WHICH LAYER RUNS THE PROGRAM?

NORMALIZATION, NOT DEPTH, DECIDES WHERE A TRANSFORMER EXECUTES EACH STEP

Anonymous authors
Paper under double-blind review

ABSTRACT

When a transformer is trained from random initialization to execute a Turing-complete instruction set, it reaches the input–output behavior of a hand-designed reference circuit, while realizing it through a different internal mechanism (Anonymous, 2026; Nanda et al., 2023). That comparison fixes *which* algorithm gradient descent learns; it leaves open a second, equally basic question: *where, across the network's depth, is each step of that algorithm computed, and what decides the placement?*

We study this question on the same substrate as similar recent work, a four-layer transformer trained to execute SUBLEQ (Mavaddat & Parhami, 1988), using the same hand-built circuit $\boldsymbol{\theta}_C$, now as a per-layer *answer key*: for every datapath signal (operand fetch, dereference, difference, branch, program-counter update), $\boldsymbol{\theta}_C$ fixes the layer at which it could be computed, and we require every probe to recover that layout before trusting it. Read this way, the trained network reproduces $\boldsymbol{\theta}_C$'s pipeline in the same dependency order. Where that pipeline runs, however, is decided by an unexpected component: *normalization*. In a paired, multi-seed comparison whose only difference is a pre-normalization layer, the dereference and the subtraction it feeds are deferred to the final layer without normalization and, with it, staged early at the layer $\boldsymbol{\theta}_C$ uses, under both LayerNorm and RMSNorm. The shift holds on a target no affine function of the operands can supply, so it is not an artifact of when the operands become readable.

We give an idealized-model account of the effect and turn it into a control. A short analysis shows a block's gradient scales with the norm of its input, which normalization holds to a fixed value (up to a learned gain) at every depth. Acting on this, we place normalization at a chosen layer to *relocate* the computation to that depth, show the contrast persists across depth, the normalized share decreasing toward $1/L$, and find it dissolves under a norm-free replacement. As by-products we localize that stage to attention rather than the feed-forward block and show the effect recurs across one-instruction-set variants and on modular addition.

1 INTRODUCTION

When gradient descent trains a transformer to execute an algorithmic task that admits a clean hand-designed solution, does it build the engineer's circuit? The constructive literature shows that a few carefully chosen weights suffice for Turing-complete computation (Pérez et al., 2021; Weiss et al., 2021; Giannou et al., 2023); trained from random initialization on the same task, the network is instead dense, magnitudinally larger, and mechanistically distinct (Nanda et al., 2023; Chughtai et al., 2023; Zhong et al., 2023). A recent study made the comparison exact on SUBLEQ, a one-instruction Turing-complete language (Mavaddat & Parhami, 1988): a trained transformer recovers a hand-built circuit's content-addressing *function* through a different key geometry (Anonymous, 2026).

Yet, that answers *which* algorithm gradient descent finds. That same study also noticed, but did not explain, a symptom of a different question: the learned computation clusters in the network's later layers. The question underneath is just as basic and far less studied: *where*, across the network's depth, is each step of the learned algorithm actually computed, and what decides that placement?

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The question is usually unanswerable, because we seldom know where a computation *ought* to live: mechanistic interpretability must recover its ground truth *post hoc*, proposing a circuit and then defending it (Elhage et al., 2021; Olsson et al., 2022; Wang et al., 2023; Nanda et al., 2023). The SUBLEQ setting removes the obstacle. The hand-built circuit $\boldsymbol{\theta}_C$ that served as a functional baseline also fixes, by construction, the layer at which each datapath signal could be produced (Table 1); it is a per-layer answer key against which we read a trained network's algorithm depth by depth, after requiring every probe to first recover $\boldsymbol{\theta}_C$'s known layout. We ask not what the network computes, but how it arranges the computing.

The question is also live. Recent work documents that depth is used unevenly in pretrained transformers and reads it as a geometric consequence of pre-normalization (Sun et al., 2025; Csordás et al., 2025), and Gurnee et al. (2026) report a large model's computation organized *by depth*, an intermediate-layer "global workspace" between an early sensory and a late output-aligned regime. Those are observations; we ask the interventional question on a task whose target layout is known: does normalization *cause* the placement, and can we move it? The stakes are broader than one component: as computation is increasingly obtained by *training* substrates we cannot hand-design, where a computation is realized shifts from the circuit to the optimizer, and asking what fixes that placement, in the one setting where the intended layout is known, is a first step toward reading it in models we cannot hand-build.

Our findings.

1. **Gradient descent rediscovers the pipeline.** From input–output examples alone, the trained network reproduces $\boldsymbol{\theta}_C$'s datapath in the same dependency order, gated by a sanity check on $\boldsymbol{\theta}_C$ itself (Section 4).
2. **Normalization decides where each step runs.** At fixed depth, width, activation, task, and accuracy, adding a normalization layer moves the dereference and arithmetic from the final layer to the layer $\boldsymbol{\theta}_C$ uses, under both LayerNorm and RMSNorm and across five seeds (Section 5).
3. **The cause is normalization, not a magnitude budget, and not SUBLEQ-specific.** Scaling the initialization or the residual branch changes the signal budget but leaves the late layout intact, while a mean-free normalizer (RMSNorm) reproduces the early staging exactly; the same depth-concentration signature appears across one-instruction-set variants and on modular addition, a circuit we did not hand-build (Section 5).
4. **What update-concentration measures.** A short analysis shows a block's gradient scales with the norm of its input, which normalization pins at every depth to $\sqrt{d}$ up to a learned gain (d the model width), and that the metric partly reflects the growing residual stream (Section 6).
5. **Placement is a control knob.** Placing normalization at a chosen block moves that stage to a chosen depth; the contrast holds from 2 to 12 layers, the normalized share tracking $1/L$, and a norm-free replacement (DyT) does not reproduce it, isolating the normalization operation (Section 7).
6. **The arithmetic is in attention, and depth becomes load-bearing.** Ablation localizes the instruction's arithmetic to attention, not the feed-forward block (Section 8); and while the reorganization leaves accuracy and endurance untouched, the unnormalized executor breaks when its last block is removed while the normalized one survives (Section 9).

2 Related Work

Transformers that execute algorithms. A line of work establishes that transformers can *represent* programs and algorithms. Hand-built constructions turn a transformer into a universal computer with function calls, a program counter, and conditional branches (Giannou et al., 2023), and attention is Turing-complete in the limit (Pérez et al., 2021). RASP and its compiler Tracr express algorithms as transformer programs and compile them to weights (Weiss et al., 2021; Lindner et al., 2023), and RASP-style analysis predicts which algorithmic tasks a transformer learns to length-generalize (Zhou et al., 2023). These results concern *expressivity*, what weights *can* compute by construction. We study what gradient descent *does* learn from examples, and read the *depth* at which each step of the learned program runs against a hand-built reference for the same machine.

Mechanistic interpretability with a ground truth. Reverse-engineering learned circuits is a thriving program: the Fourier algorithm a transformer groks for modular addition (Nanda et al., 2023), induction heads (Olsson et al., 2022), the indirect-object-identification circuit (Wang et al.,

2023), superposition in analytically understood toy models (Elhage et al., 2022), and automated circuit discovery (Conmy et al., 2023). In almost all of these the ground truth is recovered *post hoc* rather than fixed in advance. Our advantage is a *pre-specified* answer key: $\boldsymbol{\theta}_C$ states, before any probing, which layer could compute each datapath signal, turning "where the computation lives" into a measurement against a known target rather than an interpretation.

Normalization beyond training stability, and depth usage. Normalization layers entered the transformer to stabilize optimization (Ba et al., 2016; Zhang & Sennrich, 2019; Xiong et al., 2020), but recent work shows they do more: LayerNorm is provably part of what a *single* attention layer can express (Brody et al., 2023). A separate strand documents that depth is used unevenly, attributing it to Pre-LN's effect on trainable variance (Sun et al., 2025) or to a passive residual-stream geometry under which later layers matter less (Csordás et al., 2025). Ours is a *cross-depth* statement (*where* a computation is placed), not a within-layer expressivity one, and it is interventional rather than observational: we compare LayerNorm, RMSNorm, and no normalization at fixed depth, width, task, and accuracy, and show that normalization sets the layer at which a learned computation is performed, read against a ground-truth circuit.

The residual stream and where computation lives. A complementary literature reads or edits the residual stream of *real* language models; localizing and editing factual associations (Meng et al., 2022) and decoding intermediate predictions with learned lenses (Belrose et al., 2023). Most recently, Gurnee et al. (2026) localize a residual-stream "global workspace", a middle band of layers that holds and broadcasts abstract concepts, in a large pretrained model, operationalizing the residual-stream-as-channel view of Elhage et al. (2021). To our knowledge these methods lack a ground truth for a full algorithm; SUBLEQ supplies one, at the price of a small, transparent computer rather than a pretrained model.

Implicit bias of optimization. Gradient descent does not choose arbitrarily among the many parameters that fit the data: it is biased toward particular solutions, classically maximum-margin or low-frequency ones (Soudry et al., 2018; Rahaman et al., 2019). That literature characterizes the bias on the *function* a network computes. We surface a bias on the *algorithm*'s internal form, the depth at which each stage of a genuine program is placed, and conjecture architectural causes for it.

3 Setup: a computer with an answer key

Asking *where* a learned algorithm lives requires a task whose algorithm we can write down in advance: not just its inputs and outputs, but its internal steps and the layer each belongs to. A programmable computer with exactly one instruction supplies this.

SUBLEQ. SUBLEQ, *subtract and branch if less-or-equal-to-zero*, has a single instruction: a triple of addresses (a, b, c) that subtracts cell a from cell b, clamps and stores the result in b, and jumps to c if it is non-positive, else falls through. This alone suffices for any computation at all (Mavaddat & Parhami, 1988). Because the instruction never changes, every step exercises the *same* small datapath, so the network must learn one reusable circuit whose signals we can name and place. We train a transformer to be this machine: it reads the program counter and surrounding memory and produces the one-step update. From input–output examples alone it becomes a faithful interpreter; this paper asks *how*, across its layers, it gets there.

The answer key: a hand-built circuit. The same computation can be built by hand as a transformer of the same shape, following the looped-transformer construction of Giannou et al. (2023); call it $\boldsymbol{\theta}_C$. It is not trained and does not compete with a trained model: it is a reference whose layer assignment follows the instruction's data dependencies.

Executing one step is a chain of *content-addressed reads* and $\boldsymbol{\theta}_C$ spends one layer per link. The program counter names the instruction, so the first read *fetches* the operand addresses a, b, c (layer 1); only with a and b in hand can it *dereference* them for $\mathrm{mem}[a], \mathrm{mem}[b]$ and subtract into v, and because this read's query is the first read's output, it cannot fire until layer 2. The branch bit and next program counter follow at the output. The ordering is forced, not stylistic: the dereference and the difference cannot precede layer 2 because the addresses they query were read only at layer 1 (Section A). Throughout, $\boldsymbol{\theta}_C$ plays one role: telling us *where to look* for each signal and letting us check that we look correctly; we never report it as a result about learning. It is *a* correct construction, not the only one, so we treat its layout as a known reference, not a unique target (Section 10).

The datapath signals. Five quantities constitute one step, and we probe for each (Table 1): the *operand fetch* a, b, c; the *dereference* $\mathrm{mem}[a], \mathrm{mem}[b]$; the *difference* $\mathrm{mem}[b] - \mathrm{mem}[a]$, whose clamped form is the ALU value $v = \mathrm{clamp}(\mathrm{mem}[b] - \mathrm{mem}[a])$; the *branch bit* $\mathbf{1}[v \le 0]$; and the *next program counter* (c if the branch is taken, else the fall-through).

Reading depth with linear probes. We locate each signal by its *emergence layer*: the shallowest depth at which a linear probe on the frozen residual stream, trained against interpreter labels, recovers it above a fixed threshold. The probe is the closed-form ridge estimatoron 3,000 states, 70/30 split, averaged over three split seeds. Continuous signals are scored by R^2 and called decodable above 0.5; categorical ones are scored by accuracy corrected against their majority class, $(\mathrm{acc} - \mathrm{base})/(1 - \mathrm{base})$, with the same 0.5 bar. A signal that emerges early is computed early; one that emerges only at the last layer is deferred. The probes are read-only.

Table 1: The answer key. By construction, the hand-built circuit $\boldsymbol{\theta}_C$ computes each datapath signal at a known layer; recovering this layout with a linear probe is our *sanity gate*. The clamp itself is not in $\boldsymbol{\theta}_C$'s residual stream: the stream carries the raw difference and the clamp is applied by the output head's decode, so the answer key fixes where the operands and their difference are formed.

Datapath signal	Layer
operand fetch (a, b, c)	1
dereference $\mathrm{mem}[a], \mathrm{mem}[b]$	2
difference $\mathrm{mem}[b] - \mathrm{mem}[a]$	2
branch bit $\mathbf{1}[v \le 0]$	2
program-counter update	4

The sanity gate. On $\boldsymbol{\theta}_C$ the dereference and the difference are undecodable through layer 1 ($R^2 \le 0.03$) and jump to near-perfect recovery at layer 2 ($R^2 \ge 0.99$; Table 1), and the branch bit, scored against its majority class, does the same. The program-counter update is the one row the gate cannot pin: it is partly predictable from operand c two layers before $\boldsymbol{\theta}_C$ forms it, so we read the gate on the three signals the paper uses. We pair this positive control with a negative one: on a network trained to reverse its input, no datapath signal reaches the bar at any depth (the largest value anywhere is 0.47 for $\mathrm{mem}[b]$, and the branch bit sits at its majority class throughout). Only a probe passing both gates is trusted on the trained models, so the layer maps below measure genuine SUBLEQ computation.

Model, data, and training. The networks are four-layer, width-256 transformers with eight heads and ReLU feed-forward blocks (≈ 3.3M parameters), run without a causal mask since the task maps a whole state to the next. The state, one program-counter byte and $N = 32$ signed 8-bit cells, is a length-33 sequence of byte tokens (vocabulary 256) with learned positional embeddings.

Normalization, when present, is pre-block: we compare *LayerNorm* (Ba et al., 2016), $\mathrm{LN}(\mathbf{x}) = \boldsymbol{\gamma} \odot (\mathbf{x} - \mu\mathbf{1})/\sigma + \boldsymbol{\beta}$ with μ, σ the mean and standard deviation over the $d = 256$ coordinates of $\mathbf{x}$; *RMSNorm* (Zhang & Sennrich, 2019), $\mathrm{RMSNorm}(\mathbf{x}) = \boldsymbol{\gamma} \odot \mathbf{x}/\sqrt{\frac{1}{d}\sum_i x_i^2}$, which drops the mean-centering; and no normalization, where the gain $\boldsymbol{\gamma}$ and bias $\boldsymbol{\beta}$ are learned.

We train from Gaussian initialization (standard deviation 0.02) with AdamW (learning rate 3×10^{-4}, weight decay 0.01, gradient clip 1), a 1000-step warmup then cosine decay, batch size 256, for 80,000 steps, on states drawn on the fly under a curriculum that grows programs from one to eight instructions; the loss is a per-token cross-entropy that up-weights the positions the instruction changes. Every comparison is *matched and paired*: the networks are identical in task, depth, width, activation, optimizer, and seed, differing only in the one factor under study, and each reaches essentially perfect single-step accuracy. Full details, with a hyperparameter table, are in Section B.

The update-concentration metric. One architecture-level summary, read from the forward pass alone, captures how front- or back-loaded a network's computation is. A residual network carries a running *residual stream* $\mathbf{x}_0, \ldots, \mathbf{x}_L$ to which each block adds an update (Elhage et al., 2021); the size of block l's update $u_l = \|\mathbf{x}_{l+1} - \mathbf{x}_l\|$ is how much that block changes the ongoing computation, and $\sum_l u_l$ is the network's total change. We report the last block's share, the *update concentration* $u_{L-1}/\sum_l u_l$: near-uniform when work is spread across depth, near one when the real computation is deferred to the end, and near the low extreme for the hand-built $\boldsymbol{\theta}_C$. We prefer it to per-layer weight norms or to the probe's decodability (which a larger-amplitude signal can shift a layer earlier with no computation having moved) because it needs no labels and is seed-robust; see Section 6.

4 Gradient Descent Discovers the Pipeline

Shown only input–output examples of SUBLEQ, gradient descent rebuilds the engineer's staged datapath, each layer preparing what the next consumes, rather than some other, less legible way to be correct. Read layer by layer against the oracle, the trained network recovers every stage.

The map (Figure 1) reports, for each layer and each of the interpreter's exact one-step quantities, whether that signal is linearly present in the frozen residual stream at the compute position. It passes both gates of Section 3: on $\boldsymbol{\theta}_C$ the probes recover the known layout, placing the dereference and difference at exactly the layer $\boldsymbol{\theta}_C$ computes them, and on the reversal control no signal is decodable at any depth. The map therefore registers genuine SUBLEQ computation. manufacture.

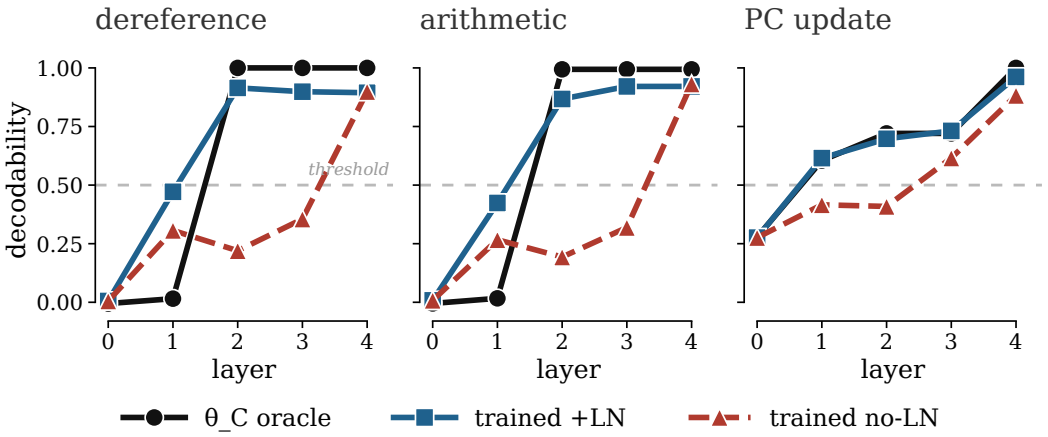

Figure 1: **Gradient descent rediscovers the engineer's datapath, but stages it at different depths.** *Black:* the hand-built oracle $\boldsymbol{\theta}_C$, whose layout is known by construction. *Blue:* a trained network with normalization. *Red:* a trained network without it. The horizontal line marks the decodability threshold at which we call a stage computed. The middle panel is the dereference-and-difference stage, the operands in hand and their subtraction formed: the oracle and the normalized network both form it at layer 2, while the unnormalized network carries its operands only diffusely through the interior and defers the arithmetic to the final layer. Every trained network here recovers the full datapath; they disagree only about *where* along their depth each stage lives.

The recovered pipeline is the engineer's, in the engineer's order: the arithmetic never becomes decodable before the dereference it consumes, the two surfacing together. What the task does *not* fix is where that pipeline lives. The two trained networks in Figure 1, both correct and running the same datapath, differing only in a standard normalization layer, place their computation at visibly different depths: one stages the dereference and arithmetic early, at layer 2 like the oracle; the other withholds them until its final layer. What controls that placement is the next section.

5 Normalization decides where computation happens

Adding a normalization layer moves the arithmetic from the last layer to layer 2. We change exactly one thing. Two families of four-layer networks are identical in task, depth, width, ReLU activation, and optimizer, and differ *only* in whether a pre-normalization layer is present. Without it, the dereference $\mathrm{mem}[b]$ and the difference $\mathrm{mem}[b] - \mathrm{mem}[a]$ do not become readable until the *final* layer; with it, both surface at layer 2, the layer $\boldsymbol{\theta}_C$ uses, and later layers merely carry the values forward. The move is robust because it holds for all five seeds with no overlap.

The arithmetic movement. Since v is affine in its operands on all but $\approx 0.5\%$ of states, a probe for v nearly repeats the probe for $\mathrm{mem}[b]$. We therefore probe what no affine function of the operands can supply: v with that fit removed, on states where the clamp fires half the time. It emerges in the *final* layer without normalization and two layers earlier with it, the arms not overlapping, so the relocation is not an artifact of operand availability (Section C.2). The branch bit tells the same story on its own scale: scored against its majority class it first exceeds chance at layer 2 in all four normalized networks and only at layer 4 in all four unnormalized ones.

Update-concentration is a seed-robust signature. Where a signal can be *seen* and where the network *does its work* need not agree. *Update-concentration*, the fraction of a network's total residual-stream change contributed by the last block, is a seed-robust signature we lead with, treating emergence as a coarser corroborating view (Figure 1). It is not by itself a measure of computation, though: it also reflects the growing stream each block acts on (Section 6), so we confirm the placement causally, by removing the last block, in Section 9. The two regimes are starkly separated: unnormalized networks

bank a last-block share of 0.88 ± 0.01 across five seeds, normalized ones 0.38 ± 0.02, and $\boldsymbol{\theta}_C$ sits at 0.01 for calibration.

The effect is normalization, not LayerNorm in particular. Retraining the early-staging family with RMSNorm, which rescales by the root-mean-square and, unlike LayerNorm, performs no mean-centering, gives a last-layer update concentration of 0.38 ± 0.03 across five seeds, on top of LayerNorm's 0.38 ± 0.02 and nowhere near the no-normalization 0.88 ± 0.01 (Figure 2). The phenomenon belongs to normalization broadly, not to one variant.

The effect is not special to SUBLEQ. For ADDLEQ (whose ALU adds) and SUBNEG (which branches on the sign of its result), normalization de-concentrates computation as it does for SUBLEQ: the last-block share falls from 0.85 to 0.37 on ADDLEQ and from 0.86 to 0.38 on SUBNEG, against SUBLEQ's $0.88 \rightarrow 0.38$, each paired across *five seeds* with standard deviations below 0.03. Normalization drives every one-instruction executor to the same ≈ 0.38 regardless of its arithmetic, and leaves every unnormalized one near 0.85.

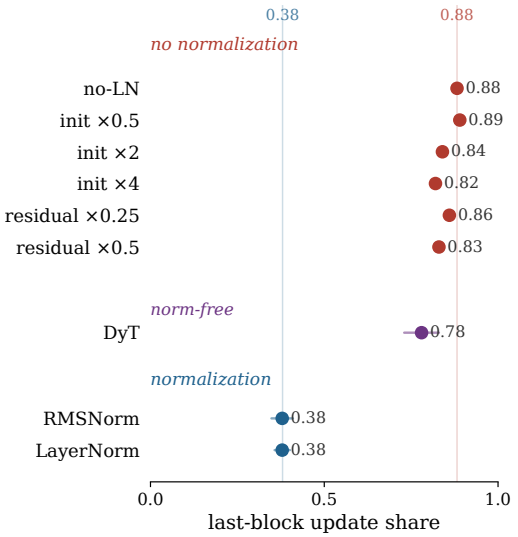

Figure 2: **Only normalization de-concentrates.** Fraction of the residual update in the last block (lower means spread across depth). No-norm sits at 0.88 ($n = 5$); changing the residual-magnitude *budget* (initialization scale $\{0.5, 2, 4\}$, residual down-scaling $\{0.25, 0.5\}$) leaves it at 0.82–0.89, falsifying the budget account, while RMSNorm and LayerNorm (0.38 each) halve it. A norm-free operation in the same slot (DyT, 0.78) does not de-concentrate (Section 7).

6 What Update Concentration Measures

Take the idealized residual network $\mathbf{x}_{l+1} = \mathbf{x}_l + \mathbf{W}_l N(\mathbf{x}_l)$, with $\mathbf{W}_l \in \mathbb{R}^{d \times d}$ the trainable parameters of block l, N the fixed normalization map ($N = \mathrm{Id}$ when it is absent), and $\mathbf{g}_{l+1} := \nabla_{\mathbf{x}_{l+1}} \mathcal{L}$ the back-propagated signal. Section C proves two facts about it, and a balancedness remark between them. They characterize the metric and explain its mechanism; they are not the paper's evidence, which is the interventions of Sections 5, 7 and 9.

One can compute that, at every layer and parameter value, $\nabla_{\mathbf{W}_l} \mathcal{L} = \mathbf{g}_{l+1} N(\mathbf{x}_l)^\top$, so $\|\nabla_{\mathbf{W}_l} \mathcal{L}\|_F = \|\mathbf{g}_{l+1}\|_2 \|N(\mathbf{x}_l)\|_2$ (Observation 1): gradient size is (back-propagated signal) $\times$ (input norm).

Normalization controls only the second factor, pinning it to $\sqrt{d}$ at every depth up to the learned gain, where without it the factor is the residual-stream norm, which grows about a hundredfold across the four layers. It leaves the first untouched, so normalization removes one of the two sources of across-depth gradient variation, not the gradient itself.

Measured in the trained networks, the input factor behaves as predicted: block inputs grow across depth without normalization ($\approx [1.0, 1.1, 5.0, 11.2]$, the last block seeing the largest) but are pinned to ≈ 11 at every depth under LayerNorm, $\sqrt{d}\gamma$ at the learned $\gamma \approx 0.7$, not the idealized $\sqrt{d} = 16$. At convergence $\|\mathbf{g}_{l+1}\| \approx 0$ (the nets reach 100% accuracy), so the factorization shapes the layout *during* training rather than at the fixed point.

Update-concentration is stream-weighted. In the scalar version of the model, gradient flow from a balanced start keeps every layer's gain equal, spreading it across depth rather than banking it in the last block (Remark 1). Even so, with those equal maps the last block's share is $a^{L-1}(a-1)/(a^L - 1) > 1/L$, produced entirely by the geometric stream $x_l = a^l$, while under pre-normalization the per-block update is depth-independent and the share is exactly $1/L$ (Corollary 1). The update $u_l = \|\mathbf{W}_l N(\mathbf{x}_l)\|$ is a block's map times its input norm, so the metric conflates how much a block transforms its input with how large that input is: a last block with no larger learned map than its predecessors still scores above $1/L$. See also Section C.1).

7 PLACING, SCALING, AND REPLACING NORMALIZATION

Placement dials the depth. If normalization sets where a step is computed, placing it should move that location. We train four-layer networks with a single normalization layer at one chosen block K and read where the arithmetic emerges. Moving the norm later moves the computation later, monotonically: with it at block 0 the dereference and ALU first decode at layer 2, at block 1 at layer 3, at block 2 at layer 4, and the update-concentration climbs in step from 0.57 to 0.90 across five seeds (Figure 3). The lone exception: placing the single norm at the *last* block leaves the network concentrated (0.90, essentially the unnormalized value) with a seed-dependent emergence layer, since no early normalization is present to stage the computation. The depth of a learned step is thus not fixed by the task; it is set by where we put the normalizer.

The effect grows with depth. Sweeping depth from 2 to 12, the unnormalized last-block share stays high (0.91 at depth 2 falling only to 0.79 at depth 12) while the normalized share *decreases toward* the $1/L$ floor ($0.54, 0.38, 0.30, 0.23, 0.19$ against $0.50, 0.25, 0.17, 0.13, 0.08$), tracking it with a persistent excess of ≈ 0.10–0.13 at depths ≥ 4 (Figure 4). That excess is not a discrepancy: Corollary 1 predicts it, since trained networks are not the flat-weight ideal, so their last block genuinely does more than the uniform share. The gap to the unnormalized share widens with depth, from 0.37 to 0.60: deeper unnormalized networks concentrate just as sharply, and normalization spreads the update toward, though not exactly to, the $1/L$ share.

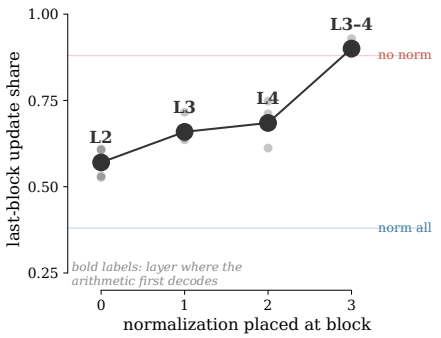

Figure 3: Update-concentration (dots, five seeds; line, mean) and the layer at which the arithmetic first decodes (bold labels) as the single normalization layer moves from block 0 to block 3.

A norm-free operation does not reproduce it. We test one representative norm-free replacement, Dynamic Tanh (Zhu et al., 2025) (DyT, $\gamma \odot \tanh(\alpha \mathbf{x}) + \beta$), which saturates but does not rescale to a fixed norm. Its learned α ranges over $[0.44, 3.43]$ across blocks ($\alpha = 3.43$ at block 3, firmly in the saturating regime), so it is a genuine saturating map, not a near-linear gain. It stays concentrated, at 0.78 ± 0.05 across five seeds, far from the 0.38 of true normalization and near the unnormalized 0.88 (Figure 2). This one learned, saturating map does not de-concentrate, consistent with the input-norm account of Observation 1. We have tested a single operation, not the space of them, so we read this as evidence that the rescaling is what matters, not proof that normalization is the only operation that could relocate computation.

Not an effective-learning-rate effect. A competing account attributes normalization's optimization role to scale-invariance coupled with weight decay, an implicit per-layer effective learning rate (van Laarhoven, 2017; Li & Arora, 2020). If that drove the depth-layout, removing weight decay should disturb it. It does not: retrained at $\lambda_{\text{wd}} = 0$ (both nets still reach 100% accuracy), the last-block shares are 0.86/0.56 (no-norm, raw/stream-normalized) and 0.35/0.016 (norm), within noise of the $\lambda_{\text{wd}} = 0.01$ values (0.88/0.60 and 0.38/0.02). The dissociation is weight-decay-independent, so the effective-learning-rate mechanism is not its cause.

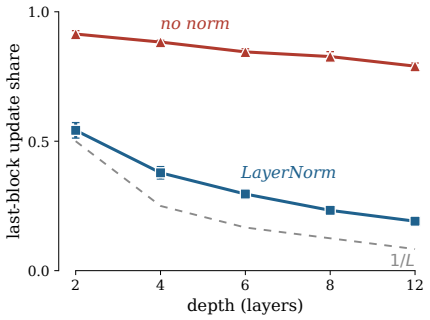

Figure 4: **Depth sweep.** Last-block update share versus depth (five seeds). Unnormalized networks stay high; normalized ones track the $1/L$ line of Corollary 1.

Placement-agnostic. The depth-usage literature ties uneven depth to *pre*-normalization specifically (Sun et al., 2025). Ours is not pre-specific: post-normalization (the norm applied to each block's *output*, $\mathbf{x} \leftarrow N(\mathbf{x} + \text{block}(\mathbf{x}))$) de-concentrates just as much: last-block share 0.28 ± 0.05 raw and 0.05 ± 0.03 stream-normalized ($n = 2$, both nets at 100% accuracy), in the same early-staging regime as pre-LN (0.38/0.02) and RMSNorm and far from the unnormalized 0.88/0.60. It is the normalization operation, at either placement, that sets the depth-layout.

8 THE ARITHMETIC LIVES IN ATTENTION

Having an idea *where in depth* each stage is computed, we ask *which part of the layer* does it. The block offers two candidates: attention, which moves information between positions, and the feed-forward (FFN) path, which transforms each position in place. Thus far, in every network the dereference-and-difference stage (v before the clamp, $\mathrm{mem}[b] - \mathrm{mem}[a]$) first becomes decodable at one identifiable layer (the last without normalization, an early one with it), so we know which block does the subtraction. Holding the rest of the network fixed, we zero that block's attention and measure v's decodability at the output, then separately zero its FFN.

It turns out the arithmetic travels with attention. Ablating the block's *attention* collapses v, its decodability falling by roughly half, a mean drop of 0.56 ± 0.09 in the unnormalized networks (arithmetic at the last layer) and 0.47 ± 0.05 in the normalized ones (arithmetic early). Ablating that same block's *FFN* does essentially nothing, a drop within noise of zero (Figure 5).

This is consistent across all ten networks (five unnormalized, five normalized) and holds *regardless of where the arithmetic landed*: in both regimes the decisive contribution is the attention of whichever block does the work, and its FFN is dispensable. Normalization sets the depth of the arithmetic; *that* attention performs it is invariant to that choice: two orthogonal findings. So, the attention path does double duty, gathering the operands and forming their clamped difference. *One caveat on what "arithmetic" means here.* The memory cells span a narrow range (standard deviation 16.7), so the clamp saturates on only $\approx 0.5\%$ of operand pairs and v is affine in the operands over the remaining 99.5%. The ablation therefore localizes the operand gathering-and-combination to attention.

9 WHEN DEPTH IS OF STRUCTURAL IMPORTANCE

Does the depth-layout matter? Not for endurance: over 300 random programs of up to 40 steps, normalized and unnormalized networks run equally long before diverging from the interpreter, a median of about five steps and indistinguishable across five seeds. It matters in a different way: a network that banks its computation in the last block cannot be shortened. We truncate each network to its first k of four blocks and read the output head. Without normalization, written-cell accuracy is flat near 0.30 until the last block and then jumps to 1.00; with normalization it climbs steadily, already reaching 0.82 (LayerNorm) and 0.81 (RMSNorm) once the last block is removed (Figure 6, five seeds). The unnormalized network breaks when its last block is dropped because all of its work is there; the normalized one degrades gracefully.[1]

This is a per-component effect: the branch and next program counter resolve late in both networks, late even in $\boldsymbol{\theta}_C$ by data dependency, so what moves is the operand-and-arithmetic path. Normalization sets how much of a network's depth is load-bearing, without changing its accuracy or endurance.

10 DISCUSSION, LIMITATIONS, AND OUTLOOK

What the result is. Because the same SUBLEQ computation can be written by hand as a transformer of the same shape, we read gradient descent's implicit bias off the *algorithm*, where each stage is computed, rather than off the function's margin or norm (Soudry et al., 2018), and $\boldsymbol{\theta}_C$ makes this a measurement rather than a post-hoc circuit. What we find supports a growing view that normalization does more than stabilize training (Brody et al., 2023; Sun et al., 2025; Csordás et al., 2025). Two simpler explanations fail: it does not cap the residual norm, which grows by two orders of magnitude either way because the pre-norm layer rescales each block's *input* and not the stream, and it is not a magnitude budget, since initialization- and residual-scaling leave the late binding intact whereas mean-free RMSNorm removes it.

Limitations. *One task family.* Every result concerns SUBLEQ-like architectures, replicating across SUBLEQ, ADDLEQ, and SUBNEG under both LayerNorm and RMSNorm, each paired across five seeds. *One size regime.* We study a single band of width and depth; our depth control shows only that, with normalization present, four versus six layers does not move where the stages land. *No accuracy or endurance hook.* The reorganization leaves single-step accuracy and rollout endurance untouched

[1] The unnormalized stream grows about tenfold in its last block, so truncation hands the output head a scale it never saw, while the normalized network ends in a norm and is scale-invariant to truncation. Part of the gap is therefore a readout mismatch, and the per-depth recalibration that would separate the two is not run.

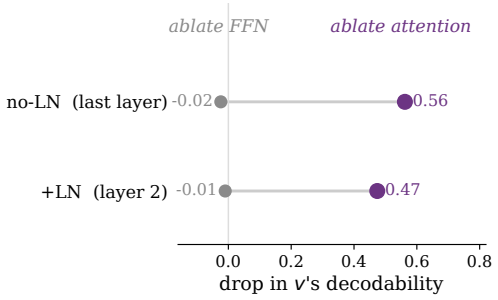

Figure 5: **Arithmetic is carried by attention, not the FFN.** Drop in v's output decodability when we zero the *attention* (purple) versus the *FFN* (grey) of the block that computes v. Five seeds per condition (ten networks).

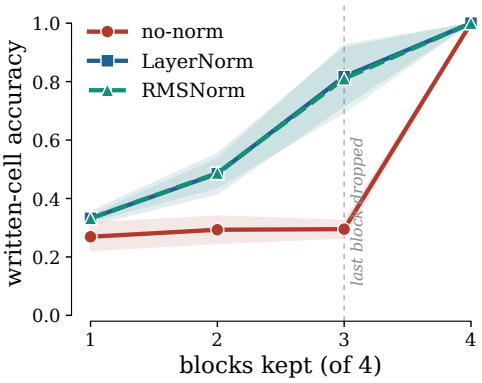

Figure 6: **Depth becomes load-bearing.** Written-cell output accuracy when the network is truncated to its first k blocks; $k = 4$ is full depth. Five seeds; bands are ± 1 s.d.

(Section 9); what it changes is how much of a network's depth is load-bearing, read off the write-back stage, not a task-performance gain.

Outlook. Normalization fixing depth-placement is a design rule we can act on, re-targeting the normalizer to relocate the computation (Section 7). Whether such rules hold where we cannot hand-build the circuit is the open question, and one cheap test needs no construction: on modular addition, whose Fourier-multiplication circuit was independently reverse-engineered (Nanda et al., 2023), removing normalization again banks the update late (0.89 ± 0.05 over five seeds) while LayerNorm and RMSNorm distribute it (0.31 and 0.32), tracking SUBLEQ's 0.88 and 0.38 (Figure 7). The step we leave open is real pretrained models: with a capability whose location is fixed in advance (factual recall and ROME (Meng et al., 2022), indirect-object identification (Wang et al., 2023), induction heads (Olsson et al., 2022)), one could re-target normalization at chosen layers and test whether the capability moves. Whether the dissociation survives a model whose normalization is learned and entangled is what SUBLEQ cannot tell us.

AI USE STATEMENT

We used large language model assistants to write, test, and independently re-implement analysis code; to run experiments and check their results; to search the literature and check references against their sources; to produce figures; and to draft and edit text. The authors chose the research question, approved the experimental design and every claim, and checked AI-assisted code, numbers, and citations against results files and sources. We take responsibility for the final content of this work, including text, claims and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The supplementary material contains the code that trains every network, builds the hand-coded circuit $\boldsymbol{\theta}_C$, and runs the probing, ablation, truncation, and concentration experiments, with unit tests for the probe estimator and the datapath labels. Architecture, task encoding, and training hyperparameters are in Section 3 and Table 2; the construction that fixes the answer key is specified in Section A; the probe protocol and both control gates are in Section 3, with the transport-immune variant in Section C.2; proofs are in Section C.

REFERENCES

Anonymous. Can we train a computer? Two mechanisms for content addressing in transformers, 2026. Under review.

Sanjeev Arora, Nadav Cohen, Wei Hu, and Yuping Luo. Implicit regularization in deep matrix factorization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. arXiv:1905.13655.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Francis Bach. Closed-form dynamics beyond quadratics. Machine Learning Research Blog, 2026. https://francisbach.com/closed-form-dynamics/.

Nora Belrose, Igor Ostrovsky, Lev McKinney, Zach Furman, Logan Smith, Danny Halawi, Stella Biderman, and Jacob Steinhardt. Eliciting latent predictions from transformers with the tuned lens. *arXiv preprint arXiv:2303.08112*, 2023.

Shaked Brody, Uri Alon, and Eran Yahav. On the expressivity role of LayerNorm in transformers' attention. In *Findings of the ACL*, 2023. arXiv:2305.02582.

Bilal Chughtai, Lawrence Chan, and Neel Nanda. A toy model of universality: Reverse engineering how networks learn group operations. In *ICML*, 2023.

Arthur Conmy, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards automated circuit discovery for mechanistic interpretability. In *NeurIPS*, 2023.

Róbert Csordás, Christopher D. Manning, and Christopher Potts. Do language models use their depth efficiently? In *NeurIPS*, 2025. arXiv:2505.13898.

Simon S Du, Wei Hu, and Jason D Lee. Algorithmic regularization in learning deep homogeneous models: Layers are automatically balanced. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. arXiv:1806.00900.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. Anthropic transformer-circuits.pub, 2021.

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy models of superposition. Anthropic transformer-circuits.pub, 2022. arXiv:2209.10652.

Angeliki Giannou, Shashank Rajput, Jy-Yong Sohn, Kangwook Lee, Jason D. Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *ICML*, 2023.

Wes Gurnee, Nicholas Sofroniew, Adam Pearce, Mateusz Piotrowski, Isaac Kauvar, Runjin Chen, Anna Soligo, Paul Bogdan, Euan Ong, Rowan Wang, Ben Thompson, David Abrahams, Subhash Kantamneni, Emmanuel Ameisen, Joshua Batson, and Jack Lindsey. Verbalizable representations form a global workspace in language models. Transformer Circuits Thread, 2026. https://transformer-circuits.pub/2026/workspace/.

Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. In *International Conference on Learning Representations (ICLR)*, 2020. URL https://openreview.net/forum?id=rJg8TeSFDH.

David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Thomas McGrath, and Vladimir Mikulik. Tracr: Compiled transformers as a laboratory for interpretability. In *NeurIPS*, 2023. arXiv:2301.05062.

Farhad Mavaddat and Behrooz Parhami. URISC: The ultimate reduced instruction set computer. *International Journal of Electrical Engineering & Education*, 25(4):327–334, 1988. doi: 10.1177/002072098802500408.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in GPT. In *NeurIPS*, 2022. arXiv:2202.05262.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. In *ICLR*, 2023.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context learning and induction heads. Anthropic transformer-circuits.pub, 2022.

Jorge Pérez, Pablo Barceló, and Javier Marinković. Attention is Turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021.

Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *ICML*, 2019. arXiv:1806.08734.

Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The implicit bias of gradient descent on separable data. *Journal of Machine Learning Research*, 19(70):1–57, 2018.

Wenfang Sun, Xinyuan Song, Pengxiang Li, Lu Yin, Yefeng Zheng, and Shiwei Liu. The curse of depth in large language models. In *NeurIPS*, 2025. arXiv:2502.05795.

Twan van Laarhoven. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350*, 2017.

Kevin Ro Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the wild: A circuit for indirect object identification in GPT-2 small. In *ICLR*, 2023.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *ICML*, 2021.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. In *ICML*, 2020. arXiv:2002.04745.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *NeurIPS*, 2019. arXiv:1910.07467.

Ziqian Zhong, Ziming Liu, Max Tegmark, and Jacob Andreas. The clock and the pizza: Two stories in mechanistic explanation of neural networks. In *NeurIPS*, 2023.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Joshua Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? A study in length generalization. *arXiv preprint arXiv:2310.16028*, 2023.

Jiachen Zhu, Xinlei Chen, Kaiming He, Yann LeCun, and Zhuang Liu. Transformers without normalization. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025. arXiv:2503.10622.