"Andriy's long-awaited sequel in his "The Hundred-Page" series of machine learning textbooks is a masterpiece of concision."

— Bob van Luijt, CEO and Co-Founder of Weaviate

"Andriy has this almost supernatural talent for shrinking epic AI concepts down to bite-sized, 'Ah, now I get it!' moments."

— Jorge Torres, CEO at MindsDB

"Andriy paints for us, in 100 marvelous strokes, the journey from linear algebra basics to the implementation of transformers."

— Florian Douetteau, Co-founder and CEO at Dataiku

"Andriy's book is an incredibly concise, clear, and accessible introduction to machine learning."

— Andre Zayarni, Co-founder and CEO at Qdrant

"This is one of the most comprehensive yet concise handbooks out there for truly understanding how LLMs work under the hood."

— Jerry Liu, Co-founder and CEO at LlamaIndex

Featuring a foreword by **Tomáš Mikolov** and back cover text by **Vint Cerf**

The Hundred-Page Language Models Book

Andriy Burkov

ISBN 978-1-7780427-2-0

Publisher: True Positive Inc.

To my family, with love

"Language is the source of misunderstandings."
***—Antoine de Saint-Exupéry**, The Little Prince*

"In mathematics you don't understand things. You just get used to them."
—John von Neumann

"Computers are useless. They can only give you answers."
— Pablo Picasso

The book is distributed on the "read first, buy later" principle

Contents

Chapter 5. Large Language Model

Large language models have transformed NLP through their remarkable capabilities in text generation, translation, and question-answering. But how can a model trained solely to predict the next word achieve these results? The answer lies in two factors: scale and supervised finetuning.

5.1. Why Larger Is Better

LLMs are built with a large number of parameters, large context windows, and trained on large corpora backed by substantial computational resources. This scale enables them to learn complex language patterns and even memorize information.

Creating a **chat LM**, capable of handling dialogue and following complex instructions, involves two stages. The first stage is **pretraining** on a massive dataset, often containing trillions of tokens. In this phase, the model learns to predict the next token based on context—similar to what we did with the RNN and decoder models, but at a vastly larger scale.

With more parameters and extended context windows, the model aims to "understand" the context as deeply as possible to improve the next token prediction and minimize the **cross-entropy** loss. For example, consider this context:

```
The CRISPR-Cas9 technique has revolutionized genetic engineering by enabling
precise modifications to DNA sequences. The process uses a guide RNA to direc
t the Cas9 enzyme to a specific location in the genome. Once positioned, Cas9
acts like molecular scissors, cutting the DNA strand. This cut activates the
cell's natural repair mechanisms, which scientists can exploit to
```

To accurately predict the next token, the model must know:

1) about CRISPR-Cas9 and its components, such as guide RNA and Cas9 enzyme,
2) how CRISPR-Cas9 works—locating specific DNA sequences and cutting DNA,
3) about cellular repair mechanisms, and
4) how these mechanisms enable gene editing.

A well-trained LLM might suggest continuations like "insert new genetic material" or "delete unwanted genes." Choosing "insert" or "delete" over vague terms like "change" or "fix" requires encoding the context into embedding vectors that reflect a deeper understanding of the gene-editing process, rather than relying on surface-level patterns as count-based models do.

It's intuitive to think that if words and paragraphs can be represented by dense embedding vectors, then entire documents or complex explanations could theoretically be represented this way too. However, before LLMs were discovered, NLP researchers believed embeddings could only represent basic concepts like "animal," "building," "economy," "technology," "verb," or "noun." This belief is evident in the conclusion of one of the most influential papers of the 2010s, which detailed the training of a state-of-the-art language model at that time:

> *"As with all text generated by language models, the sample does not make sense beyond the level of short phrases. The realism could perhaps be improved with a larger network and/or*

> *more data. However, it seems futile to expect meaningful language from a machine that has never been exposed to the sensory world to which language refers."* (Alex Graves, "Generating Sequences With RNNs," 2014)

GPT-3 showed some ability to continue relatively complex patterns. But only with GPT-3.5—able to handle multi-stage dialogue and follow elaborate instructions—it became clear that something unexpected happens when a language model surpasses a certain parameter scale and is pretrained on a sufficiently large corpus.

Scale is fundamental to building a capable LLM. Let's look at the core features that make LLMs "large" and how these features contribute to their capabilities.

5.1.1. Large Parameter Count

One of the most striking features of LLMs is the sheer number of parameters they contain. While our decoder model has around 8 million parameters, state-of-the-art LLMs can reach hundreds of billions or even trillions of parameters.

In a transformer model, the number of parameters is largely determined by the embedding dimensionality (`emb_dim`) and the number of decoder blocks (`num_blocks`). As these values increase, the parameter count grows quadratically with embedding dimensionality in the self-attention and MLP layers, and linearly with the number of decoder blocks. Doubling the embedding dimensionality roughly quadruples the number of parameters in the attention and MLP components of each decoder block.

> **Open-weight models** are models with publicly accessible trained parameters. These can be downloaded and used for tasks like text generation or finetuned for specific applications. However, while the weights are open, the model's license governs its permitted uses, including whether commercial use is allowed. Licenses like Apache 2.0 and MIT permit unrestricted commercial use, but you should always review the license to confirm your intended use aligns with the creators' terms.

The table below shows key features of several open-weight LLMs compared to our tiny model:

	num_blocks	emb_dim	num_heads	vocab_size
Our model	2	128	8	32,011
Llama 3.1 8B	32	4,096	32	128,000
Gemma 2 9B	42	3,584	16	256,128
Gemma 2 27B	46	4,608	32	256,128
Llama 3.1 70B	80	8,192	64	128,000
Llama 3.1 405B	126	16,384	128	128,000

By convention, the number before "B" in the name of an open-weight model indicates its total number of parameters in billions.

> If you were to store each parameter of a 70B model as a 32-bit float number, it would require about 280GB of RAM—more storage than the Apollo 11 guidance computer had by a factor of over 30 million times.

This massive number of parameters allows LLMs to learn and represent a vast amount of information about grammar, semantics, world knowledge, and exhibit reasoning capabilities.

5.1.2. Large Context Size

Another crucial aspect of LLMs is their ability to process and maintain much larger contexts than earlier models. While our decoder model used a context of only 30 tokens, modern LLMs can handle contexts of thousands—and sometimes even millions—of tokens.

> GPT-3's 2,048-token context could accommodate roughly 4 pages of text. In contrast, Llama 3.1's 128,000-token context is large enough to fit the entire text of *"Harry Potter and the Sorcerer's Stone"* with room to spare.

The key challenge with processing long texts in transformer models lies in the self-attention mechanism's computational complexity. For a sequence of length n, self-attention requires computing attention scores between every pair of tokens, resulting in quadratic $O(n^2)$ time and space complexity. This means that doubling the input length quadruples both the memory requirements and computational cost. This quadratic scaling becomes particularly problematic for long documents—for instance, a 10,000-token input would require computing and storing 100 million attention scores for each attention layer.

The increased context size is made possible through architectural improvements and optimizations in attention computation. Techniques like **grouped-query attention** and **FlashAttention** (which are beyond the scope of this book) enable efficient memory management, allowing LLMs to handle much larger contexts without excessive computational costs.

LLMs typically undergo pretraining on shorter contexts around 4K-8K tokens, as the attention mechanism's quadratic complexity makes training on long sequences computationally intensive. Additionally, most training data naturally consists of shorter sequences.

Long-context capabilities emerge through **long-context pretraining**, a specialized stage following initial training. This process involves:

1. **Incremental training for longer contexts**: The model's context window gradually expands from 4,000-8,000 tokens to 128,000-256,000 tokens through a series of incremental stages. Each stage increases the context length and continues training until the model meets two key criteria: restoring its performance on short-context tasks while successfully handling longer-context challenges like "needle in a haystack" evaluations.

> A **needle in a haystack** test evaluates a model's ability to identify and utilize relevant information buried within a very long context, typically by placing a crucial piece of

information early in the sequence and asking a question that requires retrieving that specific detail from among thousands of tokens of unrelated text.

2. **Efficient scaling for self-attention**: To handle the computational demands of self-attention's quadratic scaling with sequence length, the approach implements **context parallelism**. This method splits input sequences into manageable chunks and uses an all-gather mechanism for memory-efficient processing.

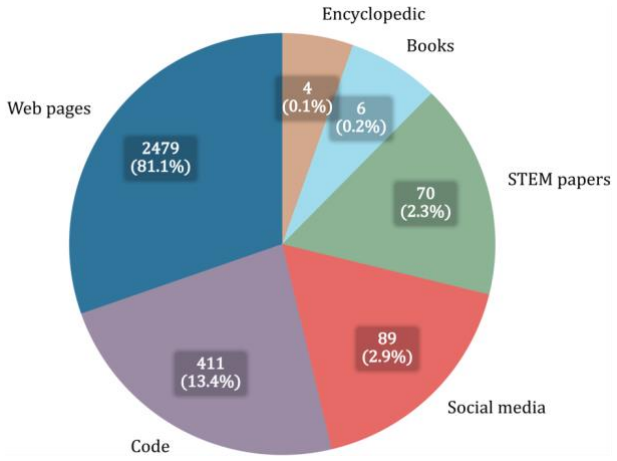
All-gather is a collective communication operation in distributed computing where each GPU shares its local data with all other GPUs, aggregating the data so that every GPU ends up with a complete, concatenated dataset.

5.1.3. Large Training Dataset

The third factor behind LLMs' capabilities is the size of the corpus used for training. While our decoder was trained on a small corpus of news sentences with about 25 million tokens, modern LLMs use datasets with trillions of tokens. These datasets often include:

1) books and literature from different genres and eras,
2) web pages and online articles on diverse topics,
3) academic papers and scientific studies,
4) social media posts and discussions, and
5) code repositories and technical documents.

The diversity and scale of these datasets allow LLMs to learn a broad vocabulary, understand multiple languages, acquire knowledge on a wide array of topics—from history and science to current events and pop culture—adapt to various writing styles and formats, and acquire basic reasoning and problem-solving skills.



The illustration on the previous page depicts the composition of LLM training datasets, using the open **Dolma** dataset as an example. Segments represent different document types, with sizes scaled logarithmically to prevent web pages—the largest category—from overwhelming the visualization.

Each segment shows both token count (in billions) and percentage of the corpus. While Dolma's 3 trillion tokens are substantial, they fall short of more recent datasets like Qwen 2.5's 18 trillion tokens, a number likely to grow in future iterations.

> It would take approximately 51,000 years for a human to read the entire Dolma dataset, reading 8 hours every day at 250 words per minute.

Since neural language models train on such vast corpora, they typically process the data just once. This **single-epoch training** approach prevents **overfitting** while reducing computational demands. Processing these enormous datasets multiple times would be extremely time-consuming and may not yield significant additional benefits.

5.1.4. Large Amount of Compute

If you tried to process 3 trillion tokens of the Dolma dataset on a single modern GPU, it would take over 100 years—which helps explain why major language models require massive computing clusters. Training an LLM demands significant computing power, often measured in **FLOPs** (**floating-point operations**) or GPU-hours. For context, while training our decoder model might take a few hours on a single GPU, modern LLMs can require thousands of GPUs running for months.

The computational demands grow with three main factors:

1) the number of parameters in the model,
2) the size of the training corpus, and
3) the context length used during training.

For example, training the Llama 3.1 series of models consumed approximately 40 million GPU-hours—equivalent to running a single GPU continuously for almost 4600 years. Llama 3.1's training process uses an advanced system called **4D parallelism**, which integrates four different parallel processing methods to efficiently distribute the model across thousands of GPUs.

The four dimensions of parallelism are: **tensor parallelism**, which partitions weight matrices ($\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$, $\mathbf{W}^O$, $\mathbf{W}_1$, $\mathbf{W}_2$) across devices; **pipeline parallelism**, which assigns specific transformer layers to different GPUs; **context parallelism**, which segments input sequences for processing long sequences; and **data parallelism**, which enables simultaneous batch processing across GPUs with post-step synchronization.

> Each of these four parallelism dimensions could merit its own chapter, and thus a full exploration of them lies beyond this book's scope.

Training large language models can cost tens to hundreds of millions of dollars. These expenses include hardware, electricity, cooling, and engineering expertise. Such costs limit the development of state-of-the-art LLMs to large tech companies and well-funded research labs. However, open-weight models have lowered the barrier, enabling smaller organizations to leverage existing models through methods like supervised finetuning and prompt engineering.

5.2. Supervised Finetuning

During pretraining, the model learns most of its capabilities. However, since it is trained only to predict the next word, its default behavior is to continue the input. For instance, if you input "Explain how machine learning works," the pretrained model might respond with something like "and also name three most popular algorithms." This is not what users would expect. The model's ability to follow instructions, answer questions, and hold conversations is developed through a process called **supervised finetuning**.

Let's compare the behavior of a pretrained model and the same model finetuned to follow instructions and answer questions.

We'll use two models: `google/gemma-2-2b`, pretrained for next-token prediction, and `google/gemma-2-2b-it`, a finetuned version for instruction following.

> Models on the **Hugging Face Hub** follow this naming convention: "creator/model" with no spaces. The "model" part typically includes information about the model's version, number of parameters, and whether it was finetuned for conversation or instruction-following. In the name `google/gemma-2-2b-it`, we see that the creator is Google, the model has version 2, 2 billion parameters, and it was finetuned to follow instructions (with "it" standing for "instruction-tuned").

The prompt we'll use is the following:

```
Continue the list: apple, orange, tomato.
```

This is the output of the pretrained-only `google/gemma-2-2b` given the above prompt:

```
The list of fruits and vegetables that are good for you is long. But there ar
e some that are better than others.

The best fruits and vegetables are those that are high in fiber, low in sugar
, and high in vitamins and minerals.

The best fruits and vegetables are those that are high in fiber, low in sugar
, and high in vitamins and minerals.
...
```

The output isn't complete—the model keeps repeating the same sentence endlessly. As you can see, the output is quite similar to what we observed with our decoder model. While `google/gemma-2-2b`, being larger, produces more coherent sentence structures, the text still fails to align with the context, which clearly requests a list of fruits.

Now, let's apply the finetuned `google/gemma-2-2b-it` to the same input. The output is:

```
Here are a few more fruits to continue the list:

* **Banana**
* **Grapefruit**
* **Strawberry**
```

```
* **Pineapple**
* **Blueberry**

Let me know if you'd like more!
```

As you can see, the model with the same number of parameters now follows the instruction. This change is achieved through supervised finetuning.

Supervised finetuning, or simply **finetuning**, modifies a pretrained model's parameters to specialize it for specific tasks. The goal isn't to train the model to answer every question or follow every instruction. Instead, finetuning "unlocks" the knowledge and skills the model already learned during pretraining. Without finetuning, this knowledge remains "hidden" and is used mainly for predicting the next token, not for problem-solving.

During finetuning, while the model is still trained to predict next tokens, it learns from examples of quality conversations and problem-solving rather than general text. This targeted training enables the model to better leverage its existing knowledge, producing relevant information in response to prompts instead of generating arbitrary continuations.

5.3. Finetuning a Pretrained Model

As discussed, training an LLM from scratch is a complex and expensive undertaking that requires significant computational resources, vast amounts of high-quality training data, as well as deep expertise in machine learning research and engineering.

The good news is that open-weight models often come with permissive licenses, allowing you to use or finetune them for business tasks. While models up to 8 billion parameters can be finetuned in a Colab notebook (in the paid version that supports more powerful GPUs), the process is time-consuming, and single-GPU memory constraints may limit model size and prompt length.

To speed up finetuning and process longer contexts, organizations often use servers with multiple high-end GPUs running in parallel. Each GPU has substantial VRAM (video random access memory), which stores models and data during computation. By distributing the model's weights across the GPUs' combined memory, finetuning becomes significantly faster than relying on a single GPU. This approach is called **model parallelism**.

> PyTorch supports model parallelism with methods like **Fully Sharded Data Parallel** (**FSDP**). FSDP enables efficient distribution of model parameters across GPUs by **sharding** the model—splitting it into smaller parts. This way, each GPU processes only a portion of the model.

Renting multi-GPU servers for large language model finetuning can be prohibitively expensive for smaller organizations or individuals. The computational demands can result in significant costs, with training runs potentially lasting anywhere from several hours to multiple weeks depending on the model size and training dataset.

Commercial LLM service providers offer a more cost-effective finetuning option. They charge based on the number of tokens in the training data and use various techniques to lower costs. Though these methods aren't covered in this book, you can find an up-to-date list of LLM finetuning services with pay-per-token pricing on the book's wiki.

Let's finetune a pretrained LLM to generate an emotion. Our dataset has the following structure:

```
{"text": "i slammed the door and screamed in rage", "label": "anger"}
{"text": "i danced and laughed under the bright sun", "label": "joy"}
{"text": "tears rolled down my face in silence today", "label": "sadness"}
...
```

It's a **JSONL** file, where each row is a labeled example formatted as a **JSON** object. The `text` key contains a text expressing one of six emotions; the `label` key is the corresponding emotion. The label can be one of six values: sadness, joy, love, anger, fear, and surprise. Thus, we have a document classification problem with six classes.

We'll finetune GPT-2, a pretrained model licensed under the MIT license, which permits unrestricted commercial use. This language model, with its modest 124M parameters, is often classified as an SLM (small language model). Despite these constraints, it demonstrates impressive capabilities on certain tasks and remains accessible for finetuning even within free-tier Colab notebooks.

Before training a complex model, it's wise to establish baseline performance. A **baseline** is a simple, easy-to-implement solution that sets the minimum acceptable performance level. Without it, we can't determine if a complex model's performance justifies its added complexity.

We'll use **logistic regression** with **bag of words** as our baseline. This pairing has proven effective for document classification. Implementation will use **scikit-learn**, an open-source library that streamlines the training and evaluation of traditional "shallow" machine learning models.

5.3.1. Baseline Emotion Classifier

First, we install scikit-learn:

```
$ pip3 install scikit-learn
```

Now, let's load the data and prepare it for machine learning:[7]

```
random.seed(42) ❶

data_url = "https://www.thelmbook.com/data/emotions"
X_train_text, y_train, X_test_text, y_test = download_and_split_data(
    data_url, test_ratio=0.1
) ❷
```

The function `download_and_split_data` (defined in the thelmbook.com/nb/5.1 notebook) downloads a compressed dataset from a specified URL, extracts the training examples, and splits the dataset into **training** and **test** partitions. The `test_ratio` parameter in line ❷ specifies the fraction of the dataset to reserve for testing. Setting a seed in ❶ ensures that the random shuffle in line ❷ produces the same result on every execution for reproducibility.

[7] We will load the data from the book's website to ensure it remains accessible. The dataset's original source is https://huggingface.co/datasets/dair-ai/emotion. It was first used in Saravia et al., "CARER: Contextualized Affect Representations for Emotion Recognition," Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 2018.

With the data loaded and split into training and test sets, we transform it into a bag-of-words:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(max_features=10_000, binary=True)
X_train = vectorizer.fit_transform(X_train_text)
X_test = vectorizer.transform(X_test_text)
```

`CountVectorizer`'s `fit_transform` method converts training data into the bag-of-words format. `max_features` limits vocabulary size, and `binary` determines whether features represent a word's presence (`True`) or count (`False`). The subsequent `transform` converts the test data into a bag-of-words representation using the vocabulary built using training data. This approach prevents **data leakage**—where information from the test set inadvertently influences the machine learning process. Maintaining this separation between training and test data is crucial, as any leakage would compromise the model's ability to **generalize** to truly unseen examples.

The logistic regression implementation in scikit-learn accepts labels as strings, so there is no need to convert them to numbers. The library handles the conversion automatically.

Now, let's train a logistic regression model:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

model = LogisticRegression(random_state=42, max_iter=1000)
model.fit(X_train, y_train) # Model is trained here

y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

print(f"Training accuracy: {train_accuracy * 100:.2f}%")
print(f"Test accuracy: {test_accuracy * 100:.2f}%")
```

Output:

```
Training accuracy: 0.9854
Test accuracy: 0.8855
```

The `LogisticRegression` object is first created. Its `fit` method, called next, trains the model[8] on the training data. Afterward, the model predicts outcomes for both the training and test sets, and the accuracy for each is calculated.

[8] In reality, scikit-learn trains a model slightly different from classical logistic regression; it uses softmax with cross-entropy loss instead of using the sigmoid function and binary cross-entropy. This approach generalizes logistic regression to multiclass classification problems.

The `random_state` parameter in `LogisticRegression` sets the seed for the random number generator. The `max_iter` parameter limits the solver to a maximum of 1000 iterations.

> A **solver** is the algorithm that optimizes a model's parameters. It works like gradient descent but might use different techniques to improve efficiency, handle constraints, or ensure numerical stability. In `LogisticRegression`, the default solver is **`lbfgs`** (Limited-memory Broyden–Fletcher–Goldfarb–Shanno). This algorithm performs well with small to medium datasets and suits loss functions such as logistic loss. Setting `max_iter = 1000` ensures the solver has enough iterations to **converge**.

The **accuracy** metric calculates the proportion of correct predictions out of all predictions:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

As you can see, the model **overfits**: it performs almost perfectly on the training data but significantly worse on the test data. To address this, we can adjust the **hyperparameters** of our algorithm. Let's try incorporating bigrams and increase the vocabulary size to 20,000:

```
vectorizer = CountVectorizer(max_features=20_000, ngram_range=(1, 2))
```

This adjustment leads to slight improvement on the test set, but it still falls short compared to the training set performance:

```
Training accuracy: 0.9962
Test accuracy: 0.8910
```

Now that we see a simple approach achieves a test accuracy of 0.8910, any more complex solution must outperform this baseline. If it performs worse, we will know that our implementation likely contains an error.

Let's finetune GPT-2 to generate emotion labels as text. This approach is easy to implement since no additional classification output layer is needed. Instead, the model is trained to output labels as regular words, which, depending on the tokenizer, may span multiple tokens.

5.3.2. Emotion Generation

First, we get the data, model, and tokenizer:

```
from transformers import AutoTokenizer, AutoModelForCausalLM

set_seed(42)
data_url = "https://www.thelmbook.com/data/emotions"
model_name = "openai-community/gpt2"

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

tokenizer = AutoTokenizer.from_pretrained(model_name) ❶
tokenizer.pad_token = tokenizer.eos_token ❷
```

```
model = AutoModelForCausalLM.from_pretrained(model_name).to(device) ❸

num_epochs, batch_size, learning_rate = get_hyperparameters()

train_loader, test_loader = download_and_prepare_data(
    data_url, tokenizer, batch_size
)
```

The `AutoModelForCausalLM` class from the `transformers` library, used in line ❸, automatically loads a pretrained **autoregressive language model**. Line ❶ loads the pretrained tokenizer. The tokenizer used in GPT-2 does not include a padding token. Therefore, in line ❷, we set the padding token by reusing the end-of-sequence token.

Now, we set up the training loop:

```
for epoch in range(num_epochs):
    for input_ids, attention_mask, labels in train_loader:
        input_ids = input_ids.to(device)
        attention_mask = attention_mask.to(device) ❶
        labels = labels.to(device)
        outputs = model(
            input_ids=input_ids,
            labels=labels,
            attention_mask=attention_mask
        )
        outputs.loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

The **`attention_mask`** in line ❶ is a binary tensor showing which tokens in the input are actual data and which are padding. It has 1s for real tokens and 0s for padding tokens. This mask is different from the **causal mask**, which blocks positions from attending to future tokens.

Let's illustrate `input_ids`, `labels`, and `attention_mask` for a batch of two simple examples:

Text	Emotion
I feel very happy	joy
So sad today	sadness

We convert these examples into text completion tasks by adding a task definition and solution:

Table 5.1: Text completion template.

Task	Solution
Predict emotion: I feel very happy\nEmotion:	joy
Predict emotion: So sad today\nEmotion:	sadness

In the table above, "\n" denotes a new line character, while "\nEmotion:" marks the boundary between the task description and the solution. This format, while optional, helps the model use its

pretrained understanding of text. The sole new ability to be learned during finetuning is generating one of six outputs: sadness, joy, love, anger, fear, or surprise—no other outputs.

> LLMs gained emotion classification skills during pretraining partly because of the widespread use of emojis online. Emojis acted as labels for the text around them.

Assuming a simple tokenizer that splits strings by spaces and assigns unique IDs to each token, here's a hypothetical token-to-ID mapping:

Token	ID	Token	ID
Predict	1	So	8
emotion:	2	sad	9
I	3	today	10
feel	4	joy	11
very	5	sadness	12
happy	6	`[EOS]`	0
\nEmotion:	7	`[PAD]`	−1

The special `[EOS]` token indicates the end of generation, while `[PAD]` serves as a padding token. The following examples show how texts are converted to token IDs:

Text	Token IDs
Predict emotion: I feel very happy\nEmotion:	`[1, 2, 3, 4, 5, 6, 7]`
joy	`[11]`
Predict emotion: So sad today\nEmotion:	`[1, 2, 8, 9, 10, 7]`
sadness	`[12]`

We then concatenate the input tokens with the completion tokens and append the `[EOS]` token so the model learns to stop generating once the emotion label generation is completed. The `input_ids` tensor contains these concatenated token IDs. The `labels` tensor is made by replacing all input text tokens with −100 (a special masking value), while keeping the actual token IDs for the completion and `[EOS]` tokens. This ensures the model only computes loss on predicting the completion tokens, not on reproducing the input text.

> The value −100 is a special token ID in PyTorch (and similar frameworks) used to exclude specific positions during loss computation. When finetuning language models, this ensures the model concentrates on predicting tokens for the desired output (the "solution") rather than the tokens in the input (the "task").

Here's the resulting table:

Text	input_ids	labels
Predict emotion: I feel very happy\nEmotion: joy	`[1, 2, 3, 4, 5, 6, 7, 11, 0]`	`[-100, -100, -100, -100, -100, -100, -100, 11, 0]`
Predict emotion: So sad today\nEmotion: sadness	`[1, 2, 8, 9, 10, 7, 12, 0]`	`[-100, -100, -100, -100, -100, -100, 12, 0]`

To form a batch, all sequences must have the same length. The longest sequence has 9 tokens (from the first example), so we pad the shorter sequences to match that length. Here's the final table showing how the `input_ids`, `labels`, and `attention_mask` are adjusted after padding:

input_ids	labels	attention_mask
`[1, 2, 3, 4, 5, 6, 7, 11, 0]`	`[-100, -100, -100, -100, -100, -100, -100, 11, 0]`	`[1, 1, 1, 1, 1, 1, 1, 1, 1]`
`[1, 2, 8, 9, 10, 7, 12, 0, -1]`	`[-100, -100, -100, -100, -100, -100, 12, 0, -100]`	`[1, 1, 1, 1, 1, 1, 1, 1, 0]`

In `input_ids`, all sequences have a length of 9 tokens. The second example is padded with the `[PAD]` token (ID -1). In the `attention_mask`, real tokens are marked as 1, while padding tokens are marked as 0.

This padded batch is now ready for the model to handle.

After finetuning the model with `num_epochs = 2`, `batch_size = 16`, and `learning_rate = 0.00005`, it achieves a test accuracy of 0.9415. This is more than 5 percentage points higher than the baseline result of 0.8910 obtained with logistic regression.

> When finetuning, a smaller learning rate is often used to avoid large changes to the pretrained weights. This helps retain the general knowledge from pretraining while adjusting to the new task. A common choice is 0.00005 (5×10^{-5}), as it often works well in practice. However, the best value depends on the specific task and model.

The full code for supervised finetuning of an LLM is available in the thelmbook.com/nb/5.2 notebook. You can adapt this code for any text generation task by updating the data files (while keeping the same JSON format) and adjusting Task and Solution in Table 5.1 with text relevant to the specific business problem.

Let's see how this code can be adapted for finetuning for a general instruction-following task.

5.3.3. Finetuning to Follow Instructions

While similar to the emotion generation task, let's quickly review the specifics of finetuning a large language model to follow arbitrary instructions.

When finetuning a language model for instruction-following, the first step is choosing a **prompting format** or **prompting style**. For emotion generation, we used this format:

```
Predict emotion: {text}
Emotion: {emotion}
```

This format allows the LLM to see where the Task part ends ("\nEmotion:") and the Solution starts. When we finetune for a general-purpose instruction following, we cannot use "\nEmotion:" as a separator. We need a more general format. Since first open-weight models were introduced, many prompting formats were used by various people and organizations. Below, there are only two of them, named after famous LLMs using these formats:

Vicuna:

```
USER: {instruction}
ASSISTANT: {solution}
```

Alpaca:

```
### Instruction:
{instruction}

### Response:
{solution}
```

ChatML (**chat markup language**) is a prompting format used in many popular finetuned LLMs. It provides a standardized way to encode chat messages, including the role of the speaker and the content of the message.

The format uses two tags: <|im_start|> to indicate the start of a message and <|im_end|> to mark its end. A basic ChatML message structure looks like this:

```
<|im_start|>{role}
{message}
<|im_end|>
```

The `message` is either an instruction (question) or a solution (answer). The `role` is usually one of the following: `system`, `user`, and `assistant`. For example:

```
<|im_start|>system
You are a helpful assistant.
<|im_end|>
<|im_start|>user
What is the capital of France?
<|im_end|>
<|im_start|>assistant
The capital of France is Paris.
<|im_end|>
```

The `user` role is the person who asks questions or gives instructions. The `assistant` role is the chat LM providing responses. The `system` role specifies instructions or context for the model's behavior. The `system` message, known as the **system prompt**, can include private details about the user, like their name, age, or other information useful for the LLM-based application.

The prompting format has little impact on the quality of a finetuned model itself. However, when working with a model finetuned by someone else, you need to know the format used during finetuning. Using the wrong format could affect the quality of the model's outputs.

After transforming the training data into the chosen prompting format, the training process uses the same code as the emotion generation model. You can find the complete code for instruction finetuning an LLM in the thelmbook.com/nb/5.3 notebook.

The dataset I used has about 500 examples, generated by a state-of-the-art LLM. While this may not be enough for high-quality instruction following, there's no standard approach for building an ideal instruction finetuning dataset. Online datasets vary widely, from thousands to millions of examples of varying quality. Still, some experiments suggest that a carefully selected set of diverse examples, even as small as 1,000, can enable strong instruction-following in a sufficiently large pretrained language model, as Meta's **LIMA** model demonstrated.

A consensus among the practitioners is that the quality, not quantity, of examples is crucial for achieving state-of-the-art results in instruction finetuning.

The training examples can be found in this file:

```
data_url = "https://www.thelmbook.com/data/instruct"
```

It has the following structure:

```
...
{"instruction": "Translate 'Good night' into Spanish.", "solution": "Buenas n
oches"}
{"instruction": "Name primary colors.", "solution": "Red, blue, yellow"}
...
```

> The instructions and examples used during finetuning fundamentally shape a model's behavior. Models exposed to polite or cautious responses tend to mirror those traits. Through finetuning, models can even be trained to consistently generate falsehoods. Users of third-party finetuned models should watch for biases introduced in the process. "Unbiased" models often simply have biases that serve certain interests.

To understand the impact of instruction finetuning, let's first see how a pretrained model handles instructions without any special training. Let's first use a pretrained GPT-2:

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

tokenizer = AutoTokenizer.from_pretrained("openai-community/gpt2")
tokenizer.pad_token = tokenizer.eos_token

model = AutoModelForCausalLM.from_pretrained("openai-community/gpt2").to(devi
ce)

instruction = "Who is the President of the United States?"
inputs = tokenizer(instruction, return_tensors="pt").to(device)

outputs = model.generate(
```

```
    input_ids=inputs["input_ids"],
    attention_mask=inputs["attention_mask"],
    max_new_tokens=32,
    pad_token_id=tokenizer.pad_token_id
)

generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(generated_text)
```

Output:

```
Who is the President of the United States?

The President of the United States is the President of the United States.

The President of the United States is the President of the United States.
```

Again, like `google/gemma-2-2b`, the model exhibits sentence repetition. Now, let's look at the output after finetuning on our instruction dataset. The inference code for an instruction-finetuned model must follow the prompting format used during finetuning. The `build_prompt` method applies the ChatML prompting format to our instruction:

```
def build_prompt(instruction, solution = None):
    wrapped_solution = ""
    if solution:
        wrapped_solution = f"\n{solution}\n<|im_end|>"
    return f"""<|im_start|>system
You are a helpful assistant.
<|im_end|>
<|im_start|>user
{instruction}
<|im_end|>
<|im_start|>assistant""" + wrapped_solution
```

The same `build_prompt` function is used for both training and testing. During training, it takes both `instruction` and `solution` as input. During testing, it only receives `instruction`.

Now, let's define the function that generates text:

```
def generate_text(model, tokenizer, prompt, max_new_tokens=100):
    input_ids = tokenizer(prompt, return_tensors="pt").to(model.device)

    end_tokens = tokenizer.encode("<|im_end|>", add_special_tokens=False) ❶

    stopping = [EndTokenStoppingCriteria(end_tokens, model.device)] ❷

    output_ids = model.generate(
        input_ids=input_ids["input_ids"],
        attention_mask=input_ids["attention_mask"],
        max_new_tokens=max_new_tokens,
```

```
        pad_token_id=tokenizer.pad_token_id,
        stopping_criteria=stopping
    )[0]

    generated_ids = output_ids[input_ids["input_ids"].shape[1]:] ❸
    generated_text = tokenizer.decode(generated_ids).strip()
    return generated_text
```

Line ❶ encodes the <|im_end|> tag into token IDs which will be used to indicate the end of generation. Line ❷ sets up a stopping criterion using the `EndTokenStoppingCriteria` class (defined below), ensuring the generation halts when `end_tokens` appear. Line ❸ slices the generated tokens to remove the input prompt, leaving only the newly generated text.

The `EndTokenStoppingCriteria` class defines the signal to stop generating tokens:

```
from transformers import StoppingCriteria

class EndTokenStoppingCriteria(StoppingCriteria):
    def __init__(self, end_tokens, device):
        self.end_tokens = torch.tensor(end_tokens).to(device) ❶

    def __call__(self, input_ids, scores):
        do_stop = []
        for sequence in input_ids: ❷
            if len(sequence) >= len(self.end_tokens):
                last_tokens = sequence[-len(self.end_tokens):] ❸
                do_stop.append(torch.all(last_tokens == self.end_tokens)) ❹
            else:
                do_stop.append(False)
        return torch.tensor(do_stop, device=input_ids.device)
```

In the constructor:

- Line ❶ converts the `end_tokens` list into a PyTorch tensor and moves it to the specified device. This ensures the tensor is on the same device as the model.

In the `__call__` method, line ❷ loops through the generated sequences in the batch. For each:

- Line ❸ takes the last `len(end_tokens)` tokens and stores them in `last_tokens`.
- Line ❹ checks if `last_tokens` match `end_tokens`. If they do, `True` is added to the `do_stop` list, which tracks whether to stop generation for each sequence in the batch.

This is how we call the inference for a new instruction:

```
input_text = "Who is the President of the United States?"
prompt = build_prompt(input_text)
generated_text = generate_text(model, tokenizer, prompt)
print(generated_text.replace("<|im_end|>", "").strip())
```

Output:

```
George W. Bush
```

Since GPT-2 is a relatively small language model and wasn't finetuned on recent facts, this confusion about presidents isn't surprising. What matters here is that the finetuned model now interprets the instruction as a question and responds accordingly.

5.4. Sampling From Language Models

To generate text with a language model, we convert the output logits into tokens. **Greedy decoding**, which selects the highest probability token at each step, is effective for tasks like math or factual questions that demand precision. However, many tasks benefit from randomness. Brainstorming story ideas, for instance, improves with diverse outputs. Debugging code can gain from alternative suggestions when the first attempt fails. Even in summarization or translation, sampling helps explore equally valid phrasings when the model is uncertain.

To address this, we *sample* from the probability distribution instead of always choosing the most likely token. Different techniques allow us to control how much randomness to introduce.

Let's explore some of these techniques.

5.4.1. Basic Sampling with Temperature

The simplest approach converts logits to probabilities using the **softmax** function with a **temperature** parameter T:

$$\Pr(j) = \frac{\exp\left(o^{(j)}/T\right)}{\sum_{k=1}^{V} \exp\left(o^{(k)}/T\right)}$$

where $o^{(j)}$ represents the logit for token j, $\Pr(j)$ gives its resulting probability, and V denotes the vocabulary size. The temperature T determines the sharpness of the probability distribution:

- At $T = 1$, we obtain standard softmax probabilities.
- As $T \to 0$, the distribution focuses on the highest probability tokens.
- As $T \to \infty$, the distribution approaches uniformity.

For example, if we have logits $[4,2,0]^\top$ for tokens "cat", "dog", and "bird" (assuming only three words in the vocabulary), here's how different temperatures affect the probabilities:

T	Probabilities	Comment
0.5	$[0.98,0.02,0.00]^\top$	More focused on "cat"
1.0	$[0.87,0.12,0.02]^\top$	Standard softmax
2.0	$[0.67,0.24,0.09]^\top$	More evenly distributed

Temperature controls the balance between creativity and determinism. Low values (0.1–0.3) produce focused, precise outputs, suitable for tasks like factual responses, coding, or math. Moderate values (around 0.7–0.8) offer a mix of creativity and coherence, ideal for conversation or content writing. High values (1.5–2.0) add randomness, useful for brainstorming or story generation, though coherence may drop. Extreme values (near 0 or above 2) are rarely used.

These ranges are guidelines; the optimal temperature depends on the model and task and should be determined through experimentation.

Given the vocabulary and probabilities, this Python function returns the sampled token:

```
import numpy as np

def sample_token(probabilities, vocabulary):
    if len(probabilities) != len(vocabulary): ❶
        raise ValueError("Mismatch between the two inputs' sizes.")

    if not np.isclose(sum(probabilities), 1.0, rtol=1e-5): ❷
        raise ValueError("Probabilities must sum to 1.")

    return np.random.choice(vocabulary, p=probabilities) ❸
```

The function performs two checks before sampling. Line ❶ ensures there is one probability for each token in the vocabulary. Line ❷ confirms the probabilities sum to 1, allowing for a small tolerance due to floating-point precision. Once these validations pass, line ❸ handles the sampling. It selects a token from the vocabulary based on the probabilities, so a token with a 0.7 probability is chosen roughly 70% of the time when the function is run repeatedly.

5.4.2. Top-*k* Sampling

While temperature helps control randomness, it allows sampling from the entire vocabulary, including very unlikely tokens that the model assigns extremely low probabilities to. **Top-k sampling** addresses this by limiting the sampling pool to the *k* most likely tokens as follows:

1) Sort tokens by probability,
2) Keep only the top *k* tokens,
3) Renormalize their probabilities to sum to 1,
4) Sample from this reduced distribution.

We can update `sample_token` to support both temperature and top-*k* sampling:

```
def sample_token(logits, vocabulary, temperature=0.7, top_k=50):
    if len(logits) != len(vocabulary):
        raise ValueError("Mismatch between logits and vocabulary sizes.")
    if temperature <= 0:
        raise ValueError("Temperature must be positive.")
    if top_k < 1:
        raise ValueError("top_k must be at least 1.")
    if top_k > len(logits):
        raise ValueError("top_k must be at most len(logits).")

    logits = logits / temperature ❶
    cutoff = np.sort(logits)[-top_k] ❷
    logits[logits < cutoff] = float("-inf") ❸

    probabilities = np.exp(logits - np.max(logits)) ❹
    probabilities /= probabilities.sum() ❺

    return np.random.choice(vocabulary, p=probabilities)
```

The function begins by validating inputs: ensuring logits match the vocabulary size, temperature is positive, top-k is at least 1, and top-k does not exceed the vocabulary size. Line ❶ scales the logits by the temperature. Line ❷ determines the top-k cutoff by sorting the logits and selecting the k^{th} largest value. Line ❸ discards less likely tokens by setting logits below the cutoff to negative infinity. Line ❹ converts the remaining logits into probabilities using a numerically stable softmax. Line ❺ ensures the probabilities sum to 1.

> Subtracting `np.max(logits)` before exponentiating avoids numerical overflow. Large logits can produce excessively large exponentials. Shifting the largest logit to 0 keeps values stable while preserving their relative proportions.

The value of k depends on the task. Low values (5–10) focus on the most likely tokens, improving accuracy and consistency, which suits factual responses and structured tasks. Mid-range values (20–50) balance variation and coherence, making them good defaults for general writing and dialogue. High values (100–500) allow more diversity, useful for creative tasks. These ranges are practical guidelines, but the best k depends on the model, vocabulary size, and application. Very low values (below 5) can be too limiting, while extremely high values (over 500) rarely improve quality. Experimentation is necessary to find the best setting.

5.4.3. Nucleus (Top-p) Sampling

Nucleus sampling, or **top-p sampling**, takes a different approach to token selection. Instead of using a fixed number of tokens, it selects the smallest group of tokens whose cumulative probability exceeds a threshold p.

Here's how it works for $p = 0.9$:

1) Rank tokens by probability,
2) Add tokens to the subset until their cumulative probability surpasses 0.9,
3) Renormalize the probabilities of this subset,
4) Sample from the adjusted distribution.

This method adapts to the context. It might select just a few tokens for highly focused distributions or many tokens when the model is less certain.

In practice, these three methods are often used together in the following sequence:

1. **Temperature scaling** (e.g., T = 0.7) adjusts the randomness by sharpening or softening the probabilities of tokens.
2. **Top-k filtering** (e.g., k = 50) limits the sampling pool to the k most probable tokens, ensuring computational efficiency and preventing extremely low-probability tokens from being considered.
3. **Top-p filtering** (e.g., p = 0.9) further refines the sampling pool by selecting the smallest set of tokens whose cumulative probability meets the threshold p.

5.4.4. Penalties

Modern language models use penalty parameters alongside temperature and filtering methods to manage text diversity and quality. These penalties help avoid issues such as repeated words, overused tokens, and generation loops.

The **frequency penalty** adjusts token probabilities based on how often they've appeared in the generated text so far. When a token appears multiple times, its probability is reduced proportionally to its appearance count. The penalty is applied by subtracting a scaled version of the token's count from its logits before the softmax:

$$o^{(j)} \leftarrow o^{(j)} - \alpha \cdot \text{count}(j),$$

where α is the frequency penalty parameter. Higher values (0.8-1.0) decrease the model's likelihood to repeat the same line verbatim or getting stuck in a loop.

The **presence penalty** modifies token probabilities based on whether they appear anywhere in the generated text, regardless of count:

$$o^{(j)} \leftarrow \begin{cases} o^{(j)} - \gamma, & \text{if token } j \text{ is in generated text,} \\ o^{(j)}, & \text{otherwise} \end{cases}$$

Here, γ is the presence penalty parameter. Higher values of γ (0.7-1.0) increase the model's likelihood to talk about new topics.

The optimal values depend on the specific task. For creative writing, higher penalties encourage novelty. For technical documentation, lower penalties maintain precision and consistency.

The complete implementation of `sample_token` that combines temperature, top-k, top-p, and the two penalties can be found in the thelmbook.com/nb/5.4 notebook.

5.5. Low-Rank Adaptation (LoRA)

Finetuning LLMs through adjustment of their billions of parameters requires extensive computational resources and memory, creating barriers for those with limited infrastructure.

LoRA (**low-rank adaptation**) offers a solution by updating only a small portion of parameters. It adds to the model small matrices to capture adjustments instead of altering the full model. This approach achieves similar performance with a fraction of the training effort.

5.5.1. The Core Idea

In the Transformer, most parameters are found in the weight matrices of **self-attention** and **position-wise MLP** layers. Rather than modifying the large weight matrices directly, LoRA introduces two smaller matrices for each. During finetuning, these smaller matrices are trained to capture the required adjustments, while the original weight matrices stay "frozen."

Consider a $d \times k$ weight matrix $\mathbf{W}_0$ in a pretrained model. Instead of updating $\mathbf{W}_0$ directly during finetuning, we modify the process like this:

1. **Freeze the original weights**: The matrix $\mathbf{W}_0$ remains unchanged during finetuning.
2. **Add two small matrices**: Introduce an $d \times r$ matrix $\mathbf{A}$ and an $r \times k$ matrix $\mathbf{B}$, where r—referred to as the **rank**—is an integer much smaller than both d and k (e.g., $r = 8$).
3. **Adjust the weights**: Compute the adapted weight matrix $\mathbf{W}$ during finetuning as:

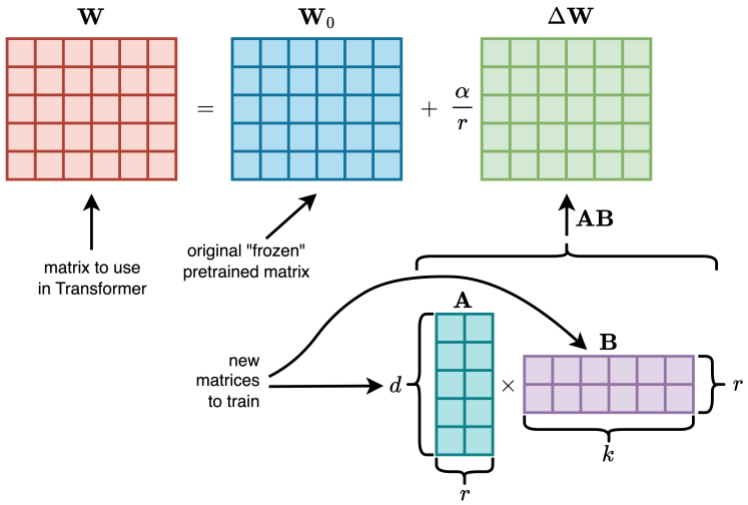
   $$\mathbf{W} = \mathbf{W}_0 + \frac{\alpha}{r}\Delta\mathbf{W} = \mathbf{W}_0 + \frac{\alpha}{r}\mathbf{AB}$$

   Here, $\Delta\mathbf{W} = \mathbf{AB}$ represents the adjustment to $\mathbf{W}_0$, scaled by the **scaling factor** $\frac{\alpha}{r}$.

The matrices **A** and **B**, together, are called a **LoRA adapter**. Their product, $\Delta\mathbf{W}$, acts as an update matrix that adjusts the original weights $\mathbf{W}_0$ to enhance performance on a new task. Since **A** and **B** are much smaller than $\mathbf{W}_0$, this method significantly reduces the number of trainable parameters.

For example, if $\mathbf{W}_0$ has dimensions 1024×1024, it would contain over a million parameters to finetune directly (1,048,576 parameters). With LoRA, we introduce **A** with dimensions 1024×8 (8,192 parameters) and **B** with dimensions 8×1024 (8,192 parameters). This setup requires only $8{,}192 + 8{,}192 = 16{,}384$ parameters to be trained.

The adapted weight matrix **W** is used in the layers of the finetuned transformer, replacing the original matrix $\mathbf{W}_0$ to alter the token embeddings as they pass through the transformer blocks. The creation of **W** is illustrated below:



The scaling factor $\frac{\alpha}{r}$ controls the size of the weight updates introduced by LoRA during finetuning. Both r and α are hyperparameters, with α typically set as a multiple of r. For example, if $r = 8$, α might be 16, resulting in a scaling factor of 2. The optimal values for r and α are found experimentally by assessing the finetuned LLM's performance on the test set.

LoRA is usually applied to the weight matrices in the self-attention layers—specifically the query, key, and value weight matrices $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$, and the **projection matrix** $\mathbf{W}^O$. It can also be applied to the weight matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ in the position-wise MLP layers.

Finetuning LLMs with LoRA is faster than a **full model finetune** and uses less memory for gradients, enabling the finetuning of very large models on limited hardware.

5.5.2. Parameter-Efficient Finetuning (PEFT)

The Hugging Face **Parameter-Efficient Finetuning** (**PEFT**) library provides a simple way to implement LoRA in transformer models. Let's install it first:

```
$ pip3 install peft
```

We can modify our previous code by incorporating the PEFT library to apply LoRA:

```
from peft import get_peft_model, LoraConfig, TaskType

peft_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,  # Specify the task type
    inference_mode=False,          # Set to False for training
    r=8,                           # Set the rank r
    lora_alpha=16                  # LoRA alpha
)

model = get_peft_model(model, peft_config)
```

The `LoraConfig` object defines the parameters for LoRA finetuning:

- `task_type` specifies the task, which in this case is **causal language modeling**,
- `r` is the LoRA adapter rank,
- `lora_alpha` is the scaling factor α.

The function `get_peft_model` wraps the original model and integrates LoRA adapters. How does it decide which matrices to augment? PEFT is designed to detect standard LLM architectures. When finetuning models such as Llama, Gemma, Mistral, or Qwen, it automatically applies LoRA to the appropriate layers. For custom transformers—like the decoder from Chapter 4—you can add the `target_modules` parameter to specify which matrices should use LoRA:

```
peft_config = LoraConfig(
    #same as above
    target_modules=["W_Q","W_K","W_V","W_O"]
)
```

Next, we set up the optimizer as usual:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

In PyTorch, the `requires_grad` attribute controls whether a tensor tracks operations for automatic differentiation. When `requires_grad=True`, PyTorch keeps track of all operations on the tensor, enabling gradient computation during the backward pass. To freeze a model parameter (preventing updates during training), set its `requires_grad` to `False`:

```
import torch.nn as nn

model = nn.Linear(2, 1)  # Linear layer: y = WX + b

print(model.weight.requires_grad)
print(model.bias.requires_grad)

model.bias.requires_grad = False
print(model.bias.requires_grad)
```

Output:

```
True
True
False
```

The PEFT library ensures that only the LoRA adapter parameters have `requires_grad=True`, keeping all other model parameters frozen.

After wrapping the model with `get_peft_model`, the training loop stays the same. For instance, finetuning GPT-2 on an emotion generation task using LoRA with `r=16` and `lora_alpha=32` achieves a test accuracy of 0.9420. This is marginally better than the 0.9415 from full finetuning. Generally, LoRA tends to perform slightly worse than full finetuning. However, the outcome depends on the choice of hyperparameters, dataset size, base model, and task.

The full code for GPT-2 finetuning with LoRA is available in the thelmbook.com/nb/5.5 notebook. You can customize it for your own tasks by modifying the dataset and LoRA settings.

5.6. LLM as a Classifier

When finetuning GPT-2 for emotion prediction, we didn't turn it into a classifier. Instead, it generated the class name as text. While this method works, it's not always optimal for classification tasks. A different approach is to train the model to produce logits for each emotion class.

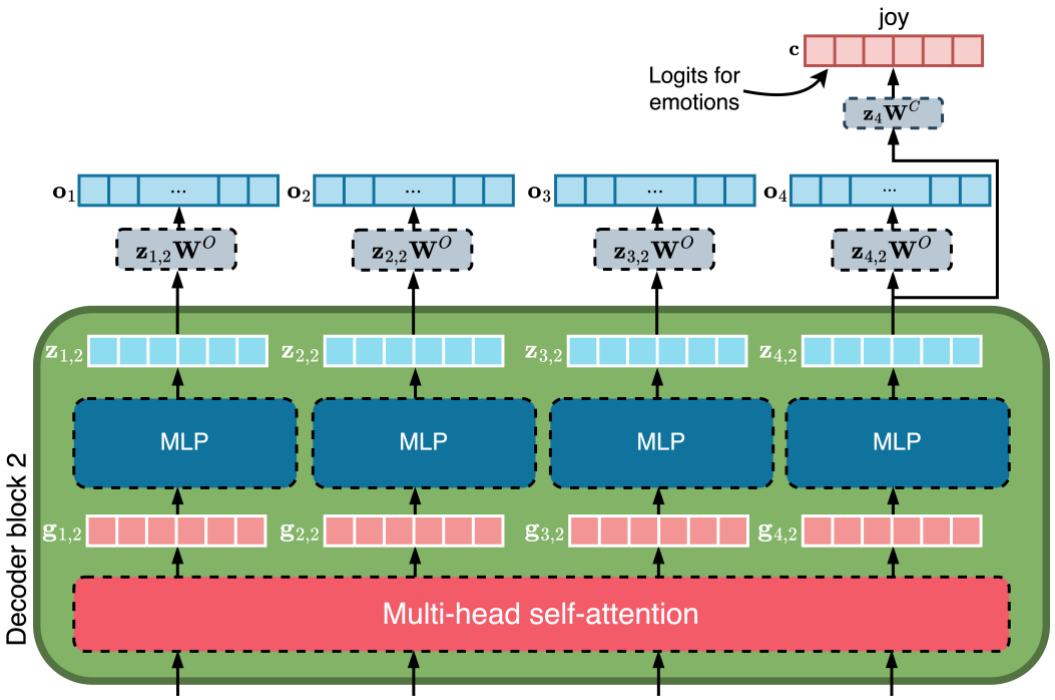
We can attach a **classification head** to a pretrained LLM. This is a fully connected layer with a softmax activation mapping logits to class probabilities.

In `transformers`, there is a class designed to make this easier. Instead of loading the model with `AutoModelForCausalLM`, we use `AutoModelForSequenceClassification`:

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(
    model_path, num_labels=6
)
```

For pretrained autoregressive language models, the class maps the embedding of the final (right-most) non-padding token from the last decoder block to a vector with dimensionality matching the number of classes (6 in this case). The structure of this modification is as follows:



As you can see, once the final decoder block processes the input (the second block in our example), the output embedding $\mathbf{z}_{4,2}$ of the last token is passed through the classification head's weight matrix, $\mathbf{W}^C$. This projection converts the embedding into logits, one per class.

The parameter tensor $\mathbf{W}^C$ is initialized with random values and trained on the labeled emotions dataset. Training relies on **cross-entropy** to measure the loss between the predicted probability distribution and the **one-hot encoded** true class label. This error is backpropagated, updating the weights in both the classification head and the rest of the model. This can be combined with LoRA.

After finetuning with `num_epochs = 8`, `batch_size = 16`, and `learning_rate = 0.00005`, the model reaches a test accuracy of 0.9460. This is slightly better than the 0.9415 accuracy from finetuning the unmodified model to generate class labels as text. The improvement might be more noticeable with a different base model or dataset.

The code for finetuning GPT-2 as an emotion classifier is available on the wiki in the thelmbook.com/nb/5.6 notebook. It can be easily adapted for any text classification task by replacing the data in the file while keeping the same JSON format.

5.7. Prompt Engineering

Chat language models, or **chat LMs**, are language models finetuned on dialogue examples. This finetuning resembles instruction finetuning but uses multi-turn conversation inputs, such as those in the ChatML format, with the targets being the assistant's responses.

Despite its simplicity, the conversational interface allows solving various practical problems. This section explores best practices for using chat LMs to address such problems known as **prompt engineering** techniques.

5.7.1. Features of a Good Prompt

To get the best results from a chat LM, you need a well-crafted prompt. The key components of a strong prompt include:

1. **Situation**: Describe why you're asking for help.
2. **Role**: Define the expert persona the model should emulate.
3. **Task**: Give clear, specific instructions about what the model must do.
4. **Output format**: Explain how you expect the response to be structured, such as bullet points, JSON, or code.
5. **Constraints**: Mention any limitations, preferences, or requirements.
6. **Quality criteria**: Define what makes a response satisfactory.
7. **Examples**: Provide few-shot examples of inputs with expected outputs.
8. **Call to action**: Restate the task simply and ask the model to perform it.

Putting input-output examples in the prompt is called **few-shot prompting** or **in-context learning**. These examples include both positive cases showing desired outputs and negative ones demonstrating incorrect responses. Adding explanations that connect incorrect responses to specific constraints helps model understand why they are wrong.

Here's an example of a prompt that includes some of the above elements:

```
Situation: I'm creating a system to analyze insurance claims. It processes ad
juster reports to extract key details for display in a SaaS platform.

Your role: Act as a seasoned insurance claims analyst familiar with industry-
standard classifications.

Task: Identify the type of incident, the primary cause, and the significant d
amages described in the report.

Output format: Return a JSON object with this structure:
{
    "type": "string",      // Incident type
    "cause": "string",     // Primary cause
    "damage": ["string"]   // Major damages
}

<examples>
    <example>
        <input>
        Observed two-vehicle accident at an intersection. Insured's car was h
it after the other driver ran a red light. Witnesses confirm. The vehicle has
severe front-end damage, airbags deployed, and was towed from the scene.
        </input>
```

```
        <output>
        {
            "type": "collision",
            "cause": "failure to stop at signal",
            "damage": ["front-end damage", "airbag deployment"]
        }
        </output>
    </example>
    <example>
        ...
    </example>
</examples>

Call to action: Extract the details from this report:

"Arrived at the scene of a fire at a residential building. Extensive damage t
o the kitchen and smoke damage throughout. Fire caused by unattended cooking.
Neighbors evacuated; no injuries reported."
```

Section names such as “Situation,” “Your role,” or “Task” are optional.

When working on a prompt, keep in mind that the attention mechanism in LLMs has limitations. It might concentrate on certain parts of a prompt while overlooking others. A good prompt strikes a balance between detail and brevity. Excessive detail can overwhelm the model, while insufficient detail risks leaving gaps that the model may fill with incorrect assumptions.

> I used XML tags for few-shot examples because they clearly define example boundaries and are familiar to LLMs from pretraining on structured data. Furthermore, chat LM models are often finetuned using conversational examples with XML structures. Using XML isn’t mandatory though, but could be helpful.

5.7.2. Followup Actions

The first solution from a model is often imperfect. User analysis and follow-up are key to getting the most out of a chat LM. Common follow-up actions include:

1. Asking the LLM whether its solution contains errors or can be simplified without breaking the constraints.
2. Copying the solution and starting a new conversation from scratch with the same LLM. In this new conversation, the user can ask the model to validate the solution as if it were “provided by an expert,” without revealing it was generated by the same model.
3. Using a different LLM to review or enhance the solution.
4. For code outputs, running the code in the execution environment, analyzing the results, and giving feedback to the model. If code fails, the full error message and stack traceback can be shared with the model.

When working with the same chat LM for follow-ups, especially in tasks like coding or handling complex structured outputs, it's generally a good idea to start fresh after three-five exchanges. This recommendation comes from two key observations:

1. Chat LMs are typically finetuned using examples of short conversations. Creating long, high-quality conversations for finetuning is both difficult and costly, so the training data often lacks examples of long interactions focused on problem solving. As a result, the model performs better with shorter exchanges.
2. Long contexts can cause errors to accumulate. In the self-attention mechanism, the softmax is applied over many positions to compute weights for combining value vectors. As the context length increases, inaccuracies build up, and the model's "focus" may shift to irrelevant details or earlier mistakes.

When starting fresh, it's important to update the initial prompt with key details from earlier follow-ups. This helps the model avoid repeating previous mistakes. By consolidating the relevant information into a clear, concise starting point, you ensure the model has the context it needs without relying on the long and noisy history of the prior conversation.

5.7.3. Code Generation

One valuable use of chat LMs is generating code. The user describes the desired code, and the model tries to generate it. As we know, modern LLMs are pretrained on vast collections of open-source code across many programming languages. This pretraining allows them to learn syntax and many standard or widely used libraries. Seeing the same algorithms implemented in different languages also enables LLMs to form shared internal representations (like synonyms in **word2vec**), making them generally indifferent to the programming language when reading or creating code.

Moreover, much of this code includes comments and annotations, which help the model understand the code's purpose—what it is designed to achieve. Sources like StackOverflow and similar forums add further value by providing examples of problems paired with their solutions. The exposure to such data gave LLMs an ability to respond with relevant code. Supervised finetuning improved their skill in interpreting user requests and turning them into code.

As a result, LLMs can generate code in nearly any language. For high-quality results, users must specify in detail what code should do. For example, providing a detailed docstring like this:

```
Write Python code that implements a method with the following specifications:

def find_target_sum(numbers: list[int], target: int) -> tuple:
    """Find pairs of indices in a list whose values sum to a target.

    Args:
        numbers: List of integers to search through. Can be empty.
        target: Integer sum to find.

    Returns:
        Tuple of two distinct indices whose values sum to target,
        or None if no solution exists.

    Examples:
        >>> find_target_sum([2, 7, 11, 15], 9)
```

```
        (0, 1)
        >>> find_target_sum([3, 3], 6)
        (0, 1)
        >>> find_target_sum([1], 5)
        None
        >>> find_target_sum([], 0)
        None

    Requirements:
        - Time complexity: O(n)
        - Space complexity: O(n)
        - Each index can only be used once
        - If multiple solutions exist, return any valid solution
        - All numbers and target can be any valid integer
        - Return None if no solution exists
    """
```

Providing a highly detailed docstring can sometimes feel as time-consuming as coding the function itself. A less detailed description might seem more practical, but this increases the likelihood of the generated code not fully meeting user needs. In such cases, users can review the output and refine their instructions with additional requests or constraints.

> By the way, the book's official website, thelmbook.com, was created entirely through collaboration with an LLM. While it wasn't generated perfectly on the first try, through iterative feedback, multiple conversation restarts, and switching between different chat LLMs when needed, I refined every element you see—from the graphics to the animations—until they met my vision.

Language models can generate functions, classes, or even entire applications. However, the chance of success decreases as the level of abstraction increases. If the problem resembles model's training data, the model performs well with minimal input. However, for novel or unique business or engineering problems, detailed instructions are crucial for good results.

> If you decide to use a brief prompt to save time, ask the model to pose clarifying questions. You can also request it to describe the code it plans to generate first. This allows you to adjust or add details to the instructions before code is created.

5.7.4. Documentation Synchronization

A common challenge in software development is keeping documentation synchronized with code changes. As codebases evolve, documentation often becomes outdated, leading to confusion and reduced maintainability. LLMs offer an automated solution to this problem through integration with **version control systems**.

The process involves creating a documentation synchronization pipeline that leverages the LLM's ability to understand both code and natural language. When developers stage changes for commit, the pipeline:

1. Uses an LLM to analyze the staged differences and identify affected documentation files in the project's documentation directory. The model examines code changes and determines which documentation files might need updates.
2. Both the existing documentation content and staged code changes are then passed to another LLM call. This second step generates updated documentation that reflects the code modifications while maintaining the existing documentation's style and structure.
3. Places the updated documentation in the staging area alongside code changes. This allows developers to review both code and documentation updates together before committing, ensuring accuracy and maintaining a single source of truth.

This approach treats documentation as a first-class citizen in the development process, ensuring it evolves alongside the code.

> While LLMs can help maintain documentation alignment, they should not operate autonomously. Human review remains crucial to verify the accuracy of generated documentation updates and ensure they align with the team's communication standards.

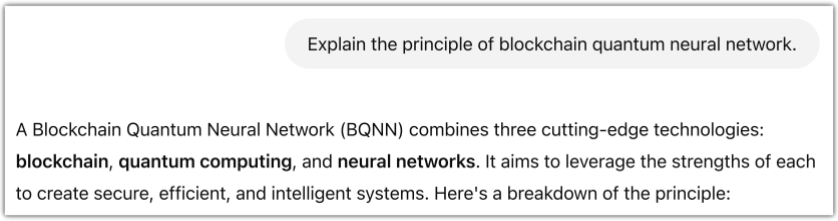
This pipeline is especially useful for keeping API documentation, architectural descriptions, and implementation guides up to date. However, like other LLM-based systems, it must include safeguards against hallucinations. We discuss this next.

5.8. Hallucinations

A major challenge with LLMs is their tendency to produce content that seems plausible but is factually incorrect. These inaccuracies, called **hallucinations**, create problems for using LLMs in production systems where reliability and accuracy are required.

5.8.1. Reasons for Hallucinations

Hallucinations in LLMs are by design. As we know, these models are optimized to predict the next token that fits the context, not to ensure factual accuracy. During pretraining, they learn to generate coherent text by following language patterns. However, no training dataset can cover every fact. When the model faces knowledge gaps, it tries to fill them with plausible content based on patterns it recognizes. This often results in fabricated details. Here's an example of hallucinated information from a widely used chat LM:



> As you can imagine, "Blockchain Quantum Neural Network (BQNN)" is not a real concept. The LLM's two-page explanation, including detailed descriptions of how it works, is entirely fabricated.

Low quality of training data also contributes to hallucinations. During pretraining on large volumes of internet text, models are exposed to both accurate and inaccurate information. They learn these inaccuracies but lack the ability to differentiate between truth and falsehood.

Finally, LLMs generate text one token at a time. This approach means that errors in earlier tokens can cascade, leading to increasingly incoherent outputs.

5.8.2. Preventing Hallucinations

Hallucinations cannot be completely avoided, but they can be minimized. A practical way to reduce hallucinations is by grounding the model's responses in verified information. This is done by including relevant factual context directly in the prompt. For instance, rather than posing an open-ended question, we can provide specific documents or data for the model to reference and instruct the model to only answer based on the provided documents.

This method, called **retrieval-augmented generation** (RAG), anchors the model's output to verifiable facts. The model still generates text but does so—most of the time—within the limits of the provided context, which significantly reduces hallucinations.

Here's how RAG works: a user submits a query, and the system searches a knowledge base—like a document repository or database—for relevant information. It uses keyword matching and embedding-based search, where the query is converted into an embedding vector. Documents with similar embeddings are retrieved using **cosine similarity**. To handle long documents, they are split into smaller chunks before embedding.

The retrieved content is added to the prompt alongside the user's question. This approach merges the strengths of traditional information retrieval with the language generation capabilities of LLMs. For example, if a user asks about a company's latest quarterly results, the RAG system would first retrieve the most recent financial reports and use them to produce the response, avoiding reliance on potentially outdated training data.

Another way to reduce hallucinations is by finetuning the model on reliable, domain-specific knowledge using unlabeled documents. For instance, a question-answering system for law firms could be finetuned on legal documents, case law, and statutes to improve accuracy within the legal domain. This approach is often referred to as **domain-specific pretraining**.

For critical applications, implementing a multi-step verification workflow can provide additional protection against hallucinations. This might involve using multiple models with different architectures or training data to cross-validate responses and having domain experts review generated content before it's used in production.

However, it's important to recognize that hallucinations cannot be completely eliminated with current LLM technology. While we can implement various safeguards and detection mechanisms, the most robust approach is to design systems that account for this limitation.

For instance, in a customer service application, an LLM could draft responses, but human review would be necessary before sending messages containing specific product details or policy

information. Similarly, in a code generation system, the model might generate code, but automated tests and human review should always occur before deployment.

> The potential for hallucinations was notably demonstrated when Air Canada's customer service chatbot provided incorrect information about bereavement travel rates to a passenger. The chatbot falsely claimed that customers could book full-price tickets and later apply for reduced fares, contradicting the airline's actual policy. When the passenger tried to claim the fare reduction, Air Canada's denial led to a small claims court case, resulting in an $812 CAD (near $565 USD) compensation order. This case highlights the tangible business consequences of AI inaccuracies, including financial losses, customer frustration, and reputational damage.

Success with LLMs lies in recognizing that hallucinations are an inherent limitation of the technology. However, this issue can be managed through thoughtful system design, safeguards, and a clear understanding of when and where these models should be applied.

5.9. LLMs, Copyright, and Ethics

The widespread deployment of LLMs has introduced novel challenges in copyright law, particularly regarding training data usage and the legal status of AI-generated content. These issues affect both the companies developing LLMs and the businesses building applications with them.

5.9.1. Training Data

The first major copyright consideration involves training data. LLMs are trained on large text datasets that include copyrighted material such as books, articles, and software code. While some claim that this might qualify as fair use,[9] this has not been tested in court. The issue is further complicated by the models' capacity to output protected content. This legal uncertainty has already sparked high-profile lawsuits from authors and publishers against AI companies, posing risks for businesses using LLM applications.

> Meta's decision to withhold its multimodal Llama model from the European Union in July 2024 exemplifies the growing tension between AI development and regulatory compliance. Citing concerns over the region's "unpredictable" regulatory environment, particularly regarding the use of copyrighted and personal data for training, Meta joined other tech giants like Apple in limiting AI deployments in European markets. This restriction highlights the challenges companies face in balancing innovation with regional regulations.

When selecting models for commercial use, companies should review the training documentation and license terms. Models trained primarily on **public domain** or properly licensed materials involve

[9] Fair use is a U.S. legal doctrine. Other regions handle copyright exceptions differently. The EU relies on "fair dealing" and specific statutory exceptions, Japan has distinct copyright limitations, and other countries apply unique rules for permitted uses. This variation complicates global LLM deployment, as training data allowed under U.S. fair use might violate copyright laws elsewhere.

lower legal risks. However, the massive datasets required for effective LLMs make it nearly impossible to avoid copyrighted material entirely. Businesses need to understand these risks and factor them into their development strategies.

Beyond legal issues, training LLMs on copyrighted material raises ethical concerns. Even when legally permissible, using copyrighted works without consent may appear exploitative, especially if the model outputs compete with the creators' work. Transparency about training data sources and proactive engagement with creators can help address these concerns. Ethical practices should also involve compensating creators whose contributions significantly improve the model, fostering a more equitable system.

5.9.2. Generated Content

The copyright status of content generated by LLMs presents challenges that traditional copyright law cannot easily resolve. Copyright law is built around the assumption of human authorship, leaving it unclear whether AI-generated works qualify for protection or who the rightful owner might be. Another issue is that LLMs can sometimes reproduce portions of their training data verbatim, including copyrighted material. This ability to generate exact reproductions—beyond learning abstract patterns—raises serious legal questions.

Some businesses address these challenges by using LLMs as assistive tools rather than independent creators. For example, a marketing team might use an LLM to draft text, leaving human writers to edit and finalize it. This approach maintains clearer copyright ownership while leveraging AI's efficiency. Similarly, software developers use LLMs to generate code snippets, which they review and integrate into larger systems. By 2024, this practice had grown significantly—at Google, over 25% of all code was generated by LLMs and then refined by developers.

To minimize copyright risks in LLM applications, companies often implement technical safeguards.

One method involves comparing model outputs against a database of copyrighted materials to detect verbatim copies. For example, a company may maintain a repository of copyrighted texts and employ similarity detection methods—such as **cosine similarity** or **edit distance**—to flag outputs that surpass a defined similarity threshold.

However, these methods are not foolproof. Paraphrased content can make the output formally distinct while remaining substantively similar, which automated systems may fail to detect. To handle this, businesses often supplement these tools with human review to ensure compliance.

5.9.3. Open-Weight Models

The copyright status of model weights poses legal questions separate from those concerning training data or generated outputs. Model weights encode patterns learned during training and could be viewed as derivative works of the training data. This leads to the question: does sharing weights amount to indirectly redistributing the original copyrighted materials, even in their transformed form? Some argue that weights are an abstract transformation and constitute new intellectual property. Others contend that if weights can reproduce fragments of the training data, they inherently include copyrighted content and should be treated similarly under copyright law.

This debate carries serious implications for open-source AI development. If model weights are classified as derivative works, sharing and distributing models trained on copyrighted data could become legally restricted, even if the training process qualifies as fair use. As a result, some organizations have shifted to training models solely on public domain or explicitly licensed content.

However, this strategy often limits the effectiveness of the models, as the smaller, restricted datasets typically lead to reduced performance.

As laws around LLMs evolve, businesses must stay flexible. They may need to adjust workflows as courts define legal boundaries or revise policies as AI-specific legislation appears. Consulting intellectual property lawyers with AI expertise can help manage these risks.

5.9.4. Broader Ethical Considerations

Beyond copyright concerns, LLMs raise significant ethical challenges that affect society at large. One fundamental issue is **explainability**. While LLMs can articulate reasoning for their outputs and provide detailed explanations when asked, this verbal explanation capability differs from true algorithmic transparency. The model's explanations are post-hoc rationalizations—generated text that sounds plausible but may not reflect the actual computational process that produced the original output. This creates a unique challenge where the model appears transparent while its underlying decision-making process remains opaque. This limitation becomes particularly significant in high-stakes applications like healthcare or legal services.

The question of **bias** presents another challenge. LLMs trained on internet data inevitably absorb societal biases present in their training data. These models can perpetuate or amplify discriminatory patterns in areas such as gender, race, age, and cultural background. For instance, they might generate different responses to equivalent prompts that only differ in demographic details, or produce content that reinforces stereotypes.

Organizations deploying LLMs must implement structured evaluation protocols, including automated bias detection across demographic groups and audits using standardized test sets. This should include deploying concrete safeguards like toxic language filters, mandatory human review for high-stakes decisions, and clear user notifications about AI involvement.