Building an LLM from Zero

So Simple You Can Teach It to Your Kids

Truong (Jack) Luu

First draft: December 17, 2025

Last edited: March 2026

Source code: github.com/jackluucoding/build-llm-from-zero
Read online: jackluu.io/book/
Contact: hi@jackluu.io

Abstract

This book is a hands-on tutorial for building a GPT-style language model entirely from scratch using Python and PyTorch. It covers every component of the Transformer architecture — tokenization, embeddings, self-attention, multi-head attention, feed-forward layers, residual connections, and layer normalization — before assembling them into a complete language model, training it on the Tiny Shakespeare dataset, and generating text. No GPU is required; the full training run completes in approximately 20-30 minutes on a standard CPU-only laptop. The book is designed for readers with basic Python knowledge and no prior machine learning background, including high school students, college instructors seeking classroom-ready materials, IT professionals, and parents teaching their children about artificial intelligence.

Keywords: large language model, GPT, Transformer, PyTorch, deep learning, natural language processing, machine learning tutorial, artificial intelligence education

Contents

Preface: Why I Wrote This Book

There are already a lot of resources out there for learning about AI and large language models. So why write another one?

Because most of them fall into one of three traps, or they push you straight into **tutorial hell**.

Tutorial hell is a state of learning paralysis where you endlessly watch coding tutorials and follow along with instructors, feeling productive, but never actually building anything on your own. You run the code, it works, you feel good, but close the laptop and ask yourself *why* any of it worked, and you draw a blank. You've watched the chef cook a hundred times but have never touched the stove yourself.

The three traps that lead there:

- **Too shallow.** Copy-and-paste the code, follow along, done. You produce an output, but you don't understand any of the decisions behind it. The moment something breaks or changes, you're stuck.
- **Too deep.** Research-paper density, prerequisite courses in linear algebra and calculus, written for people who already have a graduate-level background. Most readers bounce off in chapter two.
- **Too demanding on hardware.** Great content, but assumes you have a modern GPU, a cloud computing account, and several dedicated weekends free. That rules out most people.

This book is a different approach.

The goal is a **balance between theory and code**, enough explanation to genuinely understand what you are building and why each piece is there, paired with real, runnable code you can execute right now on the machine in front of you.

Every chapter follows the same structure: first the idea in plain language with an analogy, then the code that implements it, then a short summary. You are never asked to accept something on faith. If a line of code does something, the chapter explains why.

Right now means on a regular laptop. No GPU. No cloud. No special hardware. This entire book was written and tested on a ThinkPad T14 Gen 1 (Intel Core i7, 32 GB RAM, released 2020), a five-year-old business laptop with no dedicated GPU. The full training run completed in about 20-30 minutes. If it runs there, it will run on yours.

Setup is minimal: Python and PyTorch. That's it. No accounts to create, no clusters to configure.

0.1 Who Is This Book For?

Beginners who learn by building. You know some Python, you're curious about AI, and you want to actually understand how it works, not just use someone else's model. You want to go from zero to a working language model, line by line, without getting stuck in tutorial hell.

Instructors and professors. You want classroom-ready material: concepts clear enough to teach, code that runs on a student's laptop in a single class session, and a structure that maps cleanly to a lecture. This book is designed to be that.

IT/IS professionals. You work with AI tools every day but want to understand what's happening under the hood. You don't need a PhD, you need a clear explanation of the architecture, a working example, and code you can actually read and modify.

Parents teaching their kids. AI is everywhere. If you want to introduce your child to how it actually works, not just how to use it, this book gives you a concrete, hands-on project you can work through together. Build something real. Ask questions. Break it and fix it. That's how learning sticks.

0.2 A Note on AI Assistance

This book was written by Truong (Jack) Luu. Claude (by Anthropic) was used as an AI assistant to help with writing plans, generating code drafts, and editing prose. All code was reviewed, edited, and tested by the author on a local machine. Every example in this book runs exactly as shown.

Truong (Jack) Luu jackluu.io

Part I

Setup

Chapter 1

Setting Up Your Environment

> *No code to write yet. Just getting your computer ready.*

1.1 What You Need (and Why)

Before you can build anything, you need four tools installed on your computer:

1. **Python** - the programming language all the code in this book is written in.
2. **Git** - a tool for downloading the project code from the internet.
3. **PyTorch** - a math library that makes training neural networks fast and easy.
4. **A terminal** - the text-based window where you type commands to run code.

Think of it like setting up a kitchen before you cook. Python is the kitchen itself, PyTorch is a specialized appliance (a very powerful blender for numbers), Git is the service that delivers the recipe book to your door, and the terminal is how you talk to all of them.

You only have to do this setup once. After that, you just open the terminal and start coding.

1.2 Step 1: Open a Terminal

Before you install anything, you need to know how to open the command window where you will type instructions.

Windows:

Press `Win + R`, type `cmd`, and press Enter. A black window appears with a blinking cursor. That is the Command Prompt - your terminal on Windows. Everything you type here gets executed immediately when you press Enter.

macOS:

Press `Cmd + Space` to open Spotlight, type `Terminal`, and press Enter. A white or dark window opens with a `$` prompt. That is the Terminal on macOS.

You will use this window for every step below.

1.3 Step 2: Install Python

Python is the programming language this entire tutorial is written in. We need version 3.10 or newer.

Windows:

1. Open your web browser and go to **python.org/downloads**
2. Click the big yellow button: "Download Python 3.13.x" (the exact number after 3.13 does not matter)
3. Run the downloaded `.exe` installer
4. **CRITICAL:** On the first screen of the installer, check the box that says **"Add Python to PATH"** at the very bottom. If you miss this, Python will not be found when you type commands. Check it before clicking Install Now.
5. Click "Install Now" and wait for it to finish.

To verify it worked, open a new Command Prompt window and type:

```
python --version
```

You should see something like `Python 3.13.2`. If you do, Python is installed correctly.

macOS:

1. Open your web browser and go to **python.org/downloads** (same site)
2. Click the big yellow button: "Download Python 3.13.x"
3. Run the downloaded `.pkg` installer and follow the prompts (click Continue through all the screens)

To verify, open Terminal and type:

```
python3 --version
```

You should see `Python 3.13.x`. On macOS, the command is `python3` (with the 3) instead of just `python`.

Troubleshooting:

- **Windows: "'python' is not recognized as an internal or external command"** - This means you forgot to check the "Add Python to PATH" box during installation. Uninstall Python from Control Panel, reinstall it, and this time check that box.
- **macOS: "command not found: python3"** - Try typing `python --version` instead (without the 3). If that also fails, retry the installation steps above.

1.4 Step 3: Install Git

Git is a version control tool. For our purposes, it is simply the command we use to download the project from the internet.

Windows:

1. Go to **git-scm.com/download/win**
2. The download should start automatically. If not, click the link for "64-bit Git for Windows Setup"
3. Run the downloaded installer
4. Click "Next" through every screen - all the default settings are fine. You do not need to change anything.
5. Click "Install" at the end.

To verify, open a **new** Command Prompt window (close the old one and open a fresh one) and type:

```
git --version
```

You should see something like `git version 2.48.1.windows.1`.

macOS:

Git is often already installed on macOS. Open Terminal and type:

```
git --version
```

If Git is installed, you will see a version number. If it is not installed, macOS will pop up a dialog asking if you want to install the "Command Line Developer Tools" - click Install and wait for it to finish.

1.5 Step 4: Download the Project

Now that Git is installed, you can download the project code with a single command.

A **repository** (or "repo") is just a folder of code stored online. The `git clone` command creates a copy of that folder on your computer.

Open your terminal and type:

```
git clone https://github.com/jackluucoding/build-llm-from-zero.git
```

Then move into the project folder:

```
cd build-llm-from-zero
```

You should now be inside the project folder. You can verify this by typing `dir` (Windows) or `ls` (macOS) to see the files - you should see folders like `book`, `src`, and `checkpoints`.

Windows tip: You can paste commands into Command Prompt by right-clicking inside the window.

1.6 Step 5: Create a Virtual Environment

Here is a problem you will eventually run into without this step: different projects need different versions of the same library. Project A needs PyTorch version 1.0, Project B needs version 2.0. If you install both globally, they overwrite each other.

A **virtual environment** solves this by giving each project its own isolated box of installed packages. Think of it as each project having its own dedicated toolbox. Project A keeps its tools in its own toolbox. Project B keeps its tools in its own toolbox. They never interfere with each other.

Make sure your terminal is inside the `build-llm-from-zero` folder (the `cd` command in Step 4 should have taken you there), then run:

Windows:

```
python -m venv venv
venv\Scripts\activate
```

macOS:

```
python3 -m venv venv
source venv/bin/activate
```

After you run the activate command, you will see `(venv)` appear at the start of your terminal prompt, like this:

```
(venv) C:\Users\yourname\build-llm-from-zero>
```

That `(venv)` tag tells you the virtual environment is active and ready. Every package you install from now on goes into this isolated box, not your main Python installation.

Important: Every time you open a new terminal window to work on this project, you need to run the activate command again. The `(venv)` tag disappears when you close the terminal. It does not install anything new - it just switches your active environment back on.

1.7 Step 6: Install PyTorch and Dependencies

With the virtual environment active (you should see `(venv)` in your prompt), install the required libraries.

pip is Python's package manager - think of it as an app store for code libraries. When you type `pip install something`, it downloads and installs that library automatically.

First, install PyTorch. We use a CPU-only version because this tutorial does not require a graphics card:

```
pip install torch==2.6.0 --index-url https://download.pytorch.org/whl/cpu
```

This downloads about 200 MB, so it may take a few minutes depending on your internet speed.

Then install the other libraries:

```
pip install numpy requests matplotlib
```

- **numpy**: math utilities used alongside PyTorch
- **requests**: for downloading the Shakespeare dataset
- **matplotlib**: for plotting training loss (used in one chapter)

To verify PyTorch installed correctly:

```
python -c "import torch; print(torch.__version__)"
```

You should see `2.6.0+cpu`. If you see that, PyTorch is ready.

macOS note: Use `python3` instead of `python` if the command is not found.

1.8 Step 7: Download the Shakespeare Dataset

The tutorial trains the model on the complete works of Shakespeare, a text file about 1 MB in size. Run this to download it:

```
python src/utils/download_data.py
```

The script will print a confirmation message and show the first few lines of the text so you can see it worked. The file is saved to `src/data/shakespeare.txt`.

1.9 Step 8: Verify Everything Works

Run the setup verification script to confirm all pieces are in place:

```
python src/ch00_setup_check.py
```

If everything is set up correctly, you will see:

```
Checking your environment...

[OK] Python 3.13.x
[OK] PyTorch 2.6.0+cpu
[OK] NumPy x.x.x
[OK] Requests x.x.x
[OK] Shakespeare dataset found (1,115,394 characters)
[OK] Quick tensor test passed

All checks passed! You are ready to start Chapter 1.
```

If any line shows `[FAIL]`, read the message next to it - it will tell you exactly which step to redo.

1.10 Troubleshooting

"pip: command not found" or "pip is not recognized" Try `pip3` instead of `pip`. If neither works, try `python -m pip install ...` (Windows) or `python3 -m pip install ...` (macOS).

"ModuleNotFoundError: No module named 'torch'" when running a script Your virtual environment is not active. Open a terminal, navigate to the project folder, and run the activate

command again (`venv\Scripts\activate` on Windows, `source venv/bin/activate` on macOS). You need to do this every time you open a new terminal window.

Dataset download fails with a network error Check your internet connection. If the error persists, wait a few minutes and try again. The dataset can also be found by searching for "Tiny Shakespeare Karpathy" - download the raw text file and save it manually to `src/data/shakespeare.txt`.

"python: command not found" on Windows even after installation The PATH checkbox was not checked during installation. Go to Control Panel > Apps, find Python, uninstall it, then reinstall and check the PATH box.

1.11 Key Takeaways

- You installed four things: Python (the language), Git (code downloader), PyTorch (math library), and a terminal (how you talk to them).
- A virtual environment keeps this project's packages isolated from everything else on your computer.
- Always activate your virtual environment (`venv\Scripts\activate` or `source venv/bin/activate`) before working on this project.
- If `ch00_setup_check.py` shows all [OK], you are fully ready to begin.

Part II

Foundations

Chapter 2

What Is a Language Model?

> *No code in this chapter, just ideas. Grab a snack.*

2.1 Theory

2.1.1 The World's Most Useful Party Trick

Imagine you're texting a friend and you type:

> *"I'm so hungry, I could eat a -"*

Before you finish, your phone suggests: **horse**. Or **pizza**. Or **whole building**.

Your phone is doing something called **next-word prediction**, it guesses what word is most likely to come next based on everything you've typed so far.

A **Language Model (LM)** does exactly the same thing, just much better.

Given a sequence of words (or characters), it predicts: *"What comes next?"*

That's it. That's the whole idea. Everything else in this tutorial is just *how* to build a machine that can do this really well.

2.1.2 "But ChatGPT Talks Back to Me..."

Yes, but under the hood, it's still just predicting the next token, one at a time.

Here's how a conversation works at the model level:

1. You type: `"What is the capital of France?"`
2. The model predicts the next token: `"Paris"`, most likely.

3. Then it predicts the token after that: `" is"`, still makes sense.
4. Then: `" the"`, then `" capital"`, then `" of"`, then `" France"`, then `"."`, then it decides to stop.

The model never "thinks" about the whole answer at once. It just keeps predicting the next piece, over and over.

This is called **autoregressive generation** (auto = self, regressive = looking back). The model uses its own previous outputs as inputs for the next step.

Here is what that means concretely: when the model outputs "Paris", that word gets added to the input for the next step. Now the input is "What is the capital of France? Paris", and the model predicts what comes after that. Each new word it generates becomes part of the context for the next word. It is like writing by hand - once you write a word, you cannot erase it, you can only add what comes next.

2.1.3 What Is a "Token"?

A token is the smallest unit the model works with. Depending on the model, a token can be:

- **A character**: `H`, `e`, `l`, `l`, `o` (5 tokens for "Hello")
- **A word piece**: `Hello` (1 token)
- **A word**: `Hello` (1 token, different approach)

In this tutorial, we use **character-level tokens**, the simplest option. Every single character (letters, spaces, punctuation) is one token.

Why character-level? Because: - The vocabulary is tiny (only 65 characters in Shakespeare) - There's nothing to install or configure - You can understand it completely in 5 minutes

Real models like GPT-4 use more complex tokenization (called BPE, Byte Pair Encoding), but the *idea* is the same.

2.1.4 What Makes a Language Model "Large"?

The "L" in LLM stands for **Large**, meaning it has a lot of **parameters**.

Parameters are the numbers inside the model that get adjusted during training. Think of them like dials on a radio: training turns them to the right settings so the model produces good output.

Model	Parameters	Year
Our model (this tutorial)	~825,000	2024
GPT-2 Small	117,000,000	2019
GPT-3	175,000,000,000	2020
GPT-4 (estimated)	~1,800,000,000,000	2023

Our model is tiny by comparison. But it uses the **exact same architecture** as GPT-2, just with fewer layers and smaller dimensions.

It's like comparing a toy car to a Formula 1 racer. Same basic design, very different scale.

2.1.5 The Training Process (Big Picture)

Building an LLM happens in two phases:

Phase 1: Architecture Design We design the model's structure, how information flows through it, what operations it performs. This is what Parts 1-3 of this tutorial cover.

Phase 2: Training We show the model millions of examples of text and adjust its parameters to make it better at predicting the next token. The model never sees a human label like "this is good writing." It just sees raw text and learns patterns.

This kind of training is called **self-supervised learning**.

In most machine learning, you need labeled training data: humans manually mark "this photo contains a cat" or "this email is spam." Labels = the correct answers.

With language modeling, the labels are built right into the text. If your training text is "Hello world", you get: - Input: `H` - Label: `e` (the next character) - Input: `He` - Label: `l` - Input: `Hel` - Label: `l` - … and so on

The text itself is the answer key. No human labelers needed. That is why you can train on any raw text from the internet - billions of unlabeled pages become billions of training examples automatically. That is the superpower of self-supervised learning.

2.1.6 What We're Building

By the end of this tutorial, you'll have:

```
Input  : "To be or not to be, that is the "
Output : "q"  ← untrained
Output : "question"  ← after training!
```

The model goes from outputting gibberish to outputting Shakespeare. Not because anyone told it what Shakespeare sounds like, but because it learned the patterns from the data itself.

Pretty cool, right?

2.2 Key Takeaways

- A language model predicts "what comes next" one token at a time.
- Autoregressive generation = using your own outputs as inputs for the next step.
- Tokens are the basic units, in our case, individual characters.
- Parameters are the learnable numbers inside the model.
- Training = adjusting parameters to get better at next-token prediction.

Chapter 3

Tensors and PyTorch Basics

> ***Code file:*** *src/ch02_tensors.py* ***Run it:*** *python src/ch02_tensors.py*

3.1 Theory

3.1.1 What Is PyTorch?

PyTorch is a Python library for doing math on large arrays of numbers, really, really fast.

You might ask: "Can't we just use regular Python lists?" Yes, technically. But consider: a single 1000x1000 matrix multiply requires 1 billion individual multiplications (1000 x 1000 x 1000). Pure Python would perform these one at a time, taking several seconds per operation. PyTorch uses optimized C++ code under the hood that performs thousands of multiplications simultaneously, completing the same operation in microseconds. Our model does millions of such operations during training. Without PyTorch, training would take weeks instead of minutes.

Think of PyTorch as a turbo-charged calculator.

3.1.2 What Is a Tensor?

A **tensor** is the fundamental data type in PyTorch. It's a multi-dimensional array of numbers.

If that sounds scary, don't worry. You already know tensors:

- A **0D tensor** (scalar) is just one number: `3.14`
- A **1D tensor** (vector) is a list of numbers: `[1, 2, 3, 4]`
- A **2D tensor** (matrix) is a table of numbers: like a spreadsheet
- A **3D tensor** is a stack of tables: like a spreadsheet with multiple sheets

```
1D: [1, 2, 3]                    ← a row
```

```
2D: [[1, 2, 3],                  ← a table
     [4, 5, 6]]

3D: [[[1, 2], [3, 4]],           ← a stack of tables
     [[5, 6], [7, 8]]]
```

In this tutorial, we work with 3D tensors a lot. Their dimensions represent: - **B** = Batch size (how many text sequences we process at once) - **T** = Time (how many tokens in each sequence) - **C** = Channel size (embedding dimension: how many numbers we use to represent each token)

We write shapes like `(B, T, C)`, for example, `(32, 128, 128)` means "32 sequences, each 128 tokens long, each token described by 128 numbers."

3.1.3 Key Operations

Here are the operations that appear throughout the tutorial:

Creating tensors:

```
import torch
a = torch.tensor([1.0, 2.0, 3.0])       # from a Python list
b = torch.zeros(3, 4)                     # all zeros, shape (3, 4)
c = torch.randn(2, 5)                     # random values (bell curve)
```

Checking the shape:

```
x = torch.randn(2, 5, 8)
print(x.shape)    # torch.Size([2, 5, 8])
```

Reshaping:

```
x = torch.arange(12.0)   # [0, 1, 2, ..., 11]
y = x.reshape(3, 4)       # re-arrange into 3 rows of 4
# It's like re-folding a piece of paper: same numbers, different shape.
```

Matrix multiplication (`@`):

```
A = torch.randn(3, 4)    # shape (3, 4)
B = torch.randn(4, 5)    # shape (4, 5)
C = A @ B                # shape (3, 5)
# Rule: (m, n) @ (n, p) = (m, p)
# The inner dimensions must match!
```

Why must the inner dimensions match? Think of (3, 4) as "3 students each with 4 test scores" and (4, 5) as "4 test scores each mapped to 5 skill ratings". The 4 scores in the first matrix match the 4 scores in the second - that's what lets you combine them. You couldn't combine (3, 4) and (5, 6) because there's a mismatch: 4 scores don't connect to 5 scores.

Softmax, converts any numbers into probabilities (they sum to 1):

```
logits = torch.tensor([1.0, 2.0, 3.0])
probs  = torch.softmax(logits, dim=0)
# Output: [0.09, 0.24, 0.67]  ← sum = 1.0
# The biggest input (3.0) gets the biggest probability.
```

3.1.4 Why Does Softmax Sum to 1?

Softmax uses the formula:

```
P(i) = exp(x_i) / sum(exp(x_j) for all j)
```

`exp` means e^x, where e is a special mathematical constant (approximately 2.718). Why use it? Two reasons: 1. `exp(x)` is always positive, so we never get negative probabilities. 2. `exp(x)` amplifies differences: `exp(3) = 20` and `exp(1) = 2.7`, so 3 becomes about 7x more dominant than 1 after exp. This means the highest-scoring option gets a meaningfully higher probability.

By dividing each `exp(x_i)` by the sum of all `exp(x_j)`, we guarantee everything sums to 1. This turns raw "scores" (called **logits**) into a probability distribution.

Think of it like: you and your friends vote on pizza toppings. Each person's enthusiasm is a logit. Softmax converts "enthusiasm scores" into "probability each topping wins."

3.2 Code

> ***File:*** *src/ch02_tensors.py* ***Run it:*** *python src/ch02_tensors.py*

This file demonstrates: 1. Creating 1D, 2D, and 3D tensors 2. The `(B, T, C)` shape convention used throughout the tutorial 3. Reshaping 4. Matrix multiplication 5. Softmax and transpose

Run it and read the output carefully. Every operation here will appear again in the attention mechanism (Chapter 5).

3.2.1 What to look for

When you run it, pay attention to the shape annotations. For example:

```
3D matmul: torch.Size([2, 5, 8]) @ torch.Size([8, 4]) = torch.Size([2, 5, 4])
```

PyTorch automatically handles the batch dimension (2) when doing matrix multiplication on 3D tensors. This is called **broadcasting**: PyTorch expands the smaller tensor's shape to match the larger one so the operation works without writing loops.

You could write a loop: `for i in range(batch_size): result[i] = sequences[i] @ weights`. But that's slow - Python executes each iteration one at a time. Broadcasting tells PyTorch to apply the operation to all 32 sequences at once using optimized parallel code. Same result, much faster.

3.3 Key Takeaways

- A tensor is a multi-dimensional array of numbers.
- Shape `(B, T, C)` = batch x time x channel size, the standard convention in this tutorial.
- `@` is matrix multiplication. Inner dimensions must match.
- Softmax turns any numbers into probabilities that sum to 1.
- PyTorch handles batches automatically, no for-loops needed.

Chapter 4

Tokenization

> ***Code file:*** *src/ch03_tokenizer.py* ***Run it:*** *python src/ch03_tokenizer.py*

4.1 Theory

4.1.1 The Problem: Neural Networks Are Number Machines

Neural networks can only work with numbers. They can't eat text directly, it's like trying to feed a dog a photograph of a steak. Technically related, completely useless.

So we need to convert text into numbers. That process is called **tokenization**.

4.1.2 Our Approach: Character-Level Tokenization

The simplest tokenizer imaginable: 1. Find every unique character in the text. 2. Assign each character a unique integer. 3. To encode text, replace each character with its integer. 4. To decode, do the reverse.

For example, with a tiny vocabulary `{a: 0, b: 1, c: 2}`:

```
encode("cab") = [2, 0, 1]
decode([2, 0, 1]) = "cab"
```

That's literally it.

4.1.3 Building the Vocabulary

The Shakespeare dataset contains exactly **65 unique characters**: - Lowercase letters: `a` through `z` (26) - Uppercase letters: `A` through `Z` (26) - Digits: `3` (only one digit appears in all of Shakespeare - most numbers were written as words back then) - Punctuation: , `!`, `$`, `&`, `'`, `,`, `-`, `.`, `:`, `;`, `?` - Newlines

We sort them to get a consistent ordering:

```
chars = sorted(set(text))   # ['\\n', ' ', '!', '$', '&', "'", ',', '-', '.', ...]
stoi  = {ch: i for i, ch in enumerate(chars)}   # string to integer
itos  = {i: ch for i, ch in enumerate(chars)}   # integer to string
```

4.1.4 Why Character-Level?

Other tokenization approaches exist:

Word-level: Each word is one token. Problem: huge vocabulary (50,000+ words), rare words not handled well, punctuation messiness.

Byte-Pair Encoding (BPE): Groups frequently co-occurring characters into single tokens. For example, `th`, `ing`, and `tion` each become one token instead of 2-4 characters. GPT-4 uses this. Problem: requires a separate training pass, complex to implement.

Character-level: Simple, transparent, zero dependencies. Vocabulary of only 65. Perfect for learning.

The downside: longer sequences. "Hello" = 5 tokens character-level, but often 1 token with BPE. With our `block_size=128`, we can only "see" 128 characters at once instead of perhaps 128 words.

For a tutorial laptop model, that's totally fine.

4.1.5 The Vocabulary Is Learned From the Data

This is subtle but important: our tokenizer is *built from the training data*. We don't import a dictionary, we look at the actual text and extract the vocabulary.

If a character never appears in Shakespeare, it has no token ID. Our tokenizer only knows the 65 characters in the training data. Feed it an emoji or a Chinese character and it tries to look the character up in the `stoi` dictionary. It finds nothing, so Python raises a `KeyError: <character not found>` and your program crashes.

Production systems handle this with a special `<UNK>` (unknown) token as a fallback - any unseen character maps to `<UNK>` instead of crashing. But for our tutorial, sticking to Shakespeare keeps it simple.

4.2 Code

> ***File**: `src/ch03_tokenizer.py` **Run it**: `python src/ch03_tokenizer.py`*

This file: 1. Loads `shakespeare.txt` 2. Builds the 65-character vocabulary 3. Creates `encode()` and `decode()` functions 4. Demonstrates a round-trip: text → integers → text 5. Encodes the entire dataset as a PyTorch tensor 6. Splits it into training (90%) and validation (10%) sets

4.2.1 Sample output

```
Original : 'Hello, World!'
Encoded  : [20, 43, 50, 50, 53, 6, 1, 35, 53, 56, 50, 42, 2]
Decoded  : 'Hello, World!'
Round-trip matches: True
```

The exact numbers will match yours since we use `sorted()`, the vocabulary ordering is deterministic.

4.2.2 Training vs Validation Split

We split the data 90%/10%: - **Training set**: The model learns from this (sees it during training). - **Validation set**: We test on this to check if the model is really learning patterns, or just memorizing.

If training loss goes down but validation loss stays high, the model is **overfitting** (memorizing rather than learning). For example: training loss 1.3, validation loss 2.5 means the model has memorized specific character sequences from the training text, but fails on new text it has never seen before.

Imagine memorizing every answer to a practice test without understanding the material. You'd ace the practice test but fail the real exam. That's overfitting. Validation loss is our surprise quiz on material the model has never studied.

4.3 Key Takeaways

- Tokenization converts text to integers so neural networks can process it.
- We use character-level tokenization: each character gets a unique integer ID.
- `encode(text)` converts string to list of ints; `decode(ids)` does the reverse.
- The vocabulary (65 characters) is extracted directly from the dataset.
- We split data 90/10 into train and validation sets.

Chapter 5

Embeddings

> ***Code file:*** *src/ch04_embeddings.py* ***Run it:*** *python src/ch04_embeddings.py*

5.1 Theory

5.1.1 Why Can't We Just Use the Token IDs?

After tokenization, each character is an integer: `A=33`, `B=34`, `a=39`, etc.

But using these raw integers as input to a neural network is a problem. The number `39` for `a` doesn't mean `a` is "more" than `A` (number 33). There's no mathematical relationship between these integers that mirrors the relationships between characters.

Here is what goes wrong: neural networks learn by multiplying numbers together. If 'A' is token 33 and 'Z' is token 64, the network sees 'Z' as a bigger number than 'A' - almost twice as big. It might learn that 'Z' is more important than 'A' just because of this accident. But token IDs are assigned arbitrarily - 'Z' could have been token 1 instead. The ordering is meaningless.

Feeding raw token IDs to a neural network is like telling someone the Dewey Decimal numbers of books and expecting them to understand literature. Or, if you've never used a library catalog: it's like texting a friend "I want to talk about item number 39" without saying what item 39 is. The number alone carries no meaning.

5.1.2 The Embedding Table

The solution: a **lookup table** that maps each token ID to a vector of floating-point numbers.

```
token ID 33  →  [0.12, -0.45, 0.83, ...]   ← 128 numbers
token ID 34  →  [-0.22, 0.71, -0.34, ...]  ← 128 numbers
token ID 39  →  [0.55, 0.03, -0.89, ...]   ← 128 numbers
```

These 128 numbers (the "embedding vector") are what the model actually works with. They're initialized randomly but **learned during training**: the model adjusts them using gradient descent, the same mechanism that trains everything else. Tokens that appear in similar contexts will gradually develop similar vectors.

In PyTorch:

```
embedding = nn.Embedding(vocab_size=65, embedding_dim=128)
```

This creates a table of shape `(65, 128)`, 65 rows (one per character), 128 columns (one per embedding dimension).

When we pass token IDs, PyTorch simply looks up the corresponding rows:

```
input:  [33, 39]     ← token IDs for 'Aa'
output: [[0.12, -0.45, ...],   ← row 33 from the table
         [0.55, 0.03, ...]]    ← row 39 from the table
```

Shape change: `(B, T)` → `(B, T, C)` where `C = n_embd = 128`.

5.1.3 What Do Embeddings Learn?

After training, similar characters tend to have similar embeddings. Characters that appear in similar contexts end up with nearby vectors in the 128-dimensional space.

For words (not characters), this is even more interesting. Trained models learn that `king - man + woman` is very close to `queen`. Here is why: 'king' and 'queen' appear in similar contexts (palaces, thrones, crowns), so their embedding vectors point in similar directions. 'King' and 'man' appear in different patterns ('a man walked' vs 'a king ruled'), so their vectors differ in a specific way. Subtracting that difference and adding 'woman' bridges from 'king' to 'queen' in the vector space. Our character model learns simpler patterns (e.g., capital and lowercase versions of the same letter tend to cluster together).

Think of embeddings as assigning each character a "personality profile" of 128 numbers. The model learns these profiles during training.

5.1.4 Positional Embeddings

Here's a problem: the transformer doesn't inherently know the **order** of tokens.

Unlike reading a sentence word-by-word from left to right, the transformer processes all tokens at the same time, in parallel. It looks at the entire set of tokens at once. Without position information,

"cat eats dog" and "dog eats cat" look identical: just three tokens, shuffled. Order matters, so we need to tell the model where each token sits.

We fix this with **positional embeddings**: a second lookup table indexed by *position* rather than token ID.

```
pos_embedding = nn.Embedding(block_size=128, embedding_dim=128)
```

Position 0 gets its own 128-number vector, position 1 gets a different 128-number vector, and so on - up to position 127.

We **add** the positional embedding to the token embedding (rather than, say, concatenating them) because addition keeps the same 128 dimensions while mixing both pieces of information together. The model learns to use those shared 128 numbers to represent both 'what token is this?' and 'where does it appear?'.

```
Final token representation = token_embedding + position_embedding
```

Shape: `(B, T, C)` + `(T, C)` broadcast = `(B, T, C)`

5.1.5 Learned vs. Fixed Positional Encodings

There are two approaches:

Fixed: Use a mathematical formula to assign each position a unique fingerprint - position 0 gets one pattern, position 1 gets a different pattern, position 2 another, and so on. These fingerprints never change during training. The original Transformer paper used this approach.

Learned: Treat positions like tokens - give each position an ID (0, 1, 2, ...) and let the model learn a lookup table of position embeddings, just like it learns token embeddings. GPT-2 uses this. We use this too.

For our small model, the difference is negligible. Learned embeddings are simpler to implement.

5.2 Code

> ***File:*** *src/ch04_embeddings.py* ***Run it:*** *python src/ch04_embeddings.py*

This file: 1. Creates a token embedding table `(65, 128)` 2. Demonstrates lookup: `(B, T)` → `(B, T, C)` 3. Creates a position embedding table `(128, 128)` 4. Combines token + position embeddings 5. Prints a parameter count for the embedding layers

5.2.1 Key line to understand

```
x = token_embeddings + position_embeddings    # (B, T, C)
```

This `x` is the **input to the transformer blocks**. Every chapter after this operates on this 3D tensor.

5.3 Key Takeaways

- Embeddings are learned lookup tables that map token IDs to float vectors.
- Shape change: `(B, T)` → `(B, T, C)` where `C = n_embd = 128`.
- Position embeddings encode the position of each token (0, 1, 2, …).
- The final input to the transformer = token embedding + position embedding.
- Both embedding tables are learned from scratch during training.

Part III

The Attention Mechanism

Chapter 6

Self-Attention (Single Head)

> ***Code file:*** *src/ch05_self_attention.py* ***Run it:*** *python src/ch05_self_attention.py*

6.1 Theory

6.1.1 The Problem: Context Matters

Read this sentence: *"The trophy didn't fit in the suitcase because it was too big."*

What does "it" refer to, the trophy or the suitcase?

You figured it out in milliseconds. But how? You read the whole sentence and connected the word "it" to "trophy" because of the word "big", only the trophy could be too big to fit.

Simple word-by-word processing can't do this. Each word needs to "look at" other words and decide which ones are relevant to understanding its meaning. That's exactly what **self-attention** does.

6.1.2 The Library Analogy: Q, K, V

Self-attention uses three concepts:

Query (Q): What am I looking for? **Key (K)**: What do I offer? **Value (V)**: What information do I actually contain?

Imagine a library: - You walk in with a question: "Books about ocean ecology." That's your **Query**. - Every book has an index card describing what it's about. That's each book's **Key**. - The actual text content of each book is its **Value**.

You compare your Query against every Key to find the best matches, then read (take a weighted mix of) the Values of the matching books.

In self-attention, every token simultaneously acts as a Query ("what do I need?"), a Key ("what do I have?"), and a Value ("what's my content?"), at the same time!

Let's make this concrete. Suppose we have the sequence: `[cat, eats, the, big, dog]`. When the model processes "eats": - Its **Q** (query) asks: "What context do I need to understand my role here?" - Its **K** (key) announces: "I'm a verb, an action word." - Its **V** (value) holds: "Full details about me - a past-tense verb, an action."

The model then compares "eats" Q against every token's K: - eats-Q vs. dog-K: "A noun is very relevant to a verb." High attention score. - eats-Q vs. the-K: "An article is less relevant." Lower attention score.

So "eats" ends up borrowing more information from "dog" than from "the". That's attention in action.

6.1.3 The Math, Step by Step

One key term before we start: **head_size** is the dimension of each attention head. In our model, we split the 128-dimensional embedding across 4 heads: 128 / 4 = 32 per head. Chapter 6 explains this split. For now, just know that head_size = 32.

Given input `x` of shape `(B, T, C)`:

Step 1: Project to Q, K, V

`W_q`, `W_k`, and `W_v` are learnable weight matrices (shape: C x head_size). Think of them as filters: `W_q` multiplies each token's 128-number embedding to extract only the "query-relevant" aspects - out of 128 numbers, which ones matter for asking questions? `W_k` filters for "key" aspects. `W_v` filters for "value" aspects. These filters are learned automatically during training.

```
q = x @ W_q   # "What am I looking for?" shape: (B, T, head_size)
k = x @ W_k   # "What do I offer?"       shape: (B, T, head_size)
v = x @ W_v   # "My actual content"       shape: (B, T, head_size)
```

Step 2: Compute attention scores

We want to score every pair of tokens: "how much should token i pay attention to token j?" To do this using matrix multiply, we need shape (T, head_size) @ (head_size, T) = (T, T), which gives one score for every pair. The transpose flips K's last two dimensions to make this possible.

`k.transpose(-2, -1)` swaps K's last two dimensions, turning shape `(B, T, head_size)` into `(B, head_size, T)`. This lets us compute `(B, T, head_size) @ (B, head_size, T) = (B, T, T)` - a score for every token pair.

```
scores = q @ k.transpose(-2, -1)   # (B, T, T)
scores = scores / sqrt(head_size)   # scale down to avoid huge values
```

Token `i`'s scores at position `[i, j]` = "how much should token `i` pay attention to token `j`?"

Step 3: Apply the causal mask

```
scores = scores.masked_fill(future_positions, float("-inf"))
```

This sets scores for all future tokens to `-infinity`. Softmax will convert these to 0, so token `i` cannot "see" tokens `j > i`.

Why? If the model can see all future tokens during training, it learns a cheat: just look at position t+1 and copy whatever is there. It gets perfect training accuracy but learns absolutely nothing about language. And during generation (Chapter 14), future tokens literally do not exist yet - the model is supposed to create them. The causal mask forces the model to learn real patterns from context rather than cheating.

Step 4: Softmax → attention weights

```
weights = softmax(scores)    # (B, T, T), each row sums to 1
```

Now `weights[i, j]` = "what fraction of attention does token `i` give to token `j`?"

Step 5: Weighted sum of values

```
output = weights @ v    # (B, T, T) @ (B, T, head_size) = (B, T, head_size)
```

Each token's output = a mix of all values, weighted by how relevant each was.

6.1.4 Why Divide by sqrt(head_size)?

Here is the problem without scaling: a dot product of two 32-dimensional vectors (head_size = 32) can easily reach large numbers - imagine summing 32 multiplications, each potentially 1 or 2, the total could be 30-50 or higher.

Feed those large scores into softmax: - `softmax([50, 45, 30])` gives roughly `[99%, 1%, 0%]` - the model is overconfident, nearly certain about one token and ignoring the rest. - The model stops learning because it already thinks it knows the answer with certainty.

After dividing by `sqrt(32) = 5.66`: - `softmax([8.8, 7.9, 5.3])` gives roughly `[70%, 28%, 2%]` - still prefers one token but spreads attention across others. - Now the model can keep learning from getting the distribution wrong.

This division is a simple numerical trick to keep the model learning well. It is called "scaled dot-product attention."

6.1.5 The Causal Mask: Visualized

For a sequence of 5 tokens, the mask looks like this:

```
        look at:  0   1   2   3   4
token 0:          OK  -   -   -   -    (can only see itself)
token 1:          OK  OK  -   -   -    (can see 0 and 1)
token 2:          OK  OK  OK  -   -
token 3:          OK  OK  OK  OK  -
token 4:          OK  OK  OK  OK  OK   (can see all previous)
```

`-` means the score is set to `-inf` before softmax, so the weight becomes 0.

Why this specific lower-triangular pattern? It mirrors how you read: when you are at word 0, you have only read word 0. When you are at word 4, you have read words 0 through 4. Token 4 can see tokens 0-4. Token 0 can only see itself.

This lower-triangular mask is what makes attention **causal**, each token can only see its past, not its future.

6.2 Code

> ***File**: `src/ch05_self_attention.py` **Run it**: `python src/ch05_self_attention.py`*

This file implements `SingleHeadAttention` as a standalone `nn.Module` with all 5 steps visible in the `forward()` method. Each line has a shape comment.

6.2.1 Sample output

```
Token 5 attends to tokens 0..5 (future tokens masked):
  token 0: 0.149  ####
  token 1: 0.152  ####
  token 2: 0.146  ####
  token 3: 0.175  #####
  token 4: 0.209  ######
  token 5: 0.168  #####
  token 6: 0.000
  token 7: 0.000
  token 8: 0.000
  token 9: 0.000
```

Notice: tokens 6-9 have weight 0.000, the causal mask is working.

6.3 Key Takeaways

- Self-attention lets each token look at all other tokens and decide what's relevant.
- Q = "what I'm looking for", K = "what I offer", V = "my content".
- Scores = Q x K^T, scaled by sqrt(head_size), then softmax gives weights.
- The causal mask prevents tokens from seeing the future.
- Output = weighted sum of V vectors.

Chapter 7

Multi-Head Attention

> ***Code file:*** *src/ch06_multihead_attention.py* ***Run it:*** *python src/ch06_multihead_attention.py*

7.1 Theory

7.1.1 One Reader Isn't Enough

In Chapter 5, we built one attention head, one "reader" that scans the text and decides what's important.

But when you read a sentence, you're tracking multiple things at once: - *Who* is the subject? - *What action* is happening? - *When* is this taking place? - *What's the tone*, serious? sarcastic?

A single attention head can only "look" for one kind of pattern at a time. **Multi-head attention** fixes this by running N heads in parallel, each potentially learning to track a different aspect of the text.

Think of it as having a team of readers, each highlighting different things, then combining all their notes.

7.1.2 How It Works

Multi-head attention is actually very simple conceptually:

1. Run N independent `SingleHeadAttention` modules on the same input `x`.
2. Each head produces an output of shape `(B, T, head_size)`.
3. Concatenate all outputs along the last dimension: `(B, T, N * head_size)`.
4. Apply a final linear projection to get back to shape `(B, T, C)`.

```
# For each of the 4 heads, run it on input x and collect all 4 outputs into a list.
# This runs all heads on the same data, giving 4 different "perspectives".
head_outputs = [head(x) for head in self.heads]      # N x (B, T, head_size)
concatenated = torch.cat(head_outputs, dim=-1)        # (B, T, N * head_size)
output       = self.projection(concatenated)          # (B, T, C)
```

In our model: N=4 heads, head_size=32, so `4 × 32 = 128 = C`. The output is the same shape as the input, perfect.

7.1.3 Do the Heads Really Learn Different Things?

Yes - by architectural diversity. Each head has its own independent Q, K, V weight matrices, so they start from different random initializations and naturally learn different patterns.

Here is the intuition: all 4 heads see the same input text, but each starts with different random weights and makes different guesses during early training. As training adjusts weights to reduce loss, each head finds a different "angle" that helps. It is like a group study session where everyone reads the same chapter but each person ends up focusing on different details - one tracks the main argument, another notices examples, a third follows the logic structure.

In well-trained large models, researchers have found that different attention heads specialize: - Some heads track subject-verb agreement across long distances. - Some focus on local context (neighboring words). - Some track coreference ("it" to "the trophy").

Our small model won't show this clearly. With only 4 heads sharing 128 dimensions, there is not much room for specialization. Larger models with 96 heads and 12,288 embedding dimensions show this dramatically - researchers have identified heads that reliably track specific grammatical relationships. But the underlying mechanism is identical to what we built.

7.1.4 The Output Projection

After concatenating the heads, we apply one final linear layer:

```
self.proj = nn.Linear(n_embd, n_embd)
```

Why? Two reasons: 1. It combines the heads intelligently. After concatenation, we have 4 x 32 = 128 numbers from 4 different heads. But the heads might have redundant or conflicting information. The projection is a (128 x 128) weight matrix that learns: "for this kind of token, head 1's insight is most important, head 3 is redundant, downweight it." It blends the heads based on what actually helps. 2. It gives the model extra capacity to transform the combined multi-head representation before passing it on.

7.2 Code

> ***File:*** *src/ch06_multihead_attention.py* ***Run it:*** *python src/ch06_multihead_attention.py*

This file imports `SingleHeadAttention` from Chapter 5 and wraps N of them in a `MultiHeadAttention` module.

7.2.1 Key output to notice

```
Multi-head output has 4x more channels  -  it sees 4 perspectives at once.
```

The output shape is still `(B, T, C)`, same as the input. Multi-head attention doesn't change the shape, it just enriches the content.

7.2.2 Parameter count

```
MultiHeadAttention total parameters: 65,664
  From 4 heads: 49,152
  From output proj : 16,512
```

About 50K parameters just for the attention layer, and that's per transformer block!

7.3 Key Takeaways

- Multi-head attention = N single-head attention modules running in parallel.
- Each head can specialize in detecting different patterns.
- Outputs are concatenated, then projected back to the original shape.
- Output shape: `(B, T, C)`, same as input.
- 4 heads $\times$ 32 head_size = 128 = n_embd. It all fits together.

Chapter 8

Feed-Forward Layers and Layer Normalization

> ***Code file:*** *src/ch07_feedforward.py* ***Run it:*** *python src/ch07_feedforward.py*

8.1 Theory

8.1.1 After Attention, We Need to "Think"

Attention lets each token gather information from its neighbors: *"Who said what, and how does it relate to me?"*

But gathering information is only half the job. The token still needs to **process** that information and decide what to do with it.

The **feed-forward layer** does that processing. It's applied independently to each token's representation and gives the model capacity to transform what it learned through attention.

Analogy: Attention is listening to your classmates explain their solutions. Feed-forward is going home and thinking it over yourself.

8.1.2 Feed-Forward Network Architecture

The feed-forward layer is a small two-layer neural network:

```
x → Linear(C → 4C) → GELU → Linear(4C → C)
```

In our model (C = 128):

```
x → Linear(128 → 512) → GELU → Linear(512 → 128)
```

Why 4x? The 4x expansion gives the model a wide workspace. Think of it like a drafting table: too narrow and you can't spread out your work. The network expands to think, then compresses back to an answer. The original Transformer paper used this ratio, and it has proven to be the sweet spot across many model sizes ever since.

Important: this is applied to *each token independently*. Token 5 goes through its own copy of the feed-forward network. Token 6 goes through an identical copy. They don't interact here, interaction only happens in attention.

8.1.3 GELU Activation

Between the two linear layers, we apply **GELU** (Gaussian Error Linear Unit).

Think of it as ReLU with a smoother edge. ReLU is a hard on/off switch: - Negative input? Output 0. Done. - Positive input? Output as-is.

GELU is a gradual dimmer: - Strongly negative input? Output nearly 0. - Slightly negative input? Output reduced, but not to zero. - Positive input? Output passes through, almost unchanged.

In practice, you just need to know: - Strongly negative values: nearly 0 - Values near zero: gently reduced - Positive values: pass through

Why smooth matters: during training, we use calculus to figure out how much to adjust each weight. When the activation function has a sharp corner (like ReLU's sudden kink at zero), the calculus produces sudden jumps that can destabilize training. GELU's smooth curve makes the math nicer and training more stable. GPT-2 and most modern transformers use GELU instead of ReLU.

8.1.4 Layer Normalization

One more ingredient: **Layer Normalization** (LayerNorm).

During training, numbers flowing through many stacked layers can grow very large or shrink very small - like compound interest, small multiplications add up over 4 layers. Very large numbers cause the weight-update math to blow up (exploding gradients). Near-zero numbers cause the update signals to disappear (vanishing gradients). Both problems make training unstable or impossible.

LayerNorm fixes this by "re-centering" and "re-scaling" the values at each token position:

```
LayerNorm(x) = (x - mean) / std × gamma + beta
```

Where `gamma` and `beta` are learned parameters that let the model scale and shift after normalization.

Result: After LayerNorm, the values at each position have mean 0 and standard deviation 1.

Think of it like: every time a runner finishes a lap, you reset their stopwatch to 0. It doesn't change who's winning, it just keeps the numbers manageable.

By keeping values near mean=0 and std=1, gradients stay in a stable range as they flow backward through the layers. Without this, signals can either amplify exponentially (exploding gradients) or shrink toward zero (vanishing gradients) as they travel through 4+ stacked blocks.

8.1.5 Where Does LayerNorm Go?

There are two conventions:

Post-norm (original Transformer, 2017): Apply LayerNorm *after* attention and feed-forward.

Pre-norm (GPT-2 and later): Apply LayerNorm *before* each sub-layer.

We use **pre-norm** because it's more stable during training. Pre-norm: normalize first, then do the hard work - like stretching before exercise. Post-norm: do the hard work, then normalize. Pre-norm is more stable because attention and feed-forward receive well-behaved, normalized inputs rather than potentially messy ones.

In Chapter 8, you'll see:

```
x = x + attention(LayerNorm(x))     # normalize BEFORE attention
x = x + feedforward(LayerNorm(x)) # normalize BEFORE feedforward
```

8.2 Code

> ***File**: src/ch07_feedforward.py **Run it**: python src/ch07_feedforward.py*

This file: 1. Implements `FeedForward` as a standalone `nn.Module` 2. Demonstrates GELU vs ReLU comparison 3. Demonstrates what LayerNorm does to values

8.2.1 Output to notice

```
Before LayerNorm: mean=4.63, std=9.57
After  LayerNorm: mean=0.0000, std=1.0039
```

The values before normalization were all over the place. After: tightly controlled at mean=0, std 1. This is what we want during training.

8.3 Key Takeaways

- Feed-forward = two-layer MLP applied independently to each token.
- Architecture: `Linear(C → 4C) → GELU → Linear(4C → C)`.
- GELU is a smooth version of ReLU, better for training.
- LayerNorm re-centers and re-scales values to keep training stable.
- We use pre-norm: LayerNorm is applied *before* each sub-layer.

Part IV

The Transformer

Chapter 9

The Transformer Block

> ***Code file:*** *src/ch08_transformer_block.py* ***Run it:*** *python src/ch08_transformer_block.py*

9.1 Theory

9.1.1 Putting the Pieces Together

We now have all the ingredients: - Multi-head attention (Chapter 6), tokens talk to each other - Feed-forward layer (Chapter 7), tokens think for themselves - Layer normalization (Chapter 7), keeps training stable

One **transformer block** combines all of these into a single reusable unit. We'll stack multiple copies of this block to build the full model.

9.1.2 The Transformer Block: Forward Pass

The full forward pass of one block is just two lines of code:

```
x = x + attention(LayerNorm(x))     # "listen to others, then add it to what I know"
x = x + feedforward(LayerNorm(x))   # "think about it, then update what I know"
```

That's it. Let's unpack each piece.

9.1.3 Trick 1: Residual Connections

Notice the `x + ...` pattern. Instead of:

```
x = attention(x)   # REPLACE x with attention's output
```

We do:

```
x = x + attention(x)   # ADD attention's output to x
```

This is called a **residual connection** (or skip connection).

Why does it matter? Imagine you're in a game of telephone. Each person translates the message and passes it on. After 10 rounds, the original message is usually unrecognizable.

Residual connections are like also passing the original message separately alongside the game of telephone. Even if the translated message gets mangled, the original is still there.

In practice: gradients flow more easily during training. Deep networks without residual connections are very hard to train; with them, you can stack dozens of layers.

Here is why it matters for training: during training, "feedback" signals travel backward through all layers to adjust weights. Without shortcuts, this feedback must pass through every transformation layer in sequence - like a chain of telephone. If each layer distorts the signal slightly, after 4+ layers the signal either grows too large (exploding gradients) or shrinks to nearly nothing (vanishing gradients). Either way, the model can't learn effectively from deep layers.

With residual connections, the `+x` shortcut provides a direct highway for the feedback signal to bypass each block entirely. Even if a block's transformations get messy, the original signal travels home cleanly. This is why deep networks (many layers) became practical only after residual connections were invented.

9.1.4 Trick 2: Pre-Layer Norm

We apply LayerNorm *before* each sub-layer (the "pre-norm" style):

```
x  ->  LayerNorm  ->  Attention  ->  + x  ->  LayerNorm  ->  FFN  ->  + x
```

This is the modern convention (used in GPT-2 and all later models). It stabilizes training, especially in the early stages.

9.1.5 Stacking Blocks

One transformer block isn't enough. We stack `n_layers = 4` of them in sequence:

```
# Create a list of 4 transformer blocks, then chain them so data flows 1 -> 2 -> 3 -> 4.
# nn.Sequential chains modules in order.
# The * unpacks the list of 4 blocks into 4 separate arguments.
blocks = nn.Sequential(*[TransformerBlock() for _ in range(4)])
```

Each block refines the token representations further. Think of it like reading a book multiple times with a different question each pass. First pass: 'what is happening?' (who are the characters?). Second pass: 'why is it happening?' (motivations). Third pass: 'what does it mean?' (themes). Earlier transformer blocks tend to capture simpler patterns (which characters are nearby?), later blocks capture more abstract ones (what does this scene mean?).

In large, well-studied models, researchers have confirmed this pattern: early layers encode syntax and local context, later layers encode semantics and long-range relationships. Our small model is too tiny to show this clearly, but the mechanism is identical.

This is why deeper models (more layers) generally perform better.

9.1.6 Parameter Count

Each block has ~197,888 parameters. With 4 blocks: ~791,552 parameters, about 96% of our entire model. The attention mechanism and feed-forward layer are where most of the "knowledge" lives.

9.2 Code

> ***File**: src/ch08_transformer_block.py **Run it**: python src/ch08_transformer_block.py*

This file: 1. Implements `TransformerBlock` using the two-line forward pass 2. Shows the parameter breakdown 3. Demonstrates the residual connection 4. Stacks 4 blocks and confirms shapes are unchanged

9.2.1 Key thing to notice

```
Input  shape: torch.Size([2, 10, 128])
Output shape: torch.Size([2, 10, 128])   (same as input)
```

The transformer block is **shape-preserving**. Input and output have identical shapes. This is what allows us to stack them: the output of block 1 feeds directly into block 2.

9.3 Key Takeaways

- A transformer block = LayerNorm + Attention + Residual + LayerNorm + FFN + Residual.
- Residual connections add the input to the output: `x = x + sublayer(x)`.
- Pre-norm: LayerNorm is applied before each sub-layer.
- Blocks are shape-preserving: output shape = input shape.
- We stack 4 blocks; each refines the token representations further.

Chapter 10

The Full GPT Architecture

> ***Code file:*** *src/ch09_gpt_model.py* ***Run it:*** *python src/ch09_gpt_model.py*

10.1 Theory

10.1.1 The Final Assembly

We now have all the components. The full GPT model is just:

```
Input token IDs
     |
Token Embedding  +  Position Embedding
     |
Transformer Block 1
     |
Transformer Block 2
     |
Transformer Block 3
     |
Transformer Block 4
     |
Final LayerNorm
     |
Linear Layer (LM Head)
     |
Output logits (one score per vocabulary token)
```

Let's trace through a concrete example.

10.1.2 Tracing Through the Model

Suppose our input is the sequence `"ROMEO"` (5 characters = 5 tokens):

Input token IDs: `[18, 21, 13, 17, 21]` Shape: `(1, 5)`, one sequence, 5 tokens

After token + position embedding: Shape: `(1, 5, 128)`, each token now has 128 numbers representing it

After 4 transformer blocks: Shape: `(1, 5, 128)`, same shape, but the values now encode rich contextual meaning - Token 4 (`O`) now "knows" it's the last letter of "ROMEO", who appears in a play

After final LayerNorm: Shape: `(1, 5, 128)`, normalized, same shape

After LM head (Linear layer): Shape: `(1, 5, 65)`, 65 scores per position (one score per possible character)

Each position makes its own independent prediction, based only on what it has seen so far (thanks to the causal mask): - Position 0 (`R`): predicts the next character after `R` - Position 1 (`O`): predicts the next character after `RO` - Position 2 (`M`): predicts the next character after `ROM` - Position 3 (`E`): predicts the next character after `ROME` - Position 4 (`O`): predicts the next character after the full `ROMEO`

We get 5 independent training targets from a single forward pass. The highest score at position 4 = the model's best guess for what follows "ROMEO".

10.1.3 The LM Head

The final linear layer (called the "Language Model head" or LM head) maps from `n_embd = 128` to `vocab_size = 65`:

```
self.lm_head = nn.Linear(n_embd, vocab_size, bias=False)
```

These 65 output scores are called **logits**. Logit is just a technical term for "raw score" - these numbers can be negative or very large, and they do not add up to 1. They are not probabilities yet.

For text generation (Chapter 14), we convert logits to probabilities using softmax, then sample a character.

For training, we use cross-entropy loss directly on the logits. Why skip the softmax during training? Applying log() and softmax together (which cross-entropy does internally) is more numerically stable than converting to probabilities first and then taking log(). The math is equivalent but the computer arithmetic is safer.

10.1.4 Total Parameter Count

```
Token embedding     :     8,320  (65 × 128)
Position embedding  :    16,384  (128 × 128)
4 Transformer blocks:   791,552  (4 × 197,888)
Final LayerNorm     :       256  (128 × 2 for gamma and beta)
LM Head             :     8,320  (128 × 65)

TOTAL               :   824,832  (~825K)
```

A note on the position embedding shape (128 x 128): we have 128 possible positions (our max sequence length, called `block_size`), and each position embedding is 128 numbers long (same as the token embedding dimension). So 128 positions x 128 numbers per position = 16,384 parameters. Each position gets its own unique 128-number fingerprint.

For comparison: - GPT-2 Small: 117,000,000 parameters - Our model: 824,832 parameters (~142x smaller than GPT-2 Small) - Same architecture, just fewer layers and narrower dimensions.

10.1.5 Why No Bias in the LM Head?

```
nn.Linear(n_embd, vocab_size, bias=False)
```

Omitting the bias follows GPT-2 convention. In practice, a bias in the final layer provides no measurable quality improvement. A `nn.Linear` layer normally has two learnable components: weights (the transformation matrix) and a bias (an offset added to every output). Setting `bias=False` means we use only the weights - just pure matrix multiplication, no offset. This is a GPT-2 design choice that works well in practice.

10.2 Code

> ***File:*** *src/ch09_gpt_model.py* ***Run it:*** *python src/ch09_gpt_model.py*

This file: 1. Implements the complete `GPT` class 2. Prints a detailed parameter breakdown 3. Runs a forward pass and explains the output

10.2.1 Key output

```
TOTAL : 824,832
```

```
Input  shape: torch.Size([2, 10])   (B, T)
Output shape: torch.Size([2, 10, 65])  (B, T, vocab_size)
```

The model is complete! The output logits at each position predict the next token.

10.3 Key Takeaways

- The GPT model = Embeddings → N Transformer Blocks → LayerNorm → LM Head.
- Input shape: `(B, T)` token IDs. Output shape: `(B, T, vocab_size)` logits.
- Total parameters: ~825K (tiny compared to production models, same architecture).
- Logits at position `t` predict what token comes at position `t+1`.

Chapter 11

Causal Language Modeling

> ***Code file:*** *src/ch10_causal_lm.py* ***Run it:*** *python src/ch10_causal_lm.py*

11.1 Theory

11.1.1 The Training Trick: One Forward Pass = Many Examples

How do we train a model to predict the next token?

Naively, you might think: take one token, predict the next, check if you're right, update the weights. Then repeat for the next token.

That's 1 million training examples done one at a time. Slow.

The brilliant insight: **a single forward pass on a sequence of length T gives us T training examples at once!**

Here's why. Given the sequence `"HELLO"`:

Input context	Correct next token
`H`	`E`
`HE`	`L`
`HEL`	`L`
`HELL`	`O`

The model processes the whole sequence at once, and at each position, it predicts the next token. We get 4 training examples from one 4-token sequence.

The causal mask (from Chapter 5) ensures position 2 (`HEL`) can't "cheat" by seeing position 3 (`O`). Each position only sees what came before.

11.1.2 Input and Target: The Shifted Pair

In code, this is implemented by using the *same sequence* twice, shifted by one:

```
x = sequence[:-1]   # all but the last token  → "HELL"
y = sequence[1:]    # all but the first token → "ELLO"
```

At each position `i`, `x[i]` is the input and `y[i]` is the correct next token.

11.1.3 The Loss Function: Cross-Entropy

Once we have predictions (logits) and correct answers (targets), we measure how wrong we are using **cross-entropy loss**.

Intuitively: for each position, the model predicts a probability distribution over all 65 characters. Cross-entropy measures how much probability the model assigned to the *correct* character.

- If the model predicts 90% probability for the correct character: low loss (good!)
- If the model predicts 1% probability for the correct character: high loss (bad!)

For a random model (no training), the expected loss is `log(65)` `4.17`. Here is where that number comes from: with 65 characters and no knowledge, the model assigns equal probability to each (1/65 1.5%). Cross-entropy loss for a uniform distribution over 65 choices equals log(65) 4.17. Think of it as the "complete ignorance" baseline - this is the worst possible score a model can get while still trying. After training to 3000 steps, we expect the loss to drop to around `1.3-1.5`. At that point the model assigns roughly 4-5x higher probability to the correct character than a random guesser would. A much better model would hit `1.0` or below. For reference: a human reading Shakespeare might score around 0.8-1.0 on this exact task.

11.1.4 Autoregressive Generation

Once trained, we use the model to *generate* new text:

1. Start with a prompt: `"ROMEO:\n"` (encoded as token IDs)
2. Feed the prompt to the model → get logits at the last position
3. Convert logits to probabilities (softmax)
4. Sample one token from the probability distribution
5. Append that token to the sequence
6. Repeat from step 2 until you have enough text

Each new token is added to the context, which influences what comes next. This is why it's called **autoregressive**: "auto" means self, and "regressive" means going back to previous data. So autoregressive means the model uses its own previous outputs as inputs for the next step - like writing a sentence one word at a time, where each word you choose influences what word you write next.

11.1.5 "No Cheating": Why the Causal Mask Matters

Without the causal mask, at training position `t` the model could peek at the input token at position `t+1` and trivially copy it to the output. Perfect training accuracy, zero actual language understanding - the model learned to cheat, not to predict.

The mask prevents this technically: inside the attention calculation (Chapter 5), it sets attention weights to zero for all future positions. Position 2 literally cannot receive any information from positions 3, 4, 5, etc. - they are blocked, not just discouraged. The model has no choice but to predict from context.

The mask forces the model to actually learn: "Given everything I've seen so far, what's the most likely next character?"

This is the same constraint you'd impose on a student during an exam: cover up the answers and actually *think*.

11.2 Code

> ***File:*** *`src/ch10_causal_lm.py`* ***Run it:*** *`python src/ch10_causal_lm.py`*

This file: 1. Shows how input/target pairs are constructed 2. Computes cross-entropy loss on a random (untrained) model 3. Implements the `generate()` function 4. Demonstrates generation on an untrained model (random output, but correct mechanics)

11.2.1 Important output

```
Loss (random model): 4.3253
Expected loss for random: 4.1744
```

These are close, the model is essentially guessing randomly, as expected. After training, this loss will drop significantly.

11.3 Key Takeaways

- One forward pass on a T-length sequence = T training examples (huge efficiency win!).
- Input `x` and target `y` are the same sequence, shifted by one position.
- Cross-entropy loss measures how wrong the predictions are. Lower = better.
- Random model loss log(vocab_size) = log(65) 4.17.
- Generation = repeatedly predicting and appending the next token.

Part V

Training

Chapter 12

Dataset and DataLoader

> ***Code file:*** *src/ch11_dataloader.py* ***Run it:*** *python src/ch11_dataloader.py*

12.1 Theory

12.1.1 The Dishwasher Problem

Imagine washing 1 million dishes. You could wash them one at a time, but that's slow. Or you could load the dishwasher with 32 at a time, much faster, because the machine handles them in parallel.

Training a neural network is the same. Instead of processing one example at a time, we process a **batch** of 32 examples simultaneously. PyTorch runs all 32 through the model at once using parallelized matrix operations.

This is why our model works on tensors of shape `(B, T, C)`, the `B` dimension is the batch size.

12.1.2 What Is a Training Example?

For our language model, one training example = one text window: - **Input** `x`: a sequence of `block_size = 128` token IDs - **Target** `y`: the same sequence, shifted right by 1

```
Text  : "To be or not to be, that is the question"
x     : "To be or not to be, that is the questio"   ← tokens 0..127
y     : "o be or not to be, that is the question"   ← tokens 1..128
```

At each of the 128 positions, `x[i]` is the input context and `y[i]` is what we're trying to predict.

12.1.3 Sliding Windows

We create training examples by sliding a window of size `block_size` across the full text:

```
Position 0:   "To be or not to be..."  (chars 0-127)
Position 1:   "o be or not to be,..."  (chars 1-128)
Position 2:   " be or not to be, t..."  (chars 2-129)
...
```

This gives us over **1 million** overlapping training examples from the ~1.1 million characters of Shakespeare. There's plenty of data!

12.1.4 The PyTorch Dataset and DataLoader

PyTorch provides two abstractions:

`Dataset`: An object that knows how many examples exist and can return any one of them by index.

The two methods below use Python's special "dunder" (double underscore) naming: `__len__` and `__getitem__`. Python has dozens of these special methods - they are automatically called behind the scenes when you use built-in operations: - `len(my_dataset)` automatically calls `my_dataset.__len__()` - `my_dataset[42]` automatically calls `my_dataset.__getitem__(42)`

By defining these two methods, our `TextDataset` class works with all Python and PyTorch code that expects a sequence-like object.

```python
class TextDataset(Dataset):
    def __len__(self):
        return len(self.data) - self.block_size   # ~1 million examples
    def __getitem__(self, idx):
        x = self.data[idx   : idx + block_size]
        y = self.data[idx+1 : idx + block_size + 1]
        return x, y
```

`DataLoader`: Wraps a Dataset and automatically: - Groups examples into batches of size `batch_size = 32` - Shuffles the data each epoch (one epoch = one full pass through all training examples; after seeing every example once, we shuffle and start again) - Handles edge cases

```python
loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Each call to `next(iter(loader))` gives us one batch: `(x, y)` of shape `(32, 128)`.

Note: these are raw token IDs, so the shape is `(B, T)` not `(B, T, C)`. When the model receives this batch in Chapter 12, the embedding layer is the first thing it runs - that is what expands the

shape from `(B, T)` to `(B, T, C)`. Here is what that expansion means: we start with (32, 128) - 32 sequences of 128 token IDs, each token represented as a single integer. The embedding layer replaces each integer with a full 128-dimensional vector (learned in Chapter 4). Now each token is described by 128 numbers instead of 1. Result: shape (32, 128, 128) = 32 sequences, 128 tokens each, each token a 128-number vector.

12.1.5 Why Shuffle?

If examples always appear in the same order, the model might learn "character X tends to follow Y in this dataset" based on the dataset's sequential structure rather than actual language patterns. Shuffling forces the model to rely only on genuine patterns within each 128-token window, not on where that window happens to sit in the file.

12.2 Code

> ***File:*** *src/ch11_dataloader.py* ***Run it:*** *python src/ch11_dataloader.py*

This file: 1. Loads and tokenizes Shakespeare 2. Implements `TextDataset` 3. Creates train and validation `DataLoader`s 4. Inspects one batch and decodes it back to text

12.2.1 Sample output

```
x (input)  : "d the time seems thirty unto me,\nBeing all this time aband..."
y (target) : " the time seems thirty unto me,\nBeing all this time abando..."
(y is x shifted by 1 character)
```

Notice: `y` is just `x` with the first character removed and one new character appended at the end.

12.3 Key Takeaways

- A training batch = 32 text windows, each 128 tokens long.
- Input `x` and target `y` are the same window, shifted by 1.
- `Dataset` defines how to get one example; `DataLoader` batches and shuffles.
- Shuffling prevents the model from memorizing order.

- ~1 million training examples from 1 million characters of Shakespeare.

Chapter 13

The Training Loop

> ***Code file:*** *src/ch12_train.py* ***Run it:*** *python src/ch12_train.py* ***Expected time:*** *~20-30 minutes on CPU*

13.1 Theory

13.1.1 The Most Important Four Lines in Machine Learning

The training loop reduces to four operations, repeated thousands of times:

```
logits = model(x)              # 1. Forward pass: make predictions
loss   = cross_entropy(logits, y)  # 2. Compute loss: how wrong are we?
loss.backward()                    # 3. Backward pass: who's responsible?
optimizer.step()                   # 4. Update: fix the responsible parameters
```

That's it. Everything else is just bookkeeping.

13.1.2 Step 1: Forward Pass

We feed a batch of 32 text windows through the model. For each of the 32 sequences, at each of the 128 positions, the model outputs 65 scores (one per character). The overall logits shape is `(32, 128, 65)`.

13.1.3 Step 2: Compute Loss

Cross-entropy loss compares our predictions to the correct answers. Lower loss = better predictions.

On the first step, loss 4.17 (random guessing). After 3000 steps, loss drops to around 1.3-1.5. The model went from "no idea" to "pretty decent at Shakespeare."

13.1.4 Step 3: Backward Pass (Backpropagation)

`loss.backward()` computes the **gradient** of the loss with respect to every parameter in the model.

Here is how it works: during the forward pass, PyTorch secretly records every mathematical operation it performs (multiply, add, softmax, etc.). When you call `loss.backward()`, PyTorch traces backward through all those recorded steps and figures out: "If I had changed this weight by a tiny amount, how would the loss have changed?"

A gradient answers exactly that question for every single weight in the model: - Positive gradient: increasing this weight makes loss go up, so we should decrease it. - Negative gradient: increasing this weight makes loss go down, so we should increase it. - Big gradient: this weight has a strong effect on loss, adjust it more. - Small gradient: this weight barely matters right now.

Think of it like: you're lost in a hilly landscape (loss landscape) and you want to find the lowest valley. The gradient tells you which direction is downhill from where you're standing.

Important: Before each backward pass, we call `optimizer.zero_grad()`. Here is why: PyTorch does not replace gradient values when you call `backward()` - it *adds* the new gradients on top of whatever was already there. So if you skip `zero_grad()`, you accumulate gradients from batch 1, batch 2, batch 3 all added together, which is nonsense. You want only fresh gradients from this batch. Zero it first, then compute fresh.

Note: `optimizer.zero_grad()` must come *before* `loss.backward()`, not after.

13.1.5 Step 4: Optimizer Step (Adam)

`optimizer.step()` uses the gradients to update all parameters:

```
new_value = old_value - learning_rate × gradient
```

This is called **gradient descent**. The learning rate (`3e-4 = 0.0003`) controls how big each step is.

We use the **Adam optimizer**, which is smarter than basic gradient descent.

Basic gradient descent takes the same step size for every weight, every time. Adam is smarter: it keeps a memory of the last many gradient updates for each weight and uses that history to choose a better step size. Weights that keep getting large gradients get smaller steps (they are in steep

terrain, be careful). Weights that get small or inconsistent gradients get bigger steps (they need more nudging). It is like a smart hiker who adjusts stride length based on how the terrain has been - shorter strides on steep cliffs, longer strides on flat ground.

Adam usually converges faster and more reliably than plain gradient descent.

13.1.6 The Validation Loss

Every 300 steps, we compute the loss on the *validation set* (text the model hasn't trained on). This is our honest measure of how well the model has learned.

If training loss drops but validation loss doesn't: the model is **overfitting** (memorizing, not learning). For example: training loss 1.3, validation loss 2.5 means the model has memorized specific sequences from the training text but cannot apply what it learned to new text it has never seen. It learned the answers, not the concept.

If both drop together: the model is genuinely learning patterns that work on new text too.

13.1.7 What to Expect

```
step     1 | train loss: 4.2827 | val loss: 4.2xxx
step   300 | train loss: 2.xxxx | val loss: 2.xxxx
step   600 | train loss: 2.xxxx | val loss: 2.xxxx
...
step  3000 | train loss: 1.3xxx | val loss: 1.5xxx
```

Why does loss start near 4.17? On step 0, the model has learned nothing - it is essentially guessing at random among 65 possible characters. With equal probability for all 65 characters, the cross-entropy loss formula gives $\log(65) = 4.17$. Think of it as the "complete ignorance" baseline.

Loss drops fastest in the early steps because there is a lot of low-hanging fruit - the model quickly learns obvious patterns like 'spaces are common' and 'e follows many letters'. As training progresses, only harder patterns remain, and each step yields smaller gains. The model approaches the limits of what 825K parameters trained on 1MB of text can learn.

The validation loss being slightly higher than training loss is normal and expected. The model has seen training examples many times and adapted to them specifically. Validation examples are fresh - like the difference between a familiar practice test and a surprise quiz on the same material.

13.2 Code

> ***File**: `src/ch12_train.py` **Run it**: `python src/ch12_train.py`*

This file is the most important in the project. Read through it carefully, every line has a comment explaining its purpose.

Key things to notice: - `optimizer.zero_grad()` comes *before* `loss.backward()`. - We call `model.eval()` before validation (disables dropout) and `model.train()` after. - A checkpoint is saved at the end.

13.2.1 After training

Once training completes, you'll have: - A checkpoint file at `checkpoints/model.pt` - Printed loss values showing the model improving - A model ready for generation in Chapters 14-16!

13.3 Key Takeaways

- Training loop = forward pass → loss → backward pass → optimizer step. Repeat.
- Gradient = "which direction should we adjust each parameter?"
- Adam optimizer adapts the step size for each parameter automatically.
- Validation loss = honest measure of generalization (not memorization).
- ~20-30 min on a CPU-only laptop for 3000 steps.

Chapter 14

Saving and Loading Checkpoints

> ***Code file:*** *src/ch13_checkpoint.py* ***Run it:*** *python src/ch13_checkpoint.py*

14.1 Theory

14.1.1 Why Checkpoints?

Training on a CPU takes 20-30 minutes. If your laptop crashes, runs out of battery, or you accidentally close the terminal, without checkpoints, you'd lose all that work and start from scratch.

More importantly: once you have a trained model, you want to be able to load it later without retraining. That's what checkpoints are for.

A checkpoint = the model's "save file."

14.1.2 What Is a Checkpoint?

A checkpoint is just a dictionary saved to disk using `torch.save()`:

```
checkpoint = {
    "model_state": model.state_dict(),   # all the learned weights
    "gpt_cfg"    : config,               # model architecture details
    "step"       : step,                 # how far training had gone
    "val_loss"   : val_loss,             # performance at that point
}
torch.save(checkpoint, "checkpoints/model.pt")
```

`model.state_dict()` is a dictionary of all the model's parameters - every number the model has adjusted during training. This includes: all attention weight matrices (W_q, W_k, W_v for

every head in every block), all feed-forward layer weights, all biases, and all layer normalization parameters (gamma and beta). It does NOT include the architecture itself (number of layers, embedding size, etc.) - that is stored separately in `gpt_cfg`. You need both to fully restore a model: the config tells PyTorch how to build the skeleton, and the state_dict fills in all the learned values.

File size: our checkpoint is about **4 MB**. Smaller than a single photo on your phone. Your model's entire brain fits in a thumbnail.

14.1.3 Loading a Checkpoint

To load and use a saved model:

```
# Load the dictionary from disk
checkpoint = torch.load("checkpoints/model.pt", map_location="cpu")

# Rebuild the model architecture (empty, no weights yet)
config = checkpoint["gpt_cfg"]
model  = GPT(config)

# Fill in the saved weights
model.load_state_dict(checkpoint["model_state"])

# Switch to inference mode (disables dropout)
model.eval()
```

It's like loading a save file in a video game: the game loads the world (architecture) and then restores your character's state (weights).

14.1.4 `model.eval()` vs `model.train()`

Two important mode switches:

`model.train()`: Enables dropout (randomly zeroes out ~10% of activations during training). During each forward pass, random neurons get switched off. This forces the model to not rely on any specific neuron always being present - if neuron 42 might disappear, the model has to distribute knowledge across many neurons. This makes the learned features more robust and helps prevent overfitting. Dropout is on by default.

`model.eval()`: Disables dropout. All neurons are active. Use this when generating text or evaluating - you want deterministic, full-power outputs.

You can see both in the training loop from Chapter 12: `model.eval()` before computing validation loss, `model.train()` after.

Always switch to `eval()` mode before generation!

14.1.5 Resume Training (Optional)

You can also resume training from a checkpoint:

```
checkpoint = torch.load("checkpoints/model.pt")
model.load_state_dict(checkpoint["model_state"])
start_step = checkpoint["step"]
# continue training from start_step...
```

For a 3000-step run, this usually isn't needed. But for longer runs on larger models, it's essential.

14.2 Code

> ***File:*** *src/ch13_checkpoint.py* ***Run it:*** *python src/ch13_checkpoint.py*

This file: 1. Checks if a checkpoint exists (creates a dummy one if not) 2. Loads the checkpoint 3. Rebuilds the model 4. Verifies it works with a forward pass 5. Reports the file size

---

14.3 Key Takeaways

- A checkpoint = the model's learned weights + metadata, saved to disk.
- `torch.save()` saves; `torch.load()` + `model.load_state_dict()` restores.
- Always call `model.eval()` before generating text.
- Our checkpoint is ~4 MB, tiny!
- Checkpoints let you use a trained model without retraining.

Part VI

Generation

Chapter 15

Greedy Decoding and Sampling

> ***Code file:*** *src/ch14_generate_greedy.py* ***Run it:*** *python src/ch14_generate_greedy.py*
> (Requires a trained checkpoint from `ch12_train.py`)

15.1 Theory

15.1.1 From Logits to Text

Our model produces **logits** - 65 raw scores, one per character. (Recall from Chapter 9: logits are the unnormalized scores from the LM head, before any softmax.) To pick the next character, we need a strategy.

We have two basic strategies:

15.1.2 Strategy 1: Greedy Decoding

Always pick the character with the **highest score**.

```
next_token = logits.argmax()    # argmax = "index of the maximum value"
```

Pros: Simple, fast, deterministic (always produces the same output from the same prompt).

Cons: Repetitive. Once the model outputs a repetitive pattern, that pattern becomes the context for the next prediction. The same context creates nearly identical logits, which again picks the same token.

Here is a concrete example of how the loop forms. Suppose the model learned: - Given `[king, is, ]` (king + is + space), the next character is `t` - Given `[is, , t]` (is + space + t), the next character is `h` - Given `[ , t, h]` (space + t + h), the next character is `e` - … and eventually circles back to producing `king is the king is the...`

The model is not broken - it found a short repeating pattern in its training data and locked into it. This is the failure mode of greedy decoding.

Greedy decoding is like a student who always circles "C" on every multiple choice question - technically safe, usually boring, occasionally loops forever.

Example of greedy going wrong:

```
Input: "The king of England"
Greedy: "The king of England the the the the the the the..."
```

The model got stuck in a loop. This is called **repetition** and is a known failure mode of greedy decoding.

15.1.3 Strategy 2: Sampling

Instead of always picking the top token, we **sample randomly** from the probability distribution.

```
probs    = softmax(logits)
next_token = torch.multinomial(probs, num_samples=1)
```

`torch.multinomial` picks a token randomly, but tokens with higher probability are more likely to be chosen. It's like a weighted lottery: - Character "e" has 40% probability → wins the lottery 40% of the time - Character "x" has 0.1% probability → wins very rarely, but occasionally

Pros: Varied output, avoids repetition, more creative-sounding text.

Cons: Unpredictable. Sometimes picks weird tokens. Different every run (unless you set a random seed).

15.1.4 Which Is Better?

Neither is universally better, it depends on what you want:

Situation	Better Strategy
You want consistent, predictable output	Greedy
You want creative, varied output	Sampling
Filling in a specific factual answer	Greedy
Writing a poem or story	Sampling

In practice, most real applications use sampling with controls (temperature and top-k, see Chapter 15).

15.1.5 The Generation Loop

Both strategies use the same loop:

```
context = prompt_token_ids    # starting context

for _ in range(max_new_tokens):
    logits   = model(context)[:, -1, :]   # get logits at last position
    next_id  = pick_next_token(logits)    # greedy or sampling
    context  = append(context, next_id)   # grow the sequence
```

The key step: we always use the **logits at the last position** (`[:, -1, :]`). The model output shape is `(batch, sequence_length, vocab_size)`. We index `[:, -1, :]` to take the last sequence position from every batch item - that is where the model has attended to everything before it and is predicting what comes next. Positions earlier in the sequence predict what comes after *them*, not after the whole prompt.

We **crop the context** to `block_size` (128) tokens before each forward pass. Here is why: during training, the model only ever saw sequences of exactly 128 tokens. Position embeddings were learned for positions 0 through 127. If you feed the model 200 tokens, it encounters position 150, which has no embedding - the model has never seen that position during training, so the output is undefined. By always keeping the last 128 tokens (a sliding window), we stay within the model's trained range.

15.2 Code

> ***File:*** *src/ch14_generate_greedy.py* ***Run it:*** *python src/ch14_generate_greedy.py*

This file implements both `generate_greedy()` and `generate_sample()` and runs them with the same prompt for comparison.

15.2.1 If you haven't trained yet

Run `python src/ch12_train.py` first (20-30 minutes). Then come back.

With an untrained model, both strategies produce gibberish. With a trained model, you'll see a clear difference in style.

15.3 Key Takeaways

- Greedy decoding: always pick the highest-probability token. Simple but repetitive.
- Sampling: pick randomly weighted by probability. More varied but unpredictable.
- Both use the same loop: predict → pick → append → repeat.
- We always use the logits at the **last position** to predict the **next** token.

Chapter 16

Temperature and Top-k Sampling

> ***Code file:*** *src/ch15_generate_sampling.py* ***Run it:*** *python src/ch15_generate_sampling.py*
> (Requires a trained checkpoint from `ch12_train.py`)

16.1 Theory

16.1.1 Plain Sampling Has Problems Too

We saw in Chapter 14 that plain sampling (random choice weighted by probability) gives varied output. But sometimes it picks *really* bad tokens, like a comma in the middle of a word, or a rarely-seen character that breaks the grammar.

The problem: even characters with very low probability (say, 0.1%) occasionally get picked. That's one in a thousand tokens, and with 200 tokens generated, you might expect several weird ones.

We need controls.

16.1.2 Control 1: Temperature

Temperature adjusts how "peaked" or "flat" the probability distribution is before sampling.

The implementation is simple: divide the logits by a temperature value `T` before applying softmax.

```
logits = logits / temperature   # adjust before softmax
probs  = softmax(logits)
```

- **T < 1.0** (e.g., 0.5): Dividing by a small number makes logits *bigger* in magnitude. Quick math: dividing 6 by 0.5 gives 12 (dividing by a fraction multiplies). So logits [1, 2, 3] divided by 0.5 give [2, 4, 6] - the gap between highest and lowest doubled from 2 to 4. After softmax, the top token becomes overwhelmingly more likely. Output is focused and coherent, but tends toward repetition.

- **T = 1.0**: Dividing by 1 changes nothing. Normal sampling - the model's raw learned distribution.
- **T > 1.0** (e.g., 2.0): Dividing by a large number makes logits *smaller*. Logits [1, 2, 3] divided by 2.0 give [0.5, 1.0, 1.5] - the gap shrank from 2 to 1. After softmax, all tokens have more similar probabilities. Output is more random and creative, sometimes nonsensical.

Think of temperature as the "creativity dial": - Low temperature = accountant mode (safe, predictable) - High temperature = jazz musician mode (creative, unpredictable)

A sweet spot is usually between 0.7 and 1.0. To give you a feel for the range: - T=0.5: Very focused and coherent, but tends toward repetition - T=0.8: Good balance - our recommended default - T=1.0: The model's raw learned distribution, some surprising choices - T=1.5+: Gets weird fast - expect sudden topic changes and odd vocabulary

---

16.1.3 Control 2: Top-k Sampling

Top-k limits which tokens can even be considered. Before sampling, we keep only the top-k highest-scoring tokens and set all others to negative infinity (which becomes 0 after softmax).

```
k = 40
top_k_values = logits.topk(k).values
threshold    = top_k_values[:, -1, None]   # k-th largest value
logits       = logits.masked_fill(logits < threshold, float("-inf"))  # eliminated tokens get
probs        = softmax(logits)
next_token   = multinomial(probs, 1)
```

`masked_fill` is a PyTorch operation that says: "wherever this condition is True, replace the value with a specified number." Here, the condition is `logits < threshold` (score lower than the k-th best), and the replacement is `-inf` (negative infinity). Why `-inf`? Because softmax computes `e^x` for each value. `e^(-infinity) = 0`. So any token set to `-inf` gets exactly 0 probability after softmax - completely eliminated from sampling.

With `top_k = 40`: only the 40 most likely characters are in the running. We keep the top 40 and zero out the remaining 25 (out of 65).

This prevents the model from ever picking a truly bizarre token. Even if the sampling is random within the top 40, at least we know all 40 are "reasonable" choices given the context.

Common values: top_k = 40 or 50 works well. top_k = 1 = greedy decoding.

16.1.4 Combining Temperature and Top-k

Most real applications use both together:

```
1. Compute logits
2. Apply top-k (keep the top-k tokens, zero out the rest)
3. Apply temperature (adjust the spread)
4. Apply softmax (convert to probabilities)
5. Sample
```

Recommended settings: `temperature=0.8, top_k=40`

This combination gives you: - Creative and varied output (from sampling + temperature) - With a quality floor (from top-k, no bizarre picks)

16.2 Code

> ***File**: `src/ch15_generate_sampling.py` **Run it**: `python src/ch15_generate_sampling.py`*

This file generates text with 5 different configurations for direct comparison: 1. `temp=0.5, top_k=40`, focused/conservative 2. `temp=0.8, top_k=40`, balanced (recommended default) 3. `temp=1.0, top_k=40`, the model's raw learned distribution 4. `temp=1.5, top_k=40`, creative/chaotic 5. `temp=1.0, no top_k`, unrestricted sampling

After running Chapter 12's training, the differences become very obvious.

16.3 Key Takeaways

- Temperature < 1: more focused output. Temperature > 1: more random output.
- Implementation: just divide logits by temperature before softmax.
- Top-k: only keep the k highest-scoring tokens as candidates.
- Recommended sweet spot: `temperature=0.8, top_k=40`.
- Both controls are applied *before* softmax and sampling.

Chapter 17

Putting It All Together

> ***Code file:*** *src/ch16_full_pipeline.py* ***Run it:*** *python src/ch16_full_pipeline.py* ***Expected time:*** *~20-30 minutes on CPU*

17.1 Theory

17.1.1 You've Made It

Let's take a moment to look back at what we've built:

1. **Tokenizer** (Ch 3): Converts Shakespeare text to integers and back.
2. **Embeddings** (Ch 4): Maps each token to a 128-dimensional vector.
3. **Self-attention** (Ch 5): Each token gathers information from others.
4. **Multi-head attention** (Ch 6): 4 attention heads, 4 perspectives.
5. **Feed-forward layer** (Ch 7): Each token processes what it learned.
6. **Transformer block** (Ch 8): One complete unit with residual connections.
7. **GPT model** (Ch 9): 4 blocks stacked, plus embedding and LM head.
8. **Training loop** (Ch 12): Gradient descent for 3000 steps.
9. **Generation** (Ch 15): Temperature + top-k sampling.

The full pipeline runs all of these steps in sequence. This is the same architecture as GPT-2 Small (117M parameters), just about 142x smaller.

What does "same architecture" mean? GPT-2 uses the exact same building blocks you just built: character/token embeddings, self-attention with multiple heads, transformer blocks with residual connections and layer norm, and gradient descent training. The only differences are scale: GPT-2 uses 768-dimensional embeddings (we use 128), 12 transformer layers (we use 4), and it was trained on 40GB of internet text (we use 1MB of Shakespeare). The code structure is identical. If you understand what you built here, you understand GPT-2, and by extension the core of GPT-3, GPT-4, and all modern LLMs.

17.1.2 The Complete Pipeline

```
shakespeare.txt
    |
[tokenize]
    |
integer tokens  →  DataLoader (batches of 32 × 128)
    |
[model] 825K-parameter GPT
    |
[training loop] 3000 steps, Adam optimizer
    |
checkpoints/model_final.pt
    |
[generation] temperature=0.8, top_k=40
    |
Shakespeare-like text!
```

17.1.3 What "Shakespeare-like" Actually Looks Like

After 3000 training steps, the model typically produces output like:

```
ROMEO:
What is it thou dost say?
I cannot hold my peace. The king hath not
Yet made me know his pleasure.

JULIET:
Good night, good night! Parting is such sweet
sorrow that I shall say good night till it be morrow.
```

It's not perfect Shakespeare. It mixes up characters, makes grammatical mistakes, and sometimes trails off. But it: - Uses correct character name formatting - Forms mostly grammatical sentences - Uses period-appropriate vocabulary - Occasionally produces surprisingly beautiful lines

All from 825,000 numbers, trained in 20-30 minutes!

17.1.4 What Would Make It Better?

More steps and more parameters: - 10,000 steps → noticeably better - Larger model (n_embd=256, n_layers=6) → better still - GPT-2 Small (117M parameters) → impressively good

These would require a GPU to train in a reasonable time. But the *code* is essentially the same, just bigger numbers in the config.

17.1.5 What You've Actually Learned

By completing this tutorial, you now understand: - How language models work (next-token prediction) - What the Transformer architecture actually does, layer by layer - How training works (gradient descent, backpropagation, Adam) - What "parameters" are and how they're learned - How text generation works (autoregressive sampling)

The fundamental building blocks - embeddings, self-attention, transformer blocks, and gradient descent - are the same ones used to build GPT-4, Llama, Gemini, and every other modern LLM. Modern models add refinements on top, but these are optimizations, not different fundamentals: - **Rotary positional embeddings**: A better way to encode positions than the lookup table we built. Works more reliably at very long sequences. - **Grouped-query attention**: A faster version of multi-head attention that shares some weight matrices across heads. Reduces memory and compute. - **Instruction tuning**: After pretraining (which is what we did), models are fine-tuned on human-written question/answer examples to be more helpful and less likely to produce harmful content. - **RLHF (Reinforcement Learning from Human Feedback)**: Further training that uses human preference rankings to make the model's responses more helpful and aligned with what people actually want.

The foundation - transformer blocks, self-attention, gradient descent training - is exactly what you built. Everything else is refinement.

17.2 Code

> ***File**: `src/ch16_full_pipeline.py` **Run it**: `python src/ch16_full_pipeline.py`*

This is the single self-contained end-to-end script. It: - Downloads the data if not present - Tokenizes - Creates DataLoaders - Builds the model (prints param count) - Trains for 3000 steps (logs loss every 300 steps) - Saves a checkpoint - Generates 400 characters of Shakespeare-like text

You can also use it as a "smoke test" - a quick end-to-end check that nothing is broken. (The term comes from electronics: plug in a new circuit, if it doesn't literally smoke, you're probably fine.) If `ch16_full_pipeline.py` runs start-to-finish without errors, the whole tutorial is working correctly.

17.3 Key Takeaways

- The full pipeline = tokenize → embed → 4 transformer blocks → train → generate.
- 825K parameters, trained in ~20-30 minutes on CPU.
- Same architecture as GPT-2, just much smaller.
- Generation uses temperature=0.8, top_k=40 for good balance.
- The fundamentals here are the same ones used in production LLMs.

17.4 Congratulations!

You started with an empty folder and ended with a working language model.

You didn't just run someone else's code - you built every component yourself, from character-level tokenization through to text generation, and you understand why each piece is there.

You didn't just follow a recipe. You built the kitchen, designed the recipe, and cooked a meal that generates Shakespeare. Not bad for a weekend project.

That's the real achievement.