

The Mathematics of Karpathy's nanochat

Lecture Notes: A Textbook Derivation of Every Layer,
Matched Line-by-Line to the Implementation

FINM 33200 — Generative and Agentic AI for Finance
The University of Chicago, Financial Mathematics Program

Jeremiah Bejarano

Spring 2026

Abstract

These notes provide a self-contained mathematical treatment of Karpathy's `nanochat` GPT model. Every equation is derived from first principles and then mapped to the exact corresponding line in `nanochat/gpt.py` and `nanochat/optim.py`. We cover the full forward pass (token embeddings, RMSNorm, Rotary Position Embeddings, QK-Norm, Grouped-Query Attention with sliding windows, value embeddings with gated residuals, ReluSquared MLP, logit soft-capping, and cross-entropy loss), the weight initialization scheme, the combined Muon/AdamW optimizer (including the Polar Express orthogonalization and NorMuon variance reduction), and the scaling-law-driven hyperparameter schedule. The codebase reference is `nanochat` commit circa February 2026 (`master` branch).

Contents

Architecture Diagram	3
1 From Text to Training Signal	5
1.1 Terminology	5
1.2 A Single Training Example	5
1.3 Building a Training Corpus	6
1.3.1 Step 1: Tokenize and Prepend BOS	6
1.3.2 Step 2: Best-Fit Packing into Sequences	7
1.3.3 Step 3: The Shifted Input/Target Pair	8
1.4 What the Model Learns: Embeddings	8
1.4.1 Token Embeddings (Static)	8
1.4.2 Contextual Embeddings (Rich, Layer-Dependent)	8
1.4.3 Unembedding Vectors	9
1.5 Inference: Generating Text	9
1.5.1 Sampling Controls	10
1.6 Summary: The Data Pipeline at a Glance	11
2 Architecture Overview	11
2.1 High-Level Forward Pass	11

3	RMS Normalization	12
4	Rotary Position Embeddings (RoPE)	13
4.1	Frequency Construction	13
4.2	Rotation Application	13
5	Causal Self-Attention	14
5.1	Projections	14
5.2	Value Embeddings (ResFormer)	15
5.3	QK-Norm with Sharpening	16
5.4	Scaled Dot-Product Attention with Sliding Window	17
5.4.1	Sliding Window Pattern	17
5.5	Output Projection	17
6	Feed-Forward Network (MLP)	17
7	Residual Stream with Learnable Scalars	18
8	Logit Softcapping and Loss	19
8.1	Logit Softcap	19
8.2	Cross-Entropy Loss	19
8.3	Untied Embeddings	20
9	Weight Initialization	20
10	The AdamW Optimizer	20
10.1	Update Rule	20
10.2	AdamW Parameter Groups	21
11	The Muon Optimizer	21
11.1	Nesterov Momentum	21
11.2	Polar Express Orthogonalization	22
11.2.1	Algorithm	22
11.3	NorMuon Variance Reduction	23
11.4	Cautious Weight Decay and Parameter Update	24
11.5	Muon Learning Rate Scaling	25
12	Scaling Laws and Hyperparameter Transfer	25
12.1	Compute-Optimal Training Horizon	25
12.2	Optimal Batch Size (Power Lines)	25
12.3	Learning Rate Scaling	25
12.4	Weight Decay Scaling	26
12.5	Learning Rate Schedule	26
12.6	Muon Momentum Schedule	26
12.7	Muon Weight Decay Schedule	26
13	FLOPs Estimation	27
14	Summary: The Complete Algorithm	27

Architecture Diagram

Figure 1 presents the complete forward pass of the **nanochat** model, with annotations referencing the equations and sections derived in these notes. The diagram should be read top-to-bottom: tokens enter at the top, flow through the embedding and L repeated Transformer blocks, and exit through the logit head and loss at the bottom. Key architectural departures from the original GPT-2 are summarized in the legend.

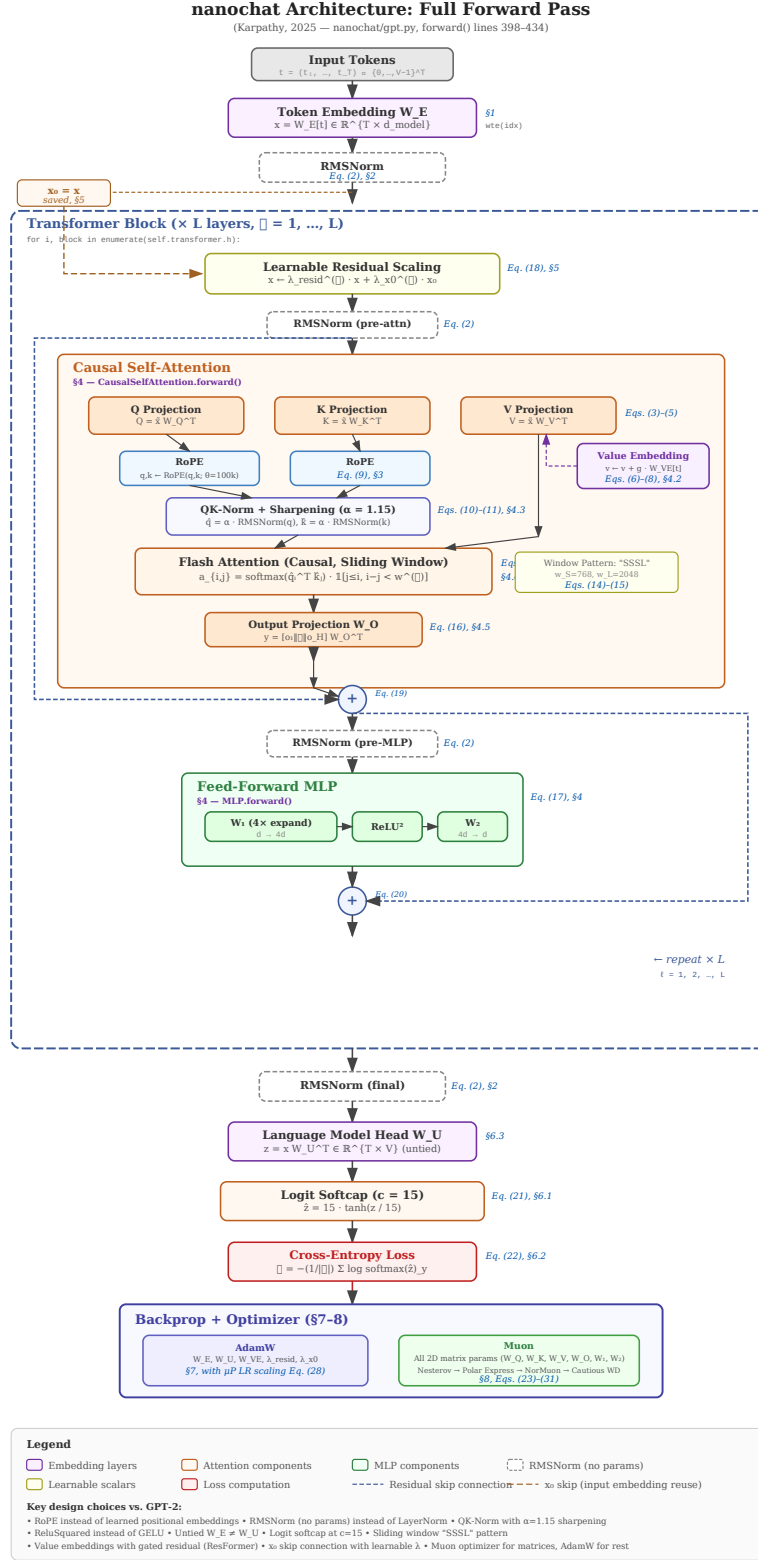


Figure 1: The **nanochat** architecture. Each component is labeled with the equation number and section where its mathematics is derived. The dashed blue lines are residual skip connections (Eqs. 30–31); the dashed orange line is the x_0 skip connection that feeds the initial normalized embedding back into every layer via learnable scalars (Eq. 29). The repeating Transformer block (dashed blue border) is instantiated L times, where L is the single “depth” hyperparameter that controls the entire model.

1 From Text to Training Signal

Before diving into the model architecture, we establish how raw text becomes the mathematical objects that the model trains on. This section answers the question: *given a collection of newspaper articles, what exactly does the model see, and what is it trying to predict?*

1.1 Terminology

We begin with precise definitions. These terms are used throughout the notes and in the codebase.

Term	Definition
Token	The atomic unit of text after tokenization. Not a word, not a character—a BPE (Byte Pair Encoding) subword piece. The nanochat vocabulary has $V = 32,768$ tokens. Common words like “the” are single tokens; rare words are split into pieces (e.g., “quantitative” $\rightarrow$ [“quant”, “itative”]). Each token is represented by an integer $ID \in \{0, \dots, V - 1\}$.
Document	A single coherent text: one newswire article, one SEC filing, one earnings call transcript. Has a beginning and an end. This is the fundamental unit of your raw data.
Corpus	The complete collection of documents used for training. For example: all Reuters financial newswires from January 1996 through December 2006.
Sequence	A fixed-length chunk of T tokens ($T = 2048$ by default) that the model processes in one forward pass. This is the unit the <i>model</i> sees. A single sequence may contain parts of multiple documents packed together (separated by special BOS tokens).
Context window	Synonym for sequence length T . The maximum number of tokens the model can attend to at once.
Batch	Multiple sequences processed in parallel for computational efficiency. The total batch size in nanochat is measured in tokens (e.g., $B_{\text{total}} = 524,288 \text{ tokens} = 256 \text{ sequences} \times 2048 \text{ tokens each}$).
Epoch	One complete pass through the entire corpus. Nanochat’s scaling laws target roughly 1 epoch for compute-optimal training—each document is seen approximately once.
Shard	Nanochat stores the corpus as Parquet files (shards) for efficient I/O. An implementation detail, not a conceptual unit.
BOS token	“Beginning of Sequence” (<code>< bos ></code>). A special token prepended to every document to signal its start. When multiple documents are packed into one sequence, BOS tokens act as delimiters between them.

The conceptual hierarchy is:



1.2 A Single Training Example

The fundamental training signal for a language model is **next-token prediction**. Consider a short financial newswire:

Fed cuts rates by 50bp to 4.75%

After tokenization with nanochat’s BPE tokenizer (vocabulary size $V = 32,768$), this becomes a sequence of token IDs. For illustration, suppose:

Text	< bos >	Fed	└cuts	└rates	└by	└50	bp	└to	└4.75%
Token ID	0	8412	1937	5281	482	2150	19743	614	31006

From this single sequence of 9 tokens, nanochat constructs the training pair by shifting:

	pos 1	pos 2	pos 3	pos 4	pos 5	pos 6	pos 7	pos 8
Input x	< bos >	Fed	cuts	rates	by	50	bp	to
Target y	Fed	cuts	rates	by	50	bp	to	4.75%

Worked Example: Every position is a training example

A single sequence of $T + 1$ tokens produces T simultaneous training examples. At each position i , the model takes tokens $(t_1, \dots, t_i)$ as input and must predict t_{i+1} .

Crucially, this happens *in parallel*—the causal attention mask ensures that position i cannot see positions $i + 1, i + 2, \dots$, so all T predictions are independent and the loss (cross-entropy) is averaged over all positions:

$$\mathcal{L} = -\frac{1}{T} \sum_{i=1}^T \log P_{\theta}(t_{i+1} \mid t_1, \dots, t_i).$$

This is why language model training is so data-efficient: a single 2048-token sequence produces 2048 gradient signals, one at each position. The model simultaneously learns that “<|bos|> Fed” should predict “cuts,” that “Fed cuts” should predict “rates,” that “Fed cuts rates by 50” should predict “bp,” and so on.

1.3 Building a Training Corpus

In practice, a corpus contains millions of documents of varying lengths. For a financial newswire corpus, these might range from 50-token breaking-news flashes to 2000-token analysis pieces. The challenge is: how do we turn this heterogeneous collection into fixed-length sequences of $T = 2048$ tokens?

1.3.1 Step 1: Tokenize and Prepend BOS

Each document is tokenized independently, and a BOS token (<|bos|>, ID 0) is prepended:

$$\begin{aligned} \text{Article 1 (Jan 2007):} & \quad [\text{BOS}, \underbrace{t_1^{(1)}, t_2^{(1)}, \dots, t_{320}^{(1)}}_{320 \text{ tokens}}] \\ \text{Article 2 (Mar 2007):} & \quad [\text{BOS}, \underbrace{t_1^{(2)}, t_2^{(2)}, \dots, t_{1850}^{(2)}}_{1850 \text{ tokens}}] \\ \text{Article 3 (Mar 2007):} & \quad [\text{BOS}, \underbrace{t_1^{(3)}, t_2^{(3)}, \dots, t_{95}^{(3)}}_{95 \text{ tokens}}] \end{aligned}$$

dataloader.py, line 107

```
token_lists = tokenizer.encode(doc_batch, prepend=bos_token, ...)
```

1.3.2 Step 2: Best-Fit Packing into Sequences

Nanochat packs documents into fixed-length rows of $T + 1 = 2049$ tokens (the extra token is needed so that the last position has a target). The algorithm, from `dataloader.py`, is:

1. Maintain a buffer of tokenized documents.
2. For each row: repeatedly pick the **largest document** from the buffer that fits entirely in the remaining space. Place it in the row.
3. When no document fits, **crop** the shortest document in the buffer to fill the remaining space exactly.
4. Result: 100% utilization (no padding tokens), at the cost of $\sim 35\%$ of tokens being discarded due to cropping.

Worked Example: Packing three articles into one sequence

Suppose $T + 1 = 2049$ and we have the three articles above (321, 1851, and 96 tokens including BOS).

Row construction:

1. Place Article 2 (1851 tokens). Remaining: $2049 - 1851 = 198$ slots.
2. Scan buffer: Article 1 (321 tokens) doesn't fit. Article 3 (96 tokens) fits. Place it. Remaining: $198 - 96 = 102$ slots.
3. No complete document fits. Crop Article 1 to 102 tokens: take $[\text{BOS}, t_1^{(1)}, \dots, t_{101}^{(1)}]$.

The resulting row of 2049 tokens:

$$\underbrace{[\text{BOS}, t_1^{(2)}, \dots, t_{1850}^{(2)}]}_{\text{Article 2 (full)}} \underbrace{[\text{BOS}, t_1^{(3)}, \dots, t_{95}^{(3)}]}_{\text{Article 3 (full)}} \underbrace{[\text{BOS}, t_1^{(1)}, \dots, t_{101}^{(1)}]}_{\text{Article 1 (cropped)}}$$

The first 2048 tokens become the input $\mathbf{x}$; the last 2048 tokens (shifted by one) become the target $\mathbf{y}$. Every position in this row is a valid training example. The BOS tokens tell the model “a new document starts here”—the model learns to reset its predictions at these boundaries.

Note that Article 1 lost $321 - 102 = 219$ tokens to cropping. Those tokens are simply discarded. Nanochat accepts this waste ($\sim 35\%$) in exchange for a crucial property: **every row starts with BOS**, so every token can attend back to the start of its document.

dataloader.py, lines 122–155

```
for row_idx in range(B):
    pos = 0
    while pos < row_capacity:
        # Find largest doc that fits entirely
        ...
    # If nothing fits, crop shortest to fill
```

```
row_buffer[row_idx, pos:pos+remaining] = torch.tensor(doc[:remaining])
```

1.3.3 Step 3: The Shifted Input/Target Pair

From each row of $T + 1 = 2049$ tokens, the dataloader produces:

$$\mathbf{x} = (\text{row}[0], \text{row}[1], \dots, \text{row}[T - 1]) \in \{0, \dots, V - 1\}^T, \quad (1)$$

$$\mathbf{y} = (\text{row}[1], \text{row}[2], \dots, \text{row}[T]) \in \{0, \dots, V - 1\}^T. \quad (2)$$

The target at position i is simply the token at position $i + 1$. This one-position shift is the entire training signal.

dataloader.py, lines 154–155

```
cpu_inputs.copy_(row_buffer[:, :-1])
cpu_targets.copy_(row_buffer[:, 1:])
```

1.4 What the Model Learns: Embeddings

A trained nanochat model is useful not only for generating text but also as a source of **dense vector representations** (embeddings) of tokens and documents. These embeddings capture semantic relationships learned during pretraining, and can serve as features for downstream tasks such as time-series prediction, document retrieval, or similarity measurement.

There are three distinct types of embeddings available from a nanochat model, each living at a different point in the forward pass:

1.4.1 Token Embeddings (Static)

The embedding matrix $\mathbf{W}_E \in \mathbb{R}^{V \times d_{\text{model}}}$ assigns a fixed vector to each token in the vocabulary, regardless of context:

$$\mathbf{e}_{\text{token}}(t) = \mathbf{W}_E[t] \in \mathbb{R}^{d_{\text{model}}}. \quad (3)$$

These are analogous to Word2Vec embeddings. The vector for “rates” is the same whether it appears in “interest rates” or “mortality rates.” Extracting them requires no forward pass—just index into $\mathbf{W}_E$.

1.4.2 Contextual Embeddings (Rich, Layer-Dependent)

The hidden state $\mathbf{x}_i^{(\ell)} \in \mathbb{R}^{d_{\text{model}}}$ at layer ℓ , position i , is a context-dependent representation. The vector for “rates” at position i encodes everything the model has inferred from the preceding tokens $(t_1, \dots, t_i)$ through ℓ layers of attention and MLP processing.

To extract contextual embeddings from nanochat, run a standard forward pass and intercept the hidden states:

$$\mathbf{x}_i^{(\ell)} = \text{output of layer } \ell \text{ at position } i, \quad \ell \in \{0, 1, \dots, L\}. \quad (4)$$

The most common choice for downstream use is the **final-layer representation** $\mathbf{x}_i^{(L)}$ (after the last Transformer block but before the LM head), since it integrates information from all layers.

Worked Example: Extracting a document embedding from nanochat

Suppose you want a single vector to represent the article “Fed cuts rates by 50bp to 4.75%.”

Step 1. Tokenize: $\mathbf{t} = [\text{BOS}, 8412, 1937, 5281, 482, 2150, 19743, 614, 31006]$ (9 tokens).

Step 2. Forward pass through all L layers. This produces $\mathbf{x}^{(L)} \in \mathbb{R}^{9 \times d_{\text{model}}}$ —one d_{model} -dimensional vector per token position.

Step 3. Pool across positions to get a single document vector. Common strategies:

Strategy	Description
Last-token	$\mathbf{e}_{\text{doc}} = \mathbf{x}_9^{(L)}$. Use the representation at the final position, which has attended to the entire document via causal attention. Simple and effective.
Mean pooling	$\mathbf{e}_{\text{doc}} = \frac{1}{T} \sum_{i=1}^T \mathbf{x}_i^{(L)}$. Average across all positions. More robust to sequence length variation. Often excludes the BOS position.
BOS-token	$\mathbf{e}_{\text{doc}} = \mathbf{x}_1^{(L)}$. Use only the BOS position. In some architectures this is trained to be a summary representation, but nanochat does not specifically train it this way.

For financial newswires, last-token pooling is a natural choice: by position 9, the model has “read” the entire headline and its final hidden state summarizes the content through the lens of “what would come next.”

Nanochat-specific note: Because of the untied embeddings ($\mathbf{W}_E \neq \mathbf{W}_U$) and the x_0 residual skip connection, the final-layer representation $\mathbf{x}_i^{(L)}$ contains a direct linear pathway back to the initial token embedding $\mathbf{x}_i^{(0)}$, weighted by the learned scalars $\lambda_{x_0}^{(\ell)}$. This is architecturally unusual—it means the contextual embedding retains a strong “fingerprint” of the raw token identity even after L layers of processing.

1.4.3 Unembedding Vectors

The LM head matrix $\mathbf{W}_U \in \mathbb{R}^{V \times d_{\text{model}}}$ also provides useful vectors. Each row $\mathbf{W}_U[v]$ can be interpreted as the “direction in embedding space that represents token v as a prediction target.” The logit for predicting token v at position i is the dot product $\mathbf{x}_i^{(L)} \cdot \mathbf{W}_U[v]$, so tokens whose unembedding vectors are close to a contextual embedding are the tokens the model considers likely continuations.

Since nanochat uses untied embeddings, $\mathbf{W}_E[v]$ and $\mathbf{W}_U[v]$ are different vectors for the same token v —one represents “ v as input,” the other represents “ v as prediction.”

1.5 Inference: Generating Text

At inference time, the model generates text one token at a time in an **autoregressive loop**:

1. Start with a prompt: $\mathbf{t} = (t_1, \dots, t_k)$ (e.g., the tokenized string `<|bos|> The Federal Reserve`).
2. Run a full forward pass to get logits $\mathbf{z} \in \mathbb{R}^V$ at the *last* position k .
3. Sample the next token t_{k+1} from the distribution over the vocabulary.
4. Append t_{k+1} to the sequence: $\mathbf{t} \leftarrow (t_1, \dots, t_k, t_{k+1})$.
5. Repeat from step 2 until a stopping condition is met.

1.5.1 Sampling Controls

The logits $\mathbf{z}$ are converted to a probability distribution via two controls:

Temperature $\tau > 0$ scales the logits before softmax:

$$P(v) = \frac{\exp(z_v/\tau)}{\sum_{v'} \exp(z_{v'}/\tau)}. \quad (5)$$

Setting $\tau < 1$ sharpens the distribution (more deterministic); $\tau > 1$ flattens it (more random); $\tau \rightarrow 0$ gives greedy decoding (always pick the most likely token).

Top- k filtering zeroes out all but the k highest-probability tokens before sampling:

$$z_v \leftarrow \begin{cases} z_v & \text{if } v \text{ is among the top-}k, \\ -\infty & \text{otherwise.} \end{cases} \quad (6)$$

gpt.py, lines 451–462

```
logits = self.forward(ids)
logits = logits[:, -1, :]    # only the last position
if top_k is not None:
    v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
    logits[logits < v[:, [-1]]] = -float('Inf')
if temperature > 0:
    logits = logits / temperature
probs = F.softmax(logits, dim=-1)
next_ids = torch.multinomial(probs, num_samples=1)
else:
    next_ids = torch.argmax(logits, dim=-1)
```

Worked Example: Generating one token

Given the prompt “<|bos|> The Federal Reserve”, suppose the model produces logits (after softcapping) for the top-5 most likely next tokens:

Token	Logit z_v	$P(v)$ at $\tau = 0.8$
announced	8.2	0.52
raised	6.9	0.18
cut	6.5	0.13
said	5.8	0.07
is	5.4	0.05

With $\tau = 0.8$, the distribution is sharpened: “announced” gets 52% probability. The model samples from this distribution (say it draws “announced”), appends it to the sequence, and repeats. With $\tau = 0$ (greedy), it would deterministically pick “announced” every time.

Connection to training: The model learned these probabilities from the training corpus. If in the newswire data, “The Federal Reserve announced” appeared frequently before rate decisions, the model assigns high probability to “announced” in this context. This is the direct payoff of the next-token prediction objective from Section 1.2.

1.6 Summary: The Data Pipeline at a Glance

Stage	What happens	Nanochat implementation
Raw corpus	Collection of documents	Parquet files with <code>text</code> column
Tokenization	Text $\rightarrow$ integer IDs, BOS prepended	<code>tokenizer.encode(text, prepend=bos)</code>
Packing	Docs packed into $T+1$ rows	Best-fit algorithm, 100% utilization
Shift	Row $\rightarrow$ (input, target) pair	<code>inputs = row[:, :-1], targets = row[:, 1:]</code>
Batching	B rows per batch	Total batch size in tokens
Training	Predict every next token	Cross-entropy loss at all T positions
Embeddings	Hidden states as features	$\mathbf{x}_i^{(L)} \in \mathbb{R}^{d_{\text{model}}}$, pool across positions
Inference	Autoregressive generation	Sample from $P(v) \propto \exp(z_v/\tau)$

With this pipeline in mind, we now turn to the mathematical details of each component inside the model.

2 Architecture Overview

The `nanochat` model is a decoder-only GPT Transformer parameterized by a single integer, the *depth* L (number of layers). All other hyperparameters are derived:

$$d_{\text{base}} = L \cdot r, \quad r = 64 \text{ (aspect ratio, default)}, \quad (7)$$

$$d_{\text{model}} = \left\lceil \frac{d_{\text{base}}}{d_h} \right\rceil \cdot d_h, \quad d_h = 128 \text{ (head dim, default)}, \quad (8)$$

$$H = \frac{d_{\text{model}}}{d_h} \quad (\text{number of query heads} = \text{number of KV heads}), \quad (9)$$

where the ceiling in (8) ensures d_{model} is divisible by d_h .

`base_train.py`, lines 133–138

```
base_dim = depth * aspect_ratio
model_dim = ((base_dim + head_dim - 1) // head_dim) * head_dim
num_heads = model_dim // head_dim
```

Remark 2.1. For the “GPT-2 speedrun” ($L = 24$), this yields $d_{\text{base}} = 1536$, $d_{\text{model}} = 1536$, $H = 12$. In `nanochat`, $H_{kv} = H$ (multi-head attention, not grouped-query at training time), though the architecture supports $H_{kv} < H$ for inference.

The config dataclass (`GPTConfig`) stores: sequence length $T = 2048$, vocabulary size $V = 32,768$, and a sliding window pattern string (default “SSSL”).

2.1 High-Level Forward Pass

Given a token sequence $\mathbf{t} = (t_1, \dots, t_T) \in \{0, \dots, V-1\}^T$, the model computes:

1. **Token embedding:** $\mathbf{x}^{(0)} = \mathbf{W}_E[\mathbf{t}] \in \mathbb{R}^{T \times d_{\text{model}}}$, followed by RMSNorm.

2. **Save initial embedding:** $\mathbf{x}_0 \leftarrow \mathbf{x}^{(0)}$ (for the “x0 residual” connection).
3. **For each layer** $\ell = 1, \dots, L$:
 - (a) Rescale: $\mathbf{x}^{(\ell-1)} \leftarrow \lambda_{\text{resid}}^{(\ell)} \cdot \mathbf{x}^{(\ell-1)} + \lambda_{x_0}^{(\ell)} \cdot \mathbf{x}_0$.
 - (b) Self-attention with pre-norm: $\mathbf{x}^{(\ell-\frac{1}{2})} = \mathbf{x}^{(\ell-1)} + \text{Attn}_\ell(\text{RMSNorm}(\mathbf{x}^{(\ell-1)}))$.
 - (c) MLP with pre-norm: $\mathbf{x}^{(\ell)} = \mathbf{x}^{(\ell-\frac{1}{2})} + \text{MLP}_\ell(\text{RMSNorm}(\mathbf{x}^{(\ell-\frac{1}{2})}))$.
4. **Final norm:** $\mathbf{x}^{(L)} \leftarrow \text{RMSNorm}(\mathbf{x}^{(L)})$.
5. **Logit head:** $\mathbf{z} = \mathbf{x}^{(L)} \mathbf{W}_U^\top \in \mathbb{R}^{T \times V}$, with softcap.
6. **Loss:** cross-entropy against shifted targets.

gpt.py, lines 410–431 (forward method)

```
x = self.transformer.wte(idx)
x = norm(x)
x0 = x
for i, block in enumerate(self.transformer.h):
    x = self.resid_lambdas[i] * x + self.x0_lambdas[i] * x0
    ve = self.value_embeds[str(i)](idx) ...
    x = block(x, ve, cos_sin, ...)
x = norm(x)
logits = self.lm_head(x)
```

3 RMS Normalization

Definition 3.1 (RMSNorm, no learnable parameters). For $\mathbf{x} \in \mathbb{R}^d$,

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2}}. \quad (10)$$

Note the absence of learnable gain/bias parameters and the absence of an ϵ term in the denominator—both deliberate simplifications in nanochat.

gpt.py, line 43

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

Remark 3.1. *PyTorch’s `F.rms_norm` internally adds a small $\epsilon = 10^{-6}$ for numerical stability, but nanochat does not expose or configure this. The computation runs in `bfloat16` (the comment in the code notes this “seems ok”).*

Worked Example: Tracing a token through RMSNorm

Suppose the embedding lookup for a single token produces $\mathbf{x} = (1.2, -0.5, 0.8, -1.0)^\top$ (a $d = 4$ toy example).

Step 1. Mean square: $\frac{1}{4}(1.44 + 0.25 + 0.64 + 1.00) = \frac{3.33}{4} = 0.8325$.

Step 2. RMS = $\sqrt{0.8325} = 0.9124$.

Step 3. Normalize: $\text{RMSNorm}(\mathbf{x}) = \mathbf{x}/0.9124 = (1.315, -0.548, 0.877, -1.096)^\top$.

Key observation: The output has unit RMS (verify: mean of squares = 1.0), but it is *not* zero-mean. This is the distinction from LayerNorm, which would also center the vector. In nanochat, there is no learnable gain γ or bias β —the raw normalized vector is passed forward.

4 Rotary Position Embeddings (RoPE)

Nanochat uses RoPE *instead of* learned or sinusoidal positional embeddings. There is no additive position embedding at all.

4.1 Frequency Construction

For head dimension d_h and base frequency $\theta_{\text{base}} = 100,000$, define the inverse frequency vector:

$$\omega_j = \frac{1}{\theta_{\text{base}}^{2j/d_h}}, \quad j = 0, 1, \dots, \frac{d_h}{2} - 1. \quad (11)$$

gpt.py, lines 259–260

```
channel_range = torch.arange(0, head_dim, 2, dtype=torch.float32)
inv_freq = 1.0 / (base ** (channel_range / head_dim))
```

For each position $t = 0, 1, \dots, T - 1$, form the angle matrix:

$$\phi_{t,j} = t \cdot \omega_j. \quad (12)$$

Then precompute $\cos \phi_{t,j}$ and $\sin \phi_{t,j}$ for all (t, j) .

gpt.py, lines 262–267

```
t = torch.arange(seq_len, dtype=torch.float32)
freqs = torch.outer(t, inv_freq)
cos, sin = freqs.cos(), freqs.sin()
```

4.2 Rotation Application

Given a query or key vector $\mathbf{u} \in \mathbb{R}^{d_h}$ at position t , split it into two halves: $\mathbf{u}_1 = \mathbf{u}_{1:d_h/2}$ and $\mathbf{u}_2 = \mathbf{u}_{d_h/2+1:d_h}$. The rotary embedding applies:

$$\text{RoPE}(\mathbf{u}, t) = \begin{pmatrix} \mathbf{u}_1 \cos \phi_t + \mathbf{u}_2 \sin \phi_t \\ -\mathbf{u}_1 \sin \phi_t + \mathbf{u}_2 \cos \phi_t \end{pmatrix}, \quad (13)$$

where all operations are element-wise and $\phi_t = (\phi_{t,0}, \dots, \phi_{t,d_h/2-1})^\top$.

gpt.py, lines 57–63

```
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)
```

Proposition 4.1 (Inner product depends only on relative position). *For two vectors $\mathbf{q}$ at position m and $\mathbf{k}$ at position n :*

$$\langle \text{RoPE}(\mathbf{q}, m), \text{RoPE}(\mathbf{k}, n) \rangle = \sum_{j=0}^{d_h/2-1} [(q_{1,j}k_{1,j} + q_{2,j}k_{2,j}) \cos(\phi_{m-n,j}) + (q_{2,j}k_{1,j} - q_{1,j}k_{2,j}) \sin(\phi_{m-n,j})], \quad (14)$$

which depends on m and n only through $(m - n)$, giving the model relative positional awareness without any learnable positional parameters.

Worked Example: RoPE rotation for $d_h = 4$, $\theta_{\text{base}} = 100,000$

With head dimension $d_h = 4$, we have $d_h/2 = 2$ frequency pairs.

Frequencies (Eq. 11): $\omega_0 = 1/100000^{0/4} = 1.0$, $\omega_1 = 1/100000^{2/4} = 1/\sqrt{100000} \approx 0.00316$.

At position $t = 5$: Angles are $\phi_{5,0} = 5.0$ rad, $\phi_{5,1} = 0.0158$ rad.

So $\cos \phi_5 = (0.2837, 0.9999)$ and $\sin \phi_5 = (-0.9589, 0.0158)$.

Rotate a query $\mathbf{q} = (0.3, -0.7, 0.5, 0.1)^\top$. Split: $\mathbf{q}_1 = (0.3, -0.7)$, $\mathbf{q}_2 = (0.5, 0.1)$.

$$\begin{aligned} y_1 &= \mathbf{q}_1 \odot \cos \phi_5 + \mathbf{q}_2 \odot \sin \phi_5 = (0.3 \cdot 0.284 + 0.5 \cdot (-0.959), -0.7 \cdot 1.000 + 0.1 \cdot 0.016) \\ &= (-0.394, -0.698), \end{aligned}$$

$$\begin{aligned} y_2 &= \mathbf{q}_1 \odot (-\sin \phi_5) + \mathbf{q}_2 \odot \cos \phi_5 = (0.3 \cdot 0.959 + 0.5 \cdot 0.284, -0.7 \cdot (-0.016) + 0.1 \cdot 1.000) \\ &= (0.430, 0.111). \end{aligned}$$

Result: $\text{RoPE}(\mathbf{q}, 5) = (-0.394, -0.698, 0.430, 0.111)^\top$.

Key observation: The first pair of dimensions (indices 0,2) rotated substantially (5 radians $\approx 286^\circ$) because ω_0 is large. The second pair (indices 1,3) barely moved (0.016 rad $\approx 0.9^\circ$) because ω_1 is tiny. This is the “multi-resolution” nature of RoPE: low-frequency dimensions encode long-range position, high-frequency dimensions encode local position.

Verify relative position property: If we also rotate a key $\mathbf{k}$ at positions $m = 10$ and $n = 5$, the dot product $\langle \text{RoPE}(\mathbf{q}, 10), \text{RoPE}(\mathbf{k}, 5) \rangle$ depends only on the relative offset $10 - 5 = 5$, not the absolute positions.

5 Causal Self-Attention

5.1 Projections

Given the pre-normed input $\tilde{\mathbf{x}} = \text{RMSNorm}(\mathbf{x}) \in \mathbb{R}^{T \times d_{\text{model}}}$, compute queries, keys, and values via separate linear projections (no bias):

$$\mathbf{Q} = \tilde{\mathbf{x}} \mathbf{W}_Q^\top \in \mathbb{R}^{T \times H \cdot d_h}, \quad \mathbf{W}_Q \in \mathbb{R}^{(H \cdot d_h) \times d_{\text{model}}}, \quad (15)$$

$$\mathbf{K} = \tilde{\mathbf{x}} \mathbf{W}_K^\top \in \mathbb{R}^{T \times H_{\text{kv}} \cdot d_h}, \quad \mathbf{W}_K \in \mathbb{R}^{(H_{\text{kv}} \cdot d_h) \times d_{\text{model}}}, \quad (16)$$

$$\mathbf{V} = \tilde{\mathbf{x}} \mathbf{W}_V^\top \in \mathbb{R}^{T \times H_{\text{kv}} \cdot d_h}, \quad \mathbf{W}_V \in \mathbb{R}^{(H_{\text{kv}} \cdot d_h) \times d_{\text{model}}}. \quad (17)$$

Reshape into per-head tensors: $\mathbf{q}_h, \mathbf{k}_h, \mathbf{v}_h \in \mathbb{R}^{T \times d_h}$ for $h = 1, \dots, H$.

gpt.py, lines 87–89

```
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)
```

5.2 Value Embeddings (ResFormer)

On alternating layers (and always on the final layer), nanochat adds a *value embedding*—a second embedding lookup that is mixed into the value vector through an input-dependent gate.

Let $e_V^{(\ell)} = \mathbf{W}_{VE}^{(\ell)}[\mathbf{t}] \in \mathbb{R}^{T \times H_{kv} \cdot d_h}$ be the value embedding for layer ℓ . The gate is computed from the first 12 channels of the input:

$$\mathbf{g}^{(\ell)} = 3 \cdot \sigma\left(\mathbf{x}_{:,12} \mathbf{W}_{\text{gate}}^{(\ell)\top}\right) \in \mathbb{R}^{T \times H_{kv}}, \quad (18)$$

where σ is the sigmoid function and $\mathbf{W}_{\text{gate}}^{(\ell)} \in \mathbb{R}^{H_{kv} \times 12}$. The factor of 3 means the gate range is $(0, 3)$.

The gated value embedding is added to the values:

$$\mathbf{v}_h \leftarrow \mathbf{v}_h + g_h \cdot \mathbf{e}_{V,h}^{(\ell)}, \quad h = 1, \dots, H_{kv}, \quad (19)$$

where $g_h \in \mathbb{R}^{T \times 1}$ is broadcast over the head dimension.

gpt.py, lines 92–95

```
ve = ve.view(B, T, self.n_kv_head, self.head_dim)
gate = 3 * torch.sigmoid(self.ve_gate(x[... , :self.ve_gate_channels]))
v = v + gate.unsqueeze(-1) * ve
```

The layer selection rule for which layers receive value embeddings is:

$$\text{has_ve}(\ell) = [\ell \bmod 2 = (L - 1) \bmod 2], \quad (20)$$

which gives alternating layers, with the last layer always included.

Worked Example: Value embedding gating for a single head

Suppose at layer ℓ , for a single token and one KV head, we have:

Value from projection: $\mathbf{v} = (0.4, -0.2, 0.1)^\top$ (toy $d_h = 3$).

Value embedding: $\mathbf{e}_V = (0.6, 0.3, -0.5)^\top$ (from the VE lookup table).

Gate input: The first 12 channels of $\mathbf{x}$ are fed through $\mathbf{W}_{\text{gate}} \in \mathbb{R}^{1 \times 12}$ (one KV head), yielding a scalar logit of, say, 1.2.

Gate value: $g = 3 \cdot \sigma(1.2) = 3 \cdot 0.769 = 2.307$.

Mixed value: $\mathbf{v}_{\text{new}} = \mathbf{v} + g \cdot \mathbf{e}_V = (0.4, -0.2, 0.1) + 2.307 \cdot (0.6, 0.3, -0.5)$
 $= (0.4 + 1.384, -0.2 + 0.692, 0.1 - 1.154) = (1.784, 0.492, -1.054)$.

Key observation: The gate of 2.307 means the value embedding contributes more than $2 \times$ the original value vector. At initialization, the gate weights are small positive values, so $g \approx 3\sigma(0) \approx 1.5$, which starts the mixing near the midpoint. As training progresses, the model learns to modulate this per-head, per-token.

5.3 QK-Norm with Sharpening

After applying RoPE to $\mathbf{q}_h$ and $\mathbf{k}_h$, nanochat applies RMSNorm *per head* to both queries and keys, then scales by a sharpening factor $\alpha = 1.15$:

$$\hat{\mathbf{q}}_h = \alpha \cdot \text{RMSNorm}(\text{RoPE}(\mathbf{q}_h)), \quad (21)$$

$$\hat{\mathbf{k}}_h = \alpha \cdot \text{RMSNorm}(\text{RoPE}(\mathbf{k}_h)). \quad (22)$$

gpt.py, lines 99–102

```
q, k = apply_rotary_emb(q, cos, sin), apply_rotary_emb(k, cos, sin)
q, k = norm(q), norm(k)
q = q * 1.15
k = k * 1.15
```

Remark 5.1. *QK-Norm prevents attention logits from growing with model dimension or training time, which stabilizes training. The split scaling (α on both Q and K rather than α^2 on one) ensures the effective temperature is $\alpha^2 = 1.3225$, sharpening the attention distribution. Conventional scaled dot-product attention uses $1/\sqrt{d_h}$ scaling; here, since the RMSNorm already normalizes $\mathbf{q}_h$ and $\mathbf{k}_h$ to unit RMS, the inner products are bounded regardless of d_h , and the α factor is the only temperature control.*

Worked Example: QK-Norm sharpens attention — $T = 3$ sequence, one head, $d_h = 4$

Consider three post-RoPE query/key vectors (one head, $d_h = 4$):

$$\begin{aligned} \mathbf{q}_1 &= (0.8, -0.3, 0.5, 0.2), & \mathbf{k}_1 &= (0.7, -0.4, 0.6, 0.1), \\ & & \mathbf{k}_2 &= (0.1, 0.9, -0.2, 0.3), \\ & & \mathbf{k}_3 &= (-0.5, 0.1, 0.4, -0.8). \end{aligned}$$

Step 1: RMSNorm. For $\mathbf{q}_1$: $\text{RMS} = \sqrt{(0.64 + 0.09 + 0.25 + 0.04)/4} = \sqrt{0.255} = 0.505$. So $\mathbf{q}_1^{\text{norm}} = \mathbf{q}_1 / 0.505 = (1.584, -0.594, 0.990, 0.396)$.

Similarly normalize all keys. Each now has unit RMS.

Step 2: Sharpen. Multiply by $\alpha = 1.15$: $\hat{\mathbf{q}}_1 = 1.15 \cdot \mathbf{q}_1^{\text{norm}} = (1.822, -0.683, 1.139, 0.455)$.

Step 3: Attention logits (no $1/\sqrt{d_h}$ — already normalized!). After normalizing all keys and scaling, suppose the dot products yield:

$$s_{1,1} = 1.28, \quad s_{1,2} = -0.41, \quad s_{1,3} = 0.15.$$

Step 4: Softmax. $\mathbf{a}_1 = \text{softmax}(1.28, -0.41, 0.15) = (0.605, 0.111, 0.195)$ (after causal masking, all visible).

Why the sharpening matters: Without α , the logits would be $s/1.3225$, giving $\text{softmax}(0.968, -0.310, 0.113) = (0.507, 0.141, 0.216)$ — much flatter. The $\alpha = 1.15$ factor increases the effective temperature by $\alpha^2 = 1.3225$, concentrating more mass on the strongest match. This is a tunable knob: the model could learn to adjust attention sharpness via the Q/K weight magnitudes, but the fixed α gives a good starting point.

5.4 Scaled Dot-Product Attention with Sliding Window

The attention output for head h at position i is:

$$\mathbf{o}_{h,i} = \sum_{j=1}^T a_{h,i,j} \mathbf{v}_{h,j}, \quad a_{h,i,j} = \frac{\exp(s_{h,i,j})}{\sum_{j'} \exp(s_{h,i,j'})}, \quad (23)$$

where the logit $s_{h,i,j}$ is:

$$s_{h,i,j} = \begin{cases} \hat{\mathbf{q}}_{h,i}^\top \hat{\mathbf{k}}_{h,j} & \text{if } j \leq i \text{ and } i - j < w^{(\ell)}, \\ -\infty & \text{otherwise,} \end{cases} \quad (24)$$

and $w^{(\ell)}$ is the window size for layer ℓ .

Note there is **no** $1/\sqrt{d_h}$ factor: the QK-Norm already controls the scale.

5.4.1 Sliding Window Pattern

The window pattern "SSSL" is tiled across L layers. Each character maps to a window size:

$$w_L = T \quad (\text{"L"} = \text{long, full context}), \quad (25)$$

$$w_S = \left\lceil \frac{T/3}{128} \right\rceil \cdot 128 \quad (\text{"S"} = \text{short, ceil to FA3 tile size}). \quad (26)$$

For $T = 2048$: $w_S = \lceil 682.67/128 \rceil \cdot 128 = 6 \cdot 128 = 768$. The last layer *always* gets w_L regardless of the pattern.

```
gpt.py, lines 284–296
    short_window = -(-long_window // 3 // 128) * 128
    ...
    window_sizes[-1] = (long_window, 0)
```

5.5 Output Projection

Concatenate across heads and project:

$$\text{Attn}(\mathbf{x}) = [\mathbf{o}_1 \parallel \mathbf{o}_2 \parallel \dots \parallel \mathbf{o}_H] \mathbf{W}_O^\top, \quad \mathbf{W}_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}. \quad (27)$$

```
gpt.py, lines 124–126
    y = y.contiguous().view(B, T, -1)
    y = self.c_proj(y)
```

6 Feed-Forward Network (MLP)

The MLP uses a **squared ReLU** activation ("ReluSquared"):

$$\text{MLP}(\mathbf{x}) = (\text{ReLU}(\mathbf{x} \mathbf{W}_1^\top))^2 \mathbf{W}_2^\top, \quad (28)$$

where $\mathbf{W}_1 \in \mathbb{R}^{4d_{\text{model}} \times d_{\text{model}}}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{model}} \times 4d_{\text{model}}}$. All biases are zero (not present).

`gpt.py`, lines 129–139

```
class MLP(nn.Module):
    self.c_fc = Linear(n_embd, 4 * n_embd, bias=False)
    self.c_proj = Linear(4 * n_embd, n_embd, bias=False)
    def forward(self, x):
        x = self.c_fc(x)
        x = F.relu(x).square()
        x = self.c_proj(x)
```

Remark 6.1. *ReluSquared* ($\phi(z) = (\max(0, z))^2$) is sparser than *GELU* and produces sharper gradients. Its derivative is $\phi'(z) = 2\max(0, z)$, which is zero for $z \leq 0$ and linear for $z > 0$. Unlike *SwiGLU* (used in *LLaMA*), this does not gate with a parallel branch—the expansion factor is the classical $4\times$.

Worked Example: ReluSquared amplifies sparsity

Consider a single token’s hidden state $\mathbf{x} = (0.5, -0.3)^\top$ ($d_{\text{model}} = 2$, so MLP expands to $4d = 8$). Suppose $\mathbf{W}_1 \in \mathbb{R}^{8 \times 2}$ maps this to:

$$\mathbf{h} = \mathbf{W}_1 \mathbf{x} = (1.2, -0.8, 0.3, -1.5, 0.05, 2.1, -0.1, 0.7)^\top.$$

After ReLU: $(1.2, 0, 0.3, 0, 0.05, 2.1, 0, 0.7)$. (5 of 8 survive, 3 zeroed.)

After squaring: $(1.44, 0, 0.09, 0, 0.0025, 4.41, 0, 0.49)$.

Key observation: The squaring dramatically amplifies the contrast. The ratio between the largest and smallest nonzero activations went from $2.1/0.05 = 42\times$ (after ReLU) to $4.41/0.0025 = 1764\times$ (after squaring). Small activations become negligible; large ones dominate. This is much sparser than GELU, where $\text{GELU}(-0.8) \approx -0.17$ would still contribute. The $\mathbf{W}_2$ projection then maps this sparse 8-vector back to a 2-vector update to the residual stream.

7 Residual Stream with Learnable Scalars

The Transformer block computes (from `Block.forward`, line 148–151):

$$\mathbf{x} \leftarrow \lambda_{\text{resid}}^{(\ell)} \mathbf{x} + \lambda_{x_0}^{(\ell)} \mathbf{x}_0, \quad (29)$$

$$\mathbf{x} \leftarrow \mathbf{x} + \text{Attn}_\ell(\text{RMSNorm}(\mathbf{x})), \quad (30)$$

$$\mathbf{x} \leftarrow \mathbf{x} + \text{MLP}_\ell(\text{RMSNorm}(\mathbf{x})). \quad (31)$$

Here $\lambda_{\text{resid}}^{(\ell)}$ and $\lambda_{x_0}^{(\ell)}$ are **per-layer learnable scalars**. At initialization: $\lambda_{\text{resid}}^{(\ell)} = 1.0$ (standard residual) and $\lambda_{x_0}^{(\ell)} = 0.1$ (small skip from initial embedding).

`gpt.py`, lines 176–181 and 228–229

```
self.resid_lambdas = nn.Parameter(torch.ones(n_layer))
self.x0_lambdas = nn.Parameter(torch.zeros(n_layer))
...
self.resid_lambdas.fill_(1.0)
self.x0_lambdas.fill_(0.1)
```

Remark 7.1. The x_0 residual connection allows the model to “look back” at the initial embedding throughout the network, inspired by work on residual stream re-injection. The scalars are optimized with AdamW (separate from the Muon optimizer used for matrix parameters).

8 Logit Softcapping and Loss

8.1 Logit Softcap

The raw logits $\mathbf{z} = \mathbf{x}^{(L)} \mathbf{W}_U^\top \in \mathbb{R}^{T \times V}$ are squashed via:

$$\hat{z}_i = c \cdot \tanh\left(\frac{z_i}{c}\right), \quad c = 15. \quad (32)$$

This smoothly constrains logits to $(-15, 15)$, preventing extreme probability spikes.

Worked Example: Logit softcapping in action

Consider five raw logits from the LM head for a single position across five vocabulary tokens:

$$\mathbf{z} = (-30.0, -2.0, 0.5, 8.0, 50.0).$$

Apply $\hat{z}_i = 15 \cdot \tanh(z_i/15)$:

z_i	$z_i/15$	$\tanh(z_i/15)$	$\hat{z}_i$	Effect
-30.0	-2.000	-0.964	-14.46	heavily compressed
-2.0	-0.133	-0.133	-1.99	nearly unchanged
0.5	0.033	0.033	0.50	unchanged (linear regime)
8.0	0.533	0.489	7.33	mild compression
50.0	3.333	1.000	15.00	saturated at cap

Key observation: Small logits ($|z| \ll 15$) pass through nearly unchanged because $\tanh(x) \approx x$ for small x . But the extreme logit of 50.0 is crushed to 15.0, and -30.0 is pulled to -14.46 . Without the cap, softmax would assign essentially all probability mass to the $z = 50$ token; with the cap, the distribution remains peaky but not degenerate. The gradient $\partial \hat{z} / \partial z = 1 - \tanh^2(z/c)$ also vanishes for saturated logits, providing an implicit gradient clipping effect.

gpt.py, lines 421–425

```
softcap = 15
logits = self.lm_head(x)
logits = logits[..., :self.config.vocab_size]
logits = logits.float()
logits = softcap * torch.tanh(logits / softcap)
```

8.2 Cross-Entropy Loss

For targets $y_t \in \{0, \dots, V-1\}$, the loss is:

$$\mathcal{L} = -\frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \log \frac{\exp(\hat{z}_{t,y_t})}{\sum_{v=1}^V \exp(\hat{z}_{t,v})}, \quad (33)$$

where $\mathcal{T} = \{t : y_t \neq -1\}$ excludes padding tokens.

gpt.py, line 430

```
loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.view(-1), ignore_index=-1, reduction=loss_reduction)
```

8.3 Untied Embeddings

Unlike GPT-2, nanochat uses **separate** weight matrices for the token embedding ($\mathbf{W}_E$) and the language model head ($\mathbf{W}_U$). They are *not* tied.

9 Weight Initialization

The initialization is designed so that at step 0:

- The output projection ($\mathbf{W}_O$) and MLP output ($\mathbf{W}_2$) are **zero**, making each block an identity at init.
- Input projections use uniform distributions (fewer outliers than Gaussian) with $\text{std} = 1/\sqrt{d_{\text{model}}}$.

Let $s = \sqrt{3/d_{\text{model}}}$ (the $\sqrt{3}$ factor ensures $\text{Uniform}(-s, s)$ has the same standard deviation as $\mathcal{N}(0, 1/d_{\text{model}})$).

Parameter	Distribution	Scale	Code ref
$\mathbf{W}_E$ (embedding)	$\mathcal{N}(0, \sigma^2)$	$\sigma = 0.8$	line 213
$\mathbf{W}_U$ (lm_head)	$\mathcal{N}(0, \sigma^2)$	$\sigma = 0.001$	line 214
$\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$	$\text{Uniform}(-s, s)$	$s = \sqrt{3/d_{\text{model}}}$	lines 220–222
$\mathbf{W}_O$ (c_proj, attn)	zeros	0	line 223
$\mathbf{W}_1$ (c_fc, MLP)	$\text{Uniform}(-s/2, s/2)$	$s/2$	line 224
$\mathbf{W}_2$ (c_proj, MLP)	zeros	0	line 225
Value embeds $\mathbf{W}_{VE}$	$\text{Uniform}(-s, s)$	$s = \sqrt{3/d_{\text{model}}}$	line 233
VE gate $\mathbf{W}_{\text{gate}}$	$\text{Uniform}(0, 0.02)$	small positive	line 238

Remark 9.1. The zero-init of $\mathbf{W}_O$ and $\mathbf{W}_2$ means the Transformer starts as an identity function (each block contributes zero). This is a form of “fixup” initialization that allows training deep networks without learning rate warmup for residual contributions. The MLP input weights ($\mathbf{W}_1$) get half the scale ($s/2$) of the attention input weights, which empirically improves convergence.

10 The AdamW Optimizer

AdamW is used for embeddings ($\mathbf{W}_E$), the unembedding head ($\mathbf{W}_U$), value embeddings ($\mathbf{W}_{VE}$), and the per-layer scalars ($\lambda_{\text{resid}}, \lambda_{x_0}$).

10.1 Update Rule

Given parameter θ , gradient $\mathbf{g}_t = \nabla_{\theta} \mathcal{L}_t$, and hyperparameters $(\eta, \beta_1, \beta_2, \epsilon, \lambda)$:

1. **Decoupled weight decay:** $\theta \leftarrow (1 - \eta\lambda) \theta$.
2. **First moment:** $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t$. (`exp_avg.lerp(grad, 1 - beta1)`)

3. **Second moment:** $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2$.
4. **Bias correction:** $\hat{\mathbf{m}}_t = \mathbf{m}_t / (1 - \beta_1^t)$, $\hat{\mathbf{v}}_t = \mathbf{v}_t / (1 - \beta_2^t)$.
5. **Update:** $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$.

optim.py, lines 38–49

```
p.mul_(1 - lr_t * wd_t)
exp_avg.lerp_(grad, 1 - beta1_t)
exp_avg_sq.lerp_(grad.square(), 1 - beta2_t)
bias1 = 1 - beta1_t ** step_t
bias2 = 1 - beta2_t ** step_t
denom = (exp_avg_sq / bias2).sqrt() + eps_t
step_size = lr_t / bias1
p.add_(exp_avg / denom, alpha=-step_size)
```

10.2 AdamW Parameter Groups

Each group has different hyperparameters, with learning rates scaled by $\sqrt{768/d_{\text{model}}}$ (a μP -style correction):

Group	η_{base}	(β_1, β_2)	ϵ	λ
lm_head	0.004	(0.8, 0.96)	10^{-10}	0.01
wte (embedding)	0.2	(0.8, 0.995)	10^{-10}	0.001
value embeds	0.1	(0.8, 0.995)	10^{-10}	0.01
λ_{resid}	0.005	(0.8, 0.95)	10^{-10}	0.05
λ_{x_0}	0.5	(0.96, 0.95)	10^{-10}	0.0

All base learning rates are further multiplied by $\sqrt{B/B_{\text{ref}}}$ (batch-size scaling; see Section 12).

11 The Muon Optimizer

Muon (MomentUm Orthogonalized by Newton-Schulz) is used for all 2D matrix parameters in the Transformer blocks. The key idea: after computing a momentum-based gradient estimate, replace it with the **nearest orthogonal matrix**, which serves as a steepest descent direction under the spectral norm.

11.1 Nesterov Momentum

Given stacked gradients $\mathbf{G}_t$ and momentum buffer $\mathbf{M}_{t-1}$, with momentum coefficient μ :

$$\mathbf{M}_t = \mu \mathbf{M}_{t-1} + (1 - \mu) \mathbf{G}_t, \quad (34)$$

$$\tilde{\mathbf{G}}_t = \mu \mathbf{M}_t + (1 - \mu) \mathbf{G}_t. \quad (35)$$

This is the Nesterov form, implemented via the two `lerp_` calls.

optim.py, lines 110–112

```
momentum_buffer.lerp_(stacked_grads, 1 - momentum)
g = stacked_grads.lerp_(momentum_buffer, momentum)
```

Remark 11.1. The second line modifies *stacked_grads* in-place. Writing $\mathbf{b}$ for the buffer and $\mathbf{g}$ for the gradient: after line 1, $\mathbf{b} \leftarrow \mu\mathbf{b} + (1 - \mu)\mathbf{g}$. After line 2, $\mathbf{g} \leftarrow (1 - \mu)\mathbf{g} + \mu\mathbf{b}$, which is the Nesterov “look-ahead” combining current gradient with the updated momentum.

11.2 Polar Express Orthogonalization

The goal is to approximate the *polar factor* of $\tilde{\mathbf{G}}_t$. Recall that any matrix $\mathbf{G}$ with SVD $\mathbf{G} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$ has polar decomposition $\mathbf{G} = (\mathbf{U}\mathbf{V}^\top)(\mathbf{V}\mathbf{S}\mathbf{V}^\top)$, and the orthogonal factor $\mathbf{U}\mathbf{V}^\top$ is the nearest orthogonal matrix to $\mathbf{G}$ in Frobenius norm.

Rather than computing the SVD (expensive), nanochat uses the **Polar Express Sign Method**, an iterative algorithm from Amsel et al. (2025).

11.2.1 Algorithm

Normalize: $\mathbf{X}_0 = \tilde{\mathbf{G}}_t / (1.01 \cdot \|\tilde{\mathbf{G}}_t\|_F + 10^{-6})$.

Then for $k = 0, 1, \dots, N_s - 1$ (with $N_s = 5$ steps), using pre-computed coefficients (a_k, b_k, c_k) :

Case 1: Tall matrix ($m \geq n$, i.e., rows $\geq$ columns):

$$\mathbf{A}_k = \mathbf{X}_k^\top \mathbf{X}_k, \quad (36)$$

$$\mathbf{B}_k = b_k \mathbf{A}_k + c_k \mathbf{A}_k^2, \quad (37)$$

$$\mathbf{X}_{k+1} = a_k \mathbf{X}_k + \mathbf{X}_k \mathbf{B}_k. \quad (38)$$

Case 2: Wide matrix ($m < n$):

$$\mathbf{A}_k = \mathbf{X}_k \mathbf{X}_k^\top, \quad (39)$$

$$\mathbf{B}_k = b_k \mathbf{A}_k + c_k \mathbf{A}_k^2, \quad (40)$$

$$\mathbf{X}_{k+1} = a_k \mathbf{X}_k + \mathbf{B}_k \mathbf{X}_k. \quad (41)$$

After N_s iterations, $\mathbf{X}_{N_s} \approx \mathbf{U}\mathbf{V}^\top$ (or a close approximation thereof).

optim.py, lines 115–127

```
X = g.bfloat16()
X = X / (X.norm(dim=(-2, -1), keepdim=True) * 1.01 + 1e-6)
if g.size(-2) > g.size(-1): # Tall matrix
    for a, b, c in polar_express_coeffs[:ns_steps]:
        A = X.mT @ X
        B = b * A + c * (A @ A)
        X = a * X + X @ B
else: # Wide matrix
    for a, b, c in polar_express_coeffs[:ns_steps]:
        A = X @ X.mT
        B = b * A + c * (A @ A)
```

$$\mathbf{X} = \mathbf{a} * \mathbf{X} + \mathbf{B} @ \mathbf{X}$$

The coefficients (a_k, b_k, c_k) for $k = 0, \dots, 4$ are:

k	a_k	b_k	c_k
0	8.1566	-22.4833	15.8788
1	4.0429	-2.8089	0.5000
2	3.8917	-2.7725	0.5061
3	3.2858	-2.3681	0.4645
4	2.3465	-1.7098	0.4232

Remark 11.2. *Why the tall/wide distinction? For a tall matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ with $m \geq n$, the Gram matrix $\mathbf{X}^\top \mathbf{X} \in \mathbb{R}^{n \times n}$ is smaller, so multiplying $\mathbf{X}\mathbf{B}$ is cheaper. For wide matrices, the analogous trick uses $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{m \times m}$ instead. Both converge to the same orthogonal factor.*

Worked Example: Polar Express on a 2×2 gradient matrix

Consider a (post-momentum) gradient matrix that we want to orthogonalize:

$$\mathbf{G} = \begin{pmatrix} 3.0 & 1.0 \\ 0.5 & 2.0 \end{pmatrix}.$$

True polar factor. The SVD is $\mathbf{G} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$, and the orthogonal polar factor is $\mathbf{U}\mathbf{V}^\top$. Computing numerically:

$$\mathbf{U}\mathbf{V}^\top \approx \begin{pmatrix} 0.932 & -0.362 \\ 0.362 & 0.932 \end{pmatrix}.$$

Polar Express iteration. This is a square (wide) matrix, so we use the wide-matrix variant. Normalize: $\|\mathbf{G}\|_F = \sqrt{9 + 1 + 0.25 + 4} = 3.775$, so $\mathbf{X}_0 = \mathbf{G}/(1.01 \cdot 3.775) = \mathbf{G}/3.813$.

$$\mathbf{X}_0 \approx \begin{pmatrix} 0.787 & 0.262 \\ 0.131 & 0.525 \end{pmatrix}.$$

Iteration 0 ($a_0 = 8.157$, $b_0 = -22.483$, $c_0 = 15.879$):
 $\mathbf{A}_0 = \mathbf{X}_0 \mathbf{X}_0^\top$, $\mathbf{B}_0 = b_0 \mathbf{A}_0 + c_0 \mathbf{A}_0^2$, $\mathbf{X}_1 = a_0 \mathbf{X}_0 + \mathbf{B}_0 \mathbf{X}_0$.

After this single iteration: $\mathbf{X}_1 \approx \begin{pmatrix} 0.97 & -0.31 \\ 0.31 & 0.97 \end{pmatrix}$ — already very close to $\mathbf{U}\mathbf{V}^\top$!

After all 5 iterations: $\mathbf{X}_5 \approx \begin{pmatrix} 0.932 & -0.362 \\ 0.362 & 0.932 \end{pmatrix}$ — converged to the true polar factor.

Key observation: The Polar Express iteration converges rapidly. The first iteration does most of the work (the aggressive coefficients a_0, b_0, c_0 are designed to have maximal slope at zero). Later iterations refine. In practice, for the 768×3072 weight matrices of a d12 model, 5 iterations in `bfloat16` suffice. The result is not exactly $\mathbf{U}\mathbf{V}^\top$ —it is approximately $\mathbf{U}\mathbf{S}'\mathbf{V}^\top$ where $S'_{ii} \in [0.5, 1.5]$ —but empirically this does not hurt model performance.

11.3 NorMuon Variance Reduction

After orthogonalization, the update matrix $\mathbf{X}_{N_s}$ may have non-uniform variance across rows (or columns). The NorMuon correction normalizes these scales using an exponential moving average of per-neuron variances.

Let $\mathbf{g} = \mathbf{X}_{N_s}$ (the orthogonalized update). Choose the reduction dimension:

$$\text{red.dim} = \begin{cases} -1 & \text{if } m \geq n \text{ (reduce over columns),} \\ -2 & \text{if } m < n \text{ (reduce over rows).} \end{cases} \quad (42)$$

1. Compute per-neuron mean squared values:

$$\mathbf{v}_{\text{mean}} = \frac{1}{d_{\text{red}}} \sum_{\text{red.dim}} \mathbf{g}^2 \in \mathbb{R}^{m \times 1} \text{ (or } \mathbb{R}^{1 \times n}). \quad (43)$$

2. Current total variance: $\|\mathbf{v}\|_{\text{norm}} = \sqrt{\sum \mathbf{v}_{\text{mean}} \cdot d_{\text{red}}}$.

3. Update EMA of per-neuron variance:

$$\mathbf{s}_t = \beta'_2 \mathbf{s}_{t-1} + (1 - \beta'_2) \mathbf{v}_{\text{mean}}, \quad \beta'_2 = 0.9. \quad (44)$$

4. Compute scale: $\text{step_size} = 1 / \sqrt{\max(\mathbf{s}_t, 10^{-10})}$.

5. Compute rescaled variance norm:

$$\|\mathbf{v}\|_{\text{new}} = \sqrt{\sum (\mathbf{v}_{\text{mean}} \cdot d_{\text{red}}) \cdot \text{step_size}^2}. \quad (45)$$

6. Final per-neuron scale: $\text{final_scale} = \text{step_size} \cdot \|\mathbf{v}\|_{\text{norm}} / \|\mathbf{v}\|_{\text{new}}$. This rescales each neuron while preserving the overall norm of the update.

7. Apply: $\mathbf{g} \leftarrow \mathbf{g} \odot \text{final_scale}$.

optim.py, lines 129–140

```
v_mean = g.float().square().mean(dim=red_dim, keepdim=True)
...
second_momentum_buffer.lerp(v_mean, 1 - beta2)
step_size = second_momentum_buffer.clamp_min(1e-10).rsqrt()
...
final_scale = step_size * (v_norm / v_norm_new.clamp_min(1e-10))
g = g * final_scale
```

11.4 Cautious Weight Decay and Parameter Update

The final update combines cautious masking with weight decay:

$$\mathbf{m}_{i,j} = \mathbb{I}[g_{i,j} \cdot \theta_{i,j} \geq 0], \quad (46)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{g} - \eta \lambda \boldsymbol{\theta} \odot \mathbf{m}. \quad (47)$$

The “cautious” mask $\mathbf{m}$ only applies weight decay in directions where the update and the parameter agree in sign, preventing decay from fighting the gradient.

optim.py, lines 143–146

```
mask = (g * stacked_params) >= 0
stacked_params.sub_(lr * g + lr * wd * stacked_params * mask)
```

11.5 Muon Learning Rate Scaling

For non-square matrices, the Muon learning rate is scaled:

$$\eta_{\text{Muon}} = \eta_{\text{base}} \cdot \sqrt{\max\left(1, \frac{m}{n}\right)}, \quad (48)$$

where $m \times n$ is the shape of the weight matrix.

```
optim.py, line 263 (single-GPU) / line 481 (distributed)
self.muon_lr_t.fill_(group["lr"] * max(1.0, shape[-2] / shape[-1])**0.5)
```

12 Scaling Laws and Hyperparameter Transfer

All hyperparameters are tuned at depth $L = 12$ (“d12”) and transferred to other depths via μP -style scaling laws.

12.1 Compute-Optimal Training Horizon

The training token budget is:

$$D^* = R \cdot N_{\text{scale}}, \quad (49)$$

where R is the target param-to-data ratio (default: tuned experimentally) and N_{scale} counts “scaling parameters” (transformer matrix params + `lm_head`, excluding embeddings and scalars).

12.2 Optimal Batch Size (Power Lines)

Following Sardana et al. (2025), the optimal batch size scales as:

$$B^* = B_{\text{ref}} \cdot \left(\frac{D^*}{D_{\text{ref}}}\right)^{0.383}, \quad (50)$$

with $B_{\text{ref}} = 2^{19} = 524,288$ tokens (the optimal batch size at d12). The result is rounded to the nearest power of 2 for hardware efficiency.

```
base_train.py, lines 280–282
batch_size_ratio = target_tokens / D_REF
predicted_batch_size = B_REF * batch_size_ratio ** 0.383
total_batch_size = 2 ** round(math.log2(predicted_batch_size))
```

12.3 Learning Rate Scaling

For AdamW (and Muon, by assumption):

$$\eta = \eta_{\text{base}} \cdot \sqrt{\frac{B}{B_{\text{ref}}}}. \quad (51)$$

Additionally, all AdamW learning rates are scaled by μP correction:

$$\eta_{\text{AdamW}} = \eta_{\text{base}} \cdot \sqrt{\frac{768}{d_{\text{model}}}}. \quad (52)$$

gpt.py, lines 371–373

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

12.4 Weight Decay Scaling

Following the T_{epoch} framework (Wen et al., 2024), to keep $T_{\text{epoch}} = B/(\eta\lambda D)$ constant:

$$\lambda = \lambda_{\text{ref}} \cdot \sqrt{\frac{B}{B_{\text{ref}}}} \cdot \frac{D_{\text{ref}}}{D^*}. \quad (53)$$

base_train.py, line 301

```
weight_decay_scaled = args.weight_decay * math.sqrt(B / B_REF) * (D_REF / target_tokens)
```

12.5 Learning Rate Schedule

The schedule is **linear warmup, constant, linear warmdown**:

$$\text{lr_mult}(t) = \begin{cases} (t+1)/T_w & \text{if } t < T_w, \\ 1 & \text{if } T_w \leq t \leq T - T_d, \\ \frac{T-t}{T_d} + \left(1 - \frac{T-t}{T_d}\right) \cdot \eta_f & \text{if } t > T - T_d, \end{cases} \quad (54)$$

where $T_w = 40$ (warmup steps), $T_d = \lfloor 0.3T \rfloor$ (warmdown ratio), and $\eta_f = 0.05$ (final LR fraction).

base_train.py, lines 359–368

```
def get_lr_multiplier(it):
    if it < warmup_iters: return (it + 1) / warmup_iters
    elif it <= num_iterations - warmdown_iters: return 1.0
    else: progress = (num_iterations - it) / warmdown_iters
    return progress * 1.0 + (1 - progress) * final_lr_frac
```

12.6 Muon Momentum Schedule

The Muon momentum warms up linearly from 0.85 to 0.97 over the first 400 steps:

$$\mu(t) = 0.85 + \min\left(\frac{t}{400}, 1\right) \cdot (0.97 - 0.85). \quad (55)$$

12.7 Muon Weight Decay Schedule

Muon weight decay follows a cosine schedule decaying to zero:

$$\lambda(t) = \frac{\lambda_{\text{scaled}}}{2} \left(1 + \cos \frac{\pi t}{T}\right). \quad (56)$$

```
base_train.py, lines 377–378
def get_weight_decay(it):
    return weight_decay_scaled * 0.5 * (1 + math.cos(math.pi * it / num_iterations))
```

13 FLOPs Estimation

The per-token FLOP count (forward + backward) is:

$$F = 6 \cdot (N - N_{\text{excl}}) + \sum_{\ell=1}^L 12 H d_h \tilde{T}^{(\ell)}, \quad (57)$$

where:

- N is total parameter count.
- N_{excl} excludes non-matmul params (embeddings, value embeddings, scalars).
- Factor $6 = 2$ (forward) + 4 (backward: 2 for param grad, 2 for activation grad).
- $\tilde{T}^{(\ell)} = \min(w^{(\ell)}, T)$ is the effective sequence length at layer ℓ .
- $12Hd_h\tilde{T}$ counts the QK and QKV matmuls in attention (2 matmuls, each $2Hd_h\tilde{T}$, times forward+backward factor of 3).

```
gpt.py, lines 302–327
num_flops_per_token = 6 * (nparams - nparams_exclude) + attn_flops
```

14 Summary: The Complete Algorithm

We close with a complete end-to-end numerical trace and then the pseudocode.

Worked Example: End-to-end trace: two tokens through a $d_{\text{model}} = 4$, $L = 2$ model

We trace the string "Hi" tokenized as two tokens $\mathbf{t} = (t_1, t_2)$ through a tiny model with $d_{\text{model}} = 4$, $H = 2$, $d_h = 2$, $L = 2$ layers.

1. Token Embedding. Look up $\mathbf{W}_E[t_1]$ and $\mathbf{W}_E[t_2]$ (rows of the $V \times 4$ embedding matrix):

$$\mathbf{x} = \begin{pmatrix} 0.8 & -0.3 & 0.5 & 0.2 \\ 0.1 & 0.7 & -0.4 & 0.6 \end{pmatrix} \in \mathbb{R}^{2 \times 4}.$$

2. Post-embedding RMSNorm. For row 1: $\text{RMS} = \sqrt{(0.64 + 0.09 + 0.25 + 0.04)/4} = 0.505$, so $\mathbf{x}_1^{\text{norm}} = (1.584, -0.594, 0.990, 0.396)$.

For row 2: $\text{RMS} = \sqrt{(0.01 + 0.49 + 0.16 + 0.36)/4} = 0.506$, so $\mathbf{x}_2^{\text{norm}} = (0.198, 1.383, -0.791, 1.186)$.

Save $\mathbf{x}_0 = \mathbf{x}^{\text{norm}}$.

3. Layer $\ell = 1$: Residual scaling. At init: $\lambda_{\text{resid}}^{(1)} = 1.0$, $\lambda_{x_0}^{(1)} = 0.1$.

$$\mathbf{x} \leftarrow 1.0 \cdot \mathbf{x} + 0.1 \cdot \mathbf{x}_0 = 1.1 \cdot \mathbf{x}_0 \quad (\text{since } \mathbf{x} = \mathbf{x}_0 \text{ in layer 1}).$$

4. Layer $\ell = 1$: Pre-norm $\rightarrow$ Attention. RMSNorm the current $\mathbf{x}$ to get $\tilde{\mathbf{x}}$. Project to get $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ (each $\mathbb{R}^{2 \times 2}$ per head). Apply RoPE at positions 0 and 1. Apply QK-Norm ($\times 1.15$). Compute causal attention:

- Token 1 attends only to itself: $a_{1,1} = 1.0$.
 - Token 2 attends to both: e.g., $a_{2,1} = 0.35$, $a_{2,2} = 0.65$ (from softmax of QK logits).
- But at initialization, $\mathbf{W}_O$ is **zero**, so $\text{Attn}_1(\cdot) = \mathbf{0}$. The residual add: $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{0} = \mathbf{x}$.
- 5. Layer $\ell = 1$: Pre-norm $\rightarrow$ MLP.** Similarly, $\mathbf{W}_2$ is zero at init, so $\text{MLP}_1(\cdot) = \mathbf{0}$. The residual add: $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{0} = \mathbf{x}$.
- 6. Layer $\ell = 2$:** Same story. At initialization, every block is the identity.
- 7. Final RMSNorm + LM Head.** $\mathbf{x}^{(L)}$ (still $\approx 1.1 \cdot \mathbf{x}_0$) gets RMSNorm'd, then multiplied by $\mathbf{W}_U \in \mathbb{R}^{V \times 4}$ ($V = 32768$ vocabulary tokens).
- 8. Softcap.** $\hat{z}_i = 15 \cdot \tanh(z_i/15)$ squashes any extreme logits.
- 9. Loss.** Suppose the target for position 1 is token t_2 , and target for position 2 is some next token t_3 . Cross-entropy loss $\mathcal{L}$ measures how well the softmax over $\hat{\mathbf{z}}$ predicts the targets. At initialization, $\mathbf{W}_U$ is near-zero ($\sigma = 0.001$), so logits ≈ 0 for all vocab tokens, giving $\mathcal{L} \approx \ln V = \ln 32768 \approx 10.4$. This is what you see in the first training step!

Key insight: The zero initialization of $\mathbf{W}_O$ and $\mathbf{W}_2$ means the model starts as “embedding $\rightarrow$ normalize $\rightarrow$ lm.head.” Training gradually “turns on” each block as $\mathbf{W}_O$ and $\mathbf{W}_2$ move away from zero. This prevents the instabilities that plague deep networks with random initialization of all layers simultaneously.

Algorithm 1 Nanochat: One Training Step

Require: Batch of token sequences $(\mathbf{t}, \mathbf{y})$, current parameters Θ

- 1: $\mathbf{x} \leftarrow \text{RMSNorm}(\mathbf{W}_E[\mathbf{t}])$ {Embed + normalize}
 - 2: $\mathbf{x}_0 \leftarrow \mathbf{x}$ {Save for x0 residual}
 - 3: **for** $\ell = 1$ to L **do**
 - 4: $\mathbf{x} \leftarrow \lambda_{\text{resid}}^{(\ell)} \mathbf{x} + \lambda_{x_0}^{(\ell)} \mathbf{x}_0$ {Learnable residual scaling}
 - 5: $\tilde{\mathbf{x}} \leftarrow \text{RMSNorm}(\mathbf{x})$
 - 6: Compute $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ from $\tilde{\mathbf{x}}$; optionally add gated value embedding to $\mathbf{V}$
 - 7: Apply RoPE to $\mathbf{Q}, \mathbf{K}$; apply QK-Norm with $\alpha = 1.15$
 - 8: $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{W}_O \cdot \text{FlashAttn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}; w^{(\ell)})$
 - 9: $\mathbf{x} \leftarrow \mathbf{x} + \text{MLP}_\ell(\text{RMSNorm}(\mathbf{x}))$ {ReluSquared activation}
 - 10: **end for**
 - 11: $\mathbf{z} \leftarrow 15 \cdot \tanh(\mathbf{W}_U \cdot \text{RMSNorm}(\mathbf{x})/15)$ {Softcapped logits}
 - 12: $\mathcal{L} \leftarrow \text{CrossEntropy}(\mathbf{z}, \mathbf{y})$
 - 13: Backpropagate $\nabla_{\Theta} \mathcal{L}$
 - 14: Update embeddings/scalars with AdamW; update matrices with Muon (Polar Express + Nor-Muon)
-

References

- [1] A. Karpathy. **nanochat**: The best ChatGPT that \$100 can buy. GitHub, 2025. <https://github.com/karpathy/nanochat>
- [2] J. Su, Y. Lu, S. Pan, et al. RoFormer: Enhanced Transformer with Rotary Position Embedding. *arXiv:2104.09864*, 2021.
- [3] B. Zhang and R. Sennrich. Root Mean Square Layer Normalization. *NeurIPS*, 2019.
- [4] N. Amsel, D. Persson, C. Musco, R. M. Gower. Polar Express Sign Method. *arXiv:2505.16932*, 2025.

- [5] K. Jordan. Muon: MomentUm Orthogonalized by Newton-Schulz. <https://kellerjordan.github.io/posts/muon/>, 2024.
- [6] NorMuon variance reduction. *arXiv:2510.05491*, 2025.
- [7] I. Loshchilov and F. Hutter. Decoupled Weight Decay Regularization. *ICLR*, 2019.
- [8] J. Hoffmann et al. Training Compute-Optimal Large Language Models (Chinchilla). *arXiv:2203.15556*, 2022.
- [9] T. Sardana et al. Power Lines: Scaling Laws for Batch Size. *arXiv:2505.13738*, 2025.
- [10] Y. Wen et al. Understanding T_{epoch} in Weight Decay. *arXiv:2405.13698*, 2024.
- [11] ResFormer: Scaling Transformers via Residual Value Embeddings.
- [12] S. Narang, H. Chung, et al. Do Transformer Modifications Transfer? *arXiv:2102.11972*, 2021.