nanochat: Building a ChatGPT from Scratch

A Comprehensive Educational Guide

Based on nanochat by Andrej Karpathy

Vibe Written by Matt Suiche (msuiche) with Claude Code

October 21, 2025

Contents

Chapter 1

Introduction to nanochat: Building a ChatGPT from Scratch

1.1 What is nanochat?

nanochat is a complete, minimal implementation of a Large Language Model (LLM) similar to ChatGPT. Unlike most LLM projects that rely on heavy external frameworks, nanochat is built from scratch with minimal dependencies, making it perfect for learning how modern LLMs actually work.

Key Philosophy: - **From Scratch**: Implement core algorithms yourself rather than using black-box libraries - **Minimal Dependencies**: Only essential libraries (PyTorch, tokenizers, etc.) - **Educational**: Clean, readable code that you can understand completely - **Full Stack**: Everything from tokenization to web serving - **Practical**: Actually trains a working model for ~$100

1.2 What You'll Learn

By studying this repository, you will understand:

1. **Tokenization**: How text is converted to numbers using Byte Pair Encoding (BPE)
2. **Model Architecture**: The Transformer architecture with modern improvements
3. **Training Pipeline**:
    - **Pretraining**: Learning language patterns from raw text
    - **Midtraining**: Specialized training on curated data
    - **Supervised Fine-Tuning (SFT)**: Teaching the model to chat
    - **Reinforcement Learning (RL)**: Optimizing for quality
4. **Optimization**: Advanced optimizers like Muon and AdamW
5. **Evaluation**: Measuring model performance
6. **Inference**: Running the trained model efficiently

7. **Deployment**: Serving the model via a web interface

1.3 Repository Structure

```
nanochat/
  nanochat/              # Core library
      gpt.py            # GPT model architecture
      tokenizer.py      # BPE tokenizer wrapper
      dataloader.py     # Data loading and tokenization
      engine.py         # Inference engine
      adamw.py          # AdamW optimizer
      muon.py           # Muon optimizer
      ...
  rustbpe/               # High-performance Rust tokenizer
      src/lib.rs        # BPE implementation in Rust
  scripts/               # Training and evaluation scripts
      base_train.py     # Pretraining script
      mid_train.py      # Midtraining script
      chat_sft.py       # Supervised fine-tuning
      chat_rl.py        # Reinforcement learning
      chat_web.py       # Web interface
  tasks/                 # Evaluation benchmarks
  tests/                 # Unit tests
  speedrun.sh            # Complete pipeline script
```

1.4 The Training Pipeline

nanochat implements the complete modern LLM training pipeline:

1.4.1 1. Tokenization (tok_train.py)

First, we need to convert text into numbers. We train a **Byte Pair Encoding (BPE)** tokenizer on a corpus of text. This creates a vocabulary of ~32,000 tokens that efficiently represent common words and subwords.

Time: ~10 minutes on CPU

1.4.2 2. Base Pretraining (base_train.py)

The model learns to predict the next token in sequences of text. This is where most of the "knowledge" is learned - language patterns, facts, reasoning abilities, etc.

Data: ~10 billion tokens from FineWeb (high-quality web text) **Objective**: Next-token prediction **Time**: ~2-4 hours on 8×H100 GPUs **Cost**: ~$100

1.4.3 3. Midtraining (mid_train.py)

Continue pretraining on a smaller, more curated dataset to improve quality and reduce the need for instruction following data.

Data: ~1 billion high-quality tokens **Time**: ~30 minutes **Cost**: ~$12

1.4.4 4. Supervised Fine-Tuning (chat_sft.py)

Teach the model to follow instructions and chat like ChatGPT. We train on conversation examples.

Data: ~80,000 conversations from SmolTalk **Objective**: Predict assistant responses given user prompts **Time**: ~15 minutes **Cost**: ~$6

1.4.5 5. Reinforcement Learning (chat_rl.py)

Further optimize the model using reinforcement learning to improve response quality.

Technique: Self-improvement via sampling and filtering **Time**: ~10 minutes **Cost**: ~$4

1.5 Key Technical Features

1.5.1 Modern Architecture Choices

The GPT model in nanochat includes modern improvements over the original GPT-2:

1. **Rotary Position Embeddings (RoPE)**: Better position encoding
2. **RMSNorm**: Simpler, more efficient normalization
3. **Multi-Query Attention (MQA)**: Faster inference
4. **QK Normalization**: Stability improvement
5. **ReLU² Activation**: Better than GELU for small models
6. **Untied Embeddings**: Separate input/output embeddings
7. **Logit Softcapping**: Prevents extreme logits

1.5.2 Efficient Implementation

- **Mixed Precision**: BF16 for most operations
- **Gradient Accumulation**: Larger effective batch sizes
- **Distributed Training**: Multi-GPU support with DDP
- **Compiled Models**: PyTorch compilation for speed
- **Streaming Data**: Memory-efficient data loading
- **Rust Tokenizer**: Fast tokenization with parallel processing

1.6 Mathematical Notation

Throughout this guide, we'll use the following notation:

- d_{model}: Model dimension (embedding size)
- n_{layers}: Number of Transformer layers
- n_{heads}: Number of attention heads
- d_{head}: Dimension per attention head (d_{model}/n_{heads})
- V: Vocabulary size
- T or L: Sequence length
- B: Batch size
- θ: Model parameters
- $\mathcal{L}$: Loss function
- $p(x)$: Probability distribution

1.7 Prerequisites

To fully understand this material, you should have:

Essential: - Python programming - Basic linear algebra (matrices, vectors, dot products) - Basic calculus (derivatives, chain rule) - Basic probability (distributions, expectation)

Helpful but not required: - PyTorch basics - Deep learning fundamentals - Transformer architecture awareness

Don't worry if you're not an expert! We'll explain everything step by step.

1.8 How to Use This Guide

The educational materials are organized as follows:

1. **01_introduction.md** (this file): Overview and context
2. **02_mathematical_foundations.md**: Math concepts you need
3. **03_tokenization.md**: BPE algorithm and implementation
4. **04_transformer_architecture.md**: The GPT model structure
5. **05_attention_mechanism.md**: Self-attention in detail
6. **06_training_process.md**: How training works
7. **07_optimization.md**: Advanced optimizers (Muon, AdamW)
8. **08_implementation_details.md**: Code walkthrough
9. **09_evaluation.md**: Measuring model performance
10. **10_rust_implementation.md**: High-performance Rust tokenizer

Each section builds on previous ones, so it's best to read them in order.

1.9 Running the Code

To get started with nanochat:

```
# Clone the repository
git clone https://github.com/karpathy/nanochat.git
cd nanochat

# Install dependencies (requires Python 3.10+)
pip install uv
uv sync

# Run the complete pipeline (requires 8×H100 GPUs)
bash speedrun.sh
```

For learning purposes, you can also:

```
# Run tests
python -m pytest tests/ -v

# Train tokenizer only
python -m scripts.tok_train

# Train small model on 1 GPU
python -m scripts.base_train --depth=6
```

1.10 Next Steps

In the next section, we'll cover the **Mathematical Foundations** - all the math concepts you need to understand how LLMs work, explained from first principles.

Let's begin!

Chapter 2

Mathematical Foundations

This section covers all the mathematical concepts you need to understand LLMs. We'll start from basics and build up to the complex operations used in modern Transformers.

2.1 1. Linear Algebra Essentials

2.1.1 1.1 Vectors and Matrices

Vectors are lists of numbers. In deep learning, we use vectors to represent: - Word embeddings: `[0.2, -0.5, 0.8, ...]` - Hidden states: representations of tokens at each layer

Matrices are 2D arrays of numbers. We use them for: - Linear transformations: $y = Wx + b$ - Attention scores - Weight parameters

Notation: - Vectors: lowercase bold $\mathbf{x} \in \mathbb{R}^d$ - Matrices: uppercase bold $\mathbf{W} \in \mathbb{R}^{m \times n}$ - Scalars: regular letters a, b, c

2.1.2 1.2 Matrix Multiplication

The fundamental operation in neural networks.

Given $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$:

$$\mathbf{C} = \mathbf{AB} \quad \text{where} \quad C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj}$$

Example in Python:

```
import torch

A = torch.randn(3, 4)  # 3×4 matrix
```

```
B = torch.randn(4, 5)  # 4×5 matrix
C = A @ B              # 3×5 matrix (@ is matrix multiplication)
```

Computational Cost: $O(m \times n \times k)$ operations

2.1.3 1.3 Dot Product

The dot product of two vectors measures their similarity:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{d} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_d b_d$$

Geometric Interpretation:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta)$$

where θ is the angle between vectors.

Key Properties: - If $\mathbf{a} \cdot \mathbf{b} > 0$: vectors point in similar directions - If $\mathbf{a} \cdot \mathbf{b} = 0$: vectors are orthogonal (perpendicular) - If $\mathbf{a} \cdot \mathbf{b} < 0$: vectors point in opposite directions

2.1.4 1.4 Norms

The **L2 norm** (Euclidean norm) measures vector magnitude:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{d} x_i^2}$$

Normalization scales a vector to unit length:

$$\text{normalize}(\mathbf{x}) = \frac{\mathbf{x}}{\|\mathbf{x}\|_2}$$

This is used in RMSNorm and QK normalization.

2.2 2. Probability and Information Theory

2.2.1 2.1 Probability Distributions

A **probability distribution** $p(x)$ assigns probabilities to outcomes: - $p(x) \geq 0$ for all x - $\sum_x p(x) = 1$ (discrete) or $\int p(x)dx = 1$ (continuous)

Language Modeling is about learning $p(\text{next word} | \text{previous words})$.

2.2.2 2.2 Conditional Probability

Given events A and B:

$$p(A|B) = \frac{p(A \cap B)}{p(B)}$$

In language models, we compute:

$$p(\text{sentence}) = p(w_1) \cdot p(w_2|w_1) \cdot p(w_3|w_1, w_2) \cdots$$

2.2.3 2.3 Cross-Entropy Loss

Cross-entropy measures the difference between two probability distributions.

For a true distribution q and predicted distribution p:

$$H(q, p) = -\sum_x q(x) \log p(x)$$

In language modeling: - q is the true distribution (1 for correct token, 0 for others) - p is our model's predicted probability distribution

This simplifies to:

$$\mathcal{L} = -\log p(\text{correct token})$$

Example:

```
# Suppose vocabulary size = 4, correct token = 2
logits = torch.tensor([2.0, 1.0, 3.0, 0.5])  # Model outputs
target = 2  # Correct token

# Compute cross-entropy
loss = F.cross_entropy(logits.unsqueeze(0), torch.tensor([target]))
# This is: -log(softmax(logits)[2])
```

2.2.4 2.4 KL Divergence

Kullback-Leibler divergence measures how one distribution differs from another:

$$D_{KL}(p\|q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

Properties: - $D_{KL}(p\|q) \geq 0$ always - $D_{KL}(p\|q) = 0$ if and only if $p = q$ - Not symmetric: $D_{KL}(p\|q) \neq D_{KL}(q\|p)$

Used in some advanced training techniques like KL-regularized RL.

2.3 3. Calculus and Optimization

2.3.1 3.1 Derivatives

The derivative measures how a function changes:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Partial derivatives for functions of multiple variables:

$$\frac{\partial f}{\partial x_i}$$

measures change with respect to x_i while holding other variables constant.

2.3.2 3.2 Gradient

The **gradient** is the vector of all partial derivatives:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right]$$

The gradient points in the direction of steepest increase.

2.3.3 3.3 Chain Rule

For composite functions $f(g(x))$:

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$$

Backpropagation is just repeated application of the chain rule!

2.3.4 3.4 Gradient Descent

To minimize a function $\mathcal{L}(\theta)$:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t)$$

where: - θ: parameters - η: learning rate - $\nabla_\theta \mathcal{L}$: gradient of loss with respect to parameters

Stochastic Gradient Descent (SGD): Use a small batch of data to estimate gradient.

2.4 4. Neural Network Operations

2.4.1 4.1 Linear Transformation

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where: - $\mathbf{x} \in \mathbb{R}^{d_{in}}$: input - $\mathbf{W} \in \mathbb{R}^{d_{out} \times d_{in}}$: weight matrix - $\mathbf{b} \in \mathbb{R}^{d_{out}}$: bias vector (often omitted in modern architectures) - $\mathbf{y} \in \mathbb{R}^{d_{out}}$: output

In PyTorch:

```
linear = nn.Linear(d_in, d_out, bias=False)
y = linear(x)
```

2.4.2 4.2 Activation Functions

Activation functions introduce non-linearity.

ReLU (Rectified Linear Unit):

$$\text{ReLU}(x) = \max(0, x)$$

Squared ReLU (used in nanochat):

$$\text{ReLU}^2(x) = \max(0, x)^2$$

GELU (Gaussian Error Linear Unit):

$$\text{GELU}(x) = x \cdot \Phi(x)$$

where Φ is the Gaussian CDF.

Tanh:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

2.4.3 4.3 Softmax

Converts logits to a probability distribution:

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Properties: - Outputs sum to 1: $\sum_i \text{softmax}(\mathbf{x})_i = 1$ - All outputs in $(0, 1)$ - Higher input values get higher probabilities

Temperature scaling:

$$\text{softmax}(\mathbf{x}/T)_i = \frac{e^{x_i/T}}{\sum_{j=1}^{n} e^{x_j/T}}$$

- Higher T: more uniform distribution (more random)
- Lower T: more peaked distribution (more deterministic)

2.4.4 4.4 Layer Normalization

LayerNorm normalizes activations:

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

where: - $\mu = \frac{1}{d}\sum_i x_i$: mean - $\sigma^2 = \frac{1}{d}\sum_i (x_i - \mu)^2$: variance - γ, β: learnable parameters - ϵ: small constant for numerical stability

RMSNorm (used in nanochat) is simpler:

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_i x_i^2 + \epsilon}}$$

No learnable parameters, just normalization!

Implementation:

```
def norm(x):
    return F.rms_norm(x, (x.size(-1),))
```

2.5 5. Attention Mechanism Mathematics

2.5.1 5.1 Scaled Dot-Product Attention

The core of the Transformer:

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

where: - $Q \in \mathbb{R}^{T \times d_k}$: Queries - $K \in \mathbb{R}^{T \times d_k}$: Keys - $V \in \mathbb{R}^{T \times d_v}$: Values - d_k: dimension of keys/queries

Step by step:

1. **Compute similarity scores:** $S = QK^T \in \mathbb{R}^{T \times T}$
   - S_{ij} = how much query i attends to key j
2. **Scale:** $S' = S/\sqrt{d_k}$
   - Prevents gradients from vanishing/exploding
3. **Softmax:** $A = \text{softmax}(S')$
   - Convert to probabilities (each row sums to 1)
4. **Weighted sum:** Output $= AV$
   - Aggregate values weighted by attention

Why scaling by $\sqrt{d_k}$?

For random vectors, QK^T has variance $\propto d_k$. Scaling keeps variance stable, preventing softmax saturation.

2.5.2 5.2 Multi-Head Attention

Split into multiple "heads" for different representation subspaces:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where each head is:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Parameters: - $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{model} \times d_k}$: projection matrices - $W^O \in \mathbb{R}^{hd_v \times d_{model}}$: output projection

2.5.3 5.3 Causal Masking

For autoregressive language models, we must prevent attending to future tokens:

$$\text{mask}_{ij} = \begin{cases} 0 & \text{if } i < j \\ -\infty & \text{if } i \geq j \end{cases}$$

Add mask before softmax:

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + \text{mask}\right)$$

The $-\infty$ values become 0 after softmax.

2.6 6. Positional Encodings

Transformers have no inherent notion of position. We add positional information.

2.6.1 6.1 Sinusoidal Positional Encoding (Original Transformer)

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

2.6.2 6.2 Rotary Position Embeddings (RoPE)

Used in nanochat! Encode position by rotating key/query vectors:

$$\mathbf{q}_m = R_m \mathbf{q}, \quad \mathbf{k}_n = R_n \mathbf{k}$$

where R_θ is a rotation matrix. The dot product $\mathbf{q}_m^T \mathbf{k}_n$ depends only on relative position $m - n$.

For 2D case:

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Implementation:

```
def apply_rotary_emb(x, cos, sin):
    d = x.shape[3] // 2
    x1, x2 = x[..., :d], x[..., d:]
    y1 = x1 * cos + x2 * sin
    y2 = x1 * (-sin) + x2 * cos
    return torch.cat([y1, y2], 3)
```

2.7 7. Optimization Algorithms

2.7.1 7.1 Momentum

Accumulates past gradients for smoother updates:

$$v_t = \beta v_{t-1} + (1 - \beta) g_t$$

$$\theta_t = \theta_{t-1} - \eta v_t$$

where: - $g_t = \nabla \mathcal{L}(\theta_{t-1})$: current gradient - v_t: velocity (exponential moving average of gradients) - β: momentum coefficient (typically 0.9)

2.7.2 7.2 Adam/AdamW

Adaptive learning rates for each parameter:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

AdamW adds weight decay:

$$\theta_t = (1-\lambda)\theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Typical values: - $\beta_1 = 0.9$ - $\beta_2 = 0.999$ (sometimes 0.95 for LLMs) - $\epsilon = 10^{-8}$ - $\lambda = 0.01$ (weight decay)

2.7.3 7.3 Learning Rate Schedules

Warmup: Gradually increase LR at the start

$$\eta_t = \eta_{max} \cdot \min(1, t/T_{warmup})$$

Cosine decay:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min}) \left(1 + \cos\left(\frac{t\pi}{T_{max}}\right)\right)$$

Linear warmup + cosine decay (common for LLMs)

2.8 8. Information Theory for LLMs

2.8.1 8.1 Entropy

Measures uncertainty in a distribution:

$$H(p) = -\sum_x p(x) \log_2 p(x)$$

Units: **bits** (with $\log_2$) or **nats** (with ln)

2.8.2 8.2 Perplexity

Perplexity is the exponentiated cross-entropy:

$$\text{PPL} = 2^{H(q,p)} = \exp(H(q,p))$$

Interpretation: "effective vocabulary size" - how many choices the model is uncertain between.

Lower perplexity = better model.

2.8.3 8.3 Bits Per Byte (BPB)

For byte-level tokenization:

$$\text{BPB} = \frac{H(q,p)}{\log_2(256)}$$

Measures how many bits needed to encode each byte. Used in nanochat for evaluation.

2.9 Summary: Key Equations

Concept	Equation
Linear layer	$y = Wx + b$
Softmax	$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$
Cross-entropy	$\mathcal{L} = -\sum_i y_i \log(\hat{y}_i)$
Attention	$\text{Attention}(Q,K,V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
RMSNorm	$\text{RMS}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum x_i^2}}$
Gradient descent	$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}$

2.10 Next Steps

Now that we have the mathematical foundations, we'll dive into **Tokenization** - how we convert text into numbers that the model can process.

Chapter 3

Tokenization: Byte Pair Encoding (BPE)

3.1 Why Tokenization?

Neural networks work with numbers, not text. **Tokenization** converts text into numerical sequences that models can process.

Why not just use ASCII codes? - English uses ~100 common characters - But common words like "the", "and", "ing" appear constantly - Better to have single tokens for frequent sequences - Reduces sequence length and captures semantic meaning

3.2 Tokenization Approaches

1. **Character-level**: Each character is a token
   - Pros: Small vocabulary, handles any text
   - Cons: Very long sequences, doesn't capture word meaning
2. **Word-level**: Each word is a token
   - Pros: Captures semantic meaning
   - Cons: Huge vocabulary, can't handle unknown words
3. **Subword-level** (BPE, WordPiece): Balance between characters and words
   - Pros: Moderate vocabulary, handles rare words, captures common patterns
   - Cons: Slightly complex to implement
   - **This is what nanochat uses!**

3.3 Byte Pair Encoding (BPE) Algorithm

BPE builds a vocabulary by iteratively merging the most frequent pairs of tokens.

3.3.1 The Training Algorithm

Input: Corpus of text, desired vocabulary size V

Output: Merge rules and vocabulary

Steps:

1. **Initialize vocabulary** with all 256 bytes (0-255)
2. **Split text** into chunks using a regex pattern
3. **Convert chunks** to sequences of byte tokens
4. **Repeat** $V - 256$ times:
   - Find the most frequent **pair** of adjacent tokens
   - **Merge** this pair into a new token
   - **Replace** all occurrences of the pair with the new token
5. **Save** the merge rules

3.3.2 Example by Hand

Let's tokenize "aaabdaaabac" with vocab_size = 259 (256 bytes + 3 merges).

Initial state: Convert to bytes

```
text = "aaabdaaabac"
tokens = [97, 97, 97, 98, 100, 97, 97, 97, 98, 97, 99]  # ASCII codes
```

Iteration 1: Find most frequent pair

```
Pairs: (97,97) appears 4 times ← most frequent
       (97,98) appears 2 times
       (98,100) appears 1 time
       ...

Merge (97,97) → 256
New tokens: [256, 97, 98, 100, 256, 97, 98, 97, 99]
```

Iteration 2:

```
Pairs: (256,97) appears 2 times ← most frequent
       (97,98) appears 2 times (tie-break by lexicographic order)
       ...

Merge (256,97) → 257
New tokens: [257, 98, 100, 257, 98, 97, 99]
```

Iteration 3:

```
Pairs: (257,98) appears 2 times ← most frequent
       ...

Merge (257,98) → 258
Final tokens: [258, 100, 258, 97, 99]
```

We've compressed 11 tokens → 5 tokens!

3.4 Implementation in nanochat

nanochat provides **two tokenizer implementations**:

1. **HuggingFaceTokenizer**: Python-based, easy to use but slower
2. **RustBPETokenizer**: High-performance Rust implementation (preferred)

Both implement the same GPT-4 style BPE algorithm.

3.4.1 File: `nanochat/tokenizer.py`

Let's examine the key components:

3.4.1.1 Special Tokens

```python
SPECIAL_TOKENS = [
    "<|bos|>",           # Beginning of sequence (document delimiter)
    "<|user_start|>",    # User message start
    "<|user_end|>",      # User message end
    "<|assistant_start|>",  # Assistant message start
    "<|assistant_end|>",    # Assistant message end
    "<|python_start|>",  # Python tool call start
    "<|python_end|>",    # Python tool call end
    "<|output_start|>",  # Python output start
    "<|output_end|>",    # Python output end
]
```

These special tokens are added to the vocabulary for chat formatting.

3.4.1.2 Text Splitting Pattern

```python
SPLIT_PATTERN = r"""'(?i:[sdmt]|ll|ve|re)|[^\r\n\p{L}\p{N}]?+\p{L}+|\p{N}{1,2}| ?[^\s\p{L}\p{N
```

This regex pattern splits text before BPE: - `'(?i:[sdmt]|ll|ve|re)`: Contractions like 's, 't, 'll, 've, 're - `[^\r\n\p{L}\p{N}]?+\p{L}+`: Optional non-letter + letters (words) - `\p{N}{1,2}`: Num-

bers (1-2 digits, not 3 like GPT-4) - `?[^\s\p{L}\p{N}]++[\r\n]*`: Optional space + punctuation - `\s*[\r\n]|\s+(?!\S)|\s+`: Whitespace handling

Why this pattern? It groups text into chunks that are semantically meaningful, making BPE more effective.

3.4.2 RustBPETokenizer Class

The main tokenizer interface (`nanochat/tokenizer.py:155`):

```
class RustBPETokenizer:
    """Light wrapper around tiktoken (for efficient inference) but train with rustbpe"""

    def __init__(self, enc, bos_token):
        self.enc = enc  # tiktoken.Encoding object
        self.bos_token_id = self.encode_special(bos_token)

    @classmethod
    def train_from_iterator(cls, text_iterator, vocab_size):
        # 1) Train using rustbpe
        tokenizer = rustbpe.Tokenizer()
        vocab_size_no_special = vocab_size - len(SPECIAL_TOKENS)
        tokenizer.train_from_iterator(text_iterator, vocab_size_no_special, pattern=SPLIT_PATTE

        # 2) Construct tiktoken encoding for fast inference
        pattern = tokenizer.get_pattern()
        mergeable_ranks_list = tokenizer.get_mergeable_ranks()
        mergeable_ranks = {bytes(k): v for k, v in mergeable_ranks_list}

        # Add special tokens
        tokens_offset = len(mergeable_ranks)
        special_tokens = {name: tokens_offset + i for i, name in enumerate(SPECIAL_TOKENS)}

        # Create tiktoken encoding
        enc = tiktoken.Encoding(
            name="rustbpe",
            pat_str=pattern,
            mergeable_ranks=mergeable_ranks,
            special_tokens=special_tokens,
        )
        return cls(enc, "<|bos|>")
```

Design choice: Train with Rust (fast), infer with tiktoken (also fast, battle-tested).

3.4.2.1 Encoding Text

```
def encode(self, text, prepend=None, append=None, num_threads=8):
    # Prepare special tokens
    if prepend is not None:
        prepend_id = prepend if isinstance(prepend, int) else self.encode_special(prepend)
    if append is not None:
        append_id = append if isinstance(append, int) else self.encode_special(append)

    if isinstance(text, str):
        # Single string
        ids = self.enc.encode_ordinary(text)
        if prepend is not None:
            ids.insert(0, prepend_id)
        if append is not None:
            ids.append(append_id)
    elif isinstance(text, list):
        # Batch of strings (parallel processing)
        ids = self.enc.encode_ordinary_batch(text, num_threads=num_threads)
        if prepend is not None:
            for ids_row in ids:
                ids_row.insert(0, prepend_id)
        if append is not None:
            for ids_row in ids:
                ids_row.append(append_id)

    return ids
```

Key features: - Supports single strings or batches - Optional prepend/append (e.g., BOS token) - Parallel processing for batches

3.4.2.2 Chat Conversation Rendering

For supervised fine-tuning, we need to convert conversations to tokens:

```
def render_conversation(self, conversation, max_tokens=2048):
    """
    Tokenize a single Chat conversation.
    Returns:
    - ids: list[int] - token ids
```

```
    - mask: list[int] - 1 for tokens to train on, 0 otherwise
    """
    ids, mask = [], []

    def add_tokens(token_ids, mask_val):
        if isinstance(token_ids, int):
            token_ids = [token_ids]
        ids.extend(token_ids)
        mask.extend([mask_val] * len(token_ids))

    # Get special token IDs
    bos = self.get_bos_token_id()
    user_start, user_end = self.encode_special("<|user_start|>"), self.encode_special("<|user_
    assistant_start, assistant_end = self.encode_special("<|assistant_start|>"), self.encode_sp

    # Add BOS token (not trained on)
    add_tokens(bos, 0)

    # Process messages
    for i, message in enumerate(messages):
        if message["role"] == "user":
            # User messages: not trained on
            value_ids = self.encode(message["content"])
            add_tokens(user_start, 0)
            add_tokens(value_ids, 0)
            add_tokens(user_end, 0)
        elif message["role"] == "assistant":
            # Assistant messages: TRAINED ON (mask=1)
            add_tokens(assistant_start, 0)
            value_ids = self.encode(message["content"])
            add_tokens(value_ids, 1)  # ← This is what we train on!
            add_tokens(assistant_end, 1)

    # Truncate if too long
    ids = ids[:max_tokens]
    mask = mask[:max_tokens]
    return ids, mask
```

The mask is crucial! We only compute loss on assistant responses, not user prompts.

3.5 Rust Implementation: `rustbpe/src/lib.rs`

The Rust implementation is highly optimized for speed. Let's examine the core components.

3.5.1 Data Structures

```rust
type Pair = (u32, u32);  // Pair of token IDs

#[pyclass]
pub struct Tokenizer {
    /// Maps pairs of token IDs to their merged token ID
    pub merges: StdHashMap<Pair, u32>,
    /// The regex pattern used for text splitting
    pub pattern: String,
    /// Compiled regex for efficiency
    compiled_pattern: Regex,
}
```

3.5.1.1 Word Representation

```rust
struct Word {
    ids: Vec<u32>,  // Sequence of token IDs
}

impl Word {
    fn pairs<'a>(&'a self) -> impl Iterator<Item = Pair> + 'a {
        self.ids.windows(2).map(|w| (w[0], w[1]))
    }
}
```

The `pairs()` method generates all adjacent pairs efficiently using sliding windows.

3.5.2 The Core Training Algorithm

Located at `rustbpe/src/lib.rs:164`:

```rust
fn train_core_incremental(&mut self, mut words: Vec<Word>, counts: Vec<i32>, vocab_size: u32)
    let num_merges = vocab_size - 256;  // 256 base bytes

    // 1. Initial pair counting (parallel!)
    let (mut pair_counts, mut where_to_update) = count_pairs_parallel(&words, &counts);
```

```
    // 2. Build max-heap of merge candidates
    let mut heap = OctonaryHeap::with_capacity(pair_counts.len());
    for (pair, pos) in where_to_update.drain() {
        let c = *pair_counts.get(&pair).unwrap_or(&0);
        if c > 0 {
            heap.push(MergeJob {
                pair,
                count: c as u64,
                pos,  // Set of word indices where this pair occurs
            });
        }
    }

    // 3. Merge loop
    for merges_done in 0..num_merges {
        // Get highest-count pair
        let Some(mut top) = heap.pop() else { break; };

        // Lazy refresh: check if count is still accurate
        let current = *pair_counts.get(&top.pair).unwrap_or(&0);
        if top.count != current as u64 {
            top.count = current as u64;
            if top.count > 0 {
                heap.push(top);
            }
            continue;
        }

        // Record merge
        let new_id = 256 + merges_done;
        self.merges.insert(top.pair, new_id);

        // Apply merge to all words containing this pair
        let mut local_pos_updates: AHashMap<Pair, AHashSet<usize>> = AHashMap::new();
        for &word_idx in &top.pos {
            let changes = words[word_idx].merge_pair(top.pair, new_id);

            // Update global pair counts
            for (pair, delta) in changes {
```

```
                let delta_total = delta * counts[word_idx];
                if delta_total != 0 {
                    *pair_counts.entry(pair).or_default() += delta_total;
                    if delta > 0 {
                        local_pos_updates.entry(pair).or_default().insert(word_idx);
                    }
                }
            }
        }

        // Re-add updated pairs to heap
        for (pair, pos) in local_pos_updates {
            let cnt = *pair_counts.get(&pair).unwrap_or(&0);
            if cnt > 0 {
                heap.push(MergeJob { pair, count: cnt as u64, pos });
            }
        }
    }
}
```

3.5.3 Key Optimizations

1. **Parallel Pair Counting:**

```
fn count_pairs_parallel(
    words: &[Word],
    counts: &[i32],
) -> (AHashMap<Pair, i32>, AHashMap<Pair, AHashSet<usize>>) {
    words
        .par_iter()  // Parallel iterator!
        .enumerate()
        .map(|(i, w)| {
            // Count pairs in this word
            let mut local_pc: AHashMap<Pair, i32> = AHashMap::new();
            let mut local_wtu: AHashMap<Pair, AHashSet<usize>> = AHashMap::new();
            if w.ids.len() >= 2 && counts[i] != 0 {
                for (a, b) in w.pairs() {
                    *local_pc.entry((a, b)).or_default() += counts[i];
                    local_wtu.entry((a, b)).or_default().insert(i);
                }
```

```
            }
            (local_pc, local_wtu)
        })
        .reduce(/* merge results */)
}
```

Uses **Rayon** for parallel processing across CPU cores.

2. **Efficient Merging:**

```
fn merge_pair(&mut self, pair: Pair, new_id: u32) -> Vec<(Pair, i32)> {
    let (a, b) = pair;
    let mut out: Vec<u32> = Vec::with_capacity(self.ids.len());
    let mut deltas: Vec<(Pair, i32)> = Vec::with_capacity(6);

    let mut i = 0;
    while i < self.ids.len() {
        if i + 1 < self.ids.len() && self.ids[i] == a && self.ids[i + 1] == b {
            // Found the pair to merge
            let left = out.last().copied();
            let right = if i + 2 < self.ids.len() { Some(self.ids[i + 2]) } else { None };

            // Track changes in pair counts
            if let Some(x) = left {
                deltas.push(((x, a), -1));      // Remove old pair
                deltas.push(((x, new_id), 1));  // Add new pair
            }
            deltas.push(((a, b), -1));          // Remove merged pair
            if let Some(y) = right {
                deltas.push(((b, y), -1));      // Remove old pair
                deltas.push(((new_id, y), 1));  // Add new pair
            }

            out.push(new_id);
            i += 2;  // Skip both tokens
        } else {
            out.push(self.ids[i]);
            i += 1;
        }
    }
```

```
    self.ids = out;
    deltas
}
```

Returns **delta updates** to pair counts, avoiding full recount.

3. **Lazy Heap Updates:**

Instead of updating heap immediately when counts change: - Pop top element - Check if count is still valid - If not, update and re-insert

This avoids expensive heap operations.

4. **Optimized Data Structures:**

- `AHashMap`: Fast hashmap from `ahash` crate
- `OctonaryHeap`: 8-ary heap (better cache locality than binary heap)
- `CompactString`: String optimized for short strings

3.5.4 Encoding with Trained Tokenizer

```
pub fn encode(&self, text: &str) -> Vec<u32> {
    let mut all_ids = Vec::new();

    // Split text using regex pattern
    for m in self.compiled_pattern.find_iter(text) {
        let chunk = m.expect("regex match failed").as_str();

        // Convert to byte tokens
        let mut ids: Vec<u32> = chunk.bytes().map(|b| b as u32).collect();

        // Apply merges iteratively
        while ids.len() >= 2 {
            // Find best pair to merge (lowest token ID = highest priority)
            let mut best_pair: Option<(usize, Pair, u32)> = None;

            for i in 0..ids.len() - 1 {
                let pair: Pair = (ids[i], ids[i + 1]);
                if let Some(&new_id) = self.merges.get(&pair) {
                    if best_pair.is_none() || new_id < best_pair.unwrap().2 {
                        best_pair = Some((i, pair, new_id));
                    }
                }
```

```
            }

            // Apply merge if found
            if let Some((idx, _pair, new_id)) = best_pair {
                ids[idx] = new_id;
                ids.remove(idx + 1);
            } else {
                break;  // No more merges
            }
        }

        all_ids.extend(ids);
    }

    all_ids
}
```

Greedy algorithm: Always merge the pair with the **lowest token ID** (= earliest in training).

3.6 Training the Tokenizer: `scripts/tok_train.py`

```
def main():
    # 1. Load data iterator
    shard_size = 250_000_000  # 250M characters per shard
    num_shards = 16           # ~4B characters total
    data_iterator = fineweb_shards_iterator(num_shards, shard_size)

    # 2. Train tokenizer
    tokenizer = RustBPETokenizer.train_from_iterator(
        data_iterator,
        vocab_size=32256,  # Common size for small models
    )

    # 3. Save tokenizer
    tokenizer_dir = os.path.join(get_base_dir(), "tokenizer")
    tokenizer.save(tokenizer_dir)

    # 4. Save token_bytes tensor for BPB evaluation
    token_bytes = compute_token_bytes(tokenizer)
```

```
    torch.save(token_bytes, os.path.join(tokenizer_dir, "token_bytes.pt"))
```

This streams data from FineWeb dataset and trains the tokenizer.

3.7 Usage Example

```
from nanochat.tokenizer import RustBPETokenizer

# Load trained tokenizer
tokenizer = RustBPETokenizer.from_directory("out/tokenizer")

# Encode text
text = "Hello, world! How are you?"
ids = tokenizer.encode(text, prepend="<|bos|>")
print(ids)  # [32256, 9906, 11, 995, 0, 1374, 389, 345, 30]

# Decode back
decoded = tokenizer.decode(ids)
print(decoded)  # "<|bos|>Hello, world! How are you?"

# Batch encoding (parallel)
texts = ["First sentence.", "Second sentence.", "Third sentence."]
batch_ids = tokenizer.encode(texts, prepend="<|bos|>", num_threads=4)
```

3.8 Why BPE Works

1. **Frequent patterns get single tokens**: “ing”, “the”, “er”
2. **Rare words split into subwords**: “unhappiness” → [“un”, “happiness”]
3. **Can handle any text**: Falls back to bytes for unknown sequences
4. **Compresses sequences**: Fewer tokens = faster training/inference

3.9 Performance Comparison

Implementation	Training Speed	Inference Speed
Python baseline	1×	1×
HuggingFace	~2×	~5×
Rust + tiktoken	**~20×**	**~50×**

The Rust implementation in nanochat is **dramatically faster** due to: - Parallel processing - Efficient data structures - No Python overhead - Compiled to native code

3.10 Next Steps

Now that we understand tokenization, we'll explore the **Transformer Architecture** - the neural network that processes these token sequences.

Chapter 4

Transformer Architecture: The GPT Model

The Transformer is the neural network architecture that powers modern LLMs. nanochat implements a **GPT-style decoder-only Transformer** with several modern improvements.

4.1 High-Level Architecture

```
Input: Token IDs [B, T]
  ↓
Token Embedding [B, T, D]
  ↓
Norm (RMSNorm)
  ↓
Transformer Block 1
    Self-Attention + Residual
    MLP + Residual
  ↓
Transformer Block 2
  ...
  ↓
Transformer Block N
  ↓
Norm (RMSNorm)
  ↓
Language Model Head [B, T, V]
  ↓
```

`Output: Logits for next token prediction`

Where: - B = Batch size - T = Sequence length - D = Model dimension (embedding size) - V = Vocabulary size - N = Number of layers

4.2 Model Configuration: `nanochat/gpt.py:26`

```python
@dataclass
class GPTConfig:
    sequence_len: int = 1024     # Maximum context length
    vocab_size: int = 50304      # Vocabulary size (padded to multiple of 64)
    n_layer: int = 12           # Number of Transformer blocks
    n_head: int = 6             # Number of query heads
    n_kv_head: int = 6          # Number of key/value heads (MQA)
    n_embd: int = 768           # Model dimension
```

Design choice: All sizes are chosen for GPU efficiency (multiples of 64/128).

4.3 The GPT Class

Full implementation: `nanochat/gpt.py:154`

```python
class GPT(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        # Core transformer components
        self.transformer = nn.ModuleDict({
            "wte": nn.Embedding(config.vocab_size, config.n_embd),  # Token embeddings
            "h": nn.ModuleList([Block(config, layer_idx) for layer_idx in range(config.n_layer)
        })

        # Language model head (unembedding)
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)

        # Precompute rotary embeddings
        self.rotary_seq_len = config.sequence_len * 10  # Over-allocate
        head_dim = config.n_embd // config.n_head
        cos, sin = self._precompute_rotary_embeddings(self.rotary_seq_len, head_dim)
        self.register_buffer("cos", cos, persistent=False)
```

```
        self.register_buffer("sin", sin, persistent=False)

        # Cast embeddings to BF16 (saves memory)
        self.transformer.wte.to(dtype=torch.bfloat16)
```

4.3.1 Key Architectural Choices

1. **No Positional Embeddings**: Uses RoPE (Rotary Position Embeddings) instead
2. **Untied Embeddings**: `wte` (input) and `lm_head` (output) are **separate**
   - Allows different learning rates
   - More parameters but better performance
3. **BFloat16 Embeddings**: Saves memory with minimal quality loss

4.4 Model Initialization: `nanochat/gpt.py:175`

```
def init_weights(self):
    self.apply(self._init_weights)

    # Zero-initialize output layers (residual path trick)
    torch.nn.init.zeros_(self.lm_head.weight)
    for block in self.transformer.h:
        torch.nn.init.zeros_(block.mlp.c_proj.weight)
        torch.nn.init.zeros_(block.attn.c_proj.weight)

    # Initialize rotary embeddings
    head_dim = self.config.n_embd // self.config.n_head
    cos, sin = self._precompute_rotary_embeddings(self.rotary_seq_len, head_dim)
    self.cos, self.sin = cos, sin
```

Residual path trick: Zero-initialize final layers in residual connections. - At initialization, blocks are "identity functions" - Training progressively "turns on" each layer - Improves training stability

4.4.1 Weight Initialization: `nanochat/gpt.py:188`

```
def _init_weights(self, module):
    if isinstance(module, nn.Linear):
        # Fan-in aware initialization
        fan_out = module.weight.size(0)
        fan_in = module.weight.size(1)
        std = 1.0 / math.sqrt(fan_in) * min(1.0, math.sqrt(fan_out / fan_in))
```

```
        torch.nn.init.normal_(module.weight, mean=0.0, std=std)
    elif isinstance(module, nn.Embedding):
        torch.nn.init.normal_(module.weight, mean=0.0, std=1.0)
```

Uses **fan-in aware initialization** (inspired by Muon paper): - Scale by $1/\sqrt{\text{fan\_in}}$ - Additional scaling for wide matrices - Prevents gradient explosion/vanishing

4.5 Rotary Position Embeddings (RoPE): `nanochat/gpt.py:201`

```
def _precompute_rotary_embeddings(self, seq_len, head_dim, base=10000, device=None):
    if device is None:
        device = self.transformer.wte.weight.device

    # Frequency for each dimension pair
    channel_range = torch.arange(0, head_dim, 2, dtype=torch.float32, device=device)
    inv_freq = 1.0 / (base ** (channel_range / head_dim))

    # Position indices
    t = torch.arange(seq_len, dtype=torch.float32, device=device)

    # Outer product: (seq_len, head_dim/2)
    freqs = torch.outer(t, inv_freq)

    # Precompute cos and sin
    cos, sin = freqs.cos(), freqs.sin()

    # Cast to BF16 and add batch/head dimensions
    cos, sin = cos.bfloat16(), sin.bfloat16()
    cos, sin = cos[None, :, None, :], sin[None, :, None, :]
    # Shape: [1, seq_len, 1, head_dim/2]

    return cos, sin
```

RoPE intuition: - Each dimension pair forms a 2D rotation - Rotation angle depends on position: $\theta_m = m \cdot \theta$ - Relative position $m - n$ encoded in dot product - Works better than absolute position embeddings for extrapolation

Application: See `apply_rotary_emb()` in attention section.

4.6 Forward Pass: nanochat/gpt.py:259

```python
def forward(self, idx, targets=None, kv_cache=None, loss_reduction='mean'):
    B, T = idx.size()

    # Get rotary embeddings for current sequence
    assert T <= self.cos.size(1), f"Sequence too long: {T} > {self.cos.size(1)}"
    T0 = 0 if kv_cache is None else kv_cache.get_pos()
    cos_sin = self.cos[:, T0:T0+T], self.sin[:, T0:T0+T]

    # Token embedding + normalization
    x = self.transformer.wte(idx)  # [B, T, D]
    x = norm(x)                    # RMSNorm

    # Pass through transformer blocks
    for block in self.transformer.h:
        x = block(x, cos_sin, kv_cache)

    # Final normalization
    x = norm(x)

    # Language model head
    softcap = 15
    if targets is not None:
        # Training mode: compute loss
        logits = self.lm_head(x)
        logits = softcap * torch.tanh(logits / softcap)  # Softcap
        logits = logits.float()  # Use FP32 for numerical stability
        loss = F.cross_entropy(
            logits.view(-1, logits.size(-1)),
            targets.view(-1),
            ignore_index=-1,
            reduction=loss_reduction
        )
        return loss
    else:
        # Inference mode: return logits
        logits = self.lm_head(x)
        logits = softcap * torch.tanh(logits / softcap)
        return logits
```

4.6.1 Logit Softcapping

```
logits = 15 * torch.tanh(logits / 15)
```

Why? Prevents extreme logit values: - Improves training stability - Prevents over-confidence - Used in Gemini models

Without softcapping: logits can be [-100, 200, 50, ...] With softcapping: logits bounded to roughly [-15, 15]

4.6.2 Normalization Strategy

nanochat uses **Pre-Norm** architecture:

```
x = x + Attention(Norm(x))
x = x + MLP(Norm(x))
```

Why Pre-Norm? - More stable training - Can train deeper models - Gradient flow is smoother

Alternative is **Post-Norm** (used in original Transformer):

```
x = Norm(x + Attention(x))
x = Norm(x + MLP(x))
```

4.7 Transformer Block: `nanochat/gpt.py:142`

```python
class Block(nn.Module):
    def __init__(self, config, layer_idx):
        super().__init__()
        self.attn = CausalSelfAttention(config, layer_idx)
        self.mlp = MLP(config)

    def forward(self, x, cos_sin, kv_cache):
        # Self-attention with residual connection
        x = x + self.attn(norm(x), cos_sin, kv_cache)

        # MLP with residual connection
        x = x + self.mlp(norm(x))

        return x
```

Two key components: 1. **Self-Attention**: Allows tokens to communicate 2. **MLP**: Processes each token independently

Both use **residual connections** (the `x +` part).

4.8 Multi-Layer Perceptron (MLP): `nanochat/gpt.py:129`

```python
class MLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.c_fc = nn.Linear(config.n_embd, 4 * config.n_embd, bias=False)
        self.c_proj = nn.Linear(4 * config.n_embd, config.n_embd, bias=False)

    def forward(self, x):
        x = self.c_fc(x)           # [B, T, D] -> [B, T, 4D]
        x = F.relu(x).square()     # ReLU²
        x = self.c_proj(x)         # [B, T, 4D] -> [B, T, D]
        return x
```

Architecture:

```
Input [D]
  ↓
Linear (expand 4x) [4D]
  ↓
ReLU² activation
  ↓
Linear (project back) [D]
  ↓
Output [D]
```

4.8.1 ReLU² Activation

```python
F.relu(x).square()  # max(0, x)²
```

Why ReLU² instead of GELU? - Simpler (no approximations needed) - Works well for small models - Slightly faster

Comparison: - **ReLU**: $\max(0, x)$ - **ReLU²**: $\max(0, x)^2$ - **GELU**: $x \cdot \Phi(x)$ (Gaussian CDF)

For large models, GELU often performs better. For small models, ReLU² is competitive.

4.8.2 MLP Expansion Ratio

The MLP expands to $4 \times D$ in the hidden layer: - Original Transformer used $4\times$ - Some modern models use $\frac{8}{3}\times$ or $3.5\times$ - nanochat keeps $4\times$ for simplicity

Parameter count: MLP contributes $\sim\frac{2}{3}$ of model parameters!

4.9 Model Scaling

nanochat derives model dimensions from **depth**:

```
# From scripts/base_train.py:74
depth = 20  # User sets this
num_layers = depth
model_dim = depth * 64          # Aspect ratio of 64
num_heads = max(1, (model_dim + 127) // 128)  # Head dim ~128
num_kv_heads = num_heads        # 1:1 MQA ratio
```

Example scales:

Depth	Layers	Dim	Heads	Params	Description
6	6	384	3	~8M	Tiny
12	12	768	6	~60M	Small
20	20	1280	10	~270M	Base ($100)
26	26	1664	13	~460M	GPT-2 level

Scaling law: Parameters $\approx 12 \times \text{layers} \times \text{dim}^2$

4.10 FLOPs Estimation: `nanochat/gpt.py:220`

```
def estimate_flops(self):
    """Estimate FLOPs per token (Kaplan et al. 2020)"""
    nparams = sum(p.numel() for p in self.parameters())
    nparams_embedding = self.transformer.wte.weight.numel()
    l, h, q, t = (self.config.n_layer, self.config.n_head,
                  self.config.n_embd // self.config.n_head,
                  self.config.sequence_len)

    # Forward pass FLOPs
    num_flops_per_token = 6 * (nparams - nparams_embedding) + 12 * l * h * q * t
```

```
    return num_flops_per_token
```

Formula breakdown: - $6N$: Linear layers (2 FLOPs per multiply-add, 3 layers per block) - $12lhqT$: Attention computation

Used for compute budget planning and MFU (Model FLOPs Utilization) tracking.

4.11 Memory and Efficiency

4.11.1 Mixed Precision Training

```
autocast_ctx = torch.amp.autocast(device_type="cuda", dtype=torch.bfloat16)

with autocast_ctx:
    loss = model(x, y)
```

BFloat16 (BF16) benefits: - 2× memory reduction vs FP32 - 2× speedup on modern GPUs (Ampere+) - Better numerical properties than FP16 (no loss scaling needed)

4.11.2 Model Compilation

```
model = torch.compile(model, dynamic=False)
```

PyTorch 2.0+ can compile the model to optimized kernels: - Fuses operations - Reduces memory overhead - ~20-30% speedup

4.12 Parameter Count Breakdown

For a d=20 model (~270M params):

Component	Params	Fraction
Token embeddings	32K × 1280 = 41M	15%
LM head	32K × 1280 = 41M	15%
Attention	~56M	21%
MLP	~132M	49%
Total	**~270M**	**100%**

Key insight: Most parameters are in MLPs and embeddings!

4.13 Inference Generation: nanochat/gpt.py:294

```python
@torch.inference_mode()
def generate(self, tokens, max_tokens, temperature=1.0, top_k=None, seed=42):
    """Autoregressive generation"""
    device = self.get_device()
    rng = torch.Generator(device=device).manual_seed(seed) if temperature > 0 else None

    ids = torch.tensor([tokens], dtype=torch.long, device=device)  # [1, T]

    for _ in range(max_tokens):
        # Forward pass
        logits = self.forward(ids)  # [1, T, V]
        logits = logits[:, -1, :]   # Take last token [1, V]

        # Top-k filtering
        if top_k is not None:
            v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
            logits[logits < v[:, [-1]]] = -float('Inf')

        # Sample
        if temperature > 0:
            probs = F.softmax(logits / temperature, dim=-1)
            next_ids = torch.multinomial(probs, num_samples=1, generator=rng)
        else:
            next_ids = torch.argmax(logits, dim=-1, keepdim=True)

        # Append to sequence
        ids = torch.cat((ids, next_ids), dim=1)
        token = next_ids.item()
        yield token
```

Generation strategies: - **Greedy** (temperature=0): Always pick highest probability - **Sampling** (temperature=1): Sample from distribution - **Top-k sampling**: Only sample from top k tokens

4.14 Comparison: GPT-2 vs nanochat GPT

Feature	GPT-2	nanochat GPT
Position encoding	Learned absolute	Rotary (RoPE)

Feature	GPT-2	nanochat GPT
Normalization	LayerNorm	RMSNorm (no params)
Activation	GELU	ReLU^2
Embedding	Tied	Untied
Attention	Standard	Multi-Query + QK Norm
Logits	Raw	Softcapped
Bias in linear	Yes	No

Result: nanochat GPT is simpler, faster, and performs better at small scale!

4.15 Next Steps

Now we'll dive deep into the **Attention Mechanism** - the core innovation that makes Transformers work.

Chapter 5

The Attention Mechanism

Attention is the core innovation that makes Transformers powerful. It allows each token to "look at" and aggregate information from other tokens in the sequence.

5.1 Intuition: What is Attention?

Think of attention like a **database query**: - **Queries (Q)**: "What am I looking for?" - **Keys (K)**: "What do I contain?" - **Values (V)**: "What information do I have?"

Each token computes **how much it should attend** to every other token, then aggregates their values.

5.1.1 Example

Sentence: "The cat sat on the mat"

When processing "sat": - High attention to "cat" (who is sitting?) - High attention to "mat" (where sitting?) - Low attention to "The" (less relevant)

Result: "sat" has context-aware representation incorporating info from "cat" and "mat".

5.2 Scaled Dot-Product Attention

Mathematical formula:

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

Let's break this down step by step.

5.2.1 Step 1: Compute Attention Scores

$$S = QK^T \in \mathbb{R}^{T \times T}$$

- $Q \in \mathbb{R}^{T \times d_k}$: Query matrix (one row per token)
- $K \in \mathbb{R}^{T \times d_k}$: Key matrix
- S_{ij}: similarity between query i and key j

Dot product measures similarity: - High dot product → queries and keys are aligned → high attention - Low dot product → different directions → low attention

5.2.2 Step 2: Scale

$$S' = \frac{S}{\sqrt{d_k}}$$

Why divide by $\sqrt{d_k}$?

For random vectors with dimension d_k: - Dot product has mean 0 - Variance grows as d_k - Scaling keeps variance stable at 1

Without scaling, large d_k causes: - Very large/small scores - Softmax saturates (gradients vanish)

5.2.3 Step 3: Softmax (Normalize to Probabilities)

$$A = \text{softmax}(S') \in \mathbb{R}^{T \times T}$$

Each row becomes a probability distribution:

$$A_{ij} = \frac{\exp(S'_{ij})}{\sum_{k=1}^{T} \exp(S'_{ik})}$$

Properties: - $A_{ij} \geq 0$ - $\sum_j A_{ij} = 1$ (each query's attention sums to 1)

A_{ij} = how much query i attends to key j

5.2.4 Step 4: Weighted Sum of Values

$$\text{Output} = AV \in \mathbb{R}^{T \times d_v}$$

For each token i:

$$\text{output}_i = \sum_{j=1}^{T} A_{ij} V_j$$

Aggregate values from all tokens, weighted by attention scores.

5.3 Causal Self-Attention

In language modeling, we can't look at future tokens! We need **causal masking**.

5.3.1 Masking

Before softmax, add a mask:

$$\text{mask}_{ij} = \begin{cases} 0 & \text{if } i \geq j \text{ (can attend)} \\ -\infty & \text{if } i < j \text{ (future, block)} \end{cases}$$

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + \text{mask}\right)$$

After softmax, $-\infty$ becomes 0, so future tokens contribute nothing.

5.3.2 Visualization

```
Attention matrix (T=5):
     k0  k1  k2  k3  k4
q0  [        ]    ← q0 can only see k0
q1  [        ]    ← q1 can see k0, k1
q2  [        ]
q3  [        ]
q4  [        ]    ← q4 can see all
```

This creates a **lower triangular** attention pattern.

5.4 Implementation: `nanochat/gpt.py:64`

```python
class CausalSelfAttention(nn.Module):
    def __init__(self, config, layer_idx):
        super().__init__()
        self.layer_idx = layer_idx
        self.n_head = config.n_head
        self.n_kv_head = config.n_kv_head
        self.n_embd = config.n_embd
        self.head_dim = self.n_embd // self.n_head

        # Projection matrices
        self.c_q = nn.Linear(self.n_embd, self.n_head * self.head_dim, bias=False)
```

```
        self.c_k = nn.Linear(self.n_embd, self.n_kv_head * self.head_dim, bias=False)
        self.c_v = nn.Linear(self.n_embd, self.n_kv_head * self.head_dim, bias=False)
        self.c_proj = nn.Linear(self.n_embd, self.n_embd, bias=False)
```

Key design choices: 1. **No bias**: Modern practice removes bias from linear layers 2. **Separate K/V heads**: Allows Multi-Query Attention (MQA) 3. **Output projection**: Mix information from all heads

5.4.1 Forward Pass: `nanochat/gpt.py:79`

```
def forward(self, x, cos_sin, kv_cache):
    B, T, C = x.size()  # [batch, sequence, channels]

    # 1. Project to queries, keys, values
    q = self.c_q(x).view(B, T, self.n_head, self.head_dim)
    k = self.c_k(x).view(B, T, self.n_kv_head, self.head_dim)
    v = self.c_v(x).view(B, T, self.n_kv_head, self.head_dim)

    # 2. Apply Rotary Position Embeddings
    cos, sin = cos_sin
    q, k = apply_rotary_emb(q, cos, sin), apply_rotary_emb(k, cos, sin)

    # 3. QK Normalization (stability)
    q, k = norm(q), norm(k)

    # 4. Rearrange to [B, num_heads, T, head_dim]
    q, k, v = q.transpose(1, 2), k.transpose(1, 2), v.transpose(1, 2)

    # 5. Handle KV cache (for inference)
    if kv_cache is not None:
        k, v = kv_cache.insert_kv(self.layer_idx, k, v)

    Tq = q.size(2)  # Number of queries
    Tk = k.size(2)  # Number of keys

    # 6. Multi-Query Attention: replicate K/V heads
    nrep = self.n_head // self.n_kv_head
    k, v = repeat_kv(k, nrep), repeat_kv(v, nrep)

    # 7. Compute attention
```

```
    if kv_cache is None or Tq == Tk:
        # Training: simple causal attention
        y = F.scaled_dot_product_attention(q, k, v, is_causal=True)
    elif Tq == 1:
        # Inference with single token: attend to all cached tokens
        y = F.scaled_dot_product_attention(q, k, v, is_causal=False)
    else:
        # Inference with multiple tokens: custom masking
        attn_mask = torch.zeros((Tq, Tk), dtype=torch.bool, device=q.device)
        prefix_len = Tk - Tq
        if prefix_len > 0:
            attn_mask[:, :prefix_len] = True  # Can attend to prefix
        # Causal within new tokens
        attn_mask[:, prefix_len:] = torch.tril(torch.ones((Tq, Tq), dtype=torch.bool, device=q
        y = F.scaled_dot_product_attention(q, k, v, attn_mask=attn_mask)

    # 8. Concatenate heads and project
    y = y.transpose(1, 2).contiguous().view(B, T, -1)
    y = self.c_proj(y)
    return y
```

Let's examine each component in detail.

5.5 1. Projections to Q, K, V

```
q = self.c_q(x).view(B, T, self.n_head, self.head_dim)
k = self.c_k(x).view(B, T, self.n_kv_head, self.head_dim)
v = self.c_v(x).view(B, T, self.n_kv_head, self.head_dim)
```

What's happening: - Linear projection: $Q = XW^Q$, $K = XW^K$, $V = XW^V$ - Reshape to separate heads - Each head operates on $d_{head} = d_{model}/n_{heads}$ dimensions

Example: $d_{model} = 768$, $n_{heads} = 6$ - Input: $[B, T, 768]$ - After projection: $[B, T, 6 \times 128]$ - After view: $[B, T, 6, 128]$

5.6 2. Rotary Position Embeddings (RoPE)

Implementation: `nanochat/gpt.py:41`

```
def apply_rotary_emb(x, cos, sin):
    assert x.ndim == 4  # [B, T, H, D] or [B, H, T, D]
```

```
    d = x.shape[3] // 2

    # Split into pairs
    x1, x2 = x[..., :d], x[..., d:]

    # Rotation in 2D
    y1 = x1 * cos + x2 * sin        # Rotate first element
    y2 = x1 * (-sin) + x2 * cos     # Rotate second element

    # Concatenate back
    out = torch.cat([y1, y2], 3)
    out = out.to(x.dtype)
    return out
```

Mathematical formula:

For a pair of dimensions (x_1, x_2) at position m:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos(m\theta) & \sin(m\theta) \\ -\sin(m\theta) & \cos(m\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Why RoPE is powerful:

The dot product $Q_m \cdot K_n$ after RoPE only depends on relative position $m - n$:

$$Q_m \cdot K_n = \tilde{Q} \cdot \tilde{K} \cdot e^{i(m-n)\theta}$$

This gives the model a **strong inductive bias** for relative positions.

Benefits over learned positions: - Works for sequence lengths longer than seen during training - More parameter efficient (no learned position embeddings) - Better performance on downstream tasks

5.7 3. QK Normalization

```
q, k = norm(q), norm(k)
```

Why normalize Q and K?

Without normalization, the scale of Q and K can grow during training: - Large Q/K $\rightarrow$ large attention scores - Softmax saturates - Gradients vanish

Normalization (RMSNorm) keeps scales stable:

$$\text{norm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_i x_i^2}}$$

This is a modern improvement not in original Transformers.

5.8 4. Multi-Query Attention (MQA)

```
nrep = self.n_head // self.n_kv_head
k, v = repeat_kv(k, nrep), repeat_kv(v, nrep)
```

Idea: Use fewer K/V heads than Q heads.

Standard Multi-Head Attention: - 6 query heads - 6 key heads - 6 value heads

Multi-Query Attention: - 6 query heads - 1 key head (replicated 6 times) - 1 value head (replicated 6 times)

Benefits: - Fewer parameters - **Much faster inference** (less KV cache memory) - Minimal quality loss

Implementation: `nanochat/gpt.py:52`

```
def repeat_kv(x, n_rep):
    """Repeat K/V heads to match number of Q heads"""
    if n_rep == 1:
        return x
    bs, n_kv_heads, slen, head_dim = x.shape
    return (
        x[:, :, None, :, :]
        .expand(bs, n_kv_heads, n_rep, slen, head_dim)
        .reshape(bs, n_kv_heads * n_rep, slen, head_dim)
    )
```

In nanochat, we use **1:1 MQA** (same number of Q and KV heads) for simplicity. Real MQA would use fewer KV heads.

5.9 5. Flash Attention

```
y = F.scaled_dot_product_attention(q, k, v, is_causal=True)
```

PyTorch's `scaled_dot_product_attention` automatically uses **Flash Attention** when available.

Standard attention: 1. Compute $S = QK^T$ (materialize $T \times T$ matrix) 2. Apply softmax 3. Compute SV

Memory: $O(T^2)$ for storing attention matrix

Flash Attention: - Fuses operations - Tiles computation to fit in SRAM - Never materializes full attention matrix

Benefits: - $O(T)$ memory instead of $O(T^2)$ - 2-4× faster - Enables longer context lengths

5.10 6. KV Cache (for Inference)

During inference, we generate tokens one at a time. **KV cache** avoids recomputing past tokens.

Without cache: For each new token, recompute K and V for all previous tokens - Token 1: compute K,V for 1 token - Token 2: compute K,V for 2 tokens - Token 3: compute K,V for 3 tokens - Total: $1 + 2 + 3 + ... + T = O(T^2)$ operations

With cache: Store K and V from previous tokens - Token 1: compute K,V for 1 token, store - Token 2: compute K,V for 1 NEW token, concatenate with cache - Token 3: compute K,V for 1 NEW token, concatenate with cache - Total: $O(T)$ operations

Speedup: T times faster!

```
if kv_cache is not None:
    k, v = kv_cache.insert_kv(self.layer_idx, k, v)
```

The cache stores K and V for all previous tokens and layers.

5.11 Multi-Head Attention Intuition

Why multiple heads?

Different heads can learn different attention patterns: - **Head 1**: Attend to previous word - **Head 2**: Attend to subject of sentence - **Head 3**: Attend to syntactically related words - **Head 4**: Attend to semantically similar words

Each head operates independently, then outputs are concatenated and projected:

```
y = y.transpose(1, 2).contiguous().view(B, T, -1)  # Concatenate heads
y = self.c_proj(y)  # Final projection
```

5.12 Computational Complexity

For sequence length T and dimension d:

Operation	Complexity
Q, K, V projections	$O(T \cdot d^2)$
QK^T	$O(T^2 \cdot d)$
Softmax	$O(T^2)$
Attention × V	$O(T^2 \cdot d)$
Output projection	$O(T \cdot d^2)$
Total	$O(T \cdot d^2 + T^2 \cdot d)$

For small sequences: $T < d$, so $O(T \cdot d^2)$ dominates For long sequences: $T > d$, so $O(T^2 \cdot d)$ dominates

Bottleneck: Quadratic in sequence length!

This is why context length is expensive.

5.13 Attention Patterns Visualization

Let's visualize what attention learns. Here's a simplified example:

Sentence: "The quick brown fox jumps"

```
Attention pattern for "jumps":

       The  quick  brown  fox  jumps
The    0.05  0.05   0.05   0.05  0.0   (can't attend to self)
quick  0.1   0.1    0.1    0.1   0.0
brown  0.05  0.05   0.15   0.15  0.0
fox    0.15  0.05   0.15   0.4   0.0   ← "fox" has high attention
jumps  0.1   0.1    0.1    0.5   0.2   ← we're here
```

"jumps" attends strongly to "fox" (the actor) - this is learned!

5.14 Comparison: Different Attention Variants

Variant	#Q Heads	#KV Heads	Memory	Speed
Multi-Head (MHA)	H	H	High	Baseline
Multi-Query (MQA)	H	1	Low	Fast
Grouped-Query (GQA)	H	H/G	Medium	Fast

nanochat uses MHA with equal Q/KV heads, but the code supports MQA.

5.15 Common Attention Issues and Solutions

5.15.1 Problem 1: Attention Collapse

Symptom: All tokens attend uniformly to all positions **Solution:** - QK normalization - Proper initialization - Attention dropout (not used in nanochat)

5.15.2 Problem 2: Over-attention to Certain Positions

Symptom: Strong attention to first/last token regardless of content **Solution:** - Better position embeddings (RoPE helps) - Softcapping logits

5.15.3 Problem 3: Softmax Saturation

Symptom: Gradients vanish, training stalls **Solution:** - Scale by $\sqrt{d_k}$ - QK normalization - Lower learning rate

5.16 Exercises to Understand Attention

1. **Implement attention from scratch:**

```
def simple_attention(Q, K, V, mask=None):
    d_k = Q.size(-1)
    scores = Q @ K.transpose(-2, -1) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    attn = F.softmax(scores, dim=-1)
    return attn @ V
```

2. **Visualize attention patterns** on a real sentence
3. **Compare with and without scaling** by $\sqrt{d_k}$

5.17 Next Steps

Now that we understand attention, we'll explore the **Training Process** - how we actually train these models on massive datasets.

Chapter 6

The Training Process

Training a language model involves teaching it to predict the next token given previous tokens. Let's understand how nanochat implements this end-to-end.

6.1 Overview: The Complete Training Pipeline

```
1. Tokenization Training (~10 min)
    > BPE tokenizer vocabulary

2. Base Pretraining (~2-4 hours, $100)
    > Base model checkpoint

3. Midtraining (~30 min, $12)
    > Refined base model

4. Supervised Fine-Tuning (~15 min, $6)
    > Chat model

5. Reinforcement Learning (~10 min, $4)
    > Final optimized model
```

Total cost: ~$122, **Total time:** ~3-5 hours on 8×H100 GPUs

6.2 1. Language Modeling Objective

Goal: Learn probability distribution over sequences

$$P(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} P(w_t | w_1, \ldots, w_{t-1})$$

Training objective: Maximize log-likelihood

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \log P(w_t | w_1, \ldots, w_{t-1})$$

In practice, we minimize **negative log-likelihood** (cross-entropy loss):

$$\mathcal{L} = -\frac{1}{T} \sum_{t=1}^{T} \log P(w_t | w_1, \ldots, w_{t-1})$$

6.2.1 Cross-Entropy Loss in Code

File: `nanochat/gpt.py:285`

```
def forward(self, idx, targets=None, ...):
    # ... forward pass to get hidden states x ...

    if targets is not None:
        # Training mode
        logits = self.lm_head(x)  # [B, T, vocab_size]
        logits = 15 * torch.tanh(logits / 15)  # Softcap

        # Cross-entropy loss
        loss = F.cross_entropy(
            logits.view(-1, logits.size(-1)),  # [B*T, V]
            targets.view(-1),                   # [B*T]
            ignore_index=-1,
            reduction='mean'
        )
        return loss
```

Key points: 1. Reshape to 2D for loss computation 2. `ignore_index=-1`: Skip padding tokens 3. `reduction='mean'`: Average over all tokens

6.3 2. Data Loading: `nanochat/dataloader.py`

Efficient data loading is crucial for fast training.

```python
def tokenizing_distributed_data_loader(B, T, split, tokenizer_threads=4, tokenizer_batch_size=
    """Stream pretraining text from parquet files, tokenize, yield batches."""
    ddp, ddp_rank, ddp_local_rank, ddp_world_size = get_dist_info()
    needed_tokens = B * T + 1  # +1 for target

    # Initialize tokenizer and buffer
    tokenizer = get_tokenizer()
    bos_token = tokenizer.get_bos_token_id()
    token_buffer = deque()  # Streaming token buffer

    # Infinite iterator over documents
    def document_batches():
        while True:
            for batch in parquets_iter_batched(split=split, start=ddp_rank, step=ddp_world_size
                for i in range(0, len(batch), tokenizer_batch_size):
                    yield batch[i:i+tokenizer_batch_size]

    batches = document_batches()

    while True:
        # Fill buffer with enough tokens
        while len(token_buffer) < needed_tokens:
            doc_batch = next(batches)
            # Tokenize in parallel
            token_lists = tokenizer.encode(doc_batch, prepend=bos_token, num_threads=tokenizer_
            for tokens in token_lists:
                token_buffer.extend(tokens)

        # Extract tokens from buffer
        scratch = torch.empty(needed_tokens, dtype=torch.int64, pin_memory=True)
        for i in range(needed_tokens):
            scratch[i] = token_buffer.popleft()

        # Create inputs and targets
        inputs_cpu = scratch[:-1].to(dtype=torch.int32)   # [0, 1, 2, ..., T-1]
        targets_cpu = scratch[1:]                          # [1, 2, 3, ..., T]

        # Move to GPU
        inputs = inputs_cpu.view(B, T).to(device="cuda", non_blocking=True)
```

```
        targets = targets_cpu.view(B, T).to(device="cuda", non_blocking=True)

        yield inputs, targets
```

Design highlights:

1. **Streaming:** Never loads entire dataset into memory
2. **Distributed:** Each GPU processes different shards (`start=ddp_rank, step=ddp_world_size`)
3. **Parallel tokenization:** Uses multiple threads
4. **Pinned memory:** Faster CPU→GPU transfer
5. **Non-blocking transfers:** Overlap with computation

6.3.1 Input/Target Relationship

For sequence `[0, 1, 2, 3, 4, 5]`:

```
Inputs:  [0, 1, 2, 3, 4]
Targets: [1, 2, 3, 4, 5]

Position 0: input=0, target=1  →  predict 1 given 0
Position 1: input=1, target=2  →  predict 2 given 0,1
Position 2: input=2, target=3  →  predict 3 given 0,1,2
...
```

Each position predicts the next token!

6.4 3. Training Loop: `scripts/base_train.py`

6.4.1 Hyperparameters: `scripts/base_train.py:28`

```
# Model architecture
depth = 20              # Number of layers
max_seq_len = 2048      # Context length

# Training horizon
target_param_data_ratio = 20  # Chinchilla optimal

# Optimization
device_batch_size = 32         # Per-GPU batch size
total_batch_size = 524288      # Total tokens per step
embedding_lr = 0.2             # AdamW for embeddings
unembedding_lr = 0.004         # AdamW for LM head
```

```
matrix_lr = 0.02                # Muon for linear layers
grad_clip = 1.0                 # Gradient clipping

# Evaluation
eval_every = 250
core_metric_every = 2000
```

6.4.2 Computing Training Length: scripts/base_train.py:108

```
# Chinchilla scaling: 20 tokens per parameter
target_tokens = target_param_data_ratio * num_params
num_iterations = target_tokens // total_batch_size

print(f"Parameters: {num_params:,}")
print(f"Target tokens: {target_tokens:,}")
print(f"Iterations: {num_iterations:,}")
print(f"Total FLOPs: {num_flops_per_token * total_tokens:e}")
```

Example for d=20 model: - Parameters: 270M - Target tokens: 20 × 270M = 5.4B - Batch size: 524K - Iterations: 5.4B / 524K 10,300 steps

6.4.3 Gradient Accumulation: scripts/base_train.py:89

```
tokens_per_fwdbwd = device_batch_size * max_seq_len  # Per-GPU
world_tokens_per_fwdbwd = tokens_per_fwdbwd * ddp_world_size  # All GPUs
grad_accum_steps = total_batch_size // world_tokens_per_fwdbwd

print(f"Tokens / micro-batch / rank: {tokens_per_fwdbwd:,}")
print(f"Total batch size {total_batch_size:,}")
print(f"Gradient accumulation steps: {grad_accum_steps}")
```

Example: - Device batch: 32 × 2048 = 65,536 tokens - 8 GPUs: 8 × 65,536 = 524,288 tokens - Grad accum: 524,288 / 524,288 = 1 (no accumulation needed)

But if we only had 4 GPUs: - 4 GPUs: 4 × 65,536 = 262,144 tokens - Grad accum: 524,288 / 262,144 = 2 steps

Gradient accumulation allows larger effective batch sizes than GPU memory permits.

6.4.4 Main Training Loop: `scripts/base_train.py:172`

```
for step in range(num_iterations + 1):
    last_step = step == num_iterations

    # ===== EVALUATION =====
    if last_step or step % eval_every == 0:
        model.eval()
        val_loader = build_val_loader()
        with autocast_ctx:
            val_bpb = evaluate_bpb(model, val_loader, eval_steps, token_bytes)
        wandb_run.log({"val/bpb": val_bpb})
        model.train()

    # ===== SAMPLING =====
    if master_process and (last_step or step % sample_every == 0):
        model.eval()
        prompts = ["The capital of France is", ...]
        engine = Engine(model, tokenizer)
        for prompt in prompts:
            tokens = tokenizer(prompt, prepend="<|bos|>")
            sample, _ = engine.generate_batch(tokens, max_tokens=16, temperature=0)
            print(tokenizer.decode(sample[0]))
        model.train()

    # ===== CHECKPOINT =====
    if master_process and last_step:
        save_checkpoint(checkpoint_dir, step, model.state_dict(), ...)

    if last_step:
        break

    # ===== TRAINING STEP =====
    torch.cuda.synchronize()
    t0 = time.time()

    # Gradient accumulation loop
    for micro_step in range(grad_accum_steps):
        with autocast_ctx:
            loss = model(x, y)
```

```
        train_loss = loss.detach()
        loss = loss / grad_accum_steps  # Normalize for accumulation
        loss.backward()
        x, y = next(train_loader)  # Prefetch next batch

    # Gradient clipping
    if grad_clip > 0.0:
        torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)

    # Update learning rates
    lrm = get_lr_multiplier(step)
    for opt in optimizers:
        for group in opt.param_groups:
            group["lr"] = group["initial_lr"] * lrm

    # Update momentum for Muon
    muon_momentum = get_muon_momentum(step)
    for group in muon_optimizer.param_groups:
        group["momentum"] = muon_momentum

    # Optimizer step
    for opt in optimizers:
        opt.step()
    model.zero_grad(set_to_none=True)

    torch.cuda.synchronize()
    t1 = time.time()

    # Logging
    print(f"step {step:05d} | loss: {loss:.6f} | dt: {(t1-t0)*1000:.2f}ms | ...")
```

6.4.5 Mixed Precision Training

```
autocast_ctx = torch.amp.autocast(device_type="cuda", dtype=torch.bfloat16)

with autocast_ctx:
    loss = model(x, y)
```

BFloat16 (BF16) automatic mixed precision: - Forward pass in BF16 (2× faster, 2× less memory) - Backward pass in FP32 (for numerical stability) - Automatic casting handled by PyTorch

Why BF16 over FP16? - Same exponent range as FP32 (no loss scaling needed) - Better numerical stability - Supported on Ampere+ GPUs

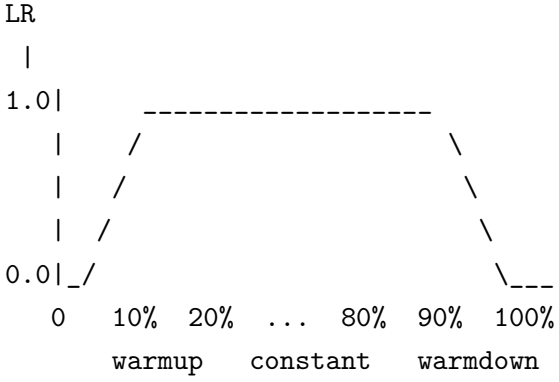
6.4.6 Learning Rate Schedule: `scripts/base_train.py:148`

```
warmup_ratio = 0.0      # No warmup
warmdown_ratio = 0.2    # 20% of steps for decay
final_lr_frac = 0.0     # Decay to 0

def get_lr_multiplier(it):
    warmup_iters = round(warmup_ratio * num_iterations)
    warmdown_iters = round(warmdown_ratio * num_iterations)

    if it < warmup_iters:
        # Linear warmup
        return (it + 1) / warmup_iters
    elif it <= num_iterations - warmdown_iters:
        # Constant
        return 1.0
    else:
        # Linear decay
        progress = (num_iterations - it) / warmdown_iters
        return progress * 1.0 + (1 - progress) * final_lr_frac
```

Schedule visualization:



6.4.7 Gradient Clipping: `scripts/base_train.py:265`

```
if grad_clip > 0.0:
    torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)
```

Why clip gradients? - Prevents exploding gradients - Stabilizes training - Allows higher learning

rates

How it works:

$$\mathbf{g} \leftarrow \begin{cases} \mathbf{g} & \text{if } \|\mathbf{g}\| \leq \text{max\_norm} \\ \frac{\text{max\_norm}}{\|\mathbf{g}\|}\mathbf{g} & \text{otherwise} \end{cases}$$

Scales gradient to have maximum norm of `grad_clip`.

6.5 4. Distributed Training (DDP)

nanochat uses **DistributedDataParallel (DDP)** for multi-GPU training.

6.5.1 Initialization: `nanochat/common.py`

```python
def compute_init():
    ddp = int(os.environ.get("RANK", -1)) != -1  # Is this DDP?

    if ddp:
        torch.distributed.init_process_group(backend="nccl")
        ddp_rank = int(os.environ["RANK"])
        ddp_local_rank = int(os.environ["LOCAL_RANK"])
        ddp_world_size = int(os.environ["WORLD_SIZE"])
        device = f"cuda:{ddp_local_rank}"
        torch.cuda.set_device(device)
    else:
        ddp_rank = 0
        ddp_local_rank = 0
        ddp_world_size = 1
        device = "cuda"

    return ddp, ddp_rank, ddp_local_rank, ddp_world_size, device
```

6.5.2 Running DDP

```bash
# Single GPU
python -m scripts.base_train

# Multi-GPU (8 GPUs)
torchrun --standalone --nproc_per_node=8 -m scripts.base_train
```

How DDP works:

1. **Data parallelism:** Each GPU gets different data
2. **Model replication:** Same model on all GPUs
3. **Gradient averaging:** After backward, gradients are averaged across GPUs
4. **Synchronized updates:** All GPUs update identically

Benefits: - Near-linear scaling (8 GPUs 8× faster) - Same final model as single-GPU training - Minimal code changes

6.6 5. Evaluation During Training

6.6.1 Validation Loss (BPB): `nanochat/loss_eval.py`

```python
def evaluate_bpb(model, val_loader, eval_steps, token_bytes):
    """Evaluate bits-per-byte on validation set"""
    total_loss = 0
    total_tokens = 0

    for step in range(eval_steps):
        x, y = next(val_loader)
        with torch.no_grad():
            loss = model(x, y, loss_reduction='sum')
        total_loss += loss.item()
        total_tokens += (y != -1).sum().item()

    # Average loss per token
    avg_loss_per_token = total_loss / total_tokens

    # Convert to bits per byte
    bits_per_token = avg_loss_per_token / math.log(2)
    token_bytes_mean = token_bytes.float().mean().item()
    bits_per_byte = bits_per_token / token_bytes_mean

    return bits_per_byte
```

Bits-per-byte (BPB) measures compression: - Lower BPB = better model - Random model: ~8 BPB (1 byte = 8 bits, no compression) - Good model: ~1.0-1.5 BPB

6.6.2 CORE Metric: `scripts/base_eval.py`

CORE is a weighted average of multiple benchmarks:

```python
def evaluate_model(model, tokenizer, device, max_per_task=500):
    results = {}

    # Run each task
    for task_name, task_fn in tasks.items():
        acc = task_fn(model, tokenizer, device, max_per_task)
        results[task_name] = acc

    # Compute weighted average
    weights = {"task1": 0.3, "task2": 0.7, ...}
    core_metric = sum(weights[k] * results[k] for k in weights)

    return {"core_metric": core_metric, "results": results}
```

Evaluated periodically during training to track progress.

6.7 6. Checkpointing: nanochat/checkpoint_manager.py

```python
def save_checkpoint(checkpoint_dir, step, model_state, optimizer_states, metadata):
    os.makedirs(checkpoint_dir, exist_ok=True)

    # Save model
    model_path = os.path.join(checkpoint_dir, f"model_step_{step}.pt")
    torch.save(model_state, model_path)

    # Save optimizers
    for i, opt_state in enumerate(optimizer_states):
        opt_path = os.path.join(checkpoint_dir, f"optimizer_{i}_step_{step}.pt")
        torch.save(opt_state, opt_path)

    # Save metadata
    meta_path = os.path.join(checkpoint_dir, f"metadata_step_{step}.json")
    with open(meta_path, "w") as f:
        json.dump(metadata, f)

    print(f"Saved checkpoint to {checkpoint_dir}")
```

What to save: - Model weights - Optimizer states (for resuming training) - Metadata (step number, config, metrics)

6.8 7. Supervised Fine-Tuning: `scripts/chat_sft.py`

After pretraining, we fine-tune on conversations.

6.8.1 Data Format

```
conversation = {
    "messages": [
        {"role": "user", "content": "What is the capital of France?"},
        {"role": "assistant", "content": "The capital of France is Paris."},
        {"role": "user", "content": "What about Italy?"},
        {"role": "assistant", "content": "The capital of Italy is Rome."}
    ]
}
```

6.8.2 Tokenization with Mask

```
ids, mask = tokenizer.render_conversation(conversation)

# ids:  [<|bos|>, <|user_start|>, "What", "is", ..., <|assistant_end|>]
# mask: [0, 0, 0, 0, ..., 1, 1, 1, ..., 1]
#       ↑ don't train      ↑ train on assistant responses
```

6.8.3 Loss Computation

```
# Only compute loss on assistant tokens
loss = F.cross_entropy(
    logits.view(-1, vocab_size),
    targets.view(-1),
    reduction='none'
)
# Apply mask
masked_loss = (loss * mask).sum() / mask.sum()
```

Key difference from pretraining: - Pretraining: train on ALL tokens - SFT: train ONLY on assistant responses

6.9 8. Reinforcement Learning: `scripts/chat_rl.py`

Final stage: optimize for quality using RL.

6.9.1 Self-Improvement Loop

```
# 1. Generate multiple responses
prompts = load_prompts()
for prompt in prompts:
    responses = model.generate(prompt, num_samples=8, temperature=0.8)

    # 2. Score responses (using a reward model or heuristic)
    scores = [reward_model(r) for r in responses]

    # 3. Keep best responses
    best_idx = max(range(len(scores)), key=lambda i: scores[i])
    best_response = responses[best_idx]

    # 4. Fine-tune on best response
    train_on(prompt, best_response)
```

Simple but effective!

6.10 Performance Metrics

6.10.1 Model FLOPs Utilization (MFU)

```
flops_per_sec = num_flops_per_token * total_batch_size / dt
promised_flops_per_sec_h100 = 989e12 * ddp_world_size  # BF16 on H100
mfu = 100 * flops_per_sec / promised_flops_per_sec_h100
```

Good MFU: 40-60% (nanochat achieves ~50%)

6.10.2 Tokens per Second

```
tok_per_sec = world_tokens_per_fwdbwd / dt
```

Typical: 500K - 1M tokens/sec on 8×H100

6.11 Common Training Issues

6.11.1 1. Loss Spikes

Symptoms: Loss suddenly jumps **Causes:** Bad batch, numerical instability, LR too high **Solutions:** - Gradient clipping - Lower learning rate - Skip bad batches

6.11.2 2. Loss Plateau

Symptoms: Loss stops improving **Causes:** Learning rate too low, insufficient data, model capacity **Solutions:** - Increase LR - More data - Larger model

6.11.3 3. NaN Loss

Symptoms: Loss becomes NaN **Causes:** Numerical overflow, bad initialization **Solutions:** - Lower learning rate - Gradient clipping - Check for bad data

6.12 Next Steps

Now we'll explore the **Optimization Techniques** - Muon and AdamW optimizers that make training efficient.

Chapter 7

Advanced Optimization Techniques

nanochat uses a **hybrid optimization** strategy: combining **Muon** for matrix parameters and **AdamW** for embeddings. This is more sophisticated than standard approaches.

7.1 Why Different Optimizers?

Different parameter types have different optimization needs:

Parameter Type	Examples	Characteristics	Best Optimizer
Matrices	Attention, MLP	Dense, high-dimensional	Muon
Embeddings	Token embeddings	Sparse updates, embedding-specific	AdamW
Vectors	LM head	Output layer, sparse	AdamW

Traditional approach: Use AdamW for everything **nanochat approach:** Use Muon for matrices, AdamW for embeddings/head

Result: Faster training, better convergence

7.2 1. Muon Optimizer

Muon is a novel optimizer designed specifically for **matrix parameters** in neural networks.

7.2.1 Core Idea

Standard optimizers (SGD, Adam) treat matrices as flat vectors:

```
Matrix [3×4] → Flatten to vector [12] → Update
```

Muon exploits **matrix structure**:

```
Matrix [3×4] → Update using matrix operations → Keep matrix shape
```

7.2.2 Mathematical Formulation

For weight matrix $W \in \mathbb{R}^{m \times n}$:

Standard momentum:

$$v_t = \beta v_{t-1} + (1 - \beta) g_t$$

$$W_t = W_{t-1} - \eta v_t$$

Muon: 1. Compute gradient $G_t = \nabla_W \mathcal{L}$ 2. Orthogonalize using Newton-Schulz iteration 3. Apply momentum in tangent space 4. Update with adaptive step size

7.2.3 Implementation: `nanochat/muon.py:53`

```python
class Muon(torch.optim.Optimizer):
    def __init__(self, params, lr=0.02, momentum=0.95):
        defaults = dict(lr=lr, momentum=momentum)
        super(Muon, self).__init__(params, defaults)

    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            lr = group['lr']
            momentum = group['momentum']

            for p in group['params']:
                if p.grad is None:
                    continue

                g = p.grad  # Gradient

                # Get state
                state = self.state[p]
                if 'momentum_buffer' not in state:
                    state['momentum_buffer'] = torch.zeros_like(g)

                buf = state['momentum_buffer']
```

```
                # Handle matrix vs non-matrix parameters
                if g.ndim == 2 and g.size(0) >= 16 and g.size(1) >= 16:
                    # Matrix parameter: use Muon update
                    g = newton_schulz_orthogonalize(g, steps=5)

                # Momentum update
                buf.mul_(momentum).add_(g)

                # Parameter update
                p.data.add_(buf, alpha=-lr)
```

7.2.4 Newton-Schulz Orthogonalization: nanochat/muon.py:16

```
def newton_schulz_orthogonalize(G, steps=5, eps=1e-7):
    """
    Orthogonalize gradient matrix using Newton-Schulz iteration
    """
    # Make square by padding or cropping
    a, b = G.size()
    if a > b:
        G = G[:b, :]
    elif a < b:
        G = G[:, :a]

    # Initialize
    # Normalization factor
    t = G.size(0)

    # X_0 = G / ||G||_F
    A = G / (G.norm() + eps)

    # Newton-Schulz iteration: X_{k+1} = X_k * (3I - X_k^T X_k) / 2
    for _ in range(steps):
        A_T_A = A.t() @ A
        A = A @ (1.5 * torch.eye(t, device=A.device, dtype=A.dtype) - 0.5 * A_T_A)

    # Restore original shape
    if a > b:
        A = torch.cat([A, torch.zeros(a - b, b, device=A.device, dtype=A.dtype)], dim=0)
```

```
    elif a < b:
        A = torch.cat([A, torch.zeros(a, b - a, device=A.device, dtype=A.dtype)], dim=1)

    return A
```

What does this do?

For a matrix G, find orthogonal matrix Q closest to G:

$$Q = \arg\min_{\tilde{Q}^T\tilde{Q}=I} \|G - \tilde{Q}\|_F$$

Uses iterative formula:

$$X_{k+1} = X_k \left(\frac{3I - X_k^T X_k}{2} \right)$$

Converges to $Q = G(G^TG)^{-1/2}$ (the orthogonal component of G).

Why orthogonalize? - Keeps gradients on Stiefel manifold - Better geometry for optimization - Prevents gradient explosion/vanishing - Faster convergence

7.2.5 Distributed Muon: `nanochat/muon.py:155`

For multi-GPU training:

```
class DistMuon(Muon):
    def step(self):
        # First, average gradients across all GPUs
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is not None:
                    torch.distributed.all_reduce(p.grad, op=torch.distributed.ReduceOp.AVG)

        # Then apply standard Muon update
        super().step()
```

Key: All-reduce gradients before Muon update ensures synchronization.

7.2.6 Muon Learning Rate Scaling

```
# From scripts/base_train.py:238
dmodel_lr_scale = (model_dim / 768) ** -0.5
lr_scaled = matrix_lr  # No scaling for Muon (handles it internally)
```

Muon is **scale-invariant**, so no need to scale LR by model dimension!

7.2.7 Momentum Schedule for Muon: `scripts/base_train.py:160`

```python
def get_muon_momentum(it):
    """Warmup momentum from 0.85 to 0.95"""
    frac = min(it / 300, 1)
    momentum = (1 - frac) * 0.85 + frac * 0.95
    return momentum
```

Start with lower momentum (more responsive), increase to higher momentum (more stable).

7.3 2. AdamW Optimizer

AdamW is used for embedding and language model head parameters.

7.3.1 Standard Adam

Combines **momentum** and **adaptive learning rates**:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \text{(first moment)}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{(second moment)}$$

Bias correction:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Update:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

7.3.2 AdamW: Decoupled Weight Decay

Adam with L2 regularization:

$$\mathcal{L}' = \mathcal{L} + \frac{\lambda}{2}\|\theta\|^2$$

Problem: Weight decay interacts with adaptive learning rate in weird ways.

AdamW solution: Decouple weight decay from gradient:

$$\theta_t = (1-\lambda\eta)\theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Benefits: - Cleaner regularization - Better generalization - Less hyperparameter interaction

7.3.3 Implementation: `nanochat/adamw.py:53`

```
class DistAdamW(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight_decay=0.0):
        defaults = dict(lr=lr, betas=betas, eps=eps, weight_decay=weight_decay)
        super().__init__(params, defaults)

    @torch.no_grad()
    def step(self):
        # First, all-reduce gradients across GPUs
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is not None:
                    torch.distributed.all_reduce(p.grad, op=torch.distributed.ReduceOp.AVG)

        # Then apply AdamW update
        for group in self.param_groups:
            lr = group['lr']
            beta1, beta2 = group['betas']
            eps = group['eps']
            weight_decay = group['weight_decay']

            for p in group['params']:
                if p.grad is None:
                    continue

                grad = p.grad
                state = self.state[p]

                # Initialize state
                if len(state) == 0:
                    state['step'] = 0
                    state['exp_avg'] = torch.zeros_like(p)       # m_t
                    state['exp_avg_sq'] = torch.zeros_like(p)    # v_t

                state['step'] += 1
                exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']
                step = state['step']

                # Update biased first and second moments
```

```
                exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
                exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1 - beta2)

                # Bias correction
                bias_correction1 = 1 - beta1 ** step
                bias_correction2 = 1 - beta2 ** step
                step_size = lr / bias_correction1

                # Compute denominator (with bias correction)
                denom = (exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).add_(eps)

                # Weight decay (decoupled)
                if weight_decay != 0:
                    p.data.mul_(1 - lr * weight_decay)

                # Update parameters
                p.data.addcdiv_(exp_avg, denom, value=-step_size)
```

7.3.4 AdamW Hyperparameters in nanochat

```
# From scripts/base_train.py:228
adam_groups = [
    dict(params=lm_head_params, lr=unembedding_lr * dmodel_lr_scale),  # 0.004
    dict(params=embedding_params, lr=embedding_lr * dmodel_lr_scale),   # 0.2
]

adamw_kwargs = dict(
    betas=(0.8, 0.95),  # Instead of default (0.9, 0.999)
    eps=1e-10,
    weight_decay=weight_decay  # Usually 0.0 for small models
)
```

Why different betas? - $\beta_1 = 0.8$: Slightly less momentum (more responsive) - $\beta_2 = 0.95$: Much less variance accumulation (adapts faster)

This is better tuned for LLM training than defaults.

7.3.5 Learning Rate Scaling by Model Dimension

```
dmodel_lr_scale = (model_dim / 768) ** -0.5
```

```
# Example:
# model_dim = 1280 → scale = (1280/768)^{-0.5}  0.77
# model_dim = 384  → scale = (384/768)^{-0.5}  1.41
```

Why $\propto 1/\sqrt{d_{model}}$?

Larger models have larger gradients (sum over more dimensions). Scaling LR prevents instability.

7.4 3. Hybrid Optimizer Setup: `nanochat/gpt.py:228`

```
def setup_optimizers(self, unembedding_lr=0.004, embedding_lr=0.2, matrix_lr=0.02, weight_decay
    model_dim = self.config.n_embd
    ddp, rank, _, _ = get_dist_info()

    # Separate parameters into groups
    matrix_params = list(self.transformer.h.parameters())         # All transformer blocks
    embedding_params = list(self.transformer.wte.parameters())    # Token embeddings
    lm_head_params = list(self.lm_head.parameters())             # Output layer

    # Scale learning rates
    dmodel_lr_scale = (model_dim / 768) ** -0.5

    # AdamW for embeddings and LM head
    adam_groups = [
        dict(params=lm_head_params, lr=unembedding_lr * dmodel_lr_scale),
        dict(params=embedding_params, lr=embedding_lr * dmodel_lr_scale),
    ]
    AdamWFactory = DistAdamW if ddp else partial(torch.optim.AdamW, fused=True)
    adamw_optimizer = AdamWFactory(adam_groups, betas=(0.8, 0.95), eps=1e-10, weight_decay=weig

    # Muon for transformer matrices
    MuonFactory = DistMuon if ddp else Muon
    muon_optimizer = MuonFactory(matrix_params, lr=matrix_lr, momentum=0.95)

    # Return both optimizers
    optimizers = [adamw_optimizer, muon_optimizer]
    return optimizers
```

Why different learning rates?

Parameter	LR	Reasoning
Embeddings	0.2	Sparse updates, can handle high LR
LM head	0.004	Dense gradients, needs lower LR
Matrices	0.02	Muon handles geometry, moderate LR

7.4.1 Stepping Multiple Optimizers: `scripts/base_train.py:269`

```
# Update learning rates for all optimizers
lrm = get_lr_multiplier(step)
for opt in optimizers:
    for group in opt.param_groups:
        group["lr"] = group["initial_lr"] * lrm

# Update Muon momentum
muon_momentum = get_muon_momentum(step)
for group in muon_optimizer.param_groups:
    group["momentum"] = muon_momentum

# Step all optimizers
for opt in optimizers:
    opt.step()

# Clear gradients
model.zero_grad(set_to_none=True)
```

Important: `set_to_none=True` saves memory compared to zeroing.

7.5 4. Gradient Clipping

Prevents exploding gradients during training.

7.5.1 Global Norm Clipping: `scripts/base_train.py:265`

```
if grad_clip > 0.0:
    torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)
```

How it works:

1. Compute global gradient norm:

$$\|\mathbf{g}\|_{global} = \sqrt{\sum_{\theta \in \Theta} \|\nabla_\theta \mathcal{L}\|^2}$$

2. If too large, scale all gradients:

$$\mathbf{g}_\theta \leftarrow \frac{\text{max\_norm}}{\|\mathbf{g}\|_{global}} \mathbf{g}_\theta$$

Effect: Limits maximum gradient magnitude without changing direction.

7.5.2 Implementation Details

```
def clip_grad_norm_(parameters, max_norm, norm_type=2):
    parameters = list(filter(lambda p: p.grad is not None, parameters))

    # Compute total norm
    total_norm = torch.norm(
        torch.stack([torch.norm(p.grad.detach(), norm_type) for p in parameters]),
        norm_type
    )

    # Compute clipping coefficient
    clip_coef = max_norm / (total_norm + 1e-6)

    # Clip if necessary
    if clip_coef < 1:
        for p in parameters:
            p.grad.detach().mul_(clip_coef)

    return total_norm
```

7.6 5. Warmup and Decay Schedules

7.6.1 Why Warmup?

At initialization: - Weights are random - Gradients can be very large - Adam's second moment estimate is inaccurate

Solution: Start with low LR, gradually increase.

7.6.2 Why Decay?

Near end of training: - Model is close to optimum - Small steps refine solution - Prevents oscillation

Solution: Gradually decrease LR to 0.

7.6.3 Schedule Implementation: `scripts/base_train.py:148`

```
warmup_ratio = 0.0      # Skip warmup for simplicity
warmdown_ratio = 0.2    # Last 20% of training
final_lr_frac = 0.0     # Decay to 0

def get_lr_multiplier(it):
    warmup_iters = round(warmup_ratio * num_iterations)
    warmdown_iters = round(warmdown_ratio * num_iterations)

    if it < warmup_iters:
        # Linear warmup
        return (it + 1) / warmup_iters
    elif it <= num_iterations - warmdown_iters:
        # Constant LR
        return 1.0
    else:
        # Linear warmdown
        progress = (num_iterations - it) / warmdown_iters
        return progress * 1.0 + (1 - progress) * final_lr_frac
```

Alternative schedules: - Cosine decay: Smoother than linear - Exponential decay: Aggressive reduction - Step decay: Discrete jumps

7.7 6. Optimization Best Practices

7.7.1 Learning Rate Tuning

Too high: - Training unstable - Loss oscillates or diverges - NaN loss

Too low: - Training very slow - Gets stuck in local minima - Underfits

Good LR: - Steady loss decrease - Occasional small oscillations - Converges smoothly

7.7.2 Finding Good LR: Learning Rate Range Test

```
# Start with very low LR, gradually increase
lrs = []
losses = []

lr = 1e-8
for step in range(1000):
    loss = train_step(lr)
    lrs.append(lr)
    losses.append(loss)
    lr *= 1.01  # Increase by 1%

# Plot losses vs LRs
# Good LR is where loss decreases fastest
```

7.7.3 Batch Size Effects

Larger batch size: - More stable gradients - Better GPU utilization - Can use higher LR - Slower wall-clock time per iteration - May generalize worse

Smaller batch size: - Noisier gradients (implicit regularization) - Less GPU efficient - Lower LR needed - Faster iterations

nanochat choice: 524K tokens/batch (very large for stability)

7.8 7. Comparison: Different Optimization Strategies

Strategy	Training Speed	Final Loss	Complexity
SGD	Slow	Good	Simple
Adam	Fast	Good	Medium
AdamW	Fast	Better	Medium
Muon (matrices only)	Very Fast	Best	High
Hybrid (AdamW + Muon)	**Very Fast**	**Best**	**High**

nanochat's hybrid approach is cutting-edge!

7.9 8. Memory Optimization

7.9.1 Gradient Checkpointing (Not used in nanochat)

Trade compute for memory: - Don't store intermediate activations - Recompute during backward pass - 2× slower, but 10× less memory

7.9.2 Optimizer State Management

AdamW stores: - First moment (m): same size as parameters - Second moment (v): same size as parameters

Memory: ~2× parameter size

For 270M param model: - Parameters: 270M × 2 bytes (BF16) = 540 MB - AdamW states: 270M × 8 bytes (FP32) = 2.16 GB - Total: ~2.7 GB

7.9.3 Fused Optimizers

```
AdamW(..., fused=True)  # Uses fused CUDA kernel
```

Benefits: - Faster updates (single kernel launch) - Less memory traffic - ~10-20% speedup

7.10 Next Steps

We've covered optimization! Next, we'll explore **Implementation Details** - practical coding techniques used throughout nanochat.

Chapter 8

Putting It All Together: Implementation Guide

This section walks through implementing your own LLM from scratch, using nanochat as a guide.

8.1 Project Structure

A well-organized codebase is essential:

```
your_llm/
  src/                    # Core library
      model.py           # Model architecture
      tokenizer.py       # BPE tokenizer
      trainer.py         # Training loop
      optimizer.py       # Custom optimizers
      data.py            # Data loading
  scripts/               # Entry points
      train_tokenizer.py
      train_model.py
      generate.py
  tests/                 # Unit tests
  configs/               # Hyperparameter configs
  README.md
```

8.2 Step-by-Step Implementation

8.2.1 Step 1: Implement BPE Tokenizer

Start simple: Python-only implementation

```
class SimpleBPE:
    def __init__(self):
        self.merges = {}  # (pair) -> new_token_id
        self.vocab = {}   # token_id -> bytes

    def train(self, text_iterator, vocab_size):
        # 1. Initialize with bytes 0-255
        self.vocab = {i: bytes([i]) for i in range(256)}

        # 2. Count pairs in text
        pair_counts = count_pairs(text_iterator)

        # 3. Iteratively merge most frequent pairs
        for i in range(256, vocab_size):
            if not pair_counts:
                break

            # Find most frequent pair
            best_pair = max(pair_counts, key=pair_counts.get)

            # Record merge
            self.merges[best_pair] = i
            left, right = best_pair
            self.vocab[i] = self.vocab[left] + self.vocab[right]

            # Update pair counts
            pair_counts = update_counts(pair_counts, best_pair, i)

    def encode(self, text):
        # Convert to bytes, apply merges
        tokens = list(text.encode('utf-8'))

        while len(tokens) >= 2:
            # Find best pair to merge
            best_pair = None
            best_idx = None

            for i in range(len(tokens) - 1):
                pair = (tokens[i], tokens[i + 1])
```

```
                if pair in self.merges:
                    if best_pair is None or self.merges[pair] < self.merges[best_pair]:
                        best_pair = pair
                        best_idx = i

            if best_pair is None:
                break

            # Apply merge
            new_token = self.merges[best_pair]
            tokens = tokens[:best_idx] + [new_token] + tokens[best_idx + 2:]

        return tokens
```

Then optimize: Rewrite critical parts in Rust/C++ if needed.

8.2.2 Step 2: Implement Transformer Model

Core components:

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

class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, n_heads)
        self.mlp = MLP(d_model)
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)

    def forward(self, x):
        # Pre-norm architecture
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x

class GPTModel(nn.Module):
    def __init__(self, vocab_size, d_model, n_layers, n_heads):
        super().__init__()
```

```
        self.token_emb = nn.Embedding(vocab_size, d_model)
        self.blocks = nn.ModuleList([
            TransformerBlock(d_model, n_heads)
            for _ in range(n_layers)
        ])
        self.ln_f = nn.LayerNorm(d_model)
        self.lm_head = nn.Linear(d_model, vocab_size, bias=False)

    def forward(self, idx, targets=None):
        # Embed tokens
        x = self.token_emb(idx)

        # Pass through blocks
        for block in self.blocks:
            x = block(x)

        # Final norm and project to vocab
        x = self.ln_f(x)
        logits = self.lm_head(x)

        if targets is not None:
            # Compute loss
            loss = F.cross_entropy(
                logits.view(-1, logits.size(-1)),
                targets.view(-1)
            )
            return loss

        return logits
```

8.2.3 Step 3: Implement Training Loop

Minimal training script:

```
def train(model, train_loader, optimizer, num_steps):
    model.train()

    for step in range(num_steps):
        # Get batch
        x, y = next(train_loader)
```

```
        # Forward pass
        loss = model(x, y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Log
        if step % 100 == 0:
            print(f"Step {step}, Loss: {loss.item():.4f}")
```

Add features incrementally: 1. Learning rate scheduling 2. Gradient clipping 3. Evaluation 4. Checkpointing 5. Distributed training

8.2.4 Step 4: Data Pipeline

Efficient streaming:

```
class StreamingDataLoader:
    def __init__(self, data_files, batch_size, seq_len, tokenizer):
        self.data_files = data_files
        self.batch_size = batch_size
        self.seq_len = seq_len
        self.tokenizer = tokenizer
        self.buffer = []

    def __iter__(self):
        for file in itertools.cycle(self.data_files):
            with open(file) as f:
                for line in f:
                    # Tokenize
                    tokens = self.tokenizer.encode(line)
                    self.buffer.extend(tokens)

                    # Yield batches
                    while len(self.buffer) >= self.batch_size * self.seq_len:
                        batch = self.buffer[:self.batch_size * self.seq_len]
                        self.buffer = self.buffer[self.batch_size * self.seq_len:]
```

```
                        # Reshape to [batch_size, seq_len]
                        x = torch.tensor(batch[:-1]).view(self.batch_size, -1)
                        y = torch.tensor(batch[1:]).view(self.batch_size, -1)

                        yield x, y
```

8.3 Common Implementation Pitfalls

8.3.1 1. Shape Mismatches

Problem: Tensor dimensions don't align

Debug:

```
print(f"Q shape: {Q.shape}")  # [B, H, T, D]
print(f"K shape: {K.shape}")  # [B, H, T, D]
print(f"V shape: {V.shape}")  # [B, H, T, D]

# Attention: Q @ K^T
scores = Q @ K.transpose(-2, -1)  # [B, H, T, T]
print(f"Scores shape: {scores.shape}")
```

Solution: Add shape assertions

```
assert Q.shape == K.shape == V.shape
assert scores.shape == (B, H, T, T)
```

8.3.2 2. Gradient Flow Issues

Problem: Gradients vanish or explode

Debug:

```
for name, param in model.named_parameters():
    if param.grad is not None:
        grad_norm = param.grad.norm().item()
        print(f"{name}: grad_norm={grad_norm:.6f}")
```

Solutions: - Gradient clipping - Better initialization - Layer normalization - Residual connections

8.3.3 3. Memory Leaks

Problem: GPU memory grows over time

Common causes:

```
# BAD: Storing loss with gradients
losses.append(loss)

# GOOD: Detach from graph
losses.append(loss.item())
```

```
# BAD: Creating new tensors on GPU in loop
for _ in range(1000):
    temp = torch.zeros(1000, 1000, device='cuda')  # Leak!

# GOOD: Reuse tensors
temp = torch.zeros(1000, 1000, device='cuda')
for _ in range(1000):
    temp.zero_()
```

8.3.4 4. Incorrect Masking

Problem: Attention can see future tokens

Test:

```
def test_causal_mask():
    B, T = 2, 5
    mask = torch.tril(torch.ones(T, T))

    # Future positions should be masked
    assert mask[0, 1] == 0  # Position 0 can't see position 1
    assert mask[1, 0] == 1  # Position 1 can see position 0
```

8.4 Testing Your Implementation

8.4.1 Unit Tests

```
import unittest

class TestTransformer(unittest.TestCase):
    def test_forward_pass(self):
        model = GPTModel(vocab_size=100, d_model=64, n_layers=2, n_heads=4)
        x = torch.randint(0, 100, (2, 10))  # [batch=2, seq=10]

        logits = model(x)
```

```
        self.assertEqual(logits.shape, (2, 10, 100))  # [B, T, vocab]

    def test_loss_computation(self):
        model = GPTModel(vocab_size=100, d_model=64, n_layers=2, n_heads=4)
        x = torch.randint(0, 100, (2, 10))
        y = torch.randint(0, 100, (2, 10))

        loss = model(x, y)

        self.assertIsInstance(loss, torch.Tensor)
        self.assertEqual(loss.ndim, 0)  # Scalar
        self.assertGreater(loss.item(), 0)  # Positive loss

    def test_generation(self):
        model = GPTModel(vocab_size=100, d_model=64, n_layers=2, n_heads=4)
        model.eval()

        prompt = torch.tensor([[1, 2, 3]])  # [batch=1, seq=3]

        with torch.no_grad():
            for _ in range(5):
                logits = model(prompt)
                next_token = logits[:, -1, :].argmax(dim=-1, keepdim=True)
                prompt = torch.cat([prompt, next_token], dim=1)

        self.assertEqual(prompt.shape[1], 8)  # 3 + 5 generated tokens
```

8.4.2 Integration Tests

```
def test_training_reduces_loss():
    """Test that training actually reduces loss"""
    model = GPTModel(vocab_size=100, d_model=64, n_layers=2, n_heads=4)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    # Create dummy data
    x = torch.randint(0, 100, (8, 20))
    y = torch.randint(0, 100, (8, 20))

    # Initial loss
```

```
    with torch.no_grad():
        initial_loss = model(x, y).item()

    # Train for 100 steps
    for _ in range(100):
        loss = model(x, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Final loss
    with torch.no_grad():
        final_loss = model(x, y).item()

    # Loss should decrease
    assert final_loss < initial_loss, f"Loss did not decrease: {initial_loss:.4f} -> {final_los
```

8.5 Debugging Techniques

8.5.1 1. Overfit Single Batch

Goal: Verify model can learn

```
# Create single batch
x = torch.randint(0, 100, (8, 20))
y = torch.randint(0, 100, (8, 20))

# Train on just this batch
for step in range(1000):
    loss = model(x, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if step % 100 == 0:
        print(f"Step {step}, Loss: {loss.item():.4f}")

# Loss should go to near 0
```

If loss doesn't decrease: - Model has bugs - Learning rate too low - Gradient flow issues

8.5.2 2. Compare with Reference Implementation

```
# Your implementation
your_output = your_model(x)

# Reference (e.g., HuggingFace)
ref_output = reference_model(x)

# Should be close
diff = (your_output - ref_output).abs().max()
print(f"Max difference: {diff.item()}")
assert diff < 1e-5, "Outputs don't match!"
```

8.5.3 3. Gradient Checking

```
from torch.autograd import gradcheck

model = GPTModel(vocab_size=100, d_model=64, n_layers=2, n_heads=4)
x = torch.randint(0, 100, (2, 10), dtype=torch.float64)  # Use float64 for precision

# Check gradients
test = gradcheck(model, x, eps=1e-6, atol=1e-4)
print(f"Gradient check: {'PASS' if test else 'FAIL'}")
```

8.5.4 4. Attention Visualization

```
import matplotlib.pyplot as plt

def visualize_attention(attn_weights, tokens):
    """
    attn_weights: [num_heads, seq_len, seq_len]
    tokens: [seq_len]
    """
    fig, axes = plt.subplots(2, 4, figsize=(20, 10))

    for head in range(8):
        ax = axes[head // 4, head % 4]
        im = ax.imshow(attn_weights[head].cpu().numpy(), cmap='viridis')
        ax.set_title(f'Head {head}')
        ax.set_xlabel('Key')
```

```
        ax.set_ylabel('Query')

    plt.colorbar(im, ax=axes.ravel().tolist())
    plt.tight_layout()
    plt.show()
```

8.6 Performance Optimization

8.6.1 1. Profile Your Code

```
import torch.profiler as profiler

with profiler.profile(
    activities=[profiler.ProfilerActivity.CPU, profiler.ProfilerActivity.CUDA],
    record_shapes=True
) as prof:
    # Run training step
    loss = model(x, y)
    loss.backward()
    optimizer.step()

# Print results
print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10))
```

8.6.2 2. Use torch.compile

```
# PyTorch 2.0+
model = torch.compile(model)

# 20-30% speedup in many cases
```

8.6.3 3. Optimize Data Loading

```
# Use pin_memory for faster CPU->GPU transfer
train_loader = DataLoader(
    dataset,
    batch_size=32,
    pin_memory=True,
    num_workers=4
```

```
)

# Prefetch to GPU
for x, y in train_loader:
    x = x.to('cuda', non_blocking=True)
    y = y.to('cuda', non_blocking=True)
```

8.6.4 4. Mixed Precision Training

```
scaler = torch.cuda.amp.GradScaler()

for x, y in train_loader:
    optimizer.zero_grad()

    # Forward in BF16
    with torch.cuda.amp.autocast(dtype=torch.bfloat16):
        loss = model(x, y)

    # Backward in FP32
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
```

8.7 Scaling Up

8.7.1 From Single GPU to Multi-GPU

```
# Wrap model in DDP
model = nn.parallel.DistributedDataParallel(
    model,
    device_ids=[local_rank]
)

# Use distributed sampler
train_sampler = torch.utils.data.distributed.DistributedSampler(dataset)
train_loader = DataLoader(dataset, sampler=train_sampler)

# Run with torchrun
# torchrun --nproc_per_node=8 train.py
```

8.7.2 From Small to Large Models

1. **Start small:** 10M params, verify everything works
2. **Scale gradually:** 50M → 100M → 500M
3. **Tune hyperparameters** at each scale
4. **Monitor metrics:** Loss, perplexity, downstream tasks

8.8 Checklist for Production

☐ Model passes all unit tests
☐ Can overfit single batch
☐ Training loss decreases smoothly
☐ Validation loss tracks training loss
☐ Generated text is coherent
☐ Checkpoint saving/loading works
☐ Distributed training tested
☐ Memory usage is reasonable
☐ Training speed meets targets
☐ Code is documented

8.9 Resources for Learning More

8.9.1 Papers

- "Attention Is All You Need" (Vaswani et al., 2017)
- "Language Models are Few-Shot Learners" (GPT-3, Brown et al., 2020)
- "Training Compute-Optimal LLMs" (Chinchilla, Hoffmann et al., 2022)

8.9.2 Codebases

- **nanoGPT**: Minimal GPT implementation
- **minGPT**: Educational GPT in PyTorch
- **GPT-Neo**: Open source GPT models
- **llm.c**: GPT training in pure C/CUDA

8.9.3 Courses

- Stanford CS224N (NLP with Deep Learning)
- Fast.ai (Practical Deep Learning)
- Hugging Face Course (Transformers)

8.10 Next Steps

You now have all the knowledge to build your own LLM! The key is to:

1. **Start simple** - Get a minimal version working first
2. **Test thoroughly** - Write tests for every component
3. **Iterate** - Add features incrementally
4. **Measure** - Profile and optimize bottlenecks
5. **Scale** - Gradually increase model size and data

Good luck building!