TEST YOURSELF ON

BUILD A

Large Language Model

(FROM SCRATCH)

Manning Editorial Team

FREE

MANNING

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

ISBN: 9781638357629

brief contents

CHAPTERS

about the book

Sebastian Raschka's bestselling book *Build a Large Language Model (From Scratch)* is the best way to learn how Large Language Models function. With over 350 pages in seven chapters and five Appendixes, it guides you step-by-step in building an entire large language model similar to GPT-2. It uses Python and the PyTorch deep learning library. It's a unique way to learn this subject, which some believe is the only way to truly learn: you build a model yourself.

Even with the clear explanations, diagrams and code in the book, learning a complex subject is still hard. This Test Yourself guide intends to make it a little easier. The structure mirrors the structure of Build a Large Language Model (From Scratch), focusing on key concepts from each chapter. You can test yourself with multiple-choice quizzes, questions on code and key concepts, and questions with longer answers that push you to think critically. The answers to all questions are provided.

Depending on what you know at any point, this Test Yourself guide can help you in different ways. It will solidify your knowledge if used after reading a chapter. But it will also benefit you if you digest it before reading. By testing yourself on the main concepts and their relationships you are primed to navigate a chapter more easily and be ready for its messages.

We recommend using it before and after reading, as well as later when you have started forgetting. Repeated learning solidifies our knowledge and integrates it with related knowledge already in our long-term memory.

1 Understanding large language models

Chapter 1 provides a high-level introduction to **large language models (LLMs**), exploring their applications, building stages, and the underlying transformer architecture. The chapter discusses the concept of **pretraining** and **fine-tuning**, which are two crucial steps in developing effective LLMs. It introduces the **transformer architecture** and its key components, including the **encoder** and **decoder** modules, as well as the self-attention mechanism. The chapter also provides a plan for building an LLM from scratch, outlining the three stages involved: data preparation and sampling, attention mechanism implementation, and pretraining on unlabeled data to obtain a foundational model for further fine-tuning.

All the answers to the questions can be found at the end of this document.

Quick questions on main concepts

1. **What is the primary difference between deep learning and traditional machine learning in the context of LLMs?**
   - A. Deep learning is better suited for handling structured data, while traditional machine learning is better for unstructured data.
   - B. Deep learning does not require manual feature extraction, while traditional machine learning does.
   - C. Deep learning is more accurate than traditional machine learning for all tasks.
   - D. Deep learning is more computationally efficient than traditional machine learning.

2. **What is the primary function of a large language model (LLM)?**

A. To analyze and interpret images.
B. To predict future events.
C. To understand, generate, and respond to human-like text.
D. To control and operate robots.

3. **What is the main advantage of using custom-built LLMs over general-purpose LLMs?**
   A. They can outperform general-purpose LLMs in specific tasks or domains.
   B. They are more versatile and can be used for a wider range of tasks.
   C. They are more efficient in handling large datasets.
   D. They are less expensive to train.

4. **What is the significance of the "transformer" architecture in LLMs?**
   A. It provides a faster processing speed for large datasets.
   B. It allows the model to selectively focus on different parts of the input text when making predictions.
   C. It enables the model to learn from unlabeled data.
   D. It allows the model to translate languages without specific training.

5. **What is the main purpose of pretraining an LLM?**
   A. To fine-tune the model for specific tasks.
   B. To evaluate the model's performance on various tasks.
   C. To create a model that can translate languages.
   D. To develop a broad understanding of language by training on a large, diverse dataset.

Questions by chapter section

Now we'll move through the chapter in more detail.

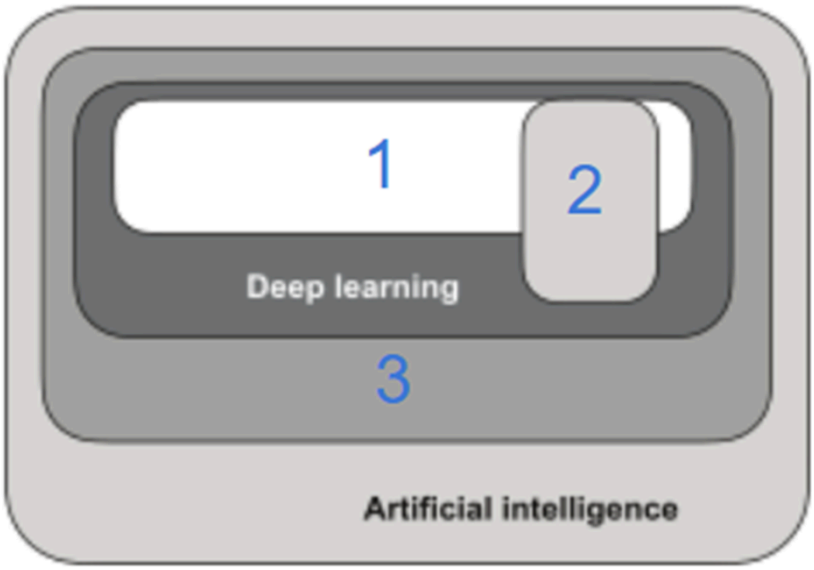
1.1 What is an LLM?

1. What is a large language model (LLM) and how does it work?

2. What is the significance of the 'large' in 'large language model'?

3. How do LLMs relate to generative AI?

4. Label this diagram:



Fill the table with the label descriptions:

Label	Description
1	
2	
3	

5. What is the difference between traditional machine learning and deep learning in terms of feature extraction?

6. Match the terms to its description on the right:

Large Language Model (LLM)		A subset of machine learning that uses deep neural networks to model complex patterns and abstractions in data.
Transformer		A type of artificial intelligence that can create new content, such as text, images, or audio.
Generative AI		An architecture used in LLMs that allows them to pay selective attention to different parts of the input when making predictions, making them adept at handling the nuances of human language.
Deep Learning		A type of artificial intelligence that uses deep learning to understand, generate, and respond to human-like text.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

1.2 Applications of LLMs

1. What are some of the key applications of LLMs in various domains?

2. How do LLMs contribute to the development of chatbots and virtual assistants?

3. Explain the role of LLMs in knowledge retrieval from specialized fields.

4. What is the potential impact of LLMs on our relationship with technology?

5. Match the terms on the left to its description on the right:

Natural Language Processing		a field of computer science that focuses on enabling computers to understand and process human language.
Machine Translation		computer programs designed to simulate conversation with human users.
Chatbots		the process of extracting relevant information from large amounts of data.
Knowledge Retrieval		the automatic translation of text from one language to another.

Fill the table with the column mappings:

Left Hand Column	1	2	3
Right Hand Column			

1.3 Stages of building and using LLMs

1. What are the main advantages of building custom LLMs compared to using general-purpose LLMs like ChatGPT?

2. Describe the two-stage training process involved in creating an LLM.

3. What is the purpose of pretraining an LLM, and what type of data is used in this stage?

4. Explain the concept of self-supervised learning in the context of pretraining LLMs.

5. What are the two main categories of fine-tuning LLMs, and how do they differ in terms of the labeled data used?

6. Match the term on the left to its description on the right:

Pretraining		A type of fine-tuning where the labeled dataset consists of instruction and answer pairs, such as a query to translate a text accompanied by the correctly translated text.
Fine-tuning		A type of fine-tuning where the labeled dataset consists of texts and associated class labels, for example, emails associated with "spam" and "not spam" labels.
Instruction Fine-tuning		The process of further training a pretrained LLM on a narrower dataset that is more specific to particular tasks or domains.
Classification Fine-tuning		The initial phase of training an LLM on a large, diverse dataset to develop a broad understanding of language.

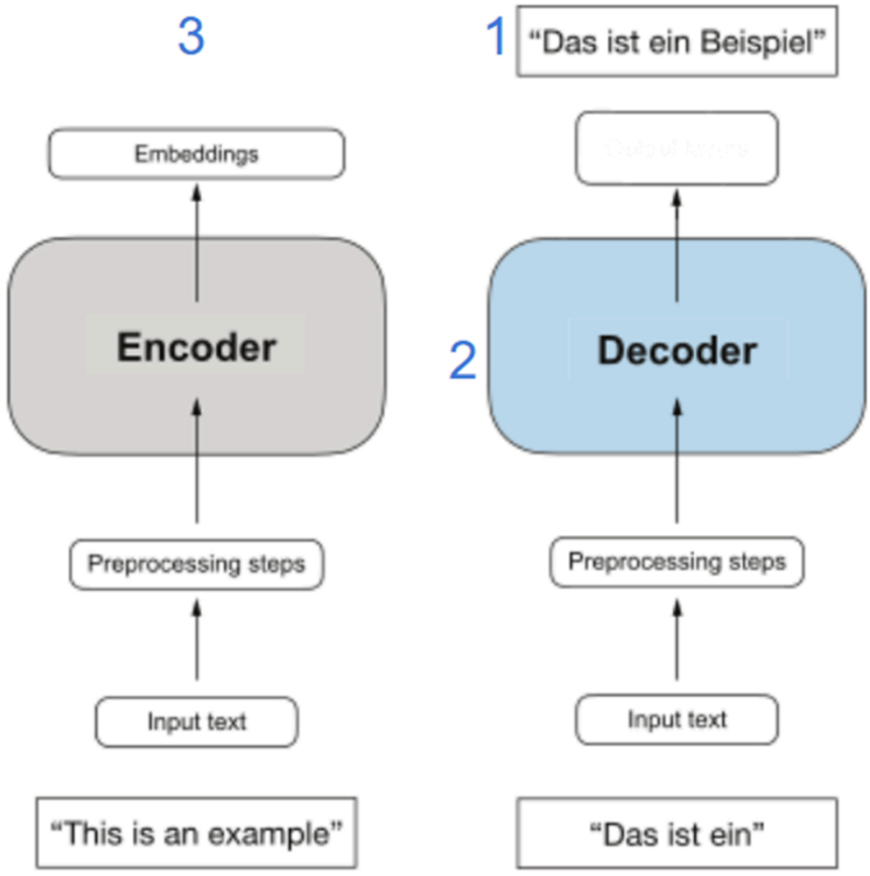
Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

1.4 Introducing the transformer architecture

1. What is the transformer architecture and what is its significance in the development of LLMs?

2. Which label in the diagram does the output of the Embeddings stage go to?



3. Describe the two main components of the transformer architecture and their roles in language processing.

4. What is the self-attention mechanism and how does it contribute to the transformer's effectiveness?

5. Explain the key differences between BERT and GPT models in terms of their training approaches and primary applications.

6. What are zero-shot and few-shot learning, and how do they relate to GPT models?

1.5 Utilizing large datasets

1. What are the key characteristics of the training datasets used for large language models like GPT-3 and BERT?

2. Explain the significance of the size and diversity of the training dataset for the performance of large language models.

3. What is the concept of 'tokenization' in the context of large language models?

4. Describe the concept of 'pretraining' in the context of large language models and its significance.

5. Explain the concept of 'fine-tuning' in the context of large language models and its advantages.

6.

7.

8. Match the term on the left to its description on the right:

Encoder		The ability of a model to generalize to completely unseen tasks without any prior specific examples.
Decoder		The part of the transformer architecture that takes the encoded vectors from the encoder and generates the output text.
Self-attention mechanism		The part of the transformer architecture that processes the input text and encodes it into a series of numerical representations or vectors that capture the contextual information of the input.
Zero-shot learning		Allows the model to weigh the importance of different words or tokens in a sequence relative to each other, enabling it to capture long-range dependencies and contextual relationships within the input data.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

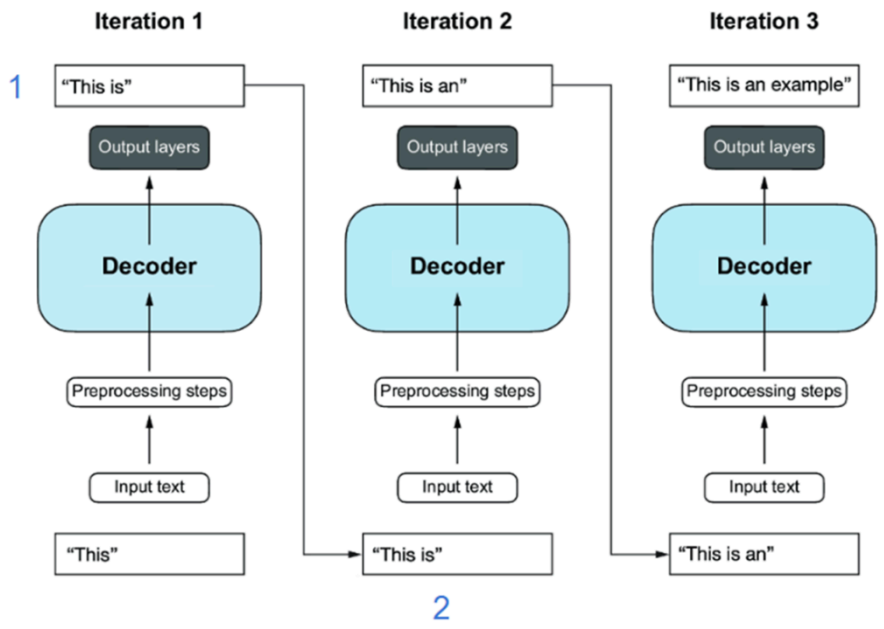
1.6 A closer look at the GPT architecture

1. What is the primary task that GPT models are trained on, and how does this relate to their ability to perform other tasks like translation?

2. Explain the concept of self-supervised learning in the context of GPT models.

3. How does the GPT architecture differ from the original transformer architecture, and what are the implications of this difference?

4. What is happening at the labels 1 and 2 in the diagram?



Fill out the table with your answers:

Label	Description
1	
2	

5. What is the significance of GPT models being considered autoregressive models?

6. Describe the relationship between the size and complexity of GPT models and their capabilities.

7. Match the term on the left to its description on the right:

Pretrained Models		Pretrained models that serve as a foundation for further fine-tuning on specific tasks.
Fine-tuning		The models are trained on massive datasets of text and code, allowing them to perform well on various tasks, including language syntax, semantics, and context.
Base Models		A process of adapting a pretrained model to a specific task by training it on a smaller dataset related to that task.

Fill the table with the column mappings:

Left Hand Column	1	2	3
Right Hand Column			

1.7 Building a large language model

1. What are the three main stages involved in building a large language model from scratch?

2. What is the key idea behind the transformer architecture used in LLMs?

3. What is the primary task used for pretraining LLMs like GPT-3?

4. Explain the concept of emergent properties in LLMs.

5. Why is fine-tuning a pretrained LLM beneficial for specific tasks?

6. Put these stages of creating a pretrained LLM (base model) in order:
   A. Evaluate the model's performance on text generation tasks.

B. Implement the transformer decoder architecture (GPT-like).
C. Prepare the text data by cleaning and tokenizing it.
D. Train the model using a next-word prediction task on a large text dataset.

Fill the table with your answers:

Step Order	Step
1	
2	
3	
4	

7. Match the term to its description on the right:

Autoregressive Model		The task of predicting the next word in a sequence, which is used to train GPT models.
Self-Supervised Learning		A type of machine learning where the model learns from the data itself, without requiring explicit labels.
Next-Word Prediction		A type of model that generates text by predicting the next word in a sequence based on the words that have already been generated.
Decoder-Only Architecture		The architecture of GPT models, which uses only the decoder portion of the transformer architecture, making it suitable for text generation.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

Answers

Multiple Choice

1. B
2. C
3. A
4. B
5. D

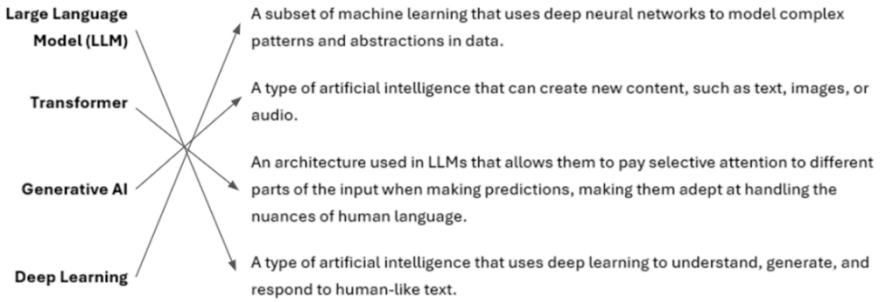
Answers by section

1.1 What is an LLM?

1. An LLM is a deep neural network trained on massive amounts of text data to understand, generate, and respond to human-like text. It uses a transformer architecture to pay attention to different parts of the input, making it adept at handling language nuances. LLMs are trained on the task of predicting the next word in a sequence, which allows them to learn context, structure, and relationships within text.
2. The 'large' refers to both the model's size in terms of parameters (adjustable weights) and the immense dataset it's trained on. LLMs often have tens or hundreds of billions of parameters, which are optimized during training to predict the next word in a sequence.
3. LLMs are often considered a form of generative AI because they can generate text. Generative AI is a broader term encompassing AI systems that create new content, such as text, images, or music.
4. AI is the broader field of creating machines that can perform tasks requiring human-like intelligence. Machine learning is a subset of AI that focuses on algorithms that learn from data. Deep learning is a subset of machine learning that uses deep neural networks with multiple layers to model complex patterns in data.
5. Here are the label descriptions:

Label	Description
1	Large Language Models (LLMs)
2	Generative AI (GenAI)
3	Machine Learning

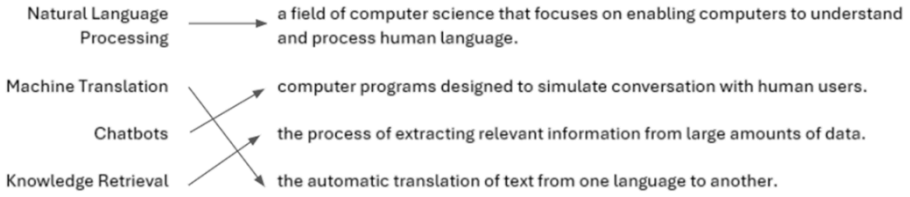
6. Traditional machine learning requires manual feature extraction, where human experts identify and select relevant features for the model. Deep learning, on the other hand, does not require this manual process, allowing the model to learn features directly from the data.
7. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	4	3	2	1

1.2 Applications of LLMs

1. LLMs are used in machine translation, text generation, sentiment analysis, text summarization, content creation, powering chatbots and virtual assistants, and knowledge retrieval from specialized areas like medicine and law.
2. LLMs enable chatbots and virtual assistants like ChatGPT and Gemini to understand and respond to user queries in a natural language, enhancing their ability to provide information and complete tasks.
3. LLMs can analyze vast amounts of text in fields like medicine or law, summarizing lengthy passages, answering technical questions, and facilitating efficient knowledge retrieval.
4. LLMs have the potential to make technology more conversational, intuitive, and accessible by automating text-based tasks and enabling natural language interactions with AI systems.
5. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	1	4	2	3

1.3 Stages of building and using LLMs

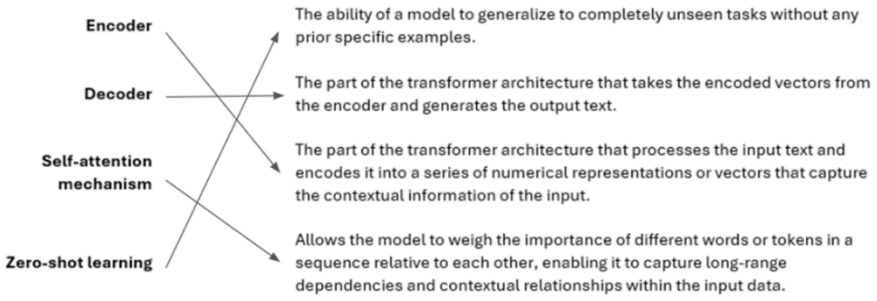
1. Custom LLMs offer advantages like improved performance for specific tasks or domains, enhanced data privacy by avoiding reliance on third-party providers, and the ability to deploy models locally on devices for reduced latency and costs.
2. The process begins with pretraining, where the model learns a broad understanding of language from a large, diverse dataset. This pretrained model is then fine-tuned on a smaller, more specific dataset to adapt it to particular tasks or domains.
3. Pretraining aims to develop a foundational model with a general understanding of language. It uses large amounts of unlabeled text data, often referred to as 'raw' text, to train the model to predict the next word in a sequence.
4. Self-supervised learning allows LLMs to generate their own labels from the input data during pretraining. This eliminates the need for manually labeled data, which is a common requirement in traditional supervised learning.
5. Instruction fine-tuning uses labeled data consisting of instruction-answer pairs, while classification fine-tuning uses labeled data with texts and associated class labels.

1.4 Introducing the transformer architecture

1. The transformer architecture is a deep neural network architecture that revolutionized natural language processing. It's the foundation for most modern LLMs, enabling them to process and understand language effectively.
2. The output of the Embeddings stage goes to the Decoder, label 2.
3. The transformer architecture consists of an encoder and a decoder. The encoder processes the input text and converts it into numerical representations, while the decoder uses these representations to generate the output text.
4. The self-attention mechanism allows the transformer to weigh the importance of different words in a sequence relative to each other. This helps the model capture long-range dependencies and contextual relationships, leading to more coherent and relevant output.
5. BERT focuses on masked word prediction and excels in tasks like text classification, while GPT is designed for generative tasks like text completion, translation, and summarization.
6. Zero-shot learning allows GPT models to perform tasks without prior training on specific examples, while few-shot learning enables them to learn from a minimal number of examples. These capabilities demonstrate GPT's versatility and adaptability.

1.5 Utilizing large datasets

1. These datasets are vast, encompassing billions of words and covering a wide range of topics and languages. They are designed to expose the models to diverse text, enabling them to learn language syntax, semantics, and context.
2. The scale and diversity of the training data allow these models to perform well on various tasks, including those requiring general knowledge. The models learn to understand and generate text that reflects the real-world complexities of language.
3. Tokenization is the process of converting text into individual units called tokens, which are the basic building blocks that the model reads and processes. These tokens can be words, punctuation marks, or other meaningful units of text.
4. Pretraining involves training a large language model on a massive dataset to learn general language patterns and knowledge. This pre-trained model serves as a foundation, making it adaptable for various downstream tasks through fine-tuning, which involves further training on specific datasets for specific applications.
5. Fine-tuning involves further training a pre-trained large language model on a smaller, task-specific dataset. This process adapts the model to perform well on specific tasks, such as text summarization or question answering, while leveraging the general knowledge learned during pretraining.
6. The terms are connected like this:



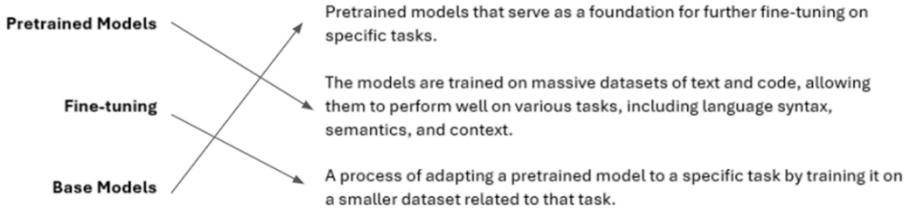
Left Hand Column	1	2	3	4
Right Hand Column	3	2	4	1

1.6 A closer look at the GPT architecture

1. GPT models are primarily trained on a next-word prediction task, which involves predicting the next word in a sequence. This seemingly simple task allows the models to learn the relationships between words and phrases, enabling them to perform other tasks like translation, even though they were not explicitly trained for it.
2. GPT models utilize self-supervised learning, where the model learns from the data itself without requiring explicit labels. In the case of GPT, the next word in a sentence serves as the label for the model to predict, allowing for training on massive unlabeled text datasets.
3. The GPT architecture uses only the decoder portion of the transformer, making it a decoder-only model. This design makes it suitable for text generation and next-word prediction tasks, as it generates text one word at a time in a unidirectional, left-to-right manner.
4. This is what is happening in the diagram:

Label	Description
1	The next word is created based on the input text
2	The output of the previous round becomes the input to the next round

5. Autoregressive models, like GPT, incorporate their previous outputs as inputs for future predictions. This means that each new word generated by GPT is based on the preceding sequence, ensuring coherence and fluency in the generated text.
6. GPT models, particularly GPT-3, are significantly larger than the original transformer model, with a greater number of layers and parameters. This increased size and complexity contribute to their ability to perform a wider range of tasks and achieve higher accuracy.
7. The terms are connected like this:



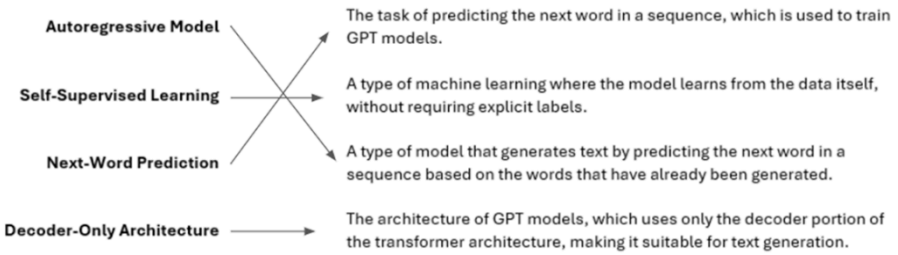
Left Hand Column	1	2	3
Right Hand Column	2	3	1

1.7 Building a large language model

1. The three stages are implementing the LLM architecture and data preparation process, pretraining an LLM to create a foundation model, and fine-tuning the foundation model for specific tasks.
2. The transformer architecture utilizes an attention mechanism that allows the LLM to selectively access the entire input sequence when generating output, word by word.
3. LLMs like GPT-3 are pretrained on a massive corpus of text by predicting the next word in a sentence, using this prediction as a label.
4. While the primary pretraining task for GPT-like models is next-word prediction, they exhibit emergent properties, meaning they can perform tasks like classification, translation, and summarization without explicit training for those tasks.
5. Fine-tuning a pretrained LLM on a custom dataset allows it to specialize in specific tasks and potentially outperform general LLMs on those tasks.
6. Here is the correct order of the steps:

Step Order	Step
1	C
2	B
3	D
4	A

7. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	3	2	1	4

2 Working with text data

Chapter 2 focuses on **preparing text data for LLM training** by converting it into numerical representations called **embeddings**. The chapter explores the process of **tokenization**, which involves splitting text into individual words or subword units, and converting these tokens into numerical IDs using a **vocabulary**. It covers different tokenization techniques, including **byte pair encoding (BPE)** used in models like GPT. The chapter also explains creating token embeddings, which are vector representations of tokens, and adding positional embeddings to encode the position of tokens within a sequence, providing the necessary input for subsequent LLM modules.

All the answers to the questions can be found at the end of this document.

Quick questions on main concepts

1. **What is the primary purpose of tokenization in the context of LLMs?**
   - A. Tokenization is used to convert text into lowercase.
   - B. Tokenization splits text into individual words or special characters
   - C. Tokenization is used to identify the parts of speech in a sentence.
   - D. Tokenization is used to remove stop words from text.

2. **What is the purpose of the <|unk|> token in a vocabulary used for LLMs?**
   - A. The `<|unk|>` token is used to represent punctuation marks.

B. The <|unk|> token is used to mark the beginning of a sentence.
C. The <|unk|> token represents unknown words that were not present in the training data.
D. The <|unk|> token is used to mark the end of a sentence.

3. **What is the primary task of an LLM during training?**
   A. LLMs are trained to translate text from one language to another.
   B. LLMs are trained to summarize text.
   C. LLMs are trained to answer questions based on a given text.
   D. LLMs are trained to predict the next word in a sequence, given a preceding context.

4. **What is the difference between absolute positional embeddings and relative positional embeddings?**
   A. Absolute positional embeddings encode the exact position of a token in a sequence, while relative positional embeddings encode the relative distance between tokens.
   B. Absolute positional embeddings are only used for short sequences, while relative positional embeddings are used for longer sequences.
   C. Absolute positional embeddings are more efficient than relative positional embeddings.
   D. Relative positional embeddings are more accurate than absolute positional embeddings.

5. **The purpose of _____ in the context of LLMs is to provide information about the order and location of tokens within a sequence, helping the LLM understand the relationships between words.**
   A. attention mechanism
   B. positional embeddings
   C. tokenization

6. **What is the final output of the input processing pipeline for an LLM, before it is fed into the main LLM layers?**
   A. The final output is a tensor of probabilities for each word in the vocabulary.
   B. The final output is a tensor of text tokens.
   C. The final output is a tensor of input embeddings, created by combining token embeddings and positional embeddings.
   D. The final output is a tensor of token IDs.

Questions by chapter section

2.1 Understanding word embeddings

1. Why are word embeddings necessary for processing text data in deep learning models?

2. What is the main idea behind the Word2Vec approach to generating word embeddings?

3. Explain the trade-off involved in choosing the dimensionality of word embeddings.

4. How do LLMs typically handle word embeddings compared to using pretrained models like Word2Vec?

5. What is the primary challenge associated with visualizing high-dimensional word embeddings?

2.2 Tokenizing text

1. What is the purpose of tokenizing text in the context of building a large language model?

2. Describe the process of tokenizing text using Python's regular expression library 're'.

3. Why is it important to consider capitalization when tokenizing text for LLM training?

4. Explain the trade-off between removing whitespaces during tokenization and keeping them.

5. Match the term on the left to its description on the right:

Word Embeddings		The process of converting various data types, such as text, audio, or video, into a dense vector representation that deep learning models can understand.
Embedding		The dimensionality of a word embedding, which determines the number of dimensions used to represent each word, influencing the complexity and computational efficiency of the model.
Word2Vec		An algorithm that generates word embeddings by predicting the context of a word given the target word or vice versa, based on the idea that words appearing in similar contexts tend to have similar meanings.
Embedding Size		A method of representing words as continuous-valued vectors, allowing deep learning models to process text data.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

2.3 Converting tokens into token IDs

1. What is the purpose of converting tokens into token IDs?

2. How is a vocabulary created for tokenization?

3. What is the purpose of the `encode` method in the `SimpleTokenizerV1` class?

4. What is the purpose of the `decode` method in the `SimpleTokenizerV1` class?

5. What is the limitation of using a vocabulary built from a small training set?

6. Match the term on the left to its description on the right:

Tokenization		individual units of text that result from tokenization, representing words, punctuation, or other special characters.
Tokens		used to define patterns in text, allowing for flexible and precise text manipulation, including tokenization.
Regular Expressions		initial steps taken to prepare text data for further processing, such as tokenization, which makes the text suitable for use in language models.
Preprocessing		splitting text into individual units, called tokens, which can be words, punctuation marks, or other special characters.

Left Hand Column	1	2	3	4
Right Hand Column				

2.4 Adding special context tokens

1. What are the two special tokens added to the vocabulary and what are their purposes?

2. How does the modified `SimpleTokenizerV2` handle unknown words?

3. Explain the purpose of the `<|endoftext|>` token when training on multiple independent documents.

4. A piece of the code has been removed from this listing. Which of these terms has been removed and where should it go?

```
A unk   B \n   C <|unk|>   D |unk|
```

```
def encode(self, text):
    preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)
    preprocessed = [
        item.strip() for item in preprocessed if item.strip()
    ]
    preprocessed = [item if item in self.str_to_int
                    else "__1__" for item in preprocessed]

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

Fill out the table with your answer

Position	1
Term	

5. What are the additional special tokens commonly used in LLMs, and what are their functions?

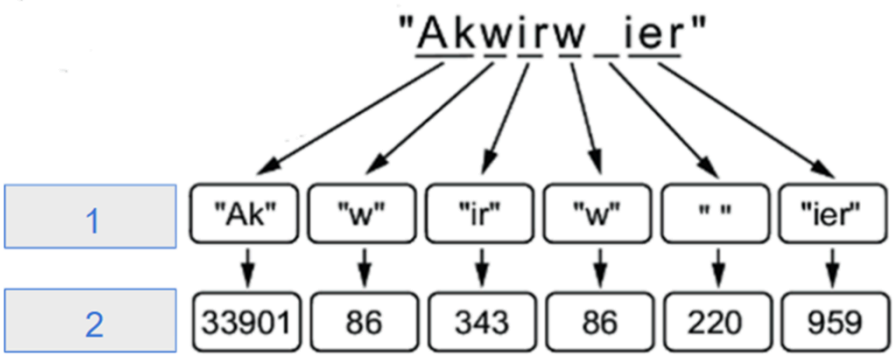
6. Match the term on the left to its description on the right:

Vocabulary		integer representations of tokens, used as an intermediate step before converting tokens into embedding vectors.
Token IDs		the dataset used to build the vocabulary and train the language model.
Tokenizer		a mapping from unique tokens to unique integer values, created by tokenizing the entire training dataset and sorting the tokens alphabetically.
Training Set		a class that implements methods for encoding text into token IDs and decoding token IDs back into text.

Left Hand Column	1	2	3	4
Right Hand Column				

2.5 Byte pair encoding

1. What are the two stages in this diagram?



Fill out the table with your answers:

Stage	Description
1	
2	

2. What is the primary advantage of using Byte Pair Encoding (BPE) for tokenization, especially when dealing with unknown words?

3. What is the total vocabulary size of the BPE tokenizer used in models like GPT-2, GPT-3, and the original ChatGPT?

4. How does the BPE tokenizer handle unknown words, such as `someunknownPlace`, without using `<|unk|>` tokens?

5. What Python library is used to implement the BPE tokenizer in the provided code example?

2.6 Data sampling with a sliding window

1. Explain the purpose of creating input-target pairs in the context of training a large language model (LLM).

2. Describe the sliding window approach used for generating input-target pairs and how it works.

3. Pieces of the code have been removed from three places in this listing. Which of these terms have been removed and where should they go?

```
A torch.vector   B tiktoken   C tokenizer   D  torch.tensor
```

Fill out the table with your answers

```
import torch
from torch.utils.data import Dataset, DataLoader
class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []

        token_ids = ____1____.encode(txt)

        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(____2____(input_chunk))
            self.target_ids.append(____3____(target_chunk))

    def __len__(self):
        return len(self.input_ids)

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

Position	1	2	3
Term			

4. What is the role of the `stride` parameter in the `GPTDatasetV1` class, and how does it affect the generation of input-target pairs?

5. Explain the purpose of the `max_length` parameter in the `GPTDatasetV1` class and its impact on the input-target pairs.

6. Pieces of the code have been removed from two places in this listing. Which of these terms have been removed and where should they go?

`A tokenizer` `B tiktoken` `C dataset` `D dataloader`

```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True,
                         num_workers=0):
    tokenizer = ___1___.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
    dataloader = DataLoader(
        ___2___,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last,
        num_workers=num_workers
    )

    return dataloader
```

Fill out the table with your answers

Position	1	2
Term		

7. What is the significance of using PyTorch's `Dataset` and `DataLoader` classes for creating a data loader for LLM training?

8. Match the term on the left to its description on the right:

Byte Pair Encoding (BPE)		words that are not present in the tokenizer's predefined vocabulary.
Subword Units		total number of unique tokens that a tokenizer can recognize and represent.
Out-of-Vocabulary (OOV) Words		smaller units of text that a BPE tokenizer breaks down words into, which can be individual characters or combinations of characters.
Vocabulary Size		a tokenization scheme that breaks down words into smaller subword units or individual characters, allowing it to handle unknown words by representing them as sequences of subword tokens or characters.

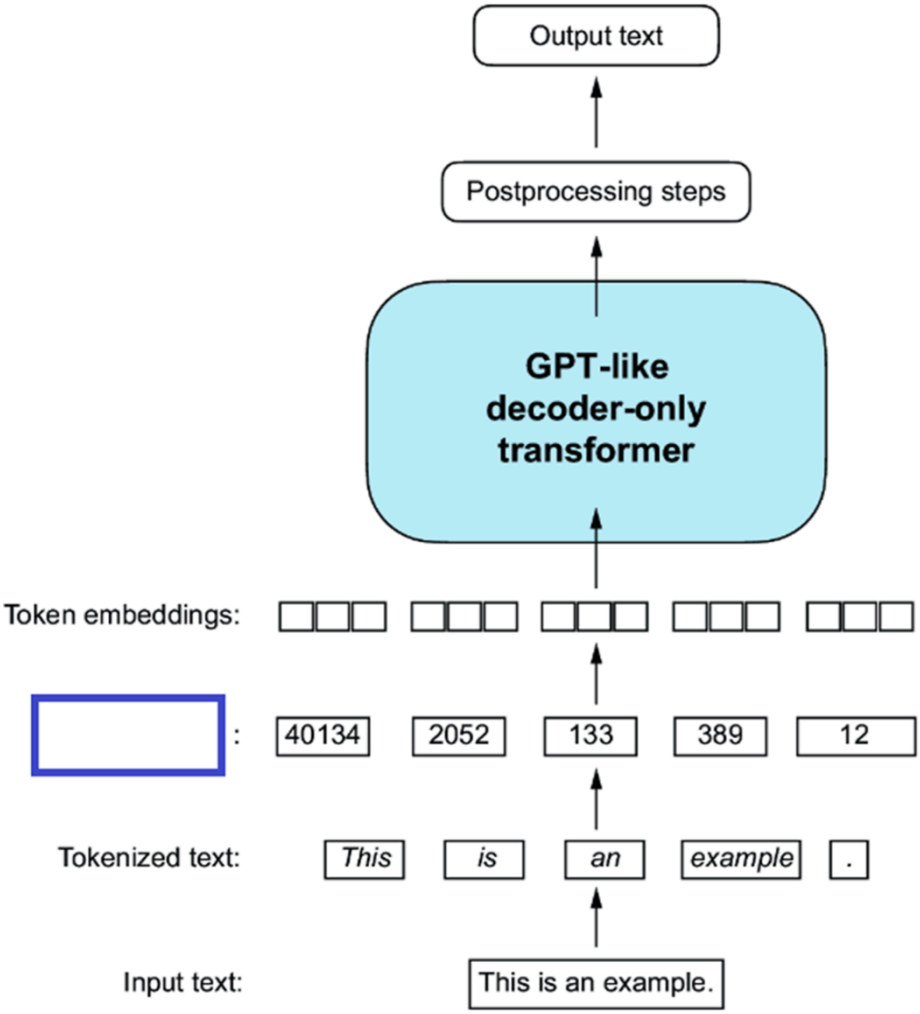
Left Hand Column	1	2	3	4
Right Hand Column				

2.7 Creating token embeddings

1. Why are embedding vectors necessary for training GPT-like LLMs?

2. How are embedding weights initialized in the beginning of LLM training?

3. What is the missing stage from this diagram?



4. Describe the process of converting a token ID into an embedding vector using an embedding layer.

5. How does the embedding layer's weight matrix relate to the vocabulary size and embedding dimension?

6. Match the term on the left to its description on the right:

Context Size		number of positions the input window is shifted when creating the next batch of input–target pairs.
Input–Target Pairs		a technique used to create input–target pairs from a text dataset by moving a window of tokens across the text.
Sliding Window		number of tokens that the LLM uses as input to predict the next word.
Stride		a set of data used to train an LLM, where the input is a sequence of tokens and the target is the next token in the sequence.

2.8 Encoding word positions

1. What is the main shortcoming of LLMs in terms of token order and how is it addressed?

2. Explain the difference between absolute and relative positional embeddings.

3. How are positional embeddings used in OpenAI's GPT models?

4. Describe the process of creating input embeddings for an LLM using token embeddings and positional embeddings.

5. What is the purpose of the `token_embedding_layer` and `pos_embedding_layer` in the code provided?

In Chapter Exercises

These are the exercises from Chapter 2. They are reproduced verbatim.

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."

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`.

Answers

Multiple Choice

1. B
2. C
3. D
4. A
5. B
6. C

Answers by section

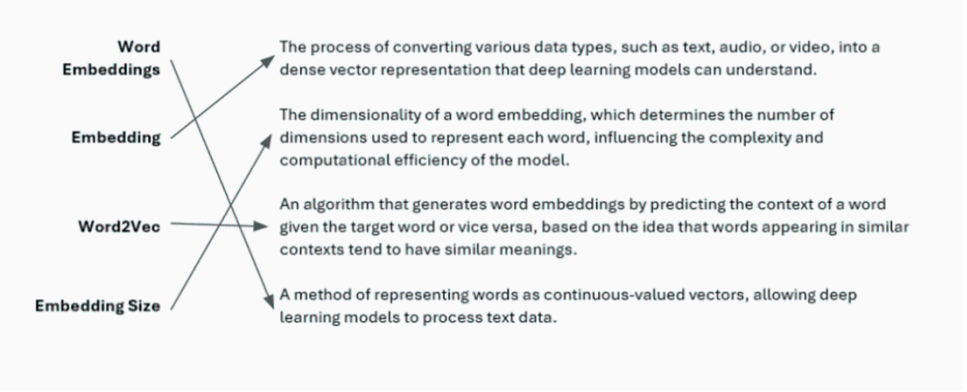
2.1 Understanding word embeddings

1. Deep learning models operate on numerical data, while text is categorical. Word embeddings convert words into continuous-valued vectors, making them compatible with the mathematical operations used in neural networks.
2. Word2Vec trains a neural network to predict the context of a word given the target word or vice versa. This approach assumes that words appearing in similar contexts tend to have similar meanings, resulting in clustered representations of related words in the embedding space.
3. Higher dimensionality in word embeddings can capture more nuanced relationships between words but comes at the cost of computational efficiency. Lower dimensionality offers faster processing but may sacrifice some semantic detail.

4. LLMs often generate their own embeddings as part of the input layer and optimize them during training. This allows for embeddings tailored to the specific task and data, potentially leading to better performance than using pre-trained embeddings.
5. Our visual perception and common graphical representations are limited to three dimensions or fewer. Visualizing high-dimensional embeddings requires specialized techniques or dimensionality reduction methods.

2.2 Tokenizing text

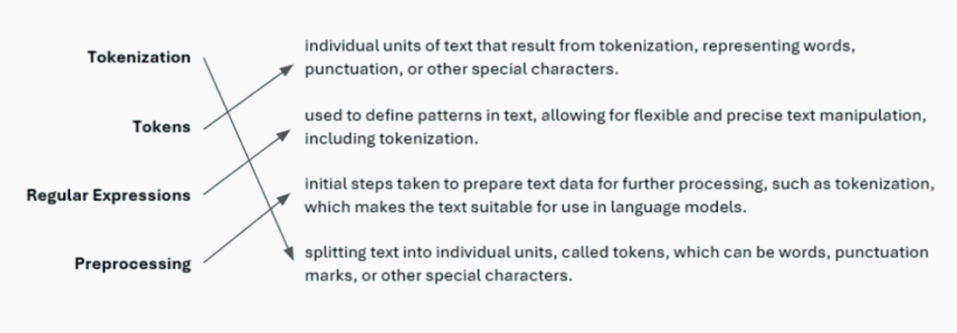
1. Tokenization is a crucial preprocessing step for creating embeddings for an LLM. It involves splitting input text into individual tokens, which are either words or special characters, to prepare the text for further processing and embedding creation.
2. The 're.split' function can be used to split text based on specific patterns. By defining a regular expression that matches whitespace characters, punctuation marks, and other special characters, we can separate the text into individual tokens. The resulting list can then be further processed to remove redundant whitespace characters.
3. Capitalization helps LLMs distinguish between proper nouns and common nouns, understand sentence structure, and learn to generate text with proper capitalization. Therefore, preserving capitalization during tokenization is beneficial for training effective language models.
4. Removing whitespaces reduces memory and computing requirements. However, keeping whitespaces can be useful for training models that are sensitive to the exact structure of the text, such as Python code, which relies on indentation and spacing.
5. The terms match like this:



Left Hand Column	1	2	3	4
Right Hand Column	4	1	3	2

2.3 Converting tokens into token IDs

1. Converting tokens into token IDs is an intermediate step before converting them into embedding vectors. This process allows for efficient representation and processing of text data within a language model.
2. A vocabulary is created by tokenizing the entire training dataset, sorting the unique tokens alphabetically, and assigning a unique integer to each token. This mapping allows for efficient conversion between tokens and their corresponding integer representations.
3. The `encode` method takes text as input, splits it into tokens, and uses the vocabulary to convert these tokens into their corresponding integer IDs. This process allows for representing text data as a sequence of integers, which can be processed by the language model.
4. The `decode` method takes a sequence of token IDs as input and uses the inverse vocabulary to convert these IDs back into their corresponding text tokens. This process allows for converting the output of the language model, which is a sequence of integers, back into human-readable text.
5. Using a vocabulary built from a small training set can lead to issues when encountering new words or phrases not present in the training data. This can result in errors during tokenization and decoding, highlighting the importance of using large and diverse training sets for building robust language models.
6. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	4	1	2	3

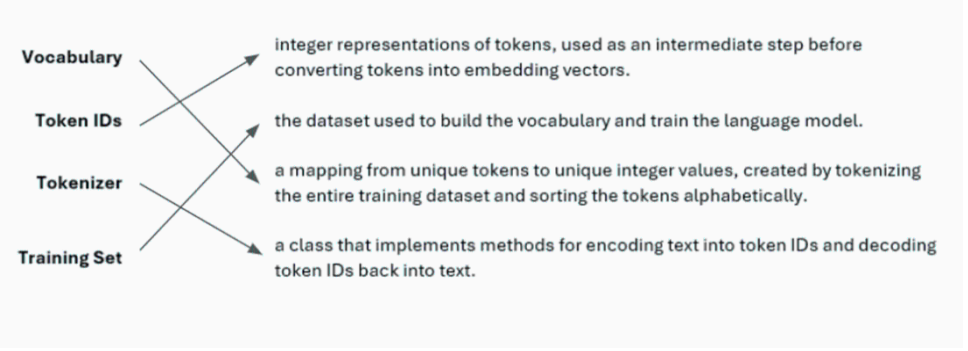
2.4 Adding special context tokens

1. The two special tokens added are `<|unk|>` and `<|endoftext|>`. `<|unk|>` represents unknown words not in the training data, while `<|endoftext|>` separates unrelated text sources, helping the LLM understand their distinct nature.

2. When encountering a word not in the vocabulary, `SimpleTokenizerV2` replaces it with the `<|unk|>` token, ensuring that all words are represented in the encoded text.
3. The `<|endoftext|>` token acts as a marker between unrelated text sources, signaling the start or end of a particular segment. This helps the LLM understand that these texts, though concatenated for training, are distinct entities.
4. The removed piece map to the position like this:

Position	1
Term	C

5. Other common special tokens include `[BOS]` (beginning of sequence), `[EOS]` (end of sequence), and `[PAD]` (padding). `[BOS]` marks the start of a text, `[EOS]` indicates the end, and `[PAD]` is used to extend shorter texts to match the length of the longest text in a batch for training.
6. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	3	1	4	2

2.5 Byte pair encoding

1. These are the stages:

Stage	Description
1	Tokens
2	Token IDs

2. BPE tokenizers break down unknown words into smaller subword units or even individual characters. This allows them to handle any word without needing a special `<|unk|>` token, ensuring that the tokenizer and the LLM can process any text, even if it contains words not present in the training data.
3. The BPE tokenizer used in these models has a vocabulary size of 50,257, with the `<|endoftext|>` token assigned the largest token ID.
4. The BPE tokenizer breaks down unknown words into smaller subword units or individual characters. This allows it to represent any word as a sequence of known subword tokens or characters, enabling it to process any text without needing a special token for unknown words.
5. The code uses the `tiktoken` library, which is an open-source Python library that efficiently implements the BPE algorithm based on Rust code.

2.6 Data sampling with a sliding window

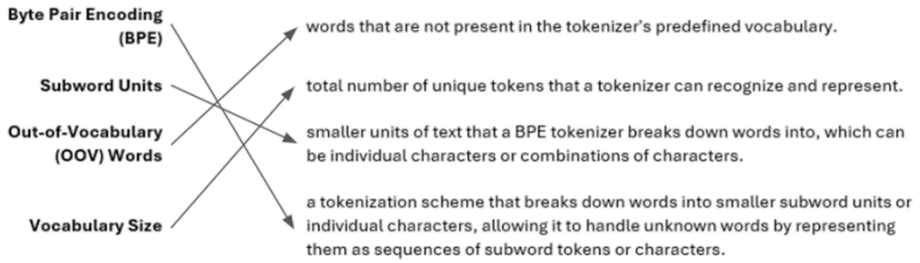
1. Input-target pairs are essential for training LLMs because they provide the model with examples of text sequences and their corresponding next words. This allows the LLM to learn the relationships between words and predict the most likely next word in a given context.
2. The sliding window approach involves iterating through a text sequence and extracting overlapping chunks of text as inputs. Each input chunk is paired with the corresponding next word as the target. The window slides across the text, creating multiple input-target pairs for training.
3. The removed pieces map to the positions like this:

Position	1	2	3
Term	C	D	D

4. The `stride` parameter determines the step size of the sliding window. A smaller stride results in more overlapping input chunks, while a larger stride creates less overlap. The choice of stride influences the amount of data generated and the potential for capturing long-range dependencies in the text.
5. The `max_length` parameter defines the size of the input chunks extracted from the text. It determines the number of tokens included in each input sequence. A larger `max_length` allows the LLM to process longer contexts, but it also increases the computational cost of training.
6. The removed pieces maps to the positions like this:

Position	1	2
Term	B	C

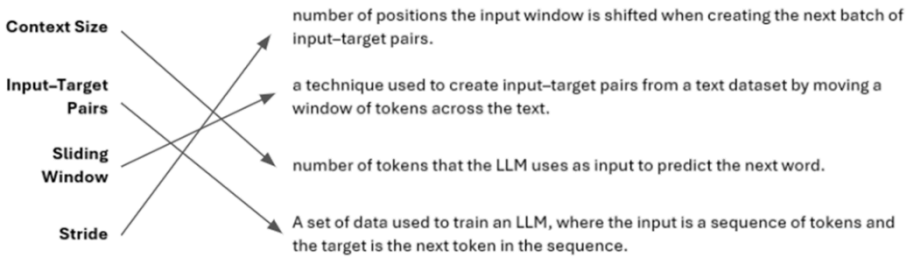
7. PyTorch's `Dataset` and `DataLoader` classes provide a convenient and efficient way to manage and iterate over large datasets. They allow for batching, shuffling, and parallel data loading, which are crucial for optimizing the training process of LLMs.
8. The terms match like this:



Left Hand Column	1	2	3	4
Right Hand Column	4	3	1	2

2.7 Creating token embeddings

1. Embedding vectors are essential for training GPT-like LLMs because these models are deep neural networks that rely on the backpropagation algorithm for learning. Backpropagation requires continuous vector representations, which embedding vectors provide.
2. Embedding weights are initially assigned random values. These random values serve as the starting point for the LLM's learning process. During training, the embedding weights are optimized through backpropagation to improve the model's performance.
3. The missing stage in the diagram is Token IDs.
4. The embedding layer acts as a lookup table. When given a token ID, it retrieves the corresponding embedding vector from its weight matrix. This embedding vector is a continuous representation of the token, allowing the LLM to process it effectively.
5. The embedding layer's weight matrix has a number of rows equal to the vocabulary size, representing each unique token. The number of columns corresponds to the embedding dimension, which determines the size of the embedding vector for each token.
6. The terms match like this:



Left Hand Column	1	2	3	4
Right Hand Column	3	4	2	1

2.8 Encoding word positions

1. LLMs' self-attention mechanism lacks a notion of token order. To address this, positional embeddings are introduced, which provide information about the position of each token within a sequence.
2. Absolute positional embeddings assign a unique embedding to each position in a sequence, indicating its exact location. Relative positional embeddings focus on the relative distance between tokens, allowing the model to generalize better to sequences of varying lengths.
3. GPT models use absolute positional embeddings that are optimized during training. These embeddings are not fixed or predefined but are learned alongside the model's other parameters.
4. Token embeddings are generated by mapping token IDs to vectors. Positional embeddings are then added to these token embeddings, resulting in input embeddings that incorporate both token identity and positional information.
5. The `token_embedding_layer` converts token IDs into embedding vectors, while the `pos_embedding_layer` generates positional embeddings based on the position of each token in the sequence.

In Chapter Exercise Solutions

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]))
copy
```

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=2, stride=2
)
```

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=8, stride=2
)
```

An example batch looks like

```
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]])
```

3 Coding Attention Mechanisms

Chapter 3 focuses on **coding different types of attention mechanisms** that are crucial in LLMs, specifically focusing on the **self-attention mechanism used in the GPT architecture**. Self-attention is a mechanism that allows each position in the input sequence to consider the relevancy of, or "attend to," all other 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.

All the answers to the questions can be found at the end of this document.

Quick Questions on main concepts

1. **What is the main problem with using traditional encoder-decoder RNN architectures for language translation, especially when dealing with long sentences?**
   - A. RNNs struggle to retain context from earlier parts of the input sequence due to the reliance on a single hidden state.
   - B. RNNs are prone to vanishing gradients, making it difficult to learn long-term dependencies.
   - C. RNNs are computationally expensive and slow to train.
   - D. RNNs are not suitable for handling sequential data like text.

2. **What is the purpose of the "query" vector in self-attention?**

A. The query vector is used to combine the key and value vectors.
B. The query vector is used to calculate the attention weights.
C. The query vector represents the current element in the sequence that the model is focusing on and is used to probe the other elements for relevance.
D. The query vector is used to generate the output sequence.

3. **What is the primary function of an attention mechanism in a language model?**
   A. To improve the model's ability to handle long sequences.
   B. To reduce the computational complexity of the model.
   C. To generate more creative and diverse outputs.
   D. To selectively focus on specific parts of the input sequence.

4. **The main advantage of using multiple attention heads in a _____ attention mechanism is to allow the model to attend to different aspects of the input sequence simultaneously, improving its ability to capture complex relationships.**
   A. multi-head attention
   B. single-head attention
   C. neural attention

5. **The purpose of _____ in the attention mechanism is to randomly drop out attention weights during training, preventing the model from becoming overly reliant on specific connections.**
   A. dropout
   B. regularization
   C. encoding

6. **The _____ class integrates the multi-head functionality within a single class, while the MultiHeadAttentionWrapper class uses a list of separate CausalAttention objects.**
   A. CausalAttention
   B. MultiHeadAttention
   C. Functionality

7. **What is the purpose of the output projection layer in the MultiHeadAttention class?**
   A. To transform the combined output of the attention heads into a desired output dimension.
   B. To reduce the computational complexity of the attention mechanism.
   C. To ensure that the attention weights sum to 1.
   D. To make the model more robust to noise.

Questions by chapter section

Now we'll move through the chapter in more detail.

3.1 The problem with modeling long sequences

1. What is the main challenge in translating text from one language to another, and why can't we simply translate word by word?

2. Describe the role of the encoder and decoder in a language translation model.

3. What is the primary limitation of encoder-decoder RNNs in handling long sequences?

4. Explain the concept of a hidden state in the context of encoder-decoder RNNs.

3.2 Capturing data dependencies with attention mechanisms

1. What is the main limitation of RNNs for translating longer texts?

2. What is the significance of self-attention in the context of LLMs?

3. Match the term on the left to its description on the right

Attention Mechanism		A neural network architecture that relies on self-attention mechanisms to process sequential data, enabling it to handle long-range dependencies and outperform traditional RNNs in tasks like machine translation and text generation.
Self-Attention		A family of large language models based on the transformer architecture, known for their impressive capabilities in generating human-like text, translating languages, and performing various other language-related tasks.
Transformer Architecture		A technique that allows a neural network to focus on specific parts of an input sequence when generating an output, enabling it to capture long-range dependencies and improve performance on tasks like machine translation.
GPT Series		A type of attention mechanism where each element in a sequence can attend to all other elements in the same sequence, allowing the model to learn relationships and dependencies within the input.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

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

1. What is the purpose of the 'self' in self-attention?

2. Explain the concept of a context vector in self-attention.

3. What is the role of attention scores in self-attention?

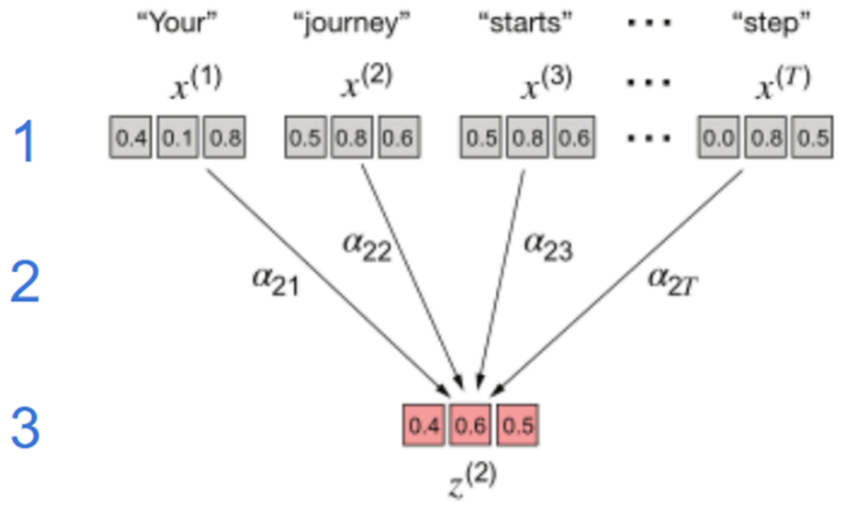
4. Why is normalization applied to attention scores?

5. What function has been removed from this code at position 1?

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

6. What is the purpose of the softmax function in self-attention?

7. The diagram shows a simple attention mechanism. What does each numbered stage represent?



8. Fill out the table with your answers:

Label	Description
1	
2	
3	

9. How are context vectors calculated using attention weights?

10. Match the term on the left to its description on the right:

Context Vector		Intermediate values calculated as dot products between the query element and all other input elements, representing the similarity or attention between them.
Attention Scores		Normalized attention scores that sum up to 1, representing the relative importance or contribution of each input element to the context vector.
Attention Weights		An enriched embedding vector that incorporates information about a specific input element and all other elements in the input sequence, creating a more comprehensive representation.

Fill the table with the column mappings:

Left Hand Column	1	2	3
Right Hand Column			

3.4 Implementing self-attention with trainable weights

1. Explain the purpose of the weight matrices Wq, Wk, and Wv in the self-attention mechanism.

2. Describe the process of computing attention scores and attention weights in the self-attention mechanism.

3. Why is the scaling by the square root of the embedding dimension (d_k) important in the scaled-dot product attention mechanism?

4. Pieces of the code have been removed from three places in this listing. Which of these terms have been removed and where should they go?

```
A softmax   B dim=-1   C W_value   D Linear   E torch
```

Fill out the table with your answers

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        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.___1___ = 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 = ___2___.softmax(
            attn_scores / keys.shape[-1]**0.5, ___3___
        )
        context_vec = attn_weights @ values
        return context_vec
```

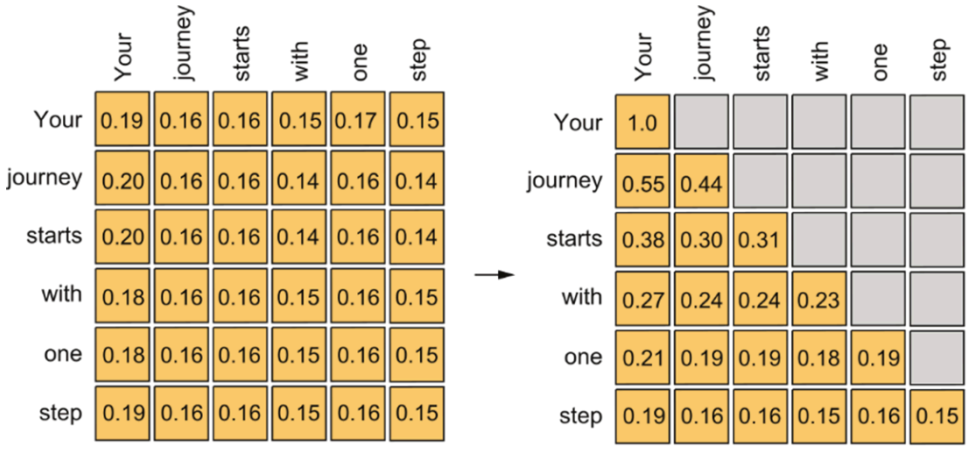
Position	1	2	3
Term			

5. What is the purpose of the `SelfAttention_v1` and `SelfAttention_v2` classes, and how do they differ in their implementation?

3.5 Hiding future words with causal attention

1. What is the purpose of causal attention, and how does it differ from standard self-attention?

2. What does this diagram show?



3. Describe the process of applying a causal attention mask to attention weights. What is the goal of this masking?

4. Explain the concept of information leakage in the context of causal attention and how it is addressed.

5. What is the purpose of dropout in the attention mechanism, and how is it applied in the context of causal attention?

6. What is the significance of the `register_buffer` method in the `CausalAttention` class?

7. Match the term on the left to its description on the right:

Causal Attention		Used to project input tokens into query, key, and value vectors, which are then used to compute attention scores and weights.
Trainable Weight Matrices		Used to selectively hide certain values during computation, often used to implement causal attention.
Mask		A specialized form of self-attention that restricts a model to only consider previous and current inputs in a sequence when processing any given token.

Fill the table with the column mappings:

Left Hand Column	1	2	3
Right Hand Column			

3.6 Extending single-head attention to multi-head attention

1. What is the main purpose of multi-head attention in LLMs?

2. How does the `MultiHeadAttentionWrapper` class implement multi-head attention?

3. Explain the key difference between the `MultiHeadAttentionWrapper` and the `MultiHeadAttention` class.

4. Pieces of the code have been removed from four places in this listing. Which of these terms have been removed and where should they go?

```
A 'mask'    B torch    C dropout    D -torch.inf    E W_value    F Linear
```

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, 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)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
              1  ,
           torch.triu(torch.ones(context_length, context_length),
           diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.   2   (x)

        attn_scores = queries @ keys.transpose(1, 2)
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens],     3      )
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        attn_weights = self.   4   (attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

Fill out the table with your answers

Position	1	2	3	4
Term				

5. What is the purpose of the output projection layer in the `MultiHeadAttention` class?

6. Why is the `MultiHeadAttention` class more efficient than the `MultiHeadAttentionWrapper`?

7. Match the term on the left to its description on the right:

Masked Attention		The process of dividing the attention mechanism into multiple independent heads, each with its own set of weights, to enhance pattern recognition and improve model performance.
Dropout		A technique used during training to randomly ignore hidden layer units, preventing overfitting by ensuring that a model does not become overly reliant on any specific set of units.
Multi-Head Attention		A specialized form of self-attention that restricts a model to only consider previous and current inputs in a sequence when processing any given token.

In Chapter Exercises

These are the exercises from Chapter 3. They are reproduced verbatim.

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.

Exercise 3.2 Returning two-dimensional embedding vectors

Change the input arguments for the MultiHeadAttentionWrapper(..., num_ heads=2) call such that the output context vectors are two-dimensional instead of four 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.

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 1,024 tokens.

Answers

Quick questions on main concepts

1. A
2. C
3. D
4. A
5. A
6. B
7. A

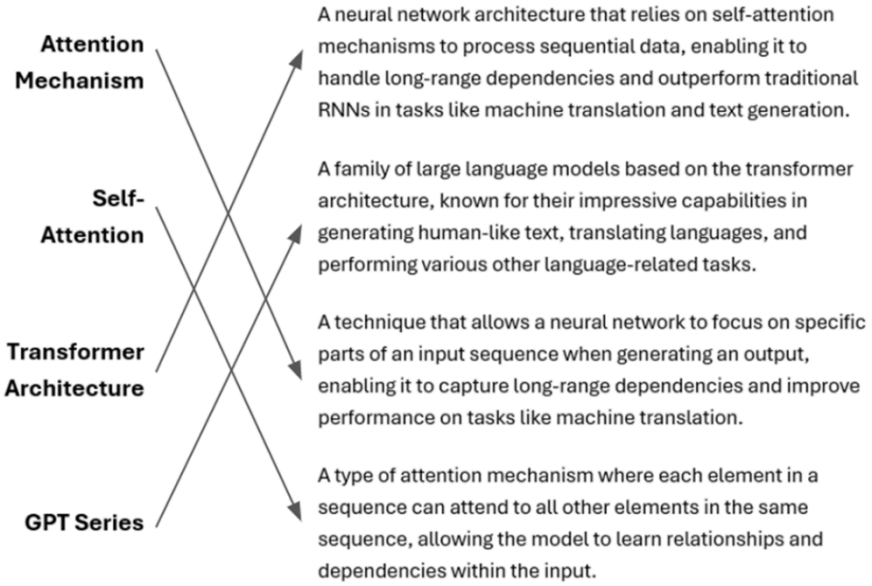
Questions by chapter section

3.1 The problem with modeling long sequences

1. The challenge lies in the grammatical structures of different languages. Direct word-by-word translation often fails to capture the meaning and context due to differences in sentence structure and word order.
2. The encoder processes the entire input text and encodes its meaning into a hidden state. The decoder then uses this hidden state to generate the translated text, one word at a time.
3. Encoder-decoder RNNs rely solely on the current hidden state during decoding, which can lead to a loss of context, especially when dependencies span long distances in complex sentences.
4. The hidden state is a compressed representation of the input sequence, capturing the meaning of the entire text. It acts as a memory cell that the decoder uses to generate the translated output.

3.2 Capturing data dependencies with attention mechanisms

1. RNNs struggle with long texts because they need to remember the entire encoded input in a single hidden state before decoding, making it difficult to retain information from earlier parts of the input.
2. Self-attention is a crucial component of contemporary LLMs based on the transformer architecture, such as the GPT series, enabling them to capture long-range dependencies and understand the relationships between words in a sentence.
3. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	3	4	1	2

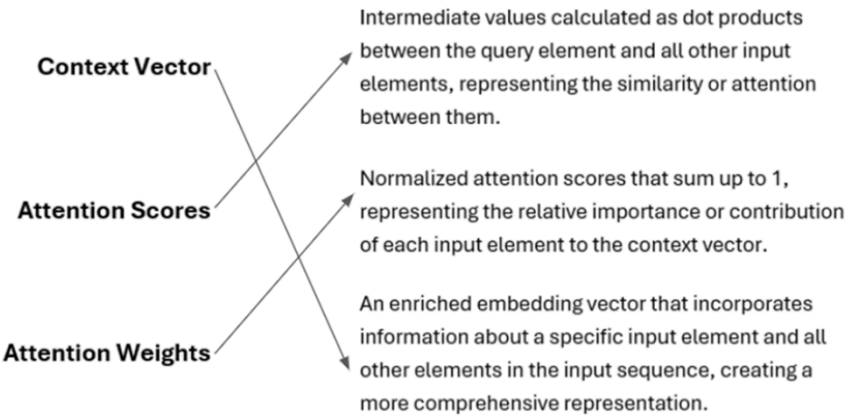
3.3 Attending to different parts of the input with self-attention

1. The 'self' in self-attention 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.
2. A context vector is an enriched embedding vector that incorporates information from all other elements in the input sequence. It represents an enhanced understanding of each element by considering its relationships with other elements.
3. Attention scores are intermediate values that represent the similarity between the query element and each other element in the input sequence. They are calculated using dot products and indicate the degree of attention or focus on each element.
4. Normalization is applied to attention scores to obtain attention weights that sum up to 1. This normalization is a convention that is useful for interpretation and maintaining training stability in an LLM.
5. The `dot()` function has been removed. This calculates the dot product of the query

6. The softmax function is used to normalize attention scores, ensuring that the attention weights are always positive and sum to 1. It provides a more stable and interpretable representation of attention weights as probabilities or relative importance.
7. The labels in the diagram represent (Figure 3.7):

Label	Description
1	Input vector corresponding to the first token
2	Attention weight to weigh the importance of input vector x (1)
3	The context vector z (2) is computed as a combination of all input vectors weighted with respect to input element x (2)

8. Context vectors are calculated by multiplying the embedded input tokens with their corresponding attention weights and then summing the resulting vectors. This weighted sum combines information from all input elements, creating an enriched representation of each element.
9. The terms are connected like this:



Left Hand Column	1	2	3
Right Hand Column	3	1	2

3.4 Implementing self-attention with trainable weights

1. These matrices are used to project the embedded input tokens into query, key, and value vectors, respectively. This projection allows the model to learn relationships between different parts of the input sequence and determine the importance of each input element for generating context vectors.
2. Attention scores are calculated by taking the dot product of the query vector with each key vector. These scores are then normalized using the softmax function to obtain attention weights, which represent the relative importance of each input element for the current query.
3. Scaling by the square root of d_k helps to prevent small gradients during backpropagation, especially when dealing with large embedding dimensions. This scaling ensures that the softmax function operates in a more stable range, leading to more effective model training.
4. The removed pieces map to the positions like this:

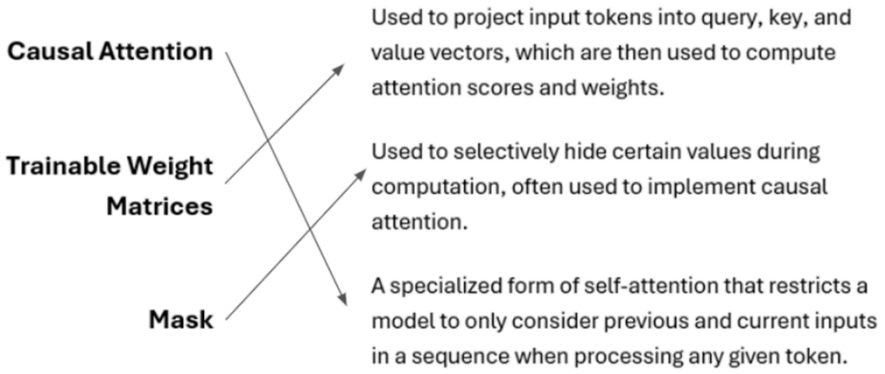
Position	1	2	3
Term	C	E	B

5. Both classes implement the self-attention mechanism. `SelfAttention_v1` uses manual weight initialization with `nn.Parameter`, while `SelfAttention_v2` utilizes `nn.Linear` layers for weight matrices, which offers optimized weight initialization and improved training stability.

3.5 Hiding future words with causal attention

1. Causal attention, also known as masked attention, restricts a model to consider only previous and current inputs in a sequence when processing any given token. This is in contrast to standard self-attention, which allows access to the entire input sequence at once.
2. The diagram shows that in causal attention, we mask out the attention weights above the diagonal so that for a given input, the LLM can’t access future tokens when computing the context vectors using the attention weights. (Figure 3.19)
3. The causal attention mask is applied to the attention weights by zeroing out the elements above the diagonal, effectively preventing the model from attending to future tokens. This ensures that the model's predictions are based solely on past and current information.
4. Information leakage occurs when masked positions still influence the softmax calculation. However, renormalizing the attention weights after masking effectively nullifies the effect of masked positions, ensuring that there is no information leakage from future tokens.

5. Dropout is a technique used to prevent overfitting by randomly dropping out hidden layer units during training. In causal attention, dropout is typically applied after calculating the attention weights, randomly zeroing out some of the weights and scaling up the remaining ones.
6. The `register_buffer` method ensures that the causal mask is automatically moved to the appropriate device (CPU or GPU) along with the model, avoiding device mismatch errors during training.
7. The terms are connected like this:



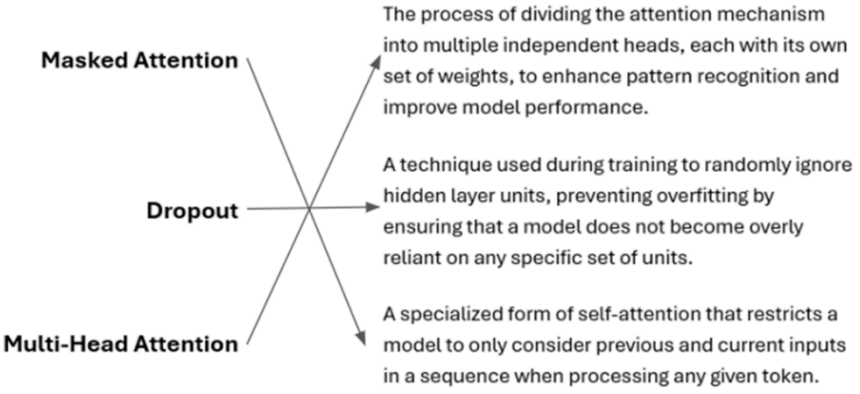
Left Hand Column	1	2	3
Right Hand Column	3	1	2

3.6 Extending single-head attention to multi-head attention

1. Multi-head attention allows LLMs to process information from different perspectives by running the attention mechanism multiple times with different learned linear projections. This enables the model to capture more complex patterns and relationships within the input data.
2. The `MultiHeadAttentionWrapper` class creates multiple instances of the `CausalAttention` module, each representing a separate attention head. It then combines the outputs from these heads by concatenating them.
3. The `MultiHeadAttentionWrapper` stacks multiple single-head attention modules, while the `MultiHeadAttention` class integrates multi-head functionality within a single class. The `MultiHeadAttention` class 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.
4. The removed pieces map to the positions like this:

Position	1	2	3	4
Term	A	E	D	C

5. The output projection layer in the `MultiHeadAttention` class is used to project the combined outputs from all attention heads back to the original embedding dimension. This layer is not strictly necessary but is commonly used in many LLM architectures.
6. The `MultiHeadAttention` class is more efficient because it performs matrix multiplications for the queries, keys, and values only once, instead of repeating them for each attention head as in the `MultiHeadAttentionWrapper`.
7. The terms are connected like this:



Left Hand Column	1	2	3
Right Hand Column	3	2	1

In Chapter Exercise Solutions

Exercise 3.1

The correct weight assignment is

```
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_heads=2)
```

Exercise 3.3

The initialization for the smallest GPT-2 model is

```
block_size = 1024
d_in, d_out = 768, 768
num_heads = 12
mha = MultiHeadAttention(d_in, d_out, block_size, 0.0, num_heads)
```

4 Implementing a GPT model from scratch to generate text

Chapter 4 focuses on implementing a **GPT-like large language model (LLM)** architecture, including its **transformer blocks** which consist of the **masked multi-head attention module** from the previous chapter. The chapter explains concepts such as **layer normalization**, **feed-forward networks**, and **shortcut connections**. This chapter also covers how to assemble a GPT model and use it to generate text one token at a time. The chapter uses the GPT-2 model with 124 million parameters as a reference, specifying the configurations and demonstrating how to instantiate the model.

All the answers to the questions can be found at the end of this document.

Quick questions on main concepts

1. **What is the purpose of the context_length parameter in the GPT_CONFIG_124M dictionary?**
   - A. It specifies the number of transformer blocks in the model.
   - B. It indicates the count of attention heads in the multi-head attention mechanism.
   - C. It represents the embedding size, transforming each token into a vector.
   - D. It denotes the maximum number of input tokens the model can handle via the positional embeddings.

2. **The primary purpose of layer normalization in a GPT model is to adjust the activations of a neural network layer to have a mean of 0 and a _____ of 1.**
   A. variance
   B. gradient
   C. weight

3. **What is the main advantage of layer normalization over batch normalization in LLMs?**
   A. Layer normalization is more suitable for handling sequential data than batch normalization.
   B. Layer normalization normalizes each input independently of the batch size
   C. Layer normalization is computationally more efficient than batch normalization.
   D. Layer normalization is more effective at preventing overfitting than batch normalization.

4. **Which activation function is commonly used in LLMs like GPT-2, offering a smoother alternative to ReLU?**
   A. ReLU (Rectified Linear Unit)
   B. GELU (Gaussian Error Linear Unit)
   C. Sigmoid
   D. Tanh

5. **What is the primary purpose of shortcut connections in a deep neural network?**
   A. To reduce the number of parameters in the model.
   B. To preserve the flow of gradients during the backward pass in training.
   C. To prevent overfitting by dropping out neurons during training.
   D. To increase the computational efficiency of the model.

6. **What are the main components of a transformer block in a GPT model?**
    A. Convolutional layers, pooling layers, and maxout activations.
    B. Linear layers, ReLU activations, and batch normalization.
    C. Multi-head attention, layer normalization, dropout, feed forward layers, and GELU activations.
    D. Token embedding, positional embedding, and dropout.

7. **What is the primary reason why the GPT model generates gibberish when it is not trained?**
    A. The model is not using the correct activation function.
    B. The model is not using the correct tokenizer.
    C. The model has not learned the relationships between words and patterns in language.
    D. The model is not using the correct dropout rate.

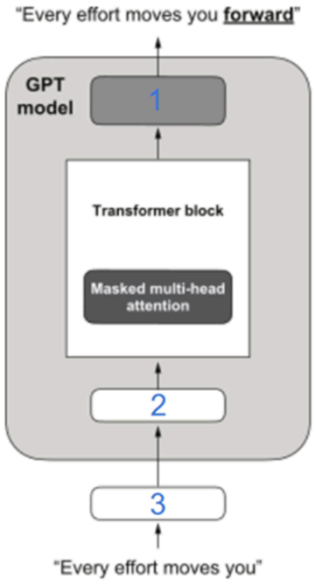
Questions by chapter section

Now we'll move through the chapter in more detail.

4.1 Coding an LLM architecture

1. What is the main purpose of a GPT model, and how does it achieve this?

2. Label this diagram:



Fill out the table with your answers

Label	1	2	3
Description			

3. What are the key components of a GPT model, and how do they contribute to the model's functionality?

4. Explain the concept of 'parameters' in the context of LLMs like GPT.

5. What are the main differences between GPT-2 and GPT-3, and why is GPT-2 a better choice for learning LLM implementation?

6. Describe the purpose of the `GPT_CONFIG_124M` dictionary and explain the meaning of each key-value pair.

7. What is the role of the `DummyGPTModel` class in the code, and how does it contribute to the overall implementation of the GPT model?

4.2 Normalizing activations with layer normalization

1. What is the main purpose of layer normalization in neural networks, and how does it contribute to improved training dynamics?

2. Describe the typical placement of layer normalization within GPT-2 and modern transformer architectures.

3. Pieces of the code have been removed from three places in this listing. Which of these terms have been removed and where should they go?

`A mean` `B zeroes` `C norm_x` `D ones` `E scale` `F zeros`

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.__1__(emb_dim))
        self.shift = nn.Parameter(torch.__2__(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 * __3__ + self.shift
```

Fill out the table with your answers

Position	1	2	3
Term			

4. Explain the difference between biased and unbiased variance calculation, and why is the biased approach preferred in the context of LLMs?

5. What are the key differences between layer normalization and batch normalization, and why is layer normalization often favored in LLMs?

6. Match the term on the left to its description on the right:

Transformer Block		The trainable weights of a neural network model, which are adjusted during the training process to minimize a specific loss function.
Layer Normalization		The output of a neural network model before applying the softmax function, representing the unnormalized probabilities of each possible output class.
Parameters		The core building block of a GPT model, consisting of a masked multi-head attention module and a feedforward neural network, which are applied sequentially to the input data.
Logits		A technique used to normalize the output of each layer in a neural network, ensuring that the data has a mean of zero and a standard deviation of one, which helps to improve the stability and performance of the model.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

4.3 Implementing a feed forward network with GELU activations

1. What is the GELU activation function and how does it differ from the ReLU activation function?

2. Explain the purpose and structure of the `FeedForward` module in the context of an LLM.

3. Pieces of the code have been removed from two places in this listing. Which of these terms have been removed and where should they go?

A RELU **B** Linear **C** Embedding **D** GELU **E** Sequential

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.  1  (
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            2 (),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

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

Fill out the table with your answers

Position	1	2
Term		

4. How does the `FeedForward` module contribute to the model's ability to learn and generalize data?

5. What is the significance of the `FeedForward` module having the same input and output dimensions?

4.4 Adding shortcut connections

1. What is the vanishing gradient problem and how does it affect training deep neural networks?

2. Explain the concept of shortcut connections and how they address the vanishing gradient problem.

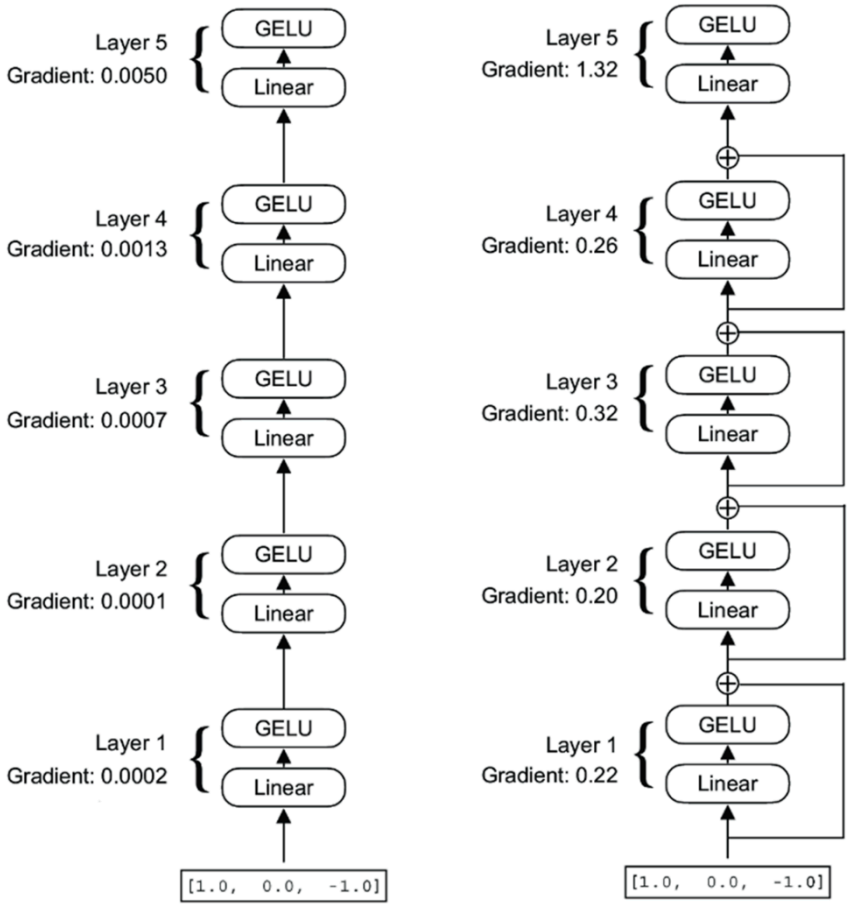
3. How are shortcut connections implemented in this code listing:

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]),
                          GELU())
        ])

    def forward(self, x):
        for layer in self.layers:
            layer_output = layer(x)
            if self.use_shortcut and x.shape == layer_output.shape:
                x = x + layer_output
            else:
                x = layer_output
        return x
```

4. What is the purpose of the `print_gradients` function and how does it demonstrate the effectiveness of shortcut connections?

5. This diagram shows (on the left) a deep neural network consisting of five layers and one with shortcut connections (on the right). Why are the values of the gradients bigger in the network on the right?



4.5 Connecting attention and linear layers in a transformer block

1. What are the key components of a transformer block and how do they contribute to the processing of input sequences?

2. Explain the concept of Pre-LayerNorm and its significance in the transformer block architecture.

3. Describe the role of shortcut connections in the transformer block and their impact on gradient flow.

4. How does the transformer block preserve the input dimensions in its output?

5. Explain how the transformer block integrates contextual information from the entire input sequence into its output.

4.6 Coding the GPT model

1. What is the purpose of the `GPTModel` class and how does it relate to the `TransformerBlock` class?

2. Explain the role of the `LayerNorm` layer in the `GPTModel` architecture.

3. What is weight tying and how does it affect the number of parameters in a GPT model?

4. Describe the process of converting the output of the `GPTModel` into text.

5. Pieces of the code have been removed from three places in this listing. Which of these terms have been removed and where should they go?

```
A Embedding   B Dropout   C LayerNorm   D Linear   E Sequential
```

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.___1___(cfg["drop_rate"])

        self.trf_blocks = nn.___2___ (
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.___3___(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )
```

Fill out the table with your answers

Position	1	2	3
Term			

6. How does the number of transformer blocks affect the complexity and performance of a GPT model?

7. Match the terms on the left to the descriptions on the right:

Multi-Head Attention		a technique that normalizes the activations of a layer to have a mean of zero and a standard deviation of one, improving the stability and performance of the model.
Layer Normalization		allows the gradient to flow directly from the input to the output of a layer, preventing the vanishing gradient problem and enabling the training of deeper models.
Shortcut Connection		allows the model to attend to different parts of the input sequence simultaneously, capturing complex relationships between words and phrases.

Fill the table with the column mappings:

Left Hand Column	1	2	3
Right Hand Column			

4.7 Generating text

1. Explain the process by which a GPT model generates text, starting from its output tensors.

2. Describe the role of the `softmax` function in the text generation process.

3. What is the purpose of the `generate_text_simple` function, and how does it work?

4. Why is the `softmax` step technically redundant in the generate_text_simple function?

5. What is the significance of the greedy decoding approach in text generation?

6. Why does the GPT model generate gibberish when it hasn't been trained?

7. Put these steps to implement the GPTModel architecture into the correct order:
   A. Initialize token and positional embeddings, dropout, and a linear output layer.

B. Implement the forward pass, combining embeddings, transformer blocks, layer normalization, and the output layer.
C. Create a sequential container for TransformerBlock instances.
D. Create a GPTModel class inheriting from nn.Module.

Fill the table with your answers:

Step Order	Step
1	
2	
3	
4	

In Chapter Exercises

These are the exercises from Chapter 4. They are reproduced verbatim.

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.

Exercise 4.2 Initializing larger GPT models

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 1,024-dimensional embeddings, 24 transformer blocks, 16 multi-head attention heads), GPT-2 large (1,280-dimensional embeddings, 36 transformer blocks, 20 multi-head attention heads), and GPT-2 XL (1,600-dimensional embeddings, 48 transformer blocks, 25 multi-head attention heads). As a bonus, calculate the total number of parameters in each GPT model.

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.)

Answers

Multiple Choice

1. D
2. A
3. B
4. B
5. B
6. B
7. C
8. C

Answers by section

4.1 Coding an LLM architecture

1. These are the diagram labels:

Label	1	2	3
Description	Output layers	Enbedding layers	Tokenized text

2. A GPT model is designed to generate new text one word at a time. It achieves this by using a large deep neural network architecture that learns patterns from a massive dataset of text and then uses these patterns to predict the next word in a sequence.
3. The key components of a GPT model include embedding layers, transformer blocks, and an output layer. Embedding layers convert words into numerical representations, transformer blocks process these representations to capture relationships between words, and the output layer predicts the probability of each word in the vocabulary.
4. Parameters in LLMs refer to the trainable weights of the model. These weights are adjusted during training to minimize a loss function, allowing the model to learn from the training data and improve its ability to generate text.
5. GPT-3 is a larger model with more parameters and trained on more data than GPT-2. However, GPT-2 is more suitable for learning LLM implementation because its pretrained weights are publicly available, and it can be run on a single laptop computer, unlike GPT-3, which requires a GPU cluster.

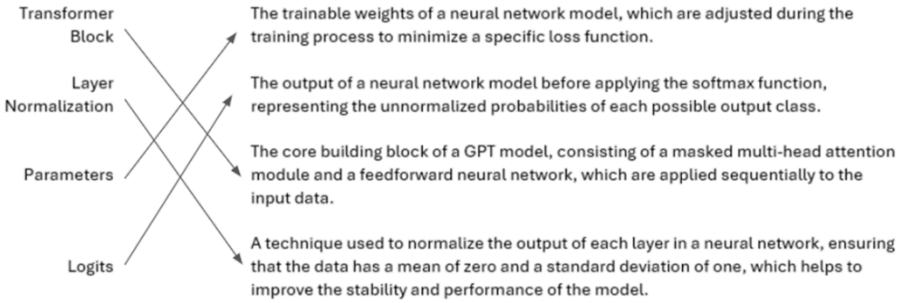
6. The `GPT_CONFIG_124M` dictionary defines the configuration of the small GPT-2 model. It includes parameters like vocabulary size, context length, embedding dimension, number of attention heads, number of layers, dropout rate, and query-key-value bias, which determine the model's architecture and behavior.
7. The `DummyGPTModel` class provides a placeholder architecture for the GPT model, outlining the main components and their order. It serves as a starting point for building the full GPT model by defining the data flow and providing a framework for implementing the individual components.

4.2 Normalizing activations with layer normalization

1. Layer normalization aims to stabilize and accelerate neural network training by adjusting the activations of a layer to have a mean of 0 and a variance of 1. This normalization process helps prevent vanishing or exploding gradients, ensuring consistent and reliable training.
2. Layer normalization is commonly applied both before and after the multi-head attention module in GPT-2 and modern transformer architectures. It is also applied before the final output layer.
3. The removed pieces map to the positions like this

Position	1	2	3
Term	D	F	C

4. Biased variance calculation divides by the number of inputs *n*, while unbiased variance uses *n* – *1* in the denominator to correct for bias. In LLMs, where the embedding dimension *n* is large, the difference between these approaches is negligible, and the biased method is preferred for compatibility with GPT-2's normalization layers and TensorFlow's default behavior.
5. Layer normalization normalizes across the feature dimension, while batch normalization normalizes across the batch dimension. Layer normalization offers greater flexibility and stability, especially in scenarios with varying batch sizes or resource constraints, making it suitable for LLMs.
6. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	3	4	1	2

4.3 Implementing a feed forward network with GELU activations

1. The GELU activation function is a smooth, nonlinear function that approximates ReLU but with a non-zero gradient for almost all negative values. Unlike ReLU, which outputs zero for any negative input, GELU allows for a small, non-zero output for negative values.
2. The `FeedForward` module is a small neural network consisting of two linear layers and a GELU activation function. It expands the embedding dimension into a higher-dimensional space, applies a nonlinear transformation, and then contracts back to the original dimension, allowing for richer representation learning.
3. The removed pieces map to the positions like this

Position	1	2
Term	E	D

4. The `FeedForward` module enhances the model's ability to learn and generalize data by exploring a richer representation space through the expansion and contraction of the embedding dimension. This allows the model to capture more complex relationships within the data.
5. The uniformity in input and output dimensions simplifies the architecture by enabling the stacking of multiple layers without the need to adjust dimensions between them, making the model more scalable.

4.4 Adding shortcut connections

1. The vanishing gradient problem occurs when gradients become progressively smaller as they propagate backward through the layers of a deep neural network, making it difficult to effectively train earlier layers. This can hinder the learning process and prevent the model from achieving optimal performance.
2. Shortcut connections, also known as skip connections, create alternative paths for gradients to flow through the network by bypassing certain layers. This helps preserve the flow of gradients during the backward pass, mitigating the vanishing gradient problem and enabling more effective training of deeper networks.
3. In the code, shortcut connections are implemented by adding the output of a layer to the output of a later layer. This is done conditionally based on the `use_shortcut` attribute, allowing for the inclusion or exclusion of shortcut connections during the forward pass.
4. The `print_gradients` function calculates and displays the mean absolute gradient values for each layer in the model. By comparing the gradient values of models with and without shortcut connections, we can see that shortcut connections help maintain a more consistent gradient flow across layers, preventing the gradients from vanishing in earlier layers.
5. The bigger gradient values on the right are because of the shortcut connections -these stop the gradient values from getting vanishingly small through the layers.

4.5 Connecting attention and linear layers in a transformer block

1. A transformer block consists of multi-head attention, feed forward layers, layer normalization, and dropout. The multi-head attention analyzes relationships between elements in the input sequence, while the feed forward network modifies the data individually at each position. Layer normalization ensures consistent scaling, and dropout prevents overfitting.
2. Pre-LayerNorm refers to applying layer normalization before the multi-head attention and feed forward layers. This approach, unlike Post-LayerNorm, has been shown to improve training dynamics and lead to better performance in transformer models.
3. Shortcut connections add the input of the block to its output, allowing gradients to flow more easily through the network during training. This helps to prevent vanishing gradients and improves the learning of deep models.
4. The transformer block maintains the input dimensions by applying operations that do not alter the shape of the input sequence. This allows for a one-to-one correspondence between input and output vectors, enabling its use in various sequence-to-sequence tasks.

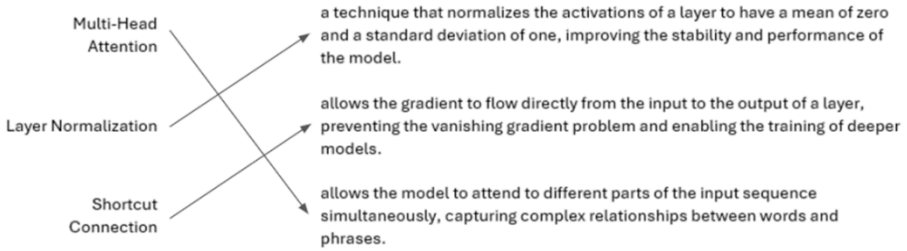
5. While the transformer block preserves the physical dimensions of the input sequence, the content of each output vector is re-encoded to incorporate contextual information from across the entire input sequence. This allows the model to capture complex relationships between elements in the sequence.

4.6 Coding the GPT model

1. The `GPTModel` class assembles the complete GPT architecture, utilizing the previously defined `TransformerBlock` class as a building block. It combines token and positional embeddings, applies multiple `TransformerBlock` layers, and finally projects the output into the vocabulary space to predict the next token.
2. The `LayerNorm` layer is applied after the transformer blocks to normalize the output, ensuring that the data has a consistent scale and distribution. This helps to stabilize the learning process and improve the model's performance.
3. Weight tying is a technique where the weights of the token embedding layer are reused in the output layer. This reduces the number of trainable parameters, leading to a smaller model footprint and potentially faster training.
4. The output of the `GPTModel` is a tensor of shape `[batch_size, num_tokens, vocab_size]`, representing the logits for each token in the vocabulary. To convert these logits into text, we need to apply a softmax function to obtain probabilities, then select the token with the highest probability for each position in the sequence.
5. The removed pieces map to the positions like this:

Position	1	2	3
Term	B	E	D

6. Increasing the number of transformer blocks in a GPT model generally leads to a larger model with more parameters, requiring more computational resources. However, it can also improve the model's ability to capture long-range dependencies in the input text, potentially leading to better performance on tasks like text generation.
7. The terms are connected like this:



Left Hand Column	1	2	3
Right Hand Column	3	1	2

4.7 Generating text

1. The GPT model converts its output tensors into text by decoding them, selecting tokens based on a probability distribution derived from the softmax function, and then converting these tokens back into human-readable text.
2. The `softmax` function transforms the output logits into a probability distribution, where each value represents the likelihood of a particular token being the next in the sequence. This allows the model to select the most probable token for generation.
3. The `generate_text_simple` function implements a simple generative loop for a language model. It iteratively predicts the next token based on the current context, appends it to the input sequence, and repeats this process until a specified number of new tokens are generated.
4. The `softmax` function is monotonic, meaning it preserves the order of its inputs. Therefore, applying `torch.argmax` directly to the logits tensor would yield the same result as applying it to the softmax output, as the position of the highest value remains unchanged.
5. Greedy decoding refers to the strategy of always selecting the most likely token at each step. While this approach can be efficient, it can also lead to repetitive or predictable text, as the model always chooses the most obvious continuation.
6. The model generates incoherent text because it has not yet learned the relationships between words and their contexts. Training is crucial for the model to develop the ability to generate meaningful and coherent text.
7. Here is the correct order of the steps:

Step Order	Step
1	D
2	A
3	C
4	B

In Chapter Exercise Solutions

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_params:,}")

total_params = sum(p.numel() for p in block.att.parameters())
print(f"Total number of parameters in attention module: {total_params:,}")
```

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

```
gpt2-xl:
Total number of parameters: 1,637,792,000
Number of trainable parameters considering weight tying: 1,557,380,800
Total size of the model: 6247.68 MB
```

Exercise 4.3

There are three distinct places in chapter 4 where we used dropout layers: the embedding layer, shortcut layer, and multi-head attention module. We can control the dropout rates for each of the layers by coding them separately in the config file and then modifying the code implementation accordingly.

The modified configuration is as follows:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 1024,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate_attn": 0.1,
    "drop_rate_shortcut": 0.1,
    "drop_rate_emb": 0.1,
    "qkv_bias": False
}
```

The modified TransformerBlock and GPTModel look like

```
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate_attn"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(
            cfg["drop_rate_shortcut"]
        )
```

```
    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut

        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        return x


class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"]
        )
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"]
        )
        self.drop_emb = nn.Dropout(cfg["drop_rate_emb"])

        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)
        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        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 logitss
```

5 Pretraining on unlabeled data

Chapter 5 focuses on **pretraining a large language model (LLM)** and evaluating its performance using techniques **like calculating training** and **validation losses**. The chapter also explores different decoding strategies, including **temperature scaling** and **top-k sampling**, to control the randomness and enhance the originality of generated text. Additionally, it covers practical steps for saving and loading model weights, allowing users to resume training or load pretrained weights from sources like OpenAI's GPT model. These steps are crucial for developing and fine-tuning LLMs for various downstream tasks.

All the answers to the questions can be found at the end of this document.

Quick questions on main concepts

1. **What is the primary purpose of the cross-entropy loss function in LLM training?**
   - A. To generate text samples for evaluation purposes.
   - B. To prevent the model from overfitting to the training data.
   - C. To evaluate the model's performance on a specific task, such as text classification.
   - D. To measure the difference between the model's predicted probability distribution of tokens and the actual distribution of tokens in the training data.

2. **What is the purpose of temperature scaling in text generation?**

A. To control the randomness and diversity of the generated text by adjusting the probability distribution of tokens.
B. To improve the model's accuracy in predicting the next token.
C. To reduce the computational cost of text generation.
D. To prevent the model from overfitting to the training data.

3. **What is the primary goal of pretraining a large language model (LLM)?**
   A. To improve the model's ability to generate coherent and grammatically correct text.
   B. To fine-tune the model for a specific task, such as text classification.
   C. To learn general language patterns and representations from a massive amount of text data.
   D. To reduce the computational cost of training the model.

4. **What is the main benefit of using pretrained weights from OpenAI for LLMs?**
   A. It eliminates the need for extensive and expensive pretraining from scratch
   B. It reduces the risk of overfitting to the training data.
   C. It guarantees that the model will perform well on any task.
   D. It simplifies the process of fine-tuning the model for specific tasks.

5. **What is the primary advantage of saving the model's state dictionary in PyTorch?**
   A. It improves the model's performance on unseen data.
   B. It allows you to load and reuse the trained model without retraining it from scratch.
   C. It prevents the model from overfitting to the training data.
   D. It reduces the computational cost of training the model.

Questions by chapter section

Now we'll move through the chapter in more detail.

5.1 Evaluating generative text models

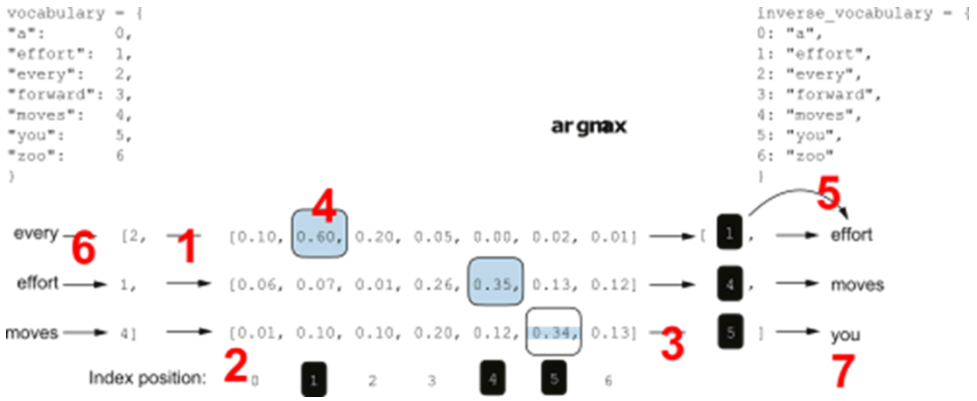
1. What is the purpose of the `generate_text_simple` function and how does it work?

2. Explain the concept of text generation loss and its significance in evaluating the quality of generated text.

3. Describe the role of backpropagation in training an LLM and how it relates to the text generation loss.

4. What is cross entropy loss and how is it used in evaluating LLMs?

5. Label this diagram by assigning the label numbers to the matching description:



Here are the descriptions for you to assign the label numbers to – any shown in italics are already correctly assigned:

Label	Description
1	Predicted token IDs are index positions with highest probability
2	Input text
3	Map input text to token IDs with vocabulary
4	Output text generated by LLM
5	Map index positions back into text with inverse vocabulary
6	Seven-dimensional probability row vector for each input vector
7	Index position with highest probability from argmax function

6. Explain the concept of perplexity and its relationship to cross entropy loss.

7. Pieces of the code have been removed from two places in this listing. Which of these terms have been removed and where should they go?

`A softmax` `B tensor` `C no_grad` `D grad`

```
with torch.____1____():
    logits = model(inputs)
probas = torch.____2____(logits, dim=-1)
print(probas.shape)
```

Fill out the table with your answers

Position	1	2
Term		

5.2 Training an LLM

1. What is the purpose of the `train_model_simple` function and what are its key components?

2. Explain the role of the `evaluate_model` function in the training process.

3. Pieces of the code have been removed from four places in this listing. Which of these terms have been removed and where should they go?

A `tokens_seen` **B** `train_losses` **C** `Epochs` **D** `Loss` **E** `epochs_seen`

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(
        epochs_seen, val_losses, linestyle="-.", label="Validation loss"
    )
    ax1.set_xlabel("    1    ")
    ax1.set_ylabel("    2    ")
    ax1.legend(loc="upper right")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax2 = ax1.twiny()
    ax2.plot(          3          ,          4          , alpha=0)
    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)
```

Fill out the table with your answers

Position	1	2	3	4
Term				

4. What is the purpose of the `generate_and_print_sample` function and how does it work?

5. What is AdamW and why is it preferred over Adam for training LLMs?

6. What is the significance of the training and validation loss curves in Figure 5.12?

7. Match the terms on the left with the description on the right:

Text Generation Loss		a common measure in machine learning that quantifies the difference between two probability distributions, typically the true distribution of labels and the predicted distribution from a model.
Cross Entropy Loss		a standard technique for training deep neural networks that involves updating the model's weights to minimize the difference between the model's predicted output and the actual desired output.
Perplexity		a measure used alongside cross entropy loss to evaluate the performance of models in tasks like language modeling, providing a more interpretable way to understand the uncertainty of a model in predicting the next token in a sequence.
Backpropagation		a numerical measure used to assess the quality of text generated during training, indicating how well the model's predictions align with the desired target text.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

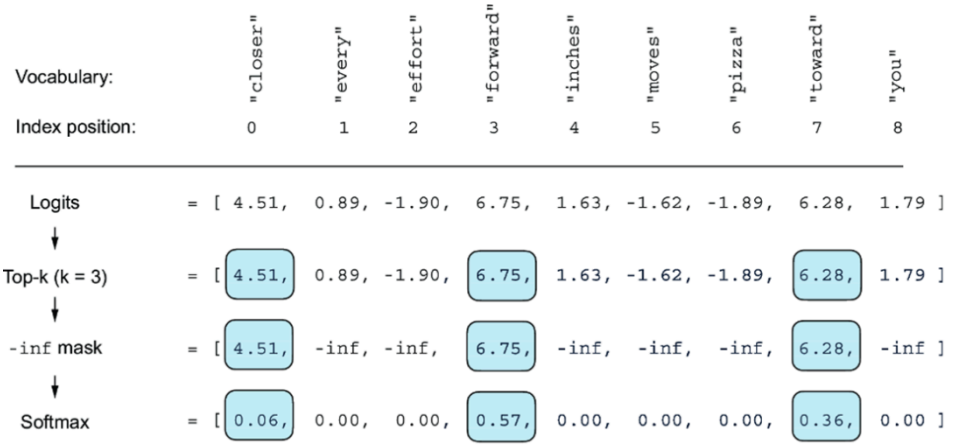
5.3 Decoding strategies to control randomness

1. What is the purpose of temperature scaling in text generation?

2. Explain how top-k sampling works and its benefit in text generation.

3. Describe the difference between greedy decoding and probabilistic sampling in text generation.

4. What is the purpose of the Inf Masking step in the top-k sampling process?



A. To assign probabilities to all words in the vocabulary
B. To select the top k logits with the highest scores
C. To set logits not in the top-k to negative infinity, removing them from consideration
D. To convert logits into probabilities using the softmax function

5. How does the `generate` function incorporate temperature scaling and top-k sampling?

5.4 Loading and saving model weights in PyTorch

1. Why is it important to save the model weights after training a large language model?

2. What is the recommended way to save a PyTorch model's weights?

3. What is the purpose of the `model.eval()` method in PyTorch?

4. Why is it important to save the optimizer state when saving a model?

5. Pieces of the code have been removed from two places in this listing. Which of these terms have been removed and where should they go?

A Adam **B** load **C** save **D** AdamW **E** AdamWest

```
checkpoint = torch.__1__("model_and_optimizer.pth", map_location=device)
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.__2__(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

Fill out the table with your answers

Position	1	2
Term		

6. Match the terms on the left with the description on the right:

AdamW		the process of iterating over the training data multiple times, calculating the loss for each batch, and updating the model weights to minimize the loss
Training Loop		a variant of the Adam optimizer that improves the weight decay approach, which aims to minimize model complexity and prevent overfitting by penalizing larger weights
Overfitting		the loss calculated on a separate dataset that is not used for training, which provides an estimate of the model's performance on unseen data
Validation Loss		a phenomenon that occurs when a model learns the training data too well and performs poorly on unseen data

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

5.5 Loading pretrained weights from OpenAI

1. What is the primary advantage of using pretrained weights from OpenAI for a GPT-2 model?

2. What are the key components of the `settings` and `params` dictionaries obtained from OpenAI's GPT-2 model weights?

3. How are the pretrained weights from OpenAI loaded into a custom `GPTModel` instance?

4. What is the significance of the `model_configs` dictionary in the context of loading pretrained weights?

5. Why is it necessary to update the `NEW_CONFIG` dictionary with `context_length` and `qkv_bias` settings?

6. Match the terms on the left with the description on the right:

State Dict		a dictionary containing the parameters of each layer in a PyTorch model.
Model Weights		a dictionary containing the internal state of the optimizer, such as the learning rate and momentum.
Optimizer State		the parameters that are learned during the training process and are used to make predictions.
Evaluation Mode		a mode in which the model is used for inference, and dropout layers are disabled.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

In Chapter Exercises

These are the exercises from Chapter 5. They are reproduced verbatim.

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 5.14. 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?

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? Likewise, 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?

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 one more epoch using the train_model_simple function.

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

Experiment with GPT-2 models of different sizes—for example, the largest 1,558 million parameter model—and compare the generated text to the 124 million model.

Answers

Quick questions on main concepts

1. D
2. A
3. C
4. A
5. B

Questions by chapter section

5.1 Evaluating generative text models

1. The `generate_text_simple` function generates text by taking a starting context, converting it to token IDs, feeding it into the GPT model, and then converting the model's output logits back into token IDs, which are then decoded into text.
2. Text generation loss is a numerical measure used to evaluate the quality of generated text. It quantifies the difference between the model's predicted output (token probabilities) and the actual desired output (target tokens). A lower loss indicates better text generation quality.

3. Backpropagation is a technique used to update the model's weights during training. It uses the text generation loss to adjust the weights so that the model produces outputs closer to the target tokens, thereby minimizing the loss and improving the quality of generated text.
4. Cross entropy loss is a measure of the difference between two probability distributions, typically the true distribution of labels (tokens) and the predicted distribution from the model. It is used to evaluate the performance of LLMs by quantifying how well the model's predicted probability distribution matches the actual distribution of tokens in the dataset.
5. This is the correct assignment of the descriptions to labels:

Label	Description
1	Map input text to token IDs with vocabulary
2	Seven-dimensional probability row vector for each input vector
3	Predicted token IDs are index positions with highest probability
4	Index position with highest probability from argmax function
5	Map index positions back into text with inverse vocabulary
6	Input text
7	Output text generated by LLM

6. Perplexity is a measure derived from cross entropy loss that provides a more interpretable way to understand the uncertainty of a model in predicting the next token. It represents the effective vocabulary size about which the model is uncertain at each step. A lower perplexity indicates less uncertainty and better model performance.
7. The removed pieces map to the positions like this:

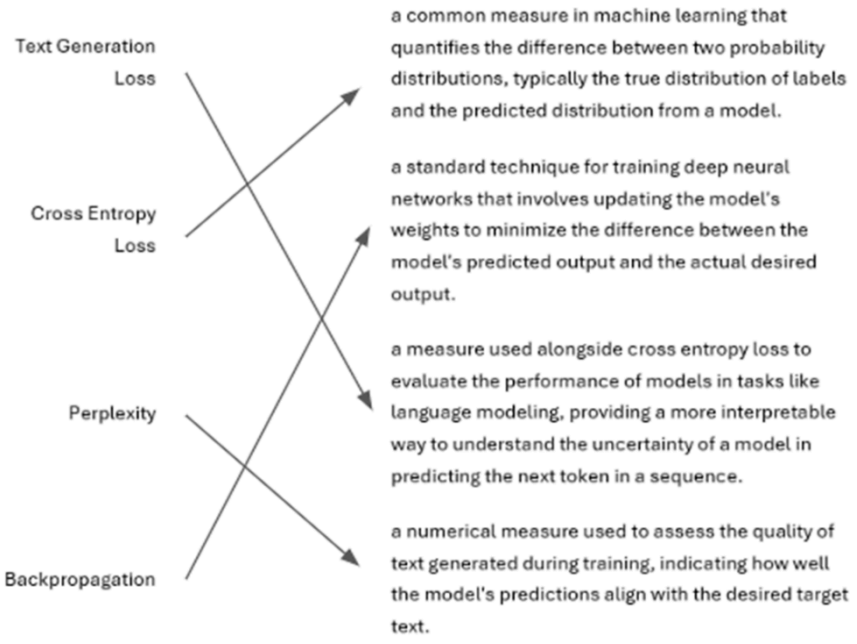
Position	1	2
Term	C	A

5.2 Training an LLM

1. The `train_model_simple` function implements a basic training loop for the LLM. It iterates over epochs, processes batches, calculates loss, updates weights, and evaluates the model's performance using validation data.
2. The `evaluate_model` function calculates the loss on both the training and validation sets to assess the model's performance. It ensures the model is in evaluation mode, disabling gradient tracking and dropout for accurate evaluation.
3. The removed pieces map to the positions like this:

Position	1	2	3	4
Term	C	D	A	B

4. The `generate_and_print_sample` function generates text samples from the model to visually assess its progress during training. It takes a text snippet as input, converts it to token IDs, and uses the `generate_text_simple` function to generate new text.
5. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	3	1	4	2

6. AdamW is a variant of the Adam optimizer that improves weight decay, a technique that penalizes larger weights to prevent overfitting. This makes AdamW more effective for regularizing and generalizing LLMs.
7. The curves show that the model initially learns well, but the training loss continues to decrease while the validation loss stagnates. This indicates overfitting, where the model memorizes the training data instead of generalizing to new data.

5.3 Decoding strategies to control randomness

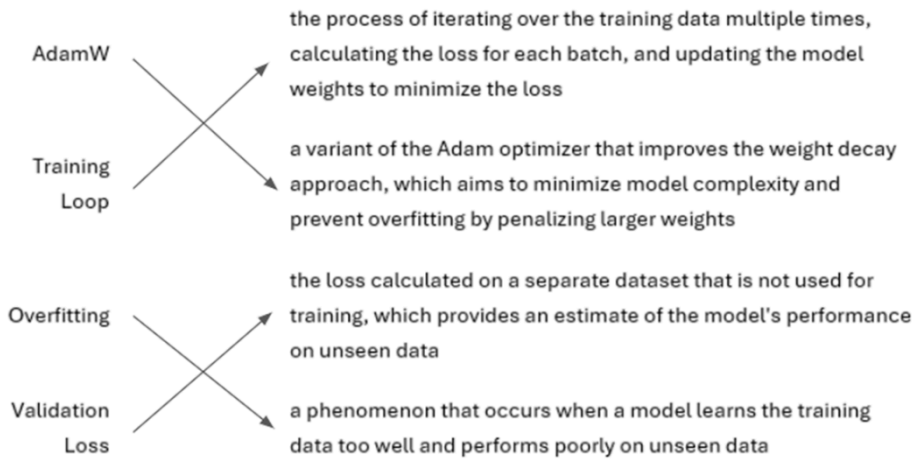
1. Temperature scaling adjusts the probability distribution of the next token by dividing the logits by a temperature value. Higher temperatures create a more uniform distribution, leading to more diverse outputs, while lower temperatures sharpen the distribution, favoring the most likely tokens.
2. Top-k sampling selects only the top-k most likely tokens for the next token prediction. This helps to reduce the generation of nonsensical or grammatically incorrect text by focusing on the most probable options.
3. Greedy decoding always selects the token with the highest probability, while probabilistic sampling chooses tokens based on their probability distribution. Probabilistic sampling introduces randomness and can lead to more diverse outputs.
4. The correct answer is C, "To set logits not in the top-k to negative infinity, removing them from consideration"
5. The `generate` function allows for the specification of temperature and top-k values. If a temperature is provided, the logits are scaled accordingly. If a top-k value is given, the logits are masked to exclude tokens outside the top-k most likely options.

5.4 Loading and saving model weights in PyTorch

1. Saving the model weights allows you to reuse the trained model without retraining it from scratch, saving significant computational time and resources. This is especially important for large language models, which can take a long time to train.
2. The recommended way is to save the model's `state_dict`, which is a dictionary mapping each layer to its parameters, using the `torch.save` function. This allows you to easily load the weights into a new model instance later.
3. The `model.eval()` method switches the model to evaluation mode, disabling dropout layers. This is important for inference, as we don't want to randomly drop out information during prediction.
4. Saving the optimizer state allows you to resume training from where you left off. This is important for adaptive optimizers like AdamW, which store historical data to adjust learning rates dynamically. Without the optimizer state, the model may learn suboptimally or fail to converge properly.
5. The removed pieces map to the positions like this:

Position	1	2
Term	B	D

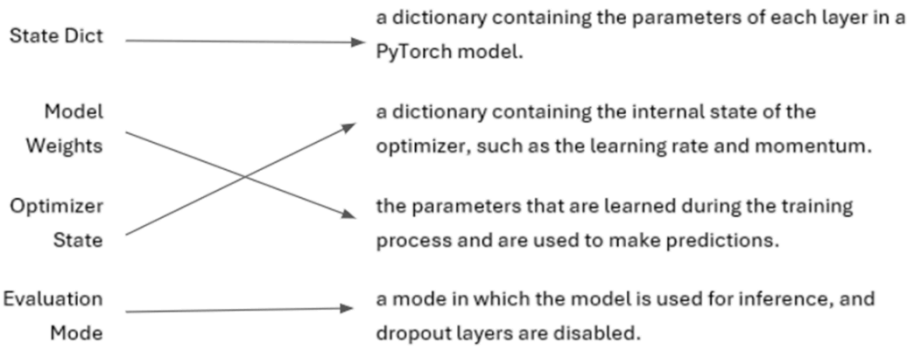
6. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	2	1	4	3

5.5 Loading pretrained weights from OpenAI

1. Using pretrained weights from OpenAI eliminates the need for extensive training on a large corpus, saving significant time and computational resources.
2. The `settings` dictionary contains the LLM architecture settings, while the `params` dictionary holds the actual weight tensors for the model's layers.
3. The `load_weights_into_gpt` function carefully matches the weights from OpenAI's implementation with the corresponding layers in the custom `GPTModel` instance, ensuring consistency and functionality.
4. The `model_configs` dictionary provides the specific architectural settings for different GPT-2 model sizes, allowing for the selection and loading of weights for the desired model.
5. The pretrained GPT-2 models from OpenAI were trained with a different `context_length` and used bias vectors in the attention module, requiring these settings to be updated for compatibility.
6. The terms are connected like this:



Left Hand Column	1	2	3	4
Right Hand Column	1	3	2	4

In Chapter Exercise Solutions

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 is 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 when 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:

```
Setting to top_k=None and applying no temperature scaling
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_decay=0.1)
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 124-million 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:

"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 training and validation set portions of "The Verdict." (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).

"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 124-million 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 1,558 million instead of 124 million model weights in chapter 5, the only two lines of code that we have to change are the following:

```
hparams, params = download_and_load_gpt2(model_size="124M", models_dir="gpt2")
model_name = "gpt2-small (124M)"
```

The updated code is

```
hparams, params = download_and_load_gpt2(model_size="1558M", models_dir="gpt2")
model_name = "gpt2-xl (1558M)"
```

6 Fine-tuning for classification

Chapter 6 focuses on **classification fine-tuning**, a technique to adapt a pretrained LLM for specific classification tasks, such as identifying spam messages. The chapter guides readers through preparing a text message dataset, modifying the pretrained LLM by replacing the output layer for classification, and implementing a training function to fine-tune the model for spam classification. It also covers evaluating the fine-tuned model's accuracy, demonstrating its use in classifying new text messages as either spam or not spam.

All the answers to the questions can be found at the end of this document.

Quick questions on main concepts

1. **The primary purpose of _____ is to train a model to recognize and predict specific class labels.**
   - A. classification-finetuning
   - B. regression
   - C. clustering

2. **What are the two main categories of finetuning for language models?**
   - A. Instruction-finetuning and classification-finetuning
   - B. Generative and discriminative models
   - C. Pretraining and fine-tuning
   - D. Supervised and unsupervised learning

3. **What is the purpose of padding text messages in the spam dataset?**
   A. To ensure that all text messages have the same length for batch processing.
   B. To reduce the computational cost of processing the text messages.
   C. To remove irrelevant information from the text messages.
   D. To improve the model's ability to understand the context of the text messages.

4. **What is the role of the padding token in the SpamDataset class?**
   A. To mark the beginning and end of a text message.
   B. To indicate the start of a new sentence.
   C. To represent unknown or out-of-vocabulary words.
   D. To fill in the gaps in shorter text messages to match the length of the longest message.

5. **Why is the last output token of the GPT model chosen for classification-finetuning?**
   A. Because the last token is the most likely to contain the class label information.
   B. Because the last token is the easiest to process for the model.
   C. Because the last token is the most important token in the text message.
   D. Because the last token accumulates information from all preceding tokens due to the causal attention mask.

6. **What is the purpose of the cross entropy loss function in the spam classification task?**
   A. To determine the number of epochs required for training.
   B. To calculate the accuracy of the model's predictions.
   C. To measure the difference between the model's predicted probabilities and the actual class labels.

D. To identify the most important features in the text messages.

7. **What is the significance of the model's accuracy on the test set?**
   A. It indicates how well the model generalizes to new, unseen data.
   B. It determines the number of epochs required for training.
   C. It reflects the model's performance on the training data.
   D. It indicates the model's ability to learn from the training data.

Questions by chapter section

Now we'll move through the chapter in more detail.

6.1 Different categories of fine-tuning

1. What are the two most common ways to fine-tune language models?

2. Describe the purpose of instruction fine-tuning and provide an example.

3. Explain the concept of classification fine-tuning and give an example.

4. What is the key limitation of a classification fine-tuned model?

5. Compare the flexibility of instruction fine-tuned models and classification fine-tuned models.

6. When is instruction fine-tuning the preferred approach?

6.2 Preparing the dataset

1. What is the purpose of the dataset used in this section and what type of data does it contain?

2. Why is the dataset undersampled to include an equal number of 'spam' and 'non-spam' messages?

3. How are the 'string' class labels ('ham' and 'spam') converted into integer class labels?

4. Describe the purpose of the `random_split` function and how it divides the dataset.

5. Pieces of the code have been removed from two places in this listing. Which of these terms have been removed and where should they go?

```
A sample    B shape    C concat    D merge
```

Fill out the table with your answers

```
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].__1__[0]
    ham_subset = df[df["Label"] == "ham"].sample(
        num_spam, random_state=123
    )
    balanced_df = pd.__2__([
        ham_subset, df[df["Label"] == "spam"]
    ])
    return balanced_df

balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

Position	1	2
Term		

6. What is the significance of saving the dataset as CSV files?

7. Match the terms on the left to their descriptions on the right:

Instruction Fine-Tuning		The process of training a language model on a set of tasks using specific instructions to improve its ability to understand and execute tasks described in natural language prompts.
Classification Fine-Tuning		A model that can perform well across a variety of tasks.
Generalist Model		A model that is highly trained to perform a specific task.
Specialized Model		A method of training a model to recognize a specific set of class labels, such as 'spam' and 'not spam'.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

6.3 Creating data loaders

1. What are the two primary options for batching text messages of varying lengths?

2. Pieces of the code have been removed from four places in this listing. Which of these terms have been removed and where should they go? Note that a term may appear more than once!

`A train_loader` `B val_loader` `C train_end` `D validation_end`

```
def random_split(df, train_frac, validation_frac):

    df = df.sample(
        frac=1, random_state=123
    ).reset_index(drop=True)
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)


    train_df = df[:    1    ]
    validation_df = df[   2   :   3   ]
    test_df = df[   4   :]

    return train_df, validation_df, test_df

train_df, validation_df, test_df = random_split(
    balanced_df, 0.7, 0.1)
```

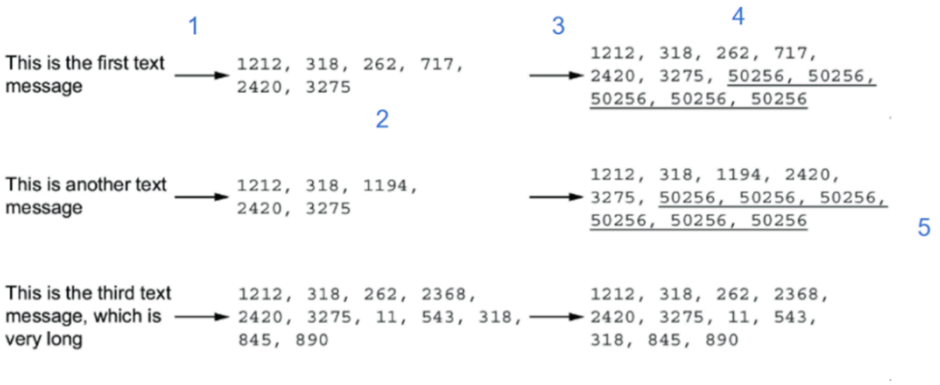
Fill out the table with your answers

Position	1	2	3	4
Term				

3. What is the purpose of the padding token and how is it used in the `SpamDataset` class?

4. What is the role of the `SpamDataset` class in the data loading process?

5. Label this diagram by assigning the label numbers to the matching description



Here are the descriptions for you to assign the label numbers to – any shown in italics are already correctly assigned:

Label	Description
1	Pad to longest sequence
2	Token Ids
3	Longest message has no padding
4	Tokenize text
5	Padded Token Ids

6. How are the validation and test sets padded and truncated in relation to the training set?

7. Describe the structure of a single training batch in terms of input and target tensors.

8. What is the purpose of the `DataLoader` class and how is it used to create training, validation, and test loaders?

6.4 Initializing a model with pretrained weights

1. What is the purpose of initializing a pretrained model before fine-tuning for classification?

2. Describe the process of loading pretrained weights into the GPT model.

3. How is the model's ability to generate coherent text used to verify that the pretrained weights have been loaded correctly?

4. What is the purpose of prompting the model with a spam message before fine-tuning?

6.5 Adding a classification head

1. Why is it necessary to modify the output layer of a pretrained LLM for classification fine-tuning?

2. Explain the rationale behind using a number of output nodes equal to the number of classes in a classification task.

3. Why is it often sufficient to fine-tune only the last layers of a pretrained LLM for a new task?

4. Describe the process of freezing and unfreezing layers in a pretrained LLM for fine-tuning.

5. Why is the last token in a sequence considered the most informative for classification tasks using a causal attention mask?

6. Explain the significance of focusing on the last output token for classification fine-tuning in a GPT-like model.

6.6 Calculating the classification loss and accuracy

1. Explain how the model's output is converted into a class label prediction for spam classification.

2. What is the purpose of the `softmax` function in this context, and why is it optional?

3. Describe the function of the `calc_accuracy_loader` function and how it is used to calculate the classification accuracy.

4. Why is cross-entropy loss used as a proxy for maximizing classification accuracy?

5. Explain the key difference between the `calc_loss_batch` function used for classification and the one used for language modeling.

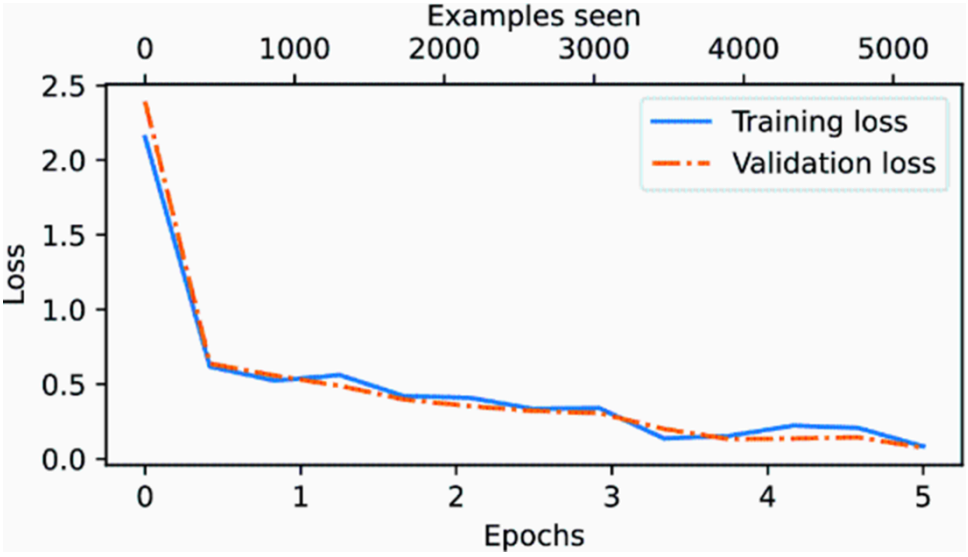
6.7 Fine-tuning the model on supervised data

1. What is the primary difference between the training loop used for fine-tuning and the one used for pretraining?

2. Describe the key modifications made to the `train_classifier_simple` function compared to the `train_model_simple` function used for pretraining.

3. What is the purpose of the `evaluate_model` function in the context of fine-tuning?

4. Explain the significance of the loss curves plotted here:



5. What factors influence the choice of the number of epochs during fine-tuning?

6. Pieces of the code have been removed from three places in this listing. Which of these terms have been removed and where should they go?

```
A train_loader   B test_loader   C AdamW   D val_loader   E Adam
```

```
import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.____1____(model.parameters(), lr=5e-5, weight_decay=0.1)
num_epochs = 5


train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, ____2____, ____3____, optimizer, device,
        num_epochs=num_epochs, eval_freq=50,
        eval_iter=5
    )

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

Fill out the table with your answers

Position	1	2	3
Term			

7. How does the `eval_iter` parameter affect the accuracy estimations during training?

8. Put these steps to prepare and fine-tune the GPT-2 Model into the correct order:
   A. Load pretrained GPT-2 model weights.
   B. Train the model using the training data loader for a specified number of epochs.
   C. Freeze all model parameters except the output layer and the last transformer block.
   D. Initialize an AdamW optimizer.

E. Define functions to calculate loss and accuracy for a batch and a data loader.
F. Set the `requires_grad` attribute to `True` for the new output layer and the last transformer block.
G. Replace the original output layer with a new linear layer mapping to two classes (spam\/not spam).
H. Evaluate the model's performance on the validation set after each epoch.
Fill the table with your answers – two are already in place:

Step Order	Step
1	A
2	
3	
4	
5	E
6	
7	
8	H

6.8 Using the LLM as a spam classifier

1. Describe the process of using a fine-tuned LLM for spam classification.

2. Explain the role of the `classify_review` function in spam classification.

3. How is the classification accuracy of a spam classification model evaluated?

4. In this code listing, order the stages to match the numbers in the listing:

```
def classify_review(
        text, model, tokenizer, device, max_length=None,
        pad_token_id=50256):
    model.eval()

1   input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[1]

2   input_ids = input_ids[:min(
        max_length, supported_context_length
    )]

3   input_ids += [pad_token_id] * (max_length - len(input_ids))

    input_tensor = torch.tensor(
        input_ids, device=device
4   ).unsqueeze(0)

5   with torch.no_grad():
6       logits = model(input_tensor)[:, -1, :]
    predicted_label = torch.argmax(logits, dim=-1).item()

7   return "spam" if predicted_label == 1 else "not spam"
```

Stage	Number in listing
Models inference without gradient tracking	
Add batch dimension	
Truncate sequences if they are too long	
Prepare inputs to the model	
Pad sequences to the longest sequence	
Logits of the last output token	
Return the classified result	

5. What is the purpose of saving a fine-tuned spam classification model?

6. How does the classification fine-tuning process differ from the pretraining process of an LLM?

In Chapter Exercises

These are the exercises from Chapter 6. They are reproduced verbatim.

Exercise 6.1 Increasing the context length

Pad the inputs to the maximum number of tokens the model supports and observe how it affects the predictive performance.

Exercise 6.2 Fine-tuning the whole model

Instead of fine-tuning just the final transformer block, fine-tune the entire model and assess the effect on predictive performance.

Exercise 6.3 Fine-tuning the first vs. last token

Try fine-tuning the first output token. Notice the changes in predictive performance compared to fine-tuning the last output token.

Answers

Quick questions on main concepts

1. A
2. A
3. A
4. D
5. D
6. C
7. A

Questions by chapter section

6.1 Different categories of fine-tuning

1. The two most common ways to fine-tune language models are instruction fine-tuning and classification fine-tuning. Instruction fine-tuning focuses on training the model to understand and execute tasks based on natural language prompts, while classification fine-tuning trains the model to recognize specific class labels.
2. Instruction fine-tuning aims to improve a model's ability to understand and execute tasks based on natural language instructions. For example, training a model to translate English sentences into German using specific instructions would be an instance of instruction fine-tuning.

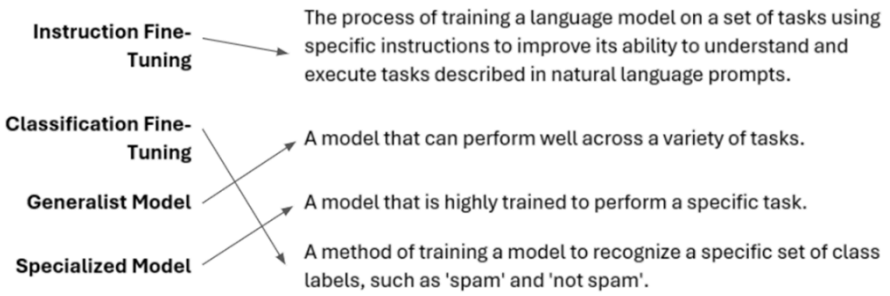
3. Classification fine-tuning involves training a model to recognize specific class labels. For instance, training a model to identify whether an email is spam or not spam is an example of classification fine-tuning. This approach is also used in tasks like image classification and sentiment analysis.
4. A classification fine-tuned model is restricted to predicting classes it has encountered during training. It can only classify data into the predefined categories it was trained on, limiting its ability to handle tasks outside its training scope.
5. Instruction fine-tuned models are more flexible and can handle a broader range of tasks based on user instructions. Classification fine-tuned models are highly specialized and excel at categorizing data into predefined classes, but they lack the flexibility of instruction fine-tuned models.
6. Instruction fine-tuning is best suited for models that need to handle a variety of tasks based on complex user instructions, improving flexibility and interaction quality. It is ideal for applications requiring adaptability and the ability to respond to diverse user requests.

6.2 Preparing the dataset

1. The dataset used is a collection of text messages classified as either 'spam' or 'non-spam' (also known as 'ham'). This dataset is used to demonstrate the process of fine-tuning a large language model for classification tasks.
2. The original dataset has a significant imbalance between 'spam' and 'non-spam' messages. Undersampling creates a balanced dataset, which is beneficial for training a classification model as it prevents the model from being biased towards the majority class.
3. The 'string' class labels are converted into integer class labels (0 and 1) using a mapping dictionary. This process is similar to converting text into token IDs, but instead of using the GPT vocabulary, it uses only two token IDs.
4. The `random_split` function divides the dataset into three parts: training, validation, and testing. The training set is used to train the model, the validation set is used to adjust hyperparameters and prevent overfitting, and the testing set is used to evaluate the model's performance on unseen data.
5. The removed pieces map to the positions like this:

Position	1	2
Term	B	C

6. Saving the dataset as CSV files allows for easy reuse of the data in future steps. This ensures that the prepared dataset can be readily accessed for further analysis and model training.
7. The terms match like this:



6.3 Creating data loaders

1. The two options are truncating all messages to the length of the shortest message or padding all messages to the length of the longest message. Truncation is computationally cheaper but can lead to information loss, while padding preserves the entire content of all messages.
2. The removed pieces map to the positions like this:

Position	1	2	3	4
Term	C	C	D	D

3. The padding token is used to ensure that all text messages in a batch have the same length. In the `SpamDataset` class, shorter messages are padded with the padding token ID (50256) to match the length of the longest message.
4. The `SpamDataset` class handles several key tasks: loading data from CSV files, tokenizing text messages using the GPT-2 tokenizer, and padding or truncating sequences to a uniform length. It also provides methods to access individual data samples and the overall dataset length.
5. This is the correct assignment of the descriptions to labels:

Label	Description
1	Tokenize text
2	Token Ids
3	Pad to longest sequence
4	Padded Token Ids
5	Longest message has no padding

6. The validation and test sets are padded to match the length of the longest training sequence. Any samples exceeding this length are truncated. This ensures consistency in input lengths across all datasets.
7. A single training batch consists of eight text messages represented as token IDs, each with 120 tokens. The corresponding class labels for each message are stored in a separate tensor. This structure allows for efficient processing of multiple training examples simultaneously.
8. The `DataLoader` class is used to create data loaders that efficiently load and process data in batches. It takes a dataset as input and allows for customization of parameters like batch size, shuffling, and number of workers. Separate data loaders are created for training, validation, and testing, each with specific configurations.

6.4 Initializing a model with pretrained weights

1. Initializing a pretrained model prepares it for classification fine-tuning by loading the weights learned during pretraining on unlabeled data. This allows the model to leverage its existing knowledge and accelerate the learning process for the specific classification task.
2. The process involves using the `download_and_load_gpt2` function to retrieve the pretrained weights based on the chosen model size. These weights are then loaded into the `GPTModel` using the `load_weights_into_gpt` function, ensuring the model is ready for fine-tuning.
3. By providing a prompt and generating text using the `generate_text_simple` function, we can assess whether the model produces coherent and meaningful output. If the generated text is sensible, it indicates that the pretrained weights have been loaded successfully.
4. Prompting the model with a spam message allows us to evaluate its initial ability to classify spam. This provides a baseline understanding of the model's performance before fine-tuning and helps identify areas for improvement.

6.5 Adding a classification head

1. The original output layer of a pretrained LLM is designed for language generation, mapping hidden representations to a large vocabulary of tokens. For classification, we need a smaller output layer that maps to the specific classes we want to predict.
2. Using a separate output node for each class allows for a more general approach to classification, as it avoids the need to modify the loss function for binary tasks. This approach is easily adaptable to multi-class problems.

3. The lower layers of a pretrained LLM typically capture general language structures and semantics, while the upper layers learn task-specific features. Fine-tuning only the last layers allows for efficient adaptation to new tasks without disrupting the learned general language knowledge.
4. Freezing layers prevents their weights from being updated during training. This is done by setting the `requires_grad` attribute of their parameters to `False`. Unfreezing layers allows their weights to be updated by setting `requires_grad` to `True`.
5. The causal attention mask restricts each token's attention to itself and preceding tokens. As a result, the last token accumulates information from all previous tokens, making it the most comprehensive representation of the input sequence.
6. Since the last token in a sequence has access to information from all preceding tokens due to the causal attention mask, it provides the most relevant context for classification. Therefore, fine-tuning based on the last token's output allows for more accurate predictions.

6.6 Calculating the classification loss and accuracy

1. The model's output for the last token is a 2-dimensional tensor representing the probability scores for each class (spam or not spam). The class label is determined by finding the index of the highest probability score using the `argmax` function.
2. The `softmax` function converts the model's output into probabilities that sum to 1. It is optional because the largest output values directly correspond to the highest probability scores, so `argmax` can be applied directly to the output tensor.
3. The `calc_accuracy_loader` function iterates through a data loader, applies the `argmax` prediction to each input, and calculates the proportion of correct predictions. It returns the classification accuracy as a percentage.
4. Classification accuracy is not a differentiable function, making it unsuitable for optimization. Cross-entropy loss is a differentiable function that can be used to optimize the model's parameters to indirectly maximize classification accuracy.
5. The classification `calc_loss_batch` function focuses on optimizing only the last token's output, while the language modeling version optimizes the output of all tokens. This difference reflects the focus on predicting the class label for the last token in classification.

6.7 Fine-tuning the model on supervised data

1. The primary difference is that during fine-tuning, we calculate the classification accuracy instead of generating sample text to evaluate the model's performance.

2. The `train_classifier_simple` function tracks the number of training examples seen instead of tokens and calculates accuracy after each epoch. It also omits the step of printing a sample text.
3. The `evaluate_model` function calculates the loss for both the training and validation sets, providing insights into the model's performance on both seen and unseen data.
4. The loss curves in Figure 6.16 show a sharp decline in both training and validation loss during the initial epochs, indicating effective learning. The close proximity of the curves suggests that the model is generalizing well to unseen data and not overfitting.
5. The number of epochs depends on the dataset's complexity and the task's difficulty. Overfitting may necessitate reducing the number of epochs, while insufficient training might require increasing them.
6. The removed pieces map to the positions like this:

Position	1	2	3
Term	C	A	D

7. The `eval_iter` parameter determines the number of batches used to calculate training and validation accuracy. A smaller `eval_iter` value leads to faster training but less accurate performance estimations.
8. Here is the correct order of the steps:

Step Order	Step
1	A
2	C
3	G
4	F
5	E
6	D
7	B
8	H

6.8 Using the LLM as a spam classifier

1. The process involves using a pre-trained LLM, fine-tuned for classification, to predict the class label (spam or not spam) for a given text message. This is achieved by converting the text into token IDs, feeding it to the model, and obtaining the predicted class label based on the model's output.
2. The `classify_review` function takes a text message as input, preprocesses it, converts it into token IDs, feeds it to the fine-tuned model, and predicts the class label (spam or not spam) based on the model's output. It then returns the corresponding class name.

3. The classification accuracy is evaluated by calculating the fraction or percentage of correct predictions made by the model on a test dataset. This metric indicates how well the model is able to correctly classify spam and non-spam messages.
4. Saving the model allows for its reuse later without the need for retraining. This is useful for deploying the model for real-time spam detection or for further experimentation and analysis.
5. The stages map to the numbers like this:

Stage	Number in listing
Models inference without gradient tracking	5
Add batch dimension	4
Truncate sequences if they are too long	2
Prepare inputs to the model	1
Pad sequences to the longest sequence	3
Logits of the last output token	6
Return the classified result	7

6. While both processes involve converting text into token IDs and using a cross-entropy loss function, classification fine-tuning focuses on training the model to output a correct class label, whereas pretraining aims to predict the next token in a sequence. Classification fine-tuning also involves replacing the output layer of the LLM with a smaller classification layer.

In Chapter Exercise Solutions

Exercise 6.1

We can pad the inputs to the maximum number of tokens the model supports by setting the max length to max_length = 1024 when initializing the datasets:

```
train_dataset = SpamDataset(..., max_length=1024, ...)
val_dataset = SpamDataset(..., max_length=1024, ...)
test_dataset = SpamDataset(..., max_length=1024, ...)
```

However, the additional padding results in a substantially worse test accuracy of 78.33% (vs. the 95.67% in the main chapter).

Exercise 6.2

Instead of fine-tuning just the final transformer block, we can fine-tune the entire model by removing the following lines from the code:

```
for param in model.parameters():
    param.requires_grad = False
```

This modification results in a 1% improved test accuracy of 96.67% (vs. the 95.67% in the main chapter).

Exercise 6.3

Rather than fine-tuning the last output token, we can fine-tune the first output token by changing model(input_batch)[:, -1, :] to model(input_batch)[:, 0, :] everywhere in the code.

As expected, since the first token contains less information than the last token, this change results in a substantially worse test accuracy of 75.00% (vs. the 95.67% in the main chapter).

7 Fine-tuning to follow instructions

Chapter 7 explores **instruction fine-tuning**, a process that enables a pretrained LLM to follow specific instructions and generate desired responses, going beyond its initial text completion capabilities. The chapter covers preparing an instruction dataset, organizing data into batches for efficient training, loading a pretrained LLM, and fine-tuning the model to follow instructions. It also explains extracting and evaluating the LLM-generated responses to assess the model's performance after fine-tuning, as well as techniques like scoring responses using another LLM, such as Llama 3. Chapter 7 emphasizes the goal of improving the LLM's ability to comprehend and execute instructions effectively, similar to how Chapter 6 focused on fine-tuning for classification.

All the answers to the questions can be found at the end of this document.

Quick questions on main concepts

1. **What is the primary challenge that pretrained LLMs often face regarding instructions?**
   - A. Inability to complete sentences.
   - B. Difficulty in generating coherent text.
   - C. Limited vocabulary size.
   - D. Struggling with specific instructions like grammar correction or voice conversion.

2. **What is the key component in preparing a dataset for supervised instruction fine-tuning?**
   - A. Pre-trained language model.

B. Optimization algorithm.
C. Instruction-response pairs.
D. Tokenization algorithm.

3. **What data format is commonly used for instruction datasets due to its human and machine readability?**
   A. JSON (JavaScript Object Notation).
   B. CSV (Comma Separated Values).
   C. YAML (YAML Ain't Markup Language).
   D. XML (Extensible Markup Language).

4. **What is the purpose of a custom collate function in the context of instruction fine-tuning?**
   A. To optimize the model's architecture.
   B. To handle the specific formatting and requirements of the instruction fine-tuning dataset.
   C. To pre-process the input data.
   D. To evaluate the model's performance.

5. **What is the purpose of using the ignore_index parameter (-100) in the custom collate function?**
   A. To mark the end of a sequence.
   B. To indicate an unknown token.
   C. To exclude padding tokens from the loss calculation.
   D. To identify the start of a sequence.

6. **What is the purpose of saving the fine-tuned model's state dictionary?**
   A. To compress the model for efficient storage.
   B. To visualize the model's architecture.
   C. To save the model's parameters for later use or reuse in other projects.
   D. To improve the model's performance.

Questions by chapter section

Now we'll move through the chapter in more detail.

7.1 Introduction to instruction fine-tuning

1. What is the primary function of a pretrained LLM?

2. What is the challenge that pretrained LLMs often face?

3. What is the purpose of instruction fine-tuning?

4. What is a crucial aspect of instruction fine-tuning?

7.2 Preparing a dataset for supervised instruction fine-tuning

1. What is the purpose of the instruction dataset used for fine-tuning a pretrained LLM?

2. Describe the format of the instruction dataset used in this section.

3. What are the two prompt styles mentioned in the section, and how do they differ?

4. What is the purpose of the `format_input` function, and how does it work?

5. How is the instruction dataset divided into training, validation, and test sets?

6. Match the terms on the left to their description on the right:

Instruction–Response Pairs		Different formats for presenting instructions and inputs to a language model, influencing how the model interprets and responds to the task.
Prompt Styles		A simpler format for instruction fine-tuning that uses designated tokens to mark user input and assistant output, allowing for more flexibility in task presentation.
Alpaca Prompt Style		The dataset consists of pairs of instructions and their corresponding responses, providing examples of how to complete tasks.
Phi-3 Prompt Style		A structured format for instruction fine-tuning that uses separate sections for the instruction, input, and response, making it clear for the model to understand the task.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

7.3 Organizing data into training batches

1. What is the purpose of a custom collate function in instruction fine-tuning?

2. How does the custom collate function handle padding in instruction fine-tuning?

3. Explain the role of target token IDs in instruction fine-tuning and how they are generated.

4. Why are padding tokens replaced with the `-100` placeholder value in the target token IDs?

5. What is the purpose of retaining one end-of-text token in the target sequence?

6. Pieces of the code have been removed from four places in this listing. Which of these terms have been removed and where should they go? Note that a term may appear more than once!

```
A encode   B data   C format   D encoded_texts E full_text
```

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

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.encoded_texts = []
        for entry in __1__:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text
            self.encoded_texts.append(
                tokenizer.__2__(full_text)
            )

    def __getitem__(self, index):
        return self.__3__[index]

    def __len__(self):
        return len(self.data)
```

Fill out the table with your answers

Position	1	2	3
Term			

7.4 Creating data loaders for an instruction dataset

1. What is the purpose of the `custom_collate_fn` function in the context of instruction fine-tuning?

2. Explain the advantage of moving data to the target device within the `custom_collate_fn` function instead of the main training loop.

3. How is the `device` setting determined and used in the `custom_collate_fn` function?

4. What is the purpose of the `allowed_max_length` parameter in the `customized_collate_fn` function?

5. Describe the process of creating data loaders for the training, validation, and test sets using the `DataLoader` class.

6. Pieces of the code have been removed from four places in this listing. Which of these terms have been removed and where should they go? Note that a term may appear more than once!

```
A 50265   B padded[1:]   C padded[]   D 50256   E padded[:-1]
```

```
def custom_collate_draft_2(
    batch,
    pad_token_id=____1____,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]

        padded = (
            new_item + [pad_token_id] *
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(____2____)
        targets = torch.tensor(____3____)
        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor

inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)
```

Fill out the table with your answers

Position	1	2	3
Term			

7. Match the terms on the left with their descriptions on the right:

Batching Process		A function that defines how to combine individual data samples into a batch, specifically tailored for the requirements of instruction fine-tuning.
Custom Collate Function		The process of organizing training data into groups of samples, called batches, to improve training efficiency.
Padding Tokens		Special tokens added to the end of sequences to ensure all inputs in a batch have the same length, allowing for efficient processing by the model.
Ignore Index		A special value, typically -100, used to indicate padding tokens in the target sequence, preventing them from contributing to the loss calculation during training.

Fill the table with the column mappings:

Left Hand Column	1	2	3	4
Right Hand Column				

7.5 Loading a pretrained LLM

1. Why is a larger pretrained model, like 'gpt2-medium', preferred for instruction fine-tuning compared to the smaller 'gpt2-small' model?

2. What is the purpose of loading a pretrained LLM before starting instruction fine-tuning?

3. How does the code for loading a pretrained model differ from the code used for pretraining or classification fine-tuning?

4. What is the purpose of evaluating the pretrained LLM's performance on a validation task before fine-tuning?

5. How is the model's generated response isolated from the input instruction in the provided code?

7.6 Fine-tuning the LLM on instruction data

1. What is the purpose of fine-tuning an LLM on instruction data?

2. Describe the process of fine-tuning an LLM on instruction data, highlighting the key steps involved.

3. What are some potential challenges encountered during fine-tuning an LLM on instruction data, and how can these challenges be addressed?

4. How is the effectiveness of fine-tuning evaluated during the training process?

5. What is the significance of the Alpaca dataset in the context of fine-tuning LLMs?

7.7 Extracting and saving responses

1. Describe the process of evaluating the performance of an instruction-fine-tuned Large Language Model (LLM) after training.

2. What are the different methods for evaluating the performance of instruction-fine-tuned LLMs, and what are their relative strengths and weaknesses?

3. What is meant by 'conversational performance' in the context of LLMs, and why is it important?

4. How can the responses generated by an LLM be automatically evaluated using another LLM, and what are the advantages of this approach?

5. Explain the process of appending generated model responses to a test set and saving the updated data for later analysis.

7.8 Evaluating the fine-tuned LLM

1. What is the purpose of using a larger LLM to evaluate the responses of a fine-tuned model?

2. Describe the process of using Ollama to evaluate the responses of a fine-tuned model.

3. What are some alternative LLMs that can be used for evaluating model responses, besides the 8-billion-parameter Llama 3 model?

4. How can the `generate_model_scores` function be used to assess the performance of a fine-tuned model?

5. What are some strategies for improving the performance of a fine-tuned model?

6. Match the terms on the left with their descriptions on the right:

Test Set		A method of evaluating the conversational performance of a language model using another language model to assess the quality of responses.
Conversational Performance		The ability of a language model to engage in human-like communication, understanding context, nuance, and intent.
Automated Conversational Benchmarks		A portion of data that is held out from the training process and used to evaluate the performance of a trained model.

Fill the table with the column mappings:

Left Hand Column	1	2	3
Right Hand Column			

In Chapter Exercises

These are the exercises from Chapter7. They are reproduced verbatim.

Exercise 7.1 Changing prompt styles

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

Exercise 7.2 Instruction and input masking

After completing the chapter and fine-tuning the model with InstructionDataset, replace the instruction and input tokens with the -100 mask to use the instruction masking method illustrated in figure 7.13. Then evaluate whether this has a positive effect on model performance.

Exercise 7.3 Fine-tuning on the original Alpaca dataset

The Alpaca dataset, by researchers at Stanford, is one of the earliest and most popular openly shared instruction datasets, consisting of 52,002 entries. As an alternative to the instruction-data.json file we use here, consider fine-tuning an LLM on this dataset. The dataset is available at https://mng.bz/NBnE.

This dataset contains 52,002 entries, which is approximately 50 times more than those we used here, and most entries are longer. Thus, I highly recommend using a GPU to conduct the training, which will accelerate the fine-tuning process. If you encounter out-of-memory errors, consider reducing the batch_size from 8 to 4, 2, or even 1. Lowering the allowed_max_length from 1,024 to 512 or 256 can also help manage memory problems.

Exercise 7.4 Parameter-efficient fine-tuning with LoRA

To instruction fine-tune an LLM more efficiently, modify the code in this chapter to use the low-rank adaptation method (LoRA) from appendix E. Compare the training run time and model performance before and after the modification.

Answers

Quick questions on main concepts

1. D
2. C
3. A
4. B
5. C
6. C

Questions by chapter section

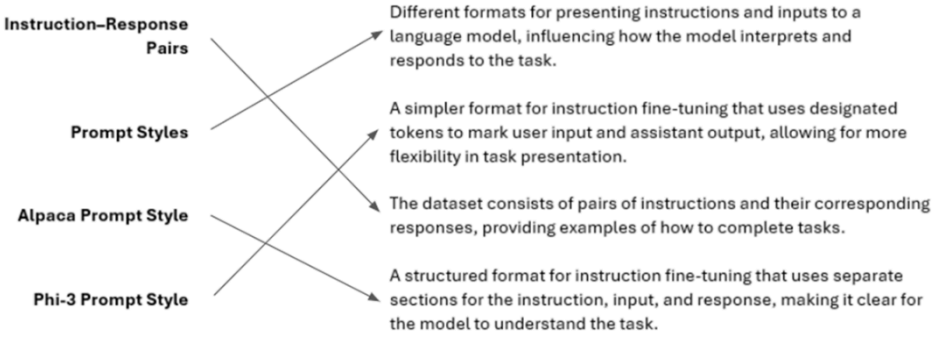
7.1 Introduction to instruction fine-tuning

1. A pretrained LLM is primarily capable of text completion, meaning it can finish sentences or write text paragraphs given a fragment as input.
2. Pretrained LLMs often struggle with specific instructions, such as grammar correction or voice conversion, requiring further fine-tuning.
3. Instruction fine-tuning aims to improve an LLM's ability to follow specific instructions and generate desired responses based on those instructions.
4. Preparing a suitable dataset is a key aspect of instruction fine-tuning, providing the model with examples of instructions and desired responses.

7.2 Preparing a dataset for supervised instruction fine-tuning

1. The instruction dataset consists of instruction-response pairs that are used to train the LLM to follow instructions and generate appropriate responses based on given inputs.
2. The dataset is stored in a JSON file and contains entries that are Python dictionary objects. Each entry includes an 'instruction', 'input', and 'output' field, representing the task, input data, and desired response, respectively.

3. The two prompt styles are Alpaca and Phi-3. Alpaca uses a structured format with defined sections for instruction, input, and response, while Phi-3 employs a simpler format with designated <|user|> and <|assistant|> tokens.
4. The `format_input` function converts the entries in the instruction dataset into the Alpaca-style input format. It constructs a formatted string that includes the instruction, input (if available), and a placeholder for the response.
5. The dataset is divided into training, validation, and test sets using a specific ratio. The training set is used to train the model, the validation set is used to evaluate the model's performance during training, and the test set is used to evaluate the model's final performance.
6. The terms match like this:



Left Hand Column	1	2	3	4
Right Hand Column	3	1	4	2

7.3 Organizing data into training batches

1. A custom collate function is used to handle the specific requirements and formatting of the instruction fine-tuning dataset. It ensures that training examples are padded to the same length within each batch, allowing for efficient processing by the model.
2. The custom collate function pads training examples to the length of the longest example in each batch, using the `<|endoftext|>` token ID (50256). This minimizes unnecessary padding by only extending sequences to match the longest one in each batch.

3. Target token IDs represent the desired output sequence that the model should generate. They are created by shifting the input token IDs one position to the right, omitting the first token and appending an end-of-text token. This setup allows the model to learn how to predict the next token in a sequence.
4. Replacing padding tokens with `-100` allows the cross entropy loss function to ignore them during training. This ensures that only meaningful data influences model learning and prevents padding tokens from contributing to the loss calculation.
5. Retaining one end-of-text token in the target sequence helps the LLM learn to generate end-of-text tokens, which act as an indicator that the generated response is complete.
6. The removed pieces map to the positions like this:

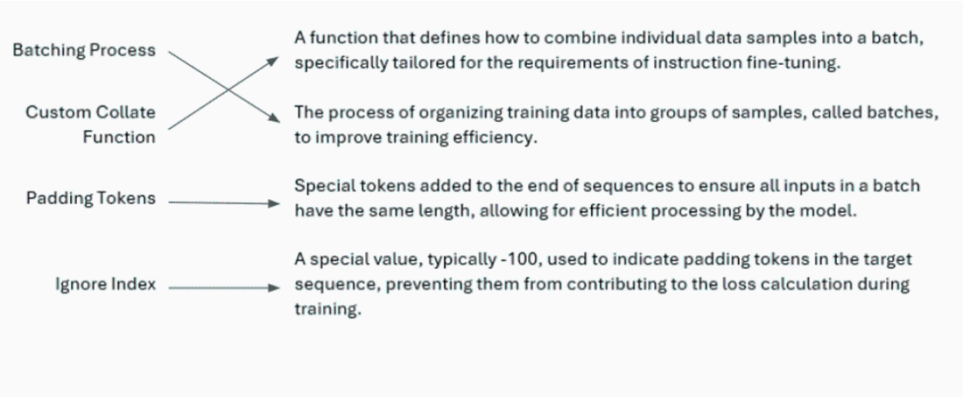
Position	1	2	3
Term	B	A	D

7.4 Creating data loaders for an instruction dataset

1. The `custom_collate_fn` function is used to batch the instruction dataset, ensuring that the input and target tensors are moved to the specified device (CPU, GPU, or MPS) before being fed into the LLM for fine-tuning.
2. By performing the device transfer within the `custom_collate_fn`, the process becomes a background task, preventing it from blocking the GPU during model training and improving efficiency.
3. The `device` setting is determined based on the availability of a GPU or MPS. The `partial` function from `functools` is used to create a new version of the `custom_collate_fn` with the `device` argument prefilled, ensuring that the function uses the correct device for data transfer.
4. The `allowed_max_length` parameter is used to truncate the data to the maximum context length supported by the LLM model being fine-tuned, in this case, the GPT-2 model.
5. The `DataLoader` class is used to create data loaders for each set (training, validation, and test). The `batch_size`, `collate_fn`, `shuffle`, `drop_last`, and `num_workers` parameters are configured to control the batching process, shuffling, and data loading behavior.
6. The removed pieces map to the positions like this:

Position	1	2	3
Term	D	B	E

7. The terms match like this:



Left Hand Column	1	2	3	4
Right Hand Column	2	1	3	4

7.5 Loading a pretrained LLM

1. The smaller model lacks the capacity to learn and retain the complex patterns and nuanced behaviors required for high-quality instruction-following tasks. The larger model has more parameters, allowing it to handle more intricate instructions and generate more accurate responses.
2. Loading a pretrained LLM provides a foundation for the fine-tuning process. It allows the model to leverage existing knowledge and patterns learned during pretraining, enabling it to learn new tasks more efficiently and effectively.
3. The code remains largely the same, but instead of specifying the 'gpt2-small' model, we now specify 'gpt2-medium' to load the larger model with 355 million parameters. This change reflects the choice of a more capable model for instruction fine-tuning.
4. Evaluating the pretrained LLM's performance provides a baseline understanding of its capabilities before fine-tuning. This allows us to assess the impact of fine-tuning on the model's ability to follow instructions and generate accurate responses.
5. The code subtracts the length of the input instruction from the start of the generated text, effectively removing the input text and leaving only the model's generated response. The `strip()` function is then applied to remove any extra whitespace.

7.6 Fine-tuning the LLM on instruction data

1. Fine-tuning an LLM on instruction data aims to improve its ability to understand and follow instructions, leading to more accurate and relevant responses to user prompts.

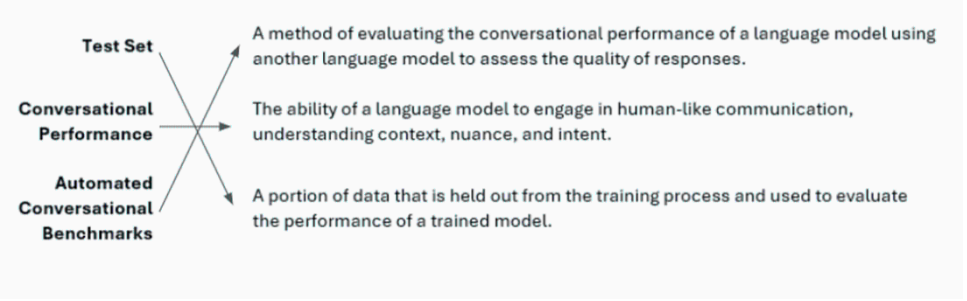
2. The process involves loading a pretrained LLM, preparing an instruction dataset, and training the model on this dataset using a suitable loss function and optimizer. The training process aims to minimize the loss, indicating improved performance in following instructions.
3. Challenges include hardware limitations, such as memory constraints, which can be addressed by using a smaller model, reducing the batch size, or utilizing a GPU for faster training. Additionally, managing the length of input sequences can be crucial for efficient training.
4. The effectiveness is evaluated by monitoring the training and validation losses. A decrease in these losses indicates that the model is learning to follow instructions better. Additionally, inspecting the generated responses during training provides qualitative insights into the model's progress.
5. The Alpaca dataset is a valuable resource for fine-tuning LLMs on instruction data. It provides a large collection of diverse instructions and corresponding responses, enabling the model to learn a wider range of task-specific behaviors.

7.7 Extracting and saving responses

1. Evaluation involves extracting model-generated responses from a held-out test set, manually analyzing them, and then quantifying the response quality using various methods such as benchmarks, human comparison, or automated metrics.
2. Methods include short-answer benchmarks (e.g., MMLU), human preference comparisons, and automated conversational benchmarks (e.g., AlpacaEval). Human evaluation provides valuable insights but is time-consuming, while automated methods are efficient but may lack the nuance of human judgment.
3. Conversational performance refers to an LLM's ability to engage in human-like communication, understanding context, nuance, and intent. It's crucial for applications like chatbots where natural and coherent interaction is essential.
4. An approach similar to AlpacaEval can be used, employing another LLM to evaluate the responses. This automated method is efficient, saving time and resources compared to manual human evaluation while still providing meaningful performance indicators.
5. The generated model responses are added to a dictionary containing the test data. This updated data is then saved as a JSON file (e.g., 'instruction-data-with-response.json') for easy access and analysis in future sessions.

7.8 Evaluating the fine-tuned LLM

1. A larger LLM, like Llama 3 or GPT-4, can be used to automatically assess the quality of responses generated by a fine-tuned model. This provides a more objective and scalable method for evaluating model performance compared to manually reviewing a few examples.
2. Ollama, an open-source application, allows you to run LLMs locally. You can use the `query_model` function to send prompts to the LLM, such as asking it to score a model's response on a scale of 0 to 100. The LLM's evaluation can then be used to assess the overall performance of the fine-tuned model.
3. Other LLMs, such as the 3.8-billion-parameter `phi3` model or the larger 70-billion-parameter Llama 3 model, can be used with Ollama. The choice of model depends on the available computational resources and the desired level of performance.
4. The `generate_model_scores` function iterates through a set of test data, sending prompts to the LLM to evaluate each model response. It then calculates the average score across all responses, providing a quantitative measure of the model's performance.
5. Strategies for improving model performance include adjusting hyperparameters during fine-tuning, increasing the size or diversity of the training dataset, experimenting with different prompts or instruction formats, and using a larger pretrained model.
6. The terms match like this:



Left Hand Column	1	2	3
Right Hand Column	3	2	1

In Chapter Exercise Solutions

Exercise 7.1

The Phi-3 prompt format, which is shown in figure 7.4, looks like the following for a given example input:

```
<user>
Identify the correct spelling of the following word: 'Occasion'

<assistant>
The correct spelling is 'Occasion'.
```

To use this template, we can modify the format_input function as follows:

```
def format_input(entry):
    instruction_text = (
        f"<|user|>\n{entry['instruction']}"
    )
    input_text = f"\n{entry['input']}" if entry["input"] else ""
    return instruction_text + input_text
```

Lastly, we also have to update the way we extract the generated response when we collect the test set responses:

```
for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)
    tokenizer=tokenizer
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)
    response_text = (
        generated_text[len(input_text):]
        .replace("<|assistant|>:", "")
        .strip()
    )
    test_data[i]["model_response"] = response_text
```

Fine-tuning the model with the Phi-3 template is approximately 17% faster since it results in shorter model inputs. The score is close to 50, which is in the same ballpark as the score we previously achieved with the Alpaca-style prompts.

Exercise 7.2

To mask out the instructions as shown in figure 7.13, we need to make slight modifications to the InstructionDataset class and custom_collate_fn function. We can modify the InstructionDataset class to collect the lengths of the instructions, which we will use in the collate function to locate the instruction content positions in the targets when we code the collate function, as follows:

```
class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.instruction_lengths = []
        self.encoded_texts = []

        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\\n### Response:\\n{entry['output']}"
            full_text = instruction_plus_input + response_text

            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )
            instruction_length = (
                len(tokenizer.encode(instruction_plus_input)
            )
            self.instruction_lengths.append(instruction_length)

    def __getitem__(self, index):
        return self.instruction_lengths[index], self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

Next, we update the custom_collate_fn where each batch is now a tuple containing (instruction_length, item) instead of just item due to the changes in the InstructionDataset dataset. In addition, we now mask the corresponding instruction tokens in the target ID list:

```
def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):

    batch_max_length = max(len(item)+1 for instruction_length, item in batch)
    inputs_lst, targets_lst = [], []

    for instruction_length, item in batch:
        new_item = item.copy()
```

```
        new_item += [pad_token_id]
        padded = (
            new_item + [pad_token_id] * (batch_max_length - len(new_item)
        )
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])
        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index

        targets[:instruction_length-1] = -100

        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length]

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)

    return inputs_tensor, targets_tensor
```

When evaluating a model fine-tuned with this instruction masking method, it performs slightly worse (approximately 4 points using the Ollama Llama 3 method from chapter 7). This is consistent with observations in the "Instruction Tuning With Loss Over Instructions" paper (https://arxiv.org/abs/2405.14394).

Exercise 7.3

To fine-tune the model on the original Stanford Alpaca dataset (https://github.com/tatsu-lab/stanford_alpaca), we just have to change the file URL from

```
url = "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/main/ch07/01_main-chapter-code/instruction-data.json"
```

to

```
url = "https://raw.githubusercontent.com/tatsu-lab/stanford_alpaca/main/alpaca_data.json"
```

Note that the dataset contains 52,000 entries (50x more than in chapter 7), and the entries are longer than the ones we worked with in chapter 7.

Thus, it's highly recommended that the training be run on a GPU.

If you encounter out-of-memory errors, consider reducing the batch size from 8 to 4, 2, or 1. In addition to lowering the batch size, you may also want to consider lowering the allowed_max_length from 1024 to 512 or 256.

Below are a few examples from the Alpaca dataset, including the generated model responses:

Exercise 7.4

To instruction fine-tune the model using LoRA, use the relevant classes and functions from appendix E:

```
from appendix_E import LoRALayer, LinearWithLoRA, replace_linear_with_lora
```

Next, add the following lines of code below the model loading code in section 7.5:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
replace_linear_with_lora(model, rank=16, alpha=16)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
model.to(device)
```

Note that, on an Nvidia L4 GPU, the fine-tuning with LoRA takes 1.30 min to run on an L4. On the same GPU, the original code takes 1.80 minutes to run. So, LoRA is approximately 28% faster in this case. The score, evaluated with the Ollama Llama 3 method from chapter 7, is around 50, which is in the same ballpark as the original model.

Appendix A. Introduction to PyTorch

Appendix A provides a primer on PyTorch, a popular Python-based deep learning library, emphasizing its three core components: a **tensor library**, an **automatic differentiation engine**, and **deep learning utilities.** PyTorch's tensor library, similar to NumPy, enables efficient handling of array-like data structures, including scalars, vectors, matrices, and higher-dimensional arrays, with the added advantage of GPU support for accelerated computations. The automatic differentiation (autograd) engine in PyTorch simplifies neural network training by automatically calculating gradients for tensor operations, eliminating the need for manual gradient derivation. Finally, the appendix describes PyTorch's deep learning utilities, which offer building blocks like pretrained models, loss functions, and optimizers, for designing and training a variety of deep learning models.

All the answers to the questions can be found at the end of this document.

Questions

1. What is PyTorch and why is it popular?

2. Name the three main components of PyTorch.

3. How does PyTorch relate to NumPy?

4. What is a tensor in the context of PyTorch?

5. How are PyTorch tensors created?

6. Explain the concept of a computation graph in PyTorch.

7. What is automatic differentiation (autograd) in PyTorch?

8. How are gradients computed in PyTorch?

9. How do you define a multilayer neural network in PyTorch?

10. What is the purpose of the `torch.nn.Sequential` class?

11. How do you access the parameters of a PyTorch model?

12. What is the `forward` pass in a neural network?

13. Explain the use of `torch.no_grad()`.

14. How are datasets and data loaders used in PyTorch?

15. What are the key methods in a custom PyTorch `Dataset` class?

16. What is the purpose of the `num_workers` parameter in `DataLoader`?

17. Describe a typical training loop in PyTorch.

18. What is the role of an optimizer in PyTorch?

19. How do you save and load a PyTorch model?

20. How do you use GPUs for training in PyTorch?

21. What is `torch.cuda.is_available()` used for?

22. What is distributed training in PyTorch?

23. How does `DistributedDataParallel` (DDP) work?

24. What is the `DistributedSampler` in PyTorch?

Answers

1. PyTorch is an open-source Python-based deep learning library. Its popularity stems from its user-friendly interface, efficiency, and flexibility, offering a balance between usability and advanced features.
2. PyTorch comprises a tensor library for GPU-accelerated computation, an automatic differentiation engine (autograd) for gradient calculation, and a deep learning library with tools for building and training models.
3. PyTorch's tensor operations largely adopt NumPy's API and syntax, making it familiar to NumPy users. PyTorch extends NumPy's functionality with GPU acceleration and automatic differentiation.
4. A tensor is a multi-dimensional array that serves as a data container in PyTorch, generalizing vectors and matrices to higher dimensions. Its rank indicates the number of dimensions.
5. PyTorch tensors are created using the `torch.tensor()` function, specifying the data as a list, nested list, or other iterable. The data type can be specified or inferred from the input.
6. A computation graph represents the sequence of calculations in a neural network. PyTorch automatically builds this graph, tracking operations on tensors to enable automatic differentiation.
7. Autograd is PyTorch's system for automatically computing gradients of tensor operations. It uses the computation graph to efficiently apply the chain rule for backpropagation.
8. Gradients are computed using the `grad()` function or by calling `.backward()` on a loss tensor. The `.backward()` method automatically computes gradients for all leaf nodes in the computation graph.
9. You subclass the `torch.nn.Module` class, defining layers in the `__init__` method and their interaction in the `forward` method. The `forward` method describes the data flow through the network.
10. The `torch.nn.Sequential` class simplifies the definition of neural networks by allowing you to chain layers together in a specific order.

11. You access a model's parameters using `model.parameters()`. This returns an iterator over all trainable parameters (weights and biases).
12. The forward pass is the process of calculating output tensors from input tensors by passing the input data through all the layers of the neural network.
13. The `torch.no_grad()` context manager disables gradient tracking, saving memory and computation when you're only making predictions (inference) and not training.
14. A custom `Dataset` class defines how individual data records are loaded. A `DataLoader` handles batching, shuffling, and parallel data loading using worker processes.
15. The key methods are `__init__` (to initialize), `__getitem__` (to get a single data item by index), and `__len__` (to get the dataset's length).
16. The `num_workers` parameter in `DataLoader` specifies the number of subprocesses to use for data loading. Increasing this can speed up training by parallelizing data loading.
17. A training loop iterates over epochs and batches. For each batch, it performs a forward pass, calculates the loss, performs backpropagation using `.backward()`, and updates model parameters using an optimizer's `.step()` method.
18. An optimizer (e.g., `torch.optim.SGD`) updates model parameters based on the calculated gradients to minimize the loss function. The learning rate is a key hyperparameter of the optimizer.
19. You save a model's state using `torch.save(model.state_dict(), 'filename.pth')` and load it using `model.load_state_dict(torch.load('filename.pth'))`.
20. You transfer tensors to the GPU using `.to('cuda')` and ensure all tensors involved in a computation are on the same device. The `torch.device` context manager helps manage device placement.
21. The `torch.cuda.is_available()` function checks if CUDA (GPU computation) is available in the current environment.
22. Distributed training involves distributing model training across multiple GPUs or machines to speed up the process. PyTorch's `DistributedDataParallel` (DDP) is a common approach.
23. DDP creates copies of the model on each GPU, splits the data across them, and synchronizes gradients after each batch to update the model parameters efficiently.
24. The `DistributedSampler` ensures that each GPU in a distributed training setup receives a different, non-overlapping subset of the training data.

Appendix B. References and further reading

Appendix B is a collection of references and further reading, and so we do not provide any questions on this topic here.

Appendix C. Exercise solutions

Appendix C contains the solutions to the exercises in the original book, and so we do not provide any questions on this topic here. The solutions to the exercises themselves can be found in the relevant chapter in this guide.

Appendix D. Adding bells and whistles to the training loop

Appendix D aims to improve the training function used for pretraining and fine-tuning LLMs. It focuses on three specific techniques: learning rate warmup, cosine decay, and gradient clipping. Guidance is provided on how to integrate these techniques into the training function, enabling readers to experiment with these enhancements and observe their impact on the performance of LLMs during pretraining.

All the answers to the questions can be found at the end of this document.

Questions

1. Explain the purpose of learning rate warmup in training large language models.

2. How does cosine decay modify the learning rate during training?

3. What is the purpose of gradient clipping in training LLMs?

4. How is the learning rate calculated during the warmup phase?

5. Describe the formula used for cosine decay of the learning rate.

6. What is the L2 norm, and how is it used in gradient clipping?

7. How does the `clip_grad_norm_` function in PyTorch work?

8. What is the benefit of combining learning rate warmup, cosine decay, and gradient clipping?

9. How is the peak learning rate determined in the provided training function?

10. When is gradient clipping applied in the modified training function?

Answers

1. Learning rate warmup gradually increases the learning rate from a low initial value to a peak value. This prevents large, destabilizing updates early in training, improving stability.
2. Cosine decay adjusts the learning rate to follow a cosine curve after a warmup period, gradually decreasing it to near zero. This helps avoid overshooting loss minima and improves training stability.
3. Gradient clipping prevents excessively large gradients by scaling them down to a maximum magnitude. This maintains stability by ensuring parameter updates remain within a manageable range.

4. During warmup, the learning rate linearly increases from the initial learning rate to the peak learning rate over a specified number of warmup steps.
5. After warmup, cosine decay uses a formula involving a cosine function to reduce the learning rate gradually, combining a minimum learning rate with a scaling factor based on the cosine of the training progress.
6. The L2 norm measures the magnitude of a vector (or matrix). In gradient clipping, it calculates the length of the gradient vector; if it exceeds a threshold, the gradients are scaled down.
7. The `clip_grad_norm_` function calculates the L2 norm of the gradients. If this norm exceeds a specified `max_norm`, it scales down all gradients proportionally to ensure the norm equals `max_norm`.
8. Combining these techniques enhances the stability of LLM training by mitigating the risks of early instability, overshooting minima, and excessively large gradient updates.
9. The peak learning rate is obtained directly from the optimizer's parameter groups, reflecting the initial learning rate set for the optimizer.
10. Gradient clipping is applied after the warmup phase, only when the global step exceeds the number of warmup steps, ensuring that the learning rate has stabilized before applying gradient clipping.

Appendix E. Parameter-efficient fine-tuning with LoRA

Appendix E introduces Low-Rank Adaptation (LoRA) as an efficient technique for fine-tuning large language models by updating only a small subset of parameters, using low-rank matrices to approximate weight changes during backpropagation. The appendix emphasizes that LoRA's ability to separate LoRA weights from pretrained model weights allows for model customization without altering original weights, reducing storage needs and improving scalability. The appendix then practically demonstrates LoRA's application in spam classification by integrating LoRA layers into the GPT model, showing comparable performance to traditional fine-tuning but with considerably fewer trainable parameters.

All the answers to the questions can be found at the end of this document.

Questions

1. What is LoRA and what is its primary advantage in fine-tuning large language models?

2. How does LoRA approximate the weight update matrix (DW) in a linear layer?

3. Explain the role of the 'rank' hyperparameter in LoRA.

4. What is the purpose of the 'alpha' hyperparameter in LoRA?

5. How does the `LoRALayer` class contribute to the LoRA fine-tuning process?

6. Describe the function of the `LinearWithLoRA` class.

7. What is the significance of initializing matrix B in the `LoRALayer` with zeros?

8. Explain the purpose of the `replace_linear_with_lora` function.

9. Why is freezing the original model's parameters important before applying LoRA?

10. What is the primary practical advantage of keeping LoRA weights separate from the original model weights?

Answers

1. LoRA (Low-Rank Adaptation) is a parameter-efficient fine-tuning technique that adjusts only a small subset of a model's weights, significantly reducing computational costs and resources compared to full fine-tuning.
2. Instead of directly calculating DW, LoRA uses two smaller matrices, A and B, such that their product AB approximates DW. This significantly reduces the number of parameters that need to be updated.
3. The 'rank' hyperparameter determines the inner dimension of matrices A and B, controlling the number of additional parameters introduced by LoRA. A higher rank allows for greater model adaptability but increases computational cost.
4. The 'alpha' hyperparameter acts as a scaling factor for the output of the low-rank adaptation (AB), regulating the influence of the LoRA update on the original layer's output.
5. The `LoRALayer` class implements the core LoRA mechanism, creating and applying the smaller matrices A and B to the input, generating the low-rank approximation of the weight update.
6. The `LinearWithLoRA` class combines a standard `Linear` layer with a `LoRALayer`. Its forward pass adds the output of the original linear layer and the LoRA layer's output, integrating the low-rank adaptation.
7. Initializing matrix B with zeros ensures that the initial LoRA adaptation doesn't alter the pretrained weights, as adding the zero matrix (AB) leaves the original weights unchanged.
8. This function iterates through a model's layers, replacing all `Linear` layers with `LinearWithLoRA` layers, integrating the LoRA adaptation into the model's architecture.
9. Freezing the original parameters prevents them from being updated during LoRA fine-tuning, ensuring that only the smaller LoRA matrices are trained, maintaining the knowledge of the pretrained model.
10. Keeping LoRA weights separate allows for efficient model customization without storing multiple complete model versions. This reduces storage requirements and improves scalability, especially beneficial for applications requiring many customized models.