Fully Nested Transformers

Avi Trost [1] **Alexander Yun** [1] **John Cooper** [1] **Gabriel Orlanski** [1] **Frederic Sala** [1]

Abstract

Matryoshka Representation Learning produces representations that can be truncated at different granularities to adapt to diverse downstream requirements, but computing them still requires a full forward pass through the model. Elastic architectures enable adaptive inference budgets, but their smaller models do not generally produce representations that are prefixes of those produced by the larger models. To unify these two forms of adaptivity within a single language model, we design StairFormer: a Transformer architecture that nests a hierarchy of prefix submodels by preserving prefix structure at each layer. Running a larger model produces, as prefixes, the same representations that would have been produced by independently running each smaller model. Prefix representations are efficiently computed using the computational paths of smaller submodels, which can then be progressively refined by activating larger portions of the model. This full nesting enables cascading inference and reuse of intermediate computations across model scales because submodels share the same forward pass. Empirically, we show that StairFormer maintains competitive language modeling performance at 1.13B parameters, achieving a CORE metric within an 11.1% relative gap of a standard Transformer baseline while satisfying these full nesting constraints.

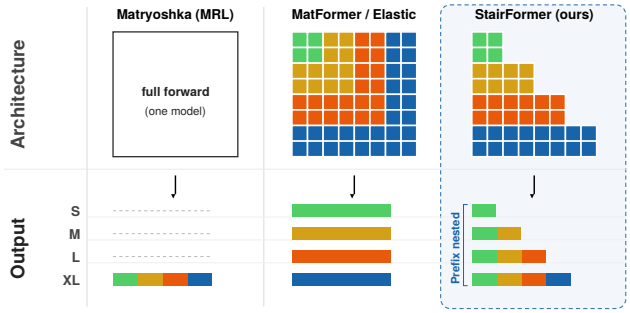

Figure 1. Matryoshka Representation Learning (left) hierarchically nests output representations, but not the model producing them. MatFormer (middle) nests models, but not their output representations. StairFormer (right) nests *both* output representations and the models producing them.

1. Introduction

Matryoshka representation learning (MRL) is a popular approach for training flexible representations, i.e., coarse-to-fine embeddings (Kusupati et al., 2024). While this technique produces representations that can be used flexibly at inference time, the cost of obtaining these representations is nevertheless constant. Running a model to produce coarse-grained representations is just as expensive as using it for fine-grained representations. A natural question is whether inference for coarser representations can be made cheaper than inference for refined representations.

In contrast, *elastic* architectures such as MatFormer (Devvrit et al., 2024) have been proposed that can be trained once and deployed at different inference budgets. While promising, the representations produced by this approach lack the downstream flexibility of Matryoshka-style representations. This highlights a tension between MRL and elastic architectures: models can adapt to either inference or downstream budgets, but not both.

We study this tradeoff, seeking to understand whether there is a way to sidestep it. Our study starts from the architectural viewpoint. In a typical Transformer-based model, components mix information nonlinearly, preventing nested models from being fully nested, which is why existing methods are unable to achieve the best of both worlds. This incompatibility makes full nesting difficult to obtain by directly modifying standard architectures. To tackle this issue, we identify sufficient conditions that enable full nestedness, and design a Transformer-based architecture that achieves this.

[1]Department of Computer Sciences, University of Wisconsin–Madison, Madison, WI, USA. Correspondence to: Avi Trost <atrost@cs.wisc.edu>.

Proceedings of the 43^{rd} International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026.

We propose **StairFormer**[1], a Transformer-based architecture with both inference and downstream flexibility. StairFormer maintains a nested hierarchy of submodels that share intermediate computations. By making the model itself nested, StairFormer unifies the flexibility of Matryoshka representations with the efficiency of elastic inference. Coarse predictions can be produced by cheaper submodels, while later blocks refine these computations without changing the earlier outputs. This property has several possible use cases, including efficient cascading inference where a query's KV cache can be reused between submodels.

The key architectural challenge is that standard Transformer components do not preserve prefix subspaces: dense linear maps, attention heads, and normalization statistics can all allow later coordinates to influence earlier ones. Structured matrices emerge as a natural way to approach this problem, given their flexibility and efficiency (Qiu et al., 2024). We show that block lower triangular linear maps give a simple sufficient condition for preserving prefix submodels, and we extend this condition to Transformers by aligning attention heads with block boundaries and replacing normalization layers with a prefix-preserving variant. Together, these modifications yield an architecture whose layers preserve the prefix filtration, so the full model is output-nested by construction.

We make the following contributions:

1. We formalize full nesting for Transformer families as prefix preservation across layers, giving conditions under which the output of a smaller model is exactly the prefix of the output of a larger model.

2. We design StairFormer, a fully nested Transformer architecture, and empirically demonstrate competitive language modeling performance at the 1.13B-parameter scale, achieving a CORE metric within an 11.1% relative gap of a standard Transformer baseline.

3. We show that StairFormer's full nesting property can be used for cascading inference with computation reuse across model scales.

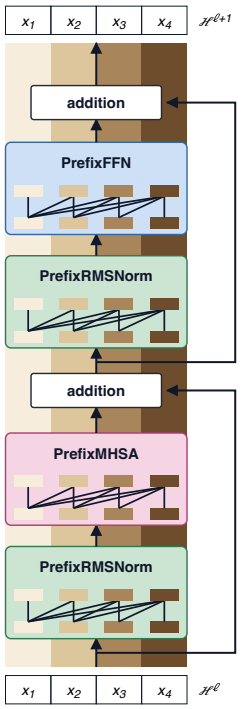

Figure 2. StairFormer architecture. Each Transformer layer preserves the prefix filtration by using PrefixRMSNorm, head-aligned PrefixMHSA, PrefixFFN, and residual additions that do not allow later coordinate blocks to influence earlier ones.

2. Fully Nested Transformers

2.1. Preliminaries

We begin with some definitions of useful properties for nested models. These lay the groundwork for what is desired in nested Transformers. We will then describe an architecture which achieves these properties.

[1]The block lower triangular structure of StairFormer's weights resembles a staircase.

Let $M_\alpha : \mathcal{X} \to \mathcal{H}_\alpha$ denote the model of budget α (a value representing the size of the hidden state), mapping an input to its final hidden representation. This model can be decomposed layer-wise as $M_\alpha = T_\alpha^{L-1} \circ \cdots \circ T_\alpha^0$, with each layer T_α^ℓ mapping from $\mathcal{H}_\alpha^\ell$ to $\mathcal{H}_\alpha^{\ell+1}$ with $\mathcal{H}_\alpha = \mathcal{H}_\alpha^L$ and $\mathcal{X} = \mathcal{H}_\alpha^0$. Fix a sequence $\mathcal{A} = (\alpha_i)_{i=1}^m$ representing the budgets of the nested models. Their final hidden spaces are nested as a *filtration*:

$$\mathcal{H}_{\alpha_1} \subset \mathcal{H}_{\alpha_2} \subset \cdots \subset \mathcal{H}_{\alpha_m}.$$

Additionally, we are equipped with projection maps $\pi_{\beta\to\alpha} : \mathcal{H}_\beta \to \mathcal{H}_\alpha$ for $\alpha \le \beta$.

Definition 2.1. *The family* $M = \{M_\alpha\}_{\alpha\in\mathcal{A}}$ *is* ***output nested*** *if, for every* $\alpha \le \beta$,

$$M_\alpha(x) = \pi_{\beta\to\alpha}\left(M_\beta(x)\right) \quad \forall x \in \mathcal{X}.$$

In plain words, an output nested family M implies that a single forward pass of a model M_β includes the forward pass of every smaller model M_α in the family.

We will primarily be interested in a modification of output nesting, where the hidden state spaces for a specific layer

Table 1. Summary of architectural modifications beyond block triangular weights needed to make StairFormer fully nested.

COMPONENT	ISSUE	FIX
MULTI-HEAD SELF-ATTENTION	NON-LINEAR MIXING OF INFORMATION	HEAD-ALIGNED BLOCKS
RMSNORM	LATER-COORDINATE STATISTICS AFFECT ALL COORDINATES	PrefixRMSNorm

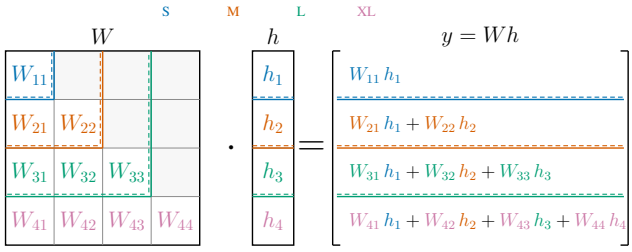

Figure 3. Block lower triangular linear layers preserve prefix submodels. The first k output blocks depend only on the first k input blocks, so each leading principal block submatrix defines a complete smaller model.

ℓ across the family M are also a simultaneous filtration. Specifically, we are equipped with projection maps $\pi^{\ell}_{\beta\to\alpha} : \mathcal{H}^{\ell}_{\beta} \to \mathcal{H}^{\ell}_{\alpha}$. For consistency, $\pi^{L}_{\beta\to\alpha} = \pi_{\beta\to\alpha}$ from the filtration for output nesting, and $\pi^{0}_{\beta\to\alpha} = \mathrm{id}_{\mathcal{X}}$, so all submodels observe the same input embeddings.

Definition 2.2. *We say the family $T^{\ell} = \{T^{\ell}_{\alpha}\}_{\alpha\in\mathcal{A}}$ is **filtration-preserving** if, for every $\alpha \le \beta$,*

$$\pi^{\ell+1}_{\beta\to\alpha} \circ T^{\ell}_{\beta} = T^{\ell}_{\alpha} \circ \pi^{\ell}_{\beta\to\alpha}.$$

In other words, projecting into the smaller hidden state after the larger layer produces the same output as projecting before the smaller layer.

A consequence of this definition is that the composition of filtration-preserving layers is filtration-preserving. It follows that a model with filtration-preserving layers is output nested.

Proposition 2.3. *If T^{ℓ} is filtration-preserving for $\ell < L$, then M is output-nested.*

Proof. See Appendix C. □

Benefits of these properties. This property enables us to extract submodels of varying sizes out of a single trained model. Unlike prior architectures, the extracted submodels are simple sub-architectures within the larger model. With a single forward pass, we obtain the forward passes of each submodel simultaneously.

A challenge in realizing this structure for modern Transformer-based LLMs is that layers are highly nonlinear. In particular, the self-attention mechanism contains a variety of components, all of which need to be carefully considered to verify filtration preservation. In situations where the components are not already filtration-preserving, additional constraints are needed to enforce this desired property.

2.2. Handling Specific Layer Types

To design a Transformer with the required parameter structure, we investigate each component of these models separately, detailing the modifications necessary to have filtration-preservation for all subcomponents.

We will fix $\mathcal{H}_{\alpha} = \mathbb{R}^{\alpha}$ and π such that

$$\pi_{\beta\to\alpha}(x_{1:\beta}) = x_{1:\alpha},$$

i.e. the projection outputting the first α coordinates of its input. In the case where the input space is matrices $X \in \mathbb{R}^{\alpha\times L}$ for some context length L, the induced π projects each column separately.

This choice of π has the nice additional property that it respects all element-wise operations, such as vector addition and element-wise multiplication. This means that a residual connection of the form $T(x) = x + f(x)$ is filtration-preserving if and only if $f(x)$ is.

Linear layers. We first consider the simplest and most fundamental setting: a single linear layer. This case provides the basic structural constraint that we will later impose throughout the Transformer. This is central to our design: informally, **block lower triangular weights are filtration-preserving**.

Consider the linear layer $T(x) = Wx$. In a standard dense parameterization, any output coordinate may depend on any input coordinate. This violates filtration preservation: the first k output blocks of the large model may depend on input blocks $x_{k+1}, \dots, x_m$, which are unavailable to the smaller model. Therefore, simply truncating the output of a dense linear layer generally does not recover the output of the corresponding smaller layer.

A sufficient way to prevent this leakage is to **restrict W to the class of block lower triangular matrices**:

$$W = \begin{bmatrix} W_{11} & 0 & \cdots & 0 \\ W_{21} & W_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ W_{m1} & W_{m2} & \cdots & W_{mm} \end{bmatrix}.$$

Then the i-th output block is

$$T(x)_i = \sum_{j \le i} W_{ij} x_j.$$

Let $\alpha_0 = 0$. Then block i has width $b_i = \alpha_i - \alpha_{i-1}$, so $W_{ij} \in \mathbb{R}^{b_i \times b_j}$ and $x_i \in \mathbb{R}^{b_i}$, ensuring the first k output blocks depend only on the first k input blocks. This property enables extracted smaller models to be materially cheaper than the larger model.

For any budget α_k, define the corresponding submodel layer $W^{(k)}$ as the leading principal block submatrix

$$W^{(k)} = \begin{bmatrix} W_{11} & 0 & \cdots & 0 \\ W_{21} & W_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ W_{k1} & W_{k2} & \cdots & W_{kk} \end{bmatrix}.$$

Proposition 2.4. *The family of linear layers $\{W^{(k)}\}_{k=1}^{m}$ is filtration-preserving.*

Proof. See Appendix C. □

Earlier coordinate blocks form a complete smaller model, while later coordinate blocks add refinements that may depend on the earlier representation. Computation can therefore proceed block by block: evaluating the first k blocks produces the same result as running the α_k-budget submodel, and continuing to later blocks extends this computation to a larger model, without affecting the smaller output.

Self-attention. While imposing a block lower-triangular weight structure is sufficient to make linear layers filtration-preserving, this restriction is not enough to make an entire Transformer block satisfy this property, see MHSA:

$$\mathrm{head}_i(X) = \mathrm{softmax}\Big(Q_i K_i^\top / \sqrt{d_h}\Big) V_i.$$

Here, softmax mixes information from later coordinates into earlier coordinates, so self-attention is generally not a filtration-preserving operation. However, a key observation is that the dependencies are contained within heads, so as long as block boundaries coincide with head boundaries, information cannot flow from later blocks into earlier blocks.[2] Each submodel will activate a different number of heads.

The Q, K, and V projections are block lower triangular with respect to this partition, so every active head depends only on the active prefix of the input representation. Additionally, the output projection W^O is also block lower triangular, so the first k output blocks depend only on the first k head outputs. Smaller models only need to activate a subset of the heads, and therefore can maintain a smaller KV cache. When refining computations to a larger model, the KV cache from smaller models can be directly reused without recomputation.

Thus, for any budget α_k, running self-attention on the prefix representation gives exactly the prefix of the full self-attention output:

$$\pi_{\beta \to \alpha_k} \mathrm{MHSA}_\beta(X) = \mathrm{MHSA}_{\alpha_k}\left(\pi_{\beta \to \alpha_k} X\right).$$

Therefore, **head-aligned multi-head self-attention is filtration-preserving**.

RMSNorm. Standard $\mathrm{RMSNorm}$(Zhang & Sennrich, 2019) normalizes a vector coordinate-wise, but its normalization factor is computed from the vector as a whole. Given $x \in \mathbb{R}^n$, $\mathrm{RMSNorm}$ is defined as

$$\overline{x}_i = \frac{x_i}{\mathrm{RMS}(x)} \odot g_i, \qquad \mathrm{RMS}(x) = \sqrt{\frac{1}{n}\|x\|_2^2}.$$

The RMS factor is problematic in our setting because it can leak information from later blocks into earlier ones. To avoid this, we define $\mathrm{PrefixRMSNorm}$, a variant of $\mathrm{RMSNorm}$ whose normalization statistic depends only on prefix coordinate blocks. This removes the need for recomputation when increasing model capacity.

Let $x = [x_1; \ldots; x_m]$, where x_k denotes the k-th coordinate block. For coordinates in block k, $\mathrm{PrefixRMSNorm}$ is defined as

$$\overline{x}_k = \frac{x_k}{\mathrm{PrefixRMS}_k(x)} \odot g_k,$$

$$\mathrm{PrefixRMS}_k(x) = \sqrt{\frac{1}{n_{\le k}} \sum_{j=1}^{k} \|x_j\|_2^2}$$

and $n_{\le k}$ denotes the number of coordinates in the first k blocks of x. Acting coordinate-wise and without leaking information from later blocks into earlier ones, **this change makes the normalization layer filtration-preserving**.

2.3. StairFormer

To achieve our goal of designing a fully nested architecture, we introduce **StairFormer** as the architecture that incorporates these modifications. Put succinctly, StairFormer is an architecture consisting of:

1. Block lower triangular weight matrices, where block boundaries coincide with attention head boundaries.
2. $\mathrm{PrefixRMSNorm}$ over blocks instead of $\mathrm{RMSNorm}$.

Our main result effectively summarizes our work up to this point: **StairFormer is fully nested by design.** These modifications together suffice for transforming a Transformer block into one that is fully nested. Formally:

[2] Block boundaries must coincide with group boundaries instead of head boundaries for GQA.

Theorem 2.5. *All StairFormer layers are filtration-preserving, and therefore StairFormer is output-nested.*

Proof. See Appendix C. □

Implications of Theorem 2.5. Such an architecture may enable several possible benefits, including: speculative decoding, cascading model routing, simultaneous training across scales, proxy models that update alongside a target model during post-training, and dynamic model routing. In this work, we focus on the cascading router use case.

3. Experiments

We claim that StairFormer, as a Transformer-based architecture, can achieve strong language modeling performance. Additionally, we claim that the full nesting of submodels provides a language model with novel benefits. In this section, we empirically test these claims:

1. Can StairFormer achieve competitive language modeling performance compared to a standard Transformer-based architecture?
2. Can we leverage StairFormer's computational sharing between submodels to enable efficient cascading inference?

3.1. Setup

Training details. We pretrain a 1.13B StairFormer model with 4 submodels (including the largest model) on NVIDIA ClimbMix (Diao et al., 2025) for 3.8B tokens. After evaluating base model performance, we further fine-tune the model with SFT for 509M tokens. The SFT data mixture is a combination of SmolTalk, identity conversations, MMLU auxiliary training data, GSM8K, SimpleSpelling, and SpellingBee (Allal et al., 2025; Hendrycks et al., 2021; Cobbe et al., 2021; Karpathy, 2025). The training is conducted with a mix of H100, A100, and L40 GPUs, and is built on the Nanochat repository (Karpathy, 2025). The models are trained using a convex combination of the different submodels' losses, where the largest model has a weight of 0.9, and the remaining 0.1 weight is split between the three other submodels. This loss scaling is as a heuristic that appeared meaningful during prelimnary testing. We use a block-row-wise version of Muon that supports block triangular weight matrices. Additional training details are included in Appendix B.

Baseline models. We additionally train three baseline families: a family of standard Transformers, and two nested MatFormer models (Devvrit et al., 2024). One MatFormer model nests FFN hidden dimension only, and the other nests FFN hidden dimension and attention heads. For each set of baselines, we produce a model that closely mirrors the active parameter count and inference FLOPs of each budget of the StairFormer models (besides the FFN-only MatFormer baseline, which cannot extend to low parameter counts). All models are trained with the same amount of tokens as StairFormer. Additional baseline details are provided in Appendix B.

3.2. Language Modeling Performance

We evaluate each model's language modeling performance after pretraining, in addition to task performance after SFT. We evaluate the CORE metric from DataComp-LM (Li et al., 2025), which measures base model capabilities. After SFT, models are evaluated on GSM8K (Cobbe et al., 2021). Results are shown in Figure 4.

3.3. Cascading Inference

For cascading inference, a router observes the input prompt and the smaller model's response, then decides whether to accept that response or escalate based on text embeddings. We compare an oracle router, which defers exactly when the smaller model is incorrect, against lightweight MLP routers trained on frozen `all-MiniLM-L6-v2` sentence embeddings of the prompt–response pair from SentenceTransformers (Reimers & Gurevych, 2019). Accuracy–FLOP tradeoffs are reported on held-out GSM8K, relative to the largest standard baseline. Since StairFormer is able to reuse KV cache information between submodels, we count prefill computation as shareable across escalations. Decoding computation cannot generally be reused because models will sample different tokens during generation. Results are in Figure 5.

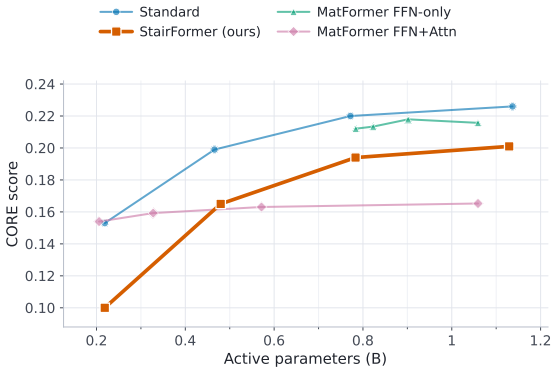

Figure 4. Pretrained model performance. CORE metric on base models plotted versus active parameters for each model in a family. StairFormer models can adapt to a wider dynamic range of inference budgets than MatFormer while still maintaining reasonable performance.

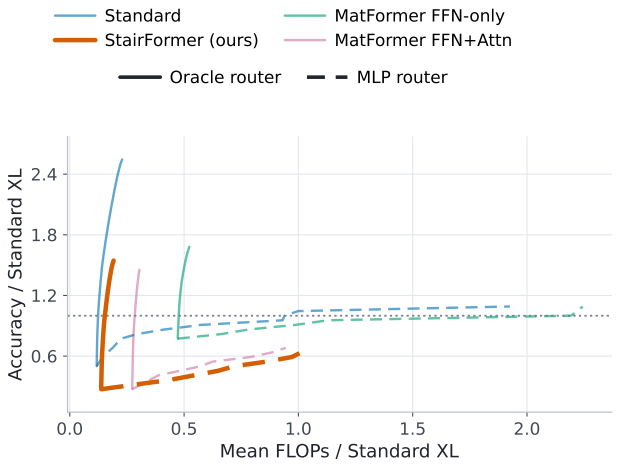

Figure 5. Cascading inference tradeoffs on GSM8K after SFT. Routers begin with the smallest model and selectively defer to larger models. The lines shown refer to Pareto-optimal router decisions for both the oracle and MLP routers. Accuracies are relative to the XL standard Transformer baseline.

4. Discussion

Language modeling performance. StairFormer's performance remained competitive, despite the degree of constraints imposed on it, while supporting a wider range of usable inference budgets than the MatFormer variants. The FFN-only MatFormer baseline retains stronger performance, but its elastic range is limited. Conversely, extending MatFormer-style slicing to both FFN and attention increases the budget range, but significantly weakens the larger extracted models. With more principled training methods tailored to StairFormer's structure, such as carefully designed loss functions or optimizers, we expect that StairFormer is capable of performing closer to standard architectures.

Cascading inference. The cascading inference experiments demonstrate a practical advantage of full nestedness. Under oracle routing, StairFormer achieves performance superior to the MatFormer baselines, despite having weaker language modeling performance. The gap between oracle routing and MLP routing suggests that to realize these efficiencies, more sophisticated routing methods should be used. StairFormer remains competitive while enabling prompt-side computation sharing across escalations. This highlights the potential benefits of computation sharing and reuse in StairFormer models, which would further improve with dedicated training recipes.

5. Related Work

Matryoshka Embeddings. Matryoshka Representation Learning (MRL) (Kusupati et al., 2024) is a paradigm that encourages the prefix of an embedding vector to operate as a coarse-grained embedding vector by itself after truncation. This strategy is the predominant strategy for training embedding models, due to its flexibility and low overhead.

Nested Models. Nested models can be trained once and deployed with an adaptive inference budget. A nested architecture closely related to MRL is MatFormer (Devvrit et al., 2024), which applies MRL-style truncation inside a Transformer's FFN or self-attention layers (Vaswani et al., 2023). MatFormer-style training has grown in popularity, with industrial-scale efforts implementing it (Team, 2025; Taghibakhshi et al., 2025). The Nested Subspace Property described in (Rauba & van der Schaar, 2026) details the exact nesting case for linear layers. The block triangular matrices used in our work can be thought of as similar to a special case of matrices satisfying the Nested Subspace Property, while also enabling the Transformer as a whole to become nested. The fMRLRec method for recommendation systems outlined in (Wang et al., 2024) utilizes block triangular matrices similarly to our work, which we build upon by laying out the theoretical conditions sufficient for exact nestedness, as well as extending the property to Transformer models. ThinkingViT enables computation reuse among nested submodels, but unlike in our work, its submodel outputs are not nested (Hojjat et al., 2026).

6. Limitations

Extending this design to other architectures will take careful consideration to ensure each component is filtration-preserving. Additionally, modern foundation model infrastructure is built around dense matrix multiplications. Even though block triangular matrix multiplications are GPU-friendly in principle, dense matrix multiplications are much more heavily optimized in modern systems.

7. Conclusion

We presented StairFormer, showing that it is possible to design a Transformer architecture that fully nests submodels within one another, while maintaining competitive performance. In addition to elastic inference, the output representation of larger models contains each smaller model's output as a prefix, maintaining a consistent Matryoshka representation, and enabling applications such as cascading inference. Finally, this architecture can extend beyond language models, and investigating the potential of StairFormer embedding models to settings where Matryoshka embeddings are currently deployed is a natural future direction.

Acknowledgements

We would like to thank Nicholas Roberts and Albert Ge for their discussions and feedback on the project. We are grateful for the support of the National Science Foundation (NSF) (CCF2106707), the Defense Advanced Research Projects Agency (DARPA Young Faculty Award), and the Wisconsin Alumni Research Foundation (WARF).

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Allal, L. B., Lozhkov, A., Bakouch, E., Blázquez, G. M., Penedo, G., Tunstall, L., Marafioti, A., Kydlíček, H., Lajarín, A. P., Srivastav, V., Lochner, J., Fahlgren, C., Nguyen, X.-S., Fourrier, C., Burtenshaw, B., Larcher, H., Zhao, H., Zakka, C., Morlon, M., Raffel, C., von Werra, L., and Wolf, T. Smollm2: When smol goes big – data-centric training of a small language model, 2025. URL https://arxiv.org/abs/2502.02737.

Bajpai, D. J. and Hanawal, M. K. A survey of early exit deep neural networks in nlp, 2025. URL https://arxiv.org/abs/2501.07670.

Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. Once-for-all: Train one network and specialize it for efficient deployment, 2020. URL https://arxiv.org/abs/1908.09791.

Cai, M., Yang, J., Gao, J., and Lee, Y. J. Matryoshka multimodal models, 2024. URL https://arxiv.org/abs/2405.17430.

Chen, Y., Pan, X., Li, Y., Ding, B., and Zhou, J. Ee-llm: Large-scale training and inference of early-exit large language models with 3d parallelism, 2024. URL https://arxiv.org/abs/2312.04916.

Chen, Z., Lu, X., Li, J., Chen, P., Li, Z., Sun, K., Luo, Y., Mao, Q., Li, M., Xiao, L., Yang, D., Huang, X., Ban, Y., Sun, H., and Yu, P. S. Harnessing multiple large language models: A survey on llm ensemble, 2026. URL https://arxiv.org/abs/2502.18036.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Dekoninck, J., Baader, M., and Vechev, M. A unified approach to routing and cascading for llms, 2025. URL https://arxiv.org/abs/2410.10347.

Devvrit, Kudugunta, S., Kusupati, A., Dettmers, T., Chen, K., Dhillon, I., Tsvetkov, Y., Hajishirzi, H., Kakade, S., Farhadi, A., and Jain, P. Matformer: Nested transformer for elastic inference, 2024. URL https://arxiv.org/abs/2310.07707.

Diao, S., Yang, Y., Fu, Y., Dong, X., Su, D., Kliegl, M., Chen, Z., Belcak, P., Suhara, Y., Yin, H., Patwary, M., Lin, C., Kautz, J., and Molchanov, P. Climb: Clustering-based iterative data mixture bootstrapping for language model pre-training. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2504.13161.

Elhoushi, M., Shrivastava, A., Liskovich, D., Hosmer, B., Wasti, B., Lai, L., Mahmoud, A., Acun, B., Agarwal, S., Roman, A., Aly, A., Chen, B., and Wu, C.-J. Layerskip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.681. URL http://dx.doi.org/10.18653/v1/2024.acl-long.681.

Haberer, J., Hojjat, A., and Landsiedel, O. Hydravit: Stacking heads for a scalable vit, 2024. URL https://arxiv.org/abs/2409.17978.

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding, 2021. URL https://arxiv.org/abs/2009.03300.

Hojjat, A., Haberer, J., Pirk, S., and Landsiedel, O. Thinkingvit: Matryoshka thinking vision transformer for elastic inference, 2026. URL https://arxiv.org/abs/2507.10800.

Hou, L., Huang, Z., Shang, L., Jiang, X., Chen, X., and Liu, Q. Dynabert: Dynamic bert with adaptive width and depth, 2020. URL https://arxiv.org/abs/2004.04037.

Hu, Q. J., Bieker, J., Li, X., Jiang, N., Keigwin, B., Ranganath, G., Keutzer, K., and Upadhyay, S. K. Routerbench: A benchmark for multi-llm routing system, 2024. URL https://arxiv.org/abs/2403.12031.

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

Kolawole, S., Dennis, D., Talwalkar, A., and Smith, V. Agreement-based cascading for efficient inference, 2025. URL https://arxiv.org/abs/2407.02348.

Kusupati, A., Bhatt, G., Rege, A., Wallingford, M., Sinha, A., Ramanujan, V., Howard-Snyder, W., Chen, K., Kakade, S., Jain, P., and Farhadi, A. Matryoshka representation learning, 2024. URL https://arxiv.org/abs/2205.13147.

Li, H., Zhang, Y., Guo, Z., Wang, C., Tang, S., Zhang, Q., Chen, Y., Qi, B., Ye, P., Bai, L., Wang, Z., and Hu, S. Llmrouterbench: A massive benchmark and unified framework for llm routing, 2026. URL https://arxiv.org/abs/2601.07206.

Li, J., Fang, A., Smyrnis, G., Ivgi, M., Jordan, M., Gadre, S., Bansal, H., Guha, E., Keh, S., Arora, K., Garg, S., Xin, R., Muennighoff, N., Heckel, R., Mercat, J., Chen, M., Gururangan, S., Wortsman, M., Albalak, A., Bitton, Y., Nezhurina, M., Abbas, A., Hsieh, C.-Y., Ghosh, D., Gardner, J., Kilian, M., Zhang, H., Shao, R., Pratt, S., Sanyal, S., Ilharco, G., Daras, G., Marathe, K., Gokaslan, A., Zhang, J., Chandu, K., Nguyen, T., Vasiljevic, I., Kakade, S., Song, S., Sanghavi, S., Faghri, F., Oh, S., Zettlemoyer, L., Lo, K., El-Nouby, A., Pouransari, H., Toshev, A., Wang, S., Groeneveld, D., Soldaini, L., Koh, P. W., Jitsev, J., Kollar, T., Dimakis, A. G., Carmon, Y., Dave, A., Schmidt, L., and Shankar, V. Datacomp-lm: In search of the next generation of training sets for language models, 2025. URL https://arxiv.org/abs/2406.11794.

Li, X., Li, Z., Li, J., Xie, H., and Li, Q. 2d matryoshka sentence embeddings, 2024. URL https://arxiv.org/abs/2402.14776.

Miao, R., Yan, Y., Yao, X., and Yang, T. An efficient inference framework for early-exit large language models, 2024. URL https://arxiv.org/abs/2407.20272.

Moslem, Y. and Kelleher, J. D. Dynamic model routing and cascading for efficient llm inference: A survey, 2026. URL https://arxiv.org/abs/2603.04445.

Ong, I., Almahairi, A., Wu, V., Chiang, W.-L., Wu, T., Gonzalez, J. E., Kadous, M. W., and Stoica, I. Routellm: Learning to route llms with preference data, 2025. URL https://arxiv.org/abs/2406.18665.

Qiu, S., Potapczynski, A., Finzi, M., Goldblum, M., and Wilson, A. G. Compute better spent: Replacing dense layers with structured matrices, 2024. URL https://arxiv.org/abs/2406.06248.

Rauba, P. and van der Schaar, M. Deep hierarchical learning with nested subspace networks for large language models, 2026. URL https://arxiv.org/abs/2509.17874.

Reimers, N. and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks, 2019. URL https://arxiv.org/abs/1908.10084.

Shukla, A., Vemprala, S., Kusupati, A., and Kapoor, A. Matmamba: A matryoshka state space model, 2024. URL https://arxiv.org/abs/2410.06718.

Taghibakhshi, A., Sreenivas, S. T., Muralidharan, S., Cai, R., Chochowski, M., Mahabaleshwarkar, A. S., Suhara, Y., Olabiyi, O., Korzekwa, D., Patwary, M., Shoeybi, M., Kautz, J., Catanzaro, B., Aithal, A., Tajbakhsh, N., and Molchanov, P. Nemotron elastic: Towards efficient many-in-one reasoning llms, 2025. URL https://arxiv.org/abs/2511.16664.

Team, G. Gemma 3n. 2025. URL https://ai.google.dev/gemma/docs/gemma-3n.

Valipour, M., Rezagholizadeh, M., Rajabzadeh, H., Kavehzadeh, P., Tahaei, M., Chen, B., and Ghodsi, A. Sortednet: A scalable and generalized framework for training modular deep neural networks, 2024. URL https://arxiv.org/abs/2309.00255.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Wang, Y., Yue, Z., Zeng, H., Wang, D., and McAuley, J. Train once, deploy anywhere: Matryoshka representation learning for multimodal recommendation. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 13461–13472, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.786. URL https://aclanthology.org/2024.findings-emnlp.786/.

Wang, Y., Hu, Q., Ding, Y., Wang, R., Gong, Y., Jiao, J., Shen, Y., Cheng, P., and Su, J. Training matryoshka mixture-of-experts for elastic inference-time expert utilization, 2025. URL https://arxiv.org/abs/2509.26520.

Yu, J. and Huang, T. Universally slimmable networks and improved training techniques, 2019. URL https://arxiv.org/abs/1903.05134.

Yu, J., Yang, L., Xu, N., Yang, J., and Huang, T. Slimmable neural networks, 2018. URL https://arxiv.org/abs/1812.08928.

Zhang, B. and Sennrich, R. Root mean square layer normalization, 2019. URL https://arxiv.org/abs/1910.07467.

Zhang, Y., Coskun, H., Ma, X., Wang, H., Ma, K., Xi, Chen, Hu, D. H., and Fu, Y. Slicing vision transformer for flexible inference, 2024. URL https://arxiv.org/abs/2412.04786.

A. Notation

Symbol	Meaning
α, β	Model budgets/scales.
$\mathcal{A}$	The collection of budgets for the model family of interest.
M_α	Model with budget α.
T_α^ℓ	Layer ℓ of model M_α.
$\mathcal{X}$	Model input space.
$\mathcal{H}_\alpha$	Model output space for M_α.
$\pi_{\beta\to\alpha}$	A projection map from $\mathcal{H}_\beta$ to $\mathcal{H}_\alpha$. Part of a filtration.
$\mathcal{H}_\alpha^\ell$	Intermediate representation space of M_α at layer ℓ.
$\pi_{\beta\to\alpha}^\ell$	A projection map from $\mathcal{H}_\beta^\ell$ to $\mathcal{H}_\alpha^\ell$. Part of a filtration.
L	Total number of layers in M_α.
m	Number of submodels, $\lvert\mathcal{A}\rvert$.

Table 2. Notation.

B. Implementation Details

B.1. Training Details

All training code was built on Andrej Karpathy's Nanochat (commit 0aaca56) and inherits its defaults (Karpathy, 2025).

Pretraining.

Models use sequence length 2048 and the Nanochat tokenizer with vocabulary size 32768. Pretraining uses next-token cross-entropy loss on NVIDIA ClimbMix for 3.8B tokens (Diao et al., 2025). Unless otherwise specified, optimization uses the Nanochat AdamW/Muon optimizer split: embeddings, value embeddings, output embeddings, scalar parameters, and normalization parameters are optimized with AdamW, while matrix parameters are optimized with Muon. Matrix parameters are optimized with Muon using learning rate 2e-2, momentum warmed from 0.85 to 0.97, 5 Newton–Schulz steps, $\beta_2 = 0.9$, and matrix weight decay 0.28 cosine-decayed to zero. Token embeddings use learning rate 0.3, output embeddings use 8×10^{-3}, and scalar parameters use 0.5, with AdamW learning rates scaled by $(d_{\text{model}}/768)^{-1/2}$. Pretraining uses 40 warmup steps, a warmdown over the final 65% of training, and final learning-rate fraction 0.05. We use a total batch size of 2^{20} tokens. All compared models use the same pretraining token budget of 3.8B tokens.

Supervised Finetuning.

During SFT, we initialize from the pretrained checkpoint, use the same optimizer partition, set Muon weight decay to zero, multiply inherited learning rates by 0.8, use no warmup, and linearly decay the learning rate to zero over the final 50% of SFT. All compared models use the same SFT token budget of 509M total tokens. The SFT mixture contains 42.9% SmolTalk, 28.0% MMLU auxiliary data, 18.7% SimpleSpelling, 7.5% SpellingBee, 2.8% GSM8K, and 0.2% identity conversations by example count (Allal et al., 2025; Hendrycks et al., 2021; Cobbe et al., 2021).

B.2. Architecture-Specific Details

In addition to the details specified below, we summarize architecture configs in Table 3, and their language modeling performance in Table 4.

Family	Size	Layers	d_{model}	Heads	d_{head}	d_{attn}	d_{ff}	Params (M)
Dense	S	24	384	6	64	384	1536	218.6
Dense	M	24	704	11	64	704	2816	465.7
Dense	L	24	1024	16	64	1024	4096	771.8
Dense	XL	24	1344	21	64	1344	5376	1136.8
StairFormer	S / k=1	24	384	6	64	384	1536	218.6
StairFormer	M / k=2	24	768	12	64	768	3072	479.7
StairFormer	L / k=3	24	1152	18	64	1152	4608	783.3
StairFormer	XL / k=4	24	1536	24	64	1536	6144	1129.3
MatFormer FFN+Attn	S	24	1280	2	80	160	640	205.8
MatFormer FFN+Attn	M	24	1280	4	80	320	1280	327.7
MatFormer FFN+Attn	L	24	1280	8	80	640	2560	571.5
MatFormer FFN+Attn	XL	24	1280	16	80	1280	5120	1059.1
MatFormer FFN-only	S	24	1280	16	80	1280	640	783.8
MatFormer FFN-only	M	24	1280	16	80	1280	1280	823.1
MatFormer FFN-only	L	24	1280	16	80	1280	2560	901.8
MatFormer FFN-only	XL	24	1280	16	80	1280	5120	1059.1

Table 3. Architecture configurations for the compared model families. d_{attn} denotes the active attention width, i.e. heads times head dimension.

Family	Model	Active Params (M)	Forward FLOPs/token (M)	CORE	GSM8K (%)
Dense	S	218.6	185.6	0.153	2.0
Dense	M	465.7	470.0	0.199	3.3
Dense	L	771.8	872.4	0.220	6.3
Dense	XL	1136.8	1,393	0.226	6.1
StairFormer	S	218.6	185.6	0.100	1.0
StairFormer	M	479.7	456.1	0.165	1.4
StairFormer	L	783.3	811.6	0.194	2.3
StairFormer	XL	1129.3	1,252	0.201	2.7
MatFormer FFN-only	S	783.8	728.8	0.212	4.5
MatFormer FFN-only	M	823.1	807.4	0.213	4.5
MatFormer FFN-only	L	901.8	964.7	0.218	5.0
MatFormer FFN-only	XL	1059.1	1279.3	0.216	4.7
MatFormer FFN+Attn	S	205.8	233.3	0.154	1.1
MatFormer FFN+Attn	M	327.7	382.7	0.159	2.7
MatFormer FFN+Attn	L	571.5	681.6	0.163	3.9
MatFormer FFN+Attn	XL	1059.1	1279.3	0.165	2.5

Table 4. Model budgets and evaluation results on CORE metric and GSM8K. Forward FLOPs are estimated per token.

Standard Baseline Architecture.

We use the Nanochat GPT-style decoder architecture with rotary positional embeddings, QK normalization, untied input/output embeddings, ReLU^2 MLP activations, RMSNorm, no linear biases, value embeddings, residual scaling, and full-context causal attention. We train 4 standard Transformer baseline models: one corresponding to each budget of our StairFormer model. These model configurations are chosen to be close to the active parameter counts and inference FLOPs of their corresponding StairFormer models.

StairFormer Architecture.

The StairFormer model we train is similar to the baseline architecture, except:

1. Weight matrices are made to be block lower triangular, with four row blocks, i.e., a 4×4 block matrix with blocks above the diagonal masked to zero.
2. Instances of $\mathrm{RMSNorm}$ are replaced with $\mathrm{PrefixRMSNorm}$, corresponding to the filtration induced by these blocks.

All residual-stream linear maps in the Transformer blocks are block lower triangular, including the attention projections and MLP projections. We refer to the block lower triangular variants of MHSA and FFN as $\mathrm{PrefixMHSA}$ and $\mathrm{PrefixFFN}$, respectively. Token embeddings, value embeddings, the unembedding layer, and small auxiliary gates are kept dense. To extract submodel k, we retain the first k width blocks and, for each block-lower-triangular linear map, keep exactly the row-block parameters whose output block index is at most k.

MatFormer Architecture.

We have two variants of MatFormer models to use as baselines. One variant applies MatFormer-style nesting on the FFN hidden dimension only, and one variant applies the nesting on the FFN hidden dimension and attention heads. Both variants are trained as a single shared parent model, from which smaller submodels are extracted after pretraining and SFT. In the FFN-only variant, all submodels share the same residual width and attention computation, while smaller budgets use prefixes of the MLP hidden dimension. In the FFN+Attention variant, submodels additionally use prefixes of the attention heads, reducing the active QKV projections, output projection input width, value embeddings, and attention computation.

The MatFormer architecture used is the same as the large baseline, except trained MatFormer-style: each batch selects one submodel uniformly from $\{S, M, L, XL\}$. The XL MatFormer submodel corresponds to the full shared parent model. The S model corresponds to the first 1/8 of the hidden size of the FFN (and intermediate size for attention, for that variant), M corresponds to the first 1/4, and L corresponds to the first 1/2.

Training Losses. For the baseline model, we train with standard cross-entropy loss. The MatFormer models are trained with cross-entropy loss for the uniformly sampled submodel selected for each batch. The StairFormer model is trained using weighted cross-entropy loss
$(1-\lambda)\mathcal{L}_{\text{XL}} + \frac{\lambda}{3}\left(\mathcal{L}_L + \mathcal{L}_M + \mathcal{L}_S\right)$ with $\lambda = 0.1$. Investigation into different loss functions is left as future work.

B.3. Cascading Inference Experiment Details

For cascading inference, we use the SFT evaluation traces from GSM8K. For each example and each submodel, we cache the input prompt, generated response, task correctness, output length, and measured prefill and decoding FLOPs. A cascade begins with the smallest model in a family. After each response, a lightweight router decides whether to accept the current answer or defer to the next larger model. If all smaller models are rejected, the cascade terminates at the largest model in the family.

Our learned router is a per-task per-family per-transition MLP. That is, we train one router for each task, for each model family, for each of the family's cascade transitions. The router input is a frozen text embedding of the prompt–response pair, computed with `sentence-transformers/all-MiniLM-L6-v2` (Reimers & Gurevych, 2019). The MLP is trained as a binary classifier predicting whether the current response is correct. At inference time, we sweep a threshold on the predicted correctness probability: responses below the threshold are deferred, while responses above it are accepted.

We split evaluation examples into 40% router-training, 30% validation, and 30% test sets, using the same split across experiments. Thresholds are selected or swept on the validation split, and all reported accuracy–FLOP tradeoffs are computed on the held-out test split. We also report an oracle cascade, which defers exactly when the current model's

response is incorrect, as a theoretical upper bound. For StairFormer, we count prompt-side prefill computation as reusable across escalations, since the KV cache can be shared across these models.

C. Proofs

Proof of Proposition 2.3. Note that $\pi_{\beta\to\alpha} = \pi^L_{\beta\to\alpha}$ and $\pi^0_{\beta\to\alpha} = \mathrm{id}_{\mathcal{X}}$. In the definition of output nesting, each layer being filtration-preserving lets the projection be pushed through the model:

$$\begin{aligned}
\pi_{\beta\to\alpha} \circ M_\beta &= \pi^L_{\beta\to\alpha} \circ T^{L-1}_\beta \circ T^{L-2}_\beta \circ \cdots \circ T^1_\beta \circ T^0_\beta, \\
&= T^{L-1}_\alpha \circ \pi^{L-1}_{\beta\to\alpha} \circ T^{L-2}_\beta \circ \cdots \circ T^1_\beta \circ T^0_\beta, \\
&\vdots \\
&= T^{L-1}_\alpha \circ T^{L-2}_\alpha \circ \cdots \circ T^1_\alpha \circ \pi^1_{\beta\to\alpha} \circ T^0_\beta, \\
&= T^{L-1}_\alpha \circ T^{L-2}_\alpha \circ \cdots \circ T^1_\alpha \circ T^0_\alpha \circ \pi^0_{\beta\to\alpha}, \\
&= T^{L-1}_\alpha \circ T^{L-2}_\alpha \circ \cdots \circ T^1_\alpha \circ T^0_\alpha, \\
&= M_\alpha.
\end{aligned}$$

□

Proof of Proposition 2.4. Let $x = [x_1; \ldots; x_m]$. Since $W^{(m)}$ is block lower triangular, its i-th output block is $y_i = \sum_{j\le i} W_{ij}x_j$. Projecting the output of the larger layer onto the first k blocks gives

$$\pi_{\alpha_m\to\alpha_k} W^{(m)} x = \pi_{\alpha_m\to\alpha_k} \begin{bmatrix} T(x)_1 \\ \vdots \\ T(x)_m \end{bmatrix} = \begin{bmatrix} T(x)_1 \\ \vdots \\ T(x)_k \end{bmatrix} = W^{(k)} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} = W^{(k)} \pi_{\alpha_m\to\alpha_k} x,$$

as desired. □

Proof of Theorem 2.5. The projections π preserve vector addition. Therefore, it is sufficient to show that every component of StairFormer is filtration-preserving. Proposition 2.3 will imply that StairFormer is output-nested.

The components of StairFormer are linear layers, self-attention layers, $\mathrm{PrefixRMSNorm}$, and non-linearities in the MLP. The first of these was shown in Proposition 2.4. The next two fall out directly from the design of StairFormer, separating the different blocks into different heads and separating the normalization factor in $\mathrm{PrefixRMSNorm}$. The Q, K, and V projections for active head groups depend only on the active prefix, and attention is computed independently per head group. W^O is block lower triangular, so the projected full output is filtration-preserving.

What remains to be shown is that the non-linearity of the MLP is filtration-preserving. These nonlinearities are applied element-wise, so

$$\pi_{\alpha_m\to\alpha_k}(\phi(x)) = \begin{bmatrix} \phi(x_1) \\ \vdots \\ \phi(x_{\alpha_k}) \end{bmatrix} = \phi\left(\begin{bmatrix} x_1 \\ \vdots \\ x_{\alpha_k} \end{bmatrix} \right) = \phi(\pi_{\alpha_m\to\alpha_k}(x)).$$

□

D. Additional Details

D.1. Adapting Muon for Block Triangular Parameters

Since StairFormer uses block triangular weights, special care must be taken to adapt Muon to its structure. Rather than storing a dense matrix with an explicit mask, we parameterize each logical matrix W by independent row blocks

$$R_i = [W_{i,1}, W_{i,2}, \ldots, W_{i,i}] \in \mathbb{R}^{d^{\mathrm{out}}_i \times \sum_{j\le i} d^{\mathrm{in}}_j},$$

so that

$$y_i = R_i[x_1; \ldots; x_i].$$

Thus, the masked weights are not stored or updated during training.

To modify Muon accordingly, we do the following: each R_i is an ordinary dense matrix parameter. We place row blocks into separate Muon parameter groups by row-block index and shape, and use

$$\eta_i = \frac{\eta}{\sqrt{i+1}},$$

with $i = 0$ for the first row block. This preserves the block-triangular structure exactly while keeping Muon's orthogonalized update well-scaled across all prefix submodels. In other words, we treat each row of blocks as independent 2D matrices for Muon to optimize.

E. Extended Related Works

Elastic models. Many forms of elastic networks have been studied. One prominent work is Once-for-all (Cai et al., 2020), where a model can be trained once and specialized subnetworks can be extracted from it. Other works include slimmable networks (Yu et al., 2018; Yu & Huang, 2019) that can adjust their width on the fly, DynaBERT (Hou et al., 2020) which can adapt both width and depth, and SortedNet (Valipour et al., 2024) which extends nesting capabilities to normalization layers. Elastic networks can also slice embedding layers, as in HydraViT and Scala (Haberer et al., 2024; Zhang et al., 2024). Nesting capabilities have also been extended to modern components like mixture-of-experts layers as in M-MoE (Wang et al., 2025), or Mamba as in MatMamba (Shukla et al., 2024). Properties of MRL have also been extended to other modalities, such as vision in M3 (Cai et al., 2024).

Routing and cascading inference. Routing methods route inputs between various models, e.g. at the token level or at the query level (Hu et al., 2024; Li et al., 2026). These routing methods often rely on lightweight MLP modules trained on embeddings, and some work has explored using ensemble-based strategies that rely on running full models (Ong et al., 2025; Chen et al., 2026). Other methods employ cascading strategies, where a family of progressively more expensive models is used during inference (Dekoninck et al., 2025; Moslem & Kelleher, 2026; Kolawole et al., 2025).

Early exit. Models have been explored that can exit a forward pass early, saving computational resources (Miao et al., 2024; Bajpai & Hanawal, 2025; Chen et al., 2024). One particularly relevant method is LayerSkip (Elhoushi et al., 2024), which skips later layers in an LLM. Our method is orthogonal to LayerSkip and can be thought of as similar to a width-wise analogue of it. 2DMSE extends this idea by combining it with MRL for the early exit outputs, (Li et al., 2024) but not all the Matryoshka outputs are consistent with each other.