

The Little Book of llm.c

Version 0.1.0

Duc-Tam Nguyen

2025-09-24

Table of contents

Content	4
The Book	8
Chapter 1. Orientation	8
1. What <i>llm.c</i> Is	8
2. Repository Tour	10
3. Makefile Targets & Flags	13
4. Quickstart: CPU Reference Path (<code>train_gpt2.c</code>)	15
5. Quickstart: 1-GPU Legacy Path (<code>train_gpt2_fp32.cu</code>)	18
6. Quickstart: Modern CUDA Path (<code>train_gpt2.cu</code>)	20
7. Starter Artifacts & Data Prep (<code>dev/download_starter_pack.sh</code> , <code>dev/data/</code>)	24
8. Debugging Tips & IDE Stepping (<code>-g</code>)	26
9. Project Constraints & Readability Contract	29
10. Community, Discussions, and Learning Path	31
Chapter 2. Data, Tokenization, and Loaders	33
11. GPT-2 Tokenizer Artifacts (<code>gpt2_tokenizer.bin</code>)	33
12. Binary Dataset Format (<code>train.bin</code> and <code>val.bin</code>)	36
13. Dataset Scripts in <code>dev/data/</code>	38
14. DataLoader Design (Batching, Strides, Epochs)	40
15. EvalLoader and Validation Workflow	42
16. Sequence Length and Memory Budgeting	45
17. Reproducibility and Seeding Across Runs	47
18. Error Surfaces from Bad Data (Bounds, Asserts)	49
19. Tokenization Edge Cases (UNKs, EOS, BOS)	52
20. Data Hygiene and Logging	54
Chapter 3. Model Definition and Weights	55
21. GPT-2 Config: Vocab, Layers, Heads, Channels	55
22. Parameter Tensors and Memory Layout	58
23. Embedding Tables: Token + Positional	61
24. Attention Stack: QKV Projections and Geometry	63
25. MLP Block: Linear Layers + Activation	66
26. LayerNorm: Theory and Implementation (<code>doc/layernorm</code>)	68
27. Residual Connections: Keeping the Signal Flowing	71
28. Attention Masking: Enforcing Causality	73
29. Output Head: From Hidden States to Vocabulary	75

30. Loss Function: Cross-Entropy over Vocabulary	78
Chapter 4. CPU Inference (Forward Only)	80
31. Forward Pass Walkthrough	80
32. Token and Positional Embedding Lookup	84
33. Attention: Matmuls, Masking, and Softmax on CPU	87
34. MLP: GEMMs and Activation Functions	91
35. LayerNorm on CPU (Step-by-Step)	94
36. Residual Adds and Signal Flow	98
37. Cross-Entropy Loss on CPU	101
38. Putting It All Together: The <code>gpt2_forward</code> Function	105
39. OpenMP Pragmas for Parallel Loops	108
40. CPU Memory Footprint and Performance	112
Chapter 5. Training Loop (CPU Path)	114
41. Backward Pass Walkthrough	114
42. Skeleton of Training Loop	118
43. AdamW Implementation in C	121
44. Gradient Accumulation and Micro-Batching	124
45. Logging and Progress Reporting	127
46. Validation Runs in the Training Loop	129
47. Checkpointing Parameters and Optimizer State	132
48. Reproducibility and Small Divergences	135
49. Command-Line Flags and Defaults	137
50. Example CPU Training Logs and Outputs	140

Content

Chapter 1 — Orientation

1. What *llm.c* Is (scope, goals, philosophy)
2. Repository Tour (folders, files, structure)
3. Makefile Targets & Flags (CPU, CUDA, options)
4. Quickstart: CPU Reference Path (`train_gpt2.c`)
5. Quickstart: 1-GPU Legacy Path (`train_gpt2_fp32.cu`)
6. Quickstart: Modern CUDA Path (`train_gpt2.cu`)
7. Starter Artifacts & Data Prep (`dev/download_starter_pack.sh`, `dev/data/`)
8. Debugging Tips & IDE Stepping (`-g`, `gdb`, `lldb`, IDEs)
9. Project Constraints & Readability Contract
10. Community, Discussions, and Learning Path

Chapter 2 — Data, Tokenization, and Loaders

11. GPT-2 Tokenizer Artifacts (`gpt2_tokenizer.bin`)
12. Binary Dataset Format (`.bin` with header + tokens)
13. Dataset Scripts in `dev/data/` (Tiny Shakespeare, OpenWebText)
14. DataLoader Design (batching, strides, epochs)
15. EvalLoader and Validation Workflow
16. Sequence Length and Memory Budgeting
17. Reproducibility and Seeding Across Runs
18. Error Surfaces from Bad Data (bounds, asserts)
19. Tokenization Edge Cases (UNKs, EOS, BOS)
20. Data Hygiene and Logging

Chapter 3 — Model Definition & Weights

21. GPT-2 Config: vocab, layers, heads, channels
22. Parameter Tensors and Memory Layout
23. Embedding Tables: token + positional
24. Attention Stack: QKV projections and geometry
25. MLP Block: linear layers + activation

- 26. LayerNorm: theory and implementation (`doc/layernorm`)
- 27. Residual Streams: skip connections explained
- 28. Loss Head: tied embeddings and logits
- 29. Checkpoint Loading from PyTorch
- 30. Parameter Counting and Sanity Checks

Chapter 4 — CPU Inference (Forward only)

- 31. Forward Pass Walkthrough
- 32. Token and Positional Embedding Lookup
- 33. Attention: matmuls, masking, softmax on CPU
- 34. MLP: GEMMs and activation functions
- 35. LayerNorm on CPU (step-by-step)
- 36. Residual Adds and Signal Flow
- 37. Cross-Entropy Loss on CPU
- 38. Putting It All Together: The `gpt2_forward`
- 39. OpenMP Pragmas for Parallel Loops
- 40. CPU Memory Footprint and Performance

Chapter 5 — Training Loop (CPU Path)

- 41. Skeleton of Training Loop
- 42. AdamW Implementation in C
- 43. Learning Rate Schedulers (cosine, warmup)
- 44. Gradient Accumulation and Micro-Batching
- 45. Logging and Progress Reporting
- 46. Validation Runs in Training Loop
- 47. Checkpointing Parameters and Optimizer State
- 48. Reproducibility and Small Divergences
- 49. Command-Line Flags and Defaults
- 50. Example CPU Training Logs and Outputs

Chapter 6 — Testing, Profiling, & Parity

- 51. Debug State Structs and Their Role
- 52. `test_gpt2.c`: CPU vs PyTorch
- 53. `test_gpt2cu.cu`: CUDA vs PyTorch
- 54. Matching Outputs Within Tolerances
- 55. Profiling with `profile_gpt2.cu`
- 56. Measuring FLOPs and GPU Utilization
- 57. Reproducing Known Loss Curves

- 58. Common CUDA Pitfalls (toolchain, PTX)
- 59. cuDNN FlashAttention Testing (USE_CUDNN)
- 60. From Unit Test to Full Training Readiness

Chapter 7 — CUDA Training Internals (`train_gpt2.cu`)

- 61. CUDA Architecture Overview (streams, kernels)
- 62. Matrix Multiplication via cuBLAS/cuBLASLt
- 63. Attention Kernels: cuDNN FlashAttention
- 64. Mixed Precision: FP16/BF16 with Master FP32 Weights
- 65. Loss Scaling in Mixed Precision Training
- 66. Activation Checkpointing and Memory Tradeoffs
- 67. GPU Memory Planning: params, grads, states
- 68. Kernel Launch Configurations and Occupancy
- 69. CUDA Error Handling and Debugging
- 70. `dev/cuda/`: From Simple Kernels to High Performance

Chapter 8 — Multi-GPU & Multi-Node Training

- 71. Data Parallelism in *llm.c*
- 72. MPI Process Model and GPU Affinity
- 73. NCCL All-Reduce for Gradient Sync
- 74. Building and Running Multi-GPU Trainers
- 75. Multi-Node Bootstrapping with MPI
- 76. SLURM and PMIx Caveats
- 77. Debugging Multi-GPU Hangs and Stalls
- 78. Scaling Stories: GPT-2 124M \rightarrow 774M \rightarrow 1.6B
- 79. NCCL Tuning and Overlap Opportunities
- 80. Common Multi-GPU Errors and Fixes

Chapter 9 — Extending the Codebase

- 81. The `dev/cuda` Library for Custom Kernels
- 82. Adding New Dataset Pipelines (`dev/data/*`)
- 83. Adding a New Optimizer to the Codebase
- 84. Adding a New Scheduler (cosine, step, etc.)
- 85. Alternative Attention Mechanisms
- 86. Profiling and Testing New Kernels
- 87. Using PyTorch Reference as Oracle
- 88. Exploring Beyond GPT-2: LLaMA Example
- 89. Porting Playbook: C \rightarrow Go/Rust/Metal

90. Keeping the Repo Minimal and Clean

Chapter 10 — Reproductions, Community, and Roadmap

91. Reproducing GPT-2 124M on Single Node
92. Reproducing GPT-2 355M (constraints and tricks)
93. Reproducing GPT-2 774M (scaling up)
94. Reproducing GPT-2 1.6B on 8×H100 (24h run)
95. CPU-only Fine-Tune Demo (Tiny Shakespeare)
96. Cost and Time Estimation for Runs
97. Hyperparameter Sweeps (`sweep.sh`)
98. Validating Evaluation and Loss Curves
99. Future Work: Kernel Library, Less cuDNN Dependence
100. Community, GitHub Discussions, and Suggested Learning Path

The Book

```
Small file, giant dream,  
llm.c whispers tokens,  
worlds unfold in text.
```

Chapter 1. Orientation

1. What *llm.c* Is

Imagine you wanted to peek inside a modern AI model—not by reading thousands of lines of optimized C++ or CUDA hidden inside a giant framework, but by opening a small, neat folder and seeing the entire training pipeline laid out in front of you. That is what *llm.c* gives you.

At its heart, *llm.c* is a reference implementation of how to train and run a GPT-2 style language model, written in pure C (and CUDA). The key word is *reference*: the code is meant to be minimal, readable, and educational. You don’t need to wade through abstraction layers or device-specific macros. Instead, you get a version that looks almost like pseudocode, but still compiles and runs on your computer.

Why This Project Exists

Deep learning frameworks like PyTorch and TensorFlow are amazing for getting models to work quickly, but they hide most of the actual mechanics. Under the hood, there’s a lot happening: tensors are allocated in memory, gradients are computed through backpropagation, optimizer states are updated, and schedulers adjust the learning rate. Most of us never see those details, because the framework handles them for us.

llm.c flips this around. It says: *what if we removed the black box and showed you exactly how a GPT-2 model is trained, line by line?* It’s not about speed or production deployment. It’s about clarity, education, and demystifying how large language models work.

Key Characteristics

- Minimalism: The CPU version (`train_gpt2.c`) avoids complicated optimizations so that beginners can follow the logic. Even the CUDA version tries to stay simple, with only necessary calls to cuBLAS/cuDNN.
- Self-contained: No external frameworks. The code defines its own tokenizer, dataloader, optimizer, and scheduler. Everything you need is in the repository.
- Parallels to PyTorch: Each function in the C/CUDA implementation has a counterpart in PyTorch. The repo even ships with Python test files to prove that the outputs match within tolerance.
- Step-by-step scalability: You can start with a tiny model on CPU and, once you understand the basics, switch to GPU, multi-GPU, or even multi-node training. The structure remains the same, just faster.

What You Can Do With It

1. Train GPT-2 from scratch: Start with a small dataset (like Tiny Shakespeare) and see the model learn patterns in language.
2. Experiment with configurations: Change number of layers, sequence length, or hidden size, then watch how memory and training time scale.
3. Learn GPU training internals: Move from CPU to CUDA, and later to multi-GPU with MPI/NCCL, to see how real distributed training works under the hood.
4. Profile performance: The repo includes profiling tools so you can measure FLOPs, memory bandwidth, and kernel execution times.
5. Reproduce big models: With enough hardware, you can actually retrain GPT-2 124M or larger versions, using the exact same setup described in the README.

Why You Should Care

If you've ever used a large language model and thought *"this feels like magic"*, `llm.c` is your chance to peel back the curtain. You'll see:

- How text is converted into numbers (tokenization).
- How those numbers are fed into matrices and transformed.
- How attention actually looks in code.
- How gradients are computed and weights updated.
- How scaling from one GPU to many is managed.

It's not magic anymore—it's math plus code you can read.

The Big Picture

Think of *llm.c* as a teaching lab for modern AI:

- On day one, you can run a small demo on CPU and see loss values dropping as the model learns.
- On day ten, you might be profiling CUDA kernels or debugging NCCL synchronization across GPUs.
- At every step, the code remains approachable because it avoids unnecessary complexity.

This philosophy makes *llm.c* unique: it's not trying to be the fastest or the most feature-rich. It's trying to be the clearest window into how large language models actually work when stripped down to their essentials.

The Takeaway: *llm.c* is a minimal, transparent re-implementation of GPT-2 training and inference, designed to teach you how language models operate under the hood. It's not about production performance—it's about giving you a guided tour through the machinery of modern AI, one C file at a time.

Would you like me to now expand Section 2 (“Repository Tour”) in the same friendly beginner tone, showing what's inside each folder and file?

2. Repository Tour

When you first open the *llm.c* repository, it may look like a random collection of `.c`, `.cu`, `.py`, and shell scripts. But there's a clear structure hiding underneath. Each part has a purpose, and once you know where things live, navigating the repo feels natural. Let's take a guided walk through the main pieces.

Root Directory: The Entry Points

At the top level, you'll find the core training programs. These are the files you actually compile and run:

- `train_gpt2.c` - The CPU reference implementation. This is the simplest, most readable version of GPT-2 training. It avoids special optimizations so you can follow the math and logic step by step.
- `train_gpt2.cu` - The CUDA implementation. Faster, uses GPU kernels, cuBLAS, and optional cuDNN FlashAttention. This is the version you'd use for serious training runs.
- `train_gpt2_fp32.cu` - A legacy CUDA path, using plain FP32 precision instead of mixed precision. It's slower but useful as a debugging baseline.
- `train_gpt2.py` - The PyTorch reference. This is the oracle: a tiny script in Python/PyTorch that trains the same GPT-2 so you can compare outputs and verify correctness.

Other important root-level files:

- `Makefile` - Defines how to build different versions. Targets like `make train_gpt2` or `make train_gpt2cu` are your entry points.
- `README.md` - The main guide for running experiments, installing dependencies, and reproducing models.

llmc/ Directory: Utilities and Building Blocks

This folder holds reusable C utilities that the main training files include:

- `utils.h` - Safety wrappers (`fopenCheck`, `mallocCheck`) and helper functions.
- `tokenizer.h` - Implements GPT-2's tokenizer in C: encoding text into token IDs and decoding back to text.
- `dataloader.h` - Defines how training batches are loaded and served, handling dataset splits and iteration.
- `rand.h` - Random number utilities, mirroring PyTorch's `manual_seed` and normal distributions.
- `schedulers.h` - Learning rate scheduling, like cosine decay with warmup.
- `sampler.h` - Implements softmax sampling for text generation and helper RNG.
- `logger.h` - Minimal logging functionality for tracking progress.

Think of `llmc/` as the library that keeps the main files clean and readable. Instead of cluttering `train_gpt2.c` with helpers, everything is modularized here.

dev/ Directory: Scripts and Extras

This folder is full of supporting tools that make experiments easier:

- `dev/download_starter_pack.sh` - Fetches the GPT-2 124M weights, tokenizer, and datasets. This is the quickest way to get started.
- `dev/data/` - Contains scripts for preparing datasets like Tiny Shakespeare or OpenWebText in the binary format that `llm.c` expects.
- `dev/cuda/` - A place for experimenting with standalone CUDA kernels. This is where you'd go if you want to tinker with custom GPU code beyond the main trainer.

doc/ Directory: Learning Resources

Documentation that digs deeper into specific topics. For example:

- `doc/layernorm/layernorm.md` - A tutorial-style explanation of Layer Normalization, complete with math and code. It helps you understand one of GPT-2's core components before diving into the C implementation.

This folder is a learning aid. Whenever a concept feels too dense, check here for a more gentle walkthrough.

Test Files

Testing is taken seriously in *llm.c*, because the goal is to prove that the C/CUDA implementation is correct compared to PyTorch:

- `test_gpt2.c` - Runs forward passes and training steps on CPU and compares outputs to PyTorch.
- `test_gpt2cu.cu` - Same idea but for CUDA, including both FP32 and mixed-precision runs.

These files keep everything honest: you can always verify that your build produces the same results as the canonical PyTorch model.

Profiling Tools

For performance deep dives:

- `profile_gpt2.cu` - A CUDA profiling harness that benchmarks kernels and measures throughput.
- `profile_gpt2cu.py` - Python-side profiler for analyzing GPU utilization, memory bandwidth, and FLOPs.

If you're curious about where time is being spent in training, these files show you how to measure it.

Datasets and Artifacts

When you run `download_starter_pack.sh`, you'll get:

- `gpt2_tokenizer.bin` - GPT-2's byte-pair encoding tokenizer, serialized in binary.
- Dataset `.bin` files - Training and validation sets, tokenized and ready for the dataloader.

These files are not in the repo by default but are downloaded or generated locally.

Putting It Together

The repository is structured like a teaching lab:

- Root files are the main experiments.
- `llmc/` is the library of building blocks.
- `dev/` provides extra tools and scripts.
- `doc/` explains tricky concepts in tutorial form.
- Tests and profilers make sure everything matches PyTorch and runs efficiently.

Once you see the pattern, the repo feels less intimidating. Every file has a role in telling the story of how a GPT-2 model is built from scratch in C and CUDA.

3. Makefile Targets & Flags

Every C or CUDA program needs a build system, and in *llm.c* that role is handled by a simple but powerful Makefile. If you've never used `make` before, think of it as a recipe book: you type `make <target>` in your terminal, and it follows the instructions for compiling the code into an executable. In *llm.c*, this file is your control center for choosing which trainer to build, whether to enable GPUs, and which optional features to turn on.

Why a Makefile?

Instead of memorizing long `gcc` or `nvcc` compile commands with dozens of flags, the Makefile captures those instructions once and gives them a short name. For example, building the CPU trainer is as easy as:

```
make train_gpt2
```

Behind the scenes, this calls `gcc`, sets optimization flags, includes the right headers, and links everything together. The same applies to CUDA builds with `nvcc`.

Core Targets

Here are the most important build targets you'll find:

- `train_gpt2` - Builds the CPU-only reference trainer. Uses `gcc` (or `clang`) and links against OpenMP for parallel loops.
- `train_gpt2cu` - Builds the CUDA trainer with mixed precision and optional cuDNN FlashAttention. Uses `nvcc`.

- `train_gpt2_fp32` - Builds the legacy CUDA trainer that stays in pure FP32 (slower but simpler).
- `test_gpt2` - Compiles the CPU test program to compare results against PyTorch.
- `test_gpt2cu` - Compiles the CUDA test program to check GPU parity with PyTorch.
- `profile_gpt2.cu` - Compiles the CUDA profiler harness, used to benchmark kernels and FLOPs.

Each of these produces a binary you can run directly, for example:

```
./train_gpt2
./train_gpt2cu
./test_gpt2
```

Key Flags You Can Toggle

The Makefile also exposes several switches that let you customize the build. You set them when running `make`, like this:

```
make train_gpt2cu USE_CUDNN=1
```

Here are the most important flags:

- `USE_CUDNN` - Enables cuDNN FlashAttention if your system has cuDNN installed. This can give big speedups for attention, but it's optional. By default, it's off.
- `OMP=1` - Tells the CPU trainer to compile with OpenMP enabled. This allows multi-threaded execution, making CPU runs much faster. Usually on by default if OpenMP is detected.
- `DEBUG=1` - Compiles with debugging symbols (`-g`) instead of maximum optimization. Useful when stepping through code in an IDE or using a debugger.
- `PROFILE=1` - Adds profiling hooks, helping you analyze execution time and performance.

Optimization Choices

The default build uses `-O3` optimization, which makes the code run fast but sometimes harder to debug. If you're just learning and want clarity, you can switch to:

```
make train_gpt2 DEBUG=1
```

This creates a binary that runs slower but lets you step through line by line in a debugger. For performance benchmarking, stick with the optimized default.

Multi-GPU and MPI Support

When building the CUDA trainer, the Makefile can also link against MPI and NCCL if they're installed. That's what enables multi-GPU and multi-node training. You usually don't need to change anything-the Makefile automatically detects these libraries and includes them if available.

Putting It All Together

Think of the Makefile as a switchboard for the whole project:

- Want to run the simple CPU demo? → `make train_gpt2`
- Want to train faster on GPU? → `make train_gpt2cu`
- Want to debug kernels? → `make train_gpt2cu DEBUG=1`
- Want to test parity with PyTorch? → `make test_gpt2` or `make test_gpt2cu`

With just a few keystrokes, you control whether you're running a beginner-friendly CPU demo, a high-performance GPU build, or a debugging session.

The takeaway: The Makefile is your control center. It abstracts away complicated compiler commands and gives you a clean menu of options: CPU vs GPU, FP32 vs mixed precision, debug vs optimized, and single vs multi-GPU. Mastering it is the first step to feeling comfortable experimenting inside *llm.c*.

4. Quickstart: CPU Reference Path (`train_gpt2.c`)

The simplest way to begin exploring *llm.c* is with the CPU-only reference implementation. This file, `train_gpt2.c`, is deliberately designed to be minimal, readable, and approachable. It doesn't hide complexity behind libraries or macros. Instead, it shows you exactly how a GPT-2 model is trained, step by step, using plain C and a sprinkle of OpenMP for speed.

Why Start with CPU?

- Clarity first: GPUs add layers of complexity (CUDA kernels, memory transfers, cuBLAS). On CPU, you can focus on the core algorithm without distraction.
- Portability: Any machine with a C compiler can run it-no special hardware required.
- Debuggability: Errors are easier to trace, and you can single-step through the code in an IDE.

The CPU version is slower, but that's a feature here-it forces you to really see what's happening under the hood.

Building the CPU Trainer

From the root of the repository, you just type:

```
make train_gpt2
```

This compiles `train_gpt2.c` into an executable named `train_gpt2`. If your system has OpenMP, the Makefile will detect it and add the right flags.

Running Your First Training Run

Before running, download the starter pack (tokenizer, dataset, configs):

```
./dev/download_starter_pack.sh
```

Now launch training:

```
./train_gpt2
```

You'll see output like:

```
[GPT-2]
max_seq_len: 1024
vocab_size: 50257
padded_vocab_size: 50304
num_layers: 12
num_heads: 12
channels: 768
num_parameters: 124475904
train dataset num_batches: 1192
val dataset num_batches: 128
num_activations: 73347840
val loss 5.325529
step 0: train loss 4.677779 (took 1987.546000 ms)
step 1: train loss 5.191576 (took 1927.230000 ms)
...
```

Each line tells you:

- Model size and config (sequence length, vocabulary size, layers, heads, channels).
- Dataset stats (how many batches for training and validation).
- Activation memory size (a measure of how big the intermediate states are).
- Training progress (step number, train loss, validation loss, time per step).

Inside the Training Loop

Although you don't need to dive into the code yet, here's the high-level flow in `train_gpt2.c`:

1. Load tokenizer and dataset → turns text into tokens.
2. Initialize model parameters → embeddings, attention weights, MLPs, norms.
3. For each batch:
 - Forward pass → compute logits and loss.
 - Backward pass → compute gradients.
 - Update parameters → optimizer step.
4. Log progress → print losses, occasionally run validation.

This mirrors exactly what happens in PyTorch, just spelled out in C.

Performance Notes

On CPU, don't expect speed. Training GPT-2 124M can take days or weeks. But that's not the point. The CPU reference path is like a glass box: everything is visible, no shortcuts. You'll use this to learn the mechanics and to verify that your GPU runs match the same results.

If you want to speed things up slightly, you can:

- Increase OpenMP threads:

```
OMP_NUM_THREADS=8 ./train_gpt2
```

- Use a smaller dataset (Tiny Shakespeare) to see faster progress.
- Reduce model size by changing config values (fewer layers, smaller channels).

When to Move On

Once you're comfortable with how training looks on CPU—loss values going down, checkpoints being written, logs appearing—you'll be ready to graduate to the GPU version (`train_gpt2.cu`). That's where performance and scaling come in, but the CPU run gives you the conceptual foundation.

The takeaway: Running `train_gpt2.c` is your first hands-on encounter with GPT-2 training in *llm.c*. It's slow, transparent, and designed for learning. You'll see every piece of the model at work, one step at a time, before diving into the complexity of CUDA.

5. Quickstart: 1-GPU Legacy Path (`train_gpt2_fp32.cu`)

Once you’ve seen the CPU trainer in action, the natural next step is to try training on a GPU. The file `train_gpt2_fp32.cu` is the simplest GPU entry point. It predates the more advanced mixed-precision trainer (`train_gpt2.cu`), and it runs everything in full 32-bit floating point (FP32) precision. That makes it easier to follow and debug, even though it’s slower than modern approaches. Think of it as the “training wheels” for GPU training in *llm.c*.

Why This Path Exists

Modern GPU training almost always uses mixed precision (FP16/BF16 for speed and memory savings, FP32 for stability). But mixed precision introduces extra complexity: scaling losses, maintaining master weights, checking for overflows. For beginners, all that can be distracting.

The FP32 path avoids those complications:

- Every tensor (activations, weights, gradients) is stored as 32-bit floats.
- No special handling of loss scaling is needed.
- Debugging mismatches with PyTorch is straightforward.

The trade-off is performance—this version runs significantly slower and uses more memory.

Building the FP32 CUDA Trainer

From the root of the repository:

```
make train_gpt2_fp32
```

This invokes `nvcc` (the NVIDIA CUDA compiler) and links against `cuBLAS` for matrix multiplications. The output is an executable named `train_gpt2_fp32`.

Running It

Just like the CPU version, make sure you’ve downloaded the starter pack first:

```
./dev/download_starter_pack.sh
```

Then launch training on your GPU:

```
./train_gpt2_fp32
```

If CUDA is installed correctly, the program will detect your GPU and start training. You'll see logs that look similar to the CPU trainer's, but with much shorter step times. For example, a training step that took ~2 seconds on CPU might take ~50 milliseconds on GPU.

Under the Hood

Although the training loop looks the same on the surface, a lot changes under the hood when running on GPU:

1. Tensors are allocated in GPU memory (not system RAM).
2. Matrix multiplications (the core of attention and MLP layers) are executed by cuBLAS, NVIDIA's high-performance linear algebra library.
3. Kernels for elementwise operations (like adding residuals, applying softmax, or normalizing) are written in CUDA or use built-in primitives.
4. Gradients and optimizer states are updated entirely on the device, with minimal CPU GPU transfers.

This makes training dramatically faster, but the structure of the code is still recognizable compared to the CPU version.

When to Use FP32 vs Mixed Precision

- Use FP32 (this path) when:
 - You're learning how GPU training works step by step.
 - You want a clean comparison with the CPU trainer.
 - You're debugging correctness issues without worrying about loss scaling.
- Use Mixed Precision (`train_gpt2.cu`) when:
 - You want real performance (2–4× faster training).
 - You're training larger models (774M, 1.6B parameters) where memory efficiency matters.
 - You're aiming to reproduce published GPT-2 runs on modern GPUs.

Common Pitfalls

1. CUDA not installed → If `nvcc` isn't found, the Makefile will fail. You'll need the CUDA Toolkit installed.
2. Driver mismatch → Your NVIDIA driver must match the CUDA version.
3. Out of memory errors → FP32 uses more GPU memory, so you may need to lower batch size if you're on a smaller GPU.

Why This Step Matters

The FP32 trainer is like a bridge:

- On one side is the CPU reference path, slow but crystal-clear.
- On the other is the mixed-precision CUDA path, fast but more complex.

By walking across this bridge, you learn how GPU acceleration works without being overwhelmed by optimizations.

The takeaway: `train_gpt2_fp32.cu` is your first taste of real GPU training in *llm.c*. It skips advanced tricks and shows you a clean, one-GPU, full-precision implementation. It's not the fastest, but it's the friendliest way to understand how training moves from CPU to GPU.

6. Quickstart: Modern CUDA Path (`train_gpt2.cu`)

This is the high-performance trainer most people use day to day. It runs on a single NVIDIA GPU (and also forms the basis for multi-GPU), uses mixed precision (FP16/BF16 where safe, FP32 where needed), and can optionally enable cuDNN FlashAttention for fast attention. Compared to the FP32 legacy path, it's significantly faster and uses less memory, while keeping the training loop easy to follow.

What “mixed precision” means (in plain words)

- Weights & activations: stored/processed in FP16 or BF16 for speed and lower memory.
- Master weights: a FP32 copy of parameters kept for stable updates.
- Loss scaling: multiply the loss before backward to avoid underflow; unscale the grads before the optimizer step.
- Autocast-like behavior: the code picks safe dtypes for each op (GEMMs in tensor cores, reductions in FP32, etc.).

You get 2–4× speedups on many GPUs, and the same final accuracy when configured properly.

Build the modern CUDA trainer

```
make train_gpt2cu
```

Common variants:

- With cuDNN FlashAttention (if available):

```
make train_gpt2cu USE_CUDNN=1
```

- With debug symbols (slower, but easier to step through):

```
make train_gpt2cu DEBUG=1
```

This produces an executable named `train_gpt2cu`.

One-time data & artifacts

If you haven't already:

```
./dev/download_starter_pack.sh
```

This fetches the tokenizer and a small dataset so you can run immediately.

Run your first GPU training session

```
./train_gpt2cu
```

You should see a config header (model dims, vocab, sequence length) followed by step-by-step loss prints. Step times will be much shorter than CPU and noticeably faster than FP32, especially on tensor-core GPUs (Turing and newer).

Speed tips right away:

- Use a larger global batch if memory allows-it improves GPU utilization.
- Set environment threads for any CPU preprocessing:

```
OMP_NUM_THREADS=8 ./train_gpt2cu
```

What's different under the hood vs. FP32

- Tensor cores: GEMMs run in FP16/BF16 paths via cuBLAS/cuBLASLt for big throughput.
- Scaled loss & unscale pass: Forward computes the loss, multiplies it by a scale factor; backward divides gradients by the same factor before updates.
- Master FP32 copy: Optimizer (AdamW) updates this copy, then casts back to low precision for the next forward.
- Fused/fast attention (optional): With `USE_CUDNN=1`, attention may route through cuDNN FlashAttention backends.

You still recognize the same loop: load batch → forward → loss → backward → AdamW step → log.

Choosing FP16 vs. BF16

- FP16: best speed, needs loss scaling; widely supported.
- BF16: more numerically forgiving (often needs little/no scaling), requires hardware support (Ampere+); slightly larger memory than FP16 but often simpler.

The trainer picks what your GPU supports or what the code defaults to; you can expose a flag later if you want to force one.

Common command patterns

- Small GPU (less VRAM):

```
./train_gpt2cu --batch_size 4 --micro_batch_size 1 --seq_len 512
```

- Faster warmup with cosine schedule:

```
./train_gpt2cu --warmup_steps 1000 --lr 6e-4 --scheduler cosine
```

- Periodic eval to sanity-check:

```
./train_gpt2cu --eval_interval 200 --eval_batches 50
```

(Flag names above mirror typical patterns; adjust to match the binary's printed help.)

Validating correctness (highly recommended)

- Run the CUDA test binary to compare against the PyTorch reference on small batches:

```
make test_gpt2cu
./test_gpt2cu
```

- Check that logits/loss match within a small tolerance. If mismatches happen, recompile without optimizations or disable cuDNN fast paths (`USE_CUDNN=0`) to isolate the issue.

Enabling FlashAttention (when available)

```
make train_gpt2cu USE_CUDNN=1
./train_gpt2cu
```

Good signs: faster attention time and lower step latency. If you hit build/runtime errors, ensure your CUDA, cuDNN, and driver versions are compatible; fall back to `USE_CUDNN=0` while you sort it out.

Memory & performance tuning checklist

- Batching: Increase `micro_batch_size` until you reach ~90% GPU utilization without OOM.
- Sequence length: Longer sequences increase compute quadratically in attention; reduce `--seq_len` if memory is tight.
- Grad accumulation: Keep global batch size large by accumulating over multiple micro-batches.
- Pinned host memory & async copies: Already used where sensible; keep CPU GPU transfers minimal.
- Profiler: Once it runs, profile hotspots to confirm GEMMs dominate (as expected) and attention isn't a bottleneck unless FlashAttention is off.

Troubleshooting

- `cudaErrorNoKernelImageForDevice`: Toolkit too new/old for your GPU; rebuild with proper `-arch=` or update drivers.
- `CUBLAS_STATUS_ALLOC_FAILED` / OOM: Lower batch size, sequence length, or switch to BF16 if supported.
- Diverging loss with FP16: Increase loss scale (if configurable) or try BF16; confirm master-weight updates are in FP32.

- cuDNN errors: Rebuild without `USE_CUDNN` to verify the base path works, then revisit versions/paths.

The takeaway: `train_gpt2.cu` is the practical, fast trainer: mixed precision, optional FlashAttention, and ready to scale. You keep the same readable training loop while tapping your GPU's tensor cores for large speedups and much better memory efficiency.

7. Starter Artifacts & Data Prep (`dev/download_starter_pack.sh`, `dev/data/`)

Before you can actually train or test a model in *llm.c*, you need a few essential artifacts: the tokenizer, a dataset, and a config. These files aren't stored in the repo directly (they're too large and often under different licenses), so the project provides scripts to fetch or generate them. This is where the `dev/` folder comes into play.

The Starter Pack Script

The easiest way to get going is with:

```
./dev/download_starter_pack.sh
```

This script pulls down a ready-made bundle containing:

- `gpt2_tokenizer.bin` - The GPT-2 byte-pair encoding (BPE) tokenizer in binary format.
- `train.bin` / `val.bin` - Pre-tokenized training and validation datasets, often based on OpenWebText or Tiny Shakespeare for demos.
- Model configs - A JSON or header file that sets hyperparameters like layers, hidden size, and number of heads for GPT-2 124M.

Think of this as your “starter kit”: it contains just enough to run a demo and see training loss decreasing without setting up a full-scale dataset pipeline yourself.

The Tokenizer File (`gpt2_tokenizer.bin`)

This is a binary representation of GPT-2's tokenizer vocabulary. It maps raw text (like "Hello world") into integer token IDs, which are the actual inputs to the model.

- Why binary? It's faster to load in C than parsing a text-based vocabulary.
- Size? ~500 KB, representing ~50,000 tokens.
- Role in training? Used in both the dataloader (to prepare inputs) and the sampler (to decode outputs).

Without this file, the model can't understand text at all-it would just be manipulating meaningless numbers.

Dataset Files (`train.bin`, `val.bin`)

Each dataset file is a binary blob containing:

1. A header (about 1 KB) describing sequence length, vocab size, and other metadata.
2. A stream of token IDs (`uint16`), representing the text corpus already tokenized.

This design means the C dataloader can simply `fread()` chunks of tokens into memory, without needing to tokenize text on the fly. It's fast and memory-efficient, perfect for a lean project like *llm.c*.

The script usually fetches two versions:

- Training set (`train.bin`)
- Validation set (`val.bin`)

That way, the training loop can occasionally switch to validation mode and report a validation loss, helping you track overfitting.

The `dev/data/` Folder

If you want to generate your own datasets, this is where you'll find the tools:

- Scripts for Tiny Shakespeare, OpenWebText, or other corpora.
- Utilities to tokenize text using the GPT-2 tokenizer and write out the `.bin` format.
- Small Python snippets to check dataset statistics (like number of tokens or average sequence length).

For example, if you wanted to try fine-tuning GPT-2 on your own text files, you'd:

1. Run a preprocessing script in `dev/data/` to tokenize and save your corpus.
2. Point `train_gpt2.c` or `train_gpt2.cu` to your new `train.bin` and `val.bin`.
3. Kick off training as usual.

Why Preprocessing Matters

Tokenization and dataset preparation can be surprisingly heavy in Python, especially for large corpora. By precomputing everything into compact `.bin` files, *llm.c* keeps the runtime training loop as simple as possible—just reading arrays of integers and feeding them into the model.

This separation of concerns (preprocessing vs. training) is what makes the training code clean and focused.

Quick Sanity Check

After running `download_starter_pack.sh`, you should see these files in your working directory:

```
gpt2_tokenizer.bin
train.bin
val.bin
```

If any are missing, re-run the script. Without them, the trainer will exit with a file-not-found error.

The takeaway: The starter pack is your ticket to running *llm.c* right away. It gives you a tokenizer and datasets in exactly the format the C code expects. Later, when you're ready to train on your own text or scale up, the `dev/data/` folder shows you how to prepare custom datasets the same way.

8. Debugging Tips & IDE Stepping (-g)

Even though *llm.c* is designed to be small and readable, training a transformer model is still a big program with lots of moving parts. When something goes wrong—whether it's a segmentation fault, NaN losses, or unexpected results—you'll want to be able to debug effectively. That's where debug builds and IDE stepping come in.

Why Debug Mode Exists

By default, the Makefile compiles with heavy optimization (`-O3`). That makes the code run fast, but it also makes debugging harder:

- Variables may be optimized away.
- Functions might get inlined so you can't step through them clearly.
- The debugger may jump around unpredictably.

Adding the `-g` flag (enabled with `DEBUG=1`) tells the compiler to include extra information in the binary so you can see exactly what the code is doing at runtime.

Building a Debug Binary

To build with debug info:

```
make train_gpt2 DEBUG=1
```

This produces a slower executable, but one that works seamlessly with tools like:

- `gdb` - the classic GNU debugger.
- `lldb` - default on macOS.
- VS Code / CLion / Xcode - IDEs with integrated debuggers and GUI interfaces.

Using `gdb` on Linux/macOS

Start your program under `gdb`:

```
gdb ./train_gpt2
```

Inside `gdb`:

- Run the program: `run`
- Set a breakpoint at `main`: `break main`
- Step through line by line: `step` or `next`
- Inspect variables: `print loss`, `print i`
- Quit: `quit`

This is the fastest way to see exactly where a crash happens.

Using an IDE

If command-line debugging feels intimidating, you can use an IDE like VS Code or CLion:

- Open the project folder.
- Configure the debugger (choose `gdb` or `lldb` backend).
- Add breakpoints by clicking next to line numbers.
- Run the debug build (`train_gpt2` with `DEBUG=1`).
- Step through forward pass, backward pass, or optimizer updates.

This way, you can visually watch variables update with each step.

Debugging CUDA Code

CUDA debugging is a bit trickier, but still possible:

- `cuda-gdb` - NVIDIA's GPU debugger, works like `gdb` but supports stepping into kernels.
- Nsight Systems / Nsight Compute - graphical profilers/debuggers that let you trace kernel launches, memory transfers, and GPU utilization.

If your CUDA code crashes with cryptic messages like `illegal memory access`, `cuda-gdb` can help pinpoint the kernel and even the exact line.

Debugging Common Issues in `llm.c`

1. File not found → Make sure `gpt2_tokenizer.bin`, `train.bin`, and `val.bin` are downloaded.
2. Segfault at `malloc/fread` → Check file paths and dataset sizes.
3. Loss becomes NaN →
 - On CPU: check for division by zero in normalization.
 - On GPU: check loss scaling (mixed precision) or try FP32 path for comparison.
4. Mismatch with PyTorch tests → Run `test_gpt2` or `test_gpt2cu` and compare outputs; this usually isolates whether the bug is in forward pass, backward pass, or optimizer.

Logging & Sanity Checks

When debugging, it helps to add extra logging. The repo already has a lightweight logger, but you can also sprinkle `printfs` (on CPU) or `cudaDeviceSynchronize(); printf(...)` (on GPU) to track values. For example:

```
printf("Step %d: loss=%f\n", step, loss);
```

Sometimes the quickest fix is just to print and see what's going on.

Best Practices for Beginners

- Start with the CPU build when learning—it’s easier to debug than CUDA.
- Always keep a small dataset (like Tiny Shakespeare) for fast iteration.
- Compare against the PyTorch reference for the same batch to catch subtle errors.
- Use `DEBUG=1` whenever you hit strange behavior—you’ll trade speed for clarity, which is usually worth it when learning.

The takeaway: Debug builds (`-g`) turn *llm.c* from a black box into a step-through learning tool. With `gdb`, `lldb`, or an IDE, you can pause at any line, inspect variables, and understand exactly how GPT-2 training works inside C or CUDA. It’s slower, but it’s the clearest way to learn and fix issues.

9. Project Constraints & Readability Contract

The *llm.c* project isn’t trying to be the fastest or most feature-rich GPT-2 trainer. Instead, it has a very deliberate set of constraints—rules the author imposes on the codebase to keep it approachable and educational. You can think of these as the “contract” between the code and the reader: certain things are kept simple on purpose, even if they cost some performance.

Minimalism over Optimizations

- No processor-specific intrinsics: You won’t see AVX, NEON, or other hardware-tuned assembly calls in the CPU path.
- No fancy template metaprogramming: Unlike in C++ frameworks, here you get plain C structs and functions.
- No exotic libraries: Aside from cuBLAS/cuDNN for GPU acceleration, most functionality is implemented directly.

This means the code runs almost anywhere, and you don’t need to understand deep compiler tricks to follow what’s going on.

Transparency over Abstraction

- Every operation is visible in the source. For example, instead of calling a framework function like `nn.CrossEntropyLoss`, you’ll find an explicit forward and backward pass coded in C.
- Data loading, tokenization, optimizer steps, and schedulers are all implemented as separate, small modules in `llmc/`.
- You don’t need to guess what’s happening—if you’re curious, you can open the corresponding `.h` file and see the exact code.

The guiding idea: if something is central to training GPT-2, you should be able to read and understand it.

Performance Where It Matters (but No More)

- OpenMP pragmas are allowed in CPU builds, because they give large speedups with minimal extra code.
- cuBLAS/cuDNN are used for GPU matmuls and attention, because re-implementing them would be a distraction and would make the project impossibly large.
- But the project avoids unnecessary complexity-no kernel fusion, no elaborate caching layers, no half-implemented “framework” abstractions.

This balance ensures you can still run experiments at a reasonable speed, but the code never sacrifices readability.

Educational First

The code is written to teach, not to win benchmarks. That means:

- Variable names are descriptive, not cryptic.
- Comments explain not just *what* happens, but also *why*.
- Files are kept small and focused, rather than sprawling across dozens of layers of abstraction.
- There’s a matching PyTorch reference implementation so you can always check your understanding against a familiar baseline.

Limitations You Should Expect

- Training is slower than PyTorch/XLA/JAX or DeepSpeed-tuned runs.
- Multi-GPU scaling is functional but not heavily optimized.
- Only GPT-2 architectures are covered-don’t expect GPT-3 or transformer variants.
- Features like dataset streaming, checkpoint sharding, or advanced distributed tricks are intentionally left out.

These are not bugs-they’re conscious trade-offs to keep the codebase small, sharp, and didactic.

Why This Matters for You

If you're learning how transformers work, this contract is a gift:

- You won't get lost in performance hacks.
- You won't fight through an abstraction jungle.
- You'll always know that what you're reading is close to the "pure" algorithmic idea.

On the flip side, if you're aiming for production-grade speed, you'll need to layer more on top. But that's outside the mission of *llm.c*.

The takeaway: *llm.c* is bound by a readability contract: clarity over raw speed, transparency over abstraction, minimalism over complexity. These constraints keep the project small enough to fit in your head, while still powerful enough to reproduce GPT-2 training. It's a teaching lab, not a racing car-and that's exactly why it's valuable.

10. Community, Discussions, and Learning Path

The last piece of the quickstart isn't about code at all-it's about the people and resources around the project. *llm.c* has grown into more than just a single repository; it has become a meeting point for learners, tinkerers, and researchers who want to strip large language models down to their essentials. Understanding this community layer is just as important as understanding the code itself.

Discussions and Issues on GitHub

The project's Discussions tab is full of valuable context:

- Developers asking about build errors on different platforms (Linux, macOS, Windows).
- Explorations of how to extend *llm.c* to train larger GPT-2 models (355M, 774M, 1.6B).
- Reports on multi-GPU and MPI runs, including troubleshooting NCCL hangs and performance bottlenecks.
- Debates on mixed precision vs FP32 vs BF16 stability.

Reading these threads is like looking over the shoulders of hundreds of other learners. You'll see not only the official answers but also the thought process of people solving problems in real time.

Roadmap and Contributions

The README and issues sometimes hint at where the project might grow:

- Making the CUDA kernels more modular in `dev/cuda/`.
- Simplifying multi-GPU startup for clusters.
- Adding small tutorial-style docs (like the LayerNorm walkthrough).

The project is open to contributions, but it follows the same minimalist philosophy. If you're thinking of contributing, remember: the goal is clarity first, performance second.

External Learning Resources

While *llm.c* is self-contained, it pairs nicely with outside material:

- The PyTorch reference implementation in `train_gpt2.py` is your canonical oracle for correctness.
- The GPT-2 paper gives the architecture background.
- CUDA and cuBLAS/cuDNN docs explain the GPU APIs that the project calls into.
- Community blog posts often walk through specific sections of the code in plain English, making it easier to digest.

By combining the code, the paper, and these resources, you can triangulate a much deeper understanding.

A Suggested Learning Path

If you're coming to *llm.c* as a beginner, here's a natural progression:

1. Run the CPU trainer (`train_gpt2.c`) on Tiny Shakespeare. Watch the loss decrease.
2. Step through the code with `DEBUG=1`, confirming that you understand forward, backward, and optimizer steps.
3. Move to the FP32 CUDA trainer to see how the same loop runs on GPU.
4. Switch to the modern CUDA trainer (`train_gpt2.cu`) and learn how mixed precision works.
5. Experiment with dataset scripts in `dev/data/`-try your own text corpus.
6. Read the LayerNorm doc in `doc/` to deepen your theory-practice connection.
7. Explore multi-GPU runs with MPI/NCCL if you have access to multiple GPUs.
8. Follow GitHub Discussions for real-world debugging and scaling stories.

Why This Matters

Code alone is not enough. The community context, the discussions, and the learning path make *llm.c* a living project. By engaging with them, you avoid the feeling of learning in isolation. You'll see others wrestling with the same challenges, and you'll have a clearer sense of what to try next.

The takeaway: Beyond the files and scripts, *llm.c* is a community-driven learning environment. GitHub issues, discussions, reference docs, and external tutorials all form part of the “extended classroom.” If the code is the lab bench, the community is the set of lab partners who help you figure things out along the way.

Chapter 2. Data, Tokenization, and Loaders

11. GPT-2 Tokenizer Artifacts (`gpt2_tokenizer.bin`)

A language model like GPT-2 doesn't directly understand English, Vietnamese, or any other natural language. Instead, it understands numbers. These numbers are called tokens. A tokenizer is the tool that translates between human text and tokens. In *llm.c*, the GPT-2 tokenizer is stored in a small file called `gpt2_tokenizer.bin`. This file is the key that lets the model read input text and produce output text that we can understand.

What This File Contains

The file `gpt2_tokenizer.bin` is a binary version of GPT-2's tokenizer. It includes:

Component	Purpose
Byte vocabulary (0–255)	Makes sure every possible character can be represented.
Merge rules (BPE)	Combines frequent sequences like “ing” or ” the” into single tokens for efficiency.
Vocabulary size (~50,257)	Defines how many distinct tokens GPT-2 can work with.
Mapping IDs → text	Lets the program turn model outputs (numbers) back into human-readable strings.

Instead of being written as JSON or text, the tokenizer is stored in binary form. This allows *llm.c* to load it very quickly using a simple file read, which keeps the code clean and fast.

Where It Comes From

You don't need to build this file by hand. The repository provides a script to download it, along with small training and validation datasets:

```
./dev/download_starter_pack.sh
```

After running the script, you should see `gpt2_tokenizer.bin`, `train.bin`, and `val.bin` in your working directory. If the tokenizer is missing, the program cannot run because it won't know how to interpret text.

How the Code Uses It

During training, the tokenizer is not active because the datasets (`train.bin` and `val.bin`) are already pre-tokenized into integers. This keeps the training loop fast and simple.

During sampling or evaluation, the tokenizer becomes important again. After the model predicts a sequence of token IDs, the tokenizer translates those numbers back into text that you can read on your screen.

The C API for the tokenizer, defined in `llmc/tokenizer.h`, provides just three main functions:

```
int tokenizer_init(Tokenizer *t, const char *filename);
int tokenizer_decode(Tokenizer *t, const int *ids, int n, char *out);
void tokenizer_free(Tokenizer *t);
```

This is all you need: initialize the tokenizer from the file, decode tokens into text, and free memory when done.

Example Workflow in Practice

1. Initialize the tokenizer:

```
Tokenizer t;
tokenizer_init(&t, "gpt2_tokenizer.bin");
```

2. Decode a sequence of tokens back to text:

```
char buf[512];
tokenizer_decode(&t, tokens, ntokens, buf);
printf("%s\n", buf);
```

3. Clean up memory when you no longer need it:

```
tokenizer_free(&t);
```

This small cycle is enough to turn model outputs into readable sentences.

Why It Matters

Without the tokenizer, the model cannot communicate. The tokenizer is like a shared dictionary between humans and the neural network. If you give the model text, the tokenizer converts it into numbers the model understands. When the model responds, the tokenizer converts its numbers back into text. If the tokenizer does not match the dataset, the model's predictions will come out as gibberish. Keeping the tokenizer and dataset in sync is essential for correct training and evaluation.

Try It Yourself

Here are a few small exercises you can do to understand the tokenizer better:

1. Check that the file exists: After running the starter pack script, verify that `gpt2_tokenizer.bin` is in your directory. Try running the trainer without it and observe the error message.
2. Inspect vocab size: Run the trainer and look for the line that prints `vocab_size: 50257`. Compare this with `padded_vocab_size: 50304`. Why do you think padding helps GPUs?
3. Decode a sequence manually: Write a short C program that loads the tokenizer and decodes a fixed list of token IDs (for example `[464, 3290, 318]`). Observe what text you get.
4. Mismatch experiment: If you build your own dataset with a different tokenizer (say, a custom vocabulary), try decoding it with `gpt2_tokenizer.bin`. Notice how the output becomes meaningless, showing why consistency matters.
5. Dataset + tokenizer link: Open `train.bin` in a hex viewer. You'll see it's just numbers. Use the tokenizer to decode the first few hundred tokens and see real text emerge.

The Takeaway

`gpt2_tokenizer.bin` is a tiny but vital file. It is the bridge that allows the model and humans to speak the same language. Training is efficient because all data is pre-tokenized, and when you want to see what the model has written, the tokenizer turns raw numbers back into words. Without it, the entire system would be silent.

12. Binary Dataset Format (`train.bin` and `val.bin`)

Just like the tokenizer turns text into numbers, the datasets in *llm.c* are stored as numbers too. Instead of reading plain text files like `.txt`, the training and validation data are kept in simple binary files: `train.bin` and `val.bin`. These files are the fuel for the training loop.

What These Files Look Like

At first glance, `train.bin` and `val.bin` look like unreadable blobs if you open them in a text editor. That's because they are not meant to be human-readable. They contain:

Part	Description
A tiny header (about 1 KB)	Stores metadata such as sequence length and vocab size.
A stream of token IDs (<code>uint16</code>)	Every tokenized word piece from the dataset, saved as 16-bit integers.

Each integer represents one token from the tokenizer's vocabulary. Since GPT-2 has a vocabulary of about 50,000 tokens, 16-bit integers (`uint16_t`) are enough to store them all.

Why Binary Format?

- Efficiency: Instead of re-tokenizing text every time, the data is pre-tokenized once and stored as numbers. The trainer just reads them directly.
- Speed: Reading integers from a file is faster than parsing and processing raw text.
- Simplicity: The training loop only has to deal with arrays of integers-no string handling, no parsing, no surprises.

This choice makes the training code in *llm.c* much cleaner and faster.

How the Dataloader Uses Them

When training starts, the dataloader reads chunks of numbers from `train.bin`. Each chunk corresponds to one batch of size $B \times T$:

- B = batch size (number of examples in a batch).
- T = sequence length (number of tokens per example).

For example, if $B = 8$ and $T = 1024$, the dataloader will read $8 \times 1024 = 8192$ token IDs from the file, reshape them into sequences, and feed them to the model.

The validation file (`val.bin`) works the same way but is only used occasionally during training to measure validation loss. This helps detect overfitting.

Workflow in Code

Inside the repo, you'll see functions like these in `llmc/dataloader.h`:

```
int dataloader_init(Dataloader *loader, const char *filename, int B, int T);
int dataloader_next_batch(Dataloader *loader, int *inputs, int *targets);
void dataloader_reset(Dataloader *loader);
void dataloader_free(Dataloader *loader);
```

Here's what happens step by step:

1. Initialize with the binary file and batch/sequence sizes.
2. Next batch reads the next $B \times T$ tokens into an input array and a target array.
3. Reset allows re-reading from the beginning when you start a new epoch.
4. Free cleans up resources when training ends.

The target array is simply the same sequence shifted by one token-because language modeling predicts the *next* token.

Why It Matters

The dataset format is what makes *llm.c* practical. Without it, the code would need to handle messy text, encodings, and tokenization during every training step. By storing clean arrays of token IDs, the training loop becomes very short and easy to follow. It's a design decision that keeps the project minimal yet faithful to real training pipelines.

Try It Yourself

1. Check file size: Run `ls -lh train.bin` and notice how large it is compared to a plain `.txt` file. Why is it smaller or larger?
2. Peek inside: Use a hex viewer (`xxd train.bin | head`) to see raw numbers. They won't look like text, but they are the tokens the model trains on.
3. Count tokens: Write a short Python or C script to count how many token IDs are stored in `train.bin`. This gives you a sense of dataset size.
4. Mini-dataset: Try generating your own dataset from a small `.txt` file using the scripts in `dev/data/`. See how the `.bin` file is created.
5. Validation experiment: During training, reduce the validation set to only a few batches and observe how the validation loss stabilizes or fluctuates compared to training loss.

The Takeaway

`train.bin` and `val.bin` may look like gibberish, but they are carefully prepared binary files containing token IDs. They make training faster, simpler, and more reproducible. The dataloader in *llm.c* reads these numbers in neat chunks and serves them directly to the model, letting you focus on learning how transformers work instead of wrestling with raw text parsing.

13. Dataset Scripts in `dev/data/`

The repository doesn't just give you ready-made binary datasets like `train.bin` and `val.bin`. It also provides scripts inside the `dev/data/` folder that show you how to create your own. These scripts are important because they demonstrate how raw text gets transformed into the binary format that the dataloader in *llm.c* expects.

What's Inside `dev/data/`

This folder contains small Python scripts that:

Script	Purpose
<code>prepare_shakespeare.py</code>	Turns the Tiny Shakespeare dataset into <code>train.bin</code> and <code>val.bin</code> .
<code>prepare_openwebtext.py</code>	Prepares a large-scale dataset similar to the one GPT-2 was trained on.
Other helpers	Tokenize raw <code>.txt</code> files, split them into train/val, and save to binary.

Each script follows the same basic recipe:

1. Read raw text from a source file.
2. Apply the GPT-2 tokenizer to turn text into token IDs.
3. Split the tokens into training and validation portions.
4. Write the IDs into binary files that *llm.c* can read directly.

Why Preprocessing Happens Outside C

In C, handling text files with Unicode, punctuation, and different encodings is messy. Instead, preprocessing is done once in Python, where tokenizers are easier to use. The results are saved in a simple binary format (uint16 IDs). From then on, C only has to deal with arrays of integers-clean and efficient.

This design keeps the training loop minimal: no text parsing, no string handling, just numbers.

Example: Tiny Shakespeare

One of the simplest datasets is Tiny Shakespeare, about 1 MB of text from Shakespeare's plays. The script `prepare_shakespeare.py` will:

- Read `input.txt` (the raw text).
- Use the GPT-2 tokenizer (`gpt2_tokenizer.bin`) to turn every word and symbol into token IDs.
- Split 90% of the data into `train.bin` and 10% into `val.bin`.

After running the script, you'll have small binary files that let you train GPT-2 from scratch in minutes on CPU or GPU.

Example: OpenWebText

The script `prepare_openwebtext.py` shows how to tokenize a much larger dataset, closer to what GPT-2 was originally trained on. This is heavier and requires more disk space, but it's useful if you want to try scaling up training to bigger models.

Why It Matters

These scripts are more than convenience tools—they are examples of how to adapt `llm.c` to your own data. If you have a collection of emails, poems, or programming code, you can:

1. Put them into a single `.txt` file.
2. Modify one of the scripts in `dev/data/`.
3. Generate new `train.bin` and `val.bin` files.
4. Train GPT-2 on your own text.

By separating dataset creation from training, *llm.c* keeps the C code small and makes experimentation flexible.

Try It Yourself

1. Run the Shakespeare script:

```
python dev/data/prepare_shakespeare.py
```

Then check that `train.bin` and `val.bin` were created.

2. Open the binary files with a hex viewer and confirm that they contain only numbers.
3. Modify the script to tokenize a different text file (for example, your own writing).

4. Compare dataset sizes: Tiny Shakespeare is tiny (MBs), OpenWebText is huge (GBs). Observe how training speed changes depending on dataset size.
5. Re-run training with your custom dataset and watch how the model starts generating text in your style.

The Takeaway

The `dev/data/` scripts are the bridge between raw human text and the binary datasets used in training. They let you prepare small demo datasets or scale up to larger corpora. By experimenting with these scripts, you learn how to bring your own data into *llm.c* and train a GPT-style model on anything you like.

14. DataLoader Design (Batching, Strides, Epochs)

Now that the datasets are prepared as `.bin` files, we need a way to feed them into the model during training. This is the job of the `DataLoader` in *llm.c*. You'll find its interface in `llmc/dataloader.h`, and its purpose is very simple: take a big stream of token IDs from `train.bin` or `val.bin`, cut it into manageable chunks, and serve those chunks to the training loop as batches.

The Core Idea

Training a language model requires two arrays for every batch:

- Inputs: a sequence of token IDs, like `[The, cat, sat, on]`
- Targets: the same sequence shifted by one, like `[cat, sat, on, the]`

The model learns to predict each next token in the sequence. The `DataLoader` automates slicing these arrays out of the giant dataset file.

The Interface

In the code you'll see function declarations like these:

```
int dataloader_init(Dataloader *loader, const char *filename, int B, int T);
int dataloader_next_batch(Dataloader *loader, int *inputs, int *targets);
void dataloader_reset(Dataloader *loader);
void dataloader_free(Dataloader *loader);
```

Here's what each does:

- `dataloader_init`: opens the dataset file, remembers batch size B and sequence length T .
- `dataloader_next_batch`: returns the next chunk of $B \times T$ tokens (inputs) and their shifted version (targets).
- `dataloader_reset`: rewinds to the start of the file when an epoch ends.
- `dataloader_free`: closes the file and releases memory.

This design keeps the training loop clean: just call `next_batch` and you get the data ready for forward/backward passes.

$B \times T$ Explained

The two most important parameters are:

Symbol	Meaning	Example
B	Batch size (how many sequences per step)	16
T	Sequence length (how many tokens per sequence)	1024

So one batch contains $B \times T$ tokens. For example, with $B = 16$ and $T = 1024$, each batch holds 16,384 tokens. The `DataLoader` simply reads that many numbers from the binary file and arranges them in memory.

Strides Through the Dataset

As you call `dataloader_next_batch`, the loader moves forward through the dataset by $B \times T$ tokens each time. When it reaches the end of the dataset file, it either:

- Resets back to the beginning (`dataloader_reset`), or
- Switches from training to validation, depending on the training loop's needs.

This stride-based reading is efficient: no random access, just sequential reads from a file.

Epochs and Shuffling

In deep learning, an epoch means one full pass through the dataset. The `DataLoader` in *llm.c* is simple: it goes linearly from start to finish. It doesn't shuffle data like PyTorch's `DataLoader`. Why? Because language data is already very diverse, and the project values minimal code over extra features. If you want shuffling, you can preprocess the dataset differently before creating `.bin` files.

Why It Matters

The `DataLoader` is the quiet workhorse of training. It ensures that every step sees a fresh batch of token sequences, always with matching inputs and targets. By separating dataset reading from the training loop, the code stays clean and focused. This design also makes it easy to swap datasets—once you generate a `.bin` file, the loader doesn't care where it came from.

Try It Yourself

1. Print the first batch: Modify the code to print the first 20 input tokens and their targets. See how each input token aligns with the next target token.
2. Experiment with `B` and `T`: Set `B = 2` and `T = 8` and observe how the loader slices the dataset into tiny chunks. Then try larger values and see how memory usage changes.
3. Check epoch length: Write a small loop to count how many batches you get before `dataloader_reset` is called. Does this match the total tokens divided by $B \times T$?
4. Validation check: Observe how often the training loop switches to `val.bin`. How does validation loss compare to training loss over time?
5. Custom stride: Modify the code so the `DataLoader` skips some tokens between batches. What effect does this have on training?

The Takeaway

The `DataLoader` in *llm.c* is intentionally simple. It streams token IDs in fixed-sized batches, moves forward stride by stride, and resets when done. This straightforward design avoids complexity and keeps the focus on the model itself, while still teaching you the essential mechanics of batching and sequence handling in language model training.

15. EvalLoader and Validation Workflow

Training a model isn't just about watching the training loss go down. To know whether the model is actually learning patterns that generalize—and not just memorizing the training data—you need to run validation. In *llm.c*, validation is handled by a component called the `EvalLoader`, which works just like the `DataLoader` but reads from the validation dataset (`val.bin`) instead of the training dataset (`train.bin`).

Why We Need Validation

Imagine teaching a student only by drilling them with the same math problems over and over. They might get really good at those problems, but fail completely when given new ones.

Validation is like giving the student a pop quiz with unseen questions. If they do well, you know they’ve actually learned the concepts.

For language models, validation helps detect overfitting: when the training loss keeps improving but the validation loss stays flat or even gets worse.

How EvalLoader Works

EvalLoader lives in the same code file as the DataLoader (`llmc/dataloader.h`), but it points to a different dataset file. Its workflow is nearly identical:

1. Open `val.bin` and prepare for reading.
2. Serve up batches of size $B \times T$ (batch size \times sequence length).
3. Provide inputs and targets the same way as the training DataLoader.
4. Reset after one full pass through the file.

The training loop typically calls the EvalLoader at intervals—for example, every few hundred steps—so you get a snapshot of validation loss during training.

What Happens During Validation

When validation is triggered:

1. The current model parameters are frozen (no gradient updates).
2. A few batches are read from `val.bin`.
3. The model runs forward passes only, computing the loss on each batch.
4. The losses are averaged and reported as validation loss.

This doesn’t take long because it usually samples just a subset of the validation dataset, not the entire file.

Training Loop with Validation

In pseudocode, the loop looks like this:

```
for (step = 0; step < max_steps; step++) {
    dataloader_next_batch(&train_loader, inputs, targets);
    forward_backward_update(model, inputs, targets);

    if (step % eval_interval == 0) {
        float val_loss = 0.0f;
        for (int i = 0; i < eval_batches; i++) {
```

```

        evalloader_next_batch(&val_loader, inputs, targets);
        val_loss += forward_only(model, inputs, targets);
    }
    val_loss /= eval_batches;
    printf("step %d: val loss %.4f\n", step, val_loss);
}
}

```

This is simplified, but it shows the idea: the validation loop is nested inside the training loop, running occasionally instead of every step.

Why It Matters

Validation is the reality check of training. Without it, you could train forever and celebrate low training losses, only to discover that your model produces nonsense on new text. By tracking validation loss, you can:

- Detect overfitting early.
- Adjust hyperparameters (like learning rate or batch size).
- Know when training has plateaued and it's time to stop.

In professional setups, validation curves are often plotted live, but in *llm.c*, the minimalist approach is to just print numbers to the console.

Try It Yourself

1. Watch val loss: Run training and note how validation loss compares to training loss. Do they both decrease together?
2. Overfitting demo: Train on a very tiny dataset (like 10 KB of text). Notice how training loss plummets but validation loss stalls or rises.
3. Change eval interval: Reduce `eval_interval` so validation runs every step. How much slower does training feel?
4. Change eval batches: Set `eval_batches` to 1 vs 100. What difference does this make in the stability of the reported validation loss?
5. Validation as stopping rule: Stop training when validation loss stops improving for many intervals. How does this affect final performance?

The Takeaway

The EvalLoader is a twin of the DataLoader, but for validation. It feeds the model data it has never seen during training, and the resulting validation loss tells you whether your model is learning useful patterns or just memorizing. It's the simplest safeguard against wasted compute, and it's an essential part of every training loop—even in the stripped-down world of *llm.c*.

16. Sequence Length and Memory Budgeting

When training GPT-2 in *llm.c*, one of the most important decisions you make is choosing the sequence length (often called *T*). This value determines how many tokens the model processes in a single forward pass. It might sound like just another parameter, but sequence length has a huge impact on what the model can learn, how much memory it uses, and how fast training runs.

What Sequence Length Means

Sequence length is simply the number of tokens per training example. If *T* = 1024, the model reads 1,024 tokens in a row (like words or subwords) and tries to predict the next token at each position.

Think of it like this: if you give the model a paragraph of text, sequence length is how much of that paragraph it sees at once. Shorter lengths give the model less context, while longer lengths allow it to capture bigger patterns, like whole paragraphs or even multiple pages.

Where It Appears in the Code

In the logs, you'll often see lines like:

```
max_seq_len: 1024
```

This number is defined in the GPT-2 configuration and passed into the DataLoader. The DataLoader slices chunks of exactly *T* tokens from `train.bin` and `val.bin`. The model itself has fixed positional embeddings of size *T*, so it cannot process sequences longer than this maximum.

Memory Costs of Longer Sequences

Transformers are powerful but expensive. The attention mechanism compares every token to every other token in the sequence. This means memory and compute scale with the square of sequence length:

Sequence Length (T)	Relative Attention Cost
256	1×
512	4×
1024	16×
2048	64×

So doubling T doesn't just double the cost—it multiplies it by four. That's why training at long context lengths requires a lot of GPU memory.

Trade-offs

- Shorter sequences: Faster, less memory, but limited context. Good for quick experiments or tiny datasets like Tiny Shakespeare.
- Longer sequences: More memory, slower, but the model can understand larger spans of text. Required for large-scale GPT-2 training.

You can think of sequence length as a dial: turning it up increases the model's ability to “remember,” but it also makes training much heavier.

Practical Choices in *llm.c*

- Tiny Shakespeare example: often trained with $T = 64$ or 128 for speed.
- GPT-2 small (124M): typically uses $T = 1024$, the same as the original paper.
- If your GPU has limited memory, you might need to shrink T and/or batch size B.

Why It Matters

Choosing sequence length is about balancing learning power against hardware limits. A too-small sequence length can prevent the model from capturing long-term dependencies. A too-large one can make training impossible on your hardware. Every run of *llm.c* is a negotiation between what you'd like the model to see and what your system can handle.

Try It Yourself

1. Short vs long: Train Tiny Shakespeare with $T = 64$ and then $T = 256$. Compare both the speed and the coherence of generated text.
2. Memory test: Increase T step by step until you hit an out-of-memory (OOM) error. Note the maximum your GPU can handle.
3. Batch trade-off: Try reducing batch size B while increasing T . Can you keep GPU memory stable while giving the model more context?
4. Validation impact: Run with different T values and watch how validation loss behaves. Does longer context always help?
5. Inspect embeddings: Print out the shape of the positional embeddings. Notice how they are always tied to T .

The Takeaway

Sequence length (T) controls how much context the model sees. It directly determines the size of the positional embeddings, the structure of batches, and the memory required for attention. In *llm.c*, adjusting T is one of the fastest ways to explore the trade-offs between speed, memory, and model capability.

17. Reproducibility and Seeding Across Runs

When training machine learning models, it's common to notice that two runs—using the same code and the same dataset—don't produce exactly the same results. This happens because many parts of training involve randomness. In *llm.c*, reproducibility is controlled by random seeds. A seed is a starting point for a random number generator. If you always start from the same seed, the sequence of “random” numbers will be identical, and so will the training run.

Where Randomness Appears

Even in a small project like *llm.c*, randomness shows up in several places:

Component	Random Role
Weight initialization	The model's parameters (like attention matrices) are set randomly at the start.
Optimizer states	Some optimizers use random noise (though AdamW is mostly deterministic).
Sampling outputs	When generating text, randomness decides which token to pick if probabilities are close.

Component	Random Role
Parallelism	On GPU, threads may execute in slightly different orders, sometimes introducing small nondeterminism.

Without a fixed seed, every training run can drift apart, even if all settings look the same.

How *llm.c* Handles Seeds

The repository provides a small random utilities module: `llmc/rand.h`. Inside you'll find functions such as:

```
void manual_seed(uint64_t seed);
float normal_(float mean, float std);
```

- `manual_seed` sets the seed for the internal random number generator, ensuring reproducibility.
- `normal_` is used for initializing weights with Gaussian noise, similar to PyTorch's `torch.nn.init.normal_`.

When you call `manual_seed(1337);`, the model weights will be initialized the same way every time.

Why Seeds Don't Guarantee Perfect Reproducibility

Even with a fixed seed, you may still see small differences:

- GPU kernels sometimes use parallel algorithms that are not bitwise deterministic.
- Floating-point math can produce slightly different rounding on different hardware.
- Multi-GPU runs (via NCCL/MPI) may introduce nondeterministic reduce operations.

These differences are usually tiny-validation loss might vary by 0.001-but they exist. For most educational purposes, *llm.c* seeds are enough to make experiments repeatable.

Typical Defaults

In many examples, you'll see:

```
manual_seed(1337);
```


This “magic number” 1337 is just a convention. You can change it to any integer. Using the same seed across runs guarantees the same starting weights, which helps when comparing hyperparameters.

Why It Matters

Reproducibility is crucial in machine learning because it lets you:

- Debug effectively: If a bug appears, you want it to appear consistently.
- Compare settings: You can test learning rates or batch sizes fairly by keeping everything else the same.
- Share results: Other people can run your exact setup and see the same outcomes.

Without seeds, it becomes hard to tell whether a difference came from your hyperparameter change or just random luck.

Try It Yourself

1. Run twice with same seed: Train GPT-2 with `manual_seed(1337)` set. Do you get identical training loss curves?
2. Change the seed: Try `manual_seed(42)` and compare the loss curve. How similar are they? Do they converge to about the same final validation loss?
3. Remove seeding: Comment out the seed line and run again. Notice how runs diverge.
4. Sampling experiment: With a fixed seed, generate text multiple times. Then change the seed and generate again. See how outputs change.
5. Multi-GPU test: If you have more than one GPU, run the same seed across devices. Do results stay exactly the same or only approximately?

The Takeaway

Reproducibility in *llm.c* comes from setting seeds for random number generators. While floating-point quirks mean you can’t always get perfect bit-for-bit matches, seeds let you control the biggest source of randomness: weight initialization and sampling. With seeding, you can debug, compare, and share results confidently.

18. Error Surfaces from Bad Data (Bounds, Asserts)

When training a model in *llm.c*, everything depends on the quality and correctness of the data you feed in. If the dataset or batches contain mistakes, the training process can go off track quickly—sometimes by crashing outright, other times by producing strange loss values like NaN. To guard against this, the code uses bounds checks and asserts that catch problems early.

What Can Go Wrong with Data

There are several common data issues:

Problem	What Happens
Token ID out of range	The model expects IDs between 0 and <code>vocab_size-1</code> . A wrong ID can cause array indexing errors.
Empty or short dataset	The DataLoader may run out of tokens before filling a batch.
Mis-matched tokenizer	If you build a dataset with a different tokenizer, IDs may not correspond to the GPT-2 tokenizer in <code>gpt2_tokenizer.bin</code> . This produces nonsense outputs.
Corrupt <code>.bin</code> files	If files are incomplete or written incorrectly, the DataLoader might read garbage values.

These errors show up as segfaults, invalid memory access, or exploding losses during training.

How *llm.c* Defends Against Bad Data

The repository makes heavy use of asserts—simple checks that stop the program immediately if something unexpected happens. For example, in `llmc/utils.h`, functions like `freadCheck` and `mallocCheck` ensure that file reads and memory allocations succeed. If not, they print an error message and abort instead of silently failing.

Inside the DataLoader, token IDs are often validated to make sure they fall inside the expected vocabulary range. If you try to access an invalid index in the embedding table, the program will crash quickly, which is better than continuing with corrupted values.

Example: Vocab Range Check

During training, every input token is used to look up a row in the embedding matrix. If a token ID is too large, you'd access memory outside the matrix. This is why checking `0 <= id < vocab_size` is essential. In C, asserts provide this safety net.

```
assert(id >= 0 && id < vocab_size);
```

This kind of check may look simple, but it saves hours of debugging mysterious crashes.

Error Surfaces in Loss

Even if your program doesn't crash, bad data can create "error surfaces" in the loss function:

- NaNs: Appear when invalid values propagate through softmax, layernorm, or division operations.
- Flat loss: If the dataset is empty or repetitive, the model never improves.
- Mismatch behavior: Training loss decreases but validation loss stays high if training and validation sets use inconsistent tokenization.

These are signs that something is wrong with the dataset or preprocessing.

Why It Matters

C is a low-level language with very little safety by default. One out-of-range index can corrupt memory and cause unpredictable bugs. By aggressively checking assumptions (file sizes, vocab bounds, token IDs), *llm.c* turns hard-to-find errors into immediate, clear failures. For learners, this makes it much easier to understand what went wrong.

Try It Yourself

1. Corrupt a dataset: Open `train.bin` and delete a few bytes. Run training and see what error appears. Notice how quickly asserts catch it.
2. Force a bad ID: Modify the DataLoader to add `+100000` to a token. Does the model crash with an assertion?
3. Skip asserts: Temporarily disable checks and rerun. Compare how much harder it is to figure out what went wrong.
4. Validation mismatch: Tokenize a file with a different tokenizer and save it as `val.bin`. Watch how the validation loss behaves compared to training loss.
5. Print debug info: Add logging to display the first 20 tokens of each batch. Can you spot bad data before it crashes?

The Takeaway

Bad data can silently sabotage training, but *llm.c* uses asserts and bounds checks to make errors loud and clear. This design choice helps learners focus on the real logic of transformers instead of chasing hidden bugs caused by corrupted or mismatched datasets. In machine learning, good data hygiene and strict validation are as important as the model itself.

19. Tokenization Edge Cases (UNKs, EOS, BOS)

Tokenization looks simple at first: take text, split it into tokens, and assign each token an ID. But in practice, there are always tricky situations. *llm.c* inherits the quirks of the GPT-2 tokenizer, which is byte-level BPE (Byte Pair Encoding). This design mostly avoids “unknown” tokens, but it still has details you need to understand when preparing datasets or interpreting outputs.

No True “UNK” in GPT-2

Some tokenizers, like those used in earlier NLP systems, include a special UNK (unknown) token for words that aren’t in the vocabulary. GPT-2 avoids this problem by working at the byte level:

- Every possible byte (0–255) is in the base vocabulary.
- If the tokenizer doesn’t know how to split a character or word, it just falls back to raw bytes.

That means you will never see an UNK token in *llm.c*. Any input text is always representable. This is one of the main reasons GPT-2’s tokenizer is so robust.

Special Tokens: EOS and BOS

Even though GPT-2 doesn’t use UNK, it does use other special tokens:

Token	ID	Purpose
EOS (End of Sequence)	50256	Marks the end of a text segment. Used during training and sampling.
BOS (Beginning of Sequence)	Not explicit in GPT-2	GPT-2 doesn’t use a fixed BOS token. Instead, the model assumes generation starts at position 0.

In *llm.c*, you’ll often see EOS at the end of training sequences or when sampling text. If you generate text and see strange endings, it’s usually because the model predicted EOS.

Whitespace Quirks

The tokenizer also handles whitespace in a slightly unusual way. For example, the word “hello” and the word ” hello” (with a leading space) map to different tokens. This is why generated text sometimes starts with a space—it’s part of the token definition.

Example:

- `"hello"` → token ID 31373
- `" hello"` → token ID 15496

This is normal behavior for GPT-2. It helps the model capture spacing and punctuation consistently.

Unicode and Rare Characters

Because it's byte-level, GPT-2 can encode emojis, accented characters, or even binary junk data. But the BPE merges are optimized for English, so rare characters often get split into multiple byte tokens. That means sequences with lots of rare symbols (like Chinese or emojis) will use more tokens than plain English text.

Why It Matters

Edge cases in tokenization affect both dataset preparation and model outputs. If you see weird spacing or early EOS tokens, it's not a bug—it's just how the tokenizer works. Understanding these quirks helps you debug outputs and prepare datasets without surprises.

Try It Yourself

1. EOS inspection: Open `val.bin` with a hex viewer and look for token ID 50256. These mark the ends of text segments.
2. Whitespace check: Use the tokenizer to encode `"hello"` and `" hello"`. Compare the token IDs.
3. Emoji test: Encode a string with emojis (e.g., `" 🐼 "`) and see how many tokens it becomes.
4. Rare character dataset: Create a small `.txt` file with accented characters and tokenize it. How many bytes does each character consume?
5. Sampling experiment: Generate text until you see the EOS token appear. Notice how the model “knows” to stop.

The Takeaway

Tokenization in GPT-2 is robust, but it has quirks. There are no unknown tokens thanks to byte-level encoding, but whitespace and special tokens like EOS play important roles. By experimenting with these edge cases, you'll develop an intuition for how raw text is mapped into the numbers that drive training and generation in *llm.c*.

20. Data Hygiene and Logging

When training with *llm.c*, having clean data is just as important as having the right model code. If the dataset contains errors, duplicates, or formatting issues, the model may waste capacity memorizing noise instead of learning useful patterns. This is where data hygiene comes in—making sure your training and validation sets are prepared properly. Alongside this, logging ensures you can monitor what’s happening during training and catch problems early.

What Data Hygiene Means

Data hygiene is about making sure your dataset is both valid and useful. For language models, this includes:

Check	Why It Matters
Correct tokenization	Must match the tokenizer (<code>gpt2_tokenizer.bin</code>), otherwise IDs won’t line up.
No corrupt files	Binary <code>.bin</code> files must be complete; partial writes cause crashes.
Balanced splits	Training and validation sets should come from the same distribution.
Reasonable size	Too small → overfitting. Too large → slow or infeasible.
Deduplication	Repeated passages (e.g., web scrapes) make models memorize instead of generalize.

The scripts in `dev/data/` handle basic hygiene by tokenizing consistently and splitting into train/val sets. But if you bring your own dataset, you are responsible for cleaning it first.

Logging During Training

Once training starts, logging becomes your window into what’s happening. *llm.c* uses a minimal logging system (`llmc/logger.h`) to print progress to the console. Typical logs include:

```
step 0: train loss 5.19, val loss 5.32
step 100: train loss 4.87, val loss 5.01
step 200: train loss 4.62, val loss 4.88
```

These numbers let you track:

- Training loss: Is the model fitting the data?
- Validation loss: Is it generalizing, or overfitting?
- Step timing: How long each batch takes, useful for profiling.

Even in such a small project, this logging loop gives you most of what you need to debug runs.

Why Hygiene and Logging Go Together

Bad data often reveals itself in the logs. For example:

- If validation loss is much higher than training loss, your validation set may be mismatched.
- If loss suddenly becomes NaN, your dataset might contain corrupt tokens.
- If loss plateaus at a high value, you may have too little data or poor preprocessing.

By keeping your data clean and watching logs closely, you can detect these issues early instead of wasting hours of compute.

Try It Yourself

1. Dirty dataset test: Take a `.txt` file, add random symbols or binary junk, and prepare a `.bin` dataset. What happens to training loss?
2. Duplicate passages: Copy the same paragraph 100 times into a training file. Does validation loss improve, or does the model just memorize?
3. Log frequency: Modify the code to log every step instead of every N steps. How noisy are the results?
4. Custom logger: Extend the logger to also print gradient norms or learning rate values. Does this help you understand training dynamics better?
5. Compare splits: Build two datasets with different train/val splits. Which one gives more stable validation losses?

The Takeaway

Data hygiene ensures the model learns from clean, consistent input, while logging ensures you can see whether learning is actually happening. Together, they form the foundation of reliable experiments in *llm.c*. If you clean your data carefully and pay attention to the logs, you'll catch most problems before they become serious.

Chapter 3. Model Definition and Weights

21. GPT-2 Config: Vocab, Layers, Heads, Channels

Every GPT-2 model, no matter how large or small, is defined by a handful of configuration numbers. These numbers decide how big the model is, how much memory it needs, and how powerful it can become. In *llm.c*, these settings are stored in a simple config struct and printed at the start of training. They describe the “blueprint” of the transformer.

The Core Parameters

Here are the most important values you'll see in the logs:

Parameter	Meaning	Example (GPT-2 Small)
<code>vocab_size</code>	Number of distinct tokens (from tokenizer).	50,257
<code>padded_vocab_size</code>	Vocab size rounded up to nearest multiple (for GPU efficiency).	50,304
<code>max_seq_len</code>	Longest sequence of tokens the model can handle.	1,024
<code>num_layers</code>	Number of transformer blocks stacked on top of each other.	12
<code>num_heads</code>	Number of attention heads per block.	12
<code>channels</code>	Width of hidden states (embedding dimension).	768
<code>num_parameters</code>	Total trainable weights in the model.	~124M

Together, these values define both the structure and the capacity of the model.

What They Control

- Vocabulary size connects the model to the tokenizer. Every input token ID must be less than `vocab_size`. The padded version makes GPU matrix multiplications easier.
- Max sequence length fixes the size of the positional embeddings. If you set this to 1024, the model can't read beyond 1024 tokens in one pass.
- Layers control model depth. Each layer contains an attention block and an MLP. More layers = more representational power.
- Heads divide attention into parallel “subspaces.” With 12 heads, the model can track different types of relationships in the text at the same time.
- Channels set the dimensionality of embeddings and hidden vectors. Larger channels mean more expressive representations but also more computation.
- Parameters are the sum of it all. This number tells you how heavy the model is to train and how much memory it will consume.

Configs Across GPT-2 Sizes

The original GPT-2 models come in several sizes:

Model	Layers	Heads	Channels	Parameters
GPT-2 Small	12	12	768	124M
GPT-2 Medium	24	16	1024	355M

Model	Layers	Heads	Channels	Parameters
GPT-2 Large	36	20	1280	774M
GPT-2 XL	48	25	1600	1.6B

llm.c can scale between these by just changing a few numbers in the config struct.

Where Config Appears in the Code

In `train_gpt2.c` and `train_gpt2.cu`, you'll see something like:

```
GPT2Config config = {
    .vocab_size = 50257,
    .max_seq_len = 1024,
    .num_layers = 12,
    .num_heads = 12,
    .channels = 768,
};
```

Later, the model is initialized using this struct, and the log prints all the derived information (like `num_parameters`).

Why It Matters

The config is the contract between your dataset and your model.

- If `vocab_size` doesn't match your tokenizer, you'll get crashes.
- If `max_seq_len` is too small, you'll lose context.
- If `num_layers` or `channels` are too large for your GPU, you'll run out of memory.

By tweaking the config, you decide whether you want a tiny model for learning or a massive one closer to GPT-2 XL.

Try It Yourself

1. Print config: Run the trainer and note the printed values. Compare them with the GPT-2 sizes in the table.
2. Shrink the model: Change `num_layers = 4`, `num_heads = 4`, and `channels = 256`. Train on Tiny Shakespeare and see how fast it runs.

3. Increase sequence length: Try setting `max_seq_len = 2048`. Does your GPU still handle it, or do you get out-of-memory errors?
4. Parameter count check: Compute how many parameters your custom config has. Compare it to the reported `num_parameters`.
5. Tokenizer mismatch test: Intentionally set `vocab_size = 30000` and watch what error appears when loading the tokenizer.

The Takeaway

The GPT-2 config struct in *llm.c* is small but powerful. It defines everything about the model's architecture: vocabulary, sequence length, depth, width, and total parameters. By adjusting just a few integers, you can scale from a toy model that runs on CPU to a billion-parameter giant (if your hardware allows it). Understanding these numbers is the first step to understanding how transformer capacity is controlled.

22. Parameter Tensors and Memory Layout

Once the GPT-2 configuration is set, the next big step is to allocate the parameters of the model. These are the trainable numbers—weights and biases—that define how the model processes input tokens. In *llm.c*, parameters are stored in flat arrays of floats rather than in deeply nested objects like in PyTorch. This choice makes the code easier to read and keeps memory access predictable.

What Are Parameters?

Every part of the transformer has its own trainable weights:

- Embedding tables: one for tokens and one for positions.
- Attention layers: query, key, value, and output projections.
- MLP layers: two linear layers plus their biases.
- LayerNorms: scale (`gamma`) and shift (`beta`) values.
- Final projection: maps hidden states back to vocab size for logits.

Together, these add up to hundreds of millions of numbers, even for GPT-2 Small.

Flat Memory Design in *llm.c*

Instead of allocating each parameter separately, *llm.c* stores all parameters in one contiguous block of memory. Each layer is given a slice of this big array.

This has two benefits:

1. Simplicity: You only need one malloc (or cudaMalloc) for all parameters.
2. Performance: Contiguous memory access is faster on both CPU and GPU.

To keep track of where each layer’s weights live inside the block, the code uses offsets.

Example in Code

In `train_gpt2.c`, parameters are packed into a single array:

```
float* params = (float*)mallocCheck(config.num_parameters * sizeof(float));
```

Later, helper functions compute pointers into this array for each sub-module. For example, the token embedding weights are just the first slice:

```
float* token_embedding_table = params;
```

Then the program moves forward, assigning chunks to positional embeddings, attention weights, and so on.

Shapes of the Tensors

Even though parameters are stored in 1D memory, they conceptually form 2D or 3D tensors. For example:

Parameter	Shape	Purpose
Token embeddings	[vocab_size, channels]	Maps token IDs to vectors.
Positional embeddings	[max_seq_len, channels]	Adds position info.
Attention weights (Q, K, V, O)	[channels, channels]	Project hidden states.
MLP layers	[channels, 4×channels] and [4×channels, channels]	Expand and contract hidden states.
LayerNorm scale/shift	[channels]	Normalize and rescale features.

When you look at the code, remember: these shapes are “virtual.” They’re just views into slices of the big 1D array.

Why This Layout Works Well

PyTorch or TensorFlow manage parameter tensors with lots of abstractions. *llm.c* strips this away: you see the raw memory, the exact number of parameters, and the order they're laid out in. This makes it clear how large the model really is and why it uses so much RAM or VRAM.

It also means you can easily save and load checkpoints by writing or reading the flat array directly to disk. No need for complicated serialization formats.

Why It Matters

Understanding parameter layout helps you:

- See how the model's size explodes as you increase layers, heads, or channels.
- Debug memory issues by checking how big each slice is.
- Realize how much of training is just linear algebra on big arrays of floats.

This perspective is powerful because it demystifies deep learning: at its core, GPT-2 is just multiplying slices of one giant float array again and again.

Try It Yourself

1. Print parameter count: Add a line in the code to print `config.num_parameters`. Compare it with the table for GPT-2 Small/Medium/Large.
2. Inspect a slice: Print the first 10 numbers of the embedding table. They'll look random (from initialization).
3. Change precision: Modify the code to allocate `half` (FP16) instead of `float`. How much memory do you save?
4. Checkpoint peek: Save a checkpoint, then open it in a hex viewer. It's just raw floats-proof that parameters are stored flat.
5. Parameter scaling: Double the number of layers and see how `num_parameters` changes. Can you predict the increase?

The Takeaway

In *llm.c*, parameters are not hidden inside classes or objects. They live in one flat block of memory, sliced up by convention into embeddings, attention matrices, MLP weights, and norms. This design makes the relationship between model architecture and memory crystal clear-and reminds you that even a billion-parameter transformer is “just” a giant array of numbers.

23. Embedding Tables: Token + Positional

Before a transformer can reason about text, it first needs to turn tokens into vectors. In *llm.c*, this job is handled by the embedding tables: one for tokens, one for positions. These tables are the very first layer of GPT-2, and they transform plain integer IDs into continuous values that the neural network can process.

Token Embedding Table

When you feed in a batch of token IDs, the model looks up their corresponding vectors in the token embedding table.

- Shape: [vocab_size, channels]
 - vocab_size 50,257 (for GPT-2)
 - channels = hidden size (768 for GPT-2 Small)
- Each row corresponds to one token in the vocabulary.
- Each row is a dense vector of size channels.

So if your input batch has size (B, T), looking up embeddings gives you a tensor of shape (B, T, channels).

In the code, this is implemented as an array slice from the flat parameter block:

```
float* token_embedding_table = params; // first slice of parameters
```

At runtime, token IDs index directly into this table.

Positional Embedding Table

Transformers don't inherently know about word order. That's what positional embeddings are for.

- Shape: [max_seq_len, channels]
 - max_seq_len = 1024 in GPT-2 Small
 - Same channel dimension as token embeddings
- Each position (0, 1, 2, ..., 1023) has its own vector.

During training, when the model sees token *i* at position *j*, it takes the token embedding vector and adds the positional embedding vector for *j*. This gives the model both word identity and word position.

In *llm.c*, positional embeddings immediately follow the token embeddings in the flat parameter array.

Adding Them Together

The embedding layer’s forward pass is simple:

```
embedding_out[token, pos] = token_embedding[token] + positional_embedding[pos]
```

This results in a (B, T, channels) tensor that becomes the input to the first transformer block.

Why This Matters

Embeddings are the bridge between discrete tokens and continuous math. Without them, the model couldn’t use linear algebra to learn patterns. By adding positional embeddings, GPT-2 knows the difference between:

- “dog bites man” → **dog** comes first, **man** comes last
- “man bites dog” → same tokens, but swapped positions change the meaning

This small step is essential: order and identity must both be captured before attention can begin.

Try It Yourself

1. Inspect shapes: Print the sizes of the token and positional embedding tables during initialization. Confirm they match [vocab_size, channels] and [max_seq_len, channels].
2. Look at first rows: Print the first 5 vectors of the token embedding table. They should look like small random floats from initialization.
3. Change max_seq_len: Double max_seq_len in the config. How does this change the size of the positional table? Does training still work?
4. Overwrite embeddings: Try setting the token embedding table to all zeros. What happens to training loss?
5. Sampling experiment: After training a few steps, decode outputs without adding positional embeddings. Do the results become nonsensical or repetitive?

The Takeaway

The embedding tables are the foundation of GPT-2. Token embeddings give meaning to symbols, while positional embeddings give structure to sequences. In *llm.c*, they are just two slices of the flat parameter array, added together at the very start of the forward pass-but without them, the transformer would be blind to both words and order.

24. Attention Stack: QKV Projections and Geometry

After embeddings, the real magic of transformers begins: the attention mechanism. In GPT-2, every transformer block contains an attention stack. This is where the model learns how each token relates to others in the sequence-whether it's paying attention to the previous word, the beginning of a sentence, or even punctuation marks far away.

What Attention Does

Attention lets the model answer the question:

“Given the current word, which other words in the context should I care about, and how much?”

Instead of treating words independently, the model uses attention to build connections across the sequence.

The Q, K, V Projections

Each attention block starts with three linear projections:

Name	Shape	Purpose
Q (Query)	[channels, channels]	Represents what each token is <i>asking</i> about.
K (Key)	[channels, channels]	Represents how each token can be <i>recognized</i> .
V (Value)	[channels, channels]	Represents the actual <i>information</i> to pass along.

Here's the flow:

1. Each input vector (from embeddings or previous block) is multiplied by these three matrices to produce Q, K, and V vectors.

2. Attention scores are computed by comparing Qs with Ks.
3. These scores are used to weight the Vs, mixing information from other tokens into the current one.

Geometry of Attention

- Q and K define a similarity score: how well does this token match another one?
- V carries the actual features (like meaning, grammar cues).
- The result is a weighted sum: tokens borrow information from others based on attention scores.

In equations:

```
scores = Q × KT / sqrt(d_k)
weights = softmax(scores + mask)
output  = weights × V
```

The division by `sqrt(d_k)` normalizes scores so they don't blow up as dimensions grow.

Multi-Head Attention

GPT-2 doesn't use just one attention projection-it uses many in parallel, called heads. Each head learns to focus on different types of relationships:

- One head might track subject-verb agreement.
- Another might watch punctuation and quotes.
- Another might connect pronouns to their referents.

For GPT-2 Small:

- 12 heads per layer
- Each head works on a reduced dimension (`channels / num_heads`)
- Outputs are concatenated and projected back to `channels`

This setup is what gives transformers their flexibility.

Implementation in *llm.c*

In the parameter array, each transformer block has slices for Q, K, V, and output projection (O). During forward pass:

1. Multiply input by Q, K, V matrices.
2. Reshape into heads.
3. Compute attention scores (masked to prevent looking forward).
4. Apply softmax.
5. Multiply by V to get weighted values.
6. Concatenate heads and apply the O projection.

All of this is done with plain matrix multiplications and softmax calls—no magic beyond linear algebra.

Why It Matters

Attention is the beating heart of GPT-2. It’s how the model captures dependencies across text, from short-term grammar to long-range coherence. Without QKV, embeddings would stay isolated, and the model could never build context-aware representations.

Try It Yourself

1. Print shapes: Log the shapes of Q, K, V matrices in one layer. Confirm they match `[channels, channels]`.
2. Visualize scores: After a forward pass, print the attention weights for one head. Do they concentrate on recent tokens or spread across the sequence?
3. Reduce heads: Change `num_heads` from 12 to 4. What happens to validation loss?
4. Break symmetry: Initialize all Q, K, V matrices with zeros. Does training loss decrease at all?
5. Mask experiment: Disable the causal mask (allow looking ahead). Does the model “cheat” by predicting future tokens perfectly?

The Takeaway

The attention stack is where tokens stop being isolated and start talking to each other. Q, K, and V projections turn context into weighted relationships, and multi-head attention lets the model juggle many types of dependencies at once. In *llm.c*, this is implemented with straightforward linear algebra, making the most powerful idea in modern NLP visible and accessible.

25. MLP Block: Linear Layers + Activation

After attention mixes information across tokens, GPT-2 applies a second transformation inside each block: the MLP (Multi-Layer Perceptron). This part doesn't look at other tokens-it processes each position independently. But it's just as important because it gives the model extra capacity to transform and refine the hidden features before passing them to the next layer.

What the MLP Looks Like

Every transformer block contains an MLP with two linear layers and a nonlinear activation in between:

```
hidden = Linear1(x)
hidden = GELU(hidden)
out     = Linear2(hidden)
```

This structure expands the feature dimension and then compresses it back down, which lets the network learn richer representations.

Shapes of the Layers

If the hidden size (channels) is `d_model`, the MLP works as follows:

Step	Shape	Purpose
Input	[B, T, d_model]	Output of attention for each token.
Linear1	[d_model, 4 × d_model]	Expands features 4× wider.
GELU	elementwise	Introduces nonlinearity.
Linear2	[4 × d_model, d_model]	Projects back to original size.
Output	[B, T, d_model]	Same shape as input, ready for residual add.

For GPT-2 Small (`d_model` = 768), Linear1 expands to 3072 channels, then Linear2 reduces back to 768.

Activation: GELU

The activation function in GPT-2 is GELU (Gaussian Error Linear Unit). It's smoother than ReLU, giving the model a more nuanced way of handling values around zero. In code, GELU looks like:

```
float gelu(float x) {  
    return 0.5f * x * (1.0f + tanhf(0.79788456f * (x + 0.044715f * x * x * x)));  
}
```

This formula may look complicated, but the idea is simple: it smoothly squashes negative values toward zero and keeps positive values flowing through.

Why Expand and Shrink?

The expansion to $4 \times d_{\text{model}}$ may seem wasteful, but it's deliberate:

- Expanding gives the model more capacity to represent patterns at each token.
- Shrinking keeps the overall parameter count manageable.
- Together, they act like a bottleneck layer that forces the model to transform information more effectively.

This “expand \rightarrow activate \rightarrow shrink” design is one of the main reasons transformers scale so well.

Implementation in *llm.c*

Just like attention, the MLP parameters live in the flat array of floats. Each block stores two weight matrices and two bias vectors. During forward pass:

1. Multiply input by **Linear1** weights, add bias.
2. Apply GELU elementwise.
3. Multiply by **Linear2** weights, add bias.
4. Pass result through residual connection.

Because each position is processed independently, the MLP is easy to parallelize across tokens.

Why It Matters

The MLP is the nonlinear refiner of transformer blocks. Attention spreads information, but MLPs transform it in-place, giving the model more expressive power. Without the MLP, the network would be mostly linear, limiting its ability to capture complex patterns in text.

Try It Yourself

1. Print shapes: Log the dimensions of Linear1 and Linear2 weights in one block. Do they match [768, 3072] and [3072, 768] for GPT-2 Small?
2. Swap activation: Replace GELU with ReLU in the code. Does training still work? How does validation loss compare?
3. Reduce expansion: Change expansion from $4\times$ to $2\times$ ([768, 1536]). What effect does this have on parameter count and performance?
4. Zero out MLP: Set MLP weights to zero. Does the model still learn anything, or does performance collapse?
5. Compare speed: Measure training step time with and without the MLP enabled. How much slower is it?

The Takeaway

The MLP block in GPT-2 is a simple two-layer network with GELU activation, applied independently to each token. It expands, activates, and compresses features, giving the model nonlinear power to reshape hidden states. In *llm.c*, it's implemented with basic matrix multiplications and a smooth GELU function, proving that even small building blocks can have a big impact on the model's ability to learn language.

26. LayerNorm: Theory and Implementation (doc/layernorm)

Deep neural networks often suffer from unstable training if activations drift too high or too low. To stabilize this, GPT-2 uses Layer Normalization (LayerNorm) inside every transformer block. In *llm.c*, LayerNorm is implemented directly in C, and there's even a detailed explanation in the repo's `doc/layernorm` file to help learners understand how it works.

The Idea of Normalization

When you pass vectors through many layers, their values can become unbalanced-some features dominate while others shrink. Normalization fixes this by:

1. Centering: subtracting the mean of the vector.
2. Scaling: dividing by the standard deviation.

This makes every feature vector have mean 0 and variance 1, improving stability.

Why “Layer” Norm?

There are different kinds of normalization (BatchNorm, InstanceNorm, etc.). LayerNorm is special because:

- It normalizes across the features of a single token (the “layer”), not across the batch.
- This makes it independent of batch size, which is important for NLP where batch sizes can vary.

So if a hidden vector has 768 channels, LayerNorm computes the mean and variance over those 768 numbers for each token.

Trainable Parameters

LayerNorm isn’t just normalization-it also has two trainable vectors:

- (gamma): scales each feature after normalization.
- (beta): shifts each feature after normalization.

These allow the network to “undo” normalization when necessary, giving it flexibility.

Formula

For each input vector \mathbf{x} of size d :

```
mean = (1/d) *  $\Sigma$  x_i  
var  = (1/d) *  $\Sigma$  (x_i - mean)^2  
x_norm = (x - mean) / sqrt(var + eps)  
y = * x_norm +
```

Where `eps` is a tiny constant (like `1e-5`) to avoid dividing by zero.

Implementation in `llm.c`

In the code, LayerNorm is implemented as a simple function that loops over features, computes mean and variance, and applies the formula above. It’s not hidden inside a framework-it’s right there in C, so you can step through it line by line.

For example, the forward pass looks like this (simplified):

```

void layernorm_forward(float* out, float* inp, float* weight, float* bias, int N) {
    float mean = 0.0f, var = 0.0f;
    for (int i = 0; i < N; i++) mean += inp[i];
    mean /= N;
    for (int i = 0; i < N; i++) var += (inp[i] - mean) * (inp[i] - mean);
    var /= N;
    float inv_std = 1.0f / sqrtf(var + 1e-5f);
    for (int i = 0; i < N; i++) {
        out[i] = (inp[i] - mean) * inv_std * weight[i] + bias[i];
    }
}

```

This is the kind of clear, low-level implementation that makes *llm.c* educational.

Where It Fits in GPT-2

Each transformer block contains two LayerNorms:

- One before attention.
- One before the MLP.

GPT-2 uses Pre-LN architecture: inputs are normalized before each sublayer. This makes training more stable and gradients flow better.

Why It Matters

LayerNorm may look like a small detail, but without it, GPT-2 would fail to train reliably. It smooths out the flow of activations so attention and MLP layers can do their job. In practice, this is one of the critical “glue” components that makes deep transformers trainable at scale.

Try It Yourself

1. Print statistics: After applying LayerNorm, print the mean and variance of the output. Do they stay close to 0 and 1?
2. Remove `weight` and `bias`: Force gamma to 1 and beta to 0. Does the model still train? Compare losses.
3. Disable normalization: Comment out LayerNorm and train. How unstable does training become?
4. Compare positions: Try switching to Post-LN (apply normalization after attention/MLP). Does this change convergence speed?
5. Vary epsilon: Change `1e-5` to `1e-2` or `1e-8`. How sensitive is training?

The Takeaway

LayerNorm is the quiet stabilizer of GPT-2. It makes sure each token's features stay balanced, while `keep` keep flexibility. In *llm.c*, it's implemented directly with clear C code, letting you see exactly how normalization is calculated. It's a small but indispensable piece of the transformer puzzle.

27. Residual Connections: Keeping the Signal Flowing

Transformers like GPT-2 don't just stack layers on top of each other blindly. They use residual connections—a trick that allows the input of a layer to be added back to its output. This simple addition helps signals flow through the network without vanishing or exploding, and it makes training deep models possible.

The Basic Idea

Imagine you have a function $F(x)$ representing some transformation (like attention or an MLP). Instead of just computing:

$$y = F(x)$$

the transformer does:

$$y = F(x) + x$$

This means the layer learns only the *difference* it needs to add to the input, instead of replacing it entirely.

Why This Helps

Residuals solve two big problems in deep networks:

1. Gradient flow: During backpropagation, gradients can get smaller and smaller as they pass through many layers. Adding the input back ensures gradients always have a path straight through.
2. Information preservation: Even if $F(x)$ distorts the signal, the original x is still there. This prevents the model from “forgetting” important information.
3. Faster training: The network doesn't have to re-learn identity mappings—it can just pass them through the skip connection.

Implementation in *llm.c*

Residuals in *llm.c* are implemented as a straightforward elementwise addition:

```
void residual_forward(float* out, float* inp1, float* inp2, int N) {
    for (int i = 0; i < N; i++) {
        out[i] = inp1[i] + inp2[i];
    }
}
```

Here:

- `inp1` is the output of the layer (like attention).
- `inp2` is the original input.
- `out` is the combined result.

This is done for every token position and feature channel.

Where Residuals Are Used

In GPT-2, every transformer block has two residuals:

1. Attention residual: Adds the input of the attention layer to its output.
2. MLP residual: Adds the input of the MLP to its output.

So the data flowing through the network always carries both the new transformation and the original signal.

Why It Matters

Without residual connections, stacking 12–48 transformer blocks would be nearly impossible to train. Gradients would vanish, and the model would either stop learning or take forever to converge. Residuals let deep transformers scale smoothly.

They also add an intuitive interpretation: each block is like a “refinement step” rather than a full rewrite of the representation.

Try It Yourself

1. Remove residuals: Comment out the addition in the code. Does training collapse?
2. Scale residuals: Multiply the input by 0.5 before adding. Does this slow convergence?
3. Check loss curves: Compare training with and without residuals for the first 500 steps.
4. Inspect outputs: Print the norms of `inp1`, `inp2`, and `out`. Are the scales balanced?
5. Deeper models: Increase the number of layers from 12 to 24. Does the importance of residuals become more obvious?

The Takeaway

Residual connections are the “lifeline” of deep transformers. By simply adding inputs back into outputs, they make it possible to train very deep networks without losing gradients or information. In *llm.c*, the implementation is as simple as looping over arrays and adding them-but the effect is profound: residuals are what let GPT-2 go deep and still work.

28. Attention Masking: Enforcing Causality

One of the defining traits of GPT-2 is that it’s a causal language model. That means it predicts the *next* token given all the tokens before it, but never cheats by looking ahead. To enforce this, GPT-2 applies an attention mask inside every attention layer.

Why a Mask Is Needed

Without a mask, attention is free to connect any token to any other, including future ones. For example:

- Input: “The cat sat on the”
- Target: “mat”

If the model could peek at “mat” while computing attention, the task would be trivial-it could just copy the next word. That would break the training objective.

The mask forces the model to only use tokens at or before the current position when making predictions.

How the Mask Works

When computing attention scores ($Q \times K^T / \sqrt{d_k}$), the result is a matrix of size $[T, T]$ where each row corresponds to one token attending to all others.

The mask modifies this matrix:

- Allowed positions (past and present): keep scores as is.
- Disallowed positions (future): set scores to $-\text{inf}$.

After applying softmax, those $-\text{inf}$ entries become zero probability, effectively blocking attention to the future.

Implementation in *llm.c*

The causal mask is applied during the attention forward pass. The code uses a loop to zero out invalid positions:

```
for (int t = 0; t < T; t++) {  
    for (int u = t + 1; u < T; u++) {  
        scores[t][u] = -1e9; // block future positions  
    }  
}
```

Here T is the sequence length. This ensures that token t can only attend to itself and earlier tokens.

Visualizing the Mask

Think of the mask as a triangular matrix:

	0	1	2	3
0				
1				
2				
3				

Each row shows which past tokens a given position can look at. Future positions remain blank.

Why It Matters

The mask is what makes GPT-2 a predictive model instead of a bidirectional encoder like BERT. Without it, the model could “cheat” and the training objective would no longer match how it’s used at inference time (generating text step by step).

This small detail—just filling part of a matrix with `-inf`—is critical to making autoregressive text generation possible.

Try It Yourself

1. Disable the mask: Comment out the masking code. Watch validation loss drop unrealistically, then notice that text generation produces garbage.
2. Reverse the mask: Block the past and allow the future. Does the model still train? What does it predict?
3. Partial mask: Only allow attention to the previous 5 tokens (a sliding window). How does this affect learning long-range structure?
4. Print scores: Before and after masking, log a row of attention scores. Notice how future positions become huge negatives.
5. Visualize: Write a small script to plot the attention mask as a matrix. It should look strictly lower-triangular.

The Takeaway

Attention masking is a simple but essential trick. By filling future positions with `-inf` before softmax, GPT-2 ensures that each token can only attend to its past. In *llm.c*, this is implemented with just a couple of loops—but it’s what turns a generic transformer into a true causal language model.

29. Output Head: From Hidden States to Vocabulary

After tokens pass through embeddings, attention, MLPs, LayerNorm, and residuals, we end up with hidden states for every position in the sequence. But GPT-2’s final job is not to output vectors—it must predict the next token from the vocabulary. This is handled by the output head, the last stage of the model.

What the Output Head Does

The output head maps hidden states of shape (B, T, channels) into logits of shape (B, T, vocab_size). Each logit represents the model’s “raw score” for how likely a particular token is at the next step.

The pipeline looks like this:

hidden states → Linear projection → Logits → Softmax → Probabilities

- Logits: real numbers, one per token in the vocabulary.
- Softmax: converts logits into probabilities that sum to 1.
- Predicted token: the token with the highest probability (or sampled from the distribution).

Tied Weights with Embeddings

In GPT-2, the token embedding table and the output head share weights. This means the same matrix is used both for:

- Mapping tokens to vectors at the start (embedding lookup).
- Mapping vectors back to tokens at the end (output head).

Mathematically, this improves efficiency and helps align input and output representations.

In *llm.c*, this is done by simply pointing both embedding and output head to the same parameter slice.

```
// token embedding table
float* token_embedding_table = params;
// output head reuses the same memory
float* output_head = token_embedding_table;
```

When the model projects hidden states back to vocab space, it does a matrix multiply with this shared matrix.

Shapes in Action

For GPT-2 Small:

- Hidden states: [B, T, 768]
- Output projection (embedding transpose): [768, 50257]
- Logits: [B, T, 50257]

That’s more than 50k scores per position, one for each token in the vocabulary.

Why Weight Tying Helps

1. Memory efficiency: You don't need a separate giant matrix for the output head.
2. Better learning: The same vectors that represent tokens going in also represent them going out, which reinforces consistency.
3. Simpler code: Just reuse the same parameter slice.

This trick is why GPT-2 can scale vocab sizes without blowing up parameter counts too much.

Why It Matters

The output head is where everything comes together. For each position, the model collapses its hidden representation into a distribution over possible next tokens. This is how GPT-2 generates text one step at a time. Without this step, you'd only have abstract hidden states-useful internally, but not something you can read.

Try It Yourself

1. Print logits: After a forward pass, print the logits for the last token. Do they look like random floats at initialization?
2. Check probability sum: Apply softmax to logits and verify the probabilities sum to 1.
3. Untie weights: Make the output head its own matrix instead of reusing embeddings. Does training still work? How does the parameter count change?
4. Top-k sampling: Modify sampling to keep only the top 5 logits before softmax. What kind of text does this produce?
5. Greedy vs random: Compare greedy decoding (argmax) vs random sampling from probabilities. Which one gives more interesting outputs?

The Takeaway

The output head is the final bridge between hidden vectors and actual words. By reusing the token embedding matrix, GPT-2 projects hidden states back into vocabulary space and produces logits for every possible token. In *llm.c*, this step is just another matrix multiplication-but it's the one that turns internal math into real text predictions.

30. Loss Function: Cross-Entropy over Vocabulary

Training GPT-2 means teaching it to predict the next token in a sequence. To measure how well it's doing, we need a loss function that compares the model's predicted probabilities with the true token IDs. In *llm.c*, this is done with the cross-entropy loss—a standard choice for classification tasks.

From Logits to Probabilities

After the output head, we have logits of shape (B, T, vocab_size). These are raw scores. To turn them into probabilities:

```
probs = softmax(logits)
```

Softmax ensures two things:

- All values are between 0 and 1.
- They sum to 1 across the vocabulary.

So for each position, you get a probability distribution over all possible next tokens.

Cross-Entropy Definition

Cross-entropy compares the predicted distribution p with the true distribution q . For language modeling:

- q is a one-hot vector (all zeros, except 1 at the true token index).
- p is the probability vector from softmax.

The formula for one token:

```
loss = -log(p[true_token])
```

For a batch, you average across all tokens in all sequences.

Implementation in *llm.c*

In C, this boils down to:

```

float loss = 0.0f;
for (int b = 0; b < B; b++) {
    for (int t = 0; t < T; t++) {
        int target = targets[b*T + t];
        float logit_max = -1e9;
        for (int v = 0; v < vocab_size; v++) {
            if (logits[b*T*vocab_size + t*vocab_size + v] > logit_max) {
                logit_max = logits[b*T*vocab_size + t*vocab_size + v];
            }
        }
        // compute softmax denominator
        float sum = 0.0f;
        for (int v = 0; v < vocab_size; v++) {
            sum += expf(logits[b*T*vocab_size + t*vocab_size + v] - logit_max);
        }
        float logprob = logits[b*T*vocab_size + t*vocab_size + target] - logit_max - logf(sum);
        loss += -logprob;
    }
}
loss /= (B * T);

```

This snippet shows how *llm.c* explicitly computes softmax and cross-entropy in loops. No black boxes—just raw math.

Intuition

- If the model assigns high probability to the correct token \rightarrow loss is small.
- If the model assigns low probability to the correct token \rightarrow loss is large.
- Minimizing loss means pushing probability mass toward the right answers.

Why Cross-Entropy Works for Language

Language modeling is essentially a huge multi-class classification problem: at each step, which word comes next? Cross-entropy is perfect here because it directly penalizes wrong predictions proportional to how confident the model was.

Why It Matters

The loss function is the only signal the model gets about how well it's doing. Everything else—parameter updates, weight tuning, learning dynamics—flows from this single number. A

well-implemented cross-entropy ensures training is stable and meaningful.

Try It Yourself

1. Check values: Print the loss after the first few steps. It should be close to $\log(\text{vocab_size})$ (10.8 for 50k vocab) before training.
2. Overfit tiny batch: Train on just one sequence. Does the loss go near 0 after enough steps?
3. Change target: Replace the true token with a random one. Does the loss increase immediately?
4. Compare vocab sizes: Train with a smaller vocabulary (e.g., 100 tokens). Does initial loss drop to $\log(100)$ 4.6?
5. Inspect probabilities: For one token, print the top 5 predicted probabilities. Does the true token climb to the top as training progresses?

The Takeaway

The cross-entropy loss is the compass guiding GPT-2 during training. It turns raw logits into probabilities and measures how well the model predicts the correct next token. In *llm.c*, it's implemented with explicit loops and math, letting you see exactly how probabilities and losses are computed. Without this step, the model would have no way to learn from its mistakes.

Chapter 4. CPU Inference (Forward Only)

31. Forward Pass Walkthrough

When we talk about the *forward pass* in GPT-2, we mean the process of turning an input sentence (like “The cat sat on the”) into predictions for the next word. In simple terms, it's how the model “thinks” before giving an answer. In `train_gpt2.c`, this happens inside the function `gpt2_forward`. Let's walk through it slowly, step by step, so you can see how numbers flow through the model and transform along the way.

1. From Words to Numbers

Computers don't understand words like *cat* or *sat*. They only understand numbers. Before the forward pass starts, text is already tokenized into IDs (integers). For example:

"The cat sat" → [464, 3290, 616]

Each number is a token ID. The model doesn't yet know what "464" means in plain English-it just knows it's a number that points into a table.

2. Embedding: Giving Words Meaning

The first real step in the forward pass is embedding lookup. Imagine we have a huge dictionary, but instead of definitions in English, each word ID points to a long vector of numbers (say, 768 numbers for GPT-2 small).

- Word embeddings (**wte**): Each token ID becomes a vector that captures the meaning of the word.
- Position embeddings (**wpe**): Each token also gets a vector for its position: first word, second word, third word, etc.

The model adds these two vectors together. This way, it knows not just what the word is, but also where it is in the sentence.

For example:

To-ken	Word	Word Embedding (shortened)	Position Embedding (shortened)	Combined Vector
464	"The"	[0.2, -0.5, 0.1, ...]	[0.0, 0.1, -0.3, ...]	[0.2, -0.4, -0.2, ...]
3290	"cat"	[0.9, -0.2, 0.4, ...]	[0.1, -0.1, -0.2, ...]	[1.0, -0.3, 0.2, ...]

Now every token is a vector with both meaning and position built in.

3. Transformer Layers: The Thinking Steps

GPT-2 has multiple identical layers stacked on top of each other. Each layer has two big parts: attention and MLP (feed-forward network).

Attention (looking around):

- Each word asks: "Which other words should I pay attention to right now?"
- For "sat," attention might focus heavily on "cat," because those words are related.
- The code computes *queries*, *keys*, and *values* for every word, then does dot-products, softmax, and weighted sums to mix information.

MLP (processing deeply):

- After attention, each token passes through a mini neural network (two matrix multiplications with a nonlinear GELU function in between).

- This helps each word refine its understanding, even if it doesn't directly attend to another word.

Both blocks have residual connections: the input is added back to the output, like keeping the original notes while adding new insights. This prevents information loss.

4. Normalization: Keeping Numbers Stable

At many points, the model normalizes vectors so they don't explode in size or shrink too small. This is called LayerNorm. It ensures training is stable, like making sure your cooking pot doesn't boil over or dry out.

5. The Final Prediction Layer

After all layers, the model produces a final vector for each position. Then:

- It multiplies those vectors by the embedding table again (but transposed).
- This gives logits: raw scores for each word in the vocabulary (about 50k options).

Example: for the last token “on the,” the logits might be:

Word	Logit	Probability (after softmax)
“mat”	7.2	0.85
“dog”	5.1	0.10
“car”	3.0	0.05

The highest probability is “mat.”

6. Softmax: Turning Scores into Probabilities

The logits are big numbers, but they don't mean much until we apply softmax. Softmax makes them into probabilities that sum to 1. This way, we can interpret them as chances: “There's an 85% chance the next word is *mat*.”

7. Cross-Entropy Loss: Measuring Mistakes

If we're training, we also give the model the correct next word. The model checks how much probability it gave to that word. If it gave it high probability, the loss is low. If it gave it low probability, the loss is high.

- Correct: “mat” (probability 0.85 \rightarrow loss 0.16, small).
- Wrong: “car” (probability 0.05 \rightarrow loss 3.0, large).

This loss is averaged across all tokens, and it's the signal that tells the backward pass how to update the model.

8. Why It Matters

The forward pass is the part of GPT-2 that generates predictions. Without it, the model can't “think” or make sense of input. It's like the brain processing sensory input before deciding what to do. In `train_gpt2.c`, the forward pass is written with plain C loops, which makes the math crystal clear instead of hidden inside deep learning libraries.

9. Try It Yourself

1. Print embeddings: Modify the code to print the vector for the first token. See how it's just numbers, but those numbers are the “meaning” of the word.
2. Inspect probabilities: After the forward pass, print the softmax probabilities for one position. They should sum to 1.0.
3. Change sequence length: Increase `T` from 64 to 128. Notice how validation slows down, because attention compares all tokens with all others (T^2 scaling).
4. Baseline loss: Before training, measure the loss. It should be around `log(vocab_size)` (10.8 for GPT-2 small). That's the loss of random guessing.
5. Mask experiment: Temporarily remove the causal mask in attention. The model will “cheat” by looking ahead, and loss will drop unrealistically.

The Takeaway

The forward pass is like the thought process of GPT-2. Input words become vectors, vectors mix through attention and MLPs, everything gets normalized, and finally the model produces probabilities for the next word. It's a carefully choreographed dance of math operations, all coded in plain C loops in `train_gpt2.c`. Once you understand this flow, you can follow exactly how GPT-2 turns raw tokens into intelligent predictions.

32. Token and Positional Embedding Lookup

Before GPT-2 can do anything intelligent with text, it needs to turn raw numbers (token IDs) into vectors that capture meaning and context. This is the role of embeddings. In `train_gpt2.c`, this step is handled by the function `encoder_forward`. Let's take a closer look at how it works and why it matters.

Tokens Are Just Numbers

Suppose you type:

```
"The cat sat on the mat."
```

After tokenization, this sentence might look like:

```
[464, 3290, 616, 319, 262, 1142, 13]
```

These are just IDs. The model doesn't inherently know that 3290 means "cat." It only knows it needs to use these numbers to fetch vectors from a table.

The Embedding Tables

The model has two important tables stored in memory:

1. Word Token Embeddings (`wte`)
 - Size: (`V`, `C`) where `V` is vocab size (~50,000 for GPT-2 small) and `C` is channels (768).
 - Each row corresponds to a token ID.
 - Example: row 3290 might be `[0.12, -0.45, 0.88, ...]`.
2. Positional Embeddings (`wpe`)
 - Size: (`maxT`, `C`) where `maxT` is the maximum sequence length (e.g. 1024).
 - Each row corresponds to a position index: 0 for the first token, 1 for the second, etc.
 - Example: position 2 might be `[0.07, 0.31, -0.22, ...]`.

Both tables are filled with trainable values. At the start, they're random. As training progresses, the optimizer updates them so they encode useful patterns.

Adding Them Together

For each token at position t :

- Look up its word vector from wte .
- Look up its position vector from wpe .
- Add them elementwise.

This gives a final vector of size C that represents what the token is and where it is.

Example with simplified numbers:

Token ID	Word	Word Embedding	Position	Combined
464	The	[0.1, -0.2, 0.3]	[0.2, 0.0, -0.1]	[0.3, -0.2, 0.2]
3290	cat	[0.4, 0.5, -0.3]	[0.0, 0.1, 0.2]	[0.4, 0.6, -0.1]

Now the vector doesn't just mean "cat," it means "cat at position 1."

Why Position Matters

Without positions, the model would treat:

- "The cat sat"
- "Sat cat the"

as identical, because they use the same tokens. But word order is essential in language. By adding positional embeddings, GPT-2 knows the difference between "dog bites man" and "man bites dog."

Inside the Code

The embedding lookup is written explicitly with loops in C:

```
for (int b = 0; b < B; b++) {
    for (int t = 0; t < T; t++) {
        float* out_bt = out + b * T * C + t * C;
        int ix = inp[b * T + t];
        float* wte_ix = wte + ix * C;
        float* wpe_t = wpe + t * C;
        for (int i = 0; i < C; i++) {
            out_bt[i] = wte_ix[i] + wpe_t[i];
        }
    }
}
```

```
    }  
  }  
}
```

What's happening here:

- Loop over batches (**b**) and sequence positions (**t**).
- Find the token ID **ix**.
- Fetch its embedding **wte_ix**.
- Fetch its position embedding **wpe_t**.
- Add them element by element.

The result, **out_bt**, is the vector for this token at this position.

Analogy

Think of it like name tags at a conference:

- The word embedding is your name: "Alice."
- The position embedding is your table number: "Table 7."
- Together, they tell the conference staff who you are and where you are seated.

Without the table number, they might know who you are but not where to find you. Without your name, they just know there's someone at Table 7 but not who. Both are needed for proper context.

Why It Matters

Embeddings are the foundation of the whole model. If this step is wrong, everything else collapses. They transform meaningless IDs into rich vectors that carry semantic and positional information. This is the entry point where language starts becoming something a neural network can reason about.

Try It Yourself

1. Print a token embedding: Modify the code to print out **wte_ix** for a specific token ID like "cat." You'll see a vector of floats, the learned representation.
2. Print a position embedding: Do the same for **wpe_t** at position 0, 1, 2... Notice how positions have unique but consistent patterns.
3. Check the sum: Verify that **out_bt[i] = wte_ix[i] + wpe_t[i]**. This is literally how word and position are fused.

4. Shuffle words: Try feeding “cat sat” vs. “sat cat.” The embeddings will differ because the position vectors change, even though the words are the same.
5. Observe growth during training: After some training steps, dump the embeddings again. You’ll notice they stop being random and start showing structure.

The Takeaway

The embedding lookup is the very first step of the forward pass. It takes raw numbers and makes them meaningful by combining token identity and position. This prepares the input for the deeper transformer layers. Even though the C code looks simple-a few nested loops-it’s doing the crucial work of giving words a mathematical shape the model can understand.

33. Attention: Matmuls, Masking, and Softmax on CPU

The attention mechanism is the heart of GPT-2. It’s where each word in the input sequence decides which other words to look at when forming its representation. In `train_gpt2.c`, this happens inside the `attention_forward` function, which implements multi-head self-attention using plain C loops and matrix multiplications. Let’s break it down carefully, step by step, so even an absolute beginner can follow the flow.

The Big Idea of Attention

Imagine you’re reading:

“The cat sat on the mat.”

When the model is trying to understand the word *sat*, it doesn’t just look at *sat* by itself. It wants to consider other words like *cat* (the subject) and *mat* (likely the object). Attention gives each token a way to “consult” earlier tokens and decide how important they are.

This is done mathematically by projecting each token into three roles: Query (Q), Key (K), and Value (V).

- Query (Q): “What am I looking for?”
- Key (K): “What do I offer?”
- Value (V): “What information do I carry?”

Step 1: Creating Q, K, and V

For every input vector of size C (e.g., 768), the code performs three separate linear projections (matrix multiplications). These produce Q , K , and V vectors of smaller size, divided among attention heads.

In the code:

```
matmul_forward(acts.q, acts.ln1, params.wq, params.bq, B, T, C, C);
matmul_forward(acts.k, acts.ln1, params.wk, params.bk, B, T, C, C);
matmul_forward(acts.v, acts.ln1, params.wv, params.bv, B, T, C, C);
```

Here:

- `acts.ln1` is the normalized input from the previous step.
- `params.wq`, `params.wk`, `params.wv` are the weight matrices.
- The output shapes are (B, T, C) .

So each token now has three new representations: Q , K , and V .

Step 2: Computing Attention Scores

For each token at position t , we want to know how much it should pay attention to every earlier token (including itself). This is done with a dot product between its Query and all Keys.

Mathematically:

$$\text{score}[t][u] = (Q[t] \cdot K[u]) / \sqrt{d_k}$$

- t = current token.
- u = another token at or before t .
- $\sqrt{d_k}$ is a scaling factor (d_k = size of each head) to keep values stable.

In the code, these dot products are done explicitly in loops.

Step 3: Applying the Causal Mask

GPT-2 is an autoregressive model, meaning it only predicts the future from the past, not the other way around. To enforce this, the attention matrix is masked:

- Token at position t can only look at positions $\leq t$.
- Anything beyond t is set to a very negative value ($-1e9$), which becomes effectively zero after softmax.

This ensures, for example, that when predicting the 3rd word, the model doesn't cheat by looking at the 4th.

Step 4: Turning Scores into Probabilities

The scores are raw numbers that can be large and unstable. To convert them into meaningful weights, the code applies softmax:

```
attention_weights[t][u] = exp(score[t][u]) /  $\sum$  exp(score[t][v])
```

This makes all weights positive and ensures they sum to 1. Now each token has a probability distribution over earlier tokens.

Example for the word *sat*:

Attended Token	Raw Score	After Softmax
The	1.2	0.10
cat	3.4	0.80
sat (itself)	0.7	0.10
on	masked	0.00

Clearly, *sat* focuses most strongly on *cat*.

Step 5: Weighted Sum of Values

Once the attention weights are computed, the model uses them to take a weighted sum of the Value vectors:

```
output[t] =  $\sum$  attention_weights[t][u] * V[u]
```

This produces a new representation for each token that blends in information from others.

For *sat*, its new vector will be mostly influenced by *cat*, but also a little by *The* and itself.

Step 6: Multi-Head Attention

In practice, attention is split into multiple heads (12 for GPT-2 small). Each head works on smaller chunks of the vector (C/heads).

- Head 1 might focus on subject–verb relationships.
- Head 2 might track distances (like “how far back was this token?”).
- Head 3 might specialize in punctuation.

After all heads compute their outputs, the results are concatenated and projected back into size C with another matrix multiply.

Step 7: Residual Connection

Finally, the output of the attention block is added back to the original input (residual connection). This keeps the original signal flowing, even if the attention introduces distortions.

```
residual_forward(acts.residual2, acts.ln1, acts.att, B*T, C);
```

This ensures information isn’t lost and gradients flow smoothly during training.

Why It Matters

Attention is the mechanism that lets GPT-2 capture relationships between words. Without it, the model would treat each token independently, losing context. By explicitly computing “who should I look at?” for every token, GPT-2 learns patterns like subject–verb agreement, long-distance dependencies, and even stylistic nuances.

Try It Yourself

1. Inspect the attention mask: Print out the scores before and after masking. Notice how future tokens are set to huge negative values.
2. Visualize weights: Run attention on a short sentence and plot the weights. You’ll see which words attend to which.
3. Change sequence length: Try increasing T and observe how computation grows quadratically (T^2). Attention is expensive!
4. Experiment with heads: Force the model to use only 1 head instead of 12. See how this limits the diversity of patterns it can capture.
5. Check sum of weights: For one token, verify that all attention weights add up to 1.0 after softmax.

The Takeaway

Attention is what makes transformers powerful. It allows each word to dynamically decide which other words matter for understanding its role in a sentence. In `train_gpt2.c`, this process is spelled out with explicit loops and matrix multiplications, so you can follow every step of the math. Understanding this section gives you the key to why GPT-2-and all modern LLMs-work so well.

34. MLP: GEMMs and Activation Functions

After the attention block lets tokens “talk to each other,” GPT-2 applies a second kind of transformation called the MLP block (multi-layer perceptron). Unlike attention, which mixes information between tokens, the MLP processes each token independently, enriching its internal representation. Even though it looks simpler than attention, the MLP is essential for capturing complex relationships in language.

What the MLP Does

Every token’s vector (size C , e.g., 768 for GPT-2 small) goes through:

1. Linear expansion: project from size C to size $4C$ (3072 in GPT-2 small).
2. Nonlinear activation: apply the GELU function, which adds flexibility.
3. Linear projection back: reduce size from $4C$ back to C .
4. Residual connection: add the input vector back to the output, keeping the original signal intact.

This allows the model to not only share information between tokens (via attention) but also refine how each token represents itself.

Step 1: Expanding with a Matrix Multiply

The first step is to expand each token’s vector from 768 to 3072 dimensions. This is done with a general matrix multiply (GEMM):

```
matmul_forward(acts.mlp_in, acts.ln2, params.wfc, params.bfc, B, T, C, 4*C);
```

- `acts.ln2`: the normalized input from the previous residual.
- `params.wfc`: the weight matrix of size $(C, 4C)$.
- `params.bfc`: bias vector of size $(4C)$.
- `acts.mlp_in`: the result, shape $(B, T, 4C)$.

Think of it like stretching a rubber band-suddenly, the token has much more room to express richer features.

Step 2: GELU Activation

After expansion, each number passes through GELU (Gaussian Error Linear Unit).

The formula:

$$\text{GELU}(x) = 0.5 * x * (1 + \tanh(\sqrt{2/\pi} * (x + 0.044715 * x^3)))$$

This looks complicated, but the key idea is:

- For small negative numbers, output 0 (ignore weak signals).
- For large positive numbers, output x (pass strong signals).
- For numbers in between, it smoothly blends.

Unlike ReLU, which just chops off negatives, GELU lets small signals through in a probabilistic way. This makes it better for language, where even small hints matter.

Analogy: Imagine you're grading homework. If an answer is completely wrong, you give 0 points (ReLU style). If it's perfect, you give full credit. But if it's partially right, you give partial credit. GELU behaves like that-soft, nuanced grading.

Step 3: Projecting Back Down

Once the token vector has been expanded and passed through GELU, it's projected back to the original size C:

```
matmul_forward(acts.mlp_out, acts.mlp_in_gelu, params.wproj, params.bproj, B, T, 4*C, C);
```

- `params.wproj`: the projection weights, size (4C, C).
- `params.bproj`: bias, size (C).
- `acts.mlp_out`: result, shape (B, T, C).

Now each token has gone through a non-linear “thinking step,” mixing and reshaping features.

Step 4: Residual Connection

Just like with attention, the MLP output is added back to the input:

```
residual_forward(acts.residual3, acts.residual2, acts.mlp_out, B*T, C);
```

This means the token keeps its old representation while adding the new refinements. If the MLP makes a mistake early in training, the residual ensures the token doesn't lose all its meaning.

Inside the Code: Simplicity

Even though MLPs in deep learning libraries like PyTorch are one-liners (`nn.Linear + nn.GELU + nn.Linear`), here in C you see every step spelled out:

- First GEMM expands to 4C.
- Loop applies GELU element by element.
- Second GEMM projects back to C.
- Residual adds input and output.

It's like watching a magician reveal the trick instead of just seeing the final illusion.

Why the Expansion Matters

You might ask: why expand to 4C and then shrink back? Why not just keep the size the same?

The expansion allows the model to capture more complicated combinations of features. By spreading information out, applying a nonlinear transformation, and then compressing it again, the model can discover patterns that wouldn't fit in the smaller space.

Think of it like brainstorming on a huge whiteboard. You spread out all your ideas (4C), reorganize them, and then condense the best ones into a neat summary (C).

Example Walkthrough

Let's say we're processing the token "cat" in the sentence *The cat sat*.

1. Input vector (size 768): [0.12, -0.08, 0.33, ...]
2. After first matrix multiply: expanded to [1.2, -0.9, 0.5, ...] (size 3072).
3. After GELU: [1.1, -0.0, 0.4, ...] (smooth nonlinearity).
4. After projection: back to [0.15, -0.02, 0.27, ...] (size 768).
5. Add back original input: [0.27, -0.10, 0.60, ...].

Now "cat" has been enriched with new internal features that help the model predict what comes next.

Why It Matters

The MLP is the part of GPT-2 that lets each token refine itself. Attention gives context from neighbors, but the MLP deepens the representation of the token itself. Without it, the model would lack the ability to detect fine-grained patterns.

Try It Yourself

1. Print intermediate sizes: Add debug prints to see how token vectors grow to 4C and shrink back to C.
2. Swap activation: Replace GELU with ReLU in the code and train. Compare losses-you'll notice GPT-2 prefers GELU.
3. Disable residual: Temporarily remove the residual add. Watch how the model struggles to learn, because it can't preserve information.
4. Visualize values: Track how many values are near 0 before and after GELU. You'll see GELU softly zeroes out weak signals.
5. Smaller expansion: Try changing 4C to 2C in the code. You'll save memory but lose accuracy, since the MLP has less expressive power.

The Takeaway

The MLP block is a token's personal deep thinker. It stretches the representation wide, filters it through GELU, compresses it again, and then adds it back to the original. While attention handles the conversation between words, the MLP ensures each word processes and refines its own role. Together, they create the layered reasoning ability that makes GPT-2 so powerful.

35. LayerNorm on CPU (Step-by-Step)

One of the most important but often overlooked ingredients in GPT-2 is Layer Normalization, or LayerNorm for short. While attention and MLPs are the big stars, LayerNorm is like the stage crew keeping everything running smoothly behind the scenes. It ensures the numbers flowing through the network stay stable and balanced, preventing explosions or collapses that could make training impossible. In `train_gpt2.c`, LayerNorm is implemented with explicit loops so you can see every calculation. Let's walk through it carefully.

Why Do We Need Normalization?

Imagine a classroom where every student talks at different volumes. Some whisper, some shout. If you try to listen to all of them at once, the loud voices drown out the quiet ones.

Neural networks face a similar problem. The outputs of layers can have wildly different scales. If one dimension of a vector is much larger than the others, it dominates. Training becomes unstable, and gradients may vanish or explode.

LayerNorm fixes this by ensuring that, for each token at each layer, the vector has:

- Mean = 0 (centered around zero)
- Variance = 1 (consistent spread of values)

After that, trainable parameters scale and shift the result so the model can still learn flexible transformations.

The Math Behind LayerNorm

For a given token vector x of size C (e.g., 768):

1. Compute mean:

$$\mu = (1/C) * \sum x[i]$$

2. Compute variance:

$$\sigma^2 = (1/C) * \sum (x[i] - \mu)^2$$

3. Normalize:

$$x_{\text{norm}}[i] = (x[i] - \mu) / \sqrt{\sigma^2 + \epsilon}$$

where ϵ is a tiny constant (like $1e-5$) to avoid division by zero.

4. Scale and shift with trainable weights g (gamma) and b (beta):

$$y[i] = g[i] * x_{\text{norm}}[i] + b[i]$$

So the final output has controlled statistics, but still enough flexibility for the model to adjust.

The Code in `train_gpt2.c`

Here's a simplified version from the repository:

```

void layernorm_forward(float* out, float* inp, float* weight, float* bias, int B, int T, int C) {
    for (int b = 0; b < B; b++) {
        for (int t = 0; t < T; t++) {
            float* x = inp + b*T*C + t*C;
            float* o = out + b*T*C + t*C;

            // mean
            float mean = 0.0f;
            for (int i = 0; i < C; i++) mean += x[i];
            mean /= C;

            // variance
            float var = 0.0f;
            for (int i = 0; i < C; i++) {
                float diff = x[i] - mean;
                var += diff * diff;
            }
            var /= C;

            // normalize, scale, shift
            for (int i = 0; i < C; i++) {
                float norm = (x[i] - mean) / sqrtf(var + 1e-5f);
                o[i] = norm * weight[i] + bias[i];
            }
        }
    }
}

```

Notice the structure:

- Outer loops over batch B and sequence length T.
- Inner loops compute mean, variance, and then apply normalization per token vector of length C.
- **weight** and **bias** are the learnable gamma and beta.

This is exactly what LayerNorm means: normalize *each layer's inputs per token*.

Example Walkthrough

Suppose we have a single token vector ($C=4$) = [2.0, -1.0, 3.0, 0.0].

1. Mean: $(2 - 1 + 3 + 0)/4 = 1.0$

2. Variance: $((2-1)^2 + (-1-1)^2 + (3-1)^2 + (0-1)^2)/4 = (1 + 4 + 4 + 1)/4 = 2.5$

3. Normalize: subtract mean and divide by $\sqrt{2.5}$:

[$(2-1)/1.58$, $(-1-1)/1.58$, $(3-1)/1.58$, $(0-1)/1.58$]
= [0.63, -1.26, 1.26, -0.63]

4. Scale and shift (say $\text{weight}=[1,1,1,1]$, $\text{bias}=[0,0,0,0]$):

[0.63, -1.26, 1.26, -0.63]

Now the vector has mean 0, variance 1, and is ready for the next layer.

Analogy

Think of LayerNorm like a baking recipe. If one ingredient is way too strong (like adding five times too much salt), the whole dish is ruined. LayerNorm tastes the mixture, balances all the flavors, and then lets you adjust the seasoning with learnable gamma (scale) and beta (shift).

Why It Matters

Without LayerNorm, the model would quickly become unstable:

- Some tokens would dominate, while others fade.
- Gradients could explode, making loss jump wildly.
- Training would be inconsistent between batches.

With LayerNorm, each layer works with clean, normalized inputs. This allows deeper stacks of attention and MLP blocks to learn reliably.

Try It Yourself

1. Print statistics: Add debug code to check the mean and variance before and after LayerNorm. Before: mean 0, variance 1. After: mean 0, variance 1.
2. Remove LayerNorm: Comment out LayerNorm in the code. Watch training collapse-loss will not decrease properly.
3. Change epsilon: Try making $\epsilon = 1e-1$ or $\epsilon = 1e-12$. See how too large or too small values can break stability.
4. Observe gamma and beta: Initialize $\text{gamma}=1$, $\text{beta}=0$. During training, watch how these parameters drift, fine-tuning normalization.
5. Experiment with batch norm: Replace LayerNorm with BatchNorm (not typical for transformers). You'll see it doesn't work well, because transformers process variable-length sequences where per-batch statistics vary too much.

The Takeaway

LayerNorm is the quiet but critical stabilizer in GPT-2. It ensures every token vector is balanced, centered, and scaled before moving into attention or MLP. In `train_gpt2.c`, you see exactly how it works: compute mean, compute variance, normalize, then scale and shift. Even though it's just a few lines of C code, it's one of the main reasons deep transformers can stack dozens of layers without breaking.

36. Residual Adds and Signal Flow

Once embeddings, attention, and MLP blocks are computed, there's still one piece left to keep the whole network stable and effective: residual connections. In `train_gpt2.c`, these appear in functions like `residual_forward`, where outputs of a layer are added back to their inputs. This simple-looking step is one of the key reasons GPT-2 and other deep transformer models can stack many layers without collapsing.

The Core Idea

A residual connection says:

```
output = input + transformation(input)
```

Instead of replacing the old representation with the new one, the model adds them together. That way, the original signal always survives, even if the new transformation is noisy or imperfect.

Think of it like taking lecture notes. Each time the teacher explains more, you don't throw away your old notes. You add new details next to them. That way, you preserve everything learned so far, while layering new insights on top.

Why Residuals Are Crucial

1. Preventing information loss: If you only applied transformations, some features might vanish forever. Adding the input back ensures no information is lost.
2. Helping gradients flow: During backpropagation, gradients must travel backward through many layers. Without shortcuts, they can vanish or explode. Residuals create direct paths for gradients, making learning stable.
3. Improving training speed: With residuals, deeper networks converge faster because the model can “skip” bad transformations while still using the identity mapping.

The Code in `train_gpt2.c`

Here's the implementation of residual addition:

```
void residual_forward(float* out, float* inp1, float* inp2, int N) {
    for (int i = 0; i < N; i++) {
        out[i] = inp1[i] + inp2[i];
    }
}
```

It's deceptively simple:

- `inp1` is the original input.
- `inp2` is the new transformation (from attention or MLP).
- `out` stores the sum.
- `N` is the total number of floats ($B * T * C$).

Even though it's just a single line inside a loop, this is what makes stacking 12+ transformer blocks possible.

Example Walkthrough

Suppose we're processing the token "cat."

- Input vector (simplified, size=3): [0.5, -0.3, 0.7]
- After attention block: [0.2, 0.1, -0.4]

Residual addition:

$$[0.5 + 0.2, -0.3 + 0.1, 0.7 - 0.4] = [0.7, -0.2, 0.3]$$

Now the representation of "cat" contains both the original signal and the contextual information from attention.

Later, after the MLP:

- Input again: [0.7, -0.2, 0.3]
- MLP output: [0.1, -0.5, 0.4]
- Residual: [0.8, -0.7, 0.7]

Step by step, the vector grows richer without losing its foundation.

Analogy

Residual connections are like building layers in Photoshop. Each new layer adds adjustments, but you always keep the original photo underneath. If a new adjustment is bad, you can still see the original. This makes the final composition stronger and safer to experiment with.

Residuals Across the Model

In GPT-2's forward pass, residuals appear in two main places inside each transformer block:

1. After attention:

```
x = x + Attention(x)
```

2. After MLP:

```
x = x + MLP(x)
```

Together with LayerNorm before each block, these form the backbone of the transformer architecture:

```
x → LayerNorm → Attention → Residual Add → LayerNorm → MLP → Residual Add
```

Why It Matters

Without residual connections, GPT-2 would struggle to train past a few layers. Deeper stacks would lose track of the original signal, gradients would vanish, and performance would stall. Residuals are the glue that holds the whole architecture together, enabling models with billions of parameters to train effectively.

Try It Yourself

1. Remove residuals: Temporarily comment out the `residual_forward` calls. Training will quickly fail-the loss won't decrease properly.
2. Print before/after: Inspect a token's vector before and after residual add. Notice how the numbers change smoothly rather than being overwritten.
3. Experiment with scaling: Try replacing `out[i] = inp1[i] + inp2[i];` with `out[i] = inp1[i] + 0.1 * inp2[i];`. This reduces the impact of the new transformation-sometimes used in advanced architectures.
4. Compare to skip-less RNNs: Research how older recurrent networks without residuals had trouble scaling deep. You'll see why residuals are a game changer.
5. Chain of signals: Track how a single token's vector evolves across all 12 layers. You'll notice it keeps its core identity while absorbing new context.

The Takeaway

Residual connections may look like a simple addition, but they're the key to deep learning's success in transformers. They preserve information, stabilize training, and allow GPT-2 to stack many layers without falling apart. In `train_gpt2.c`, this idea is laid bare: a few lines of C code implementing one of the most powerful tricks in modern neural networks.

37. Cross-Entropy Loss on CPU

After embeddings, attention, MLPs, LayerNorm, and residuals have done their job, the model produces logits-raw scores for every word in the vocabulary at every position in the sequence. But logits alone don't tell us if the model is "good" or "bad" at predicting the right word. To measure performance and guide learning, GPT-2 uses the cross-entropy loss function. In `train_gpt2.c`, this is implemented in the function `crossentropy_forward`.

What Are Logits?

At the final stage of the forward pass, each token position has a vector of length V (vocabulary size, $\sim 50k$). For example, the model might produce these logits for a tiny vocabulary of 3 words:

```
logits = [5.2, 1.1, -2.7]
```

Logits are just numbers-bigger means "more likely," smaller means "less likely"-but they aren't probabilities yet.

Step 1: Softmax – Turning Scores Into Probabilities

To compare predictions with the true target, we first convert logits into probabilities. The tool for this is the softmax function:

$$p[i] = \exp(\text{logits}[i]) / \sum \exp(\text{logits}[j])$$

Softmax has two important effects:

1. It makes all values positive.
2. It normalizes them so they sum to 1, forming a probability distribution.

Example:

- Logits: [5.2, 1.1, -2.7]
- Subtract max (5.2) for stability \rightarrow [0.0, -4.1, -7.9]
- Exponentiate \rightarrow [1.0, 0.017, 0.0004]
- Normalize \rightarrow [0.98, 0.017, 0.0004]

Now the model is saying:

- Word 0: 98% chance
- Word 1: 1.7% chance
- Word 2: 0.04% chance

Step 2: Cross-Entropy – Measuring Mistakes

Cross-entropy compares the predicted probability for the correct word against the ideal case (probability = 1).

Formula:

`loss = -log(probability_of_correct_word)`

- If the model assigns high probability to the correct word, loss is small.
- If the model assigns low probability, loss is large.

Example:

- Correct word = Word 0, probability = 0.98 \rightarrow loss = $-\log(0.98)$ 0.02 (excellent).
- Correct word = Word 1, probability = 0.017 \rightarrow loss = $-\log(0.017)$ 4.1 (bad).

Step 3: Averaging Over the Batch

In practice, we don't train on just one word, but a batch of sequences. The code loops over every token in every batch, collects their losses, and averages them.

From `train_gpt2.c`:

```
float loss = 0.0f;
for (int i = 0; i < B*T; i++) {
    int target = targets[i];
    float logit_max = -1e9;
    for (int j = 0; j < Vp; j++) {
        if (logits[i*Vp + j] > logit_max) logit_max = logits[i*Vp + j];
    }
    float sum = 0.0f;
```

```

    for (int j = 0; j < Vp; j++) {
        sum += expf(logits[i*Vp + j] - logit_max);
    }
    float log_sum = logf(sum);
    float correct_logit = logits[i*Vp + target];
    loss += (log_sum + logit_max - correct_logit);
}
loss /= (B*T);

```

What's happening here:

1. For each token, find the max logit (`logit_max`) to improve numerical stability.
2. Compute softmax denominator (`sum`).
3. Calculate log probability of the correct token.
4. Accumulate losses across all tokens.
5. Divide by total tokens (`B*T`) to get the average.

Numerical Stability Tricks

Without subtracting `logit_max`, `exp(logits)` can overflow. For example, `exp(1000)` is infinite. By subtracting the max, the largest logit becomes 0, so its exponential is 1, and all others are 1. This keeps numbers manageable while preserving the probability ratios.

Example With a Sentence

Sentence: *The cat sat on the mat.*

Suppose the model predicts probabilities for the last token:

- “mat”: 0.85
- “dog”: 0.10
- “car”: 0.05

Correct word = “mat.”

Loss = $-\log(0.85)$ = 0.16.

If instead the model guessed “dog” with 0.10, loss = $-\log(0.10)$ = 2.3. Higher penalty for being wrong.

Analogy

Cross-entropy is like grading multiple-choice exams. If the student picks the right answer confidently (high probability), they lose almost no points. If they're hesitant or wrong, they lose more points. Over many questions (tokens), you calculate their average score-the training loss.

Why It Matters

Cross-entropy loss is the guiding signal for the entire training process. It tells the optimizer:

- “Increase probability for the right words.”
- “Decrease probability for the wrong words.”

Without it, GPT-2 would have no way of knowing whether its predictions are improving.

Try It Yourself

1. Check baseline loss: Before training, print the loss. It should be close to `log(vocab_size)` (~10.8 for GPT-2 small), which corresponds to random guessing.
2. Inspect softmax sums: For one token, sum all probabilities. It should equal ~1.0.
3. Force the wrong answer: Temporarily change the target to an incorrect word. Watch how the loss shoots up.
4. Observe loss during training: Print loss every step. It should steadily decrease as the model learns.
5. Compare with accuracy: Track how often the model's top prediction matches the target. Loss and accuracy will move together, but loss is smoother and more informative.

The Takeaway

Cross-entropy loss turns raw model scores into a clear training signal. It penalizes wrong predictions, rewards confident correct ones, and ensures the optimizer knows exactly how to adjust weights. In `train_gpt2.c`, you see this implemented explicitly, without any library shortcuts-just loops, exponentials, and logs. Understanding this section is key to understanding how GPT-2 learns from its mistakes.

38. Putting It All Together: The `gpt2_forward` Function

Up to this point, we've explored the forward pass piece by piece - embeddings, attention, feed-forward layers, layer normalization, residual connections, and finally the loss. But a model doesn't live as disconnected pieces; they all come together in a single function that drives inference: `gpt2_forward`. This function is where the code actually executes the story we've been telling. Let's walk through it carefully so you can see how every building block plugs into the whole picture.

The role of `gpt2_forward`

Think of `gpt2_forward` as the director of the play. The actors (embeddings, attention, MLP, layernorm, etc.) already know their roles. The director calls them on stage in the right order and makes sure they hand the script off smoothly to the next actor. In our case:

1. Tokens come in as integers (word IDs).
2. They're turned into embeddings (token + position).
3. Each transformer block processes the sequence through attention, MLP, layernorm, and residuals.
4. The final hidden states are mapped back into vocabulary space.
5. If labels are provided, a loss is computed.

Code skeleton

Here's a simplified excerpt of the real function from `train_gpt2.c` (slightly shortened for readability):

```
void gpt2_forward(GPT2 *model, int *tokens, int *labels, int B, int T) {
    // Step 1: Embedding lookup
    embedding_forward(model->token_embedding, tokens, B, T);
    embedding_forward(model->position_embedding, positions, B, T);

    // Step 2: Transformer blocks
    for (int l = 0; l < model->config->n_layer; l++) {
        attention_forward(&model->blocks[l].attn, ...);
        mlp_forward(&model->blocks[l].mlp, ...);
        layernorm_forward(&model->blocks[l].ln, ...);
        residual_forward(...);
    }

    // Step 3: Final normalization + logits
}
```

```

    layernorm_forward(model->final_ln, ...);
    matmul_forward(model->lm_head, ...); // project to vocab

    // Step 4: Loss (optional)
    if (labels != NULL) {
        crossentropy_forward(...);
    }
}

```

Don't worry if this looks intimidating - we'll decode each part in plain language.

Step 1: Embedding lookup

Before the model can reason about words, it has to map token IDs into continuous vectors. That's where embedding tables come in:

- `token_embedding` converts each integer token ID into a dense vector of size `C` (the channel dimension).
- `position_embedding` does the same for positions (0, 1, 2, ..., T-1).
- These two are added together, giving each token both a meaning (word identity) and a place in the sentence.

Step 2: Transformer blocks

Each block is like a mini-pipeline that processes the sequence and passes it forward. Inside the loop:

1. Attention: compares tokens with each other, weighted by learned Q/K/V projections.
2. MLP: expands each token vector, applies a nonlinear GELU activation, then projects back down.
3. LayerNorm: normalizes values for stable training and inference.
4. Residual: adds the input of the block back to its output to keep information flowing.

This loop runs `n_layer` times - for GPT-2 124M, that's 12 blocks.

Step 3: Final normalization and logits

After the last block, the sequence of token representations goes through a final layer normalization. Then, a large matrix multiplication (`lm_head`) projects each token's hidden state into the size of the vocabulary (50,000 for GPT-2). The result is a tensor of shape (B, T, vocab_size) containing the raw prediction scores for each next token.

Step 4: Optional loss computation

If you pass `labels` (the correct next tokens) into `gpt2_forward`, the function calls `crossentropy_forward`. This compares the predicted scores with the true tokens and outputs a single number: the loss. The loss tells you “how wrong” the model was, which is critical during training. But if you’re only doing inference, you don’t need this step.

How the pieces connect

Here’s a table that maps our earlier sections to the parts of `gpt2_forward`:

Code Step	Concept	Section Covered Earlier
Embeddings	token + positional vectors	32
Attention	QKV projections, masking, softmax	33
MLP	feed-forward expansion and compression	34
LayerNorm	normalization for stability	35
Residual	skip connections for signal flow	36
CrossEntropy	comparing predictions with labels	37

So `gpt2_forward` is really just a neat orchestration of everything you’ve already learned.

Why it matters

Understanding `gpt2_forward` gives you the complete mental picture of inference. It shows how embeddings, attention, MLP, normalization, and residuals work together in code to turn a batch of tokens into predictions. Without this integration step, the model would just be a collection of disconnected parts.

Try it yourself

1. Print shapes: Add `printf` statements inside `gpt2_forward` to print tensor shapes after embeddings, after each block, and after logits. This helps you see the data flow.
2. Use a single block: Change the loop to run only 1 transformer block instead of all 12. Watch how the outputs degrade - the model loses depth of reasoning.
3. Disable position embeddings: Comment out the line that adds `position_embedding`. Try running inference. You’ll notice the model becomes worse at handling word order.
4. Loss vs no loss: Call `gpt2_forward` with and without labels. Compare the difference - with labels you get a scalar loss, without labels you just get logits.
5. Smaller vocab: Try using a toy tokenizer with a small vocabulary and rerun the projection step. You’ll see the logits shrink to (B, T, `tiny_vocab_size`).

The takeaway

`gpt2_forward` is where GPT-2 inference really happens. It ties together every concept - embeddings, attention, feed-forward layers, normalization, residuals, and the final projection into vocabulary space. Once you understand this function, you don't just know the pieces of GPT-2, you know how they actually work together to produce predictions. It's the "main stage" of inference, and mastering it means you can confidently say you understand how a transformer runs forward on CPU.

39. OpenMP Pragmas for Parallel Loops

CPU training in `train_gpt2.c` is intentionally "plain C," but it still squeezes out a lot of speed by adding a few OpenMP pragmas (`#pragma omp ...`) in the hottest loops. OpenMP lets the compiler split a loop's iterations across multiple CPU cores-no threads to create by hand, no locks to manage. If you compile without OpenMP support, these pragmas are simply ignored and the code still runs (just slower).

Below we'll (1) show exactly where OpenMP is used, (2) explain why those loops are good candidates, and (3) offer practical tips to get solid speedups on your machine.

OpenMP in this file: where it appears and why

Location			
/			
Function	Pragma used	What's parallelized	Why it's a great fit
<code>matmul_forward_naive</code>	<code>#pragma omp parallel for collapse(2)</code>	Outer loops over <code>b</code> (batch) and <code>t</code> (time)	Each <code>(b,t)</code> row computes an independent output vector; no write conflicts. Large, regular work = easy scaling.
<code>matmul_forward</code> (tiled)	<code>#pragma omp parallel for</code>	Collapsed <code>B*T</code> loop in tiles of <code>LOOP_UNROLL</code>	Heaviest compute in the model; tiling + per-thread tiles keep caches warm.
<code>matmul_backward</code> (part 1)	<code>#pragma omp parallel for collapse(2)</code>	Backprop into <code>inp</code> over <code>(b,t)</code>	Each <code>(b,t)</code> reads weights and <code>dout</code> , writes a private slice of <code>dinp</code> → no overlap.
<code>matmul_backward</code> (part 2)	<code>#pragma omp parallel for</code>	Backprop into <code>weight/bias</code> over <code>o</code> (output channel)	Each thread owns one output channel's gradient row, avoiding atomics.

Location			
/			
Function	Pragma used	What's parallelized	Why it's a great fit
<code>softmax_forward</code>	<code>#pragma omp parallel for collapse(2)</code>	Over (b, t) positions	Each softmax is independent; perfect “embarrassingly parallel” loop.
<code>attention_forward</code>	<code>#pragma omp parallel for collapse(3)</code>	Over (b, t, h) = batch, time, head	Per (b, t, h) head's work is independent; big 3-D grid parallelizes extremely well.

A few key patterns to notice:

- Collapse clauses (`collapse(2)` / `collapse(3)`) fuse nested loops into one big iteration space so the scheduler can distribute more, smaller chunks—great for load-balancing when B , T , or NH are modest.
- Parallelizing along independent dimensions avoids race conditions. For example, in `matmul_backward` the pass that writes `dinp[b,t,:]` is parallelized over (b, t) so no two threads update the same memory.
- Own-your-row strategy: when accumulating `dweight`, the loop goes over `o` (output channels) so each thread writes its own gradient row `dweight[o,:]`. No atomics needed.

Quick refresher: what OpenMP is doing

A typical pattern looks like:

```
#pragma omp parallel for collapse(2)
for (int b = 0; b < B; b++) {
    for (int t = 0; t < T; t++) {
        // compute outputs for (b, t) independently
    }
}
```

When compiled with OpenMP, the compiler creates a team of threads and divides the iteration space ($B \times T$ in this example) among them. Each thread executes its assigned iterations; when the loop finishes, threads sync at an implicit barrier.

Because each (b, t) (or (b, t, h)) writes to a disjoint slice of the output arrays, there's no need for locks or atomics. This is why these loops scale cleanly across cores.

Enabling OpenMP, safely

- The source guards the OpenMP header with:

```
#ifdef OMP
#include <omp.h>
#endif
```

so you can define `OMP` in your build and add your compiler switch. Example (GCC/Clang):

```
-D OMP -fopenmp
```

- If you forget `-fopenmp` (or your platform's equivalent), the pragmas are ignored and the program runs single-threaded.
- You can control threads at runtime:

```
export OMP_NUM_THREADS=8
```

A good rule of thumb is to start with the number of physical cores on your CPU.

Why these loops benefit the most

1. Matrix multiplies dominate runtime. `matmul_forward/matmul_backward` consume the bulk of CPU time. Parallelizing them yields the largest end-to-end speedups.
2. Softmax is independent per position. Each `(b,t)` softmax computes a max, then exponentials and a sum-no cross-talk between positions.
3. Attention splits across batch/time/head. The triple loop over `(b,t,h)` has lots of work per iteration ($Q \cdot K$, softmax, weighted sum), making thread overhead negligible compared to useful compute.
4. Minimal synchronization and no atomics. By choosing iteration spaces that own exclusive output slices, we avoid costly synchronization.

Practical tips for better scaling

- Set `OMP_NUM_THREADS` to your CPU. Too many threads can hurt (oversubscription). Start with physical cores, then experiment.
- Pin threads (optional, advanced). Some OpenMP runtimes support `OMP_PROC_BIND=close` to improve cache locality.
- Mind memory bandwidth. On wide CPUs, GEMMs may become bandwidth-bound. Bigger B/T improves arithmetic intensity; tiny batches underutilize cores.

- Warm caches with tiling. The “tiled” `matmul_forward` keeps small accumulators in registers and reuses loaded weights across `LOOP_UNROLL` inner iterations.
- Avoid hidden sharing. If you add new parallel loops, ensure each thread writes to unique memory regions. If you must accumulate to the same place, restructure (like “own-your-row”) or use per-thread scratch buffers then reduce.

Micro-walkthrough: why collapse helps

Consider `softmax_forward`:

```
#pragma omp parallel for collapse(2)
for (int b = 0; b < B; b++) {
    for (int t = 0; t < T; t++) {
        // 1) find max over V
        // 2) exp/log-sum
        // 3) normalize first V entries; zero out padded [V..Vp)
    }
}
```

If `B=4`, `T=64`, that’s 256 independent softmaxes. With `collapse(2)`, OpenMP sees a single loop of 256 iterations to distribute evenly; without `collapse`, it might chunk by `b` first (only 4 big chunks), which can load-imbalance.

Common pitfalls (and how this code avoids them)

- Race conditions: Two threads writing the same `out[i]`. *Avoided by design*: each parallel loop writes distinct slices (e.g., per `(b,t)` or per `o`).
- False sharing: Threads write adjacent memory locations on the same cache line. It’s minimized by the large, contiguous slices per thread (entire rows/tiles), but if you extend the code with fine-grained parallelism, keep this in mind.
- Tiny loops: Overhead can exceed work. The file parallelizes only large, hot loops (GEMMs, attention, softmax), not small scalar ops.

Try it yourself

1. Change thread count: Run with `OMP_NUM_THREADS=1,2,4,8,...` and log step time. Plot speedup vs. threads.
2. Toggle a pragma: Comment out `#pragma omp` in `matmul_forward` only. Measure the slowdown; you’ll see where most time goes.

3. Experiment with `collapse`: Remove `collapse(2)` in `softmax_forward`. On small B, you'll likely see worse scaling.
4. Per-layer profiling: Print elapsed time around `matmul_forward`, `attention_forward`, and `softmax_forward` to see which benefits most on your CPU.
5. Schedule policy (advanced): Try `#pragma omp parallel for schedule(static)` vs. `dynamic` on a heavy loop to see if it changes load balance (defaults are usually fine here).

The takeaway

A handful of well-placed OpenMP pragmas deliver big wins on CPU by parallelizing the most expensive loops (GEMMs, attention, softmax) across cores-without complicating the code. The design ensures each thread works on independent slices, so there's no locking, no atomics, and very little overhead. If you compile with OpenMP enabled, you get fast, multi-core training; if not, you still have a clean, readable reference implementation.

40. CPU Memory Footprint and Performance

When you train GPT-2 on your CPU using `train_gpt2.c`, two big questions usually pop up almost immediately: *how much memory is this going to take?* and *how fast will it run?* Let's walk through both of these in a beginner-friendly way, so you understand not just what happens in the code, but why it behaves the way it does.

Memory: where does it all go?

Imagine training GPT-2 is like cooking a big meal in a small kitchen. You need space for ingredients, bowls for mixing, and counter space for preparing. Memory on your CPU is that kitchen. GPT-2 needs several “bowls” to hold different parts of the computation:

1. Parameters (the weights of the model). These are the “fixed recipe” - the actual numbers the network learns. They come from the checkpoint file you load at the start. For GPT-2 124M, this is about 124 million floating-point numbers. Each one takes 4 bytes, so just the weights are around 500 MB.
2. Optimizer state (AdamW). Training doesn't just adjust weights blindly; it keeps track of two extra moving averages for each weight, called m and v . That means for every single parameter, you store three numbers: the weight, m , and v . So memory for optimizer state is often double the size of the weights themselves. For GPT-2 124M, that's about 1 GB more.
3. Gradients. Every time we run a backward pass, we store how much each weight should change. That's another buffer roughly the same size as the weights - another 500 MB.

4. Activations (intermediate results). This is the sneaky one. Every forward pass produces temporary tensors like embeddings, attention maps, and feed-forward outputs. Their size depends on batch size (B) and sequence length (T). If B=4 and T=64, activations are a few hundred MB. If B=32 and T=1024, they can balloon to many gigabytes.

Here's a rough mental budget for GPT-2 124M with a small setup (B=4, T=64):

- Parameters: ~500 MB
- Optimizer state: ~1 GB
- Gradients: ~500 MB
- Activations: ~200–300 MB Total: ~2–2.5 GB

Even for the “tiny” GPT-2, you already need a couple gigabytes of RAM to train. On a laptop, this can quickly push you to the limit.

Performance: where does time go?

Now let's talk speed. When you run `train_gpt2.c` on CPU, you'll see lines like:

```
step 1: train loss 5.191576 (took 1927.230000 ms)
```

That “took X ms” tells you how long one step took. Why is it slow? Three main reasons:

1. Matrix multiplications (matmuls). These are the heart of neural networks. Every attention head and every MLP layer does them. On CPU, most of your step time is spent here. That's why the code uses OpenMP pragmas (`#pragma omp`) to parallelize loops across cores.
2. Attention softmax. Attention compares every token in a sequence with every other token. If your sequence length is 1024, that's over a million comparisons per head per layer. On CPU, this quadratic growth is painful.
3. Memory bandwidth. CPUs can only move numbers from RAM to cores so fast. Even if you had infinite FLOPs, you'd still be slowed down by how quickly you can fetch and store these huge tensors.

A simple experiment

You can see these effects yourself:

- Change batch size (B). Run with B=1, then with B=8. Notice how memory usage and step time scale up.

- Change sequence length (T). Try T=16, then T=256. You'll see attention costs grow dramatically.
- Change threads. Set `OMP_NUM_THREADS=1` versus `OMP_NUM_THREADS=8`. With more threads, you'll often see speedups, but only up to the number of physical cores your CPU has.

Why this matters

For beginners, CPU runs are perfect for learning:

- You can debug with small batches and short sequences.
- You can step into functions with a debugger and watch tensors being created.
- You don't need a GPU just to understand how training works.

But when it comes to *serious* training - larger GPT-2 models or even long sequences - CPU is simply too slow. What takes seconds on GPU may take minutes on CPU. That's why in practice, people use CPUs for learning and testing, and GPUs for large-scale training.

The takeaway

Training GPT-2 on CPU is like practicing piano on a small keyboard. It's slower, limited, and you can't play the biggest pieces, but it's great for learning the fundamentals. Memory usage comes from weights, optimizer state, gradients, and activations, and performance is dominated by matmuls and attention. Once you understand where the resources go, you can adjust batch size, sequence length, and threads to find the sweet spot for your machine.

Chapter 5. Training Loop (CPU Path)

41. Backward Pass Walkthrough

Up until now, we've spent all our time looking at the forward pass. That's the part of the model that takes tokens, pushes them through embeddings, attention, feed-forward layers, and finally produces logits or a loss. For inference, forward pass alone is enough. But if you want to train a model, forward is only half the story.

Training means adjusting the weights of the model so that its predictions become better over time. To do this, we need a way to figure out how wrong each weight was and in what direction it should move to reduce the loss. That's the job of the backward pass.

The backward pass is also called backpropagation. It's the algorithm that moves information in reverse through the network: from the loss, back through the final logits, through every

transformer block, down to the embeddings. Along the way, it calculates gradients - small numbers that tell us how much each weight contributed to the error.

The big idea: chain rule in action

At the heart of backpropagation is something very familiar from calculus: the chain rule. If the output of the network depends on many functions stacked together (embedding \rightarrow attention \rightarrow MLP \rightarrow ... \rightarrow loss), then the derivative of the loss with respect to an early parameter is a product of partial derivatives through the entire chain.

Instead of writing long formulas, the code in `train_gpt2.c` simply calls each layer's backward function in reverse order. The gradient flows backward, step by step, and each layer computes its own contribution using local rules.

Think of it like a relay race, but run backwards: the loss hands a “blame baton” to the output head, which hands it back to the last transformer block, and so on, until it reaches the very first embedding table.

Walking through `gpt2_backward`

Here's a simplified sketch of how the backward function looks in the code (names shortened for readability):

```
void gpt2_backward(GPT2 *model, int *tokens, int *labels, int B, int T) {
    // Step 1: loss gradient
    crossentropy_backward(...);

    // Step 2: final projection (lm_head)
    matmul_backward(model->lm_head, ...);

    // Step 3: transformer blocks in reverse
    for (int l = model->config->n_layer - 1; l >= 0; l--) {
        residual_backward(...);
        layernorm_backward(&model->blocks[l].ln, ...);
        mlp_backward(&model->blocks[l].mlp, ...);
        attention_backward(&model->blocks[l].attn, ...);
    }

    // Step 4: embeddings
    embedding_backward(model->token_embedding, tokens, ...);
    embedding_backward(model->position_embedding, positions, ...);
}
```

Let's unpack this line by line.

Step 1: Starting from the loss

The journey begins with the loss function. In training, the most common loss is cross-entropy. Its backward function compares the predicted probabilities with the true labels and produces a gradient for the logits.

- If the model predicted “cat” with high confidence and the true label was “dog,” the gradient will push the logits away from “cat” and toward “dog.”
- This gradient is the starting signal that propagates backward through the entire network.

Step 2: Back through the output head

After the loss, the next stop is the final linear projection (`lm_head`). This is just a big matrix multiply that turns hidden states into vocabulary logits. Its backward function computes two things:

1. The gradient with respect to the weights of `lm_head`.
2. The gradient with respect to the hidden states that fed into it.

This hidden-state gradient is then passed back to the last transformer block.

Step 3: Transformer blocks in reverse

Here comes the heavy lifting. Each block has multiple components, and their backward functions are called in the exact opposite order of the forward pass.

1. Residual backward: the skip connection splits the gradient into two paths - one flowing back into the transformed output, one flowing back into the original input.
2. LayerNorm backward: computes gradients with respect to its scale (`gamma`) and shift (`beta`), and also passes gradients back to the normalized input.
3. MLP backward: applies the chain rule to the two linear layers and the GELU activation. The code reuses temporary values from the forward pass (like activations) to make this efficient.
4. Attention backward: this is the trickiest. It computes gradients for Q, K, and V projections, as well as for the softmaxed attention weights. It has to apply the causal mask again to ensure no illegal gradient flows.

This loop continues until all transformer blocks have been processed.

Step 4: Back to embeddings

Finally, the gradient reaches the embedding tables. This is where the model first looked up vectors for tokens and positions. Now it calculates how much each embedding contributed to the error. These gradients are added into the embedding matrices, telling the optimizer how to update them.

Why this matters

The backward pass is what makes learning possible. Without it, the model would forever output the same predictions, never improving. By flowing “blame” backwards, each parameter learns how to nudge itself so that the next forward pass is a little bit better.

Even though the code looks like a lot of function calls, the principle is simple: start from the loss, step backward through each layer, apply the chain rule locally, and collect gradients.

Try it yourself

1. Print gradient norms: Add a `printf` to see the average gradient magnitude at each layer. Notice how they change - sometimes exploding, sometimes vanishing.
2. Freeze a layer: Comment out `mlp_backward` for one block and see how the model fails to update properly.
3. Inspect embeddings: After training a few steps, dump a few rows of the token embedding matrix. You’ll see the numbers changing because of gradient updates.
4. Tiny dataset experiment: Train on a very small dataset (like a 10-word corpus) and watch how the backward pass quickly pushes embeddings to memorize it.
5. Check symmetry: Compare the order of calls in `gpt2_forward` with `gpt2_backward`. They’re exact opposites - forward builds, backward unbuilds.

The takeaway

Backpropagation is the learning engine of neural networks. In `llm.c`, the backward pass is written out explicitly, showing how gradients flow from the loss, through the output head, back through every transformer block, and finally into embeddings. Once you understand this flow, you can see how training stitches forward and backward together to slowly shape a random model into a working language model.

42. Skeleton of Training Loop

The backward pass gave us gradients, but gradients by themselves don't train a model. Training requires a loop: a cycle that repeatedly runs forward, backward, and update steps over and over until the model improves. This cycle is called the training loop, and it is the heartbeat of every deep learning program. In `train_gpt2.c`, the loop is written explicitly in C, which means you can see every piece instead of it being hidden away in a framework.

The basic rhythm

Every training step follows the same rhythm:

1. Get a batch of data (input tokens and their labels).
2. Run the forward pass to compute predictions and loss.
3. Run the backward pass to compute gradients.
4. Update weights using an optimizer like AdamW.
5. Log progress and, occasionally, validate.

This rhythm repeats thousands or millions of times. With each repetition, the weights shift slightly, nudging the model toward lower loss and better predictions.

How the loop looks in code

Here's a simplified sketch from `train_gpt2.c` (with some details omitted for clarity):

```
for (int step = 0; step < max_steps; step++) {
    // 1. Load batch of tokens and labels
    dataloader_next_batch(&train_loader, tokens, labels, B, T);

    // 2. Forward pass
    gpt2_forward(&model, tokens, labels, B, T);

    // 3. Zero gradients
    gpt2_zero_grad(&model);

    // 4. Backward pass
    gpt2_backward(&model, tokens, labels, B, T);

    // 5. Optimizer step (AdamW)
    adamw_update(&opt, &model, learning_rate);

    // 6. Logging and validation
```

```

    if (step % log_interval == 0) { print_loss(step, model.loss); }
    if (step % val_interval == 0) { run_validation(...); }
}

```

This loop captures the full training lifecycle: data, forward, backward, update, and monitoring.

Step 1: Batching data

The dataloader feeds the loop with small chunks of tokens. Instead of sending the whole dataset at once, it breaks it down into batches of size B (number of sequences per batch) and length T (number of tokens per sequence).

- Example: if $B=4$ and $T=128$, each batch is 512 tokens long.
- Each sequence has a matching set of labels, which are simply the same tokens shifted one position ahead (so the model always predicts the *next* word).

This batching keeps memory use manageable and helps the model see many small samples instead of a few giant ones.

Step 2: Forward pass

The forward pass computes predictions for all tokens in the batch and calculates the loss. This is the “evaluation” step - how well did the model do on this batch? The result is stored in `model.loss`.

Step 3: Zeroing gradients

Before calculating new gradients, the old ones must be cleared out. If you skip this step, gradients from previous batches would accumulate and corrupt the update. In frameworks like PyTorch you’d call `optimizer.zero_grad()`. Here it’s a plain C function:

```
gpt2_zero_grad(&model);
```

It walks through all parameters and resets their gradient buffers to zero.

Step 4: Backward pass

Now the backward function is called. It pushes gradients back through the network, computing how each weight influenced the error. At this point, every parameter has an associated gradient stored in memory.

Step 5: Optimizer update

With gradients ready, the optimizer (AdamW in this code) updates each parameter:

```
new_weight = old_weight - learning_rate * gradient (with AdamW tweaks)
```

This step is what actually changes the model. Without it, the model would never learn - the forward and backward passes would just repeat the same results forever.

Step 6: Logging and validation

Every few steps, the loop prints out useful numbers: current step, loss, time taken, and sometimes throughput (tokens per second). This feedback is important to check whether training is actually working.

Every few hundred or thousand steps, the loop also runs a validation pass on held-out data. This tells you whether the model is just memorizing training data or genuinely learning patterns that generalize.

Why the training loop matters

The training loop is deceptively simple, but it is the engine room of machine learning. Every improvement in model performance happens because this loop runs many times. By writing it explicitly in C, `llm.c` exposes details that high-level frameworks usually hide: zeroing gradients, passing pointers to arrays, calling backward and optimizer functions directly.

This makes it a perfect learning tool. You can see clearly:

- Where the data comes in,
- Where predictions are made,
- Where gradients are calculated,
- And where learning actually happens.

Try it yourself

1. Print the loss curve: Add a `printf` inside the loop and write the loss to a file. Plot it - you should see it decrease over time.
2. Change batch size: Set `B=1` vs. `B=8`. Notice how the loop becomes noisier with smaller batches but smoother with larger ones.
3. Skip backward: Comment out `gpt2_backward` and optimizer update. Run the loop. You'll see the loss never decreases - a clear demonstration that forward alone doesn't train.
4. Experiment with steps: Try `max_steps=10` vs. `max_steps=1000`. Short runs show no improvement; longer runs start to reduce the loss.
5. Slow it down: Insert a `sleep(1);` inside the loop. This makes the rhythm visible step by step, so you can literally watch the model "breathe" as it trains.

The takeaway

The skeleton of the training loop is the core cycle of learning. It feeds data into the model, computes predictions, finds errors, sends them backward, updates weights, and logs progress. Everything else - optimizers, schedulers, distributed training, mixed precision - is just an enhancement of this basic loop. If you understand how this loop works in `llm.c`, you understand the beating heart of deep learning training.

43. AdamW Implementation in C

Training a neural network is about adjusting millions of parameters so that the model gradually becomes better at predicting text. The function `gpt2_update` in `train_gpt2.c` is responsible for this adjustment. It implements the AdamW optimizer, one of the most widely used algorithms in deep learning. Let's walk through both the theory and the actual implementation.

From Gradient Descent to AdamW

The most basic optimizer is gradient descent:

```
new_param = old_param - learning_rate * gradient
```

This approach works, but it has weaknesses. The step size (learning rate) must be tuned carefully: too small and training is slow, too large and training diverges. Moreover, all parameters use the same step size, even though some may need gentler updates.

AdamW improves this by keeping track of moving averages of gradients and scaling updates adaptively. It also introduces weight decay, which prevents parameters from growing too large and helps regularize the model.

How AdamW Works

AdamW combines several techniques into a single update rule. First, it uses momentum: instead of relying only on the current gradient, it averages recent gradients. This smooths noisy updates. Second, it maintains a running estimate of the squared gradient, which scales down steps in directions where gradients are consistently large. These are sometimes called the first and second moments.

Since both running averages start at zero, the algorithm applies bias correction during the first few steps. Without this, the early updates would be too small. Finally, AdamW applies weight decay directly in the update, shrinking parameter values slightly each step.

Putting it together, each parameter update looks like this:

```
m_t = 1 * m_(t-1) + (1 - 1) * g_t
v_t = 2 * v_(t-1) + (1 - 2) * g_t^2
m_hat_t = m_t / (1 - 1^t)
v_hat_t = v_t / (1 - 2^t)

new_param = old_param - lr * ( m_hat_t / (sqrt(v_hat_t) + ) + * old_param )
```

Here m is momentum, v is variance, lr is learning rate, ϵ is a small constant for stability, and λ is the weight decay factor.

The Implementation in train_gpt2.c

```
void gpt2_update(GPT2 *model, float learning_rate, float beta1, float beta2,
                 float eps, float weight_decay, int t) {
    if (model->m_memory == NULL) {
        model->m_memory = (float*)calloc(model->num_parameters, sizeof(float));
        model->v_memory = (float*)calloc(model->num_parameters, sizeof(float));
    }

    for (size_t i = 0; i < model->num_parameters; i++) {
        float param = model->params_memory[i];
        float grad = model->grads_memory[i];
```

```

float m = beta1 * model->m_memory[i] + (1.0f - beta1) * grad;
float v = beta2 * model->v_memory[i] + (1.0f - beta2) * grad * grad;

float m_hat = m / (1.0f - powf(beta1, t));
float v_hat = v / (1.0f - powf(beta2, t));

model->m_memory[i] = m;
model->v_memory[i] = v;
model->params_memory[i] -= learning_rate *
    (m_hat / (sqrtf(v_hat) + eps) + weight_decay * param);
}
}

```

The first `if` block allocates memory for the moving averages `m` and `v` the first time the optimizer runs. Then, for each parameter, the code computes the new averages, applies bias correction, and finally updates the parameter with the AdamW formula.

Example Walkthrough

Suppose we have a parameter $w = 0.5$ with gradient $g = 0.2$ on the first training step. Using $\beta_1 = 0.9$ and $\beta_2 = 0.999$:

- Momentum:

$$m = 0.9 * 0 + 0.1 * 0.2 = 0.02$$

- Variance:

$$v = 0.999 * 0 + 0.001 * 0.04 = 0.00004$$

- Bias correction:

$$\hat{m} = 0.02 / (1 - 0.9) = 0.2$$

$$\hat{v} = 0.00004 / (1 - 0.999) = 0.04$$

- Final update ($lr = 0.001$, $weight_decay = 0.01$):

$$\begin{aligned}
 \text{update} &= 0.001 * (0.2 / \sqrt{0.04}) + 0.01 * 0.5 \\
 &= 0.001 * (1.0 + 0.005) \\
 &= 0.001005
 \end{aligned}$$

So the parameter becomes $w = 0.498995$.

Intuition

Think of a ball rolling down a slope. The gradient is the slope itself. Momentum makes the ball keep rolling even if the slope flattens briefly. The variance term makes the ball slow down on rocky ground where the slope changes rapidly. Bias correction ensures the ball doesn't move too timidly at the start. Weight decay adds friction so the ball doesn't roll out of control.

Why It Matters

Optimizers are the difference between a model that trains smoothly and one that diverges or gets stuck. AdamW became popular because it combines stability with efficiency. It automatically adapts to each parameter's scale, reduces the need for manual learning rate tuning, and includes weight decay in a principled way. For GPT-style models with hundreds of millions of parameters, these qualities make training feasible.

Try It Yourself

1. Change the learning rate from 0.001 to 0.01 in the code and see how quickly the model diverges.
2. Set `weight_decay = 0` and compare validation loss after a few epochs. The model might overfit more quickly.
3. Print out the first 10 values of `m_memory` and `v_memory` during training to watch how they evolve over steps.
4. Replace AdamW with plain SGD (just `param -= lr * grad`) and compare training speed and stability.
5. Experiment with `1 = 0` (no momentum) or `2 = 0` (no variance smoothing) and see how noisy updates become.

The Takeaway

AdamW provides a balance of speed, stability, and generalization. In practice, it allows models like GPT-2 to train much more reliably than with vanilla gradient descent. The C implementation in `llm.c` demonstrates that beneath the math, it's just a simple loop applying a few arithmetic operations for each parameter.

44. Gradient Accumulation and Micro-Batching

Modern language models are enormous, and so are the batches of text we would like to feed them during training. But real hardware has limits: a single GPU or CPU may not have enough memory to process a large batch in one go. To solve this, training code often uses

gradient accumulation and micro-batching. Both ideas allow us to simulate training with larger batches without requiring more memory than our hardware can provide.

What Problem Are We Solving?

When you process a batch of data, you run forward and backward passes to calculate gradients. If your batch size is very large, you get smoother gradients (less noisy), which often helps the model converge better. But large batches may not fit in memory.

Imagine trying to train with a batch of 1024 sequences on a GPU that can only handle 128 sequences at once. Without tricks, you would be forced to use the smaller batch size and give up the benefits of larger batches. Gradient accumulation fixes this problem by letting you split the big batch into smaller micro-batches, process them one at a time, and accumulate the results as if you had processed the big batch all at once.

How It Works in Practice

Let's say we want an effective batch size of 1024, but our hardware only supports 128. We split the big batch into 8 micro-batches of 128 each:

1. Run forward + backward on micro-batch 1, store the gradients.
2. Run forward + backward on micro-batch 2, add its gradients to the stored ones.
3. Repeat until all 8 micro-batches are processed.
4. Once gradients for all 8 are accumulated, perform the optimizer update.

The important part is step 4: we only update the parameters once per effective batch, not after each micro-batch. This preserves the effect of training with a large batch.

Pseudocode Example

Here's how this might look in simplified pseudocode:

```
int accumulation_steps = 8;
for (int step = 0; step < total_steps; step++) {
    zero_grad(&model);
    for (int i = 0; i < accumulation_steps; i++) {
        dataloader_next_batch(&train_loader, tokens, labels, B, T);
        forward(&model, tokens, labels, B, T);
        backward(&model, tokens, labels, B, T);
        // do NOT call optimizer update yet
    }
}
```

```
adamw_update(&model, lr, beta1, beta2, eps, weight_decay, step+1);  
}
```

Notice how the optimizer only runs once per outer loop iteration, even though gradients were accumulated across multiple micro-batches.

Why Gradient Accumulation Helps

- Memory efficiency: You can train with larger effective batch sizes without needing more hardware.
- Training stability: Larger batches reduce the variance of gradients, making training less noisy.
- Flexibility: You can scale effective batch size up or down depending on your needs without changing hardware.

Micro-Batching vs. Accumulation

Micro-batching refers to the act of splitting a batch into smaller parts. Gradient accumulation is what you do after micro-batching: sum up the gradients across those parts. Together, they allow you to simulate training with any batch size you want, within memory constraints.

Why It Matters

The quality of training often depends on batch size. If you can't fit a large batch directly, gradient accumulation ensures you still reap the benefits. It's one of those "engineering hacks" that makes training state-of-the-art models possible on limited resources.

Try It Yourself

1. Run training with batch size = 16 and no accumulation. Watch how noisy the loss curve looks.
2. Now set micro-batch size = 4 and accumulation_steps = 4. This simulates batch size = 16, but in smaller chunks. Compare the loss curve.
3. Increase accumulation_steps to simulate batch size = 32. Observe if training becomes smoother.
4. Experiment with turning accumulation off and on while keeping the same effective batch size. Notice how optimizer updates per epoch differ.
5. Print out how many times the optimizer is called. With accumulation, it should be fewer than the number of micro-batches.

The Takeaway

Gradient accumulation and micro-batching are techniques that let you train with large effective batch sizes while staying within the limits of your hardware. They preserve the benefits of large batches—stability and smoother gradients—without demanding extra memory. In `llm.c`, the simplicity of the training loop means you can clearly see where accumulation fits: gradients are summed across micro-batches, and only then does the optimizer step in. This is a small adjustment in code but a huge enabler in practice.

45. Logging and Progress Reporting

Every training loop needs a way to show what’s happening under the hood. Without logs, you wouldn’t know if the model is improving, if the code is running efficiently, or if something has silently gone wrong. In `train_gpt2.c`, logging is intentionally minimal but highly informative: each training step prints the step number, the current training loss, and how long that step took to run.

The Real Code for Logging

Here’s the relevant snippet from `train_gpt2.c`:

```
// do a training step
clock_gettime(CLOCK_MONOTONIC, &start);
dataloader_next_batch(&train_loader);
gpt2_forward(&model, train_loader.inputs, train_loader.targets, B, T);
gpt2_zero_grad(&model);
gpt2_backward(&model);
gpt2_update(&model, 1e-4f, 0.9f, 0.999f, 1e-8f, 0.0f, step+1);
clock_gettime(CLOCK_MONOTONIC, &end);

double time_elapsed_s = (end.tv_sec - start.tv_sec) +
                        (end.tv_nsec - start.tv_nsec) / 1e9;
printf("step %d: train loss %f (took %f ms)\n",
       step, model.mean_loss, time_elapsed_s * 1000);
```

This small block accomplishes two things:

1. It measures how long the training step took using `clock_gettime`.
2. It reports the step number, the loss, and the elapsed time in milliseconds.

The output looks like this when training:

```
step 0: train loss 4.677779 (took 1987.546 ms)
step 1: train loss 5.191576 (took 1927.230 ms)
step 2: train loss 4.438685 (took 1902.987 ms)
```

Understanding What's Reported

- Step number (`step`) Tells you where you are in training. Since deep learning often runs for thousands of steps, this acts like a progress bar.
- Training loss (`model.mean_loss`) Shows how well the model is fitting the training batch. A lower value generally means better predictions. Watching this number decrease over time is the main signal that learning is happening.
- Step duration (`time_elapsed_s * 1000`) Measures performance. If one step takes 2000 ms, then 5000 steps would take about 3 hours. Monitoring this helps you estimate total training time and spot performance regressions (e.g., if a new change suddenly doubles the step time).

Why It Matters

Logs are your window into the training process. If the loss goes down smoothly, training is healthy. If it suddenly spikes or stays flat, something is wrong-maybe the learning rate is too high, or the model has run out of capacity. Timing information also matters: you need to know whether the code is running efficiently or wasting cycles.

Try It Yourself

1. Change the learning rate from `1e-4` to `1e-2` and watch how the loss behaves. If it jumps or becomes unstable, you'll see it directly in the logs.
2. Add validation logging by running the model on a held-out dataset every 100 steps and printing `val_loss`. Compare it to `train_loss`.
3. Record the log output to a file with:

```
./train_gpt2 > log.txt
```

Then plot `train_loss` over steps in Python or Excel to visualize the curve.

4. Add throughput reporting: divide the batch size times sequence length (`B*T`) by the step time to print tokens per second. This gives a clearer sense of efficiency.
5. Try disabling `clock_gettime` and only print loss. Notice how much harder it becomes to judge performance without timing information.

The Takeaway

Even the simplest logs can tell you a lot. With just a single line-step, loss, and duration-you know how fast training is, whether it's converging, and how long it will take. In larger frameworks, this kind of information is often hidden behind dashboards and monitoring tools, but the core idea is the same: training is only useful if you can see and interpret its progress.

46. Validation Runs in the Training Loop

When you train a model, it is not enough to look only at how well it does on the training data. The real test is whether the model has learned patterns that apply to new, unseen data. This is where validation comes in. Validation is like a quiz the model takes from time to time during training. It does not count toward learning-it is just a check to see how much the model has really understood.

In `train_gpt2.c`, validation is built right into the training loop. Every so often, instead of updating weights, the program pauses and runs the model on a set of tokens it has never trained on. It then prints out the average validation loss. This number tells you if the model is actually generalizing, not just memorizing.

How the validation code looks

Here is the actual block of code that handles validation:

```
if (step % 10 == 0) {
    float val_loss = 0.0f;
    dataloader_reset(&val_loader);
    for (int i = 0; i < val_num_batches; i++) {
        dataloader_next_batch(&val_loader);
        gpt2_forward(&model, val_loader.inputs, val_loader.targets, B, T);
        val_loss += model.mean_loss;
    }
    val_loss /= val_num_batches;
    printf("val loss %f\n", val_loss);
}
```

At first glance, this might look like just a few lines of C code. But behind it are several important ideas about how machine learning models are tested while they learn. Let's go through this step by step.

Step-by-step explanation

The first line checks whether it is time to run validation:

```
if (step % 10 == 0) {
```

This means that validation happens every 10 steps. The % operator is “modulo,” which returns the remainder of a division. If the step number is divisible by 10 (like 0, 10, 20, 30), then the block runs. By spacing it out this way, validation does not slow training too much but still gives you regular updates.

Next, the code sets up a place to store the running total of the validation loss:

```
float val_loss = 0.0f;
```

Then it resets the validation dataloader:

```
dataloader_reset(&val_loader);
```

This makes sure the validation dataset starts from the beginning each time. That way, the results are consistent—you’re always checking the model on the same set of text, rather than starting from a random place.

Now comes the loop over validation batches:

```
for (int i = 0; i < val_num_batches; i++) {  
    dataloader_next_batch(&val_loader);  
    gpt2_forward(&model, val_loader.inputs, val_loader.targets, B, T);  
    val_loss += model.mean_loss;  
}
```

Here’s what’s happening inside:

- `dataloader_next_batch` fetches the next chunk of tokens and labels from the validation set.
- `gpt2_forward` runs the model forward on those tokens, predicting the next word for each one, and computes the loss against the true labels.
- The loss from that batch is added to `val_loss`.

Notice that there is no call to `gpt2_zero_grad`, no `gpt2_backward`, and no `gpt2_update`. That is because validation does not train the model. It only measures performance.

Finally, the program averages the loss across the number of batches:

```
val_loss /= val_num_batches;
```

And prints the result:

```
printf("val loss %f\n", val_loss);
```

This gives you a single number that summarizes how well the model is performing on unseen data at this point in training.

How to read validation loss

Imagine you are training and see logs like this:

```
step 0: train loss 4.677779 (took 1987.546 ms)
val loss 4.901234
step 1: train loss 5.191576 (took 1927.230 ms)
step 2: train loss 4.438685 (took 1902.987 ms)
...
step 10: train loss 3.912342 (took 1890.321 ms)
val loss 4.100321
```

The training loss is printed every step, while the validation loss appears every 10 steps. If both numbers are going down, that is a sign the model is genuinely learning. If training loss drops but validation loss stays the same or starts going up, the model is probably memorizing the training set-this is called overfitting.

Why validation is important

Without validation, you could be tricked into thinking the model is improving just because the training loss is going down. But that might only mean it has memorized the training data. Validation checks prevent this by showing you whether the model can handle data it has not seen before. It is like a student practicing with old exam papers (training) versus being tested with new problems (validation).

Small details that matter

The code averages validation loss over `val_num_batches`, which is set earlier to 5. That means it only checks 5 batches, not the entire validation dataset. This is a shortcut-it makes validation much faster, at the cost of some accuracy in the measurement. But for training feedback, this is usually enough.

The batch size `B` and sequence length `T` for validation are the same as training. This keeps the loss comparable between training and validation.

Try it yourself

You can experiment with the validation process to understand it better. Here are some ideas:

1. Change the frequency from every 10 steps to every 5 or even every step. You'll see more validation updates, but training will slow down.
2. Increase `val_num_batches` to 20. The validation loss will become less noisy, but each check will take longer.
3. Comment out the validation block and train again. Notice how you lose a sense of whether the model is really generalizing.
4. Save validation loss values to a file and plot them. Compare the curve against the training loss curve. You'll see how they move together or diverge.
5. Try using a very small validation dataset. Watch how the loss jumps around more compared to a larger, more stable dataset.

The takeaway

Validation runs are short forward-only tests that give you confidence the model is learning patterns that apply to new text. They are easy to implement-a few lines of code in `train_gpt2.c`-but they are one of the most important tools for monitoring training. By checking validation loss regularly, you make sure your model is not just memorizing but actually becoming better at language modeling.

47. Checkpointing Parameters and Optimizer State

Training a model can take hours, days, or even weeks. If you stop the program halfway-whether by accident (a crash, a power cut) or on purpose (pausing to save compute)-you don't want to start over from scratch. Checkpointing solves this problem by saving the model's parameters and optimizer state to disk so you can resume training later.

What a checkpoint contains

A checkpoint is like a “save game” for machine learning. At a minimum, it needs:

1. Model parameters – the actual weights of the neural network, stored as floating-point numbers in memory. These define what the model has learned so far.
2. Optimizer state – for AdamW, this includes the running averages of gradients (`m_memory`) and squared gradients (`v_memory`). Without these, the optimizer would lose its “memory” of past updates, which could destabilize resumed training.
3. Step counter – the number of steps completed so far. This matters for bias correction in AdamW and for scheduling the learning rate.

Together, these three things capture the full training state.

Saving a checkpoint

Although `train_gpt2.c` is kept minimal and does not include full checkpointing code, the idea is straightforward. You allocate a file, write all parameters, optimizer buffers, and metadata, then close the file. In pseudocode, it looks like this:

```
FILE *f = fopen("checkpoint.bin", "wb");
fwrite(model.params_memory, sizeof(float), model.num_parameters, f);
fwrite(model.m_memory, sizeof(float), model.num_parameters, f);
fwrite(model.v_memory, sizeof(float), model.num_parameters, f);
fwrite(&step, sizeof(int), 1, f);
fclose(f);
```

This is a binary dump of the model and optimizer. Later, you can load the file back with `fread` calls into the same memory locations.

Loading a checkpoint

Loading is the reverse:

```
FILE *f = fopen("checkpoint.bin", "rb");
fread(model.params_memory, sizeof(float), model.num_parameters, f);
fread(model.m_memory, sizeof(float), model.num_parameters, f);
fread(model.v_memory, sizeof(float), model.num_parameters, f);
fread(&step, sizeof(int), 1, f);
fclose(f);
```

Once loaded, training can continue exactly where it left off.

Why optimizer state matters

It might seem enough to save only the model's parameters. But AdamW depends on moving averages of past gradients. If you throw those away and restart with only the parameters, the optimizer will behave differently. Learning may suddenly become unstable, or the effective learning rate may feel wrong. That's why saving both the parameters and optimizer state gives the most faithful restart.

Why checkpointing is essential

Training is rarely smooth. Servers reboot, experiments are interrupted, bugs are found. Without checkpoints, any interruption means wasted compute and lost progress. With checkpoints, you can pause and resume at will. They also let you archive important moments in training—for example, saving the model when validation loss is lowest, not just at the end.

Try it yourself

1. Write a small function that saves the model's parameters after every 100 steps. Then kill the program midway and reload from the saved file. Confirm that resumed training picks up where it left off.
2. Try saving only parameters but not optimizer state. Resume training and compare loss curves. You'll see that the run diverges from the original.
3. Save checkpoints at multiple steps and later reload them to compare model generations (does the model produce more fluent text after 10 steps, 100 steps, 1000 steps?).
4. Intentionally corrupt part of a checkpoint file (flip a few bytes) and try reloading. This helps you understand why consistency checks or checksums are often added in real systems.
5. Store checkpoints in a versioned way (e.g., `checkpoint_step100.bin`, `checkpoint_step200.bin`) so you can roll back if a later training phase degrades performance.

The takeaway

Checkpointing is what makes long-running training practical. By saving parameters, optimizer state, and the step counter, you preserve not just what the model knows but how it is learning. In real projects, checkpoints are the bridge between experiments and production: they let you stop, resume, compare, and deploy models without ever starting from scratch. Even though `llm.c` does not fully implement it, the concept is simple and invaluable.

48. Reproducibility and Small Divergences

When training deep learning models, two runs that look identical on the surface can still behave differently. One run might converge quickly, another might take longer, and sometimes losses diverge even though you used the same dataset and code. This happens because of the way randomness and numerical precision interact during training. Reproducibility is about controlling these factors so that results are consistent and meaningful.

Sources of randomness

There are several places where randomness sneaks into training:

- Data order: if batches are shuffled differently, the model sees tokens in a new sequence. Early steps can influence the trajectory of training.
- Weight initialization: initial parameters are usually set randomly. Different seeds lead to slightly different starting points.
- Dropout and sampling: while `train_gpt2.c` is minimal and doesn't include dropout layers, many neural networks do. Dropout randomly disables activations during training.
- Floating-point arithmetic: on CPUs and GPUs, the order of summations or parallel reductions can cause tiny rounding differences. Over many steps, these small changes accumulate.

How llm.c handles reproducibility

The repository includes functions like `manual_seed` and `random_f32` in `llmc/rand.h`. These are simple random number generators that can be seeded with a fixed value. For example:

```
manual_seed(1337);
```

If you call this before training, the random number generator starts from the same state every run. That means weight initialization and sampling will be reproducible.

The dataloaders also have reproducibility options. When you initialize a `DataLoader`, you can decide whether it shuffles batches or not. Keeping this consistent ensures the model sees the same data order each run.

Why small divergences happen anyway

Even with fixed seeds, you might notice that two runs are not perfectly identical. On CPUs, differences often come from OpenMP parallel loops—threads may sum numbers in a different order, producing slightly different results. On GPUs, parallelism and library implementations (like cuBLAS or cuDNN) can do the same.

These differences are usually very small, but deep learning systems are chaotic: tiny changes in the early steps can grow into visible differences later. This doesn't mean the code is wrong—it just means floating-point math has limits.

Why reproducibility matters

Reproducibility isn't just about peace of mind. It has real uses:

- Debugging: if a bug appears, you want to reproduce the exact same run to diagnose it.
- Comparisons: when testing new optimizers, schedulers, or architectures, you want fair comparisons on identical conditions.
- Science: reproducible results are essential for research papers and benchmarks.

At the same time, absolute bit-for-bit reproducibility is often unrealistic in large parallel systems. Instead, the goal is practical reproducibility: ensuring that runs are *similar enough* to reach the same conclusions.

Example experiment

Suppose you seed training with `manual_seed(1337)` and use the same dataset. You might get a loss curve like this:

Run A: `step 1000 → val loss 3.42`

Run B: `step 1000 → val loss 3.43`

The numbers are not identical, but they are close. The important part is that the model's learning trajectory is stable and results are comparable.

If you remove the seed and allow full randomness, you might get:

Run A: `step 1000 → val loss 3.42`

Run B: `step 1000 → val loss 3.89`

Both are valid, but harder to compare.

Try it yourself

1. Run training twice without setting a seed. Compare how training loss and validation loss differ at step 500.
2. Set a fixed seed with `manual_seed(42)` before building the model. Run training twice and compare again. You should see closer numbers.
3. Enable OpenMP with multiple threads and then run with a single thread. Notice how results differ slightly due to floating-point summation order.
4. Save two checkpoints from runs with different seeds. Use the model to generate text and compare outputs. You'll see different wording, but both grammatically plausible.
5. Increase the dataset size and check if differences between runs shrink. With more data, randomness matters less.

The takeaway

Reproducibility in training is about controlling randomness where possible and accepting small divergences where not. In `llm.c`, reproducibility is made clear through simple seeding functions and deterministic dataloader options. Perfect bit-level reproducibility isn't the point-the goal is to ensure results are stable, comparable, and scientifically sound, even if tiny numerical differences creep in.

49. Command-Line Flags and Defaults

When you run a training program, you often want to change certain settings without editing the source code. For example, you might want to try a different batch size, adjust the learning rate, or train for more steps. Command-line flags make this possible. In `train_gpt2.c`, defaults are set inside the program, but it can also be compiled to accept arguments, giving you flexibility while keeping the code minimal.

Why flags exist

Deep learning experiments are highly sensitive to hyperparameters-values like learning rate, batch size, sequence length, or number of steps. If every change required modifying source code, recompiling, and rerunning, experimentation would be slow and error-prone. Flags allow you to configure these parameters quickly at runtime.

In many large frameworks (like PyTorch or TensorFlow), command-line arguments are parsed with helper libraries. In `llm.c`, the philosophy is simplicity: flags are either defined in code as constants, or you can extend `main` with standard C argument parsing to override defaults.

Defaults in `train_gpt2.c`

Looking at the code, here are the main defaults hardcoded in the `main` function:

- Batch size (B):

```
int B = 4; // number of sequences per batch
```

- Sequence length (T):

```
int T = 64; // tokens per sequence
```

- Validation batches:

```
int val_num_batches = 5;
```

- Training steps:

```
for (int step = 0; step <= 40; step++) {
```

By default, only 40 steps are run in this example.

- Optimizer hyperparameters (inside `gpt2_update` call):

```
gpt2_update(&model, 1e-4f, 0.9f, 0.999f, 1e-8f, 0.0f, step+1);
```

Here the learning rate is `1e-4`, beta values for AdamW are 0.9 and 0.999, epsilon is `1e-8`, and weight decay is 0.0.

These defaults are chosen to make the reference training loop run quickly and predictably, especially on small datasets like Tiny Shakespeare or Tiny Stories.

How to add flags

If you want flexibility, you can extend `main` with argument parsing:

```
int main(int argc, char argv) {
    int B = 4;
    int T = 64;
    int max_steps = 40;
    if (argc > 1) B = atoi(argv[1]);
    if (argc > 2) T = atoi(argv[2]);
    if (argc > 3) max_steps = atoi(argv[3]);
    ...
}
```

Now you can run:

```
./train_gpt2 8 128 100
```

This sets batch size to 8, sequence length to 128, and steps to 100, without changing source code.

Why it matters

Command-line flags make experimentation far more efficient. You can try multiple configurations in one day without recompiling or editing the file repeatedly. This is especially useful when running jobs on clusters where you want scripts that launch many experiments automatically with different parameters.

Defaults are equally important: they give you a safe, predictable starting point. Beginners can run the code without thinking about flags, while advanced users can override values as needed.

Try it yourself

1. Keep the default batch size of 4 and sequence length of 64. Run training and note the time per step.
2. Change batch size to 8 by editing the code. Observe how training speed changes and how memory usage increases.
3. Modify the loop to train for 200 steps instead of 40. Watch how loss decreases further.
4. Add argument parsing to accept learning rate as a flag. Experiment with $1e-3$ vs. $1e-5$ and see how quickly training diverges or stalls.
5. Create a shell script that runs training multiple times with different values for B and T. Compare results.

The takeaway

Command-line flags and defaults balance simplicity with flexibility. Defaults make the code runnable out of the box, while flags let you scale experiments without constantly editing the source. In `train_gpt2.c`, this design keeps the training loop minimal but still adaptable, encouraging both clarity and experimentation.

50. Example CPU Training Logs and Outputs

One of the best ways to understand what a training loop is doing is by reading its logs. Logs are the program's way of telling you how training is progressing: what the loss is, how fast it's running, and whether validation checks are improving. In `train_gpt2.c`, logging is deliberately minimal so you can easily see the essentials without being overwhelmed.

What the logs look like

Here's a snippet of output from running the CPU training loop on Tiny Shakespeare:

```
train dataset num_batches: 1192
val dataset num_batches: 128
[GPT-2]
max_seq_len: 1024
vocab_size: 50257
padded_vocab_size: 50304
num_layers: 12
num_heads: 12
channels: 768
num_parameters: 124475904
num_activations: 73347840
val loss 5.325529
step 0: train loss 4.677779 (took 1987.546000 ms)
step 1: train loss 5.191576 (took 1927.230000 ms)
step 2: train loss 4.438685 (took 1902.987000 ms)
...
```

Each part of this output has meaning:

- Dataset sizes: how many training and validation batches are available.
- Model config: confirmation that the GPT-2 model was loaded correctly (sequence length, vocab size, number of layers, etc.).
- Validation loss: an average measure of how well the model is doing on unseen data.
- Training step logs: for each step, you see the training loss and how long the step took in milliseconds.

Understanding loss values

Loss is the number that tells us how far the model's predictions are from the correct answers. Lower is better.

- A loss around 5.3 means the model is essentially guessing.
- As training progresses, you want to see this number slowly decrease.
- If the number gets stuck, or goes up, it can indicate problems with the learning rate, dataset, or implementation.

Think of it like a report card: at the beginning, the model is failing every test, but as it practices (trains), the grades (loss values) improve.

Speed measurements

The “took ... ms” part shows how long each step took. On CPU, this is usually slow, sometimes a couple of seconds per step. On GPU, the same step might only take tens of milliseconds.

Timing logs are useful because they help you:

- Estimate how long full training will take.
- Compare performance between machines.
- Spot problems if training suddenly slows down.

Occasional validation checks

Every few steps, the code switches to validation data and prints a `val loss`. This is crucial: training loss always goes down if the model memorizes the training set, but validation loss tells you if it is *actually learning patterns* that generalize.

If training loss goes down but validation loss stays high, that’s a sign of overfitting.

Generated samples

At certain steps, the code also prints generated text like this:

generating:

```
The King had not
that the Duke of Northumberland and the Duke of
...
```

Even though the text might look strange at first, it’s a powerful sign that the model is learning. At the beginning, output is pure gibberish, but as training continues, you start to see recognizable words and patterns.

Why it matters

Logs are your window into the training process. Without them, training would be a black box—you'd wait hours and have no idea if it was working. By watching loss curves, step times, and sample outputs, you can make informed adjustments and gain confidence that the model is on the right track.

Try it yourself

1. Run the training loop as-is and save the console output. Mark how loss changes between step 0 and step 40.
2. Increase the number of steps to 200 and compare how the losses evolve.
3. Change the batch size from 4 to 8 and note both the training speed and the loss behavior.
4. Edit the code to print validation loss every step instead of every 10 steps. Does the trend look smoother?
5. Save the generated samples at steps 20 and 40. Compare how the quality changes.

The takeaway

Training logs are like a diary of the model's progress. They show you how quickly the model is learning, how well it generalizes, and how fast the computation runs. By reading and interpreting logs carefully, you can guide experiments, detect problems early, and appreciate the progress that's happening inside the model.