Building a Large Language Model from Scratch

Independent Study

Sundar Raj Sharma
Department of Computer Science and Information Systems
Youngstown State University
`ssharma33@student.ysu.edu`

Supervisor: Feng (George) Yu, Ph.D.
Department of Computer Science and Information Systems
Youngstown State University

IS-6996 · Spring 2026

Abstract. This document is a weekly learning log for an independent study on building a large language model (LLM) from scratch. The study follows Sebastian Raschka's *Build a Large Language Model (From Scratch)* [1] as the primary reference, supplemented by hands-on practice in Google Colab and Jupyter Notebooks. The goal is not to use pre-built libraries but to implement every component — tokenization, attention mechanisms, the transformer block, pretraining, and fine-tuning — from first principles using Python and PyTorch. This is a continuous learning effort. Each weekly entry documents what was learned, key concepts understood, code written, and challenges encountered. The cumulative result is intended to serve as both a personal knowledge record and a readable summary: someone reading this document should come away with a clear understanding of what an LLM is and how one can be built, step by step.

Spring 2026

Contents

1. References & Resources

This study draws primarily on the following textbook and supplementary materials. All hands-on coding is done in Google Colab and Jupyter Notebooks, with no reliance on high-level abstractions such as Hugging Face Transformers. The objective is to understand each component by building it.

1.1 Primary Textbook

Sebastian Raschka — *Build a Large Language Model (From Scratch)*, Manning Publications, 2024.
`https://github.com/rasbt/LLMs-from-scratch`

This book is the backbone of the study. It walks through implementing a GPT-style language model entirely from scratch in PyTorch — from raw text and tokenization all the way through pretraining and instruction fine-tuning. Each chapter maps directly to a weekly learning block.

Book Contents

- **Chapter 1 Understanding Large Language Models**
  - What is an LLM?
  - Applications of LLMs
  - Stages of building and using LLMs
  - Introducing the transformer architecture
  - Utilizing large datasets
  - A closer look at the GPT architecture
  - Building a large language model
- **Chapter 2 Working with Text Data**
  - Understanding word embeddings
  - Tokenizing text
  - Converting tokens into token IDs
  - Adding special context tokens
  - Byte pair encoding
  - Data sampling with a sliding window
  - Creating token embeddings
  - Encoding word positions
- **Chapter 3 Coding Attention Mechanisms**

  - The problem with modeling long sequences
  - Capturing data dependencies with attention mechanisms
  - Self-attention without trainable weights
  - Self-attention with trainable weights
  - Causal attention and masking
  - Multi-head attention

- **Chapter 4 Implementing a GPT Model from Scratch**
  - Coding an LLM architecture
  - Layer normalization
  - Feed-forward network with GELU activations
  - Shortcut connections
  - Transformer block
  - Coding the GPT model
  - Generating text

- **Chapter 5 Pretraining on Unlabeled Data**
  - Evaluating generative text models
  - Training an LLM
  - Decoding strategies: temperature scaling and top-k sampling
  - Loading and saving model weights in PyTorch
  - Loading pretrained weights from OpenAI

- **Chapter 6 Fine-Tuning for Classification**
  - Categories of fine-tuning
  - Preparing the dataset and data loaders
  - Initializing a model with pretrained weights
  - Adding a classification head
  - Fine-tuning on supervised data
  - Using the LLM as a spam classifier

- **Chapter 7 Fine-Tuning to Follow Instructions**
  - Introduction to instruction fine-tuning

  - Preparing a dataset for supervised instruction fine-tuning
  - Organizing data into training batches
  - Fine-tuning the LLM on instruction data
  - Evaluating the fine-tuned LLM

- **Appendices**
  - A: Introduction to PyTorch
  - B: References and further reading
  - C: Exercise solutions
  - D: Adding bells and whistles to the training loop
  - E: Parameter-efficient fine-tuning with LoRA

1.2 Supplementary Papers

Vaswani et al. — *Attention Is All You Need*, NeurIPS 2017.
`https://arxiv.org/abs/1706.03762`

The foundational paper introducing the transformer architecture. Required reading alongside Chapter 3 of the textbook.

Radford et al. — *Language Models are Unsupervised Multitask Learners*, OpenAI 2019.
`https://openai.com/blog/better-language-models`

The GPT-2 paper. Provides context for the architecture implemented in this study.

1.3 Tools & Environment

- **Language:** Python 3.10+
- **Framework:** PyTorch 2.x
- **Notebooks:** Google Colab, Jupyter Notebooks
- **Version Control:** GitHub
- **Documentation:** Overleaf (this document), updated weekly

1.4 Study Structure

This is a 3-credit independent study meeting twice a week — every Tuesday and Thursday. Each week consists of approximately 3 hours of guided theory aligned with the textbook, plus an additional 10–12 hours of personal study commitment: practicing code, running experiments in Google Colab and Jupyter Notebooks, and reinforcing concepts hands-on. This is not an expert-level starting point — the approach is continuous learning, building understanding from the ground up before tackling anything complex.

Chapter	Topic	Week
1	Understanding Large Language Models	Week 1
2	Working with Text Data	Week 2
3	Coding Attention Mechanisms	Weeks 3–4
4	Implementing a GPT Model from Scratch	Weeks 5–6
5	Pretraining on Unlabeled Data	Weeks 7–8
6	Fine-Tuning for Classification	Weeks 9–10
7	Fine-Tuning to Follow Instructions	Weeks 11–12

2. Weekly Progress

Week 1 — Understanding Large Language Models

Tuesday, January 20 • Thursday, January 22, 2026 • 1.5 hrs each session

Weekly Goals

- Read Chapter 1 in full, including the preface and author background.
- Understand what an LLM is, where it came from, and why it matters.
- Explore the author's work beyond the book — YouTube, LinkedIn, blog, research.
- Build curiosity and context before writing a single line of code.
- Get a clear picture of the three-stage pipeline the entire book follows.

Tuesday, January 20 — Session 1: Starting from the Beginning

Week 1 had no code. That was intentional. Before writing anything, the goal was to understand *who* wrote this book, *why* it was written, and *what* the journey ahead actually looks like. This is the seed phase getting the soil right before planting anything.

Started by reading the preface and the author sections. Sebastian Raschka is a machine learning researcher and educator who has spent years making complex topics accessible. Reading his background made it clear this book is not just a technical manual it is written by someone who genuinely cares about the reader understanding deeply, not just copying code. That changed how I approached everything that followed.

From there: his LinkedIn, his YouTube channel where he teaches machine learning concepts, his personal blog, and his research papers. This was not procrastination — it was orientation. Understanding the author's perspective made Chapter 1 land differently when I finally read it.

Key Concepts

What is a Large Language Model? An LLM is a deep neural network trained on massive amounts of text to understand, generate, and respond to human language. The "large" in the name refers to two things: the number of parameters (often tens to hundreds of billions of adjustable weights), and the scale of the training data (sometimes the entire publicly available internet).

The training task is deceptively simple: predict the next word. Given a sequence of text, the model learns to predict what comes next. Repeating this across billions of examples forces the model to learn grammar, facts, context, reasoning patterns, and the structure of language — not because any of those things were explicitly taught, but because predicting the next word well requires understanding all of them.

Where LLMs sit in the bigger picture. AI is the broad field. Machine learning is a subset of AI focused on algorithms that learn from data. Deep learning is a subset of machine learning focused on neural networks with many layers. LLMs are a specific application of deep learning — they use transformer-based neural networks trained on text at scale. The hierarchy matters because it clarifies what LLMs actually are: not magic, but the current frontier of a well-defined progression of ideas.

Why LLMs changed everything. Before LLMs, NLP models were narrow. A spam classifier could classify spam. A translation model could translate. Each model was built for one task. LLMs broke this pattern. The same pretrained model can translate, summarize, answer questions, write code, and classify text — all without being explicitly trained on each task separately. This generality is what makes them so significant.

Thursday, January 22 — Session 2: The Architecture and the Pipeline

Second session went deeper into Chapter 1 — the transformer architecture, the GPT design specifically, and the three-stage pipeline that organizes the entire book. Also spent time reading Raschka's blog posts and watching parts of his YouTube lectures, which gave a more intuitive feel for concepts the book explains more formally.

Key Concepts

The Transformer Architecture. Introduced in the 2017 paper *Attention Is All You Need* [2], the transformer replaced recurrent networks (RNNs and LSTMs) as the dominant architecture for language tasks. The core innovation is *self-attention*: instead of processing tokens one at a time in order, the transformer processes the entire sequence simultaneously and learns which tokens should influence which other tokens. This makes it much better at capturing long-range relationships in text.

The original transformer had two parts: an **encoder** that reads and encodes the input, and a **decoder** that generates the output. GPT uses only the decoder half.

BERT vs. GPT — two different directions. Both are transformer-based, but they go different directions. BERT uses the encoder and is trained by masking random words and predicting them — it sees the full context in both directions. GPT uses the decoder and predicts the next word left-to-right — it can only see what came before. BERT is better for understanding tasks like classification. GPT is better for generation tasks like writing and conversation. This study follows the GPT path entirely.

GPT is autoregressive. Each new word is generated based on everything that came before it. The model generates one token at a time, feeding each output back as input for the next step. This is what "autoregressive" means. It is slower than parallel processing, but it is how coherent text gets built word by word.

Emergent behavior. GPT models are trained only on next-word prediction. Yet they can translate languages, summarize documents, answer questions, and write code — none of which they were explicitly trained to do. This is called *emergent behavior*: capabilities that arise from scale and exposure to diverse data, not from being specifically programmed. Researchers did not expect this. It is one of the most surprising and important discoveries in the field.

Key Concepts

The Three-Stage Pipeline — the Road Map for This Study. Raschka frames the entire book around three stages. Understanding this now makes every chapter make sense in context:

1. **Stage 1 — Architecture and Data:** Implement the data pipeline (tokenization, embeddings) and the attention mechanism. This is Chapters 1–4. No training yet — just building the machine.
2. **Stage 2 — Pretraining:** Train the model on raw, unlabeled text using next-word prediction. The model develops general language understanding. This is Chapter 5. Also covers loading pretrained weights from OpenAI's GPT-2 so expensive retraining can be skipped.
3. **Stage 3 — Fine-Tuning:** Take the pretrained model and adapt it to a specific task using labeled data. Two types: classification fine-tuning (Chapter 6) and instruction fine-tuning (Chapter 7).

This pipeline mirrors how real-world LLMs like ChatGPT are built. Pretraining costs millions of dollars and takes massive compute. Fine-tuning is much cheaper and is where most practical applications happen. Understanding both is the goal.
Why build from scratch? The book makes a strong case: custom-built LLMs can outperform general-purpose ones on specific tasks. Companies like Bloomberg have built domain-specific LLMs (BloombergGPT for finance) that beat ChatGPT in their area. Privacy is another reason — not sending sensitive data to a third-party API. And smaller, fine-tuned models can run locally on laptops or phones, reducing latency and cost.
But for this study, the reason is simpler: you cannot understand what you did not build.

Key Concepts

Training Data at Scale. GPT-3 was trained on approximately 300 billion tokens drawn from five datasets: CommonCrawl (60%), WebText2 (22%), Books1 and Books2 (8% each), and Wikipedia (3%). A token is roughly a word or punctuation character. The CommonCrawl portion alone is around 570 GB of text. The estimated cost of training GPT-3 was $4.6 million in cloud computing. This is why pretraining from scratch is not practical for this study — but understanding it is. The book provides code to pretrain on a small dataset for educational purposes, then load GPT-2's open-source weights to skip the expensive part.

Challenges & Open Questions

- **Self-attention** is mentioned throughout Chapter 1 but not yet explained mechanically. The book defers this to Chapter 3. For now the intuition is: each token learns to "pay attention" to other tokens that are relevant to it. The details — queries, keys, values, dot products — are coming.
- **Tokenization** is described as converting text into tokens but the exact mechanics (how words get split, what byte pair encoding actually does) are covered in Chapter 2. Noted as something to not skip past quickly.
- **Scale vs. understanding:** GPT-3 has 96 transformer layers and 175 billion parameters. These numbers are hard to build intuition around at this stage. The study will build a much smaller version, which should make the architecture tangible.

Reflection

Week 1 felt like turning on a light in a room you have been in before but never really seen clearly. The concepts — neural networks, training, language models — were not entirely new words. But understanding *why* they work the way they do, and *how* each piece connects to the next, was different this time.

Reading the preface before the technical content was the right call. It made the book feel like a conversation rather than a manual. Following Raschka's YouTube and blog added a layer that the text alone does not give — you hear how he *thinks* about problems, not just what the answers are. That matters for learning.

The insight that stayed with me most: predicting the next word is a trivial task on the surface. But to do it well across billions of diverse sentences, the model has to internally build something that looks a lot like understanding. The simplicity of the training objective is not a limitation — it is the whole trick. That is genuinely surprising, and it makes me want to understand the mechanism deeply, not just use it.

No code this week. That was the right call. The foundation has to be conceptual before it can be technical.

Week 2 — Working with Text Data

Tuesday, January 27 •Thursday, January 29, 2026 •1.5 hrs each session

Weekly Goals

- Understand why raw text cannot be fed directly into a neural network.
- Learn how text gets tokenized split into words and special characters.
- Build a simple tokenizer from scratch using Python's `re` module.
- Understand byte pair encoding (BPE) and use the `tiktoken` library.
- Implement the sliding window approach to create input–target training pairs.
- Understand token embeddings and why positional embeddings are needed.

Tuesday, January 27 Session 1: From Text to Tokens

This week is where things got real. Week 1 was all concepts. Week 2 is where the first lines of code got written and honestly, it felt really good. The ideas are simple, the Python is approachable, and watching each step work made the whole pipeline start to feel tangible.

Key Concepts

Why can't neural networks process raw text? Neural networks work with numbers continuous-valued vectors that can be multiplied, added, and differentiated. Text is categorical. The word "cat" has no inherent numerical value. Before an LLM can process any text, every word and punctuation mark has to be converted into numbers. The pipeline is:

$$\text{raw text} \longrightarrow \text{tokens} \longrightarrow \text{token IDs} \longrightarrow \text{embedding vectors}$$

What is an embedding? An embedding maps a discrete token into a point in a continuous vector space. Instead of representing "king" as a single integer, an embedding represents it as a vector of hundreds of numbers that encode its meaning and relationships to other words. Similar words end up close together in this space. This is what allows a neural network to reason about language mathematically.

The first hands-on task: download a real text file and load it into Python. The book uses a short story by Edith Wharton, *The Verdict*, as the training text for this chapter.

```python
import urllib.request

url = ("https://raw.githubusercontent.com/rasbt/"
       "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
       "the-verdict.txt")
urllib.request.urlretrieve(url, "the-verdict.txt")

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

```python
raw_text = f.read()

print("Total characters:", len(raw_text))
print(raw_text[:99])
```

Listing 1: Downloading and loading the training text

Output:

```
Total characters: 20479
I HAD always thought Jack Gisburn rather a cheap genius--though a good
   fellow enough--so it was no
```

Twenty thousand characters of real text. Small by real-world standards GPT-3 trained on hundreds of billions of tokens but perfect for understanding the pipeline step by step.

Next: split that text into tokens using Python's `re` module.

```python
import re

text = "Hello, world. Is this-- a test?"

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

Listing 2: Tokenizing text with regular expressions

Output:

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

The regular expression splits on whitespace, commas, periods, dashes, and question marks every punctuation mark becomes its own separate token. Punctuation carries meaning: a period signals end of sentence, a question mark changes the tone. If they stay attached to words, "test." and "test" look like different things to the model even though they are not. Separating them keeps the vocabulary clean.

Running this on the full short story produces **4,690 tokens**.

Thursday, January 29 Session 2: Vocabulary, BPE, and Embeddings

Second session picks up from tokens and goes all the way to the embedding vectors the model actually sees. Every step connected naturally to the one before it.

Key Concepts

Building a Vocabulary. Once the text is tokenized, every unique token gets mapped to a unique integer. Sort all unique tokens alphabetically, assign each one an index, store it in a Python dictionary. That dictionary is the vocabulary. Encoding means looking up each token and returning its integer. Decoding means doing the reverse.

```python
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)
vocab = {token: integer for integer, token in enumerate(all_words)}

print(vocab_size)
```

Listing 3: Building the vocabulary

Output:

```
1130
```

1,130 unique tokens from the short story. Now implement a tokenizer class with two methods encode (text to IDs) and decode (IDs back to text).

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

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

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])
        return re.sub(r'\s+([,.?!"()\'])', r'\1', text)

tokenizer = SimpleTokenizerV1(vocab)
ids = tokenizer.encode('"It\'s the last he painted, you know,"')
print(ids)
print(tokenizer.decode(ids))
```

Listing 4: Simple tokenizer with encode and decode

Output:

```
[1, 56, 2, 850, 988, 602, 533, 746, 5, 1126, 596, 5, 1]
"It's the last he painted, you know,"
```

Encode converts the sentence to integers. Decode brings it back. Clean round-trip. But there is a real limitation: try encoding a word not in the training text and it crashes with a `KeyError`. The vocabulary only knows what it has seen.

The fix is two special tokens added to the vocabulary.

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

print(len(vocab))
```

Listing 5: Adding special tokens to the vocabulary

Output:

```
1132
```

Key Concepts

Special Tokens. `<|unk|>` replaces any word not found in the vocabulary. Instead of crashing, the tokenizer substitutes the unknown word silently. The model learns that this token means "something I haven't seen before."
`<|endoftext|>` is inserted between unrelated texts that are concatenated for training. When training on multiple books or articles, this token signals: the previous document ended, a new one begins. Without it, the model might try to connect things that have nothing to do with each other.
GPT models actually do not use `<|unk|>` at all. They use a smarter tokenizer byte pair encoding that never encounters an unknown word because it breaks any word into subword pieces or individual characters.

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

tokenizer2 = SimpleTokenizerV2(vocab)
print(tokenizer2.decode(tokenizer2.encode(text)))
```

Listing 6: Testing the tokenizer with unknown words and special tokens

Output:

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

"Hello" and "palace" were not in *The Verdict*, so both became `<|unk|>`. The `<|endoftext|>` separator passed through correctly.

Key Concepts

Byte Pair Encoding (BPE). BPE is the tokenization scheme used by GPT-2, GPT-3, and ChatGPT. It builds a vocabulary of subword pieces by starting with individual characters and iteratively merging the most frequent pairs. “de” might become one token because it appears in thousands of words: *define*, *depend*, *made*, *hidden*.
The result: an unknown word like “someunknownPlace” does not crash the tokenizer it gets broken into familiar subword chunks. BPE can handle any text, including words that never appeared in training. The GPT-2 BPE vocabulary has **50,257 tokens**.

```python
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
text = "Hello, do you like tea? <|endoftext|> In the sunlit terraces of
    someunknownPlace."
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
print(tokenizer.decode(integers))
```

Listing 7: Using the tiktoken BPE tokenizer

Output:

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250,
    8812, 2114, 286, 617, 34680, 27271, 13]
Hello, do you like tea? <|endoftext|> In the sunlit terraces of
   someunknownPlace.
```

“someunknownPlace” encodes and decodes perfectly. BPE split it into subword pieces internally and reassembled it cleanly. No <|unk|> needed.

Key Concepts

Sliding Window Creating Training Pairs. The LLM learns by predicting the next token. To train it, every sequence needs to be paired with its target: the same sequence shifted one position forward. A context size of 4 means: input is tokens $[t_1, t_2, t_3, t_4]$, target is $[t_2, t_3, t_4, t_5]$. The window slides across the entire text generating thousands of these pairs.

```python
context_size = 4
enc_sample = tokenizer.encode(raw_text)[50:]

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

Listing 8: Sliding window to generate input–target pairs

Output:

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

This is the entire learning signal. Predict the next word given everything before it. Repeated across billions of examples that is how an LLM learns.

Key Concepts

Token Embeddings. Token IDs are integers. Neural networks need vectors. An embedding layer is a lookup table: each token ID maps to a row in a weight matrix, and that row is the token's embedding vector. These weights start random and get updated during training.
Positional Embeddings. The embedding layer gives every occurrence of the same token the exact same vector no matter where in the sentence it appears. But position matters. "Dog bites man" and "Man bites dog" use the same words but mean opposite things. A second embedding layer encodes the position of each token (0, 1, 2, . . .) and adds it to the token embedding. Final input to the model = token embedding + positional embedding.

```python
import torch

vocab_size = 50257
output_dim = 256
max_length = 4

token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
pos_embedding_layer   = torch.nn.Embedding(max_length, output_dim)

token_embeddings = token_embedding_layer(inputs)
pos_embeddings   = pos_embedding_layer(torch.arange(max_length))

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

Listing 9: Token and positional embeddings

Output:

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

8 training examples, each with 4 tokens, each token as a 256-dimensional vector. This tensor is what the model actually receives. The entire data pipeline raw text to numbers is complete. Everything from here goes into the transformer.

Challenges & Open Questions

- **BPE merge decisions:** How exactly does BPE decide which pairs to merge and when to stop? The frequency cutoff logic is still fuzzy. Will explore more.
- **Stride vs. overlap:** A stride smaller than the context size creates overlapping examples. More data, but risk of overfitting. The right stride is a hyperparameter to tune per task.
- **Embedding dimensions:** GPT-2 uses 768, GPT-3 uses 12,288. How does the number of dimensions affect what the model can represent? Will become clearer when building the full model in Chapter 4.

Reflection

Week 2 was genuinely fun. The Python here is not complicated regular expressions, dictionaries, a class with two methods and yet by the end of the session raw text has been turned into the numerical tensors an LLM actually trains on. That felt significant.

The concept that surprised me most: positional embeddings. It had not occurred to me that the self-attention mechanism is completely position-blind without them. The same word in position 1 and position 10 looks identical to the model. Adding a separate learned embedding for position is such a clean fix just add two vectors together and the model has both meaning and order in one input.

BPE also clicked in a way I did not expect. The idea that you never need an unknown token if you can always fall back to individual characters is simple but powerful. Every word in every language is made of characters. BPE can handle anything.

Looking forward to Chapter 3. That is where attention mechanisms live and that is where the real intelligence of the transformer is supposed to come from.

Week 3 — Coding Attention Mechanisms

Tuesday, February 3 • Thursday, February 5, 2026 • 1.5 hrs each session

Weekly Goals

- Understand why attention mechanisms were invented and what problem they solve.
- Learn the basics of PyTorch tensors before diving into attention code.
- Implement simple self-attention step by step dot products, softmax, context vectors.
- Implement self-attention with trainable weight matrices (queries, keys, values).
- Understand causal masking and why it is essential for GPT-style models.
- Build toward multi-head attention and understand what it adds.

Tuesday, February 3 Session 1: PyTorch Basics + Simple Self-Attention

Honest note to start: this was the hardest week so far. Chapter 3 is where the complexity jumped significantly. The concepts are deep, the code has more moving parts, and there were moments of genuine confusion especially around tensor shapes, the meaning of context vectors, and why all these steps were needed. But the chapter is also where the real magic of transformers lives. So the head stayed up.

Before touching any attention code, time was spent on PyTorch basics. The book uses PyTorch throughout, and Chapter 3 assumes comfort with tensors, dot products, and matrix multiplication. This was the first week where that assumption started to feel real.

Key Concepts

PyTorch Tensors the Foundation of Everything. PyTorch tensors are like NumPy arrays but with two critical extras: they can run on a GPU, and they support automatic differentiation (autograd), which is how neural networks learn. A 1D tensor is a vector. A 2D tensor is a matrix. A 3D tensor is a batch of matrices and that is what shows up constantly in LLM code. The shape of a tensor is everything. When code breaks in PyTorch, it is almost always a shape mismatch. Getting comfortable reading shapes like `[batch, tokens, embedding_dim]` was the first real skill this week.
Dot product the core operation of attention. A dot product multiplies two vectors element-wise and sums the results into a single number. It measures similarity: two vectors pointing in the same direction give a large dot product. Two perpendicular vectors give zero. In attention, dot products between token vectors tell the model how much one token should "attend to" another.

Key Concepts

Why do we need attention mechanisms at all? Before transformers, the dominant architecture for language tasks was the RNN (recurrent neural network). RNNs process text one token at a time, passing a hidden state forward. The problem: by the time the model reaches the end of a long sentence, the hidden state has compressed everything that came before into a single fixed-size vector. Early context gets squeezed out. Long-range dependencies where a word at the start of a paragraph determines the meaning of a word at the end get lost.
Attention was invented to fix this. Instead of compressing everything into one vector, attention lets every token look directly at every other token and decide how much to weight each one. No compression. No forgetting. Direct access.

The sentence used throughout this chapter: *"Your journey starts with one step."* Six tokens, each embedded as a 3-dimensional vector. Small enough to follow every number.

```python
import torch

inputs = torch.tensor(
  [[0.43, 0.15, 0.89],  # Your     (x1)
   [0.55, 0.87, 0.66],  # journey  (x2)
   [0.57, 0.85, 0.64],  # starts   (x3)
   [0.22, 0.58, 0.33],  # with     (x4)
   [0.77, 0.25, 0.10],  # one      (x5)
   [0.05, 0.80, 0.55]]  # step     (x6)
)
```

Listing 10: Input sentence as embedded token vectors

Output: No output yet just defining the inputs. Each row is one token's embedding. Six rows, three dimensions each. Shape: `[6, 3]`.

Key Concepts

Simple Self-Attention Step by Step. The goal: for each token, compute a *context vector* an enriched representation that blends information from all other tokens, weighted by relevance. For token **x2** ("journey"), the steps are:

1. Compute **attention scores**: dot product of x2 with every other token. Higher score = more similar = more relevant.
2. **Normalize** with softmax to get attention weights that sum to 1.
3. Compute the **context vector**: weighted sum of all input vectors, using the attention weights.

```python
query = inputs[1]   # "journey" is the query token
```

```python

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

print(attn_scores_2)
```

Listing 11: Step 1: Compute attention scores for token x2

Output:

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

Each number is how similar that token is to “journey.” The token “starts” (x3) scores high at 1.4754 makes sense, it is semantically close to “journey.” The token “one” (x5) scores lowest least related.

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

Listing 12: Step 2: Normalize with softmax to get attention weights

Output:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082,
    0.1581])
Sum: tensor(1.)
```

Softmax converts raw scores into weights that sum to 1. Now they can be interpreted as: “pay 23.8% attention to ‘journey’ itself, 23.3% to ‘starts’, 13.9% to ‘Your’. . . ”

```python
context_vec_2 = torch.zeros(query.shape)
for i, x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i] * x_i

print(context_vec_2)
```

Listing 13: Step 3: Compute the context vector for x2

Output:

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

This 3-dimensional vector is the context vector for “journey.” It is no longer just the raw embedding of “journey” it now contains blended information from all six tokens, weighted by how relevant each one is. This is what self-attention produces.

Then the same computation is generalized to all tokens at once using matrix multiplication replacing the slow Python for-loop with a single efficient operation:

```python
attn_scores  = inputs @ inputs.T
attn_weights = torch.softmax(attn_scores, dim=-1)
all_context_vecs = attn_weights @ inputs
print(all_context_vecs)
```

Listing 14: All attention weights and context vectors at once

Output:

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

Row 2 matches the manually computed `context_vec_2` exactly. The matrix multiplication does in one line what the loop did step by step.

Thursday, February 5 Session 2: Trainable Weights, Causal Masking, Multi-Head

Second session was the steeper climb. This is where the real complexity of Chapter 3 comes in. The simple attention from Session 1 has no learnable parameters it uses the raw input vectors directly. Real LLMs need the model to *learn* how to attend. That requires weight matrices.

Key Concepts

Self-Attention with Trainable Weights Queries, Keys, and Values. The core idea: instead of computing dot products directly between input vectors, first transform each input through three separate learned weight matrices:

$$Q = X \cdot W_q \qquad K = X \cdot W_k \qquad V = X \cdot W_v$$

Query what this token is looking for.
Key what this token is offering to others.
Value the actual content this token contributes to the output.
Attention scores are computed between queries and keys. Context vectors are computed as weighted sums of values. The weight matrices W_q, W_k, W_v are learned during training the model figures out the best way to transform inputs so that relevant tokens end up with high attention scores.
Why scale by $\sqrt{d_k}$? When d_k (the key dimension) is large, dot products grow large in magnitude and push softmax into saturation near-zero gradients, learning stalls. Dividing by $\sqrt{d_k}$ keeps values in a stable range. This is the "scaled" in scaled dot-product attention.

```python
import torch.nn as nn

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

    def forward(self, x):
        keys    = self.W_key(x)
        queries = self.W_query(x)
        values  = self.W_value(x)

        attn_scores  = queries @ keys.T
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        return attn_weights @ values

torch.manual_seed(789)
sa = SelfAttention_v2(d_in=3, d_out=2)
print(sa(inputs))
```

Listing 15: Implementing self-attention with trainable weight matrices

Output:

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

Six context vectors, each 2-dimensional (d_out=2). The weight matrices project from 3D input space into 2D output space. These weights will be updated during training.

Key Concepts

Causal Attention Hiding the Future. GPT generates text left-to-right. When predicting the next token, it must not be allowed to look at tokens that come after the current position that would be cheating. Causal attention (also called masked attention) enforces this constraint. The fix: mask out all attention scores above the diagonal. Replace them with $-\infty$ before applying softmax. Since $e^{-\infty} = 0$, those positions get zero weight effectively invisible to the model. The result: each token can only attend to itself and everything before it.

```python
context_length = attn_scores.shape[0]

# Upper triangle mask (ones above diagonal)
```

```python
mask = torch.triu(torch.ones(context_length, context_length), diagonal
    =1)

# Fill upper triangle with -inf before softmax
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

Listing 16: Applying the causal mask with negative infinity

Output:

```
tensor([[0.2899,   -inf,   -inf,   -inf,   -inf,   -inf],
        [0.4656, 0.1723,   -inf,   -inf,   -inf,   -inf],
        [0.4594, 0.1703, 0.1731,   -inf,   -inf,   -inf],
        [0.2642, 0.1024, 0.1036, 0.0186,   -inf,   -inf],
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786,   -inf],
        [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]])
```

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

Listing 17: Apply softmax the -inf values become zero automatically

Output:

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

The triangle shape is clear: token 1 can only see itself. Token 6 can see all six. Everything above the diagonal is zero. No future leakage.

All of this the weight matrices, the masking, dropout for regularization gets packaged into a clean `CausalAttention` class. That class is then stacked into `MultiHeadAttention`.

Key Concepts

Multi-Head Attention Why Run Attention Multiple Times? A single attention head looks at the input through one "lens" one set of learned W_q, W_k, W_v matrices. Multi-head attention runs several attention operations in parallel, each with its own weight matrices. Each head can learn to focus on different aspects: one head might track syntactic relationships, another semantic similarity, another long-range coreference.

The outputs of all heads are concatenated and projected back to the original dimension. The result: a richer, more expressive context vector that captures multiple perspectives on the input simultaneously.

In GPT-2 (small), there are 12 heads operating in parallel. In GPT-3, 96 heads. The efficient implementation does not run them sequentially it reshapes the tensors so all heads compute in parallel through a single batched matrix multiplication.

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert d_out % num_heads == 0
        self.d_out     = d_out
        self.num_heads = num_heads
        self.head_dim  = d_out // num_heads   # split d_out across
           heads
        self.W_query   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key     = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj  = nn.Linear(d_out, d_out)
        self.dropout   = nn.Dropout(dropout)
        self.register_buffer("mask",
            torch.triu(torch.ones(context_length, context_length),
               diagonal=1))

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys    = self.W_key(x).view(b, num_tokens, self.num_heads,
           self.head_dim).transpose(1,2)
        queries = self.W_query(x).view(b, num_tokens, self.num_heads,
           self.head_dim).transpose(1,2)
        values  = self.W_value(x).view(b, num_tokens, self.num_heads,
           self.head_dim).transpose(1,2)

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

        context_vec = (attn_weights @ values).transpose(1,2).contiguous
            ()
        context_vec = context_vec.view(b, num_tokens, self.d_out)
        return self.out_proj(context_vec)

torch.manual_seed(123)
batch = torch.stack((inputs, inputs), dim=0)    # batch of 2 inputs
mha = MultiHeadAttention(d_in=3, d_out=2, context_length=6, dropout
    =0.0, num_heads=2)
context_vecs = mha(batch)
print(context_vecs)
print("Shape:", context_vecs.shape)
```

Listing 18: Multi-head attention efficient implementation

Output:

```
tensor([[[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]],
        [[0.3190, 0.4858],
         ...
Shape: torch.Size([2, 6, 2])
```

Batch of 2 sentences, 6 tokens each, 2-dimensional context vector per token. The two sentences produce identical output because they are duplicates. This is the attention module that will plug directly into the GPT model in Chapter 4.

Challenges & Open Questions

- **Tensor shapes were the hardest part.** Going from `[b, tokens, d_out]` to `[b, heads, tokens, head_dim]` through `.view()` and `.transpose()` is not immediately obvious. Had to draw these transformations out on paper to follow what was happening at each step.
- **Why values and not just keys?** The Q/K/V split felt arbitrary at first. The intuition that helped: Q and K determine *how much* to attend, V determines *what* gets passed forward. They serve different roles.
- **register_buffer vs. nn.Parameter:** The mask uses `register_buffer` because it is not a learnable parameter it should not be updated by the optimizer. But it still needs to move to GPU with the model. This distinction took a moment to understand.
- **Motivation dip:** The jump in complexity this week was real. There were moments where the number of steps felt overwhelming. What helped: going back to the simple version first, getting that working, then moving to the complex version. Building up, not jumping in.

Reflection

Week 3 was the toughest so far and also the most rewarding once things clicked.

The moment that helped most: understanding that the entire chapter is really just three steps repeated at different levels of complexity. (1) Compute similarity scores. (2) Normalize with softmax. (3) Take a weighted sum. That is self-attention. Everything else the weight matrices, the masking, the multi-head structure is built on top of that same core loop.

The demotivation was real. When the tensor shapes started flying and the `.view().transpose().contiguous()` chains appeared, it felt like the gap between "I understand the idea" and "I understand the code" was suddenly very large. What got me through: running each piece in isolation, printing shapes, checking that intermediate outputs matched what was expected before moving on.

PyTorch clicked more this week than in any previous session. The difference between `nn.Parameter` (learned) and `register_buffer` (fixed but device-aware) is a small thing, but understanding it made the code feel less like magic.

One sentence that kept things in perspective: every complex LLM in the world GPT-4, Gemini, Claude is built on this same attention mechanism. Understanding it at this level, from the dot products up, is understanding the foundation everything else rests on. That is worth the difficulty.

Week 4 — Attention Mechanisms: Deep Revision

Tuesday, February 10 • Thursday, February 12, 2026 • 1.5 hrs each session

Weekly Goals

- Revisit Chapter 3 fully with a clearer head and stronger intent.
- Read the original *Attention Is All You Need* paper for context.
- Watch YouTube explanations to build visual intuition before returning to code.
- Rebuild attention from dot products to multi-head without referencing notes.
- Feel positioned solidly before moving into Chapter 4.

The Decision to Revisit

Week 3 ended with understanding that was real but not solid. The code worked. The outputs made sense. But if someone asked to explain causal masking or why the value matrix exists without looking at notes, it would have been difficult. That is not good enough. Chapter 4 builds directly on this. Chapter 5 builds on Chapter 4. Getting lost here means getting more lost later.

So week 4 became a revision week. Same chapter, different approach. Less following the book line by line, more going out and finding the intuition from other sources first, then coming back to the code with that foundation.

Tuesday, February 10 — Session 1: External Resources and the Paper

The first session was entirely off the book. Read *Attention Is All You Need* [2], watched several YouTube explanations, and read a few blog articles. This was the most useful single session of the study so far in terms of building genuine understanding.

Key Concepts

What the original paper taught that the book did not emphasize. *Attention Is All You Need* (Vaswani et al., 2017) introduced the transformer and replaced the dominant RNN-based architectures entirely. The paper's title is a statement: you do not need recurrence or convolution. Attention alone is sufficient.
The key formula in the paper:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Reading this in the paper after implementing it piece by piece in code made the formula feel different. Each symbol now had a concrete meaning. Q is the matrix of queries for every token. K is the matrix of keys. V is the values. The dot product QK^T computes all pairwise similarities in one shot. Dividing by $\sqrt{d_k}$ stabilizes the gradients. Softmax turns scores into weights. Multiplying by V blends the values. Seven symbols. One formula. The entire attention mechanism.

Key Concepts

The Query / Key / Value intuition that finally clicked. The database analogy helped more than anything else this week. Think of attention as a soft database lookup:

- The **Query** is what you are searching for. It represents the current token asking: "what information do I need from the other tokens?"
- The **Key** is what each token is advertising. It says: "here is what I contain, in case anyone finds it relevant."
- The **Value** is what actually gets retrieved. Once the query finds a relevant key, the corresponding value is what gets passed forward.

In a hard database lookup, you either find an exact match or you do not. In attention, every key matches every query to some degree — the dot product score determines how much. The result is a *soft* retrieval: a blend of all values, weighted by relevance. This is why attention is so powerful. It never says "not relevant." It just says "less relevant."

Key Concepts

What multi-head attention is actually doing. One head looks at the input through one learned perspective. The weight matrices W_q, W_k, W_v define that perspective. Different random initializations lead to different perspectives being learned.
Multiple heads run in parallel, each looking at the same input but through different lenses. One head might learn to track which noun a pronoun refers to. Another might learn syntactic dependencies. Another might track topic continuity across sentences. None of this is programmed in. It emerges from training. The model discovers what each head is useful for by gradient descent.
Concatenating the outputs of all heads and projecting them back down with W_o gives the final context vector: richer than anything a single head could produce.

Key Concepts

Causal masking: the clean mental model. Causal attention exists because GPT generates tokens one at a time, left to right. At generation time, future tokens do not exist yet. So during training, the model must be prevented from using future tokens — otherwise it would learn a shortcut (just copy the answer from the future), which would not generalize to real generation.
The mask enforces this constraint by zeroing out all attention weights above the diagonal. Token 1 attends only to token 1. Token 3 attends to tokens 1, 2, 3. Token n attends to all n tokens. The lower-triangular structure of the attention weight matrix is what makes the model genuinely autoregressive.
Using $-\infty$ before softmax is the elegant implementation: $e^{-\infty} = 0$, so masked positions contribute nothing to the weighted sum. No separate re-normalization needed. Softmax handles it automatically.

Thursday, February 12 — Session 2: Rebuilding the Code from Memory

Second session: close the notes, open a blank notebook, rebuild the attention mechanism from scratch. Start from dot products. End at `MultiHeadAttention`. If anything cannot be rebuilt, that is the gap.

```python
import torch

# Six tokens, 3-dimensional embeddings
inputs = torch.tensor(
  [[0.43, 0.15, 0.89],
   [0.55, 0.87, 0.66],
   [0.57, 0.85, 0.64],
   [0.22, 0.58, 0.33],
   [0.77, 0.25, 0.10],
   [0.05, 0.80, 0.55]]
)
```

```

# All pairwise attention scores: inputs @ inputs.T
attn_scores = inputs @ inputs.T
print(attn_scores.shape)
```

Listing 19: Step 1 rebuilt: attention scores via dot product

Output:

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

A 6x6 matrix. Row i, column j is the dot product between token i and token j. High value means high similarity. This is the raw attention score matrix.

```
attn_weights = torch.softmax(attn_scores, dim=-1)
print(attn_weights)
print("Row sums:", attn_weights.sum(dim=-1))
```

Listing 20: Step 2 rebuilt: normalize with softmax

Output:

```
tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],
        [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],
        ...])
Row sums: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

Every row sums to 1. Each row is now a probability distribution over the six tokens: how much attention this token pays to each other token.

```
context_vecs = attn_weights @ inputs
print(context_vecs)
print("Shape:", context_vecs.shape)
```

Listing 21: Step 3 rebuilt: context vectors as weighted sum

Output:

```
tensor([[0.4421, 0.5931, 0.5790],
        [0.4419, 0.6515, 0.5683],
        [0.4431, 0.6496, 0.5671],
        [0.4304, 0.6298, 0.5510],
        [0.4671, 0.5910, 0.5266],
        [0.4177, 0.6503, 0.5645]])
Shape: torch.Size([6, 3])
```

Six context vectors. Each is a blend of all six input vectors, weighted by the attention scores. The shape stayed the same: 6 tokens, 3 dimensions. The numbers changed because each token now carries information from its neighbors.

```python
import torch.nn as nn

context_length = 6
mask = torch.triu(torch.ones(context_length, context_length), diagonal
    =1)

# Apply to attention scores: fill upper triangle with -inf
masked_scores = attn_scores.masked_fill(mask.bool(), -torch.inf)
causal_weights = torch.softmax(masked_scores, dim=-1)
print(causal_weights)
```

Listing 22: Causal mask rebuilt from scratch

Output:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5384, 0.4616, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3438, 0.3312, 0.3250, 0.0000, 0.0000, 0.0000],
        [0.2714, 0.2528, 0.2542, 0.2216, 0.0000, 0.0000],
        [0.2258, 0.2059, 0.2076, 0.1850, 0.1757, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]])
```

The lower-triangular structure is clean. Token 1 sees only itself. Token 6 sees everything. This is what makes the model autoregressive. Rebuilt from memory without errors. That felt like a real milestone.

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads,
        qkv_bias=False):
        super().__init__()
        assert d_out % num_heads == 0
        self.num_heads = num_heads
        self.head_dim  = d_out // num_heads
        self.d_out     = d_out
        self.W_query   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key     = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj  = nn.Linear(d_out, d_out)
        self.dropout   = nn.Dropout(dropout)
        self.register_buffer("mask",
            torch.triu(torch.ones(context_length, context_length),
               diagonal=1))

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        # Project and split into heads
        def split_heads(tensor):
            return tensor.view(b, num_tokens, self.num_heads, self.head_dim).transpose(1, 2)

        Q = split_heads(self.W_query(x))
        K = split_heads(self.W_key(x))
        V = split_heads(self.W_value(x))

        # Scaled dot-product attention with causal mask
        scores = Q @ K.transpose(2, 3) / self.head_dim**0.5
        scores.masked_fill_(self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
        weights = self.dropout(torch.softmax(scores, dim=-1))

        # Combine heads and project
        out = (weights @ V).transpose(1, 2).contiguous().view(b, num_tokens, self.d_out)
        return self.out_proj(out)

torch.manual_seed(123)
batch = torch.stack((inputs, inputs), dim=0)
mha = MultiHeadAttention(d_in=3, d_out=2, context_length=6, dropout=0.0, num_heads=2)
print(mha(batch).shape)
```

Listing 23: Full MultiHeadAttention class rebuilt

Output:

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

Batch of 2, six tokens, 2-dimensional output per token. Rebuilt cleanly. The `split_heads` helper made the reshape logic easier to read and reason about.

Challenges & Open Questions

- **The output projection layer:** `self.out_proj` is a linear layer applied after combining the heads. The book mentions it is not strictly necessary but is standard in all real LLMs. Understanding why: it gives the model one more learned transformation to mix information across heads before passing the output forward. Makes sense in retrospect.
- **.contiguous():** Called before `.view()` when the tensor is not stored contiguously in memory after `.transpose()`. Without it, `.view()` throws a runtime error. This is a PyTorch memory layout detail. Not conceptually deep but important to know.
- **dropout on attention weights:** Randomly zeroing some attention weights during training forces the model not to over-rely on any particular token relationship. Only applied during training, disabled at inference.

Reflection

Taking a full week to revise was the right call. No question.
Week 3 ended with surface understanding. Week 4 ends with something that feels more like ownership. The difference is being able to write the code without looking, being able to explain each step in plain words, and being able to trace a specific tensor through all its shape transformations from input to output.
Reading the original paper was what pushed understanding from “I follow the steps” to “I understand why the steps are what they are.” The formula in the paper is not abstract anymore. Every symbol maps to code that has been written.
The YouTube videos helped build visual intuition that the book could not give in text alone. Seeing the attention weight matrix drawn as a heatmap, watching the heads specialize during training in animated form — that kind of visual explanation fills in gaps that pure code cannot.
Going into Chapter 4 feeling genuinely ready. The attention mechanism is understood. The data pipeline from Chapter 2 is understood. The next step is putting them together into a full GPT architecture and watching it generate text. That is where it gets real.

Week 5 — Implementing a GPT Model from Scratch

Tuesday, February 17 • Thursday, February 19, 2026 • 1.5 hrs each session

Weekly Goals

- Understand the full GPT architecture from a top-down view.
- Learn how all previous components fit together into one model.
- Understand layer normalization, feed-forward networks, and shortcut connections.
- Understand what a transformer block is and how it is assembled.
- Study the GPT-2 (124M) configuration and understand what each hyperparameter controls.
- Plan to write code once the Overleaf documentation is pushed and professor meeting is done.

Where Things Stand

Week 4 ended with a solid understanding of attention mechanisms. Week 5 is Chapter 4, which is where everything so far gets assembled into a full GPT-style model. No code was written this week. The focus was on reading the chapter carefully, understanding the architecture from top to bottom, and building a clear mental model before the professor meeting next week.

This chapter felt different from the previous ones. Less confusing, more satisfying. Having a solid foundation in attention meant that the new pieces — layer normalization, GELU activations, shortcut connections, the transformer block — could be understood in terms of what problem each one solves rather than just what the code looks like.

Tuesday, February 17 — Session 1: The GPT Architecture Top-Down

Key Concepts

The Big Picture: What a GPT Model Actually Is. A GPT model is a stack of transformer blocks sitting between two embedding layers at the input and a linear output layer at the end. That is the entire architecture. Every capability — translation, summarization, question answering, code generation — comes from this same structure trained on enough text.
The data flow through the model is:

1. **Token embedding:** Input token IDs are converted to vectors via a learned embedding matrix. Shape: `[batch, tokens, d_model]`.
2. **Positional embedding:** A learned position vector is added to each token embedding. Same shape. Now each token knows where it is in the sequence.
3. **Transformer blocks:** The combined embeddings pass through N stacked transformer blocks. Each block refines the representations using attention and a feed-forward network.
4. **Final layer norm:** Applied once after the last transformer block.
5. **Output projection:** A linear layer maps from `d_model` to `vocab_size`. Each position produces a score for every token in the vocabulary. The highest score is the predicted next token.

Key Concepts

GPT-2 (124M) Configuration. The smallest GPT-2 model is defined by a small set of hyperparameters. Understanding what each one controls is essential before looking at any code.

Hyperparameter	Value and Meaning
`vocab_size`	50,257 BPE vocabulary from tiktoken
`context_length`	1,024 maximum tokens the model can see at once
`emb_dim`	768 embedding dimension for every token
`n_heads`	12 attention heads per transformer block
`n_layers`	12 stacked transformer blocks
`drop_rate`	0.1 dropout probability during training
`qkv_bias`	False no bias terms in Q, K, V projections

Total parameters: **124 million**. Each parameter is a floating-point number updated by gradient descent during training. The embedding matrix alone accounts for $50{,}257 \times 768 \approx 38.6$ million parameters.

Key Concepts

Layer Normalization: Stabilizing Training. Deep networks are hard to train because the distribution of values flowing through each layer shifts constantly as weights update. This is called internal covariate shift. Layer normalization fixes this by normalizing the activations at each layer to have mean 0 and variance 1, then applying learned scale and shift parameters. Unlike batch normalization (which normalizes across the batch dimension), layer norm normalizes across the feature dimension. This makes it independent of batch size and well-suited for language tasks where sequences vary in length.
GPT-2 applies layer norm *before* the attention and feed-forward sublayers (pre-norm), rather than after. Research found pre-norm leads to more stable training in deep transformer models.

Key Concepts

The Feed-Forward Network and GELU Activation. Each transformer block contains a small two-layer feed-forward network applied independently to each token position. The network expands the dimension by a factor of 4, applies an activation function, then projects back down:

$$\text{FFN}(x) = W_2 \cdot \text{GELU}(W_1 x + b_1) + b_2$$

For GPT-2 with `emb_dim=768`: the hidden layer is $768 \times 4 = 3072$ dimensions wide. This expansion gives the model capacity to learn complex non-linear transformations of each token's representation.
GELU (Gaussian Error Linear Unit) is the activation function used instead of ReLU. GELU is smoother than ReLU and allows small negative values through rather than cutting everything below zero to exactly zero. In practice this leads to better performance in transformer models.

Thursday, February 19 — Session 2: Transformer Blocks and Shortcut Connections

Key Concepts

Shortcut Connections: Why They Exist. Deep networks suffer from the vanishing gradient problem: gradients shrink as they backpropagate through many layers, making the earliest layers learn almost nothing. Residual (shortcut) connections solve this by adding the input of a sublayer directly to its output:

$$\text{output} = x + \text{Sublayer}(x)$$

The gradient now has a direct path back through the addition operation, bypassing the sublayer. Even if the sublayer's gradients vanish, the shortcut keeps the signal alive. This is what made training very deep networks (12, 24, 96 layers) practical.
Each transformer block has two shortcut connections: one around the attention sublayer and one around the feed-forward sublayer.

Key Concepts

The Transformer Block: One Complete Unit. A transformer block is the repeating unit that gets stacked N times. Inside each block:

1. **Layer Norm** applied to the input.
2. **Multi-Head Causal Attention** applied to the normalized input.
3. **Dropout** applied to the attention output.
4. **Shortcut connection:** add the original input back.
5. **Layer Norm** applied to the result.
6. **Feed-Forward Network** applied to the normalized result.
7. **Dropout** applied to the FFN output.
8. **Shortcut connection:** add back the result from step 4.

The output has the same shape as the input: `[batch, tokens, emb_dim]`. Stacking 12 of these blocks gives the model 12 chances to refine every token's representation before producing the final output.

Key Concepts

GPT-2 vs. GPT-3: Scale Is the Difference. Architecturally, GPT-2 and GPT-3 are the same model. The difference is scale. GPT-2 (largest variant) has 1.5 billion parameters. GPT-3 has 175 billion. GPT-3 was trained on more data with more compute.
GPT-3's weights are not public. GPT-2's are. This is why the book builds GPT-2: it is small enough to run on a laptop, the weights can be loaded for free, and the architecture is identical to the much larger models. Understanding GPT-2 deeply means understanding the foundation of every GPT-family model that came after it.
Training GPT-3 from scratch on a single consumer GPU would take an estimated 665 years. Training the small GPT-2 on the dataset used in the book takes minutes on a laptop CPU. That difference in scale is what makes this study tractable.

Key Concepts

Parameter Count: Where the 124M Numbers Come From. For the 124M GPT-2 model with `emb_dim=768`, `n_layers=12`:

Component	Approximate Parameters
Token embedding matrix (50257×768)	38.6M
Positional embedding matrix (1024×768)	0.8M
Each transformer block (attention + FFN + norms)	7.1M
12 transformer blocks total	85.2M
Output projection layer	0 (tied to token embeddings)
Total	$\approx$ **124M**

The majority of parameters live in the transformer blocks. Each block's attention module has four weight matrices (W_q, W_k, W_v, W_o), each 768×768. The FFN has two weight matrices (768×3072 and 3072×768). That is where the bulk of the learned knowledge is stored.

Challenges & Open Questions

- **Weight tying:** The output projection layer in GPT-2 shares weights with the token embedding matrix. This is called weight tying. It reduces parameters and reportedly improves performance. The intuition is that the same matrix that maps token IDs to vectors can map vectors back to token scores. This is still conceptually fuzzy and needs revisiting when the code is written.
- **Pre-norm vs. post-norm:** The book uses pre-norm (layer norm before the sublayer). The original 2017 transformer used post-norm. Understanding why pre-norm became standard in modern LLMs is something to read more about.
- **No code yet:** This was a reading and understanding week. The architecture is clear conceptually. The plan is to write the full GPT class once the Overleaf documentation is submitted and after the professor meeting next week. Coming back to this with fresh energy makes more sense than rushing the implementation.

Reflection

Week 5 felt like a turning point in a quiet way. There was no code, no debugging, no shape errors. Just reading and thinking. And for the first time since week 1, that felt completely fine rather than like falling behind.

The reason is that the architecture now makes sense as a whole. In weeks 2 and 3, each component felt isolated: here is tokenization, here is attention, here are embeddings. Week 5 is where those pieces connected into a single coherent picture. The token embedding feeds into the transformer blocks. The transformer blocks use exactly the attention module from Chapter 3. The output projection maps back to vocabulary space. It is one pipeline, end to end.

The parameter count exercise was unexpectedly useful. Working through where 124 million numbers actually live in the model, which layers account for which portion, made the architecture feel concrete rather than abstract. A 768x768 weight matrix is not a vague concept. It is 589,824 numbers, each learned from data.

The professor meeting next week is good timing. Having gone through four complete chapters, built the data pipeline, implemented attention from scratch, and now understood the full model architecture, there is something real to show and discuss. The code implementation of Chapter 4 will follow after that meeting.

References

[1] S. Raschka. *Build a Large Language Model (From Scratch)*. Manning Publications, 2024. https://github.com/rasbt/LLMs-from-scratch

[2] A. Vaswani, N. Shazeer, N. Parmar, et al. *Attention Is All You Need*. NeurIPS, 2017. https://arxiv.org/abs/1706.03762

[3] A. Radford, J. Wu, R. Child, et al. *Language Models are Unsupervised Multitask Learners*. OpenAI, 2019. https://openai.com/blog/better-language-models