RETHINKING PRE-PRETRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

We report that allocating compute to synthetic pre-pretraining (PPT) never outperforms pure natural-language training in compute-matched experiments, across four PPT curricula (k-Dyck, neural cellular automata (NCA), declarative reference corpora, and code) and three training regimes: data-constrained, data-unconstrained, and a high token-to-parameter-ratio regime (881M parameters trained on 50B tokens). When natural language is scarce, repeated epochs over a 100M-token corpus beat every synthetic curriculum on natural-language validation loss, with no consistent downstream accuracy advantage from PPT. We find that broader PPT gains reported in prior work can be attributed to the weight initialization scheme. In an NCA reproduction, we recover a 5.4% PPT advantage under the original initialization; changing the initialization improves both methods (the no-PPT baseline by 9.0%), leaving them indistinguishable. However, when PPT compute is *additive* to, rather than *competing* with, the natural-language budget, PPT gains persist even after improving the baseline initialization at small natural-language budgets, but vanish by 1B natural-language tokens. Finally, knowledge acquired during PPT fails to survive subsequent natural-language training; interleaving the same data throughout training preserves it, though it does not significantly improve performance. Our results suggest limited benefit from synthetic pre-pretraining as a distinct training stage and motivate rethinking how such synthetic data are used in language model training.

1 INTRODUCTION

Rich computational structure can be learned from naturally occurring text. The benefits of scaling training on such tokens have yet to hit a ceiling, though increasingly large amounts of data are required for further improvements in performance (Kaplan et al., 2020). By contrast, humans acquire language from far less linguistic experience (Warstadt et al., 2023), and their reasoning abilities are largely dissociable from linguistic proficiency (Fedorenko et al., 2024). This contrast has motivated recent work to ask whether useful linguistic and computational priors can instead be acquired from synthetic data. Beyond making language learning more sample-efficient, synthetic distributions supply effectively unlimited training data once high-quality natural language becomes scarce.

This idea is operationalized in prior work through pre-pretraining (PPT): a training phase on synthetic data that precedes conventional pretraining (PT) and is intended to produce an initialization with useful priors for learning language. Different synthetic distributions target different computational priors: k-Dyck encourages hierarchical dependency tracking (Hu et al., 2025), neural cellular automata (NCA) expose models to latent transition rules and long-range dependencies (Lee et al., 2026), and procedural tasks train operations such as copying, sorting, and retrieval (Jiang et al., 2026). These approaches have reported gains in natural-language modeling and context recall, including up to 8.6% lower perplexity for NCA, both when PPT is given additional compute on top of a fixed natural-language budget and when PPT and natural-language training share a fixed total compute budget.

Although prior findings establish that PPT can improve the performance of a fixed training recipe, it remains an open question whether it is a better use of training resources than natural-language training. We therefore revisit synthetic pre-pretraining, asking when it helps, when it does not, and when it is a better use of compute than natural-language training. We study three synthetic distributions (k-Dyck, NCA, and declarative reference corpora (DeclRef)), alongside code, which

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.

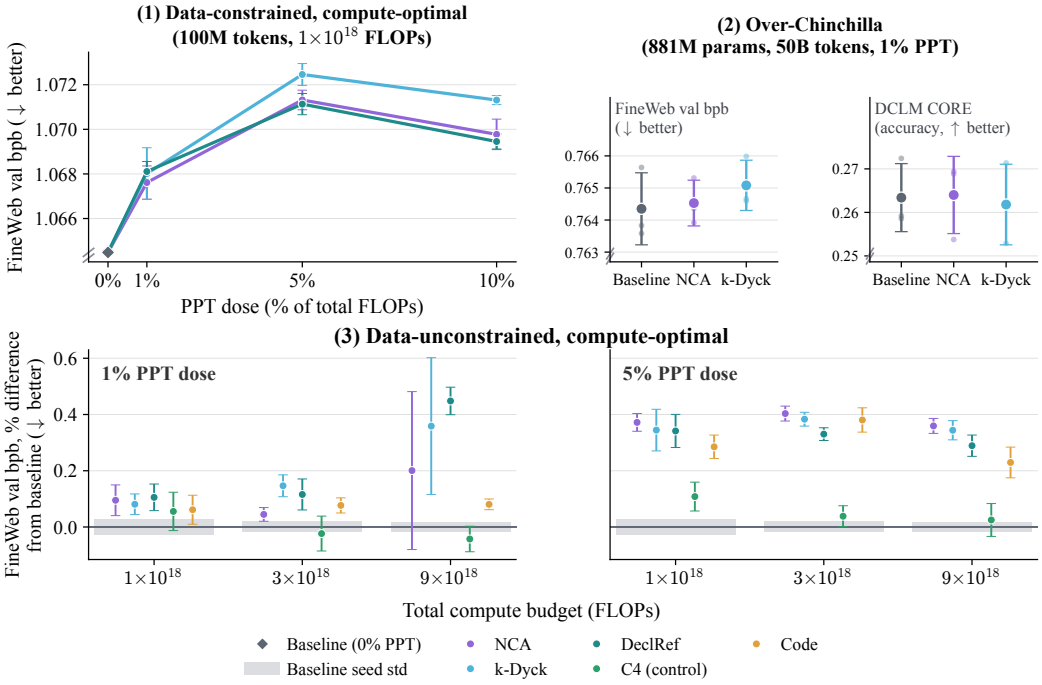

Figure 1: **Across all regimes, allocating compute to PPT never beats spending it on natural language.** (1) Data-constrained: FineWeb validation bits per byte (BPB) against the fraction of a fixed 1×10^{18}-FLOP budget spent on PPT, with repetition over 100M natural-language tokens. Optimal model sizes range from 561M to 708M parameters. (2) Over-Chinchilla: an 881M-parameter model trained on 50B tokens, $\sim12\times$ the compute-optimal token count, scored on FineWeb BPB and DCLM CORE. The PPT methods spend 0.5B tokens on PPT and 49.5B on FineWeb. (3) Data-unconstrained: the optimal model size found at each budget varies with the PPT method, from 179M to 186M parameters at 1×10^{18} FLOPs, 252M to 336M at 3×10^{18}, and 438M to 561M at 9×10^{18}.

prior work has found can improve downstream reasoning beyond coding tasks (Aryabumi et al., 2025).

We begin by asking whether synthetic pre-pretraining can provide useful inductive biases when it is not competing with natural-language training for compute. In this additive-compute setting, where PPT is added on top of a fixed natural-language budget, we first reproduce a prior NCA result as a positive control before testing the setting more broadly. We then ask whether PPT is worth its compute when it competes directly with natural-language training for a fixed FLOP budget. We evaluate this compute-matched condition across three regimes: a data-constrained setting, where the alternative is repeated training on the same 100M natural-language tokens; a data-unconstrained setting, which tests whether any PPT recipe remains competitive as training compute scales; and a high token-to-parameter ratio setting, which probes whether PPT becomes useful far beyond the usual compute-optimal frontier. Our findings are as follows:

1. **Gains from PPT are sensitive to the baseline initialization scheme.** We reproduce a previously reported NCA result under its original setup, recovering a 5.4% perplexity improvement from PPT. However, changing the token embedding initialization from $\mathcal{N}(0, 1)$ to $\mathcal{N}(0, 0.02)$ closes the gap between NCA and the no-PPT baseline.
2. **Additive compute: PPT can help at small natural-language budgets.** When each method is tuned over initialization schemes, k-Dyck and NCA improve over the best no-PPT baseline at 50M and 100M natural-language tokens (budgets reported to parallel the amount of language humans may learn from, Warstadt et al., 2023). However, by 1B tokens the advantage disappears. This suggests synthetic PPT can instill useful inductive biases, but its benefits diminish at larger natural-language budgets.

3. **Compute-matched: natural-language training performs as well as or better than PPT across all compute-matched regimes.** This includes the strongest scarcity case, where multiple epochs on 100M unique natural-language tokens beat every synthetic curriculum. The same holds on most of our downstream evaluations, both DCLM CORE (Li et al., 2024) and BLiMP (Warstadt et al., 2020), with the exceptions confined to our smallest compute budget (Table 1).
4. **Mechanism: learned PPT priors may not survive the phase transition to natural language.** We find that gains acquired during PPT on the PPT distribution disappear over the PPT-to-PT phase boundary. Uniformly interleaving the same data throughout training instead preserves performance on those distributions, but we do not find significant improvement in natural-language validation loss.

These findings highlight the surprising effectiveness of naturally occurring data as a training signal: even when unique natural-language tokens are scarce, repeated training on them remains more useful than training on the synthetic corpora we test. Importantly, we do not take this to imply that synthetic data cannot help. Instead, our results point to front-loading synthetic data as a poor way to use it, since what is learned during PPT is progressively lost once natural-language training begins. It may instead be better integrated gradually, through interleaving that avoids the abrupt distribution shift between PPT and PT. Our results therefore motivate rethinking the PPT phase, rather than ruling out synthetic data as a training corpus.

2 Background & Related Work

2.1 Pre-pretraining

All prior work in synthetic pre-pretraining shares a common hypothesis: training on a synthetic distribution instills language-modeling priors that transfer to natural language. The corpora used often contain no memorizable content, so the next-token objective reduces to inferring the governing structure from the context, which would intuitively be useful for developing in-context learning capabilities. PPT corpora in prior work can be divided broadly into three categories: formal languages (Hu et al., 2025; Mita et al., 2026), algorithmic tasks (Wu et al., 2022; Jiang et al., 2026), and data sampled from randomly initialized neural networks (Lee et al., 2026; Guo et al., 2026). Of these, we focus on k-Dyck and NCA, for which prior work reports the largest improvements in natural-language perplexity.

Hu et al. (2025) introduce PPT on k-Dyck languages, formal languages over a vocabulary of k bracket types whose grammar requires that every opening bracket is eventually closed. For example, at $k = 2$, with vocabulary $\{$ `(`, `)`, `[`, `]` $\}$, the string `( [ ( ) ] [ ] )` is valid. They find the k-Shuffle Dyck variant performs best at $k = 128$, and it is this variant we refer to as k-Dyck throughout the paper. Lee et al. (2026) instead use NCA for PPT. A cellular automaton is a grid of cells, each holding a value from a finite vocabulary—for example, one of ten symbols. The grid evolves over discrete time steps: at each step, every cell takes a new value as a function of the cells in its immediate neighborhood, using a single update rule applied identically to every cell. This can be defined as follows: let x_t denote the state of the whole grid at time step t, then an initial state x_0 and an update rule f give $x_t = f(x_{t-1})$. An NCA replaces f with a randomly initialized convolutional neural network.

To construct an NCA PPT corpus, Lee et al. (2026) sample a random initial grid, with each cell assigned a value drawn from the fixed vocabulary, together with a randomly initialized convolutional neural network as the update rule. They then run the simulation forward for a fixed number of time steps. At this point, the resulting rollouts range from low-entropy static grids to more complex, visually interesting ones. Hence, each rollout is further filtered by a proxy-complexity metric, the gzip compression ratio, and only those falling within the highest compression band are kept. Each grid in a retained rollout is then flattened exactly as a 2D image tensor would be, and the flattened grids are concatenated in time step order into a one-dimensional token sequence for training.

The experimental protocol is largely shared across the PPT methods in the literature. The natural-language token budget and training recipe are fixed, and a synthetic data phase is prepended, with the number of PPT tokens varied to optimize the downstream natural-language loss after PT. The

usual control replaces the synthetic tokens with an equal number of natural-language tokens, and a gain is reported if PPT reaches a lower downstream loss than this control. Methods are therefore compared at equal total tokens and equal model size.

In their largest-scale experiment, training a 1.6B-parameter model, Lee et al. (2026) report that PPT on 164M NCA tokens improves perplexity by 5.7% compared to the no-PPT baseline. In Hu et al. (2025)'s largest experiment, PPT a 1B-parameter model on k-Dyck before 1.6B natural-language tokens reduces final cross-entropy loss by around 0.6% ($\sim$2% in perplexity). Both works additionally report gains on public downstream benchmarks: reasoning benchmarks for NCA like BIG-bench Lite (Lee et al., 2026), and BLiMP for k-Dyck (Hu et al., 2025). We evaluate both fronts in this work, scoring every method on BLiMP (Warstadt et al., 2020), and on the DCLM CORE suite (Li et al., 2024), which contains BIG-bench tasks, among them a subset of BIG-bench Lite. A recent study reproduced the k-Dyck gain (Hu et al., 2025) under the original conditions. However, it found that the gain did not consistently persist across all model sizes, tokenizers, and languages tested (Riazhskykh et al., 2026). These mixed results motivate our study, and since k-Dyck has already been reproduced, our own reproduction (Section 4) focuses on NCA.

2.2 Training regimes

How a compute budget should be spent depends on which resource is scarce. When natural language is the constraint, additional compute buys only repeated passes over the same tokens. Muennighoff et al. (2023) characterize this setting and find that, for their models and data, up to four epochs of repetition are close to lossless, after which the gains from additional compute rapidly diminish and new data is more useful. The slowrun benchmark (Vegesna et al., 2026) isolates the regime in its extreme form, fixing a 100M-token sample of FineWeb and leaving compute essentially unconstrained. Work in this setting has shown that heavy regularization and ensembling continue to buy substantial data efficiency well past the point at which ordinary training overfits (Kim et al., 2025). PPT is motivated by the same scarcity, so these are competing answers to one question: what to do with compute once the natural-language data has run out. When data is instead plentiful, compute becomes the constraint, which is the setting where Hoffmann et al. (2022) show that model size and token count should be scaled together. Deployed models, however, are rarely trained at that ratio, because training compute is not the only cost; once inference is accounted for, this moves the optimum toward smaller models trained on more tokens (Sardana et al., 2024; Touvron et al., 2023). These are the three compute-matched regimes we study: data-constrained, data-unconstrained, and over-Chinchilla, a comparison that prior PPT work has not explicitly made. Because the compute-optimal model size may itself depend on data quality and curriculum (Bi et al., 2024), in the data-constrained and data-unconstrained regimes we search model size separately for every method and budget.

3 Experimental Setup

The research question we explore is: **in additive and compute-matched settings, does PPT yield a measurable improvement over a natural-language baseline?** We test this question in four settings, first the additive setting, where we also ablate weight initialization, followed by three compute-matched configurations. We then describe the architecture, corpora, hyperparameter optimization, and evaluation protocol shared across all four settings.

Additive compute. We first evaluate the additive setting, where PPT is prepended to a natural-language run rather than substituting any part of its training budget (Lee et al., 2026; Hu et al., 2025; Jiang et al., 2026). We fix model size at 336M parameters, and set the natural-language PT budgets to $\{50\text{M}, 100\text{M}, 1\text{B}\}$. Then, for each PT budget, we vary the number of PPT tokens over $\{5\text{M}, 20\text{M}, 50\text{M}, 100\text{M}, 200\text{M}, 500\text{M}, 1\text{B}\}$, asking which quantity of PPT minimizes final natural-language validation loss. We ablate over three weight-initialization schemes: the scheme used by Hu et al. (2025), which initializes every parameter from a Normal distribution $\mathcal{N}(0, 0.02)$; the scheme used by Lee et al. (2026), which initializes token embeddings from the standard Normal distribution $\mathcal{N}(0, 1)$ and all remaining parameters from $\mathcal{N}(0, 0.02)$; and the spectral-based initialization of NanoChat used throughout the rest of this paper (Yang et al., 2024b; Karpathy, 2025), which draws linear-layer weights from $\mathcal{N}(0, \sigma)$ with $\sigma = \min(1, \sqrt{d_{\text{out}}/d_{\text{in}}})/\sqrt{d_{\text{in}}}$, token embeddings from $\mathcal{N}(0, 1)$, and zero-initializes the language-modeling head and all residual projection weights.

This ablation is additional to prior work, and turns out to be crucial for answering our research question.

Compute-matched. For the rest of the experiments, we instead fix the total compute budget and ask whether PPT can justify a share of it. Each experiment begins from a total training budget C, measured in FLOPs, and α giving the fraction of that budget spent on PPT. A run with α receives αC FLOPs of PPT followed by $(1-\alpha)C$ FLOPs of natural-language pretraining. Hence, a run with $\alpha = 0$ spends the entire budget on natural language. We use $\alpha \in \{0, 0.01, 0.05, 0.10\}$, with the largest percentage used only in the data-constrained regime: early experiments at the larger budgets showed no benefit that justified the extra runs, so $\alpha = 0.10$ was not run there. Note that α is a fraction of compute, not of tokens: the number of tokens each phase consumes follows from the allocation rather than being set directly. Our FLOP measurement method is covered in Appendix C.

The compute-matched experiments span three configurations. The *data-constrained* configuration fixes natural language at 100M tokens and the total budget at 1×10^{18} FLOPs. The *data-unconstrained* configuration leaves natural language abundant and sweeps over 1×10^{18}, 3×10^{18} and 9×10^{18} FLOP budgets. At each budget, we search over model size to minimize validation loss so models are trained at compute-optimality. The *over-Chinchilla* configuration trains a model of 881M parameters on 50B tokens, $\sim$57 tokens per parameter, evaluating NCA and k-Dyck, the two curricula from prior work we revisit, each at 1% (500M tokens). In all three compute-matched settings we use only the spectral-based default initialization of the NanoChat version we build on (Karpathy, 2025), as it has been well tuned by the NanoChat community and is supported by our own experiments (Figure 3).

Architecture and data. All our models are trained using a fork of NanoChat, with the Muon optimizer (Jordan et al., 2024; Liu et al., 2025; Karpathy, 2025), which updates the transformer blocks' weight matrices, while the embedding and unembedding tables are trained under AdamW. However, we find that in our smaller-scale experiments the PPT phase often does better under AdamW, while at larger scales Muon performs on par, and so we ablate AdamW, applied to all parameters, only for the additive-compute experiments and the 1×10^{18} data-constrained experiment. NanoChat sets the hidden size to $64d$ for a model of depth d, so the parameter count follows from the depth alone. Depth is therefore the only capacity-related hyperparameter we search.

For our main natural-language corpus we follow NanoChat and use FineWeb (Penedo et al., 2024). For k-Dyck and NCA, we generate the data using the same code released by prior work (Appendices A.1 and A.2). We additionally introduce DeclRef, a synthetic corpus that, like the procedural tasks of Jiang et al. (2026), targets retrieval from context (Appendix A.3). Alongside the synthetic curricula, we pre-pretrain on two real-text corpora, C4 (Raffel et al., 2023) as a natural-language control and the GitHub Code dataset (CodeParrot, 2022). The additive setting is aimed at revisiting the additive-compute claims made for the synthetic distributions of prior work, NCA and k-Dyck, so for this setting we do not test DeclRef. Our data-constrained experiments add DeclRef but leave out code and C4, since these supply additional real text, which would break the 100M-token constraint on natural language. The data-unconstrained experiments do include these other corpora, while the over-Chinchilla experiments return to NCA and k-Dyck alone. Following prior work (Lee et al., 2026), the synthetic languages are encoded with their own, far smaller vocabularies. At the phase boundary we therefore expand the vocabulary to its natural-language size and reinitialize both the token embedding table and the language-modeling head, carrying only the transformer body across; each phase has its own learning-rate schedule and the optimizer state is reset at the boundary. Code and C4 share the natural-language tokenizer, so their embedding table and head carry over unchanged. Further details are in Appendices A and D.

Hyperparameter optimization. A subset of our experiment's hyperparameters are optimized with `LLMTuna`, a language-model-based hyperparameter optimizer we release alongside this paper. `LLMTuna` optimizes held-out validation bits per byte (BPB) by proposing each successive hyperparameter configuration from a rolling context of prior trials and their results, together with any files in the codebase one may want to include as context; this is in line with recent work using language models as hyperparameter optimizers (Yang et al., 2024a; Jiang & Zhao, 2026; Karpathy, 2026a). We use it because we found it more sample-efficient than Bayesian hyperparameter opti-

mization using Optuna (Akiba et al., 2019) (see Appendix D.2). The hyperparameters searched and held fixed in each setting are detailed in Appendix D.

Evaluation. Our primary metric is BPB on a held-out split of FineWeb (∼10M tokens); hyperparameter tuning uses a separate validation set. We additionally evaluate on held-out splits of C4 and GitHub code (Appendix E), on the DCLM CORE suite (Li et al., 2024), and on BLiMP (Warstadt et al., 2020), the grammaticality benchmark used by Hu et al. (2025). DCLM CORE aggregates 22 benchmarks covering knowledge, commonsense, reading comprehension, and symbolic reasoning (Appendix E.5). Every metric is reported as the mean over three seeds, with ± values and error bars giving one sample standard deviation.

4 Additive Compute

The reproduced NCA advantage vanishes under an alternative initialization. We begin by reproducing one of the experimental setups of Lee et al. (2026), from their Section 5.1. This setup is not described in Section 3, but fits within the additive-compute regime. Under the original setup, a 400M-parameter Llama model trained on ∼9B tokens of OpenWebText, PPT on 164M NCA tokens delivers a 5.4% perplexity improvement over the no-PPT baseline. However, changing only the scale at which the embeddings are initialized (Hugging Face, 2024) improves the baseline by 9.0%—more than the reported gain itself—and applying the same initialization to the NCA method leaves the two indistinguishable (Figure 2). For more details on the reproduction see Appendix B. Note that this experiment isolates sensitivity to initialization, but does not identify the best NCA configuration under the alternative initialization, since we do not retune the number of PPT tokens in that setting. However, this experiment is meant to show how much performance gain is available on the baseline before any PPT data is introduced, rather than to establish the best achievable NCA configuration under the original training scheme. The latter is studied in the additive-compute and compute-matched experiments that follow.

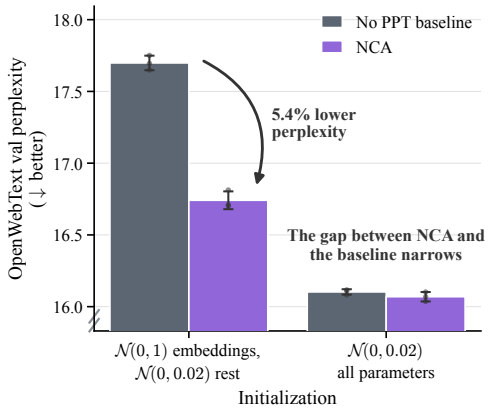

Figure 2: **Gains from NCA PPT are sensitive to baseline initialization.** A 400M-parameter Llama model trained on ∼9B tokens of OpenWebText, reproducing an NCA PPT experiment of Lee et al. (2026); we plot perplexity to match how prior work reports it. The groups differ only in weight initialization. With $\mathcal{N}(0,1)$ embeddings and $\mathcal{N}(0,0.02)$ elsewhere, NCA PPT improves perplexity over the no-PPT baseline by 5.4%. Initializing the embeddings from $\mathcal{N}(0,0.02)$ instead improves the baseline itself by 9.0%, and the two methods become inseparable despite the additional compute the NCA phase receives.

PPT helps at small natural-language budgets, but not at larger ones. We next ask whether PPT can help in the additive setting, where the natural-language budget is tightly constrained and the synthetic phase does not compete for it. Figure 3 shows, for each PPT method at each natural-language token budget, the best PPT token count and weight initialization. At 50M and 100M natural-language tokens both independently tuned PPT pipelines improve over the best no-PPT baseline, whose winning configuration uses spectral-based initialization. This means that at these low budgets the synthetic languages do instill useful inductive biases. This is also seen in Table 1, where k-Dyck PPT at the 50M natural-language token budget improves DCLM CORE accuracy over the no-PPT baseline. However, at 1B all PPT methods fall within the baseline's seed spread of validation BPB. Importantly, under the alternative initializations, the picture never changes: both curricula improve over the baseline at every budget, including 1B. What the additive setting shows is therefore two effects overlaid, a benefit robust to the weight initialization scheme but confined to small natural-language budgets and a benefit at the larger budget that no longer exists when changing the baseline's weight initialization.

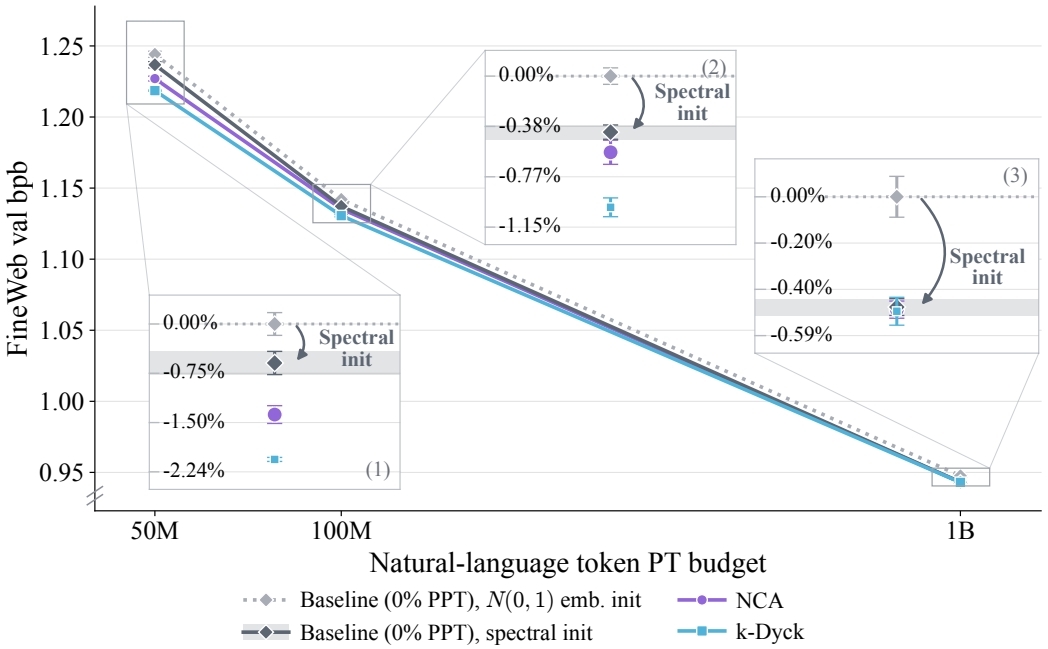

Figure 3: **PPT is useful at 50M–100M natural-language tokens, but its gain at 1B disappears under spectral-based initialization.** Under the additive-compute regime, we find that NCA and k-Dyck, each at its lowest-loss configuration, improve on the optimal no-PPT baseline at 50M and 100M natural-language tokens. Under spectral-based initialization this gain disappears by 1B tokens, but under the other initializations it persists at every budget (Appendix E.1 shows the full breakdown). Because spectral-based initialization yields the lowest loss for every method at this largest budget, we use it in all compute-matched experiments, which run at comparable FLOP budgets.

5 Compute-Matched

Although the previous section shows that PPT can improve performance when compute is added on top of a fixed natural-language budget, this does not establish that synthetic data is the best use of those resources. In practice, the same compute could instead be spent on further natural-language training, including repeated epochs when unique language is scarce. We therefore turn to compute-matched experiments, asking whether PPT can justify diverting part of the training budget away from natural language.

Across the tested regimes, allocating compute to synthetic PPT never beats spending it on natural language. Figure 1 shows results across the three compute-matched configurations. In the data-constrained regime, diverting any fraction of the budget to synthetic PPT increases validation BPB, with the effect leveling off beyond 5%. The compute is better spent on additional epochs over the same natural-language data. In the over-Chinchilla regime, neither NCA nor k-Dyck improves over the baseline on validation loss or DCLM CORE accuracy. The data-unconstrained scaling confirms this pattern holds across budgets: no PPT method significantly outperforms the baseline in terms of natural-language validation BPB, with 1% closer to the baseline than 5% at the two smaller budgets, and the natural-language C4 control matching the baseline. This is broadly reflected in the downstream DCLM CORE and BLiMP benchmarks, with the exceptions shown in Table 1. At the lowest compute budget we study, DeclRef and code show some improvement over the baseline. However, how significant these gains are remains unclear. PPT on C4, a natural-language baseline, scores similarly to these synthetic PPT methods on DCLM CORE. In addition, these are the only experiments in which we observe an aggregate gain; every other method performs at or below the baseline (see Appendix E). When breaking the two benchmarks into their individual tasks, we find that some PPT methods do improve on specific sub-tasks, but often at the cost of others, which is

why most perform on par on aggregate (Appendix E.5, Figure 9). This is especially the case for BLiMP, where some phenomena improve under PPT, in line with the BLiMP gains Hu et al. (2025) report for k-Dyck.

We also observe no consistent ranking among PPT methods across configurations or scales. For example, DeclRef is marginally better than NCA and k-Dyck in the data-constrained regime, but in the data-unconstrained regime the ordering varies with both the PPT percentage and the total compute budget. This is accentuated in the over-Chinchilla regime: with large token budgets and parameter counts, final validation loss between the baseline and PPT methods becomes very similar.

	Setting	PPT method	Baseline (0% PPT)	PPT score
BLiMP (↑ better)	Data-constrained (100M NL, 1×10^{18})	DeclRef 10%	0.721 ± 0.004	0.731 ± 0.005
DCLM CORE (↑ better)	Additive (50M NL)	k-Dyck	0.024 ± 0.004	0.032 ± 0.003
	Data-unconstrained (1×10^{18})	C4 1%	0.088 ± 0.006	0.106 ± 0.004
		Code 1%		0.104 ± 0.004
		DeclRef 1%		0.097 ± 0.002
		DeclRef 5%		0.108 ± 0.011

Table 1: **Downstream benchmark improvement.** Across the models reported in Figures 1 and 3, these are the cases whose aggregate BLiMP or DCLM CORE score sits above their baseline. However, C4, the natural-language control, reaches a DCLM CORE score in the same range as the PPT methods. In this table, NL denotes natural-language tokens.

6 Retention Across the PPT-to-PT Transition

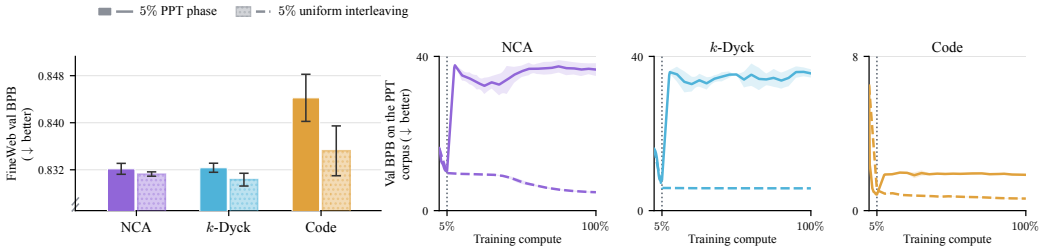

Figure 4: **Uniformly distributing the PPT data across training, rather than front-loading it, prevents forgetting on the PPT distribution.** In the interleaved runs, 5% of every batch consists of NCA, k-Dyck, or code throughout training. In the PPT runs, the same 5% of compute is instead spent up front, at 2.7×10^{19} FLOPs. Left: FineWeb validation BPB at the end of training. Right: validation BPB on the PPT corpus itself across the whole run, one panel per corpus on its own scale, with the gray dotted line marking 5% of total compute. Under the PPT schedule the models lose this ability at the PPT-to-PT boundary; under interleaving they keep it and continue to improve on it. Elsewhere in this work the vocabulary is replaced at the phase boundary for the synthetic PPT distributions, which means the PPT corpus cannot be scored after the PPT phase. Every run here instead shares a single vocabulary, extended to hold the NCA and k-Dyck tokens, so the corpus can be tracked throughout training. These runs are not hyperparameter-tuned (configurations in Appendix D.5, Table 7), so the FineWeb panel is included only to show that the interleaved runs sit in a similar range to their PPT counterparts.

One reason PPT may fail to improve over the baseline is that what it learns does not carry over into PT. The shift from PPT to PT is abrupt: the PPT data differs substantially from natural language, and is seen only at the start of training and never again. Ibrahim et al. (2024) show that even a shift between two natural languages causes forgetting. After pretraining on 300B tokens of English and continuing on German, ability on the original English distribution degrades, and by more than

under a milder shift between two English corpora. The shift from a synthetic distribution to natural language is arguably even larger.

We test this hypothesis by tracking the validation BPB of the NCA, k-Dyck and code PPT distributions across the phase boundary, at 2.7×10^{19} FLOPs, a larger budget than the ones run in previous sections. This gives us a longer horizon over which to observe performance on the PPT distribution. The three right-hand panels in Figure 4 show the result: during the PPT phase all three methods learn their respective distributions, but at the PPT-to-PT boundary the BPB immediately starts increasing, signaling that the models forget their ability on the PPT distribution. The same figure suggests that this stems from front-loading the data in the PPT phase. Holding 5% of every batch as PPT data throughout training instead improves performance on the PPT distribution over the entire run. This is despite both schedules spending the same total compute on the PPT distribution. Forgetting at the boundary is therefore preventable by simply mixing in the synthetic distribution throughout PT. Interleaving improves FineWeb BPB for the PPT methods, though not significantly for all, and as these runs are not hyperparameter-tuned at this scale, how large the gain could be remains open.

7 Limitations

Hyperparameter search at larger scales. We could not run a full hyperparameter search above 9×10^{18} FLOPs, where each trial becomes expensive; the over-Chinchilla runs, for instance, sweep only the peak learning rates and use other hyperparameters from the smaller budgets (Appendix D.6). However, the baseline and PPT methods receive the same reduced sweep, so the comparison between them remains matched.

Effect sizes relative to prior work. The differences we report between PPT and the baseline are small, and in some settings smaller than those previously reported. One reason is that we use NanoChat, which uses a different tokenizer than prior work. The tokenizer alone has been shown to change the effect of PPT (Riazhskykh et al., 2026). However, within our own experiments, the setup is held fixed across the baseline and the PPT methods, so any difference between them is attributable to PPT rather than to these details. Another reason is the metric we report. In order to make our results easily comparable for future work, we evaluate using validation BPB. Prior work often reports results in perplexity, which is the exponential of the loss and hence yields larger relative differences than BPB. Hence, our BPB differences are not directly comparable to the perplexity ones shown in prior work.

8 Conclusion

We set out to test whether PPT improves PT efficiency. Across three synthetic corpora and multiple training regimes, allocating compute to PPT never outperforms spending it on natural language. This holds even under a multi-epoch setting over 100M tokens. However, in the additive setting at the smallest natural-language budgets, a benefit persists even against a well-initialized baseline. Prior studies hold the initialization scheme fixed while varying the curriculum, a natural design choice for isolating the curriculum effect. Our results suggest that future robustness checks should additionally vary this axis for both PPT and no-PPT methods.

Moreover, our findings leave open questions for future work. As natural-language data is increasingly constrained relative to available compute (Kim et al., 2025), synthetic data is a compelling prospect. That said, our results motivate rethinking whether such data should be integrated via an explicit PPT phase. For such a phase to be worthwhile, the forgetting across the phase boundary must first be solved. Alternatively, this forgetting can be avoided by interleaving synthetic data throughout training. We therefore recommend that future work investigate better interleaving schedules. However, whether such schedules make synthetic data a better use of compute than natural language remains an open question.

Reproducibility Statement

Anonymized links to our code are given in Appendix F; they will be de-anonymized after review. The repositories contain the full training factory, the data-preparation and generation scripts for

every corpus, and the evaluation code, together with a manifest holding one record per experiment we report. Identifying details are redacted for review, so some links and paths, such as the dataset locations, will not resolve as written until then.

AI USE STATEMENT

In this work, we used generative AI tools to help write the code in the repositories we release (Appendix F), find related work, and polish the writing of this paper, all under close supervision by us, who reviewed and edited all AI-assisted output. In addition, our hyperparameter optimizer, `LLMTuna`, uses a language model to propose hyperparameter configurations within a purpose-built search loop and not simply a coding agent (Appendix D.1). We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2623–2631, 2019.

Viraat Aryabumi, Yixuan Su, Raymond Ma, Adrien Morisot, Ivan Zhang, Acyr Locatelli, Marzieh Fadaee, Ahmet Üstün, and Sara Hooker. To code or not to code? exploring impact of code in pre-training. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=zSfeN1uAcx.

Xiao Bi, Deli Chen, Guanting Chen, et al. DeepSeek LLM: Scaling open-source language models with longtermism, 2024. URL https://arxiv.org/abs/2401.02954.

Johan Bjorck, Alon Benhaim, Vishrav Chaudhary, Furu Wei, and Xia Song. Scaling optimal LR across token horizons. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=WYL4eFLcxG.

CodeParrot. GitHub Code dataset. https://huggingface.co/datasets/codeparrot/github-code, 2022.

Evelina Fedorenko, Anna Ivanova, and Tamar Regev. The language network as a natural kind within the broader landscape of the human brain. *Nature Reviews Neuroscience*, 25:289–312, 04 2024. doi: 10.1038/s41583-024-00802-4.

Xu Guo, Runyu Peng, Jian Tong, Yunhua Zhou, Haijun Lv, Zhihui Lu, and Qipeng Guo. Synthetic pre-pre-training improves language model robustness to noisy pre-training data, 2026. URL https://arxiv.org/abs/2605.10129.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Michael Y. Hu, Jackson Petty, Chuan Shi, William Merrill, and Tal Linzen. Between circuits and Chomsky: Pre-pretraining on formal languages imparts linguistic biases. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. arXiv:2502.19249.

Hugging Face. transformers: Llama configuration and modeling (v4.40.0). https://github.com/huggingface/transformers/blob/v4.40.0/src/transformers/models/llama/configuration_llama.py, 2024. `initializer_range=0.02`, applied as $\mathcal{N}(0, 0.02)$.

Adam Ibrahim, Benjamin Thérien, Kshitij Gupta, Mats L. Richter, Quentin Anthony, Eugene Belilovsky, Timothée Lesort, and Irina Rish. Simple and scalable strategies to continually pre-train large language models. *Transactions on Machine Learning Research*, 2024.

Liangze Jiang, Zachary Shinnick, Anton van den Hengel, Hemanth Saratchandran, and Damien Teney. Procedural pretraining: Warming up language models with abstract data. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2601.21725.

Zhengyao Jiang and Bingchen Zhao. AutoResearch vs classical hyperparameter tuning. *Weco AI: Blog*, 2026. https://www.weco.ai/blog/autoresearch-vs-classical-hpo.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks. https://kellerjordan.github.io/posts/muon/, 2024.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

Andrej Karpathy. nanochat: The best ChatGPT that $100 can buy, 2025. URL https://github.com/karpathy/nanochat.

Andrej Karpathy. AutoResearch: AI agents running research on single-GPU nanochat training automatically. https://github.com/karpathy/autoresearch, 2026a. URL https://github.com/karpathy/autoresearch.

Andrej Karpathy. nanochat miniseries v1. https://github.com/karpathy/nanochat/discussions/420, 2026b. GitHub discussion, 7 January 2026.

Konwoo Kim, Suhas Kotha, Percy Liang, and Tatsunori Hashimoto. Pre-training under infinite compute. *arXiv preprint arXiv:2509.14786*, 2025.

Dan Lee, Seungwook Han, Akarsh Kumar, and Pulkit Agrawal. Training language models via neural cellular automata. *arXiv preprint arXiv:2603.10055*, 2026.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. DataComp-LM: In search of the next generation of training sets for language models. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, volume 37, 2024.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for LLM training. *arXiv preprint arXiv:2502.16982*, 2025.

Masato Mita, Taiga Someya, Ryo Yoshida, and Yohei Oseki. Language acquisition device in large language models. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19564–19577, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.895. URL https://aclanthology.org/2026.acl-long.895/.

Niklas Muennighoff, Alexander M. Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. Scaling data-constrained language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The FineWeb datasets: Decanting the web for the finest text data at scale. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024. arXiv:2406.17557.

PyTorch. torch.nn.Linear. https://docs.pytorch.org/docs/2.14/generated/torch.nn.Linear.html, 2026a. PyTorch 2.14 documentation.

PyTorch. torch.nn.Embedding. https://docs.pytorch.org/docs/2.14/generated/torch.nn.Embedding.html, 2026b. PyTorch 2.14 documentation.

PyTorch. torch.utils.flop_counter: FLOP counting utilities in PyTorch. https://docs.pytorch.org/docs/2.14/utils.html, 2026c. PyTorch 2.14 documentation.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023. URL https://arxiv.org/abs/1910.10683.

Sofiia Riazhskykh, Nam Luu, and Ondřej Bojar. Instability of LLM pre-pretraining: It doesn't always help. an investigation on multiple languages, 2026. URL https://arxiv.org/abs/2608.08800.

Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. Beyond Chinchilla-optimal: Accounting for inference in language model scaling laws. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024. arXiv:2401.00448.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Akshay Vegesna, Samip Dahal, Chinmay Karkar, Bishwas Mandal, Shmuel Berman, and Zhiwei Xu. NanoGPT slowrun: Language modeling with limited data and infinite compute. https://github.com/qlabs-eng/slowrun, 2026.

Alex Warstadt, Alicia Parrish, Haokun Liu, Anhad Mohananey, Wei Peng, Sheng-Fu Wang, and Samuel R. Bowman. BLiMP: The benchmark of linguistic minimal pairs for English. *Transactions of the Association for Computational Linguistics*, 8:377–392, 2020. doi: 10.1162/tacl_a_00321. URL https://aclanthology.org/2020.tacl-1.25/.

Alex Warstadt, Aaron Mueller, Leshem Choshen, Ethan Wilcox, Chengxu Zhuang, Juan Ciro, Rafael Mosquera, Bhargavi Paranjabe, Adina Williams, Tal Linzen, and Ryan Cotterell. Findings of the BabyLM challenge: Sample-efficient pretraining on developmentally plausible corpora. In *Proceedings of the BabyLM Challenge at the 27th Conference on Computational Natural Language Learning*, pp. 1–34, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.conll-babylm.1. URL https://aclanthology.org/2023.conll-babylm.1/.

Yuhuai Wu, Felix Li, and Percy Liang. Insights into pre-training via simpler synthetic tasks. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=Bb4VGOWELI.

Greg Yang, James B. Simon, and Jeremy Bernstein. A spectral condition for feature learning, 2024b. URL https://arxiv.org/abs/2310.17813.

A DATA

A.1 k-DYCK

We generate k-Shuffle Dyck with the code released by Hu et al. (2025), at their published settings. At each position the generator opens a new bracket with probability 0.5 or closes one of the currently open brackets, chosen uniformly among the open types, forcing a close once the maximum nesting depth is reached and an open at depth zero. We use the variant and setting they report as best, $k = 128$ bracket types, which gives a vocabulary of $2k = 256$, with a maximum nesting depth of 16 and sequences of 2048 tokens.

A.2 NEURAL CELLULAR AUTOMATA

We generate the NCA corpus with the code released by Lee et al. (2026). The automaton is a 12×12 grid of cells, each holding one of ten colors, and the update rule is a randomly initialized convolutional network applied identically to every cell. A rule and an initial grid are sampled for every sequence, and a rule is kept only when the gzip compression ratio of its rollout exceeds 50%, which selects for dynamics that are more complex rather than trivially predictable (low entropy).

Grids are not tokenized cell by cell. Each grid is serialized in row-major order as non-overlapping 2×2 patches, so its 144 cells become 36 tokens, and since a patch holds four cells over ten colors the patch vocabulary is 10^4 token IDs. A start and an end token delimit each grid, so one grid costs 38 tokens, and successive grids are concatenated into a single sequence tracing the automaton's evolution.

The context length therefore fixes how many grids a sequence holds. Lee et al. (2026) train at 1024 tokens, which is 27 grids ($27 \times 38 = 1026$, with the tail truncated), whereas we train at 2048, which is 54 grids ($54 \times 38 = 2052$, again with its tail truncated). Relative to a sequence length of 1024, each of our sequences shows twice as much of the automaton's evolution as a result. This does not apply to the NCA reproduction of Appendix B.

A.3 DECLARATIVE-REFERENCE DATA

Prior work uses procedural tasks such as copying and retrieval as PPT data (Jiang et al., 2026). This "declarative-reference" corpus follows that idea, aiming at a distribution with little semantics that forces context recall, but at a higher entropy than a pure copying task, where everything is determined once the prefix has been seen.

In this synthetic corpus, each training document is a sequence of statements, built from four keywords:

- `declare` declares a new *Name* with its *Value*
- `reference` repeats a *Name* that is already in scope (i.e., that has been declared), followed by its *Value*.
- `open` starts a new *Scope*, and `close` ends it. A Scope is the part of a document between a matching `open` and `close`. Scopes can be nested, like balanced brackets in k-Dyck. Names declared inside a Scope are available until that Scope is closed; Names declared in enclosing Scopes remain available.

Here are the first sixteen tokens of a training document, with token IDs above and a possible decoding below. The choice of letters is arbitrary and only makes the example more readable: lowercase letters stand for *Name* tokens, uppercase for *Value* tokens.

token IDs	2	163	873	1027	977	2	305	383	73	747	730	3	163	873	1027	977
decoded	`declare`	r	Q	S	T	`declare`	n	b	m	U	V	`reference`	r	Q	S	T

Reading it as three statements:

1. `declare r Q S T`. At the start of a document nothing has been declared yet, so a `reference` is impossible and the generator emits a `declare`. It then draws a random

Name, here the single token r, and a random *Value*, here the three tokens $Q\,S\,T$. The length of each is itself drawn at random, from one to three tokens, so statements vary in length.

2. `declare n b m U V`. A *Name* is now in scope, so this time the generator could have emitted a `reference`. The keyword is chosen at random from those available, and here it is a `declare` again, giving the *Name* $n\,b\,m$ and the *Value* $U\,V$.
3. `reference r Q S T`. This time the keyword drawn is a `reference`. A second random choice picks a *Name* from those in scope, either r or $n\,b\,m$, and here picks r. Neither choice can be predicted. Once both are made, though, the rest of the statement is fixed: the only possible continuation is the *Value* $Q\,S\,T$ associated with r, so a model predicts these three tokens correctly only by recalling that pairing from the context.

The *Names* and *Values* occupy disjoint blocks of the vocabulary, IDs $[4, 516)$ and $[516, 1028)$, and the four keywords occupy IDs 0 to 3, so the letter case in the decoding stands in for a split that is real in the data. No `open` or `close` appears in this example window, although both occur elsewhere in the document. When a Scope is closed, Names declared inside it are no longer available for `reference`; Names declared in enclosing Scopes remain available.

The generator has the following parameters:

- How often a *Scope* opens, and how deeply *Scopes* can nest.
- How frequently each *Name* and *Value* token is drawn. The frequencies follow a Zipf distribution drawn newly for every document.
- How a `reference` chooses among the *Names* currently in scope, which favors recently declared ones.
- How the mix of `declare` to `reference` statements shifts from the start of a document to its end.
- The fraction of references that pair the *Name* with a fresh *Value* instead of repeating the original one, so that no position is perfectly deterministic.
- How often the *Name* frequencies are reshuffled part-way through a document: a *Name* that occurs every few statements early on may, after the reshuffle, become as rare as any other, so frequency statistics inferred from the opening tokens go stale and must be re-estimated from later context

We choose these parameters with one goal in mind: that predicting a token should require using the current document, and never anything memorized from the corpus (except for the four keyword tokens). No *Name* is paired with any particular *Value* across documents, and the token frequencies are redrawn for each document, so nothing learned about one document carries to the next. What a `reference` emits is fixed by the `declare` that preceded it in the same document, and which *Names* can be referenced at all depends on the *Scopes* currently open, so both the answer and the space of possible answers are properties of the local context. The parameter values are provided in the generator code released with this paper.

B NCA Reproduction Experiment

We follow one of the configurations described in the paper, a 400M-parameter Llama model trained on $\sim$9B tokens of OpenWebText, for which the original work reports an 8.6% perplexity improvement over the no-PPT baseline. The exact Llama configuration is not specified for the 400M model, so we estimate it and choose the following: 24 layers, a hidden size of 1024 with 16 attention heads, and a multilayer perceptron intermediate size of 4096. The OpenWebText training and validation splits are not specified either, so we choose these ourselves too. Under their initialization we recover the effect, a 5.4% gain, and we attribute the remaining difference from the reported 8.6% to the choices above. Beyond those choices we make no changes to the released codebase, and the only thing we vary is the weight initialization.

We compare three initializations, shown in Figure 5. The first is that of the original work: the embedding table is drawn from $\mathcal{N}(0, 1)$, the transformer body from $\mathcal{N}(0, 0.02)$, and the language-modeling head keeps PyTorch's default uniform initialization (PyTorch, 2026a;b), whose standard

deviation here is ≈ 0.02. The second draws both the token embedding table, and language-modeling head from $\mathcal{N}(0, 0.02)$. The third is the spectral-based initialization from the version of NanoChat we use: linear layers from $\mathcal{N}(0, \sigma)$ with $\sigma = \min(1, \sqrt{d_{\text{out}}/d_{\text{in}}})/\sqrt{d_{\text{in}}}$, the embedding table from $\mathcal{N}(0, 1)$, and the residual output projections and head zeroed. It does not transfer to Llama exactly, as RMSNorm weights have no counterpart and are left at ones.

Every run is initialized under the method being tested, including the PPT phase. NCA therefore trains its PPT phase under that method, and at the transition the new token embedding table and language-modeling head are drawn under the same one, while the body carries over. As shown in Figure 5, the initialization that best suits the baseline is itself architecture-dependent: the spectral-based initialization is the default in the NanoChat version we use, but is not the best-performing of the three here. This provides further motivation to ablate initialization when evaluating PPT.

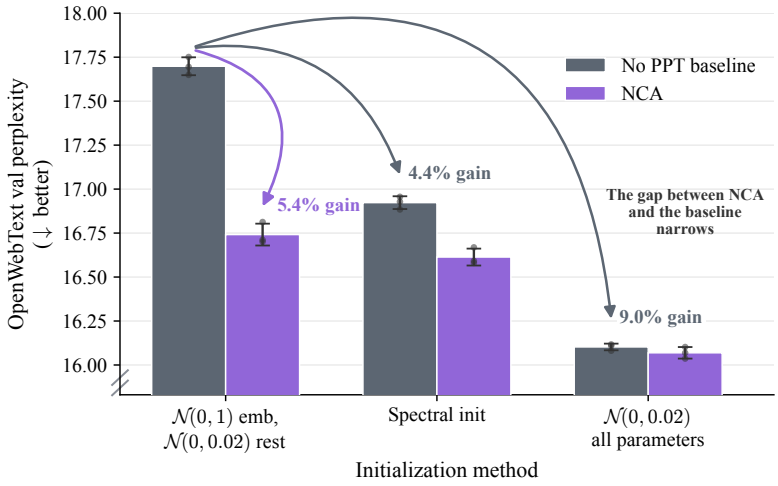

Figure 5: **The NCA positive control under three baseline weight initializations.** A 400M-parameter Llama model trained on 164M NCA tokens, and then ∼9B tokens of OpenWebText, following Lee et al. (2026). Validation perplexity for the no-PPT baseline and NCA under three initializations, namely that of the original work ($\mathcal{N}(0, 1)$ embeddings and $\mathcal{N}(0, 0.02)$ elsewhere), $\mathcal{N}(0, 0.02)$ embeddings and head, and the spectral-based initialization of the NanoChat version we use. Under each alternative initialization the baseline improves and the gap between NCA and the baseline narrows.

C FLOP Measurement

To convert a FLOP budget into a training configuration, we measure the FLOPs of one forward and backward pass (one optimization step) for the model using the FLOP-counting utilities in PyTorch (PyTorch, 2026c). We then divide the total budget by the FLOPs per step to determine the number of optimization steps and tokens consumed by each phase. Working backwards from a FLOP budget to a configuration is the standard framing of compute-optimal PT (Hoffmann et al., 2022), and it is what allows a PPT method and its baseline to be compared on equal terms for the compute-matched experiments.

D Hyperparameters

D.1 How LLMTuna works

`LLMTuna` is a language-model-based hyperparameter optimizer that we release with this work. The approach is inspired by prior work that also used language models for hyperparameter optimization and autoresearch more broadly (Yang et al., 2024a; Jiang & Zhao, 2026; Karpathy, 2026a). `LLMTuna` maintains an ordered history of configurations tried so far and their corresponding scores,

which in our experiments are validation BPB. At each step, a language model uses this history together with any additional context provided by the human engineer, such as training code, notes on hyperparameter behavior, or constraints on the search, to propose the next configuration. This gives the model semantic context from the codebase that can inform which hyperparameters are likely to matter. The search space is specified using three parameter types, `Float`, `Int`, and `Choice`, each with a name, a natural-language description, and optional bounds and initial value.

Following the API design of Optuna (Akiba et al., 2019), the optimization loop consists of two calls. `suggest` provides the accumulated context to the model together with a tool whose schema is generated from the search space, so the proposal is returned as JSON in a fixed shape rather than as prose. `observe` then records the configuration and its resulting score, adding the trial to the history available to subsequent calls to `suggest`.

D.2 Benchmark against Optuna

We compare LLMTuna against a Tree-structured Parzen Estimator (TPE) search using the Optuna library (Akiba et al., 2019) on one of our experiment configurations: NCA at 1% and a total budget of 1×10^{18} FLOPs. Both searches minimize the same objective (validation set BPB) over the same search space and initial hyperparameter configuration.

Optuna is run with the TPE sampler and a budget of 30 trials. The learning rate parameters are sampled in log space for a fair comparison, and the constraint that the warmup and warmdown ratios sum to at most one is enforced by conditional sampling, drawing the warmdown ratio from $[0, 1 - \text{warmup}]$. `LLMTuna` is run under the proposer, context and stopping rule given in Appendices D.3 and F. Results are shown in Figure 6. These results suggest that `LLMTuna` can find good configurations with relatively few trials under the search constraints considered here.

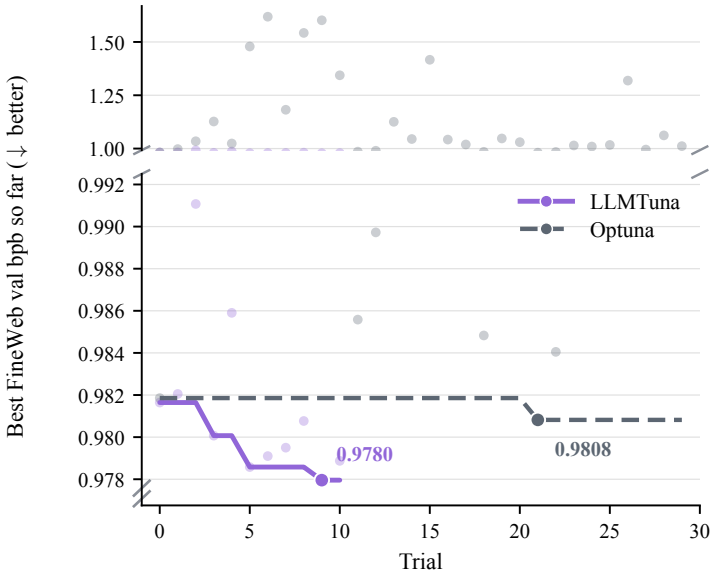

Figure 6: **LLMTuna reaches a lower validation loss in fewer trials than a TPE search.** The continuous lines for each method show the best FineWeb validation BPB so far against trial index, while the faint dots are individual trials. LLMTuna reaches 0.9780 at its tenth trial, while TPE reaches 0.9808 at its twenty-second and does not improve over the remaining eight. The axis is broken above 0.992, where the early exploratory trials of both searches sit.

D.3 Search configuration

In the additive-compute setting, we run a separate `LLMTuna` search for each combination of PT natural-language token count, PPT token count, and the three weight-initialization schemes of Sec-

tion 3. In the data-constrained and data-unconstrained settings, a search is run for each combination of budget C, PPT fraction α and PPT corpus. Table 2 lists what each search tunes.

Setting	Recipe	Tuned hyperparameters
Additive	Unified	Peak matrix learning rate, warmup fraction, warmdown fraction, PT batch size
	Separate	The above, plus PPT peak learning rate, PPT warmup fraction, PPT warmdown fraction, PPT batch size
Compute-matched	Unified	Depth, peak matrix learning rate, warmup fraction, warmdown fraction
	Separate	The above, plus PPT peak learning rate, PPT warmup fraction, PPT warmdown fraction, PPT batch size

Table 2: **Hyperparameters tuned in each setting.** *Separate* is the standard recipe of Section 3: the PPT phase gets its own learning rate and schedule, and the optimizer state is reset at the phase boundary. *Unified* covers the no-PPT baseline and the C4 control, which run both phases under a single schedule with no reset; for C4, restarting the optimizer at the boundary hurt its final loss. In the additive setting the natural-language token count is fixed, so batch size sets the number of optimizer steps; we therefore tune it there, and fix it at 262,144 tokens per step in the compute-matched settings. The PPT-phase optimizer is not searched; in the additive and data-constrained settings we run one search under each of Muon and AdamW and report the lower-BPB result.

Every search is given a budget of at most 30 trials and stops early once five consecutive proposals fail to improve the best validation BPB by more than 10^{-3}. This tolerance is finer than almost every baseline-versus-PPT difference we report, and in practice the stopping criterion is what ends the search.

Proposals come from `claude-sonnet-4-6` with a thinking budget of 300 tokens and a response cap of 4000, called through a tool interface that forces it to return a value for every tuned hyperparameter. The proposer sees (i) a system prompt naming only the hyperparameters that the study tunes, (ii) a description of each, with its bounds or permitted options, (iii) a summary of the training stack, generated once from ten source files and restricted to what bears on the tuned hyperparameters, and (iv) every trial so far, with its full configuration and validation BPB, ranked best first. In the data-unconstrained setting, each FLOP budget runs its own search, warm-started from the previous budget's best configuration and carrying that search's history forward as context, so the proposer can see how an optimum moved as the budget grew.

D.4 Proposer prompt

The system prompt for a study tuning all eight hyperparameters begins:

```
You are a hyperparameter optimizer minimizing val_bpb
(validation bits-per-byte; lower is better).

You see all prior trials ranked by val_bpb.  Balance
exploiting the current best config with informative
exploration; the trial budget is small, so every proposal
should either improve the best or teach you something
specific.

Most of the result at a given scale comes from three knobs:
depth, matrix_lr, and ppt_lr.  Each has its own optimum, and
each optimum shifts with scale.  Getting them near their
minimum for THIS scale matters more than refining the other
knobs, so make sure each has been searched widely enough
that you would notice if it were off.

...
```

The rest of the prompt covers step size and keeping proposals attributable; the full prompts and hyperparameter definitions are in the released code.

D.5 HYPERPARAMETERS FOUND

Tables 3–7 list the configuration of every cell the paper reports, one table per setting. Each table lists only the hyperparameters that differ between the cells of its setting; the rest are shared by every run in the paper and are stated here once. All models use a vocabulary of $65{,}536$ tokens (extended to hold the PPT tokens in the shared-vocabulary retention runs) and train at a sequence length of 2048 with a trapezoidal learning-rate schedule (linear warmdown to zero over the stated fraction of the phase, no warmup unless noted), gradient clipping at 1.0 and no weight decay. During natural-language pretraining Muon updates the matrix parameters at the stated peak rate η_{PT} while the embedding and unembedding tables train under AdamW at 0.3 and 0.004; during a PPT phase all three groups share a single rate η_{PPT} (Appendix D.3). Every value is read from the logged configuration of the three seed replicas, which agree on every field.

Method	d	η_{PT}	PPT optimizer	η_{PPT}	PPT batch
Baseline (0% PPT)	20	3e-2	–	–	–
NCA 1%	20	3e-2	AdamW	3e-5	512k
NCA 5%	20	3e-2	AdamW	3e-5	512k
NCA 10%	22	2e-2	AdamW	3e-6	1M
k-Dyck 1%	20	3e-2	Muon	8e-6	512k
k-Dyck 5%	20	3e-2	Muon	2e-2	512k
k-Dyck 10%	22	3e-2	Muon	1e-3	1M
DeclRef 1%	20	3e-2	AdamW	3e-5	256k
DeclRef 5%	20	3e-2	AdamW	3e-5	512k
DeclRef 10%	21	2e-2	AdamW	1e-6	128k

Table 3: **Data-constrained setting: reported configurations.** 100M unique natural-language tokens at 1×10^{18} FLOPs, the experiments of panel (1) of Figure 1, each at the lower-BPB PPT optimizer. Every PPT run warms down over 0.4 of both its PPT and natural-language phases, and the baseline over 0.2 of training.

	d			η_{PT}			η_{PPT}			PT warmdown			PPT warmdown			PPT batch		
Method	1×10^{18}	3×10^{18}	9×10^{18}	1×10^{18}	3×10^{18}	9×10^{18}	1×10^{18}	3×10^{18}	9×10^{18}	1×10^{18}	3×10^{18}	9×10^{18}	1×10^{18}	3×10^{18}	9×10^{18}	1×10^{18}	3×10^{18}	9×10^{18}
Baseline (0% PPT)	12	16	20	3e-2	2e-2	1.5e-2	–	–	–	0.4	0.4	0.5	–	–	–	–	–	–
C4 1%	11	15	19	3e-2	1.5e-2	1.5e-2	–	–	–	0.4	0.4	0.4	–	–	–	–	–	–
C4 5%	12	15	19	3e-2	1.5e-2	1.5e-2	–	–	–	0.4	0.4	0.4	–	–	–	–	–	–
NCA 1%	11	15	19	2e-2	2e-2	2e-2	6e-3	2e-2	1.2e-2	0.4	0.4	0.4	0.4	0.4	0.4	1M	1M	1M
NCA 5%	12	16	20	3e-2	3e-2	3e-2	2e-3	6e-5	6e-5	0.4	0.4	0.4	0.4	0.4	0.4	256k	256k	256k
k-Dyck 1%	11	16	20	3e-2	3e-2	4e-2	3e-3	1e-2	1e-2	0.4	0.4	0.4	0.4	0.4	0.4	1M	1M	1M
k-Dyck 5%	12	16	18	3e-2	3e-2	3e-2	3e-5	3e-5	3e-5	0.4	0.4	0.4	0.4	0.4	0.4	512k	512k	512k
DeclRef 1%	12	16	18	3e-2	1.5e-2	1.5e-2	1.2e-2	1.2e-2	1.2e-2	0.4	0.4	0.4	0.4	0.4	0.4	256k	256k	256k
DeclRef 5%	12	16	20	2e-2	1.5e-2	1.5e-2	2e-3	1e-3	1e-3	0.4	0.4	0.4	0.4	0.4	0.4	256k	256k	256k
Code 1% (unified optimizer)	11	15	18	3e-2	3e-2	1.5e-2	–	–	–	0.4	0.5	0.5	–	–	–	–	–	–
Code 5% (unified optimizer)	12	16	20	1.5e-2	1.5e-2	1.5e-2	–	–	–	0.4	0.5	0.5	–	–	–	–	–	–
Code 1% (separate PPT optimizer)	11	15	18	3e-2	3e-2	2e-2	1e-4	3e-4	3e-4	0.4	0.4	0.4	0.4	0.7	0.7	1M	1M	1M
Code 5% (separate PPT optimizer)	12	14	18	3e-2	3e-2	2e-2	1e-5	1e-5	1e-5	0.4	0.4	0.4	0.4	0.8	0.8	256k	256k	512k

Table 4: **Data-unconstrained setting: reported configurations.** The methods of Table 10 at each budget. Two schedule details have no column: k-Dyck 1% warms its PPT phase up over the first 0.1 of the phase at every budget, and the unified code 5% method warms up over 0.02 of training at 3×10^{18} and 9×10^{18}; every other experiment has no warmup.

D.6 OVER-CHINCHILLA CONFIGURATION

The over-Chinchilla configuration's 50B tokens are $\sim12\times$ the compute-optimal token count for its 881M-parameter model, based on the ≈4.6 tokens per parameter of the baseline's searched optima at 1×10^{18}–9×10^{18} FLOPs. This is well below the 20 token to parameter ratio reported by Hoffmann et al. (2022), as NanoChat's own fits also find (Karpathy, 2026b). This is thanks to differences in data, model architecture, and optimizer. Nonetheless, even by the Chinchilla ratio, the run's ≈57 tokens per parameter is greater than the chinchilla optimal ratio. We run the PPT methods at 1% of total compute at this scale, which amounts to 0.5B tokens on PPT and 49.5B on FineWeb.

No `LLMTuna` search is run at this budget. Training uses no warmup and a linear warmdown over the second half of training, and for the PPT runs the warmdown fraction is set so that decay begins at the same 25B natural-language token mark as in the baseline. We sweep only the peak learning rates, since the optimum moves with the training horizon and this horizon is an order of magnitude longer

NL tokens	Method	Init	PPT tokens	PT batch	PPT optimizer	η_{PPT}	PPT warmdown	PPT batch
50M	Baseline	$\mathcal{N}(0,1)$ emb.	–	64k	–	–	–	–
50M	Baseline	spectral	–	64k	–	–	–	–
50M	NCA	$\mathcal{N}(0,0.02)$	500M	64k	Muon	4e-3	1	256k
50M	k-Dyck	spectral	20M	64k	AdamW	1.2e-4	1	64k
100M	Baseline	$\mathcal{N}(0,1)$ emb.	–	64k	–	–	–	–
100M	Baseline	spectral	–	64k	–	–	–	–
100M	NCA	$\mathcal{N}(0,0.02)$	500M	64k	Muon	6e-3	1	512k
100M	k-Dyck	spectral	1B	64k	AdamW	5e-5	1	512k
1B	Baseline	$\mathcal{N}(0,1)$ emb.	–	256k	–	–	–	–
1B	Baseline	spectral	–	256k	–	–	–	–
1B	NCA	spectral	100M	256k	Muon	6e-6	1	256k
1B	k-Dyck	spectral	5M	256k	Muon	6e-6	0.4	256k

Table 5: **Additive setting: reported configurations.** The twelve points of Figure 3: per natural-language (NL) budget, the two baselines the figure draws and each curriculum at its lowest-loss initialization, PPT token count and PPT optimizer. Every experiment settled on $\eta_{\mathrm{PT}} = 0.02$ with a warmdown of 0.4, and the PPT phase runs with no warmup.

Method	η_{PT}	PT warmdown	η_{PPT}	PPT warmup	PPT warmdown	PPT batch
Baseline (0% PPT)	6e-3	0.5	–	–	–	–
NCA 1%	7.5e-3	0.495	1.3e-3	0	0.6	1M
k-Dyck 1%	6e-3	0.495	1e-2	0.1	0.4	1M

Table 6: **Over-Chinchilla setting: configurations.** Depth 24 (881M parameters) on 50B tokens, the methods of panel (2) of Figure 1. No search is run at this budget; see Appendix D.6.

than any we searched. Every configuration is first run at 0.006, extrapolated from the baseline's searched optima, and then at three further rates, 0.00748, 0.00563 and 0.00368, obtained by rescaling the peak rate with the training horizon T as T^p for $p \in \{-0.2, -0.32, -0.5\}$, a range informed by the power law between optimal learning rate and token horizon reported by Bjorck et al. (2025), with each PPT run's PPT-phase rate rescaled by the same rule in its own horizon. Each rate is run at a single seed, and the rate minimizing FineWeb validation BPB is replicated across the remaining seeds: 0.006 for the baseline and k-Dyck, 0.00748 for NCA.

E Full results

This appendix collects the complete results for each setting in turn. For our reported BLiMP results, we score it following the released evaluation code of Hu et al. (2025). Each of the 67,000 minimal pairs is scored by total sentence log-probability, and a pair is counted correct only if the grammatical sentence scores strictly higher than the ungrammatical one, so ties count as errors. No beginning-of-sequence token is prepended, which means the first token of each sentence is unscored. DCLM CORE is evaluated with the NanoChat evaluation code (Karpathy, 2025).

Run	d	η_{PT}	PT warmdown	η_{PPT}	PPT warmdown	PPT batch
NCA 5%, interleaved	24	1.5e-2	0.5	–	–	–
k-Dyck 5%, interleaved	24	1.5e-2	0.5	–	–	–
Code 5%, interleaved	24	1.5e-2	0.5	–	–	–
NCA 5%, PPT phase	24	3e-2	0.4	6e-5	0.4	256k
k-Dyck 5%, PPT phase	21	3e-2	0.4	3e-5	0.4	512k
Code 5%, PPT phase	24	1.5e-2	0.5	–	–	–

Table 7: **Retention experiment: configurations.** The runs of Figure 4 at 2.7×10^{19} FLOPs. The PPT runs use each method's 5% configuration, extrapolated from the smaller budgets, with a shared vocabulary. The interleaved runs train under one schedule, so they have no PPT-phase entries, and all three share the code 5% configuration so that only the interleaved corpus differs.

E.1 ADDITIVE SETTING

Figure 7 shows the full results behind Figure 3, and Tables 8 and 9 report BLiMP and DCLM CORE accuracy for the same runs, with each curriculum at its lowest-loss PPT token count.

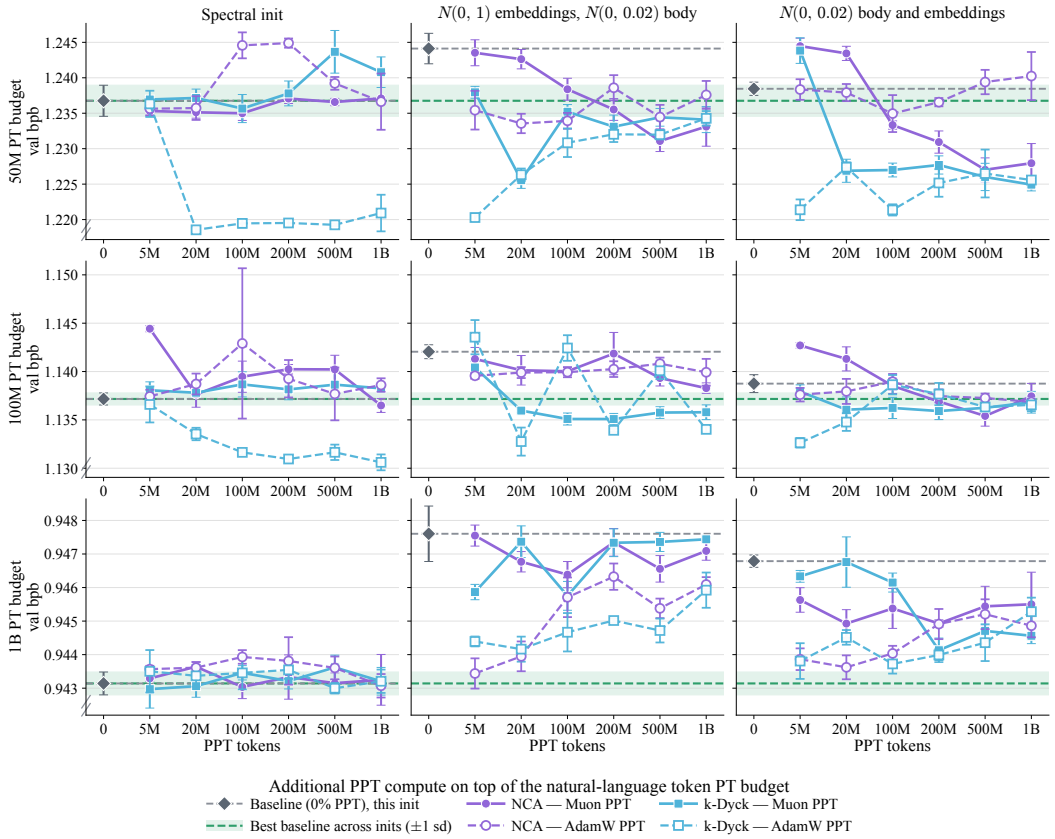

Figure 7: **Full synthetic PPT token sweep behind Figure 3.** FineWeb validation bits-per-byte against the amount of synthetic PPT added on top of a fixed natural-language token budget. Rows are that budget, columns are the initialization scheme: two as used in the prior literature that introduced these curricula, and the spectral-based initialization. The model size here is 336M parameters. Within each panel the leftmost point is that configuration's own baseline with no PPT, marked by the gray dashed line. The green dashed line and band mark the lowest-loss no-PPT baseline across the three initialization schemes, providing a reference for comparisons across initializations. Solid lines with filled markers are Muon PPT and dashed lines with open markers are AdamW PPT. AdamW performs better than Muon for the PPT methods at the lower natural-language budgets; at 1B tokens, however, the two are indistinguishable under spectral-based initialization.

Initialisation	Method	50M	100M	1B
Spectral init	Baseline	0.691 ± 0.005	0.719 ± 0.001	0.774 ± 0.005
	NCA	0.691 ± 0.002	0.719 ± 0.008	0.769 ± 0.007
	k-Dyck	0.688 ± 0.006	0.723 ± 0.007	0.778 ± 0.007
$N(0, 1)$ embeddings	Baseline	0.687 ± 0.003	0.717 ± 0.004	0.779 ± 0.003
	NCA	0.683 ± 0.003	0.717 ± 0.007	0.776 ± 0.002
	k-Dyck	0.695 ± 0.007	0.718 ± 0.011	0.775 ± 0.001
$N(0, 0.02)$ all	Baseline	0.695 ± 0.005	0.720 ± 0.006	0.779 ± 0.005
	NCA	0.691 ± 0.008	0.724 ± 0.006	0.774 ± 0.004
	k-Dyck	0.690 ± 0.005	0.722 ± 0.011	0.783 ± 0.005

Table 8: **BLiMP in the additive setting.** BLiMP accuracy for the additive-compute experiments of Figure 3: for each natural-language budget and initialization, the baseline and each curriculum at its best (lowest-loss) PPT token count (Table 5 lists the counts). Higher is better, and bold marks values whose one-standard-deviation interval lies entirely above that of the baseline with the same initialization.

Initialisation	Method	50M	100M	1B
Spectral init	Baseline	0.024 ± 0.004	0.042 ± 0.011	0.110 ± 0.009
	NCA	$\mathbf{0.032 \pm 0.001}$	0.046 ± 0.006	0.120 ± 0.002
	k-Dyck	$\mathbf{0.032 \pm 0.003}$	0.046 ± 0.012	0.116 ± 0.003
$N(0, 1)$ embeddings	Baseline	0.030 ± 0.002	0.040 ± 0.007	0.107 ± 0.010
	NCA	0.035 ± 0.010	0.045 ± 0.009	0.117 ± 0.014
	k-Dyck	0.030 ± 0.005	0.044 ± 0.006	0.115 ± 0.015
$N(0, 0.02)$ all	Baseline	0.031 ± 0.007	0.049 ± 0.008	0.121 ± 0.003
	NCA	0.027 ± 0.006	0.047 ± 0.012	0.116 ± 0.006
	k-Dyck	0.037 ± 0.012	0.044 ± 0.004	0.117 ± 0.012

Table 9: **DCLM CORE in the additive setting.** Centered CORE accuracy for the same experiments as Table 8: for each natural-language budget and initialization, the baseline and each curriculum at its best (lowest-loss) PPT token count (Table 5 lists the counts). Higher is better, and bold marks values whose one-standard-deviation interval lies entirely above that of the baseline with the same initialization.

Method	1×10^{18}	3×10^{18}	9×10^{18}
Baseline (0% PPT)	$\mathbf{0.9794}_{\pm 0.0003}$	$\mathbf{0.9234}_{\pm 0.0002}$	$\mathbf{0.8733}_{\pm 0.0002}$
C4 1%	$\mathbf{0.9799}_{\pm 0.0006}$	$\mathbf{0.9232}_{\pm 0.0005}$	$\mathbf{0.8729}_{\pm 0.0004}$
C4 5%	$0.9804_{\pm 0.0004}$	$\mathbf{0.9238}_{\pm 0.0003}$	$\mathbf{0.8735}_{\pm 0.0005}$
NCA 1%	$0.9803_{\pm 0.0005}$	$\mathbf{0.9239}_{\pm 0.0001}$	$\mathbf{0.8751}_{\pm 0.0024}$
NCA 5%	$0.9830_{\pm 0.0002}$	$0.9272_{\pm 0.0002}$	$0.8765_{\pm 0.0002}$
k-Dyck 1%	$0.9802_{\pm 0.0002}$	$0.9248_{\pm 0.0003}$	$0.8765_{\pm 0.0021}$
k-Dyck 5%	$0.9828_{\pm 0.0007}$	$0.9270_{\pm 0.0001}$	$0.8763_{\pm 0.0003}$
DeclRef 1%	$0.9804_{\pm 0.0004}$	$0.9245_{\pm 0.0005}$	$0.8772_{\pm 0.0004}$
DeclRef 5%	$0.9827_{\pm 0.0005}$	$0.9265_{\pm 0.0001}$	$0.8758_{\pm 0.0003}$
Code 1% (unified optimizer)	$0.9831_{\pm 0.0003}$	$0.9266_{\pm 0.0002}$	$0.8790_{\pm 0.0056}$
Code 5% (unified optimizer)	$0.9869_{\pm 0.0003}$	$0.9305_{\pm 0.0002}$	$0.8817_{\pm 0.0016}$
Code 1% (separate PPT optimizer)	$\mathbf{0.9800}_{\pm 0.0004}$	$0.9242_{\pm 0.0002}$	$0.8740_{\pm 0.0001}$
Code 5% (separate PPT optimizer)	$0.9822_{\pm 0.0003}$	$0.9270_{\pm 0.0004}$	$0.8753_{\pm 0.0005}$

Table 10: **Allocating a larger share of a fixed compute budget to synthetic PPT generally increases loss.** FineWeb bits-per-byte. Hyperparameters were searched independently for each method at every budget. Code appears under both the separate and the unified optimizer recipe. Bold marks values whose one-standard-deviation interval overlaps that of the column's lowest.

E.2 Data-unconstrained setting

Validation bits-per-byte on all three held-out splits, followed by the two benchmarks, across the data-unconstrained sweep.

Method	1×10^{18}	3×10^{18}	9×10^{18}
Baseline (0% PPT)	$\mathbf{1.1461}_{\pm 0.0002}$	$1.0900_{\pm 0.0002}$	$1.0409_{\pm 0.0004}$
C4 1%	$\mathbf{1.1465}_{\pm 0.0007}$	$1.0898_{\pm 0.0007}$	$1.0399_{\pm 0.0005}$
C4 5%	$\mathbf{1.1456}_{\pm 0.0004}$	$\mathbf{1.0885}_{\pm 0.0004}$	$\mathbf{1.0382}_{\pm 0.0008}$
NCA 1%	$1.1471_{\pm 0.0006}$	$1.0905_{\pm 0.0003}$	$1.0424_{\pm 0.0027}$
NCA 5%	$1.1497_{\pm 0.0003}$	$1.0937_{\pm 0.0003}$	$1.0433_{\pm 0.0002}$
k-Dyck 1%	$1.1470_{\pm 0.0002}$	$1.0914_{\pm 0.0003}$	$1.0436_{\pm 0.0021}$
k-Dyck 5%	$1.1493_{\pm 0.0006}$	$1.0935_{\pm 0.0001}$	$1.0443_{\pm 0.0005}$
DeclRef 1%	$1.1472_{\pm 0.0006}$	$1.0912_{\pm 0.0006}$	$1.0446_{\pm 0.0005}$
DeclRef 5%	$1.1493_{\pm 0.0007}$	$1.0931_{\pm 0.0001}$	$1.0428_{\pm 0.0005}$
Code 1% (unified optimizer)	$1.1500_{\pm 0.0005}$	$1.0935_{\pm 0.0003}$	$1.0468_{\pm 0.0054}$
Code 5% (unified optimizer)	$1.1537_{\pm 0.0003}$	$1.0973_{\pm 0.0002}$	$1.0487_{\pm 0.0017}$
Code 1% (separate PPT optimizer)	$1.1468_{\pm 0.0004}$	$1.0907_{\pm 0.0002}$	$1.0420_{\pm 0.0005}$
Code 5% (separate PPT optimizer)	$1.1488_{\pm 0.0004}$	$1.0938_{\pm 0.0002}$	$1.0432_{\pm 0.0006}$

Table 11: **The data-unconstrained comparison on the held-out C4 split.** C4 bits-per-byte for the methods of Table 10, same checkpoints, same protocol. Bold marks values whose one-standard-deviation interval overlaps that of the column's lowest.

E.3 Data-constrained setting

Panel (1) of Figure 1 reports, for each curriculum and PPT percentage, the best configuration found by the search. Because we run one search under each PPT-phase optimizer and report the lower-BPB result, that panel selects over the optimizer as well as over depth and learning rate. Figure 8 separates the two optimizers, and the benchmark scores for the same experiments follow.

Method	1×10^{18}	3×10^{18}	9×10^{18}
Baseline (0% PPT)	$\mathbf{1.9435}_{\pm0.0260}$	$\mathbf{1.7078}_{\pm0.0785}$	$\mathbf{1.6702}_{\pm0.0347}$
C4 1%	$\mathbf{1.9738}_{\pm0.0498}$	$\mathbf{1.7890}_{\pm0.0528}$	$\mathbf{1.6590}_{\pm0.0286}$
C4 5%	$\mathbf{1.9012}_{\pm0.0132}$	$\mathbf{1.7914}_{\pm0.0732}$	$\mathbf{1.6832}_{\pm0.0217}$
NCA 1%	$\mathbf{1.9356}_{\pm0.0883}$	$\mathbf{1.7793}_{\pm0.0469}$	$\mathbf{1.7353}_{\pm0.0295}$
NCA 5%	$\mathbf{1.9640}_{\pm0.0603}$	$\mathbf{1.8043}_{\pm0.0862}$	$\mathbf{1.7272}_{\pm0.0227}$
k-Dyck 1%	$\mathbf{1.9867}_{\pm0.0929}$	$\mathbf{1.7675}_{\pm0.0580}$	$\mathbf{1.8248}_{\pm0.1379}$
k-Dyck 5%	$1.9536_{\pm0.0281}$	$\mathbf{1.8378}_{\pm0.1098}$	$\mathbf{1.8205}_{\pm0.1192}$
DeclRef 1%	$1.9933_{\pm0.0670}$	$\mathbf{1.7080}_{\pm0.1051}$	$\mathbf{1.7438}_{\pm0.0528}$
DeclRef 5%	$\mathbf{1.9250}_{\pm0.0147}$	$\mathbf{1.7385}_{\pm0.0628}$	$\mathbf{1.7206}_{\pm0.0545}$
Code 1% (unified optimizer)	$\mathbf{1.9601}_{\pm0.0834}$	$\mathbf{1.8063}_{\pm0.1156}$	$\mathbf{1.7206}_{\pm0.0346}$
Code 5% (unified optimizer)	$\mathbf{1.8892}_{\pm0.0361}$	$\mathbf{1.7491}_{\pm0.0304}$	$\mathbf{1.6568}_{\pm0.0585}$
Code 1% (separate PPT optimizer)	$\mathbf{1.9697}_{\pm0.0891}$	$\mathbf{1.7372}_{\pm0.0510}$	$\mathbf{1.6882}_{\pm0.0374}$
Code 5% (separate PPT optimizer)	$\mathbf{1.9426}_{\pm0.0390}$	$1.7975_{\pm0.0099}$	$\mathbf{1.6924}_{\pm0.0661}$

Table 12: **The data-unconstrained comparison on the held-out GitHub-code split.** GitHub-code bits-per-byte for the methods of Table 10, same checkpoints, same protocol. Bold marks values whose one-standard-deviation interval overlaps that of the column's lowest.

Method	1×10^{18}	3×10^{18}	9×10^{18}
Baseline (0% PPT)	0.761 ± 0.004	0.780 ± 0.003	0.794 ± 0.004
C4 1%	0.756 ± 0.007	0.779 ± 0.004	0.791 ± 0.009
C4 5%	0.757 ± 0.009	0.780 ± 0.005	0.794 ± 0.008
NCA 1%	0.753 ± 0.006	0.781 ± 0.007	0.792 ± 0.002
NCA 5%	0.759 ± 0.006	0.773 ± 0.010	0.796 ± 0.002
k-Dyck 1%	0.755 ± 0.007	0.782 ± 0.006	0.798 ± 0.002
k-Dyck 5%	0.762 ± 0.006	0.783 ± 0.001	0.797 ± 0.002
DeclRef 1%	0.758 ± 0.013	0.779 ± 0.010	0.791 ± 0.009
DeclRef 5%	0.754 ± 0.002	0.778 ± 0.003	0.797 ± 0.008
Code 1% (unified optimizer)	0.753 ± 0.004	0.782 ± 0.002	0.789 ± 0.007
Code 5% (unified optimizer)	0.762 ± 0.009	0.784 ± 0.004	0.793 ± 0.003
Code 1% (separate PPT optimizer)	0.758 ± 0.007	0.783 ± 0.007	0.796 ± 0.006
Code 5% (separate PPT optimizer)	0.755 ± 0.009	0.780 ± 0.005	0.789 ± 0.004

Table 13: **BLiMP across the data-unconstrained sweep.** BLiMP accuracy for the data-unconstrained sweep; higher is better, and bold marks values whose one-standard-deviation interval lies entirely above the baseline's at that budget.

Method	1×10^{18}	3×10^{18}	9×10^{18}
Baseline (0% PPT)	0.088 ± 0.006	0.121 ± 0.015	0.165 ± 0.008
C4 1%	$\mathbf{0.106 \pm 0.004}$	0.131 ± 0.004	0.156 ± 0.004
C4 5%	0.101 ± 0.007	0.137 ± 0.005	0.154 ± 0.007
NCA 1%	0.092 ± 0.005	0.127 ± 0.011	0.162 ± 0.007
NCA 5%	0.082 ± 0.003	0.133 ± 0.004	0.155 ± 0.012
k-Dyck 1%	0.096 ± 0.012	0.135 ± 0.010	0.153 ± 0.006
k-Dyck 5%	0.098 ± 0.004	0.135 ± 0.007	0.153 ± 0.007
DeclRef 1%	$\mathbf{0.097 \pm 0.002}$	0.130 ± 0.014	0.152 ± 0.005
DeclRef 5%	$\mathbf{0.108 \pm 0.011}$	0.125 ± 0.008	0.159 ± 0.005
Code 1% (unified optimizer)	0.098 ± 0.005	0.135 ± 0.002	0.159 ± 0.005
Code 5% (unified optimizer)	$\mathbf{0.103 \pm 0.005}$	0.130 ± 0.008	0.158 ± 0.007
Code 1% (separate PPT optimizer)	$\mathbf{0.104 \pm 0.004}$	0.132 ± 0.010	0.159 ± 0.006
Code 5% (separate PPT optimizer)	0.096 ± 0.008	0.127 ± 0.012	0.156 ± 0.002

Table 14: **DCLM CORE across the data-unconstrained sweep.** Centered CORE accuracy; higher is better, and bold marks values whose one-standard-deviation interval lies entirely above the baseline's.

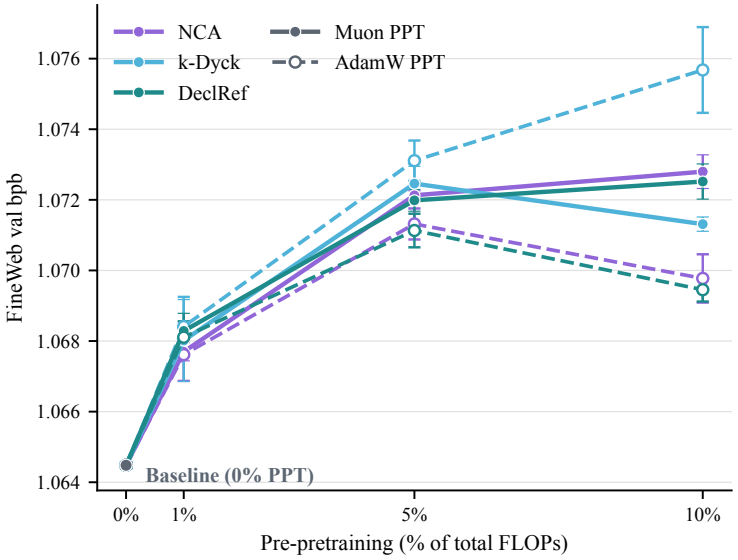

Figure 8: **Data-constrained results separated by PPT-phase optimizer.** The experiments behind panel (1) of Figure 1, with Muon PPT drawn as solid lines with filled markers and AdamW PPT as dashed lines with open markers. Each experiment was searched independently under the same search space and stopping rule, so the two differ in the optimizer used during PPT and in the configuration each search found; every point shows that experiment's tuned configuration, on the same evaluation window as the baseline. Neither optimizer is always better than the other: AdamW is lower for NCA and DeclRef at every PPT percentage, while k-Dyck is lower under Muon. No experiment of either optimizer reaches the no-PPT baseline.

Method	BLiMP	DCLM CORE
Baseline (0% PPT)	0.721 ± 0.004	0.068 ± 0.012
NCA 1%	0.718 ± 0.003	0.077 ± 0.007
NCA 5%	0.722 ± 0.001	0.074 ± 0.006
NCA 10%	0.730 ± 0.006	0.069 ± 0.002
k-Dyck 1%	0.714 ± 0.007	0.068 ± 0.009
k-Dyck 5%	0.724 ± 0.001	0.059 ± 0.010
k-Dyck 10%	0.728 ± 0.006	0.060 ± 0.008
DeclRef 1%	0.727 ± 0.003	0.068 ± 0.007
DeclRef 5%	0.720 ± 0.007	0.070 ± 0.009
DeclRef 10%	$\mathbf{0.731 \pm 0.005}$	0.070 ± 0.004

Table 15: **BLiMP and DCLM CORE in the data-constrained regime.** 100M unique natural-language tokens at a fixed 1×10^{18}-FLOP budget, the experiments of panel (1) of Figure 1, each at the lower-BPB PPT optimizer. Higher is better, and bold marks values whose one-standard-deviation interval lies entirely above the baseline's.

E.4 Over-Chinchilla setting

Benchmark scores for the 881M-parameter runs on 50B tokens.

Method	BLiMP	DCLM CORE
Baseline (0% PPT)	0.809 ± 0.006	0.263 ± 0.008
NCA 1%	0.812 ± 0.005	0.264 ± 0.009
k-Dyck 1%	0.809 ± 0.002	0.262 ± 0.009

Table 16: **Over-Chinchilla benchmark summary.** BLiMP and DCLM CORE for the over-Chinchilla regime: 881M parameters on 50B tokens, NCA and k-Dyck at 1%. Higher is better, and bold marks values whose one-standard-deviation interval lies entirely above the baseline's. No method separates from the baseline on either benchmark.

E.5 Downstream benchmarks

The DCLM CORE suite from Li et al. (2024) aggregates 22 tasks: AGIEval LSAT-AR, ARC-Easy, ARC-Challenge, six BIG-bench tasks (QA Wikidata, Dyck languages, Operators, Repeat Copy Logic, CS Algorithms, and Language Identification), BoolQ, CommonsenseQA, COPA, CoQA, HellaSwag (zero-shot and 10-shot), Jeopardy, LAMBADA, OpenBookQA, PIQA, SQuAD, the Winograd Schema Challenge, and Winogrande. Please refer to the original paper for more details on each benchmark.

F Code release

The training and evaluation code is available at `rethinking-pre-pretraining`, and `LLMTuna`, the hyperparameter optimizer, at `llmtuna`.

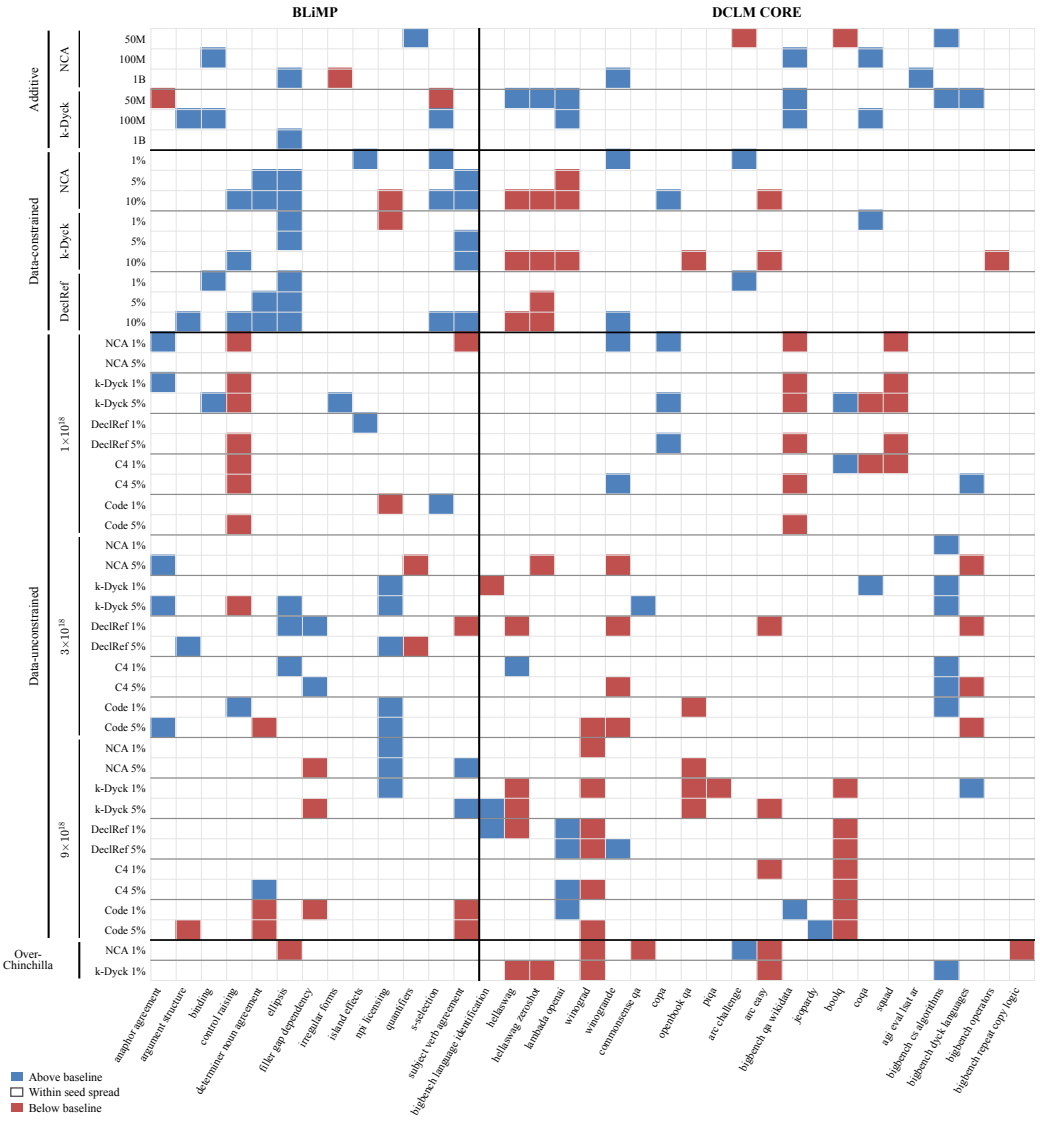

Figure 9: **Breakdown of downstream performance by benchmark sub-tasks.** Each row is a PPT model from Figures 1 and 3, and each column is one of the 13 BLiMP or 22 DCLM CORE tasks. Blue cells are significantly above the baseline, red cells significantly below, and white cells on par. Most cells are white, so on most sub-scores the PPT models perform on par with their baseline. The colored cells show that, even so, the PPT method changes how the model performs on some sub-tasks of these benchmarks.