying the attention mechanism to prevent the model from accessing future information in the sequence, which is crucial for tasks like language modeling, where each word prediction should only depend on previous words.

The multi-head component involves splitting the attention mechanism into multiple "heads." Each head learns different aspects of the data, allowing the model to simultaneously attend to information from different representation subspaces at different positions. This improves the model's performance in complex tasks.

3.5 Hiding future words with causal attention

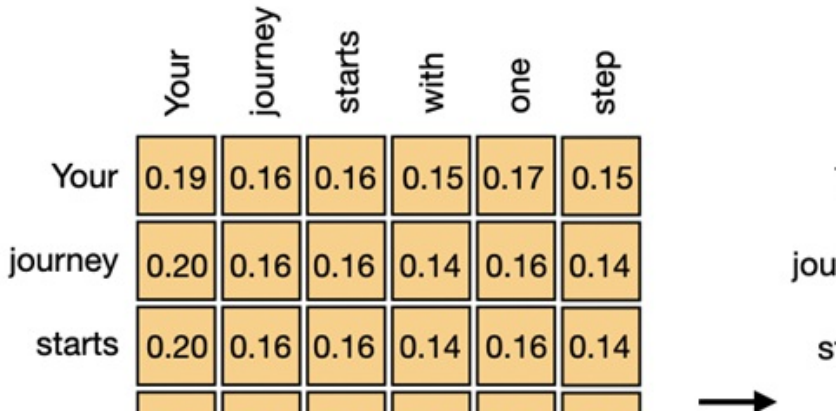
In this section, we modify the standard self-attention mechanism to create a *causal attention* mechanism, which is essential for developing an LLM in the subsequent chapters.

Causal attention, also known as *masked attention*, is a specialized form of self-attention. It restricts a model to only consider previous and current inputs in a sequence when processing any given token. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.

Consequently, when computing attention scores, the causal attention mechanism ensures that the model only factors in tokens that occur at or before the current token in the sequence.

To achieve this in GPT-like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text, as illustrated in Figure 3.19.

Figure 3.19 In causal attention, we mask out the attention weights above the diagonal such that for a given input, the LLM can't access future tokens when computing the context vectors using the attention weights. For example, for the word "journey" in the second row, we only keep the attention weights for the words before ("Your") and in the current position ("journey").

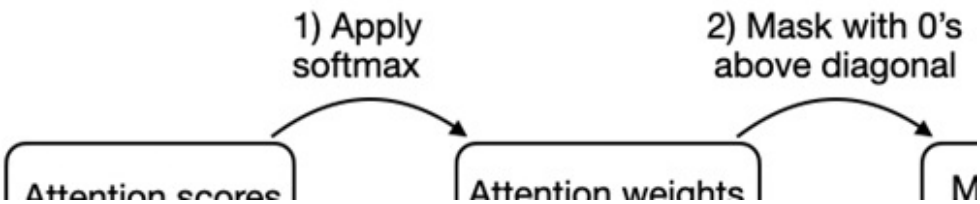


As illustrated in Figure 3.19, we mask out the attention weights above the diagonal, and we normalize the non-masked attention weights, such that the attention weights sum to 1 in each row. In the next section, we will implement this masking and normalization procedure in code.

3.5.1 Applying a causal attention mask

In this section, we implement the causal attention mask in code. We start with the procedure summarized in Figure 3.20.

Figure 3.20 One way to obtain the masked attention weight matrix in causal attention is to apply the softmax function to the attention scores, zeroing out the elements above the diagonal and normalizing the resulting matrix.



To implement the steps to apply a causal attention mask to obtain the masked attention weights as summarized in Figure 3.20, let's work with the attention scores and weights from the previous section to code the causal attention mechanism.

In the first step illustrated in Figure 3.20, we compute the attention weights using the softmax function as we have done in previous sections:

```
queries = sa_v2.W_query(inputs)  #A
keys = sa_v2.W_key(inputs)
attn_scores = queries @ keys.T
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, d
print(attn_weights)
```

This results in the following attention weights:

```
tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
        [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
        [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<SoftmaxBackward0>)
```

We can implement step 2 in Figure 3.20 using PyTorch's `tril` function to create a mask where the values above the diagonal are zero:

```
context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_lengt
print(mask_simple)
```

The resulting mask is as follows:

```
tensor([[1., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0.],
        [1., 1., 1., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1., 1.]])
```

Now, we can multiply this mask with the attention weights to zero out the values above the diagonal:

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

As we can see, the elements above the diagonal are successfully zeroed out:

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<MulBackward0>)
```

The third step in Figure 3.20 is to renormalize the attention weights to sum up to 1 again in each row. We can achieve this by dividing each element in each row by the sum in each row:

```
row_sums = masked_simple.sum(dim=1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

The result is an attention weight matrix where the attention weights above the diagonal are zeroed out and where the rows sum to 1:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<DivBackward0>)
```

Information leakage

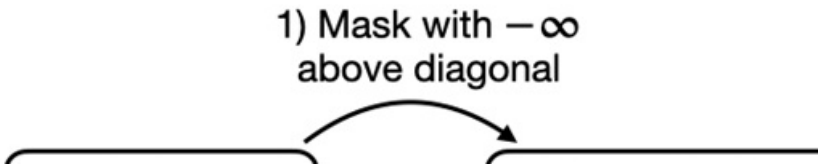
When we apply a mask and then renormalize the attention weights, it might initially appear that information from future tokens (which we intend to mask) could still influence the current token because their values are part of the softmax calculation. However, the key insight is that when we renormalize the attention weights after masking, what we're essentially doing is recalculating the softmax over a smaller subset (since masked positions don't contribute to the softmax value).

The mathematical elegance of softmax is that despite initially including all positions in the denominator, after masking and renormalizing, the effect of the masked positions is nullified — they don't contribute to the softmax score in any meaningful way.

In simpler terms, after masking and renormalization, the distribution of attention weights is as if it was calculated only among the unmasked positions to begin with. This ensures there's no information leakage from future (or otherwise masked) tokens as we intended.

While we could be technically done with implementing causal attention at this point, we can take advantage of a mathematical property of the softmax function and implement the computation of the masked attention weights more efficiently in fewer steps, as shown in Figure 3.21.

Figure 3.21 A more efficient way to obtain the masked attention weight matrix in causal attention is to mask the attention scores with negative infinity values before applying the softmax function.



The softmax function converts its inputs into a probability distribution. When negative infinity values (-∞) are present in a row, the softmax function treats them as zero probability. (Mathematically, this is because $e^{-\infty}$ approaches 0.)

We can implement this more efficient masking "trick" by creating a mask with 1's above the diagonal and then replacing these 1's with negative infinity (`-inf`) values:

```
mask = torch.triu(torch.ones(context_length, context_length), dia
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

This results in the following mask:

```
tensor([[0.2899,   -inf,   -inf,   -inf,   -inf,   -inf],
        [0.4656, 0.1723,   -inf,   -inf,   -inf,   -inf],
```

```
        [0.4594, 0.1703, 0.1731,   -inf,   -inf,   -inf],
        [0.2642, 0.1024, 0.1036, 0.0186,   -inf,   -inf],
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786,   -inf],
        [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]],
       grad_fn=<MaskedFillBackward0>)
```

Now, all we need to do is apply the softmax function to these masked results, and we are done:

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
```

As we can see based on the output, the values in each row sum to 1, and no further normalization is necessary:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<SoftmaxBackward0>)
```

We could now use the modified attention weights to compute the context vectors via `context_vec = attn_weights @ values`, as in section 3.4. However, in the next section, we first cover another minor tweak to the causal attention mechanism that is useful for reducing overfitting when training LLMs.

3.5.2 Masking additional attention weights with dropout

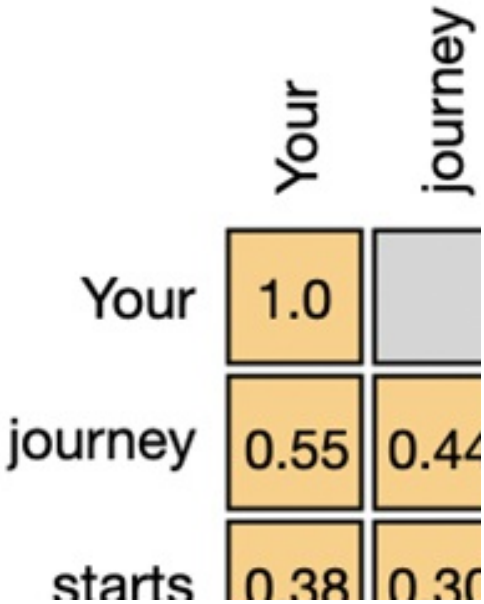
Dropout in deep learning is a technique where randomly selected hidden layer units are ignored during training, effectively "dropping" them out. This method helps prevent overfitting by ensuring that a model does not become overly reliant on any specific set of hidden layer units. It's important to emphasize that dropout is only used during training and is disabled afterward.

In the transformer architecture, including models like GPT, dropout in the attention mechanism is typically applied in two specific areas: after calculating the attention scores or after applying the attention weights to the

value vectors.

Here, we will apply the dropout mask after computing the attention weights, as illustrated in Figure 3.22, because it's the more common variant in practice.

Figure 3.22 Using the causal attention mask (upper left), we apply an additional dropout mask (upper right) to zero out additional attention weights to reduce overfitting during training.



In the following code example, we use a dropout rate of 50%, which means masking out half of the attention weights. (When we train the GPT model in later chapters, we will use a lower dropout rate, such as 0.1 or 0.2.)

In the following code, we apply PyTorch's dropout implementation first to a 6×6 tensor consisting of ones for illustration purposes:

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5) #A
example = torch.ones(6, 6) #B
print(dropout(example))
```

As we can see, approximately half of the values are zeroed out:

```
tensor([[2., 2., 0., 2., 2., 0.],
        [0., 0., 0., 2., 0., 2.],
```

```
        [2., 2., 2., 2., 0., 2.],
        [0., 2., 2., 0., 0., 2.],
        [0., 2., 0., 2., 0., 2.],
        [0., 2., 2., 2., 2., 0.]])
```

When applying dropout to an attention weight matrix with a rate of 50%, half of the elements in the matrix are randomly set to zero. To compensate for the reduction in active elements, the values of the remaining elements in the matrix are scaled up by a factor of 1/0.5 =2. This scaling is crucial to maintain the overall balance of the attention weights, ensuring that the average influence of the attention mechanism remains consistent during both the training and inference phases.

Now, let's apply dropout to the attention weight matrix itself:

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

The resulting attention weight matrix now has additional elements zeroed out and the remaining ones rescaled:

```
tensor([[2.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],
        [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],
       grad_fn=<MulBackward0>
```

Note that the resulting dropout outputs may look different depending on your operating system; you can read more about this inconsistency [here on the PyTorch issue tracker at https://github.com/pytorch/pytorch/issues/121595.

Having gained an understanding of causal attention and dropout masking, we will develop a concise Python class in the following section. This class is designed to facilitate the efficient application of these two techniques.

3.5.3 Implementing a compact causal attention class

In this section, we will now incorporate the causal attention and dropout modifications into the `SelfAttention` Python class we developed in section

3.4. This class will then serve as a template for developing *multi-head attention* in the upcoming section, which is the final attention class we implement in this chapter.

But before we begin, one more thing is to ensure that the code can handle batches consisting of more than one input so that the `CausalAttention` class supports the batch outputs produced by the data loader we implemented in chapter 2.

For simplicity, to simulate such batch inputs, we duplicate the input text example:

```
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape) #A
```

This results in a 3D tensor consisting of 2 input texts with 6 tokens each, where each token is a 3-dimensional embedding vector:

```
torch.Size([2, 6, 3])
```

The following `CausalAttention` class is similar to the `SelfAttention` class we implemented earlier, except that we now added the dropout and causal mask components as highlighted in the following code:

Listing 3.3 A compact causal attention class

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, qkv_
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)  #A
        self.register_buffer(
           'mask',
           torch.triu(torch.ones(context_length, context_length),
           diagonal=1)
        )  #B

    def forward(self, x):
        b, num_tokens, d_in = x.shape  #C
```

```
New batch dimension b
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2)  #C
        attn_scores.masked_fill_(  #D
            self.mask.bool()[:num_tokens, :num_tokens], -torch.in
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

While all added code lines should be familiar from previous sections, we now added a `self.register_buffer()` call in the `__init__` method. The use of `register_buffer` in PyTorch is not strictly necessary for all use cases but offers several advantages here. For instance, when we use the `CausalAttention` class in our LLM, buffers are automatically moved to the appropriate device (CPU or GPU) along with our model, which will be relevant when training the LLM in future chapters. This means we don't need to manually ensure these tensors are on the same device as your model parameters, avoiding device mismatch errors.

We can use the `CausalAttention` class as follows, similar to `SelfAttention` previously:

```
torch.manual_seed(123)
context_length = batch.shape[1]
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)
```

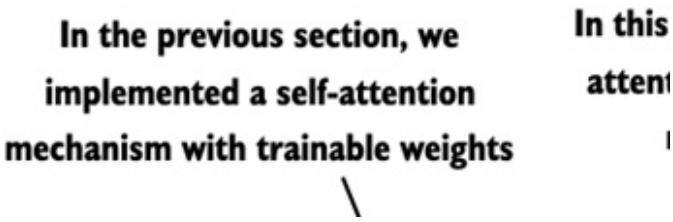
The resulting context vector is a 3D tensor where each token is now represented by a 2D embedding:

```
context_vecs.shape: torch.Size([2, 6, 2])
```

Figure 3.23 provides a mental model that summarizes what we have accomplished so far.

Figure 3.23 A mental model summarizing the four different attention modules we are coding in

this chapter. We began with a simplified attention mechanism, added trainable weights, and then added a casual attention mask. In the remainder of this chapter, we will extend the causal attention mechanism and code multi-head attention, which is the final module we will use in the LLM implementation in the next chapter.



As illustrated in Figure 3.23, in this section, we focused on the concept and implementation of causal attention in neural networks. In the next section, we will expand on this concept and implement a multi-head attention module that implements several of such causal attention mechanisms in parallel.

3.6 Extending single-head attention to multi-head attention

In this final section of this chapter, we are extending the previously implemented causal attention class over multiple-heads. This is also called *multi-head attention*.

The term "multi-head" refers to dividing the attention mechanism into multiple "heads," each operating independently. In this context, a single causal attention module can be considered single-head attention, where there is only one set of attention weights processing the input sequentially.

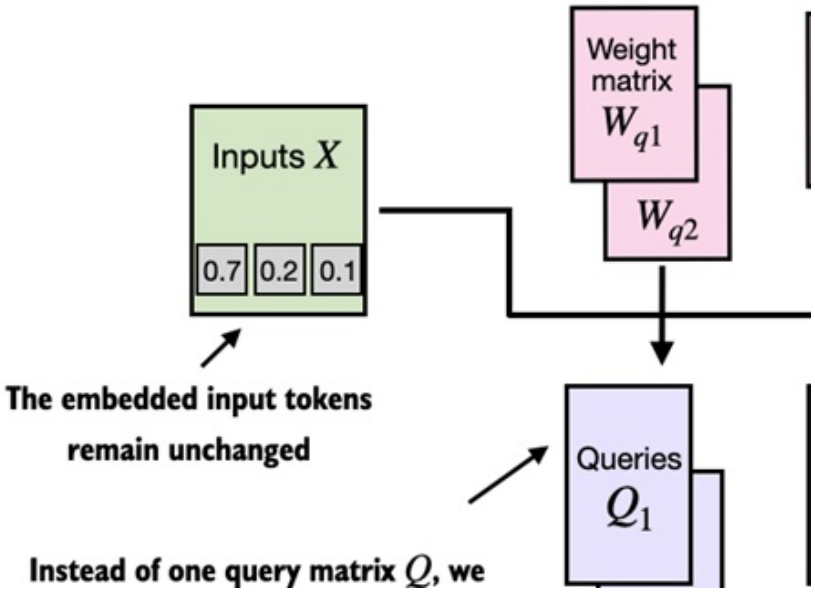
In the following subsections, we will tackle this expansion from causal attention to multi-head attention. The first subsection will intuitively build a multi-head attention module by stacking multiple `CausalAttention` modules for illustration purposes. The second subsection will then implement the same multi-head attention module in a more complicated but computationally more efficient way.

3.6.1 Stacking multiple single-head attention layers

In practical terms, implementing multi-head attention involves creating multiple instances of the self-attention mechanism (depicted earlier in Figure 3.18 in section 3.4.1), each with its own weights, and then combining their outputs. Using multiple instances of the self-attention mechanism can be computationally intensive, but it's crucial for the kind of complex pattern recognition that models like transformer-based LLMs are known for.

Figure 3.24 illustrates the structure of a multi-head attention module, which consists of multiple single-head attention modules, as previously depicted in Figure 3.18, stacked on top of each other.

Figure 3.24 The multi-head attention module in this figure depicts two single-head attention modules stacked on top of each other. So, instead of using a single matrix W_v for computing the value matrices, in a multi-head attention module with two heads, we now have two value weight matrices: W_{v1} and W_{v2}. The same applies to the other weight matrices, W_q and W_k. We obtain two sets of context vectors Z_1 and Z_2 that we can combine into a single context vector matrix Z.



As mentioned before, the main idea behind multi-head attention is to run the attention mechanism multiple times (in parallel) with different, learned linear projections -- the results of multiplying the input data (like the query, key, and value vectors in attention mechanisms) by a weight matrix.

In code, we can achieve this by implementing a simple `MultiHeadAttentionWrapper` class that stacks multiple instances of our

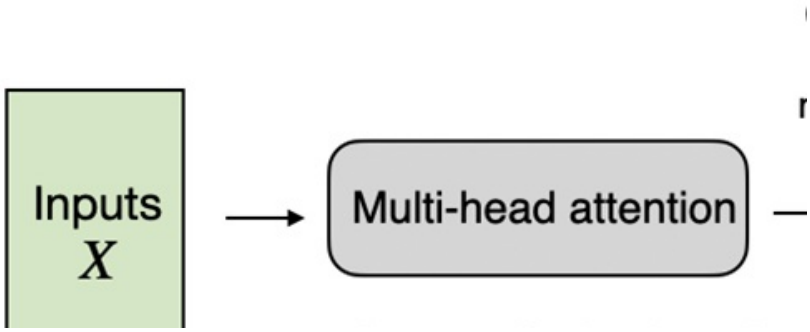
previously implemented `CausalAttention` module:

Listing 3.4 A wrapper class to implement multi-head attention

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(d_in, d_out, context_length, dropout
             for _ in range(num_heads)]
        )

    def forward(self, x):
        return torch.cat([head(x) for head in self.heads], dim=-1
```

For example, if we use this MultiHeadAttentionWrapper class with two attention heads (via `num_heads=2`) and CausalAttention output dimension `d_out=2`, this results in a 4-dimensional context vectors (`d_out*num_heads=4`), as illustrated in Figure 3.25.

Figure 3.25 Using the `MultiHeadAttentionWrapper`, we specified the number of attention heads (`num_heads`). If we set `num_heads=2`, as shown in this figure, we obtain a tensor with two sets of context vector matrices. In each context vector matrix, the rows represent the context vectors corresponding to the tokens, and the columns correspond to the embedding dimension specified via `d_out=4`. We concatenate these context vector matrices along the column dimension. Since we have 2 attention heads and an embedding dimension of 2, the final embedding dimension is 2 × 2 = 4.



To illustrate Figure 3.25 further with a concrete example, we can use the `MultiHeadAttentionWrapper` class similar to the `CausalAttention` class before:

```
torch.manual_seed(123)
context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
mha = MultiHeadAttentionWrapper(d_in, d_out, context_length, 0.0,
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

This results in the following tensor representing the context vectors:

```
tensor([[[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]],

        [[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]]], grad_fn=<CatBack
context_vecs.shape: torch.Size([2, 6, 4])
```

The first dimension of the resulting `context_vecs` tensor is 2 since we have two input texts (the input texts are duplicated, which is why the context vectors are exactly the same for those). The second dimension refers to the 6 tokens in each input. The third dimension refers to the 4-dimensional embedding of each token.

Exercise 3.2 Returning 2-dimensional embedding vectors

Change the input arguments for the `MultiHeadAttentionWrapper(..., num_heads=2)` call such that the output context vectors are 2-dimensional instead of 4-dimensional while keeping the setting `num_heads=2`. Hint: You don't have to modify the class implementation; you just have to change one of the other input arguments.

In this section, we implemented a MultiHeadAttentionWrapper that combined multiple single-head attention modules. However, note that these

are processed sequentially via `[head(x) for head in self.heads]` in the forward method. We can improve this implementation by processing the heads in parallel. One way to achieve this is by computing the outputs for all attention heads simultaneously via matrix multiplication, as we will explore in the next section.

3.6.2 Implementing multi-head attention with weight splits

In the previous section, we created a `MultiHeadAttentionWrapper` to implement multi-head attention by stacking multiple single-head attention modules. This was done by instantiating and combining several `CausalAttention` objects.

Instead of maintaining two separate classes, `MultiHeadAttentionWrapper` and `CausalAttention`, we can combine both of these concepts into a single `MultiHeadAttention` class. Also, in addition to just merging the `MultiHeadAttentionWrapper` with the `CausalAttention` code, we will make some other modifications to implement multi-head attention more efficiently.

In the `MultiHeadAttentionWrapper`, multiple heads are implemented by creating a list of `CausalAttention` objects (`self.heads`), each representing a separate attention head. The `CausalAttention` class independently performs the attention mechanism, and the results from each head are concatenated. In contrast, the following `MultiHeadAttention` class integrates the multi-head functionality within a single class. It splits the input into multiple heads by reshaping the projected query, key, and value tensors and then combines the results from these heads after computing attention.

Let's take a look at the `MultiHeadAttention` class before we discuss it further:

Listing 3.5 An efficient multi-head attention class

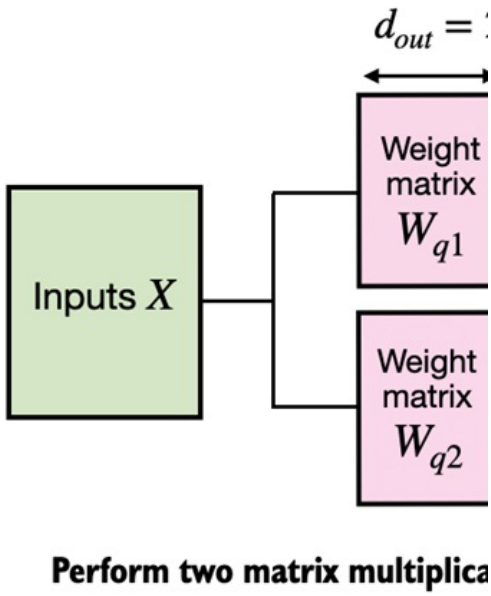
```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length, dropout, num_heads, qkv_bias=Fal
        super().__init__()
        assert d_out % num_heads == 0, "d_out must be divisible b
```

```
        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads #A
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out) #B
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            'mask',
             torch.triu(torch.ones(context_length, context_length
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x) #C
        queries = self.W_query(x) #C
        values = self.W_value(x) #C

        keys = keys.view(b, num_tokens, self.num_heads, self.head
        values = values.view(b, num_tokens, self.num_heads, self.
        queries = queries.view(b, num_tokens, self.num_heads, sel

        keys = keys.transpose(1, 2) #E
        queries = queries.transpose(1, 2) #E
        values = values.transpose(1, 2) #E

        attn_scores = queries @ keys.transpose(2, 3)  #F
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens] #G

        attn_scores.masked_fill_(mask_bool, -torch.inf) #H

        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = (attn_weights @ values).transpose(1, 2) #I
        #J
        context_vec = context_vec.contiguous().view(b, num_tokens
        context_vec = self.out_proj(context_vec) #K
        return context_vec
```

Even though the reshaping (`.view`) and transposing (`.transpose`) of tensors inside the `MultiHeadAttention` class looks very complicated, mathematically, the `MultiHeadAttention` class implements the same concept as the `MultiHeadAttentionWrapper` earlier.

On a big-picture level, in the previous `MultiHeadAttentionWrapper`, we stacked multiple single-head attention layers that we combined into a multi-head attention layer. The `MultiHeadAttention` class takes an integrated approach. It starts with a multi-head layer and then internally splits this layer into individual attention heads, as illustrated in Figure 3.26.

Figure 3.26 In the `MultiheadAttentionWrapper` class with two attention heads, we initialized two weight matrices W_{q1} and W_{q2} and computed two query matrices Q_1 and Q_2 as illustrated at the top of this figure. In the `MultiheadAttention` class, we initialize one larger weight matrix W_q, only perform one matrix multiplication with the inputs to obtain a query matrix Q, and then split the query matrix into Q_1 and Q_2 as shown at the bottom of this figure. We do the same for the keys and values, which are not shown to reduce visual clutter.



The splitting of the query, key, and value tensors, as depicted in Figure 3.26, is achieved through tensor reshaping and transposing operations using PyTorch's `.view` and `.transpose` methods. The input is first transformed (via linear layers for queries, keys, and values) and then reshaped to represent multiple heads.

The key operation is to split the `d_out` dimension into `num_heads` and `head_dim`, where `head_dim = d_out / num_heads`. This splitting is then achieved using the `.view` method: a tensor of dimensions `(b, num_tokens, d_out)` is reshaped to dimension `(b, num_tokens, num_heads, head_dim)`.

The tensors are then transposed to bring the `num_heads` dimension before the `num_tokens` dimension, resulting in a shape of `(b, num_heads, num_tokens, head_dim)`. This transposition is crucial for correctly aligning the queries, keys, and values across the different heads and performing batched matrix multiplications efficiently.

To illustrate this batched matrix multiplication, suppose we have the following example tensor:

```
a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573], #A
                    [0.8993, 0.0390, 0.9268, 0.7388],
                    [0.7179, 0.7058, 0.9156, 0.4340]],

                   [[0.0772, 0.3565, 0.1479, 0.5331],
                    [0.4066, 0.2318, 0.4545, 0.9737],
                    [0.4606, 0.5159, 0.4220, 0.5786]]]])
```

Now, we perform a batched matrix multiplication between the tensor itself and a view of the tensor where we transposed the last two dimensions, `num_tokens` and `head_dim`:

```
print(a @ a.transpose(2, 3))
```

The result is as follows:

```
tensor([[[[1.3208, 1.1631, 1.2879],
          [1.1631, 2.2150, 1.8424],
          [1.2879, 1.8424, 2.0402]],

         [[0.4391, 0.7003, 0.5903],
          [0.7003, 1.3737, 1.0620],
          [0.5903, 1.0620, 0.9912]]]])
```

In this case, the matrix multiplication implementation in PyTorch handles the 4-dimensional input tensor so that the matrix multiplication is carried out between the 2 last dimensions `(num_tokens, head_dim)` and then repeated for the individual heads.

For instance, the above becomes a more compact way to compute the matrix multiplication for each head separately:

```
first_head = a[0, 0, :, :]
```

```
first_res = first_head @ first_head.T
print("First head:\n", first_res)

second_head = a[0, 1, :, :]
second_res = second_head @ second_head.T
print("\nSecond head:\n", second_res)
```

The results are exactly the same results that we obtained when using the batched matrix multiplication `print(a @ a.transpose(2, 3))` earlier:

```
First head:
 tensor([[1.3208, 1.1631, 1.2879],
        [1.1631, 2.2150, 1.8424],
        [1.2879, 1.8424, 2.0402]])

Second head:
 tensor([[0.4391, 0.7003, 0.5903],
        [0.7003, 1.3737, 1.0620],
        [0.5903, 1.0620, 0.9912]])
```

Continuing with MultiHeadAttention, after computing the attention weights and context vectors, the context vectors from all heads are transposed back to the shape `(b, num_tokens, num_heads, head_dim)`. These vectors are then reshaped (flattened) into the shape `(b, num_tokens, d_out)`, effectively combining the outputs from all heads.

Additionally, we added a so-called output projection layer (`self.out_proj`) to `MultiHeadAttention` after combining the heads, which is not present in the `CausalAttention` class. This output projection layer is not strictly necessary (see the References section in Appendix B for more details), but it is commonly used in many LLM architectures, which is why we added it here for completeness.

Even though the `MultiHeadAttention` class looks more complicated than the `MultiHeadAttentionWrapper` due to the additional reshaping and transposition of tensors, it is more efficient. The reason is that we only need one matrix multiplication to compute the keys, for instance, `keys = self.W_key(x)` (the same is true for the queries and values). In the MultiHeadAttentionWrapper, we needed to repeat this matrix multiplication, which is computationally one of the most expensive steps, for each attention head.

The `MultiHeadAttention` class can be used similar to the `SelfAttention` and `CausalAttention` classes we implemented earlier:

```
torch.manual_seed(123)
batch_size, context_length, d_in = batch.shape
d_out = 2
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_he
context_vecs = mha(batch)
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

As we can see based on the results, the output dimension is directly controlled by the `d_out` argument:

```
tensor([[[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]],

        [[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]]], grad_fn=<ViewBackward0>)
context_vecs.shape: torch.Size([2, 6, 2])
```

In this section, we implemented the `MultiHeadAttention` class that we will use in the upcoming sections when implementing and training the LLM itself. Note that while the code is fully functional, we used relatively small embedding sizes and numbers of attention heads to keep the outputs readable.

For comparison, the smallest GPT-2 model (117 million parameters) has 12 attention heads and a context vector embedding size of 768. The largest GPT-2 model (1.5 billion parameters) has 25 attention heads and a context vector embedding size of 1600. Note that the embedding sizes of the token inputs and context embeddings are the same in GPT models (`d_in = d_out`).

Exercise 3.3 Initializing GPT-2 size attention modules

Using the `MultiHeadAttention` class, initialize a multi-head attention module that has the same number of attention heads as the smallest GPT-2 model (12 attention heads). Also ensure that you use the respective input and output embedding sizes similar to GPT-2 (768 dimensions). Note that the smallest GPT-2 model supports a context length of 1024 tokens.

3.7 Summary

- Attention mechanisms transform input elements into enhanced context vector representations that incorporate information about all inputs.
- A self-attention mechanism computes the context vector representation as a weighted sum over the inputs.
- In a simplified attention mechanism, the attention weights are computed via dot products.
- A dot product is just a concise way of multiplying two vectors element-wise and then summing the products.
- Matrix multiplications, while not strictly required, help us to implement computations more efficiently and compactly by replacing nested for-loops.
- In self-attention mechanisms that are used in LLMs, also called scaled-dot product attention, we include trainable weight matrices to compute intermediate transformations of the inputs: queries, values, and keys.
- When working with LLMs that read and generate text from left to right, we add a causal attention mask to prevent the LLM from accessing future tokens.
- Next to causal attention masks to zero out attention weights, we can also add a dropout mask to reduce overfitting in LLMs.
- The attention modules in transformer-based LLMs involve multiple instances of causal attention, which is called multi-head attention.
- We can create a multi-head attention module by stacking multiple instances of causal attention modules.
- A more efficient way of creating multi-head attention modules involves batched matrix multiplications.

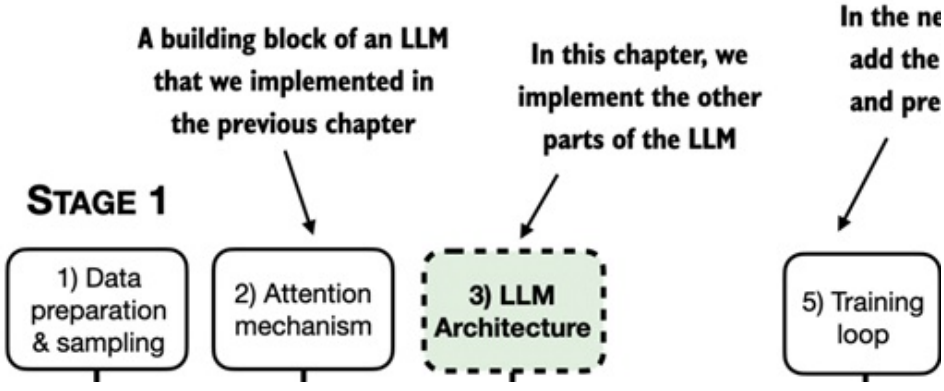
4 Implementing a GPT model from Scratch To Generate Text

This chapter covers

- Coding a GPT-like large language model (LLM) that can be trained to generate human-like text
- Normalizing layer activations to stabilize neural network training
- Adding shortcut connections in deep neural networks to train models more effectively
- Implementing transformer blocks to create GPT models of various sizes
- Computing the number of parameters and storage requirements of GPT models

In the previous chapter, you learned and coded the *multi-head attention* mechanism, one of the core components of LLMs. In this chapter, we will now code the other building blocks of an LLM and assemble them into a GPT-like model that we will train in the next chapter to generate human-like text, as illustrated in Figure 4.1.

Figure 4.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset, and finetuning it on a labeled dataset. This chapter focuses on implementing the LLM architecture, which we will train in the next chapter.



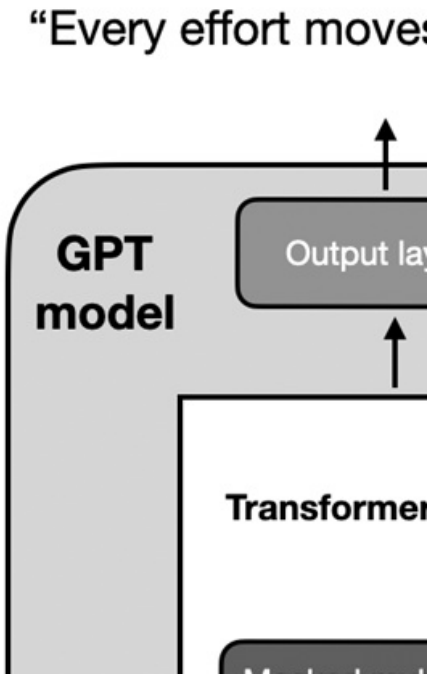
The LLM architecture, referenced in Figure 4.1, consists of several building

blocks that we will implement throughout this chapter. We will begin with a top-down view of the model architecture in the next section before covering the individual components in more detail.

4.1 Coding an LLM architecture

LLMs, such as GPT (which stands for *Generative Pretrained Transformer*), are large deep neural network architectures designed to generate new text one word (or token) at a time. However, despite their size, the model architecture is less complicated than you might think, since many of its components are repeated, as we will see later. Figure 4.2 provides a top-down view of a GPT-like LLM, with its main components highlighted.

Figure 4.2 A mental model of a GPT model. Next to the embedding layers, it consists of one or more transformer blocks containing the masked multi-head attention module we implemented in the previous chapter.



As you can see in Figure 4.2, we have already covered several aspects, such as input tokenization and embedding, as well as the masked multi-head attention module. The focus of this chapter will be on implementing the core structure of the GPT model, including its *transformer blocks*, which we will

then train in the next chapter to generate human-like text.

In the previous chapters, we used smaller embedding dimensions for simplicity, ensuring that the concepts and examples could comfortably fit on a single page. Now, in this chapter, we are scaling up to the size of a small GPT-2 model, specifically the smallest version with 124 million parameters, as described in Radford *et al.*'s paper, "Language Models are Unsupervised Multitask Learners." Note that while the original report mentions 117 million parameters, this was later corrected.

Chapter 6 will focus on loading pretrained weights into our implementation and adapting it for larger GPT-2 models with 345, 762, and 1,542 million parameters. In the context of deep learning and LLMs like GPT, the term "parameters" refers to the trainable weights of the model. These weights are essentially the internal variables of the model that are adjusted and optimized during the training process to minimize a specific loss function. This optimization allows the model to learn from the training data.

For example, in a neural network layer that is represented by a 2,048x2,048-dimensional matrix (or tensor) of weights, each element of this matrix is a parameter. Since there are 2,048 rows and 2,048 columns, the total number of parameters in this layer is 2,048 multiplied by 2,048, which equals 4,194,304 parameters.

GPT-2 versus GPT-3

Note that we are focusing on GPT-2 because OpenAI has made the weights of the pretrained model publicly available, which we will load into our implementation in chapter 6. GPT-3 is fundamentally the same in terms of model architecture, except that it is scaled up from 1.5 billion parameters in GPT-2 to 175 billion parameters in GPT-3, and it is trained on more data. As of this writing, the weights for GPT-3 are not publicly available. GPT-2 is also a better choice for learning how to implement LLMs, as it can be run on a single laptop computer, whereas GPT-3 requires a GPU cluster for training and inference. According to Lambda Labs, it would take 355 years to train GPT-3 on a single V100 datacenter GPU, and 665 years on a consumer RTX 8000 GPU.

We specify the configuration of the small GPT-2 model via the following Python dictionary, which we will use in the code examples later:

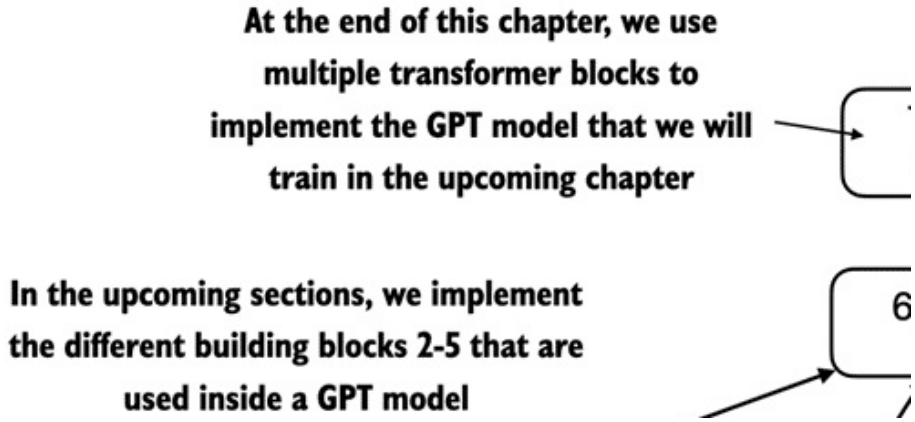
```
GPT_CONFIG_124M = {
    "vocab_size": 50257,  # Vocabulary size
    "context_length": 1024,      # Context length
    "emb_dim": 768,       # Embedding dimension
    "n_heads": 12,        # Number of attention heads
    "n_layers": 12,       # Number of layers
    "drop_rate": 0.1,     # Dropout rate
    "qkv_bias": False     # Query-Key-Value bias
}
```

In the `GPT_CONFIG_124M` dictionary, we use concise variable names for clarity and to prevent long lines of code:

- `"vocab_size"` refers to a vocabulary of 50,257 words, as used by the BPE tokenizer from chapter 2.
- `"context_length"` denotes the maximum number of input tokens the model can handle, via the positional embeddings discussed in chapter 2.
- `"emb_dim"` represents the embedding size, transforming each token into a 768-dimensional vector.
- `"n_heads"` indicates the count of attention heads in the multi-head attention mechanism, as implemented in chapter 3.
- `"n_layers"` specifies the number of transformer blocks in the model, which will be elaborated on in upcoming sections.
- `"drop_rate"` indicates the intensity of the dropout mechanism (0.1 implies a 10% drop of hidden units) to prevent overfitting, as covered in chapter 3.
- `"qkv_bias"` determines whether to include a bias vector in the `Linear` layers of the multi-head attention for query, key, and value computations. We will initially disable this, following the norms of modern LLMs, but will revisit it in chapter 6 when we load pretrained GPT-2 weights from OpenAI into our model.

Using the configuration above, we will start this chapter by implementing a GPT placeholder architecture (`DummyGPTModel`) in this section, as shown in Figure 4.3. This will provide us with a big-picture view of how everything fits together and what other components we need to code in the upcoming sections to assemble the full GPT model architecture.

Figure 4.3 A mental model outlining the order in which we code the GPT architecture. In this chapter, we will start with the GPT backbone, a placeholder architecture, before we get to the individual core pieces and eventually assemble them in a transformer block for the final GPT architecture.



The numbered boxes shown in Figure 4.3 illustrate the order in which we tackle the individual concepts required to code the final GPT architecture. We will start with step 1, a placeholder GPT backbone we call `DummyGPTModel`:

Listing 4.1 A placeholder GPT model architecture class

```
import torch
import torch.nn as nn

class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_d
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["e
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg) for _ in range(cfg["n_la
        self.final_norm = DummyLayerNorm(cfg["emb_dim"]) #B
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
```

```
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

class DummyTransformerBlock(nn.Module): #C
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x): #D
        return x

class DummyLayerNorm(nn.Module): #E
    def __init__(self, normalized_shape, eps=1e-5): #F
        super().__init__()

    def forward(self, x):
        return x
```

The `DummyGPTModel` class in this code defines a simplified version of a GPT-like model using PyTorch's neural network module (`nn.Module`). The model architecture in the `DummyGPTModel` class consists of token and positional embeddings, dropout, a series of transformer blocks (`DummyTransformerBlock`), a final layer normalization (`DummyLayerNorm`), and a linear output layer (`out_head`). The configuration is passed in via a Python dictionary, for instance, the `GPT_CONFIG_124M` dictionary we created earlier.

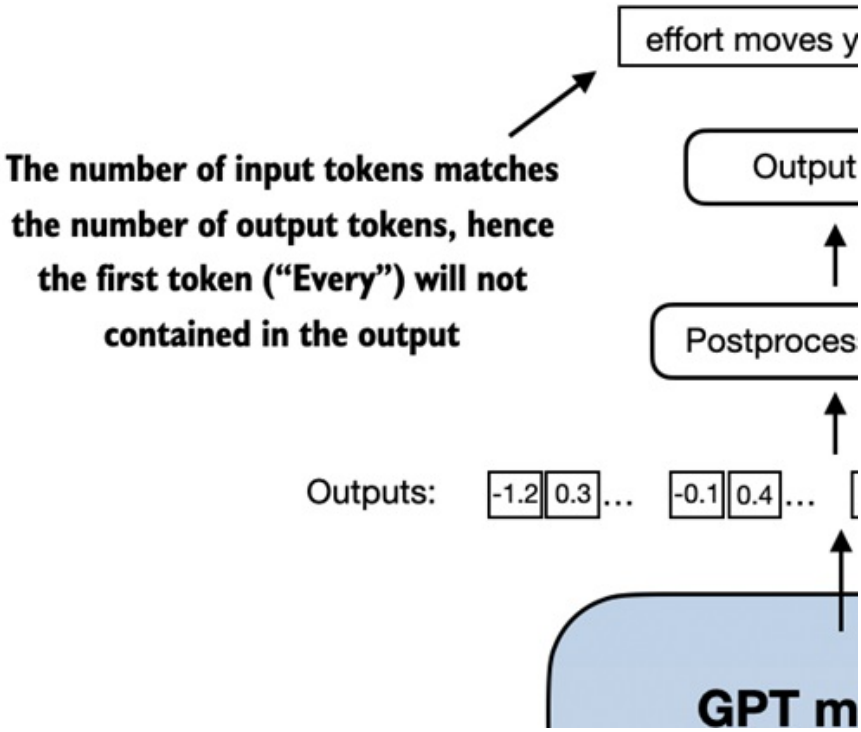
The `forward` method describes the data flow through the model: it computes token and positional embeddings for the input indices, applies dropout, processes the data through the transformer blocks, applies normalization, and finally produces logits with the linear output layer.

The code above is already functional, as we will see later in this section after we prepare the input data. However, for now, note in the code above that we have used placeholders (`DummyLayerNorm` and `DummyTransformerBlock`) for the transformer block and layer normalization, which we will develop in later sections.

Next, we will prepare the input data and initialize a new GPT model to illustrate its usage. Building on the figures we have seen in chapter 2, where we coded the tokenizer, Figure 4.4 provides a high-level overview of how

data flows in and out of a GPT model.

Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our `DummyGPTClass` coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded input token dimension typically matches the output dimension. The output embeddings here represent the context vectors we discussed in chapter 3.



To implement the steps shown in Figure 4.4, we tokenize a batch consisting of two text inputs for the GPT model using the tiktoken tokenizer introduced in chapter 2:

```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Every effort moves you"
txt2 = "Every day holds a"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
```

The resulting token IDs for the two texts are as follows:

```
tensor([[ 6109,  3626,  6100,   345], #A
        [ 6109,  1110,  6622,   257]])
```

Next, we initialize a new 124 million parameter `DummyGPTModel` instance and feed it the tokenized `batch`:

```
torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
print("Output shape:", logits.shape)
print(logits)
```

The model outputs, which are commonly referred to as logits, are as follows:

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[-1.2034,  0.3201, -0.7130,  ..., -1.5548, -0.2390, -0.4
         [-0.1192,  0.4539, -0.4432,  ...,  0.2392,  1.3469,  1.2
         [ 0.5307,  1.6720, -0.4695,  ...,  1.1966,  0.0111,  0.5
         [ 0.0139,  1.6755, -0.3388,  ...,  1.1586, -0.0435, -1.0

        [[-1.0908,  0.1798, -0.9484,  ..., -1.6047,  0.2439, -0.4
         [-0.7860,  0.5581, -0.0610,  ...,  0.4835, -0.0077,  1.6
         [ 0.3567,  1.2698, -0.6398,  ..., -0.0162, -0.1296,  0.3
         [-0.2407, -0.7349, -0.5102,  ...,  2.0057, -0.3694,  0.1
       grad_fn=<UnsafeViewBackward0>)
```

The output tensor has two rows corresponding to the two text samples. Each text sample consists of 4 tokens; each token is a 50,257-dimensional vector, which matches the size of the tokenizer's vocabulary.

The embedding has 50,257 dimensions because each of these dimensions refers to a unique token in the vocabulary. At the end of this chapter, when we implement the postprocessing code, we will convert these 50,257-dimensional vectors back into token IDs, which we can then decode into words.

Now that we have taken a top-down look at the GPT architecture and its in- and outputs, we will code the individual placeholders in the upcoming sections, starting with the real layer normalization class that will replace the `DummyLayerNorm` in the previous code.

4.2 Normalizing activations with layer normalization

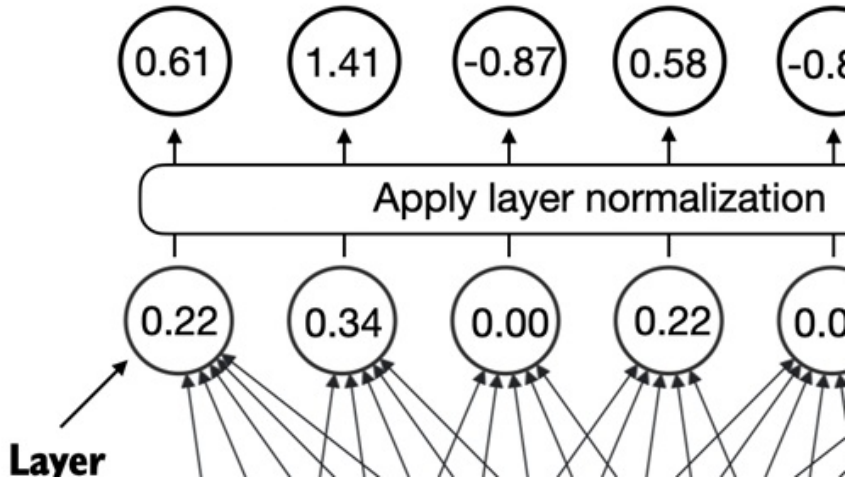
Training deep neural networks with many layers can sometimes prove challenging due to issues like vanishing or exploding gradients. These issues lead to unstable training dynamics and make it difficult for the network to effectively adjust its weights, which means the learning process struggles to find a set of parameters (weights) for the neural network that minimizes the loss function. In other words, the network has difficulty learning the underlying patterns in the data to a degree that would allow it to make accurate predictions or decisions. (If you are new to neural network training and the concepts of gradients, a brief introduction to these concepts can be found in *Section A.4, Automatic Differentiation Made Easy* in *Appendix A: Introduction to PyTorch*. However, a deep mathematical understanding of gradients is not required to follow the contents of this book.)

In this section, we will implement *layer normalization* to improve the stability and efficiency of neural network training.

The main idea behind layer normalization is to adjust the activations (outputs) of a neural network layer to have a mean of 0 and a variance of 1, also known as unit variance. This adjustment speeds up the convergence to effective weights and ensures consistent, reliable training. As we have seen in the previous section, based on the `DummyLayerNorm` placeholder, in GPT-2 and modern transformer architectures, layer normalization is typically applied before and after the multi-head attention module and before the final output layer.

Before we implement layer normalization in code, Figure 4.5 provides a visual overview of how layer normalization functions.

Figure 4.5 An illustration of layer normalization where the 5 layer outputs, also called activations, are normalized such that they have a zero mean and variance of 1.



We can recreate the example shown in Figure 4.5 via the following code, where we implement a neural network layer with 5 inputs and 6 outputs that we apply to two input examples:

```
torch.manual_seed(123)
batch_example = torch.randn(2, 5) #A
layer = nn.Sequential(nn.Linear(5, 6), nn.ReLU())
out = layer(batch_example)
print(out)
```

This prints the following tensor, where the first row lists the layer outputs for the first input and the second row lists the layer outputs for the second row:

```
tensor([[0.2260, 0.3470, 0.0000, 0.2216, 0.0000, 0.0000],
        [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]],
       grad_fn=<ReluBackward0>)
```

The neural network layer we have coded consists of a `Linear` layer followed by a non-linear activation function, `ReLU` (short for Rectified Linear Unit), which is a standard activation function in neural networks. If you are unfamiliar with `ReLU`, it simply thresholds negative inputs to 0, ensuring that a layer outputs only positive values, which explains why the resulting layer output does not contain any negative values. (Note that we will use another, more sophisticated activation function in GPT, which we will introduce in the next section).

Before we apply layer normalization to these outputs, let's examine the mean and variance:

```
mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The output is as follows:

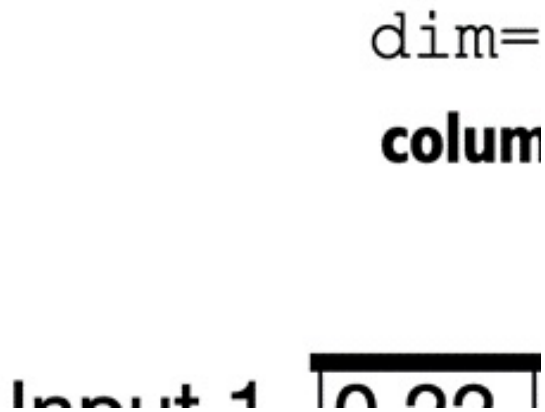
```
Mean:
  tensor([[0.1324],
          [0.2170]], grad_fn=<MeanBackward1>)
Variance:
  tensor([[0.0231],
          [0.0398]], grad_fn=<VarBackward0>)
```

The first row in the mean tensor above contains the mean value for the first input row, and the second output row contains the mean for the second input row.

Using `keepdim=True` in operations like mean or variance calculation ensures that the output tensor retains the same shape as the input tensor, even though the operation reduces the tensor along the dimension specified via `dim`. For instance, without `keepdim=True`, the returned mean tensor would be a 2-dimensional vector `[0.1324, 0.2170]` instead of a 2×1-dimensional matrix `[[0.1324], [0.2170]]`.

The `dim` parameter specifies the dimension along which the calculation of the statistic (here, mean or variance) should be performed in a tensor, as shown in Figure 4.6.

Figure 4.6 An illustration of the dim parameter when calculating the mean of a tensor. For instance, if we have a 2D tensor (matrix) with dimensions `[rows, columns]`, using `dim=0` will perform the operation across rows (vertically, as shown at the bottom), resulting in an output that aggregates the data for each column. Using `dim=1` or `dim=-1` will perform the operation across columns (horizontally, as shown at the top), resulting in an output aggregating the data for each row.



As Figure 4.6 explains, for a 2D tensor (like a matrix), using `dim=-1` for operations such as mean or variance calculation is the same as using `dim=1`. This is because -1 refers to the tensor's last dimension, which corresponds to the columns in a 2D tensor. Later, when adding layer normalization to the GPT model, which produces 3D tensors with shape `[batch_size, num_tokens, embedding_size]`, we can still use `dim=-1` for normalization across the last dimension, avoiding a change from `dim=1` to `dim=2`.

Next, let us apply layer normalization to the layer outputs we obtained earlier. The operation consists of subtracting the mean and dividing by the square root of the variance (also known as standard deviation):

```
out_norm = (out - mean) / torch.sqrt(var)
mean = out_norm.mean(dim=-1, keepdim=True)
var = out_norm.var(dim=-1, keepdim=True)
print("Normalized layer outputs:\n", out_norm)
print("Mean:\n", mean)
print("Variance:\n", var)
```

As we can see based on the results, the normalized layer outputs, which now also contain negative values, have zero mean and a variance of 1:

```
Normalized layer outputs:
 tensor([[ 0.6159,  1.4126, -0.8719,  0.5872, -0.8719, -0.8719],
        [-0.0189,  0.1121, -1.0876,  1.5173,  0.5647, -1.0876]],
       grad_fn=<DivBackward0>)
Mean:
 tensor([[2.9802e-08],
        [3.9736e-08]], grad_fn=<MeanBackward1>)
Variance:
 tensor([[1.],
```

```
       [1.]], grad_fn=<VarBackward0>)
```

Note that the value 2.9802e-08 in the output tensor is the scientific notation for 2.9802 × 10-8, which is 0.0000000298 in decimal form. This value is very close to 0, but it is not exactly 0 due to small numerical errors that can accumulate because of the finite precision with which computers represent numbers.

To improve readability, we can also turn off the scientific notation when printing tensor values by setting `sci_mode` to False:

```
torch.set_printoptions(sci_mode=False)
print("Mean:\n", mean)
print("Variance:\n", var)
Mean:
 tensor([[    0.0000],
        [    0.0000]], grad_fn=<MeanBackward1>)
Variance:
 tensor([[1.],
        [1.]], grad_fn=<VarBackward0>)
```

So far, in this section, we have coded and applied layer normalization in a step-by-step process. Let's now encapsulate this process in a PyTorch module that we can use in the GPT model later:

Listing 4.2 A layer normalization class

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

This specific implementation of layer Normalization operates on the last dimension of the input tensor x, which represents the embedding dimension (`emb_dim`). The variable `eps` is a small constant (epsilon) added to the

variance to prevent division by zero during normalization. The `scale` and `shift` are two trainable parameters (of the same dimension as the input) that the LLM automatically adjusts during training if it is determined that doing so would improve the model's performance on its training task. This allows the model to learn appropriate scaling and shifting that best suit the data it is processing.

Biased variance

In our variance calculation method, we have opted for an implementation detail by setting `unbiased=False`. For those curious about what this means, in the variance calculation, we divide by the number of inputs *n* in the variance formula. This approach does not apply Bessel's correction, which typically uses *n-1* instead of *n* in the denominator to adjust for bias in sample variance estimation. This decision results in a so-called biased estimate of the variance. For large-scale language models (LLMs), where the embedding dimension n is significantly large, the difference between using n and n-1 is practically negligible. We chose this approach to ensure compatibility with the GPT-2 model's normalization layers and because it reflects TensorFlow's default behavior, which was used to implement the original GPT-2 model. Using a similar setting ensures our method is compatible with the pretrained weights we will load in chapter 6.

Let's now try the `LayerNorm` module in practice and apply it to the batch input:

```
ln = LayerNorm(emb_dim=5)
out_ln = ln(batch_example)
mean = out_ln.mean(dim=-1, keepdim=True)
var = out_ln.var(dim=-1, unbiased=False, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

As we can see based on the results, the layer normalization code works as expected and normalizes the values of each of the two inputs such that they have a mean of 0 and a variance of 1:

```
Mean:
 tensor([[    -0.0000],
        [     0.0000]], grad_fn=<MeanBackward1>)
```

```
Variance:
 tensor([[1.0000],
         [1.0000]], grad_fn=<VarBackward0>)
```

In this section, we covered one of the building blocks we will need to implement the GPT architecture, as shown in the mental model in Figure 4.7.

Figure 4.7 A mental model listing the different building blocks we implement in this chapter to assemble the GPT architecture.

In the next section, we will look at the GELU activation function, which is one of the activation functions used in LLMs, instead of the traditional ReLU function we used in this section.

Layer normalization versus batch normalization

If you are familiar with batch normalization, a common and traditional normalization method for neural networks, you may wonder how it compares to layer normalization. Unlike batch normalization, which normalizes across the batch dimension, layer normalization normalizes across the feature dimension. LLMs often require significant computational resources, and the available hardware or the specific use case can dictate the batch size during training or inference. Since layer normalization normalizes each input independently of the batch size, it offers more flexibility and stability in these scenarios. This is particularly beneficial for distributed training or when deploying models in environments where resources are constrained.

4.3 Implementing a feed forward network with

GELU activations

In this section, we implement a small neural network submodule that is used as part of the transformer block in LLMs. We begin with implementing the *GELU* activation function, which plays a crucial role in this neural network submodule. (For additional information on implementing neural networks in PyTorch, please see section A.5 Implementing multilayer neural networks in Appendix A.)

Historically, the ReLU activation function has been commonly used in deep learning due to its simplicity and effectiveness across various neural network architectures. However, in LLMs, several other activation functions are employed beyond the traditional ReLU. Two notable examples are GELU (*Gaussian Error Linear Unit*) and SwiGLU (*Sigmoid-Weighted Linear Unit*).

GELU and SwiGLU are more complex and smooth activation functions incorporating Gaussian and sigmoid-gated linear units, respectively. They offer improved performance for deep learning models, unlike the simpler ReLU.

The GELU activation function can be implemented in several ways; the exact version is defined as GELU(x)=x Φ(x), where Φ(x) is the cumulative distribution function of the standard Gaussian distribution. In practice, however, it's common to implement a computationally cheaper approximation (the original GPT-2 model was also trained with this approximation):

GELU(x) ≈ 0.5 · x · (1 + tanh[√((2/π)) · (x + 0.044715 · x^3])

In code, we can implement this function as PyTorch module as follows:

Listing 4.3 An implementation of the GELU activation function

```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
```

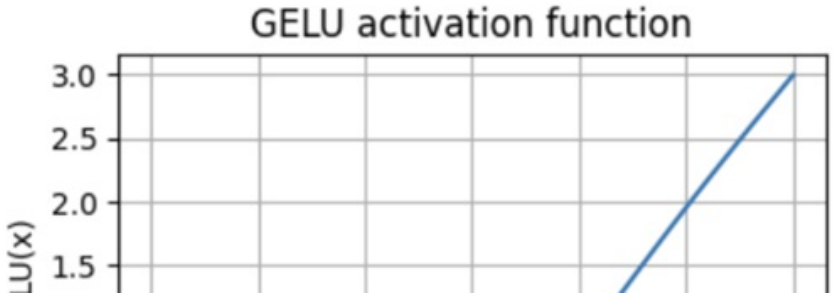
```
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))
```

Next, to get an idea of what this GELU function looks like and how it compares to the ReLU function, let's plot these functions side by side:

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100) #A
y_gelu, y_relu = gelu(x), relu(x)
plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "Re
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

As we can see in the resulting plot in Figure 4.8, ReLU is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero. GELU is a smooth, non-linear function that approximates ReLU but with a non-zero gradient for negative values.

Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.



The smoothness of GELU, as shown in Figure 4.8, can lead to better optimization properties during training, as it allows for more nuanced adjustments to the model's parameters. In contrast, ReLU has a sharp corner

at zero, which can sometimes make optimization harder, especially in networks that are very deep or have complex architectures. Moreover, unlike RELU, which outputs zero for any negative input, GELU allows for a small, non-zero output for negative values. This characteristic means that during the training process, neurons that receive negative input can still contribute to the learning process, albeit to a lesser extent than positive inputs.

Next, let's use the GELU function to implement the small neural network module, `FeedForward`, that we will be using in the LLM's transformer block later:

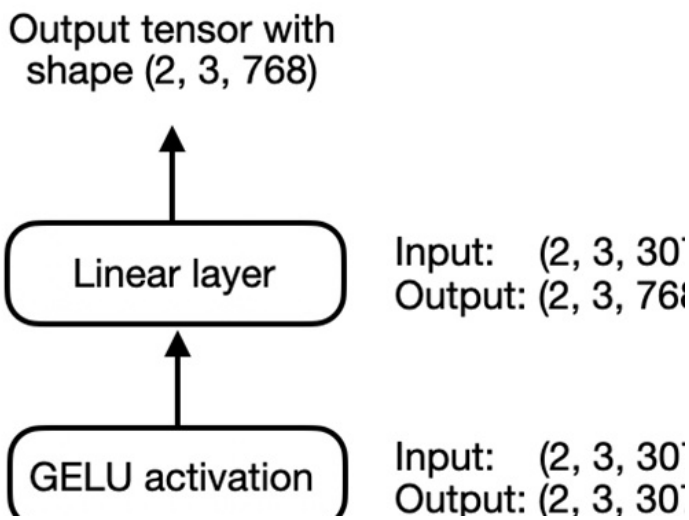
Listing 4.4 A feed forward neural network module

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```

As we can see in the preceding code, the `FeedForward` module is a small neural network consisting of two `Linear` layers and a `GELU` activation function. In the 124 million parameter GPT model, it receives the input batches with tokens that have an embedding size of 768 each via the `GPT_CONFIG_124M` dictionary where `GPT_CONFIG_124M["emb_dim"] = 768`.

Figure 4.9 shows how the embedding size is manipulated inside this small feed forward neural network when we pass it some inputs.

Figure 4.9 provides a visual overview of the connections between the layers of the feed forward neural network. It is important to note that this neural network can accommodate variable batch sizes and numbers of tokens in the input. However, the embedding size for each token is determined and fixed when initializing the weights.



Following the example in Figure 4.9, let's initialize a new `FeedForward` module with a token embedding size of 768 and feed it a batch input with 2 samples and 3 tokens each:

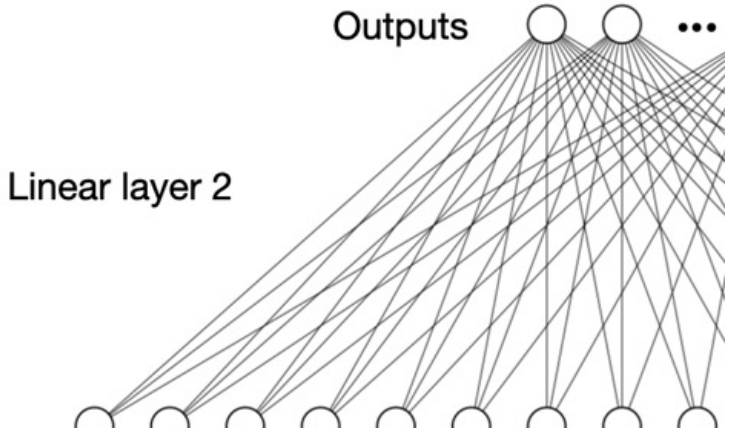
```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768) #A
out = ffn(x)
print(out.shape)
```

As we can see, the shape of the output tensor is the same as that of the input tensor:

```
torch.Size([2, 3, 768])
```

The `FeedForward` module we implemented in this section plays a crucial role in enhancing the model's ability to learn from and generalize the data. Although the input and output dimensions of this module are the same, it internally expands the embedding dimension into a higher-dimensional space through the first linear layer as illustrated in Figure 4.10. This expansion is followed by a non-linear GELU activation, and then a contraction back to the original dimension with the second linear transformation. Such a design allows for the exploration of a richer representation space.

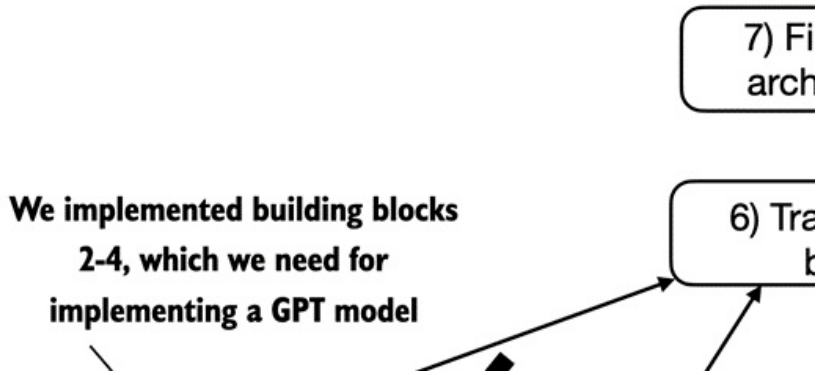
Figure 4.10 An illustration of the expansion and contraction of the layer outputs in the feed forward neural network. First, the inputs expand by a factor of 4 from 768 to 3072 values. Then, the second layer compresses the 3072 values back into a 768-dimensional representation.



Moreover, the uniformity in input and output dimensions simplifies the architecture by enabling the stacking of multiple layers, as we will do later, without the need to adjust dimensions between them, thus making the model more scalable.

As illustrated in Figure 4.11, we have now implemented most of the LLM's building blocks.

Figure 4.11 A mental model showing the topics we cover in this chapter, with the black checkmarks indicating those that we have already covered.

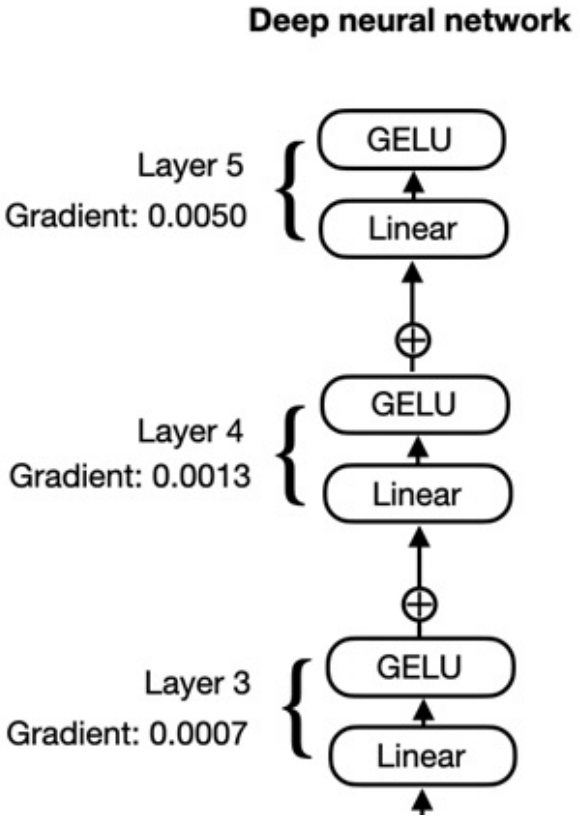


In the next section, we will go over the concept of shortcut connections that we insert between different layers of a neural network, which are important for improving the training performance in deep neural network architectures.

4.4 Adding shortcut connections

Next, let's discuss the concept behind *shortcut connections*, also known as skip or residual connections. Originally, shortcut connections were proposed for deep networks in computer vision (specifically, in residual networks) to mitigate the challenge of vanishing gradients. The vanishing gradient problem refers to the issue where gradients (which guide weight updates during training) become progressively smaller as they propagate backward through the layers, making it difficult to effectively train earlier layers, as illustrated in Figure 4.12.

Figure 4.12 A comparison between a deep neural network consisting of 5 layers without (on the left) and with shortcut connections (on the right). Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. The gradient illustrated in Figure 1.1 denotes the mean absolute gradient at each layer, which we will compute in the code example that follows.



As illustrated in Figure 4.12, a shortcut connection creates an alternative, shorter path for the gradient to flow through the network by skipping one or more layers, which is achieved by adding the output of one layer to the output of a later layer. This is why these connections are also known as skip

connections. They play a crucial role in preserving the flow of gradients during the backward pass in training.

In the code example below, we implement the neural network shown in Figure 4.12 to see how we can add shortcut connections in the `forward` method:

Listing 4.5 A neural network to illustrate shortcut connections

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
            # Implement 5 layers
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2
            nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3
            nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4
            nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5
        ])

    def forward(self, x):
        for layer in self.layers:
            # Compute the output of the current layer
            layer_output = layer(x)
            # Check if shortcut can be applied
            if self.use_shortcut and x.shape == layer_output.shap
                x = x + layer_output
            else:
                x = layer_output
        return x
```

The code implements a deep neural network with 5 layers, each consisting of a `Linear` layer and a `GELU` activation function. In the forward pass, we iteratively pass the input through the layers and optionally add the shortcut connections depicted in Figure 4.12 if the `self.use_shortcut` attribute is set to `True`.

Let's use this code to first initialize a neural network without shortcut connections. Here, each layer will be initialized such that it accepts an example with 3 input values and returns 3 output values. The last layer returns a single output value:

```
layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123) # specify random seed for the initial weig
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)
```

Next, we implement a function that computes the gradients in the the model's backward pass:

```
def print_gradients(model, x):
    # Forward pass
    output = model(x)
    target = torch.tensor([[0.]])

    # Calculate loss based on how close the target
    # and output are
    loss = nn.MSELoss()
    loss = loss(output, target)

    # Backward pass to calculate the gradients
    loss.backward()

    for name, param in model.named_parameters():
        if 'weight' in name:
            # Print the mean absolute gradient of the weights
            print(f"{name} has gradient mean of {param.grad.abs()
```

In the preceding code, we specify a loss function that computes how close the model output and a user-specified target (here, for simplicity, the value 0) are. Then, when calling `loss.backward()`, PyTorch computes the loss gradient for each layer in the model. We can iterate through the weight parameters via `model.named_parameters()`. Suppose we have a 3×3 weight parameter matrix for a given layer. In that case, this layer will have 3×3 gradient values, and we print the mean absolute gradient of these 3×3 gradient values to obtain a single gradient value per layer to compare the gradients between layers more easily.

In short, the `.backward()` method is a convenient method in PyTorch that computes loss gradients, which are required during model training, without implementing the math for the gradient calculation ourselves, thereby making working with deep neural networks much more accessible. If you are unfamiliar with the concept of gradients and neural network training, I

recommend reading sections *A.4, Automatic differentiation made easy* and *A.7 A typical training loop* in *appendix A*.

Let's now use the `print_gradients` function and apply it to the model without skip connections:

```
print_gradients(model_without_shortcut, sample_input)
```

The output is as follows:

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.00012011116101583466
layers.2.0.weight has gradient mean of 0.0007152041653171182
layers.3.0.weight has gradient mean of 0.001398873864673078
layers.4.0.weight has gradient mean of 0.005049646366387606
```

As we can see based on the output of the `print_gradients` function, the gradients become smaller as we progress from the last layer (`layers.4`) to the first layer (`layers.0`), which is a phenomenon called the vanishing gradient problem.

Let's now instantiate a model with skip connections and see how it compares:

```
torch.manual_seed(123)
model_with_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=True
)
print_gradients(model_with_shortcut, sample_input)
```

The output is as follows:

```
layers.0.0.weight has gradient mean of 0.22169792652130127
layers.1.0.weight has gradient mean of 0.20694105327129364
layers.2.0.weight has gradient mean of 0.32896995544433594
layers.3.0.weight has gradient mean of 0.2665732502937317
layers.4.0.weight has gradient mean of 1.3258541822433472
```

As we can see, based on the output, the last layer (`layers.4`) still has a larger gradient than the other layers. However, the gradient value stabilizes as we progress towards the first layer (`layers.0`) and doesn't shrink to a vanishingly small value.

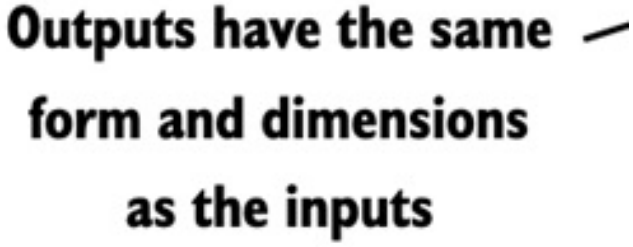
In conclusion, shortcut connections are important for overcoming the limitations posed by the vanishing gradient problem in deep neural networks. Shortcut connections are a core building block of very large models such as LLMs, and they will help facilitate more effective training by ensuring consistent gradient flow across layers when we train the GPT model in the next chapter.

After introducing shortcut connections, we will now connect all of the previously covered concepts (layer normalization, GELU activations, feed forward module, and shortcut connections) in a transformer block in the next section, which is the final building block we need to code the GPT architecture.

4.5 Connecting attention and linear layers in a transformer block

In this section, we are implementing the *transformer block*, a fundamental building block of GPT and other LLM architectures. This block, which is repeated a dozen times in the 124 million parameter GPT-2 architecture, combines several concepts we have previously covered: multi-head attention, layer normalization, dropout, feed forward layers, and GELU activations, as illustrated in Figure 4.13. In the next section, we will then connect this transformer block to the remaining parts of the GPT architecture.

Figure 4.13 An illustration of a transformer block. The bottom of the diagram shows input tokens that have been embedded into 768-dimensional vectors. Each row corresponds to one token's vector representation. The outputs of the transformer block are vectors of the same dimension as the input, which can then be fed into subsequent layers in an LLM.



As shown in Figure 4.13, the transformer block combines several components, including the masked multi-head attention module from chapter 3 and the `FeedForward` module we implemented in Section 4.3.

When a transformer block processes an input sequence, each element in the sequence (for example, a word or subword token) is represented by a fixed-size vector (in the case of Figure 4.13, 768 dimensions). The operations within the transformer block, including multi-head attention and feed forward layers, are designed to transform these vectors in a way that preserves their dimensionality.

The idea is that the self-attention mechanism in the multi-head attention block identifies and analyzes relationships between elements in the input sequence. In contrast, the feed forward network modifies the data individually at each position. This combination not only enables a more nuanced understanding and processing of the input but also enhances the model's overall capacity for handling complex data patterns.

In code, we can create the `TransformerBlock` as follows:

Listing 4.6 The transformer block component of GPT

```
from previous_chapters import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            block_size=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_resid = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        #A
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_resid(x)
        x = x + shortcut  # Add the original input back

        shortcut = x #B
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_resid(x)
        x = x + shortcut  #C
        return x
```

The given code defines a `TransformerBlock` class in PyTorch that includes a multi-head attention mechanism (`MultiHeadAttention`) and a feed forward network (`FeedForward`), both configured based on a provided configuration dictionary (`cfg`), such as `GPT_CONFIG_124M`.

Layer normalization (`LayerNorm`) is applied before each of these two components, and dropout is applied after them to regularize the model and prevent overfitting. This is also known as *Pre-LayerNorm*. Older architectures, such as the original transformer model, applied layer normalization after the self-attention and feed-forward networks instead, known as *Post-LayerNorm*, which often leads to worse training dynamics.

The class also implements the forward pass, where each component is followed by a shortcut connection that adds the input of the block to its output. This critical feature helps gradients flow through the network during training and improves the learning of deep models as explained in section 4.4.

Using the `GPT_CONFIG_124M` dictionary we defined earlier, let's instantiate a transformer block and feed it some sample data:

```
torch.manual_seed(123)
x = torch.rand(2, 4, 768)  #A
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print("Input shape:", x.shape)
print("Output shape:", output.shape)
```

The output is as follows:

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])
```

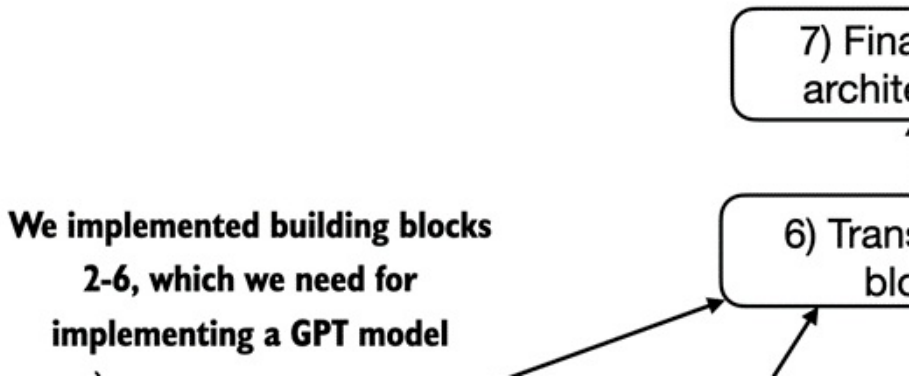
As we can see from the code output, the transformer block maintains the input dimensions in its output, indicating that the transformer architecture processes sequences of data without altering their shape throughout the network.

The preservation of shape throughout the transformer block architecture is not incidental but a crucial aspect of its design. This design enables its effective application across a wide range of sequence-to-sequence tasks, where each output vector directly corresponds to an input vector, maintaining a one-to-one relationship. However, the output is a context vector that encapsulates information from the entire input sequence, as we learned in chapter 3. This means that while the physical dimensions of the sequence (length and feature size) remain unchanged as it passes through the transformer block, the content of each output vector is re-encoded to integrate contextual information from across the entire input sequence.

With the transformer block implemented in this section, we now have all the building blocks, as shown in Figure 4.14, needed to implement the GPT

architecture in the next section.

Figure 4.14 A mental model of the different concepts we have implemented in this chapter so far.



As illustrated in Figure 4.14, the transformer block combines layer normalization, the feed forward network, including GELU activations, and shortcut connections, which we already covered earlier in this chapter. As we will see in the upcoming chapter, this transformer block will make up the main component of the GPT architecture we will implement

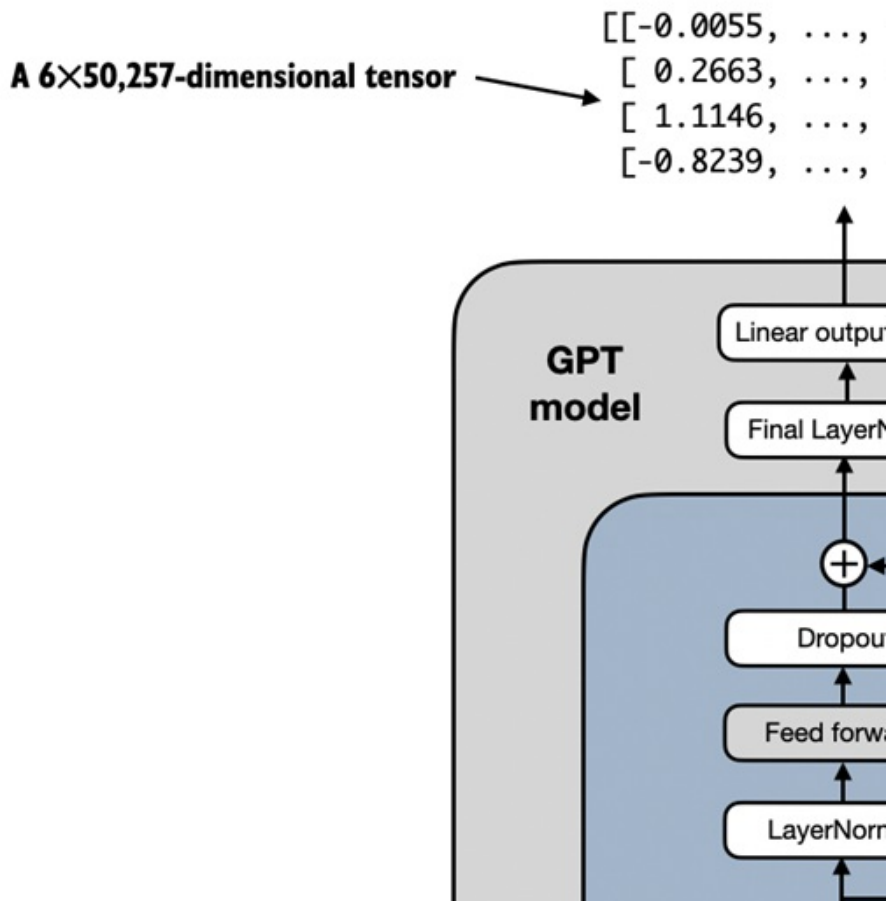
4.6 Coding the GPT model

We started this chapter with a big-picture overview of a GPT architecture that we called `DummyGPTModel`. In this `DummyGPTModel` code implementation, we showed the input and outputs to the GPT model, but its building blocks remained a black box using a `DummyTransformerBlock` and `DummyLayerNorm` class as placeholders.

In this section, we are now replacing the `DummyTransformerBlock` and `DummyLayerNorm` placeholders with the real `TransformerBlock` and `LayerNorm` classes we coded later in this chapter to assemble a fully working version of the original 124 million parameter version of GPT-2. In chapter 5, we will pretrain a GPT-2 model, and in chapter 6, we will load in the pretrained weights from OpenAI.

Before we assemble the GPT-2 model in code, let's look at its overall structure in Figure 4.15, which combines all the concepts we covered so far in this chapter.

Figure 4.15 An overview of the GPT model architecture. This figure illustrates the flow of data through the GPT model. Starting from the bottom, tokenized text is first converted into token embeddings, which are then augmented with positional embeddings. This combined information forms a tensor that is passed through a series of transformer blocks shown in the center (each containing multi-head attention and feed forward neural network layers with dropout and layer normalization), which are stacked on top of each other and repeated 12 times.



As shown in Figure 4.15, the transformer block we coded in Section 4.5 is repeated many times throughout a GPT model architecture. In the case of the 124 million parameter GPT-2 model, it's repeated 12 times, which we specify via the `"n_layers"` entry in the `GPT_CONFIG_124M` dictionary. In the case of the largest GPT-2 model with 1,542 million parameters, this transformer block is repeated 36 times.

As shown in Figure 4.15, the output from the final transformer block then goes through a final layer normalization step before reaching the linear output layer. This layer maps the transformer's output to a high-dimensional space

(in this case, 50,257 dimensions, corresponding to the model's vocabulary size) to predict the next token in the sequence.

Let's now implement the architecture we see in Figure 4.15 in code:

Listing 4.7 The GPT model architecture implementation

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_d
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["e
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        #A
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Thanks to the `TransformerBlock` class we implemented in Section 4.5, the `GPTModel` class is relatively small and compact.

The `__init__` constructor of this `GPTModel` class initializes the token and positional embedding layers using the configurations passed in via a Python dictionary, `cfg`. These embedding layers are responsible for converting input token indices into dense vectors and adding positional information, as discussed in chapter 2.

Next, the `__init__` method creates a sequential stack of `TransformerBlock` modules equal to the number of layers specified in `cfg`. Following the transformer blocks, a `LayerNorm` layer is applied, standardizing the outputs from the transformer blocks to stabilize the learning process. Finally, a linear output head without bias is defined, which projects the transformer's output into the vocabulary space of the tokenizer to generate logits for each token in the vocabulary.

The forward method takes a batch of input token indices, computes their embeddings, applies the positional embeddings, passes the sequence through the transformer blocks, normalizes the final output, and then computes the logits, representing the next token's unnormalized probabilities. We will convert these logits into tokens and text outputs in the next section.

Let's now initialize the 124 million parameter GPT model using the `GPT_CONFIG_124M` dictionary we pass into the cfg parameter and feed it with the batch text input we created at the beginning of this chapter:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

The preceding code prints the contents of the input batch followed by the output tensor:

```
Input batch:
 tensor([[ 6109,  3626,  6100,   345], # token IDs of text 1
         [ 6109,  1110,  6622,   257]]) # token IDs of text 2

Output shape: torch.Size([2, 4, 50257])
tensor([[[ 0.3613,  0.4222, -0.0711,  ...,  0.3483,  0.4661, -0.2
         [-0.1792, -0.5660, -0.9485,  ...,  0.0477,  0.5181, -0.3
         [ 0.7120,  0.0332,  0.1085,  ...,  0.1018, -0.4327, -0.2
         [-1.0076,  0.3418, -0.1190,  ...,  0.7195,  0.4023,  0.0

        [[-0.2564,  0.0900,  0.0335,  ...,  0.2659,  0.4454, -0.6
         [ 0.1230,  0.3653, -0.2074,  ...,  0.7705,  0.2710,  0.2
         [ 1.0558,  1.0318, -0.2800,  ...,  0.6936,  0.3205, -0.3
```

```
          [-0.1565,  0.3926,  0.3288,  ...,  1.2630, -0.1858,  0.0
        grad_fn=<UnsafeViewBackward0>)
```

As we can see, the output tensor has the shape `[2, 4, 50257]`, since we passed in 2 input texts with 4 tokens each. The last dimension, 50,257, corresponds to the vocabulary size of the tokenizer. In the next section, we will see how to convert each of these 50,257-dimensional output vectors back into tokens.

Before we move on to the next section and code the function that converts the model outputs into text, let's spend a bit more time with the model architecture itself and analyze its size.

Using the `numel()` method, short for "number of elements," we can collect the total number of parameters in the model's parameter tensors:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total number of parameters: {total_params:,}")
```

The result is as follows:

```
Total number of parameters: 163,009,536
```

Now, a curious reader might notice a discrepancy. Earlier, we spoke of initializing a 124 million parameter GPT model, so why is the actual number of parameters 163 million, as shown in the preceding code output?

The reason is a concept called weight tying that is used in the original GPT-2 architecture, which means that the original GPT-2 architecture is reusing the weights from the token embedding layer inits output layer. To understand what this means, let's take a look at the shapes of the token embedding layer and linear output layer that we initialized on the `model` via the `GPTModel` earlier:

```
print("Token embedding layer shape:", model.tok_emb.weight.shape)
print("Output layer shape:", model.out_head.weight.shape)
```

As we can see based on the print outputs, the weight tensors for both these layers have the same shape:

```
Token embedding layer shape: torch.Size([50257, 768])
Output layer shape: torch.Size([50257, 768])
```

The token embedding and output layers are very large due to the number of rows for the 50,257 in the tokenizer's vocabulary. Let's remove the output layer parameter count from the total GPT-2 model count according to the weight tying:

```
total_params_gpt2 =  total_params - sum(p.numel() for p in model.
print(f"Number of trainable parameters considering weight tying:
```

The output is as follows:

```
Number of trainable parameters considering weight tying: 124,412,
```

As we can see, the model is now only 124 million parameters large, matching the original size of the GPT-2 model.

Weight tying reduces the overall memory footprint and computational complexity of the model. However, in my experience, using separate token embedding and output layers results in better training and model performance; hence, we are using separate layers in our `GPTModel` implementation. The same is true for modern LLMs. However, we will revisit and implement the weight tying concept later in chapter 6 when we load the pretrained weights from OpenAI.

Exercise 4.1 Number of parameters in feed forward and attention modules

Calculate and compare the number of parameters that are contained in the feed forward module and those that are contained in the multi-head attention module.

Lastly, let us compute the memory requirements of the 163 million parameters in our `GPTModel` object:

```
total_size_bytes = total_params * 4  #A
total_size_mb = total_size_bytes / (1024 * 1024)  #B
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

The result is as follows:

```
Total size of the model: 621.83 MB
```

In conclusion, by calculating the memory requirements for the 163 million parameters in our `GPTModel` object and assuming each parameter is a 32-bit float taking up 4 bytes, we find that the total size of the model amounts to 621.83 MB, illustrating the relatively large storage capacity required to accommodate even relatively small LLMs.

In this section, we implemented the GPTModel architecture and saw that it outputs numeric tensors of shape `[batch_size, num_tokens, vocab_size]`. In the next section, we will write the code to convert these output tensors into text.

Exercise 4.2 Initializing larger GPT models

In this chapter, we initialized a 124 million parameter GPT model, which is known as "GPT-2 small." Without making any code modifications besides updating the configuration file, use the GPTModel class to implement GPT-2 medium (using 1024-dimensional embeddings, 24 transformer blocks, 16 multi-head attention heads), GPT-2 large (1280-dimensional embeddings, 36 transformer blocks, 20 multi-head attention heads), and GPT-2 XL (1600-dimensional embeddings, 48 transformer blocks, 25 multi-head attention heads). As a bonus, calculate the total number of parameters in each GPT model.

4.7 Generating text

In this final section of this chapter, we will implement the code that converts the tensor outputs of the GPT model back into text. Before we get started, let's briefly review how a generative model like an LLM generates text one word (or token) at a time, as shown in Figure 4.16.

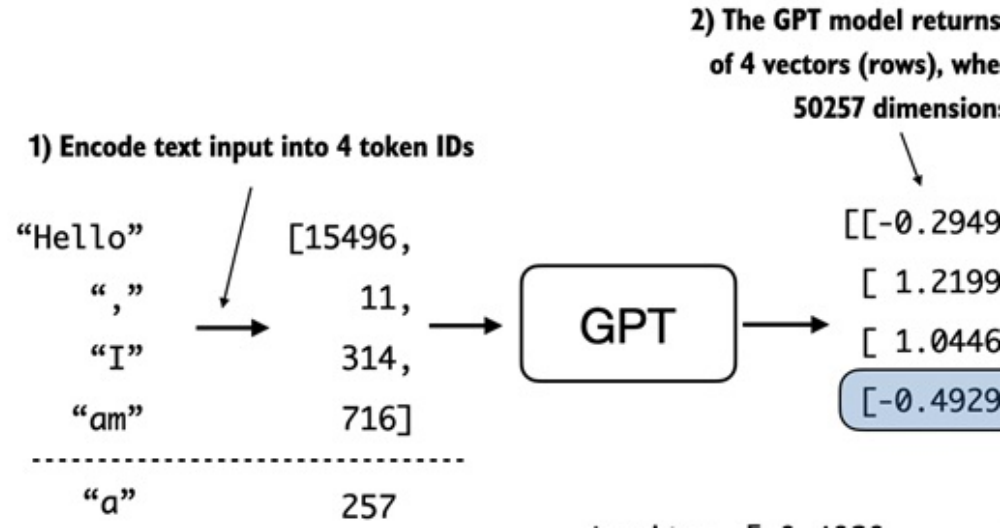
Figure 4.16 This diagram illustrates the step-by-step process by which an LLM generates text, one token at a time. Starting with an initial input context ("Hello, I am"), the model predicts a subsequent token during each iteration, appending it to the input context for the next round of prediction. As shown, the first iteration adds "a", the second "model", and the third "ready", progressively building the sentence.

Figure 4.16 illustrates the step-by-step process by which a GPT model generates text given an input context, such as "Hello, I am," on a big-picture level. With each iteration, the input context grows, allowing the model to generate coherent and contextually appropriate text. By the 6th iteration, the model has constructed a complete sentence: "Hello, I am a model ready to help."

In the previous section, we saw that our current `GPTModel` implementation outputs tensors with shape `[batch_size, num_token, vocab_size]`. Now, the question is, how does a GPT model go from these output tensors to the generated text shown in Figure 4.16?

The process by which a GPT model goes from output tensors to generated text involves several steps, as illustrated in Figure 4.17. These steps include decoding the output tensors, selecting tokens based on a probability distribution, and converting these tokens into human-readable text.

Figure 4.17 details the mechanics of text generation in a GPT model by showing a single iteration in the token generation process. The process begins by encoding the input text into token IDs, which are then fed into the GPT model. The outputs of the model are then converted back into text and appended to the original input text.



The next-token generation process detailed in Figure 4.17 illustrates a single step where the GPT model generates the next token given its input.

In each step, the model outputs a matrix with vectors representing potential next tokens. The vector corresponding to the next token is extracted and converted into a probability distribution via the softmax function. Within the vector containing the resulting probability scores, the index of the highest value is located, which translates to the token ID. This token ID is then decoded back into text, producing the next token in the sequence. Finally, this token is appended to the previous inputs, forming a new input sequence for the subsequent iteration. This step-by-step process enables the model to generate text sequentially, building coherent phrases and sentences from the initial input context.

In practice, we repeat this process over many iterations, such as shown in Figure 4.16 earlier, until we reach a user-specified number of generated tokens.

In code, we can implement the token-generation process as follows:

Listing 4.8 A function for the GPT model to generate text

```
def generate_text_simple(model, idx, max_new_tokens, context_size
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:] #B
```

```
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :] #C
        probas = torch.softmax(logits, dim=-1)  #D
        idx_next = torch.argmax(probas, dim=-1, keepdim=True) #E
        idx = torch.cat((idx, idx_next), dim=1)  #F

    return idx
```

The code snippet provided demonstrates a simple implementation of a generative loop for a language model using PyTorch. It iterates for a specified number of new tokens to be generated, crops the current context to fit the model's maximum context size, computes predictions and then selects the next token based on the highest probability prediction.

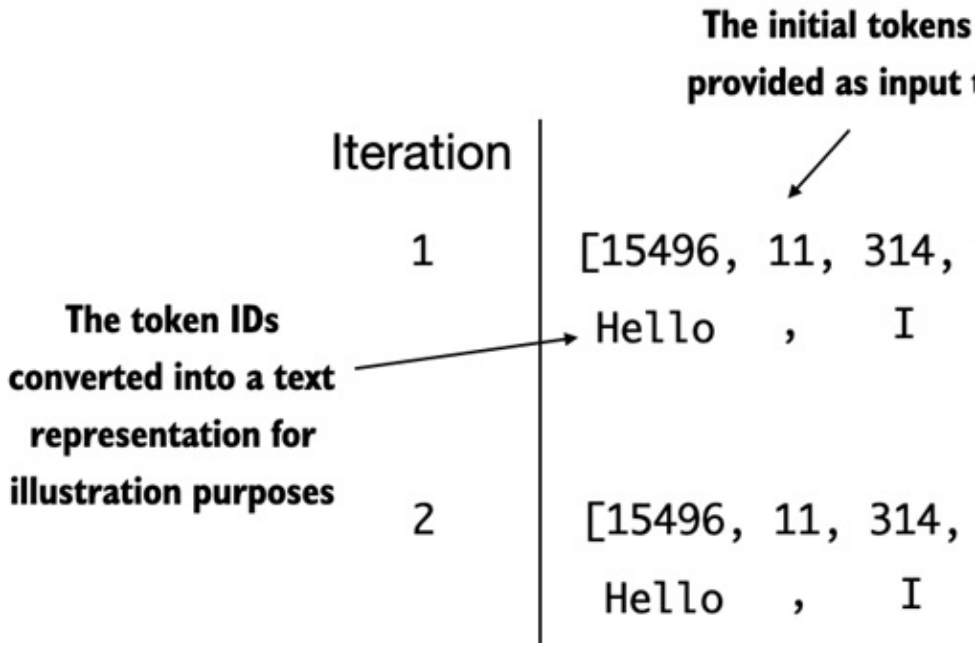
In the preceeding code, the `generate_text_simple` function, we use a softmax function to convert the logits into a probability distribution from which we identify the position with the highest value via `torch.argmax`. The softmax function is monotonic, meaning it preserves the order of its inputs when transformed into outputs. So, in practice, the softmax step is redundant since the position with the highest score in the softmax output tensor is the same position in the logit tensor. In other words, we could apply the `torch.argmax` function to the logits tensor directly and get identical results. However, we coded the conversion to illustrate the full process of transforming logits to probabilities, which can add additional intuition, such as that the model generates the most likely next token, which is known as *greedy decoding*.

In the next chapter, when we will implement the GPT training code, we will also introduce additional sampling techniques where we modify the softmax outputs such that the model doesn't always select the most likely token, which introduces variability and creativity in the generated text.

This process of generating one token ID at a time and appending it to the context using the `generate_text_simple` function is further illustrated in Figure 4.18. (The token ID generation process for each iteration is detailed in Figure 4.17.

Figure 4.18 An illustration showing six iterations of a token prediction cycle, where the model

takes a sequence of initial token IDs as input, predicts the next token, and appends this token to the input sequence for the next iteration. (The token IDs are also translated into their corresponding text for better understanding.)



As shown in Figure 4.18, we generate the token IDs in an iterative fashion. For instance, in iteration 1, the model is provided with the tokens corresponding to "Hello , I am", predicts the next token (with ID 257, which is "a"), and appends it to the input. This process is repeated until the model produces the complete sentence "Hello, I am a model ready to help." after six iterations.

Let's now try out the `generate_text_simple` function with the `"Hello, I am"` context as model input, as shown in Figure 4.18, in practice.

First, we encode the input context into token IDs:

```
start_context = "Hello, I am"
encoded = tokenizer.encode(start_context)
print("encoded:", encoded)
encoded_tensor = torch.tensor(encoded).unsqueeze(0) #A
print("encoded_tensor.shape:", encoded_tensor.shape)
```

The encoded IDs are as follows:

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

Next, we put the model into `.eval()` mode, which disables random components like dropout, which are only used during training, and use the `generate_text_simple` function on the encoded input tensor:

```
model.eval() #A
out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output:", out)
print("Output length:", len(out[0]))
```

The resulting output token IDs are as follows:

```
Output: tensor([[15496,    11,   314,   716, 27018, 24086, 47843,
Output length: 10
```

Using the `.decode` method of the tokenizer, we can convert the IDs back into text:

```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

The model output in text format is as follows:

```
Hello, I am Featureiman Byeswickattribute argue logger Normandy C
```

As we can see, based on the preceding output, the model generated gibberish, which is not at all like the coherent text shown in Figure 4.18. What happened? The reason why the model is unable to produce coherent text is that we haven't trained it yet. So far, we just implemented the GPT architecture and initialized a GPT model instance with initial random weights.

Model training is a large topic in itself, and we will tackle it in the next chapter.

Exercise 4.3 Using separate dropout parameters

At the beginning of this chapter, we defined a global `"drop_rate"` setting in the `GPT_CONFIG_124M` dictionary to set the dropout rate in various places throughout the GPTModel architecture. Change the code to specify a separate dropout value for the various dropout layers throughout the model architecture. (Hint: there are three distinct places where we used dropout layers: the embedding layer, shortcut layer, and multi-head attention module.)

4.8 Summary

- Layer normalization stabilizes training by ensuring that each layer's outputs have a consistent mean and variance.
- Shortcut connections are connections that skip one or more layers by feeding the output of one layer directly to a deeper layer, which helps mitigate the vanishing gradient problem when training deep neural networks, such as LLMs.
- Transformer blocks are a core structural component of GPT models, combining masked multi-head attention modules with fully connected feed-forward networks that use the GELU activation function.
- GPT models are LLMs with many repeated transformer blocks that have millions to billions of parameters.
- GPT models come in various sizes, for example, 124, 345, 762, and 1542 million parameters, which we can implement with the same `GPTModel` Python class.
- The text generation capability of a GPT-like LLM involves decoding output tensors into human-readable text by sequentially predicting one token at a time based on a given input context.
- Without training, a GPT model generates incoherent text, which underscores the importance of model training for coherent text generation, which is the topic of subsequent chapters.

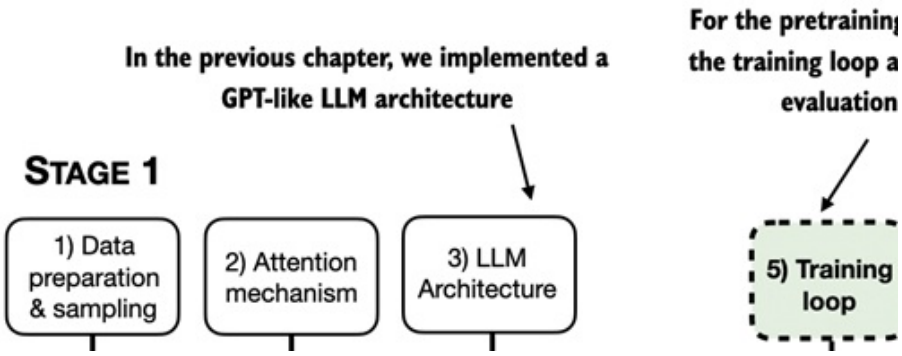
5 Pretraining on Unlabeled Data

This chapter covers

- Computing the training and validation set losses to assess the quality of LLM-generated text during training
- Implementing a training function and pretraining the LLM
- Saving and loading model weights to continue training an LLM
- Loading pretrained weights from OpenAI

In the previous chapters, we implemented the data sampling, attention mechanism and coded the LLM architecture. The core focus of this chapter is to implement a training function and pretrain the LLM, as illustrated in Figure 5.1.

Figure 5.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset and finetuning it on a labeled dataset. This chapter focuses on pretraining the LLM, which includes implementing the training code, evaluating the performance, and saving and loading model weights.



As illustrated in Figure 5.1, we will also learn about basic model evaluation techniques to measure the quality of the generated text, which is a requirement for optimizing the LLM during the training process. Moreover, we will discuss how to load pretrained weights, giving our LLM a solid starting point for finetuning in the upcoming chapters.

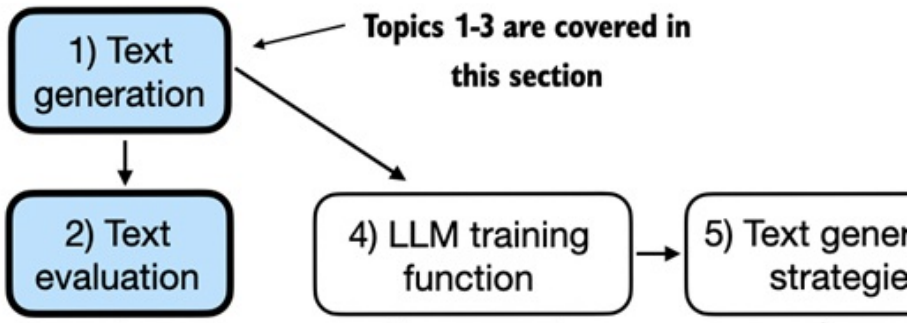
Weight parameters

In the context of LLMs and other deep learning models, *weights* refer to the trainable parameters that the learning process adjusts. These weights are also known as *weight parameters* or simply *parameters*. In frameworks like PyTorch, these weights are stored in linear layers, for example, which we used to implement the multi-head attention module in chapter 3 and the `GPTModel` in chapter 4. After initializing a layer (`new_layer = torch.nn.Linear(...)`), we can access its weights through the `.weight` attribute, `new_layer.weight`. Additionally, for convenience, PyTorch allows direct access to all a model's trainable parameters, including weights and biases, through the method `model.parameters()`, which we will use later when implementing the model training.

5.1 Evaluating generative text models

We begin this chapter by setting up the LLM for text generation based on code from the previous chapter and discuss basic ways to evaluate the quality of the generated text in this section. The content we cover in this section and the remainder of this chapter is outlined in Figure 5.2.

Figure 5.2 An overview of the topics covered in this chapter. We begin by recapping the text generation from the previous chapter and implementing basic model evaluation techniques that we can use during the pretraining stage.



As shown in Figure 5.2, the next subsection recaps the text generation we set up at the end of the previous chapter before we dive into the text evaluation and calculation of the training and validation losses in the subsequent subsections.

5.1.1 Using GPT to generate text

In this section, we set up the LLM and briefly recap the text generation process we implemented in chapter 4. We begin by initializing the GPT model that we will evaluate and train in this chapter, using the `GPTModel` class and `GPT_CONFIG_124M` dictionary from chapter 4:

```
import torch
from chapter04 import GPTModel
GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256,  #A
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,     #B
    "qkv_bias": False
}
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()
```

Considering the `GPT_CONFIG_124M` dictionary, the only adjustment we have made compared to the previous chapter is reducing the context length (`context_length`) to 256 tokens. This modification reduces the computational demands of training the model, making it possible to carry out the training on a standard laptop computer.

Originally, the GPT-2 model with 124 million parameters was configured to handle up to 1,024 tokens. After the training process, at the end of this chapter, we will update the context size setting and load pretrained weights to work with a model configured for a 1,024-token context length.

Using the `GPTmodel` instance, we adopt the `generate_text_simple` function introduced in the previous chapter and introduce two handy functions, `text_to_token_ids` and `token_ids_to_text`. These functions facilitate the conversion between text and token representations, a technique we will utilize throughout this chapter. To provide a clearer understanding, Figure 5.3 illustrates this process before we dive into the code.

Figure 5.3 Generating text involves encoding text into token IDs that the LLM processes into logit vectors. The logit vectors are then converted back into token IDs, detokenized into a text representation.

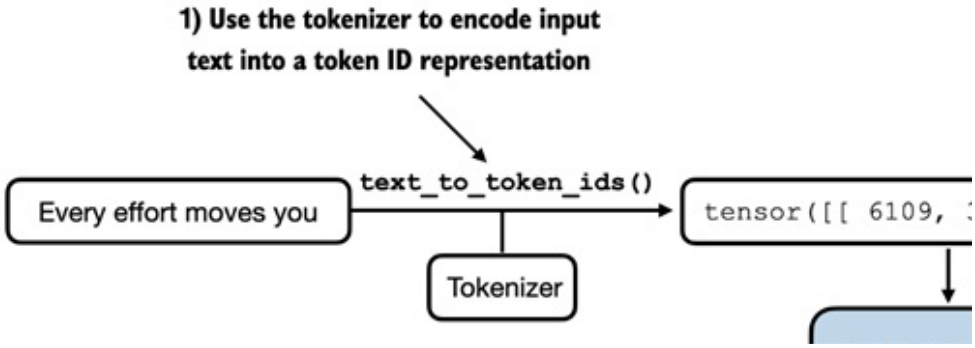


Figure 5.3 illustrates a three-step text generation process using a GPT model. First, the tokenizer converts input text into a series of token IDs, as discussed in chapter 2. Second, the model receives these token IDs and generates corresponding logits, which are vectors representing the probability distribution for each token in the vocabulary, as discussed in chapter 4. Third, these logits are converted back into token IDs, which the tokenizer decodes into human-readable text, completing the cycle from textual input to textual output.

In code, we implement the text generation process as follows:

Listing 5.1 Utility functions for text to token ID conversion

```
import tiktoken
from chapter04 import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftex
    encoded_tensor = torch.tensor(encoded).unsqueeze(0) # add bat
    return encoded_tensor

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0) # remove batch dimension
    return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
```

```
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

Using the preceding code, the `model` generates the following text:

```
Output text:
 Every effort moves you rentingetic wasn refres RexMeCHicular st
```

Based on the output, it's clear the model isn't yet producing coherent text because it hasn't undergone training. To define what makes text "coherent" or "high quality," we have to implement a numerical method to evaluate the generated content. This approach will enable us to monitor and enhance the model's performance throughout its training process.

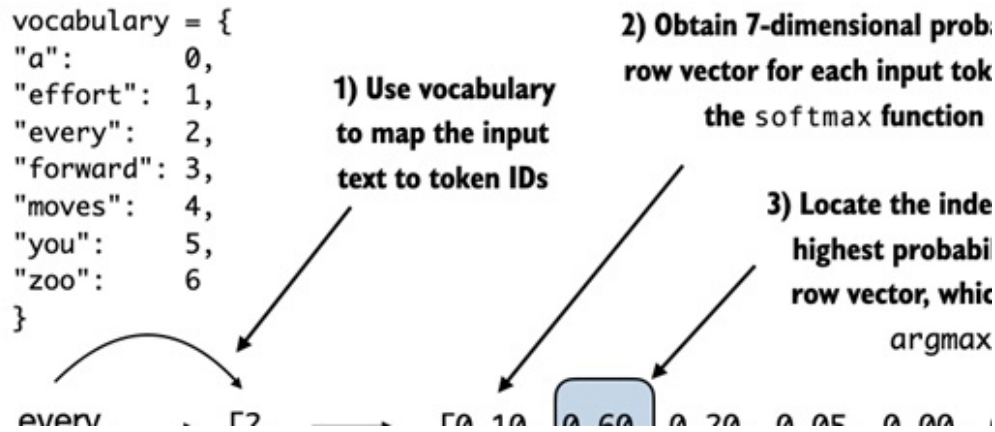
The following section introduces how we calculate a *loss metric* for the generated outputs. This loss serves as a progress and success indicator of the training progress. Furthermore, in subsequent chapters on finetuning LLMs, we will review additional methodologies for assessing model quality.

5.1.2 Calculating the text generation loss

This section explores techniques for numerically assessing text quality generated during training by calculating a so-called text generation loss. We go over this topic step-by-step with a practical example to make the concepts clear and applicable, beginning with a short recap of how the data is loaded from chapter 2 and how the text is generated via the `generate_text_simple` function from chapter 4.

Figure 5.4 illustrates the overall flow from input text to LLM-generated text using a five-step procedure.

Figure 5.4 For each of the 3 input tokens, shown on the left, we compute a vector containing probability scores corresponding to each token in the vocabulary. The index position of the highest probability score in each vector represents the most likely next token ID. These token IDs associated with the highest probability scores are selected and mapped back into a text that represents the text generated by the model.



The text generation process in Figure 5.4 outlines what the `generate_text_simple` function from chapter 4 does internally. We need to perform these same initial steps before we can compute a loss that measures the generated text quality later in this section.

Figure 5.4 outlines the text generation process with a small 7-token vocabulary to fit this image on a single page. However, our `GPTModel` works with a much larger vocabulary consisting of 50,257 words; hence, the token IDs in the following codes will range from 0 to 50,256 rather than 0 to 6.

Also, Figure 5.4 only shows a single text example (`"every effort moves"`) for simplicity. In the following hands-on code example that implements the steps in Figure 5.4, we will work with two input examples (`"every effort moves"` and `"I really like"`) as inputs for the GPT model:

Consider the two input examples, which which have already been mapped to token IDs, corresponding to step 1 in Figure 5.4:

```
inputs = torch.tensor([[16833, 3626, 6100],   # ["every effort mo
                       [40,    1107, 588]])   #  "I really like"]
```

Matching these inputs, the `targets` contain the token IDs we aim for the model to produce:

```
targets = torch.tensor([[3626, 6100, 345  ],  # [" effort moves y
                        [588,  428,  11311]]) #  " really like ch
```

Note that the targets are the inputs but shifted one position forward, a concept

we covered chapter 2 during the implementation of the data loader. This shifting strategy is crucial for teaching the model to predict the next token in a sequence.

When we feed the `inputs` into the model to calculate logit vectors for the two input examples, each comprising three tokens, and apply the softmax function to transform these logit values into probability scores, which corresponds to step 2 in Figure 5.4:

```
with torch.no_grad(): #A
    logits = model(inputs)
probas = torch.softmax(logits, dim=-1) # Probability of each toke
print(probas.shape)
```

The resulting tensor dimension of the probability score (`probas`) tensor is as follows:

```
torch.Size([2, 3, 50257])
```

The first number, 2, corresponds to the two examples (rows) in the `inputs`, also known as batch size. The second number, 3, corresponds to the number of tokens in each input (row). Finally, the last number corresponds to the embedding dimensionality, which is determined by the vocabulary size, as discussed in previous chapters.

Following the conversion from logits to probabilities via the softmax function, the `generate_text_simple` function from chapter 4 then converts the resulting probability scores back into text, as illustrated in steps 3-5 in Figure 5.4.

We can implement steps 3 and 4 by applying the argmax function to the probability scores to obtain the corresponding token IDs:

```
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)
```

Given that we have 2 input batches, each containing 3 tokens, applying the argmax function to the probability scores (step 3 in Figure 5.4) yields 2 sets of outputs, each with 3 predicted token IDs:

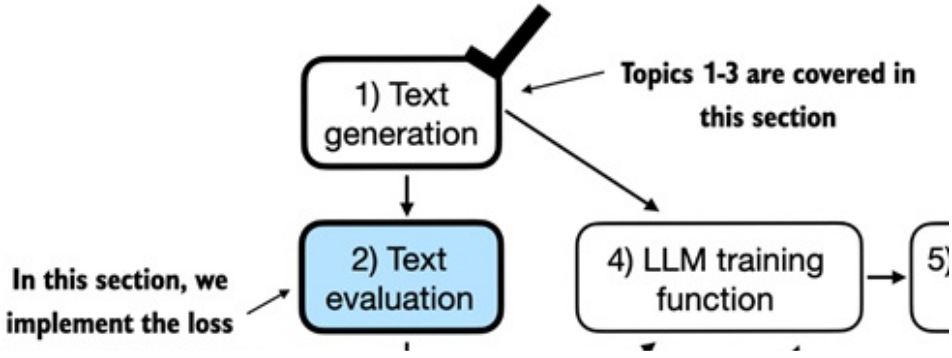
```
Token IDs:
 tensor([[[16657], # First batch
         [  339],
         [42826]],
        [[49906], # Second batch
         [29669],
         [41751]]])
Finally, step 5 converts the token IDs back into text:
print(f"Targets batch 1: {token_ids_to_text(targets[0], tokenizer
print(f"Outputs batch 1: {token_ids_to_text(token_ids[0].flatten(
```

When we decode these tokens, we find that these output tokens are quite different from the target tokens we want the model to generate:

```
Targets batch 1:  effort moves you
Outputs batch 1:  Armed heNetflix
```

The model produces random text that is different from the target text because it has not been trained yet. We now get to the part where we evaluate the performance of the model's generated text numerically via a so-called loss as illustrated in Figure 5.4. Not only is this useful for measuring the quality of the generated text, but it's also a building block for implementing the training function later, which we use to update the model's weight to improve the generated text.

Figure 5.5 We now implement the text evaluation function in the remainder of this section. In the next section, we apply this evaluation function to the entire dataset we use for model training.

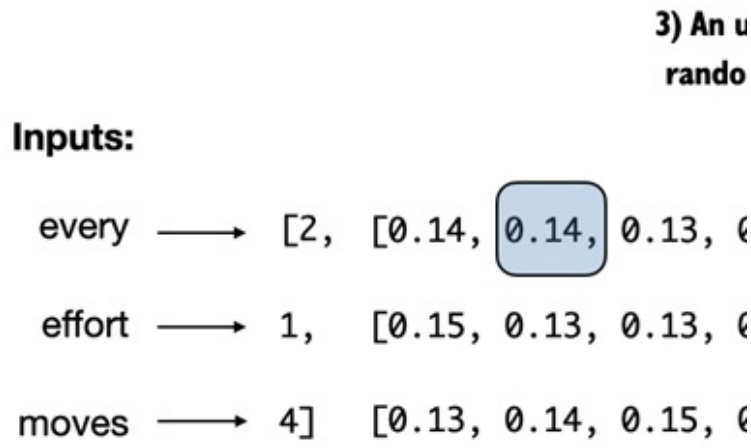


Part of the text evaluation process that we implement in the remainder of this section, as shown in Figure 5.5, is to measure "how far" the generated tokens are from the correct predictions (targets). The training function we implement

later in this chapter will use this information to adjust the model weights to generate text that is more similar to (or ideally matches) the target text.

The model training aims to increase the softmax probability in the index positions corresponding to the correct target token IDs, as illustrated in Figure 5.6. This softmax probability is also used in the evaluation metric we are implementing in the remainder of this section to numerically assess the model's generated outputs: the higher the probability in the correct positions, the better.

Figure 5.6 Before training, the model produces random next-token probability vectors. The goal of model training is to ensure that the probability values corresponding to the highlighted target token IDs are maximized.



Remember that Figure 5.6 displays the softmax probabilities for a compact 7-token vocabulary to fit everything into a single figure. This implies that the starting random values will hover around 1/7, which equals approximately 0.14.

However, the vocabulary we are using for our GPT-2 model has 50,257 tokens, so most of the initial probabilities will hover around 0.00002 via 1/50,257.

For each of the two input texts, we can print the initial softmax probability scores corresponding to the target tokens via the following code:

```
text_idx = 0
target_probas_1 = probas[text_idx, [0, 1, 2], targets[text_idx]]
```

```
print("Text 1:", target_probas_1)

text_idx = 1
target_probas_2 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 2:", target_probas_2)
```

The 3 target token ID probabilities for each batch are as follows:

```
Text 1: tensor([7.4541e-05, 3.1061e-05, 1.1563e-05])
Text 2: tensor([3.9836e-05, 1.6783e-05, 4.7559e-06])
```

The goal of training an LLM is to maximize these values, aiming to get them as close to a probability of 1. This way, we ensure the LLM consistently picks the target token—essentially the next word in the sentence—as the next token it generates.

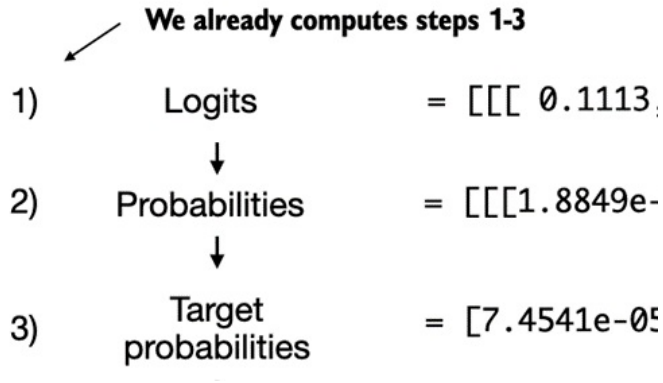
Backpropagation

How do we maximize the softmax probability values corresponding to the target tokens? The big picture is that we update the model weights so that the model outputs higher values for the respective token IDs we want to generate. The weight update is done via a process called *backpropagation*, a standard technique for training deep neural networks (see sections A.3 to A.7 in Appendix A for more details about backpropagation and model training).

Backpropagation requires a loss function, which calculates the difference between the model's predicted output (here, the probabilities corresponding to the target token IDs) and the actual desired output. This loss function measures how far off the model's predictions are from the target values.

In the remainder of this section, we calculate the loss for the probability scores of the two example batches, `target_probas_1` and `target_probas_2`. The main steps are illustrated in Figure 5.7.

Figure 5.7 Calculating the loss involves several steps. Steps 1 to 3 calculate the token probabilities corresponding to the target tensors. These probabilities are then transformed via a logarithm and averaged in steps 4-6.



Since we already applied steps 1-3 listed in Figure 5.7 to obtain `target_probas_1` and `target_probas_2`, we proceed with step 4, applying the *logarithm* to the probability scores:

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_
print(log_probas)
```

This results in the following values:

```
tensor([ -9.5042, -10.3796, -11.3677, -10.1308, -10.9951, -12.256
```

Working with logarithms of probability scores is more manageable in mathematical optimization than handling the scores directly. This topic is outside the scope of this book, but I've detailed it further in a lecture, which is linked in the reference section in appendix B.

Next, we combine these log probabilities into a single score by computing the average (step 5 in Figure 5.7):

```
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
```

The resulting average log probability score is as follows:

```
tensor(-10.7722)
```

The goal is to get the average log probability as close to 0 as possible by updating the model's weights as part of the training process, which we will implement later in section 5.2.

However, in deep learning, the common practice isn't to push the average log probability up to 0 but rather to bring the negative average log probability down to 0. The negative average log probability is simply the average log probability multiplied by -1, which corresponds to step 6 in Figure 5.7:

```
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
```

This prints `tensor(-10.7722)`.

The term for this negative value, -10.7722 turning into 10.7722, is known as the *cross entropy* loss in deep learning.

PyTorch comes in handy here, as it already has a built-in `cross_entropy` function that takes care of all these 6 steps in Figure 5.7 for us.

Cross entropy loss

At its core, the cross entropy loss is a popular measure in machine learning and deep learning that measures the difference between two probability distributions--typically, the true distribution of labels (here, tokens in a dataset) and the predicted distribution from a model (for instance, the token probabilities generated by an LLM).

In the context of machine learning and specifically in frameworks like PyTorch, the `cross_entropy` function computes this measure for discrete outcomes, which is similar to the negative average log probability of the target tokens given the model's generated token probabilities, making the terms cross entropy and negative average log probability related and often used interchangeably in practice.

Before we apply the cross entropy function, let's briefly recall the shape of the logits and target tensors:

```
print("Logits shape:", logits.shape)
print("Targets shape:", targets.shape)
```

The resulting shapes are as follows:

```
Logits shape: torch.Size([2, 3, 50257])
Targets shape: torch.Size([2, 3])
```

As we can see, the `logits` tensor has three dimensions: batch size, number of tokens, and vocabulary size. The `targets` tensor has two dimensions: batch size and number of tokens.

For the cross `entropy_loss` function in PyTorch, we want to flatten these tensors by combining them over the batch dimension:

```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)
```

The resulting tensor dimensions are as follows:

```
Flattened logits: torch.Size([6, 50257])
Flattened targets: torch.Size([6])
```

Remember that the `targets` are the token IDs we want the LLM to generate, and the `logits` contain the unscaled model outputs before they enter the softmax function to obtain the probability scores.

Previously, we applied the softmax function, selected the probability scores corresponding to the target IDs, and computed the negative average log probabilities. PyTorch's `cross_entropy` function will take care of all these steps for us:

```
loss = torch.nn.functional.cross_entropy(logits_flat, targets_fla
print(loss)
```

The resulting loss is the same that we obtained previously when applying the individual steps shown in Figure 5.7 manually:

```
tensor(10.7722)
```

Perplexity

Perplexity is a measure often used alongside cross entropy loss to evaluate the performance of models in tasks like language modeling. It can provide a

more interpretable way to understand the uncertainty of a model in predicting the next token in a sequence.

Perplexity measures how well the probability distribution predicted by the model matches the actual distribution of the words in the dataset. Similar to the loss, a lower perplexity indicates that the model predictions are closer to the actual distribution.

Perplexity can be calculated as `perplexity = torch.exp(loss)`, which returns `tensor(47678.8633)` when applied to the previously calculated loss.

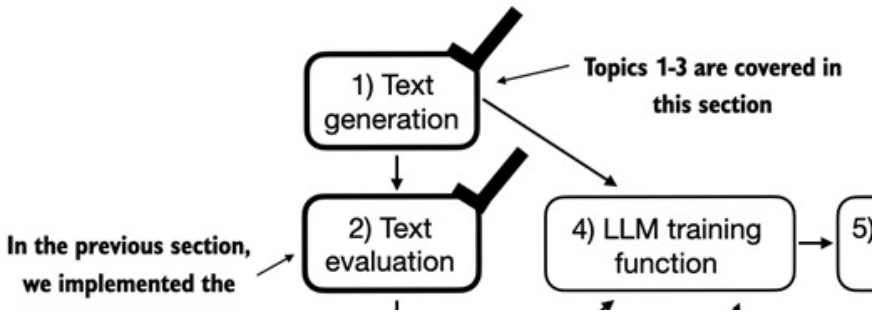
Perplexity is often considered more interpretable than the raw loss value because it signifies the effective vocabulary size about which the model is uncertain at each step. In the given example, this would translate to the model being unsure about which among 47,678 words or tokens in the vocabulary to generate as the next token.

In this section, we calculated the loss for two small text inputs for illustration purposes. In the next section, we apply the loss computation to the entire training and validation sets.

5.1.3 Calculating the training and validation set losses

In this section, we first prepare the training and validation datasets that we will use to train the LLM later in this chapter. Then, we calculate the cross entropy for the training and validation sets, as illustrated in Figure 5.8, which is an important component of the model training process.

Figure 5.8 After computing the cross entropy loss in the previous section, we now apply this loss computation to the entire text dataset that we will use for model training.



To compute the loss on the training and validation datasets as illustrated in Figure 5.8, we use a very small text dataset, the "The Verdict" short story by Edith Wharton, which we have already worked with in chapter 2. By selecting a text from the public domain, we circumvent any concerns related to usage rights. Additionally, the reason why we use such a small dataset is that it allows for the execution of code examples on a standard laptop computer in a matter of minutes, even without a high-end GPU, which is particularly advantageous for educational purposes.

Interested readers can also use the supplementary code of this book to prepare a larger-scale dataset consisting of more than 60,000 public domain books from Project Gutenberg and train an LLM on these (see appendix D for details).

The cost of pretraining LLMs

To put the scale of our project into perspective, consider the training of the 7 billion parameter Llama 2 model, a relatively popular openly available LLM. This model required 184,320 GPU hours on expensive A100 GPUs, processing 2 trillion tokens. At the time of writing, running an 8xA100 cloud server on AWS costs around $30 per hour. A rough estimate puts the total training cost of such an LLM at around $690,000 (calculated as 184,320 hours divided by 8, then multiplied by $30).

The following code loads the "The Verdict" short story we used in chapter 2:

```
file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()
```

After loading the dataset, we can check the number of characters and tokens in the dataset:

```
total_characters = len(text_data)
total_tokens = len(tokenizer.encode(text_data))
print("Characters:", total_characters)
print("Tokens:", total_tokens)
```

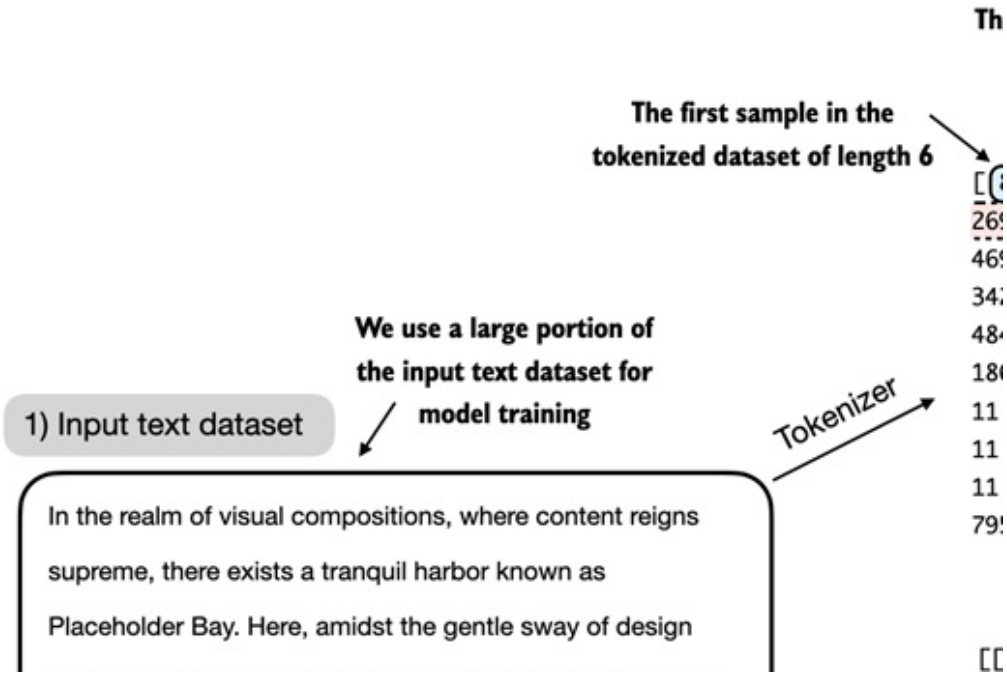
The output is as follows:

```
Characters: 20479
Tokens: 5145
```

With just 5,145 tokens, the text might seem too small to train an LLM, but as mentioned earlier, it's for educational purposes so that we can run the code in minutes instead of weeks. Plus, we will be loading pretrained weights from OpenAI into our `GPTModel` code at the end of this chapter.

Next, we divide the dataset into a training and a validation set and use the data loaders from chapter 2 to prepare the batches for LLM training. This process is visualized in Figure 5.9.

Figure 5.9 When preparing the data loaders, we split the input text into training and validation set portions. Then, we tokenize the text (only shown for the training set portion for simplicity) and divide the tokenized text into chunks of a user-specified length (here 6). Finally, we shuffle the rows and organize the chunked text into batches (here, batch size 2), which we can use for model training.



For visualization purposes, Figure 5.9 uses a `max_length=6` due to spatial constraints. However, for the actual data loaders we are implementing, we set the `max_length` equal to the 256-token context length that the LLM supports so that the LLM sees longer texts during training.

Training with variable lengths

We are training the model with training data presented in similarly-sized chunks for simplicity and efficiency. However, in practice, it can also be beneficial to train an LLM with variable-length inputs to help the LLM to better generalize across different types of inputs when it is being used.

To implement the data splitting and loading visualized in Figure 5.9, we first define a `train_ratio` to use 90% of the data for training and the remaining 10% as validation data for model evaluation during training:

```
train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

Using the `train_data` and `val_data` subsets, we can now create the respective data loader reusing the `create_dataloader_v1` code from chapter 2:

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False
)
```

We used a relatively small batch size in the preceding code to reduce the computational resource demand because we were working with a very small dataset. In practice, training LLMs with batch sizes of 1,024 or larger is not

uncommon.

As an optional check, we can iterate through the data loaders to ensure that they were created correctly:

```
print("Train loader:")
for x, y in train_loader:
    print(x.shape, y.shape)

print("\nValidation loader:")
for x, y in val_loader:
    print(x.shape, y.shape)
```

We should see the following outputs:

```
Train loader:
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])

Validation loader:
torch.Size([2, 256]) torch.Size([2, 256])
```

Based on the preceding code output, we have 9 training set batches with 2 samples and 256 tokens each. Since we allocated only 10% of the data for validation, there is only one validation batch consisting of 2 input examples.

As expected, the input data (`x`) and target data (`y`) have the same shape (the batch size times the number of tokens in each batch) since the targets are the inputs shifted by one position, as discussed in chapter 2.

Next, we implement a utility function to calculate the cross entropy loss of a given batch returned via the training and validation loader:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch, target_batch = input_batch.to(device), target_ba
    logits = model(input_batch)
```

```
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten()
    )
    return loss
```

We can now use this `calc_loss_batch` utility function, which computes the loss for a single batch, to implement the following `calc_loss_loader` function that computes the loss over all the batches sampled by a given data loader:

Listing 5.2 Function to compute the training and validation loss

```
def calc_loss_loader(data_loader, model, device, num_batches=None
    total_loss = 0.
    if num_batches is None:
        num_batches = len(data_loader) #A
    else:
        num_batches = min(num_batches, len(data_loader)) #B
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(input_batch, target_batch, mod
            total_loss += loss.item() #C
        else:
            break
    return total_loss / num_batches #D
```

By default, the `calc_loss_batch` function iterates over all batches in a given data loader, accumulates the loss in the `total_loss` variable, and then computes and averages the loss over the total number of batches. Alternatively, we can specify a smaller number of batches via `num_batches` to speed up the evaluation during model training.

Let's now see this `calc_loss_batch` function in action, applying it to the training and validation set loaders:

```
device = torch.device("cuda" if torch.cuda.is_available() else "c
model.to(device)
train_loss = calc_loss_loader(train_loader, model, device) #B
val_loss = calc_loss_loader(val_loader, model, device)
print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

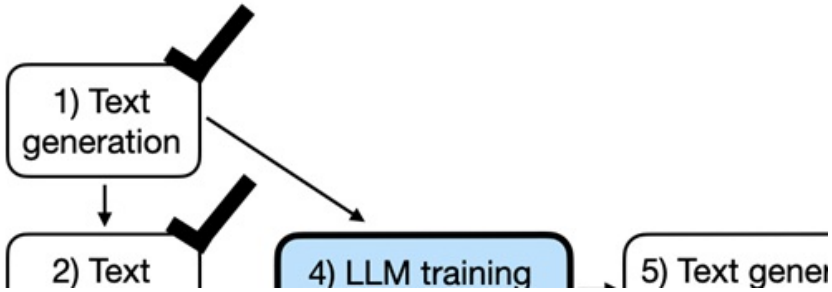
The resulting loss values are as follows:

```
Training loss: 10.98758347829183
Validation loss: 10.98110580444336
```

The loss values are relatively high because the model has not yet been trained. For comparison, the loss approaches 0 if the model learns to generate the next tokens as they appear in the training and validation sets.

Now that we have a way to measure the quality of the generated text, in the next section, we train the LLM to reduce this loss so that it becomes better at generating text, as illustrated in Figure 5.10.

Figure 5.10 We have recapped the text generation process and implemented basic model evaluation techniques to compute the training and validation set losses. Next, we will go to the training functions and pretrain the LLM.



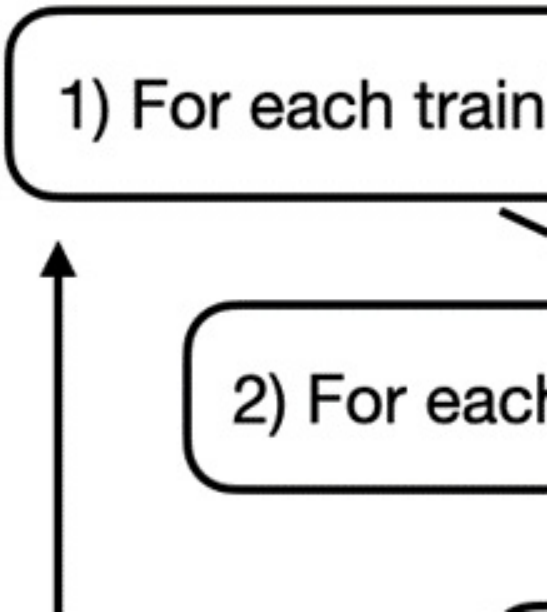
As shown in Figure 5.10, the next section focuses on pretraining the LLM. After model training, we implement alternative text generation strategies and save and load pretrained model weights.

5.2 Training an LLM

In this section, we finally implement the code for pretraining the LLM, our `GPTModel`. For this, we focus on a straightforward training loop, as illustrated in Figure 5.11, to keep the code concise and readable. However, interested readers can learn about more advanced techniques, including l*earning rate warmup*, *cosine annealing*, and *gradient clipping*, in *Appendix D, Adding Bells and Whistles to the Training Loop.*

Figure 5.11 A typical training loop for training deep neural networks in PyTorch consists of several steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update

the model weights so that the training set loss is minimized.



The flowchart in Figure 5.11 depicts a typical PyTorch neural network training workflow, which we use for training an LLM. It outlines eight steps, starting with iterating over each epoch, processing batches, resetting and calculating gradients, updating weights, and concluding with monitoring steps like printing losses and generating text samples. If you are relatively new to training deep neural networks with PyTorch and any of these steps are unfamiliar, consider reading sections A.5 to A.8 in *Appendix A, Introduction to PyTorch*.

In code, we can implement this training flow via the following `train_model_simple` function:

Listing 5.3 The main function for pretraining LLMs

```
def train_model_simple(model, train_loader, val_loader, optimizer
                       eval_freq, eval_iter, start_context):
    train_losses, val_losses, track_tokens_seen = [], [], [] #A
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs): #B
        model.train()
        for input_batch, target_batch in train_loader:
```

```
            optimizer.zero_grad() #C
            loss = calc_loss_batch(input_batch, target_batch, mod
            loss.backward() #D
            optimizer.step() #E
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0: #F
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, Val loss {va

        generate_and_print_sample(  #G
            model, train_loader.dataset.tokenizer, device, start_
        )
    return train_losses, val_losses, track_tokens_seen
```

Note that the `train_model_simple` function we just created uses two functions we have not defined yet: `evaluate_model` and `generate_and_print_sample`.

The `evaluate_model` function corresponds to step 7 in Figure 5.11. It prints the training and validation set losses after each model update so we can evaluate whether the training improves the model.

More specifically, the `evaluate_model` function calculates the loss over the training and validation set while ensuring the model is in evaluation mode with gradient tracking and dropout disabled when calculating the loss over the training and validation sets:

```
def evaluate_model(model, train_loader, val_loader, device, eval_
    model.eval() #A
    with torch.no_grad(): #B
        train_loss = calc_loss_loader(train_loader, model, device
        val_loss = calc_loss_loader(val_loader, model, device, nu
    model.train()
    return train_loss, val_loss
```

Similar to `evaluate_model`, the `generate_and_print_sample` function is a convenience function that we use to track whether the model improves during

the training. In particular, the `generate_and_print_sample` function takes a text snippet (`start_context`) as input, converts it into token IDs, and feeds it to the LLM to generate a text sample using the `generate_text_simple` function we used earlier:

```
def generate_and_print_sample(model, tokenizer, device, start_con
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(devi
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
        decoded_text = token_ids_to_text(token_ids, tokenizer)
        print(decoded_text.replace("\n", " "))  # Compact print f
    model.train()
```

While the `evaluate_model` function gives us a numeric estimate of the model's training progress, this `generate_and_print_sample` text function provides a concrete text example generated by the model to judge its capabilities during training.

AdamW

Adam optimizers are a popular choice for training deep neural networks. However, in our training loop, we opt for the *AdamW* optimizer. AdamW is a variant of Adam that improves the weight decay approach, which aims to minimize model complexity and prevent overfitting by penalizing larger weights. This adjustment allows AdamW to achieve more effective regularization and better generalization and is thus frequently used in the training of LLMs.

Let's see this all in action by training a GPTModel instance for 10 epochs using an AdamW optimizer and the `train_model_simple` function we defined earlier.

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0004, weig
```

```
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=1,
    start_context="Every effort moves you"
)
```

Executing the `training_model_simple` function starts the training process, which takes about 5 minutes on a MacBook Air or a similar laptop to complete. The output printed during this execution is as follows:

```
Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933
Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339
Every effort moves you,,,,,,,,,,,,.
Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616
Every effort moves you, and, and, and, and, and, and, and, and, a
[...]  #A
Ep 9 (Step 000080): Train loss 0.541, Val loss 6.393
Every effort moves you?"  "Yes--quite insensible to the irony. Sh
Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452
Every effort moves you know," was one of the axioms he laid down
```

As we can see, based on the results printed during the training, the training loss improves drastically, starting with a value of 9.558 and converging to 0.762. The language skills of the model have improved quite a lot. In the beginning, the model is only able to append commas to the start context (`"Every effort moves you,,,,,,,,,,,,"`) or repeat the word `"and"`. At the end of the training, it can generate grammatically correct text.

Similar to the training set loss, we can see that the validation loss starts high (9.856) and decreases during the training. However, it never becomes as small as the training set loss and remains at 6.372 after the 10th epoch.

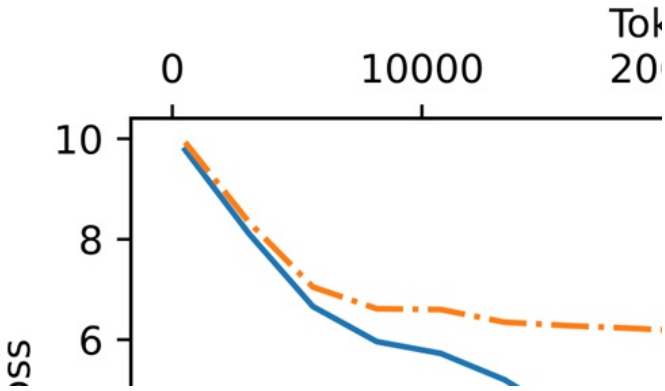
Before discussing the validation loss in more detail, let's create a simple plot that shows the training and validation set losses side by side:

```
import matplotlib.pyplot as plt
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losse
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(epochs_seen, val_losses, linestyle="-.", label="Vali
    ax1.set_xlabel("Epochs")
```

```
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax2 = ax1.twiny()  #A
    ax2.plot(tokens_seen, train_losses, alpha=0)  #B
    ax2.set_xlabel("Tokens seen")
    fig.tight_layout()
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

The resulting training and validation loss plot is shown in Figure 5.12.

Figure 5.12 At the beginning of the training, we observe that both the training and validation set losses sharply decrease, which is a sign that the model is learning. However, the training set loss continues to decrease past the second epoch, whereas the validation loss stagnates. This is a sign that the model is still learning, but it's overfitting to the training set past epoch 2.



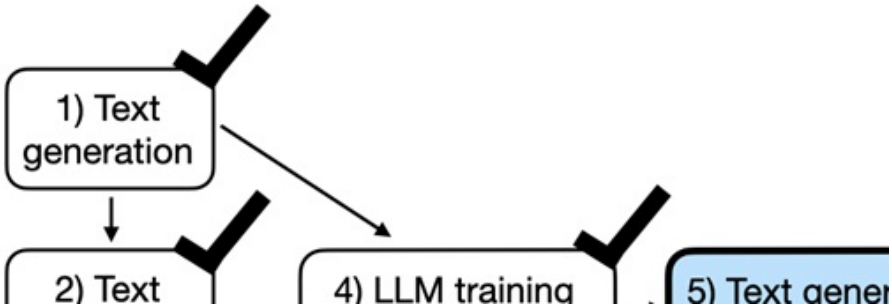
As Figure 5.12 shows, both the training and validation losses start to improve for the first epoch. However, the losses start to diverge past the second epoch. This divergence and the fact that the validation loss is much larger than the training loss indicate that the model is overfitting to the training data. We can confirm that the model memorizes the training data verbatim by searching for the generated text snippets, such as `"quite insensible to the irony"` in the `"The Verdict"` text file.

This memorization is expected since we are working with a very, very small training dataset and training the model for multiple epochs. Usually, it's common to train a model on a much, much larger dataset for only one epoch.

As mentioned earlier, interested readers can try to train the model on 60,000 public domain books from Project Gutenberg, where this overfitting does not occur; see appendix B for details.

In the upcoming section, as shown in Figure 5.13, we explore sampling methods employed by LLMs to mitigate memorization effects, resulting in more novel generated text.

Figure 5.13 Our model can generate coherent text after implementing the training function. However, it often memorizes passages from the training set verbatim. The following section covers strategies to generate more diverse output texts.



As illustrated in Figure 5.13, the next section will cover text generation strategies for LLM to reduce training data memorization and increase the originality of the LLM-generated text before we cover weight loading and saving and loading pretrained weights from OpenAI's GPT model.

5.3 Decoding strategies to control randomness

In this section, we will cover text generation strategies (also called decoding strategies) to generate more original text. First, we briefly revisit the `generate_text_simple` function from the previous chapter that we used inside the `generate_and_print_sample` earlier in this chapter. Then, we will cover two techniques, *temperature scaling*, and *top-k sampling*, to improve this function.

We begin by transferring the model back from the GPU to the CPU since inference with a relatively small model does not require a GPU. Also, after training, we put the model into evaluation model to turn off random

components such as dropout:

```
model.to("cpu")
model.eval()
```

Next, we plug the `GPTModel` instance (`model`) into the `generate_text_simple` function, which uses the LLM to generate one token at a time:

```
tokenizer = tiktoken.get_encoding("gpt2")
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The generated text is as follows:

```
Output text:
Every effort moves you know," was one of the axioms he laid down
```

As explained earlier in section 5.1.2, the generated token is selected at each generation step corresponding to the largest probability score among all tokens in the vocabulary.

This means that the LLM will always generate the same outputs even if we run the `generate_text_simple` function above multiple times on the same start context (`"Every effort moves you"`).

The following subsections introduce two concepts to control the randomness and diversity of the generated text: temperature scaling and top-k sampling.

5.3.1 Temperature scaling

This section introduces temperature scaling, a technique that adds a probabilistic selection process to the next-token generation task.

Previously, inside the `generate_text_simple` function, we always sampled the token with the highest probability as the next token using `torch.argmax`, also known as *greedy decoding*. To generate text with more variety, we can

replace the argmax with a function that samples from a probability distribution (here, the probability scores the LLM generates for each vocabulary entry at each token generation step).

To illustrate the probabilistic sampling with a concrete example, let's briefly discuss the next-token generation process using a very small vocabulary for illustration purposes:

```
vocab = {
    "closer": 0,
    "every": 1,
    "effort": 2,
    "forward": 3,
    "inches": 4,
    "moves": 5,
    "pizza": 6,
    "toward": 7,
    "you": 8,
}
inverse_vocab = {v: k for k, v in vocab.items()}
```

Next, assume the LLM is given the start context `"every effort moves you"` and generates the following next-token logits:

```
next_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)
```

As discussed in the previous chapter, Inside the `generate_text_simple`, we convert the logits into probabilities via the softmax function and obtain the token ID corresponding the generated token via the argmax function, which we can then map back into text via the inverse vocabulary:

```
probas = torch.softmax(next_token_logits, dim=0)
next_token_id = torch.argmax(probas).item()
print(inverse_vocab[next_token_id])
```

Since the largest logit value, and correspondingly the largest softmax probability score, is in the fourth position (index position 3 since Python uses 0-indexing), the generated word is `"forward"`.

To implement a probabilistic sampling process, we can now replace the

argmax with the `multinomial` function in PyTorch:

```
torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print(inverse_vocab[next_token_id])
```

The printed output is `"forward"` just like before. What happened? The `multinomial` function samples the next token proportional to its probability score. In other words, `"forward"` is still the most likely token and will be selected by `multinomial` most of the time but not all the time. To illustrate this, let's implement a function that repeats this sampling 1000 times:

```
def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item() for
    sampled_ids = torch.bincount(torch.tensor(sample))
    for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {inverse_vocab[i]}")
print_sampled_tokens(probas)
The sampling output is as follows:
73 x closer
0 x every
0 x effort
582 x forward
2 x inches
0 x moves
0 x pizza
343 x toward
```

As we can see based on the output, the word `"forward"` is sampled most of the time (582 out of 1000 times), but other tokens such as `"closer"`, `"inches"`, and `"toward"` will also be sampled some of the time. This means that if we replaced the `argmax` function with the `multinomial` function inside the `generate_and_print_sample` function, the LLM would sometimes generate texts such as "`every effort moves you toward`", "`every effort moves you inches`", and "`every effort moves you closer`" instead of "`every effort moves you forward`".

We can further control the distribution and selection process via a concept called temperature scaling, where *temperature scaling* is just a fancy description for dividing the logits by a number greater than 0:

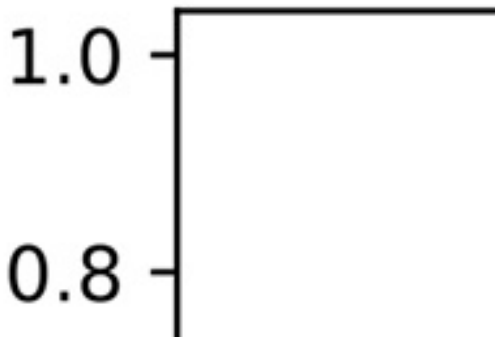
```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

Temperatures greater than 1 result in more uniformly distributed token probabilities, and Temperatures smaller than 1 will result in more confident (sharper or more peaky) distributions. Let's illustrate this by plotting the original probabilities alongside probabilities scaled with different temperature values:

```
temperatures = [1, 0.1, 5]  # Original, higher, and lower tempera
scaled_probas = [softmax_with_temperature(next_token_logits, T) f
x = torch.arange(len(vocab))
bar_width = 0.15
fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):
    rects = ax.bar(x + i * bar_width, scaled_probas[i],
                   bar_width, label=f'Temperature = {T}')
ax.set_ylabel('Probability')
ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend()
plt.tight_layout()
plt.show()
```

The resulting plot is shown in Figure 5.14.

Figure 5.14 A temperature of 1 represents the unscaled probability scores for each token in the vocabulary. Decreasing the temperature to 0.1 sharpens the distribution, so the most likely token (here "forward") will have an even higher probability score. Vice versa, increasing the temperature to 5 makes the distribution more uniform.



A temperature of 1 divides the logits by 1 before passing them to the softmax function to compute the probability scores. In other words, using a temperature of 1 is the same as not using any temperature scaling. In this case, the tokens are selected with a probability equal to the original softmax probability scores via the `multinomial` sampling function in PyTorch.

For example, for the temperature setting 1, the token corresponding to "forward" would be selected with about 60% of the time, as we can see in Figure 5.14.

Also, as we can see in Figure 5.14, applying very small temperatures, such as 0.1, will result in sharper distributions such that the behavior of the `multinomial` function selects the most likely token (here: `"forward"`) almost 100% of the time, approaching the behavior of the argmax function. Vice versa, a temperature of 5 results in a more uniform distribution where other tokens are selected more often. This can add more variety to the generated texts but also more often results in nonsensical text. For example, using the temperature of 5 results in texts such as `"every effort moves you pizza"` about 4% of the time.

Exercise 5.1

Use the `print_sampled_tokens` function to print the sampling frequencies of the softmax probabilities scaled with the temperatures shown in Figure 5.13. How often is the word `"pizza"` sampled in each case? Can you think of a faster and more accurate way to determine how often the word `"pizza"` is sampled?

5.3.2 Top-k sampling

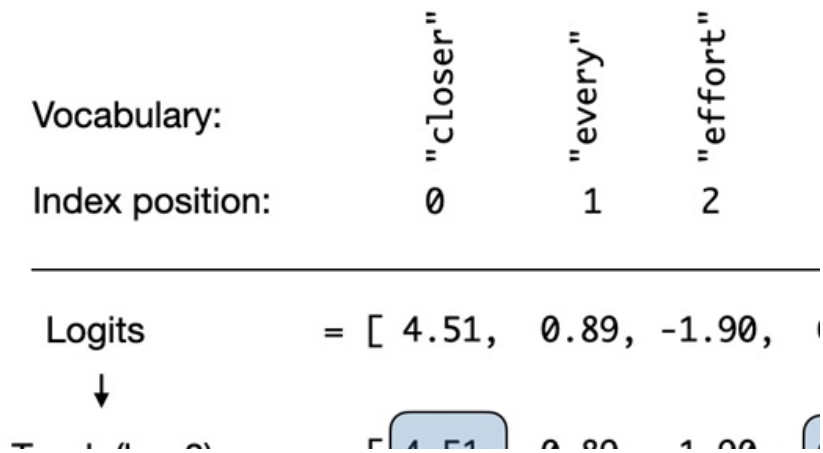
In the previous section, we implemented a probabilistic sampling approach coupled with temperature scaling to increase the diversity of the outputs. We saw that higher temperature values result in more uniformly distributed next-token probabilities, which result in more diverse outputs as it reduces the likelihood of the model repeatedly selecting the most probable token. This method allows for exploring less likely but potentially more interesting and creative paths in the generation process. However, One downside of this

approach is that it sometimes leads to grammatically incorrect or completely nonsensical outputs such as `"every effort moves you pizza"`.

In this section, we introduce another concept called *top-k sampling*, which, when combined with probabilistic sampling and temperature scaling, can improve the text generation results.

In top-k sampling, we can restrict the sampled tokens to the top-k most likely tokens and exclude all other tokens from the selection process by masking their probability scores, as illustrated in Figure 5.15.

Figure 5.15 Using top-k sampling with k=3, we focus on the 3 tokens associated with the highest logits and mask out all other tokens with negative infinity (-inf) before applying the softmax function. This results in a probability distribution with a probability value 0 assigned to all non-top-k tokens.



The approach outlined in Figure 5.15 replaces all non-selected logits with negative infinity value (`-inf`), such that when computing the softmax values, the probability scores of the non-top-k tokens are 0, and the remaining probabilities sum up to 1. (Careful readers may remember this masking trick from the causal attention module we implemented in chapter 3 in section 3.5.1 *Applying a causal attention mask*.)

In code, we can implement the top-k procedure outlined in Figure 5.15 as follows, starting with the selection of the tokens with the largest logit values:

```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
```

```
print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

The logits values and token IDs of the top 3 tokens, in descending order, are as follows:

```
Top logits: tensor([6.7500, 6.2800, 4.5100])
Top positions: tensor([3, 7, 0])
```

Subsequently, we apply PyTorch's `where` function to set the logit values of tokens that are below the lowest logit value within our top-3 selection to negative infinity (`-inf`).

```
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1],  #A
    input=torch.tensor(float('-inf')),  #B
    other=next_token_logits  #C
)
print(new_logits)
```

The resulting logits for the next token in the 9-token vocabulary are as follows:

```
tensor([4.5100,   -inf,   -inf, 6.7500,   -inf,   -inf,   -inf, 6
```

Lastly, let's apply the softmax function to turn these into next-token probabilities:

```
topk_probas = torch.softmax(new_logits, dim=0)
print(topk_probas)
```

As we can see, the result of this top-3 approach are 3 non-zero probability scores:

```
tensor([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0
```

We can now apply the temperature scaling and multinomial function for probabilistic sampling introduced in the previous section to select the next token among these 3 non-zero probability scores to generate the next token. We do this in the next section by modifying the text generation function.

5.3.3 Modifying the text generation function

The previous two subsections introduced two concepts to increase the diversity of LLM-generated text: temperature sampling and top-k sampling. In this section, we combine and add these concepts to modify the `generate_simple` function we used to generate text via the LLM earlier, creating a new `generate` function:

Listing 5.4 A modified text generation function with more diversity

```
def generate(model, idx, max_new_tokens, context_size, temperatur
    for _ in range(max_new_tokens):  #A
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]
        if top_k is not None:  #B
            top_logits, _ = torch.topk(logits, top_k)
            min_val = top_logits[:, -1]
            logits = torch.where(
                logits < min_val,
                torch.tensor(float('-inf')).to(logits.device),
                logits
            )
        if temperature > 0.0:  #C
            logits = logits / temperature
            probs = torch.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
        else:  #D
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

Let's now see this new `generate` function in action:

```
torch.manual_seed(123)
token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)
```

```
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The generated text is as follows:

```
Output text:
 Every effort moves you stand to work on surprise, a one of us ha
```

As we can see, the generated text is very different from the one we previously generated via the `generate_simple` function at the beginning of section 5.3 (`"Every effort moves you know," was one of the axioms he laid...!"`), which was a memorized passage from the training set.

Exercise 5.2

Play around with different temperatures and top-k settings. Based on your observations, can you think of applications where lower temperature and top-k settings are desired? Vice versa, can you think of applications where higher temperature and top-k settings are preferred? (It's recommended to also revisit this exercise at the end of the chapter after loading the pretrained weights from OpenAI.)

Exercise 5.3

What are the different combinations of settings for the `generate` function to force deterministic behavior, that is, disabling the random sampling such that it always produces the same outputs similar to the `generate_simple` function?

So far, we covered how to pretrain LLMs and use them to generate text. The last two sections of this chapter will discuss how we save and load the trained LLM and how we load pretrained weights from OpenAI.

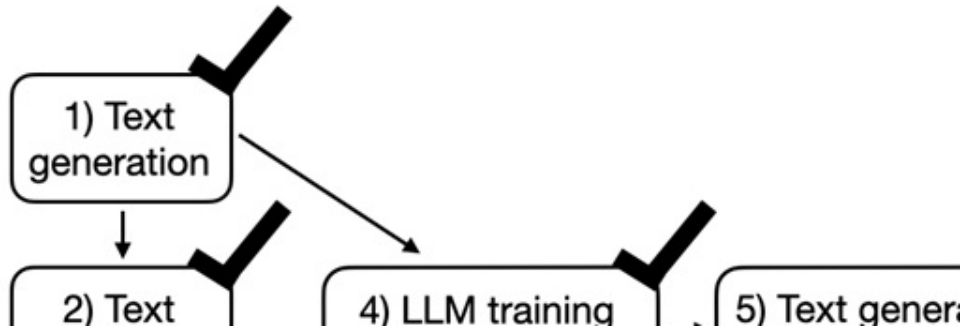
5.4 Loading and saving model weights in PyTorch

In this chapter, we have discussed how to numerically evaluate the training progress and pretrain an LLM from scratch. Even though both the LLM and dataset were relatively small, this exercise showed that pretraining LLMs is computationally expensive. Thus, it is important to be able to save the LLM

so that we don't have to rerun the training every time we want to use it in a new session.

As illustrated in the chapter overview in Figure 5.16, we cover how to save and load a pretrained model in this section. Then, in the upcoming section, we will load a more capable pretrained GPT model from OpenAI into our `GPTModel` instance.

Figure 5.16 After training and inspecting the model, it is often helpful to save the model so that we can use or continue training it later, which is the topic of this section before we load the pretrained model weights from OpenAI in the final section of this chapter.



Fortunately, saving a PyTorch model is relatively straightforward. The recommended way is to save a model's so-called `state_dict`, a dictionary mapping each layer to its parameters, using the `torch.save` function as follows:

```
torch.save(model.state_dict(), "model.pth")
```

In the preceding code, `"model.pth"` is the filename where the `state_dict` is saved. The `.pth` extension is a convention for PyTorch files, though we could technically use any file extension.

Then, after saving the model weights via the `state_dict`, we can load the model weights into a new `GPTModel` model instance as follows:

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth"))
model.eval()
```

As discussed in chapter 4, dropout helps prevent the model from overfitting

to the training data by randomly "dropping out" of a layer's neurons during training. However, during inference, we don't want to randomly drop out any of the information the network has learned. Using `model.eval()` switches the model to evaluation mode for inference, disabling the dropout layers of the `model`.

If we plan to continue pretraining a model later, for example, using the `train_model_simple` function we defined earlier in this chapter, saving the optimizer state is also recommended.

Adaptive optimizers such as AdamW store additional parameters for each model weight. AdamW uses historical data to adjust learning rates for each model parameter dynamically. Without it, the optimizer resets, and the model may learn suboptimally or even fail to converge properly, which means that it will lose the ability to generate coherent text. . Using `torch.save`, we can save both the model and optimizer `state_dict` contents as follows:

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

Then, we can restore the model and optimizer states as follows by first loading the saved data via `torch.load` and then using the `load_state_dict` method:

```
checkpoint = torch.load("model_and_optimizer.pth")
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

Exercise 5.4

After saving the weights, load the model and optimizer in a new Python session or Jupyter notebook file and continue pretraining it for 1 more epoch using the `train_model_simple` function.

5.5 Loading pretrained weights from OpenAI

Previously, for educational purposes, we trained a small GPT-2 model using a limited dataset comprising a short-story book. This approach allowed us to focus on the fundamentals without the need for extensive time and computational resources.

Fortunately, OpenAI openly shared the weights of their GPT-2 models, thus eliminating the need to invest tens to hundreds of thousands of dollars in retraining the model on a large corpus ourselves.

In the remainder of this section, we load these weights into our GPTModel class and use the model for text generation. Here, *weights* refer to the weight parameters that are stored in the `.weight` attributes of PyTorch's `Linear` and `Embedding` layers, for example. We accessed them earlier via `model.parameters()` when training the model.

In the next chapters, we will reuse these pretrained weights to finetune the model for a text classification task and follow instructions similar to ChatGPT.

Note that OpenAI originally saved the GPT-2 weights via TensorFlow, which we have to install to load the weights in Python. Moreover, the following code will use a progress bar tool called `tqdm` to track the download process, which we also have to install.

You can install these libraries by executing the following command in your terminal:

```
pip install tensorflow>=2.15.0  tqdm>=4.66
```

The download code is relatively long, mostly boilerplate, and not very interesting. Hence, instead of devoting precious space in this chapter to discussing Python code for fetching files from the internet, we download the `gpt_download.py` Python module directly from this chapter's online repository:

```
import urllib.request
```

```
url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/"
    "01_main-chapter-code/gpt_download.py"
)
filename = url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

Next, after downloading this file to the local directory of your Python session, readers are encouraged to briefly inspect the contents of this file to ensure that it was saved correctly and contains valid Python code.

We can now import the `download_and_load_gpt2` function from the `gpt_download.py` file as follows, which will load the GPT-2 architecture settings (`settings`) and weight parameters (`params`) into our Python session:

```
from gpt_download import download_and_load_gpt2
settings, params = download_and_load_gpt2(model_size="124M", mode
```

Executing the proceeding codel downloads the following 7 files associated with the 124M parameter GPT-2 model:

```
checkpoint: 100%|███████████████████████████| 77.0/77.0 [00:00<00
encoder.json: 100%|████████████████████████| 1.04M/1.04M [00:00<
hprams.json: 100%|██████████████████████████| 90.0/90.0 [00:00<00
model.ckpt.data-00000-of-00001: 100%|       | 498M/498M [01:09<00
model.ckpt.index: 100%|████████████████████| 5.21k/5.21k [00:00<
model.ckpt.meta: 100%|█████████████████████| 471k/471k [00:00<00
vocab.bpe: 100%|███████████████████████████| 456k/456k [00:00<00
```

Updated download instructions

If the download code does not work for you, it could be due to intermittent internet connection, server issues, or changes in how OpenAI shares the weights of the open-source GPT-2 model. In this case, please visit this chapter's online code repository at https://github.com/rasbt/LLMs-from-scratch for alternative and updated instructions, and please reach out via the Manning Forum for further questions.

After the execution of the previous code has been completed, let's inspect the contents of `settings` and `params`:

```
print("Settings:", settings)
print("Parameter dictionary keys:", params.keys())
```

The contents are as follows:

```
Settings: {'n_vocab': 50257, 'n_ctx': 1024, 'n_embd': 768, 'n_hea
Parameter dictionary keys: dict_keys(['blocks', 'b', 'g', 'wpe',
```

Both `settings` and `params` are Python dictionaries. The `settings` dictionary stores the LLM architecture settings similarly to our manually defined `GPT_CONFIG_124M` settings. The `params` dictionary contains the actual weight tensors. Note that we only printed the dictionary keys because printing the weight contents would take up too much screen space, however, we can inspect these weight tensors by printing the whole dictionary via `print(params)` or by selecting individual tensors via the respective dictionary keys, for example, the embedding layer weights:

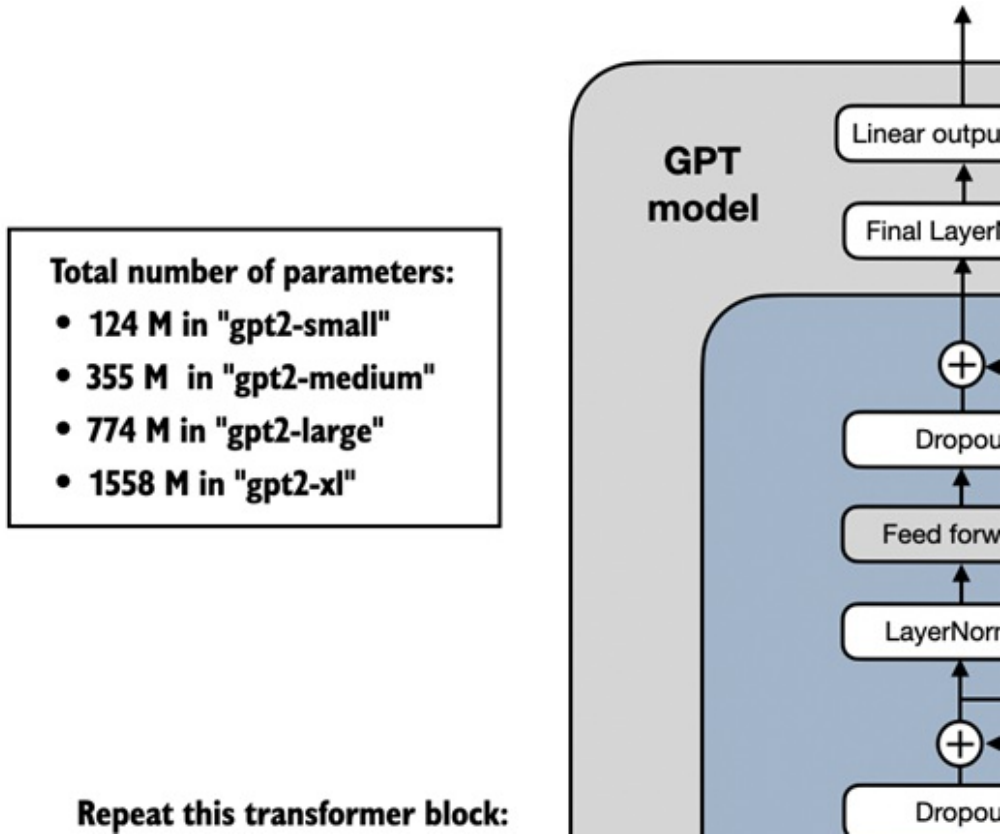
```
print(params["wte"])
print("Token embedding weight tensor dimensions:", params["wte"].
```

The weights of the token embedding layer are as follows:

```
[[-0.11010301 ... -0.1363697   0.01506208   0.04531523]
 [ 0.04034033 ...  0.08605453  0.00253983   0.04318958]
 [-0.12746179  ...  0.08991534 -0.12972379 -0.08785918]
 ...
 [-0.04453601 ...   0.10435229  0.09783269 -0.06952604]
 [ 0.1860082  ...  -0.09625227  0.07847701 -0.02245961]
 [ 0.05135201 ...   0.00704835  0.15519823  0.12067825]]
Token embedding weight tensor dimensions: (50257, 768)
```

We downloaded and loaded the weights of the smallest GPT-2 model via the `download_and_load_gpt2(model_size="124M", ...)` setting. However, note that OpenAI also shares the weights of larger models: `"355M"`, `"774M"`, and `"1558M"`. The overall architecture of these differently-sized GPT models is the same, as illustrated in Figure 5.17.

Figure 5.17 GPT-2 LLMs come in several different model sizes, ranging from 124 million to 1,558 million parameters. The core architecture is the same, with the only difference being the embedding sizes and the number of times individual components like the attention heads and transformer blocks are repeated.



As illustrated in Figure 5.17, the overall architecture of the differently-sized GPT-2 models remains the same, except that different architectural elements are repeated different numbers of times, and the embedding size differs. The remaining code in this chapter is also compatible with these larger models.

After loading the GPT-2 model weights into Python, we still need to transfer them from the `settings` and `params` dictionaries into our `GPTModel` instance.

First, we create a dictionary that lists the differences between the different GPT model sizes, as explained in Figure 5.17:

```
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_head
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_he
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_hea
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads
}
```

Suppose we are interested in loading the smallest model, `"gpt2-small (124M)"`. We can use the corresponding settings from the `model_configs`

table able to update our full-length `GPT_CONFIG_124M` we defined and used earlier throughout the chapter as follows:

```
model_name = "gpt2-small (124M)"
NEW_CONFIG = GPT_CONFIG_124M.copy()
NEW_CONFIG.update(model_configs[model_name])
```

Careful readers may remember that we used a 256-token length earlier, but the original GPT-2 models from OpenAI were trained with a 1,024-token length, so we have to update the `NEW_CONFIG` accordingly:

```
NEW_CONFIG.update({"context_length": 1024})
```

Also, OpenAI used bias vectors in the multi-head attention module's linear layers to implement the query, key, and value matrix computations. Bias vectors are not commonly used in LLMs anymore as they don't improve the modeling performance and are thus unnecessary. However, since we are working with pretrained weights, we need to match the settings for consistency and enable these bias vectors:

```
NEW_CONFIG.update({"qkv_bias": True})
```

We can now use the updated `NEW_CONFIG` dictionary to initialize a new `GPTModel` instance:

```
gpt = GPTModel(NEW_CONFIG)
gpt.eval()
```

By default, the `GPTModel` instance is initialized with random weights for pretraining. The last step to using OpenAI's model weights is to override these random weights with the weights we loaded into the `params` dictionary.

For this, we will first define a small `assign` utility function that checks whether two tensors or arrays (`left` and `right`) have the same dimensions or shape and returns the right tensor as trainable PyTorch parameters:

```
def assign(left, right):
    if left.shape != right.shape:
        raise ValueError(f"Shape mismatch. Left: {left.shape}, Ri
    return torch.nn.Parameter(torch.tensor(right))
```

Next, we define a `load_weights_into_gpt` function that loads the weights from the `params` dictionary into a `GPTModel` instance `gpt`:

Listing 5.5 Loading OpenAI weights into our GPT model code

```
import numpy as np

def load_weights_into_gpt(gpt, params):
    gpt.pos_emb.weight = assign(gpt.pos_emb.weight, params['wpe']
    gpt.tok_emb.weight = assign(gpt.tok_emb.weight, params['wte']

    for b in range(len(params["blocks"])):  #B
        q_w, k_w, v_w = np.split(  #C
            (params["blocks"][b]["attn"]["c_attn"])["w"], 3, axis
        gpt.trf_blocks[b].att.W_query.weight = assign(
            gpt.trf_blocks[b].att.W_query.weight, q_w.T)
        gpt.trf_blocks[b].att.W_key.weight = assign(
            gpt.trf_blocks[b].att.W_key.weight, k_w.T)
        gpt.trf_blocks[b].att.W_value.weight = assign(
            gpt.trf_blocks[b].att.W_value.weight, v_w.T)

        q_b, k_b, v_b = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["b"], 3, axis
        gpt.trf_blocks[b].att.W_query.bias = assign(
            gpt.trf_blocks[b].att.W_query.bias, q_b)
        gpt.trf_blocks[b].att.W_key.bias = assign(
            gpt.trf_blocks[b].att.W_key.bias, k_b)
        gpt.trf_blocks[b].att.W_value.bias = assign(
            gpt.trf_blocks[b].att.W_value.bias, v_b)

        gpt.trf_blocks[b].att.out_proj.weight = assign(
            gpt.trf_blocks[b].att.out_proj.weight,
            params["blocks"][b]["attn"]["c_proj"]["w"].T)
        gpt.trf_blocks[b].att.out_proj.bias = assign(
            gpt.trf_blocks[b].att.out_proj.bias,
            params["blocks"][b]["attn"]["c_proj"]["b"])

        gpt.trf_blocks[b].ff.layers[0].weight = assign(
            gpt.trf_blocks[b].ff.layers[0].weight,
            params["blocks"][b]["mlp"]["c_fc"]["w"].T)
        gpt.trf_blocks[b].ff.layers[0].bias = assign(
            gpt.trf_blocks[b].ff.layers[0].bias,
            params["blocks"][b]["mlp"]["c_fc"]["b"])
        gpt.trf_blocks[b].ff.layers[2].weight = assign(
            gpt.trf_blocks[b].ff.layers[2].weight,
            params["blocks"][b]["mlp"]["c_proj"]["w"].T)
```

```
            gpt.trf_blocks[b].ff.layers[2].bias = assign(
                gpt.trf_blocks[b].ff.layers[2].bias,
                params["blocks"][b]["mlp"]["c_proj"]["b"])

            gpt.trf_blocks[b].norm1.scale = assign(
                gpt.trf_blocks[b].norm1.scale,
                params["blocks"][b]["ln_1"]["g"])
            gpt.trf_blocks[b].norm1.shift = assign(
                gpt.trf_blocks[b].norm1.shift,
                params["blocks"][b]["ln_1"]["b"])
            gpt.trf_blocks[b].norm2.scale = assign(
                gpt.trf_blocks[b].norm2.scale,
                params["blocks"][b]["ln_2"]["g"])
            gpt.trf_blocks[b].norm2.shift = assign(
                gpt.trf_blocks[b].norm2.shift,
                params["blocks"][b]["ln_2"]["b"])

    gpt.final_norm.scale = assign(gpt.final_norm.scale, params["g
    gpt.final_norm.shift = assign(gpt.final_norm.shift, params["b
    gpt.out_head.weight = assign(gpt.out_head.weight, params["wte
```

In the `load_weights_into_gpt` function, we carefully match the weights from OpenAI's implementation with our `GPTModel` implementation. To pick a specific example, OpenAI stored the weight tensor for the output projection layer for the first transformer block as `params["blocks"][0]["attn"]["c_proj"]["w"]`. In our implementation, this weight tensor corresponds to `gpt.trf_blocks[b].att.out_proj.weight`, where `gpt` is a `GPTModel` instance.

Developing the `load_weights_into_gpt` function took a lot of guesswork since OpenAI used a slightly different naming convention from ours. However, the `assign` function would alert us if we try to match two tensors with different dimensions. Also, if we made a mistake in this function, we would notice this as the resulting GPT model would be unable to produce coherent text.

Let's not try the `load_weights_into_gpt` out in practice and load the OpenAI model weights into our `GPTModel` instance `gpt`:

```
load_weights_into_gpt(gpt, params)
gpt.to(device)
```

If the model is loaded correctly, we can now use it to generate new text using

our previous `generate` function:

```
torch.manual_seed(123)
token_ids = generate(
    model=gpt,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=NEW_CONFIG["context_length"],
    top_k=50,
    temperature=1.5
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The resulting text is as follows:

```
Output text:
 Every effort moves you toward finding an ideal new way to practi
What makes us want to be on top of that?
```

We can be confident that we loaded the model weights correctly because the model can produce coherent text. A tiny mistake in this process would cause the model to fail.

In the following chapters, we will work further with this pretrained model and fine-tune it to classify text and follow instructions.

Exercise 5.5

Calculate the training and validation set losses of the GPTModel with the pretrained weights from OpenAI on the "The Verdict" dataset.

Exercise 5.6

Readers are encouraged to experiment with GPT-2 models of different sizes, for example, the largest 1558M parameter model and compare the generated text to the 124M model we loaded in this chapter.

5.6 Summary

- When LLMs generate text, they output one token at a time.

- By default, the next token is generated by converting the model outputs into probability scores and selecting the token from the vocabulary that corresponds to the highest probability score, which is known as "greedy decoding."
- Using probabilistic sampling and temperature scaling, we can influence the diversity and coherence of the generated text.
- Training and validation set losses can be used to gauge the quality of text generated by LLM during training.
- Pretraining an LLM involves changing its weights to minimize the training loss.
- The training loop for LLMs itself is a standard procedure in deep learning, using a conventional cross entropy loss and AdamW optimizer.
- Pretraining an LLM on a large text corpus is time- and resource-intensive so we can load openly available weights from OpenAI as an alternative to pretraining the model on a large dataset ourselves.

Appendix A. Introduction to PyTorch

This chapter covers

- An overview of the PyTorch deep learning library
- Setting up an environment and workspace for deep learning
- Tensors as a fundamental data structure for deep learning
- The mechanics of training deep neural networks
- Training models on GPUs

This chapter is designed to equip you with the necessary skills and knowledge to put deep learning into practice and implement large language models (LLMs) from scratch.

We will introduce PyTorch, a popular Python-based deep learning library, which will be our primary tool for the remainder of this book. This chapter will also guide you through setting up a deep learning workspace armed with PyTorch and GPU support.

Then, you'll learn about the essential concept of tensors and their usage in PyTorch. We will also delve into PyTorch's automatic differentiation engine, a feature that enables us to conveniently and efficiently use backpropagation, which is a crucial aspect of neural network training.

Note that this chapter is meant as a primer for those who are new to deep learning in PyTorch. While this chapter explains PyTorch from the ground up, it's not meant to be an exhaustive coverage of the PyTorch library. Instead, this chapter focuses on the PyTorch fundamentals that we will use to implement LLMs throughout this book. If you are already familiar with deep learning, you may skip this appendix and directly move on to chapter 2, working with text data.

A.1 What is PyTorch

PyTorch (https://pytorch.org/) is an open-source Python-based deep learning library. According to *Papers With Code* (https://paperswithcode.com/trends), a platform that tracks and analyzes research papers, PyTorch has been the most widely used deep learning library for research since 2019 by a wide margin. And according to the *Kaggle Data Science and Machine Learning Survey 2022* (https://www.kaggle.com/c/kaggle-survey-2022), the number of respondents using PyTorch is approximately 40% and constantly grows every year.

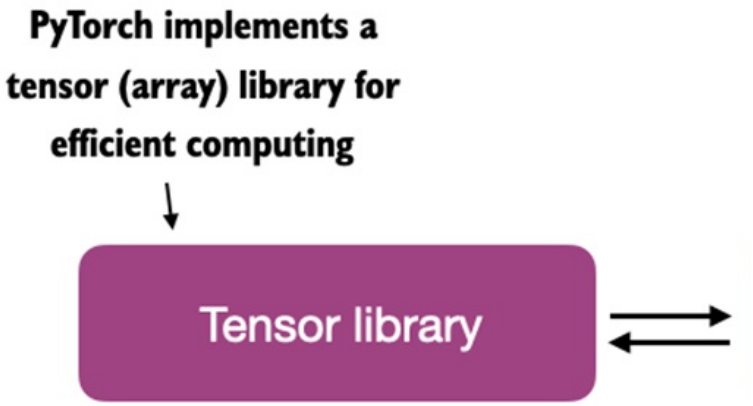
One of the reasons why PyTorch is so popular is its user-friendly interface and efficiency. However, despite its accessibility, it doesn't compromise on flexibility, providing advanced users the ability to tweak lower-level aspects of their models for customization and optimization. In short, for many practitioners and researchers, PyTorch offers just the right balance between usability and features.

In the following subsections, we will define the main features PyTorch has to offer.

A.1.1 The three core components of PyTorch

PyTorch is a relatively comprehensive library, and one way to approach it is to focus on its three broad components, which are summarized in figure A.1.

Figure A.1 PyTorch's three main components include a tensor library as a fundamental building block for computing, automatic differentiation for model optimization, and deep learning utility functions, making it easier to implement and train deep neural network models.



Firstly, PyTorch is a *tensor library* that extends the concept of array-oriented programming library NumPy with the additional feature of accelerated computation on GPUs, thus providing a seamless switch between CPUs and GPUs.

Secondly, PyTorch is an *automatic differentiation engine*, also known as autograd, which enables the automatic computation of gradients for tensor operations, simplifying backpropagation and model optimization.

Finally, PyTorch is a *deep learning library*, meaning that it offers modular, flexible, and efficient building blocks (including pre-trained models, loss functions, and optimizers) for designing and training a wide range of deep learning models, catering to both researchers and developers.

After defining the term deep learning and installing PyTorch in the two following subsections, the remainder of this chapter will go over these three core components of PyTorch in more detail, along with hands-on code examples.

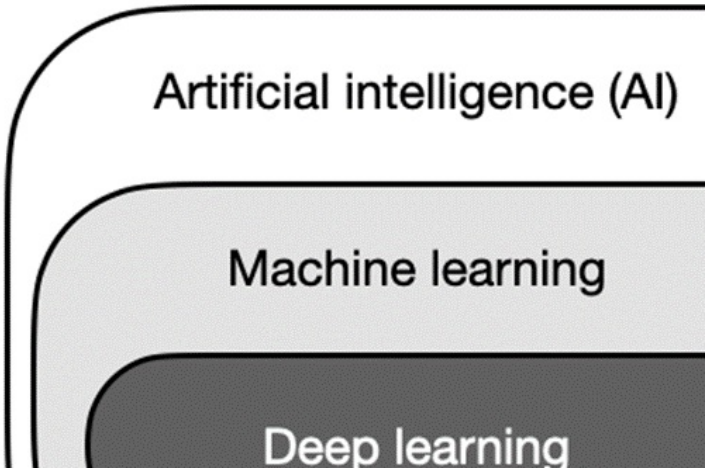
A.1.2 Defining deep learning

LLMs are often referred to as *AI* models in the news. However, as illustrated in the first section of chapter 1 (*1.1 What is an LLM?*) LLMs are also a type of deep neural network, and PyTorch is a deep learning library. Sounds confusing? Let's take a brief moment and summarize the relationship between these terms before we proceed.

AI is fundamentally about creating computer systems capable of performing tasks that usually require human intelligence. These tasks include understanding natural language, recognizing patterns, and making decisions. (Despite significant progress, AI is still far from achieving this level of general intelligence.)

*Machine learnin*g represents a subfield of AI (as illustrated in figure A.2) that focuses on developing and improving learning algorithms. The key idea behind machine learning is to enable computers to learn from data and make predictions or decisions without being explicitly programmed to perform the task. This involves developing algorithms that can identify patterns and learn from historical data and improve their performance over time with more data and feedback.

Figure A.2 Deep learning is a subcategory of machine learning that is focused on the implementation of deep neural networks. In turn, machine learning is a subcategory of AI that is concerned with algorithms that learn from data. AI is the broader concept of machines being able to perform tasks that typically require human intelligence.



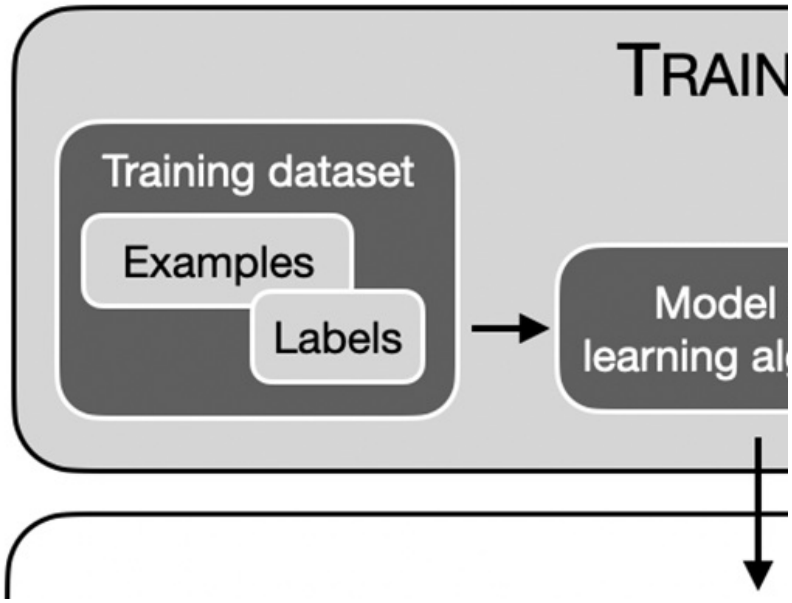
Machine learning has been integral in the evolution of AI, powering many of the advancements we see today, including LLMs. Machine learning is also behind technologies like recommendation systems used by online retailers and streaming services, email spam filtering, voice recognition in virtual assistants, and even self-driving cars. The introduction and advancement of machine learning have significantly enhanced AI's capabilities, enabling it to move beyond strict rule-based systems and adapt to new inputs or changing environments.

Deep learning is a subcategory of machine learning that focuses on the training and application of deep neural networks. These deep neural networks were originally inspired by how the human brain works, particularly the interconnection between many neurons. The "deep" in deep learning refers to the multiple hidden layers of artificial neurons or nodes that allow them to model complex, nonlinear relationships in the data.

Unlike traditional machine learning techniques that excel at simple pattern recognition, deep learning is particularly good at handling unstructured data like images, audio, or text, so deep learning is particularly well suited for LLMs.

The typical predictive modeling workflow (also referred to as *supervised learning*) in machine learning and deep learning is summarized in figure A.3.

Figure A.3 The supervised learning workflow for predictive modeling consists of a training stage where a model is trained on labeled examples in a training dataset. The trained model can then be used to predict the labels of new observations.



Using a learning algorithm, a model is trained on a training dataset consisting of examples and corresponding labels. In the case of an email spam classifier, for example, the training dataset consists of emails and their *spam* and *not-spam* labels that a human identified. Then, the trained model can be used on new observations (new emails) to predict their unknown label (*spam* or *not*

spam).

Of course, we also want to add a model evaluation between the training and inference stages to ensure that the model satisfies our performance criteria before using it in a real-world application.

Note that the workflow for training and using LLMs, as we will see later in this book, is similar to the workflow depicted in figure A.3 if we train them to classify texts. And if we are interested in training LLMs for generating texts, which is the main focus of this book, figure A.3 still applies. In this case, the labels during pretraining can be derived from the text itself (the next-word prediction task introduced in chapter 1). And the LLM will generate entirely new text (instead of predicting labels) given an input prompt during inference.

A.1.3 Installing PyTorch

PyTorch can be installed just like any other Python library or package. However, since PyTorch is a comprehensive library featuring CPU- and GPU-compatible codes, the installation may require additional explanation.

Python version

Many scientific computing libraries do not immediately support the newest version of Python. Therefore, when installing PyTorch, it's advisable to use a version of Python that is one or two releases older. For instance, if the latest version of Python is 3.13, using Python 3.10 or 3.11 is recommended.

For instance, there are two versions of PyTorch: a leaner version that only supports CPU computing and a version that supports both CPU and GPU computing. If your machine has a CUDA-compatible GPU that can be used for deep learning (ideally an NVIDIA T4, RTX 2080 Ti, or newer), I recommend installing the GPU version. Regardless, the default command for installing PyTorch is as follows in a code terminal:

```
pip install torch
```

Suppose your computer supports a CUDA-compatible GPU. In that case, this

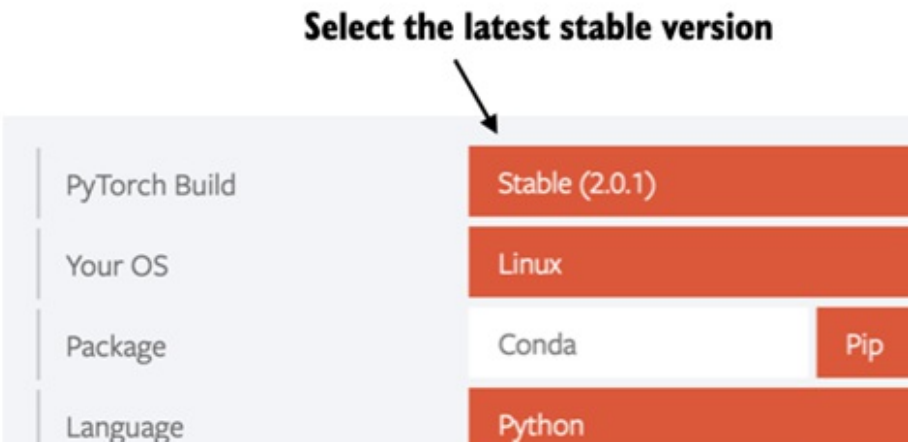
will automatically install the PyTorch version that supports GPU acceleration via CUDA, given that the Python environment you're working on has the necessary dependencies (like pip) installed.

AMD GPUs for deep learning

As of this writing, PyTorch has also added experimental support for AMD GPUs via ROCm. Please see https://pytorch.org for additional instructions.

However, to explicitly install the CUDA-compatible version of PyTorch, it's often better to specify the CUDA you want PyTorch to be compatible with. PyTorch's official website (https://pytorch.org) provides commands to install PyTorch with CUDA support for different operating systems as shown in figure A.4.

Figure A.4 Access the PyTorch installation recommendation on https://pytorch.org to customize and select the installation command for your system.



(Note that the command shown in figure A.4 will also install the `torchvision` and `torchaudio` libraries, which are optional for this book.)

As of this writing, this book is based on PyTorch 2.0.1, so it's recommended to use the following installation command to install the exact version to guarantee compatibility with this book:

```
pip install torch==2.0.1
```

However, as mentioned earlier, given your operating system, the installation command might slightly differ from the one shown above. Thus, I recommend visiting the https://pytorch.org website and using the installation menu (see figure A4) to select the installation command for your operating system and replace `torch` with `torch==2.0.1` in this command.

To check the version of PyTorch, you can execute the following code in PyTorch:

```
import torch
torch.__version__
```

This prints:

```
'2.0.1'
```

PyTorch and Torch

Note that the Python library is named "torch" primarily because it's a continuation of the Torch library but adapted for Python (hence, "PyTorch"). The name "torch" acknowledges the library's roots in Torch, a scientific computing framework with wide support for machine learning algorithms, which was initially created using the Lua programming language.

If you are looking for additional recommendations and instructions for setting up your Python environment or installing the other libraries used later in this book, I recommend visiting the supplementary GitHub repository of this book at https://github.com/rasbt/LLMs-from-scratch.

After installing PyTorch, you can check whether your installation recognizes your built-in NVIDIA GPU by running the following code in Python:

```
import torch
torch.cuda.is_available()
```

This returns:

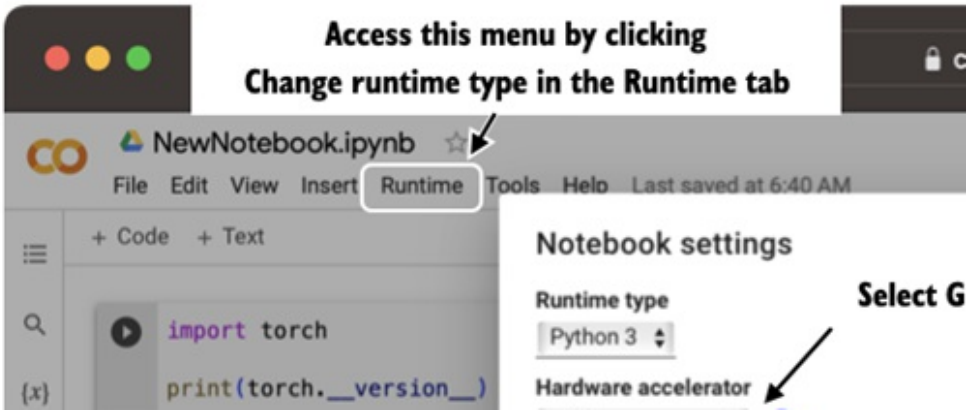
```
True
```

If the command returns True, you are all set. If the command returns False,

your computer may not have a compatible GPU, or PyTorch does not recognize it. While GPUs are not required for the initial chapters in this book, which are focused on implementing LLMs for educational purposes, they can significantly speed up deep learning-related computations.

If you don't have access to a GPU, there are several cloud computing providers where users can run GPU computations against an hourly cost. A popular Jupyter-notebook-like environment is Google Colab (https://colab.research.google.com), which provides time-limited access to GPUs as of this writing. Using the "Runtime" menu, it is possible to select a GPU, as shown in the screenshot in figure A.5.

Figure A.5 Select a GPU device for Google Colab under the *Runtime/Change runtime type* menu.



PyTorch on Apple Silicon

If you have an Apple Mac with an Apple Silicon chip (like the M1, M2, M3, or newer models), you have the option to leverage its capabilities to accelerate PyTorch code execution. To use your Apple Silicon chip for PyTorch, you first need to install PyTorch as you normally would. Then, to check if your Mac supports PyTorch acceleration with its Apple Silicon chip, you can run a simple code snippet in Python:

```
print(torch.backends.mps.is_available())
```

If it returns `True`, it means that your Mac has an Apple Silicon chip that can be used to accelerate PyTorch code.

Exercise A.1

Install and set up PyTorch on your computer.

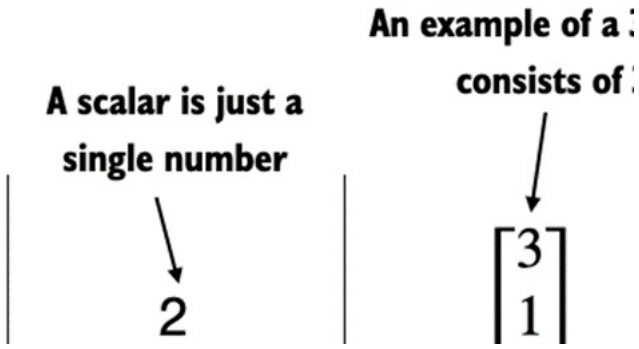
Exercise A.2

Run the supplementary Chapter 2 code at https://github.com/rasbt/LLMs-from-scratch that checks whether your environment is set up correctly..

A.2 Understanding tensors

Tensors represent a mathematical concept that generalizes vectors and matrices to potentially higher dimensions. In other words, tensors are mathematical objects that can be characterized by their order (or rank), which provides the number of dimensions. For example, a scalar (just a number) is a tensor of rank 0, a vector is a tensor of rank 1, and a matrix is a tensor of rank 2, as illustrated in figure A.6

Figure A.6 An illustration of tensors with different ranks. Here 0D corresponds to rank 0, 1D to rank 1, and 2D to rank 2. Note that a 3D vector, which consists of 3 elements, is still a rank 1 tensor.



From a computational perspective, tensors serve as data containers. For instance, they hold multi-dimensional data, where each dimension represents a different feature. Tensor libraries, such as PyTorch, can create, manipulate, and compute with these multi-dimensional arrays efficiently. In this context, a tensor library functions as an array library.

PyTorch tensors are similar to NumPy arrays but have several additional features important for deep learning. For example, PyTorch adds an automatic differentiation engine, simplifying *computing gradients*, as discussed later in section 2.4. PyTorch tensors also support GPU computations to speed up deep neural network training, which we will discuss later in section 2.8.

PyTorch's has a NumPy-like API

As you will see in the upcoming sections, PyTorch adopts most of the NumPy array API and syntax for its tensor operations. If you are new to NumPy, you can get a brief overview of the most relevant concepts via my article Scientific Computing in Python: Introduction to NumPy and Matplotlib at https://sebastianraschka.com/blog/2020/numpy-intro.html.

The following subsections will look at the basic operations of the PyTorch tensor library, showing how to create simple tensors and going over some of the essential operations.

A.2.1 Scalars, vectors, matrices, and tensors

As mentioned earlier, PyTorch tensors are data containers for array-like structures. A scalar is a 0-dimensional tensor (for instance, just a number), a vector is a 1-dimensional tensor, and a matrix is a 2-dimensional tensor. There is no specific term for higher-dimensional tensors, so we typically refer to a 3-dimensional tensor as just a 3D tensor, and so forth.

We can create objects of PyTorch's `Tensor` class using the `torch.tensor` function as follows:

Listing A.1 Creating PyTorch tensors

```
import torch


tensor0d = torch.tensor(1) #A


tensor1d = torch.tensor([1, 2, 3]) #B
```

```
tensor2d = torch.tensor([[1, 2], [3, 4]]) #C


tensor3d = torch.tensor([[[1, 2], [3, 4]], [[5, 6], [7, 8]]]) #D
```

A.2.2 Tensor data types

In the previous section, we created tensors from Python integers. In this case, PyTorch adopts the default 64-bit integer data type from Python. We can access the data type of a tensor via the `.dtype` attribute of a tensor:

```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

This prints:

```
torch.int64
```

If we create tensors from Python floats, PyTorch creates tensors with a 32-bit precision by default, as we can see below:

```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

The output is:

```
torch.float32
```

This choice is primarily due to the balance between precision and computational efficiency. A 32-bit floating point number offers sufficient precision for most deep learning tasks, while consuming less memory and computational resources than a 64-bit floating point number. Moreover, GPU architectures are optimized for 32-bit computations, and using this data type can significantly speed up model training and inference.

Moreover, it is possible to readily change the precision using a tensor's `.to` method. The following code demonstrates this by changing a 64-bit integer tensor into a 32-bit float tensor:

```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

This returns:

```
torch.float32
```

For more information about different tensor data types available in PyTorch, I recommend checking the official documentation at https://pytorch.org/docs/stable/tensors.html.

A.2.3 Common PyTorch tensor operations

Comprehensive coverage of all the different PyTorch tensor operations and commands is outside the scope of this book. However, we will briefly describe relevant operations as we introduce them throughout the book.

Before we move on to the next section covering the concept behind computation graphs, below is a list of the most essential PyTorch tensor operations.

We already introduced the `torch.tensor()` function to create new tensors.

```
tensor2d = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(tensor2d)
```

This prints:

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

In addition, the `.shape` attribute allows us to access the shape of a tensor:

```
print(tensor2d.shape)
```

The output is:

```
torch.Size([2, 3])
```

As you can see above, `.shape` returns `[2, 3]`, which means that the tensor has 2 rows and 3 columns. To reshape the tensor into a 3 by 2 tensor, we can

use the `.reshape` method:

```
print(tensor2d.reshape(3, 2))
```

This prints:

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

However, note that the more common command for reshaping tensors in PyTorch is `.view()`:

```
print(tensor2d.view(3, 2))
```

The output is:

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

Similar to `.reshape` and `.view`, there are several cases where PyTorch offers multiple syntax options for executing the same computation. This is because PyTorch initially followed the original Lua Torch syntax convention but then also added syntax to make it more similar to NumPy upon popular request.

Next, we can use `.T` to transpose a tensor, which means flipping it across its diagonal. Note that this is similar from reshaping a tensor as you can see based on the result below:

```
print(tensor2d.T)
```

The output is:

```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

Lastly, the common way to multiply two matrices in PyTorch is the `.matmul` method:

```
print(tensor2d.matmul(tensor2d.T))
```

The output is:

```
tensor([[14, 32],
        [32, 77]])
```

However, we can also adopt the `@` operator, which accomplishes the same thing more compactly:

```
print(tensor2d @ tensor2d.T)
```

This prints:

```
tensor([[14, 32],
        [32, 77]])
```

As mentioned earlier, we will introduce additional operations throughout this book when needed. For readers who'd like to browse through all the different tensor operations available in PyTorch (hint: we won't need most of these), I recommend checking out the official documentation at https://pytorch.org/docs/stable/tensors.html.

A.3 Seeing models as computation graphs

In the previous section, we covered one of the major three components of PyTorch, namely, its tensor library. Next in line is PyTorch's automatic differentiation engine, also known as autograd. PyTorch's autograd system provides functions to compute gradients in dynamic computational graphs automatically. But before we dive deeper into computing gradients in the next section, let's define the concept of a computational graph.

A computational graph (or computation graph in short) is a directed graph that allows us to express and visualize mathematical expressions. In the context of deep learning, a computation graph lays out the sequence of calculations needed to compute the output of a neural network -- we will need this later to compute the required gradients for backpropagation, which is the main training algorithm for neural networks.

Let's look at a concrete example to illustrate the concept of a computation graph. The following code implements the forward pass (prediction step) of a

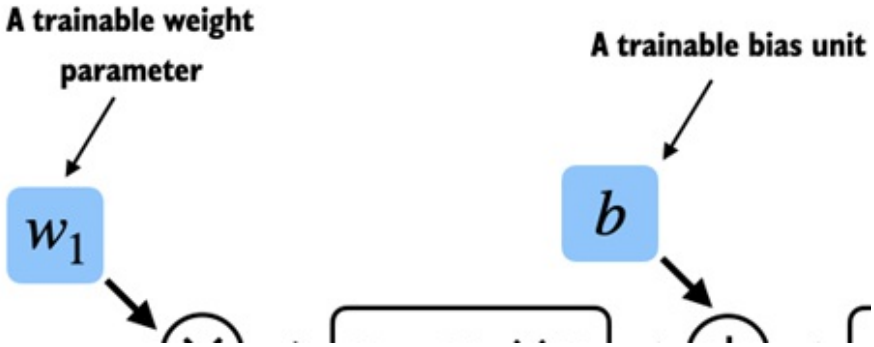
simple logistic regression classifier, which can be seen as a single-layer neural network, returning a score between 0 and 1 that is compared to the true class label (0 or 1) when computing the loss:

Listing A.2 A logistic regression forward pass

```
import torch.nn.functional as F   #A

y = torch.tensor([1.0])  #B
x1 = torch.tensor([1.1]) #C
w1 = torch.tensor([2.2]) #D
b = torch.tensor([0.0])  #E
z = x1 * w1 + b          #F
a = torch.sigmoid(z)     #G

loss = F.binary_cross_entropy(a, y)
```

If not all components in the code above make sense to you, don't worry. The point of this example is not to implement a logistic regression classifier but rather to illustrate how we can think of a sequence of computations as a computation graph, as shown in figure A.7.

Figure A.7 A logistic regression forward pass as a computation graph. The input feature x_1 is multiplied by a model weight w_1 and passed through an activation function σ after adding the bias. The loss is computed by comparing the model output a with a given label y.

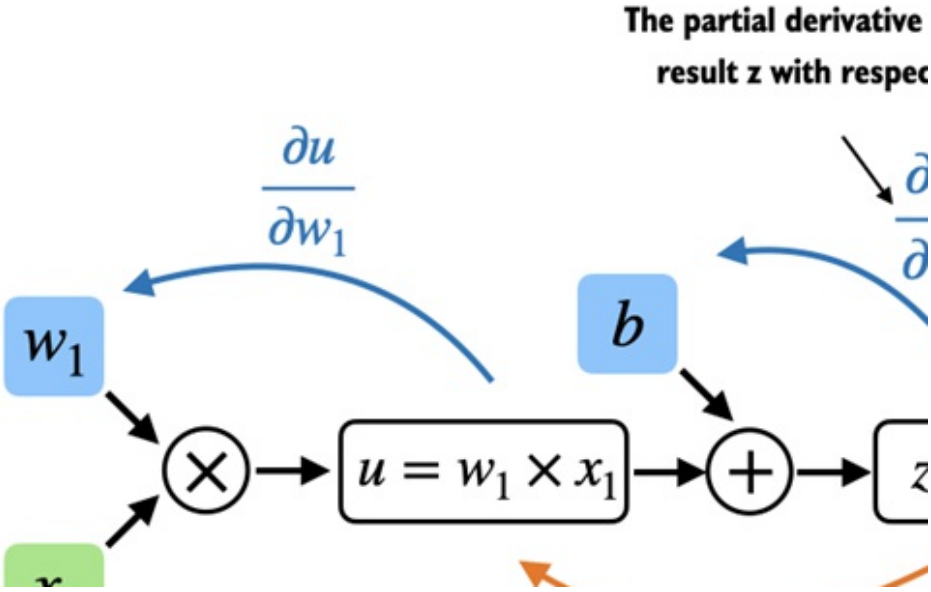


In fact, PyTorch builds such a computation graph in the background, and we can use this to calculate gradients of a loss function with respect to the model parameters (here w1 and b) to train the model, which is the topic of the upcoming sections.

A.4 Automatic differentiation made easy

In the previous section, we introduced the concept of computation graphs. If we carry out computations in PyTorch, it will build such a graph internally by default if one of its terminal nodes has the `requires_grad` attribute set to `True`. This is useful if we want to compute gradients. Gradients are required when training neural networks via the popular backpropagation algorithm, which can be thought of as an implementation of the *chain rule* from calculus for neural networks, which is illustrated in figure A.8.

Figure A.8 The most common way of computing the loss gradients in a computation graph involves applying the chain rule from right to left, which is also called reverse-model automatic differentiation or backpropagation. It means we start from the output layer (or the loss itself) and work backward through the network to the input layer. This is done to compute the gradient of the loss with respect to each parameter (weights and biases) in the network, which informs how we update these parameters during training.



Partial derivatives and gradients

Figure A.8 shows partial derivatives, which measure the rate at which a function changes with respect to one of its variables. A gradient is a vector containing all of the partial derivatives of a multivariate function, a function with more than one variable as input.

If you are not familiar or don't remember the partial derivatives, gradients, or the chain rule from calculus, don't worry. On a high level, all you need to know for this book is that the chain rule is a way to compute gradients of a loss function with respect to the model's parameters in a computation graph. This provides the information needed to update each parameter in a way that minimizes the loss function, which serves as a proxy for measuring the model's performance, using a method such as gradient descent. We will revisit the computational implementation of this training loop in PyTorch in section 2.7, *A typical training loop*.

Now, how is this all related to the second component of the PyTorch library we mentioned earlier, the automatic differentiation (autograd) engine? By tracking every operation performed on tensors, PyTorch's autograd engine constructs a computational graph in the background. Then, calling the `grad` function, we can compute the gradient of the loss with respect to model parameter `w1` as follows:

Listing A.3 Computing gradients via autograd

```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True)  #A
grad_L_b = grad(loss, b, retain_graph=True)
```

Let's show the resulting values of the loss with respect to the model's parameters:

```
print(grad_L_w1)
print(grad_L_b)
```

The prints:

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

Above, we have been using the grad function "manually," which can be useful for experimentation, debugging, and demonstrating concepts. But in practice, PyTorch provides even more high-level tools to automate this process. For instance, we can call `.backward` on the loss, and PyTorch will compute the gradients of all the leaf nodes in the graph, which will be stored via the tensors' `.grad` attributes:

```
loss.backward()
print(w1.grad)
print(b.grad)
```

The outputs are:

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

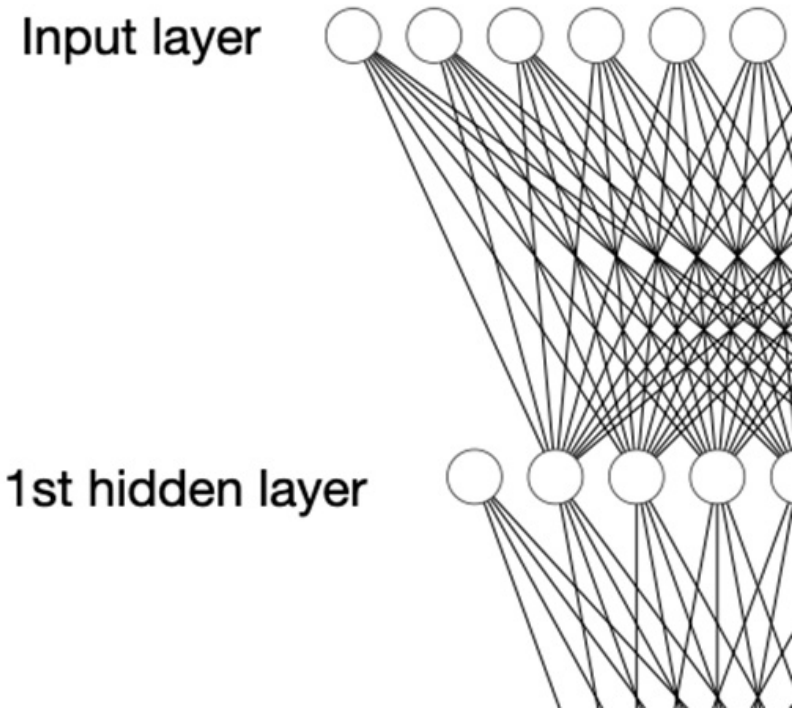
If this section is packed with a lot of information and you may be overwhelmed by the calculus concepts, don't worry. While this calculus jargon was a means to explain PyTorch's autograd component, all you need to take away from this section is that PyTorch takes care of the calculus for us via the `.backward` method -- we won't need to compute any derivatives or gradients by hand in this book.

A.5 Implementing multilayer neural networks

In the previous sections, we covered PyTorch's tensor and autograd components. This section focuses on PyTorch as a library for implementing deep neural networks.

To provide a concrete example, we focus on a multilayer perceptron, which is a fully connected neural network, as illustrated in figure A.9.

Figure A.9 An illustration of a multilayer perceptron with 2 hidden layers. Each node represents a unit in the respective layer. Each layer has only a very small number of nodes for illustration purposes.



When implementing a neural network in PyTorch, we typically subclass the `torch.nn.Module` class to define our own custom network architecture. This `Module` base class provides a lot of functionality, making it easier to build and train models. For instance, it allows us to encapsulate layers and operations and keep track of the model's parameters.

Within this subclass, we define the network layers in the `__init__` constructor and specify how they interact in the forward method. The forward method describes how the input data passes through the network and comes together as a computation graph.

In contrast, the backward method, which we typically do not need to implement ourselves, is used during training to compute gradients of the loss function with respect to the model parameters, as we will see in section 2.7, *A typical training loop*.

The following code implements a classic multilayer perceptron with two hidden layers to illustrate a typical usage of the `Module` class:

Listing A.4 A multilayer perceptron with two hidden layers

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):  #A
        super().__init__()

        self.layers = torch.nn.Sequential(

            # 1st hidden layer
            torch.nn.Linear(num_inputs, 30),  #B
            torch.nn.ReLU(),  #C

            # 2nd hidden layer
            torch.nn.Linear(30, 20),  #D
            torch.nn.ReLU(),

            # output layer
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits   #E
```

We can then instantiate a new neural network object as follows:

```
model = NeuralNetwork(50, 3)
```

But before using this new `model` object, it is often useful to call `print` on the model to see a summary of its structure:

```
print(model)
```

This prints:

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Note that we used the `Sequential` class when we implemented the

`NeuralNetwork` class. Using Sequential is not required, but it can make our life easier if we have a series of layers that we want to execute in a specific order, as is the case here. This way, after instantiating `self.layers = Sequential(...)` in the `__init__` constructor, we just have to call the `self.layers` instead of calling each layer individually in the `NeuralNetwork`'s `forward` method.

Next, let's check the total number of trainable parameters of this model:

```
num_params = sum(p.numel() for p in model.parameters() if p.requi
print("Total number of trainable model parameters:", num_params)
```

This prints:

```
Total number of trainable model parameters: 2213
```

Note that each parameter for which `requires_grad=True` counts as a trainable parameter and will be updated during training (more on that later in section 2.7, *A typical training loop*).

In the case of our neural network model with the two hidden layers above, these trainable parameters are contained in the `torch.nn.Linear` layers. A *linear* layer multiplies the inputs with a weight matrix and adds a bias vector. This is sometimes also referred to as a *feedforward* or *fully connected* layer.

Based on the `print(model)` call we executed above, we can see that the first `Linear` layer is at index position 0 in the layers attribute. We can access the corresponding weight parameter matrix as follows:

```
print(model.layers[0].weight)
```

This prints:

```
Parameter containing:
tensor([[ 0.1174, -0.1350, -0.1227,  ...,  0.0275, -0.0520, -0.01
        [-0.0169,  0.1265,  0.0255,  ..., -0.1247,  0.1191, -0.06
        [-0.0973, -0.0974, -0.0739,  ..., -0.0068, -0.0892,  0.10
        ...,
        [-0.0681,  0.1058, -0.0315,  ..., -0.1081, -0.0290, -0.13
        [-0.0159,  0.0587, -0.0916,  ..., -0.1153,  0.0700,  0.07
        [-0.1019,  0.1345, -0.0176,  ...,  0.0114, -0.0559, -0.00
```

```
        requires_grad=True)
```

Since this is a large matrix that is not shown in its entirety, let's use the `.shape` attribute to show its dimensions:

```
print(model.layers[0].weight.shape)
```

The result is:

```
torch.Size([30, 50])
```

(Similarly, you could access the bias vector via `model.layers[0].bias`.)

The weight matrix above is a 30x50 matrix, and we can see that the `requires_grad` is set to `True`, which means its entries are trainable -- this is the default setting for weights and biases in `torch.nn.Linear`.

Note that if you execute the code above on your computer, the numbers in the weight matrix will likely differ from those shown above. This is because the model weights are initialized with small random numbers, which are different each time we instantiate the network. In deep learning, initializing model weights with small random numbers is desired to break symmetry during training -- otherwise, the nodes would be just performing the same operations and updates during backpropagation, which would not allow the network to learn complex mappings from inputs to outputs.

However, while we want to keep using small random numbers as initial values for our layer weights, we can make the random number initialization reproducible by seeding PyTorch's random number generator via `manual_seed`:

```
torch.manual_seed(123)
model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

The result is:

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702,  ...,  0.0222,  0.1260,  0.08
         [ 0.0502,  0.0307,  0.0333,  ...,  0.0951,  0.1134, -0.02
         [ 0.1077, -0.1108,  0.0122,  ...,  0.0108, -0.1049, -0.10
```

```
        ...,
        [-0.0787,  0.1259,  0.0803,  ...,  0.1218,  0.1303, -0.13
        [ 0.1359,  0.0175, -0.0673,  ...,  0.0674,  0.0676,  0.10
        [ 0.0790,  0.1343, -0.0293,  ...,  0.0344, -0.0971, -0.05
       requires_grad=True)
```

Now, after we spent some time inspecting the `NeuraNetwork` instance, let's briefly see how it's used via the forward pass:

```
torch.manual_seed(123)
X = torch.rand((1, 50))
out = model(X)
print(out)
```

The result is:tensor([[-0.1262, 0.1080, -0.1792]], `grad_fn=<AddmmBackward0>)`

In the code above, we generated a single random training example `X` as a toy input (note that our network expects 50-dimensional feature vectors) and fed it to the model, returning three scores. When we call `model(x)`, it will automatically execute the forward pass of the model.

The forward pass refers to calculating output tensors from input tensors. This involves passing the input data through all the neural network layers, starting from the input layer, through hidden layers, and finally to the output layer.

These three numbers returned above correspond to a score assigned to each of the three output nodes. Notice that the output tensor also includes a `grad_fn` value.

Here, `grad_fn=<AddmmBackward0>` represents the last-used function to compute a variable in the computational graph. In particular, `grad_fn=<AddmmBackward0>` means that the tensor we are inspecting was created via a matrix multiplication and addition operation. PyTorch will use this information when it computes gradients during backpropagation. The `<AddmmBackward0>` part of `grad_fn=<AddmmBackward0>` specifies the operation that was performed. In this case, it is an `Addmm` operation. `Addmm` stands for matrix multiplication (`mm`) followed by an addition (`Add`).

If we just want to use a network without training or backpropagation, for

example, if we use it for prediction after training, constructing this computational graph for backpropagation can be wasteful as it performs unnecessary computations and consumes additional memory. So, when we use a model for inference (for instance, making predictions) rather than training, it is a best practice to use the `torch.no_grad()` context manager, as shown below. This tells PyTorch that it doesn't need to keep track of the gradients, which can result in significant savings in memory and computation.

```
with torch.no_grad():
    out = model(X)
print(out)
```

The result is:

```
tensor([[-0.1262,  0.1080, -0.1792]])
```

In PyTorch, it's common practice to code models such that they return the outputs of the last layer (logits) without passing them to a nonlinear activation function. That's because PyTorch's commonly used loss functions combine the softmax (or sigmoid for binary classification) operation with the negative log-likelihood loss in a single class. The reason for this is numerical efficiency and stability. So, if we want to compute class-membership probabilities for our predictions, we have to call the `softmax` function explicitly:

```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

This prints:

```
tensor([[0.3113, 0.3934, 0.2952]]))
```

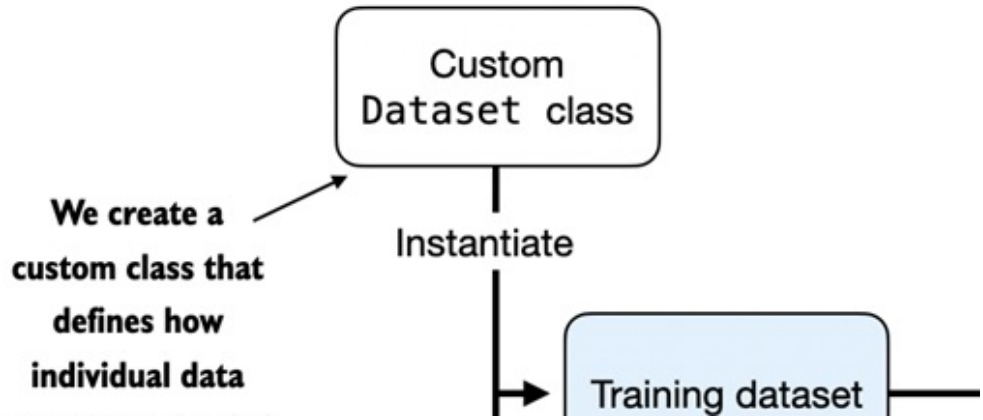
The values can now be interpreted as class-membership probabilities that sum up to 1. The values are roughly equal for this random input, which is expected for a randomly initialized model without training.

In the following two sections, we will learn how to set up an efficient data loader and train the model.

A.6 Setting up efficient data loaders

In the previous section, we defined a custom neural network model. Before we can train this model, we have to briefly talk about creating efficient data loaders in PyTorch, which we will iterate over when training the model. The overall idea behind data loading in PyTorch is illustrated in figure A.10.

Figure A.10 PyTorch implements a Dataset and a DataLoader class. The Dataset class is used to instantiate objects that define how each data record is loaded. The DataLoader handles how the data is shuffled and assembled into batches.



Following the illustration in figure A.10, in this section, we will implement a custom Dataset class that we will use to create a training and a test dataset that we'll then use to create the data loaders.

Let's start by creating a simple toy dataset of five training examples with two features each. Accompanying the training examples, we also create a tensor containing the corresponding class labels: three examples below to class 0, and two examples belong to class 1. In addition, we also make a test set consisting of two entries. The code to create this dataset is shown below.

Listing A.5 Creating a small toy dataset

```
X_train = torch.tensor([
    [-1.2, 3.1],
    [-0.9, 2.9],
    [-0.5, 2.6],
    [2.3, -1.1],
```

```
    [2.7, -1.5]
])
y_train = torch.tensor([0, 0, 0, 1, 1])

X_test = torch.tensor([
    [-0.8, 2.8],
    [2.6, -1.6],
])
y_test = torch.tensor([0, 1])
```

Class label numbering

PyTorch requires that class labels start with label 0, and the largest class label value should not exceed the number of output nodes minus 1 (since Python index counting starts at 0. So, if we have class labels 0, 1, 2, 3, and 4, the neural network output layer should consist of 5 nodes.

Next, we create a custom dataset class, `ToyDataset`, by subclassing from PyTorch's `Dataset` parent class, as shown below.

Listing A.6 Defining a custom Dataset class

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):     #A
        one_x = self.features[index]  #A
        one_y = self.labels[index]    #A
        return one_x, one_y           #A

    def __len__(self):
        return self.labels.shape[0]   #B

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

This custom `ToyDataset` class's purpose is to use it to instantiate a PyTorch `DataLoader`. But before we get to this step, let's briefly go over the general structure of the `ToyDataset` code.

In PyTorch, the three main components of a custom Dataset class are the `__init__` constructor, the `__getitem__` method, and the `__len__` method, as shown in code listing A.6 above.

In the `__init__` method, we set up attributes that we can access later in the `__getitem__` and `__len__` methods. This could be file paths, file objects, database connectors, and so on. Since we created a tensor dataset that sits in memory, we are simply assigning `X` and `y` to these attributes, which are placeholders for our tensor objects.

In the `__getitem__` method, we define instructions for returning exactly one item from the dataset via an `index`. This means the features and the class label corresponding to a single training example or test instance. (The data loader will provide this `index`, which we will cover shortly.)

Finally, the `__len__` method constrains instructions for retrieving the length of the dataset. Here, we use the `.shape` attribute of a tensor to return the number of rows in the feature array. In the case of the training dataset, we have five rows, which we can double-check as follows:

```
print(len(train_ds))
```

The result is:

```
5
```

Now that we defined a PyTorch Dataset class we can use for our toy dataset, we can use PyTorch's `DataLoader` class to sample from it, as shown in the code listing below:

Listing A.7 Instantiating data loaders

```
from torch.utils.data import DataLoader


torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,  #A
    batch_size=2,
```

```
    shuffle=True,  #B
    num_workers=0  #C
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,  #D
    num_workers=0
)
```

After instantiating the training data loader, we can iterate over it as shown below. (The iteration over the `test_loader` works similarly but is omitted for brevity.)

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

The result is:

```
Batch 1: tensor([[-1.2000,  3.1000],
                 [-0.5000,  2.6000]]) tensor([0, 0])
Batch 2: tensor([[ 2.3000, -1.1000],
                 [-0.9000,  2.9000]]) tensor([1, 0])
Batch 3: tensor([[ 2.7000, -1.5000]]) tensor([1])
```

As we can see based on the output above, the `train_loader` iterates over the training dataset visiting each training example exactly once. This is known as a training epoch. Since we seeded the random number generator using `torch.manual_seed(123)` above, you should get the exact same shuffling order of training examples as shown above. However if you iterate over the dataset a second time, you will see that the shuffling order will change. This is desired to prevent deep neural networks getting caught in repetitive update cycles during training.

Note that we specified a batch size of 2 above, but the 3rd batch only contains a single example. That's because we have five training examples, which is not evenly divisible by 2. In practice, having a substantially smaller batch as the last batch in a training epoch can disturb the convergence during training. To prevent this, it's recommended to set `drop_last=True`, which will drop the last batch in each epoch, as shown below:

Listing A.8 A training loader that drops the last batch

```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

Now, iterating over the training loader, we can see that the last batch is omitted:

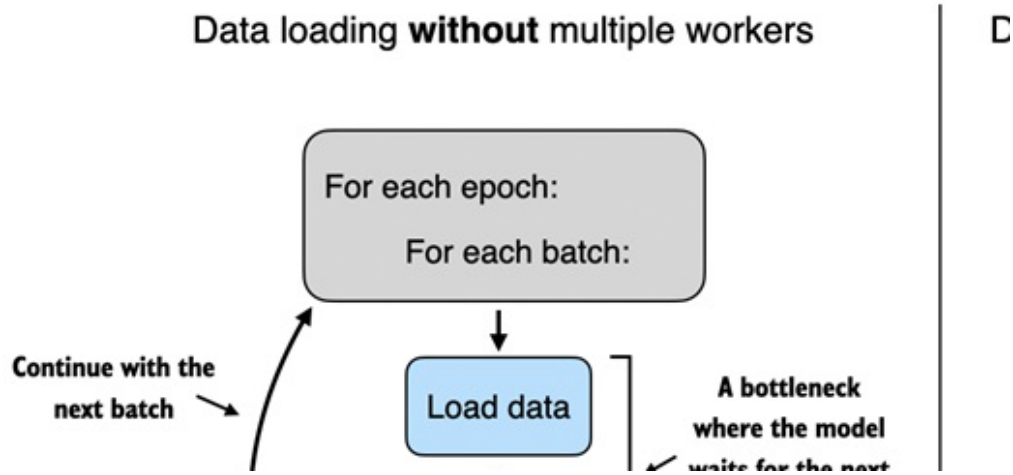
```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

The result is:

```
Batch 1: tensor([[-0.9000,  2.9000],
        [ 2.3000, -1.1000]]) tensor([0, 1])
Batch 2: tensor([[ 2.7000, -1.5000],
        [-0.5000,  2.6000]]) tensor([1, 0])
```

Lastly, let's discuss the setting `num_workers=0` in the `DataLoader`. This parameter in PyTorch's `DataLoader` function is crucial for parallelizing data loading and preprocessing. When `num_workers` is set to 0, the data loading will be done in the main process and not in separate worker processes. This might seem unproblematic, but it can lead to significant slowdowns during model training when we train larger networks on a GPU. This is because instead of focusing solely on the processing of the deep learning model, the CPU must also take time to load and preprocess the data. As a result, the GPU can sit idle while waiting for the CPU to finish these tasks. In contrast, when `num_workers` is set to a number greater than zero, multiple worker processes are launched to load data in parallel, freeing the main process to focus on training your model and better utilizing your system's resources, which is illustrated in figure A.11

Figure A.11 Loading data without multiple workers (setting `num_workers=0`) will create a data loading bottleneck where the model sits idle until the next batch is loaded as illustrated in the left subpanel. If multiple workers are enabled, the data loader can already queue up the next batch in the background as shown in the right subpanel.



However, if we are working with very small datasets, setting `num_workers` to 1 or larger may not be necessary since the total training time takes only fractions of a second anyway. On the contrary, if you are working with tiny datasets or interactive environments such as Jupyter notebooks, increasing `num_workers` may not provide any noticeable speedup. They might, in fact, lead to some issues. One potential issue is the overhead of spinning up multiple worker processes, which could take longer than the actual data loading when your dataset is small.

Furthermore, for Jupyter notebooks, setting `num_workers` to greater than 0 can sometimes lead to issues related to the sharing of resources between different processes, resulting in errors or notebook crashes. Therefore, it's essential to understand the trade-off and make a calculated decision on setting the `num_workers` parameter. When used correctly, it can be a beneficial tool but should be adapted to your specific dataset size and computational environment for optimal results.

In my experience, setting `num_workers=4` usually leads to optimal performance on many real-world datasets, but optimal settings depend on your hardware and the code used for loading a training example defined in the `Dataset` class.

A.7 A typical training loop

So far, we've discussed all the requirements for training neural networks:

PyTorch's tensor library, autograd, the `Module` API, and efficient data loaders. Let's now combine all these things and train a neural network on the toy dataset from the previous section. The training code is shown in code listing A.9 below.

Listing A.9 Neural network training in PyTorch

```
import torch.nn.functional as F


torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)  #A
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)  #B

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):

        logits = model(features)

        loss = F.cross_entropy(logits, labels)

        optimizer.zero_grad()   #C
        loss.backward()         #D
        optimizer.step()        #E

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train Loss: {loss:.2f}")

    model.eval()
    # Optional model evaluation
```

Running the code in listing A.9 above yields the following outputs:

```
Epoch: 001/003 | Batch 000/002 | Train Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Trainl Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train Loss: 0.00
```

As we can see, the loss reaches zero after 3 epochs, a sign that the model converged on the training set. However, before we evaluate the model's predictions, let's go over some of the details of the preceding code listing.

First, note that we initialized a model with two inputs and two outputs. That's because the toy dataset from the previous section has two input features and two class labels to predict. We used a stochastic gradient descent (`SGD`) optimizer with a learning rate (`lr`) of 0.5. The learning rate is a hyperparameter, meaning it's a tunable setting that we have to experiment with based on observing the loss. Ideally, we want to choose a learning rate such that the loss converges after a certain number of epochs -- the number of epochs is another hyperparameter to choose.

Exercise A.3

How many parameters does the neural network introduced at the beginning of this section have?

In practice, we often use a third dataset, a so-called validation dataset, to find the optimal hyperparameter settings. A validation dataset is similar to a test set. However, while we only want to use a test set precisely once to avoid biasing the evaluation, we usually use the validation set multiple times to tweak the model settings.

We also introduced new settings called `model.train()` and `model.eval()`. As these names imply, these settings are used to put the model into a training and an evaluation mode. This is necessary for components that behave differently during training and inference, such as *dropout* or *batch normalization* layers. Since we don't have dropout or other components in our `NeuralNetwork` class that are affected by these settings, using `model.train()` and `model.eval()` is redundant in our code above. However, it's best practice to include them anyway to avoid unexpected behaviors when we change the model architecture or reuse the code to train a different model.

As discussed earlier, we pass the logits directly into the `cross_entropy` loss function, which will apply the softmax function internally for efficiency and numerical stability reasons. Then, calling `loss.backward()` will calculate the gradients in the computation graph that PyTorch constructed in the

background. The `optimizer.step()` method will use the gradients to update the model parameters to minimize the loss. In the case of the SGD optimizer, this means multiplying the gradients with the learning rate and adding the scaled negative gradient to the parameters.

Preventing undesired gradient accumulation

It is important to include an `optimizer.zero_grad()` call in each update round to reset the gradients to zero. Otherwise, the gradients will accumulate, which may be undesired.

After we trained the model, we can use it to make predictions, as shown below:

```
model.eval()
with torch.no_grad():
    outputs = model(X_train)
print(outputs)
```

The results are as follows:

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]])
```

To obtain the class membership probabilities, we can then use PyTorch's softmax function, as follows:

```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

This outputs:

```
tensor([[    0.9991,     0.0009],
        [    0.9982,     0.0018],
        [    0.9949,     0.0051],
        [    0.0491,     0.9509],
        [    0.0307,     0.9693]])
```

Let's consider the first row in the code output above. Here, the first value (column) means that the training example has a 99.91% probability of belonging to class 0 and a 0.09% probability of belonging to class 1. (The `set_printoptions` call is used here to make the outputs more legible.)

We can convert these values into class labels predictions using PyTorch's argmax function, which returns the index position of the highest value in each row if we set `dim=1` (setting `dim=0` would return the highest value in each column, instead):

```
predictions = torch.argmax(probas, dim=1)
print(predictions)
```

This prints:

```
tensor([0, 0, 0, 1, 1])
```

Note that it is unnecessary to compute softmax probabilities to obtain the class labels. We could also apply the `argmax` function to the logits (outputs) directly:

```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

The output is:

```
tensor([0, 0, 0, 1, 1])
```

Above, we computed the predicted labels for the training dataset. Since the training dataset is relatively small, we could compare it to the true training labels by eye and see that the model is 100% correct. We can double-check this using the == comparison operator:

```
predictions == y_train
```

The results are:

```
tensor([True, True, True, True, True])
```

Using `torch.sum`, we can count the number of correct prediction as follows:

```
torch.sum(predictions == y_train)
```

The output is:

```
5
```

Since the dataset consists of 5 training examples, we have 5 out of 5 predictions that are correct, which equals 5/5 × 100% = 100% prediction accuracy.

However, to generalize the computation of the prediction accuracy, let's implement a `compute_accuracy` function as shown in the following code listing.

Listing A.10 A function to compute the prediction accuracy

```
def compute_accuracy(model, dataloader):

    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):

        with torch.no_grad():
            logits = model(features)

        predictions = torch.argmax(logits, dim=1)
        compare = labels == predictions  #A
        correct += torch.sum(compare)  #B
        total_examples += len(compare)

    return (correct / total_examples).item() #C
```

Note that the following code listing iterates over a data loader to compute the number and fraction of the correct predictions. This is because when we work with large datasets, we typically can only call the model on a small part of the dataset due to memory limitations. The `compute_accuracy` function above is a general method that scales to datasets of arbitrary size since, in each iteration, the dataset chunk that the model receives is the same size as the batch size seen during training.

Notice that the internals of the `compute_accuracy` function are similar to what we used before when we converted the logits to the class labels.

We can then apply the function to the training as follows:

```
print(compute_accuracy(model, train_loader))
```

The results is:

```
1.0
```

Similarly, we can apply the function to the test set as follows:

```
>>> print(compute_accuracy(model, test_loader))
```

This prints:

```
1.0
```

In this section, we learned how we can train a neural network using PyTorch. Next, let's see how we can save and restore models after training.

A.8 Saving and loading models

In the previous section, we successfully trained a model. Let's now see how we can save a trained model to reuse it later.

Here's the recommended way how we can save and load models in PyTorch:

```
torch.save(model.state_dict(), "model.pth")
```

The model's state_dict is a Python dictionary object that maps each layer in the model to its trainable parameters (weights and biases). Note that `"model.pth"` is an arbitrary filename for the model file saved to disk. We can give it any name and file ending we like; however, `.pth` and `.pt` are the most common conventions.

Once we saved the model, we can restore it from disk as follows:

```
model = NeuralNetwork(2, 2)
```

```
model.load_state_dict(torch.load("model.pth"))
```

The `torch.load("model.pth")` function reads the file `"model.pth"` and reconstructs the Python dictionary object containing the model's parameters while `model.load_state_dict()` applies these parameters to the model, effectively restoring its learned state from when we saved it.

Note that the line `model = NeuralNetwork(2, 2)` above is not strictly necessary if you execute this code in the same session where you saved a model. However, I included it here to illustrate that we need an instance of the model in memory to apply the saved parameters. Here, the `NeuralNetwork(2, 2)` architecture needs to match the original saved model exactly.

Now, we are well equipped to use PyTorch to implement large language models in the upcoming chapters. However, before we jump to the next chapter, the last section will show you how to train PyTorch models faster using one or more GPUs (if available).

A.9 Optimizing training performance with GPUs

In this last section of this chapter, we will see how we can utilize GPUs, which will accelerate deep neural network training compared to regular CPUs. First, we will introduce the main concepts behind GPU computing in PyTorch. Then, we will train a model on a single GPU. Finally, we'll then look at distributed training using multiple GPUs.

A.9.1 PyTorch computations on GPU devices

As you will see, modifying the training loop from section 2.7 to optionally run on a GPU is relatively simple and only requires changing three lines of code.

Before we make the modifications, it's crucial to understand the main concept behind GPU computations within PyTorch. First, we need to introduce the notion of devices. In PyTorch, a device is where computations occur, and data resides. The CPU and the GPU are examples of devices. A PyTorch

tensor resides in a device, and its operations are executed on the same device.

Let's see how this works in action. Assuming that you installed a GPU-compatible version of PyTorch as explained in section 2.1.3, Installing PyTorch, we can double-check that our runtime indeed supports GPU computing via the following code:

```
print(torch.cuda.is_available())
```

The result is:

```
True
```

Now, suppose we have two tensors that we can add as follows -- this computation will be carried out on the CPU by default:

```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])
print(tensor_1 + tensor_2)
```

This outputs:

```
tensor([5., 7., 9.])
```

We can now use the `.to()` method[1] to transfer these tensors onto a GPU and perform the addition there:

```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")
print(tensor_1 + tensor_2)
```

The output is as follows:

```
tensor([5., 7., 9.], device='cuda:0')
```

Notice that the resulting tensor now includes the device information, `device='cuda:0'`, which means that the tensors reside on the first GPU. If your machine hosts multiple GPUs, you have the option to specify which GPU you'd like to transfer the tensors to. You can do this by indicating the device ID in the transfer command. For instance, you can use `.to("cuda:0")`, `.to("cuda:1")`, and so on.

However, it is important to note that all tensors must be on the same device. Otherwise, the computation will fail, as shown below, where one tensor resides on the CPU and the other on the GPU:

```
tensor_1 = tensor_1.to("cpu")
print(tensor_1 + tensor_2)
```

This results in the following:

```
RuntimeError      Traceback (most recent call last)
<ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
      1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)
RuntimeError: Expected all tensors to be on the same device, but
```

In this section, we learned that GPU computations on PyTorch are relatively straightforward. All we have to do is transfer the tensors onto the same GPU device, and PyTorch will handle the rest. Equipped with this information, we can now train the neural network from the previous section on a GPU.

A.9.2 Single-GPU training

Now that we are familiar with transferring tensors to the GPU, we can modify the training loop from *section 2.7, A typical training loop*, to run on a GPU. This requires only changing three lines of code, as shown in code listing A.11 below.

Listing A.11 A training loop on a GPU

```
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

device = torch.device("cuda")   #A
model = model.to(device)   #B

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
```

```
    for batch_idx, (features, labels) in enumerate(train_loader):

        features, labels = features.to(device), labels.to(device)
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    # Optional model evaluation
```

Running the above code will output the following, similar to the results obtained on the CPU previously in section 2.7:

```
Epoch: 001/003 | Batch 000/002 | Train/Val Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train/Val Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train/Val Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.00
```

We can also use `.to("cuda")` instead of `device = torch.device("cuda")`. As we saw in section 2.9.1, transferring a tensor to "cuda" instead of `torch.device("cuda")` works as well and is shorter. We can also modify the statement to the following, which will make the same code executable on a CPU if a GPU is not available, which is usually considered best practice when sharing PyTorch code:

```
device = torch.device("cuda" if torch.cuda.is_available() else "c
```

In the case of the modified training loop above, we probably won't see a speed-up because of the memory transfer cost from CPU to GPU. However, we can expect a significant speed-up when training deep neural networks, especially large language models.

As we saw in this section, training a model on a single GPU in PyTorch is relatively easy. Next, let's introduce another concept: training models on multiple GPUs.

PyTorch on macOS

On an Apple Mac with an Apple Silicon chip (like the M1, M2, M3, or newer models) instead of a computer with an Nvidia GPU, you can change

```
device = torch.device("cuda" if torch.cuda.is_available() else "c
```

to

```
device = torch.device("mps" if torch.backends.mps.is_available()
```

to take advantage of this chip.

Exercise A.4

Compare the runtime of matrix multiplication on a CPU to a GPU. At what matrix size do you begin to see the matrix multiplication on the GPU being faster than on the CPU? Hint: I recommend using the `%timeit` command in Jupyter to compare the runtime. For example, given matrices `a` and `b`, run the command `%timeit a @ b` in a new notebook cell.

A.9.3 Training with multiple GPUs

In this section, we will briefly go over the concept of distributed training. Distributed training is the concept of dividing the model training across multiple GPUs and machines.

Why do we need this? Even when it is possible to train a model on a single GPU or machine, the process could be exceedingly time-consuming. The training time can be significantly reduced by distributing the training process across multiple machines, each with potentially multiple GPUs. This is particularly crucial in the experimental stages of model development, where numerous training iterations might be necessary to finetune the model parameters and architecture.

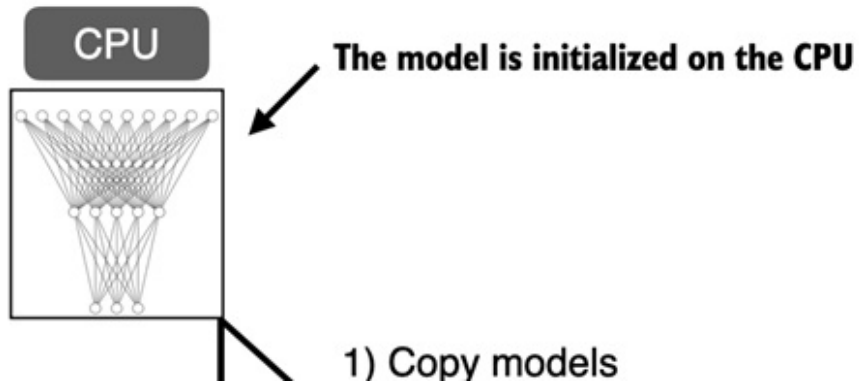
Multi-GPU computing is optional

For this book, it is not required to have access to or use multiple-GPU. This section is included for those who are interested in how multi-GPU computing works in PyTorch.

In this section, we will look at the most basic case of distributed training: PyTorch's `DistributedDataParallel` (DDP) strategy. DDP enables parallelism by splitting the input data across the available devices and processing these data subsets simultaneously.

How does this work? PyTorch launches a separate process on each GPU, and each process receives and keeps a copy of the model -- these copies will be synchronized during training. To illustrate this, suppose we have two GPUs that we want to use to train a neural network, as shown in figure A.12.

Figure A.12 The model and data transfer in DDP involves two key steps. First, we create a copy of the model on each of the GPUs. Then we divide the input data into unique minibatches that we pass on to each model copy.

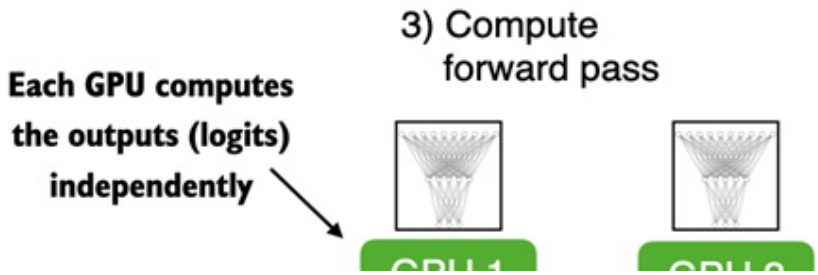


Each of the two GPUs will receive a copy of the model. Then, in every training iteration, each model will receive a minibatch (or just batch) from the data loader. We can use a `DistributedSampler` to ensure that each GPU will receive a different, non-overlapping batch when using DDP.

Since each model copy will see a different sample of the training data, the

model copies will return different logits as outputs and compute different gradients during the backward pass. These gradients are then averaged and synchronized during training to update the models. This way, we ensure that the models don't diverge, as illustrated in figure A.13.

Figure A.13 The forward and backward pass in DDP are executed independently on each GPU with its corresponding data subset. Once the forward and backward passes are completed, gradients from each model replica (on each GPU) are synchronized across all GPUs. This ensures that every model replica has the same updated weights.



The benefit of using DDP is the enhanced speed it offers for processing the dataset compared to a single GPU. Barring a minor communication overhead between devices that comes with DDP use, it can theoretically process a training epoch in half the time with two GPUs compared to just one. The time efficiency scales up with the number of GPUs, allowing us to process an epoch eight times faster if we have eight GPUs, and so on.

Multi-GPU computing in interactive environments

DDP does not function properly within interactive Python environments like Jupyter notebooks, which don't handle multiprocessing in the same way a standalone Python script does. Therefore, the following code should be executed as a script, not within a notebook interface like Jupyter. This is because DDP needs to spawn multiple processes, and each process should have its own Python interpreter instance.

Let's now see how this works in practice. For brevity, we will only focus on the core parts of the previous code that need to be adjusted for DDP training. However, for readers who want to run the code on their own multi-GPU

machine or a cloud instance of their choice, it is recommended to use the standalone script provided in this book's GitHub repository at https://github.com/rasbt/LLMs-from-scratch.

First, we will import a few additional submodules, classes, and functions for distributed training PyTorch as shown in code listing A.13 below.

Listing A.12 PyTorch utilities for distributed training

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process
```

Before we dive deeper into the changes to make the training compatible with DDP, let's briefly go over the rationale and usage for these newly imported utilities that we need alongside the `DistributedDataParallel` class.

PyTorch's `multiprocessing` submodule contains functions such as `multiprocessing.spawn`, which we will use to spawn multiple processes and apply a function to multiple inputs in parallel. We will use it to spawn one training process per GPU.

If we spawn multiple processes for training, we will need a way to divide the dataset among these different processes. For this, we will use the `DistributedSampler`.

The `init_process_group` and `destroy_process_group` are used to initialize and quit the distributed training mods. The `init_process_group` function should be called at the beginning of the training script to initialize a process group for each process in the distributed setup, and `destroy_process_group` should be called at the end of the training script to destroy a given process group and release its resources.

The following code in listing A.13 below illustrates how these new components are used to implement DDP training for the `NeuralNetwork` model we implemented earlier.

Listing A.13 Model training with DistributedDataParallel strategy

```
def ddp_setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost"   #A
    os.environ["MASTER_PORT"] = "12345"       #B
    init_process_group(
        backend="nccl",                       #C
        rank=rank,                            #D
        world_size=world_size                 #E
    )
    torch.cuda.set_device(rank)               #F
def prepare_dataset():
    ...
    train_loader = DataLoader(
        dataset=train_ds,
        batch_size=2,
        shuffle=False,                        #G
        pin_memory=True,                      #H
        drop_last=True,
        sampler=DistributedSampler(train_ds)  #I
    )
    return train_loader, test_loader
def main(rank, world_size, num_epochs):       #J
    ddp_setup(rank, world_size)
    train_loader, test_loader = prepare_dataset()
    model = NeuralNetwork(num_inputs=2, num_outputs=2)
    model.to(rank)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
    model = DDP(model, device_ids=[rank])
    for epoch in range(num_epochs):
    for features, labels in train_loader:
            features, labels = features.to(rank), labels.to(rank)
            ...
            print(f"[GPU{rank}] Epoch: {epoch+1:03d}/{num_epochs:
                  f" | Batchsize {labels.shape[0]:03d}"
                  f" | Train/Val Loss: {loss:.2f}")
    model.eval()
    train_acc = compute_accuracy(model, train_loader, device=rank
    print(f"[GPU{rank}] Training accuracy", train_acc)
    test_acc = compute_accuracy(model, test_loader, device=rank)
    print(f"[GPU{rank}] Test accuracy", test_acc)
    destroy_process_group()                    #L

if __name__ == "__main__":
    print("Number of GPUs available:", torch.cuda.device_count())
    torch.manual_seed(123)
    num_epochs = 3
    world_size = torch.cuda.device_count()
    mp.spawn(main, args=(world_size, num_epochs), nprocs=world_si
```

Before we run the code from listing A.13, here is a summary of how it works, in addition to the annotations above. We have a `__name__ == "__main__"` clause at the bottom containing code that is executed when we run the code as a Python script instead of importing it as a module. This code first prints the number of available GPUs using `torch.cuda.device_count()`, sets a random seed for reproducibility and then spawns new processes using PyTorch's `multiprocesses.spawn` function. Here, the `spawn` function launches one process per GPU setting `nproces=world_size`, where the world size is the number of available GPUs. This `spawn` function launches the code in the `main` function we define in the same script with some additional arguments provided via `args`. Note that the `main` function has a `rank` argument that we don't include in the `mp.spawn()` call. That's because the `rank`, which refers to the process ID we use as the GPU ID, is already passed automatically.

The `main` function sets up the distributed environment via `ddp_setup` -- another function we defined, loads the training and test sets, sets up the model, and carries out the training. Compared to the single-GPU training in section 2.12, we now transfer the model and data to the target device via `.to(rank)`, which we use to refer to the GPU device ID. Also, we wrap the model via `DDP`, which enables the synchronization of the gradients between the different GPUs during training. After the training finishes and we evaluate the models, we use `destroy_process_group()` to cleanly exit the distributed training and free up the allocated resources.

Earlier, we mentioned that each GPU will receive a different subsample of the training data. To ensure this, we set `sampler=DistributedSampler(train_ds)` in the training loader.

The last function to discuss is `ddp_setup`. It sets the main node's address and port to allow for communication between the different processes, initializes the process group with the NCCL backend (designed for GPU-to-GPU communication), and sets the `rank` (process identifier) and world size (total number of processes). Finally, it specifies the GPU device corresponding to the current model training process rank.

Selecting available GPUs on a multi-GPU machine

If you wish to restrict the number of GPUs used for training on a multi-GPU machine, the simplest way is to use the `CUDA_VISIBLE_DEVICES` environment variable. To illustrate this, suppose your machine has multiple GPUs, and you only want to use one GPU, for example, the GPU with index 0. Instead of `python some_script.py`, you can run the code from the terminal as follows:

```
CUDA_VISIBLE_DEVICES=0 python some_script.py
```

Or, if your machine has four GPUs and you only want to use the first and third GPU, you can use

```
CUDA_VISIBLE_DEVICES=0,2 python some_script.py
```

Setting `CUDA_VISIBLE_DEVICES` in this way is a simple and effective way to manage GPU allocation without modifying your PyTorch scripts.

Let's now run this code and see how it works in practice by launching the code as a script from the terminal:

```
python ch02-DDP-script.py
```

Note that it should work on both single- and multi-GPU machines. If we run this code on a single GPU, we should see the following output:

```
PyTorch version: 2.0.1+cu117
CUDA available: True
Number of GPUs available: 1
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.62
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.32
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.11
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.07
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.02
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.03
[GPU0] Training accuracy 1.0
[GPU0] Test accuracy 1.0
```

The code output looks similar to the one in section 2.9.2, which is a good sanity check.

Now, if we run the same command and code on a machine with two GPUs,

we should see the following:

```
PyTorch version: 2.0.1+cu117
CUDA available: True
Number of GPUs available: 2
[GPU1] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.60
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.59
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.16
[GPU1] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.17
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Training accuracy 1.0
[GPU0] Training accuracy 1.0
[GPU1] Test accuracy 1.0
[GPU0] Test accuracy 1.0
```

As expected, we can see that some batches are processed on the first GPU (`GPU0`) and others on the second (`GPU1`). However, we see duplicated output lines when printing the training and test accuracies. This is because each process (in other words, each GPU) prints the test accuracy independently. Since DDP replicates the model onto each GPU and each process runs independently, if you have a print statement inside your testing loop, each process will execute it, leading to repeated output lines.

If this bothers you, you can fix this using the rank of each process to control your print statements.

if rank == 0: # only print in the first process

print("Test accuracy: ", accuracy)

This is, in a nutshell, how distributed training via DDP works. If you are interested in additional details, I recommend checking the official API documentation at https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataPara

Alternative PyTorch APIs for multi-GPU training

If you prefer more straightforward ways to use multiple GPUs in PyTorch, you can also consider add-on APIs like the open-source Fabric library, which I've written about in Accelerating PyTorch Model Training: Using Mixed-

Precision and Fully Sharded Data Parallelism
https://magazine.sebastianraschka.com/p/accelerating-pytorch-model-training.

A.10 Summary

- PyTorch is an open-source library that consists of three core components: a tensor library, automatic differentiation functions, and deep learning utilities.
- PyTorch's tensor library is similar to array libraries like NumPy
- In the context of PyTorch, tensors are array-like data structures to represent scalars, vectors, matrices, and higher-dimensional arrays.
- PyTorch tensors can be executed on the CPU, but one major advantage of PyTorch's tensor format is its GPU support to accelerate computations.
- The automatic differentiation (autograd) capabilities in PyTorch allow us to conveniently train neural networks using backpropagation without manually deriving gradients.
- The deep learning utilities in PyTorch provide building blocks for creating custom deep neural networks.
- PyTorch includes `Dataset` and `DataLoader` classes to set up efficient data loading pipelines.
- It's easiest to train models on a CPU or single GPU.
- Using `DistributedDataParallel` is the simplest way in PyTorch to accelerate the training if multiple GPUs are available.

A.11 Further reading

While this chapter should be sufficient to get you up to speed, in addition, 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 min 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://sebastianraschka.com/pdf/supplementary/calculus.pdf

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://sebastianraschka.com/blog/2023/llm-grad-accumulation.html

This chapter covered 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://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

A.12 Exercise answers

Exercise A.3:

The network has 2 inputs and 2 outputs. In addition, there are 2 hidden layers with 30 and 20 nodes, respectively. Programmatically, we can calculate the number of parameters as follows:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requi
print("Total number of trainable model parameters:", num_params)
```

This returns:

```
752
```

We can also calculate this manually as follows:

- first hidden layer: 2 inputs times 30 hidden units plus 30 bias units.
- second hidden layer: 30 incoming units times 20 nodes plus 20 bias units.
- output layer: 20 incoming nodes times 2 output nodes plus 2 bias units.

Then, adding all the parameters in each layer results in $2 \times 30+30 + 30 \times 20+20 + 20 \times 2+2 = 752$.

Exercise A.4:

The exact runtime results will be specific to the hardware used for this experiment. In my experiments, I observed significant speed-ups even for small matrix multiplications as the following one when using a Google Colab instance connected to a V100 GPU:

```
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b
```

On the CPU this resulted in:

```
63.8 µs ± 8.7 µs per loop
```

When executed on a GPU:

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

The result was:

```
13.8 µs ± 425 ns per loop
```

In this case, on a V100, the computation was approximately four times faster.

[1] This is the same .to() method we previously used to change a tensor's datatype in section 2.2.2, Tensor data types.

Appendix B. References and Further Reading

B.1 Chapter 1

Custom-built LLMs are able to outperform general-purpose LLMs as a team at Bloomberg showed via a version of GPT pretrained on finance data from scratch. The custom LLM outperformed ChatGPT on financial tasks while maintaining good performance on general LLM benchmarks:

- *BloombergGPT: A Large Language Model for Finance* (2023) by Wu *et al.*, https://arxiv.org/abs/2303.17564

Existing LLMs can be adapted and finetuned to outperform general LLMs as well, which teams from Google Research and Google DeepMind showed in a medical context:

- *Towards Expert-Level Medical Question Answering with Large Language Models* (2023) by Singhal *et al.*, https://arxiv.org/abs/2305.09617

The paper that proposed the original transformer architecture:

- *Attention Is All You Need* (2017) by Vaswani *et al.*, https://arxiv.org/abs/1706.03762

The original encoder-style transformer, called BERT:

- *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* (2018) by Devlin *et al.*, https://arxiv.org/abs/1810.04805

The paper describing the decoder-style GPT-3 model, which inspired modern LLMs and will be used as a template for implementing an LLM from scratch in this book:

- *Language Models are Few-Shot Learners* (2020) by Brown *et al.*, https://arxiv.org/abs/2005.14165

The original vision transformer for classifying images, which illustrates that transformer architectures are not only restricted to text inputs:

- *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* (2020) by Dosovitskiy *et al.*, https://arxiv.org/abs/2010.11929

Two experimental (but less popular) LLM architectures that serve as examples that not all LLMs need to be based on the transformer architecture:

- *RWKV: Reinventing RNNs for the Transformer Era* (2023) by Peng *et al.*, https://arxiv.org/abs/2305.13048
- *Hyena Hierarchy: Towards Larger Convolutional Language Models (2023)* by Poli *et al.*, https://arxiv.org/abs/2302.10866
- *Mamba: Linear-Time Sequence Modeling with Selective State Spaces* (2023) by Gu and Dao, https://arxiv.org/abs/2312.00752

Meta AI's model is a popular implementation of a GPT-like model that is openly available in contrast to GPT-3 and ChatGPT:

- *Llama 2: Open Foundation and Fine-Tuned Chat Models* (2023) by Touvron *et al.*, https://arxiv.org/abs/2307.092881

For readers interested in additional details about the dataset references in section 1.5, this paper describes the publicly available *The Pile* dataset curated by Eleuther AI:

- *The Pile: An 800GB Dataset of Diverse Text for Language Modeling* (2020) by Gao *et al.*, https://arxiv.org/abs/2101.00027.

The following paper provides the reference for InstructGPT for finetuning GPT-3, which was mentioned in section 1.6 and will be discussed in more detail in chapter 7:

- *Training Language Models to Follow Instructions with Human Feedback* (2022) by *Ouyang et al.*, https://arxiv.org/abs/2203.02155

B.2 Chapter 2

Readers who are interested in discussion and comparison of embedding spaces with latent spaces and the general notion of vector representations can find more information in the first chapter of my book Machine Learning Q and AI:

- *Machine Learning Q and AI* (2023) by Sebastian Raschka, https://leanpub.com/machine-learning-q-and-ai

The following paper provides more in-depth discussions of how how byte pair encoding is used as a tokenization method:

- Neural Machine Translation of Rare Words with Subword Units (2015) by Sennrich at al., https://arxiv.org/abs/1508.07909

The code for the byte pair encoding tokenizer used to train GPT-2 was open-sourced by OpenAI:

- https://github.com/openai/gpt-2/blob/master/src/encoder.py

OpenAI provides an interactive web UI to illustrate how the byte pair tokenizer in GPT models works:

- https://platform.openai.com/tokenizer

For readers interested in coding and training a BPE tokenizer from the ground up, Andrej Karpathy's GitHub repository `minbpe` offers a minimal and readable implementation:

- A minimal implementation of a BPE tokenizer, https://github.com/karpathy/minbpe

Readers who are interested in studying alternative tokenization schemes that are used by some other popular LLMs can find more information in the SentencePiece and WordPiece papers:

- SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing (2018) by Kudo and Richardson, https://aclanthology.org/D18-2012/
- Fast WordPiece Tokenization (2020) by Song et al., https://arxiv.org/abs/2012.15524

B.3 Chapter 3

Readers interested in learning more about Bahdanau attention for RNN and language translation can find detailed insights in the following paper:

- *Neural Machine Translation by Jointly Learning to Align and Translate* (2014) by Bahdanau, Cho, and Bengio, https://arxiv.org/abs/1409.0473

The concept of self-attention as scaled dot-product attention was introduced in the original transformer paper:

- *Attention Is All You Need* (2017) by Vaswani et al., https://arxiv.org/abs/1706.03762

*FlashAttentio*n is a highly efficient implementation of self-attention mechanism, which accelerates the computation process by optimizing memory access patterns. FlashAttention is mathematically the same as the standard self-attention mechanism but optimizes the computational process for efficiency:

- *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awarenes*s (2022) by Dao *et al.*, https://arxiv.org/abs/2205.14135
- *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning* (2023) by Dao, https://arxiv.org/abs/2307.08691

PyTorch implements a function for self-attention and causal attention that supports FlashAttention for efficiency. This function is beta and subject to change:

- `scaled_dot_product_attention` documentation: https://pytorch.org/docs/stable/generated/torch.nn.functional.scaled_dot_

PyTorch also implements an efficient `MultiHeadAttention` class based on the `scaled_dot_product` function:

- `MultiHeadAttention` documentation: https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.ht

Dropout is a regularization technique used in neural networks to prevent overfitting by randomly dropping units (along with their connections) from the neural network during training:

- *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014) by Srivastava *et al.*, https://jmlr.org/papers/v15/srivastava14a.html

While using the multi-head attention based on scaled-dot product attention remains the most common variant of self-attention in practice, authors found that it's possible to also achieve good performance without the value weight matrix and projection layer:

- *Simplifying Transformer Blocks* (2023) by He and Hofmann, https://arxiv.org/abs/2311.01906

B.4 Chapter 4

The layer normalization paper, titled "Layer Normalization," introduces a technique that stabilizes the hidden state dynamics neural networks by normalizing the summed inputs to the neurons within a hidden layer, significantly reducing training time compared to previously published methods:

- *Layer Normalization* (2016) by Ba, Kiros, and Hinton, https://arxiv.org/abs/1607.06450

Post-LayerNorm, used in the original Transformer model, applies layer normalization after the self-attention and feed forward networks. In contrast, Pre-LayerNorm, as adopted in models like GPT-2 and newer LLMs, applies layer normalization before these components, which can lead to more stable training dynamics and has been shown to improve performance in some cases, as discussed in the following papers:

- *On Layer Normalization in the Transformer Architecture* (2020) by Xiong *et al.*, https://arxiv.org/abs/2002.04745
- *ResiDual: Transformer with Dual Residual Connections* (2023) by Tie *et al.*, https://arxiv.org/abs/2304.14802

A popular variant of LayerNorm used in modern LLMs is RMSNorm due to its improved computing efficiency. This variant simplifies the normalization process by normalizing the inputs using only the root mean square of the inputs, without subtracting the mean before squaring. This means it does not center the data before computing the scale. RMSNorm is described in more detail in the following paper:

- *Root Mean Square Layer Normalization* (2019) by Zhang and Sennrich, https://arxiv.org/abs/1910.07467

The GELU (Gaussian Error Linear Unit) activation function combines the properties of both the classic ReLU activation function and the normal distribution's cumulative distribution function to model layer outputs,

allowing for stochastic regularization and non-linearities in deep learning models, as introduced in the following paper:

- *Gaussian Error Linear Units (GELUs)* (2016) by Hendricks and Gimpel, https://arxiv.org/abs/1606.08415

The GPT-2 paper introduced a series of transformer-based LLMs with varying sizes—124M, 355M, 774M, and 1.5B parameters:

- *Language Models are Unsupervised Multitask Learners* (2019) by Radford *et al.*, https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

OpenAI's GPT-3 uses fundamentally the same architecture as GPT-2, except that the largest version (175 billion) is 100x larger than the largest GPT-2 model and has been trained on much more data. Interested readers can refer to the official GPT-3 paper by OpenAI and the technical overview by Lambda Labs, which calculates that training GPT-3 on a single RTX 8000 consumer GPU would take 665 years:

- Language Models are Few-Shot Learners (2023) by Brown et al., https://arxiv.org/abs/2005.14165
- OpenAI's GPT-3 Language Model: A Technical Overview, https://lambdalabs.com/blog/demystifying-gpt-3

NanoGPT is a code repository with a minimalist yet efficient implementation of a GPT-2 model, similar to the model implemented in this book. While the code in this book is different from nanoGPT, this repository inspired the reorganization of a large GPT Python parent class implementation into smaller submodules:

- NanoGPT, a repository for training medium-sized GPTs, https://github.com/karpathy/nanoGPT

An informative blog post showing that most of the computation in LLMs is spent in the feed forward layers rather than attention layers when the context size is smaller than 32,000 tokens:

- "In the long (context) run" by Harm de Vries, https://www.harmdevries.com/post/context-length/

B.5 Chapter 5

A video lecture by the author detailing the loss function and applying a log transformation to make it easier to handle for mathematical optimization:

- L8.2 Logistic Regression Loss Function, https://www.youtube.com/watch?v=GxJe0DZvydM

The following two papers detail the dataset, hyperparameter, and architecture details used for pretraining LLMs:

- Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling (2023) by Biderman *et al.*, https://arxiv.org/abs/2304.01373
- OLMo: Accelerating the Science of Language Models (2024) by Groeneveld *et al.*, https://arxiv.org/abs/2402.00838

The following supplementary code available for this book contains instructions for preparing 60,000 public domain books from Project Gutenberg for LLM training:

- Pretraining GPT on the Project Gutenberg Dataset, https://github.com/rasbt/LLMs-from-scratch/tree/main/ch05/03_bonus_pretraining_on_gutenberg

Chapter 5 discusses the pretraining of LLMs, and Appendix D covers more advanced training functions, such as linear warmup and cosine annealing. The following paper finds that similar techniques can be successfully applied to continue pretraining already pretrained LLMs, along with additional tips and insights:

- Simple and Scalable Strategies to Continually Pre-train Large Language Models (2024) by Ibrahim *et al.*, https://arxiv.org/abs/2403.08763

BloombergGPT is an example of a domain-specific large language model (LLM) created by training on both general and domain-specific text corpora, specifically in the field of finance:

- BloombergGPT: A Large Language Model for Finance (2023) by Wu *et al.*, https://arxiv.org/abs/2303.17564

GaLore is a recent research project that aims to make LLM pretraining more efficient. The required code change boils down to just replacing PyTorch's `AdamW` optimizer in the training function with the `GaLoreAdamW` optimizer provided by the `galore-torch` Python package.

- GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection (2024) by Zhao *et al.*, https://arxiv.org/abs/2403.03507
- GaLore code repository, https://github.com/jiaweizzhao/GaLore

The following papers and resources share openly available, large-scale pretraining datasets for LLMs that consist of hundreds of gigabytes to terabytes of text data:

- Dolma: an Open Corpus of Three Trillion Tokens for LLM Pretraining Research by Soldaini *et al.* 2024, https://arxiv.org/abs/2402.00159
- The Pile: An 800GB Dataset of Diverse Text for Language Modeling by Gao et al. 2020, https://arxiv.org/abs/2101.00027
- The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only, by Penedo *et al.* (2023) https://arxiv.org/abs/2306.01116
- RedPajama by Together AI, https://github.com/togethercomputer/RedPajama-Data

The paper that originally introduced top-k sampling:

- Hierarchical Neural Story Generation by Fan *et al.* (2018), https://arxiv.org/abs/1805.04833

Beam search (not cover in chapter 5) is an alternative decoding algorithm that generates output sequences by keeping only the top-scoring partial sequences at each step to balance efficiency and quality:

- Diverse Beam Sea*rch: Decoding Diverse Solutions from Neural Sequence Models by Vijayakumar* et al. (2016), https://arxiv.org/abs/1610.02424

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.

C.1 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'
```

Exercise 2.2

The code for the data loader with `max_length=2 and stride=2`:

```
dataloader = create_dataloader(raw_text, batch_size=4, max_length
```

It produces batches of the following format:

```
tensor([[  40,  367],
        [2885, 1464],
        [1807, 3619],
        [ 402,  271]])
```

The code of the second data loader with `max_length=8 and stride=2`:

```
dataloader = create_dataloader(raw_text, batch_size=4, max_length
```

An example batch looks like as follows:

```
tensor([[   40,   367,  2885,  1464,  1807,  3619,   402,   271],
        [ 2885,  1464,  1807,  3619,   402,   271, 10899,  2138],
        [ 1807,  3619,   402,   271, 10899,  2138,   257,  7026],
        [  402,   271, 10899,  2138,   257,  7026, 15632,   438]]
```

C.2 Chapter 3

Exercise 3.1

The correct weight assignment is as follows:

```
sa_v1.W_query = torch.nn.Parameter(sa_v2.W_query.weight.T)
sa_v1.W_key = torch.nn.Parameter(sa_v2.W_key.weight.T)
sa_v1.W_value = torch.nn.Parameter(sa_v2.W_value.weight.T)
```

Exercise 3.2

To achieve an output dimension of 2, similar to what we had in single-head attention, we need to change the projection dimension `d_out` to 1.

```
d_out = 1
mha = MultiHeadAttentionWrapper(d_in, d_out, block_size, 0.0, num
```

Exercise 3.3

The initialization for the smallest GPT-2 model is as follows:

```
block_size = 1024
d_in, d_out = 768, 768
num_heads = 12
mha = MultiHeadAttention(d_in, d_out, block_size, 0.0, num_heads)
```

C.3 Chapter 4

Exercise 4.1

We can calculate the number of parameters in the feed forward and attention modules as follows:

```
block = TransformerBlock(GPT_CONFIG_124M)

total_params = sum(p.numel() for p in block.ff.parameters())
print(f"Total number of parameters in feed forward module: {total

total_params = sum(p.numel() for p in block.att.parameters())
print(f"Total number of parameters in attention module: {total_pa
```

As we can see, the feed forward module contains approximately twice as many parameters as the attention module:

```
Total number of parameters in feed forward module: 4,722,432
Total number of parameters in attention module: 2,360,064
```

Exercise 4.2

To instantiate the other GPT model sizes, we can modify the configuration dictionary as follows (here shown for GPT-2 XL):

```
GPT_CONFIG = GPT_CONFIG_124M.copy()
GPT_CONFIG["emb_dim"] = 1600
GPT_CONFIG["n_layers"] = 48
GPT_CONFIG["n_heads"] = 25
model = GPTModel(GPT_CONFIG)
```

Then, reusing the code from Section 4.6 to calculate the number of parameters and RAM requirements, we find the following:

```
gpt2-xl:
Total number of parameters: 1,637,792,000
Number of trainable parameters considering weight tying: 1,557,38
Total size of the model: 6247.68 MB
```

C.4 Chapter 5

Exercise 5.1

We can print the number of times the token (or word) "pizza" is sampled using the `print_sampled_tokens` function we defined in this section. Let's start with the code we defined in section 5.3.1.

The "pizza" token is sampled 0x if the temperature is 0 or 0.1, and it is sampled 32× if the temperature is scaled up to 5. The estimated probability is 32/1000 × 100% = 3.2%.

The actual probability is 4.3% and contained in the rescaled softmax probability tensor (`scaled_probas[2][6]`).

Exercise 5.2

Top-k sampling and temperature scaling are settings that have to be adjusted based on the LLM and the desired degree of diversity and randomness in the output.

When using relatively small top-k values (e.g., smaller than 10) and the temperature is set below 1, the model's output becomes less random and more deterministic. This setting is useful when we need the generated text to be more predictable, coherent, and closer to the most likely outcomes based on the training data.

Applications for such low k and temperature settings include generating formal documents or reports where clarity and accuracy are most important. Other examples of applications include technical analysis or code generation tasks, where precision is crucial. Also, question answering and educational content require accurate answers where a temperature below 1 is helpful.

On the other hand, larger top-k values (e.g., values in the range of 20 to 40) and temperature values above 1 are useful when using LLMs for brainstorming or generating creative content, such as fiction.

Exercise 5.3

There are multiple ways to force deterministic behavior with the `generate` function:

1. Setting to `top_k=None` and applying no temperature scaling;
2. Setting `top_k=1`.

Exercise 5.4

In essence, we have to load the model and optimizer that we saved in the main chapter:

```
checkpoint = torch.load("model_and_optimizer.pth")
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
```

Then, call the `train_simple_function` with `num_epochs=1` to train the model for another epoch.

Exercise 5.5

We can use the following code to calculate the training and validation set losses of the GPT model:

```
train_loss = calc_loss_loader(train_loader, gpt, device)
val_loss = calc_loss_loader(val_loader, gpt, device)
```

The resulting losses for the 124M parameter are as follows:

```
Training loss: 3.754748503367106
Validation loss: 3.559617757797241
```

The main observation is that the training and validation set performances are in the same ballpark. This can have multiple explanations.

1. The Verdict was not part of the pretraining dataset when OpenAI trained

GPT-2. Hence, the model is not explicitly overfitting to the training set and performs similarly well on The Verdict's training and validation set portions. (The validation set loss is slightly lower than the training set loss, which is unusual in deep learning. However, it's likely due to random noise since the dataset is relatively small. In practice, if there is no overfitting, the training and validation set performances are expected to be roughly identical).

2. The Verdict was part of GPT -2's training dataset. In this case, we can't tell whether the model is overfitting the training data because the validation set would have been used for training as well. To evaluate the degree of overfitting, we'd need a new dataset generated after OpenAI finished training GPT-2 to make sure that it couldn't have been part of the pretraining.

Exercise 5.6

In the main chapter, we experimented with the smallest GPT-2 model, which has only 124M parameters. The reason was to keep the resource requirements as low as possible. However, you can easily experiment with larger models with minimal code changes. For example, instead of loading the 1558M instead of 124M model in chapter 5, the only 2 lines of code that we have to change are the following:

```
hparams, params = download_and_load_gpt2(model_size="124M", model
model_name = "gpt2-small (124M)"
```

The updated code is as follows:

```
hparams, params = download_and_load_gpt2(model_size="1558M", mode
model_name = "gpt2-xl (1558M)"
```

Appendix D. Adding Bells and Whistles to the Training Loop

In the appendix, we enhance the training function for the pretraining and finetuning processes covered in chapters 5-7. This appendix, in particular, covers *learning rate warmup*, *cosine decay*, and *gradient clipping* in the first three sections.

The final section then incorporates these techniques into the training function developed in chapter 5 and pretrains an LLM.

To make the code in this appendix self-contained, we reinitialize the model we trained in chapter 5.

```
import torch
from previous_chapters import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,  # Vocabulary size
    "ctx_len": 256,       # Shortened context length (orig: 1024)
    "emb_dim": 768,       # Embedding dimension
    "n_heads": 12,        # Number of attention heads
    "n_layers": 12,       # Number of layers
    "drop_rate": 0.1,     # Dropout rate
    "qkv_bias": False     # Query-key-value bias
}
device = torch.device("cuda" if torch.cuda.is_available() else "c
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()
```

After initializing the model, we also need to initialize the data loaders we used in chapter 5. First, we load the "The Verdict" short story:

```
import os
import urllib.request

file_path = "the-verdict.txt"
url = "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/
```

```
if not os.path.exists(file_path):
    with urllib.request.urlopen(url) as response:
        text_data = response.read().decode('utf-8')
    with open(file_path, "w", encoding="utf-8") as file:
        file.write(text_data)
else:
    with open(file_path, "r", encoding="utf-8") as file:
        text_data = file.read()
```

Next, we load the `text_data` into the data loaders:

```
from previous_chapters import create_dataloader_v1

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
torch.manual_seed(123)
train_loader = create_dataloader_v1(
    text_data[:split_idx],
    batch_size=2,
    max_length=GPT_CONFIG_124M["ctx_len"],
    stride=GPT_CONFIG_124M["ctx_len"],
    drop_last=True,
    shuffle=True
)
val_loader = create_dataloader_v1(
    text_data[split_idx:],
    batch_size=2,
    max_length=GPT_CONFIG_124M["ctx_len"],
    stride=GPT_CONFIG_124M["ctx_len"],
    drop_last=False,
    shuffle=False
)
```

Now that we have re-instantiated the model and data loaders we used in chapter 5, the next section will introduce the enhancements we make to the training function.

D.1 Learning rate warmup

The first technique we introduce is *learning rate warmup*. Implementing a learning rate warmup can stabilize the training of complex models such as LLMs. This process involves gradually increasing the learning rate from a very low initial value (`initial_lr`) to a maximum value specified by the user (`peak_lr`). Starting the training with smaller weight updates decreases the

risk of the model encountering large, destabilizing updates during its training phase.

Suppose we plan to train an LLM for 15 epochs, starting with an initial learning rate of 0.0001 and increasing it to a maximum learning rate of 0.01. Furthermore, we define 20 warmup steps to increase the initial learning rate from 0.0001 to 0.01 in the first 20 training steps:

```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
warmup_steps = 20
```

Next, we implement a simple training loop template to illustrate this warmup process:

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.
lr_increment = (peak_lr - initial_lr) / warmup_steps #A

global_step = -1
track_lrs = []

for epoch in range(n_epochs):  #B
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps: #C
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

        for param_group in optimizer.param_groups: #D
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])
        #E
```

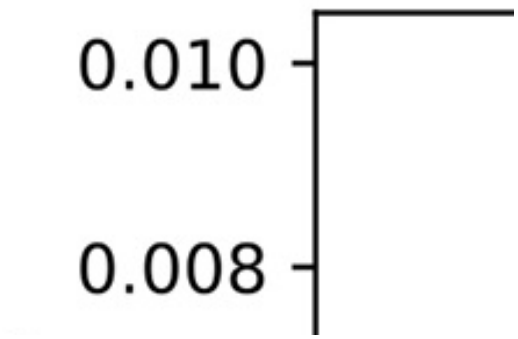
After running the preceding code, we visualize how the learning rate was changed by the training loop above to verify that the learning rate warmup works as intended:

```
import matplotlib.pyplot as plt
plt.ylabel("Learning rate")
plt.xlabel("Step")
```

```
total_training_steps = len(train_loader) * n_epochs
plt.plot(range(total_training_steps), track_lrs);
plt.show()
```

The resulting plot is shown in Figure D.1.

Figure D.1 The learning rate warmup increases the learning rate for the first 20 training steps. After 20 steps, the learning rate reaches the peak of 0.01 and remains constant for the rest of the training.



As shown in Figure D.1, the learning rate starts with a low value and increases for 20 steps until it reaches the maximum value after 20 steps.

In the next section, we will modify the learning rate further so that it decreases after reaching the maximum learning rate, which further helps improve the model training.

D.2 Cosine decay

Another widely adopted technique for training complex deep neural networks and LLMs is *cosine decay*. This method modulates the learning rate throughout the training epochs, making it follow a cosine curve after the warmup stage.

In its popular variant, cosine decay reduces (or decays) the learning rate to nearly zero, mimicking the trajectory of a half-cosine cycle. The gradual learning decrease in cosine decay aims to decelerate the pace at which the model updates its weights. This is particularly important as it helps minimize the risk of overshooting the loss minima during the training process, which is essential for ensuring the stability of the training during its later phases.

We can modify the training loop template from the previous section, adding cosine decay as follows:

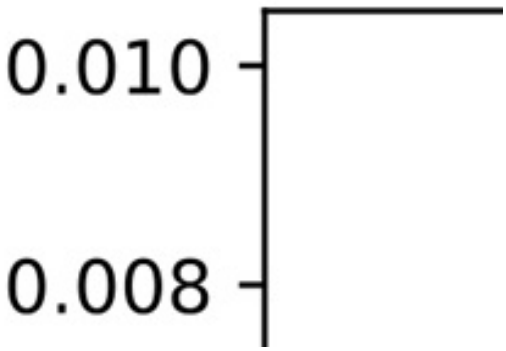
```
import math

min_lr = 0.1 * initial_lr
track_lrs = []
lr_increment = (peak_lr - initial_lr) / warmup_steps
global_step = -1

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:# #B
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
            lr = min_lr + (peak_lr - min_lr) * 0.5 * (1 + math.co

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])
```

Again, to verify that the learning rate has changed as intended, we plot the learning rate:

```
plt.ylabel("Learning rate")
plt.xlabel("Step")
plt.plot(range(total_training_steps), track_lrs)
plt.show()
```

The resulting learning rate plot is shown in Figure D.2.

Figure D.2 The first 20 steps of linear learning rate warmup are followed by a cosine decay, which reduces the learning rate in a half-cosine cycle until it reaches its minimum point at the end of training.



As shown in Figure D.2, the learning rate starts with a linear warmup phase, which increases for 20 steps until it reaches the maximum value after 20 steps. After the 20 steps of linear warmup, cosine decay kicks in, reducing the learning rate gradually until it reaches its minimum.

D.3 Gradient clipping

In this section, we introduce *gradient clipping*, another important technique for enhancing stability during LLM training. This method involves setting a threshold above which gradients are downscaled to a predetermined maximum magnitude. This process ensures that the updates to the model's parameters during backpropagation stay within a manageable range.

For example, applying the `max_norm=1.0` setting within PyTorch's `clip_grad_norm_` function ensures that the norm of the gradients does not surpass 1.0. Here, the term "norm" signifies the measure of the gradient vector's length, or magnitude, within the model's parameter space, specifically referring to the L2 norm, also known as the Euclidean norm.

In mathematical terms, for a vector $\boldsymbol{v}$ composed of components $\boldsymbol{v} = [v_1, v_2, ..., v_n]$, the L2 norm is described as:

$$|v|_2 = \sqrt{{v_1}^2 + {v_2}^2 + \ldots + {v_n}^2}$$

This calculation method is also applied to matrices.

For instance, consider a gradient matrix given by:

$$G = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

If we aim to clip these gradients to a max_norm of 1, we first compute the L2 norm of these gradients, which is

$$|G|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

Given that $|\mathbf{G}|_2 = 5$ exceeds our `max_norm` of 1, we scale down the gradients to ensure their norm equals exactly 1. This is achieved through a scaling factor, calculated as `max_norm`/$|\mathbf{G}|_2 = 1/5$. Consequently, the adjusted gradient matrix **G'** becomes

$$G' = \frac{1}{5} \times G \begin{bmatrix} 1/1 & 2/5 \\ 2/5 & 4/5 \end{bmatrix}$$

To illustrate this gradient clipping process, we would begin by initializing a new model and calculating the loss for a training batch, similar to the procedure in a standard training loop:

```
from previous_chapters import calc_loss_batch
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()
```

Upon calling the `.backward()` method in the preceding code snippet, PyTorch calculates the loss gradients and stores them in a `.grad` attribute for each model weight (parameter) tensor.

For illustration purposes, we can define the following `find_highest_gradient` utility function to identify the highest gradient value by scanning all the `.grad` attributes of the model's weight tensors after calling `.backward()`:

```
def find_highest_gradient(model):
    max_grad = None
    for param in model.parameters():
        if param.grad is not None:
```

```
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            if max_grad is None or max_grad_param > max_grad:
                max_grad = max_grad_param
    return max_grad
print(find_highest_gradient(model))
```

The largest gradient value identified by the preceding code is as follows:

```
tensor(0.0373)
```

Let's now apply gradient clipping, which can be implemented with one line of code, and see how this affects the largest gradient value:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
print(find_highest_gradient(model))
```

The largest gradient value after applying the gradient clipping with the max norm of 1 is substantially smaller than before:

```
tensor(0.0166)
```

In the next section, we will put all the concepts covered in this appendix so far into action and modify the LLM training function.

D.4 The modified training function

In this final section of this appendix, we improve the `train_model_simple` training function we used in chapter 5 by adding the three concepts we introduced: linear warmup, cosine decay, and gradient clipping. Together, these methods help stabilize LLM training.

The code is as follows, with the changes compared to the `train_model_simple` annotated:

```
from previous_chapters import evaluate_model, generate_and_print_

def train_model(model, train_loader, val_loader, optimizer, devic
                eval_freq, eval_iter, start_context, warmup_steps
                initial_lr=3e-05, min_lr=1e-6):
```

```
train_losses, val_losses, track_tokens_seen, track_lrs = [],
tokens_seen, global_step = 0, -1

peak_lr = optimizer.param_groups[0]["lr"] #A
total_training_steps = len(train_loader) * n_epochs #B
lr_increment = (peak_lr - initial_lr) / warmup_steps #C

for epoch in range(n_epochs):
    model.train()
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps: #D
            lr = initial_lr + global_step * lr_increment
        else:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps)
            lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                1 + math.cos(math.pi * progress))

        for param_group in optimizer.param_groups: #E
            param_group["lr"] = lr
        track_lrs.append(lr)
        loss = calc_loss_batch(input_batch, target_batch, mod
        loss.backward()

        if global_step > warmup_steps: #F
            torch.nn.utils.clip_grad_norm_(model.parameters()
        #G
        optimizer.step()
        tokens_seen += input_batch.numel()

        if global_step % eval_freq == 0:
            train_loss, val_loss = evaluate_model(
                model, train_loader, val_loader,
                device, eval_iter
            )
            train_losses.append(train_loss)
            val_losses.append(val_loss)
            track_tokens_seen.append(tokens_seen)
            print(f"Ep {epoch+1} (Iter {global_step:06d}): "
                  f"Train loss {train_loss:.3f}, Val loss {va

    generate_and_print_sample(
        model, train_loader.dataset.tokenizer,
        device, start_context
```

```
        )

    return train_losses, val_losses, track_tokens_seen, track_lrs
```

After defining the `train_model` function, we can use it in a similar fashion to train the model compared to the `train_model_simple` method in chapter 5:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
peak_lr = 5e-4
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.

n_epochs = 15
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=
    eval_freq=5, eval_iter=1, start_context="Every effort moves y
    warmup_steps=10, initial_lr=1e-5, min_lr=1e-5
)
```

The training will take about 5 minutes to complete on a MacBook Air or similar laptop and print the following outputs:

```
Ep 1 (Iter 000000): Train loss 10.934, Val loss 10.939
Ep 1 (Iter 000005): Train loss 8.529, Val loss 8.843
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
Ep 2 (Iter 000010): Train loss 6.400, Val loss 6.825
Ep 2 (Iter 000015): Train loss 6.116, Val loss 6.861
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
...
the irony. She wanted him vindicated--and by me!"  He laughed aga
Ep 15 (Iter 000130): Train loss 0.101, Val loss 6.707
Every effort moves you?"  "Yes--quite insensible to the irony. Sh
```

Like chapter 5, the model begins to overfit after a few epochs since it is a very small dataset, and we iterate over it multiple times. However, we can see that the function is working since it minimizes the training set loss.

Readers are encouraged to train the model on a larger text dataset and compare the results obtained with this more sophisticated training function to the results that can be obtained with the `train_model_simple` function used in chapter 5.