CLOSING THE LOOP: PRACTICAL TRAINING RECIPES FOR LOOPED LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Looped language models increase effective depth by repeatedly applying a shared block of layers, but existing large-scale recipes require multi-stage training over trillions of tokens, while the benefits of recurrence remain difficult to separate from differences in data and training. In this work, we establish practical training recipes for looped language models, with three main results. (1) We develop a compute-efficient from-scratch pipeline that reduces the training budget from 7.7T tokens in Ouro to 310B tokens while retaining strong reasoning performance. Pretraining followed by high-quality mid-training, together with learning-rate warmup and stronger exit-gate regularization, enables stable recurrent training without prior multi-stage schedules. (2) Under controlled comparisons, our 1.4B LoopLM outperforms a parameter-matched dense model trained on the same data and token budget on all 12 evaluated benchmarks, including +14 points on GSM8K, +10 on MATH, and +22 on DROP. At matched inference compute, it approaches a 3.9B dense model on mathematical reasoning and reading comprehension while using only 36% as many parameters. (3) We introduce a minimal recipe for converting pretrained dense models into looped ones: a single learned input-mixing scalar and a smoothed exit loss, with no step-specific parameters. Applied to Qwen3-1.7B-Base, Looped Qwen improves over an identically continued dense baseline on every evaluated benchmark across two data regimes, with statistically clear gains on GSM8K, MATH, and MMLU-Pro on the curated mixture. Together, these results make looped language models substantially cheaper to train from scratch and practical to introduce into existing pretrained checkpoints, while isolating the gains due to recurrence itself.

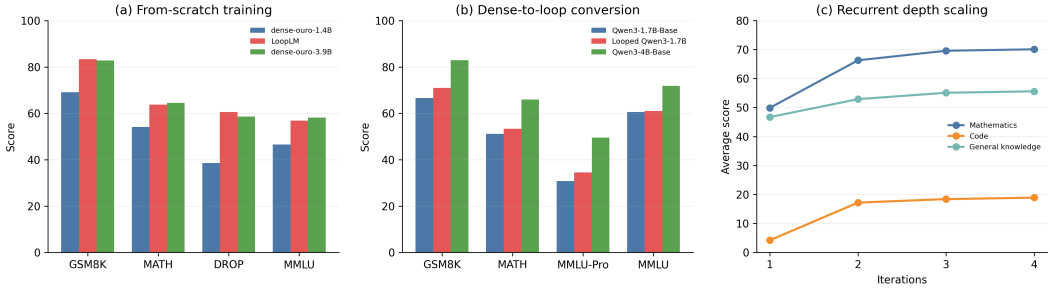


Figure 1: Looped language models trained from scratch or converted from a pretrained dense model, each compared with dense models trained on the same data. (a) A 1.4B LoopLM (ours) trained from scratch on 310B tokens beats its parameter-matched dense twin on every benchmark and stays close to its inference-compute-matched 3.9B twin on math and reading comprehension; LoopLM is shown at four iterations, the 3.9B twin matches three. (b) Looped Qwen, converted from Qwen3-1.7B-Base with one learned scalar and a smoothed exit loss, improves on that checkpoint given the same continued training. (c) Mean score per group at each recurrent step: mathematics (GSM8K, MATH, MATH500), code execution (CRUXEval) and knowledge and commonsense (the six multiple-choice benchmarks of Table 2).

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.

1 INTRODUCTION

Transformers spend the same computation on every token. Looped language models apply a shared transformer block repeatedly, which raises effective depth without adding parameters, and with adaptive exits they can spend extra computation only where it is needed (Dehghani et al., 2019; Zhu et al., 2025).

Ouro showed that looped models can be pretrained at scale and compete with much larger dense models (Zhu et al., 2025). Its recipe runs through five stages and 7.7T tokens, it starts from scratch while pretrained dense checkpoints go unused, and its gains are measured against dense models trained on other data, so the share that comes from recurrence is unknown.

We address all three points: a simpler and much cheaper from-scratch recipe, a conversion of dense checkpoints into recurrent models with minimal architectural changes, and a comparison against dense counterparts trained on the same data with the same training stages, matched in parameters or in inference compute.

Our recipe has two stages where Ouro has five, one pretraining run and one high-quality mid-training stage, and trains on 310B tokens against Ouro's 7.7T. At this budget the model is already competitive on reasoning benchmarks, while the 25 times longer training of Ouro-1.4B still shows on knowledge benchmarks (Table 3). Without the intermediate stages the exit gate can collapse early in training; a learning-rate warmup removes this early collapse, and a larger KL coefficient lets the gate recover if it does collapse. The resulting 1.4B LoopLM beats its parameter-matched dense twin on all 12 evaluated benchmarks, with the largest gains on DROP, CRUXEval-I and GSM8K, and at three recurrent steps, the inference cost of a 3.9B dense twin, it stays competitive with that model on math and reading comprehension while trailing it on knowledge and commonsense benchmarks.

Applying the Qwen3-1.7B-Base decoder (Yang et al., 2025) recurrently without further changes fails: each later step makes the representation worse. One learned scalar that mixes the original token embedding into every later step, with a smoothed exit loss that keeps every step training, is enough, and step-specific LoRA adapters are optional. Without step-specific adapters, Looped Qwen is ahead on every benchmark of Qwen3-1.7B-Base continued on the same data for the same number of steps, in two independent runs. On the curated mixture its gains on GSM8K, MATH and MMLU-Pro exceed two standard errors. An inference-compute-matched Qwen3-4B-Base stays ahead, as expected from a checkpoint more than twice the size.

The main contributions are:

- A simplified from-scratch recipe: one pretraining run and one mid-training stage, 310B tokens against Ouro's 7.7T, with a learning-rate warmup that removes the early collapse of the exit gate and a larger KL coefficient that lets it recover.
- A dense-to-loop conversion with minimal architectural changes: one learned scalar and a smoothed exit loss, no step-specific parameters, and Looped Qwen, the converted Qwen3-1.7B-Base, improves on the source checkpoint given the same continued training.
- A controlled comparison in both regimes: every looped model is measured against dense counterparts that share its data, training stages and token budget, matched in parameters or in inference compute, for LoopLM and Looped Qwen alike. The large looped models so far report their headline numbers against published dense models.

2 RELATED WORK

Looped and recurrent language models. Universal Transformers introduced the construction behind most looped models, a shared block applied repeatedly with adaptive halting (Dehghani et al., 2019; Graves, 2016); fixed-point formulations replace the halting with a convergence test on the hidden state (Movahedi et al., 2026). Looped Transformers use the shared block to extend depth and length generalization (Saunshi et al., 2025; Fan et al., 2025), recurrent-depth models gain from more test-time iterations (Geiping et al., 2025), and Mixture-of-Recursions gives each token its own depth (Bae et al., 2025b). At scale, Ouro pretrains looped models that match much larger dense models (Zhu et al., 2025), HRM-Text applies hierarchical recurrence to open-domain text (Wang et al., 2026), and Loop the Loopies loops a mixture-of-experts model (Gao et al., 2026).

Training stability. Each of these works hits an instability of its own. Ouro saw loss spikes at eight recurrent steps and cut the depth to four and halved the KL coefficient; the first large run of Geiping et al. (2025) stalled with hidden states that became the same for every token; HRM-Text adds a normalization scheme and warms up its backward horizon; Mixture-of-Recursions needs balancing losses to keep its router alive. None gives a from-scratch recipe without a multi-stage schedule or names the ingredient that keeps the exit gate alive once that schedule is gone (Section 3).

Comparing looped and dense models. Comparisons against dense models trained on the same data exist: Saunshi et al. (2025) and Mixture-of-Recursions train parameter-matched and compute-matched baselines, HRM-Text matches training FLOPs, Loop the Loopies matches pretraining cost, and Ouro's own small-scale study finds a deeper dense model ahead of the looped one at matched forward FLOPs. Ouro and the recurrent-depth model report their headline numbers against published dense models trained on other data, so the share of the gain that comes from recurrence stays open. We measure the model we report against dense twins in both regimes, and do the same for Looped Qwen, our converted model, where prior work matched parameters only.

Converting pretrained models. Relaxed Recursive Transformers tie the layers of a pretrained model into looping blocks with one LoRA adapter per iteration and distill from the original (Bae et al., 2025a); Think-at-Hard starts from Qwen3-Base and adds depth-specific adapters and a router for hard tokens (Fu et al., 2025). Both rely on step-specific adapters and compare against parameter-matched baselines only. We show that looping the pretrained decoder as it is degrades the representation, that one learned scalar stops this degradation and that adapters are optional.

Reinforcement learning. GRPO and DAPO (Shao et al., 2024; Yu et al., 2025) assume a fixed computation path; RLTT distributes credit across recurrent iterations (Williams & Tureci, 2026). Ouro reports no gain from RL on its final model. A separate study asks when RL adds to a looped model (Appendix C).

3 A SIMPLER FROM-SCRATCH RECIPE

3.1 OURO'S STAGED TRAINING

Ouro (Zhu et al., 2025) trains LoopLMs through five stages: pretraining at eight recurrent steps, continued pretraining at four, high-quality annealing, context extension, and supervised mid-training. Its exit gate is optimized jointly with the language model using an entropy-regularized ELBO objective with KL coefficient β.

We simplify this recipe by collapsing the pretraining stages into a single run, reducing the total token budget, and training the dense baselines of Section 5.1 through the same stages on the same data.

3.2 A SINGLE PRETRAINING STAGE

Ouro's pretraining uses three regimes before fine-tuning: pretraining at $T{=}8$, continued pretraining at $T{=}4$, and annealing on a higher-quality mixture (Zhu et al., 2025). We replace all three with one run from random initialization on a fixed mixture dominated by filtered web text, with mathematics and instruction-following data (Table 12).

As in Loop the Loopies (Gao et al., 2026), high-quality data is present from the start. We keep only one mid-training stage afterwards (Section 3.4).

3.3 DEPTH AND STABILITY

Our single run departs from Ouro's first-stage configuration in depth and in schedule.

Ouro explores $T{=}8$ in its first stage and drops to four afterwards, citing loss spikes at the larger depth (Zhu et al., 2025). Our own $T{=}8$ run under that configuration points the same way: the per-loop loss stops improving after the third or fourth iteration, and the gate spreads its mass evenly over loops four to eight (Figure 2). The extra iterations cost compute without measurable gain, so depth is fixed at $T{=}4$ from the start.

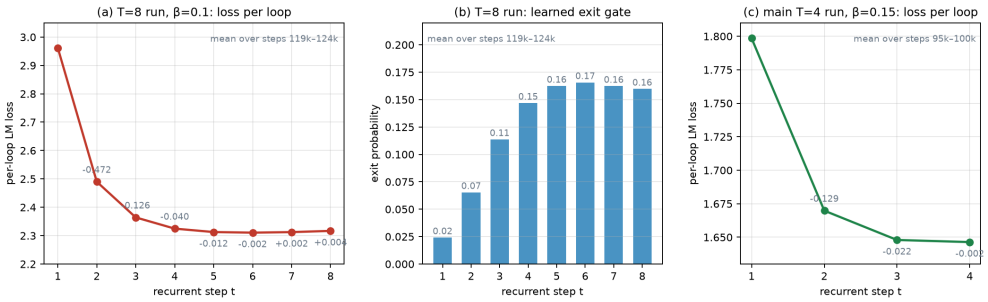


Figure 2: Why T=4. (a) Per-loop LM loss of a T=8 run (β=0.1, 124k steps) and the gain of each loop over the previous one: loops five to eight change the loss by about 0.01 nats or less. (b) Exit probability per loop in the same run; the gate spreads its mass evenly over loops four to eight. (c) The same for the main T=4 run. Values are means over the last 5k steps.

Ouro also lowers β after its first stage and trains with a constant learning rate from then on. Our run has no such transition, so we ask what keeps the exit gate alive once the schedule is gone. The failure is a gate collapse: early in training the gate learns to exit after the first loop, later iterations stop receiving gradient, and the model trains on as a plain transformer. Figure 3 compares β=0.05, the value Ouro uses after its first stage, with our β=0.15, each trained without warmup and with our warmup-plus-cosine schedule. Without warmup the smaller β collapses for good and the larger one comes close before it recovers. With the warmup schedule the larger β never comes near the collapsed regime, while the smaller one stays near $p_0 = 0.9$ for its first thousand steps before recovering. Thus, warmup prevents early collapse at our β, while larger β enables recovery if collapse occurs: without warmup every $\beta \geq 0.10$ we tried recovers, and only β=0.05 stays collapsed (Appendix B). The main run therefore uses β=0.15 with warmup and cosine decay (Table 1). Notably, the language-modelling loss does not reveal the collapse while it occurs, so the exit distribution must be monitored directly.

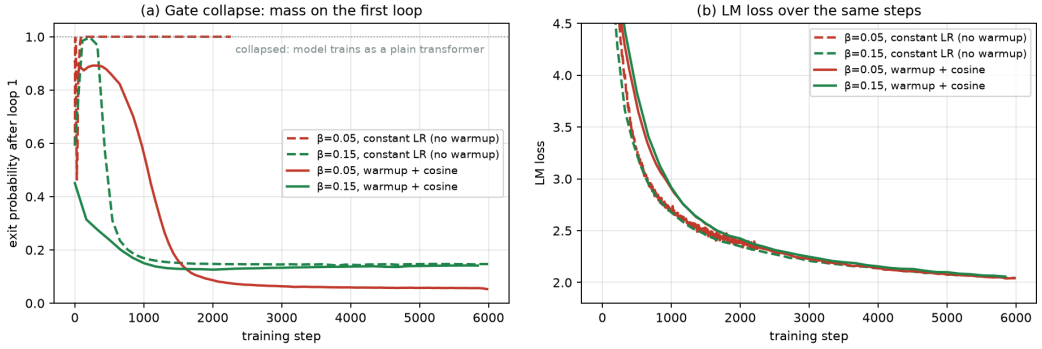


Figure 3: Gate collapse and what prevents it. (a) Probability of exiting after the first loop over the first 6k steps for $\beta \in \{0.05, 0.15\}$, with a 1k-step warmup and with Ouro's constant schedule; the constant-schedule β=0.15 run is separate from the cosine-schedule run of Appendix B. (b) LM loss over the same steps: the collapsed run is indistinguishable by loss alone.

3.4 High-quality mid-training

Despite the mathematics and instruction-following data in pretraining, the resulting checkpoint reaches only 35% accuracy on 5-shot GSM8K (Appendix C). We therefore add a high-quality mid-training stage using instruction–response pairs with loss on response tokens only. The mixture emphasizes mathematics and reasoning while retaining a knowledge-preserving component.

Mid-training continues for 100B tokens on the HRM-Text mixture (Wang et al., 2026), whose largest components are FLAN, SYNTH, and OpenMathInstruct-2 (Table 13). Recurrent depth, β, weight decay, optimizer, and architecture remain unchanged from pretraining. The same stage is applied to both dense baselines with identical data, schedule, and token budget.

3.5 Our improved setup

The final recipe keeps Ouro's shared recurrent architecture and trains it in two stages instead of five: a single 210B-token pretraining run from random initialization followed by 100B tokens of high-quality mid-training. The resulting 310B-token budget is 25× smaller than the 7.7T tokens used for Ouro-1.4B and needs about 3% of the training compute of Ouro's schedule on the same architecture (Appendix I). Training uses $T{=}4$ from the start, a learning-rate warmup to prevent early exit-gate collapse, and $\beta{=}0.15$ to enable recovery if collapse occurs (Section 3.3). Table 1 summarizes the recipe; Appendix D provides the architecture, gate objective, and data mixtures.

Table 1: Our two-stage training recipe. Both stages share the architecture, recurrent depth, and optimizer; the main difference is the data and learning-rate schedule.

	Pretraining	**Mid-training**
Initialization	From scratch	Pretraining checkpoint
Peak learning rate	3×10^{-4}	1×10^{-4}
Final learning rate	3×10^{-5}	1×10^{-5}
LR scheduler	Warmup + cosine decay	
LR warmup steps	1,000	
Weight decay	0.1	
Gradient norm clip	1.0	
Optimizer	AdamW ($\beta_1{=}0.9,\ \beta_2{=}0.95$)	
Sequence length	4K	
Training steps	100,000	47,684
Training tokens	210B	100B
Loss	Next token	Response tokens only
Recurrent steps (T)	4	
KL coefficient β	0.15	

The mid-training stage contributes most of the reasoning improvement: relative to the step-100,000 pretraining checkpoint, it more than doubles GSM8K accuracy and improves MMLU by about nine points.

3.6 Reinforcement learning as a separate study

None of the models compared in Section 5 uses reinforcement learning; we study it separately, since Ouro reported no gain from RL with verifiable rewards on its final checkpoint (Zhu et al., 2025). We use RLTT (Williams & Tureci, 2026), a GRPO variant for looped models that distributes policy gradients across recurrent iterations while keeping rollout and training execution paths consistent. With group-based methods a prompt contributes gradient only when its rollouts contain both successes and failures. A mid-trained checkpoint of an earlier run of this recipe already solved about three quarters of its MATH rollouts, and RLTT left MATH unchanged; the step-100,000 pretraining checkpoint of the main run solved about a third of its GSM8K rollouts, and the same method added twelve points. What separates the two runs is the solve rate at rollout time. Appendix C reports these two runs and their training dynamics, and summarizes four runs at a lower learning rate that did not move the policy.

4 Looping pretrained LLMs

Only a few pretrained looped checkpoints are publicly available, making dense-to-loop conversion an attractive alternative to training from scratch. We find that naive recurrence degrades later steps, while a single input-preserving scalar together with a smoothed exit loss is sufficient to convert Qwen3-1.7B-Base, at under 1% of its pretraining compute (Appendix I). Appendix G reports the corresponding training dynamics.

4.1 CONVERSION RECIPE

We start from Qwen3-1.7B-Base (Yang et al., 2025) and apply the full decoder recurrently for $T{=}3$ steps. Without input re-injection, later steps are consistently worse than the first: after the initial phase, the LM losses of steps 2 and 3 remain about 0.01 and 0.02 nats higher, respectively (Figure 9a). A learned per-step projection of the original embedding and recurrent state also fails to produce refinement: the gate abandons the middle step and the final step does not improve over the first (Figure 9c).

We instead preserve direct access to the original embedding with a single learned scalar. Let e denote the token embedding and h_{t-1} the output of the previous recurrent step. For $t > 0$,

$$z_t = we + (1 - w)h_{t-1}, \qquad w = \sigma(a), \tag{1}$$

where a is the only additional parameter. With the same data, batch, and seed as the naive run, steps 2 and 3 finish about 0.02 nats below step 1 (Figure 9b). We initialize $w = 0.99$; it increases slowly throughout training in all runs (Figure 9e; Appendix F).

4.2 TRAINING STABILITY

The exit gate introduces a separate instability. Under the standard LoopLM objective (Zhu et al., 2025), $\sum_t p_t \mathcal{L}_t$, each step is trained in proportion to its current exit probability; once the gate assigns little mass to a step, that step receives little learning signal. In an unsmoothed run with rank-8 stepwise LoRA, the gate drops the middle exit and its loss rises to about 10 nats (Figure 9d).

We therefore smooth only the exit distribution used for the LM loss:

$$\mathcal{L}_{\mathrm{LM}}^{\mathrm{smooth}} = (1 - \epsilon) \sum_{t=0}^{T-1} p_t \mathcal{L}_t + \frac{\epsilon}{T} \sum_{t=0}^{T-1} \mathcal{L}_t. \tag{2}$$

Gate regularization and inference still use the original exit probabilities. With $\epsilon{=}0.05$ and $T{=}3$, every step retains at least 0.0167 LM weight, preventing the later-step losses from drifting (Figure 9d).

4.3 DATA MIXES

We use a curated continued-training mixture covering mathematics, code, STEM, science, and instruction following, with final-answer supervision and no explicit chain-of-thought traces (Appendix F). To separate recurrence gains from data effects, we repeat the conversion on a second mixture dominated by filtered web text. Each mixture is shared unchanged between the looped model and its dense baselines.

4.4 STEPWISE LORA

We test whether step-specific parameters are necessary by adding independent rank-32 LoRA adapters to each recurrent step (Bae et al., 2025a). They provide little benefit: training losses nearly match the shared model (Figure 9f), and benchmark differences stay within two points with neither variant consistently ahead (Section 5.3). We therefore use the fully shared model by default.

4.5 FINAL ARCHITECTURE

Figure 4 summarizes the final model: a shared pretrained decoder, scalar input injection from Eq. 1, and a shared LM head and exit gate trained with Eq. 2. Because the third step improves training loss by less than 0.001 nats over the second on the instruction mixture (Figure 9f), evaluation uses a fixed exit after two steps.

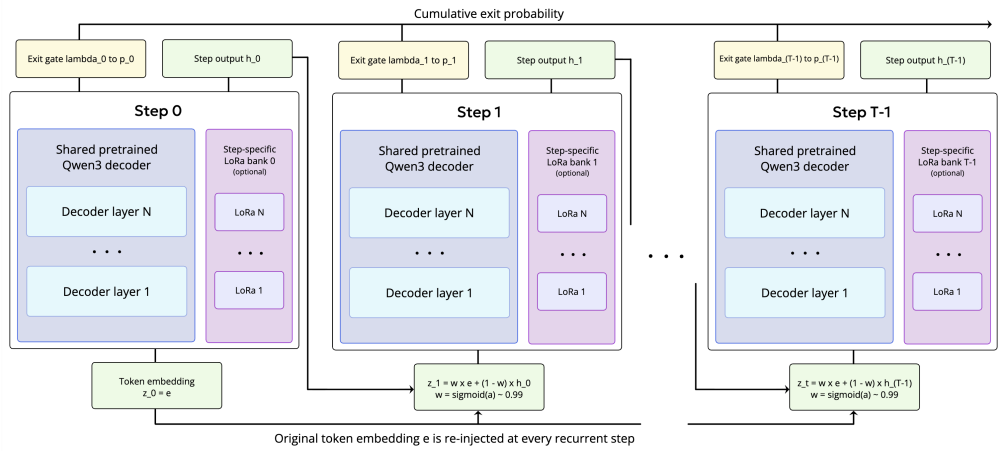


Figure 4: Final Looped Qwen architecture. The pretrained decoder is shared across steps; optional LoRA adapters are step-specific. Later steps mix the original embeddings with the preceding hidden state, while a shared exit gate accumulates stopping probability.

5 EXPERIMENTS

5.1 BASELINES

We evaluate both from-scratch and converted looped models against dense baselines trained on the same data, stages, and token budget. For each track, we use a parameter-matched baseline and a larger inference-compute-matched baseline.

5.1.1 FROM-SCRATCH LOOPLM

Both dense baselines remove recurrence (T=1) while retaining the same transformer block and two-stage training procedure. `dense-ouro-1.4B` uses one 24-layer block and matches LoopLM in parameter count; `dense-ouro-3.9B` stacks three such blocks into 72 layers and matches the inference compute of LoopLM at three recurrent steps. Because LoopLM trains with all four iterations, its per-token training compute is about $4\times$ that of `dense-ouro-1.4B` and $1.4\times$ that of `dense-ouro-3.9B` (Appendix I).

5.1.2 LOOPED DENSE

For conversion, Qwen3-1.7B-Base is the parameter-matched baseline and Qwen3-4B-Base the inference-compute-matched reference (Yang et al., 2025). All models are continued on the same conversion mixture with identical optimization and evaluation settings. At its two-step exit, Looped Qwen uses about 20% fewer inference FLOPs than Qwen3-4B-Base (Appendix I).

5.2 LOOPLM RESULTS

5.2.1 AGAINST DENSE MODELS TRAINED ON THE SAME DATA

Table 2 compares LoopLM with both controlled dense baselines.

At equal parameter count, four-step LoopLM outperforms `dense-ouro-1.4B` on all 12 benchmarks. The gains are largest on reasoning-intensive tasks: +22 points on DROP, +17 on CRUXEval-I, +14 on GSM8K, and about +10 on MATH, MMLU, and CRUXEval-O. Thus, recurrence provides a consistent advantage over applying the same parameter budget only once.

At equal inference compute the larger dense model is ahead or level on every benchmark, and the advantage of LoopLM at this point is its size. With 1.4B parameters, 36% of those of `dense-ouro-3.9B`, three-step LoopLM comes within one standard error of the difference on GSM8K, MATH and MATH500 and within about one point on DROP. A fourth iteration, roughly

Table 2: Benchmark scores after mid-training (Section 3.4). LoopLM is evaluated at a fixed exit after k iterations; `dense-ouro-3.9B` matches the inference compute of three iterations.

Benchmark	dense-ouro-1.4B	LoopLM (ours)				dense-ouro-3.9B
		1 iter.	2 iter.	3 iter.	4 iter.	
Math and reading comprehension						
GSM8K	69.14 ± 1.27	60.65 ± 1.35	76.50 ± 1.17	81.58 ± 1.07	$\mathbf{83.32 \pm 1.03}$	82.79 ± 1.04
MATH	54.10 ± 0.67	44.18 ± 0.67	60.90 ± 0.65	63.64 ± 0.64	63.80 ± 0.64	$\mathbf{64.54 \pm 0.64}$
MATH500	55.00 ± 2.22	45.00 ± 2.22	61.40 ± 2.18	$\mathbf{63.60 \pm 2.15}$	63.20 ± 2.16	$\mathbf{63.60 \pm 2.15}$
DROP (F1)	38.59 ± 0.48	22.20 ± 0.41	45.02 ± 0.49	57.40 ± 0.49	$\mathbf{60.57 \pm 0.48}$	58.57 ± 0.49
Knowledge and commonsense						
MMLU	46.52 ± 0.41	40.77 ± 0.41	52.61 ± 0.41	55.93 ± 0.40	56.84 ± 0.40	$\mathbf{58.15 \pm 0.40}$
MMLU-Pro	17.48 ± 0.34	13.80 ± 0.31	20.53 ± 0.36	24.27 ± 0.38	24.53 ± 0.38	$\mathbf{28.22 \pm 0.40}$
ARC-Challenge	46.16 ± 1.46	45.99 ± 1.46	50.60 ± 1.46	50.51 ± 1.46	51.02 ± 1.46	$\mathbf{55.03 \pm 1.45}$
ARC-Easy	74.75 ± 0.89	72.18 ± 0.92	76.47 ± 0.87	77.78 ± 0.85	77.99 ± 0.85	$\mathbf{79.67 \pm 0.83}$
HellaSwag	57.28 ± 0.49	51.02 ± 0.50	56.94 ± 0.49	58.92 ± 0.49	59.29 ± 0.49	$\mathbf{67.91 \pm 0.47}$
Winogrande	59.43 ± 1.38	56.67 ± 1.39	60.22 ± 1.38	62.90 ± 1.36	63.93 ± 1.35	$\mathbf{65.19 \pm 1.34}$
Code execution						
CRUXEval-O	6.15 ± 0.76	6.03 ± 0.74	18.20 ± 1.28	17.36 ± 1.23	16.60 ± 1.22	$\mathbf{18.48 \pm 1.28}$
CRUXEval-I	4.61 ± 0.60	2.34 ± 0.45	16.20 ± 1.15	19.46 ± 1.22	21.21 ± 1.29	$\mathbf{21.80 \pm 1.35}$

one third more inference compute, moves it ahead on DROP and level on GSM8K. The gap that remains is in knowledge and commonsense benchmarks, 2 to 9 points, where the extra parameters of the dense model store more facts.

The depth sweep shows that these gains come from using recurrent computation rather than from the underlying 1.4B checkpoint alone. One iteration trails the parameter-matched dense model on every benchmark; two iterations outperform it on 11 of 12, three on all 12, and the fourth iteration improves 10 of 12 scores further.

5.2.2 Against published models

Because published models differ substantially in evaluation protocol and pretraining budget, Table 3 is intended as contextual rather than controlled comparison.

Table 3: LoopLM after mid-training against published models between 1B and 4B parameters. Bold marks the best score in each column; underline marks the second best.

Model	Tokens	GSM8K	MATH	MATH500	DROP	MMLU	MMLU-Pro	ARC-C	ARC-E	HellaSwag	Winogrande
LoopLM (ours), 4 iter.	**0.31**	<u>83.3</u>	<u>63.8</u>	63.2	**60.6**	56.8	24.5	51.0	78.0	59.3	63.9
LoopLM (ours), 3 iter.	**0.31**	81.6	63.6	63.6	57.4	55.9	24.3	50.5	77.8	58.9	62.9
Ouro-1.4B	7.7	78.9	**70.9**	**82.4**	49.7	<u>67.4</u>	<u>48.6</u>	60.9	<u>84.0</u>	74.3	**72.3**
Huginn-3.5B	0.8	32.6	12.6	13.2	17.8	31.4	—	38.2	69.9	65.2	59.4
Gemma-3-1B	2	2.1	3.7	41.0	42.4	39.9	11.3	38.4	73.0	62.3	58.2
Llama-3.2-1B	9	7.1	3.3	7.4	28.0	32.2	11.8	32.8	—	59.4	62.8
Qwen2.5-1.5B	18	68.5	35.0	17.6	36.4	60.9	28.5	54.7	81.2	67.9	65.0
Qwen3-1.7B	36	75.4	43.5	25.8	34.3	62.6	36.8	55.7	81.9	67.1	66.3
Qwen2.5-3B	18	79.1	42.6	42.6	36.5	65.6	34.6	56.5	83.3	74.6	71.1
Llama-3.2-3B	9	67.2	7.4	40.8	45.2	58.0	33.3	**69.1**	—	73.1	69.1
Gemma-3-4B	4	38.4	24.2	<u>68.6</u>	<u>60.1</u>	59.6	29.2	56.2	82.4	**77.2**	64.7
Qwen3-4B	36	**87.8**	54.1	59.6	45.5	**73.0**	**50.6**	<u>63.7</u>	**86.4**	<u>75.7</u>	<u>71.2</u>

Despite using only 0.31T training tokens, LoopLM remains competitive with models trained on substantially larger budgets. The four-step model achieves the highest DROP score in the table and the second-highest GSM8K score. Against Ouro-1.4B, trained on 25$\times$ more tokens, LoopLM is lower on MATH and MATH500 but higher by 4.4 points on GSM8K and 10.9 points on DROP. The controlled evidence for recurrence itself remains the comparison in Table 2.

5.3 LOOPED DENSE RESULTS

We run the conversion twice from Qwen3-1.7B-Base using the same recipe: once on 84B tokens of web-scale data and once on 60B tokens of curated instruction data (Section 4.3). Each run has dense baselines continued on the same data for the same number of steps. Table 4 reports the instruction run; the web-scale run is in Appendix H. Looped Qwen trains with all three recurrent steps and is evaluated after two, where its inference cost is close to Qwen3-4B-Base.

Table 4: Conversion on the curated instruction mixture, 60B tokens, evaluated at a fixed exit after two of three recurrent steps. $\pm$ is the standard error; † denotes continued training on the same mixture for the same number of steps; bold marks Looped Qwen ahead of Qwen3-1.7B-Base†.

Benchmark	Qwen3-1.7B-Base†	Looped Qwen, shared	Looped Qwen, LoRA-32	Qwen3-4B-Base†
GSM8K	66.64 ± 1.30	$\mathbf{70.96 \pm 1.25}$	$\mathbf{69.90 \pm 1.26}$	82.87 ± 1.04
MATH	51.14 ± 0.66	$\mathbf{53.34 \pm 0.66}$	$\mathbf{54.46 \pm 0.67}$	65.92 ± 0.63
MMLU	60.62 ± 0.39	$\mathbf{61.05 \pm 0.39}$	$\mathbf{60.99 \pm 0.39}$	71.83 ± 0.36
MMLU-Pro	30.76 ± 0.41	$\mathbf{34.52 \pm 0.42}$	$\mathbf{36.24 \pm 0.43}$	49.52 ± 0.44
ARC-Challenge	53.84 ± 1.46	$\mathbf{55.46 \pm 1.45}$	$\mathbf{54.95 \pm 1.45}$	64.42 ± 1.40
ARC-Easy	79.97 ± 0.82	$\mathbf{80.13 \pm 0.82}$	$\mathbf{80.51 \pm 0.81}$	86.32 ± 0.71
HellaSwag	63.35 ± 0.48	$\mathbf{64.13 \pm 0.48}$	$\mathbf{64.29 \pm 0.48}$	70.88 ± 0.45
Winogrande	62.90 ± 1.36	$\mathbf{63.61 \pm 1.35}$	$\mathbf{64.01 \pm 1.35}$	67.80 ± 1.31

The conversion result is consistent across both data regimes. Without step-specific adapters, Looped Qwen improves over Qwen3-1.7B-Base continued on exactly the same data for the same number of steps on every benchmark in both runs. Most gains are modest, but the curated instruction mixture produces clear reasoning improvements: +4.3 points on GSM8K, +3.8 on MMLU-Pro, and +2.2 on MATH, each exceeding two standard errors of the difference. On the web-scale mixture, gains remain within about two points, with only MMLU and HellaSwag exceeding that threshold.

The larger gains on the curated mixture suggest that the benefit of recurrence depends strongly on the continued-training data. Importantly, the effect does not require step-specific capacity: the shared and LoRA-32 variants differ by at most two points, with neither consistently ahead.

Looped Qwen still trails Qwen3-4B-Base on every benchmark. The conversion therefore does not replace the knowledge and capacity obtained from substantially larger pretraining; instead, it provides a lightweight way to improve an existing 1.7B checkpoint without repeating pretraining or adding per-step parameters.

6 CONCLUSION

This work gives two practical ways to build a looped language model: a short recipe for training one from scratch, and a conversion of a pretrained dense model with minimal architectural changes.

For from-scratch training, a two-stage recipe trains a looped model at a fraction of the token budget of earlier multi-stage pipelines, a learning-rate warmup removes the early collapse of the exit gate, and a larger KL coefficient lets the gate recover if it does collapse. The resulting LoopLM improves over a parameter-matched dense model on every evaluated benchmark and, at equal inference compute, comes within about a point of a dense model with three times its parameters on math and reading comprehension; the larger model keeps its lead where parameter count stores facts.

For pretrained models, applying a dense decoder repeatedly does not produce refinement. One learned scalar that mixes the token embedding into each recurrent step, together with a smoothed exit loss, is enough to turn Qwen3-1.7B-Base into Looped Qwen, which improves on the source checkpoint given the same continued training, with no step-specific parameters. At the inference cost of Qwen3-4B-Base it still trails that model, as a checkpoint more than twice its size should.

Both results are measured against dense baselines trained on the same data with the same training stages, matched in parameters or in inference compute.

AI USE STATEMENT

Generative AI tools were used to assist with drafting and polishing the manuscript, retrieval and discovery of relevant literature, and refactoring author-written code. In particular, AI assistance was used to improve wording, clarity, and organization of the text, to help identify potentially relevant prior work, and to restructure, simplify, and improve the readability of existing implementation code without changing the intended algorithms or experimental methodology. All retrieved references were independently checked by the authors against the original sources, and all AI-assisted code changes were reviewed and tested by the authors before use.

Generative AI was not used to generate synthetic data, formulate the scientific hypotheses or methodology, derive theoretical claims or proofs, conduct data cleaning or analysis, interpret experimental results, or translate the manuscript. It was not used to generate experimental results or make scientific conclusions on behalf of the authors.

All AI-assisted text and code were reviewed, edited, and verified by the authors. Numerical results, experimental claims, citations, and descriptions of prior work were checked against the corresponding experiments, source materials, and original publications. The authors take full responsibility for the accuracy, integrity, and final content of the paper.

ETHICS STATEMENT

We adhere to the ICLR Code of Ethics. This work studies the architecture, training, and evaluation of language models and does not involve human-subject experiments or the collection of new personal or sensitive data. Training uses existing language-model pretraining and instruction-tuning datasets; we do not release additional user data or construct datasets containing newly collected personal information.

The models studied in this work are general-purpose language models and therefore inherit the usual potential risks associated with such systems, including generation of inaccurate, biased, or harmful content and possible dual-use applications. Our work does not introduce mechanisms specifically designed for harmful use, nor does it evaluate or optimize models for such applications. The proposed methods primarily change how computation is reused across recurrent steps and do not remove the need for the same safety, fairness, privacy, and deployment considerations that apply to conventional language models.

To the best of our knowledge, this work raises no additional legal, privacy, or fairness concerns beyond those associated with the underlying public pretraining and instruction-tuning corpora and pretrained checkpoints. Any funding sources and potential conflicts of interest will be disclosed in the camera-ready version as required by the conference.

REPRODUCIBILITY STATEMENT

The from-scratch LoopLM is specified in Table 1 and Appendix D: architecture, recurrent objective, optimizer, schedules and both data mixtures. The Looped Qwen conversion is specified in Section 4 and Appendix F: architecture, recurrent modules, optimizer, seeds and the curated data mixture; the web-scale run is reported in Appendix H.

For every controlled comparison, the looped model and its dense baselines draw on the same data sources and use the same training stages, token budget and evaluation procedure (Section 5.1). The depth study (Section 3.3) and the shared-versus-LoRA comparison (Section 5.3) are in the main text; the exit-gate stability sweep, the normalization ablation and the reinforcement-learning runs are in Appendices B, A and C.

Anonymized training code for both tracks, with one-GPU smoke tests on synthetic data, is provided in the supplementary material.

REFERENCES

Sangmin Bae, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Seungyeon Kim, and Tal Schuster. Relaxed recursive transformers: Effective parameter sharing with layer-wise LoRA. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=WwpYSOkkCt.

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, and Se-Young Yun. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. In *Advances in Neural Information Processing Systems*, 2025b. URL https://arxiv.org/abs/2507.10524.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *Proceedings of the 7th International Conference on Learning Representations*, 2019. URL https://arxiv.org/abs/1807.03819.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. In *Proceedings of the 13th International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2409.15647.

Tianyu Fu, Yichen You, Zekai Chen, Guohao Dai, Huazhong Yang, and Yu Wang. Think-at-hard: Dynamic looped transformers for improved reasoning, 2025. URL https://arxiv.org/abs/2511.08577.

Zitian Gao, Yilong Chen, Yihao Xiao, Xinyu Yang, Ran Tao, Joey Zhou, and Bryan Dai. Loop the loopies!, 2026. URL https://arxiv.org/abs/2607.16051.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2502.05171.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016. URL https://arxiv.org/abs/1603.08983.

Sajad Movahedi, Vera Milovanović, Shlomo Libo Feigin, Alexander Theus, Thomas Hofmann, Valentina Boeva, T. Konstantin Rusch, and Antonio Orvieto. Fixed-point reasoners: Stable and adaptive deep looped transformers, 2026. URL https://arxiv.org/abs/2606.18206. ICML 2026 Workshop.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *Proceedings of the 13th International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Pr8o5llJ1O.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Guan Wang, Changling Liu, Chenyu Wang, Cai Zhou, Yuhao Sun, Yifei Wu, Shuai Zhen, Luca Scimeca, and Yasin Abbasi Yadkori. Hrm-text: Efficient pretraining beyond scaling, 2026. URL https://arxiv.org/abs/2605.20613.

Jonathan Williams and Esin Tureci. Prioritize the process, not just the outcome: Rewarding latent thought trajectories improves reasoning in looped language models. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2602.10520.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin

Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Qiying Yu et al. DAPO: An open-source LLM reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, Lu Li, Jiajun Shi, Kaijing Ma, Shanda Li, Taylor Kergan, Andrew Smith, Xingwei Qu, Mude Hui, Bohong Wu, Qiyang Min, Hongzhi Huang, Xun Zhou, Wei Ye, Jiaheng Liu, Jian Yang, Yunfeng Shi, Chenghua Lin, Enduo Zhao, Tianle Cai, Ge Zhang, Wenhao Huang, Yoshua Bengio, and Jason Eshraghian. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025. URL https://arxiv.org/abs/2510.25741.

A LAYER-NORMALIZATION ABLATION

Weight sharing makes normalization unusually important in recurrent models: the same decoder block is traversed up to T times, and its parameters receive gradients accumulated across recurrent iterations. Prior work has likewise identified normalization as an important stability choice in deeply recurrent Transformers: HRM-Text uses a parameter-free pre-norm variant, MagicNorm (Wang et al., 2026), while recurrent-depth pretraining has shown strong sensitivity to normalization placement (Geiping et al., 2025).

We ablate two choices separately: (i) normalization inside the recurrent decoder block (*decoder norm*), and (ii) normalization of the hidden state before the shared LM head (*final norm*).

Unless stated otherwise, all runs use the architecture and optimizer of our main recipe, $T{=}4$, a 500K-token batch, and a matched budget of 19K steps.

NORMALIZATION SCHEMES

For the decoder block, we compare five variants. *Ouro-norm* is the configuration used by Zhu et al. (2025) and by our main recipe (Section 3): it normalizes the input and the output of each sub-layer and adds the normalized output to the residual stream. *Pre-norm* applies normalization before each attention or MLP sub-layer, followed by the residual addition. *Post-norm* normalizes after the residual addition. *Sandwich-norm* also normalizes the input of each sub-layer, but applies the second normalization after the residual addition, so the residual stream itself is normalized, following Geiping et al. (2025). Finally, *Magic-norm* is pre-norm with the learnable normalization scale removed, following Wang et al. (2026).

For the final hidden state, we test learnable RMSNorm (*RMS*), parameter-free RMSNorm (*FRMS*), and no normalization (*identity*).

Table 5: LM loss after 19K training steps with a 500K-token batch.

	Final normalization		
Decoder norm	**RMS**	**FRMS**	**Identity**
Ouro-norm (baseline)	1.98	–	2.11
Pre-norm	2.03	2.07	2.11
Magic-norm	–	2.04	2.07
Post-norm	–	–	7.37
Sandwich-norm	–	–	7.40

The result separates cleanly into two groups. Ouro-norm, pre-norm, and MagicNorm all train normally and finish in a narrow 1.98–2.11 range. In contrast, post-norm and sandwich-norm plateau near 7.4 and fail to train.

Figure 5 shows that the divergence appears early. Representative pre-norm-family runs converge along nearly identical trajectories, whereas post-norm and sandwich-norm plateau within roughly the first 1K steps and never recover.

WHY POST-NORM FAILS

To localize the failure, we log the parameter-gradient norm of every decoder layer separately for each recurrent iteration. Figure 6 shows the depth profile at the second iteration ($t{=}2$).

With pre-norm, the gradient decreases by only about $8\times$ from the first decoder layer to the last. With post-norm and sandwich-norm, however, layers 0–22 receive gradients that are 8–16 orders of magnitude smaller than the gradient of layer 23. Almost all learning is therefore concentrated in the final layer of the shared block. In effect, a nominally 24-layer recurrent block behaves during optimization much closer to a one-layer block.

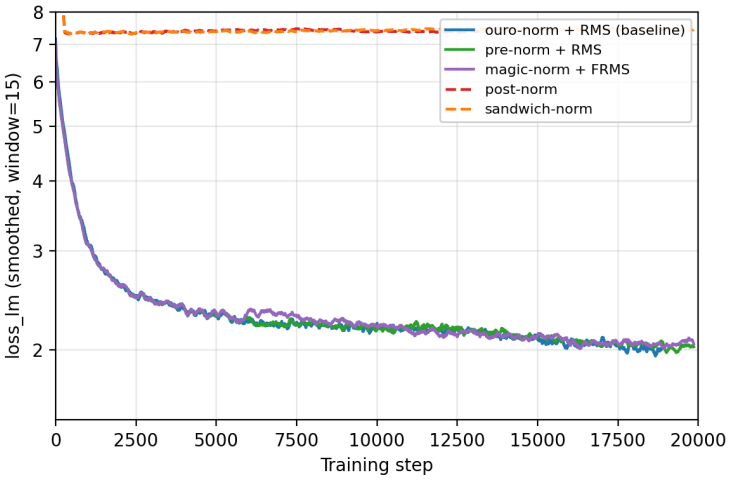


Figure 5: Training LM loss (15-step moving average) for representative pre-norm-family variants and for post-norm and sandwich-norm. The pre-norm-family curves nearly overlap, while post-norm and sandwich-norm plateau early.

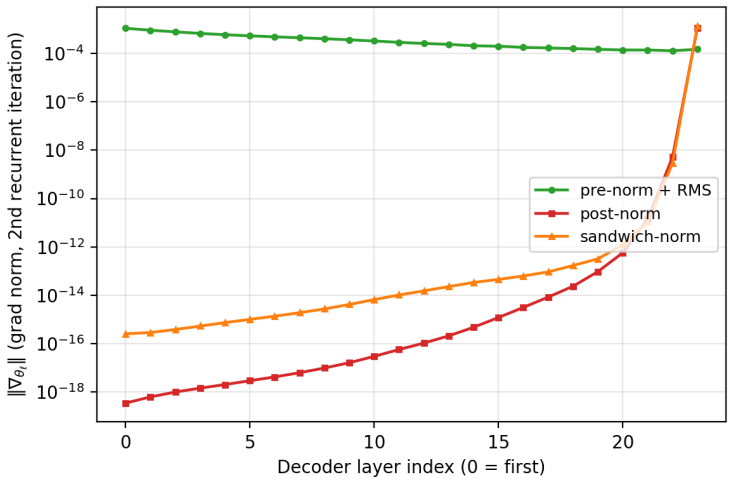


Figure 6: Gradient norm by decoder-layer depth at recurrent iteration t=2. Pre-norm preserves gradients throughout the block, whereas post-norm and sandwich-norm concentrate almost all gradient in the final decoder layer.

POST-NORM MITIGATION

We test whether the post-norm failure can be alleviated by changing the optimization schedule or shortening the recurrent path.

Increasing learning-rate warmup from 1K to 10K steps does not help: post-norm still converges to the same loss plateau near 7.

Reducing recurrent depth provides a partial workaround. Figure 7 compares post-norm at $T \in \{1, 2, 3, 4\}$ with a 10K-step warmup and otherwise identical settings. The T=1–3 runs converge to losses between 2.66 and 2.82. The T=4 run initially follows the same trajectory for roughly 2K steps, but then turns upward and returns to the ~ 7.4 plateau.

Thus, shortening the recurrent path avoids the most severe optimization failure, but does not recover the performance of the pre-norm family.

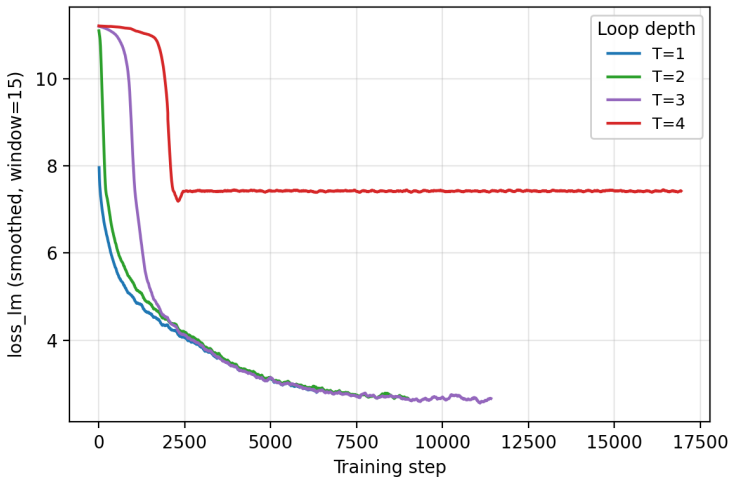


Figure 7: Post-norm training LM loss (15-step moving average) as a function of recurrent depth T, using a 500K-token batch and a 10K-step warmup. Lower recurrent depth mitigates the failure, but remains worse than the pre-norm-family configurations.

Final normalization

Once the decoder uses a stable pre-norm-family configuration, the exact choice of final normalization has only a small effect on mean loss, but a much larger effect on training stability.

Removing the final norm increases the matched-budget loss by 0.04–0.13 and introduces repeated loss spikes. We define a spike as a step-over-step loss increase greater than 0.5 above a 15-step rolling median, and count spikes over the first 25K training steps.

Table 6: Effect of final normalization on loss and training stability. Loss is measured at 19K steps; spikes and maximum gradient norm are measured over the first 25K steps.

Decoder norm	Final norm	Loss	#Spikes	Max grad norm
Ouro-norm	RMS	1.98	0	42.3
Ouro-norm	Identity	2.11	5	1.7×10^3
Pre-norm	RMS	2.03	0	15.7
Pre-norm	FRMS	2.07	0	28.4
Pre-norm	Identity	2.11	11	5.2×10^7
Magic-norm	FRMS	2.04	0	30.0
Magic-norm	Identity	2.07	10	1.3×10^6

All RMS/FRMS variants have zero spikes, whereas the identity variants produce 5–11. The difference is even clearer in gradient magnitude: with a final norm, the peak total gradient norm stays between 15.7 and 42.3; without one, it reaches as high as 5.2×10^7.

Whether the final RMSNorm has a learnable scale matters much less. For pre-norm, replacing RMS with FRMS changes the loss from 2.03 to 2.07, and both runs remain stable. The important choice is therefore whether a final normalization is present, not whether its scale is learned.

Robustness checks

Finally, we repeat the main normalization comparison at the 2M-token batch size used in our full training recipe. The conclusion does not change: post-norm, sandwich-norm, and a parameter-free post-norm variant all plateau at 7.40–7.41, closely matching the 500K-token experiments.

Table 7: No-warmup β sweep. Peak p_0 is the maximum first-step exit probability after the first 50 steps; late p_0 is its mean over steps 5K to 6K. Loss is the LM loss at step 2K, the latest step all four runs reached. The β=0.05 run is the collapsed run shown in Figure 3.

β	Schedule	Steps	Peak p_0	Late p_0	Loss at 2K	Outcome
0.05	Constant	2,267	1.00	1.00*	2.38	No recovery in 2.3K steps
0.10	Constant	54,558	1.00	0.11	2.36	Recovers
0.15	Cosine	24,579	0.99	0.14	2.39	Recovers
0.50	Constant	27,204	0.66	0.20	2.59	Never collapses

*Mean over the whole run, which was stopped at step 2,267.

Removing the learnable scale from post-norm therefore does not restore gradient flow. Together with the warmup and recurrent-depth experiments, this indicates that the failure is primarily caused by normalization placement rather than by batch size, warmup length, or the presence of learnable normalization parameters.

TAKEAWAY

Normalization placement around the shared recurrent block matters much more than the exact normalization formula. Post-norm and sandwich-norm fail because gradients vanish through almost the entire decoder block, and this failure persists across warmup lengths, batch sizes, and normalization parameterizations.

Within the stable pre-norm family, ouro-norm, standard pre-norm, and MagicNorm perform similarly. A final normalization before the LM head is important for stability, while learnable and parameter-free RMSNorm behave similarly.

Pre-norm+RMS is therefore a simple alternative to the ouro-norm configuration used in our main experiments, reaching a very similar final loss (2.03 vs. 1.98). We retain ouro-norm in the main recipe because the large-scale experiments had already been completed before this ablation.

B EARLY PRETRAINING COLLAPSE TO A DENSE REGIME

Section 3.3 shows that learning-rate warmup removes the early gate collapse. Here we examine the same failure mode more closely in the regime where it can actually occur: training without warmup. Rather than repeating the warmup $\times$ β comparison from Figure 3, we sweep β to characterize when an early collapse becomes permanent and when the model recovers.

All runs use the same 2M-token-batch configuration as the main recipe (Table 1). We train without warmup at $\beta \in \{0.05, 0.10, 0.50\}$ using a constant learning rate, and include one no-warmup cosine-schedule run at β=0.15, whose cosine spans a long budget, so its learning rate stays near the peak over the steps shown. We track the probability of exiting after the first recurrent step, p_0, which directly measures how close the model is to single-step dense training.

COLLAPSE IS USUALLY REVERSIBLE

The sweep shows that entering the dense regime does not by itself imply a training failure. Three of the four runs enter the collapsed regime early in training: at $\beta \in \{0.05, 0.10, 0.15\}$ the first-step exit probability reaches 0.99 to 1.00 within the first few hundred steps. At β=0.10 and β=0.15 the gate then returns to a recurrent regime, with late p_0 of 0.11 and 0.14; both had recovered by about step 1K. At β=0.05 it stays at p_0=1.00 for all 2,267 steps of the run. At β=0.50 the gate never enters the collapsed regime: after the first 50 steps p_0 peaks at 0.66. Without warmup, a collapse is therefore the usual start of training at moderate β, and β decides whether the gate recovers from it. At step 2K the collapsed run has the same LM loss as the recovered ones (2.38 against 2.36 and 2.39), so the loss alone does not reveal the collapse.

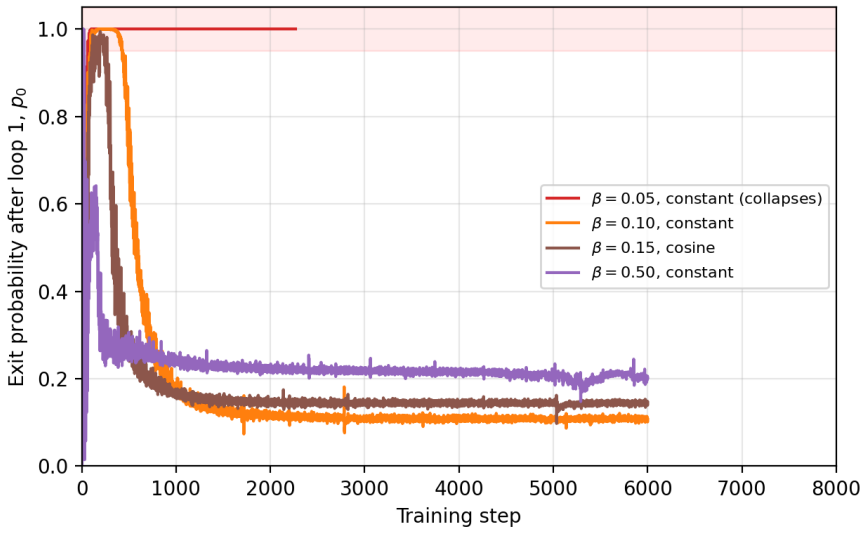


Figure 8: First-step exit probability p_0 during the first 8K training steps without warmup. The shaded region marks $p_0 > 0.95$. The β=0.05 run stays collapsed until it was stopped at step 2,267; β=0.10 and 0.15 recover, and β=0.50 never enters the shaded region after its first 50 steps.

The recovered exit distribution depends on β

The runs that train in the recurrent regime do not converge to the same exit distribution. Their late first-step exit probabilities are 0.11, 0.14, and 0.20 for $\beta \in \{0.10, 0.15, 0.50\}$, respectively.

This trend is consistent with the KL regularizer pulling the exit distribution toward the uniform T=4 prior, for which $p_0 = 0.25$. Larger β therefore both helps the gate recover from, or avoid, the early first-step collapse and places more probability on earlier exits after training has stabilized.

The role of warmup is different: at β=0.15 it keeps the gate out of the collapsed regime ($p_0 > 0.95$) from the start, and at β=0.05 it limits that regime to the first few dozen steps. Without warmup, β primarily determines whether the gate can recover once the collapse has occurred.

Gate collapse is distinct from normalization failure

This failure mode is different from the normalization failure analyzed in Appendix A. Here, the shared decoder block continues to receive gradients and train; what disappears is the contribution of later recurrent steps because the exit gate places nearly all probability on the first step.

Under post-norm and sandwich-norm, by contrast, the gate is not the primary problem. Gradients vanish through almost the entire decoder block, leaving essentially only its final layer trainable. That failure is persistent and is not repaired by changing β or extending warmup.

The two cases therefore correspond to different forms of effective model collapse: gate collapse reduces a recurrent model to single-step computation, temporarily or permanently, whereas normalization failure reduces a deep recurrent block to effectively single-layer optimization.

C When does RL help a looped model?

We ran RLTT on our 1.4B LoopLM in six configurations, varying the starting checkpoint, target task, and learning rate. All runs used temperature 0.9, groups of 8 rollouts, 32 prompts per optimizer step, and a fixed recurrent depth of T=4. Two runs used a learning rate of 10^{-5} and were evaluated with full benchmark sweeps; the remaining four used the RLTT reference learning rate of 10^{-6}.

Table 8: RLTT results at learning rate 10^{-5} after 140 optimizer steps. The reported uncertainty is the `lm-evaluation-harness` standard error and depends on the benchmark rather than the checkpoint. GSM8K in the MATH run is a guard metric, not an RL target.

Benchmark	**Base**	**Step 100**	**Step 140 (Δ)**
GSM8K target; pretraining checkpoint; solve rate ≈ 0.30			
GSM8K, 5-shot (target)	35.33 ± 1.32	46.78	47.31 ($+11.98$)
MMLU, 5-shot (guard)	47.59 ± 0.41	48.03	47.99 ($+0.40$)
HellaSwag, 10-shot (guard)	67.59 ± 0.47	68.08	68.09 ($+0.50$)
MATH target; mid-trained checkpoint; solve rate ≈ 0.77			
MATH, 0-shot (target)	55.68 ± 0.68	–	55.18 (-0.50)
GSM8K, 0-shot (guard)	73.46 ± 1.22	–	73.16 (-0.30)

Table 9: Training telemetry for the two 10^{-5} runs, averaged over the first and last 20 optimizer steps. KL is measured against the reference policy and therefore starts at zero by construction.

	GSM8K run	**MATH run**
KL, first 20 steps	0.013	< 0.001
KL, last 20 steps	0.084	0.003
Gradient norm, first 20 steps	0.58	0.12
Gradient norm, last 20 steps	0.64	0.11

RL HELPS ONLY WHEN THE ROLLOUT TASK IS NEITHER TOO EASY NOR TOO HARD

Only one of the six runs produced a clear downstream improvement. Starting from the pretraining checkpoint and optimizing GSM8K, RLTT improved 5-shot GSM8K from 35.3 to 47.3 in 140 steps, a gain of 12.0 points.

In contrast, optimizing MATH from a mid-trained checkpoint produced no measurable gain: MATH changed from 55.7 to 55.2, while the GSM8K guard metric changed from 73.5 to 73.2, both within evaluation error.

The main difference between these runs is the solve rate of the policy during rollout. Over the first 20 optimizer steps, the pretraining checkpoint solved about 30% of its GSM8K rollouts. Most groups therefore contained both correct and incorrect samples, producing a useful group-relative learning signal. The mid-trained checkpoint already solved about 77% of its MATH rollouts. Many groups were consequently uniform or nearly uniform, causing the group-relative advantage to vanish.

TRAINING TELEMETRY PREDICTS WHETHER RL IS MOVING THE POLICY

The two runs separate clearly before any downstream evaluation. Table 9 reports KL divergence from the reference policy and total gradient norm over the first and last 20 optimizer steps.

In the successful GSM8K run, KL grows from 0.013 to 0.084, while the gradient norm remains close to 0.6. In the MATH run, KL reaches only 0.003 and the gradient norm stays near 0.1. The policy therefore remains close to its initialization throughout training.

These quantities provide a cheap early diagnostic. A growing KL together with a non-negligible gradient norm indicates that RLTT is actually moving the policy; persistently tiny values suggest that a run is unlikely to produce a downstream gain.

THE POLICY MUST MOVE BEFORE TASK DIFFICULTY MATTERS

Learning rate is a second necessary condition. At 10^{-6}, none of the four additional runs meaningfully departed from the reference policy, regardless of the starting checkpoint or target task. After 94–421 optimizer steps, KL remained near 2×10^{-4}.

One of these runs started from the pretraining checkpoint and optimized MATH, providing a case in which the target task was substantially harder for the initial policy. However, because the smaller

Table 10: Backbone of the from-scratch 1.4B LoopLM, matching Ouro-1.4B (Zhu et al., 2025).

Parameter	Value
Initialization	Random, std. 0.02
Decoder layers	24, shared across recurrent steps
Hidden / FFN dimension	2048 / 5632
Attention heads	16 query, 16 key–value; no GQA
Attention head dimension	128
Attention pattern	Full causal attention; no sliding window
Activation	SiLU-gated feed-forward network
Decoder / final norm	Ouro-norm / learnable RMSNorm, $\epsilon = 10^{-6}$
Vocabulary size	49,152 (Ouro tokenizer)
Maximum positions / RoPE	4096 / base 4×10^4
Input/output embeddings	Untied; LM head shared across steps
Trainable parameters	1.4B

learning rate failed to move the policy, this experiment does not test whether RLTT is effective when the rollout solve rate is very low. The lower end of the useful solve-rate range therefore remains unresolved.

THE USEFUL UPDATE IS SHORT AND DOES NOT HURT GUARD METRICS

The GSM8K improvement also saturates quickly. Accuracy reaches 46.8 at step 100 and 47.3 at step 140, a difference smaller than the benchmark standard error. Continuing the same run therefore provides little additional benefit.

Neither guard metric changes outside its error bar: MMLU moves by $+0.4$ points and HellaSwag by $+0.5$. Within this experiment, the GSM8K improvement therefore does not come with measurable forgetting on these benchmarks.

Taken together, these runs suggest two practical requirements for RLTT in this setting. First, the learning rate must be large enough for the policy to leave its initialization. Second, the rollout task should produce mixed-success groups so that group-relative rewards contain useful variation. Our successful run satisfies both conditions; the high-solve-rate MATH run fails the second, while all 10^{-6} runs fail the first.

Whether RLTT can improve an already mid-trained checkpoint on a task selected specifically to produce a mixed rollout solve rate remains an open question.

D LOOPLM ARCHITECTURE AND TRAINING SETUP

Table 1 summarizes the optimizer and schedule of our two-stage recipe. This appendix provides the remaining details needed to reproduce the from-scratch LoopLM experiments: the backbone, recurrent objective, training mixtures, and dense baselines. We do not repeat the motivation for choosing $T{=}4$ or the warmup analysis from Sections 3 and 3.3.

D.1 SHARED-BLOCK ARCHITECTURE

Our from-scratch model follows the 1.4B Ouro LoopLM architecture of Zhu et al. (2025). A 24-layer decoder block is reused for $T{=}4$ recurrent steps, with a shared LM head and a learned per-token exit gate. Token embeddings are introduced only once: later recurrent steps consume the hidden state from the previous step, without embedding re-injection (Appendix E).

The main model uses the Ouro normalization layout. Each attention and MLP sub-layer normalizes its input and its output, and the normalized output is added to the residual stream, which is itself left unnormalized. A learnable RMSNorm is then applied to the hidden state after each recurrent step, before both the exit gate and the LM head. Appendix A studies alternative normalization schemes.

Table 11: Recurrence, exit gate, and training objective. The configuration is shared by both training stages.

Parameter	Value
Recurrent steps T	4 applications of the shared decoder
Initial state	Token embeddings e
Later-step input	Previous hidden state h_{t-1}
Embedding re-injection	None
Exit gate	Shared linear projection $2048 \to 1$ with sigmoid
Exit distribution	Learned per token; final step receives residual mass
LM objective	$\sum_t p_t \mathcal{L}_t$
Entropy coefficient β	0.15
Loss chunk size	256 positions

D.2 Recurrence and exit objective

Let e denote the token embeddings and B the shared decoder block. The model applies the same block recurrently:

$$h_0 = e, \qquad h_t = B(h_{t-1}), \quad t = 1, \ldots, T. \tag{3}$$

After each recurrent step, a shared linear gate produces a halt probability

$$\lambda_t = \sigma(w_g^\top h_t). \tag{4}$$

Following Zhu et al. (2025), these halt probabilities define the exit distribution through a survival construction:

$$p_t = \lambda_t \prod_{s<t} (1 - \lambda_s) \quad (t < T), \qquad p_T = \prod_{s<T} (1 - \lambda_s), \tag{5}$$

with the final distribution normalized to sum to one.

Training minimizes the exit-weighted language-modeling loss with entropy regularization,

$$\mathcal{L} = \sum_{t=1}^{T} p_t \, \mathcal{L}_t - \beta H(p), \tag{6}$$

using β=0.15. Here, $\mathcal{L}_t$ is the next-token cross-entropy produced by the shared LM head from h_t. During pretraining, loss is applied to all tokens; during mid-training, it is restricted to response tokens.

D.3 Pretraining

Pretraining is a single from-scratch run of 100,000 optimizer steps with 2,097,152 tokens per step, for a total of approximately 210B tokens (Table 1).

Training uses 64 GPUs with FSDP (`shard_grad_op`), bfloat16 compute, fp32 gradient reduction and buffers, FlashAttention-3, and activation checkpointing. Each GPU processes a microbatch of 8 sequences of length 4096. This already gives the full 2M-token optimizer batch, so gradient accumulation is one.

The unique pretraining corpus contains roughly 16B tokens, meaning that the 210B-token budget revisits the data for about thirteen effective epochs. The mixture is dominated by filtered web text, with additional mathematics, code, and instruction-following data.

D.4 High-quality mid-training

Mid-training starts from the step-100,000 pretraining checkpoint and continues for 47,684 steps with the same 2M-token batch, adding approximately 100B tokens.

The architecture, recurrent depth, entropy coefficient, weight decay, and optimizer are unchanged. The peak learning rate is reduced to 1×10^{-4} and cosine-decayed to 1×10^{-5}, with a 1K-step warmup.

Table 12: Pretraining mixture using mosaic-random sampling.

Source	Weight
Nemotron-CC (high quality)	66.5%
Nemotron-CC-Math v1	15.0%
MegaMath (high quality)	4.6%
Nemotron Cascade SFT Stage 1	9.6%
Nemotron Post-Training (code)	3.8%
OpenCoder annealing corpus	0.5%

Table 13: Mid-training mixture based on HRM-Text sources. The first block preserves general knowledge; the second emphasizes mathematics and reasoning.

Source	Weight
FLAN	18.0%
TaskSource	9.0%
Natural Reasoning	7.0%
WebInstruct-verified	4.0%
Platypus	2.0%
No Robots	2.0%
SYNTH	13.0%
OpenMathInstruct-2	12.0%
DeepMind Mathematics	8.0%
AceReason	5.5%
OpenThoughts-2	5.0%
TextbookReasoning	3.5%
AMPS Mathematica	3.0%
NuminaMath	2.5%
Principia collection	2.0%
Sudoku-Extreme	2.0%
AMPS Khan	0.8%
MATH train / OmniMath / GSM8K train	0.3% / 0.3% / 0.1%

Sequences remain 4K tokens long, but are formatted as instruction/response pairs and the loss is applied only to response tokens. We use the HRM-Text data collection of Wang et al. (2026), reweighted toward mathematics and reasoning while retaining a 42% knowledge-preserving portion.

Dense baselines receive the same mid-training stage with identical data, schedule, and token budget.

D.5 Dense baselines and evaluation

We use two dense counterparts to separate parameter-matched and inference-compute-matched comparisons.

`dense-ouro-1.4B` uses the same 24-layer decoder block but removes recurrence by setting T=1. It therefore matches LoopLM in parameter count.

`dense-ouro-3.9B` stacks three copies of the 24-layer block into a 72-layer dense model. Its inference cost approximately matches LoopLM evaluated at three recurrent steps.

Both dense models go through the same two training stages as LoopLM.

Unless stated otherwise, LoopLM is evaluated with a fixed exit after the third recurrent step (k=3), which defines the inference-compute-matched comparison against `dense-ouro-3.9B`. The four-step results in Table 2 use additional, unmatched recurrent compute. The learned exit gate is used during training, but fixed-depth evaluation is used for the reported controlled comparisons.

Reinforcement-learning settings are reported separately in Appendix C.

Table 14: Training LM loss with and without concatenation embedding injection, at a matched budget of 40K steps.

Projection	s_0	Loss	Δ vs. none
None (baseline)	—	1.78	—
Per-step	truncated-normal	1.90	+0.12
Shared	truncated-normal	1.91	+0.12
Shared	learned zero	1.91	+0.12
Per-step	learned zero	1.91	+0.13

E EMBEDDING INJECTIONS

We also tried re-injecting the original token embeddings at every recurrent step: concatenate the initial embedding with the current state, then project back to hidden size before the next loop, following the input-injection construction of Universal Transformers (Dehghani et al., 2019) and later recurrent-depth models (Geiping et al., 2025). We crossed two choices for the injection's initial state s_0 (a learned zero vector vs. a truncated-normal draw) with two projection schemes (one shared projection vs. one per step), and compared all four against an otherwise identical injection-free run. The comparison is on training LM loss; none of these runs was evaluated downstream.

Table 14 reports the loss at a matched budget of 40K steps (2M-token batch, β=0.15, constant learning rate, no warmup). The four injection variants sit in a band of width 0.01 around 1.90–1.91, which is 0.12–0.13 above the injection-free baseline (1.78). A per-step projection with a truncated-normal s_0 is the lowest of the four, by 0.005, which we do not treat as a real difference. Concatenation injection therefore did not help in this recipe, and we keep the injection-free formulation.

F LOOPED DENSE ARCHITECTURE AND TRAINING SETUPS

Looped Qwen starts from Qwen3-1.7B-Base and applies the conversion procedure of Section 4. The pretrained decoder, embeddings, vocabulary, and LM head are preserved and remain trainable. We add recurrent execution, a shared exit gate, scalar embedding injection, and, in one variant, step-specific LoRA adapters.

The same 28-layer decoder is reused for T=3 recurrent steps, increasing effective depth without duplicating the backbone. We run the conversion twice: once on a curated instruction mixture for 60B tokens, and once on a web-scale mixture for 84B tokens. Each run includes a fully shared model and a LoRA-32 variant, together with Qwen3-1.7B-Base and Qwen3-4B-Base baselines continued on the same data for the same number of steps.

Unless stated otherwise, the configuration below refers to the 60B-token instruction run with LoRA-32. The shared model differs only by removing the step-specific LoRA banks; the web-scale run differs only in its data mixture and training duration.

F.1 BACKBONE

The conversion keeps the architecture stored in the Qwen3-1.7B-Base checkpoint rather than relying on fallback values from the generic model configuration. All pretrained parameters remain trainable throughout continued training.

F.2 RECURRENCE AND INPUT INJECTION

The pretrained decoder is applied recurrently for T=3 steps. The first step receives the original token embeddings e. For later steps, we combine the original embeddings with the hidden state from the preceding iteration:

$$z_t = we + (1 - w)h_{t-1}, \qquad t > 0. \tag{7}$$

Table 15: Backbone configuration of the converted Qwen3-1.7B model.

Parameter	Value
Initialization	Qwen3-1.7B-Base checkpoint
Trainable backbone	Decoder, embeddings, and LM head
Decoder layers	28
Hidden / FFN dimension	2048 / 6144
Attention heads	16 query, 8 key–value
Attention head dimension	128
Attention pattern	Full causal GQA; no sliding window
Activation	SiLU-gated feed-forward network
Normalization	RMSNorm, $\epsilon = 10^{-6}$
Attention dropout / bias	0 / disabled
Vocabulary size	151,936
Maximum positions	32,768
RoPE	Base 10^6; no scaling
Input/output embeddings	Untied; LM head shared across recurrent steps
Initialization std.	0.02 for newly added standard linear modules

Table 16: Recurrent modules and objective used for Looped Qwen.

Parameter	Value
Recurrence	
Recurrent steps	$T = 3$ applications of the shared decoder
Step 0 input	Original token embeddings e
Later-step input	$z_t = we + (1 - w)h_{t-1}$
Injection parameter	One trainable scalar shared across steps
Initialization	$a = 4.6$, $w = \sigma(a) = 0.99005$
Exit gate	Shared linear projection $2048 \to 1$
Exit mode	Learned per-token distribution
Objective	
LM objective	Exit-weighted next-token cross-entropy
Entropy coefficient β	0.15
Exit smoothing ϵ	0.05
Minimum direct LM weight	0.0167 per step
Target shift	0; identical next-token target at every step
Loss mask / chunk size	Precomputed positional mask / 256 positions

The mixing coefficient is shared across all recurrent steps and parameterized as

$$w = \sigma(a). \tag{8}$$

We initialize $a = 4.6$, corresponding to $w = 0.99005$. Thus, later recurrent steps remain anchored to the original token representation while incorporating a small update from the previous hidden state.

A shared linear exit gate maps the 2048-dimensional hidden state to one scalar per token. The model is trained with the same exit-weighted objective as the from-scratch LoopLM, with entropy coefficient $\beta = 0.15$. During conversion, we additionally smooth the exit-weighted LM loss with $\epsilon = 0.05$, ensuring that every recurrent step receives a minimum direct LM weight of 0.0167.

F.3 Step-specific LoRA

The shared variant introduces no step-specific parameters beyond the scalar input injection and exit gate. The LoRA-32 variant additionally assigns an independent LoRA bank to each recurrent step.

Adapters are applied to the attention q, k, v, and output projections and to the MLP gate, up, and down projections. Each adapter has rank 32 and uses an unscaled BAx parameterization, with Kaiming-uniform initialization for A and zero initialization for B.

Table 17: Step-specific LoRA configuration.

Parameter	Value
Adapter layout	One independent bank per recurrent step
Rank	32
Attention targets	q, k, v, output projections
MLP targets	Gate, up, down projections
Parameterization	Unscaled BAx
Initialization	Kaiming-uniform A, zero B
LoRA parameters	104,595,456 across three banks
Total trainable parameters	2,136,339,458

Table 18: Optimization and batch configuration for the 60B-token conversion run.

Parameter	Value
Token budget	60B tokens
Optimizer updates	28,611
Sequence length	4096
Optimizer	Fused AdamW
AdamW coefficients	$\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$
Peak / minimum LR	3×10^{-5} / 3×10^{-6}
LR schedule	1K-step warmup, then cosine decay
Weight decay	0.1 on attention/MLP projections and LoRA
Gradient clipping	Global norm 1.0
Hardware	4 nodes $\times$ 8 H100 GPUs
Microbatch per GPU	16 sequences / 65,536 tokens
Distributed microbatch	512 sequences / 2,097,152 tokens
Gradient accumulation	1
Effective optimizer batch	2,097,152 tokens

Across three recurrent steps, the LoRA banks add 104,595,456 trainable parameters, bringing the total number of trainable parameters in the LoRA-32 model to 2,136,339,458.

F.4 Continued-training setup

The main conversion run uses the curated instruction mixture for 60B tokens, corresponding to 28,611 optimizer steps. Sequence length is 4096.

Training uses 32 H100 GPUs across four nodes. Each GPU processes 16 sequences, or 65,536 tokens. One distributed microbatch therefore contains 512 sequences, corresponding to 2,097,152 tokens. This is already the full optimizer batch, so gradient accumulation is one.

We use fused AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-8}$. The learning rate warms up linearly for 1,000 steps from 3×10^{-6} to 3×10^{-5}, then follows cosine decay to 3×10^{-6}. Weight decay is 0.1 for attention and MLP projections and for LoRA parameters, and zero otherwise. Gradients are clipped to global norm 1.0.

F.5 Training data

The 60B-token run uses a six-source curated mixture dominated by mathematics, code, and post-training data. Sampling uses mosaic-random mixing with balanced streams and shuffled random batches.

For reproducibility, training uses random seed 42, model-initialization seed 271,828, and shuffle seed 45. Shuffling is enabled with a block size of 10,737,418 samples.

Table 19: Curated instruction mixture used for the 60B-token Looped Qwen run.

Source	Weight (%)
Nemotron Post-Training Dataset v1	30.00
Nemotron Post-Training Dataset v1 (math)	27.90
Nemotron Post-Training Dataset v1 (code)	23.10
Nemotron SFT Instruction Following Chat v2	11.90
Nemotron Instruction Following Chat v1	3.86
Nemotron Science v1	3.24

Table 20: Conversion on the 84B-token web-scale mixture. Looped models are evaluated after two of three recurrent steps. $\pm$ denotes standard error; †continued on the same mixture for the same number of steps; bold marks scores above Qwen3-1.7B-Base†.

Benchmark	Qwen3-1.7B-Base†	Looped Qwen, shared	Looped Qwen, LoRA-32	Qwen3-4B-Base†
GSM8K	60.12 ± 1.35	$\mathbf{61.87 \pm 1.34}$	$\mathbf{63.68 \pm 1.32}$	76.50 ± 1.17
MATH	38.00 ± 0.65	$\mathbf{38.08 \pm 0.65}$	$\mathbf{39.22 \pm 0.65}$	47.12 ± 0.66
MMLU	61.18 ± 0.40	$\mathbf{62.73 \pm 0.39}$	$\mathbf{62.15 \pm 0.39}$	71.04 ± 0.36
MMLU-Pro	32.43 ± 0.42	$\mathbf{32.96 \pm 0.42}$	$\mathbf{33.88 \pm 0.42}$	46.28 ± 0.44
ARC-Challenge	55.29 ± 1.45	$\mathbf{55.38 \pm 1.45}$	54.61 ± 1.45	61.35 ± 1.40
ARC-Easy	80.81 ± 0.81	$\mathbf{81.27 \pm 0.80}$	80.81 ± 0.80	85.48 ± 0.72
HellaSwag	69.37 ± 0.46	$\mathbf{70.91 \pm 0.45}$	$\mathbf{71.17 \pm 0.45}$	76.47 ± 0.42
Winogrande	66.54 ± 1.33	$\mathbf{67.56 \pm 1.32}$	$\mathbf{68.19 \pm 1.31}$	71.98 ± 1.26

F.6 Baselines and evaluation

Each conversion run is paired with two dense baselines. Qwen3-1.7B-Base is the parameter-matched reference and also the checkpoint from which Looped Qwen is initialized. Qwen3-4B-Base provides a larger, inference-compute-matched reference. Both dense models continue training on exactly the same mixture for the same number of optimizer steps as their corresponding looped model.

The 60B-token instruction run is trained for 28,611 steps; the web-scale run uses 84B tokens and 40,000 steps. In both cases, reported results use the final checkpoint.

Looped Qwen is evaluated at a fixed exit after two of its three recurrent steps, the comparison point used in Section 5.3.

The learned gate is part of training, whereas the controlled benchmark comparison uses a fixed recurrent depth.

G Conversion training dynamics

Figure 9 shows the training logs behind Section 4. Panels (a) and (b) come from two web-mixture runs that share data, batch, learning rate, β=0.15 and seed and differ only in the input injection. The other panels come from single runs whose settings are given in the caption, and every configuration was trained with one seed.

H Conversion on the web-scale mixture

To separate the effect of recurrence from the effect of the curated conversion data, we repeat the same conversion procedure on a second mixture dominated by filtered web text, with smaller mathematics and code components. This run uses 84B tokens and 40,000 optimizer steps.

As in the instruction-data experiment, the looped and dense models are trained on identical data for the same number of steps and evaluated at a fixed exit after two recurrent iterations.

The shared looped model improves over Qwen3-1.7B-Base continued on the same data on every benchmark, although the gains are modest compared with the curated instruction run. The largest

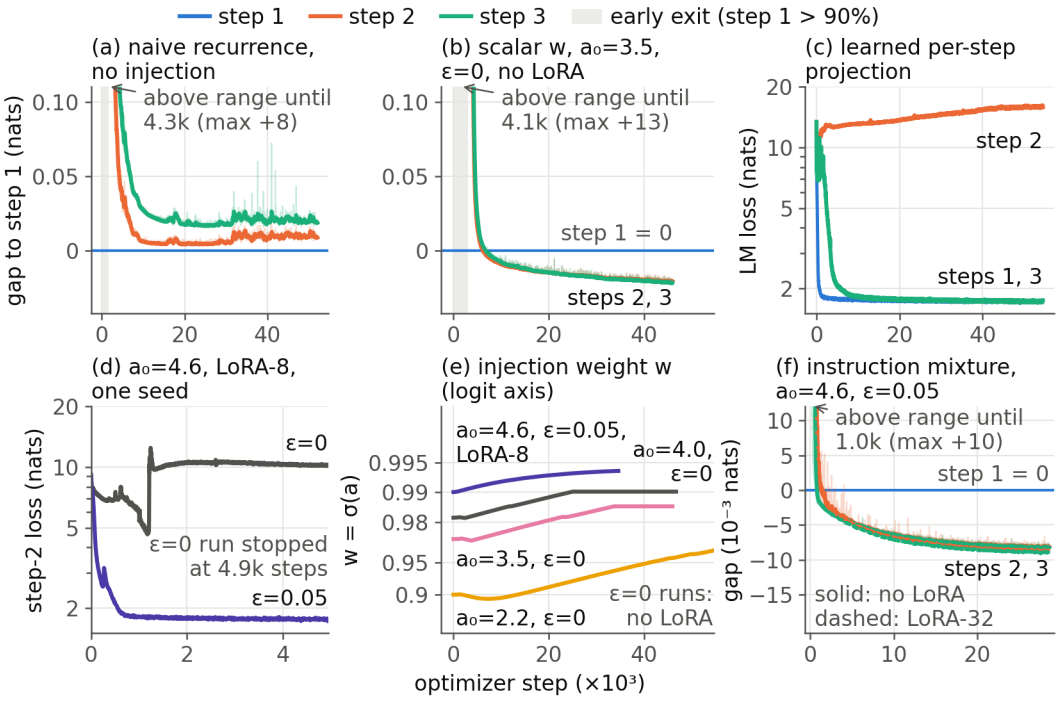


Figure 9: Conversion training logs, one seed per configuration. (a) Naive recurrence without input re-injection and (b) scalar injection $w = \sigma(a)$ with $a_0 = 3.5$, $\epsilon = 0$ and no LoRA: loss of steps 2 and 3 minus the loss of step 1 on the same batch. Shading marks where the exit probability of step 1 is above 0.9; before about 4k steps the gap is above the plotted range. (c) Per-step LM loss of the run with a learned per-step projection. (d) Step-2 loss at $a_0 = 4.6$ with rank-8 LoRA, without and with exit-loss smoothing; the unsmoothed run was stopped at 4.9k steps. (e) Learned injection weight w on a logit axis for $a_0 = 2.2$, 3.5 and 4.0 ($\epsilon = 0$, no LoRA) and $a_0 = 4.6$ ($\epsilon = 0.05$, rank-8 LoRA). (f) The gap of (a) and (b) for the instruction-mixture runs with $a_0 = 4.6$ and $\epsilon = 0.05$, without adapters (solid) and with rank-32 LoRA (dashed), in thousandths of a nat. Plotted points are logged means over 5 to 10 optimizer steps (every step in the unsmoothed run of (d)); in (a), (b) and (f) the thick line is a 31-point moving average of these points.

improvements are $+1.75$ points on GSM8K, $+1.55$ on MMLU, and $+1.54$ on HellaSwag; the remaining gains are about one point or less.

The LoRA-32 variant follows the same overall pattern and does not consistently improve over the fully shared model. Qwen3-4B-Base remains ahead across all benchmarks, consistent with the instruction-data experiment.

The two conversion runs therefore show the same qualitative effect across substantially different data mixtures: recurrence improves over Qwen3-1.7B-Base given the same continued training, while the size of the gain depends strongly on the continued-training data.

I Compute accounting

Table 21 counts the compute per token of every model in the controlled comparisons, and three points follow from it.

The from-scratch recipe is cheap. The full two-stage run costs 9.9×10^{21} training FLOPs. The same architecture trained on Ouro's schedule, 3T tokens at eight recurrent steps followed by 4.7T tokens at four, costs about 3.4×10^{23}, so our recipe needs about 3% of that training compute, 35 times less.

The inference-compute match of Section 5.1 is exact: LoopLM at three iterations and `dense-ouro-3.9B` apply the same 72 layer passes per token. Training LoopLM with all four iterations costs 1.4 times as much per token as `dense-ouro-3.9B`, which stores 2.7 times as many parameters.

Conversion is cheaper still. Converting Qwen3-1.7B-Base on the 60B-token mixture costs 1.9×10^{21} FLOPs, 0.5% of the 3.7×10^{23} of its 36T-token pretraining. At its two-step exit, Looped Qwen uses 6.3 GFLOPs per token, 22% fewer than Qwen3-4B-Base. The conversion keeps an untied LM head initialized from the input embeddings, and the counts include it.

Table 21: Compute per token, with the usual approximation of $2N$ FLOPs for a forward pass and $6N$ for a training step, where N counts the parameters applied to a token: the non-embedding layers times the recurrent steps, plus the LM head. Looped models train with all recurrent steps and apply the LM head at every step. Attention-score FLOPs are left out; they are equal within each matched pair. The Ouro-1.4B total is an estimate from its reported stage budgets and recurrent depths on the same architecture.

Model	Inference (GFLOPs/token)	Training (GFLOPs/token)	Total training (FLOPs)
LoopLM, exit k=3 / k=4	7.60 / 10.07	32.0	9.9×10^{21}
`dense-ouro-1.4B`	2.67	8.0	2.5×10^{21}
`dense-ouro-3.9B`	7.60	22.8	7.1×10^{21}
Ouro-1.4B schedule (estimate)	10.07	32.0 / 64.0	3.4×10^{23}
Looped Qwen, exit k=2 / k=3	6.26 / 9.08	31.0	1.9×10^{21} (60B)
Qwen3-1.7B-Base	3.44	10.3	3.7×10^{23} (36T)
Qwen3-4B-Base	8.04	24.1	–

J THINK-AT-HARD

We also reproduced Think-at-Hard (TaH) (Fu et al., 2025) on our training mixture. TaH uses a static oracle policy to identify tokens that a frozen reference model predicts incorrectly on its first pass. The backbone and deeper-step LoRA adapters are then trained to refine these hard tokens, after which a lightweight decider is trained to imitate the oracle policy at inference. In our reproduction, the repeated-step LM loss never fell below the first-step loss, and accuracy degraded substantially on several downstream benchmarks. Thus, we could not reproduce the gains reported in the original work under our data and training setup. Despite this negative result, correctness-directed routing is an appealing alternative to the cumulative exit probabilities used by LoopLM. A deeper step receives supervision specifically where the shallower step fails, rather than learning a stopping probability only through its contribution to the expected LM loss. This directly connects additional computation to errors that still need to be corrected and may provide a cleaner objective for token-level adaptive depth.

The binary top-1 oracle used for two-step TaH does not directly specify how many iterations a hard token needs. An exact correctness-based extension to three steps would require labels indicating whether each token is first solved at step 0, 1, or 2. If these labels are precomputed, step 1 must first be trained before its errors can be measured, followed by another labeling pass and another training run for the three-step model. Repeating this procedure up to depth T would require a sequence of shallower models and dataset relabeling stages. The current TaH paper avoids this cost for TaH-3 by assigning depths through quantiles of a frozen reference model's loss. This is efficient, but the resulting depth is a proxy for difficulty rather than the first recurrent step that actually predicts the token correctly.

We propose generating the depth mask online from the looped model itself. During training, we execute the maximum active number of recurrent steps and compare the top-1 prediction at every step with the target token. For token position j, we select

$$d_j = \min\{t : \arg\max_v q_t(v \mid x_{\leq j}) = x_{j+1}\}, \tag{9}$$

where q_t is the token distribution at recurrent step t, and apply its LM gradient only at depth d_j. If no step predicts the target correctly, we assign the token to the final active step. The discrete

mask is treated as a stop-gradient routing decision. Unlike static oracle labels, it always reflects the current model and requires neither a separate reference model nor an offline relabeling pass. Naively enabling all depths from the start can starve intermediate steps of supervision. In a newly initialized three-step model, step 1 will often remain incorrect, so many tokens are routed directly from step 0 to step 2; step 1 then receives too few updates to improve. We address this with a depth curriculum. For a three-step model, the first half of training activates only steps 0 and 1, assigning all tokens missed by both to step 1. Step 2 is introduced in the second half, after step 1 has learned a useful refinement. More generally, the maximum active depth is increased one step at a time during training. This curriculum is intended to combine TaH's correctness-directed supervision with more than two recurrent steps, while avoiding the intermediate-step collapse that makes a fully dynamic oracle unstable. We leave a controlled evaluation of this training rule to future work.