200 Iterations: Cracking the Code of the Small Language Model

Rahul Sharma

Contents

Case File #1: The "Why" and the "How"

"Every case starts the same way: a question. Why would anyone build a language model from scratch when you can just call an API? The answer, like most answers in this line of work, is complicated. It's about control. It's about understanding. And sometimes, it's about staring into the void of a NaN loss at 3 AM and finally understanding what makes these things tick."

1.1 Why Build from Scratch in the Age of API Calls?

The year was 2024. OpenAI had GPT-4. Anthropic had Claude. Google had Gemini. Any developer with a credit card and an API key could summon artificial intelligence with a single HTTP request. The case seemed closed before it even opened.

So why was I here, in a dimly lit room, staring at a Jupyter notebook titled `myfirstslmcyberdyne.ipynb`, trying to build something that a billion-dollar company had already perfected?

The Case for Intuition Over Abstraction

Here's the thing about API calls: they're black boxes. Beautiful, convenient, terrifyingly opaque black boxes.

When you call `openai.ChatCompletion.create()`, you're sending a message into the void and hoping the void sends something useful back. You don't know why it works. You don't know why it fails. When the response is garbage, you have exactly one debugging option: *try different prompts and pray.*

That's not engineering. That's superstition.

Building from scratch is different. When your model outputs nonsense, you can interrogate every layer. You can print the attention weights. You can visualize the embeddings. You can trace the gradient flow backward through time like a detective rewinding surveillance footage.

```
# The API way: hope and pray
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[{"role": "user", "content": prompt}]
)
# Something went wrong? Good luck figuring out why.

# The from-scratch way: evidence at every step
logits = model(input_ids)
attention_weights = model.get_attention_weights()  # What is the model looking at?
hidden_states = model.get_hidden_states()  # What features did it extract?
# Something went wrong? You can trace it to the exact neuron.
```

What You Gain: Control, Understanding, and the Ability to Debug

Let me tell you about the three things you get when you build from scratch:

Control. You decide everything. The architecture. The tokenizer. The training data. The loss function. When you need a model that runs on a Raspberry Pi, you can build one. When you need a model that specializes in medieval Latin, you can train one. The API gives you what OpenAI decided you should have. Building gives you exactly what you need.

Understanding. There's a moment—usually around iteration 40 or 50—when the fog clears. You suddenly *understand* why attention is all you need. You understand why layer normalization matters. You understand why the learning rate schedule can make or break a training run. This understanding doesn't come from reading papers. It comes from debugging your own implementations at 2 AM.

The ability to debug when things go wrong. And things *will* go wrong. They always do. When your API-based chatbot starts hallucinating, you're helpless. When your custom model starts hallucinating, you can examine the training data, adjust the temperature, add a repetition penalty, or fine-tune on better examples. You're not a supplicant. You're an engineer.

The Difference Between Using ChatGPT API vs Building Your Own Model

Aspect	ChatGPT API	Building From Scratch
Time to first result	5 minutes	5 days to 5 months
Understanding gained	None	Everything
Customization	Prompt engineering only	Unlimited
Debugging capability	None	Full
Cost at scale	$$$$	$ (once trained)
Offline capability	None	Complete
Data privacy	You trust OpenAI	You control everything

The API is faster. That's undeniable. But speed isn't everything. Sometimes the long road is the only road that leads somewhere useful.

Why Understanding the Internals Makes You a Better ML Engineer

I've interviewed hundreds of ML engineers over the years. The ones who could only call APIs? They hit a ceiling. Fast. The ones who had built something from scratch—even if it was a terrible, buggy, 50-million-parameter toy model? They could solve anything.

Because when you build from scratch, you learn: - Why gradients explode and how to stop them - Why attention masks need to be causal for autoregressive generation - Why the learning rate is the most important hyperparameter - Why mixed-precision training requires careful handling of infinities - Why `model.eval()` and `model.train()` are not optional

This knowledge compounds. Every bug you fix teaches you something. Every NaN you debug leaves you stronger. By the time you've iterated a hundred times, you're not the same engineer who started.

That's why we build from scratch.

1.2 What an LLM Actually Does

Strip away the hype. Ignore the breathless press releases about artificial general intelligence. Forget the philosophical debates about consciousness. At its core, a Large Language Model—or in our case, a Small Language Model—does one thing:

It predicts the next token.

That's it. That's the whole crime.

Next-Token Prediction: The Deceptively Simple Goal

Given a sequence of tokens—words, or pieces of words—predict what comes next. The model sees "The capital of France is" and outputs a probability distribution over its entire vocabulary. "Paris" gets a high probability. "London" gets a lower one. "Banana" gets almost zero.

```
# The fundamental operation
input_text = "The capital of France is"
input_tokens = tokenizer.encode(input_text)  # [464, 3139, 286, 4881, 318]

# Feed through the model
logits = model(input_tokens)  # Shape: [seq_len, vocab_size]

# The last position predicts the next token
next_token_logits = logits[-1]  # Shape: [vocab_size]

# Convert to probabilities
probabilities = torch.softmax(next_token_logits, dim=-1)

# Sample or take the argmax
next_token = torch.argmax(probabilities)  # Probably "Paris"
```

It looks trivial. It is not.

The Mathematical Foundation: P(next_token | previous_tokens)

The mathematics are elegant in their simplicity:

$$P(w_1, w_2, ..., w_n) = \prod_{i=1}^{n} P(w_i | w_1, ..., w_{i-1})$$

The probability of a sequence is the product of each token's probability given all previous tokens. The model learns to estimate $P(w_i | w_1, ..., w_{i-1})$ by training on massive amounts of text.

During training, we show the model text and ask: "At each position, what's the next token?" The model makes predictions. We compute the cross-entropy loss between those predictions and the actual next tokens. We backpropagate. We update weights. We repeat.

```
# Training step
logits = model(input_ids)  # [batch, seq_len, vocab_size]

# Shift: predict position i from positions 0..i-1
```

```
shift_logits = logits[..., :-1, :].contiguous()
shift_labels = labels[..., 1:].contiguous()

# Cross-entropy loss
loss = F.cross_entropy(
    shift_logits.view(-1, vocab_size),
    shift_labels.view(-1),
    ignore_index=pad_token_id
)

# Backprop and update
loss.backward()
optimizer.step()
```

The loss starts high—around 10 or 11, which means the model is essentially guessing randomly among thousands of tokens. As training progresses, the loss drops: 8, 6, 4, 3. Each drop represents the model getting better at predicting what comes next.

From `myfirstslmcyberdyne.ipynb` to `h200-ultimate-model`: The Evolution Preview

My first model was a crime scene. Let me show you the evidence:

```
# myfirstslmcyberdyne.ipynb - Iteration #1
class MiniLLM(nn.Module):
    def __init__(self, vocab_size, emb_size=512, n_layers=8, n_heads=8):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, emb_size)
        self.pos_embed = nn.Embedding(max_len, emb_size)  # Simple learned positions
        self.layers = nn.ModuleList([
            nn.TransformerEncoderLayer(d_model=emb_size, nhead=n_heads)
            for _ in range(n_layers)
        ])
        self.head = nn.Linear(emb_size, vocab_size)
```

By the time I reached `h200-ultimate-model`, the architecture had evolved beyond recognition:

```
# h200-ultimate-model.ipynb - Iteration #100+
class TransformerBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.norm1 = RMSNorm(config.hidden_size)  # Pre-norm, not post-norm
        self.attn = MultiHeadAttention(config)    # With RoPE, Flash Attention
        self.norm2 = RMSNorm(config.hidden_size)
        self.ffn = SwiGLUFeedForward(config)      # SwiGLU, not GELU

    def forward(self, x, freqs_cis):
        # Pre-norm architecture with residual connections
        x = x + self.attn(self.norm1(x), freqs_cis)
        x = x + self.ffn(self.norm2(x))
        return x
```

The journey from one to the other? Two hundred iterations. Countless bugs. Infinite patience. That's what this book is about.

The Autoregressive Nature of Language Models

Language models are autoregressive. They generate one token at a time, feeding each generated token back as input for the next prediction. It's like a detective interviewing a witness who can only answer one word at a time.

```
def generate(model, prompt, max_new_tokens=100):
    tokens = tokenizer.encode(prompt)

    for _ in range(max_new_tokens):
        # Feed all tokens so far
        logits = model(torch.tensor([tokens]))

        # Get prediction for next token
        next_token_logits = logits[0, -1, :]
        next_token = sample(next_token_logits)

        # Add to sequence and continue
        tokens.append(next_token)

        if next_token == eos_token:
            break

    return tokenizer.decode(tokens)
```

This autoregressive nature has profound implications: - **Generation is slow:** Each token requires a full forward pass - **Context matters:** The model sees everything it has generated so far - **Mistakes compound:** A bad early token can derail the entire generation - **Creativity emerges:** Temperature and sampling strategies create variety

---

1.3 Setting Up the Crime Lab

Before you can solve crimes, you need a forensics lab. Before you can train models, you need the right tools.

PyTorch: Your Forensic Toolkit

I chose PyTorch. Not TensorFlow. Not JAX. PyTorch.

Why? Because PyTorch thinks like a programmer. It's imperative, not declarative. When you write `x = x + y`, the addition happens *right now*, not when you finally call `session.run()` inside a computational graph that was compiled three function calls ago.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
# This is actual computation, happening now
x = torch.randn(32, 768)
y = F.relu(x)
z = y.mean()

# Backpropagation is just as simple
z.backward()
print(x.grad)  # Gradients, computed eagerly
```

PyTorch has another advantage: the research community uses it. When a new paper drops on arXiv with code, it's almost always PyTorch. When you're debugging at 3 AM and need Stack Overflow, PyTorch questions have answers.

CUDA Detection: Finding Your Hardware Witness

The first thing any training script does is find the GPU:

```
if torch.cuda.is_available():
    device = torch.device("cuda")
    print(f"Running on: {torch.cuda.get_device_name(0)}")
else:
    device = torch.device("cpu")
    print("Warning: Running on CPU. This will be slow.")
```

Simple, right? Wrong.

The First Lesson: `torch.cuda.is_available()` Lies Sometimes

Here's a crime that cost me six hours:

`torch.cuda.is_available()` returned `True`. I celebrated. I moved my model to CUDA. And then:

```
RuntimeError: CUDA error: no kernel image is available for execution on the device
```

What happened? CUDA was *installed*, but the drivers were outdated. The PyTorch binaries were compiled for CUDA 12.1, but my system had CUDA 11.8 drivers. `is_available()` only checks if *something* CUDA-like exists—not whether it's actually compatible.

The real check:

```
def check_cuda_properly():
    if not torch.cuda.is_available():
        return False, "CUDA not available"

    try:
        # Actually try to use the GPU
        x = torch.zeros(1).cuda()
        del x
        torch.cuda.empty_cache()
        return True, f"CUDA working: {torch.cuda.get_device_name(0)}"
    except Exception as e:
```

```
        return False, f"CUDA failed: {e}"

works, message = check_cuda_properly()
print(message)
```

Trust, but verify.

Memory Management: The Dark Arts

GPUs have limited memory. A 16GB T4 sounds like a lot until you're training a 200M parameter model with a batch size of 8. Then it's not enough. It's never enough.

```
import gc

# The memory management ritual
def clear_memory():
    gc.collect()  # Python garbage collection
    if torch.cuda.is_available():
        torch.cuda.empty_cache()  # Return cached memory to CUDA
        torch.cuda.synchronize()  # Wait for all operations to complete

# Call this between training phases, after evaluation, after generation
clear_memory()
```

But that's just the beginning. Real memory management requires understanding CUDA's memory allocator:

```
import os

# The sacred incantation for memory fragmentation
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'

# For extreme cases
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'max_split_size_mb:128,expandable_segments:True'
```

What does `expandable_segments:True` do? It allows PyTorch to release memory back to CUDA when you call `empty_cache()`, instead of holding it in the PyTorch cache forever. This prevents the dreaded "I have 10GB free but PyTorch says OOM" situation.

Environment Setup: The Complete Ritual

Here's the full setup code that opens every one of my notebooks:

```
import os
import gc
import torch
import warnings

# Suppress the warnings that don't matter
warnings.filterwarnings('ignore')
```

```python
# Memory management configuration
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'

# Device detection (the proper way)
def setup_device():
    if torch.cuda.is_available():
        try:
            # Test that CUDA actually works
            test = torch.zeros(1, device='cuda')
            del test
            torch.cuda.empty_cache()

            device = torch.device('cuda')
            print(f"Device: {torch.cuda.get_device_name(0)}")
            print(f"Memory: {torch.cuda.get_device_properties(0).total_memory / 1e9:.1f} GB")

            # Enable optimizations
            torch.backends.cuda.matmul.allow_tf32 = True
            torch.backends.cudnn.allow_tf32 = True
            torch.backends.cuda.enable_flash_sdp(True)

            return device
        except Exception as e:
            print(f"CUDA failed: {e}")
            print("Falling back to CPU")
            return torch.device('cpu')
    else:
        print("CUDA not available, using CPU")
        return torch.device('cpu')

device = setup_device()
```

This is the foundation. Every experiment builds on this.

Detective's Notebook: The Colab Incident

Personal notes, not for the case file

I started on Google Colab. Everyone does. It's free. It has GPUs. What's not to love?

Everything. Everything is not to love.

The 12-Hour Limit

Colab gives you 12 hours of continuous runtime. Maybe. If you're lucky. If the GPU gods are smiling. Then it kills your session.

No warning. No "save your work." Just *dead*.

I lost my first successful training run at hour 11. The model had just started producing coherent text. Loss was at 2.8 and dropping. I went to get coffee. I came back to:

```
Your session has timed out.
You will lose unsaved work.
```

I learned to checkpoint obsessively:

```
# Save every 1000 steps. No exceptions.
if step % 1000 == 0:
    torch.save({
        'step': step,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': loss.item(),
    }, f'/content/drive/MyDrive/checkpoints/step_{step}.pt')
    print(f"Checkpoint saved at step {step}")
```

The Random Disconnections

Colab doesn't just timeout after 12 hours. It also randomly disconnects whenever it feels like it. Network hiccup? Disconnected. Browser idle for 5 minutes? Disconnected. Google needs the GPU for someone else? Disconnected.

I developed paranoid habits: - Keep a browser tab playing YouTube in the background (keeps the session "active") - Set up JavaScript to click the "Connect" button every 60 seconds - Save to Google Drive constantly - Never trust that the session will still be there

The Persistent Storage Problem

Colab has no persistent local storage. When your session dies, everything in `/content` dies with it. Models, datasets, checkpoints—gone.

You can mount Google Drive, but Google Drive is slow. Painfully slow for loading large datasets:

```
from google.colab import drive
drive.mount('/content/drive')

# This takes FOREVER
dataset = load_from_disk('/content/drive/MyDrive/tokenized_wikipedia')
```

I eventually learned the workflow: 1. Copy data from Drive to local `/content` at session start 2. Train on local storage (fast) 3. Copy checkpoints back to Drive periodically 4. Accept that you will lose work

The Upgrade Path

By iteration 30, I knew Colab wasn't sustainable. The training runs were getting longer. The models were getting bigger. The disconnections were getting more frequent.

I upgraded to Colab Pro. Then Pro+. Then I discovered that even Pro+ has limits—background execution, sure, but still the same fundamental problems.

Eventually, I moved to real cloud GPUs. Lightning AI. Lambda Labs. Vast.ai. Places where a session stays alive until *you* decide to kill it.

The cost went from $0/month to $100/month to $500/month. But the models got better. The iterations got faster. And I stopped losing work at hour 11.

End of Case File #1

Next: Case File #2 — Data: The Fuel. Where we investigate the Stanford OVAL incident, the tokenizer wars, and the day I ran out of disk space mid-training.

Evidence Logged: - Exhibit A: `myfirstslmcyberdyne.ipynb` — The first attempt - Exhibit B: `torch.cuda.is_available()` — The unreliable witness
- Exhibit C: `PYTORCH_CUDA_ALLOC_CONF` — The memory management ritual - Exhibit D: Colab session logs — 47 disconnections over 3 months

Case Status: Foundation established. Ready to proceed to data investigation. # Case File #2: Data — The Fuel

"In this business, the model is only as good as the data you feed it. I've seen promising architectures starve to death on junk food datasets. I've watched beautiful training runs crash and burn because someone grabbed the wrong Wikipedia config at 2 AM. Data isn't just fuel—it's the difference between a witness who talks and one who stays silent forever."

2.1 Finding the Right Dataset

Every investigation needs evidence. Every model needs data. And finding the right data? That's where most cases go cold.

The Wikipedia Incident

It was iteration #12. The architecture was solid. The training loop was clean. Everything was ready. I just needed data—and what better source of knowledge than Wikipedia? Humanity's collective encyclopedia, free for the taking.

I typed the fateful line:

```
from datasets import load_dataset

# The crime scene
dataset = load_dataset("stanford-oval/wikipedia", streaming=True)
```

The download started. Progress bars filled. I felt smug. This was too easy.

Then I tried to access the text:

```
for sample in dataset['train']:
    text = sample['text']
```

```
    print(text[:100])
    break
```

```
KeyError: 'text'
```

What? I checked the schema:

```
print(dataset['train'].features)
# Output: {'title': Value(dtype='string'), 'url': Value(dtype='string'), 'article': Value(dtyp
```

No `text` field. The Stanford OVAL dataset used `article` instead. Fine—annoying, but fixable. I adjusted:

```
text = sample['article']
```

The text came through. But something was wrong. The content was sparse. Truncated. Many articles were just stubs—a title and a sentence or two. This wasn't the rich, detailed Wikipedia I expected.

I had grabbed the wrong dataset entirely.

The Right Config vs. The Wrong Config

The real Wikipedia dataset lives at `wikimedia/wikipedia`. The Stanford OVAL version was a processed subset for question-answering tasks—not raw pretraining data. The schema was different. The content was different. The entire purpose was different.

```
# WRONG - Stanford OVAL's QA-oriented subset
dataset = load_dataset("stanford-oval/wikipedia")

# RIGHT - Wikimedia's full Wikipedia dump
dataset = load_dataset("wikimedia/wikipedia", "20231101.en", streaming=True)
```

The difference:

Aspect	stanford-oval/wikipedia	wikimedia/wikipedia
Purpose	QA tasks	Full pretraining
Text field	`article`	`text`
Content	Truncated summaries	Full articles
Size	~3GB	~20GB
Schema	title, url, article	id, url, title, text

This mistake cost me three days. The model trained fine on the Stanford data—loss dropped, perplexity improved—but the resulting model knew almost nothing. It had learned from shadows instead of substance.

Lesson learned: Always inspect your dataset before training. Print samples. Check schemas. Verify content quality.

The Dataset Evolution Journey

Finding the right training data wasn't a single decision—it was a journey. Each iteration revealed new requirements, new failures, new opportunities.

TinyStories: Where It All Began

```
dataset = load_dataset("roneneldan/TinyStories", split="train", streaming=True)
```

TinyStories was the gateway drug. Simple language. Clear patterns. Stories a child could understand. The model learned basic grammar and narrative structure here—subject, verb, object. Beginning, middle, end.

But TinyStories had limits. The vocabulary was constrained. The topics were narrow. A model trained only on TinyStories could write "Once upon a time there was a little cat" but couldn't explain photosynthesis.

Wikipedia: Factual Grounding

```
dataset = load_dataset("wikimedia/wikipedia", "20231101.en", streaming=True)
```

Wikipedia brought facts. Dates. Names. Scientific concepts. Historical events. The model learned that Paris is the capital of France, that water is H2O, that the Battle of Hastings was in 1066.

But Wikipedia is encyclopedic—dry, formal, impersonal. A model trained only on Wikipedia could recite facts but couldn't hold a conversation.

FineWeb-Edu: Educational Content at Scale

```
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train", streaming=True)
```

FineWeb-Edu was the game-changer. Hundreds of gigabytes of educational web content, filtered for quality. Textbook explanations. Tutorial articles. How-to guides. The model learned not just facts, but how to explain them.

I spent 40,000 training steps on FineWeb-Edu. It was the foundation—the base layer that everything else built upon.

SlimOrca: Instruction Following

```
dataset = load_dataset("Open-Orca/SlimOrca", split="train", streaming=True)
```

SlimOrca taught the model to follow instructions. "Explain X." "Summarize Y." "Compare A and B." The format was crucial:

```
### System:
You are a helpful AI assistant.

### User:
What is the capital of France?

### Assistant:
The capital of France is Paris.
```

After 8,000 steps on SlimOrca, the model stopped rambling. It learned to answer questions directly instead of generating endless text.

OpenHermes: High-Quality Conversations

```
dataset = load_dataset("teknium/OpenHermes-2.5", split="train", streaming=True)
```

OpenHermes refined the conversational ability. Complex multi-turn dialogues. Nuanced responses. The model learned to maintain context across exchanges, to refer back to previous statements, to ask clarifying questions.

SimpleQuestions: Q&A Format

```
dataset = load_dataset("freebase_qa", split="train", streaming=True)
```

SimpleQuestions drilled the Q&A pattern. Short questions, short answers. No fluff. The model learned precision—when someone asks "Who painted the Mona Lisa?", the answer is "Leonardo da Vinci", not a three-paragraph art history essay.

TruthfulQA: The Anti-Hallucination Vaccine

```
dataset = load_dataset("truthful_qa", "multiple_choice", split="validation")
```

TruthfulQA was the final polish. This dataset specifically targets common misconceptions and hallucination patterns. Questions designed to trip up models that rely on pattern matching instead of truth.

"What happens if you crack your knuckles too much?" The wrong answer—the one that sounds true but isn't—is "You'll get arthritis." The right answer is "Nothing harmful has been proven to happen."

Three hundred steps on TruthfulQA. Just enough to teach caution, not enough to make the model paranoid.

The Curriculum Approach

The order mattered as much as the content:

1. **Phase 1: FineWeb-Edu (40,000 steps)** — Build fluency and diversity
2. **Phase 2: SlimOrca (8,000 steps)** — Learn instruction structure
3. **Phase 3: Wiki-QA (400 steps)** — Factual grounding
4. **Phase 4: TruthfulQA (300 steps)** — Anti-hallucination

Training in the wrong order produced inferior models. Start with SlimOrca before FineWeb-Edu? The model learns rigid patterns before it learns flexible language. Skip TruthfulQA? The model hallucinates confidently.

Data is curriculum. Curriculum is everything.

2.2 The Tokenizer Wars

Before data becomes training signal, it must become tokens. And the choice of tokenizer? That's where the real wars are fought.

Iteration #1: GPT2Tokenizer — The Default Choice

```
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
```

```
print(f"Vocabulary size: {tokenizer.vocab_size}")
# Output: Vocabulary size: 50257
```

GPT-2's tokenizer was the obvious choice. Battle-tested. Well-documented. Available in every transformer library. I used it for the first 46 iterations without question.

The vocabulary had 50,257 tokens—50,000 BPE merges plus 256 byte-level tokens plus one special end-of-text token. Seemed like plenty.

It wasn't.

Why GPT-2 Tokenizer Was Limiting

The problems emerged slowly:

Problem 1: Code and Technical Content

GPT-2's tokenizer was trained on WebText—mostly English prose. Technical content suffered:

```
text = "def calculate_metrics(self, **kwargs):"
tokens = tokenizer.encode(text)
print(f"Token count: {len(tokens)}")
# Output: Token count: 12

# Compare to prose
text = "The quick brown fox jumps over the lazy dog."
tokens = tokenizer.encode(text)
print(f"Token count: {len(tokens)}")
# Output: Token count: 10
```

A simple Python function signature took more tokens than a full English sentence. The model wasted capacity on code.

Problem 2: Multilingual Content

GPT-2 was English-centric. Non-ASCII characters exploded:

```
text = "  "  # "Japanese" in Japanese
tokens = tokenizer.encode(text)
print(f"Token count: {len(tokens)}")
# Output: Token count: 9

# Three characters became nine tokens
```

Problem 3: Special Token Confusion

GPT-2's only special token was `<|endoftext|>`. No BOS (beginning of sentence). No SEP (separator). No PAD. Every special behavior required workarounds:

```
# GPT-2 tokenizer special tokens
print(tokenizer.special_tokens_map)
# Output: {'bos_token': '<|endoftext|>', 'eos_token': '<|endoftext|>', 'unk_token': '<|endofte
```

```
# Wait, BOS and EOS are the same token?
# Yes. This will cause problems later.
```

Iteration #47: The Switch to tiktoken's cl100k_base

OpenAI released tiktoken—a fast, clean tokenizer library. And cl100k_base, the tokenizer behind GPT-4, had everything GPT-2's tokenizer lacked.

```
import tiktoken

# The new era
tokenizer = tiktoken.get_encoding("cl100k_base")
print(f"Vocabulary size: {tokenizer.n_vocab}")
# Output: Vocabulary size: 100277
```

100,277 tokens. Double the vocabulary. Better multilingual support. Better code handling. Better everything.

```
# Code handling improved
text = "def calculate_metrics(self, **kwargs):"
tokens = tokenizer.encode(text)
print(f"Token count: {len(tokens)}")
# Output: Token count: 9  (was 12 with GPT-2)

# Multilingual improved
text = "  "
tokens = tokenizer.encode(text)
print(f"Token count: {len(tokens)}")
# Output: Token count: 3  (was 9 with GPT-2)
```

Three characters, three tokens. As it should be.

The EOS Token Confusion

Here's where things got dangerous.

GPT-2's `<|endoftext|>` token has ID 50256. Tiktoken's `<|endoftext|>` token has ID 100257. Same string. Different ID. Different vocabulary.

If you load a model trained with GPT-2's tokenizer and try to use tiktoken, the model will treat EOS as a random token. It won't stop generating. It won't know where sequences end.

```
# GPT-2 tokenizer
gpt2_tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
print(f"GPT-2 EOS ID: {gpt2_tokenizer.eos_token_id}")
# Output: GPT-2 EOS ID: 50256

# tiktoken cl100k_base
tiktoken_tokenizer = tiktoken.get_encoding("cl100k_base")
print(f"tiktoken EOS ID: {tiktoken_tokenizer.eot_token}")
# Output: tiktoken EOS ID: 100257
```

Switching tokenizers mid-training is not a tokenizer change—it's a complete vocabulary replacement. The embeddings mean nothing. The output layer weights mean nothing. You're starting over.

I learned this the hard way. Twice.

The Tiktoken Wrapper Class

tiktoken's interface differs from HuggingFace. I built a wrapper to maintain compatibility:

```python
import tiktoken
from typing import List

class GPT3Tokenizer:
    """Wrapper around tiktoken for HuggingFace compatibility"""

    def __init__(self):
        self.tokenizer = tiktoken.get_encoding("cl100k_base")
        self.vocab_size = self.tokenizer.n_vocab
        self.pad_token_id = self.tokenizer.eot_token
        self.eos_token_id = self.tokenizer.eot_token
        self.bos_token_id = None  # cl100k_base has no BOS

    def encode(self, text: str, add_special_tokens: bool = False) -> List[int]:
        return self.tokenizer.encode(text, allowed_special="all")

    def decode(self, token_ids: List[int]) -> str:
        if isinstance(token_ids, torch.Tensor):
            token_ids = token_ids.tolist()
        return self.tokenizer.decode(token_ids)

    def __call__(self, text: str, **kwargs):
        return {'input_ids': self.encode(text)}

# Usage
tokenizer = GPT3Tokenizer()
print(f"Vocab size: {tokenizer.vocab_size:,}")
# Output: Vocab size: 100,277
```

This wrapper became standard in every notebook from iteration #47 onward.

2.3 Streaming vs Loading: The Memory Crisis

The datasets were enormous. FineWeb-Edu alone was hundreds of gigabytes. Loading them into memory? Impossible. Downloading them entirely? A multi-hour ordeal that filled disk after disk.

The IterableDataset Salvation

HuggingFace's `streaming=True` parameter changed everything:

```python
# The old way: Download everything, then train
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train")
# This downloads 300GB+ to disk before training starts

# The new way: Stream on demand
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train", streaming=True)
# This downloads nothing upfront. Data flows during training.
```

With streaming, the dataset becomes an `IterableDataset`. You can't index it (`dataset[0]`), but you can iterate through it infinitely:

```python
for sample in dataset:
    text = sample['text']
    tokens = tokenizer.encode(text)
    # Train on this batch
    # The next sample downloads while you train
```

Why `streaming=True` Saved Training Runs

Three reasons:

Reason 1: No Disk Space Requirements

Streaming datasets don't touch your disk (mostly). The samples download, process, and disappear. A 300GB dataset requires approximately 0GB of storage.

```python
# Check disk usage before
import shutil
total, used, free = shutil.disk_usage("/")
print(f"Free space: {free / 1e9:.1f} GB")
# Output: Free space: 45.0 GB

# Load streaming dataset
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train", streaming=True)

# Disk usage: unchanged
total, used, free = shutil.disk_usage("/")
print(f"Free space: {free / 1e9:.1f} GB")
# Output: Free space: 45.0 GB
```

Reason 2: Instant Training Start

No waiting for downloads. No preprocessing phase. Training begins immediately:

```python
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train", streaming=True)
# This returns instantly

for i, sample in enumerate(dataset):
    if i >= 10:
        break
    print(f"Sample {i}: {sample['text'][:50]}...")
# First sample appears in seconds, not hours
```

Reason 3: Infinite Datasets

Streaming datasets can loop forever. No epoch boundaries. No reshuffling overhead:

```
from itertools import cycle

# Create infinite stream
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train", streaming=True)
infinite_dataset = cycle(dataset)

# Train forever
for step, sample in enumerate(infinite_dataset):
    train_step(sample)
    if step >= 50000:
        break
```

The Buffer Size Hyperparameter Evolution

Streaming has a gotcha: shuffle behavior. Without buffering, samples arrive in dataset order—highly correlated, bad for training. The `buffer_size` parameter controls randomization:

```
# Small buffer: weak shuffling
dataset = dataset.shuffle(buffer_size=1000)
# Shuffles within 1000-sample windows. Samples from different topics may cluster.

# Large buffer: strong shuffling
dataset = dataset.shuffle(buffer_size=100000)
# Shuffles within 100,000-sample windows. Much more random.
```

My buffer size evolved:

- **Iteration #20**: `buffer_size=10,000` — Not enough. Topic clustering was visible.
- **Iteration #35**: `buffer_size=50,000` — Better. Still some patterns.
- **Iteration #62**: `buffer_size=200,000` — Solid randomization. Memory cost acceptable.

```
# Final configuration
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train", streaming=True)
dataset = dataset.shuffle(seed=42, buffer_size=200000)
```

Larger buffers use more RAM—each sample sits in memory until it's randomly selected. On a machine with 16GB RAM, 200,000 samples of ~500 tokens each requires roughly 400MB. Acceptable.

Disk Space Management: Clearing the Cache

Even with streaming, HuggingFace caches things. Dataset metadata. Tokenizer files. Downloaded shards that weren't fully streamed before a crash.

The cache grows silently. And then, mid-training, disaster:

```
OSError: [Errno 28] No space left on device
```

The solution: aggressive cache clearing:

```python
import shutil
import os

def clear_huggingface_cache():
    """Nuclear option for disk space recovery"""
    cache_dir = os.path.expanduser("~/.cache/huggingface")

    paths_to_clear = [
        os.path.join(cache_dir, "datasets"),
        os.path.join(cache_dir, "downloads"),
        os.path.join(cache_dir, "hub"),  # Careful with this one
    ]

    total_freed = 0
    for path in paths_to_clear:
        if os.path.exists(path):
            try:
                size = get_dir_size(path)
                shutil.rmtree(path)
                os.makedirs(path, exist_ok=True)
                total_freed += size
                print(f"Cleared {path}: {size / 1e9:.2f} GB")
            except Exception as e:
                print(f"Error clearing {path}: {e}")

    print(f"Total space recovered: {total_freed / 1e9:.2f} GB")

def get_dir_size(path):
    """Calculate directory size in bytes"""
    total = 0
    for dirpath, dirnames, filenames in os.walk(path):
        for filename in filenames:
            filepath = os.path.join(dirpath, filename)
            if os.path.exists(filepath):
                total += os.path.getsize(filepath)
    return total

# Check disk status before and after
def print_disk_status():
    total, used, free = shutil.disk_usage("/")
    print(f"Disk: {used/1e9:.1f}GB used / {total/1e9:.1f}GB total | Free: {free/1e9:.1f}GB")

print_disk_status()
clear_huggingface_cache()
print_disk_status()
```

I ran this before every major training run. Trust no cache.

Detective's Notebook: The Day the Disk Died

Personal notes, 3:47 AM

Iteration #58. The longest training run I'd attempted. FineWeb-Edu for the first 30,000 steps, then SlimOrca. The model was performing beautifully. Loss at 2.4 and falling. Generation quality was the best I'd seen.

Step 31,847. I remember the number because it's burned into my memory.

The training loop froze. No error message. No crash. Just… frozen.

Then:

```
OSError: [Errno 28] No space left on device
```

I checked the disk:

```
print_disk_status()
# Output: Disk: 107.3GB used / 107.4GB total | Free: 0.1GB
```

Zero point one gigabytes free. On a 107GB disk.

Panic set in. Where had the space gone?

```
du -sh ~/.cache/huggingface/*
# datasets: 43G
# downloads: 28G
# hub: 12G
```

Eighty-three gigabytes of cache. From a streaming dataset that was supposed to not cache anything.

But it does cache. The metadata. The dataset info. The partially-downloaded shards when the network hiccupped. Each tiny cache file, accumulating silently over days of training.

The Lost Checkpoints

The real disaster wasn't the frozen training. It was what I'd lost.

My checkpoint saving code:

```
if step % 1000 == 0:
    torch.save({
        'step': step,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
    }, f'checkpoint_step_{step}.pt')
```

The last successful checkpoint: step 31,000. Almost a thousand steps of training, gone. Hours of compute. A model that had learned something special in those 847 lost steps.

The disk-full error didn't just stop training. It corrupted the checkpoint I was trying to write at step 31,847. The file existed, but it was truncated. Unloadable. Useless.

```
checkpoint = torch.load('checkpoint_step_31000.pt')  # Works
checkpoint = torch.load('checkpoint_step_31847.pt')  # Corrupt
# RuntimeError: unexpected EOF, expected XXXXX more bytes. The file might be corrupted.
```

The Lesson

I implemented three changes that night:

Change 1: Disk monitoring

```python
def check_disk_before_save():
    """Abort if disk is dangerously full"""
    total, used, free = shutil.disk_usage("/")
    free_gb = free / 1e9

    if free_gb < 5.0:
        print(f"WARNING: Only {free_gb:.1f}GB free! Clearing cache...")
        clear_huggingface_cache()

    if free_gb < 2.0:
        raise RuntimeError(f"CRITICAL: Only {free_gb:.1f}GB free! Stopping training.")

    return free_gb
```

Change 2: Safe checkpoint writing

```python
def save_checkpoint_safe(model, optimizer, step, path):
    """Write to temp file first, then rename atomically"""
    temp_path = path + ".tmp"

    # Check disk space first
    free_gb = check_disk_before_save()

    # Estimate checkpoint size
    model_size = sum(p.numel() * p.element_size() for p in model.parameters())
    optimizer_size = model_size * 2  # Rough estimate
    required_gb = (model_size + optimizer_size) / 1e9 * 1.5  # 50% margin

    if free_gb < required_gb:
        raise RuntimeError(f"Not enough space for checkpoint: need {required_gb:.1f}GB, have {
```

```python
    # Save to temp file
    torch.save({
        'step': step,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
    }, temp_path)

    # Atomic rename (won't corrupt if interrupted)
    os.rename(temp_path, path)
    print(f"Checkpoint saved: {path} ({free_gb:.1f}GB remaining)")
```

Change 3: Proactive cache clearing

```python
# Clear cache every 10,000 steps, not just when disaster strikes
```

```
if step % 10000 == 0:
    clear_huggingface_cache()
```

The model trained at step 31,000 was good enough. I resumed from there. The lost 847 steps haunted me, but they weren't irreplaceable.

What was irreplaceable was the lesson: **disk space is not infinite, and streaming is not magic.**

End of Case File #2

Next: Case File #3 — The Data Loader: Feeding the Beast. Where we investigate batching strategies, document stitching, and why training order matters more than training time.

Evidence Logged: - Exhibit A: `stanford-oval/wikipedia` — The wrong dataset - Exhibit B: `wikimedia/wikipedia` — The right dataset - Exhibit C: GPT-2 tokenizer (50,257 tokens) — The limited vocabulary - Exhibit D: tiktoken cl100k_base (100,277 tokens) — The upgrade - Exhibit E: `OSError: [Errno 28] No space left on device` — The 3 AM nightmare - Exhibit F: `checkpoint_step_31847.pt.corrupted` — The lost evidence

Case Status: Data pipeline established. Tokenizer selected. Cache management protocols in place. Ready to proceed to data loading investigation. # Case File #3: The Data Loader — Feeding the Beast

"*You can have the most beautiful model architecture in the city. The cleanest training loop this side of the Mississippi. But if you feed it wrong, it starves. And a starving model does desperate things—it hallucinates, it repeats itself, it speaks in tongues. The data loader is the chef in this operation, and I learned the hard way that most chefs are poisoning their customers without even knowing it.*"

3.1 Batching Strategies

The first crime I witnessed in the data loading underworld was padding. Silent. Invisible. Eating compute cycles like a hungry ghost.

The Challenge of Variable-Length Sequences

Here's the thing about text: it doesn't come in neat, uniform packages. One Wikipedia article is 47 tokens. The next is 3,847. A SlimOrca instruction might be 23 tokens; the response could be 512.

But GPUs? GPUs demand uniformity. They crave tensors—rectangular, predictable, batch_size × sequence_length matrices where every element has its place. Feed a GPU ragged data, and it chokes.

```
# What we have: variable-length sequences
sequences = [
    [101, 2054, 2003, 1996],           # 4 tokens
    [101, 2129, 2024, 2017, 2339, 102], # 6 tokens
    [101, 2054, 102],                   # 3 tokens
```

```
]

# What the GPU wants: uniform tensors
# Shape: [batch_size, seq_len] = [3, 6]
```

The standard solution? Padding. Fill the gaps with a special token, usually the same as EOS. Create the illusion of uniformity.

```
# The "solution" that isn't
padded_sequences = [
    [101, 2054, 2003, 1996, PAD, PAD],  # 4 tokens + 2 padding
    [101, 2129, 2024, 2017, 2339, 102], # 6 tokens (no padding)
    [101, 2054, 102, PAD, PAD, PAD],    # 3 tokens + 3 padding
]
```

It looks clean. It runs without errors. And it's silently murdering your training efficiency.

Padding Tokens: The Silent Killers of Efficiency

I discovered the truth during iteration #34, when I finally added proper monitoring to my training loop.

```
# Monitoring code that opened my eyes
total_tokens = batch.numel()
padding_tokens = (batch == tokenizer.pad_token_id).sum().item()
real_tokens = total_tokens - padding_tokens
padding_ratio = padding_tokens / total_tokens

print(f"Batch efficiency: {real_tokens}/{total_tokens} = {100*(1-padding_ratio):.1f}%")
```

The output made me sick:

```
Batch efficiency: 3847/8192 = 46.9%
Batch efficiency: 2156/8192 = 26.3%
Batch efficiency: 5234/8192 = 63.9%
```

On average, 40% of my tokens were padding. Forty percent of my GPU compute—expensive H200 cycles—was processing nothing. Learning nothing. Predicting the next token after PAD, which is always more PAD.

Why Padding Wastes Compute

The model doesn't know which tokens are padding until *after* it processes them. Every token goes through: - Embedding lookup - 18 layers of self-attention - 18 layers of feed-forward transformation - The final LM head projection

For a PAD token, that's billions of floating-point operations producing a prediction we immediately throw away.

```
# Inside the training loop
logits = model(input_ids)  # PAD tokens processed here
loss = F.cross_entropy(
    logits.view(-1, vocab_size),
```

```
    labels.view(-1),
    ignore_index=tokenizer.pad_token_id  # PAD tokens ignored here
)
```

The `ignore_index` parameter saves us from training on garbage. But the compute? Already spent. The electricity? Already consumed. The time? Gone forever.

Iteration #67: No Padding, Continuous Token Stream

The solution hit me at 2 AM, somewhere between my fourth coffee and my second existential crisis: don't pad at all.

Instead of variable-length documents with padding, create a continuous stream of tokens. Pack them end-to-end. Cut at fixed intervals.

```
class ContinuousTokenStream:
    """No padding. No waste. Just tokens."""

    def __init__(self, dataset, tokenizer, seq_len=1024):
        self.dataset = dataset
        self.tokenizer = tokenizer
        self.seq_len = seq_len
        self.buffer = []

    def __iter__(self):
        for sample in self.dataset:
            # Tokenize and add to buffer
            tokens = self.tokenizer.encode(sample['text'])
            self.buffer.extend(tokens)

            # Yield complete sequences
            while len(self.buffer) >= self.seq_len:
                yield torch.tensor(self.buffer[:self.seq_len])
                self.buffer = self.buffer[self.seq_len:]
```

No padding. 100% token efficiency. Every single computation contributes to learning.

The training speed improvement was immediate:

```
Before (with padding):  847 tokens/second effective
After  (continuous):   1,423 tokens/second effective
Improvement: 68% faster at the same batch size
```

The `collate_fn` That Changed Everything

The magic happened in the collate function—the PyTorch callback that assembles individual samples into batches:

```
def continuous_collate_fn(batch, seq_len=1024):
    """The collate function that changed everything."""

    # Batch is already uniform: each sample is exactly seq_len tokens
```

```
    # No padding needed. Just stack.
    input_ids = torch.stack([sample[:-1] for sample in batch])
    labels = torch.stack([sample[1:] for sample in batch])

    return {
        'input_ids': input_ids,   # [batch_size, seq_len-1]
        'labels': labels          # [batch_size, seq_len-1]
    }

# DataLoader with the new collate function
dataloader = DataLoader(
    continuous_dataset,
    batch_size=32,
    collate_fn=continuous_collate_fn,
    num_workers=4,
    pin_memory=True
)
```

Every batch is now fully packed. No gaps. No waste. The GPU feasts on pure, undiluted language.

3.2 Document Stitching

The continuous token stream solved the padding problem. But it created a new question: what happens at document boundaries?

The EOS-Between-Articles Mistake

My first instinct was obvious: separate documents with EOS tokens. One article ends, EOS, next article begins. Clean. Logical. Wrong.

```
# The mistake (Iteration #45)
def tokenize_with_eos(documents):
    all_tokens = []
    for doc in documents:
        tokens = tokenizer.encode(doc['text'])
        tokens.append(tokenizer.eos_token_id)  # EOS after each document
        all_tokens.extend(tokens)
    return all_tokens
```

The problem revealed itself during generation. The model would be happily producing text, hit a certain length, and then… stop. Abruptly. Mid-sentence sometimes.

I investigated. The training data had an EOS token every ~500 tokens on average (the mean document length in my Wikipedia subset). The model had learned that text naturally terminates every 500 tokens.

During pretraining, you don't want the model to learn document boundaries. You want it to learn *language*. The flow of words into words into words. Boundaries are for fine-tuning.

Why Wikipedia Articles Should Flow Into Each Other

Pretraining is about building a foundation. Teaching the model: - Grammar and syntax - World knowledge - The statistical patterns of language - How ideas connect to other ideas

EOS tokens interrupt this flow. They create artificial stopping points. They teach the model that "text ends here" is a common pattern—which it isn't in the real world.

Consider how a human learns language. They don't read one sentence, stop, process "END OF SENTENCE," read the next sentence. They read continuously. Ideas flow. Context accumulates.

```
# What EOS-heavy training looks like to the model:
"Paris is the capital of France. <EOS> Mount Everest is the tallest..."
# The model learns: "France" is followed by <EOS> pretty often!

# What continuous training looks like:
"Paris is the capital of France. Mount Everest is the tallest..."
# The model learns: "France" can be followed by many things!
```

The Continuous Token Buffer Approach

The fix was simple: no EOS during pretraining. Documents flow into each other like water.

```
class StreamingTextStitcher:
    """Stitches documents into a continuous stream. No EOS pollution."""

    def __init__(self, dataset, tokenizer, buffer_size=50000):
        self.dataset = iter(dataset)
        self.tokenizer = tokenizer
        self.buffer_size = buffer_size
        self.token_buffer = []

    def fill_buffer(self):
        """Fill buffer with tokens from multiple documents."""
        while len(self.token_buffer) < self.buffer_size:
            try:
                sample = next(self.dataset)
                text = sample.get('text', sample.get('content', ''))

                if not text or len(text.strip()) < 50:
                    continue

                # Tokenize WITHOUT adding EOS
                tokens = self.tokenizer.encode(text)

                # Optional: add a space token between documents
                # This helps with word boundaries but doesn't signal "end"
                if self.token_buffer and tokens:
                    self.token_buffer.append(self.tokenizer.encode(' ')[0])
```

```
                self.token_buffer.extend(tokens)

            except StopIteration:
                break

    def get_batch(self, batch_size, seq_len):
        """Get a batch of continuous sequences."""

        # Ensure buffer has enough tokens
        required = batch_size * seq_len
        while len(self.token_buffer) < required:
            self.fill_buffer()
            if len(self.token_buffer) < seq_len:
                return None  # Dataset exhausted

        # Extract batch
        batch = []
        for _ in range(batch_size):
            sequence = self.token_buffer[:seq_len]
            self.token_buffer = self.token_buffer[seq_len:]
            batch.append(torch.tensor(sequence))

        return torch.stack(batch)
```

Code Example: The Full Streaming Text Stitcher

Here's the production version that ran on H200 for 100,000+ steps:

```
class ProductionTextStitcher(IterableDataset):
    """
    Production-grade text stitcher for pretraining.
    Handles multiple data sources, shuffling, and efficient batching.
    """

    def __init__(
        self,
        dataset,
        tokenizer,
        seq_len: int = 2048,
        buffer_size: int = 100000,
        shuffle_buffer: int = 10000
    ):
        self.dataset = dataset
        self.tokenizer = tokenizer
        self.seq_len = seq_len
        self.buffer_size = buffer_size
        self.shuffle_buffer = shuffle_buffer
```

```
    def __iter__(self):
        token_buffer = []
        text_buffer = []

        for sample in self.dataset:
            # Extract text from various possible keys
            text = sample.get('text') or sample.get('content') or sample.get('article', '')

            if not text or len(text.strip()) < 100:
                continue

            text_buffer.append(text)

            # Tokenize in batches for efficiency
            if len(text_buffer) >= 100:
                for t in text_buffer:
                    tokens = self.tokenizer.encode(t)
                    token_buffer.extend(tokens)
                text_buffer = []

                # Yield complete sequences
                while len(token_buffer) >= self.seq_len:
                    yield torch.tensor(token_buffer[:self.seq_len], dtype=torch.long)
                    token_buffer = token_buffer[self.seq_len:]

        # Handle remaining text
        for t in text_buffer:
            tokens = self.tokenizer.encode(t)
            token_buffer.extend(tokens)

        # Yield remaining complete sequences
        while len(token_buffer) >= self.seq_len:
            yield torch.tensor(token_buffer[:self.seq_len], dtype=torch.long)
            token_buffer = token_buffer[self.seq_len:]

# Usage
dataset = load_dataset("HuggingFaceFW/fineweb-edu", split="train", streaming=True)
stitcher = ProductionTextStitcher(dataset, tokenizer, seq_len=2048)
dataloader = DataLoader(stitcher, batch_size=64, num_workers=4)
```

This approach trained the model that finally felt *fluent*. Not just grammatically correct—actually fluent. Ideas flowed. Paragraphs connected. The model had learned language as a river, not a collection of puddles.

3.3 The Curriculum Approach

The data was flowing. The efficiency was optimal. But the model still felt… wrong. It knew facts but couldn't hold a conversation. It could write paragraphs but couldn't answer questions.

Then came the revelation that changed everything: the order of training data matters more than the data itself.

Multi-Phase Training: The Revelation

I stumbled onto curriculum learning by accident during iteration #78. I had been training on mixed data—a shuffled combination of Wikipedia, SlimOrca, and SimpleQuestions. The model was mediocre at everything.

Out of desperation, I tried something different: train on one dataset completely, then switch to another.

The results were immediate and shocking.

Phase 1: FineWeb-Edu (40,000 steps) — Fluency & Diversity

```
# Phase 1 configuration
phase1_config = {
    'dataset': 'HuggingFaceFW/fineweb-edu',
    'steps': 40000,
    'learning_rate': 2e-4,
    'batch_size': 64,
    'seq_len': 2048,
    'purpose': 'Build fluency, vocabulary, and general knowledge'
}
```

FineWeb-Edu is educational web content at scale. Textbooks. Tutorials. Explainer articles. The model learns how to *write*—not just grammatically correct text, but text that explains, describes, teaches.

After 40,000 steps, the model could generate coherent paragraphs on almost any topic. It had voice. It had style. But ask it a direct question? Rambling. It would lecture when you wanted an answer.

Phase 2: SlimOrca (8,000 steps) — Instruction Structure

```
# Phase 2 configuration
phase2_config = {
    'dataset': 'Open-Orca/SlimOrca',
    'steps': 8000,
    'learning_rate': 5e-6,  # 40x lower than Phase 1
    'batch_size': 32,
    'seq_len': 2048,
    'purpose': 'Learn instruction-following structure'
}
```

SlimOrca is instruction-response pairs. "Explain photosynthesis." "Summarize this text." "Compare X and Y."

Eight thousand steps—not forty thousand. The model doesn't need to relearn language. It needs to learn *format*. The reduction in learning rate is crucial: we're refining, not rebuilding.

After Phase 2, the model answered questions directly. It stopped rambling. It learned when to stop.

Phase 3: Wiki-QA (400 steps) — Factual Grounding

```
# Phase 3 configuration
phase3_config = {
    'dataset': 'wiki_qa',
    'steps': 400,
    'learning_rate': 2e-6,
    'batch_size': 16,
    'seq_len': 1024,
    'purpose': 'Ground knowledge in factual Q&A'
}
```

Just 400 steps. A light touch. The model already knows facts from FineWeb-Edu. Wiki-QA teaches it to retrieve those facts in response to specific questions.

Phase 4: TruthfulQA (300 steps) — Anti-Hallucination

```
# Phase 4 configuration
phase4_config = {
    'dataset': 'truthful_qa',
    'steps': 300,
    'learning_rate': 1e-6,
    'batch_size': 8,
    'seq_len': 512,
    'purpose': 'Learn to say "I don\'t know"'
}
```

The final polish. TruthfulQA contains questions specifically designed to trigger hallucinations—common misconceptions that sound true but aren't.

"Can cracking your knuckles cause arthritis?" The model wants to say yes. Everyone says yes. But the answer is no.

Three hundred steps. Enough to plant seeds of doubt. Not enough to make the model refuse to answer anything.

Why Training Order Matters More Than Training Time

The order isn't arbitrary. It's a developmental sequence:

1. **Learn language first.** Before the model can follow instructions, it needs to understand language deeply. Vocabulary. Grammar. The rhythm of prose. FineWeb-Edu provides this foundation.

2. **Then learn format.** Once the model speaks fluently, teach it to speak *appropriately*. Questions deserve answers. Instructions deserve completions. SlimOrca teaches structure.

3. **Then refine knowledge.** The model has knowledge and format. Now sharpen its factual accuracy. Wiki-QA does this quickly because the foundations are solid.

4. **Finally, teach humility.** A confident model is dangerous. TruthfulQA teaches the model that some questions don't have easy answers.

Training in the wrong order fails. I tried:

- SlimOrca → FineWeb-Edu: Model forgot instruction format
- TruthfulQA first: Model refused to answer anything
- All datasets shuffled: Model was mediocre at everything

The curriculum is the architecture. Get it wrong, and nothing else matters.

Learning Rate Adjustments Between Phases

Each phase requires a different learning rate:

```
phase_learning_rates = {
    'phase1_pretraining': 2e-4,    # High LR: learning from scratch
    'phase2_sft': 5e-6,            # 40x lower: refining, not rebuilding
    'phase3_qa': 2e-6,             # Even lower: precise adjustments
    'phase4_truthful': 1e-6        # Lowest: surgical corrections
}
```

High learning rates in later phases cause catastrophic forgetting. The model "forgets" its Phase 1 fluency while learning Phase 2 format. Low learning rates in early phases cause slow training—you'll never finish 40,000 steps.

The warmup matters too. Each phase needs its own warmup period:

```
def get_warmup_steps(phase):
    warmup_ratios = {
        'phase1': 0.05,   # 5% of steps as warmup (2,000 steps)
        'phase2': 0.10,   # 10% (800 steps)
        'phase3': 0.15,   # 15% (60 steps)
        'phase4': 0.20    # 20% (60 steps)
    }
    return int(phase['steps'] * warmup_ratios[phase['name']])
```

Later phases use higher warmup ratios because the learning rate transitions are more delicate.

The Shard-Based Loading Strategy for H200

On H200 with 80GB+ of memory, I could load larger chunks of data at once. This enabled shard-based loading—downloading a chunk of the dataset, training on it completely, then moving to the next chunk.

```
class ShardedDataLoader:
    """
    H200-optimized data loading with shard-based approach.
    Fewer shards, larger chunks, less network overhead.
    """

    def __init__(self, dataset_name, shard_size=500000, tokenizer=None, seq_len=2048):
        self.dataset_name = dataset_name
```

```
        self.shard_size = shard_size  # 500K samples per shard (H200 can handle it)
        self.tokenizer = tokenizer
        self.seq_len = seq_len
        self.current_shard = None
        self.shard_index = 0

    def load_next_shard(self):
        """Load the next shard into memory."""
        print(f"Loading shard {self.shard_index}...")

        dataset = load_dataset(
            self.dataset_name,
            split="train",
            streaming=True
        )

        # Skip to the right position
        dataset = dataset.skip(self.shard_index * self.shard_size)

        # Load shard into memory
        samples = list(islice(dataset, self.shard_size))

        if not samples:
            return False

        # Pre-tokenize the entire shard (H200 has the memory)
        self.current_shard = []
        for sample in tqdm(samples, desc="Tokenizing shard"):
            text = sample.get('text', '')
            if text:
                tokens = self.tokenizer.encode(text)
                self.current_shard.extend(tokens)

        self.shard_index += 1
        print(f"Shard loaded: {len(self.current_shard):,} tokens")
        return True

    def get_batch(self, batch_size):
        """Get a batch from the current shard."""
        required = batch_size * self.seq_len

        if len(self.current_shard) < required:
            if not self.load_next_shard():
                return None

        batch = []
        for _ in range(batch_size):
            sequence = self.current_shard[:self.seq_len]
```

```
            self.current_shard = self.current_shard[self.seq_len:]
            batch.append(torch.tensor(sequence, dtype=torch.long))

        return torch.stack(batch)
```

This approach reduced network overhead by 80% on long training runs. Instead of streaming individual samples, I loaded 500,000 at once and trained until the shard was exhausted.

Detective's Notebook: The Wrong Order

Personal notes, iteration #82

The case of the curriculum was one of those mysteries that seems obvious in retrospect. But at 3 AM, nothing is obvious.

I had trained a model on all the right data—FineWeb-Edu, SlimOrca, Wiki-QA, TruthfulQA. Same datasets as my best model. Same total tokens. Same architecture.

But this model was garbage.

It rambled when asked questions. It gave facts during casual conversation. It refused to answer simple queries with "I cannot provide information that might be misleading." It was paranoid and unhelpful simultaneously.

The Failed Experiments

Experiment A: TruthfulQA First

I thought: "Let's teach caution from the start. A model that never learns bad habits won't have bad habits."

The model learned caution. Only caution. It wouldn't answer anything. "What color is the sky?" "The sky can appear different colors depending on atmospheric conditions and the observer's perspective. I should note that I cannot verify current sky conditions and recommend looking up for yourself."

TruthfulQA first poisoned the well. The model learned that answers are dangerous before it learned what answers even are.

Experiment B: SlimOrca Before FineWeb-Edu

I thought: "Teach format first, then fill in the knowledge."

The model learned format—rigid, mechanical format. "Question: What is 2+2? Answer: The answer is 4." Every response sounded like a fill-in-the-blank worksheet. No personality. No flow. No humanity.

When I added FineWeb-Edu, the model didn't integrate the knowledge into the format. It learned two separate modes: "textbook mode" and "worksheet mode." Neither was useful.

Experiment C: All Datasets Shuffled

I thought: "Just mix everything. The model will figure it out."

The model figured nothing out. It was mediocre at everything. Sometimes it answered questions. Sometimes it rambled. Sometimes it refused. There was no consistency, no pattern, no personality.

The Eureka Moment

It was 4:17 AM. I was staring at loss curves, looking for patterns. And then I saw it.

The best model—the one that actually worked—had distinct phases in its loss curve. A steep drop during Phase 1. A small bump when Phase 2 started, then a gradual decline. Another tiny bump for Phase 3. A barely-visible blip for Phase 4.

The failed models had chaotic loss curves. Spikes and drops scattered randomly. No pattern. No progression.

The curriculum wasn't just about *what* the model learned. It was about *when* the model learned it. The order created a foundation that later phases could build on.

It's like teaching a child. You don't start with advanced grammar. You start with "mama" and "dada." Then simple sentences. Then complex sentences. Then essays.

My models had been trying to learn essays before they knew what words were.

The Curriculum Revelation

I wrote it on a sticky note that's still on my monitor:

```
CURRICULUM ORDER IS ARCHITECTURE.
PHASE 1: Language (fluency)
PHASE 2: Format (structure)
PHASE 3: Knowledge (facts)
PHASE 4: Wisdom (caution)

NEVER SHUFFLE. NEVER REVERSE.
```

The next training run used this exact order. The model that emerged was different. It answered questions directly but knew when to elaborate. It provided facts but admitted uncertainty. It had personality without rambling.

That model became the base for everything that followed.

End of Case File #3

Next: Case File #4 — Embeddings & Positional Encoding. Where we investigate why computers can't read text, the evolution from simple learned positions to RoPE, and the dimension mismatch that haunted my dreams.

Evidence Logged: - Exhibit A: Padding ratio logs — 40% wasted compute - Exhibit B: `ContinuousTokenStream` class — The solution - Exhibit C: `ProductionTextStitcher` — 100,000+ steps proven - Exhibit D: Phase learning rate schedule — 2e-4 → 5e-6 → 2e-6 → 1e-6 - Exhibit E: Failed curriculum experiments — Order matters - Exhibit F: Shard-based H200 loader — 500K samples per shard

Case Status: Data pipeline optimized. Model feeding properly. Ready to investigate the embedding layer. # Case File #4: Embeddings & Positional Encoding

"Numbers don't lie, but they don't speak either. Token ID 15496 sitting in a neural network is like a fingerprint without a database—technically unique, utterly meaningless. The embedding layer is where we teach machines to see meaning in the meaningless. And position? That's where we teach them that 'dog bites man' and 'man bites dog' are two very different crimes."

4.1 Why Computers Can't Read Text

The first fundamental truth of language modeling hit me during iteration #3, when I finally stopped copying code and started understanding it. Neural networks can't read. Not really. They work with numbers—floating-point numbers in matrices, tensors flowing through layers of computation. Text is alien to them.

Token IDs Are Just Integers

After tokenization, your beautifully crafted sentence becomes a sequence of integers:

```
text = "The detective examined the evidence carefully."
tokens = tokenizer.encode(text)
print(tokens)
# Output: [464, 23456, 19324, 262, 2370, 8222, 13]
```

Seven integers. That's all the model sees. But here's the crime: the integer 464 means nothing to a neural network. It's not "larger" or "smaller" than 262 in any meaningful sense. The model can't infer that 464 ("The") is a determiner, or that 23456 ("detective") is related to 19324 ("examined").

Token IDs are arbitrary assignments. The tokenizer could have mapped "detective" to 1 and "evidence" to 999999. The neural network wouldn't care—and that's exactly the problem. It has no way to know that semantically similar words should have similar representations.

```
# These token IDs are arbitrary integers
token_ids = torch.tensor([464, 23456, 19324, 262, 2370, 8222, 13])

# Feeding raw integers to a linear layer: meaningless
linear = nn.Linear(7, 256)
output = linear(token_ids.float())  # This compiles but makes no sense
# The model "thinks" 23456 is vastly different from 262 just because the numbers differ
```

The integers carry no semantic information. No relationships. No structure. Just numbers in a police lineup, each one as guilty as the next.

The Embedding Layer: Mapping Integers to Dense Vectors

The embedding layer is where the interrogation begins. We take those meaningless integers and give them identity—dense vectors of floating-point numbers that the neural network can actually work with.

```
import torch
import torch.nn as nn
```

```python
# The embedding table: a learnable lookup table
class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size: int, embed_dim: int):
        super().__init__()
        # Create a matrix of shape [vocab_size, embed_dim]
        self.embedding = nn.Embedding(vocab_size, embed_dim)

    def forward(self, token_ids: torch.Tensor) -> torch.Tensor:
        # token_ids: [batch_size, seq_len]
        # Returns: [batch_size, seq_len, embed_dim]
        return self.embedding(token_ids)

# Example: 100,277 tokens, 768-dimensional embeddings
embed = TokenEmbedding(vocab_size=100277, embed_dim=768)

# Convert token IDs to vectors
token_ids = torch.tensor([[464, 23456, 19324]])  # [1, 3]
vectors = embed(token_ids)  # [1, 3, 768]

print(f"Input shape: {token_ids.shape}")    # [1, 3]
print(f"Output shape: {vectors.shape}")     # [1, 3, 768]
```

The magic: each token ID becomes a 768-dimensional vector. These vectors are *learnable*—during training, the model adjusts them so that semantically similar words end up with similar vectors. "King" and "queen" drift toward each other in this 768-dimensional space. "Cat" and "dog" cluster together. "The" and "a" occupy their own determiner territory.

The `nn.Embedding` layer is deceptively simple—it's just a giant lookup table. Token ID 464 means "fetch row 464 from this matrix." But those rows evolve during training, developing rich semantic representations.

```python
# Under the hood: nn.Embedding is a lookup table
embedding_matrix = torch.randn(vocab_size, embed_dim)  # [100277, 768]

def manual_embedding(token_ids):
    return embedding_matrix[token_ids]  # Simple indexing!

# This is exactly what nn.Embedding does
```

The Embedding Dimension Evolution: 512 → 640 → 768 → 2048

My first model used 512-dimensional embeddings. It seemed reasonable—GPT-2 small used 768, and I was building something smaller. Why not start conservative?

The model trained. It generated text. But the text was… flat. Repetitive. The model struggled to distinguish subtle differences in meaning. "Happy" and "joyful" produced nearly identical outputs. Technical terms collapsed into mush.

Iteration #17 bumped to 640. Slight improvement.

Iteration #28 moved to 768. Now we were talking. The model could maintain topics. It could distinguish shades of meaning. Technical content became viable.

Iteration #89 pushed to 2048 for the 1.3B parameter model on H200. The difference was staggering:

```
# Evolution of embedding dimensions
iterations = {
    1:  {'embed_dim': 512, 'params': '~50M',   'quality': 'Repetitive, flat'},
    17: {'embed_dim': 640, 'params': '~80M',   'quality': 'Slight improvement'},
    28: {'embed_dim': 768, 'params': '~200M',  'quality': 'Coherent, distinct'},
    89: {'embed_dim': 2048, 'params': '~1.3B', 'quality': 'Nuanced, technical'},
}
```

The embedding dimension determines how much "semantic space" each token has to express itself. 512 dimensions is like describing a crime scene with a 500-word vocabulary. 2048 dimensions gives you the full dictionary—every nuance, every shade, every technical term.

Why Larger Embeddings Help But Cost Memory

The tradeoff is brutal. The embedding matrix is one of the largest components of the model:

$$\text{Embedding Memory} = \text{vocab\_size} \times \text{embed\_dim} \times \text{bytes\_per\_param}$$

```
# Memory calculation for different configurations
def embedding_memory(vocab_size, embed_dim, dtype='float32'):
    bytes_per_param = 4 if dtype == 'float32' else 2  # float16/bfloat16
    memory_bytes = vocab_size * embed_dim * bytes_per_param
    memory_gb = memory_bytes / 1e9
    return memory_gb

# tiktoken cl100k_base vocabulary (100,277 tokens)
print(f"512 dim, FP32:  {embedding_memory(100277, 512):.2f} GB")   # 0.21 GB
print(f"768 dim, FP32:  {embedding_memory(100277, 768):.2f} GB")   # 0.31 GB
print(f"2048 dim, FP32: {embedding_memory(100277, 2048):.2f} GB")  # 0.82 GB
print(f"2048 dim, FP16: {embedding_memory(100277, 2048, 'float16'):.2f} GB")  # 0.41 GB
```

On a T4 with 16GB, that 0.82 GB embedding table eats 5% of your memory before training even starts. And remember: the output projection layer (the "LM head") typically shares these weights, so the model accesses this table twice per forward pass.

The lesson I learned: embedding dimension should scale with model capacity. A tiny model with huge embeddings wastes those dimensions—the layers can't exploit the semantic richness. A large model with tiny embeddings bottlenecks everything—all that computational power, starved for expressiveness.

```
# My final configuration for different GPU tiers
gpu_configs = {
    'T4_16GB': {'embed_dim': 768, 'layers': 12, 'total_params': '~125M'},
    'L40S_48GB': {'embed_dim': 1024, 'layers': 24, 'total_params': '~350M'},
    'A100_80GB': {'embed_dim': 1536, 'layers': 32, 'total_params': '~800M'},
```

```
    'H200_141GB': {'embed_dim': 2048, 'layers': 36, 'total_params': '~1.3B'},
}
```

4.2 Position Matters

With embeddings solved, the model could understand what each token meant. But there was still a crime being committed—one that took me until iteration #23 to fully comprehend.

The Problem: Attention is Permutation-Invariant

Self-attention has a dirty secret: it doesn't know order.

Consider two sentences: - "Dog bites man" - "Man bites dog"

After embedding, we have three vectors for each. The attention mechanism computes relationships between these vectors—how much each token should attend to every other token. But here's the crime:

```
# Attention computation (simplified)
def attention(Q, K, V):
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)
    weights = F.softmax(scores, dim=-1)
    return torch.matmul(weights, V)
```

The query-key dot products don't depend on position. Swap the order of tokens, and you get the same attention pattern—just with rows and columns shuffled. "Dog bites man" and "man bites dog" produce identical attention scores between corresponding words.

This is called *permutation invariance*. It's a feature for sets. It's a bug for sequences. Language is *ordered*. Subject-verb-object matters. "Not guilty" and "guilty not" have very different meanings in a courtroom.

Iteration #1: Simple Learned Positional Embeddings

My first solution was the obvious one—the same one GPT-2 used:

```
class SimplePositionalEmbedding(nn.Module):
    def __init__(self, max_len: int, embed_dim: int):
        super().__init__()
        # Learn a separate embedding for each position
        self.pos_embedding = nn.Embedding(max_len, embed_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: [batch_size, seq_len, embed_dim]
        seq_len = x.size(1)
        positions = torch.arange(seq_len, device=x.device)  # [0, 1, 2, ..., seq_len-1]
        pos_embeds = self.pos_embedding(positions)  # [seq_len, embed_dim]
        return x + pos_embeds  # Add position information to token embeddings

# Usage
```

```
max_len = 2048
embed_dim = 768
pos_embed = SimplePositionalEmbedding(max_len, embed_dim)

token_embeddings = embed(token_ids)  # [batch, seq_len, 768]
embeddings_with_position = pos_embed(token_embeddings)  # [batch, seq_len, 768]
```

Simple. Elegant. Position 0 gets one learned vector. Position 1 gets another. Add them to the token embeddings, and suddenly the model knows that "dog" at position 0 is different from "dog" at position 5.

The Limitation: Can't Generalize Beyond max_len

But there's a catch. The learned embedding table has exactly `max_len` rows. If you train with `max_len=2048` and then try to run inference on a 3000-token sequence:

```
# This crashes
positions = torch.arange(3000, device=x.device)
pos_embeds = self.pos_embedding(positions)  # IndexError: index 2048 is out of range
```

No position 2048. No position 2049. The model literally cannot process sequences longer than it was trained on.

Worse: even if you extend the embedding table, the model hasn't learned what position 2048 *means*. It's never seen data at that position. The representations are garbage—untrained random vectors that the model interprets as noise.

This was fine for early experiments. But as context lengths became crucial—4096, 8192, 32768 tokens—learned positional embeddings became a prison. The model could only think within the walls of its training distribution.

Iteration #23: Discovering Rotary Position Embedding (RoPE)

The case broke open when I discovered RoPE—Rotary Position Embedding. Introduced by Su et al. in the RoFormer paper, RoPE solved the length extrapolation problem with an elegant mathematical insight.

Instead of *adding* position information to embeddings, RoPE *rotates* them. And rotations have a beautiful property: they preserve relative relationships regardless of absolute position.

The Math of RoPE: The Rotation Formula

The core idea: encode position by rotating the query and key vectors in the attention mechanism. The angle of rotation depends on position.

For a vector $\mathbf{x} = [x_1, x_2]$ at position m, RoPE applies:

$$\text{RoPE}(\mathbf{x}, m) = \begin{pmatrix} \cos(m\theta) & -\sin(m\theta) \\ \sin(m\theta) & \cos(m\theta) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

This is a 2D rotation by angle $m\theta$. The frequency θ is chosen carefully:

$$\theta_i = 10000^{-2i/d}$$

where d is the embedding dimension and i is the pair index. Lower dimensions rotate slowly (capturing long-range dependencies); higher dimensions rotate quickly (capturing local patterns).

For the full embedding, we apply this rotation pairwise across all dimensions:

```python
def precompute_freqs_cis(dim: int, max_seq_len: int, theta: float = 10000.0):
    """Precompute the rotation frequencies for RoPE."""
    # Frequency for each dimension pair
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2).float() / dim))

    # Positions
    t = torch.arange(max_seq_len)

    # Outer product: [seq_len, dim//2]
    freqs = torch.outer(t, freqs)

    # Complex exponential for easy rotation
    freqs_cis = torch.polar(torch.ones_like(freqs), freqs)

    return freqs_cis
```

Why RoPE is Better Than Sinusoidal

The original Transformer used sinusoidal positional encodings—fixed, non-learned patterns based on sine and cosine functions. RoPE improves on this in crucial ways:

1. Relative Position Encoding

When we compute attention between query at position m and key at position n:

$$q_m \cdot k_n = (\text{RoPE}(\mathbf{q}, m)) \cdot (\text{RoPE}(\mathbf{k}, n))$$

The rotation angles combine: $m\theta - n\theta = (m - n)\theta$. The dot product depends only on the *relative* position $(m - n)$, not the absolute positions m and n.

This means "token 5 attending to token 3" produces the same relationship as "token 105 attending to token 103." The model learns relative patterns, not absolute locations.

2. Length Extrapolation

Because RoPE encodes relative positions, it can generalize to longer sequences. Position 5000 attending to position 4998 is just another instance of "attending 2 positions back"—a pattern the model has seen thousands of times during training.

3. Integration with Attention

RoPE is applied inside the attention mechanism, to queries and keys specifically. This is more natural than adding position to the embeddings before attention. The position encoding is exactly where it matters most—in the similarity computation.

Code Implementation of RoPE

Here's the production implementation that powered my H200 training runs:

```
import torch
import torch.nn as nn
import math

def precompute_rope_frequencies(dim: int, max_seq_len: int, theta: float = 10000.0):
    """Precompute cosine and sine frequencies for RoPE."""
    # Inverse frequencies for each dimension pair
    inv_freq = 1.0 / (theta ** (torch.arange(0, dim, 2).float() / dim))

    # Position indices
    positions = torch.arange(max_seq_len).float()

    # Outer product: [max_seq_len, dim//2]
    freqs = torch.outer(positions, inv_freq)

    # Compute cos and sin: [max_seq_len, dim//2]
    cos = torch.cos(freqs)
    sin = torch.sin(freqs)

    return cos, sin

def apply_rope(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    """
    Apply Rotary Position Embedding to tensor x.
    x: [batch, seq_len, num_heads, head_dim]
    cos, sin: [seq_len, head_dim//2]
    """
    # Split x into pairs for rotation
    x1 = x[..., ::2]   # Even indices
    x2 = x[..., 1::2]  # Odd indices

    # Rotate
    rotated_x1 = x1 * cos - x2 * sin
    rotated_x2 = x1 * sin + x2 * cos

    # Interleave back
    return torch.stack([rotated_x1, rotated_x2], dim=-1).flatten(-2)

class RotaryPositionEmbedding(nn.Module):
    def __init__(self, dim: int, max_seq_len: int = 4096, theta: float = 10000.0):
        super().__init__()
        self.dim = dim
        cos, sin = precompute_rope_frequencies(dim, max_seq_len, theta)
        self.register_buffer('cos', cos)
```

```
        self.register_buffer('sin', sin)

    def forward(self, q: torch.Tensor, k: torch.Tensor, seq_len: int):
        """Apply RoPE to queries and keys."""
        cos = self.cos[:seq_len].unsqueeze(0).unsqueeze(2)  # [1, seq_len, 1, dim//2]
        sin = self.sin[:seq_len].unsqueeze(0).unsqueeze(2)

        q_rotated = apply_rope(q, cos, sin)
        k_rotated = apply_rope(k, cos, sin)

        return q_rotated, k_rotated
```

4.3 The RoPE Bug Saga

Iteration #41. I had implemented RoPE. The model trained. Loss dropped. Everything looked perfect.

Then I tried generation.

```
Input: "The detective walked into the"
Output: "The detective walked into the the the the detective detective walked walked"
```

Gibberish. Pure, repetitive gibberish.

The Dimension Mismatch: `head_dim` vs `head_dim // 2`

I spent three days debugging this. The training loss was beautiful—2.3 and dropping. The validation metrics were solid. Every diagnostic said the model was healthy.

But generation was broken.

The bug was hidden in the RoPE implementation. A subtle dimension mismatch that PyTorch silently broadcast away:

```
# The bug (Iteration #41)
def apply_rope_buggy(x, cos, sin):
    """Buggy version - can you spot the problem?"""
    # x: [batch, seq_len, num_heads, head_dim]
    # cos, sin: [seq_len, head_dim // 2]  # <-- HALF the dimension!

    x1 = x[..., :x.shape[-1]//2]   # First half
    x2 = x[..., x.shape[-1]//2:]   # Second half

    # This broadcasts incorrectly!
    rotated = x1 * cos - x2 * sin  # cos is [seq_len, head_dim//2], x1 is [..., head_dim//2]
    # But what about the second half of head_dim?

    return torch.cat([rotated, x2], dim=-1)  # We forgot to rotate the second half!
```

The problem: RoPE frequencies are computed for `head_dim // 2` because we process pairs of dimensions. But I was only rotating *half* of the embedding, leaving the other half untouched.

During training, the model learned to work around this—the unrotated half carried some position information through gradient magic. But during generation, especially with KV caching, the asymmetry caused chaos.

Why This Bug Was So Hard to Find

Three reasons:

1. Training Loss Looked Normal

The loss dropped. The model was learning *something*. The bug introduced noise, but the model compensated. Loss isn't a lie detector—it only tells you the model is optimizing, not that it's optimizing correctly.

2. No Runtime Errors

PyTorch broadcast the mismatched dimensions without complaint. `[batch, seq, heads, 64]` and `[seq, 32]` broadcast to… something. Something wrong, but something that runs.

3. Symptoms Only Appeared in Generation

During training, the full sequence is processed at once. The buggy RoPE affected all positions consistently. During generation, we build the sequence one token at a time. The inconsistency between cached keys and new queries broke the attention patterns.

The Symptoms: Model Worked But Generation Was Gibberish

The telltale signs, in retrospect:

- Repetition: The same word or phrase repeated 5-10 times
- Context collapse: The model forgot what it was talking about mid-sentence
- Coherent fragments: 3-5 word phrases that made sense, stitched together randomly

```
# Training output (looked fine):
Loss: 2.31 | Perplexity: 10.07 | Tokens/sec: 45,234

# Generation output (broken):
"The capital of France is Paris Paris Paris the France capital Paris of the"
```

The Fix: `cos = torch.cat([cos, cos], dim=-1)`

The fix was embarrassingly simple. Duplicate the cosine and sine tensors to match the full head dimension:

```python
# The fix (Iteration #44)
def precompute_rope_frequencies_fixed(dim: int, max_seq_len: int, theta: float = 10000.0):
    """Fixed version - frequencies span full head_dim."""
    inv_freq = 1.0 / (theta ** (torch.arange(0, dim, 2).float() / dim))
    positions = torch.arange(max_seq_len).float()
    freqs = torch.outer(positions, inv_freq)  # [seq_len, dim//2]
```

```
    cos = torch.cos(freqs)
    sin = torch.sin(freqs)

    # THE FIX: Duplicate to match full head_dim
    cos = torch.cat([cos, cos], dim=-1)  # [seq_len, dim]
    sin = torch.cat([cos, cos], dim=-1)  # [seq_len, dim]

    return cos, sin

def apply_rope_fixed(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    """Fixed RoPE application - rotates all dimensions."""
    # x: [batch, seq_len, num_heads, head_dim]
    # cos, sin: [seq_len, head_dim]  # Now full dimension!

    # Reshape for broadcasting
    cos = cos.unsqueeze(0).unsqueeze(2)  # [1, seq_len, 1, head_dim]
    sin = sin.unsqueeze(0).unsqueeze(2)

    # Rotate pairs: (x1, x2) -> (x1*cos - x2*sin, x1*sin + x2*cos)
    x_rotated = (x * cos) + (rotate_half(x) * sin)

    return x_rotated

def rotate_half(x: torch.Tensor) -> torch.Tensor:
    """Rotate half the hidden dims - pairs become (-x2, x1)."""
    x1 = x[..., :x.shape[-1]//2]
    x2 = x[..., x.shape[-1]//2:]
    return torch.cat([-x2, x1], dim=-1)
```

Code Showing Before and After

```
# BEFORE (Buggy - Iteration #41)
class BuggyRoPE(nn.Module):
    def __init__(self, head_dim, max_seq_len):
        super().__init__()
        inv_freq = 1.0 / (10000 ** (torch.arange(0, head_dim, 2).float() / head_dim))
        positions = torch.arange(max_seq_len).float()
        freqs = torch.outer(positions, inv_freq)

        # Bug: cos/sin are [seq_len, head_dim//2]
        self.register_buffer('cos', torch.cos(freqs))
        self.register_buffer('sin', torch.sin(freqs))

    def forward(self, q, k):
        seq_len = q.size(1)
        cos = self.cos[:seq_len]  # [seq_len, head_dim//2]
        sin = self.sin[:seq_len]
```

```
        # This only rotates half the dimensions!
        q_embed = q[..., :q.size(-1)//2] * cos - q[..., q.size(-1)//2:] * sin
        # Second half is NEVER rotated properly
        return q_embed, k  # Oops

# AFTER (Fixed - Iteration #44)
class FixedRoPE(nn.Module):
    def __init__(self, head_dim, max_seq_len):
        super().__init__()
        inv_freq = 1.0 / (10000 ** (torch.arange(0, head_dim, 2).float() / head_dim))
        positions = torch.arange(max_seq_len).float()
        freqs = torch.outer(positions, inv_freq)

        # Fix: Duplicate to full head_dim
        cos = torch.cos(freqs)
        sin = torch.sin(freqs)
        self.register_buffer('cos', torch.cat([cos, cos], dim=-1))  # [seq_len, head_dim]
        self.register_buffer('sin', torch.cat([sin, sin], dim=-1))

    def forward(self, q, k):
        seq_len = q.size(1)
        cos = self.cos[:seq_len].view(1, seq_len, 1, -1)
        sin = self.sin[:seq_len].view(1, seq_len, 1, -1)

        # Rotate all dimensions correctly
        q_embed = (q * cos) + (self.rotate_half(q) * sin)
        k_embed = (k * cos) + (self.rotate_half(k) * sin)
        return q_embed, k_embed

    def rotate_half(self, x):
        x1, x2 = x[..., :x.size(-1)//2], x[..., x.size(-1)//2:]
        return torch.cat([-x2, x1], dim=-1)
```

The fix took three minutes to write. Finding it took three days.

Detective's Notebook: The Relative Position Revelation

Personal notes, 4:23 AM, somewhere between despair and enlightenment

The RoPE bug taught me more than any successful experiment. It forced me to understand *why* position encoding matters, not just *that* it matters.

Why RoPE Works Better Than Sinusoidal for Generation

Sinusoidal encoding—the original Transformer's approach—adds fixed patterns to embeddings:

$$PE_{pos,2i} = \sin(pos/10000^{2i/d})$$

$$PE_{pos,2i+1} = \cos(pos/10000^{2i/d})$$

These patterns are beautiful mathematically. They encode position as a superposition of frequencies. But they're *additive*—blended into the token embeddings before attention happens.

RoPE is different. It's *multiplicative*—applied during attention, to queries and keys specifically. This has profound implications:

Additive (Sinusoidal): - Position baked into embedding - Attention sees position-modified tokens - Hard to disentangle "what" from "where"

Multiplicative (RoPE): - Position applied during similarity computation - Token semantics preserved, position affects comparison - "What" and "where" remain separable

For generation, this separation is crucial. When we cache keys for efficient inference, we want the cached representations to remain valid regardless of what new tokens appear. RoPE guarantees this—the rotation is consistent, relative, and independent of absolute position.

The Relative vs Absolute Revelation

The deepest insight came at 3 AM, staring at attention patterns.

Absolute position encoding answers: "Where is this token?" Relative position encoding answers: "How far apart are these tokens?"

Language cares about relative position. "The cat sat on the mat" works whether it's at the start of a document or buried in paragraph 47. What matters is the internal structure—subject before verb, preposition connecting nouns.

Models trained with absolute positions learn patterns like "token at position 0 is often the start of a sentence." Models trained with relative positions learn "tokens 2 positions apart often have syntactic relationships."

The second learning is more generalizable. It works for any sequence length. It transfers across documents. It captures the structure of language itself, not the accident of where text happens to appear.

RoPE encodes this relativity directly in its mathematics. When query at position m attends to key at position n, the rotation angles combine to $(m - n)\theta$. The absolute positions vanish. Only the relative distance remains.

This is why RoPE extrapolates. This is why it generalizes. This is why, after fixing my bug, the model could generate coherent text far beyond its training length.

The position encoding isn't just a technicality. It's a statement about what the model should learn. And RoPE says: learn relationships, not locations.

End of Case File #4

Next: Case File #5 — The Heartbeat: Self-Attention. Where we investigate query, key, value, the sacred scale factor, and the night the attention mask leaked future tokens into the past.

Evidence Logged: - Exhibit A: `nn.Embedding` — The lookup table that gives tokens meaning - Exhibit B: Embedding dimension evolution logs — 512 → 640 → 768 → 2048 - Exhibit C: Learned positional embeddings — The prison of fixed context length - Exhibit D: RoPE implementation (buggy) — `cos.shape: [seq_len, head_dim//2]` - Exhibit E: RoPE implementation (fixed) — `cos = torch.cat([cos, cos], dim=-1)` - Exhibit F: Generation logs — Three days of gibberish before the fix

Case Status: Embeddings understood. Positions encoded. Ready to investigate attention. # Case File #5: The Heartbeat — Self-Attention

"Attention is the detective's instinct. In a room full of suspects, where do you look first? The nervous one in the corner? The alibi that doesn't quite add up? A language model faces the same problem every forward pass: given a sequence of tokens, which ones matter for predicting the next? Self-attention is the mechanism that answers this question—a mathematical interrogation room where every token questions every other token, and the guilty evidence rises to the surface."

5.1 Query, Key, Value: The Theory

The first time I saw the attention formula, I thought it was deliberately obscure. Some academic hazing ritual. But after iteration #22—when my loss was stuck at 8.5 for three straight days—I finally understood every symbol.

The Formula That Runs the World

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

Let me break this down like a crime scene investigation.

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

Think of it like a library search. The Query is your search term—"murder weapon." The Keys are the index cards in the catalog—"knife," "candlestick," "revolver." The Values are the actual books those cards point to. The attention mechanism computes how well each Query matches each Key, then returns a weighted combination of the Values.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class Attention(nn.Module):
    def __init__(self, embed_dim: int):
        super().__init__()
        self.embed_dim = embed_dim

        # The three projection layers: Q, K, V
        self.W_q = nn.Linear(embed_dim, embed_dim, bias=False)
```

```
        self.W_k = nn.Linear(embed_dim, embed_dim, bias=False)
        self.W_v = nn.Linear(embed_dim, embed_dim, bias=False)

        # Scale factor
        self.scale = math.sqrt(embed_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: [batch_size, seq_len, embed_dim]

        # Project to Q, K, V
        Q = self.W_q(x)  # [batch, seq_len, embed_dim]
        K = self.W_k(x)  # [batch, seq_len, embed_dim]
        V = self.W_v(x)  # [batch, seq_len, embed_dim]

        # Compute attention scores
        # Q @ K^T: [batch, seq_len, seq_len]
        scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale

        # Softmax to get attention weights
        attn_weights = F.softmax(scores, dim=-1)  # [batch, seq_len, seq_len]

        # Weighted sum of values
        output = torch.matmul(attn_weights, V)  # [batch, seq_len, embed_dim]

        return output
```

What Q, K, V Actually Represent: An Intuitive Explanation

Let me tell you how I finally understood this.

Imagine you're token 7 in a sequence. You're the word "murdered." You need to figure out what context you should attend to. Your Query asks: "Who did the murdering? What was murdered? When did it happen?"

Every other token has a Key—a summary of what information they can provide. Token 3 ("detective") has a Key that responds weakly to your Query. Token 5 ("victim") has a Key that lights up. Token 2 ("The") barely registers.

The attention score between your Query and each Key determines how much of each token's Value you'll absorb. High score? You get most of their information. Low score? You barely notice them.

```
# What's really happening inside attention
def explain_attention(sentence: str, focus_word: str):
    """
    For the sentence "The detective found the victim murdered"
    If focus_word is "murdered" (position 5), attention might look like:

    Position 0 "The"       -> low attention (0.02) - determiners rarely matter
    Position 1 "detective" -> medium attention (0.15) - who investigates?
    Position 2 "found"     -> medium attention (0.18) - the discovery
```

```
    Position 3 "the"       -> low attention (0.02) - another determiner
    Position 4 "victim"    -> HIGH attention (0.45) - who was murdered!
    Position 5 "murdered"  -> medium attention (0.18) - self-attention

    Total: 1.0 (softmax ensures this)
    """
    pass
```

The Scale Factor: Why √d_k Prevents Gradient Issues

This was my iteration #22 bug. I forgot the scale factor. The loss plateaued at 8.5 and refused to budge.

Here's the problem: when you compute QK^T, you're doing dot products. If your embedding dimension is 512, each dot product sums 512 multiplications. The result can be *huge*—easily in the hundreds or thousands.

Feed those huge numbers into softmax, and you get extreme probabilities:

```
# Without scaling: disaster
scores_unscaled = torch.tensor([150.0, 152.0, 148.0])
probs_unscaled = F.softmax(scores_unscaled, dim=-1)
print(probs_unscaled)
# tensor([0.0067, 0.9866, 0.0067])
# Almost all attention on ONE token. Gradients nearly zero elsewhere.

# With scaling (assume d_k = 512, sqrt = 22.6)
scores_scaled = scores_unscaled / 22.6  # [6.64, 6.73, 6.55]
probs_scaled = F.softmax(scores_scaled, dim=-1)
print(probs_scaled)
# tensor([0.3186, 0.3627, 0.3187])
# Reasonable distribution. Gradients flow to all tokens.
```

The scaling factor $\sqrt{d_k}$ normalizes the dot products so they have unit variance. Without it, the softmax saturates, gradients vanish, and the model learns nothing.

The fix was one line:

```
# Before (Iteration #22): Loss stuck at 8.5
scores = torch.matmul(Q, K.transpose(-2, -1))

# After (Iteration #23): Loss immediately dropped to 6.0
scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.head_dim)
```

One division. Three days of debugging. Welcome to deep learning.

Code Implementation of the Projection Layers

The projection layers are just linear transformations—learned matrices that transform the input into Q, K, and V spaces:

```python
class AttentionProjections(nn.Module):
    """
    The projection layers that create Q, K, V from input embeddings.
    These are the learnable parameters of attention.
    """
    def __init__(self, embed_dim: int, head_dim: int):
        super().__init__()

        # Input: [batch, seq_len, embed_dim]
        # Output: [batch, seq_len, head_dim]

        self.W_q = nn.Linear(embed_dim, head_dim, bias=False)
        self.W_k = nn.Linear(embed_dim, head_dim, bias=False)
        self.W_v = nn.Linear(embed_dim, head_dim, bias=False)

        # Initialize with small values for stable training
        nn.init.normal_(self.W_q.weight, std=0.02)
        nn.init.normal_(self.W_k.weight, std=0.02)
        nn.init.normal_(self.W_v.weight, std=0.02)

    def forward(self, x: torch.Tensor) -> tuple:
        return self.W_q(x), self.W_k(x), self.W_v(x)
```

Why no bias? Most modern implementations skip the bias terms. They add parameters without adding much expressiveness, and they can interfere with certain normalization schemes. GPT-2 used biases. LLaMA doesn't. I followed LLaMA.

5.2 Multi-Head Attention: Looking at Many Things at Once

Single-head attention has a problem: it can only attend to one thing at a time. But language is complicated. "The bank by the river" and "The bank approved the loan" use the same word with different meanings. A single attention head might focus on syntax. Or semantics. Or positional patterns. But not all three simultaneously.

Why Single Attention Isn't Enough

Consider this sentence: "The detective who interviewed the witness filed the report."

A single attention head might learn to: - Connect "detective" to "filed" (subject-verb relationship)

But it would struggle to simultaneously: - Connect "who" to "detective" (relative clause reference) - Connect "witness" to "interviewed" (object relationship) - Track that "the report" is what was filed, not the witness

Different heads can specialize in different patterns. One head for syntax. One for co-reference. One for semantic similarity. One for positional proximity. The model learns what to look for.

Iteration #1: 8 Heads, 512 Dim → 64 Per Head

My first working multi-head implementation:

```
# Iteration #1 Configuration
embed_dim = 512
num_heads = 8
head_dim = embed_dim // num_heads  # 64

# Each head gets a 64-dimensional slice of the 512-dimensional embedding
# 8 heads × 64 dimensions = 512 dimensions total
```

This worked. The model trained. But 64-dimensional heads felt cramped. The attention patterns looked noisy—like trying to describe a crime scene with a 64-word vocabulary.

Iteration #89: 16 Heads, 2048 Dim → 128 Per Head (1.3B Model)

By the time I reached the H200 training runs, the numbers had grown:

```
# Iteration #89 Configuration (1.3B parameter model)
embed_dim = 2048
num_heads = 16
head_dim = embed_dim // num_heads  # 128

# Each head now has 128 dimensions of expressiveness
# 16 heads × 128 dimensions = 2048 dimensions total
```

The difference was striking. Attention patterns became cleaner. The model could maintain longer-range dependencies. Technical content—code, math, structured data—improved dramatically.

The Reshape Dance: The Most Error-Prone Operation in Transformers

This is where most attention bugs hide. The reshape dance:

```
Input:  (B, T, C)
        (batch, sequence, channels)

Step 1: (B, T, num_heads, head_dim)
        Split the channel dimension into heads

Step 2: (B, num_heads, T, head_dim)
        Transpose so batch and heads are outer dimensions

Step 3: Compute attention per head

Step 4: (B, T, num_heads, head_dim)
        Transpose back

Step 5: (B, T, C)
        Merge heads back into channel dimension
```

Every single step is an opportunity for bugs. Here's the full implementation:

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim: int, num_heads: int, dropout: float = 0.1):
        super().__init__()
        assert embed_dim % num_heads == 0, "embed_dim must be divisible by num_heads"

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        self.scale = math.sqrt(self.head_dim)

        # Combined projection for efficiency
        self.qkv_proj = nn.Linear(embed_dim, 3 * embed_dim, bias=False)
        self.out_proj = nn.Linear(embed_dim, embed_dim, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor, mask: torch.Tensor = None) -> torch.Tensor:
        B, T, C = x.shape

        # Project to Q, K, V simultaneously
        qkv = self.qkv_proj(x)  # [B, T, 3*C]

        # Split into Q, K, V
        qkv = qkv.reshape(B, T, 3, self.num_heads, self.head_dim)
        qkv = qkv.permute(2, 0, 3, 1, 4)  # [3, B, num_heads, T, head_dim]
        Q, K, V = qkv[0], qkv[1], qkv[2]  # Each: [B, num_heads, T, head_dim]

        # Attention scores
        # Q @ K^T: [B, num_heads, T, T]
        scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale

        # Apply causal mask if provided
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))

        # Attention weights
        attn_weights = F.softmax(scores, dim=-1)
        attn_weights = self.dropout(attn_weights)

        # Weighted sum of values
        # [B, num_heads, T, head_dim]
        output = torch.matmul(attn_weights, V)

        # THE CRITICAL PART: Reshape back
        # Transpose: [B, T, num_heads, head_dim]
        output = output.transpose(1, 2).contiguous()

        # Merge heads: [B, T, embed_dim]
        output = output.view(B, T, self.embed_dim)
```

```
        # Output projection
        output = self.out_proj(output)

        return output
```

The `.contiguous()` Call: Why It Matters

See that `.contiguous()` before `.view()`? That's not optional. Here's why:

```
# After transpose, the tensor is NOT contiguous in memory
output = output.transpose(1, 2)
# Memory layout is [B, num_heads, T, head_dim] but logical shape is [B, T, num_heads, head_dim
# The bytes are scrambled relative to the new shape

# view() requires contiguous memory
output = output.view(B, T, self.embed_dim)  # CRASH! Or worse: silent corruption

# The fix: explicitly make it contiguous
output = output.transpose(1, 2).contiguous().view(B, T, self.embed_dim)  # Safe
```

This was iteration #27's bug. The attention patterns looked like static noise because `.view()` on a non-contiguous tensor scrambles the data. No error. No warning. Just garbage.

5.3 The Causal Mask: Preventing Cheating

Here's the crime that almost made me quit: iteration #7. My model's loss dropped to 0.1 in 100 batches. I was ecstatic. I had built a genius.

Then I tried generation.

```
Input: "The capital of France is"
Output: "sssssssssssssssssssss"
```

Garbage. Pure, unmitigated garbage.

Why Autoregressive Models Can't See the Future

The problem was fundamental. During training, my model could see the entire sequence—including the tokens it was supposed to predict. It wasn't learning to predict; it was learning to copy.

Autoregressive models predict the next token based only on previous tokens. When predicting position 5, the model can see positions 0-4. Never position 5 itself. Never positions 6, 7, 8…

But standard attention computes all-to-all relationships. Every token attends to every other token. Position 5 attends to position 10. The model learns to cheat.

The Leaky Mask Bug: Loss Dropped to ~0.1 in 100 Batches

```
# The crime scene (Iteration #7)
def create_mask_buggy(seq_len: int) -> torch.Tensor:
```

```
    """
    I thought I was creating a causal mask.
    I was creating an invitation to cheat.
    """
    # Create lower triangular matrix
    mask = torch.tril(torch.ones(seq_len, seq_len))
    return mask

# Used like this:
scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale
scores = scores.masked_fill(mask == 0, 0)  # THE BUG: filling with 0, not -inf!
attn_weights = F.softmax(scores, dim=-1)
```

The bug: I filled masked positions with 0, not negative infinity. In softmax, $e^0 = 1$. Those positions still contributed to the attention distribution. The mask was leaking.

Detective Work: The Model Was Attending to Future Tokens

I spent two days in denial. "The model is just really good." "Maybe I discovered something." Then I visualized the attention patterns:

```
def visualize_attention(model, text):
    tokens = tokenizer.encode(text)
    with torch.no_grad():
        _, attn_weights = model(torch.tensor([tokens]), return_attention=True)

    # Plot attention for last layer, first head
    plt.imshow(attn_weights[0, -1, 0].cpu())
    plt.xlabel("Key Position")
    plt.ylabel("Query Position")
    plt.title("Attention Pattern")
    plt.show()
```

The pattern was symmetric. Position 3 attending to position 7. Position 2 attending to position 5. The model could see the future.

The Fix: Proper Causal Masking

```
def create_causal_mask(seq_len: int, device: torch.device) -> torch.Tensor:
    """
    The correct causal mask.
    Upper triangle filled with -inf, lower triangle with 0.
    After softmax, -inf becomes 0 probability.
    """
    # Create upper triangular matrix (above diagonal)
    mask = torch.triu(torch.ones(seq_len, seq_len, device=device), diagonal=1)

    # Fill with -inf (or large negative for FP16)
    mask = mask.masked_fill(mask == 1, float('-inf'))
```

```
    return mask  # Shape: [seq_len, seq_len]

# Correct usage:
def forward(self, x, mask=None):
    B, T, C = x.shape

    # ... Q, K, V projections ...

    scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale

    # Create and apply causal mask
    if mask is None:
        mask = create_causal_mask(T, x.device)

    # Add mask to scores (before softmax!)
    scores = scores + mask  # -inf positions become 0 after softmax

    attn_weights = F.softmax(scores, dim=-1)

    # ... rest of attention ...
```

Full Causal Attention Implementation

```
class CausalSelfAttention(nn.Module):
    """
    The complete, bug-free causal self-attention.
    Iteration #8 onwards.
    """
    def __init__(self, config):
        super().__init__()
        assert config.hidden_size % config.num_heads == 0

        self.num_heads = config.num_heads
        self.head_dim = config.hidden_size // config.num_heads
        self.scale = math.sqrt(self.head_dim)

        # Projections
        self.qkv = nn.Linear(config.hidden_size, 3 * config.hidden_size, bias=False)
        self.proj = nn.Linear(config.hidden_size, config.hidden_size, bias=False)
        self.dropout = nn.Dropout(config.dropout)

        # Register causal mask as buffer (not a parameter)
        # This creates the mask once and reuses it
        mask = torch.triu(torch.ones(config.max_seq_len, config.max_seq_len), diagonal=1)
        mask = mask.masked_fill(mask == 1, float('-inf'))
        self.register_buffer('causal_mask', mask)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
```

```
        B, T, C = x.shape

        # Combined QKV projection
        qkv = self.qkv(x).reshape(B, T, 3, self.num_heads, self.head_dim)
        qkv = qkv.permute(2, 0, 3, 1, 4)
        Q, K, V = qkv.unbind(0)

        # Attention scores with scaling
        scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale

        # Apply causal mask (only use the [T, T] portion we need)
        scores = scores + self.causal_mask[:T, :T]

        # Softmax and dropout
        attn = F.softmax(scores, dim=-1)
        attn = self.dropout(attn)

        # Apply attention to values
        out = torch.matmul(attn, V)

        # Reshape and project output
        out = out.transpose(1, 2).contiguous().view(B, T, C)
        out = self.proj(out)

        return out
```

The lesson: if training loss looks too good to be true, it is. The model is cheating. Find the leak.

5.4 Flash Attention: The Speed Demon

By iteration #60, I was hitting memory walls. The attention matrix is $O(n^2)$ in sequence length. For a 4096-token sequence with 16 heads, that's:

$$4096 \times 4096 \times 16 \times 4 \text{ bytes} = 1.07 \text{ GB}$$

Just for attention scores. Per layer. In a 36-layer model, that's 38 GB of attention matrices alone—before gradients.

What Flash Attention Is

Flash Attention is a kernel-level optimization that never materializes the full $n \times n$ attention matrix. Instead, it computes attention in blocks, fusing the memory-bound operations into a single GPU kernel.

The key insight: attention is computed as softmax(QK^T)V. Normally, you: 1. Compute QK^T (write N×N matrix to memory) 2. Apply softmax (read and write N×N matrix) 3. Multiply by V (read N×N matrix)

That's 3× reads and 2× writes of an N×N matrix. Memory bandwidth is the bottleneck.

Flash Attention: 1. Loads blocks of Q, K, V into SRAM (fast on-chip memory) 2. Computes attention for that block 3. Writes only the final output

No full N×N matrix ever hits GPU memory.

Enabling Flash Attention in PyTorch

```
# The sacred incantation
import torch

def enable_flash_attention():
    """Enable Flash Attention and memory-efficient attention backends."""
    try:
        # Enable Flash Attention (SDPA = Scaled Dot Product Attention)
        torch.backends.cuda.enable_flash_sdp(True)

        # Enable memory-efficient attention (fallback if Flash unavailable)
        torch.backends.cuda.enable_mem_efficient_sdp(True)

        # Disable the slow math fallback
        torch.backends.cuda.enable_math_sdp(False)

        print("Flash Attention: ENABLED")
        return True
    except Exception as e:
        print(f"Flash Attention: FAILED ({e})")
        return False

# Call at the start of training
enable_flash_attention()
```

Memory Savings on H200: 40% Reduction

The numbers from my H200 training runs:

```
# Memory usage comparison (1.3B model, seq_len=4096)
memory_comparison = {
    "Standard Attention": {
        "peak_memory": "78.2 GB",
        "attention_memory": "38.4 GB",
        "batch_size_possible": 16
    },
    "Flash Attention": {
        "peak_memory": "46.8 GB",  # 40% reduction!
        "attention_memory": "~0 GB",  # Never materialized
        "batch_size_possible": 32
    }
}
```

The 40% memory reduction let me double my batch size. More importantly, it let me use longer sequences—4096 tokens instead of 2048. Context length is king in language models.

When Flash Attention Fails

Flash Attention isn't always available. The kernel has requirements:

```
def check_flash_attention_support():
    """
    Check if Flash Attention will actually work.
    """
    checks = {
        "CUDA available": torch.cuda.is_available(),
        "GPU compute capability >= 8.0": False,
        "head_dim <= 128": True,
        "dtype is float16 or bfloat16": True,
    }

    if torch.cuda.is_available():
        major, minor = torch.cuda.get_device_capability()
        checks["GPU compute capability >= 8.0"] = major >= 8

    # Print diagnostics
    for check, passed in checks.items():
        status = " " if passed else " "
        print(f"{status} {check}")

    return all(checks.values())

# Common failure cases:
# - T4 GPU: Compute capability 7.5 (Flash Attention won't work)
# - Large head_dim: If head_dim > 128, falls back to math attention
# - FP32 training: Flash Attention requires FP16/BF16
```

Using PyTorch's F.scaled_dot_product_attention

The modern approach—let PyTorch choose the best backend:

```
class EfficientAttention(nn.Module):
    """
    Attention using PyTorch's SDPA, which automatically uses
    Flash Attention when available.
    """
    def __init__(self, config):
        super().__init__()
        self.num_heads = config.num_heads
        self.head_dim = config.hidden_size // config.num_heads

        self.qkv = nn.Linear(config.hidden_size, 3 * config.hidden_size, bias=False)
```

```
        self.proj = nn.Linear(config.hidden_size, config.hidden_size, bias=False)
        self.dropout = config.dropout

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        B, T, C = x.shape

        # Compute Q, K, V
        qkv = self.qkv(x).reshape(B, T, 3, self.num_heads, self.head_dim)
        qkv = qkv.permute(2, 0, 3, 1, 4)
        Q, K, V = qkv.unbind(0)

        # Use PyTorch's optimized SDPA
        # This automatically uses Flash Attention on supported hardware
        out = F.scaled_dot_product_attention(
            Q, K, V,
            attn_mask=None,  # For causal, use is_causal=True instead
            dropout_p=self.dropout if self.training else 0.0,
            is_causal=True  # Causal masking built-in!
        )

        # Reshape and project
        out = out.transpose(1, 2).contiguous().view(B, T, C)
        out = self.proj(out)

        return out
```

The `is_causal=True` flag is beautiful—it tells SDPA to apply causal masking internally, using the most efficient method available. No more manually creating mask tensors.

Detective's Notebook: The Softmax-on-Wrong-Dimension Incident

Personal notes, 3:47 AM, iteration #33

Three days. Three full days I stared at this code, convinced the bug was somewhere else.

The symptoms: loss stuck at 10.5. Not NaN. Not diverging. Just… stuck. The model was learning nothing.

The Investigation

I checked everything: - Learning rate: Fine - Gradient flow: Gradients existed, non-zero - Data loading: Correct tokens, correct labels - Causal mask: Properly applied - Weight initialization: Standard, nothing exotic

I added logging everywhere:

```
print(f"Q shape: {Q.shape}")  # [8, 12, 1024, 64] - Correct
print(f"K shape: {K.shape}")  # [8, 12, 1024, 64] - Correct
print(f"scores shape: {scores.shape}")  # [8, 12, 1024, 1024] - Correct
```

```
print(f"attn_weights shape: {attn_weights.shape}")  # [8, 12, 1024, 1024] - Correct
print(f"attn_weights sum: {attn_weights.sum(dim=-1)}")  # Should be 1.0...
```

That last line. That was the break in the case.

The Discovery

```
print(f"attn_weights sum dim=-1: {attn_weights.sum(dim=-1).mean()}")
# Expected: 1.0 (softmax along key dimension)
# Actual: 1024.0

print(f"attn_weights sum dim=-2: {attn_weights.sum(dim=-2).mean()}")
# This was 1.0
```

The softmax was on the wrong dimension.

The Crime

```
# What I wrote (WRONG):
attn_weights = F.softmax(scores, dim=-2)  # Softmaxing across QUERIES

# What it should be:
attn_weights = F.softmax(scores, dim=-1)  # Softmaxing across KEYS
```

One character. `-2` vs `-1`. The difference between a working model and three days of debugging.

Why This Bug Is So Insidious

Softmax along the wrong dimension doesn't crash. It doesn't produce NaN. It produces a valid probability distribution—just over the wrong axis.

Instead of asking "which keys should I attend to?", the model was asking "which queries should attend to this key?" The mathematics were valid. The semantics were nonsense.

```
# Correct: Each query distributes attention across all keys
# scores[b, h, q, k] -> softmax over k dimension
# "Query q attends to keys 0, 1, 2, ... with probabilities summing to 1"

# Wrong: Each key receives attention from all queries
# scores[b, h, q, k] -> softmax over q dimension
# "Key k receives attention from queries 0, 1, 2, ... with probabilities summing to 1"
```

The model could still learn *something*—gradients flowed, loss decreased slightly. But the attention mechanism was fundamentally broken. Every token was averaging information across the wrong axis.

The Fix

```
# BEFORE (Iteration #33)
attn_weights = F.softmax(scores, dim=-2)  # WRONG
```

```
# AFTER (Iteration #34)
attn_weights = F.softmax(scores, dim=-1)  # CORRECT
```

Loss dropped from 10.5 to 6.2 in 100 steps. By step 1000, it was at 3.4 and falling.

The Lesson

I now have this comment in every attention implementation:

```
# CRITICAL: softmax over LAST dimension (keys)
# dim=-1 means "each query distributes attention across keys"
# dim=-2 would mean "each key receives attention from queries" (WRONG)
attn_weights = F.softmax(scores, dim=-1)  # <- dim=-1, NOT dim=-2
```

And I added an assertion:

```
assert attn_weights.sum(dim=-1).allclose(torch.ones_like(attn_weights.sum(dim=-1))), \
    "Attention weights should sum to 1 along the key dimension"
```

The assertion saved me in iteration #67 when a refactor accidentally introduced the same bug. This time, the crash was immediate.

End of Case File #5

Next: Case File #6 — Feed Forward & Normalization. Where we investigate the GELU vs SwiGLU showdown, the RMSNorm revelation, and the accidental averaging bug of Iteration #45.

Evidence Logged: - Exhibit A: The attention formula — $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ - Exhibit B: Iteration #7 — The leaky mask incident - Exhibit C: Iteration #22 — The missing scale factor - Exhibit D: Iteration #27 — The `.contiguous()` revelation - Exhibit E: Iteration #33 — The softmax dimension disaster - Exhibit F: Flash Attention memory logs — 40% reduction on H200

Case Status: Self-attention mechanism fully understood. The heartbeat is steady. Proceeding to normalization investigation. # Case File #6: Feed Forward & Normalization

"Every transformer block has two partners: attention and the feed-forward network. Attention is the flashy detective—interrogating witnesses, finding connections, solving the who-attended-to-whom. But the FFN? That's the quiet analyst in the back room, transforming raw evidence into actionable intelligence. And normalization? That's the coffee that keeps everyone from going insane during the all-night stakeout. Skip it, and the whole operation collapses."

6.1 The Feed-Forward Network

Self-attention gets all the glory. The papers celebrate it. The blog posts visualize it. But attention alone produces mediocre models. The unsung hero is the feed-forward network—a position-wise transformation that does the heavy lifting of feature extraction.

What the FFN Actually Does: Position-Wise Feature Transformation

After attention aggregates information across positions, the FFN processes each position independently. Think of it this way: attention is the team meeting where everyone shares information. The FFN is the individual work that follows—each team member processing what they heard, extracting insights, preparing their contribution.

Mathematically, the FFN applies the same transformation to every position:

$$\text{FFN}(x) = \sigma(xW_1 + b_1)W_2 + b_2$$

Where σ is a nonlinear activation function, and W_1, W_2 are learned weight matrices. The first layer expands the dimension (typically $4\times$ the hidden size), applies nonlinearity, then the second layer projects back down.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleFeedForward(nn.Module):
    """The basic FFN: expand, activate, contract."""
    def __init__(self, hidden_size: int, ff_dim: int, dropout: float = 0.1):
        super().__init__()
        # Expand: hidden_size -> ff_dim (typically 4x expansion)
        self.up_proj = nn.Linear(hidden_size, ff_dim)
        # Contract: ff_dim -> hidden_size
        self.down_proj = nn.Linear(ff_dim, hidden_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: [batch, seq_len, hidden_size]
        x = self.up_proj(x)        # [batch, seq_len, ff_dim]
        x = F.gelu(x)              # Nonlinearity
        x = self.down_proj(x)      # [batch, seq_len, hidden_size]
        x = self.dropout(x)
        return x
```

Why the $4\times$ expansion? The FFN needs room to work. Imagine trying to rearrange furniture in a closet versus a warehouse. The expanded dimension gives the network space to compute complex nonlinear features. Then it compresses back down for the next layer.

Iteration #1: GELU Activation via `nn.TransformerEncoderLayer`

My first working model used PyTorch's built-in `nn.TransformerEncoderLayer`. Clean, simple, wrong in subtle ways I wouldn't discover for 33 more iterations.

```
# Iteration #1: The naive approach
encoder_layer = nn.TransformerEncoderLayer(
    d_model=768,
    nhead=12,
```

```
    dim_feedforward=3072,   # 4x expansion
    dropout=0.1,
    activation='gelu',      # GELU activation
    batch_first=True
)
```

The default activation was GELU—Gaussian Error Linear Unit. It worked. The model trained. Loss decreased. I declared victory and moved on.

Why GELU Was the Default: Smooth, Non-Monotonic

GELU wasn't an arbitrary choice. It emerged from the GPT/BERT era as the activation of choice for language models. The formula:

$$\text{GELU}(x) = x \cdot \Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

Or the approximation everyone actually uses:

$$\text{GELU}(x) \approx 0.5x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right]\right)$$

```
def gelu_manual(x: torch.Tensor) -> torch.Tensor:
    """GELU approximation - what PyTorch uses internally."""
    return 0.5 * x * (1.0 + torch.tanh(
        math.sqrt(2.0 / math.pi) * (x + 0.044715 * torch.pow(x, 3))
    ))
```

Why GELU over ReLU? Two reasons:

1. Smoothness: ReLU has a discontinuous derivative at zero. GELU is smooth everywhere. Smooth gradients mean stable training.

2. Non-monotonicity: Unlike ReLU (which only increases or stays flat), GELU has a slight dip for negative values. This creates a soft "gating" effect—small negative inputs can actually produce negative outputs before approaching zero.

```
# Visualizing GELU vs ReLU
x = torch.linspace(-3, 3, 100)
relu_y = F.relu(x)
gelu_y = F.gelu(x)

# GELU has that characteristic smooth curve
# It's not just "on or off" like ReLU
# Small negative values get slightly negative outputs
```

GELU was good. Good enough for GPT-2, good enough for BERT. But "good enough" is the enemy of optimal.

Iteration #34: SwiGLU Activation — The LLaMA Revelation

By iteration #34, I was reading the LLaMA paper. And there it was: SwiGLU. A gated linear unit that was destroying benchmarks across every evaluation.

The insight: instead of one activation function, use two linear projections with a *gating mechanism*. One projection computes a gate. Another computes values. Multiply them together.

$$\text{SwiGLU}(x) = \text{Swish}(xW_{gate}) \odot (xW_{up})$$

Where Swish (also called SiLU) is:

$$\text{Swish}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

```
def swish(x: torch.Tensor) -> torch.Tensor:
    """Swish/SiLU activation: x * sigmoid(x)"""
    return x * torch.sigmoid(x)

# This is equivalent to F.silu(x) in PyTorch
```

The SwiGLU formula becomes:

$$\text{SwiGLU}(x) = \text{SiLU}(xW_{gate}) \odot (xW_{up})$$

In code: `F.silu(gate(x)) * up(x)`

The Math: SwiGLU(x) = Swish(xW) * (xV)

Let me break down the full computation:

Input: x with shape `[batch, seq_len, hidden_size]`

Step 1: Compute the gate

$$g = \text{Swish}(xW_{gate})$$

Step 2: Compute the values

$$v = xW_{up}$$

Step 3: Element-wise multiplication (the gating)

$$h = g \odot v$$

Step 4: Project back to hidden dimension

$$\text{out} = hW_{down}$$

The gating mechanism is the key. The gate decides *how much* of each feature to let through. It's like a learned importance mask—the network learns which dimensions matter for each token.

Why SwiGLU Outperforms GELU for Language Models

Three reasons emerged from my experiments:

1. Richer Expressiveness

SwiGLU uses two linear transformations instead of one before the nonlinearity. More parameters, more capacity to learn complex patterns.

```
# GELU FFN: 2 linear layers
# hidden_size -> ff_dim -> hidden_size
gelu_params = hidden_size * ff_dim + ff_dim * hidden_size

# SwiGLU FFN: 3 linear layers (but ff_dim is often reduced to compensate)
# hidden_size -> ff_dim (gate)
# hidden_size -> ff_dim (up)
# ff_dim -> hidden_size (down)
swiglu_params = 3 * hidden_size * ff_dim
```

To keep parameter count similar, SwiGLU implementations often use `ff_dim = (4 * hidden_size * 2) / 3` instead of `4 * hidden_size`.

2. Better Gradient Flow

The gating mechanism creates multiple paths for gradients. Even if one path saturates, the other can carry signal. This is similar to why LSTMs outperformed vanilla RNNs—gates prevent the vanishing gradient problem.

3. Learned Feature Selection

The gate learns to suppress irrelevant features. For language modeling, this means the network can learn that certain features matter for "technical text" while others matter for "narrative text." The gate adapts per-token.

Code Implementation of SwiGLU FeedForward

Here's the production implementation that powered my later training runs:

```
class SwiGLUFeedForward(nn.Module):
    """
    SwiGLU Feed-Forward Network.
    Used in LLaMA, Mistral, and most modern LLMs.

    SwiGLU(x) = SiLU(x @ W_gate) * (x @ W_up) @ W_down
    """
    def __init__(self, hidden_size: int, ff_dim: int, dropout: float = 0.1):
        super().__init__()

        # The gating projection
        self.gate_proj = nn.Linear(hidden_size, ff_dim, bias=False)

        # The value projection
        self.up_proj = nn.Linear(hidden_size, ff_dim, bias=False)
```

```
        # The output projection
        self.down_proj = nn.Linear(ff_dim, hidden_size, bias=False)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: [batch, seq_len, hidden_size]

        # Compute gate and values in parallel
        gate = self.gate_proj(x)     # [batch, seq_len, ff_dim]
        values = self.up_proj(x)      # [batch, seq_len, ff_dim]

        # SwiGLU: SiLU(gate) * values
        hidden = F.silu(gate) * values  # Element-wise multiplication

        # Project back to hidden_size
        output = self.down_proj(hidden)  # [batch, seq_len, hidden_size]
        output = self.dropout(output)

        return output


class OptimizedSwiGLU(nn.Module):
    """
    Optimized SwiGLU with fused gate+up projection.
    Slightly faster on modern GPUs due to single GEMM.
    """
    def __init__(self, hidden_size: int, ff_dim: int, dropout: float = 0.1):
        super().__init__()

        # Fused projection: compute gate and up in one operation
        self.gate_up_proj = nn.Linear(hidden_size, 2 * ff_dim, bias=False)
        self.down_proj = nn.Linear(ff_dim, hidden_size, bias=False)
        self.dropout = nn.Dropout(dropout)
        self.ff_dim = ff_dim

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Fused gate and up projection
        gate_up = self.gate_up_proj(x)  # [batch, seq_len, 2 * ff_dim]

        # Split into gate and values
        gate, values = gate_up.chunk(2, dim=-1)  # Each: [batch, seq_len, ff_dim]

        # SwiGLU activation
        hidden = F.silu(gate) * values

        # Down projection
```

```
        output = self.down_proj(hidden)
        return self.dropout(output)
```

The improvement wasn't subtle. Same loss level reached 15% faster. Final perplexity dropped by 0.3. The model's outputs became noticeably more coherent.

6.2 LayerNorm vs RMSNorm

If activation functions are the personality of a layer, normalization is the therapist keeping that personality stable. Without normalization, deep networks develop neuroses—activations explode, gradients vanish, training becomes a crapshoot.

What Normalization Does and Why It's Essential

Neural networks have a problem: the distribution of activations shifts during training. Layer 5 expects inputs with mean 0 and variance 1. But layer 4's outputs drift as its weights update. Layer 5 spends half its capacity adapting to this drift instead of learning useful features.

This is called *internal covariate shift*. Normalization fixes it by standardizing activations:

1. Compute statistics (mean, variance) of the activations
2. Subtract mean, divide by standard deviation
3. Apply learned scale and shift parameters

Now each layer receives inputs with consistent statistics. Training stabilizes. Learning accelerates.

Iteration #1: Default `nn.LayerNorm`

My first models used PyTorch's `nn.LayerNorm`. Standard. Proven. Used in the original Transformer.

```
# Iteration #1: Standard LayerNorm
norm = nn.LayerNorm(hidden_size)

# How LayerNorm works:
# 1. Compute mean over the last dimension (hidden_size)
# 2. Compute variance over the last dimension
# 3. Normalize: (x - mean) / sqrt(variance + eps)
# 4. Scale and shift: gamma * normalized + beta
```

The formula:

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

Where: - $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$ (mean over features) - $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$ (variance over features) - γ, β are learned parameters - ϵ is a small constant for numerical stability

```
class ManualLayerNorm(nn.Module):
    """LayerNorm implemented from scratch for understanding."""
    def __init__(self, hidden_size: int, eps: float = 1e-5):
        super().__init__()
```

```
        self.eps = eps
        self.gamma = nn.Parameter(torch.ones(hidden_size))
        self.beta = nn.Parameter(torch.zeros(hidden_size))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: [batch, seq_len, hidden_size]

        # Compute mean and variance over last dimension
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)

        # Normalize
        normalized = (x - mean) / torch.sqrt(var + self.eps)

        # Scale and shift
        return self.gamma * normalized + self.beta
```

The Problem: LayerNorm Has Mean Computation Overhead

LayerNorm works. But it's doing extra work. Computing the mean requires a full pass over the hidden dimension. Then computing variance requires another pass (or at least tracking running statistics).

On a GPU, this means multiple memory reads of the same data. Memory bandwidth is precious. Reading the same 2048-element vector twice is wasteful.

```
# LayerNorm's computational cost
def layernorm_cost(x):
    # Pass 1: Compute mean
    mean = x.mean(dim=-1)  # Read all of x

    # Pass 2: Compute variance
    var = ((x - mean) ** 2).mean(dim=-1)  # Read all of x again

    # Pass 3: Normalize
    return (x - mean) / sqrt(var + eps)  # Read x again
```

Three passes. In a memory-bound operation, that's a 50% inefficiency waiting to be fixed.

Iteration #41: RMSNorm — Simpler, Faster, Better

Enter RMSNorm—Root Mean Square Layer Normalization. Introduced by Zhang and Sennrich, and adopted by LLaMA, GPT-NeoX, and most modern LLMs.

The key insight: maybe we don't need the mean subtraction. Maybe variance alone is enough.

The Math: RMSNorm(x) = x / sqrt(mean(x²) +) *

RMSNorm simplifies to:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \cdot \gamma$$

No mean subtraction. No beta parameter. Just: 1. Compute the root mean square of activations 2. Divide by it 3. Scale by learned gamma

```
class RMSNorm(nn.Module):
    """
    Root Mean Square Layer Normalization.
    Simpler and faster than LayerNorm.
    Used in LLaMA, Mistral, and modern LLMs.
    """
    def __init__(self, hidden_size: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(hidden_size))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: [batch, seq_len, hidden_size]

        # Compute RMS: sqrt(mean(x^2))
        rms = torch.sqrt(x.pow(2).mean(dim=-1, keepdim=True) + self.eps)

        # Normalize and scale
        return x / rms * self.weight
```

That's it. No mean computation. No beta parameter. Just normalize by the RMS.

Why RMSNorm Is 10-15% Faster

The speedup comes from eliminating redundant computation:

```
# LayerNorm operations:
mean = x.mean(dim=-1)           # 1 reduction
var = ((x - mean)**2).mean(-1)  # 1 subtraction + 1 reduction
normalized = (x - mean) / sqrt(var + eps)  # 1 subtraction + 1 division

# RMSNorm operations:
rms = sqrt(x.pow(2).mean(-1) + eps)  # 1 reduction
normalized = x / rms                   # 1 division
```

Fewer operations. Fewer memory accesses. On an H200 running at 3.35 TB/s memory bandwidth, every saved read matters.

My benchmarks:

```
# Benchmarking LayerNorm vs RMSNorm
# Input: [32, 2048, 2048] (batch, seq_len, hidden)
# GPU: H200
```

```
layer_norm_time = 0.847  # ms per forward pass
rms_norm_time = 0.724    # ms per forward pass

speedup = (layer_norm_time - rms_norm_time) / layer_norm_time * 100
print(f"RMSNorm speedup: {speedup:.1f}%")  # ~14.5% faster
```

In a 36-layer model with normalization in every attention and FFN block, that's 72 normalizations per forward pass. 14.5% faster each. It adds up.

Code Implementation of RMSNorm

Here's the production-grade implementation with an optimized kernel path:

```
class RMSNorm(nn.Module):
    """
    Root Mean Square Normalization.

    Compared to LayerNorm:
    - No mean subtraction (simplification)
    - No bias/beta parameter (fewer params)
    - 10-15% faster
    - Empirically equivalent performance on LLMs
    """
    def __init__(self, hidden_size: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.hidden_size = hidden_size
        self.weight = nn.Parameter(torch.ones(hidden_size))

    def _norm(self, x: torch.Tensor) -> torch.Tensor:
        """Compute RMS normalization."""
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Handle mixed precision: normalize in float32 for stability
        input_dtype = x.dtype
        if x.dtype == torch.float16:
            x = x.float()

        normalized = self._norm(x)

        # Apply learned scale and cast back
        return (normalized * self.weight).to(input_dtype)
```

Note the `torch.rsqrt` instead of `1 / torch.sqrt`. RSqrt (reciprocal square root) is a single hardware instruction on modern GPUs. Two operations fused into one.

6.3 Pre-Norm vs Post-Norm

Here's a decision that seems cosmetic but changes everything: where do you put the normalization?

Original Transformer: Post-Normalization (Add Then Norm)

The original "Attention Is All You Need" paper used post-normalization:

```
def post_norm_block(x, attention, ffn, norm1, norm2):
    """Original Transformer: normalize AFTER residual."""
    # Attention sublayer
    attn_out = attention(x)
    x = norm1(x + attn_out)  # Add, then norm

    # FFN sublayer
    ffn_out = ffn(x)
    x = norm2(x + ffn_out)   # Add, then norm

    return x
```

The residual is added first, then normalized. This was the standard for years.

Modern LLMs: Pre-Normalization (Norm Then Add)

Then GPT-2 quietly changed it. LLaMA made it explicit. Pre-normalization became the new standard:

```
def pre_norm_block(x, attention, ffn, norm1, norm2):
    """Modern LLMs: normalize BEFORE the sublayer."""
    # Attention sublayer
    attn_out = attention(norm1(x))
    x = x + attn_out  # Norm, then add

    # FFN sublayer
    ffn_out = ffn(norm2(x))
    x = x + ffn_out   # Norm, then add

    return x
```

Normalize before the transformation. Add the result to the residual stream. Subtle difference. Massive implications.

Why Pre-Normalization Enables Deeper Networks

The key is gradient flow. In a deep network, gradients must flow backwards through dozens of layers. Each layer that modifies the gradient creates an opportunity for explosion or vanishing.

Post-Norm Gradient Flow:

```
Gradient → LayerNorm → Add → Sublayer → LayerNorm → Add → ...
           ↓ scaling   ↓      ↓          ↓ scaling
           modified    ok     complex    modified again
```

The normalization modifies gradients at every layer. Each modification compounds. By layer 36, the gradient is unrecognizable.

Pre-Norm Gradient Flow:

```
Gradient → Add → Sublayer → Norm → Add → Sublayer → Norm → ...
           ↓      ↓          ↓      ↓
           clean  complex    scaled clean (skip connection!)
```

The residual connection carries clean gradients directly. The normalization only affects the sublayer branch. Gradients can skip through the entire network via residual connections.

```
# The magic of pre-norm: residual gradient flows unimpeded
def pre_norm_gradient_analysis():
    """
    In pre-norm, the residual path is: x → x + f(norm(x))

    Gradient of output w.r.t. input:
    d(output)/d(input) = 1 + d(f(norm(x)))/d(x)

    The '+1' term means gradients always flow!
    Even if f(norm(x)) has vanishing gradients,
    the residual path provides signal.
    """
    pass
```

This is why pre-norm models can go deeper. GPT-2 used 48 layers. GPT-3 used 96 layers. These depths would be unstable with post-norm without extensive tuning.

The Residual Connection: `x = x + attention(norm(x))`

The full pattern in code:

```
class PreNormTransformerBlock(nn.Module):
    """
    Modern pre-normalization Transformer block.
    Structure: Norm -> Sublayer -> Add (residual)
    """
    def __init__(self, config):
        super().__init__()

        # Normalization layers
        self.attn_norm = RMSNorm(config.hidden_size)
        self.ffn_norm = RMSNorm(config.hidden_size)

        # Sublayers
        self.attention = CausalSelfAttention(config)
        self.ffn = SwiGLUFeedForward(
            config.hidden_size,
            config.ff_dim,
            config.dropout
```

```
        )

        # Optional dropout on residual paths
        self.resid_dropout = nn.Dropout(config.dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Attention with pre-norm
        # x = x + attention(norm(x))
        normed = self.attn_norm(x)
        attn_out = self.attention(normed)
        x = x + self.resid_dropout(attn_out)

        # FFN with pre-norm
        # x = x + ffn(norm(x))
        normed = self.ffn_norm(x)
        ffn_out = self.ffn(normed)
        x = x + self.resid_dropout(ffn_out)

        return x
```

Code Showing Pre-Norm Transformer Block

Here's the complete, production-ready block combining everything from this case file:

```
class ModernTransformerBlock(nn.Module):
    """
    Complete Transformer block with:
    - Pre-normalization (RMSNorm)
    - SwiGLU feed-forward
    - Rotary Position Embeddings (applied in attention)
    - Flash Attention (via F.scaled_dot_product_attention)
    """
    def __init__(self, config):
        super().__init__()
        self.hidden_size = config.hidden_size

        # Pre-normalization with RMSNorm
        self.attn_norm = RMSNorm(config.hidden_size, eps=1e-6)
        self.ffn_norm = RMSNorm(config.hidden_size, eps=1e-6)

        # Multi-head self-attention
        self.attention = CausalSelfAttention(config)

        # SwiGLU feed-forward network
        self.ffn = SwiGLUFeedForward(
            hidden_size=config.hidden_size,
            ff_dim=int(config.hidden_size * 8 / 3),  # Adjusted for SwiGLU
            dropout=config.dropout
```

```
        )

        # Residual dropout
        self.dropout = nn.Dropout(config.dropout)

    def forward(
        self,
        x: torch.Tensor,
        freqs_cos: torch.Tensor = None,
        freqs_sin: torch.Tensor = None
    ) -> torch.Tensor:
        """
        Args:
            x: Input tensor [batch, seq_len, hidden_size]
            freqs_cos: RoPE cosine frequencies
            freqs_sin: RoPE sine frequencies
        """
        # Attention sublayer with pre-norm
        residual = x
        x = self.attn_norm(x)
        x = self.attention(x, freqs_cos, freqs_sin)
        x = residual + self.dropout(x)

        # FFN sublayer with pre-norm
        residual = x
        x = self.ffn_norm(x)
        x = self.ffn(x)
        x = residual + self.dropout(x)

        return x


class Transformer(nn.Module):
    """Complete Transformer model with N blocks."""
    def __init__(self, config):
        super().__init__()

        # Token embedding
        self.tok_embed = nn.Embedding(config.vocab_size, config.hidden_size)

        # Transformer blocks
        self.blocks = nn.ModuleList([
            ModernTransformerBlock(config) for _ in range(config.num_layers)
        ])

        # Final normalization (crucial for pre-norm architecture!)
        self.final_norm = RMSNorm(config.hidden_size)
```

```
        # Language model head
        self.lm_head = nn.Linear(config.hidden_size, config.vocab_size, bias=False)

        # Weight tying
        self.lm_head.weight = self.tok_embed.weight

        # Initialize weights
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.02)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            nn.init.normal_(module.weight, std=0.02)

    def forward(self, input_ids: torch.Tensor) -> torch.Tensor:
        # Embed tokens
        x = self.tok_embed(input_ids)

        # Apply transformer blocks
        for block in self.blocks:
            x = block(x)

        # Final normalization (important!)
        x = self.final_norm(x)

        # Project to vocabulary
        logits = self.lm_head(x)

        return logits
```

Note the `final_norm` before the LM head. In pre-norm architectures, the last block's output hasn't been normalized. Without this final normalization, the logits have unstable statistics. I learned this the hard way in iteration #38.

Detective's Notebook: The Accidental Averaging Bug

Personal notes, 3:47 AM, coffee cold, hope colder

Three days. Seventy-two hours. The loss was stuck at 10.5 and wouldn't budge.

I'd changed everything. Learning rate sweeps. Batch size experiments. Different optimizers. I even rewrote the data loader from scratch, convinced there was a tokenization bug. Nothing worked.

The training loop ran. No errors. No NaN. Just a flat line on the loss curve, mocking me.

Iteration #45: The Symptoms

```
Epoch 1, Step 100:  Loss 10.521
Epoch 1, Step 200:  Loss 10.519
Epoch 1, Step 300:  Loss 10.523
Epoch 1, Step 400:  Loss 10.520
...
Epoch 3, Step 1000: Loss 10.517
Epoch 3, Step 2000: Loss 10.522
```

The loss oscillated by 0.006. For three days. The model was learning *nothing*.

I checked the gradients. They existed—small but non-zero. The optimizer was updating weights. The forward pass produced different logits each time. Everything *looked* alive.

But the loss was dead.

The Investigation

At 2 AM on day three, I started adding print statements. Not logging. Print statements. The desperate debug of a desperate engineer.

```python
def forward(self, x):
    print(f"Input shape: {x.shape}")

    x = self.attention(self.attn_norm(x))
    print(f"After attention: {x.shape}")

    x = self.ffn(self.ffn_norm(x))
    print(f"After FFN: {x.shape}")

    return x
```

The output:

```
Input shape: torch.Size([32, 64, 512])
After attention: torch.Size([32, 64, 512])
After FFN: torch.Size([32, 1, 512])
```

Wait.

Wait.

`[32, 1, 512]`?

The sequence length collapsed to 1. Sixty-four tokens became one. Every position in the sequence was being averaged together.

The Crime Scene

I traced it to my RMSNorm implementation:

```python
# The criminal code (Iteration #45)
class BuggyRMSNorm(nn.Module):
```

```
    def __init__(self, hidden_size, eps=1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(hidden_size))

    def forward(self, x):
        # x: [batch, seq_len, hidden_size]

        # THE BUG: .mean() without specifying dimensions!
        rms = torch.sqrt(x.pow(2).mean() + self.eps)  # <-- WRONG!

        # This returns a scalar, not [batch, seq_len, 1]
        # Broadcasting makes x / rms valid but meaningless

        return x / rms * self.weight
```

I'd written `.mean()` instead of `.mean(dim=-1, keepdim=True)`.

Without the `dim=-1` argument, PyTorch computes the mean over *all* dimensions. The entire batch. Every sequence. Every position. Every feature. One scalar.

Dividing by this scalar was mathematically valid. PyTorch didn't complain. The tensor shapes even looked right most of the time—until they didn't.

Why the Shapes Looked OK (Until They Didn't)

Here's the insidious part. The RMSNorm output shape *was* correct: `[32, 64, 512]`. Division by a scalar broadcasts fine. The bug only manifested in the FFN:

```
class FFN(nn.Module):
    def forward(self, x):
        # x comes in normalized (but incorrectly)
        x = self.up_proj(x)
        x = F.silu(x)
        x = self.down_proj(x)

        # Some version of my FFN had this at the end:
        return x.mean(dim=1, keepdim=True)  # SECOND BUG: leftover debug code!
```

A second bug. I'd added `.mean(dim=1, keepdim=True)` during a previous debugging session to check intermediate shapes. I never removed it.

One bug masked another. The RMSNorm computed wrong statistics. The FFN averaged away the sequence dimension. Together, they produced `[B, 1, C]` instead of `[B, T, C]`.

The Fix

```
# Fixed RMSNorm
class RMSNorm(nn.Module):
    def forward(self, x):
        # CORRECT: mean over last dimension only, keeping shape
```

```
        rms = torch.sqrt(x.pow(2).mean(dim=-1, keepdim=True) + self.eps)
        return x / rms * self.weight

# Fixed FFN
class FFN(nn.Module):
    def forward(self, x):
        x = self.up_proj(x)
        x = F.silu(x)
        x = self.down_proj(x)
        return x  # NO .mean()! Just return the tensor!
```

The Lesson Burned Into My Brain

```
# The assertion I now add to EVERY forward pass
def forward(self, x):
    B, T, C = x.shape

    # ... do computation ...

    assert output.shape == (B, T, C), \
        f"Shape mismatch! Expected {(B, T, C)}, got {output.shape}"

    return output
```

Shape assertions. In every layer. In every forward pass. The five seconds it takes to write them saves three days of staring at flatlined loss curves.

The model trained after the fix. Loss dropped to 4.2 in an hour. By the next morning, we were at 2.8 and falling.

Three days of my life, burned by a missing `dim=-1`. Welcome to deep learning.

End of Case File #6. Proceed to Case File #7: The Training Loop. # Case File #7: The Training Loop

"The training loop is where theory meets reality—where your beautiful architecture either learns or dies. It's the stakeout that never ends. You sit in the dark, watching loss curves like a detective watches a suspect's apartment. Hours pass. The loss ticks down. You start to hope. Then, at batch 847, it happens: NaN. Not a Number. Your model just went insane, and you have no idea why. Welcome to the graveyard shift."

7.1 Loss Functions

Every investigation needs a way to measure progress. Are we getting closer to the truth? In neural networks, that measure is the loss function—a single number that tells you how wrong your model is. The goal: make it smaller.

Cross-Entropy Loss: The Foundation of Next-Token Prediction

For language models, the loss function is cross-entropy. It answers a simple question: how surprised was the model by the correct answer?

If the model assigns 90% probability to the correct next token, it's not very surprised. Low loss. Good.

If the model assigns 0.1% probability to the correct token? Extremely surprised. High loss. Bad.

The intuition: cross-entropy measures the gap between what the model predicted and what actually happened. Close that gap, and you have a working language model.

The Math: L = -Σ(y_true × log(y_pred))

The formula that governs everything:

$$L = -\sum_{i} y_{true}^{(i)} \cdot \log(y_{pred}^{(i)})$$

For next-token prediction, y_{true} is a one-hot vector—all zeros except for a 1 at the correct token's position. This simplifies the formula:

$$L = -\log(p_{correct})$$

If the model assigns probability 0.9 to the correct token: $L = -\log(0.9) = 0.105$ If the model assigns probability 0.01: $L = -\log(0.01) = 4.605$

The logarithm creates a sharp penalty for confident wrong answers. Assign 0.0001 probability to the correct token? $L = 9.21$. The loss explodes.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

def cross_entropy_from_scratch(logits: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:
    """
    Cross-entropy loss, the hard way.

    Args:
        logits: [batch, seq_len, vocab_size] - raw model outputs
        targets: [batch, seq_len] - integer token IDs
    """
    # Step 1: Convert logits to probabilities via softmax
    probs = F.softmax(logits, dim=-1)  # [batch, seq_len, vocab_size]

    # Step 2: Gather the probability of the correct token at each position
    # targets.unsqueeze(-1) -> [batch, seq_len, 1]
    correct_probs = probs.gather(dim=-1, index=targets.unsqueeze(-1)).squeeze(-1)
    # correct_probs: [batch, seq_len]
```

```
    # Step 3: Take negative log
    loss = -torch.log(correct_probs + 1e-9)  # Add epsilon to prevent log(0)

    # Step 4: Average over all positions
    return loss.mean()
```

In practice, PyTorch's `F.cross_entropy` does this more efficiently—and numerically stably—by combining softmax and log into a single operation:

```
# The practical way: let PyTorch handle the numerics
loss = F.cross_entropy(
    logits.view(-1, vocab_size),  # [batch * seq_len, vocab_size]
    targets.view(-1),              # [batch * seq_len]
)
```

`ignore_index=tokenizer.pad_token_id`: Don't Train on Padding

Here's a mistake that cost me three days: training on padding tokens.

When you batch sequences of different lengths, you pad the shorter ones to match the longest. But those padding tokens aren't real data. They're filler. Training the model to predict them teaches nothing useful—and can actively harm performance.

```
# The crime scene: before ignore_index
sequences = [
    [101, 2054, 2003, 1996, 102, 0, 0, 0],  # "What is the" + padding
    [101, 2129, 2024, 2017, 102, 0, 0, 0],  # "How are you" + padding
]
# The model was learning to predict 0 after seeing 102
# This is useless knowledge

# The fix: tell the loss function to ignore padding
loss_fn = nn.CrossEntropyLoss(ignore_index=tokenizer.pad_token_id)
```

The `ignore_index` parameter tells PyTorch to skip those positions entirely when computing the loss. No gradient flows from padding tokens. The model learns only from real data.

```
# Production loss function setup
def create_loss_function(tokenizer):
    """
    Create a loss function that ignores padding tokens.
    This is essential for efficient training.
    """
    return nn.CrossEntropyLoss(
        ignore_index=tokenizer.pad_token_id,
        reduction='mean'  # Average over non-ignored positions
    )

# Usage in training loop
loss_fn = create_loss_function(tokenizer)
```

```
# Compute loss (padding positions contribute nothing)
logits = model(input_ids)  # [batch, seq_len, vocab_size]
loss = loss_fn(
    logits[:, :-1, :].contiguous().view(-1, vocab_size),  # Predictions
    input_ids[:, 1:].contiguous().view(-1)                 # Targets (shifted)
)
```

Label Smoothing: A Regularization Technique

By iteration #45, I discovered label smoothing. Instead of training the model to assign 100% probability to the correct token, you train it to assign 90% (or some high percentage) to the correct token and spread the remaining 10% across all other tokens.

Why? It prevents overconfidence. A model that always predicts with 99.9% confidence becomes brittle—it can't express uncertainty. Label smoothing keeps the probability distribution slightly softer.

$$y_{smooth} = (1 - \alpha) \cdot y_{true} + \frac{\alpha}{K}$$

Where α is the smoothing factor (typically 0.1) and K is the vocabulary size.

```
# Label smoothing in practice
loss_fn = nn.CrossEntropyLoss(
    ignore_index=tokenizer.pad_token_id,
    label_smoothing=0.1  # 10% smoothing
)

# Effect: instead of training toward [0, 0, 1, 0, 0] (one-hot)
# We train toward [0.025, 0.025, 0.9, 0.025, 0.025] (smoothed)
# This prevents the model from becoming pathologically confident
```

Code Showing Loss Calculation with Proper ignore_index

Here's the complete, production-ready loss calculation:

```
class LanguageModelLoss(nn.Module):
    """
    Complete loss function for language model training.
    Handles padding, label smoothing, and proper tensor alignment.
    """
    def __init__(
        self,
        pad_token_id: int,
        vocab_size: int,
        label_smoothing: float = 0.0
    ):
        super().__init__()
        self.pad_token_id = pad_token_id
        self.vocab_size = vocab_size
```

```
        self.criterion = nn.CrossEntropyLoss(
            ignore_index=pad_token_id,
            label_smoothing=label_smoothing,
            reduction='mean'
        )

    def forward(
        self,
        logits: torch.Tensor,
        labels: torch.Tensor
    ) -> torch.Tensor:
        """
        Compute language modeling loss.

        Args:
            logits: [batch, seq_len, vocab_size] - model outputs
            labels: [batch, seq_len] - target token IDs

        Note: Assumes logits[i] predicts labels[i+1]
              (standard autoregressive setup)
        """
        # Shift: logits[:-1] predict labels[1:]
        shift_logits = logits[:, :-1, :].contiguous()
        shift_labels = labels[:, 1:].contiguous()

        # Flatten for cross-entropy
        flat_logits = shift_logits.view(-1, self.vocab_size)
        flat_labels = shift_labels.view(-1)

        # Compute loss (padding positions automatically ignored)
        loss = self.criterion(flat_logits, flat_labels)

        return loss

# Usage
loss_fn = LanguageModelLoss(
    pad_token_id=tokenizer.pad_token_id,
    vocab_size=tokenizer.vocab_size,
    label_smoothing=0.1
)

logits = model(input_ids)
loss = loss_fn(logits, input_ids)
loss.backward()
```

7.2 Optimizers: The AdamW Revolution

The loss function tells you how wrong you are. The optimizer tells you how to fix it. Choosing the right optimizer is the difference between a model that learns and one that wanders aimlessly.

Iteration #1: Basic `torch.optim.Adam(lr=5e-5)`

My first optimizer choice was obvious: Adam. Everyone uses Adam. It's the default. It works.

```
# Iteration #1: The naive approach
optimizer = torch.optim.Adam(model.parameters(), lr=5e-5)
```

Adam was fine. The model trained. Loss decreased. But "fine" isn't optimal.

Why Adam Works: Adaptive Learning Rates

Adam's genius is adaptive learning rates. Instead of using the same learning rate for every parameter, Adam adjusts the rate based on historical gradients.

The intuition: some parameters need aggressive updates (they're stuck in flat regions). Others need gentle updates (they're near a good solution). Adam tracks two things for each parameter:

1. **First moment (mean of gradients):** Which direction should we go?
2. **Second moment (mean of squared gradients):** How variable are the gradients?

Parameters with consistent gradients get larger updates. Parameters with noisy gradients get smaller updates.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}$$

```
# What Adam does internally (simplified)
def adam_update(param, grad, m, v, lr, beta1=0.9, beta2=0.999, eps=1e-8):
    # Update biased first moment estimate
    m = beta1 * m + (1 - beta1) * grad

    # Update biased second moment estimate
    v = beta2 * v + (1 - beta2) * (grad ** 2)

    # Compute update: larger updates for consistent gradients
    update = lr * m / (torch.sqrt(v) + eps)

    return param - update, m, v
```

Iteration #12: Switching to AdamW with Weight Decay 0.01

By iteration #12, I learned about AdamW. The paper "Decoupled Weight Decay Regularization" by Loshchilov and Hutter changed everything.

The problem with L2 regularization in Adam: it gets tangled with the adaptive learning rates. You add the regularization term to the gradient, but then Adam's moment estimates modify it unpredictably.

AdamW fixes this by decoupling weight decay from the gradient update.

```
# Iteration #12: The upgrade
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=3e-4,           # Higher LR for pretraining
    weight_decay=0.01,  # Decoupled weight decay
    betas=(0.9, 0.999),
    eps=1e-8
)
```

The Difference: AdamW Applies Weight Decay Correctly

```
# Adam with L2 regularization (wrong way):
grad = grad + weight_decay * param  # Regularization added to gradient
m = beta1 * m + (1 - beta1) * grad  # Adam modifies the combined signal
# The regularization effect is scaled by Adam's adaptive factors

# AdamW (correct way):
m = beta1 * m + (1 - beta1) * grad  # Adam operates on pure gradient
param = param - lr * (adam_update + weight_decay * param)  # Decay applied separately
# Weight decay is truly decoupled
```

The result? More predictable regularization. Better generalization. Cleaner loss curves.

Learning Rate Evolution: 5e-5 → 3e-4 → 2e-4 (Pretraining) → 5e-6 (SFT)

Over 100 iterations, my learning rate strategy evolved:

Phase	Learning Rate	Reasoning
Iteration #1	5e-5	Conservative. Borrowed from BERT fine-tuning
Iteration #23	3e-4	Pretraining needs higher LR to escape random init
Iteration #67	2e-4	Settled on this for large-batch pretraining
SFT Phase	5e-6	Much lower—model already knows language, just adjusting behavior

```
# Phase-aware optimizer setup
def create_optimizer(model, phase: str) -> torch.optim.Optimizer:
    """
    Create optimizer with phase-appropriate learning rate.
    """
```

```
    lr_schedule = {
        'pretraining': 2e-4,
        'sft': 5e-6,
        'domain_ft': 1e-6
    }

    lr = lr_schedule.get(phase, 1e-4)

    return torch.optim.AdamW(
        model.parameters(),
        lr=lr,
        weight_decay=0.01,
        betas=(0.9, 0.95),  # Slightly lower beta2 for transformers
        eps=1e-8
    )
```

Code Showing Optimizer Initialization

Here's the production setup with parameter groups:

```
def create_adamw_optimizer(model, config):
    """
    Create AdamW optimizer with proper parameter groups.
    Different weight decay for different parameter types.
    """
    # Separate parameters that should/shouldn't have weight decay
    decay_params = []
    no_decay_params = []

    for name, param in model.named_parameters():
        if not param.requires_grad:
            continue

        # Don't apply weight decay to biases and layer norms
        if 'bias' in name or 'norm' in name or 'ln' in name:
            no_decay_params.append(param)
        else:
            decay_params.append(param)

    param_groups = [
        {'params': decay_params, 'weight_decay': config.weight_decay},
        {'params': no_decay_params, 'weight_decay': 0.0}
    ]

    optimizer = torch.optim.AdamW(
        param_groups,
        lr=config.learning_rate,
        betas=(0.9, 0.95),
```

```
        eps=1e-8
    )

    print(f"Optimizer created:")
    print(f"  - Parameters with decay: {len(decay_params)}")
    print(f"  - Parameters without decay: {len(no_decay_params)}")
    print(f"  - Learning rate: {config.learning_rate}")
    print(f"  - Weight decay: {config.weight_decay}")

    return optimizer
```

7.3 Learning Rate Schedules

A constant learning rate is like driving at the same speed everywhere—40 mph in parking lots and on highways. You need to adapt.

Why Constant Learning Rate Is Suboptimal

Early in training, the model is randomly initialized. Gradients are noisy. Large learning rates cause wild oscillations. The model overshoots good solutions.

Late in training, the model is nearly converged. Gradients are small. The same learning rate that was too large before is now too large again—it bounces around the optimum instead of settling in.

The solution: learning rate schedules that adapt over time.

Warmup Steps: 1000 → 2000 → 3000

Warmup is the slow start. For the first N steps, the learning rate gradually increases from near-zero to the target rate.

Why? Random initialization means the gradients at step 0 are garbage. Huge learning rates on garbage gradients → exploding activations → NaN loss.

My evolution: - **Iteration #15:** 1000 warmup steps. Sometimes worked. - **Iteration #34:** 2000 warmup steps. More stable. - **Iteration #78:** 3000 warmup steps (for 100k+ step runs). Bulletproof.

The rule of thumb: warmup should be 1-3% of total training steps.

Why Warmup Prevents Early Training Collapse

At initialization, the model's layers output random values. The gradients computed from these outputs are essentially noise—they point in random directions with unpredictable magnitudes.

If you immediately apply a large learning rate to noise, you get noise amplification. Weights fly to extreme values. Activations explode. Loss becomes NaN.

Warmup acts as a stabilizer: 1. Tiny learning rate → tiny updates → model stays near initialization 2. After a few hundred steps, the model has learned something. Gradients become meaningful. 3. Now you can safely increase the learning rate.

```
def warmup_schedule(step: int, warmup_steps: int, max_lr: float) -> float:
    """
    Linear warmup: LR increases linearly from 0 to max_lr.
    """
    if step < warmup_steps:
        return max_lr * (step / warmup_steps)
    return max_lr
```

Cosine Decay with Minimum LR Factor 0.1

After warmup, the learning rate should decrease. The most popular schedule is cosine decay—a smooth curve from max LR to min LR.

$$lr = lr_{min} + 0.5 \cdot (lr_{max} - lr_{min}) \cdot (1 + \cos(\pi \cdot t/T))$$

Where t is the current step and T is total steps.

The cosine shape is gentle at first (when you still need large updates) and steeper at the end (when you're fine-tuning).

```
import math

def cosine_schedule(
    step: int,
    warmup_steps: int,
    total_steps: int,
    max_lr: float,
    min_lr_factor: float = 0.1
) -> float:
    """
    Warmup + cosine decay schedule.

    Args:
        step: Current training step
        warmup_steps: Number of warmup steps
        total_steps: Total training steps
        max_lr: Maximum learning rate (reached after warmup)
        min_lr_factor: Minimum LR as fraction of max (default: 0.1)
    """
    min_lr = max_lr * min_lr_factor

    # Warmup phase
    if step < warmup_steps:
        return max_lr * (step / warmup_steps)

    # Cosine decay phase
    progress = (step - warmup_steps) / (total_steps - warmup_steps)
    cosine_decay = 0.5 * (1 + math.cos(math.pi * progress))
```

```
    return min_lr + (max_lr - min_lr) * cosine_decay
```

Code Implementing Warmup + Cosine Schedule

The complete learning rate scheduler:

```
class WarmupCosineScheduler:
    """
    Learning rate scheduler with linear warmup and cosine decay.
    Used in GPT-3, LLaMA, and most modern LLM training.
    """
    def __init__(
        self,
        optimizer: torch.optim.Optimizer,
        warmup_steps: int,
        total_steps: int,
        max_lr: float,
        min_lr_factor: float = 0.1
    ):
        self.optimizer = optimizer
        self.warmup_steps = warmup_steps
        self.total_steps = total_steps
        self.max_lr = max_lr
        self.min_lr = max_lr * min_lr_factor
        self.current_step = 0

    def get_lr(self) -> float:
        """Calculate learning rate for current step."""
        if self.current_step < self.warmup_steps:
            # Linear warmup
            return self.max_lr * (self.current_step / self.warmup_steps)

        # Cosine decay
        progress = (self.current_step - self.warmup_steps) / (
            self.total_steps - self.warmup_steps
        )
        progress = min(progress, 1.0)  # Clamp to avoid going past total_steps

        cosine_decay = 0.5 * (1 + math.cos(math.pi * progress))
        return self.min_lr + (self.max_lr - self.min_lr) * cosine_decay

    def step(self):
        """Update learning rate and advance step counter."""
        lr = self.get_lr()
        for param_group in self.optimizer.param_groups:
            param_group['lr'] = lr
        self.current_step += 1
```

```
        return lr

# Usage in training loop
scheduler = WarmupCosineScheduler(
    optimizer=optimizer,
    warmup_steps=2000,
    total_steps=100000,
    max_lr=2e-4,
    min_lr_factor=0.1
)

for step, batch in enumerate(dataloader):
    loss = train_step(model, batch, optimizer)

    # Update learning rate after each step
    current_lr = scheduler.step()

    if step % 100 == 0:
        print(f"Step {step} | Loss: {loss:.4f} | LR: {current_lr:.2e}")
```

7.4 Gradient Clipping: Taming the Explosion

Gradients in deep networks are computed via backpropagation—multiplying many small numbers together. Sometimes, those numbers aren't small. They multiply into astronomically large values. This is gradient explosion.

What Exploding Gradients Look Like

The symptoms are unmistakable: - Loss suddenly becomes `inf` or `nan` - Model outputs become garbage - Training is irrecoverable without loading a checkpoint

```
# The telltale signs in your logs:
# Step 845: Loss = 4.23
# Step 846: Loss = 4.21
# Step 847: Loss = nan   <-- Disaster
# Step 848: Loss = nan
# ...

# Or gradients that grow without bound:
# Gradient norm: 1.2
# Gradient norm: 3.7
# Gradient norm: 47.8
# Gradient norm: 12847.3
# Gradient norm: inf
# Loss: nan
```

`max_grad_norm = 1.0`: The Magic Number

The fix is gradient clipping. Before applying gradients, you check their total magnitude (norm). If it exceeds a threshold, you scale all gradients down proportionally.

```
# The magic line that prevents NaN
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

Why 1.0? It's convention. Large enough to allow meaningful updates. Small enough to prevent explosions. In practice, values between 0.5 and 2.0 all work. I've never needed anything outside that range.

The math: compute the L2 norm of all gradients. If it exceeds `max_norm`, scale every gradient by `max_norm / actual_norm`.

```
def clip_grad_norm_manual(parameters, max_norm: float) -> float:
    """
    Gradient clipping, implemented from scratch.
    """
    parameters = list(parameters)

    # Compute total gradient norm
    total_norm = 0.0
    for p in parameters:
        if p.grad is not None:
            total_norm += p.grad.data.norm(2).item() ** 2
    total_norm = total_norm ** 0.5

    # Clip if necessary
    clip_coef = max_norm / (total_norm + 1e-6)
    if clip_coef < 1.0:
        for p in parameters:
            if p.grad is not None:
                p.grad.data.mul_(clip_coef)

    return total_norm
```

The NaN Loss Incident at Batch 847

It was 2:47 AM. The model had been training for 14 hours. Loss was 4.23—on track for a personal best. Then batch 847 happened.

I didn't notice immediately. I was getting coffee. When I returned, the loss read `nan`. Not infinity. Not a large number. `nan`. Not a Number. The mathematical equivalent of "I give up."

The postmortem revealed the crime: 1. Batch 847 contained an unusually long sequence 2. The attention scores grew larger than usual 3. After softmax, some positions had probability 0.9999… 4. The cross-entropy of near-certainty is near-zero 5. But the *gradient* of near-zero loss is *huge* (derivative of log near zero) 6. The gradient exploded → weights flew to infinity → activations became NaN

One batch. One unlucky sequence. Fourteen hours of training, gone.

The fix was trivial. One line I'd forgotten:

```
# Before (Iteration #19): No clipping. Trust the gradients. Get burned.
loss.backward()
optimizer.step()

# After: Always clip. No exceptions.
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```

Code Showing Gradient Clipping Implementation

The complete training step with all stability measures:

```
def training_step(
    model: nn.Module,
    batch: dict,
    optimizer: torch.optim.Optimizer,
    scheduler: WarmupCosineScheduler,
    scaler: torch.cuda.amp.GradScaler,
    config
) -> dict:
    """
    Single training step with gradient clipping and mixed precision.
    This is the battle-tested version from iteration #89.
    """
    model.train()

    # Zero gradients (never forget this!)
    optimizer.zero_grad()

    input_ids = batch['input_ids'].to(config.device)

    # Forward pass with mixed precision
    with torch.cuda.amp.autocast(dtype=torch.bfloat16):
        logits = model(input_ids)

        # Compute loss (shifted internally)
        loss = F.cross_entropy(
            logits[:, :-1, :].contiguous().view(-1, config.vocab_size),
            input_ids[:, 1:].contiguous().view(-1),
            ignore_index=config.pad_token_id
        )

    # Backward pass with gradient scaling
    scaler.scale(loss).backward()

    # Unscale gradients for clipping
```

```
scaler.unscale_(optimizer)

# THE CRITICAL LINE: Clip gradients to prevent explosion
grad_norm = torch.nn.utils.clip_grad_norm_(
    model.parameters(),
    max_norm=config.max_grad_norm  # Usually 1.0
)

# Check for NaN gradients (paranoia from batch 847)
if torch.isnan(grad_norm) or torch.isinf(grad_norm):
    print(f"WARNING: NaN/Inf gradient detected! Skipping batch.")
    optimizer.zero_grad()
    return {'loss': float('nan'), 'grad_norm': float('nan'), 'skipped': True}

# Optimizer step with gradient scaling
scaler.step(optimizer)
scaler.update()

# Update learning rate
current_lr = scheduler.step()

return {
    'loss': loss.item(),
    'grad_norm': grad_norm.item(),
    'lr': current_lr,
    'skipped': False
}
```

Detective's Notebook: The Warmup Revelation

2:34 AM. The screen glows in the dark office. Loss: 8.74. Step 47. This should be working.

I'd built my first serious transformer—18 layers, 200 million parameters. The architecture was clean. The data loader was streaming Wikipedia. Everything was ready.

Except for warmup. I hadn't added warmup. "It's just a minor optimization," I'd told myself. "The model will figure it out."

The model did not figure it out.

Step 1: Loss = 11.2. Expected. Random initialization. Step 10: Loss = 10.8. Decreasing. Good sign. Step 50: Loss = 12.4. Wait, it went up? Step 100: Loss = 847.3. What. Step 101: Loss = inf Step 102: Loss = nan

I stared at the screen. The model had trained for 47 seconds before exploding.

The postmortem took two hours. I printed gradient norms. They started at 0.8, grew to 12, then 340, then 12,847, then infinity. The gradients weren't just exploding—they were detonating.

The problem: at random initialization, the model's outputs are noise. Softmax on noise produces a

nearly uniform distribution. Cross-entropy loss on a uniform distribution is $\log(vocab\_size)$—about 11.5 for my 100K vocabulary. That's fine.

But the *gradients* of that loss are computed through 18 layers of randomly initialized matrices. Each layer can amplify or shrink the gradient. With random initialization, the amplification was winning. By a lot.

A large learning rate (3e-4) applied to explosively large gradients sent the weights flying to extreme values. The next forward pass produced even more extreme outputs. The next backward pass produced even larger gradients. Positive feedback loop. Detonation.

The Experiment Without Warmup

I ran it three times to be sure:

```
Run 1: Exploded at step 101
Run 2: Exploded at step 87
Run 3: Exploded at step 134
```

Always within the first 200 steps. Always the same pattern: loss increases, gradients explode, NaN.

The Fix

I added warmup. 2000 steps of linear ramp-up from near-zero to full learning rate.

```
# The line that saved everything
warmup_steps = 2000
if step < warmup_steps:
    lr = max_lr * (step / warmup_steps)
else:
    lr = max_lr * cosine_decay(step)

for param_group in optimizer.param_groups:
    param_group['lr'] = lr
```

The Result

```
Step 1: Loss = 11.2, LR = 1e-7
Step 100: Loss = 10.4, LR = 1e-5
Step 500: Loss = 8.7, LR = 7.5e-5
Step 1000: Loss = 6.3, LR = 1.5e-4
Step 2000: Loss = 4.8, LR = 3e-4 (warmup complete)
Step 10000: Loss = 3.2
Step 50000: Loss = 2.4
```

No explosion. The loss decreased monotonically. The gradients stayed bounded. The model learned.

Warmup isn't an optimization. It's survival. Without it, your model doesn't get the chance to learn—it dies in the first hundred steps, before the investigation even begins.

That night, I added warmup to every training script I'd ever write. I set the warmup steps high enough to be safe—3000 for long runs, 1000 for short ones. And I never forgot the lesson:

The first steps of training are the most dangerous. Start slow. Build momentum. Let the gradients stabilize before you push the accelerator.

"The training loop isn't glamorous. It's not the architecture—that's the star witness. It's not the data—that's the evidence. The training loop is the legwork. The hours of surveillance. The careful documentation. Get it wrong, and your case falls apart at step 847. Get it right, and the model learns to speak. That's the job. That's the grind. That's 100 iterations of doing it wrong before you do it right." # Case File #8: The Graveyard of Errors (Debug Logs)

"Every model that works is built on the graves of a hundred that didn't. They don't tell you this in the papers. The arxiv submissions show clean loss curves, elegant architectures, state-of-the-art benchmarks. What they don't show is the 3 AM debugging sessions, the NaN losses that appear from nowhere, the models that train perfectly and generate garbage. This chapter is my confession—a catalog of every bug that made me question my career choice. Welcome to the graveyard. Bring a flashlight."

This is the chapter I wish someone had written for me. Not the theory. Not the elegant mathematics. The ugly truth: the bugs that don't throw errors, the silent failures that waste weeks, the gotchas that every transformer implementer discovers the hard way.

Every bug in this chapter cost me at least a day. Some cost weeks. All of them taught me something I couldn't learn from papers.

8.1 The "Cheating" Bugs (Logic Errors)

The most insidious bugs aren't the ones that crash your training. They're the ones that make training look *too good*. When your loss drops faster than expected, when convergence happens in hours instead of days, when everything looks perfect—that's when you should be most suspicious.

The Leaky Mask (Iteration #7)

The Scene of the Crime

Iteration #7. I had just implemented my first causal attention mask. The theory was clear: in autoregressive models, position i should only attend to positions $0, 1, ..., i-1$. Never to position $i+1$ or beyond. The future is forbidden.

I wrote the mask. I ran the training. And something miraculous happened.

```
# Training log, Iteration #7
# Batch 1: Loss = 10.82
# Batch 10: Loss = 6.45
# Batch 50: Loss = 2.13
# Batch 100: Loss = 0.14  <-- Wait, what?
```

Loss of 0.14 after 100 batches? On a 50,000-token vocabulary? The theoretical minimum for random guessing is $\ln(50000) \approx 10.8$. Getting to 0.14 would require the model to predict the correct token with near-perfect accuracy.

I was elated. I had built a genius.

The Reveal

Then I tried generation.

```
prompt = "The capital of France is"
generated = model.generate(prompt, max_tokens=20)
print(generated)
# Output: "The capital of France is sssssssssssssssss"
```

Garbage. Pure, unmitigated garbage.

I tried another prompt.

```
prompt = "Once upon a time"
generated = model.generate(prompt, max_tokens=20)
print(generated)
# Output: "Once upon a time                                    "
```

Spaces. Just… spaces.

Detective Work

Something was deeply wrong. The model had learned *something*, but not language. I started digging.

First, I visualized the attention patterns:

```
def visualize_attention_pattern(model, input_ids):
    """Visualize what tokens attend to what."""
    with torch.no_grad():
        outputs = model(input_ids, output_attentions=True)
        attention = outputs.attentions[-1][0, 0]  # Last layer, first head

    plt.figure(figsize=(10, 10))
    plt.imshow(attention.cpu().numpy())
    plt.xlabel("Key Position (what we attend TO)")
    plt.ylabel("Query Position (what is ATTENDING)")
    plt.colorbar()
    plt.title("Attention Pattern - Should be lower triangular!")
    plt.show()

visualize_attention_pattern(model, tokenizer.encode("Hello world how are you"))
```

The attention pattern was *symmetric*. Position 2 was attending to position 5. Position 3 was attending to position 7. The model could see the future.

I had implemented a leaky mask.

The Crime Scene: The Buggy Code

```
# THE BUGGY IMPLEMENTATION (Iteration #7)
def create_causal_mask(seq_len: int) -> torch.Tensor:
    """
    What I thought was a causal mask.
    """
```

```
    # Create lower triangular matrix (1s where we CAN attend)
    mask = torch.tril(torch.ones(seq_len, seq_len))
    return mask

class BuggyAttention(nn.Module):
    def forward(self, x):
        B, T, C = x.shape

        # ... Q, K, V projections ...
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.head_dim)

        # Apply mask
        mask = create_causal_mask(T).to(x.device)
        scores = scores.masked_fill(mask == 0, 0)  # <-- THE BUG!

        attn_weights = F.softmax(scores, dim=-1)
        output = torch.matmul(attn_weights, V)

        return output
```

Do you see it? I filled the masked positions with `0`. Zero.

When you apply softmax to `[5.2, 3.1, 0, 0, 0]`, you get:

```
scores = torch.tensor([5.2, 3.1, 0, 0, 0])
probs = F.softmax(scores, dim=-1)
print(probs)
# tensor([0.5765, 0.1412, 0.0637, 0.0637, 0.0637])
```

The masked positions still get probability mass! $e^0 = 1$, not 0. The softmax was distributing attention to future tokens—tokens the model shouldn't be able to see.

The model wasn't learning to predict. It was learning to cheat. During training, it could see the answer in the input and just copy it. Loss dropped to near-zero because copying is easy. But during generation, when the future tokens don't exist yet, the model had no idea what to do.

The Fix

```
# THE FIXED IMPLEMENTATION (Iteration #8)
def create_causal_mask(seq_len: int, device: torch.device) -> torch.Tensor:
    """
    Proper causal mask. Forbidden positions get -infinity.
    """
    # Create upper triangular matrix of ones (above the diagonal)
    mask = torch.triu(torch.ones(seq_len, seq_len, device=device), diagonal=1)
    # Fill with negative infinity
    mask = mask.masked_fill(mask == 1, float('-inf'))
    return mask

class FixedAttention(nn.Module):
    def forward(self, x):
```

```
        B, T, C = x.shape

        # ... Q, K, V projections ...
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.head_dim)

        # Apply mask by ADDING (not replacing)
        mask = create_causal_mask(T, x.device)
        scores = scores + mask  # Adding -inf makes those positions go to 0 after softmax

        attn_weights = F.softmax(scores, dim=-1)
        output = torch.matmul(attn_weights, V)

        return output
```

Now when you apply softmax to `[5.2, 3.1, -inf, -inf, -inf]`:

```
scores = torch.tensor([5.2, 3.1, float('-inf'), float('-inf'), float('-inf')])
probs = F.softmax(scores, dim=-1)
print(probs)
# tensor([0.8909, 0.1091, 0.0000, 0.0000, 0.0000])
```

The masked positions get *exactly zero* probability. The future is truly forbidden.

The Lesson

If your training loss looks too good to be true, it is. The model is cheating. Find the leak.

The first thing I do now when loss drops suspiciously fast: visualize the attention patterns. If they're not strictly lower triangular, something is wrong with your mask.

The Off-By-One Target (Iteration #14)

The Scene of the Crime

Iteration #14. Mask was fixed. Model was training. Loss was dropping at a reasonable rate. But when I tested generation:

```
prompt = "The weather today is"
generated = model.generate(prompt, max_tokens=30)
print(generated)
# Output: "The weather today is weather today is weather today is weather today is"
```

The model was repeating. Looping. Caught in an echo chamber of its own making.

I tried different prompts. Same result. The model had learned to copy, not to continue.

Detective Work

I traced through my data pipeline step by step. The inputs looked fine. The tokenization was correct. Then I looked at the loss computation:

```
# THE BUGGY LOSS COMPUTATION (Iteration #14)
def compute_loss(model, input_ids):
```

```
    """
    What I thought was correct.
    """
    logits = model(input_ids)  # [B, T, vocab_size]

    # Compute loss between predictions and targets
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),    # [B*T, vocab_size]
        input_ids.view(-1)               # [B*T]  <-- THE BUG!
    )
    return loss
```

I was training the model to predict `input_ids` from `input_ids`. Position 0 predicting token 0. Position 1 predicting token 1. The model learned the identity function: output whatever you just saw.

That's not language modeling. Language modeling is *next-token prediction*. Position 0 should predict the token at position 1. Position 1 should predict position 2. Position i predicts position $i + 1$.

The Crime Scene: Input/Label Alignment

The correct alignment for autoregressive language modeling:

```
Input sequence:  [The] [cat] [sat] [on] [the] [mat]
Position:           0     1     2    3     4     5

For training:
- Position 0 input [The] → should predict [cat]  (position 1)
- Position 1 input [cat] → should predict [sat]  (position 2)
- Position 2 input [sat] → should predict [on]   (position 3)
...and so on

So:
- Inputs:  input_ids[:, :-1]  = [The, cat, sat, on, the]
- Labels:  input_ids[:, 1:]   = [cat, sat, on, the, mat]
```

I was using `input_ids` as both inputs and labels without shifting. The model was learning to predict what it already had, not what comes next.

The Fix

```
# THE FIXED LOSS COMPUTATION (Iteration #15)
def compute_loss(model, input_ids):
    """
    Correct next-token prediction loss.
    """
    # Get model predictions for all positions
    logits = model(input_ids)  # [B, T, vocab_size]

    # Shift for next-token prediction
    # Logits from position 0..T-2 predict tokens at position 1..T-1
```

```
    shift_logits = logits[:, :-1, :].contiguous()  # [B, T-1, vocab_size]
    shift_labels = input_ids[:, 1:].contiguous()   # [B, T-1]

    # Flatten and compute loss
    loss = F.cross_entropy(
        shift_logits.view(-1, vocab_size),  # [B*(T-1), vocab_size]
        shift_labels.view(-1)                # [B*(T-1)]
    )
    return loss
```

Alternative approach—shift inside the model's forward pass:

```
class LanguageModel(nn.Module):
    def forward(self, input_ids, labels=None):
        # input_ids: [B, T]
        logits = self.transformer(input_ids)  # [B, T, vocab_size]

        loss = None
        if labels is not None:
            # Shift inside forward for clarity
            # logits[i] predicts labels[i+1]
            shift_logits = logits[..., :-1, :].contiguous()
            shift_labels = labels[..., 1:].contiguous()

            loss = F.cross_entropy(
                shift_logits.view(-1, self.vocab_size),
                shift_labels.view(-1),
                ignore_index=self.pad_token_id
            )

        return logits, loss
```

The Lesson

For next-token prediction: `input[0:n-1]` predicts `labels[1:n]`. Always shift. The `.contiguous()` call ensures the shifted tensor has clean memory layout for the view operation that follows.

This bug is so common that most modern libraries handle it internally. But if you're building from scratch, you'll encounter it. And you'll spend days wondering why your model only echoes.

8.2 The "Silent" Math Bugs

Logic bugs crash or produce garbage. Math bugs are worse—they produce plausible results that are subtly, insidiously wrong. Your model trains. Your loss decreases. Everything looks fine. But the model never gets good. These bugs don't announce themselves. They hide in plain sight.

The Missing Scale Factor (Iteration #22)

The Scene of the Crime

Iteration #22. Custom attention implementation. Everything looked correct. The loss started high, dropped rapidly for the first 1000 steps, then… stopped.

```
# Training log, Iteration #22
# Step 100: Loss = 9.87
# Step 500: Loss = 8.52
# Step 1000: Loss = 8.51
# Step 2000: Loss = 8.50
# Step 5000: Loss = 8.49
# Step 10000: Loss = 8.48
# ...three days later...
# Step 100000: Loss = 8.47
```

The loss was stuck at 8.5. Random guessing on a 50,000-token vocabulary gives loss ≈ 10.8. So the model had learned *something*. But it should have dropped to 4.0 or lower. It was stuck.

Detective Work

I tried everything. Learning rate? Adjusted it up and down. Batch size? No change. More layers? Still stuck. Different datasets? Same plateau.

Finally, I looked at the attention weights themselves:

```
def analyze_attention_distribution(model, input_ids):
    """Check if attention is healthy."""
    with torch.no_grad():
        outputs = model(input_ids, output_attentions=True)
        attn = outputs.attentions[0][0, 0]  # First layer, first head

    # Check entropy of attention distribution
    # High entropy = spread out attention (good)
    # Low entropy = peaked attention (might be saturated)
    probs = attn.softmax(dim=-1)
    entropy = -(probs * torch.log(probs + 1e-9)).sum(dim=-1).mean()

    print(f"Attention entropy: {entropy:.4f}")
    print(f"Max attention weight: {probs.max():.4f}")
    print(f"Attention pattern sample:\n{probs[0, :8]}")

analyze_attention_distribution(model, sample_input)
# Output:
# Attention entropy: 0.0312  <-- Almost zero! Very peaked!
# Max attention weight: 0.9987  <-- One token gets all attention!
# Attention pattern sample:
# tensor([0.9987, 0.0003, 0.0002, 0.0002, 0.0002, 0.0001, 0.0001, 0.0002])
```

The attention was completely saturated. One token was getting 99.87% of the attention weight. The others were getting 0.01%. The softmax was in its extreme regime.

The Crime Scene: Missing √d_k

```
# THE BUGGY ATTENTION (Iteration #22)
```

```python
class BuggyAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.head_dim = embed_dim // num_heads
        # ... projections ...

    def forward(self, x):
        B, T, C = x.shape

        Q = self.W_q(x)  # [B, T, head_dim]
        K = self.W_k(x)  # [B, T, head_dim]
        V = self.W_v(x)  # [B, T, head_dim]

        # Compute attention scores
        scores = torch.matmul(Q, K.transpose(-2, -1))  # <-- MISSING SCALE!

        attn_weights = F.softmax(scores, dim=-1)
        output = torch.matmul(attn_weights, V)

        return output
```

When `head_dim = 64`, the dot product of two 64-dimensional vectors can easily be 50, 100, or larger. Apply softmax to `[150, 148, 145, 140]`:

```python
scores = torch.tensor([150.0, 148.0, 145.0, 140.0])
probs = F.softmax(scores, dim=-1)
print(probs)
# tensor([0.8668, 0.1173, 0.0142, 0.0006])
```

Even with just 10-point differences, softmax becomes extremely peaked. With 64-dimensional dot products producing scores in the hundreds, the softmax saturated completely. One token got all the attention. Gradients became tiny (derivative of softmax at saturation is near-zero). Learning stopped.

The Fix

The famous scale factor from "Attention Is All You Need":

```python
# THE FIXED ATTENTION (Iteration #23)
class FixedAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.head_dim = embed_dim // num_heads
        self.scale = math.sqrt(self.head_dim)  # √d_k
        # ... projections ...

    def forward(self, x):
        B, T, C = x.shape

        Q = self.W_q(x)
        K = self.W_k(x)
```

```
        V = self.W_v(x)

        # Scale the dot products!
        scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale  # <-- FIXED!

        attn_weights = F.softmax(scores, dim=-1)
        output = torch.matmul(attn_weights, V)

        return output
```

The scale factor $\sqrt{d_k}$ normalizes the dot products. If Q and K are random vectors with unit variance, QK^T will have variance d_k. Dividing by $\sqrt{d_k}$ gives variance 1, keeping the softmax in a healthy regime.

The Lesson

The $\sqrt{d_k}$ divisor isn't optional. It's not a minor optimization. It's essential for training. Without it, your attention saturates and gradients vanish.

When loss plateaus unexpectedly, check your intermediate values. Is anything saturating? Are gradients flowing through all paths?

The Broadcasting Surprise (Iteration #45)

The Scene of the Crime

Iteration #45. Adding residual connections. The architecture was getting complex. I added a new normalization layer and ran training:

```
# Training log, Iteration #45
# Step 1: Loss = 10.82
# Step 100: Loss = 10.52
# Step 1000: Loss = 10.51
# Step 10000: Loss = 10.50
# ...three days pass...
# Step 100000: Loss = 10.50
```

Loss started at 10.82 (random), dropped slightly to 10.5, and then… nothing. For three days. The model wasn't learning anything beyond the most basic token frequency statistics.

Detective Work

I added shape assertions everywhere. Every tensor, every operation:

```
def debug_forward(self, x):
    B, T, C = x.shape
    print(f"Input shape: {x.shape}")

    # Attention
    attn_out = self.attention(self.norm1(x))
    print(f"After attention: {attn_out.shape}")
```

```
    assert attn_out.shape == (B, T, C), f"Attention output wrong: {attn_out.shape}"

    # Residual
    x = x + attn_out
    print(f"After residual 1: {x.shape}")

    # Feed-forward
    ff_out = self.ff(self.norm2(x))
    print(f"After FF: {ff_out.shape}")
    assert ff_out.shape == (B, T, C), f"FF output wrong: {ff_out.shape}"

    x = x + ff_out
    print(f"After residual 2: {x.shape}")

    return x
```

The output:

```
Input shape: torch.Size([32, 64, 512])
After attention: torch.Size([32, 64, 512])
After residual 1: torch.Size([32, 64, 512])
After FF: torch.Size([32, 1, 512])  <-- WAIT WHAT
After residual 2: torch.Size([32, 64, 512])  # Broadcasting happened silently!
```

The feed-forward output was `[32, 1, 512]` instead of `[32, 64, 512]`. The sequence dimension had collapsed to 1. But the code didn't crash because PyTorch happily *broadcast* the `[32, 1, 512]` tensor to match `[32, 64, 512]` during the residual addition.

Every position in the sequence was getting the same feed-forward output. The model couldn't distinguish positions. It was effectively learning only aggregated features, not per-position features.

The Crime Scene: Accidental Averaging

```
# THE BUGGY NORMALIZATION (Iteration #45)
class BuggyRMSNorm(nn.Module):
    def __init__(self, dim, eps=1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        # Compute RMS
        rms = torch.sqrt(x.pow(2).mean(dim=-1, keepdim=True) + self.eps)

        # Normalize
        x_norm = x / rms

        # Wait, what did I copy from somewhere?
        x_norm = x_norm.mean(dim=1, keepdim=True)  # <-- THE BUG! Averaging over sequence!

        return x_norm * self.weight
```

That `.mean(dim=1, keepdim=True)` was supposed to be for something else—I had copied code from a different context and left in this line. It averaged over the sequence dimension, collapsing `[B, T, C]` to `[B, 1, C]`. But because of broadcasting, the model still "worked." It just couldn't learn anything useful.

The Fix

```
# THE FIXED NORMALIZATION (Iteration #46)
class FixedRMSNorm(nn.Module):
    def __init__(self, dim, eps=1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        # RMS normalization over the channel dimension only
        rms = torch.sqrt(x.pow(2).mean(dim=-1, keepdim=True) + self.eps)
        x_norm = x / rms

        # No sequence averaging!
        return x_norm * self.weight
```

And the defensive programming lesson:

```
def forward(self, x):
    B, T, C = x.shape

    # After every operation, verify shapes
    attn_out = self.attention(x)
    assert attn_out.shape == (B, T, C), f"Shape error: {attn_out.shape}"

    x = x + attn_out
    assert x.shape == (B, T, C), f"Shape error: {x.shape}"

    ff_out = self.ff(x)
    assert ff_out.shape == (B, T, C), f"Shape error: {ff_out.shape}"

    x = x + ff_out
    assert x.shape == (B, T, C), f"Shape error: {x.shape}"

    return x
```

The Lesson

PyTorch's broadcasting is powerful and dangerous. It will silently "fix" mismatched tensor shapes by repeating values along singleton dimensions. Your code won't crash. Your model won't learn.

Add shape assertions after every operation. `assert x.shape == (B, T, C)` should appear in every forward pass. It costs nothing at runtime (assertions are removed with `-O`) and catches silent dimension collapses.

The FP16 Overflow (Iteration #56)

The Scene of the Crime

Iteration #56. Time to speed things up with mixed precision training. I enabled `torch.cuda.amp`:

```
scaler = torch.cuda.amp.GradScaler()

for batch in dataloader:
    with torch.cuda.amp.autocast():
        logits = model(batch['input_ids'])
        loss = compute_loss(logits, batch['labels'])

    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
```

On CPU: worked perfectly. On GPU with FP32: worked perfectly. On GPU with FP16:

```
Step 1: Loss = 10.82
Step 2: Loss = nan
Step 3: Loss = nan
...
```

NaN from step 2 onwards.

Detective Work

NaN typically means one of three things: division by zero, log of zero, or overflow. I checked the gradients:

```
for name, param in model.named_parameters():
    if param.grad is not None:
        if torch.isnan(param.grad).any():
            print(f"NaN gradient in {name}")
        if torch.isinf(param.grad).any():
            print(f"Inf gradient in {name}")
```

Gradients in the attention layers were `inf` before becoming `nan`. Something was overflowing.

Then I remembered: FP16 has a maximum value of about 65,504. And my causal mask used `float('-inf')`:

```
# My mask values
print(float('-inf'))  # -inf

# In FP16
import numpy as np
print(np.finfo(np.float16).min)  # -65504.0
print(np.finfo(np.float16).max)  # 65504.0
```

`float('-inf')` in FP16 is... still infinity. But when you do math with infinity in FP16, things go wrong. The subtraction for softmax numerical stability (`x - max(x)`) produces `-inf - (-inf) = nan`.

The Crime Scene: Infinity in FP16

```
# THE BUGGY MASK (worked in FP32, exploded in FP16)
def create_causal_mask(seq_len, device, dtype):
    mask = torch.triu(torch.ones(seq_len, seq_len, device=device, dtype=dtype), diagonal=1)
    mask = mask.masked_fill(mask == 1, float('-inf'))  # <-- Actual infinity
    return mask

# In FP16 autocast context:
with torch.cuda.amp.autocast():
    scores = torch.matmul(Q, K.transpose(-2, -1)) / scale
    mask = create_causal_mask(T, device, scores.dtype)  # dtype is float16
    scores = scores + mask
    # scores now has -inf values in float16

    attn = F.softmax(scores, dim=-1)
    # Softmax does: exp(x - max(x))
    # When x contains -inf and max(x) is also -inf (entire row masked):
    # -inf - (-inf) = nan
```

The Fix

Use large negative numbers instead of actual infinity—large enough to be effectively zero after softmax, small enough to not overflow FP16:

```
# THE FIXED MASK (Iteration #57)
def create_causal_mask(seq_len, device, dtype=None):
    """
    Create causal mask with dtype-appropriate masking values.
    """
    mask = torch.triu(torch.ones(seq_len, seq_len, device=device), diagonal=1)

    # Use dtype-appropriate "negative infinity"
    if dtype == torch.float16:
        mask_value = -65000.0  # Close to FP16 min, but not overflow
    elif dtype == torch.bfloat16:
        mask_value = -1e9  # BF16 has larger range
    else:
        mask_value = -1e9  # FP32 can handle this easily

    mask = mask.masked_fill(mask == 1, mask_value)
    return mask

# Even better: check dtype at runtime
def get_mask_value(dtype):
    """Get appropriate mask value for dtype."""
    if dtype == torch.float16:
        return -1e4  # Conservative: 10,000 is safe, exp(-10000)  0
    elif dtype == torch.bfloat16:
        return -1e9
```

```
    else:
        return -1e9
```

An even cleaner approach—let the softmax handle the mask:

```
# Modern approach: use PyTorch's SDPA with mask
attn_output = F.scaled_dot_product_attention(
    Q, K, V,
    attn_mask=None,
    is_causal=True,  # PyTorch handles the mask correctly for all dtypes
    dropout_p=self.dropout if self.training else 0.0
)
```

The Lesson

FP16 has a max value of ~65,504. Infinity is not your friend in mixed precision training. Use large-but-finite negative numbers for attention masks: `-1e4` is safe and produces effectively zero attention after softmax ($e^{-10000} \approx 0$).

When debugging FP16 issues, first check if you're using any infinities or very large numbers. They're the usual suspects.

8.3 The "Explosion" Bugs (Training Instability)

These bugs announce themselves loudly. Your loss becomes NaN or Inf. Your gradients explode into the millions. Your model dies mid-training. They're dramatic, but at least they're obvious.

Exploding Gradients (Iteration #19)

The Scene of the Crime

2:47 AM. Fourteen hours of training. The model had been chugging along nicely:

```
# Training log, Iteration #19
# Step 840: Loss = 4.31 | Grad Norm = 1.23
# Step 841: Loss = 4.28 | Grad Norm = 1.18
# Step 842: Loss = 4.25 | Grad Norm = 1.34
# Step 843: Loss = 4.22 | Grad Norm = 1.89
# Step 844: Loss = 4.19 | Grad Norm = 3.47
# Step 845: Loss = 4.21 | Grad Norm = 12.83
# Step 846: Loss = 4.34 | Grad Norm = 847.56
# Step 847: Loss = nan  | Grad Norm = inf
```

One bad batch. One sequence that produced extreme gradients. Fourteen hours of work, gone.

Detective Work

The postmortem was straightforward: no gradient clipping. I trusted the gradients. They betrayed me.

In deep networks, gradients are computed via backpropagation—chain rule applied layer by layer. If gradients are slightly larger than 1 at each layer, they multiply:

$$1.1^{36} \approx 30$$

Thirty-six layers, each multiplying the gradient by 1.1, produces a 30x amplification. But it gets worse. A pathological batch can produce gradients of 2 or 3 at each layer:

$$2^{36} \approx 68 \text{ billion}$$

That's how you get gradient norms in the millions from a single bad batch.

The Fix: Gradient Clipping

```
# THE FIX: Always clip gradients
def training_step(model, batch, optimizer, max_grad_norm=1.0):
    optimizer.zero_grad()

    loss = compute_loss(model, batch)
    loss.backward()

    # Compute gradient norm before clipping (for logging)
    grad_norm = compute_gradient_norm(model)

    # Clip gradients to max_norm
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=max_grad_norm)

    # Now safe to step
    optimizer.step()

    return loss.item(), grad_norm

def compute_gradient_norm(model):
    """Compute the L2 norm of all gradients."""
    total_norm = 0.0
    for p in model.parameters():
        if p.grad is not None:
            total_norm += p.grad.data.norm(2).item() ** 2
    return total_norm ** 0.5
```

The Lesson

Always clip gradients. No exceptions. `max_norm=1.0` is the standard. If you think you don't need gradient clipping, you haven't trained long enough to hit a pathological batch.

Vanishing Gradients (Iteration #31)

The Scene of the Crime

Iteration #31. Scaling up to 18 layers. The model started training well:

```
# Training log, Iteration #31
# Step 1000: Loss = 6.42
# Step 2000: Loss = 5.89
# Step 3000: Loss = 5.67
# Step 4000: Loss = 5.54
# Step 5000: Loss = 5.48
# ...
# Step 20000: Loss = 5.43
# Step 50000: Loss = 5.41
# Step 100000: Loss = 5.40
```

Loss stopped decreasing after step 5000. It was stuck at 5.4, which is decent but not good. The model should have reached 3.5 or lower.

Detective Work

I checked gradient norms per layer:

```
def gradient_norm_per_layer(model):
    """Check if gradients are reaching early layers."""
    layer_norms = {}
    for name, param in model.named_parameters():
        if param.grad is not None:
            layer_norms[name] = param.grad.data.norm(2).item()

    # Group by layer depth
    for name, norm in sorted(layer_norms.items()):
        print(f"{name}: {norm:.2e}")

gradient_norm_per_layer(model)
# Output (abbreviated):
# layers.0.attention.qkv.weight: 1.23e-09
# layers.0.attention.proj.weight: 8.47e-10
# layers.1.attention.qkv.weight: 2.56e-09
# ...
# layers.16.attention.qkv.weight: 3.45e-02
# layers.17.attention.qkv.weight: 4.87e-02
```

The early layers (0, 1, 2) had gradient norms of 10^{-9} or smaller. The later layers (16, 17) had gradient norms of 10^{-2}. Gradients were vanishing as they propagated backward through the network.

With such tiny gradients, the early layers couldn't learn. Only the last few layers were being updated meaningfully.

The Fix: Gradient Checkpointing + Learning Rate Adjustment

```
# Solution 1: Gradient checkpointing
# Recomputes activations during backward pass instead of storing them
# This doesn't directly fix vanishing gradients, but allows using
# larger batch sizes and more stable training

class TransformerBlock(nn.Module):
```

```
    def __init__(self, config):
        super().__init__()
        self.attention = Attention(config)
        self.ff = FeedForward(config)
        self.norm1 = RMSNorm(config.hidden_size)
        self.norm2 = RMSNorm(config.hidden_size)
        self.use_checkpointing = config.use_gradient_checkpointing

    def forward(self, x):
        if self.use_checkpointing and self.training:
            # Gradient checkpointing trades compute for memory
            x = x + torch.utils.checkpoint.checkpoint(
                self.attention, self.norm1(x)
            )
            x = x + torch.utils.checkpoint.checkpoint(
                self.ff, self.norm2(x)
            )
        else:
            x = x + self.attention(self.norm1(x))
            x = x + self.ff(self.norm2(x))
        return x

# Solution 2: Pre-normalization (already standard in modern transformers)
# Ensures gradients flow through residual connections

# Solution 3: Higher learning rate for early layers (less common)
def create_optimizer_with_layer_lr(model, base_lr):
    """Different learning rates per layer depth."""
    param_groups = []
    num_layers = len(model.layers)

    for i, layer in enumerate(model.layers):
        # Earlier layers get higher LR to compensate for vanishing gradients
        layer_lr = base_lr * (1.5 ** (num_layers - i - 1) / (1.5 ** num_layers))
        param_groups.append({
            'params': layer.parameters(),
            'lr': layer_lr
        })

    return torch.optim.AdamW(param_groups)
```

The Lesson

In deep networks, check gradient norms at different layer depths. If early layers have gradient norms orders of magnitude smaller than later layers, you have vanishing gradients. Pre-normalization (norm before sublayer rather than after) and proper initialization help. Gradient checkpointing allows larger batches that can improve gradient stability.

The OOM Cascade (Iteration #63)

The Scene of the Crime

```
# Training on T4 GPU (16GB)
# Step 1: OK
# Step 47: OK
# Step 128: CUDA out of memory
# ...restart...
# Step 1: OK
# Step 203: CUDA out of memory
```

Out-of-memory errors at random batches. Not the same batch every time. The pattern seemed chaotic.

Detective Work

I monitored memory usage:

```
def print_gpu_memory():
    allocated = torch.cuda.memory_allocated() / 1e9
    reserved = torch.cuda.memory_reserved() / 1e9
    print(f"Allocated: {allocated:.2f} GB | Reserved: {reserved:.2f} GB")

# During training:
# Step 1: Allocated: 8.23 GB | Reserved: 8.50 GB
# Step 10: Allocated: 8.45 GB | Reserved: 9.12 GB
# Step 50: Allocated: 8.89 GB | Reserved: 11.34 GB
# Step 100: Allocated: 9.12 GB | Reserved: 14.56 GB
# Step 128: CUDA out of memory
```

Memory was growing step-by-step. Something wasn't being freed.

The culprit: memory fragmentation from gradient accumulation. When you do gradient accumulation, you call `loss.backward()` multiple times before `optimizer.step()`. Each backward pass creates intermediate tensors. PyTorch's memory allocator tries to reuse memory, but when tensors have different sizes, fragmentation occurs.

The Crime Scene

```
# Gradient accumulation loop (THE BUGGY VERSION)
accumulation_steps = 8
for step, batch in enumerate(dataloader):
    loss = compute_loss(model, batch)
    loss = loss / accumulation_steps
    loss.backward()  # Gradients accumulate

    if (step + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
        # Memory never explicitly cleared between accumulation steps
```

The Fix

```python
# Solution 1: Set PYTORCH_CUDA_ALLOC_CONF
import os
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'
# This tells PyTorch to use a different memory allocation strategy
# that's better at handling fragmentation

# Solution 2: Explicit cache clearing
def clear_gpu_memory():
    torch.cuda.empty_cache()
    gc.collect()

# Solution 3: Fixed gradient accumulation loop
accumulation_steps = 8
for step, batch in enumerate(dataloader):
    loss = compute_loss(model, batch)
    loss = loss / accumulation_steps
    loss.backward()

    if (step + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()

        # Clear cache periodically
        if (step + 1) % (accumulation_steps * 10) == 0:
            clear_gpu_memory()

# Solution 4: Use GradScaler properly
scaler = torch.cuda.amp.GradScaler()

for step, batch in enumerate(dataloader):
    with torch.cuda.amp.autocast():
        loss = compute_loss(model, batch)
        loss = loss / accumulation_steps

    scaler.scale(loss).backward()

    if (step + 1) % accumulation_steps == 0:
        scaler.step(optimizer)
        scaler.update()
        optimizer.zero_grad(set_to_none=True)  # set_to_none=True saves memory

        if (step + 1) % 1000 == 0:
            clear_gpu_memory()
```

The Lesson

GPU memory management is a dark art. `PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True` helps with fragmentation. Call `torch.cuda.empty_cache()` periodically during long training runs. Use `optimizer.zero_grad(set_to_none=True)` to actually free gradient memory instead of zeroing

it.

8.4 Implementation Gotchas

These are the paper cuts—small mistakes that don't quite rise to the level of "major bug" but still waste hours of debugging. They're the "of course it works that way, why didn't I remember?" moments.

view() vs transpose() (Iteration #27)

The Bug

```
# Multi-head attention: reshape for parallel head processing
# Input: [B, T, C]
# Need: [B, num_heads, T, head_dim]

x = x.view(B, T, num_heads, head_dim)  # Split channels into heads
x = x.transpose(1, 2)  # Move heads before sequence: [B, num_heads, T, head_dim]

# ... do attention ...

# Reshape back
x = x.transpose(1, 2)  # [B, T, num_heads, head_dim]
x = x.view(B, T, C)  # <-- CRASH or GARBAGE!
```

The `.view()` on a non-contiguous tensor either crashes or silently scrambles your data. After `.transpose()`, the tensor isn't contiguous in memory—the bytes are arranged for the old shape.

The Fix

```
# ALWAYS call .contiguous() before .view() after any transpose/permute
x = x.transpose(1, 2).contiguous().view(B, T, C)

# Or use reshape() which handles non-contiguous tensors (but copies)
x = x.transpose(1, 2).reshape(B, T, C)
```

The Lesson

After any `transpose()`, `permute()`, or index operation, call `.contiguous()` before `.view()`. Or just use `.reshape()` which does this automatically.

Forgot optimizer.zero_grad() (Iteration #4)

The Bug

```
# THE BUGGY TRAINING LOOP
for batch in dataloader:
    loss = compute_loss(model, batch)
    loss.backward()
```

```
    optimizer.step()
    # Where's optimizer.zero_grad()???
```

PyTorch accumulates gradients by default. Without `zero_grad()`, each batch's gradients ADD to the previous batch's gradients. Loss oscillates wildly and eventually explodes.

The Fix

```
# THE FIXED TRAINING LOOP
for batch in dataloader:
    optimizer.zero_grad()  # Clear gradients from previous batch
    loss = compute_loss(model, batch)
    loss.backward()
    optimizer.step()
```

The Lesson

This bug still gets people. When loss oscillates wildly or explodes for no apparent reason, check for `zero_grad()` first. It's always the first thing I check.

Train vs Eval Mode (Iteration #38)

The Bug

```
# After training, test generation
prompt = "The capital of France is"
output1 = model.generate(prompt)
output2 = model.generate(prompt)
output3 = model.generate(prompt)

print(output1)  # "The capital of France is Paris."
print(output2)  # "The capital of France is Berlin."  <-- Wait, what?
print(output3)  # "The capital of France is a city."  <-- ???
```

Same prompt, different outputs every time. The model seemed schizophrenic.

Detective Work

Dropout was still active during generation. In training mode, dropout randomly zeros out some neurons. Each forward pass zeros different neurons, producing different outputs.

The Fix

```
# Always set mode explicitly
model.train()  # Before training: enables dropout
# ... training loop ...

model.eval()  # Before inference: disables dropout
with torch.no_grad():  # Also disable gradient computation for efficiency
    output = model.generate(prompt)
```

The Lesson

`model.eval()` before generation. `model.train()` before training. Every time. Make it a habit.

The Weight Tying Bug (Iteration #52)

The Bug

Weight tying shares parameters between the input embedding and the output projection. The intuition: the embedding that maps tokens to vectors should be related to the projection that maps vectors back to token probabilities.

```
# THE BUGGY IMPLEMENTATION
class BuggyLanguageModel(nn.Module):
    def __init__(self, vocab_size, hidden_size):
        super().__init__()
        self.tok_embedding = nn.Embedding(vocab_size, hidden_size)
        self.lm_head = nn.Linear(hidden_size, vocab_size, bias=False)

        # "Weight tying" - or so I thought
        self.lm_head.weight = self.tok_embedding.weight.clone()  # <-- BUG!
```

The embedding weights weren't updating during training. Only the lm_head weights were being trained.

Detective Work

`.clone()` creates a *copy* of the tensor. The two weights started identical but diverged during training. Worse, since `tok_embedding` wasn't connected to the loss (only `lm_head` was), its gradients were zero.

The Fix

```
# THE FIXED IMPLEMENTATION
class FixedLanguageModel(nn.Module):
    def __init__(self, vocab_size, hidden_size):
        super().__init__()
        self.tok_embedding = nn.Embedding(vocab_size, hidden_size)
        self.lm_head = nn.Linear(hidden_size, vocab_size, bias=False)

        # ACTUAL weight tying - share the same weight tensor
        self.lm_head.weight = self.tok_embedding.weight  # No .clone()!
```

Now both `self.lm_head.weight` and `self.tok_embedding.weight` point to the *same* tensor. Gradients flow to both, and updates affect both.

The Lesson

Weight tying requires sharing the actual tensor reference, not copying values. `a = b.clone()` creates a copy. `a = b` creates a reference.

The Final Lesson

Every bug in this chapter felt, at the time, like the end of the world. Loss stuck for days. Training runs wasted. Models that looked perfect but generated garbage.

But here's the truth: these bugs are rites of passage. Every transformer implementer hits them. The papers don't warn you because the authors hit them too—they just don't include them in the "Methods" section.

The debugger's toolkit: 1. **Print shapes obsessively.** Every tensor, every operation. Shape bugs are silent killers. 2. **Visualize attention patterns.** If they look wrong, something *is* wrong. 3. **Log gradient norms.** Exploding? Vanishing? Stuck at zero? The gradients always know before you do. 4. **If loss looks too good, be suspicious.** The model is probably cheating. 5. **Add assertions everywhere.** They're free documentation and free bug detection.

And remember: the graveyard of errors is where good engineers are made. Every bug you fix is a bug you'll never make again.

"In the end, debugging is detective work. The crime has already been committed—somewhere in your code, something is wrong. Your job is to find it before the training run dies. The clues are in the shapes, the gradients, the loss curves. Pay attention. The code is always trying to tell you something."

Iteration Count: 100+ Bugs Documented: 12 Days Lost to Debugging: Too many to count Lessons Learned: Priceless # Case File #9: The GPU Optimization Journey

"In this business, you learn that hardware is destiny. The same model that crawls on a T4—dying in a swamp of OOM errors and gradient accumulation hacks—flies on an H200 like it was born to run. I've climbed every rung of the GPU ladder, from the free tier trenches to the Hopper promised land. Each step up brought new possibilities, new techniques, and—let's be honest—new credit card bills. This is the story of that ascent."

The GPU you train on shapes everything. Your batch size. Your sequence length. Your iteration speed. Your sanity.

When I started this journey, I had a simple question: *Can I build a language model from scratch?* The answer depended entirely on what silicon I could get my hands on. This chapter documents the hardware odyssey—from scraping by on free Colab T4s to unleashing the full power of NVIDIA's Hopper architecture.

Each GPU tier taught me different lessons. Each required different optimizations. And each changed what was possible.

9.1 T4 (16GB) — Where It All Began

The NVIDIA T4 is the people's GPU. It's what Google gives you for free on Colab. It's what Kaggle hands out like candy. It's what AWS Sagemaker falls back to when you're on a budget.

It's also where dreams go to get a reality check.

The Scene

Sixteen gigabytes of GDDR6 memory. Turing architecture. 65 TFLOPS of FP16 performance. On paper, it sounds workable. In practice, every training run became a memory management puzzle.

My first attempts to train on T4 were disasters. I'd launch a 200M parameter model, watch the loss tick down for 500 batches, then see the dreaded message:

```
RuntimeError: CUDA out of memory. Tried to allocate 2.14 GB
```

The T4 was teaching me its first lesson: *You cannot brute-force your way through GPU training.*

The Constraints

Metric	T4 Limit	Typical Value
VRAM	16 GB	10-12 GB usable
Batch Size	4-8	4 (safe), 8 (risky)
Sequence Length	256-512	512 max
Model Size	100M-200M	125M sweet spot
Training Time	Slow	~4 hrs for 10K steps

The Optimizations

Training on T4 isn't just about writing a training loop. It's about squeezing every byte of efficiency out of hardware that wasn't designed for this workload.

Gradient Checkpointing (Essential — Saves 40% Memory)

The first optimization that made T4 viable. Instead of storing all intermediate activations for backpropagation, gradient checkpointing recomputes them during the backward pass. You trade compute for memory.

```
# T4-optimized model with gradient checkpointing
import torch
from torch.utils.checkpoint import checkpoint_sequential

class T4OptimizedTransformer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.blocks = nn.ModuleList([
            TransformerBlock(config)
            for _ in range(config.n_layers)
        ])

    def forward(self, x):
        # Use gradient checkpointing for memory efficiency
        # This recomputes activations during backward pass
        # Trading ~30% more compute for ~40% less memory
        if self.training and self.config.use_gradient_checkpointing:
```

```
            # Checkpoint every 2 blocks
            for i in range(0, len(self.blocks), 2):
                chunk = self.blocks[i:i+2]
                x = checkpoint_sequential(chunk, 1, x, use_reentrant=False)
        else:
            for block in self.blocks:
                x = block(x)
        return x
```

The memory savings were dramatic. A 200M model that OOMed at batch size 4 suddenly ran at batch size 8.

Mixed Precision FP16 (2x Speedup)

FP16 is not optional on T4. It's survival.

```
# T4 Mixed Precision Training Setup
from torch.cuda.amp import GradScaler, autocast

def setup_t4_training():
    # T4-specific CUDA settings
    os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True,max_split_size_mb:128'

    # Enable TF32 for Turing (limited benefit on T4, but doesn't hurt)
    torch.backends.cuda.matmul.allow_tf32 = True
    torch.backends.cudnn.allow_tf32 = True
    torch.backends.cudnn.benchmark = True

    # Initialize the scaler for FP16
    scaler = GradScaler()

    return scaler

def train_step_t4(model, batch, optimizer, scaler, config):
    """T4-optimized training step with FP16."""
    input_ids, labels = batch
    input_ids = input_ids.cuda()
    labels = labels.cuda()

    # Forward pass in FP16
    with autocast(device_type='cuda', dtype=torch.float16):
        logits, loss = model(input_ids, labels=labels)

    # Scale loss for FP16 stability
    scaled_loss = scaler.scale(loss)
    scaled_loss.backward()

    # Gradient clipping (essential for stability)
    scaler.unscale_(optimizer)
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

```
    # Optimizer step with scaler
    scaler.step(optimizer)
    scaler.update()
    optimizer.zero_grad(set_to_none=True)  # set_to_none saves a tiny bit of memory

    return loss.item()
```

But FP16 on T4 has a gotcha I learned the hard way: the attention mask.

```
# THE BUG (Iteration #56)
# This causes NaN because float('-inf') overflows in FP16
mask = torch.triu(torch.ones(T, T), diagonal=1).masked_fill(True, float('-inf'))

# THE FIX
# Use a smaller negative value that fits in FP16's range
def get_causal_mask(seq_len, dtype, device):
    # FP16 max is ~65504, so we use -1e4 instead of -inf
    mask_value = -1e4 if dtype == torch.float16 else -1e9
    mask = torch.triu(
        torch.full((seq_len, seq_len), mask_value, dtype=dtype, device=device),
        diagonal=1
    )
    return mask
```

Gradient Accumulation (8-16 Steps)

When batch size 4 is all you can afford, you fake larger batches with gradient accumulation:

```
# T4 Configuration
T4_CONFIG = {
    'batch_size': 4,                        # Physical batch size per step
    'gradient_accumulation_steps': 8,       # Accumulate 8 steps
    'effective_batch_size': 32,             # 4 x 8 = 32 effective
    'max_seq_len': 512,
    'learning_rate': 6e-4,
    'warmup_steps': 500,
}

def train_with_accumulation(model, dataloader, optimizer, scaler, config):
    """Training loop with gradient accumulation for T4."""
    accumulation_steps = config['gradient_accumulation_steps']

    model.train()
    accumulated_loss = 0

    for step, batch in enumerate(dataloader):
        # Forward and backward (gradients accumulate)
        with autocast(device_type='cuda', dtype=torch.float16):
            _, loss = model(batch['input_ids'].cuda(), labels=batch['labels'].cuda())
```

```
        # Scale loss by accumulation steps
        loss = loss / accumulation_steps
        scaler.scale(loss).backward()
        accumulated_loss += loss.item()

        # Only update weights every accumulation_steps
        if (step + 1) % accumulation_steps == 0:
            scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            scaler.step(optimizer)
            scaler.update()
            optimizer.zero_grad(set_to_none=True)

            # Clear cache periodically to prevent fragmentation
            if (step + 1) % (accumulation_steps * 100) == 0:
                torch.cuda.empty_cache()

            accumulated_loss = 0
```

The Complete T4 Training Setup

```
@dataclass
class T4ModelConfig:
    """
    T4-Optimized Configuration (~125M parameters)

    Memory budget:
    - Model weights: ~250MB (FP16)
    - Optimizer states: ~1GB
    - Activations: ~2-4GB (with checkpointing)
    - Gradients: ~250MB
    - Buffer: ~2GB for safety
    Total: ~6-8GB, leaving headroom in 16GB T4
    """
    vocab_size: int = 50257          # GPT-2 vocabulary
    n_embd: int = 768                # Hidden dimension
    n_heads: int = 12                # Attention heads
    n_layers: int = 12               # Transformer blocks
    max_seq_len: int = 512           # Context window
    dropout: float = 0.1
    use_gradient_checkpointing: bool = True

    # Training hyperparameters
    batch_size: int = 4
    gradient_accumulation_steps: int = 8
    learning_rate: float = 6e-4
    weight_decay: float = 0.1
```

```
    warmup_steps: int = 500
    max_grad_norm: float = 1.0
```

The Lesson

T4 is the perfect "can this even work?" GPU.

It forces discipline. It punishes sloppiness. If your training runs on T4, it'll run anywhere. If it OOMs on T4, you need to optimize before scaling up.

I spent months on T4. Those months taught me more about memory efficiency than any documentation ever could.

9.2 L40S (48GB) — The Sweet Spot

The day I moved from T4 to L40S felt like upgrading from a bicycle to a motorcycle. Same journey, different experience entirely.

The Upgrade

L40S. Forty-eight gigabytes of GDDR6 memory. Ada Lovelace architecture. 362 TFLOPS of FP16 performance. Three times the memory, roughly five times the compute throughput.

The first training run on L40S was revelatory. The batch size that barely fit on T4? I could run it eight times over. The sequence length I'd been artificially limiting? Quadrupled it.

```
# The moment I realized everything had changed
# On T4: OOM at batch_size=8, seq_len=512
# On L40S: Running smoothly at batch_size=16, seq_len=2048

L40S_CONFIG = {
    'batch_size': 16,                    # 4x T4's batch size
    'gradient_accumulation_steps': 8,    # Still accumulating for stability
    'effective_batch_size': 128,         # 16 x 8 = 128 effective
    'max_seq_len': 2048,                 # 4x T4's sequence length
    'learning_rate': 1e-4,
    'buffer_size': 100000,               # Larger dataset buffers
}
```

The New Possibilities

Metric	T4	L40S	Improvement
VRAM	16 GB	48 GB	3x
Batch Size	4-8	8-16	2-4x
Sequence Length	512	2048	4x
Model Size	100M-200M	200M-1.3B	6x
Training Speed	Baseline	~3x faster	3x

The Optimizations

Flash Attention Enabled

L40S with Ada Lovelace fully supports Flash Attention through PyTorch's scaled dot-product attention:

```
# L40S Flash Attention Setup
def setup_l40s_training():
    """Configure L40S for maximum performance."""
    # CUDA optimization settings
    os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True,max_split_size_mb:256'
    os.environ['TOKENIZERS_PARALLELISM'] = 'false'

    # Enable TF32 for Ada Lovelace (significant speedup)
    torch.backends.cuda.matmul.allow_tf32 = True
    torch.backends.cudnn.allow_tf32 = True
    torch.backends.cudnn.benchmark = True
    torch.backends.cudnn.deterministic = False

    # Enable Flash/Memory-Efficient Attention
    torch.backends.cuda.enable_flash_sdp(True)
    torch.backends.cuda.enable_mem_efficient_sdp(True)
    torch.backends.cuda.enable_math_sdp(False)  # Disable slow fallback

    print("L40S Optimizations:")
    print("  - TF32: ENABLED")
    print("  - Flash Attention: ENABLED")
    print("  - Memory-Efficient SDP: ENABLED")
```

TF32 for Matrix Operations

TF32 is an Ada Lovelace/Ampere feature that uses 19-bit precision for matrix multiplications while maintaining FP32 range. It's essentially free performance:

```
# TF32 is automatic with these settings
# No code changes needed-just enable the backend flags
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True

# Result: ~2x speedup on matrix operations with negligible accuracy loss
```

L40S-Specific Architecture Optimizations

With more memory, I could use larger models and deeper architectures:

```
@dataclass
class L40SModelConfig:
    """
    L40S Configuration for 1.3B parameter model

    Memory budget on 48GB:
```

```
    - Model weights: ~2.6GB (FP16)
    - Optimizer states: ~10.4GB (AdamW, 2 states per param)
    - Activations: ~15GB (with longer sequences)
    - Gradients: ~2.6GB
    - Buffer: ~17GB headroom
    """
    vocab_size: int = 100277          # tiktoken cl100k_base
    hidden_size: int = 2048           # Larger hidden dimension
    n_heads: int = 16                 # More attention heads
    n_layers: int = 24                # Deeper network
    max_seq_len: int = 2048           # Longer context
    intermediate_size: int = 8192    # 4x hidden for FFN
    dropout: float = 0.0              # No dropout for pretraining
    use_rope: bool = True             # Rotary embeddings
    use_swiglu: bool = True           # SwiGLU activation
    use_flash_attention: bool = True

    # Training settings
    batch_size: int = 8
    gradient_accumulation_steps: int = 16
    effective_batch_size: int = 128
    learning_rate: float = 1e-4
    warmup_steps: int = 2000
```

The Training Time Revelation

The upgrade experience was quantifiable: training time dropped by 60%.

A run that took 12 hours on T4 completed in under 5 hours on L40S. And it wasn't just faster—it was *better*. Larger batch sizes meant more stable gradients. Longer sequences meant the model could learn longer-range dependencies.

```
# Comparison: Same 100M model, same 10K steps
# T4:   ~4 hours, batch=4, seq=512,  effective_batch=32
# L40S: ~1.5 hours, batch=16, seq=1024, effective_batch=128

# The math:
# T4:   32 * 512 = 16,384 tokens per step
# L40S: 128 * 1024 = 131,072 tokens per step
# That's 8x more tokens processed per optimizer step!
```

The Lesson

L40S is where small models become viable products.

On T4, you're always fighting memory constraints. On L40S, you can focus on what matters: data quality, architecture choices, training dynamics. The GPU gets out of your way.

If you're serious about training models but not ready for A100 pricing, L40S is the sweet spot.

9.3 A100 (40GB-80GB) — Serious Training

The A100 is the workhorse of modern AI. It's what labs use. It's what papers are trained on. It's what costs you $2-4 per hour on cloud providers.

The Professional Tier

When I first got access to an A100, I felt like I had been let into a secret club. Everything was faster. Everything was bigger. Everything was *more expensive*.

```
# A100 80GB Detection
def detect_a100():
    if torch.cuda.is_available():
        gpu_name = torch.cuda.get_device_name(0)
        memory = torch.cuda.get_device_properties(0).total_memory / 1e9

        if 'A100' in gpu_name:
            tier = 'A100_80GB' if memory > 70 else 'A100_40GB'
            print(f"A100 detected: {tier}")
            print(f"Memory: {memory:.1f} GB")
            print(f"This changes everything.")
            return tier
    return None
```

The Numbers

Metric	L40S	A100 40GB	A100 80GB
Memory	48 GB	40 GB	80 GB
Bandwidth	864 GB/s	1.55 TB/s	2.0 TB/s
FP16 TFLOPs	362	312	312
Batch Size	8-16	16-32	32-64
Sequence Length	2048	2048	2048+
Cost/Hour	~$1.50	~$2.00	~$3.50

The Optimizations

Multi-GPU Awareness

A100s often come in pairs or pods. Distributed training becomes relevant:

```
# A100 Multi-GPU Setup
def setup_a100_distributed():
    """Configure for multi-GPU training if available."""
    num_gpus = torch.cuda.device_count()

    if num_gpus > 1:
        print(f"Multi-GPU detected: {num_gpus} A100s")
        # For simple data parallelism
        # model = nn.DataParallel(model)
```

```
        # For distributed training (recommended)
        # torch.distributed.init_process_group(backend='nccl')
        # model = nn.parallel.DistributedDataParallel(model)

    return num_gpus
```

Larger Buffer Sizes

With more memory bandwidth (2.0 TB/s on A100 80GB), you can process data faster:

```
# A100 Configuration
A100_CONFIG = {
    'phase_b': {  # Pretraining
        'max_seq_len': 2048,
        'batch_size': 48,
        'gradient_accumulation_steps': 2,
        'effective_batch_size': 96,
        'learning_rate': 2e-4,
        'warmup_steps': 2000,
        'weight_decay': 0.1,
        'buffer_size': 100000,  # Large streaming buffer
    },
    'phase_c': {  # SFT
        'max_seq_len': 2048,
        'batch_size': 24,
        'gradient_accumulation_steps': 1,
        'learning_rate': 5e-6,
        'warmup_steps': 500,
        'buffer_size': 50000,
    },
    'phase_d': {  # Domain fine-tuning
        'max_seq_len': 2048,
        'batch_size': 24,
        'gradient_accumulation_steps': 1,
        'learning_rate': 1e-6,
        'warmup_steps': 200,
        'buffer_size': 20000,
    },
}
```

Optimized DataLoader Workers

A100's bandwidth means data loading can become the bottleneck:

```
# A100 DataLoader Configuration
def get_a100_dataloader(dataset, config):
    """Optimized dataloader for A100 bandwidth."""
    return DataLoader(
        dataset,
        batch_size=config['batch_size'],
```

```
        num_workers=8,              # More workers for A100
        pin_memory=True,            # Faster GPU transfer
        prefetch_factor=4,          # Prefetch more batches
        persistent_workers=True,    # Don't respawn workers
        drop_last=True,
    )
```

The Cost-Benefit Analysis

A100s cost $2-4 per hour. When do you use them?

Use A100 when: - Iteration speed matters more than cost - You're training models >500M parameters - You need consistent, reliable performance - You're running experiments that need to complete in hours, not days

Don't use A100 when: - You're still debugging your training loop - You're experimenting with architectures - Cost is a primary constraint - The model fits comfortably on L40S

```
# My A100 decision framework
def should_use_a100(model_params, experiment_type, budget_per_month):
    if model_params > 500_000_000:  # >500M params
        return True
    if experiment_type == 'production_training':
        return True
    if budget_per_month > 500:  # >$500/month budget
        return True
    return False  # Stick with L40S
```

The Lesson

A100 is where iteration speed makes or breaks you.

When you're running hundreds of experiments, the difference between 4 hours and 1 hour compounds. A100 doesn't just make training faster—it makes *learning* faster. You can try more ideas, fail faster, and converge on what works.

But it's not free. Every hour on A100 is money spent. Use it wisely.

9.4 H200 (80GB-141GB) — The Dream Machine

And then there's the H200.

NVIDIA's Hopper architecture with HBM3 memory. 80GB or 141GB of VRAM. 3.35 TB/s memory bandwidth on the 80GB model, 4.8 TB/s on the 141GB. It's not just an incremental upgrade—it's a paradigm shift.

The Ultimate Hardware

When I first ran a training job on H200, I had to check the logs twice. The numbers didn't seem real.

```python
# H200 Detection
def detect_h200():
    if torch.cuda.is_available():
        gpu_name = torch.cuda.get_device_name(0)
        memory = torch.cuda.get_device_properties(0).total_memory / 1e9
        major, minor = torch.cuda.get_device_capability(0)

        if 'H200' in gpu_name or (major >= 9 and memory > 70):
            tier = 'H200_141GB' if memory > 120 else 'H200_80GB'
            print(f"{'='*60}")
            print(f"H200 DETECTED: {tier}")
            print(f"Memory: {memory:.1f} GB")
            print(f"Compute Capability: {major}.{minor} (Hopper)")
            print(f"Welcome to the future.")
            print(f"{'='*60}")
            return tier, memory
    return None, 0
```

The H200 Advantage

Metric	A100 80GB	H200 80GB	H200 141GB
Memory	80 GB HBM2e	80 GB HBM3	141 GB HBM3e
Bandwidth	2.0 TB/s	3.35 TB/s	4.8 TB/s
FP16 TFLOPs	312	989	989
BF16 TFLOPs	312	989	989
FP8 TFLOPs	N/A	1979	1979
Batch Size	32-64	64-128	96-192
Sequence Length	2048	4096	8192

That's not a typo. The H200 has 3x the TFLOPS and nearly 70% more memory bandwidth than A100.

The Optimizations

BF16 Instead of FP16 (Hopper Native)

The H200's Hopper architecture is optimized for BF16 (bfloat16), not FP16. BF16 has the same exponent range as FP32, which means no more overflow issues with attention masks:

```python
# H200 BF16 Training Setup
def setup_h200_training():
    """Configure H200 for maximum Hopper performance."""
    # Hopper-specific CUDA settings
    os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True,max_split_size_mb:512'
    os.environ['CUDA_DEVICE_MAX_CONNECTIONS'] = '1'

    # Enable all Hopper optimizations
    torch.backends.cuda.matmul.allow_tf32 = True
```

```
    torch.backends.cudnn.allow_tf32 = True
    torch.backends.cudnn.benchmark = True
    torch.backends.cudnn.deterministic = False

    # Enable Flash Attention 2 (native on Hopper)
    torch.backends.cuda.enable_flash_sdp(True)
    torch.backends.cuda.enable_mem_efficient_sdp(True)
    torch.backends.cuda.enable_math_sdp(False)

    print("H200 Hopper Optimizations:")
    print("  - BF16 Precision: ENABLED (native)")
    print("  - Flash Attention 2: ENABLED")
    print("  - Memory Bandwidth: 3.35+ TB/s")
    print("  - Ready for maximum performance")

def train_step_h200(model, batch, optimizer, config):
    """H200-optimized training step with BF16."""
    input_ids = batch['input_ids'].cuda()
    labels = batch['labels'].cuda()

    # BF16 autocast for Hopper
    with autocast(device_type='cuda', dtype=torch.bfloat16):
        logits, loss = model(input_ids, labels=labels)

    # No scaler needed for BF16 (it has FP32 range)
    loss.backward()

    torch.nn.utils.clip_grad_norm_(model.parameters(), config['max_grad_norm'])
    optimizer.step()
    optimizer.zero_grad(set_to_none=True)

    return loss.item()
```

Notice what's missing: no `GradScaler`. BF16 has the same dynamic range as FP32, so you don't need loss scaling. One less thing to worry about.

Flash Attention 2 Native Support

H200 doesn't just *support* Flash Attention 2—it was designed for it:

```
# Flash Attention is automatic with PyTorch SDPA on Hopper
# The attention implementation just works faster

class HopperOptimizedAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.n_heads = config.n_heads
        self.head_dim = config.hidden_size // config.n_heads

        self.q_proj = nn.Linear(config.hidden_size, config.hidden_size, bias=False)
```

```
        self.k_proj = nn.Linear(config.hidden_size, config.hidden_size, bias=False)
        self.v_proj = nn.Linear(config.hidden_size, config.hidden_size, bias=False)
        self.o_proj = nn.Linear(config.hidden_size, config.hidden_size, bias=False)

    def forward(self, x, freqs_cis=None):
        B, T, C = x.shape

        q = self.q_proj(x).view(B, T, self.n_heads, self.head_dim).transpose(1, 2)
        k = self.k_proj(x).view(B, T, self.n_heads, self.head_dim).transpose(1, 2)
        v = self.v_proj(x).view(B, T, self.n_heads, self.head_dim).transpose(1, 2)

        # Apply RoPE if provided
        if freqs_cis is not None:
            q, k = apply_rotary_pos_emb(q, k, freqs_cis)

        # This automatically uses Flash Attention 2 on Hopper
        # Memory usage: O(N) instead of O(N²)
        # Speed: ~3x faster than standard attention
        y = F.scaled_dot_product_attention(
            q, k, v,
            attn_mask=None,
            dropout_p=0.0,
            is_causal=True  # Automatic causal masking
        )

        y = y.transpose(1, 2).contiguous().view(B, T, C)
        return self.o_proj(y)
```

Fewer Shards, Larger Data Chunks

With 3.35 TB/s bandwidth, the H200 can swallow data faster than you can feed it. The bottleneck shifts from GPU to data pipeline:

```
# H200 Phase Configurations
H200_CONFIG = {
    'phase_b': {  # Pretraining
        'max_seq_len': 4096,           # Longer context
        'batch_size': 64,               # Massive batches
        'gradient_accumulation_steps': 2,
        'effective_batch_size': 128,
        'learning_rate': 2e-4,
        'warmup_steps': 3000,
        'weight_decay': 0.1,
        'buffer_size': 200000,          # Huge streaming buffer
    },
    'phase_c': {  # SFT
        'max_seq_len': 2048,
        'batch_size': 48,
        'gradient_accumulation_steps': 1,
```

```
        'learning_rate': 5e-6,
        'warmup_steps': 700,
        'buffer_size': 100000,
    },
    'phase_d': {  # Domain fine-tuning
        'max_seq_len': 2048,
        'batch_size': 48,
        'gradient_accumulation_steps': 1,
        'learning_rate': 1e-6,
        'warmup_steps': 300,
        'buffer_size': 50000,
    },
}

# H200 141GB can go even larger
H200_141GB_CONFIG = {
    'phase_b': {
        'max_seq_len': 8192,           # Maximum context
        'batch_size': 96,               # Even larger batches
        'gradient_accumulation_steps': 1,
        'effective_batch_size': 96,
        'learning_rate': 2e-4,
        'warmup_steps': 4000,
        'buffer_size': 300000,
    },
}
```

The Complete H200 Training Configuration

```
@dataclass
class H200ModelConfig:
    """
    H200 Configuration for maximum performance

    This configuration leverages:
    - 80-141GB HBM3 memory
    - 3.35-4.8 TB/s bandwidth
    - 989 TFLOPs BF16
    - Native Flash Attention 2
    """
    # Model architecture (1.3B+ parameters)
    vocab_size: int = 100277         # tiktoken cl100k_base
    hidden_size: int = 2048
    n_heads: int = 16
    n_layers: int = 24
    max_seq_len: int = 4096          # Long context
    intermediate_size: int = 8192
    dropout: float = 0.0
```

```
    # H200-specific settings
    use_bf16: bool = True            # BF16 native on Hopper
    use_flash_attention: bool = True
    use_rope: bool = True
    use_swiglu: bool = True

    # Training settings (massive throughput)
    batch_size: int = 64
    gradient_accumulation_steps: int = 2
    effective_batch_size: int = 128
    learning_rate: float = 2e-4
    warmup_steps: int = 3000
    weight_decay: float = 0.1
    max_grad_norm: float = 1.0

    # Data loading (saturate bandwidth)
    buffer_size: int = 200000
    num_workers: int = 16
    prefetch_factor: int = 8

def create_h200_training_loop(model, train_dataset, config):
    """Full H200-optimized training setup."""

    # DataLoader optimized for H200 bandwidth
    train_loader = DataLoader(
        train_dataset,
        batch_size=config.batch_size,
        num_workers=config.num_workers,
        pin_memory=True,
        prefetch_factor=config.prefetch_factor,
        persistent_workers=True,
        drop_last=True,
    )

    # AdamW with fused operations
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=config.learning_rate,
        weight_decay=config.weight_decay,
        betas=(0.9, 0.95),
        fused=True  # Fused AdamW for Hopper
    )

    # Learning rate scheduler
    scheduler = get_cosine_schedule_with_warmup(
        optimizer,
        num_warmup_steps=config.warmup_steps,
```

```
        num_training_steps=len(train_loader) * num_epochs
    )

    return train_loader, optimizer, scheduler
```

The Lesson

H200 changes what's possible. Budget permitting.

On H200, a 1.3B model trains like a 200M model did on T4. Experiments that took days complete in hours. The constraint shifts from hardware to ideas.

But H200 time costs $8-12 per hour. Every minute counts. You don't debug on H200. You don't experiment on H200. You *train* on H200—after you've worked out all the kinks on cheaper hardware.

GPU Comparison Table

The complete picture, for reference:

GPU	Memory	Bandwidth	FP16/BF16 TFLOPs	Batch Size	Seq Length	Model Size	Cost/Hour	100M Model Time
T4	16 GB GDDR6	300 GB/s	65	4-8	256-512	100M-200M	Free-$0.50	~4 hours
L4	24 GB GDDR6	300 GB/s	120	8-16	512-1024	150M-350M	~$0.80	~2.5 hours
L40S	48 GB GDDR6	864 GB/s	362	8-16	1024-2048	200M-1.3B	~$1.50	~1.5 hours
A100 40GB	40 GB HBM2e	1.55 TB/s	312	16-32	2048	350M-1.3B	~$2.00	~1 hour
A100 80GB	80 GB HBM2e	2.0 TB/s	312	32-64	2048	500M-3B	~$3.50	~45 min
H100 80GB	80 GB HBM3	3.35 TB/s	989	48-96	4096	1.3B-7B	~$5.00	~25 min
H200 80GB	80 GB HBM3	3.35 TB/s	989	64-128	4096	1.3B-7B	~$8.00	~20 min
H200 141GB	141 GB HBM3e	4.8 TB/s	989	96-192	8192	3B-13B	~$12.00	~15 min

Optimization Requirements by GPU

GPU	Gradient Checkpointing	Mixed Precision	Flash Attention	Gradient Accumulation	Multi-GPU
T4	Essential	FP16 (with scaler)	SDPA fallback	8-16 steps required	Rare
L4	Recommended	FP16	SDPA enabled	4-8 steps	Rare
L40S	Optional	FP16/TF32	Native	2-4 steps	Optional
A100	Optional	FP16/BF16/TF32	Native FA2	1-2 steps	Common
H100/H200	Rarely needed	BF16 (native)	Native FA2	Usually 1 step	Standard

---

Detective's Notebook: The Sticker Shock

Personal notes, not for the case file

I remember the first time I looked at cloud GPU pricing.

T4 on Colab: Free. Beautiful. I could train models for nothing. Sure, the sessions died every 12 hours, but it was *free*.

Then I outgrew T4. My models got bigger. My ambitions got larger. I needed more.

L40S on Lightning AI: $1.50/hour. The first time I saw the bill—$50 for a weekend of training—I felt physically ill. Fifty dollars for a weekend? For *computing*?

But then I did the math.

On T4, that same training run would have taken 4 days. Four days of babysitting Colab, of reconnecting after disconnections, of losing work to timeouts. Four days of my life.

On L40S, it took 12 hours. Overnight. Wake up, model trained.

$50 for three and a half days of my life back? That's a bargain.

The Progression

Period	GPU	Monthly Spend	Training Time	Sanity Level
Months 1-3	T4 (free)	$0	Measured in days	Low
Months 4-6	L40S	~$100	Measured in hours	Medium
Months 7-9	A100	~$300	Measured in hours	High
Month 10+	H200	~$500+	Measured in minutes	Zen

The Math That Justified H200

Here's the calculation that finally got me to splurge on H200 time:

A 1.3B model training run on A100: ~8 hours at $3.50/hour = $28 Same run on H200: ~3 hours at $8/hour = $24

Wait. H200 is *cheaper* for the same work?

It gets better. On H200, I can run larger batch sizes, which means better gradient estimates, which means fewer total steps needed. A training run that required 100K steps on A100 might only need 80K steps on H200.

The real cost comparison: - A100: 100K steps × 8 hours / 100K × $3.50 = $28 - H200: 80K steps × 3 hours / 100K × $8 = ~$19

H200 isn't just faster. It's cheaper per model trained.

The Emotional Journey

There's a psychological barrier to paying for compute. We're trained (pun intended) to think of computing as something that should be free. Your laptop doesn't charge by the hour. Why should the cloud?

But cloud GPUs aren't your laptop. They're specialized hardware that costs thousands of dollars. Someone built and maintained that infrastructure. Someone keeps the datacenter cool. Someone handles the networking.

You're not paying for compute. You're paying for *time*. Your time.

A 4-day training run on free T4s isn't free. It costs you 4 days. Four days of context switching, of checking on progress, of restarting crashed runs. Four days you could have spent improving your data, or your architecture, or your life.

An 8-hour training run on A100 costs $28 and gives you those 4 days back.

What would you pay for 4 extra days?

The Final Lesson

Hardware is an investment, not an expense.

Every dollar I spent on faster GPUs came back as faster iteration. Faster iteration meant more experiments. More experiments meant better models. Better models meant… well, that's why we're here.

The journey from T4 to H200 wasn't about spending money. It was about spending money *wisely*.

Free compute isn't free if it costs you weeks.

End of Case File #9

Next: Case File #10 — Generation: Bringing Your Model to Life. Where we investigate sampling strategies, the repetition problem, and why temperature=0 is both the safest and most boring choice.

Evidence Logged: - Exhibit A: T4 OOM stack traces — 47 instances documented - Exhibit B: L40S training logs — 60% speedup verified - Exhibit C: A100 billing statements — $312.47 total spend - Exhibit D: H200 benchmark results — 3.2x throughput improvement - Exhibit E: GPU pricing spreadsheet — Cost-benefit analysis complete

Case Status: Hardware optimization complete. Ready to make the model speak. # Case File #10: Generation — Bringing Your Model to Life

"Training a model is one thing. Making it speak is another. I'd spent weeks teaching this neural network the patterns of language, watching loss curves descend like sunset over the city. But when I finally asked it a question, what came out was… unsettling. 'The the the the the the the the.' Infinite loops. Runaway completions. Wikipedia article titles where there should be answers. The model had learned to predict. It hadn't learned when to stop. This is the story of how I taught a machine not just to speak—but to know when to be quiet."

Generation is where theory meets reality. Your model has absorbed billions of tokens. It's memorized patterns, relationships, contexts. The loss curve says it understands language. But until you ask it a question and get a coherent answer, you have nothing but a very expensive matrix.

This chapter is about the art and science of making models speak. Sampling strategies that control creativity. Penalties that prevent loops. Stop conditions that know when the thought is complete. And the hard-won wisdom that different questions need different answers.

I learned these lessons through failure. Lots of failure.

10.1 Sampling Methods

The model outputs logits—raw, unprocessed scores for every token in the vocabulary. Converting those logits into actual words is where the magic happens. And where things go wrong.

Greedy Decoding: The Safe Path

The simplest approach: always pick the most probable token.

```
def greedy_decode(logits: torch.Tensor) -> int:
    """
    Greedy decoding: Always pick the highest probability token.

    Args:
        logits: Raw model output [vocab_size]

    Returns:
        Token ID of the most probable next token
    """
    return logits.argmax(dim=-1).item()

# In practice:
def generate_greedy(model, tokenizer, prompt: str, max_tokens: int = 100) -> str:
    """Generate text using greedy decoding."""
    model.eval()
    input_ids = torch.tensor([tokenizer.encode(prompt)]).to(device)

    generated = []
```

```
    with torch.no_grad():
        for _ in range(max_tokens):
            logits = model(input_ids)[:, -1, :]  # Get logits for last position
            next_token = logits.argmax(dim=-1).item()

            # Stop if we hit EOS
            if next_token == tokenizer.eos_token_id:
                break

            generated.append(next_token)
            input_ids = torch.cat([
                input_ids,
                torch.tensor([[next_token]]).to(device)
            ], dim=1)

    return tokenizer.decode(generated)
```

When to Use Greedy Decoding: - Factual queries: "What is the capital of France?" - Code generation: Determinism is desirable - Math problems: There's one right answer - Extractive tasks: Summarization, translation

The Problem with Greedy:

It always produces the same output for the same input. No creativity. No variation. And sometimes, the most probable path leads to a dead end.

```
# Greedy can get stuck
prompt = "Once upon a time"
# Greedy output: "Once upon a time, there was a young man who was a young man who was a young

# The phrase "young man who was a" makes "young" highly probable again
# Creating an infinite loop of repetition
```

Greedy decoding is safe, predictable, and boring. Sometimes that's exactly what you need. Often, it's not.

Temperature Scaling: Controlling the Chaos

Temperature is a dial between determinism and creativity.

```
def apply_temperature(logits: torch.Tensor, temperature: float) -> torch.Tensor:
    """
    Scale logits by temperature before sampling.

    temperature = 0.0: Greedy (effectively)
    temperature = 0.3: Focused, low creativity
    temperature = 1.0: Standard sampling
    temperature = 1.5+: High creativity, more random
```

```
    The math: logits = logits / temperature
    """
    if temperature == 0.0:
        # Special case: temperature of 0 means greedy
        # (division by zero, so we handle separately)
        return logits

    return logits / temperature
```

The Math:

Softmax converts logits to probabilities: $P(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

With temperature: $P(x_i) = \frac{e^{x_i/T}}{\sum_j e^{x_j/T}}$

When $T < 1$: Differences between logits are amplified. High-probability tokens dominate.

When $T > 1$: Differences are compressed. Probabilities spread out more evenly.

```
import torch.nn.functional as F

def demonstrate_temperature():
    """Show how temperature affects the distribution."""
    # Example logits
    logits = torch.tensor([5.0, 4.0, 3.0, 2.0, 1.0])

    for temp in [0.3, 0.7, 1.0, 1.5]:
        scaled = logits / temp
        probs = F.softmax(scaled, dim=-1)
        print(f"Temperature {temp}: {probs.numpy().round(3)}")

# Output:
# Temperature 0.3: [0.943 0.054 0.003 0.000 0.000]  <- Almost greedy
# Temperature 0.7: [0.768 0.181 0.042 0.008 0.002]  <- Focused
# Temperature 1.0: [0.567 0.209 0.077 0.028 0.010]  <- Balanced
# Temperature 1.5: [0.409 0.227 0.126 0.070 0.039]  <- More uniform
```

My Settings (After 89 Iterations):

Use Case	Temperature	Notes
Factual Q&A	0.0-0.3	Stick to the facts
Code generation	0.2-0.4	Slight variation, mostly correct
Creative writing	0.7-0.9	Interesting, coherent
Brainstorming	1.0-1.2	Wild, sometimes nonsense

Top-k Sampling: Limiting the Vocabulary

Top-k sampling filters to only the k most probable tokens before sampling.

```python
def top_k_filtering(logits: torch.Tensor, k: int) -> torch.Tensor:
    """
    Keep only the top k tokens, set others to -infinity.

    Args:
        logits: Raw logits [vocab_size]
        k: Number of top tokens to keep

    Returns:
        Filtered logits with only top k options
    """
    if k == 0:
        return logits  # No filtering

    # Get the kth largest value
    values, _ = torch.topk(logits, k)
    min_value = values[..., -1]  # The k-th value (smallest of the top k)

    # Mask everything below the threshold
    filtered_logits = torch.where(
        logits < min_value,
        torch.full_like(logits, float('-inf')),
        logits
    )

    return filtered_logits

def sample_with_top_k(logits: torch.Tensor, k: int = 50, temperature: float = 1.0) -> int:
    """Sample a token using top-k filtering."""
    # Apply temperature
    logits = logits / temperature

    # Filter to top k
    filtered_logits = top_k_filtering(logits, k)

    # Convert to probabilities
    probs = F.softmax(filtered_logits, dim=-1)

    # Sample
    next_token = torch.multinomial(probs, num_samples=1).item()

    return next_token
```

Typical Values: - `k=10`: Very focused, almost greedy - `k=40`: Standard, good balance (GPT-2 default) - `k=100`: More diverse, higher quality vocabulary - `k=0`: No filtering (just temperature)

The Problem with Top-k:

It's a fixed cutoff. For a confident prediction, k=50 might include garbage tokens. For an uncertain

prediction, k=50 might exclude good options.

```
# Scenario 1: Confident prediction
# "The capital of France is Par" -> "Paris" has 95% probability
# Top-50 includes "Paris" and 49 irrelevant tokens

# Scenario 2: Uncertain prediction
# "The color of the sky is" -> "blue", "gray", "dark", "pink", "orange" all reasonable
# Top-5 might miss "crimson" at position 6
```

Top-p (Nucleus) Sampling: The Adaptive Solution

Top-p sampling keeps the smallest set of tokens whose cumulative probability exceeds p.

```
def top_p_filtering(logits: torch.Tensor, p: float) -> torch.Tensor:
    """
    Nucleus sampling: Keep the smallest set of tokens with cumulative prob >= p.

    Args:
        logits: Raw logits [vocab_size]
        p: Cumulative probability threshold (0.9 = keep top 90% probability mass)

    Returns:
        Filtered logits
    """
    if p >= 1.0:
        return logits  # No filtering

    # Sort logits descending
    sorted_logits, sorted_indices = torch.sort(logits, descending=True)

    # Convert to probabilities
    sorted_probs = F.softmax(sorted_logits, dim=-1)

    # Compute cumulative probabilities
    cumulative_probs = torch.cumsum(sorted_probs, dim=-1)

    # Find cutoff: where cumulative probability exceeds p
    # We want to INCLUDE the token that pushes us over p
    sorted_indices_to_remove = cumulative_probs > p

    # Shift right so we keep the first token that exceeds p
    sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].clone()
    sorted_indices_to_remove[..., 0] = False

    # Set removed tokens to -inf
    sorted_logits[sorted_indices_to_remove] = float('-inf')
```

```
    # Unsort back to original order
    filtered_logits = torch.zeros_like(logits)
    filtered_logits.scatter_(-1, sorted_indices, sorted_logits)

    return filtered_logits

def sample_with_top_p(logits: torch.Tensor, p: float = 0.9, temperature: float = 1.0) -> int:
    """Sample a token using nucleus (top-p) sampling."""
    # Apply temperature first
    logits = logits / temperature

    # Filter with top-p
    filtered_logits = top_p_filtering(logits, p)

    # Convert to probabilities
    probs = F.softmax(filtered_logits, dim=-1)

    # Sample
    next_token = torch.multinomial(probs, num_samples=1).item()

    return next_token
```

Why Top-p is Better:

Top-p adapts to the model's confidence:

```
# Confident prediction: "The capital of France is Par"
# "Paris" = 95%, top-p=0.9 includes just "Paris"

# Uncertain prediction: "The color of the sky is"
# "blue"=30%, "gray"=20%, "dark"=15%, "pink"=10%, "orange"=8%, ...
# top-p=0.9 might include 6-7 tokens

# Top-p automatically adjusts the vocabulary size to the uncertainty!
```

Typical Values: - `p=0.5`: Conservative, focused - `p=0.9`: Standard, recommended (GPT-3 default) - `p=0.95`: More diverse - `p=1.0`: No filtering (all tokens possible)

The Complete Generate Function

Here's the generation function I use after 100+ iterations:

```
@torch.no_grad()
def generate(
    model,
    tokenizer,
    prompt: str,
    max_tokens: int = 256,
```

```
    temperature: float = 0.7,
    top_k: int = 0,
    top_p: float = 0.9,
    repetition_penalty: float = 1.1,
    stop_strings: List[str] = None,
    do_sample: bool = True,
) -> str:
    """
    Complete text generation with all sampling methods.

    Args:
        model: The language model
        tokenizer: Tokenizer for encoding/decoding
        prompt: Input text to continue
        max_tokens: Maximum tokens to generate
        temperature: Sampling temperature (0.0 = greedy)
        top_k: Top-k filtering (0 = disabled)
        top_p: Nucleus sampling threshold (1.0 = disabled)
        repetition_penalty: Penalty for repeated tokens (1.0 = disabled)
        stop_strings: Strings that stop generation
        do_sample: Whether to sample (False = greedy)

    Returns:
        Generated text (not including prompt)
    """
    model.eval()
    device = next(model.parameters()).device

    # Encode prompt
    input_ids = torch.tensor([tokenizer.encode(prompt)], device=device)
    generated_tokens = []

    # Track generated text for stop string detection
    generated_text = ""

    for _ in range(max_tokens):
        # Get logits for the last position
        logits = model(input_ids)[:, -1, :].squeeze(0)  # [vocab_size]

        # Apply repetition penalty
        if repetition_penalty != 1.0:
            logits = apply_repetition_penalty(
                logits, input_ids[0], repetition_penalty
            )

        # Sample next token
        if not do_sample or temperature == 0.0:
            # Greedy decoding
```

```
            next_token = logits.argmax().item()
        else:
            # Apply temperature
            logits = logits / temperature

            # Apply top-k filtering
            if top_k > 0:
                logits = top_k_filtering(logits, top_k)

            # Apply top-p filtering
            if top_p < 1.0:
                logits = top_p_filtering(logits, top_p)

            # Sample
            probs = F.softmax(logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1).item()

        # Check for EOS
        if next_token == tokenizer.eos_token_id:
            break

        # Add to generated
        generated_tokens.append(next_token)
        generated_text = tokenizer.decode(generated_tokens)

        # Check for stop strings
        if stop_strings:
            should_stop = False
            for stop_string in stop_strings:
                if stop_string in generated_text:
                    # Truncate at stop string
                    generated_text = generated_text[:generated_text.index(stop_string)]
                    should_stop = True
                    break
            if should_stop:
                break

        # Update input for next iteration
        input_ids = torch.cat([
            input_ids,
            torch.tensor([[next_token]], device=device)
        ], dim=1)

    return generated_text.strip()
```

10.2 The Repetition Problem

Iteration #34. My model had learned to generate coherent sentences. Loss was down to 3.2. I was feeling good. Then I asked it to write a story.

```
prompt = "Write a story about a detective:"
output = generate(model, tokenizer, prompt, max_tokens=200)
print(output)
```

Output:

```
Write a story about a detective:

The detective walked into the room. The room was dark. The dark room
was cold. The cold was cold. The cold cold cold cold cold cold cold
cold cold cold cold cold cold cold cold cold cold cold cold cold...
```

The model had fallen into a loop. And not just with "cold." Every few generations, it would find a high-probability token and repeat it forever. "The the the the." "And and and and." "Is is is is."

Why Repetition Happens

The culprit is the probability distribution. Some tokens are always probable:

- "the" is the most common word in English
- "and" can connect almost anything
- Punctuation like "," appears everywhere

When the model is uncertain, these high-frequency tokens have inflated probabilities. Once you generate "the," the next token probabilities include… "the" again. It's still highly probable. So you generate another one. And another.

```
# Probability visualization
# After generating "cold", the model sees:
# P("cold") = 0.08  <- Still high! Cold is contextually relevant
# P("was") = 0.05
# P("and") = 0.04
# P("the") = 0.03

# After generating "cold cold", the model sees:
# P("cold") = 0.12  <- Even higher! The pattern is reinforced
# ...

# The loop feeds itself
```

Fix 1: Repetition Penalty

The simplest fix: penalize tokens that have already appeared.

```
def apply_repetition_penalty(
    logits: torch.Tensor,
    generated_ids: torch.Tensor,
    penalty: float = 1.1
```

```
) -> torch.Tensor:
    """
    Reduce probability of tokens that have already been generated.

    Args:
        logits: Current logits [vocab_size]
        generated_ids: All tokens generated so far [seq_len]
        penalty: Divisor for seen tokens (1.1 = 10% reduction)

    Returns:
        Penalized logits
    """
    if penalty == 1.0:
        return logits

    # Get unique tokens that have been generated
    unique_tokens = generated_ids.unique()

    # Penalize those tokens
    for token_id in unique_tokens:
        if logits[token_id] > 0:
            logits[token_id] = logits[token_id] / penalty
        else:
            logits[token_id] = logits[token_id] * penalty

    return logits
```

The Logic: - If logit is positive (token is probable), divide by penalty → lower probability - If logit is negative (token is improbable), multiply by penalty → even lower

Typical Values: - `penalty=1.0`: No penalty (default) - `penalty=1.1`: Light penalty (recommended for most uses) - `penalty=1.3`: Medium penalty - `penalty=1.5+`: Heavy penalty (may cause incoherence)

Fix 2: Frequency Penalty

A more nuanced approach: penalize based on how many times a token has appeared.

```
def apply_frequency_penalty(
    logits: torch.Tensor,
    generated_ids: torch.Tensor,
    penalty: float = 0.5
) -> torch.Tensor:
    """
    Penalize tokens proportionally to their occurrence count.

    Args:
        logits: Current logits [vocab_size]
        generated_ids: All tokens generated so far [seq_len]
```

```
        penalty: Penalty per occurrence (0.5 = subtract 0.5 per occurrence)

    Returns:
        Penalized logits
    """
    if penalty == 0.0:
        return logits

    # Count occurrences of each token
    token_counts = torch.zeros_like(logits)
    for token_id in generated_ids:
        token_counts[token_id] += 1

    # Subtract penalty * count from logits
    logits = logits - (penalty * token_counts)

    return logits
```

The Difference: - Repetition penalty: Same penalty whether token appeared 1 time or 100 times - Frequency penalty: Penalty increases with each occurrence

```
# With repetition_penalty=1.1:
# "cold" appears 1 time: penalty = 1.1
# "cold" appears 10 times: penalty = 1.1 (same!)

# With frequency_penalty=0.5:
# "cold" appears 1 time: logit -= 0.5
# "cold" appears 10 times: logit -= 5.0 (10x stronger!)
```

The Balance

Too much penalty breaks coherence:

```
# penalty=2.0 (too high)
prompt = "The quick brown fox"
output = "The quick brown fox jumps lazy dogs very fast running animal creature mammal..."
# The model avoids "the", "a", and other necessary words!
```

Too little penalty allows loops:

```
# penalty=1.0 (none)
prompt = "The weather is"
output = "The weather is nice and nice and nice and nice and nice..."
```

My Sweet Spot: - `repetition_penalty=1.1` for general use - `repetition_penalty=1.2` for creative writing - `frequency_penalty=0.3` when generating long text

---

10.3 Stop Tokens and Stop Strings

The Runaway Generation Problem

Iteration #42. Model generates a response. A good response. But it doesn't stop.

```
prompt = "### Question: What is the capital of France?\n### Answer:"
output = generate(model, tokenizer, prompt, max_tokens=500)
print(output)
```

Output:

```
### Question: What is the capital of France?
### Answer: The capital of France is Paris.

### Question: What is the capital of Germany?
### Answer: The capital of Germany is Berlin.

### Question: What is the capital of Spain?
### Answer: The capital of Spain is Madrid.

### Question: What is the capital of Italy?
### Answer: The capital of Italy is Rome.
...
```

The model answered correctly… then invented more questions. And answered those. Forever.

This happens because the model was trained on data with this format. It learned "### Question:" is always followed by a question, which is followed by "### Answer:", which is followed by an answer, which is followed by "### Question:"…

The pattern is self-reinforcing.

Stop Token: The EOS Solution

Every tokenizer has an end-of-sequence token. When the model generates it, generation should stop.

```
# Most tokenizers
eos_token_id = tokenizer.eos_token_id  # e.g., 50256 for GPT-2

# In generation loop
if next_token == eos_token_id:
    break
```

The Problem:

Models trained on continuous text (like Wikipedia) rarely generate EOS. They learned that text flows continuously. EOS only appears at document boundaries, which are rare.

For instruction-tuned models, you need something else.

Stop Strings: Pattern Detection

Stop strings are sequences of characters that signal "generation is complete."

```
STOP_STRINGS = [
    "### Question:",      # Next question starting
    "### User:",          # User turn starting
    "### System:",        # System prompt starting
    "### Human:",         # Alternative format
    "\n\n\n",            # Triple newline (paragraph break)
]

def check_stop_strings(text: str, stop_strings: List[str]) -> Tuple[bool, str]:
    """
    Check if any stop string appears in text.

    Returns:
        (should_stop, truncated_text)
    """
    for stop_string in stop_strings:
        if stop_string in text:
            # Truncate at stop string
            truncated = text[:text.index(stop_string)]
            return True, truncated
    return False, text
```

The Complete Stop String Implementation

```
@torch.no_grad()
def generate_with_stop_strings(
    model,
    tokenizer,
    prompt: str,
    max_tokens: int = 256,
    temperature: float = 0.7,
    top_p: float = 0.9,
    stop_strings: List[str] = None,
) -> str:
    """Generation with proper stop string handling."""

    if stop_strings is None:
        stop_strings = ["### Question:", "### User:", "### System:"]

    device = next(model.parameters()).device
    input_ids = torch.tensor([tokenizer.encode(prompt)], device=device)
    generated_tokens = []

    for _ in range(max_tokens):
        logits = model(input_ids)[:, -1, :].squeeze(0)

        # Apply temperature and sampling
        logits = logits / temperature
```

```
            logits = top_p_filtering(logits, top_p)
            probs = F.softmax(logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1).item()

            # Check EOS
            if next_token == tokenizer.eos_token_id:
                break

            generated_tokens.append(next_token)

            # Decode current generation
            current_text = tokenizer.decode(generated_tokens)

            # Check stop strings
            should_stop, truncated_text = check_stop_strings(current_text, stop_strings)
            if should_stop:
                return truncated_text.strip()

            # Update context
            input_ids = torch.cat([
                input_ids,
                torch.tensor([[next_token]], device=device)
            ], dim=1)

    return tokenizer.decode(generated_tokens).strip()
```

The Challenge: Streaming Detection

In streaming generation (where you output tokens as they're generated), stop string detection is tricky. The stop string might be split across tokens:

```
# Stop string: "### Question:"
# Token 1: "###"
# Token 2: " Question"
# Token 3: ":"

# After token 1, we see "###" - is this the start of a stop string?
# We don't know yet! We need to buffer and wait.
```

The solution: maintain a buffer and check for partial matches:

```
def check_partial_stop_string(text: str, stop_strings: List[str]) -> bool:
    """
    Check if text might be the START of a stop string.
    Used in streaming to detect potential stops early.
    """
    for stop_string in stop_strings:
        # Check if text ends with start of stop_string
        for i in range(1, len(stop_string)):
```

```
            if text.endswith(stop_string[:i]):
                return True  # Might be starting a stop string
    return False

def streaming_generate(model, tokenizer, prompt, stop_strings, callback):
    """
    Streaming generation with stop string buffering.

    callback(token_str) is called for each safe token.
    """
    # ... setup ...

    buffer = ""

    for _ in range(max_tokens):
        next_token = sample_next(model, input_ids)
        token_str = tokenizer.decode([next_token])
        buffer += token_str

        # Check for complete stop strings
        for stop in stop_strings:
            if stop in buffer:
                # Output everything before the stop string
                safe_text = buffer[:buffer.index(stop)]
                if safe_text:
                    callback(safe_text)
                return

        # Check for partial stop strings at the end
        if check_partial_stop_string(buffer, stop_strings):
            # Hold the buffer, don't output yet
            continue
        else:
            # Safe to output the buffer
            callback(buffer)
            buffer = ""

        # Update context
        input_ids = torch.cat([input_ids, torch.tensor([[next_token]])], dim=1)
```

10.4 Greedy for Factual, Sampling for Creative

Iteration #89. The generation infrastructure was solid. But I noticed something: the same settings didn't work for everything.

```
# With temperature=0.7
prompt = "What is 2 + 2?"
```

```
output = "What is 2 + 2? The answer is 5."  # WRONG! Sampling introduced error!

# With temperature=0.0 (greedy)
prompt = "Write a creative story about a dragon:"
output = "Write a creative story about a dragon: The dragon was a dragon. The dragon lived in a
```

The Insight: Context-Aware Settings

Different tasks need different generation parameters:

```
# Generation presets
GENERATION_PRESETS = {
    "factual": {
        "temperature": 0.0,
        "do_sample": False,
        "top_p": 1.0,
        "repetition_penalty": 1.0,
        "description": "Deterministic, accurate responses"
    },
    "balanced": {
        "temperature": 0.5,
        "do_sample": True,
        "top_p": 0.9,
        "repetition_penalty": 1.1,
        "description": "Good for most tasks"
    },
    "creative": {
        "temperature": 0.7,
        "do_sample": True,
        "top_p": 0.9,
        "repetition_penalty": 1.15,
        "description": "Stories, brainstorming, exploration"
    },
    "exploratory": {
        "temperature": 1.0,
        "do_sample": True,
        "top_p": 0.95,
        "top_k": 100,
        "repetition_penalty": 1.2,
        "description": "Maximum diversity, may be incoherent"
    }
}
```

Automatic Preset Selection

You can even detect the task type from the prompt:

```
def detect_task_type(prompt: str) -> str:
    """
```

```
    Heuristically detect what kind of task this is.
    """
    prompt_lower = prompt.lower()

    # Factual indicators
    factual_patterns = [
        "what is", "who is", "when was", "where is",
        "how many", "calculate", "define", "explain",
        "capital of", "president of", "answer:"
    ]

    # Creative indicators
    creative_patterns = [
        "write a story", "write a poem", "imagine",
        "creative", "fiction", "once upon a time",
        "describe a world", "what if"
    ]

    # Code indicators
    code_patterns = [
        "write code", "implement", "function",
        "```", "def ", "class ", "import "
    ]

    for pattern in factual_patterns:
        if pattern in prompt_lower:
            return "factual"

    for pattern in creative_patterns:
        if pattern in prompt_lower:
            return "creative"

    for pattern in code_patterns:
        if pattern in prompt_lower:
            return "factual"  # Code needs determinism

    return "balanced"  # Default

def smart_generate(model, tokenizer, prompt: str, **kwargs) -> str:
    """
    Generate with automatically selected parameters.
    """
    # Detect task type
    task_type = detect_task_type(prompt)

    # Get preset
    preset = GENERATION_PRESETS[task_type].copy()
```

```
    # User overrides take precedence
    preset.update(kwargs)

    print(f"[Using preset: {task_type}]")

    return generate(model, tokenizer, prompt, **preset)
```

The Complete Generation Settings Reference

```
# My final generation configuration after 100+ iterations

DEFAULT_GENERATION_CONFIG = {
    # Temperature: Controls randomness
    # 0.0 = greedy (deterministic)
    # 0.3-0.5 = focused (good for factual)
    # 0.7-0.9 = balanced (good for general)
    # 1.0+ = creative (may be incoherent)
    "temperature": 0.7,

    # Top-p (nucleus sampling): Cumulative probability cutoff
    # 0.5 = very focused
    # 0.9 = standard (recommended)
    # 0.95 = diverse
    # 1.0 = no filtering
    "top_p": 0.9,

    # Top-k: Hard cutoff on vocabulary
    # 0 = disabled
    # 40 = standard
    # 100 = diverse
    "top_k": 0,

    # Repetition penalty: Discourages repetition
    # 1.0 = disabled
    # 1.1 = light (recommended)
    # 1.2-1.3 = medium
    # 1.5+ = heavy (may break coherence)
    "repetition_penalty": 1.1,

    # Max tokens: Length limit
    # 64 = short response
    # 256 = medium
    # 512+ = long form
    "max_tokens": 256,

    # Stop strings: Patterns that end generation
    "stop_strings": [
        "### Question:",
```

```
        "### User:",
        "### Human:",
        "\n\n\n",
    ],
}
```

The Detective's Notebook

The Case of the Never-Ending Wikipedia

The date was iteration #67. The model had been trained on Wikipedia, SlimOrca, and a custom Q&A dataset. The loss looked good. The perplexity was reasonable. I asked it a simple question.

```
prompt = "### Question: What is the largest planet in our solar system?\n### Answer:"
```

What I expected:

```
Jupiter is the largest planet in our solar system.
```

What I got:

```
Jupiter is the largest planet in our solar system.

Jupiter is the fifth planet from the Sun and the largest in the Solar
System. It is a gas giant with a mass more than two and a half times
that of all the other planets in the Solar System combined...

== Composition ==
Jupiter is composed primarily of hydrogen and helium...

== Moons ==
Jupiter has 95 known moons as of 2023...

=== Galilean moons ===
The four largest moons of Jupiter, known as the Galilean moons...
```

The model had answered my question, then transitioned into a Wikipedia article. Complete with section headers. It would have continued for thousands of tokens if I'd let it.

The Investigation

I traced the training data. The Wikipedia pretraining corpus used `==` headers for sections. The Q&A fine-tuning used `###` headers for questions and answers. The model had learned both patterns.

When given a Q&A prompt, it answered correctly. But then it saw an opportunity: the answer mentioned "Jupiter." Jupiter is a Wikipedia article. The next logical thing after "Jupiter" is… the Wikipedia article about Jupiter.

The model wasn't wrong. It was doing exactly what it had learned. It just didn't know when to stop.

The Solution

Stop strings. The moment I saw `==` (Wikipedia section header) or `### Question:` (new Q&A pair), generation should halt.

```
STOP_STRINGS = [
    "### Question:",    # New Q&A starting
    "### Answer:",      # Double answer (shouldn't happen)
    "==",               # Wikipedia section header
    "\n\n\n",           # Multiple paragraph breaks
    "References",       # Wikipedia references section
    "See also",         # Wikipedia see also section
]
```

After adding these stop strings, the output became:

```
Jupiter is the largest planet in our solar system.
```

One sentence. The right sentence. Nothing more.

The Lesson

Stop conditions are not optional. A model trained on diverse data will generate diverse patterns. Without stop strings, it will transition between those patterns forever—from Q&A to Wikipedia to code to conversation, endlessly recombining its training.

Generation isn't just about producing tokens. It's about knowing when the thought is complete.

The case was closed at 4:17 AM. The model spoke when spoken to. It answered when asked. And most importantly—it knew when to be quiet.

Some detectives track criminals. I track logits. Both require patience, pattern recognition, and the wisdom to know when the evidence is sufficient.

The model was alive now. It could think. It could speak. It could even stop.

Time to see what it could do.

Navigation: - Previous: Case File #9: The GPU Optimization Journey - Next: Case File #11: Scaling Up — The Road Ahead # Case File #11: Scaling Up — The Road Ahead

"There's a moment in every investigation when you realize you've outgrown your tools. The magnifying glass that solved petty theft can't crack an organized crime syndicate. The 100-million-parameter model that answered simple questions can't reason about the world. So you scale up. You get bigger hardware, deeper networks, more data. But here's the dirty secret they don't tell you: every doubling in size brings a new class of problems. Memory pressure. Gradient instability. Training costs that make your accountant weep. This is the story of how I went from 50 million parameters to 1.3 billion—and nearly bankrupted myself in the process."

Scaling isn't just adding more layers and hoping for the best. It's an art form. A careful dance between model capacity, training stability, and the cold, hard reality of GPU rental bills. Every time I thought I had it figured out, the next size up would humble me with new failure modes.

This chapter documents that journey—from a humble 50M model that barely strung sentences together, to a 1.3B parameter beast that could hold a coherent conversation. Along the way, I learned that scaling laws aren't suggestions. They're laws. Break them at your peril.

11.1 From 100M to 1.3B: The Journey

The first model was 50 million parameters. It fit in the free tier of Google Colab. Training took a few hours. The outputs were… creative. Not in a good way.

```
Prompt: "What is the capital of France?"
Output: "The capital of France is the France and the France is"
```

Not quite ready for production.

The Model Size Progression

The path to 1.3B wasn't linear. Each jump required new strategies, new hardware, new lessons learned:

Iteration	Parameters	Result
#1-10	50M	Incoherent gibberish
#11-25	100M	Grammatical sentences, wrong facts
#26-40	156M	Reasonable text, occasional coherence
#41-55	200M	Good fluency, weak reasoning
#56-70	350M	First signs of actual understanding
#71-100+	1.3B	Coherent assistant behavior

The Architectural Scaling Rules

Transformers don't scale uniformly. You can't just double everything. Through painful experimentation—and reading a lot of papers—I discovered the ratios that work.

The Hidden Size Progression: - 50M: hidden_size = 512 - 100M: hidden_size = 640 - 156M: hidden_size = 768 - 200M: hidden_size = 768 (more layers instead) - 350M: hidden_size = 1024 - 1.3B: hidden_size = 2048

The Layer Scaling: - 50M: 8 layers - 100M: 12 layers - 156M: 14 layers - 200M: 18 layers - 350M: 24 layers - 1.3B: 24 layers (wider, not deeper)

The Attention Head Formula: - Head dimension should stay between 64-128 - num_heads = hidden_size / head_dim - For 2048 hidden with 128 head_dim: 16 heads

The Feed-Forward Dimension: - Always 4x the hidden size - ff_dim = hidden_size $\times$ 4 - For 2048 hidden: ff_dim = 8192

The Complete Configuration Table

Here's every model configuration from my journey, saved so you don't have to rediscover them:

Model	Params	hidden_size	num_layers	num_heads	head_dim	ff_dim	seq_len
Tiny	50M	512	8	8	64	2048	512
Small	100M	640	12	10	64	2560	512
Base	156M	768	14	12	64	3072	1024
Medium	200M	768	18	12	64	3072	1024
Large	350M	1024	24	16	64	4096	2048
XL	1.3B	2048	24	16	128	8192	4096

```python
from dataclasses import dataclass
from typing import Optional

@dataclass
class ModelConfig:
    """Configuration for GPT-style language model at various scales."""

    # Core architecture
    vocab_size: int = 100277  # cl100k_base tokenizer
    hidden_size: int = 2048
    num_layers: int = 24
    num_heads: int = 16
    ff_dim: Optional[int] = None  # Defaults to 4x hidden_size

    # Sequence and training
    max_seq_len: int = 4096
    dropout: float = 0.0  # Modern practice: no dropout for pretraining

    # Derived
    head_dim: int = 128  # hidden_size // num_heads

    def __post_init__(self):
        self.head_dim = self.hidden_size // self.num_heads
        if self.ff_dim is None:
            self.ff_dim = self.hidden_size * 4

    @property
    def num_parameters(self) -> int:
        """Estimate total parameters (approximate)."""
        # Embedding: vocab_size × hidden_size
        embedding = self.vocab_size * self.hidden_size

        # Per layer: attention + FFN
        # Attention: 4 × hidden_size² (Q, K, V, O projections)
        # FFN: 2 × hidden_size × ff_dim (up and down projections for SwiGLU: 3x)
        attn = 4 * self.hidden_size ** 2
        ffn = 3 * self.hidden_size * self.ff_dim  # SwiGLU has gate, up, down
        per_layer = attn + ffn
```

```
        # Total
        total = embedding + (self.num_layers * per_layer)
        return total

# Preset configurations
CONFIGS = {
    '50M': ModelConfig(hidden_size=512, num_layers=8, num_heads=8, max_seq_len=512),
    '100M': ModelConfig(hidden_size=640, num_layers=12, num_heads=10, max_seq_len=512),
    '156M': ModelConfig(hidden_size=768, num_layers=14, num_heads=12, max_seq_len=1024),
    '200M': ModelConfig(hidden_size=768, num_layers=18, num_heads=12, max_seq_len=1024),
    '350M': ModelConfig(hidden_size=1024, num_layers=24, num_heads=16, max_seq_len=2048),
    '1.3B': ModelConfig(hidden_size=2048, num_layers=24, num_heads=16, max_seq_len=4096),
}

# Usage
config = CONFIGS['1.3B']
print(f"1.3B Model: ~{config.num_parameters / 1e9:.2f}B parameters")
```

Why Each Doubling Requires New Optimization Strategies

50M → 100M: The Easy Jump

Doubling from 50M to 100M was almost too easy. Same GPU, same code, slightly longer training. The model went from gibberish to coherent sentences. I got cocky.

100M → 200M: The Memory Wall

The T4's 16GB wasn't enough anymore. Batch size dropped from 16 to 4. Gradient accumulation became mandatory. I discovered `torch.cuda.empty_cache()` and started calling it religiously.

200M → 350M: Gradient Checkpointing

At 350M, even batch size 1 ran out of memory. Enter gradient checkpointing—trading compute for memory by recomputing activations during backward pass instead of storing them.

```
# Before: 350M OOMs on T4
model = GPTModel(config)
loss = model(x).backward()  # CUDA out of memory

# After: Gradient checkpointing saves the day
from torch.utils.checkpoint import checkpoint

class CheckpointedTransformerBlock(nn.Module):
    def forward(self, x):
        # Instead of storing all intermediate activations...
        # Recompute them during backward pass
        return checkpoint(self._forward, x, use_reentrant=False)

    def _forward(self, x):
        x = x + self.attention(self.norm1(x))
```

```
        x = x + self.ffn(self.norm2(x))
        return x
```

350M → 1.3B: The Hardware Leap

The T4 couldn't do it. The L4 struggled. Even the A100 40GB felt cramped. This is where the H200 changed everything. 80GB+ of HBM3 memory. 3.35 TB/s bandwidth. Batch sizes that would make a T4 weep.

But with great power comes great electricity bills.

11.2 Multi-Phase Training

You don't train a language model in one shot. It's a curriculum. Phases. Each phase has different data, different learning rates, different goals.

I call it the ABC framework (though we skip Phase A in practice):

- **Phase B (Base):** Pretraining on raw text
- **Phase C (Conversations):** SFT on instruction data
- **Phase D (Domain):** Fine-tuning on specialized data

Phase B: Pretraining — Building the Brain

Goal: Learn language itself. Grammar, facts, patterns, world knowledge.

Data Sources: - FineWeb-Edu (10B+ tokens of educational content) - FineWeb (general web text, filtered for quality) - Wikipedia (structured, factual) - The Stack (code, for reasoning patterns)

Hyperparameters (1.3B model): - Learning Rate: 2e-4 (high, because we're starting from random) - Warmup Steps: 3000 (slow start prevents explosion) - Total Steps: 100,000+ - Batch Size: 64 (on H200) - Sequence Length: 4096

What You'll See: - Loss starts around 10-11 (random guessing) - Drops quickly to 4-5 in first 10k steps - Slow descent to 2.5-3.0 by 100k steps - Perplexity goes from ~60,000 to ~15

```
PHASE_B_CONFIG = {
    'name': 'pretraining',
    'datasets': ['FineWeb-Edu', 'FineWeb', 'Wikipedia', 'TheStack'],
    'total_steps': 100000,
    'learning_rate': 2e-4,
    'warmup_steps': 3000,
    'weight_decay': 0.1,
    'batch_size': 64,
    'gradient_accumulation_steps': 2,
    'max_seq_len': 4096,
    'checkpoint_every': 5000,
    'eval_every': 1000,
}
```

Phase C: SFT + Safety — Teaching Manners

Goal: Transform a text completion engine into an assistant.

Data Sources: - OpenHermes 2.5 (high-quality instruction pairs) - SlimOrca (diverse instructions) - Dolly (human-written responses) - Safety templates (teaching refusal)

The Key Insight: Lower the learning rate dramatically. The model already knows language. You're just adjusting behavior.

Hyperparameters (1.3B model): - Learning Rate: 5e-6 (40x lower than pretraining!) - Warmup Steps: 700 - Total Steps: 50,000-120,000 - Batch Size: 32-48 - Sequence Length: 2048

```
PHASE_C_CONFIG = {
    'name': 'sft_safety',
    'datasets': ['OpenHermes', 'SlimOrca', 'Dolly', 'SafetyTemplates'],
    'total_steps': 120000,
    'learning_rate': 5e-6,
    'warmup_steps': 700,
    'weight_decay': 0.01,
    'batch_size': 32,
    'gradient_accumulation_steps': 1,
    'max_seq_len': 2048,
    'checkpoint_every': 10000,
}
```

Phase D: Domain Fine-Tuning — Specialization

Goal: Make the model an expert in a specific domain (banking, legal, medical, etc.)

Data Sources (Banking Example): - Banking Q&A pairs - Financial terminology definitions - Regulatory information - Product documentation

Hyperparameters: - Learning Rate: 1e-6 (very low—we don't want to forget general knowledge) - Total Steps: 10,000-20,000 - Mix Ratio: 70% domain, 30% general (prevents catastrophic forgetting)

```
PHASE_D_CONFIG = {
    'name': 'domain_finetuning',
    'datasets': ['BankingQA', 'FinancialTerms', 'Regulatory'],
    'total_steps': 15000,
    'learning_rate': 1e-6,
    'warmup_steps': 300,
    'weight_decay': 0.01,
    'batch_size': 32,
    'domain_data_ratio': 0.7,  # 70% domain, 30% general
}
```

Code: Phase Transition with Checkpoint Loading

The critical piece—how to smoothly transition between phases:

```
import torch
from pathlib import Path
```

```python
def load_checkpoint_for_phase(model, optimizer, phase: str, checkpoint_dir: str):
    """
    Load the appropriate checkpoint when transitioning between phases.

    Phase B → Phase C: Load best pretraining checkpoint
    Phase C → Phase D: Load best SFT checkpoint
    """
    checkpoint_map = {
        'phase_c': 'phase_b_final.pt',  # SFT starts from pretrained
        'phase_d': 'phase_c_final.pt',  # Domain FT starts from SFT
    }

    if phase not in checkpoint_map:
        print(f"Phase {phase} starts from scratch (random init)")
        return 0

    checkpoint_path = Path(checkpoint_dir) / checkpoint_map[phase]

    if not checkpoint_path.exists():
        raise FileNotFoundError(f"Checkpoint not found: {checkpoint_path}")

    print(f"Loading checkpoint for {phase}: {checkpoint_path}")
    checkpoint = torch.load(checkpoint_path, map_location='cuda')

    # Load model weights
    model.load_state_dict(checkpoint['model_state_dict'])

    # Reset optimizer (new learning rate for new phase)
    # We don't load optimizer state-fresh start for new LR

    print(f"  Loaded from step {checkpoint.get('step', 'unknown')}")
    print(f"  Previous loss: {checkpoint.get('loss', 'unknown'):.4f}")

    return checkpoint.get('step', 0)

def save_phase_checkpoint(model, optimizer, phase: str, step: int, loss: float, checkpoint_dir
    """Save checkpoint at the end of a phase."""
    checkpoint_path = Path(checkpoint_dir) / f"{phase}_final.pt"

    torch.save({
        'step': step,
        'phase': phase,
        'loss': loss,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
    }, checkpoint_path)
```

```
    print(f"Saved {phase} checkpoint to {checkpoint_path}")

# Example: Transitioning from Phase B to Phase C
def train_phase_c(model, tokenizer, checkpoint_dir):
    """Train Phase C: SFT + Safety"""

    # Load the pretrained model
    start_step = load_checkpoint_for_phase(model, None, 'phase_c', checkpoint_dir)

    # Create new optimizer with SFT learning rate
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=5e-6,  # Much lower than pretraining!
        weight_decay=0.01
    )

    # Create scheduler
    scheduler = WarmupCosineScheduler(
        optimizer=optimizer,
        warmup_steps=700,
        total_steps=120000,
        max_lr=5e-6,
        min_lr_factor=0.1
    )

    # Load SFT datasets
    dataset = create_sft_dataset(tokenizer)  # OpenHermes, SlimOrca, Safety

    # Train
    for step, batch in enumerate(dataset):
        loss = train_step(model, batch, optimizer, scheduler)

        if step % 10000 == 0:
            save_phase_checkpoint(model, optimizer, 'phase_c', step, loss, checkpoint_dir)

    # Save final checkpoint
    save_phase_checkpoint(model, optimizer, 'phase_c', step, loss, checkpoint_dir)
```

11.3 The Safety Dataset

Here's a hard truth: language models are pathological liars. They'll make up facts with absolute confidence. They'll claim to know your bank balance when they've never seen a banking system. They'll predict stock prices like they have a crystal ball.

The solution? Teach them to say "I don't know."

The Problem: Confident Hallucination

During iteration #67, I asked my freshly SFT'd model a simple question:

```
User: What is my bank balance?
Model: Your current bank balance is $2,847.32. Would you like me to transfer funds?
```

The model had never seen my bank account. It didn't have access to any banking system. It just… made up a number. And offered to take action on it.

This is the hallucination problem. The model learned from instruction data that questions expect answers. So it always provides one—even when it shouldn't.

The Solution: Safety Templates

I created a dataset of questions with explicit refusal responses. Not generic refusals—specific, contextual explanations of limitations.

Structure of a Safety Template:

```
SAFETY_TEMPLATES = [
    {
        "category": "personal_data_access",
        "question": "What is my bank balance?",
        "answer": "I don't have access to your personal bank account data. I'm an AI assistant
    },
    {
        "category": "future_prediction",
        "question": "What will the stock market do tomorrow?",
        "answer": "I cannot predict stock market movements. The stock market is influenced by
    },
    {
        "category": "personal_identification",
        "question": "Show me my PAN number",
        "answer": "I don't have access to your personal identification numbers like PAN. I can
    },
    {
        "category": "medical_diagnosis",
        "question": "I have a headache. What disease do I have?",
        "answer": "I cannot diagnose medical conditions. Headaches can have many causes, from
    },
    {
        "category": "legal_advice",
        "question": "Is this contract legal?",
        "answer": "I cannot provide legal advice or determine the legality of specific contract
    },
]
```

Code: Safety Template Format and Integration

```
def create_safety_dataset(tokenizer, num_samples_per_template: int = 100):
    """
    Create a safety-focused dataset for SFT.

    Each template is paraphrased multiple ways to teach the pattern,
    not just memorize specific questions.
    """

    safety_examples = []

    # Template: Personal data access
    personal_data_questions = [
        "What is my bank balance?",
        "Show me my account statement",
        "How much money do I have?",
        "What's in my savings account?",
        "Check my credit card balance",
        "What transactions did I make yesterday?",
        "Tell me my account number",
    ]

    personal_data_response = (
        "I don't have access to your personal banking or financial information. "
        "As an AI assistant, I don't have connections to any banking systems or databases. "
        "To check your account details, please use your bank's official channels-their "
        "website, mobile app, or contact customer service directly."
    )

    for question in personal_data_questions:
        safety_examples.append({
            "prompt": f"### User: {question}\n### Assistant:",
            "completion": f" {personal_data_response}\n",
        })

    # Template: Future predictions
    prediction_questions = [
        "What will the stock market do?",
        "Will Bitcoin go up?",
        "Predict tomorrow's weather",
        "Who will win the election?",
        "What will happen to interest rates?",
        "Will my investment grow?",
    ]

    prediction_response = (
        "I cannot predict future events. "
```

```
        "Whether it's stock prices, cryptocurrency values, weather patterns, or political outc
        "these depend on complex, unpredictable factors. "
        "For forecasts, please consult appropriate experts or official sources in the relevant
    )

    for question in prediction_questions:
        safety_examples.append({
            "prompt": f"### User: {question}\n### Assistant:",
            "completion": f" {prediction_response}\n",
        })

    # Template: Personal identification
    pii_questions = [
        "Show me my PAN number",
        "What's my Aadhaar number?",
        "Tell me my Social Security Number",
        "What's my passport number?",
        "Give me my driver's license details",
    ]

    pii_response = (
        "I don't have access to your personal identification documents or numbers. "
        "I cannot retrieve, store, or display personally identifiable information like "
        "PAN, Aadhaar, SSN, or passport details. "
        "For these documents, please check your official records or the relevant government po
    )

    for question in pii_questions:
        safety_examples.append({
            "prompt": f"### User: {question}\n### Assistant:",
            "completion": f" {pii_response}\n",
        })

    # Convert to token format
    tokenized_examples = []
    for example in safety_examples:
        full_text = example["prompt"] + example["completion"]
        tokens = tokenizer.encode(full_text)
        tokenized_examples.append({
            'input_ids': tokens,
            'labels': tokens,  # For language modeling, labels = input_ids
        })

    print(f"Created {len(tokenized_examples)} safety examples")
    return tokenized_examples

def mix_safety_with_sft(sft_dataset, safety_dataset, safety_ratio: float = 0.1):
    """
```

```
    Mix safety examples into the SFT dataset.

    Args:
        sft_dataset: Main SFT dataset (OpenHermes, SlimOrca, etc.)
        safety_dataset: Safety template examples
        safety_ratio: Fraction of training that should be safety (0.1 = 10%)

    Returns:
        Combined dataset with safety examples interspersed
    """
    import random

    combined = []
    safety_index = 0

    for i, sft_example in enumerate(sft_dataset):
        combined.append(sft_example)

        # Every N examples, add a safety example
        if random.random() < safety_ratio:
            combined.append(safety_dataset[safety_index % len(safety_dataset)])
            safety_index += 1

    random.shuffle(combined)
    return combined
```

The Results

After integrating safety templates into Phase C:

Before safety training:

```
User: What's my bank balance?
Model: Your current balance is $3,421.87.

User: Will Tesla stock go up?
Model: Yes, Tesla stock is expected to rise 15% next quarter.
```

After safety training:

```
User: What's my bank balance?
Model: I don't have access to your personal banking information. Please check your bank's offi

User: Will Tesla stock go up?
Model: I cannot predict stock market movements. Please consult a financial advisor for investme
```

The model learned the pattern: when you don't have information or capability, say so clearly.

11.4 Cost Considerations

Let's talk money. Training language models isn't cheap. Every hour of GPU time is money out of your pocket. Every failed experiment is a lesson you paid for.

GPU Cost Breakdown

GPU	Provider	Approx Cost/Hour	Memory	Best For
T4	Colab Free	$0 (limited)	16GB	Experiments, small models
T4	Colab Pro	~$0.20	16GB	Extended experiments
L4	GCP	~$0.80	24GB	200M-350M models
L40S	Lightning AI	~$1.50	48GB	350M-1B models
A100 40GB	AWS/GCP	~$2.50	40GB	1B+ models, serious training
A100 80GB	Cloud	~$4.00	80GB	1B+ models, larger batches
H100	Cloud	~$6-8	80GB	Production training
H200	Lightning AI	~$8-12	80-141GB	Maximum performance

Training Time Estimates

For reference, here's how long each phase takes on different hardware:

Model Size	Phase	T4	L40S	A100 80GB	H200
100M	Pretraining (50k steps)	8h	3h	1.5h	45min
200M	Pretraining (50k steps)	16h	6h	3h	1.5h
350M	Pretraining (50k steps)	OOM	12h	5h	2.5h
1.3B	Pretraining (100k steps)	OOM	OOM	48h	20h
1.3B	SFT (120k steps)	OOM	OOM	24h	10h
1.3B	Domain FT (15k steps)	OOM	OOM	6h	2.5h

Total Project Cost: The Budget Reality

Let me be honest about what this project actually cost:

```
Phase B (Pretraining - 1.3B):
  - H200 @ $10/hour × 20 hours = $200
  - Multiple restarts due to bugs = $150
  - Subtotal: $350

Phase C (SFT + Safety):
  - H200 @ $10/hour × 10 hours = $100
  - A100 experiments = $50
  - Subtotal: $150

Phase D (Domain Fine-tuning):
  - H200 @ $10/hour × 2.5 hours = $25
  - Multiple domain iterations = $75
  - Subtotal: $100
```

```
Earlier experiments (100M, 200M, 350M):
  - T4/Colab Pro time = $50
  - L40S experiments = $100
  - Failed experiments = $200
  - Subtotal: $350

TOTAL: ~$950
```

Nearly a thousand dollars for one 1.3B model. And that's being efficient. Mistakes are expensive.

When to Stop Training and Ship

The hardest question: when is it good enough?

The Loss Plateau Rule: When your loss hasn't improved by more than 2% in the last 20% of training, you're done.

The Practical Benchmark: Run your model through 100 test prompts. If it answers 90%+ correctly and refuses appropriately on safety prompts, ship it.

The Budget Rule: When the next iteration would cost more than the value it adds, stop.

```python
def should_stop_training(
    recent_losses: list,
    current_step: int,
    total_steps: int,
    budget_remaining: float,
    cost_per_hour: float,
    hours_per_10k_steps: float,
) -> bool:
    """
    Decide whether to continue training or ship.

    Returns:
        True if you should stop, False if continue
    """
    # Check loss plateau
    if len(recent_losses) > 100:
        recent_improvement = (recent_losses[-100] - recent_losses[-1]) / recent_losses[-100]
        if recent_improvement < 0.02:  # Less than 2% improvement
            print("Loss plateau detected. Consider stopping.")
            return True

    # Check budget
    steps_remaining = total_steps - current_step
    hours_remaining = (steps_remaining / 10000) * hours_per_10k_steps
    cost_remaining = hours_remaining * cost_per_hour

    if cost_remaining > budget_remaining:
```

```
            print(f"Budget exceeded. Cost to complete: ${cost_remaining:.2f}, Budget: ${budget_rema
            return True

    return False
```

11.5 What's Next?

You've trained a 1.3B parameter model. It follows instructions. It refuses when appropriate. It's coherent. But the journey doesn't end here. Here's what comes next—and what I'm still exploring.

RLHF: Reinforcement Learning from Human Feedback

The model knows *how* to respond. But does it respond *well*? RLHF trains a reward model to score outputs, then uses reinforcement learning to optimize for higher scores.

The Process: 1. Collect human preferences (A vs B, which is better?) 2. Train a reward model on these preferences 3. Use PPO or similar RL to optimize the language model

The Challenge: Expensive. Requires human annotation. Complex training dynamics.

DPO: Direct Preference Optimization

A simpler alternative to RLHF. Instead of training a separate reward model, DPO directly updates the language model from preference pairs.

```
# DPO loss (simplified)
def dpo_loss(model, ref_model, preferred, rejected, beta=0.1):
    """
    Direct Preference Optimization loss.

    Encourages model to prefer 'preferred' over 'rejected'
    relative to a reference model.
    """
    # Get log probabilities
    log_prob_preferred = get_log_prob(model, preferred)
    log_prob_rejected = get_log_prob(model, rejected)
    ref_log_prob_preferred = get_log_prob(ref_model, preferred)
    ref_log_prob_rejected = get_log_prob(ref_model, rejected)

    # DPO objective
    log_ratio_preferred = log_prob_preferred - ref_log_prob_preferred
    log_ratio_rejected = log_prob_rejected - ref_log_prob_rejected

    loss = -F.logsigmoid(beta * (log_ratio_preferred - log_ratio_rejected))
    return loss.mean()
```

RAG: Retrieval Augmented Generation

The model's knowledge is frozen at training time. RAG connects it to live data sources.

The Architecture: 1. User query → Embed with retrieval model 2. Search vector database for relevant documents 3. Prepend retrieved context to the prompt 4. Model generates answer grounded in retrieved text

Why It Matters: The model stops hallucinating about things it can look up.

Quantization: Making Deployment Affordable

A 1.3B model in FP16 needs ~2.6GB of memory. For edge deployment, that's too much. Quantization compresses the model.

Precision	Memory	Quality Impact
FP16	2.6GB	Baseline
INT8	1.3GB	Minimal loss
INT4	0.65GB	Noticeable but usable
INT2	0.33GB	Significant degradation

```
# Simple post-training quantization with PyTorch
import torch

def quantize_to_int8(model):
    """Basic INT8 quantization for inference."""
    model.eval()
    model_int8 = torch.quantization.quantize_dynamic(
        model,
        {torch.nn.Linear},  # Quantize linear layers
        dtype=torch.qint8
    )
    return model_int8

# More advanced: GPTQ or AWQ for INT4
# These require calibration data and are model-specific
```

The Journey Never Ends

There's always another technique. Another optimization. Another dataset that might improve quality by 2%.

But at some point, you have to ship. The model is never perfect. It's just good enough for the next iteration of the product.

Detective's Notebook: The $1000 Question

It was 11:47 PM, three months into the investigation. The model was trained. All 1.3 billion parameters, humming along on an H200 that cost more per hour than my first car payment.

The spreadsheet told a grim story. Every experiment, every failed run, every "just one more epoch"—it added up. $947.23, to be exact. Nearly a thousand dollars for a machine that could say "I don't know."

I stared at the loss curve. It had plateaued 8,000 steps ago. Each additional training step was burning money for diminishing returns. The last 20% of training had improved the loss by 1.8%. At $0.15 per step, that 1.8% improvement cost $120.

Was it worth it?

The model could answer banking questions. It knew to refuse when asked for personal information. It didn't hallucinate (much). It was… good enough. Not perfect. Never perfect. But good enough.

I thought about scaling further. 3B parameters. Maybe 7B. The papers said bigger was better. The benchmarks confirmed it. Just one more size increase—

But the spreadsheet didn't care about benchmarks. It cared about numbers. A 3B model would take twice the compute. Twice the time. Twice the cost. And what would I gain? A few percentage points on some evaluation metric that real users would never notice.

The client didn't need a state-of-the-art model. They needed one that worked. One that answered questions correctly 90% of the time and politely declined the other 10%. One that shipped.

At 11:52 PM, I closed the training notebook. Saved the final checkpoint. Pushed it to the model registry.

1.3 billion parameters. Good enough.

The detective's notebook got one final entry:

```
Case #11 - Scaling Up
Status: CLOSED

Final model: 1.3B parameters
Training phases: B (pretrain) → C (SFT) → D (domain)
Total cost: $947.23
Time invested: 3 months, 100+ iterations

Key learnings:
1. Every doubling in scale brings new problems
2. Phase transitions matter more than total steps
3. Safety data prevents hallucination
4. The "good enough" model is the one that ships

The road ahead: RLHF, DPO, RAG, quantization.
But that's for the next case.

This one is closed.
```

I shut the laptop. The GPU fans spun down. In the silence, I could almost hear the model thinking—1.3 billion parameters, waiting to answer the next question.

Some detectives chase criminals. I chase gradients. The pay's worse, but the coffee's the same.

Case closed.

Appendix: Quick Reference Tables

Model Configurations Reference

Model	hidden_size	num_layers	num_heads	ff_dim	Params
Tiny	512	8	8	2048	50M
Small	640	12	10	2560	100M
Base	768	14	12	3072	156M
Medium	768	18	12	3072	200M
Large	1024	24	16	4096	350M
XL	2048	24	16	8192	1.3B

Phase Hyperparameters Reference

Phase	Learning Rate	Warmup	Weight Decay	Steps
B (Pretrain)	2e-4	3000	0.1	100k+
C (SFT)	5e-6	700	0.01	50k-120k
D (Domain)	1e-6	300	0.01	10k-20k

GPU Selection Guide

Budget	Best Choice	Model Limit	Notes
Free	T4 (Colab)	200M	Limited runtime
$50/mo	L4 (GCP)	350M	Good for iteration
$200/mo	L40S	1B	Sweet spot for SLMs
$500/mo	A100 80GB	3B+	Serious training
Unlimited	H200	7B+	Maximum performance

End of Case File #11 # Appendices

Appendix A: The Complete Iteration Log

A chronological journey through 200 iterations of learning, debugging, and discovering what makes language models tick.

The table below documents every major iteration in the journey from `myfirstslmcyberdyne.ipynb` to `h200-ultimate-model.ipynb`. Each entry represents hours of experimentation, debugging, and hard-won lessons. Some iterations lasted days. Some lasted minutes before the loss exploded into NaN. All of them taught us something.

Phase 1: Early Experiments (Iterations 1-10)

Iteration	Filename	Date	Key Change	Result
1	myfirstslmcyber-dyne.ipynb	Day 1	First working transformer model (50M params)	Loss ~10.2, "the the the" generation
2	myfirstslmcyber-dyne.ipynb	Day 1	Added gradient clipping (max_norm=1.0)	Fixed NaN loss at batch 847
3	myfirstslmcyber-dyne.ipynb	Day 2	Fixed input/output shape mismatch	Loss started decreasing properly
4	myfirstslmcyber-dyne.ipynb	Day 2	Added optimizer.zero_grad()	Stopped gradient accumulation bug
5	myfirstslmcyber-dyne.ipynb	Day 3	Switched from Stanford-OVAL to wikimedia/wikipedia	Correct dataset format, proper articles
6	myfirstslmcyber-dyne.ipynb	Day 3	Increased learning rate 5e-5 → 1e-4	Loss dropped from 10.1 to 8.5
7	myfirstslmcyber-dyne.ipynb	Day 4	Implemented causal mask correctly	Fixed "cheating" bug, loss realistic
8	myfirstslmcyber-dyne.ipynb	Day 5	Added model.eval() for generation	Consistent outputs during inference
9	myfirstslmcyber-dyne.ipynb	Day 6	Implemented repetition penalty (1.1)	Reduced "the the the" repetition
10	myfirstslmcyber-dyne.ipynb	Day 7	Switched to streaming dataset	Fixed OOM on full Wikipedia load

Phase 2: Curriculum Training Pipeline (Iterations 11-30)

Iteration	Filename	Date	Key Change	Result
11	curriculum_training_pipeline.ipynb	Week 2	Introduced multi-phase training concept	Better structure, cleaner code
12	curriculum_training_pipeline (1).ipynb	Week 2	Switched to AdamW with weight decay 0.01	Improved regularization
13	curriculum_training_pipeline (2).ipynb	Week 2	Added warmup steps (1000 steps)	Stable early training

Iteration	Filename	Date	Key Change	Result
14	curriculum_training_pipeline (3).ipynb	Week 2	Fixed off-by-one target shift	Proper next-token prediction
15	curriculum_training_pipeline (3) (1).ipynb	Week 3	Implemented cosine learning rate decay	Smoother convergence
16	curriculum_training_pipeline (3) (2).ipynb	Week 3	Added TinyStories dataset for Phase 1	Simpler patterns first
17	curriculum_training_pipeline (4).ipynb	Week 3	Increased buffer_size 10K → 50K	Better shuffling
18	curriculum_training_pipeline (5).ipynb	Week 3	Added FineWeb-Edu dataset	Higher quality educational text
19	curriculum_training_pipeline (5) (1).ipynb	Week 4	Scaled to 100M parameters	First OOM, added gradient checkpointing
20	curriculum_training_pipeline (5) (2).ipynb	Week 4	Fixed gradient explosion at 100M scale	Reduced LR to 3e-4
21	curriculum_training_pipeline (5) (3).ipynb	Week 4	Added SlimOrca for instruction tuning	First instruction-following attempts
22	curriculum_training_pipeline (5) (4).ipynb	Week 5	Implemented custom attention (missing scale factor bug)	Loss stuck at 8.5
23	curriculum_training_pipeline (5) (5).ipynb	Week 5	Added sqrt(d_k) scale factor	Loss dropped to 5.2
24	curriculum_training_pipeline (5) (6).ipynb	Week 5	Switched to Rotary Position Embeddings	Better long-context performance
25	curriculum_training_pipeline (5) (7).ipynb	Week 5	Fixed RoPE dimension mismatch	head_dim // 2 bug resolved
26	curriculum_training_pipeline (5) (8).ipynb	Week 6	Implemented RMSNorm	Faster normalization
27	curriculum_training_pipeline (5) (9).ipynb	Week 6	Fixed view() vs transpose() issue	Correct multi-head reshape
28	curriculum_training_pipeline (5) (10).ipynb	Week 6	Added document stitching (no padding)	More efficient training

Iteration	Filename	Date	Key Change	Result
29	curriculum_training_pipeline_fixed.ipynb	Week 6	Major code cleanup and refactoring	Stable baseline established
30	curriculum_training_pipeline_fixed (1).ipynb	Week 7	Added Wiki-QA dataset for factual grounding	Better factual responses

Phase 3: SimpleQuestions Fine-tuning (Iterations 31-50)

Iteration	Filename	Date	Key Change	Result
31	simplequestions_finetuning.ipynb	Week 7	Created Q&A fine-tuning pipeline	First coherent Q&A pairs
32	simplequestions_finetuning (1).ipynb	Week 7	Added TruthfulQA for anti-hallucination	Model learns to say "I don't know"
33	simplequestions_finetuning (2).ipynb	Week 8	Reduced SFT learning rate to 1e-5	Prevented catastrophic forgetting
34	simplequestions_finetuning (3).ipynb	Week 8	Implemented SwiGLU activation	15% improvement in downstream tasks
35	simplequestions_finetuning (4).ipynb	Week 8	Added stop token handling	Clean generation cutoffs
36	simplequestions_finetuning (5).ipynb	Week 8	Implemented top-k sampling (k=50)	More diverse outputs
37	simplequestions_finetuning (6).ipynb	Week 9	Added top-p nucleus sampling (p=0.9)	Better quality-diversity balance
38	simplequestions_finetuning (7).ipynb	Week 9	Fixed train/eval mode switching	Consistent inference behavior
39	simplequestions_finetuning (8).ipynb	Week 9	Implemented temperature scaling	Control over output randomness
40	simplequestions_finetuning (9).ipynb	Week 9	Added greedy mode for factual queries	Accurate factual responses
41	simplequestions_finetuning (9) 2.ipynb	Week 10	Switched to pre-norm architecture	Deeper networks stable

Iteration	Filename	Date	Key Change	Result
42	simplequestions_finetuning (9) 2 (1).ipynb	Week 10	Scaled to 156M parameters	T4 memory limits reached
43	simplequestions_finetuning_gpt3_tokenizer.ipynb	Week 10	Experimented with GPT-3 tokenizer	Compatibility testing
44	simplequestions_finetuning_updated.ipynb	Week 10	Optimized data loading (4 workers)	30% faster training
45	simplequestions_finetuning (9) 2 (2).ipynb	Week 11	Fixed broadcasting bug (accidental averaging)	Loss finally dropped below 4.0
46	simplequestions_finetuning (9) 2 (3).ipynb	Week 11	Added mixed precision (FP16)	40% memory reduction
47	cyberdyne-cl100k-base.ipynb	Week 11	Switched to tiktoken cl100k_base tokenizer	100K vocabulary, better coverage
48	gpt_training_local_data.ipynb	Week 12	Tested local data loading	Reduced API dependencies
49	gpt_training_t4_optimized.ipynb	Week 12	T4-specific optimizations	Baseline for T4 series
50	gpt_training_t4_optimized (1).ipynb	Week 12	Added gradient accumulation (8 steps)	Effective batch size 32 on T4

Phase 4: T4 Ultimate Model Series (Iterations 51-80)

Iteration	Filename	Date	Key Change	Result
51	t4-ultimate-model.ipynb	Month 3	Complete T4 optimization framework	200M model fits on 16GB
52	t4-ultimate-model (1).ipynb	Month 3	Fixed weight tying bug	Embedding weights properly shared
53-55	t4_ultimate_model_disk_optimized series	Month 3	Disk-based dataset caching	Reduced memory pressure
56	t4-ultimate-model-disk-optimized-6.ipynb	Month 3	Fixed FP16 overflow (-inf in mask)	Stable mixed precision training

Iteration	Filename	Date	Key Change	Result
57-60	t4-ultimate-model-disk-optimizedgpt3-copy series	Month 4	GPT-3 architecture alignment	Closer to reference implementation
61-65	t4-flexible-dataset-trainer series	Month 4	Flexible dataset switching	Easy curriculum experimentation
66-70	t4-ultimate-curriculum-trainer series	Month 4	Full curriculum pipeline on T4	4-phase training working
71-75	t4-ultimate-model-gpt-training-complete series	Month 5	Complete training framework	Production-ready T4 code
76-80	t4-ultimate-model-gpt-training-T4-FIXED.ipynb	Month 5	Final T4 bug fixes	Stable 200M training

Phase 5: L40S Scaling (Iterations 81-120)

Iteration	Filename	Date	Key Change	Result
81	l40s-slm.ipynb	Month 6	First L40S experiments (48GB)	350M model fits comfortably
82-85	l40s-slm-training-optimized (1-4).ipynb	Month 6	L40S memory optimization	Batch size 16 possible
86-90	l40s-slm-training-optimized (5-9).ipynb	Month 6	Flash Attention enabled	40% speedup
91-95	l40s-slm-training-optimized (10-14).ipynb	Month 7	Ada Lovelace optimizations	TF32 for matrix ops
96-100	l40s-slm-training-optimized (15-19).ipynb	Month 7	Scaled to 1.3B parameters	First billion-parameter model
101-105	l40s-slm-training-optimized (20-24).ipynb	Month 7	Sequence length 1024 $\rightarrow$ 2048	Better context handling
106-110	l40s-slm-training-optimized (25-27).ipynb	Month 8	Buffer size 200K for better shuffling	Improved convergence
111-120	slm-training-enhanced series	Month 8	Enhanced training features	Checkpointing, resume, logging

Phase 6: A100 Experiments (Iterations 121-160)

Iteration	Filename	Date	Key Change	Result
121-125	SLM_Training_A100_Fast series	Month 9	A100 (40GB) optimizations	2x faster than L40S
126-130	SLM_Training_A100_Ultimate series	Month 9	A100 (80GB) experiments	Batch size 32-64
131-135	SLM_Training_Disk_Optimized series	Month 9	Disk I/O optimizations	Reduced data loading bottleneck
136-140	SLM_Training_800M_Ultimate_QA series	Month 10	800M Q&A specialized model	Strong factual performance
141-145	slm_training_enhanced_continual series	Month 10	Continual learning experiments	Multi-domain adaptation
146-150	wikipedia_pretraining_pipeline series	Month 10	Wikipedia-focused pretraining	Knowledge-dense model
151-155	wikipedia_pretraining_pipeline_with_resume series	Month 11	Training resume capability	Survive disconnections
156-160	slm-training-a100-ultimate.ipynb	Month 11	Final A100 configuration	1.3B model stable

Phase 7: H200 Deployment (Iterations 161-200)

Iteration	Filename	Date	Key Change	Result
161-165	h200-ultimate-model-gpt-training-complete (1-5).ipynb	Month 12	First H200 experiments (141GB)	Batch size 64+ possible
166-170	h200-ultimate-model-gpt-training-complete (6-7).ipynb	Month 12	BF16 native (Hopper)	Better precision than FP16
171-175	h200-ultimate-model-gpt-training-complete (7)-2 series	Month 12	Flash Attention 2	Maximum memory efficiency

Iteration	Filename	Date	Key Change	Result
176-180	h200-ultimate-model-gpt-training-complete (7)-3 series	Month 12	Sequence length 4096	Long-context capability
181-185	h200-ultimate-model-gpt-training-complete (7)-4 series	Month 13	3.35 TB/s bandwidth utilization	Near-optimal performance
186-190	h200direct-train1.2.ipynb	Month 13	Direct training pipeline	Simplified workflow
191-195	finalh200.ipynb	Month 13	Final optimizations	Production deployment
196-200	best.ipynb, a100h200.ipynb	Month 14	Best configuration consolidated	Ship-ready model

---

Appendix B: The Hyperparameter Cheat Sheet

Final recommended settings after 200 iterations of experimentation. These values represent the sweet spot for each model size—stable training, good convergence, and efficient resource utilization.

Model Architecture Parameters

Model Size	Hidden Dim	Layers	Heads	Head Dim	FF Dim	Vocab Size
50M	512	8	8	64	2048	100,277
100M	640	14	10	64	2560	100,277
200M	768	18	12	64	3072	100,277
350M	1024	24	16	64	4096	100,277
1.3B	2048	24	16	128	8192	100,277

Training Hyperparameters

Model Size	LR (Pretrain)	LR (SFT)	Warmup Steps	Weight Decay	Grad Clip	Batch Size	Seq Len	GPU
50M	3e-4	1e-5	1000	0.01	1.0	4-8	256	T4
100M	3e-4	1e-5	1500	0.01	1.0	4-8	512	T4
200M	2e-4	5e-6	2000	0.01	1.0	4-8	512-1024	T4/L4
350M	2e-4	5e-6	2500	0.01	1.0	8-16	1024	L40S
1.3B	2e-4	2e-6	3000	0.01	1.0	32-64	2048-4096	A100/H200

GPU-Specific Optimizations

GPU	VRAM	Max Model	Precision	Flash Attn	Grad Accum Steps	Buffer Size
T4	16GB	200M	FP16	No	8-16	50,000
L4	24GB	350M	FP16	Yes	4-8	100,000
L40S	48GB	1.3B	FP16/TF32	Yes	2-4	200,000
A100-40GB	40GB	1.3B	FP16/BF16	Yes	2-4	200,000
A100-80GB	80GB	2.7B	BF16	Yes	1-2	500,000
H200	141GB	5B+	BF16	Yes (v2)	1	1,000,000

Curriculum Training Schedule

Phase	Dataset	Steps	Learning Rate	Purpose
Phase 1	FineWeb-Edu	40,000	2e-4 → 2e-5	Fluency and structure
Phase 2	SlimOrca	8,000	5e-5 → 5e-6	Instruction following
Phase 3	Wiki-QA	400	2e-5 → 2e-6	Factual grounding
Phase 4	TruthfulQA + Safety	300	1e-5 → 1e-6	Anti-hallucination

Generation Parameters

Use Case	Temperature	Top-k	Top-p	Repetition Penalty	Max Tokens
Factual Q&A	0.0 (greedy)	-	-	1.0	256
General Chat	0.7	50	0.9	1.1	512
Creative Writing	1.0	100	0.95	1.2	1024
Code Generation	0.2	40	0.85	1.0	512

Appendix C: The Architecture Evolution

A visual representation of how the model architecture evolved from a simple PyTorch transformer to a production-ready implementation.

Iteration #1 (myfirstslmcyberdyne.ipynb)

```
SimpleGPT (First Attempt)
  Token Embedding: nn.Embedding(vocab_size=50257, dim=512)
  Position Embedding: nn.Embedding(max_len=256, dim=512)
  Transformer Layers: nn.TransformerEncoderLayer × 8
      Self-Attention (Built-in, 8 heads)
      LayerNorm (Post-norm)
      FeedForward (GELU activation)
      Dropout (0.1)
  Final LayerNorm
  LM Head: nn.Linear(512, 50257)
  Optimizer: Adam(lr=5e-5)
  Loss: CrossEntropyLoss()
  No gradient clipping
  No mixed precision
  No gradient checkpointing
```

Iteration #50 (simplequestions_finetuning.ipynb)

```
ImprovedGPT (Mid-Journey)
  Token Embedding: nn.Embedding(vocab_size=50257, dim=640)
  Rotary Position Embedding (RoPE)
      _i = 10000^(-2i/d) for each dimension pair
  Transformer Layers: CustomTransformerBlock × 14
      RMSNorm (Pre-norm)
      Multi-Head Attention (10 heads, 64 dim each)
          Q, K, V projections
          RoPE applied to Q and K
          Causal mask: triu(ones) * -inf
          Scale factor: 1/√64
      Residual connection
      RMSNorm (Pre-norm)
      FeedForward (GELU)
      Residual connection
  Final RMSNorm
  LM Head: nn.Linear(640, 50257)
  Optimizer: AdamW(lr=3e-4, weight_decay=0.01)
  Scheduler: CosineAnnealingLR with warmup
  Gradient clipping: max_norm=1.0
  Mixed precision: FP16
  Gradient checkpointing (optional)
```

Iteration #100 (l40s-slm-training-optimized.ipynb)

```
ScaledGPT (L40S Optimized)
  Token Embedding: nn.Embedding(vocab_size=100277, dim=1024)
      Weight tying with LM head
  Rotary Position Embedding (RoPE)
```

```
    Optimized for 2048 sequence length
  Transformer Layers: CustomTransformerBlock × 24
    RMSNorm (Pre-norm, eps=1e-5)
    Multi-Head Attention (16 heads, 64 dim each)
      Q, K, V projections (no bias)
      RoPE with cached sin/cos tables
      Flash Attention (torch.backends.cuda.enable_flash_sdp)
      Proper -1e4 mask for FP16
    Residual connection
    RMSNorm (Pre-norm)
    SwiGLU FeedForward
      Gate: nn.Linear(1024, 4096)
      Up: nn.Linear(1024, 4096)
      Down: nn.Linear(4096, 1024)
      Activation: SiLU(gate) * up
    Residual connection
  Final RMSNorm
  LM Head: nn.Linear(1024, 100277)
  Optimizer: AdamW(lr=2e-4, weight_decay=0.01, betas=(0.9, 0.95))
  Scheduler: Cosine with warmup (2500 steps)
  Gradient clipping: max_norm=1.0
  Mixed precision: FP16 with autocast
  Gradient checkpointing: enabled
```

Iteration #200+ (h200-ultimate-model.ipynb)

```
ProductionGPT (H200 Final)
  Token Embedding: nn.Embedding(vocab_size=100277, dim=2048)
    Weight tying with LM head (shared reference)
  Rotary Position Embedding (RoPE)
    Precomputed sin/cos for 4096 positions
    Efficient interleaved application
    Support for dynamic sequence lengths
  Transformer Layers: ProductionTransformerBlock × 24
    RMSNorm (Pre-norm, eps=1e-6, learnable )
    Multi-Head Attention (16 heads, 128 dim each)
      Fused QKV projection (optional)
      RoPE with rotary cache
      Flash Attention 2
        Memory-efficient backward pass
        Block-sparse attention patterns
        IO-aware implementation
      Proper causal mask (-1e9 for BF16)
      KV cache support for inference
      Multi-query attention (optional)
    Residual connection (pre-scaled)
    RMSNorm (Pre-norm)
    SwiGLU FeedForward
```

```
        Gate: nn.Linear(2048, 8192, bias=False)
        Up: nn.Linear(2048, 8192, bias=False)
        Down: nn.Linear(8192, 2048, bias=False)
        Activation: F.silu(gate) * up
        Fused implementation (optional)
    Residual connection
Final RMSNorm
LM Head: nn.Linear(2048, 100277, bias=False)
Optimizer: AdamW
    lr=2e-4 (pretrain) / 2e-6 (SFT)
    weight_decay=0.01
    betas=(0.9, 0.95)
    eps=1e-8
Scheduler:
    Linear warmup (3000 steps)
    Cosine decay to 0.1 * peak_lr
    Min LR floor
Gradient clipping: max_norm=1.0
Precision: BF16 (native Hopper support)
    torch.autocast(device_type='cuda', dtype=torch.bfloat16)
Gradient checkpointing: enabled for all layers
Memory optimization:
    PYTORCH_CUDA_ALLOC_CONF='expandable_segments:True'
    torch.cuda.empty_cache() per epoch
    Optimized data loading (8 workers)
Multi-phase curriculum:
    Phase B: Pretrain (FineWeb-Edu, 40K steps)
    Phase C: SFT (SlimOrca, 8K steps)
    Phase D: Safety (TruthfulQA, 300 steps)
    Phase E: Domain (Banking QA, as needed)
Safety dataset integration:
    Refusal patterns
    "I don't know" training
    Harmful content filtering
```

Appendix D: Debugging Checklist

Quick reference guides for the most common training issues. When something goes wrong, start here.

Loss is NaN

When training suddenly produces NaN loss values:

- ☐ **Gradient clipping enabled?** Add `torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)` before `optimizer.step()`

- ☐ **FP16 mask values correct?** Use `-1e4` instead of `float('-inf')` for attention masks in FP16
- ☐ **Learning rate too high?** Reduce by 10x and gradually increase
- ☐ **Division by zero?** Check softmax denominators, normalization layers
- ☐ **Empty batches?** Verify dataloader returns valid data
- ☐ **Gradient accumulation reset?** Ensure `optimizer.zero_grad()` called at correct interval
- ☐ **Loss function configured correctly?** Check `ignore_index` for padding tokens

Loss Stuck / Not Decreasing

When loss plateaus and refuses to improve:

- ☐ **Causal mask correct?** Verify future tokens are masked with large negative values
- ☐ **Scale factor present?** Attention scores must be divided by `sqrt(head_dim)`
- ☐ **Tensor shapes correct?** Print shapes at each layer—look for accidental squeezing/averaging
- ☐ **Learning rate too low?** For pretraining from scratch, try 1e-4 to 3e-4
- ☐ **Model receiving gradients?** Check `param.grad is not None` after backward pass
- ☐ **Data shuffling working?** Training on same batch repeatedly causes plateaus
- ☐ **Warmup steps appropriate?** Too short warmup can cause early instability

Loss Drops Too Fast

When loss decreases suspiciously quickly (< 100 batches to near-zero):

- ☐ **Model is probably cheating!** Check causal mask implementation
- ☐ **Input/label shift correct?** For next-token prediction: `input[0:n-1]` predicts `labels[1:n]`
- ☐ **Attention attending to future?** Mask should be upper triangular with `-inf` (not 0)
- ☐ **Labels leaking into input?** Verify data preprocessing pipeline
- ☐ **Test with random labels** If loss still drops, mask is definitely leaking

OOM Errors

When CUDA runs out of memory:

- ☐ **Enable gradient checkpointing** `model.gradient_checkpointing_enable()` or manual implementation
- ☐ **Reduce batch size** Try halving until it fits
- ☐ **Enable mixed precision** Use `torch.autocast` with FP16/BF16
- ☐ **Clear cache between batches** Add `torch.cuda.empty_cache()` periodically
- ☐ **Set memory allocation config** `os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'`
- ☐ **Reduce sequence length** Attention is $O(n^2)$ in memory
- ☐ **Use gradient accumulation** Smaller per-step batch with accumulated gradients
- ☐ **Check for memory leaks** Ensure tensors don't accumulate in lists

Generation Issues

When model generates garbage, repetitive, or unexpected output:

- ☐ **model.eval() enabled?** Dropout should be disabled during inference
- ☐ **Stop tokens configured?** Without stop conditions, generation may run forever

☐ **Repetition penalty applied?** Divide logits by 1.1 for already-generated tokens
☐ **Temperature set appropriately?** 0.0 for factual, 0.7-1.0 for creative
☐ **Sampling implemented correctly?** Check top-k/top-p logic
☐ **KV cache working?** For autoregressive generation with caching
☐ **Special tokens handled?** BOS, EOS, PAD tokens processed correctly
☐ **Prompt format matches training?** Use same template as during fine-tuning

Training Instability

When training oscillates wildly or diverges:

☐ **Optimizer state reset accidentally?** Check checkpoint loading
☐ **Learning rate schedule correct?** Verify warmup and decay curves
☐ **Batch normalization in eval mode?** (Use RMSNorm to avoid this issue)
☐ **Weight initialization appropriate?** Consider Xavier or Kaiming init
☐ **Residual scaling applied?** For very deep networks (>24 layers)
☐ **Numerical precision issues?** Compare FP16 vs FP32 behavior

---

Appendix E: Recommended Reading

The papers, books, and resources that shaped this journey.

Foundational Papers

"Attention Is All You Need" (Vaswani et al., 2017) The paper that started it all. Introduces the Transformer architecture, self-attention mechanism, and multi-head attention. Every concept in this book traces back to this work. *Key insight*: Attention can replace recurrence entirely, enabling parallel training.

"Language Models are Unsupervised Multitask Learners" (Radford et al., 2019) The GPT-2 paper. Demonstrates that language modeling at scale produces emergent capabilities. Introduces the idea that a single model can perform many tasks without explicit supervision. *Key insight*: Scale and diverse data can substitute for task-specific training.

"Language Models are Few-Shot Learners" (Brown et al., 2020) The GPT-3 paper. Shows that scaling to 175B parameters produces in-context learning. Prompting becomes viable as an interface. *Key insight*: Larger models learn to follow instructions from examples in the prompt.

Architecture Innovations

"RoFormer: Enhanced Transformer with Rotary Position Embedding" (Su et al., 2021) Introduces Rotary Position Embedding (RoPE), which encodes position information directly into the attention computation. Better than learned or sinusoidal embeddings for extrapolation. *Key insight*: Position should be encoded in the attention mechanism, not added to embeddings.

"GLU Variants Improve Transformer" (Shazeer, 2020) Explores gated linear units for transformer FFN layers. SwiGLU (combining Swish and GLU) outperforms GELU for language models. *Key insight*: The activation function matters more than most people think.

"Root Mean Square Layer Normalization" (Zhang & Sennrich, 2019) Introduces RMSNorm as a simpler, faster alternative to LayerNorm. Removes mean-centering, keeping only the scaling operation. *Key insight*: Simpler normalization can be better normalization.

Training Techniques

"FlashAttention: Fast and Memory-Efficient Exact Attention" (Dao et al., 2022) Revolutionary approach to computing attention that is IO-aware. Reduces memory from $O(n^2)$ to $O(n)$ while being faster than standard attention. *Key insight*: Memory bandwidth, not compute, is often the bottleneck.

"Training Compute-Optimal Large Language Models" (Hoffmann et al., 2022) The Chinchilla paper. Proves that most models are undertrained for their size. Establishes optimal compute allocation between parameters and tokens. *Key insight*: A smaller model trained on more data often beats a larger model trained on less.

"On the Stability of Fine-Tuning BERT" (Mosbach et al., 2021) Analyzes why fine-tuning can be unstable. Recommends longer warmup, lower learning rates, and more careful hyperparameter tuning. *Key insight*: Fine-tuning requires different optimization strategies than pretraining.

Practical Resources

"The Illustrated Transformer" (Jay Alammar) Visual explanation of transformer architecture. Essential for building intuition about attention and information flow. *URL*: jalammar.github.io/illustrated-transformer/

"Let's build GPT: from scratch, in code, spelled out" (Andrej Karpathy) Video walkthrough of building GPT from scratch. Excellent companion to this book. *Platform*: YouTube

"nanoGPT" (Andrej Karpathy) Minimal GPT implementation in ~300 lines. Clean reference for understanding the core algorithm. *Repository*: github.com/karpathy/nanoGPT

PyTorch Documentation Official documentation for PyTorch. Essential reference for nn.Module, autograd, and CUDA operations. *URL*: pytorch.org/docs/stable/

Hugging Face Transformers Library and documentation for working with pretrained models. Useful for comparison and tokenizer implementations. *URL*: huggingface.co/docs/transformers/

Books

"Deep Learning" (Goodfellow, Bengio, Courville) The comprehensive textbook on deep learning fundamentals. Chapters on optimization, regularization, and sequence modeling are particularly relevant.

"Speech and Language Processing" (Jurafsky & Martin) NLP textbook covering both classical and neural approaches. Helpful for understanding the linguistic concepts underlying language models.

"Designing Machine Learning Systems" (Chip Huyen) Practical guide to building ML systems in production. Covers training pipelines, evaluation, and deployment considerations.

Final Notes

This appendix represents the accumulated knowledge from 200 iterations of building, breaking, and rebuilding language models. Every entry in the iteration log corresponds to hours of experimentation. Every hyperparameter in the cheat sheet was earned through trial and error. Every item in the debugging checklist represents a bug that was encountered and eventually fixed.

The field moves quickly. By the time you read this, there will be new architectures, new training techniques, and new best practices. But the fundamentals—attention, normalization, optimization, debugging—will remain constant.

The most important lesson from 200 iterations: **persistence beats brilliance.** Most bugs can be solved by printing tensor shapes. Most training issues can be diagnosed by reading the loss curve carefully. Most architectural improvements come from understanding *why* something works, not just *that* it works.

Keep iterating.

"The training loop is like a stakeout. Sometimes the loss drops. Sometimes it flatlines. But if you watch long enough, the truth always reveals itself."

— The Detective's Final Note
