MATRYOSHKA LANGUAGE MODEL SUITES

Anonymous authors
Paper under double-blind review

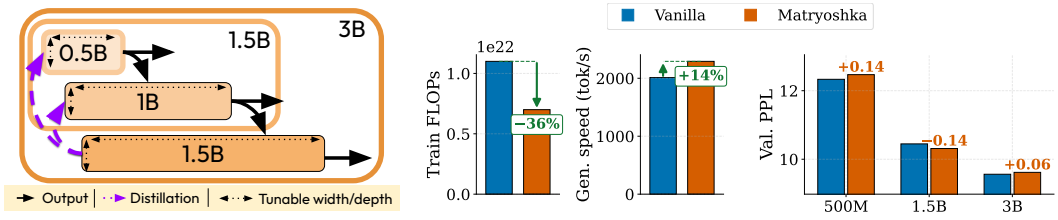
ABSTRACT

Training a language model suite classically requires training each model separately and serving them independently. We improve both training and inference efficiency by stacking sub-models of increasing size into a single nested architecture trained end-to-end. This *Matryoshka* training framework reduces the total parameter count of the suite, enables low-cost distillation from the largest to all smaller sub-models at every training step, and is well-suited for speculative decoding as the draft model is contained within the verifier. We validate our approach by training a Matryoshka suite comprising 500M, 1.5B, and 3B sub-models. Our suite is on par with independently trained baselines on benchmark performance and validation and out-of-distribution perplexities, while using 36% less training compute and improving the throughput of speculative decoding by 14–26%. We also ablate key architectural choices, offering guidance for building strong Matryoshka LM suites.

1 INTRODUCTION

The way training and inference costs of large language models (LLMs) scale steeply with model size creates a demand for models across a broad spectrum of sizes, from 100M (Allal et al., 2025) to trillion (DeepSeek-AI, 2026) parameters, for diverse deployment scenarios. Delivering models at different scales allows for different load/performance trade-offs, with smaller models being more memory- and compute-efficient than their larger counterparts. As a result, models are usually released in *suites*: groups of models trained separately on similar data, with smaller models often distilled from larger ones in the suite. The question of efficiency in training and serving model suites *jointly* is rarely studied. We propose a training framework that is particularly well-suited to the model-suite paradigm and offers several concrete benefits over the standard approach.

Instead of training separate models, our approach progressively stacks the outputs of smaller sub-models as inputs to larger ones. For instance, rather than training a $\{1\text{B}, 8\text{B}, 30\text{B}, 70\text{B}\}$ suite as four independent runs, our method trains a single architecture combining a 1B sub-model with additional 7B, 22B, and 40B parameter blocks, resulting in 70B trained parameters instead of 109B. Each sub-model can be detached as a standalone language model, with its own depth and KV cache footprint, and served independently just like a regular checkpoint. This yields a better size–performance frontier



(a) Matryoshka suite

(b) Main results for the 3B suite

Figure 1: **(a)** Matryoshka model suite: sub-models are nested into a single architecture trained end-to-end, instead of training M models independently. Each sub-model has its own width and depth, fixed at training time; smaller sub-models are distilled from the largest one as a by-product of every forward pass. **(b)** Main results for the 3B suite: training compute drops by 36%, speculative-decoding throughput on a 500M-draft / 3B-verifier pair (nucleus, T=0.4) rises by 14%, and per-size validation PPL stays within $\pm 1.4\%$ of the Vanilla baseline.

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.

than previous work such as MatFormer (Section 3.2.4), which only nests FFN widths so that all sub-models keep the KV cache footprint of the largest one. This setup also makes distillation from the largest model to all smaller ones a natural by-product, as logits from all sub-models are available during training. As a result, training the Matryoshka suite costs on the same order as training only the largest model alone, and delivers all M sub-models in a single run, with cheap knowledge distillation.

At inference time, the nested structure further benefits speculative decoding (Leviathan et al., 2023): the draft model is nested in the verifier model, and both already share KV cache, eliminating the usual memory overhead of maintaining a separate draft model. Cross-model prediction agreement is also higher than in independently trained suites, directly increasing token acceptance rates.

We verify our approach by training a 3B-parameter Matryoshka suite with 500M, 1.5B, and 3B sub-models on 35B tokens from the FineWeb-Edu dataset. Our suite matches independently trained baselines on validation perplexity, out-of-distribution perplexity, and benchmark performance while using 36% less total training compute. Moreover, the shared-weight structure and low-cost distillation substantially improve cross-model agreement and make larger draft models practical for speculative decoding. We also conduct an extensive analysis of architectural design, and show the impact of width and depth configurations in Matryoshka suites, and that both the distillation loss and our model junction mechanism are crucial for the performance of smaller sub-models. We will release trained checkpoints and a HuggingFace's `transformers`-compatible implementation upon publication.

Overall, our contributions can be summarized as:

- We introduce *Matryoshka Language Model Suites*, a pretraining framework that nests sub-models of increasing width and depth into a single jointly trained architecture (Section 2);
- We derive a low-cost distillation objective that leverages the availability of all sub-model logits at every forward pass;
- We train a 3B-parameter Matryoshka suite and show it reaches near-parity (within 0.5 average points at every size) with independently trained baselines on standard benchmarks while using 36% less training compute (Section 3);
- We demonstrate that the shared-weight structure and low-cost distillation substantially improve cross-model agreement and make larger drafts practical for speculative decoding, where speculative decoding with a Vanilla pair barely improves over autoregressive inference;
- We provide extensive ablations at smaller scale (50M / 100M / 200M suite) that provide insights for architectural hyperparameters and training objectives.

2 METHOD

We nest smaller models into larger ones, and allow different hidden sizes across models by implementing a simple junction mechanism. We then compute a cross-entropy loss and a largest-model distillation loss for each exit point, and combine them into a single objective.

Modeling A Transformer (Vaswani et al., 2017) language model maps an input sequence $x \in [1, V]^L$ to output representations $o_\theta(x) \in \mathbb{R}^{L \times D}$ via an input embedding e_θ and a stack of Transformer blocks T_θ of hidden dimension D. A linear LM head W maps each position's representation $o_\theta(x)$ to logits $l \in \mathbb{R}^V$, and the parameters (θ, W) are trained to minimize the next-token prediction cross-entropy loss.[1]

A conventional suite of M models uses M independent parameter sets $\{(\theta_m, W_m)\}_{m=1}^M$. We instead enforce a strict nesting of parameters with the exception of the LM head:

$$\theta_1 \subset \theta_2 \subset \cdots \subset \theta_M = \theta.$$

Each sub-model T_{θ_m} is a separate Transformer stack with its own hidden dimension (or width) D_m, number of layers (or depth) n_m, and an LM head W_m. We require non-decreasing hidden dimensions $D_1 \leq D_2 \leq \cdots \leq D_M$. Unlike standard early exiting, which assumes a fixed hidden dimension across exits, the Matryoshka design lets each sub-model use a width tailored to its size,

[1] We notationally distinguish the LM head parameters to simplify the definition of our Matryoshka suite.

giving more flexibility in the architectural design at every sub-model size compared to early-exit methods (Elhoushi et al., 2024). Moreover, because $\theta_m \subset \theta_{m+1}$, running the full architecture produces all M sub-model logits in a single forward pass while activating fewer parameters and writing less KV cache than M independent forward passes, facilitating distillation during training and multi-model inference methods such as speculative decoding.

Inter-model Junction The main modeling difficulty lies in passing the output $o_\theta^m(x) \in \mathbb{R}^{D_m}$ of sub-model T_{θ_m} as the input to sub-model $T_{\theta_{m+1}}$, which operates in $\mathbb{R}^{D_{m+1}}$. We design a junction mechanism that does not introduce any additional parameters.

The most naive approach is to concatenate $o_\theta^m(x)$ with a fresh embedding $e_\theta^{m+1}(x) \in \mathbb{R}^{D_{m+1}-D_m}$ extending to the "new" dimensions that sub-model $m{+}1$ adds. However, this creates a magnitude mismatch: Transformer outputs typically have much larger norms than input embeddings, which destabilizes training in the low-index channels. We address this by rescaling $o_\theta^m(x)$ to match the norm of $e_\theta^{m+1}(x)$ before concatenation. Formally, we build $o_\theta^{m+1}(x)$ as:

$$\begin{aligned} \tilde{o}_\theta^m(x) &= o_\theta^m(x) \cdot \frac{\|e_\theta^{m+1}(x)\|_2}{\|o_\theta^m(x)\|_2} \\ o_\theta^{m+1}(x) &= T_\theta^{m+1}\big(\mathrm{concat}\big(e_\theta^{m+1}(x),\ \tilde{o}_\theta^m(x)\big)\big) \quad . \end{aligned} \tag{1}$$

The first D_m channels of the input to T_θ^{m+1} carry information from sub-model m, while the remaining $D_{m+1} - D_m$ channels are freshly initialized, analogous to the input embedding of a standalone model. Each sub-model has a dedicated LM head $W_\theta^m \in \mathbb{R}^{V \times D_m}$ and is trained with its own cross-entropy loss $\mathcal{L}_{\mathrm{ce}}^m$. Both design choices matter empirically: dropping the norm rescaling, or zeroing out the fresh embedding, degrades validation PPL across all sub-model sizes (Table 3).

Integrated Distillation In classical suite training, distillation from the largest model into smaller ones (Peng et al., 2025; Gu et al., 2025) requires either storing teacher logits offline or running the teacher in parallel during student training. Each option adds significant compute or storage cost. In a Matryoshka suite, every forward pass through θ_M already produces logits for all M sub-models simultaneously, making online distillation from the largest sub-model to all smaller ones free.

Concretely, we add a distillation term to each sub-model $m < M$, defined as the cross-entropy between the teacher and student next-token distributions:

$$\mathcal{L}_d^{M \to m} = -\sum_{v=1}^{V} \texttt{stop_grad}(\sigma(l_t^M)_v)\ \log \sigma(l_t^m)_v, \tag{2}$$

where σ denotes the softmax and t the token position. This is equivalent to the KL divergence up to the teacher entropy, which carries no gradient to the student. The total loss for sub-model m is a convex combination:

$$\mathcal{L}^m = (1 - \alpha_d)\, \mathcal{L}_{\mathrm{ce}}^m + \alpha_d\, \mathcal{L}_d^{M \to m}, \tag{3}$$

and the overall training objective sums over sub-models: $\mathcal{L} = \sum_{m=1}^{M} \mathcal{L}^m$. The distillation coefficient $\alpha_d \in [0, 1]$ controls the strength of the teacher signal. We study its effect in Appendix A. We observe that α_d should be set to lower values compared to offline setups.

Speculative Decoding At inference time, any pair (m, m') with $m < m'$ forms a natural draft-verifier pair for speculative decoding (Leviathan et al., 2023). Sub-model m generates a draft sequence of γ tokens, which sub-model m' then verifies in a single forward pass. Draft-acceleration techniques such as EAGLE (Li et al., 2024a) compose with our approach (Appendix K); here, we explore how sharing weights and KV cache between draft and verifier makes larger, stronger drafts practical.

Two properties make Matryoshka suites particularly well-suited for this setup. Because $\theta_m \subset \theta_{m'}$, the verifier's first $n_1 + \cdots + n_m$ layers coincide with the draft model's layers. The KV cache computed during drafting is fully reused by the verifier for those shared layers; only the $n_{m+1} + \cdots + n_{m'}$ additional layers need to run. This eliminates the memory cost of maintaining a separate draft model. Draft-verifier alignment is higher than in independently trained suites of the same sizes, as we confirm in Section 3, which directly translates to a higher token acceptance rate.

		Params	Cumul.	D_m	Layers	Heads	Head dim	Int. size
Vanilla	500M	0.50B		1 024	24	16	64	4 096
	1.5B	1.51B		1 728	28	18	96	6 912
	3B	3.19B		2 560	28	20	128	10 240
Matryoshka	500M	0.50B	0.50B	1 024	24	16	64	4 096
	1.5B	0.98B	1.48B	2 304	10	24	96	9 216
	3B	1.72B	3.20B	4 352	5	34	128	17 408

Table 1: Architecture of the Vanilla baseline and the 3B Matryoshka suite. *Params* is the incremental block size; *Cumul.* is the running total reached at each exit point. The suite totals are 5.2B parameters for Vanilla and 3.2B for Matryoshka (-38%), at matched exit sizes.

Together, these two properties make non-tiny drafts practical. Standard speculative decoding favors draft models one or two orders of magnitude smaller than the verifier (Leviathan et al., 2023; Zhou et al., 2024; Li et al., 2024a), as larger drafts' cost and KV cache grow faster than their acceptance rate. We show in Section 3.2.3 that a 500M / 3B pair (1:6 size ratio) is already in this unfavorable regime, while the Matryoshka 500M / 3B pair turns the same configuration into a 20–40% speedup over its own standard decoding by sharing its KV cache and early layers.

3 Experiments

3.1 Experimental Setup

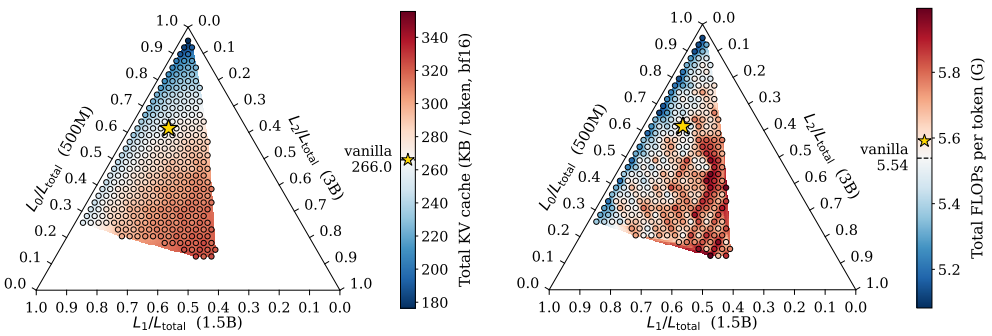
Architecture We train a Matryoshka suite from scratch with three nested sub-models totalling 3B parameters, and compare it to an equivalent traditional suite; configurations are summarized in Table 1. The main architectural design choice is how to allocate the width and depth to each sub-model while matching the target parameter count.[2] We adopt a deployment-oriented viewpoint and aim to match both memory and compute footprints across suite types.

Because Matryoshka sub-models combine blocks of different widths, the depth budget that balances per-token FLOPs and KV cache differs from a uniform-width architecture. Using closed-form formulas for parameter count, KV cache, and per-token inference FLOPs (Appendix B), we find $L = 39$ total layers to be the sweet spot for the 3B Matryoshka suite, against $L_v = 28$ for the Vanilla 3B model: shallower budgets reduce memory per token below the Vanilla footprint, while deeper ones (e.g. $L = 50$) inflate KV cache (Appendix C).

Within the $L = 39$ budget, we sweep all feasible depth triplets (n_1, n_2, n_3) summing to 39, fixing target sizes (500M, 1.5B, 3B) and attention head dimensions (64, 96, 128), and for each triplet solve for the head count that best matches each target. Figure 2 reports the resulting KV cache and per-token FLOPs of the full 3B forward pass relative to a Vanilla 3B baseline. Our chosen configuration $(n_1, n_2, n_3) = (24, 10, 5)$ closely matches both KV cache and per-token FLOPs to the Vanilla 3B baseline. This shape is selected without any quality metric, only by matching these footprints with a reasonable width/depth balance. We notably choose to use the same width and depth for the 500M sub-model in both suites, to provide a strictly comparable data point. All sub-models use the SmolLM2 tokenizer (Allal et al., 2025) (49,152 tokens) and RoPE (Su et al., 2024) ($\theta = 10^5$).

Training Setup Training uses 35B tokens from FineWeb-Edu (Penedo et al., 2024), a high-quality education-focused web corpus, with sequences packed to length 2,048. The optimizer is AdamW with $(\beta_1, \beta_2) = (0.9, 0.95)$, $\epsilon = 10^{-8}$, peak learning rate 4×10^{-4}, weight decay 0.01, and gradient clipping at 1.0. We use a warmup-stable-decay (WSD) schedule with 3,000 cooldown steps over 33,000 total steps, a batch size of 512 sequences, and bf16 mixed precision. The distillation coefficient is set to $\alpha_d = 0.3$ for all main-suite runs, chosen from a smaller-scale proxy sweep (Appendix A). Combined training across both suites amounts to 52 GPU-days on NVIDIA B200.

[2]Given a target parameter count, and using standard configurations for FFN layers ($D_m \to 4 \cdot D_m \to D_m$), fixing the depth n_m of each sub-model determines its width D_m.



(a) KV cache per token

(b) Theoretical FLOPs per token

Figure 2: Ternary sweep over all feasible depth triplets (n_1, n_2, n_3) summing to 39 layers for a (500M, 1.5B, 3B) Matryoshka suite (see Appendix D for a guide to reading these plots). Each point's position encodes the fraction of total layers assigned to each sub-model; its color encodes KV cache (left) or FLOPs (right) for the 3B forward pass relative to the Vanilla 3B baseline (dashed line, 266.0 KB/token and 5.54 GFLOPs/token). Blue points lie below the Vanilla baseline; red points lie above. The gold star marks our chosen configuration $(24, 10, 5)$, which sits near the footprint-equivalent frontier.

As our primary baseline we train a Vanilla suite of three independent Llama-style (AI@Meta, 2024) models at roughly the same sizes (0.50B, 1.51B, 3.19B; see Table 1), using the same tokenizer, data, and hyperparameters as the Matryoshka suite. The 500M and 1.5B Vanilla baselines mirror the SmolLM2 (Allal et al., 2025) configurations, and the 3B baseline follows the Llama-3.2-3B shape. Both suites are trained for 35B tokens; we additionally cool down and evaluate Vanilla checkpoints at 23B tokens to obtain a compute-matched reference.

3.2 Results and Analysis

We evaluate the Matryoshka suite against the Vanilla baseline on benchmark accuracy (Section 3.2.1), cross-model alignment (Section 3.2.2), and speculative decoding throughput (Section 3.2.3), compare against MatFormer (Section 3.2.4), and ablate our design choices on a 200M suite (Section 3.2.5).

3.2.1 Downstream Evaluation

We evaluate both suites zero-shot on seven benchmarks that are standard at this scale: ARC-Easy and ARC-Challenge (Clark et al., 2018), HellaSwag (Zellers et al., 2019), LAMBADA (Paperno et al., 2016), OpenBookQA (Mihaylov et al., 2018), PIQA (Bisk et al., 2020), and Winogrande (Sakaguchi et al., 2021). We also report average byte perplexity on five out-of-distribution corpora (WikiText-103 (Merity et al., 2017), C4 (Dodge et al., 2021), PG-19 (Rae et al., 2020), arXiv, and PubMed Central); the per-corpus values are reported in Appendix F.

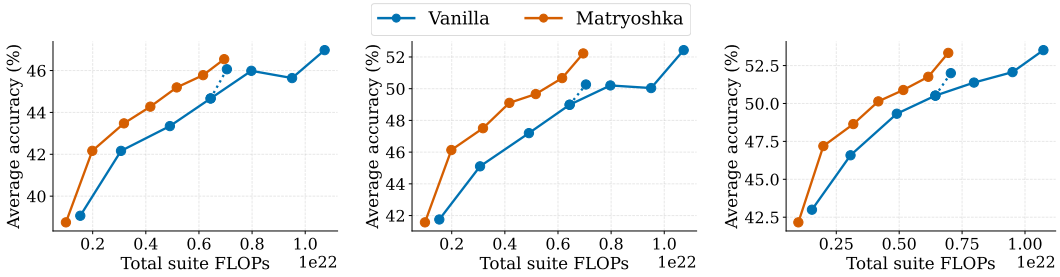
Table 2 shows that the Matryoshka suite reaches near-parity with the Token-matched Vanilla baseline on average accuracy at every size (within 0.5 points) while using 36% less training compute, and outperforms the FLOPs-matched Vanilla baseline at every size by +0.4 to +1.9 points. On average OOD byte perplexity, the Matryoshka suite outperforms the Token-matched Vanilla baseline at the 1.5B and 3B scales (and ties at 500M), with the largest single-corpus advantage on arXiv (Appendix F). Figure 3 shows the same comparison as a function of total suite training FLOPs. The Matryoshka curve dominates the Vanilla curve at every compute level for every sub-model size. Near-parity also holds after SFT on Tulu-3 (Appendix J). Our 3B runs are single-seed, but 5-seed runs at the 200M scale show a run-to-run noise of $\sim 0.1\%$ PPL (Appendix M).

3.2.2 Training Dynamics and Model Agreement

Figure 4 tracks validation perplexity, pairwise KL divergence, and pairwise top-1 next-token agreement throughout training. Matryoshka perplexity is on par with Vanilla perplexity at every checkpoint and every size (Figure 4a); the parameter sharing and the junction mechanism do not harm language modeling quality. The small per-size gaps that remain are not intrinsic to the nested architecture: a

Model	ARC-E	ARC-C	HS	LAMBADA	OBQA	PIQA	WG	**Avg acc. ↑**	**OOD PPL ↓**
500M parameters									
Vanilla (=FLOPs)	56.9	31.2	43.2	36.5	34.4	68.8	51.5	46.1	2.295
Vanilla (=Tokens)	**59.0**	**31.5**	**44.8**	**37.8**	**35.2**	68.6	52.0	**47.0**	**2.263**
Matryoshka	58.8	30.7	44.4	36.1	33.8	**69.0**	**53.0**	46.5	2.265
1.5B parameters									
Vanilla (=FLOPs)	62.2	34.6	50.3	42.6	37.6	71.8	52.7	50.3	2.171
Vanilla (=Tokens)	**65.8**	35.7	**53.5**	43.9	39.4	**72.1**	**56.7**	**52.4**	2.139
Matryoshka	63.3	**35.8**	53.3	**44.7**	**40.2**	71.9	56.4	52.2	**2.121**
3B parameters									
Vanilla (=FLOPs)	63.2	35.8	53.7	44.4	37.8	72.4	56.8	52.0	2.143
Vanilla (=Tokens)	64.4	37.3	**56.9**	**46.3**	38.2	**73.8**	**57.8**	**53.5**	2.097
Matryoshka	**64.6**	**38.0**	55.9	44.9	**39.0**	73.2	57.7	53.3	**2.067**

Table 2: Per-benchmark accuracy and out-of-distribution perplexity. Vanilla (=FLOPs) uses the same 7.0e21 FLOPs as Matryoshka (23B tokens); Vanilla (=Tokens) uses the same 35B tokens at 57% more compute. Avg acc. averages the seven benchmarks (WG = Winogrande); OOD PPL averages WikiText-103, C4, PG-19, arXiv, and PubMed Central (Appendix F). Matryoshka reaches near-parity with Token-matched on accuracy and beats it on OOD PPL at 1.5B and 3B.



(a) 500M (b) 1.5B (c) 3B

Figure 3: Average benchmark accuracy as a function of total suite training FLOPs for each sub-model size. The Matryoshka curve lies above the compute-matched Vanilla baseline at every sub-model size, with the largest gap at 1.5B. The dotted segment at the end of the Vanilla curve is a 2,000-step learning-rate cooldown branched off the 21B-token checkpoint, reaching 23B tokens.

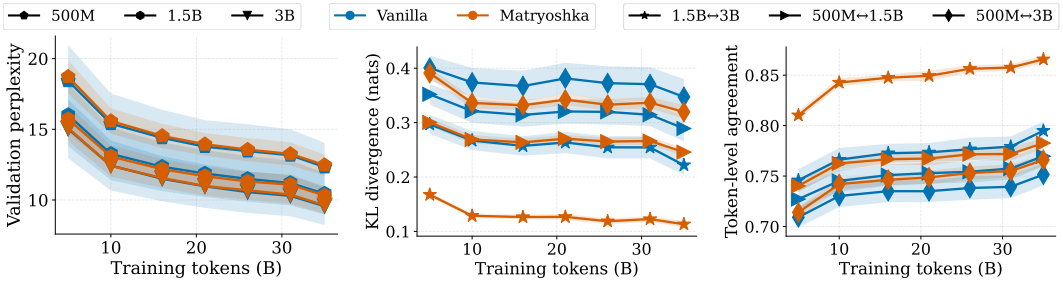
proxy-scale sweep at 200M shows that simply rebalancing the per-sub-model loss weights, currently uniform in our recipe, closes a substantial fraction of the residual gap (Appendix G).

A clear difference appears in cross-model alignment, as Matryoshka pairs exhibit noticeably lower KL divergence (Figure 4b) and higher token-level agreement (Figure 4c) than independently trained Vanilla pairs of matched sizes, with the effect strongest for the (1.5B, 3B) pair (+5.7% final agreement). This is a direct consequence of both the shared-weight structure and of online distillation across training. Independently trained models have no such constraint and develop different representations. Next-token agreement governs speculative decoding efficiency (Section 3.2.3).

3.2.3 Speculative Decoding Throughput

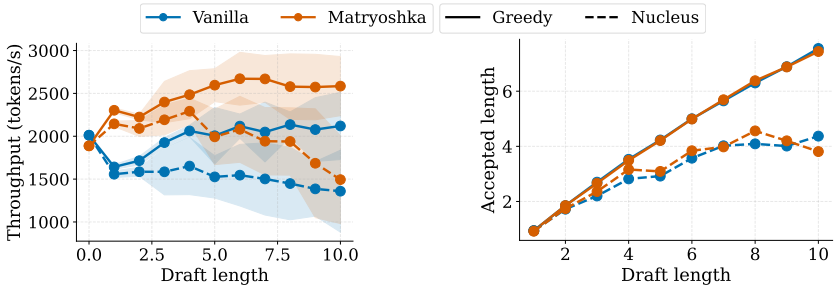
We measure speculative decoding throughput on an NVIDIA A100 GPU using the 500M sub-model as draft and the 3B sub-model as verifier, under both greedy and nucleus ($p = 0.9$) sampling. Each run generates 512 tokens per FineWeb-Edu prompt (truncated to 1,024 tokens) at temperature 0.4; each metric is averaged over roughly 1M generated tokens. The maximum batch size that fits in 80GB of VRAM is 64 for the Vanilla pair and 102 for the Matryoshka pair, thanks to the shared draft/verifier cache; larger batches make drafts faster relative to verification by reducing bandwidth latency.

Figure 5 shows that the Matryoshka pair achieves higher accepted length than the Vanilla pair across most draft lengths in the nucleus case (+5% on average), which directly reflects the agreement-rate



(a) Perplexity (b) KL divergence (c) Agreement rate

Figure 4: Training dynamics for the 3B Vanilla and Matryoshka suites. Matryoshka matches Vanilla perplexity throughout training while achieving significantly lower cross-model KL divergence and higher next-token agreement.



(a) Throughput (b) Accepted length

Figure 5: Speculative decoding analysis with a 500M draft and 3B verifier. Solid lines show greedy decoding; dashed lines show nucleus sampling ($p = 0.9$). The Matryoshka pair achieves higher throughput than the Vanilla pair across most draft lengths. In the nucleus case, the accepted length is generally higher for Matryoshka. Draft length of 0 corresponds to standard decoding.

gap in Figure 4c. The higher acceptance translates to higher throughput: at a draft length of 6, Matryoshka reaches 2,650 tokens/s in greedy mode against 2,100 for Vanilla (a 26% speedup), and the gain is preserved under nucleus sampling. Crucially, the Vanilla 500M / 3B pair barely beats plain autoregressive inference and is actually slower under nucleus sampling. At this size ratio, the draft-side KV cache and drafting FLOPs do not amortize against the acceptance rate. Matryoshka remains 9–18% faster at matched batch size (Appendix L, with implementation details).

3.2.4 Comparison with MatFormer

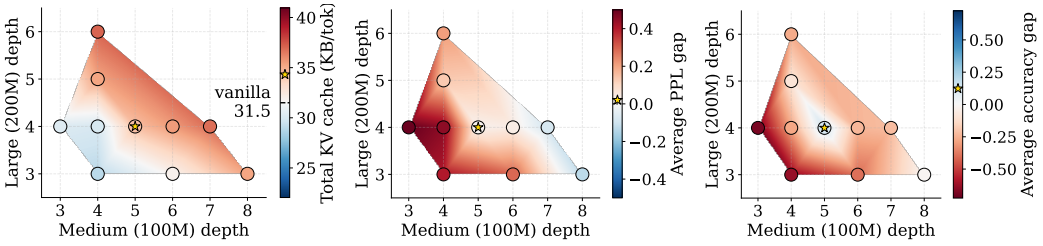
Matryoshka also compares favorably to MatFormer (Devvrit et al., 2023) on both resource footprint and quality (Appendix E). At a fixed 3B universal shape, MatFormer cannot extract sub-models below $\sim$1.05B parameters and keeps a constant 280 KB/token KV cache, whereas Matryoshka goes down to 500M parameters and 96 KB/token. At the 200M scale, Matryoshka-200M reaches 17.92 perplexity against 19.34 for MatFormer-XL, and Matryoshka-100M matches MatFormer-M (139M) with $\sim$40% less KV cache.

3.2.5 Ablations

We ablate three design choices: the inter-model junction (with and without the norm-rescaling and the fresh embedding), the use of distillation (with and without), and the allocation of layers across sub-models. Our ablations are performed on a 200M proxy suite, composed of sub-models of 50M / 100M / 200M, where Matryoshka trades 41% less training compute (on one A100, Appendix B) for producing all three sub-models in a single run. Each full run for a 200M Matryoshka suite costs roughly five days of compute on a single A100 GPU, for 20B training tokens. Appendix A reports a separate sweep of the distillation coefficient α_d.

Variant	Validation PPL ↓			
	50M	100M	200M	Average
Vanilla suite	26.07	18.87	15.21	20.05
Zero padding	26.98	18.83	16.01	20.61
Input embs. + Distil.	26.69	18.44	15.67	20.27
Input embs. + Norm	26.62	18.57	**15.46**	20.22
Input embs. + Norm + Distil. (ours)	**26.16**	**18.43**	15.62	**20.07**

Table 3: Method ablation at the 200M scale (validation PPL, lower is better). Our recipe stays within 0.02 average PPL of the Vanilla baseline; each simpler junction widens the gap.



(a) Total KV cache per token (b) Mean PPL gap (c) Mean benchmark acc. gap

Figure 6: Shape ablation at the 200M scale. Each point is one Matryoshka variant with a shared 50M base sub-model; axes are the additional layers of the 100M and 200M blocks. **(a)** Total KV cache per token, with the same-architecture Vanilla 200M baseline marked on the colorbar. **(b)** Mean perplexity gap versus the Vanilla baseline, averaged over the three sub-model sizes (50M / 100M / 200M); blue means below Vanilla. **(c)** Mean accuracy gap (in percentage points) on the seven downstream benchmarks of Table 2, averaged over the three sub-model sizes; blue means above Vanilla.

Inter-model Junction Recall that our recipe concatenates a fresh input embedding with a norm-rescaled model output. We compare it against two simpler alternatives in Table 3: concatenation without norm rescaling, and a zero junction that replaces the concatenated input embeddings with zeros. Our recipe matches the Vanilla baseline within 0.02 average PPL; removing the norm match introduces a +0.2 PPL gap, with a stronger impact on the smaller model; the zero junction underperforms at all model sizes, with a +0.54 PPL gap over our method. We also report the effect of removing distillation from the recipe, which causes a +0.15 PPL average gap more strongly noticeable on the smaller models. Both components are crucial for the performance of smaller sub-models.

Width/Depth of Sub-Models The depth triplet of the main 3B suite is one operating point in a rather large design space, as we show in Figure 2. At the 200M scale, the feasible configuration space is much less dense, which allows a coarser exploration. We sweep ten variants at the 200M scale that differ only in the depth split between the 100M and 200M blocks (Figure 6). The width of each sub-model is then fixed to match the parameter size as closely as possible (within 5% in practice). In Figure 6, we observe that several variants are comparable to the Vanilla suite both in terms of total KV cache growth and of average performance gap: the chosen $(L_1, L_2) = (5, 4)$ shape (gold star) remains within $\sim$10% of the KV cache footprint of the Vanilla baseline at very low average PPL cost, and other variants achieve a further KV reduction with comparable PPL. Additional views (mean ΔNLL, per-token FLOPs, 1D KV-cache summary) are reported in Appendix H. We also report the downstream benchmark accuracy of the chosen configuration in Appendix I, where the same near-parity pattern as in the 3B suite holds.

4 Related Work

Early Exiting Early exiting makes predictions at intermediate layers to avoid a full forward pass when the model is already confident (Teerapittayanon et al., 2016). In language modeling, this idea was applied by Schuster et al. (2022), Elbayad et al. (2020), and via sparse layer routing (Raposo

et al., 2024), while Elhoushi et al. (2024) combine layer dropout during pretraining with early-exit self-speculative decoding. These methods operate within a fixed-width architecture, whereas our sub-models differ in both width and depth, and exits correspond to fully trained sub-models with their own LM heads rather than intermediate probes.

Speculative Decoding and Distillation Speculative decoding (Leviathan et al., 2023) accelerates autoregressive inference by letting a small draft model propose candidate continuations that a larger verifier accepts or rejects in parallel. Its efficiency depends on draft-verifier alignment, which distilling the verifier into the draft improves (Zhou et al., 2024), and Zhang et al. (2024) combine it with layer skipping for self-speculative decoding. In the Matryoshka framework, the verifier contains the Transformer stack of the draft model, which is a form of static self-speculative decoding, and distillation from the largest sub-model to all smaller ones is a natural by-product of every training step, with negligible additional cost. More recently, the EAGLE line of work (Li et al., 2024a;b; 2026) trains lightweight feature-level draft heads on top of the verifier's penultimate representations, achieving 3–5$\times$ speedups, and Cai et al. (2024b) attach multiple parallel decoding heads to a single model to generate several future tokens at once. These methods are orthogonal to our work, and EAGLE composes with our models (Appendix K). For pretraining-time distillation, Peng et al. (2025) explore loss functions and teacher-student configurations, and Gu et al. (2025) show that offline teacher logits reduce data requirements by $2.4\times$. Our approach alleviates the need for offline distillation by making online distillation cheap.

Elastic Compression and Model Expansion Flextron (Cai et al., 2024a), LLaMaFlex (Cai et al., 2025), and Nemotron Elastic (Taghibakhshi et al., 2025) convert a trained LLM into a family of smaller models via elastic continued training, building on slimmable networks (Yu et al., 2019; Yu & Huang, 2019) and Once-for-All (Cai et al., 2020). They are complementary to our setting: they recover a family cheaply given a sunk pretraining cost, whereas we target the from-scratch suite-training problem. Conversely, Yano et al. (2025) build a model family small-to-large through progressive model expansion, which requires fresh data or repeated training on the same data at each stage.

Matryoshka LLMs Kusupati et al. (2022) introduce Matryoshka Representation Learning, where embedding prefixes of any length are trained to be informative. Devvrit et al. (2023) apply the same principle to Transformers via MatFormer, nesting sub-models along the FFN hidden size while keeping depth, attention heads, and embeddings fixed. This yields FLOPs and parameter savings, but the KV-cache footprint is identical across sub-models, which are not independently deployable checkpoints. MatFormer thus targets a latency knob within a single deployment regime, rather than a suite spanning various target regimes with constrained memory usage and detachable per-size checkpoints. Our framework nests both width and depth, and produces standalone checkpoints, at the cost of a discrete rather than combinatorial set of sizes. We show that this pays off on both fronts (Section 3.2.4): Matryoshka sub-models scale down in parameters and KV cache where MatFormer's cannot, and achieve a better size–perplexity frontier.

5 Conclusion

We introduce Matryoshka Language Model Suites, a pretraining framework that stacks sub-models of increasing size into a single nested architecture trained end-to-end. The key insight is that the natural structure of a model suite, a set of models sharing the same training distribution, can be exploited by nesting at lower cost than independent training. At the 3B scale, our Matryoshka suite reaches near-parity with independently trained baselines on standard benchmarks while using 36% less training compute, allowing low-cost online distillation and enabling speculative decoding with higher throughput and no memory overhead for the draft model. Our ablations produce a solid basis for scaling up Matryoshka suites to larger sizes.

Our work paves the way for a new line of research on *joint* pretraining of model suites, where sub-models of different sizes are co-designed rather than trained in isolation. Open questions include how the recipe scales with more sub-models and larger budgets, how to allocate capacity and loss weight across sub-models, and how nested suites interact with post-training beyond SFT (Appendix J).

AI USE STATEMENT

In this work, we used generative AI tools for implementing standard methods (e.g., evaluation and plotting scripts), for visualizing results, and for drafting and editing parts of the paper, including proofreading and formatting. We have not used generative AI tools to generate research ideas, design experiments, or produce experimental results: all reported numbers come from our own training and evaluation runs. We have reviewed all AI-assisted work: AI-generated code was verified and tested for correctness by the authors, AI-drafted or AI-edited text was checked and revised by the authors against our experimental results, and all references were verified manually. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects or personal data collection. All models are pretrained on FineWeb-Edu (Penedo et al., 2024), a publicly available, filtered web corpus, and fine-tuned on the publicly available Tulu-3 data (Lambert et al., 2024). As with any language model trained on web data, our models may reproduce biases or factual errors present in their training data; they are small research models that have not undergone safety alignment, and should not be deployed in user-facing applications without further evaluation. Our method primarily aims at reducing the compute required to pretrain model suites, which we expect to lower the energy cost and environmental footprint of producing families of language models.

REPRODUCIBILITY STATEMENT

Our method is fully specified in Section 2, including the inter-model junction (Equation (1)) and the training objective (Equations (2) and (3)). The architectures of both suites are given in Table 1, and the training data, tokenizer, optimizer, and schedule are described in Section 3.1. The formulas used for parameter counts, KV cache, and training and inference FLOPs are detailed in Appendix B, and the procedure used to choose the layer budget and depth allocation in Appendices C and D. Evaluation benchmarks and the out-of-distribution perplexity protocol are described in Section 3.2.1 and appendix F, the speculative decoding implementation and benchmarking setup in Section 3.2.3 and appendix L, and the fine-tuning setup in Appendix J. We report multi-seed results at the 200M scale in Appendix M. All datasets used in this work are publicly available, and we will release our trained checkpoints and a `transformers`-compatible implementation upon publication.

REFERENCES

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. Smollm2: When smol goes big – data-centric training of a small language model, 2025. URL https://arxiv.org/abs/2502.02737.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: reasoning about physical commonsense in natural language. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pp. 7432–7439. AAAI Press, 2020. doi: 10.1609/AAAI.V34I05.6239. URL https://doi.org/10.1609/aaai.v34i05.6239.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HylxE1HKwS.

Ruisi Cai, Saurav Muralidharan, Greg Heinrich, Hongxu Yin, Zhangyang Wang, Jan Kautz, and Pavlo Molchanov. Flextron: Many-in-one flexible large language model. In *Forty-first International Conference on Machine Learning*, 2024a. URL https://openreview.net/forum?id=9vKRhnflAs.

Ruisi Cai, Saurav Muralidharan, Hongxu Yin, Zhangyang Wang, Jan Kautz, and Pavlo Molchanov. LLamaflex: Many-in-one LLMs via generalized pruning and weight sharing. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=AyC4uxx2HW.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *Forty-first International Conference on Machine Learning*, 2024b. URL https://openreview.net/forum?id=PEpbUobfJv.

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, 2018. URL https://arxiv.org/abs/1803.05457.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro/blob/main/DeepSeek_V4.pdf.

Fnu Devvrit, Sneha Kudugunta, Aditya Kusupati, Tim Dettmers, Kaifeng Chen, Inderjit Dhillon, Yulia Tsvetkov, Hannaneh Hajishirzi, Sham Kakade, Ali Farhadi, and Prateek Jain. Matformer: Nested transformer for elastic inference. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@NeurIPS 2023)*, 2023. URL https://openreview.net/forum?id=93BaEweoRg.

Jesse Dodge, Maarten Sap, Ana Marasović, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 1286–1305, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.98. URL https://aclanthology.org/2021.emnlp-main.98/.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive transformer. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJg7KhVKPH.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed Aly, Beidi Chen, and Carole-Jean Wu. LayerSkip: Enabling early exit inference and self-speculative decoding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.681. URL https://aclanthology.org/2024.acl-long.681/.

Yuxian Gu, Hao Zhou, Fandong Meng, Jie Zhou, and Minlie Huang. MiniPLM: Knowledge distillation for pre-training language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=tJHDw8XfeC.

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, 2020. URL https://arxiv.org/abs/2001.08361.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. Matryoshka representation learning. In *Advances in Neural Information Processing Systems*, December 2022.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: speculative sampling requires rethinking feature uncertainty. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-2: Faster inference of language models with dynamic draft trees. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 7421–7432, Miami, Florida, USA, November 2024b. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.422. URL https://aclanthology.org/2024.emnlp-main.422/.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=4exx1hUffq.

Stephanie Lin, Jacob Hilton, and Owain Evans. TruthfulQA: Measuring how models mimic human falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3214–3252, 2022.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2381–2391, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1260. URL https://aclanthology.org/D18-1260/.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernandez. 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, Berlin, Germany, August 2016. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P16-1144.

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 *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=n6SCkn2QaG.

Hao Peng, Xin Lv, Yushi Bai, Zijun Yao, Jiajie Zhang, Lei Hou, and Juanzi Li. Pre-training distillation for large language models: A design space exploration. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3603–3618, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.181. URL https://aclanthology.org/2025.acl-long.181/.

Jackson Petty, Sjoerd Steenkiste, Ishita Dasgupta, Fei Sha, Dan Garrette, and Tal Linzen. The impact of depth on compositional generalization in transformer language models. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 7239–7252, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-long.402. URL https://aclanthology.org/2024.naacl-long.402/.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SylKikSYDH.

David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. Mixture-of-depths: Dynamically allocating compute in transformer-based language models, 2024. URL https://arxiv.org/abs/2404.02258.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106, August 2021. ISSN 0001-0782. doi: 10.1145/3474381. URL https://doi.org/10.1145/3474381.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 35:17456–17472, 2022.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomput.*, 568(C), February 2024. ISSN 0925-2312. doi: 10.1016/j.neucom.2023.127063. URL https://doi.org/10.1016/j.neucom.2023.127063.

Ali Taghibakhshi, Sharath Turuvekere Sreenivas, Saurav Muralidharan, Ruisi Cai, Marcin Chochowski, Ameya Sunil Mahabaleshwarkar, Yoshi Suhara, Oluwatobi Olabiyi, Daniel Korzekwa, Mostofa Patwary, Mohammad Shoeybi, Jan Kautz, Bryan Catanzaro, Ashwath Aithal, Nima Tajbakhsh, and Pavlo Molchanov. Nemotron elastic: Towards efficient many-in-one reasoning llms, 2025. URL https://arxiv.org/abs/2511.16664.

Yi Tay, Mostafa Dehghani, Samira Abnar, Hyung Chung, William Fedus, Jinfeng Rao, Sharan Narang, Vinh Tran, Dani Yogatama, and Donald Metzler. Scaling laws vs model architectures: How does inductive bias influence scaling? In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 12342–12364, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.825. URL https://aclanthology.org/2023.findings-emnlp.825/.

Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469, 2016. URL https://api.semanticscholar.org/CorpusID:2916466.

Oskar van der Wal, Pietro Lesci, Max Müller-Eberstein, Naomi Saphra, Hailey Schoelkopf, Willem Zuidema, and Stella Biderman. PolyPythias: Stability and outliers across fifty language model pre-training runs. In *The Thirteenth International Conference on Learning Representations*, 2025.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Kazuki Yano, Sho Takase, Sosuke Kobayashi, Shun Kiyono, and Jun Suzuki. Efficient construction of model family through progressive training using model expansion. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=fuBrcTH8NM.

Jiahui Yu and Thomas Huang. Universally Slimmable Networks and Improved Training Techniques . In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 1803–1811, Los Alamitos, CA, USA, November 2019. IEEE Computer Society. doi: 10.1109/ICCV.2019.00189. URL https://doi.ieeecomputersociety.org/10.1109/ICCV.2019.00189.

Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gMCsAqY7.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In Anna Korhonen, David Traum, and Lluís Màrquez (eds.), *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1472. URL https://aclanthology.org/P19-1472/.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. Draft & verify: Lossless large language model acceleration via self-speculative decoding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11263–11282, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.607. URL https://aclanthology.org/2024.acl-long.607/.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.

Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. Distillspec: Improving speculative decoding via knowledge distillation. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=rsY6J3ZaTF.

A Distillation Ablation

We sweep the distillation coefficient $\alpha_d \in \{0, 0.3, 0.5, 0.7\}$ on the 200M proxy suite (sub-models 50M / 100M / 200M, otherwise identical recipe to the main runs). Figure 7(a) reports the validation cross-entropy of each Matryoshka sub-model relative to its Vanilla-baseline counterpart; Figure 7(b) reports the KL divergence of each smaller sub-model to the 200M teacher, which captures how aligned each student becomes with the teacher under each setting.

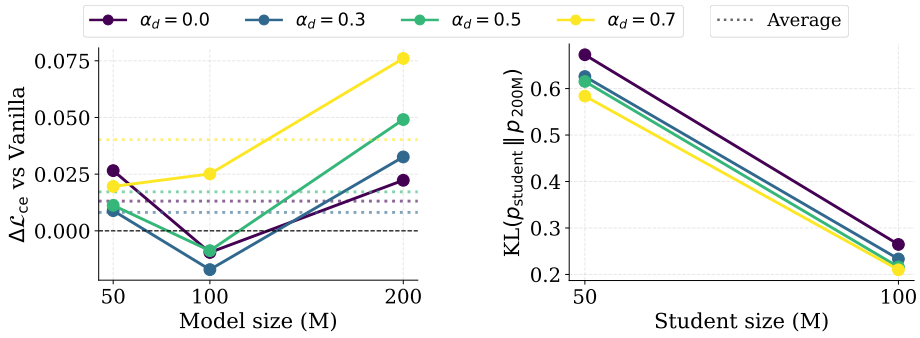
In Figure 7(a), $\alpha_d = 0$ (no distillation) leaves the 50M and 200M sub-models above Vanilla while the intermediate 100M sub-model already sits below it; turning on the distillation signal ($\alpha_d = 0.3$) closes most of the 50M gap and pushes the average gap to its minimum across the four settings. Pushing α_d higher is counter-productive: at $\alpha_d = 0.5$ the largest sub-model starts trailing Vanilla, and $\alpha_d = 0.7$ degrades every size. The pattern in Figure 7(b) mirrors this — KL drops monotonically with α_d, confirming that larger coefficients pull the student distribution closer to the teacher's, but at a cost on absolute language modeling quality once the cross-entropy term is too downweighted. We therefore use $\alpha_d = 0.3$ for the main 3B suite.

B Compute, Memory, and Training-FLOPs Accounting

This appendix details the formulas used throughout the paper to report parameter counts, KV cache footprint, theoretical inference FLOPs, and training FLOPs. We use the following notation: V is the vocabulary size, L the number of Transformer layers, D the hidden dimension (equal to the product of the number of attention heads and the head dimension), and N_{tok} the number of training tokens.

Parameter count For a standalone Transformer with input embedding, L blocks of width D with $4D$ FFN intermediate dimension and an LM head, the total parameter count is

$$N_{\text{std}}(L, D) = 2VD + 16LD^2, \tag{4}$$



(a) Validation NLL gap vs Vanilla

(b) KL($p_{\text{student}} \,\|\, p_{\,200\text{M}}$)

Figure 7: Effect of the distillation coefficient α_d on the 200M proxy suite. **(a)** Validation cross-entropy of each Matryoshka sub-model minus its Vanilla counterpart; dotted lines are the average across sub-model sizes. $\alpha_d = 0.3$ minimizes the average gap; higher values harm the larger sub-models. **(b)** KL divergence of each student sub-model (50M, 100M) to the 200M teacher, with one curve per α_d (color); KL drops monotonically as α_d increases at both student sizes.

where $2VD$ accounts for the input embedding and the LM head and $16LD^2$ accounts for the per-layer cost ($4D^2$ for self-attention projections and $12D^2$ for the FFN). For a Matryoshka suite of M nested sub-models with shapes $(n_m, D_m)_{m=1}^{M}$, $D_1 \leq \cdots \leq D_M$, the cumulative parameter count observed at sub-model m is

$$N_m^{\text{cumul}} = 2VD_m + \sum_{j \leq m} 16\, n_j D_j^2, \tag{5}$$

where the $2VD_m$ term reflects that sub-model m uses the full input embedding at width D_m together with its own LM head W_θ^m. The total trainable parameter count of the suite further adds the LM heads of the smaller sub-models, $\sum_{m<M} VD_m$.

KV Cache For a single decoded token at full attention, each Transformer layer stores both keys and values of width D, giving

$$\text{KV}(L, D) = 2\, b\, L\, D \quad \text{bytes per token}, \tag{6}$$

with b bytes per element ($b = 2$ for bf16).

Theoretical Inference FLOPs Counting two FLOPs per multiply-accumulate, the per-token forward-pass FLOPs of a standalone Transformer stack of width D and depth L is

$$F_{\text{stack}}(L, D) = 2 \cdot 16\, LD^2 = 32\, LD^2, \tag{7}$$

to which we add a single LM head application of $2VD$ FLOPs to produce the next-token logits. For the full Matryoshka suite, the per-token inference cost is the sum of the stack costs across all sub-models plus a single LM head at the deepest exit:

$$F_{\text{infer}}^{\text{Mat}} = \sum_{m=1}^{M} F_{\text{stack}}(n_m, D_m) + 2VD_M. \tag{8}$$

The values reported in Figure 2 and elsewhere use these formulas directly.

Training FLOPs We report training FLOPs by combining wall-clock GPU-hours with the nominal BF16 Tensor-Core peak of the GPU used for the run:

$$F_{\text{train}} = \frac{N_{\text{tok}}}{(\text{tokens/GPU-hour})} \times 3600 \times F_{\text{peak}}^{\text{bf16}}, \tag{9}$$

where the tokens-per-GPU-hour throughput is measured from training logs, and $F_{\text{peak}}^{\text{bf16}}$ is the GPU's nominal BF16 throughput (we use 4.5×10^{15} FLOP/s for an NVIDIA B200 and 3.12×10^{14} FLOP/s for an NVIDIA A100). This convention conflates the model's algorithmic FLOPs with hardware

utilization, but it is the most faithful comparison between Vanilla and Matryoshka training costs, as any kernel-level differences are reflected in the measured throughput. Per-suite training FLOPs are the sum of F_{train} over the runs that compose the suite (one joint run for Matryoshka, three independent runs for Vanilla). Because this convention relies on measured throughput, the same algorithmic saving can be realized differently across hardware: the 41% saving of our 200M proxy suite (Section 3.2.5) is measured on a single A100 GPU, whereas our multi-seed runs on multi-node GH200 GPUs yield a 27% saving (Appendix M).

Overhead over a Single Model While our main metric is the total compute of the suite, training the 3B Matryoshka suite only costs 16% more compute than training the Vanilla 3B model alone. This overhead mainly comes from the additional LM heads and loss computations, and should become marginal at larger scales, where LM heads represent a smaller fraction of the model.

C Choice of Total Layer Budget

The Matryoshka suite assigns a depth n_i to each sub-model under a fixed total budget $L = n_1+n_2+n_3$. Before searching the depth allocation inside L (Figure 2), we first fix L itself. The choice is constrained by two architectural costs of the 3B sub-model: per-token FLOPs and total KV cache. The Matryoshka 3B sub-model uses every layer across the three blocks, so L directly determines its KV-cache footprint and, together with the per-block widths, its FLOPs.

To make the trade-off concrete, we sweep $L \in \{28, 39, 50\}$. For each L, we randomly sample roughly 500 valid depth allocations (n_1, n_2, n_3) summing to L, and for each allocation we solve for the per-block widths that hit the (500M, 1.5B, 3B) parameter targets at fixed head dimensions $(64, 96, 128)$. We compute the resulting per-token FLOPs and total KV cache (in bf16) of the full 3B forward pass. Figure 8 plots each sample as a point colored by L; the black star marks the Vanilla 3B baseline (28 layers, $D_m = 2\,560$).

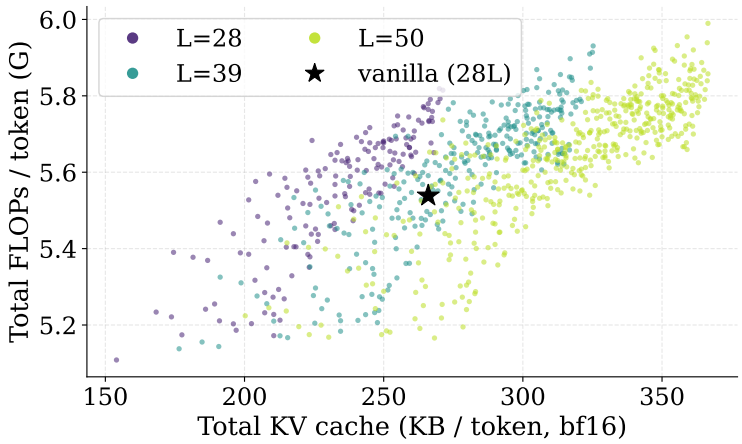


Figure 8: Per-token FLOPs and total KV cache of the Matryoshka 3B sub-model across layer budgets $L \in \{28, 39, 50\}$. Each colored point is one valid depth allocation (n_1, n_2, n_3) summing to L, with per-block widths solved to hit the (500M, 1.5B, 3B) parameter targets. The black star is the Vanilla 3B baseline (28 layers, $D_m = 2\,560$).

The three clouds occupy overlapping but distinct regions: $L = 28$ clusters at the lowest KV cache, but its FLOPs cloud sits mostly above the Vanilla baseline, because matching the 3B parameter target with only 28 shared layers forces wider matrices and higher FLOPs; $L = 50$ reaches the lowest FLOPs but pays a steep KV-cache cost ($\sim$25$-$40% above Vanilla), since KV cache scales linearly with depth and is unaffected by the FLOPs-saving width reduction; $L = 39$ contains allocations that simultaneously match or beat Vanilla on both axes — the only budget for which a strictly Pareto-dominant configuration exists. We therefore fix $L = 39$ as the depth budget and search inside it for the allocation (n_1, n_2, n_3), which is the analysis reported in Figure 2.

Note that the impact of the width-depth ratio on LM quality is also studied in the literature (Kaplan et al., 2020; Tay et al., 2023; Petty et al., 2024), with the general finding that scale matters more than shape over a reasonable range of ratios. In the case of the Matryoshka models, matching the KV cache footprint matters for matching performance with a standard counterpart (Appendix H).

D Reading the Ternary Plots

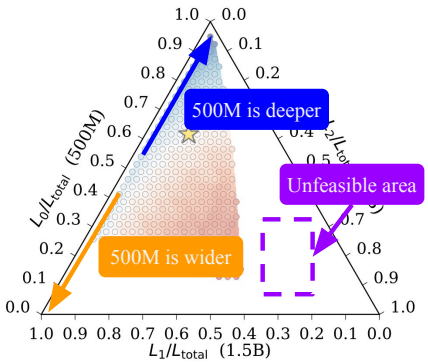


Figure 9: Reading guide for the ternary depth-allocation plots used throughout the paper. Vertices, edges, direction of motion, and the unfeasible region are annotated; the gold star marks the chosen configuration $(24, 10, 5)$.

Several figures in this paper visualize the depth allocation (L_0, L_1, L_2) across the three sub-models on a ternary diagram, with $L_0 + L_1 + L_2 = L_{\text{total}}$ fixed. Each axis carries the *fraction* of total layers assigned to one sub-model, L_i/L_{total}, so every point in the triangle corresponds to one valid normalized depth triplet. The three vertices represent the degenerate allocations where a single sub-model receives all layers; the three edges represent allocations where one sub-model receives zero layers. Figure 9 illustrates the conventions on the (500M, 1.5B, 3B) sweep at $L_{\text{total}} = 39$.

Direction of motion has a concrete architectural meaning. Moving a point toward the top vertex increases L_0/L_{total}, the fraction of layers used by the 500M sub-model. To keep the 500M parameter target fixed, the per-layer width must shrink, so points near the top vertex correspond to a 500M sub-model that is *deeper and narrower*. Moving toward the bottom-left vertex decreases L_0/L_{total} in favor of L_1/L_{total} (the 1.5B share); the 500M sub-model then has fewer layers and must be *wider* to hit the same parameter count. The same logic applies to the other two axes. The gold star marks the configuration used in our main experiments, $(L_0, L_1, L_2) = (24, 10, 5)$.

The dashed region marked "unfeasible" contains depth triplets for which no valid Matryoshka architecture exists at the prescribed parameter targets. Concretely, given the head dimensions $(d_h^{(0)}, d_h^{(1)}, d_h^{(2)}) = (64, 96, 128)$ and target sizes (500M, 1.5B, 3B), we solve for the head count of each sub-model so that the cumulative parameter count matches the target. Two constraints can fail. Solving for the head count of a sub-model with too few layers can require fewer than one head, which is not realizable. More commonly, the nesting requirement $D_0 \leq D_1 \leq D_2$ on the hidden dimensions can be violated: when a larger sub-model is assigned too many layers, the only way to hit its parameter target is for it to be *narrower* than its predecessor, which is incompatible with the constraint. In both situations, this will lead to a missing point on the ternary graph.

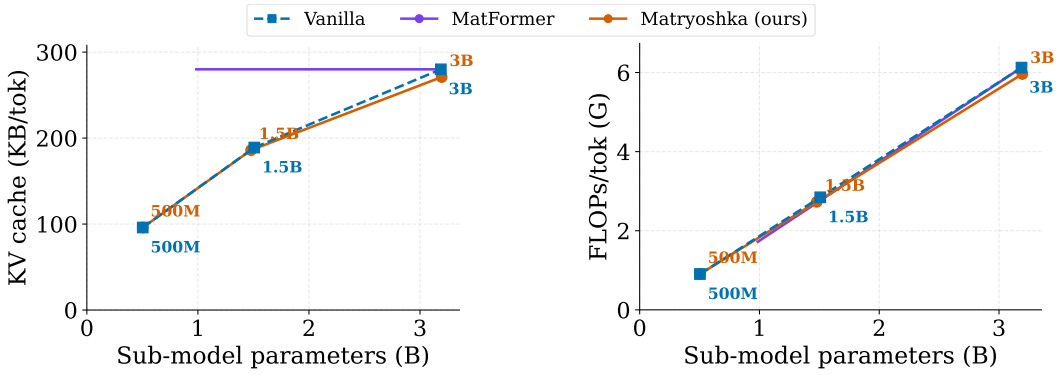
E Comparison with MatFormer

We compare Matryoshka to MatFormer (Devvrit et al., 2023) along two complementary axes: the inference-time resource footprint of the sub-models each framework can expose (Section E.1), and the language modeling quality of suites trained with each method (Section E.2).

E.1 Resource Footprint

We first compare both methods on the two resource axes that matter at deployment time: KV cache footprint and per-token inference FLOPs. MatFormer applies its nesting *only on the FFN block*, leaving attention shape, hidden width, and depth fixed at the universal-model values. Sub-models are extracted by truncating the FFN intermediate dimension to a fraction $r \in (0, 4]$ of the universal $4D$ width, with canonical granularities at $r \in \{0.5, 1, 2, 4\}$ in the original paper. MatFormer's Mix'n'Match procedure supports any positive ratio in principle, so we sweep r continuously below.

We fix the universal-model shape to the Vanilla 3B baseline (Table 1, $D = 2560$, $L = 28$) for both methods. For MatFormer we apply the standalone-Transformer formulas of Appendix B with FFN cost $3rD^2$ per layer (three SwiGLU projections (AI@Meta, 2024) of width rD). For Matryoshka we use the cumulative formulas at the three exit points (500M, 1.5B, 3B) of the 3B suite. Figure 10 reports the result.



(a) KV cache per token

(b) Theoretical inference FLOPs per token

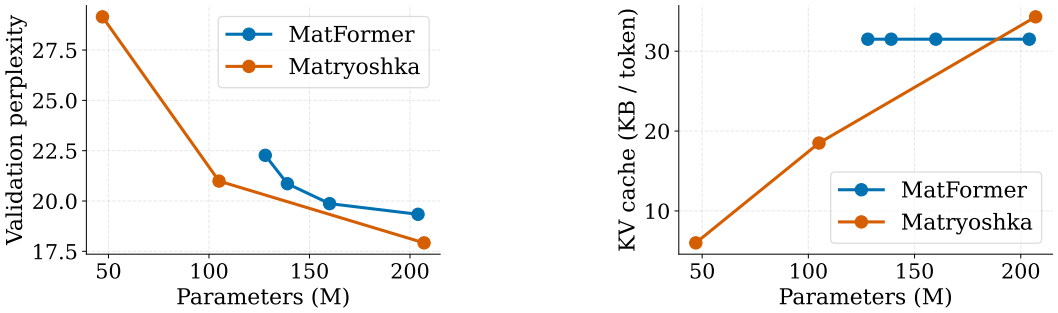
Figure 10: Resource cost of extracting a sub-model from each framework, with the universal shape fixed to the Vanilla 3B baseline ($D = 2560$, $L = 28$). The MatFormer curve is a continuous sweep of FFN ratio $r \in (0, 4]$. Matryoshka and Vanilla are the three actual sub-models / independent baselines from Table 1. Lower-left is better on both panels.

Smallest Extractable Model Floor The MatFormer trace in Figure 10 starts at roughly 1.26B parameters even at the smallest canonical granularity $r = 0.5$, and a hypothetical $r \to 0$ would still leave a $\sim$1.05B floor coming from the input embedding, the LM head, and the four attention projections at every layer, which are all kept at the universal width. This means a 3B universal MatFormer cannot expose a sub-500M extractable model at all, in either parameter count or FLOPs/tok: Figure 10b shows that MatFormer bottoms out at 2.27 GFLOPs/tok, whereas Matryoshka and Vanilla both reach 0.91 GFLOPs/tok at the 500M exit. By contrast, Matryoshka places a 500M sub-model at the cumulative exit point of the smallest block, with parameter count, FLOPs, and KV cache all scaling down from there.

Matryoshka and KV Cache On the FLOPs axis the three methods collapse onto essentially the same curve as a function of parameters, since FLOPs/tok is dominated by the same matrix-multiplication budget regardless of how the parameters are distributed across attention and FFN. KV cache, however, separates the methods cleanly. Figure 10a shows that MatFormer's KV cache is a horizontal line at 280 KB/tok regardless of which sub-model is extracted, because attention shape and depth are fixed at the universal-model values. A deployment that needs a smaller KV footprint, e.g. to fit longer contexts or larger batches into the same memory budget, gains nothing from MatFormer's elasticity. Matryoshka instead reduces KV cache from 280 KB to 96 KB at the 500M exit, identical to the 96 KB of an independently trained Vanilla 500M model, which shares its shape. Matryoshka thus matches the Vanilla curve on both axes across the full size range, confirming that the nested structure does not introduce inference-time overhead relative to a conventional suite, while delivering all sub-models from a single training run.

E.2 Language Modeling Performance

We compare our approach against MatFormer (Devvrit et al., 2023) at the 200M scale, training on 10B tokens, where both methods extract a suite of nested sub-models from a single training run (Figure 11). Because MatFormer nests along the FFN width while sharing a single attention backbone, all of its sub-models carry the same KV-cache footprint (31.5 KB/token), independent of their parameter count. In contrast, Matryoshka nests models along both width and depth, so the KV cache shrinks together with the sub-model (down to 6.0 KB/token), yielding a genuinely lighter memory profile at inference time. We also observe a better quality–size trade-off: at matched validation perplexity ($\sim$21), Matryoshka-100M matches MatFormer-M (139M) with $\sim 40\%$ less KV cache and fewer parameters, and Matryoshka-200M attains the lowest perplexity (17.92) of all compared models, including MatFormer-XL (19.34).



(a) Validation perplexity

(b) KV cache per token

Figure 11: MatFormer vs. Matryoshka suites at 200M scale, as a function of sub-model size. Matryoshka achieves a better size–performance frontier. MatFormer's shared attention backbone fixes its KV cache at 31.5 KB/token across all sub-models, whereas Matryoshka's KV cache scales with the sub-model.

F Out-of-Distribution Perplexity

To probe how well each suite generalizes outside FineWeb-Edu, we evaluate byte perplexity on five held-out corpora that span different domains: WikiText-103 (encyclopedic), C4 (general web), PG-19 (long-form fiction), arXiv (scientific full-text), and PubMed Central (biomedical full-text). All runs use a 2 048-token rolling window with stride 1 024 and the same SmolLM2 tokenizer. The Vanilla (FLOPs-matched) and Vanilla (Token-matched) baselines mirror those in Table 2. We report byte perplexity because it is invariant to tokenization differences and robust to long-tail vocabulary in PG-19, arXiv and PubMed.

Table 4 reports the per-corpus byte perplexity. The advantage is largest on arXiv, where the 3B Matryoshka sub-model lowers byte PPL by 0.13 over the Token-matched Vanilla baseline; the 1.5B and 500M Matryoshka sub-models inherit part of this advantage through the shared early layers.

G Closing the Per-Size Gap with Loss Weighting

The default Matryoshka recipe weights each sub-model's loss equally in the total loss. This is a reasonable default but not necessarily optimal: the 50M, 100M, and 200M sub-models do not contribute equally to the gradient, and a uniform weighting over-serves some sizes at the expense of others. We probe whether the residual gap to Vanilla perplexity (Figure 4a) can be closed by reweighting the per-sub-model losses, using the 200M proxy suite.

We sweep nine weight vectors $(w_{50}, w_{100}, w_{200})$ chosen to redistribute budget from sizes that are already at or below Vanilla toward those that remain above it; all vectors are normalized to sum to one. We re-run the last 1,000 cooldown steps of the 200M Matryoshka run with each weighting, keeping the rest of the recipe identical. The reference is the uniform weighting $(1/3, 1/3, 1/3)$, which corresponds to the main recipe.

Model	WikiText	C4	PG-19	arXiv	PubMed	**Average**
500M parameters						
Vanilla (FLOPs-matched)	2.244	2.023	2.582	2.551	2.075	2.295
Vanilla (Token-matched)	**2.215**	**2.003**	**2.504**	2.530	**2.065**	**2.263**
Matryoshka	2.221	2.010	2.509	**2.497**	2.090	2.265
1.5B parameters						
Vanilla (FLOPs-matched)	2.138	1.953	2.361	2.454	1.951	2.171
Vanilla (Token-matched)	2.107	1.934	2.283	2.447	**1.925**	2.139
Matryoshka	**2.100**	**1.931**	**2.228**	**2.398**	1.946	**2.121**
3B parameters						
Vanilla (FLOPs-matched)	2.097	1.924	2.206	2.423	2.066	2.143
Vanilla (Token-matched)	**2.059**	**1.906**	2.129	2.442	1.950	2.097
Matryoshka	2.064	**1.906**	**2.120**	**2.315**	**1.931**	**2.067**

Table 4: Byte perplexity on five out-of-distribution corpora. Vanilla (FLOPs-matched) and Vanilla (Token-matched) are defined as in Table 2. Lower is better; bold marks the best within each size group. Sample sizes are 1 000 (WikiText-103), 200 (C4 and PubMed Central), 40 (arXiv), and 20 (PG-19), all evaluated with a 2 048-token rolling window.

Figure 12 reports the average positive PPL excess of the weighted runs over Vanilla, defined as $\mathbb{E}_{\text{size}}\left(\max(0,\ \text{PPL}_{\text{Mat.}}(\text{size}) - \text{PPL}_{\text{Van.}}(\text{size}))\right)$. This metric only penalizes sub-models that remain above Vanilla, so it captures *how much negative gap remains* rather than rewarding improvements for already-strong sub-models. Several weighted variants improve over uniform on this aggregate score, with the best configuration cutting the residual excess by roughly 15%.

Figure 13 disaggregates the same runs by sub-model size. Each panel plots the per-size ΔPPL over Vanilla as a function of (w_{100}, w_{200}) (with $w_{50} = 1 - w_{100} - w_{200}$ implicit). Two patterns are visible. First, under the uniform weighting (the star) the 100M sub-model is already substantially below Vanilla while the 200M sub-model is the bottleneck — equal weights over-serve 100M. Second, weightings that redistribute budget from w_{100} toward w_{200} (and mildly toward w_{50}) close the 200M gap and lift the 50M sub-model above Vanilla without breaking the 100M sub-model. No single weighting is Pareto-optimal on every axis, but several variants strictly dominate uniform on the aggregate score and on the bottleneck sub-model.

We did not extend this sweep to the 3B suite, because its compute cost is prohibitive and the qualitative finding is already useful at 200M. The takeaway is that part of the residual per-size gap to Vanilla in the main results is attributable to the uniform-weighting choice rather than to the nested architecture. This suggests that a simple, near-zero-cost hyperparameter tuning of (w_m) would tighten the per-size results further; we leave a systematic per-size loss-weighting rule, including adaptive schedules, to future work.

H Additional Size-and-Shape Ablation Views

Figure 6 reports three views of the 200M size-and-shape sweep: total KV cache per token, mean ΔPPL, and mean benchmark accuracy gap relative to a Vanilla baseline. In Figure 14, we report three additional views: the mean ΔNLL counterpart of the perplexity scatter (Figure 14(a)), the per-token theoretical-FLOPs scatter (Figure 14(b)), and a 1D summary of mean NLL gap as a function of total KV cache (Figure 14(c)). These views support the same conclusion as the main figure: the Matryoshka design space contains variants that simultaneously reduce KV cache and FLOPs relative to the Vanilla baseline without harming downstream quality.

As in Figure 6, each point is one Matryoshka variant with a shared 50M base sub-model, and Δ values are averaged over the three sub-model sizes (50M / 100M / 200M) versus the Vanilla 200M baseline of matching architecture.

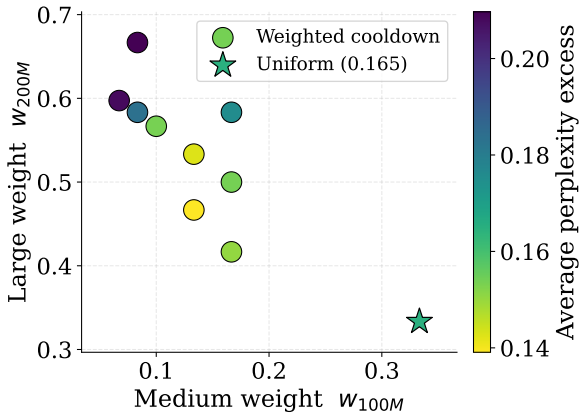
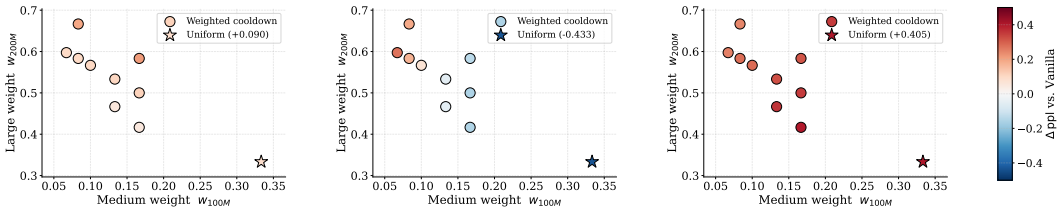


Figure 12: Average PPL excess over Vanilla (lower is better) on the 200M proxy suite for nine loss-weight configurations. Axes are the normalized weights of the medium (100M) and large (200M) sub-models; w_{50} is the residual. The star marks the uniform weighting $(1/3, 1/3, 1/3)$, which is the main-recipe choice. Several weighted variants reduce the residual excess relative to uniform.



(a) 50M (b) 100M (c) 200M

Figure 13: Per-size ΔPPL over Vanilla on the 200M proxy suite, as a function of the normalized weight vector (w_{100}, w_{200}). Stars mark the uniform weighting. Blue points beat Vanilla; red points lag it. Under uniform weights the 100M sub-model is already below Vanilla while the 200M sub-model lags; redistributing weight from w_{100} toward w_{200} closes the 200M gap without breaking 100M.

I 200M Proxy Suite Benchmarks

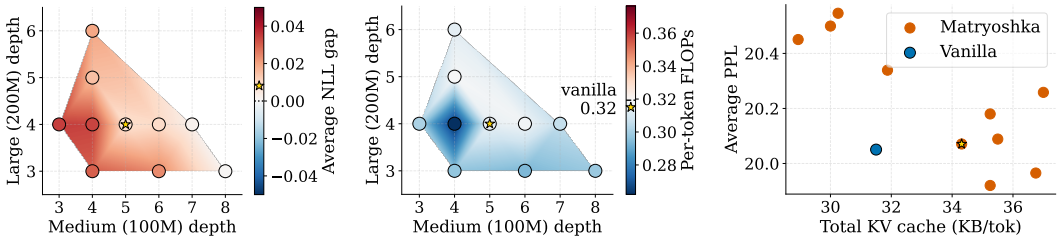
To complement the 3B-scale results, we evaluate the 200M proxy suite used throughout our ablations on the same downstream benchmarks as Table 2. The Vanilla baseline is the three independently trained 50M, 100M, and 200M models that share the recipe and tokenizer of the Matryoshka run; "Matryoshka" refers to the chosen configuration (Figure 6). Both suites are trained for 20B tokens.

In Table 5, the Matryoshka suite ties Vanilla on the 50M sub-model (-0.1 avg) and beats it at 100M and 200M ($+0.4$ and $+0.1$ avg), confirming the perplexity parity reported in Figure 14 on downstream tasks.

J Supervised Fine-Tuning

To assess whether the benefits of Matryoshka training carry over to post-training, we perform supervised fine-tuning (SFT) of the 3B suites on Tulu-3 data (Lambert et al., 2024) for 2 epochs, using a learning rate of 10^{-4}, which is common for small models (Allal et al., 2025). We compare three settings:

- **Vanilla**: each model of the Vanilla suite is fine-tuned independently.
- **Matryoshka**: the suite is fine-tuned as a whole with exactly our pretraining objective (Equation (3), summed over sub-models), with cross-entropy on the Tulu-3 targets. We add no new loss term.



(a) Mean ΔNLL scatter (b) Per-token FLOPs (G) (c) Mean NLL gap vs. KV cache

Figure 14: Additional views of the 200M size-and-shape ablation. **(a)** Same axes as Figure 6, colored by mean ΔNLL instead of ΔPPL. **(b)** Per-token theoretical FLOPs of the full Matryoshka forward pass, with the Vanilla 200M baseline marked on the colorbar. **(c)** 1D summary plotting mean NLL gap (averaged over sub-model sizes) against total KV cache.

Model	ARC-E	ARC-C	HS	LAMBADA	OBQA	PIQA	WG	**Avg acc. ↑**
50M parameters								
Vanilla	**44.2**	23.6	**28.5**	**17.5**	**29.4**	**60.1**	50.2	**36.2**
Matryoshka	43.1	**24.6**	27.9	16.8	29.0	59.6	**51.7**	36.1
100M parameters								
Vanilla	**47.4**	25.4	**32.3**	24.1	31.0	62.1	51.5	39.1
Matryoshka	47.2	**25.5**	32.1	**24.9**	**32.0**	**62.7**	**52.0**	**39.5**
200M parameters								
Vanilla	**51.8**	**27.7**	**36.7**	29.4	**33.2**	64.5	50.8	42.0
Matryoshka	51.6	26.7	36.2	**30.0**	32.2	**65.9**	**52.1**	**42.1**

Table 5: Per-benchmark zero-shot accuracy on the 200M proxy suite (50M / 100M / 200M sub-models). Vanilla is three independently trained models; Matryoshka is the configuration used as the reference point throughout our ablations. Both suites are trained for 20B tokens. Avg acc. averages the seven benchmarks (WG = Winogrande). Bold marks the better entry per row group.

- **Matryoshka (detached)**: each sub-model is fine-tuned as a standalone model. Nesting is removed: the 1.5B model has no 500M exit, and the 3B model has no 500M or 1.5B exits. This variant retains the pretraining compute savings but relaxes the nesting constraint, which corresponds to a case where only final model performance matters.

We report the validation perplexity on Tulu-3, the average accuracy on the seven benchmarks of Table 2 (Avg. MCQA), TruthfulQA (Lin et al., 2022) (MC2), and IFEval (Zhou et al., 2023) (prompt-level strict accuracy).

Table 6 shows that Vanilla and Matryoshka remain at parity overall after SFT, confirming that Matryoshka training is compatible with instruction tuning. The detached variant is also competitive: it improves SFT perplexity over joint Matryoshka fine-tuning at every size, and yields IFEval gains at 1.5B and 3B, while still using $\sim$36% less pretraining compute than the Vanilla suite.

K Combining Matryoshka Suites with EAGLE

EAGLE (Li et al., 2024a) is a draft-acceleration technique that is orthogonal to our approach. To illustrate how the two can be combined, we train an EAGLE-1 head on top of our Matryoshka-3B model, and use the 500M sub-model as a pre-verifier that trims EAGLE drafts and avoids verifying unlikely candidates with the rest of the model stack. We report throughput across generation batch sizes on an NVIDIA GH200 in Table 7.

EAGLE alone brings the largest speedups at small batch sizes (up to $1.91\times$ at batch 16), but its benefits vanish as the batch grows and inference becomes compute-bound, down to $0.98\times$ at batch

Method	Size	Tulu-3 Val PPL ↓	Avg. MCQA ↑	TruthfulQA ↑	IFEval ↑
Vanilla	500M	**3.28**	45.0	36.3	17.0
Matryoshka	500M	3.41	45.3	**40.1**	**18.1**
Matryoshka (detached)	500M	3.33	**45.9**	38.8	17.4
Vanilla	1.5B	2.90	**52.5**	37.7	22.2
Matryoshka	1.5B	2.89	51.1	40.1	23.1
Matryoshka (detached)	1.5B	**2.83**	50.8	**40.3**	**23.5**
Vanilla	3B	2.73	**52.9**	38.6	22.4
Matryoshka	3B	2.75	52.2	39.1	21.8
Matryoshka (detached)	3B	**2.68**	52.4	**40.5**	**26.1**

Table 6: Downstream results after SFT on Tulu-3. Vanilla and Matryoshka remain at parity overall. The detached variant, which removes the nesting constraint during SFT only, yields competitive models while still using $\sim$36% less pretraining compute.

256. Using the 500M sub-model as a pre-verifier slightly reduces throughput at small batch sizes, but improves it from batch 128 onwards, and retains a $1.13\times$ speedup at batch 256 where EAGLE alone is slower than normal decoding. Matryoshka suites and feature-level draft heads thus compose, with the nested sub-models being most useful in the compute-bound regime.

	Normal	EAGLE		EAGLE + Matryoshka filter	
Batch	tok/s	tok/s	Speedup	tok/s	Speedup
1	28.1	**42.1**	**1.50**×	38.7	1.38×
16	411.4	**784.1**	**1.91**×	748.0	1.82×
128	2641.0	4202.1	1.59×	**4393.7**	**1.66**×
160	3728.6	5066.2	1.36×	**5331.6**	**1.43**×
256	6498.5	6344.3	0.98×	**7317.3**	**1.13**×

Table 7: Throughput of Matryoshka-3B with an EAGLE-1 head, with and without the 500M sub-model as a pre-verifier for EAGLE drafts. Speedups are relative to normal decoding. The pre-verifier improves throughput at larger batch sizes, where inference is compute-bound.

L Speculative Decoding: Implementation and Matched-Batch Throughput

Implementation Our procedure is standard speculative decoding (Leviathan et al., 2023). The smallest sub-model (e.g. 500M) generates a draft of γ tokens and emits the corresponding probabilities. The stack of remaining sub-models (e.g. the blocks that make up the rest of the 3B model) takes the output states of the draft model as input, and computes its own probabilities for the drafted positions in a single forward pass. These probabilities are used to accept $k \leq \gamma + 1$ tokens, using argmax matching for greedy decoding and the usual rejection rule for nucleus sampling. Rollback truncates the KV cache of rejected draft tokens, the verifier's bonus token is appended, and batching is standard static batching. The only Matryoshka-specific point is structural: as the draft model is nested in the verifier, its layers and KV cache are reused by the verifier, so that drafting and verification activate 3B parameters in total, instead of 3.5B for a Vanilla 500M / 3B pair.

Matched-Batch Throughput The results of Section 3.2.3 compare each pair at its maximum feasible batch size, which is larger for Matryoshka thanks to KV cache sharing. To disentangle this effect, we also compare both pairs at matched batch sizes in Table 8 (NVIDIA GH200, draft length 4, nucleus sampling with $p = 0.9$ and $T = 0.4$). Matryoshka is faster at every matched batch size (1.09–$1.18\times$, $\sim 13\%$ on average). With the batch size fixed, this gain only comes from fewer activated parameters per step and from higher acceptance: over 100 documents, the Matryoshka pair achieves 4.4% (greedy) and 5.0% (nucleus) longer accepted lengths on average, and wins on 8/10 and 9/10 draft lengths, respectively. On top of this, KV cache sharing lets Matryoshka reach a batch size of 160, while the Vanilla pair runs out of memory from 148, so that the end-to-end gain at each

pair's best feasible batch size reaches $1.32\times$ (4,327 vs. 3,277 tok/s). We keep this matched-memory setting as our primary benchmark, since the larger batch is a direct benefit of KV cache sharing.

Batch	Matryoshka (tok/s)	Vanilla (tok/s)	Ratio
1	58 ± 7	49 ± 7	$1.18\times$
16	856 ± 96	763 ± 109	$1.12\times$
64	2892 ± 389	2503 ± 327	$1.16\times$
96	3521 ± 532	3107 ± 467	$1.13\times$
128	3563 ± 536	$\mathbf{3277} \pm 454$	$1.09\times$
136	3433 ± 440	3065 ± 400	$1.12\times$
148	3926 ± 218	OOM	—
160	$\mathbf{4327} \pm 74$	OOM	—
172	OOM	OOM	—

Table 8: Speculative decoding throughput of the 500M / 3B pairs at matched batch sizes (NVIDIA GH200, draft length 4, nucleus sampling with $p = 0.9$, $T = 0.4$; mean $\pm$ standard deviation). Bold marks each pair's best feasible throughput.

M Multi-Seed Runs at the 200M Scale

Our 3B runs are single-seed for cost reasons. To estimate run-to-run variance, we train both the Vanilla and Matryoshka 200M suites with 5 different seeds on multi-node NVIDIA GH200 GPUs, for roughly 30 GH200 hours per suite and seed, for 20B training tokens. Table 9 shows that run-to-run noise is small ($\sim 0.1\%$ of validation PPL, and at most 0.4 points of average accuracy), and that the Matryoshka suite remains on par with the Vanilla suite at every size (within 0.4 points of average accuracy and 1.4% of PPL) while using 27% less training compute on this hardware (Appendix B). This is expected, as pretraining runs are known to be consistent across seeds except for individual runs that suffer from instabilities (van der Wal et al., 2025). We note that we do not claim downstream performance gains, but rather an equivalent performance level for less training compute.

	Validation PPL ↓			Avg acc. ↑			
Model	50M	100M	200M	50M	100M	200M	Total compute
Vanilla	29.11 ± 0.05	21.56 ± 0.03	17.71 ± 0.03	36.5 ± 0.2	38.7 ± 0.2	41.3 ± 0.4	2.22×10^{19}
Matryoshka	29.17 ± 0.04	21.01 ± 0.02	17.95 ± 0.03	36.4 ± 0.2	39.1 ± 0.2	41.5 ± 0.2	1.62×10^{19} (-27%)

Table 9: Aggregated results for 200M-scale pretraining across 5 seeds (mean $\pm$ standard deviation). Avg acc. averages the seven benchmarks of Table 2.