P2N: REUSING DEEP REPRESENTATIONS FOR GREATER EFFECTIVE DEPTH

Anonymous authors
Paper under double-blind review

ABSTRACT

Mathematical problem solving and autonomous agents are increasingly important applications of language models, demanding stronger reasoning capabilities. Computational depth is a key resource for such reasoning, but adding model layers increases both training and inference costs, while test-time scaling incurs substantial additional inference computation. We introduce P2N (Previous to Next), which reuses the previous token's deep representations—already available during generation—to increase effective computational depth with negligible decoding overhead. Specifically, P2N adds the previous token's deep hidden state to the current token's shallow representation, allowing subsequent layers to build on previously completed deep computation. This recurrence extends effective computational depth across tokens, while introducing only a single vector addition per decoding step. We train the recurrence using parallel Jacobi updates restricted to a middle-layer core. Experiments with Qwen-style models show that, at 1.5B parameters, P2N improves GSM8K accuracy by over 10 percentage points under both zero-shot and few-shot prompting, with nearly unchanged decoding latency. The benefits also extend to language modeling and average performance on general downstream benchmarks.

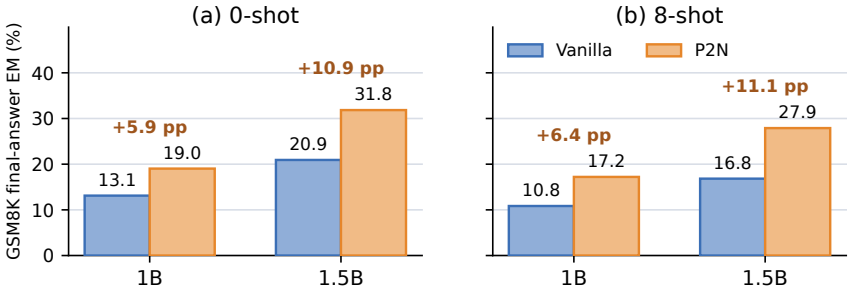


Figure 1: **Mathematical reasoning at the 1B and 1.5B scales.** At 1.5B, P2N improves GSM8K accuracy over Vanilla by 10.9 and 11.1 percentage points under zero-shot and eight-shot prompting, respectively, corresponding to relative gains of 52.2% and 65.8%.

1 INTRODUCTION

The growing use of language models for mathematical problem solving and autonomous agents places increasing demands on their reasoning capabilities. Computational depth is an important resource for reasoning, enabling representations to be refined through successive stages of computation (Dehghani et al., 2018; Geiping et al., 2026). However, adding Transformer layers increases both training and inference costs, while test-time scaling (Snell et al., 2024) incurs additional computation during generation. These costs motivate a natural question: can language models obtain greater effective depth by reusing computation that has already been performed?

We therefore introduce P2N (Previous to Next), which reuses the previous token's deep representations—already available during generation—to increase effective computational depth with negligible decoding overhead. Specifically, for a contiguous core of Transformer layers, P2N

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.

adds the previous token's core output to the current token's core input. This parameter-free feedback requires only a single vector addition per token.

Although this recurrence fits naturally into autoregressive decoding, directly applying it during training would introduce a left-to-right dependency across tokens. We address this dependency using Jacobi iteration (Saad, 2003). After an initial warm pass, each update combines the fixed core inputs with the preceding iteration's outputs shifted one position to the right, allowing all token positions to be updated in parallel. To limit the additional training cost, we restrict the recurrent core to the middle third of the network and evaluate the prefix and suffix only once, as illustrated in Figure 2.

We pretrain Qwen-style models (Yang et al., 2025) from 150M to 1.5B parameters from scratch on the Pile (Gao et al., 2020), comparing P2N with vanilla baselines at matched parameter and token budgets. At 1.5B, after matched supervised fine-tuning, P2N improves GSM8K (Cobbe et al., 2021) final-answer exact match by 10.92 and 11.07 percentage points under zero-shot and eight-shot prompting, respectively. Decoding benchmarks show nearly unchanged per-token latency. The benefits also extend to language modeling and general downstream performance: P2N consistently lowers validation loss across scales and, at 1.5B, reduces Pile validation perplexity from 6.72 to 6.52 while improving average zero-shot and five-shot accuracy across nine task configurations by 1.68 and 2.11 percentage points, respectively.

2 Related Work

Latent representations in large language models. Large language models use discrete tokens at their interfaces but represent context and intermediate computation in continuous hidden states (Vaswani et al., 2017). Prior work exploits this space by feeding hidden states back as later inputs (Hao et al., 2024), mixing text with latent tokens or continuous concepts (Su et al., 2025; Zhang et al., 2026), and combining discrete tokens with hidden representations (Yue et al., 2026; Amos et al., 2026). Other methods shape hidden states through auxiliary pretraining objectives (Tack et al., 2026; Teoh et al., 2025). Universal Transformer repeatedly applies shared computation across depth (Dehghani et al., 2018), a direction later developed by Looped Transformers (Giannou et al., 2023; Yang et al., 2024) and recurrent-depth models (Geiping et al., 2026). These approaches extend computation beyond discrete tokens, but may increase training or inference cost.

Latent feedback in Transformers. Some work makes deeper representations available to shallow computation by changing how historical hidden states are stored and reused. Segment-level approaches cache or compress hidden states (Dai et al., 2019; Rae et al., 2019), recurrent-memory models carry block states or memory tokens across longer contexts (Hutchins et al., 2022; Bulatov et al., 2022), and feedback-based models expose higher-level history through recurrent or feedback attention (Fan et al., 2020; Ju et al., 2022; Hwang et al., 2024). Recent designs create more direct cross-token, cross-layer paths through downward connections (Tang & Lu, 2026), output-derived KVs (Oncescu et al., 2026), and learned projection or fusion modules (Huang et al., 2026; Cai et al., 2026). PonderLM-2 treats the current position's final hidden state as an additional latent thought and feeds it back before predicting the next token (Zeng et al., 2025). Recirculation adds feedback after pretraining by sending deep activations back to shallow layers (Mozer et al., 2026). Its feedback path is not pretrained, and decoding runs two complete model computations in parallel at each step. However, these methods either introduce additional memory, projection, or fusion components, require extra full-model computation, or add feedback only after pretraining. P2N learns a simple additive feedback connection during pretraining, preserves sequence parallelism through Jacobi updates, and reuses cached states during decoding.

3 Method

We first describe P2N without committing to a particular subset of model layers. Let F denote the transformation performed by a causal language model, $x_t \in \mathbb{R}^d$ its input representation at token position t, and $h_t \in \mathbb{R}^d$ its output representation. For clarity, we omit the standard causal context from the notation. A vanilla model can then be written as:

$$h_t = F(x_t). \tag{1}$$

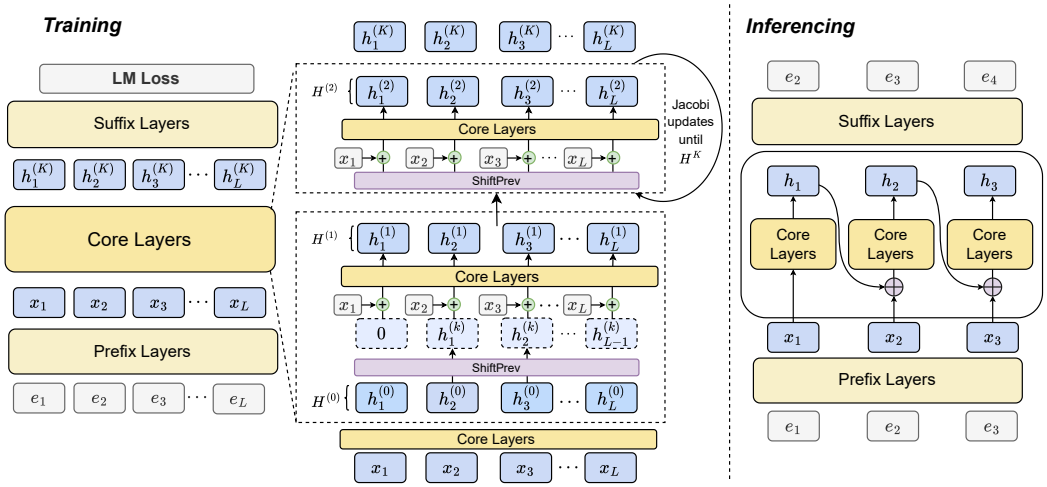


Figure 2: **Overview of P2N.** The Transformer is divided into prefix, core, and suffix layers. **Autoregressive decoding (right).** At each token, the previous token's core output is added to the current token's prefix representation before entering the core layers. Since the previous token's state is already available, this cross-token feedback is naturally compatible with autoregressive decoding; however, directly applying it during teacher-forced training would introduce a sequential dependency across token positions. **Parallel training (left).** The prefix processes the full sequence once, and an initial warm pass through the core produces the starting states. A small number of Jacobi updates then refines these states in parallel: in each iteration, the preceding iteration's core outputs are shifted one position to the right and added to the fixed prefix representations to form the new inputs. Finally, the suffix layers are applied once to the final states to compute the language-modeling loss.

3.1 Cross-token recurrence during autoregressive decoding

At autoregressive step t, the model receives the current input x_t. Because decoding proceeds from left to right, the previous token's output h_{t-1} has already been computed. P2N therefore defines:

$$h_t = F(x_t + h_{t-1}), \qquad h_0 = 0. \tag{2}$$

3.2 Parallel training with Jacobi iteration

However, directly applying Equation 2 during teacher-forced training would introduce a left-to-right sequential dependency. Although the complete input sequence is available, the state h_{t-1} is an *output* of F and is therefore unavailable until the preceding token has completed its computation. A literal implementation would consequently give up the token parallelism of standard Transformer training. We avoid this serialization with a Jacobi iteration (Zeng et al., 2025; Saad, 2003).

Let $X = [x_1, \ldots, x_T] \in \mathbb{R}^{T\times d}$ denote the input sequence, and let $H^{(k)} \in \mathbb{R}^{T\times d}$ denote the sequence of model outputs after the k-th iteration. Here, the superscript (k) indexes the Jacobi iteration, whereas a subscript t indexes a token position.

Stage 1: warm pass. We first process the full sequence without cross-token output feedback:

$$H^{(0)} = F(X). \tag{3}$$

The warm pass provides the initial output sequence from which the recurrent states are constructed.

Stage 2: Jacobi iterations. Starting from $H^{(0)}$, we perform K parallel updates:

$$H^{(k+1)} = F\Big(X + \mathrm{ShiftPrev}\Big(H^{(k)}\Big)\Big), \qquad k = 0, \ldots, K-1. \tag{4}$$

The operator $\mathrm{ShiftPrev}$ shifts the sequence $H^{(k)}$ one position to the right, so that position t receives the state from position $t-1$; the first position and packed-sequence boundaries are filled with

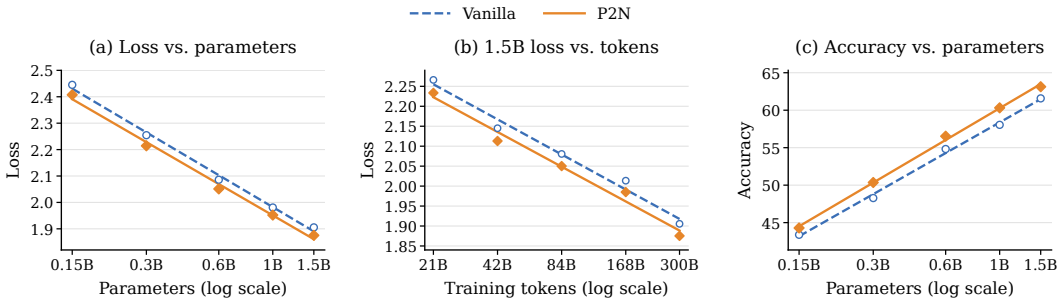


Figure 3: **Scaling behavior.** (a) Final Pile validation loss across parameter scales (lower is better). (b) Pile validation loss over training tokens at the 1.5B scale (lower is better). (c) Mean zero-shot accuracy across parameter scales (higher is better). Overall, P2N exhibits consistently favorable behavior across all three scaling views.

zeros. Each update reads only the completed outputs of the preceding iteration. Therefore, all token positions within one Jacobi iteration remain parallel, while only the small number of iterations is sequential. The final state $H^{(K)}$ is used to compute the language-modeling loss. All evaluations of F share parameters, and gradients propagate through the complete unrolled computation. In our default setting, we sample K uniformly from $\{2, 3\}$ for each training batch (Zeng et al., 2025).

3.3 Partial Architecture

To reduce the cost of Jacobi training, we apply P2N only to a contiguous subset of Transformer layers. We partition an L-layer model into three modules:

$$\mathcal{F}_{\text{full}} = \mathcal{F}_{\text{suf}} \circ \mathcal{F}_{\text{core}} \circ \mathcal{F}_{\text{pre}}, \tag{5}$$

where the recurrent core contains C layers. In subsection 3.1 and subsection 3.2, the transformation F is instantiated as $\mathcal{F}_{\text{core}}$: only the core participates in the cross-token recurrence and Jacobi iterations, while the prefix and suffix are each evaluated once.

Unless stated otherwise, the prefix, core, and suffix each contain approximately one third of the model layers, so $C \approx L/3$. For full-network Jacobi training, the estimated training compute relative to Vanilla is:

$$\mathbb{E}[K + 1] = 3.5, \qquad K \sim \text{Uniform}\{2, 3\}. \tag{6}$$

Here, K is the number of additional Jacobi iterations sampled for each training batch, and the $+1$ accounts for the initial warm pass. With the partial-loop design, only the C core layers are repeatedly evaluated, reducing this compute ratio to:

$$\frac{(L - C) + \mathbb{E}[K + 1]C}{L} = 1 + \frac{2.5C}{L} \approx 1 + \frac{2.5}{3} \approx 1.83. \tag{7}$$

4 Experiments

Our experiments are organized into four parts. First, we study scaling behavior and language modeling through large-scale pretraining. Second, we evaluate downstream capabilities, including general benchmarks, instruction following, and mathematical reasoning. Third, we measure autoregressive decoding efficiency. Finally, we examine the convergence of Jacobi iteration.

4.1 Large-Scale Pretraining

We pretrain a series of Qwen-style dense language models (Yang et al., 2025) from scratch on the Pile (Gao et al., 2020), covering model scales of 150M, 300M, 600M, 1B, and 1.5B. We use a target tokens-per-parameter ratio of 200 at each scale. For every model size, P2N and the corresponding Vanilla baseline share the same backbone architecture, training data, token budget, and optimization settings. Detailed model configurations and training hyperparameters are provided in Appendix B.

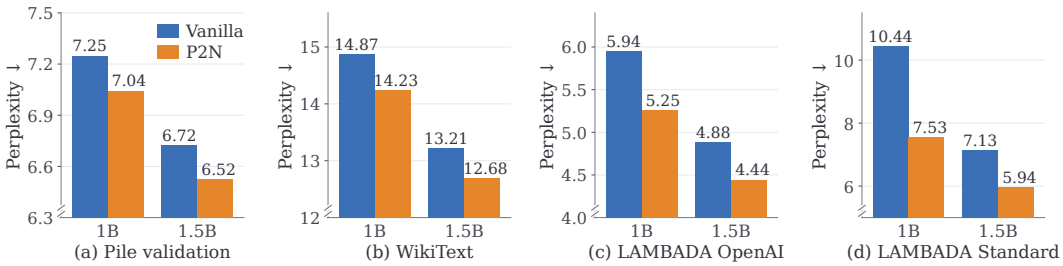


Figure 4: **Language modeling at the 1B and 1.5B scales.** Perplexity of Vanilla and P2N dense models on (a) Pile validation, (b) WikiText-2, (c) LAMBADA OpenAI, and (d) LAMBADA Standard; lower is better. WikiText-2 uses word-level perplexity, and LAMBADA uses final-word perplexity. P2N uses three Jacobi updates at evaluation time. Numbers above the bars denote perplexity values. P2N achieves lower perplexity at both scales on every benchmark, indicating the effectiveness of this cross-token feedback method.

4.1.1 Scaling Behavior

Figure 3 summarizes the scaling behavior of P2N. Panels (a) and (c) report final Pile validation loss and average downstream accuracy across model sizes, respectively, while panel (b) tracks Pile validation loss throughout pretraining for the matched 1.5B models. The downstream score is the unweighted mean of zero-shot accuracies over the seven informative task configurations in Table 1, excluding WinoGrande (Sakaguchi et al., 2021) and ARC-Challenge (Clark et al., 2018) because their small-model scores remain close to random guessing. All horizontal axes use a logarithmic scale. Across all evaluated model sizes, P2N achieves lower validation loss and higher downstream accuracy than Vanilla, while maintaining lower loss throughout pretraining at the 1.5B scale.

4.1.2 Language Modeling Ability

We evaluate the 1B and 1.5B dense models on Pile validation (Gao et al., 2020), WikiText-2 (Merity et al., 2016), the OpenAI and Standard variants of LAMBADA (Paperno et al., 2016), using matched preprocessing and scoring protocols for Vanilla and P2N. As shown in Figure 4, P2N consistently reduces perplexity across all four benchmarks at both model sizes. Relative to Vanilla, the reductions are 2.8–3.0% on Pile, 4.0–4.3% on WikiText-2, 9.1–11.6% on LAMBADA OpenAI, and 16.7–27.8% on LAMBADA Standard. The largest relative improvements occur on the two LAMBADA variants, indicating improved prediction of the final word from its preceding context. These results show that the gains extend beyond the pretraining validation set to the additional language modeling benchmarks.

4.2 Downstream Task Evaluation

We further assess the downstream capabilities of P2N at the 1B and 1.5B scales through a series of experiments. These evaluations examine whether the improvements observed during pretraining translate into better downstream performance.

4.2.1 General Downstream Tasks

We evaluate Vanilla and P2N dense models on nine task configurations: LAMBADA-OpenAI, LAMBADA-Standard (Paperno et al., 2016), PIQA (Bisk et al., 2020), WinoGrande (Sakaguchi et al., 2021), SciQ (Welbl et al., 2017), HellaSwag (Zellers et al., 2019), ARC-Easy, ARC-Challenge (Clark et al., 2018), and RACE (Lai et al., 2017). Table 1 reports zero-shot and five-shot raw accuracy, together with the unweighted average over all nine configurations. At evaluation time, P2N uses $K = 3$ Jacobi iterations after the warm pass.

In zero-shot evaluation, P2N improves average accuracy from 54.85% to 56.43% at 1B and from 57.66% to 59.34% at 1.5B. The improvements also hold in the five-shot setting, where average accuracy increases from 55.73% to 57.42% at 1B and from 58.20% to 60.31% at 1.5B. These results

Table 1: **Downstream performance at the 1B and 1.5B scales.** Zero-shot and five-shot raw accuracy (%) of dense models. Avg. is the unweighted mean over the nine task configurations. Bold indicates the higher score within each matched scale and shot setting. P2N consistently improves average accuracy at both scales in both settings.

Scale	Method	LAMBADA		PIQA	Wino Grande	SciQ	Hella Swag	ARC		RACE	Avg.
		OpenAI	Standard					Easy	Challenge		
Zero-shot											
1B	Vanilla	61.73	49.91	71.06	**59.91**	88.70	41.62	60.02	**27.39**	33.30	54.85
1B	P2N	**64.22**	**57.13**	**72.31**	58.64	**89.50**	**43.45**	**61.45**	27.05	**34.16**	**56.43**
1.5B	Vanilla	66.23	57.36	**74.16**	59.91	88.60	45.05	64.14	27.99	35.50	57.66
1.5B	P2N	**67.92**	**61.46**	73.18	**62.27**	**90.70**	**46.74**	**64.90**	**29.86**	**37.03**	**59.34**
Five-shot											
1B	Vanilla	57.62	49.18	**72.58**	60.06	93.40	41.68	63.38	29.10	34.55	55.73
1B	P2N	**59.21**	**56.47**	72.36	**61.09**	**94.10**	**43.61**	**64.77**	**29.27**	**35.89**	**57.42**
1.5B	Vanilla	61.27	52.63	**74.48**	61.80	**94.20**	45.18	66.33	32.59	35.31	58.20
1.5B	P2N	**62.90**	**60.88**	74.05	**65.43**	93.70	**47.13**	**68.18**	**33.62**	**36.94**	**60.31**

demonstrate consistent gains in overall downstream performance across both model scales and shot settings.

4.2.2 Instruction-Following Ability

We fine-tune the 1B and 1.5B dense models and their vanilla counterparts on Alpaca (Taori et al., 2023), and evaluate their instruction-following ability on MT-Bench (Zheng et al., 2023). Fine-tuning, generation, and judging settings are matched within each model-scale comparison. We use GPT-5.5 as the judge and report scores averaged over both conversation turns.

As shown in Figure 5, P2N improves the overall MT-Bench score from 2.219 to 2.613 at 1B and from 2.381 to 2.913 at 1.5B, yielding gains of 0.394 and 0.532 points, respectively. At 1B, P2N improves six of the eight categories, matches the vanilla model on roleplay, and slightly trails it on humanities. At 1.5B, improvements extend to all eight categories, with the largest gains in reasoning, extraction, and roleplay. These results suggest that the benefits of P2N transfer to instruction-following after supervised fine-tuning.

4.2.3 Mathematical Reasoning

We fine-tune the pretrained 1B and 1.5B dense models on the official GSM8K training split (Cobbe et al., 2021) and evaluate final-answer exact match (EM) on all 1,319 test examples under zero-shot and eight-shot prompting. Fine-tuning budgets are matched within each scale, and prompting formats, decoding settings, and answer-extraction procedures are matched between Vanilla and P2N within each evaluation setting.

As shown in Figure 1, P2N improves zero-shot EM by 5.91 and 10.92 percentage points at 1B and 1.5B, respectively. The improvements also hold under eight-shot prompting, with gains of 6.37 and 11.07 percentage points at the corresponding scales. These results demonstrate improved mathematical reasoning performance across both model scales and prompting settings.

4.3 Decoding Efficiency

During autoregressive decoding, P2N introduces only one additional vector addition per token, without repeating any Transformer layer. We benchmark the 1.5B dense model on a single GPU using BF16, SDPA, and the standard KV cache. Each run uses a 128-token prompt and measures 2,048 decode steps after 16 untimed warm-up steps. We report averages over five runs for each batch size.

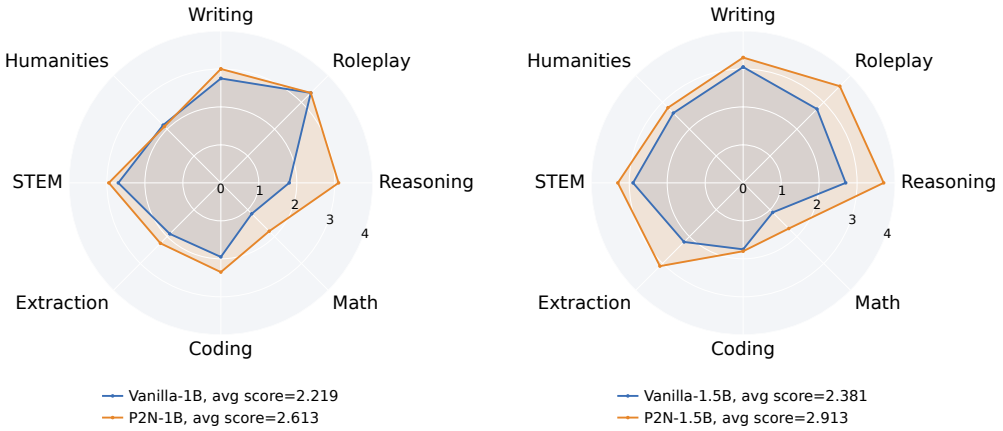


Figure 5: **Instruction-following ability.** Category-wise MT-Bench scores for 1B (left) and 1.5B (right) dense models after Alpaca fine-tuning, evaluated by GPT-5.5. Each category score averages both conversation turns; legends report the overall mean across all 80 questions and both turns. P2N improves the overall score at both scales, with gains across all eight categories at 1.5B. Both panels use the same radial scale.

Table 2: **Autoregressive decoding efficiency of the 1.5B dense model on a single GPU.** TPOT denotes time per output token, and throughput counts generated tokens per second across the batch. Results are averaged over five paired runs generating 2,048 tokens per sequence. Rel. latency is the ratio of P2N TPOT to Vanilla TPOT. P2N maintains near-vanilla latency across batch sizes.

Batch	TPOT (ms)		Throughput (tok/s)		Rel. latency
size	Vanilla	P2N	Vanilla	P2N	P2N / Vanilla
1	16.442	16.494	60.8	60.6	1.003×
4	16.983	16.991	235.5	235.4	1.000×
16	17.079	17.143	936.8	933.3	1.004×
32	22.298	22.352	1435.1	1431.7	1.002×

Table 2 reports time per output token (TPOT), aggregate decoding throughput, and relative latency. Across batch sizes, P2N consistently maintains nearly identical latency to Vanilla, indicating negligible additional decoding overhead.

4.4 Convergence Diagnostics of Jacobi Iteration

We further examine the convergence behavior of Jacobi iteration on the 1.5B P2N model. Let $H^{(k)} \in \mathbb{R}^{T\times d}$ denote the core states after k Jacobi iterations, with $H^{(0)}$ obtained from the warm pass. We also denote by $H_{\text{seq}} \in \mathbb{R}^{T\times d}$ the corresponding core states obtained by sequentially applying the same recurrence from left to right. We measure convergence using the RMSE between consecutive Jacobi states and the RMSE between Jacobi and sequential states:

$$\Delta_k = \frac{\|H^{(k)} - H^{(k-1)}\|_{\text{F}}}{\sqrt{Td}}, \qquad d_k = \frac{\|H^{(k)} - H_{\text{seq}}\|_{\text{F}}}{\sqrt{Td}}. \tag{8}$$

Figure 6 shows clear convergence as the number of Jacobi iterations increases. Validation loss decreases from 2.2180 at $k = 0$ to 1.9339 at $k = 3$ and remains nearly unchanged afterwards, reaching 1.9365 at $k = 5$. Meanwhile, both Δ_k and d_k continue to decrease toward zero: they fall below 10^{-4} by $k = 16$ and reach approximately 5.5×10^{-7} at $k = 32$. These results show that the Jacobi states progressively converge to a stable solution that closely matches sequential inference. Importantly, although P2N is trained using only random $K \in \{2, 3\}$ Jacobi iterations, this training scheme is sufficient to learn stable and convergent iterative dynamics.

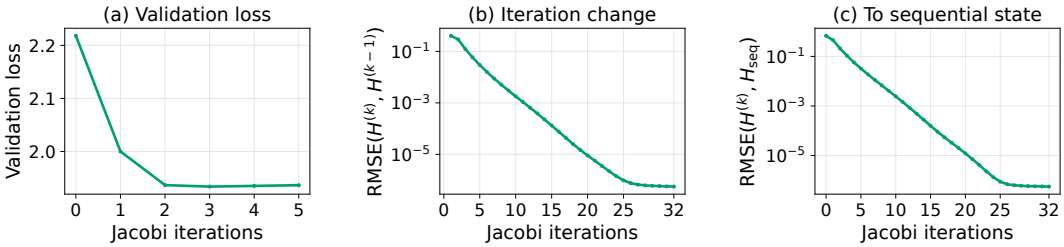


Figure 6: **Convergence diagnostics of Jacobi iteration on the 1.5B P2N model. (a)** Validation loss on 4096 eval samples under different numbers of Jacobi iterations. **(b)** RMSE between consecutive core states, $\Delta_k = \|H^{(k)} - H^{(k-1)}\|_{\mathrm{F}}/\sqrt{Td}$. **(c)** RMSE between Jacobi and sequential core states, $d_k = \|H^{(k)} - H_{\mathrm{seq}}\|_{\mathrm{F}}/\sqrt{Td}$. Validation loss quickly stabilizes, while both state errors continue decreasing toward zero as additional iterations are applied.

Table 3: **Recurrent core depth.** Validation loss and estimated training compute relative to Vanilla on 600M model trained for 120B tokens. Compute ratios use the layer-based approximation $1+2.5C/21$, where C is the number of recurrent layers. The middle-third core provides most of the gain from the 11-layer core at a lower estimated compute cost.

Method	Core layers	Relative compute	Val. loss ↓
Vanilla	0	1.00×	2.0859
Middle $1/3$ (P2N)	7	1.83×	2.0511
Middle $\approx 1/2$	11	2.31×	**2.0453**
Full	21	3.50×	2.1046

5 Ablation Studies

We conduct a series of ablations on the 21-layer, 600M dense model trained on 120B tokens from the Pile. Unless otherwise specified, we use direct sum and a middle-third recurrent core, with K sampled uniformly from $\{2, 3\}$ for each training batch. Here, K counts Jacobi iterations after the warm pass. Each ablation changes only the factor under study.

5.1 Recurrent Core Depth

We first vary the recurrent core depth to examine the trade-off between model quality and training compute. We compare Vanilla with recurrence over the middle 7 layers, the middle 11 layers (approximately half the network), and all 21 layers.

As shown in Table 3, the middle-third core reduces validation loss from 2.0859 to 2.0511. Extending the core to 11 layers provides only a further 0.0058 reduction while requiring approximately 26% more estimated training compute. Full-network recurrence is the most expensive configuration yet performs worse than Vanilla. The middle-third core therefore offers a favorable quality–compute trade-off, capturing most of the improvement achieved by the 11-layer core at a lower training cost.

5.2 Recurrent Core Placement

With the core depth fixed at seven layers, we then compare placement in the first, middle, and last thirds of the network. This isolates the effect of placement under the same recurrent-compute budget.

Table 4 shows that the middle-third placement achieves the lowest validation loss. Placing the core in the first third yields only a small improvement over Vanilla, whereas placing it in the last third performs worse than Vanilla. Together with the depth ablation, these results support applying recurrence to the middle third of the network.

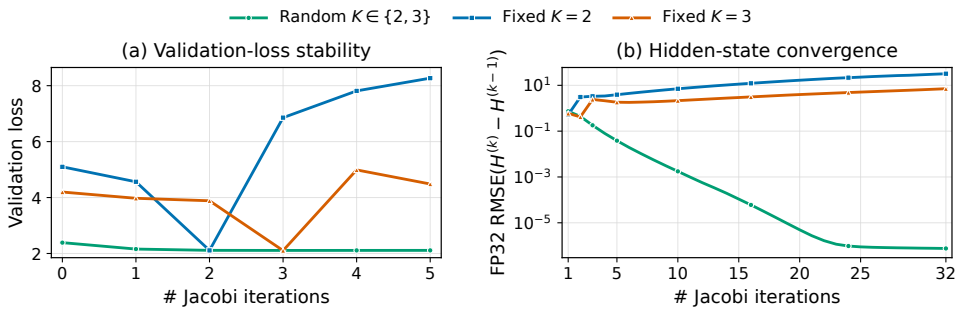


Figure 7: **Effect of sampling the number of Jacobi iterations.** **(a)** Validation loss over $k = 0, \ldots, 5$ evaluation-time Jacobi iterations. **(b)** RMSE between consecutive core states, Δ_k, over $k = 1, \ldots, 32$ iterations. Random $K \in \{2, 3\}$ yields stable loss and state differences approaching zero, whereas fixed-K models deteriorate beyond their training iteration count.

Table 4: **Recurrent core placement.** Validation loss and loss gaps relative to Vanilla on 600M model trained for 120B tokens to ablate different seven-layer core placements. Among recurrent configurations with the same compute budget, the middle-third placement achieves the lowest loss.

Core placement	Val. loss ↓	Loss gap
Vanilla (no recurrence)	2.0859	0.0000
First $1/3$	2.0792	-0.0067
Middle $1/3$ (P2N)	**2.0511**	-0.0348
Last $1/3$	2.0969	$+0.0110$

5.3 Sampling Jacobi Iterations

We compare training with fixed $K = 2$ or $K = 3$ Jacobi iterations against our default random $K \in \{2, 3\}$ schedule. Following the convergence analysis in subsection 4.4, we examine validation loss and the RMSE between consecutive core states, Δ_k, as the number of evaluation-time Jacobi iterations increases.

As shown in Figure 7, random sampling yields nearly stable validation loss while Δ_k decreases toward zero. In contrast, fixed-K models specialize to their training iteration count: beyond that count, validation loss rises sharply and Δ_k grows, indicating that the hidden states do not stabilize within the evaluated range. These results support randomizing the number of training-time Jacobi iterations to avoid depth specialization and promote stable convergence.

6 Conclusion

We introduced P2N, a parameter-free latent feedback mechanism that allows each token to build on the previous token's deep computation by adding its core output to the current core input. Jacobi iterations preserve token parallelism during training, while restricting recurrence to the middle third of the network limits the additional training cost. Experiments on Qwen-style dense models from 150M to 1.5B parameters demonstrate consistent improvements in language modeling, with gains extending to downstream tasks and mathematical reasoning after supervised fine-tuning. Architectural ablations support our choice of direct addition and a middle-third core, while convergence diagnostics show that randomized Jacobi training promotes stable iterative behavior beyond the training iteration counts. Decoding benchmarks further show that these improvements come with negligible additional per-token latency. Together, these findings highlight the value of reusing previously completed deep computation across tokens to improve language model capabilities while maintaining efficient autoregressive decoding.

AI USE STATEMENT

Generative AI tools were used to assist with manuscript drafting, language editing and translation, LaTex formatting, and the summarization and interpretation of experimental results. We also use GPT-5.5 as an automated judge for MT-Bench evaluation, as described in subsubsection 4.2.2. The authors reviewed all AI-assisted content and take full responsibility for the final manuscript, including its methods, experimental results, claims, and associated artifacts.

ETHICS STATEMENT

This work studies architectural and training modifications to language models using existing datasets and benchmarks, without collecting new human-subject data. As with other language models, the resulting models may generate biased, harmful, or privacy-sensitive content. The proposed method is not a safety intervention, and appropriate safety evaluation and safeguards remain necessary before deployment.

REPRODUCIBILITY STATEMENT

section 3 specifies the recurrence, boundary handling, Jacobi schedule, gradient flow, and training-compute accounting. Appendix B provides model configurations and pretraining hyperparameters, while Appendix C documents the supervised fine-tuning data and optimization settings. Evaluation protocols and decoding benchmark settings are described in section 4 and Appendix D. section 5 reports controlled ablations of the main design choices.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 4895–4901, 2023.

Ido Amos, Avi Caciularu, Mor Geva, Amir Globerson, Jonathan Herzig, Lior Shani, and Idan Szpektor. Latent reasoning with supervised thinking states. *arXiv preprint arXiv:2602.08332*, 2026.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. *Advances in neural information processing systems*, 35:11079–11091, 2022.

Ziyang Cai, Xingyu Zhu, Yihe Dong, Yinghui He, and Sanjeev Arora. Tˆ 2mlr: Transformer with temporal middle-layer recurrence. *arXiv preprint arXiv:2607.15178*, 2026.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 2978–2988, 2019.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Angela Fan, Thibaut Lavril, Edouard Grave, Armand Joulin, and Sainbayar Sukhbaatar. Addressing some limitations of transformers with feedback memory. *arXiv preprint arXiv:2002.09402*, 2020.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *Advances in Neural Information Processing Systems*, 38: 41340–41391, 2026.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pp. 11398–11442. PMLR, 2023.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

Zeyi Huang, Xuehai He, LiLiang Ren, Yiping Wang, Baolin Peng, Hao Cheng, Shuohang Wang, Pengcheng He, Jianfeng Gao, Yong Jae Lee, et al. Latent recurrent transformer: Architecture exploration, training strategies, and scaling behavior. *arXiv preprint arXiv:2605.26797*, 2026.

DeLesley Hutchins, Imanol Schlag, Yuhuai Wu, Ethan Dyer, and Behnam Neyshabur. Block-recurrent transformers. *Advances in neural information processing systems*, 35:33248–33261, 2022.

Dongseong Hwang, Weiran Wang, Zhuoyuan Huo, Khe Chai Sim, and Pedro Moreno Mengibar. Transformerfam: Feedback attention is working memory. *arXiv preprint arXiv:2404.09173*, 2024.

Da Ju, Stephen Roller, Sainbayar Sukhbaatar, and Jason E Weston. Staircase attention for recurrent processing of sequences. *Advances in neural information processing systems*, 35:13203–13213, 2022.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations. In *Proceedings of the 2017 conference on empirical methods in natural language processing*, pp. 785–794, 2017.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Michael C Mozer, Shoaib Ahmed Siddiqui, Danny Sawyer, Sunny Sanyal, and Rosanne Liu. Recirculation. *arXiv preprint arXiv:2608.17981*, 2026.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–15, 2021.

Costin-Andrei Oncescu, Depen Morwani, Samy Jelassi, Alexandru Meterez, Mujin Kwun, and Sham Kakade. The recurrent transformer: Greater effective depth and efficient decoding. *arXiv preprint arXiv:2604.21215*, 2026.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc-Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 1525–1534, 2016.

Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019.

Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

DiJia Su, Hanlin Zhu, Yingchen Xu, Jiantao Jiao, Yuandong Tian, and Qinqing Zheng. Token assorted: Mixing latent and text tokens for improved language model reasoning. *arXiv preprint arXiv:2502.03275*, 2025.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Jihoon Tack, Jack Lanchantin, Jane Dwivedi-Yu, Andrew Cohen, Ilia Kulikov, Janice Lan, Shibo Hao, Yuandong Tian, Jason E Weston, and Xian Li. Llm pretraining with continuous concepts. In *International Conference on Learning Representations*, volume 2026, pp. 54214–54231, 2026.

Mohan Tang and Sidi Lu. Turbo connection: Reasoning as information flow from higher to lower layers. *arXiv preprint arXiv:2602.17993*, 2026.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Stanford alpaca: An instruction-following llama model, 2023.

Jayden Teoh, Manan Tomar, Kwangjun Ahn, Edward S Hu, Tim Pearce, Pratyusha Sharma, Akshay Krishnamurthy, Riashat Islam, Alex Lamb, and John Langford. Next-latent prediction transformers learn compact world models. *arXiv preprint arXiv:2511.05963*, 2025.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Johannes Welbl, Nelson F Liu, and Matt Gardner. Crowdsourcing multiple choice science questions. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pp. 94–106, 2017.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In *International conference on learning representations*, volume 2024, pp. 42195–42214, 2024.

Zhenrui Yue, Bowen Jin, Huimin Zeng, Honglei Zhuang, Zhen Qin, Jinsung Yoon, Lanyu Shang, Jiawei Han, and Dong Wang. Hybrid latent reasoning via reinforcement learning. *Advances in Neural Information Processing Systems*, 38:5501–5530, 2026.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4791–4800, 2019.

Boyi Zeng, He Li, Shixiang Song, Yixuan Wang, Zitong Wang, Ziwei He, Xinbing Wang, and Zhouhan Lin. Ponderlm-2: Pretraining llm with latent thoughts in continuous space. *arXiv preprint arXiv:2509.23184*, 2025.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in neural information processing systems*, 32, 2019.

Zhen Zhang, Xuehai He, Weixiang Yan, Ao Shen, Chenyang Zhao, and Xin Wang. Soft thinking: Unlocking the reasoning potential of llms in continuous concept space. *Advances in Neural Information Processing Systems*, 38:168990–169012, 2026.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

A Feedback Formulation

We examine how to combine the current core input x_t with the previous token's core output h_{t-1}, considering residual-scale treatment and learnable mixing.

Residual-scale treatment. Our default formulation directly uses $x_t + h_{t-1}$ as core input. To examine whether explicit residual-scale control is beneficial, we compare it with ***RMS matching***, which rescales the sum to match the original input's RMS, and *compute-only*, which uses $x_t + h_{t-1}$ only as the attention pre-normalization input in the core block while retaining x_t on the residual shortcut.

Learnable mixing. We also compare direct sum with channel-wise and scalar gates that learn how strongly to weight the feedback. The channel-wise variant uses $x_t + g \odot h_{t-1}$, where $g \in \mathbb{R}^d$ is initialized to $\mathbf{1}$. The scalar variant instead learns a single feedback weight shared across channels.

Table 5 shows that RMS matching and compute-only increase validation loss by 0.0032 and 0.0154, respectively, relative to direct sum. Channel-wise and scalar gates provide only small gains, reducing loss by 0.0020 and 0.0017. Thus, neither explicit scale treatment nor learnable mixing offers a compelling advantage over the simpler formulation, and we retain direct sum as our default.

Table 5: **Feedback formulation.** Validation loss on 600M model trained for 120B tokens to ablate residual-scale treatments (a) and learnable mixing strategies (b). Loss gaps are measured relative to direct sum; negative values indicate improvement. Scale treatments increase loss, while learnable gates offer only small gains.

(a) Residual-scale treatment			**(b) Learnable mixing**		
Method	Val. loss ↓	Loss gap	Method	Val. loss ↓	Loss gap
Direct sum (P2N)	**2.0511**	0.0000	Direct sum (P2N)	2.0511	0.0000
RMS matching	2.0543	+0.0032	Channel-wise gate	**2.0491**	−0.0020
Compute-only	2.0665	+0.0154	Scalar gate	2.0494	−0.0017

B Model Configurations and Training Details

B.1 Model Architecture

Common backbone. All models use a Qwen3-style decoder-only Transformer architecture (Vaswani et al., 2017; Yang et al., 2025) and are trained from random initialization. The backbone uses grouped-query attention (GQA) (Ainslie et al., 2023), Q/K normalization, pre-RMSNorm (Zhang & Sennrich, 2019), SwiGLU activations with SiLU (Shazeer, 2020), and rotary position embeddings (RoPE) (Su et al., 2024). The attention projections do not use bias terms. We use a vocabulary size of 50,304 and tie the input embedding and output language-model head. Unless otherwise specified, all models use `rms_norm_eps`$= 10^{-6}$, `rope_theta`$= 10^6$, `attention_dropout`$= 0$, and `initializer_range`$= 0.02$. The attention head dimension is fixed to 64 across all model sizes.

For every model size, the Vanilla and P2N variants have exactly the same number of parameters and the same physical Transformer layers. P2N does not introduce additional parameters. Throughout the paper, Transformer layers are indexed from zero. The corresponding implementation option `p2n_core_start_layer` uses one-based indexing.

B.1.1 Dense Models

Table 6 summarizes the dense model configurations. As the model size increases, we scale depth, hidden dimension, and FFN width while preserving a middle-third P2N core.

Table 6: Dense model configurations. Vanilla and P2N use identical physical architectures and parameter counts. The P2N core is reported using zero-based layer indices.

Model	Parameters	Layers	d_{model}	d_{ff}	Q heads	KV heads	P2N core	Prefix/Core/Suffix
150M	149,541,120	12	768	3,328	12	4	L4–L7	4/4/4
300M	303,203,200	15	1,024	4,608	16	4	L5–L9	5/5/5
600M	604,627,328	21	1,280	5,632	20	5	L7–L13	7/7/7
1B	1,011,626,496	24	1,536	7,168	24	6	L8–L15	8/8/8
1.5B	1,496,092,288	27	1,792	8,192	28	7	L9–L17	9/9/9

Table 7: MoE model configurations. "Active parameters" denote the parameters activated for each token rather than the total number of model parameters. All models use 64 routed experts with Top-8 routing.

Model	Active params.	Total params.	Layers	d_{model}	$d_{\text{ff}}^{\text{expert}}$	Q/KV heads	P2N core	Prefix/Core/Suffix
A0.15B	150,130,944	794,218,752	12	768	416	12/4	L4–L7	4/4/4
A0.3B	304,186,240	1,790,542,720	15	1,024	576	16/4	L5–L9	5/5/5
A0.6B	606,347,648	3,785,499,008	21	1,280	704	20/5	L7–L13	7/7/7
A1B	1,013,985,792	6,563,049,984	24	1,536	896	24/6	L8–L15	8/8/8

B.1.2 Mixture-of-Experts Models

We additionally evaluate P2N on sparse Mixture-of-Experts (MoE) models constructed from the corresponding dense backbones. Every Transformer layer contains 64 routed experts and activates the top 8 experts for each token, with no additional shared expert. The intermediate dimension of each individual expert is set to one eighth of the FFN width of the corresponding dense model. Consequently, the aggregate FFN width of the eight activated experts matches that of the dense counterpart. The active parameter count is therefore closely matched to the corresponding dense scale, with a small increase primarily due to the routing module.

The MoE models retain the same attention configuration, normalization scheme, positional encoding, vocabulary, tied embeddings, and P2N core placement as their dense counterparts. We use Qwen-style unscaled output-projection initialization for the MoE models.

B.2 Training Setup

All models are pretrained from scratch using the standard causal next-token prediction objective. Training examples have a sequence length of 2,048 tokens. We use a global batch size of 1,024 sequences, corresponding to 2,097,152 training tokens per optimizer step, with a per-device micro-batch size of 8 sequences.

For the dense models, we use AdamW (Loshchilov & Hutter, 2017). For the MoE models, we use the Megatron Adam optimizer with the distributed optimizer (Narayanan et al., 2021). Both settings use $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, weight decay of 0.1, and global gradient-norm clipping at 1.0.

We employ a cosine learning-rate schedule with a linear warmup over the first 5% of training, followed by cosine decay to 10% of the peak learning rate. For the 150M, 300M, and 600M dense models and their corresponding MoE variants, we use a peak learning rate of 1.5×10^{-3}. For the 1B dense model and A1B MoE model, the peak learning rate is 1.0×10^{-3}, while the 1.5B dense model uses 8.0×10^{-4}.

Training uses BF16 computation throughout. For the MoE experiments, the main model parameters and accumulated gradients are maintained in FP32 by the Megatron optimizer. We use a random seed of 42 for all experiments. Activation recomputation is disabled in the reported runs.

Unless explicitly varied in an ablation, Vanilla and P2N models at the same scale use the same architecture, global batch size, optimizer hyperparameters, learning-rate schedule, and training protocol. Thus, the comparison changes only the P2N computation defined in section 3, while keeping the underlying parameterization and optimization setup fixed.

C SUPERVISED FINE-TUNING DETAILS

We conduct supervised fine-tuning (SFT) for two settings: mathematical reasoning on GSM8K and instruction tuning on Alpaca followed by evaluation on MT-Bench. For each setting, the Vanilla and P2N models are independently initialized from their corresponding pretrained checkpoints and fine-tuned with identical data and optimization hyperparameters. All experiments use full-parameter SFT. For P2N, we retain the same architecture and training recurrence as in pretraining, with the number of Jacobi updates randomly sampled from $K \in \{2, 3\}$ for each training batch.

C.1 GSM8K SUPERVISED FINE-TUNING

For mathematical reasoning, we fine-tune the models on the official GSM8K training split, which contains 7,473 examples. The official test split of 1,319 examples is reserved exclusively for evaluation and is not used during SFT. Each example is formatted as `Question:{question}\nAnswer:`, followed by the original GSM8K response as the target. The target retains both the intermediate reasoning trace and the final answer indicated by the `####` marker. Only response tokens contribute to the training objective; prompt tokens are masked from the loss.

Both Vanilla and P2N models are trained for 3 epochs with a global batch size of 128 and a maximum sequence length of 2,048 tokens. We do not use sequence packing. Optimization uses AdamW with a peak learning rate of 2×10^{-5}, $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, zero weight decay, and gradient clipping with a maximum global norm of 1.0. The learning rate follows a cosine schedule with a warmup ratio of 0.03. Training is performed in BF16 with a random seed of 42.

C.2 ALPACA SUPERVISED FINE-TUNING AND MT-BENCH EVALUATION

For instruction-following evaluation, we first perform SFT on the English Alpaca dataset and subsequently evaluate the resulting models on MT-Bench. MT-Bench itself is used only for evaluation and is not included in the training data.

We use a fixed snapshot of the `llamafactory/alpaca_en` dataset containing 51,699 training examples. We adopt the standard instruction-tuning format and optimize only over response tokens, with the instruction and input prompt masked from the loss. Models are trained for 3 epochs with a global batch size of 128, a maximum sequence length of 2,048 tokens, and sequence packing disabled.

We use AdamW with a peak learning rate of 2×10^{-5}, $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, zero weight decay, and a maximum gradient norm of 1.0. The learning rate follows a cosine schedule with a warmup ratio of 0.03. Training uses BF16 precision and a random seed of 42. As with GSM8K, Vanilla and P2N models use exactly the same SFT data and optimization settings.

After Alpaca SFT, we evaluate instruction-following and multi-turn conversational ability on MT-Bench. Models generate responses to the standard MT-Bench questions under the same generation protocol, and the resulting answers are evaluated by `GPT-5.5` as the judge model. The judge is used only for post-training evaluation and does not participate in SFT.

D ADDITIONAL EXPERIMENTS ON MIXTURE-OF-EXPERTS MODELS

We extend our experiments to sparse Mixture-of-Experts (MoE) models to examine whether the benefits of P2N generalize beyond dense Transformers. Following the organization of section 4, we evaluate scaling behavior, language modeling, downstream performance, decoding efficiency, and the effect of additional Jacobi iterations.

D.1 LARGE-SCALE PRETRAINING

We pretrain Qwen-style MoE models from scratch on the Pile, covering active parameter scales of A0.15B, A0.3B, A0.6B, and A1B. The prefix "A" denotes the number of parameters activated per token rather than the total model parameter count. Each Transformer layer contains 64 routed experts and activates the top 8 experts per token, without an additional shared expert. For each scale, Vanilla

and P2N share the same backbone architecture, active and total parameter counts, training data, token budget, and optimization settings. As in the dense experiments, P2N applies feedback to the middle third of the network and samples $K \in \{2, 3\}$ Jacobi updates during training. Detailed model configurations and training hyperparameters are provided in Appendix B.

D.1.1 Scaling Behavior

Figure 8 summarizes the scaling behavior of the MoE models. Panels (a) and (c) report final Pile validation loss and average downstream accuracy across active parameter scales, respectively, while panel (b) tracks training loss for the matched A1B models. The downstream score is the unweighted mean of raw accuracies, using the same aggregation as Table 8. All horizontal axes use a logarithmic scale.

Across all four active parameter scales, P2N achieves lower validation loss and higher average downstream accuracy than Vanilla. The validation-loss reductions range from 0.0219 to 0.0275, while the accuracy improvements range from 0.92 to 1.88 percentage points. At A1B, final validation loss decreases from 1.8259 to 1.8021, and average downstream accuracy increases from 59.92% to 61.80%. For the matched A1B training runs, P2N also maintains lower training loss at every reported token budget from 10B to 200B tokens. These results demonstrate consistent improvements across active parameter scales and the sampled stages of pretraining.

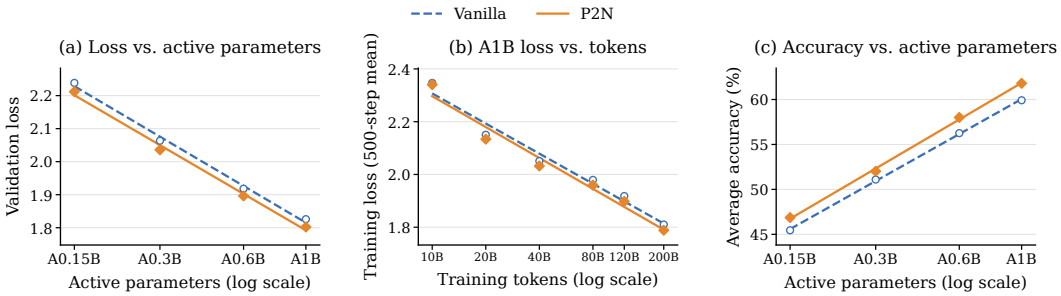


Figure 8: **Scaling behavior of MoE models. (a)** Final Pile validation loss across active parameter scales, from A0.15B to A1B. **(b)** Training loss over training tokens for the matched A1B models, with each point averaged over 500 consecutive optimizer steps. **(c)** Mean raw accuracy over all nine downstream task configurations across active parameter scales. Lower is better in panels (a) and (b), and higher is better in panel (c). P2N achieves lower final validation loss, lower training loss at the reported token budgets, and higher average downstream accuracy.

D.1.2 Language Modeling Ability

We evaluate the A0.6B and A1B MoE models on Pile validation, WikiText, and the OpenAI and Standard variants of LAMBADA. As shown in Figure 9, P2N consistently reduces perplexity on all four benchmarks at both active parameter scales.

Relative to Vanilla, the perplexity reductions are 2.2–2.4% on Pile, 2.8–3.2% on WikiText, 7.7–9.4% on LAMBADA OpenAI, and 15.0–15.7% on LAMBADA Standard. At A1B, for example, Pile perplexity decreases from 6.2081 to 6.0621, while LAMBADA Standard perplexity decreases from 6.6722 to 5.6217. The largest relative improvements occur on the two LAMBADA variants, consistent with the pattern observed in the dense experiments. These results show that the language-modeling gains extend to sparse MoE architectures.

D.2 Downstream Task Evaluation

We further assess whether the improvements observed during MoE pretraining translate into better downstream performance. We evaluate general downstream tasks at the A0.6B and A1B scales and mathematical reasoning at the A1B scale.

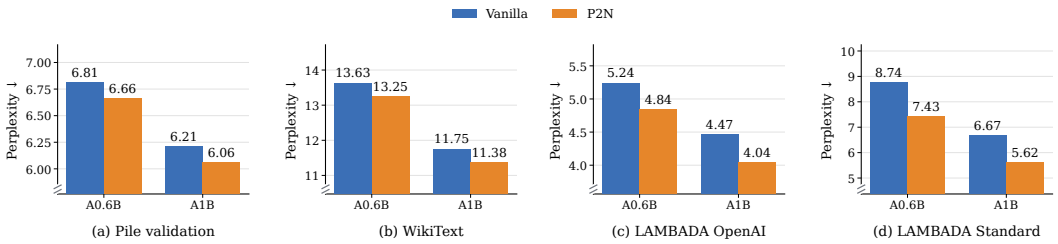


Figure 9: **Language modeling at the A0.6B and A1B scales.** Perplexity of Vanilla and P2N MoE models on **(a)** Pile validation, **(b)** WikiText, **(c)** LAMBADA OpenAI, and **(d)** LAMBADA Standard; lower is better. Numbers above the bars denote perplexity values. P2N achieves lower perplexity at both scales on every benchmark, with the largest relative improvements on the LAMBADA variants.

D.2.1 General Downstream Tasks

We evaluate the MoE models on the same nine task configurations as in subsubsection 4.2.1: LAMBADA-OpenAI, LAMBADA-Standard, PIQA, WinoGrande, SciQ, HellaSwag, ARC-Easy, ARC-Challenge, and RACE. Table 8 reports raw accuracy and the unweighted average over all nine configurations.

At A0.6B, P2N improves average accuracy from 56.26% to 58.00%, a gain of 1.75 percentage points, with improvements on all nine task configurations. At A1B, average accuracy increases from 59.92% to 61.80%, a gain of 1.88 percentage points. The A1B model improves on eight configurations and matches Vanilla on RACE. The largest absolute gains at both scales occur on LAMBADA-Standard, reaching 4.50 and 4.37 percentage points at A0.6B and A1B, respectively. These results demonstrate that the benefits of P2N extend to general downstream performance in MoE models.

Table 8: **Downstream performance at the A0.6B and A1B scales.** Raw accuracy (%) of Vanilla and P2N MoE models. Avg. denotes the unweighted mean over the nine task configurations. Bold indicates the best score within each matched scale, including ties. P2N improves average accuracy at both scales.

Scale	Method	LAMBADA		PIQA	Wino Grande	SciQ	Hella Swag	ARC		RACE	Avg.
		OpenAI	Standard					Easy	Challenge		
A0.6B	Vanilla	64.43	52.61	72.85	59.04	88.60	44.12	63.05	26.79	34.83	56.26
A0.6B	P2N	**66.68**	**57.11**	**73.18**	**61.25**	**90.60**	**45.08**	**63.51**	**29.69**	**34.93**	**58.00**
A1B	Vanilla	67.77	58.74	75.03	62.59	90.20	48.44	68.06	32.94	**35.50**	59.92
A1B	P2N	**69.84**	**63.11**	**75.79**	**65.35**	**92.30**	**49.62**	**69.70**	**34.98**	**35.50**	**61.80**

D.2.2 Mathematical Reasoning

We fine-tune the pretrained A1B MoE models on GSM8K following the SFT settings in Appendix C. We evaluate strict final-answer exact match (EM) on all 1,319 test examples under zero-shot and eight-shot prompting. Fine-tuning settings are matched between Vanilla and P2N, as are the prompting formats, decoding settings, and answer-extraction procedures within each evaluation setting.

As shown in Table 9, P2N improves zero-shot EM from 21.68% to 27.98%, a gain of 6.29 percentage points. Under eight-shot prompting, EM increases from 20.02% to 26.38%, a gain of 6.37 percentage points. These results demonstrate improved mathematical reasoning performance under both prompting settings.

D.3 Decoding Efficiency

During autoregressive decoding, P2N reuses the previous token's core output through a vector addition, without repeating any Transformer layer. We benchmark the A1B MoE models on a

Table 9: **Mathematical reasoning at the A1B MoE scale.** GSM8K test strict final-answer EM (%) under zero-shot and eight-shot prompting after supervised fine-tuning. Gains are measured in percentage points relative to Vanilla and calculated before rounding. P2N improves accuracy in both settings.

Scale	Method	Zero-shot		Eight-shot	
		EM (%) ↑	Gain (pp.)	EM (%) ↑	Gain (pp.)
A1B	Vanilla	21.68	–	20.02	–
A1B	P2N	**27.98**	+6.29	**26.38**	+6.37

Table 10: **Autoregressive decoding efficiency of the A1B MoE model on a single GPU.** Each entry is the mean of four timed runs generating 2,048 tokens per sequence after one untimed warm-up run. Each run uses a 256-token prompt. Relative latency is normalized to Vanilla TPOT.

Batch	TPOT (ms)		Throughput (tok/s)		Rel. latency
size	Vanilla	P2N	Vanilla	P2N	P2N / Vanilla
1	70.454	70.470	14.2	14.2	1.000×
4	70.537	70.464	56.7	56.8	0.999×
16	71.038	71.013	225.2	225.3	1.000×
32	70.905	70.923	451.3	451.2	1.000×

single GPU using BF16, SDPA, and the standard KV cache. Each run uses a 256-token prompt and generates 2,048 tokens per sequence. For each configuration, we perform one untimed warm-up run and report the mean over four subsequent timed runs. Prompt prefill is excluded from the decoding measurements.

Table 10 reports time per output token (TPOT), aggregate decoding throughput, and relative latency. Across batch sizes from 1 to 32, P2N maintains nearly identical TPOT and throughput to Vanilla, with relative latency remaining within 0.1% of the baseline under all reported settings. These results show that the parameter-free feedback introduces negligible decoding overhead for the A1B MoE model.

D.4 Convergence Diagnostics of Jacobi Iteration

We examine evaluation loss as a function of the number of Jacobi iterations on the A1B P2N MoE model. As shown in Figure 10, loss decreases over the first few updates and changes only slightly thereafter. In particular, it decreases from 2.2034 at $K = 0$ to 1.8656 at $K = 2$ and remains between 1.8625 and 1.8680 for $K = 2, \ldots, 5$. Thus, additional iterations beyond the training range produce only small changes in evaluation loss over the tested range.

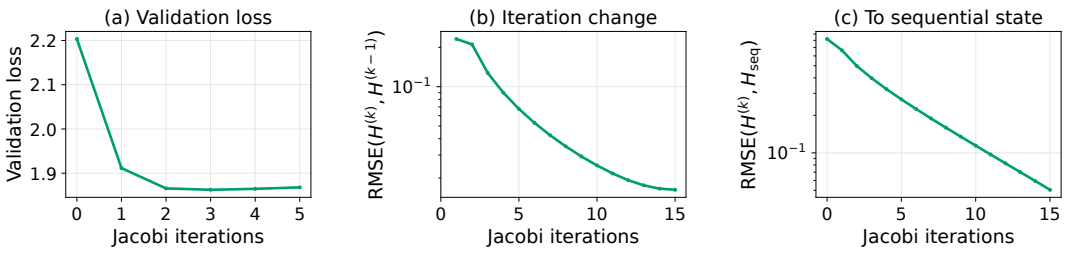


Figure 10: **Evaluation loss under additional Jacobi iterations at the A1B MoE scale.** Loss is evaluated on 4,096 examples for $K = 0, \ldots, 5$, where K counts updates after the initial warm pass. After the initial reduction, evaluation loss changes only slightly as further iterations are applied.

E LIMITATIONS AND FUTURE WORK

Jacobi iteration preserves parallel training but increases the estimated training compute to approximately $1.83\times$ that of Vanilla under our default configuration and layer-based compute accounting. In addition, the dependence on the previous token's hidden state means that efficient integration with multi-token prediction and speculative decoding requires further investigation. We are actively exploring more efficient training schedules and compatible decoding strategies to support practical deployment.