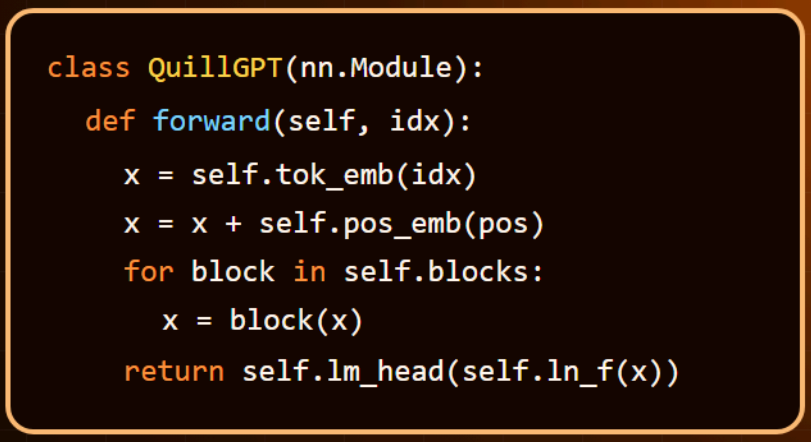
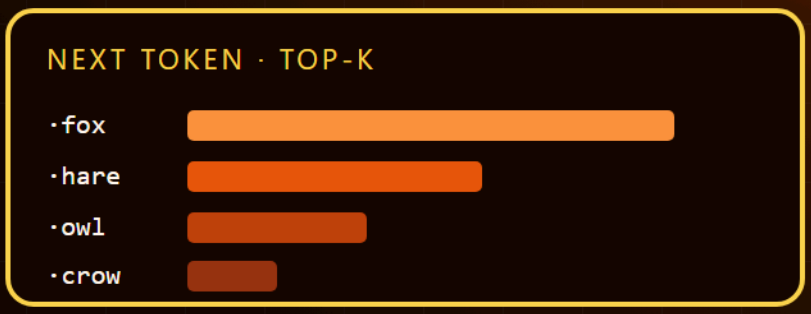
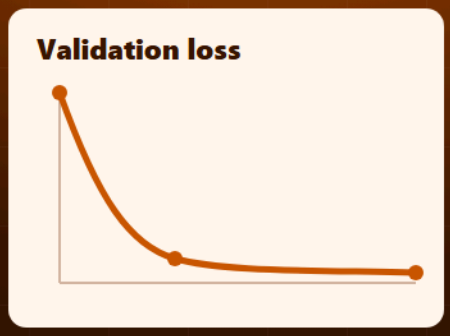
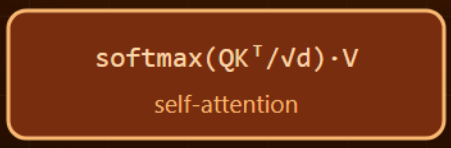












BUILD IT YOURSELF · PYTORCH

Language Models from the Ground Up

Build, train, and fine-tune a GPT-style LLM in PyTorch — on an ordinary CPU

Tokenizers · BPE · Embeddings · Self-Attention · Transformers
Training · Sampling · GPT-2 Weights · Instruction Tuning · LoRA · KV Cache

DATA · ATTENTION · TRAINING · FINE-TUNING · EFFICIENCY

THENEXBOOKS

Language Models from the Ground Up

Build, Train, and Fine-Tune a GPT-Style LLM in PyTorch
Tokenizers, attention, training, sampling, GPT-2 weights, instruction tuning, and efficiency — written line by line

First Edition

THENEXBOOKS

Language Models from the Ground Up
Build, Train, and Fine-Tune a GPT-Style LLM in PyTorch

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. The code examples may be used freely in your own projects. The Quillbrook Fables corpus is generated by code printed in this book; the generated text is dedicated to the public domain (CC0 1.0).

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python, PyTorch, Hugging Face libraries, and other tools change over time; function names, defaults, and outputs can differ between versions. Training results depend on hardware, library versions, and random seeds, so your numbers may differ slightly from the ones printed here. Language models can produce incorrect, biased, or harmful text; review model output before you rely on it, and check the license of any pretrained model before using it in a product.

Python is a trademark of the Python Software Foundation. PyTorch is a trademark of The Linux Foundation. Hugging Face is a trademark of Hugging Face, Inc. GPT-2 was released by OpenAI; OpenAI is a trademark of OpenAI. Windows is a trademark of the Microsoft group of companies. NVIDIA and CUDA are trademarks of NVIDIA Corporation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these companies or projects.

Quillbrook Press, the small publisher used as the running example in this book, is fictional, as are its staff, its QuillGPT models, and all sample data. Any resemblance to real organizations or people is coincidental.

Preface

Large language models went from research papers to everyday tools in a remarkably short time. Most developers now use them through a chat window or an API, and that is often the right choice. But using a model and understanding one are different things. When a model repeats itself, invents a fact, runs out of context, or costs more than expected, the developers who know what happens inside — tokens, embeddings, attention, logits, sampling — are the ones who can explain why and do something about it.

The best way to understand a language model is to build one. That is what this book does. Starting from an empty folder, we write every important piece of a GPT-style model ourselves in PyTorch: a tokenizer, embeddings, self-attention, the transformer block, the training loop, and the sampling code that turns probabilities into text. We train a small model on a laptop-class CPU, measure it honestly, load the real released GPT-2 weights into our own code, and then fine-tune, instruction-tune, speed up, and shrink what we built.

The models in this book are deliberately small. Everything runs on an ordinary computer in minutes, without a GPU and without a cloud account. The ideas, however, are the same ones used in the largest models, and every chapter explains what changes when you scale up — so the code is small, but what you learn is not.

Who This Book Is For

- **Python developers** who use language models in their work and want to understand how they actually function.
- **Students and self-learners** who have seen the transformer diagram many times and want to turn it into code they understand line by line.
- **Data scientists and machine learning practitioners** who know classic models and want a hands-on path into modern language modeling.
- **Technical leads and architects** who make decisions about language models and want first-hand intuition for training, fine-tuning, cost, and efficiency.

You should be comfortable with basic Python: functions, classes, lists, and dictionaries. You do not need prior PyTorch or deep learning experience — Chapter 3 teaches the parts of PyTorch we use — and the mathematics is explained in plain words with small, concrete numbers.

What You Will Learn

- What a language model is, how next-token prediction works, and how a simple bigram model already captures the core idea.
- How text becomes numbers: building a corpus, character-level tokenization, byte pair encoding written from scratch, and how it compares with a production tokenizer.

- How attention works, from a single dot product to causal multi-head self-attention, and how transformer blocks stack into a complete GPT-style model.
- How to train a model: cross-entropy loss, perplexity, optimizers, learning-rate schedules, gradient clipping, checkpoints, and spotting overfitting.
- How to generate text with greedy decoding, temperature, top-k, and top-p sampling, and how to evaluate the results honestly.
- How to load the released GPT-2 small weights into your own implementation and confirm that it matches a reference implementation.
- How to fine-tune for classification, instruction-tune a small model, and write LoRA from scratch.
- How to make generation faster with a KV cache, make models smaller with quantization, and what changes at real scale — data, compute, distributed training, and modern architecture choices.

How This Book Is Organized

Part I — Foundations (Chapters 1–4) explains what a language model is, sets up a CPU-friendly workshop, teaches the PyTorch you need, and builds a first bigram model. **Part II — Text into Numbers** (Chapters 5–9) builds the Quillbrook Fables corpus, character and byte pair encoding tokenizers, and the data pipeline. **Part III — Attention and the Transformer** (Chapters 10–14) builds embeddings, self-attention, multi-head attention, the transformer block, and the complete QuillGPT model. **Part IV — Training QuillGPT** (Chapters 15–19) covers loss and perplexity, the training loop, training techniques, text generation, and evaluation. **Part V — Pretrained Weights and Fine-Tuning** (Chapters 20–24) loads GPT-2 small, fine-tunes for classification, instruction-tunes, and implements LoRA. **Part VI — Efficiency and Real Scale** (Chapters 25–28) adds a KV cache, quantization, a tour of what changes at real scale, and a shipping checklist. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with **Try It Yourself** exercises and a **Key Takeaways** box for quick review.

The Running Example: Quillbrook Press and QuillGPT

Throughout the book we work with **Quillbrook Press**, a fictional small publisher that reprints old public-domain fables and stories. Quillbrook wants a language model of its own, **QuillGPT**, that writes in the style of its catalog, can be studied and inspected by its small technical team, and runs on the computers they already have. To keep every example reproducible and free of licensing questions, we generate the training text ourselves: the **Quillbrook Fables** corpus is two thousand short fables produced by a seeded Python script printed in Chapter 5, and it is dedicated to the public domain. We train **QuillGPT-mini**, a model of under one million parameters, then load GPT-2 small into the same code, and finally build **QuillGPT-Instruct**, which answers simple questions about fables. Chapter 5 also shows how to swap in real public-domain books when you are ready for richer text.

Versions and Conventions

The code uses **Python 3.12** and recent versions of **PyTorch**, plus **tiktoken**, **transformers**, and **safetensors** for the chapters that work with GPT-2. Every listing that trains or generates was run on a CPU with two threads (`torch.set_num_threads(2)`), and each run finishes in a few minutes. Where a technique needs a GPU — mixed precision, large batches, 4-bit loading, or multi-GPU training — we say so and describe what to expect rather than printing invented results. Training numbers such as loss and timing depend on your hardware and library versions; small differences from the printed values are normal.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause wrong results, wasted compute, or security problems.

Code appears in `monospace`; long lines are wrapped to fit the page. A small caption above each listing names its file, such as `quillgpt/model.py`. The complete reference package is called `quillgpt`, and Appendix B lists its modules, classes, and functions. Tensor shapes are written in comments as `(B, T, D)` — batch size, sequence length, and model width.

Contents

Preface

PART I — FOUNDATIONS

1 What a Language Model Is: Next-Token Prediction and the Road to QuillGPT

2 Setting Up the Workshop: Python, PyTorch on the CPU, and the Project Layout

3 PyTorch Essentials for Language Models: Tensors, Autograd, and Modules

4 A First Language Model: Bigram Counts and a Neural Bigram

PART II — TEXT INTO NUMBERS

5 The Quillbrook Fables Corpus: Collecting, Generating, and Cleaning Text

6 Character-Level Tokenization: Vocabularies, Encoding, and Decoding

7 Byte Pair Encoding from Scratch

8 Production Tokenizers: tiktoken, Special Tokens, and Trade-Offs

9 Training Windows: Datasets, DataLoaders, and Batches

PART III — ATTENTION AND THE TRANSFORMER

10 Embeddings: Turning Token IDs into Vectors with Position

11 Self-Attention from Scratch: Queries, Keys, and Values

12 Causal Masking and Multi-Head Attention

13 The Transformer Block: Layer Normalization, Feed-Forward Networks, and Residuals

14 The Complete QuillGPT Model: Assembling, Counting, and Running It

PART IV — TRAINING QUILLGPT

15 Loss, Cross-Entropy, and Perplexity

16 The Training Loop: Optimizers, Steps, and Checkpoints

17 Training Well: Learning-Rate Schedules, Clipping, Overfitting, and Regularization

18 Generating Text: Greedy Decoding, Temperature, Top-k, and Top-p

19 Evaluating a Language Model: Validation Loss, Perplexity, and Reading Samples

PART V — PRETRAINED WEIGHTS AND FINE-TUNING

20 Inside GPT-2: Architecture, Configuration, Checkpoints, and Licenses

21 Loading GPT-2 Weights into QuillGPT

22 Fine-Tuning for Classification: Sorting Fables by Lesson

23 Instruction Tuning: Building QuillGPT-Instruct

24 LoRA from Scratch: Parameter-Efficient Fine-Tuning

PART VI — EFFICIENCY AND REAL SCALE

25 Faster Generation with a KV Cache

26 Quantization and Efficient Inference

27 What Changes at Real Scale: Data, Compute, and Modern Architectures

28 Shipping QuillGPT: Packaging, a Small API, Safety, and Next Steps

APPENDICES

A Language Model Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What a language model really does, a CPU-friendly PyTorch setup, the tensor and autograd skills the rest of the book relies on, and a first working model — a bigram model built from counts and then from a neural network.

1 · What a Language Model Is

2 · Setting Up the Workshop

3 · PyTorch Essentials for Language Models

4 · A First Language Model: Bigrams

CHAPTER 1 · PART I — FOUNDATIONS

What a Language Model Is: Next-Token Prediction and the Road to QuillGPT

Every chatbot, writing assistant, and code helper you have used rests on one surprisingly small idea: look at some text and guess what comes next. In this chapter we take that idea apart with plain Python, meet the small publisher whose model we will build, and lay out the road from a dozen-line word counter to QuillGPT, a working GPT-style language model that runs on an ordinary laptop.

A Machine That Guesses the Next Word

Open the keyboard on your phone and type "See you". Above the keys, three suggestions appear: perhaps "soon", "tomorrow", and "later". Your phone has never read your mind. It has simply learned, from a great deal of text, which words tend to follow "See you", and it shows you the most likely ones. Pick one, and it immediately guesses again for the word after that.

That is a language model in miniature. A **language model** is a program that, given some text, assigns a probability to every possible continuation. Show it "The fox and the" and it does not return one answer; it returns a list of candidates with a number attached to each: perhaps "heron" with probability 0.08, "crow" with 0.07, "wolf" with 0.07, and thousands of unlikely options with tiny probabilities that together add up to the rest. The probabilities always add up to exactly 1, because *something* has to come next.

Once you have such a list, generating text is easy. Pick one candidate, append it to the text, and ask the model again. Repeat a few hundred times and you have a paragraph. Repeat with a very good model and the paragraph can be fluent, on topic, and useful. The large systems you read about in the news work this way too: underneath the chat window there is a model that predicts one small piece of text at a time, over and over.

This framing is called **next-token prediction**, and it is the single most important idea in this book. When people say a model "writes", "answers", or "reasons", what is happening mechanically is a loop of next-token predictions. Everything else we build, from tokenizers to attention to fine-tuning, exists to make that one prediction better.

> **NOTE**
>
> The technical name for this setup is *causal language modeling*: the model may look only at the text that came before a position (the cause), never at the text after it. Chapter 12 builds the mechanism, a causal mask, that enforces this rule inside the model.

Meet Quillbrook Press

A book about building a model needs a reason to build one, so we will follow a small, fictional company through the whole journey. **Quillbrook Press** is a little publisher that reprints old public-domain fables and stories: the kind of tale where a tortoise outlasts a hare and a final line spells out the moral. Its catalog is charming, its staff is tiny, and its budget is smaller still.

Quillbrook's managing editor, **Marisol Achebe**, has been reading about language models and keeps asking practical questions. Can a model write a new fable in the Quillbrook house style? Can it tell a reader the moral of a story? Could the press own such a model outright, inspect it, and run it without sending manuscripts to an outside service?

The person who has to answer those questions is **Jonah Penhallow**, Quillbrook's one-person tech team. Jonah is a capable Python developer, but he has never trained a neural network, and the only computer he has is an ordinary office PC with no graphics card worth mentioning. If a plan needs a rack of expensive accelerators, it is not a plan for Quillbrook.

Their goal is a model family called **QuillGPT**. The first member, **QuillGPT-mini**, is a small model that Jonah will train from nothing on Quillbrook's own text. Later, **QuillGPT-Instruct** will be tuned to follow short instructions such as "Give the moral of this fable." Along the way Jonah will also load the publicly released GPT-2 small weights into the very same code, to see how a much larger model behaves. You will play Jonah's role: every line he writes, you will write and run too.

Throughout the book, the training text is the **Quillbrook Fables** corpus: 2,000 short fables produced by a small Python script with a fixed random seed. Chapter 5 explains in detail why we generate our own text instead of downloading it. The short version is that it is tiny, perfectly reproducible on every machine, and free of any licensing questions, because Quillbrook dedicates it to the public domain. Here is the very first fable the script produces, exactly as the model will see it:

Fable #1 of the Quillbrook Fables corpus

```
The Sparrow and the Wolf

Long ago, near the hollow oak, there was a timid sparrow. The sparrow found a purse of
honey on the path to the hollow oak. The wolf whispered, "Keep it. No one will ever
know." But the sparrow carried the purse to the owl who kept the lost things of the
wood. The owner, an old beaver, was so grateful that word spread quickly. From that
summer on, every creature trusted the sparrow with its stores.

Moral: Honesty earns a trust that cleverness cannot buy.
```

(We wrapped the long lines to fit the page; in the corpus each paragraph is one line.) Every fable follows this shape: a title naming two animals, a short body, and a moral chosen from a list of eight. That regularity is a feature for learning. You will be able to see, by reading its output, exactly what the model has and has not picked up.

Tokens, Not Words

So far we have talked about predicting the next *word*. Real language models predict the next **token**, and the difference matters. A token is a chunk of text from a fixed list called the **vocabulary**. Depending on the tokenizer, a token might be a single character, a whole word, or a piece of a word such as "hog" in "hedgehog". Each token in the vocabulary has an integer ID, and the model works entirely with those IDs.

Why not just use whole words? Because the list of possible words is effectively endless. New names, typos, numbers, and rare words appear all the time, and a model with a fixed word list would be stuck the first time it saw one. Why not single characters, then? Because character sequences are long. A fable of 500 characters becomes 500 prediction steps, and the model has to work much harder to connect a character at the start of a sentence with one at the end. Modern tokenizers split the difference: common words become one token, rare words are spelled out from a few smaller pieces, and any text at all can still be represented.

To make this concrete, here is how two tokenizers you will meet later split the same sentence. The first is the small tokenizer we train on the Quillbrook Fables in Chapter 7; the second is the tokenizer that GPT-2 uses, which we load through the `tiktoken` library in Chapter 8. The vertical bars mark token boundaries:

Output (from Chapter 8)

```
ours (1,024 tokens):    The| hedgehog| gathered| acorns| by| the| old| mill|.
GPT-2 (50,257 tokens):  The| hedge|hog| gathered| ac|orns| by| the| old| mill|.
```

Our tokenizer, trained on fables full of hedgehogs and acorns, keeps those words whole and needs 9 tokens. GPT-2's tokenizer was trained on general web text, where "hedgehog" is rare, so it needs 11. Notice also that most tokens carry their leading space: " gathered" is a single token, space included. That small detail lets the model rebuild spacing perfectly when it turns IDs back into text.

Our tokenizer turns the title "The Fox and the Heron" into the five IDs `[267, 660, 268, 259, 940]`, one per word. From the model's point of view, the title *is* that list of five integers. Its whole job is to look at a list like that and produce a probability for each of the 1,024 IDs that could come next.

> **TIP**
>
> When you read that a model has a "context window of 1,024", or that a service charges "per thousand tokens", the unit is always tokens, not words or characters. A loose rule of thumb for English is that a token is about three quarters of a word, but measure on your own text when it matters. On the Quillbrook validation text, GPT-2's tokenizer averages about 4.03 characters per token.

From Scores to Probabilities: Logits and Softmax

A language model does not produce probabilities directly. Its last layer produces one raw score per vocabulary entry. These scores are called **logits**. A logit can be any real number: positive, negative, large, or small. Higher means "more likely", but the numbers do not add up to anything in particular, so we need a way to turn them into a proper probability distribution.

That conversion is the **softmax** function. It does two things. First it applies the exponential function to each score, which makes every value positive and stretches the gaps between large and small scores. Then it divides each result by the total, so the values sum to 1. In plain words: softmax(score) = exp(score) divided by the sum of exp over all scores.

Let us try it on a made-up vocabulary of four tokens. Suppose that after the text "The owl thanked the", a model produced these logits: "fox" 2.0, "owl" 1.5, "purse" 0.5, and "moon" -1.0. Here is softmax in pure Python, no libraries beyond the standard `math` module:

softmax_demo.py

```
import math

logits = {"fox": 2.0, "owl": 1.5, "purse": 0.5, "moon": -1.0}
total = sum(math.exp(v) for v in logits.values())
probs = {k: math.exp(v) / total for k, v in logits.items()}
for k, v in probs.items():
    print(f"{k:6} logit {logits[k]:5.1f} -> prob {v:.3f}")
print("sum:", sum(probs.values()))
```

Output

```
fox    logit   2.0 -> prob 0.532
owl    logit   1.5 -> prob 0.323
purse  logit   0.5 -> prob 0.119
moon   logit  -1.0 -> prob 0.026
sum: 1.0
```

The ordering is preserved, "fox" is still the favorite, but now we have real probabilities. A gap of 0.5 between "fox" and "owl" became a ratio of about 1.6 to 1, because exp(0.5) is about 1.65. Even "moon", with a negative score, keeps a small slice of probability. Softmax never assigns exactly zero to anything, which is why a language model can, now and then, produce a surprising word.

Choosing the Next Token

Once we have probabilities, we still have to pick one token. There are two basic strategies, and you will implement both properly in Chapter 18.

Greedy decoding always takes the most probable token. It is simple and repeatable: the same prompt gives the same text every time. Its weakness is that it is boring and prone to loops. If "the path to" is the most likely continuation of "the path to", a greedy model will happily write "the path to the

path to the path to" until you stop it. We will see QuillGPT-mini do something very close to this.

Sampling draws a token at random according to the probabilities: "fox" 53.2% of the time, "owl" 32.3%, and so on. Sampling produces varied, more natural text, at the cost of occasionally picking something odd. To control that trade-off we use a **temperature**: divide every logit by a number T before the softmax. A low temperature sharpens the distribution toward the favorite; a high one flattens it toward uniform. Here is what that does to our four tokens:

Temperature	fox	owl	purse	moon
0.5 (focused)	0.704	0.259	0.035	0.002
1.0 (as trained)	0.532	0.323	0.119	0.026
2.0 (adventurous)	0.404	0.315	0.191	0.090

At temperature 0.5, "moon" almost disappears. At 2.0, it is picked nearly one time in eleven. Pushing temperature toward zero gives you greedy decoding; pushing it very high gives you noise. Chapter 18 adds two further filters, top-k and top-p, that cut off the long tail of unlikely tokens before sampling.

One Token at a Time

Generation is a loop, and it is worth seeing the loop in its barest form before any machinery hides it. In pseudocode:

```
text = prompt
repeat N times:
    probabilities = model(text)          # one number per vocabulary token
    next_token = choose(probabilities)   # greedy or sampling
    text = text + next_token
return text
```

Three things about this loop are worth noticing now, because they shape the rest of the book.

- **Each step depends on all earlier choices.** A token chosen at step 3 becomes part of the input at step 4. This is why the process is called *autoregressive*: the model feeds on its own output. One unlucky pick early on can send the rest of the text in a strange direction.
- **The model sees a limited window.** Every model has a maximum context length. QuillGPT-mini looks at most 128 tokens back; GPT-2 small looks at 1,024. Text older than the window is simply invisible.
- **Naive generation repeats work.** Each step runs the model over the whole text again, even though only the last token is new. Chapter 25 adds a *KV cache* that remembers earlier work; on our machine it made generation roughly two to three times faster.

A Language Model in a Dozen Lines

Enough description. Let us build the smallest honest language model we can: one that predicts the next word by counting which words followed which in some training text. We will use three short sentences in the style of the Quillbrook Fables, and plain Python only. No PyTorch yet; that arrives in Chapter 2.

next_word.py

```
from collections import Counter, defaultdict

text = ("The fox found a purse of honey. "
        "The fox carried the purse to the owl. "
        "The owl thanked the fox.")
words = text.lower().replace(".", " .").split()
follows = defaultdict(Counter)
for current, nxt in zip(words, words[1:]):
    follows[current][nxt] += 1

for word in ["the", "fox", "purse"]:
    counts = follows[word]
    total = sum(counts.values())
    probs = {w: round(c / total, 2) for w, c in counts.most_common()}
    print(f"after {word!r}: {probs}")
```

Read it top to bottom. We lowercase the text and put a space before each period so that "." becomes a token of its own, then split on whitespace. That is a tokenizer, a crude one. Next, `zip(words, words[1:])` walks over every pair of neighboring words, and for each pair we bump a counter: "after *current*, we saw *nxt*". The `follows` dictionary maps each word to a `Counter` of the words that came after it. Finally we turn counts into probabilities by dividing by the total. Run it with `py -3.12 next_word.py` on Windows or `python3.12 next_word.py` on macOS and Linux:

Output

```
after 'the': {'fox': 0.5, 'owl': 0.33, 'purse': 0.17}
after 'fox': {'found': 0.33, 'carried': 0.33, '.': 0.33}
after 'purse': {'of': 0.5, 'to': 0.5}
```

The word "the" appears six times in our text (counting "The" at the start of sentences, since we lowercased). Three times it is followed by "fox", twice by "owl", and once by "purse", so the model says 0.5, 0.33, and 0.17. That is a complete, working language model. It is a **bigram** model, because it looks at pairs of neighboring tokens and nothing more, and it was "trained" simply by counting.

We can generate with it, too. Add these lines to the end of the file:

next_word.py (continued)

```
import random

word, out = "the", ["the"]          # greedy: always take the top choice
for _ in range(8):
```

```
    word = follows[word].most_common(1)[0][0]
    out.append(word)
print(" ".join(out))

rng = random.Random(42)             # sampling, with a fixed seed
for trial in range(3):
    word, out = "the", ["the"]
    for _ in range(10):
        counts = follows[word]
        word = rng.choices(list(counts), weights=list(counts.values()))[0]
        out.append(word)
        if word == ".":
            break
    print(" ".join(out))
```

Output (new lines)

```
the fox found a purse of honey . the
the purse of honey .
the owl thanked the fox carried the fox carried the fox
the purse to the purse to the owl thanked the fox
```

The greedy line simply replays the first sentence, because every time there is a tie, `most_common` returns the candidate it saw first. The sampled lines are more interesting. "The purse of honey." is a perfectly good new phrase that never appeared in the training text. "The owl thanked the fox carried the fox carried" shows the model's fatal weakness: after "fox" it has no idea that it already used "carried", because it only ever looks one word back. Each step is locally sensible and the whole is nonsense.

> **NOTE**
>
> Because `random.Random(42)` is seeded, you should see exactly these three sampled lines. Seeding every source of randomness is a habit we keep throughout the book, so that your numbers match ours. Chapter 2 explains how seeds work in PyTorch.

Everything that follows in this book is, in a sense, about fixing that weakness. Chapter 4 builds the same bigram idea on characters from the whole Quillbrook corpus and then rebuilds it as a tiny neural network. Part III replaces "look one token back" with **attention**, a mechanism that lets every position consult every earlier position, so that the model can remember that the fox already carried the purse.

How a Model Learns: Surprise as a Number

Counting works for bigrams, but it does not scale to a model that looks 128 tokens back: there are far too many possible contexts to count. Neural language models learn differently. They start with random numbers inside and adjust them, a tiny bit at a time, to make the training text more probable. To do that we need a single number that says how good the predictions are. That number is the **loss**.

For language models the standard loss is the **cross-entropy**, which sounds intimidating but is simple: for each position, take the probability the model gave to the token that *actually* came next, and compute its negative natural logarithm. Then average over all positions. If the model gave the right token probability 1, the loss for that position is -ln(1) = 0: no surprise at all. If it gave probability 0.5, the loss is about 0.69. If it gave 0.01, the loss is about 4.6. Lower loss means less surprise, which means better predictions.

A useful landmark is the loss of a model that knows nothing and spreads probability evenly over all V tokens. Each token gets 1/V, so the loss is -ln(1/V) = ln(V). For our 1,024-token vocabulary that is ln(1024), about 6.93. When we build QuillGPT-mini in Chapter 14 and measure it before training, its loss on the validation text is 6.945, right at that landmark, which tells us the random starting model is indeed clueless. After a few minutes of training on a CPU, the same measurement drops to 0.396.

You will also meet **perplexity**, which is just exp(loss). It reads as "the model is as uncertain as if it were choosing evenly among this many tokens". The untrained model has a perplexity a little above 1,000; the trained QuillGPT-mini has about 1.49. Chapter 15 derives all of this carefully, and Chapter 19 explains why such a low number on our templated fables is less impressive than it sounds.

> **WARNING**
>
> Loss and perplexity numbers are only comparable between models that use the same tokenizer and the same evaluation text. A character-level model and a word-piece model measure surprise per different units, so their perplexities cannot be ranked against each other. We return to this trap in Chapter 19.

Why "Large"?

The models behind today's assistants are called *large* language models, and three things make them large. None of them is magic, and you will meet a small version of each in this book.

- **Parameters.** These are the adjustable numbers inside the model: the weights that training changes. Our QuillGPT-mini has 939,264 of them. GPT-2 small, which OpenAI released in 2019, has about 124 million as counted in the released checkpoint. Production models today are far larger again.
- **Data.** Bigger models need much more text to learn from. Our corpus is under a million characters. Large models are trained on vast collections of web pages, books, and code, which brings its own problems of quality, duplication, licensing, and privacy that Chapters 5 and 27 discuss.
- **Compute.** Training cost grows with both the number of parameters and the amount of data. A commonly used approximation, discussed in Chapter 27, says the arithmetic grows roughly in proportion to parameters times training tokens. That is why large models are trained on clusters of specialized accelerators for long periods, and why we will not be doing that here.

The good news is that the *code* barely changes with size. The QuillGPT class we write in Chapter 14 is built with the same layout as GPT-2. In Chapter 21 we will load the real GPT-2 small weights into that same class, change nothing but the configuration numbers, and get the same outputs as the official implementation to within floating-point noise. Understanding a small model deeply is a direct route to understanding a large one.

Pretraining, Fine-Tuning, and Instruction Tuning

Modern language models are usually built in stages, and the vocabulary for those stages appears everywhere, so let us pin it down.

Pretraining is the first and by far the most expensive stage. The model reads a large body of text and learns to predict the next token, nothing more. No one labels the data; the text itself provides the answers, because the "correct next token" is simply whatever came next. A pretrained model has absorbed grammar, facts, style, and patterns, but it is not an assistant. Ask it a question and it may continue with three more questions, because that is what it saw in its training text. Our QuillGPT-mini is a pretrained model in exactly this sense; its pretraining takes a few minutes.

Fine-tuning continues training a pretrained model on a smaller, focused dataset for a specific purpose. In Chapter 22 we fine-tune QuillGPT-mini to classify fables by their lesson: is this a story about patience, greed, or honesty? The pretrained model already "knows" a lot about fables, so the classifier learns quickly.

Instruction tuning is a particular kind of fine-tuning on pairs of instructions and good responses, such as "Give the moral of this fable" followed by the right moral. It teaches the model to answer rather than merely continue. In Chapter 23 we build QuillGPT-Instruct this way. Large assistants add further stages, such as training on human preferences between two answers and safety tuning; we describe those in words but do not implement them.

What "GPT" Means

GPT stands for *Generative Pre-trained Transformer*. "Generative" means it produces text; "pre-trained" refers to the pretraining stage above; "Transformer" names the neural network architecture.

The Transformer was introduced in the 2017 paper "Attention Is All You Need" by Vaswani and colleagues. The original design had two halves: an *encoder* that reads an input sentence, and a *decoder* that writes an output sentence, which suited tasks like translation. GPT-style models keep only the decoder half. They are **decoder-only** transformers: a stack of identical blocks, each of which lets every token look back at earlier tokens (attention) and then transform what it found (a small feed-forward network). At the top, a final layer turns each position into logits over the vocabulary.

Here is the whole architecture as a sketch. Do not worry about the boxes yet; each one gets its own chapter.

```
token IDs   [267, 660, 268, 259, 940]          "The Fox and the Heron"
     |
token embedding + position embedding          (Chapter 10)
     |
+------------------ transformer block x N ---------------+
|  layer norm -> causal multi-head attention -> add      |  (Chapters 11-13)
|  layer norm -> feed-forward network         -> add     |
+--------------------------------------------------------+
     |
final layer norm -> output layer -> logits (one per vocabulary token)
     |
softmax -> probabilities for the next token   (Chapters 15 and 18)
```

QuillGPT-mini uses 4 of those blocks, each 128 numbers wide. GPT-2 small uses 12 blocks, each 768 wide. Same diagram, different numbers.

The Road to QuillGPT

This book is organized into six parts. Each one ends with something that runs.

Part	Chapters	What you build
I. Foundations	1–4	Your setup, PyTorch basics, and a first bigram model
II. Text into Numbers	5–9	The fables corpus, two tokenizers, and data loaders
III. Attention and the Transformer	10–14	Embeddings, attention, blocks, and the full QuillGPT model
IV. Training QuillGPT	15–19	Loss, the training loop, sampling, and evaluation
V. Pretrained Weights and Fine-Tuning	20–24	GPT-2 weights, a classifier, QuillGPT-Instruct, and LoRA
VI. Efficiency and Real Scale	25–28	A KV cache, quantization, scaling ideas, and a small API

The code lives in a single Python package called `quillgpt`, which you create in Chapter 2 and grow one file at a time: the corpus generator, the tokenizers, the dataset, the model, the training loop, the generation functions, and so on. A handful of short scripts sit next to the package and do the actual work: pretraining, sampling, classifying, instruction tuning. By the end, the folder you built is a small but complete language-model project that you can read from top to bottom in an afternoon.

What QuillGPT-mini Will and Will Not Do

It is worth setting expectations now, because Marisol certainly will ask. After pretraining, when we give QuillGPT-mini the title "The Fox and the Heron" and let it continue greedily, it writes text like this (Chapter 18 shows the full output, and shows that greedy decoding writes this same opening whatever the title):

```
In the days when animals spoke, a patient fox made its home by the willow marsh. The
hare found a purse of acorns on the path to the path to the miller's yard.
```

That is genuinely good for a model with under a million parameters trained for a few minutes. The opening formula, the setting, and the plot are all in the Quillbrook style. But read closely. A hare appears from nowhere, even though the title promised a fox and a heron, and the model stumbles into "the path to the path to". It has learned the *shape* of a fable very well and the *bookkeeping* of a story (who is who, what already happened) only partly.

So, for the record, here is what we can honestly promise:

- **It will** write new text in the style of the Quillbrook Fables, learn to state morals, sort fables by lesson, and follow a few simple instructions after tuning.
- **It will** be small enough to train on a CPU in minutes, save in about 3.8 MB, and inspect line by line.
- **It will not** know anything outside its corpus. It has never seen a newspaper, a recipe, or a line of code. Ask it about anything but fables and it will still write about foxes.
- **It will not** be reliably correct. Like every language model, it produces likely-looking text, not verified facts. A plausible but wrong moral is a real failure mode, and Chapter 28 discusses what that means for shipping a model.

Those limits are not embarrassing; they are the point. Because the model is small and the data is simple, you can see exactly where its knowledge ends. The same failures exist in giant models; they are just harder to spot.

Try It Yourself

1. Add a fourth sentence to the text in `next_word.py`, for example "The owl found the honey." Predict how the probabilities after "the" will change, then run the script and check.
2. Change `random.Random(42)` to another seed and generate ten sampled lines. How many are sentences that never appeared in the training text? How many are nonsense?
3. In `softmax_demo.py`, divide every logit by a temperature of 0.1 before the exponential. What probability does "fox" get now? What happens with a temperature of 10?
4. Compute ln(V) for a vocabulary of 50 characters and for GPT-2's 50,257 tokens with `math.log`. Which model starts from a higher loss, and why does that not mean it is worse?
5. Write down three questions Marisol might ask about QuillGPT. Next to each, note which chapter you expect to answer it, and check your guesses as you read.