Mechanistic Design and Scaling of Hybrid Architectures

Michael Poli [*1 2] **Armin W Thomas** [*2] **Eric Nguyen** [*2]
Pragaash Ponnusamy [1] **Björn Deiseroth** [3] **Kristian Kersting** [3] **Taiji Suzuki** [4 5]
Brian Hie [6 2] **Stefano Ermon** [2 7] **Christopher Ré** [2] **Ce Zhang** [1] **Stefano Massaroli** [4 8]

Abstract

The development of deep learning architectures is a resource-demanding process, due to a vast design space, long prototyping times, and high compute costs associated with at-scale model training and evaluation. We set out to simplify this process by grounding it in an end-to-end *mechanistic architecture design* (MAD) pipeline, encompassing small-scale capability unit tests predictive of scaling laws. Through a suite of synthetic token manipulation tasks such as compression and recall, designed to probe capabilities, we identify and test new hybrid architectures constructed from a variety of computational primitives. We experimentally validate the resulting architectures via an extensive compute-optimal and a new state-optimal scaling law analysis, training over 500 language models between 70M to 7B parameters. Surprisingly, we find MAD synthetics to correlate with compute-optimal perplexity, enabling accurate evaluation of new architectures via isolated proxy tasks. The new architectures found via MAD, based on simple ideas such as hybridization and sparsity, outperform state-of-the-art Transformer, convolutional, and recurrent architectures (Transformer++, Hyena, Mamba) in scaling, both at compute-optimal budgets and in overtrained regimes. Overall, these results provide evidence that performance on curated synthetic tasks can be predictive of scaling laws, and that an optimal architecture should leverage specialized layers via a hybrid topology.

[*]Equal contribution [1]Together AI [2]Stanford University [3]Hessian AI [4]RIKEN [5]The University of Tokyo [6]Arc Institute [7]CZ Biohub [8]Liquid AI. Correspondence to: Michael Poli <polimic03gmail.com>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

1. Introduction

Alongside data quality, the effectiveness of large-scale training is determined by the quality of a model architecture [17, 14], which is defined by the set and arrangement of the computational primitives used to form layers and functional blocks, as well as their parametrization.

Due to the combinatorial explosion of possible architecture designs and a lack of reliable prototyping pipelines – despite progress on automated neural architecture search methods [40] – architectural improvements are obtained through an opaque development process guided by heuristics and individual experience, rather than systematic procedures. Further adding to this issue are the large costs and long iteration times associated with training and testing new architectures, underscoring the need for principled and nimble design pipelines.

In spite of the wealth of possible architecture designs, the majority of models rely on variations of the same uniform Transformer recipe, based on a regular interleaving of memory-based mixers (self-attention layers) with *memoryless* mixers (shallow FFNs) [37, 16]. This particular combination of computational primitives – originating from the first Transformer design [38] – is known to improve quality, with empirical arguments supporting the notion that these primitives specialize in different sequence modeling sub-tasks e.g., in-context versus factual recall [10]. Beyond the Transformer architecture are a class of emerging computational primitives inspired by signal processing, based on gated convolutions and recurrences [18, 27, 28, 25, 11, 41], promising improved quality, cheaper scaling to long sequence length, and efficient inference. These new primitives expand the architecture design space, offering new opportunities to extend capabilities and specializations of models.

In this work, we set out to explore key questions arising from these observations:

1. *Can the architecture design process be streamlined through a set of simple pretext token manipulation tasks, providing quick and cheap performance estimates predictive of scaling laws?*

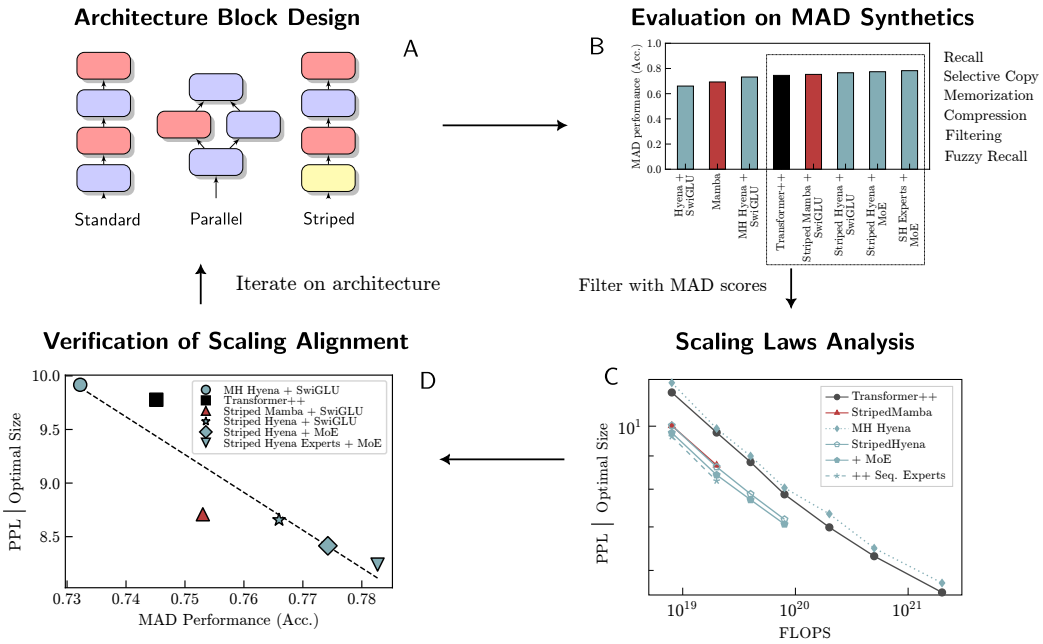

Figure 1.1: *Mechanistic architecture design* (MAD) is a framework to enable fast iterative improvement of architectures, including emerging approaches based on recurrences and convolutions. **[A]:** Design architectures via selection of computational primitives and topology. **[B]:** MAD involves an evaluation of architecture designs at small scale on a set of token manipulation synthetic tasks, curated to unit test a variety of model capabilities. The experimental setup promotes direct comparison via normalization of total state dimension for recurrent models. **[C]:** Validate scaling laws of top-performing models on MAD synthetics in compute-optimal and overtrained regimes. Results in B used to reduce the number of candidate architectures. **[D]:** Verify alignment of scaling properties and (MAD) results for each architecture e.g., correlation of compute-optimal scaling perplexity and aggregate (MAD) score (in the figure, compute-optimal perplexity at 2e19 FLOP budget is shown). If the scores between target quantity and (MAD) synthetics are correlated, iterate on a single target architecture.

2. *Is it possible to bring together the "best of all worlds" by arranging different computational primitives into hybrid architecures, leveraging their respective specialized capabilities?*

In an attempt to provide answers to these questions, we make the following core contributions:

Mechanistic architecture design We introduce a methodology for the fast prototyping and testing of new architectures, *mechanistic architecture design* (MAD). MAD is a collection of synthetic tasks – such as recall, memorization, and compression – curated to serve as isolated unit tests for key capabilities of an architecture, requiring only minutes of training time. In particular, MAD tasks are inspired by progress on understanding the inner workings of Transformers and other sequence models via in-context learning, recall, and other sequence manipulation tasks [26, 8, 3, 2, 1]. We apply MAD to test architectures built with representative computational primitives such as gated convolutions [28], gated input-varying linear recurrences [11, 41], and other operators e.g., *mixture of experts* (MoEs) [33], as well as novel ones. With MAD, we are able to filter for promising architecture candidates (Fig, 1.1, **[A,B]**). By identifying which individual tasks computational primitives excel at, we find and validate several ways to improve designs, such as *striping* i.e., sequentially interleaving blocks composed of different computational primitives with a specified interconnection topology, resulting in hybrid architectures [21, 8, 7].

Scaling laws of emerging architectures To investigate the link between MAD synthetics and real-world scaling, we execute the **largest scaling law analysis on emerging architectures to date**, training over 500 language models between 70 million and 7 billion parameters with different architectures. Our protocol builds and expands on compute-optimal scaling laws for LSTMs and Transformers [17, 35, 14]. Our findings show that hybrid architectures improve on all scaling measures, resulting in lower pretraining losses at different *floating point operation* (FLOP) compute-budgets at the compute-optimal frontier[1]. We also verify new architectures to be more robust to large pretrain-

[1]Found via the optimal allocation of compute to tokens and model size.

ing runs outside the efficient frontier e.g., smaller models trained for significantly more tokens, which make up a majority of training settings in practice due to inference cost considerations [30].

Hybridization insights at scale Building on our scaling law analysis, we investigate hybridization schedules and model topology. Our findings uncover optimal hybridization ratios for attention [38], Hyena [28], and Mamba [11] mixtures, as well as the respective placement of these layers in an architecture.

State-optimal scaling laws The size of the *state* – the analog of *kv-caches* in standard Transformers [23] – of emerging convolutional and recurrent primitives [28, 11] plays a central role in MAD and our scaling analysis, as it determines inference efficiency, memory cost, and provably has a direct effect on recall capabilities [2]. We introduce a *state-optimal scaling* analysis, with the objective of estimating how perplexity scales with the state dimension of different model architectures. We find hybrid architectures to balance the trade-off between compute requirements, state dimension, and perplexity.

New state-of-the-art architectures Leveraging MAD and new computational primitives, derived from the insights developed in this work, we design new state-of-the-art hybrid architectures, outperforming the best Transformer, convolutional, and recurrent baselines (Transformer++ [37], Hyena, Mamba) with a reduction of up to 20% in perplexity for the same compute budget.

Correlation between synthetics and scaling performance Finally, we provide the first evidence that a curated selection of MAD synthetic tasks can be used to reliably predict scaling law performance, paving the way to faster, automated architecture design. In particular, MAD accuracy is rank-correlated with compute-optimal perplexity at scale (Fig. 1.1, [D]), with particularly strong correlation for models in the same architecture class (Fig 4.1).

2. Background: Architecture Design

Architecture design refers to the selection and optimization of (a) computational primitives and their composition into layers and blocks, and (b) topology i.e., the interconnection and placement of individual blocks in an architecture.

In the following, we define the bounds of the architecture design search space explored in this work. In particular, we provide details on the emerging class of implicit subquadratic models, since their properties drive the design of the synthetic task and evaluation pipeline in MAD, and motivate the introduction of a state-optimal scaling law analysis.

2.1. Computational primitives

Architectures are compositions of linear and nonlinear functions with learnable parameters. Common choices for the former are parametric dense or structured layers $\mathsf{L} : \mathbb{R}^T \to \mathbb{R}^T$, $y = \mathsf{L}(u)$. As an example,

$$\text{dense} \qquad y_t = \sum_{t'=1}^{T} \mathsf{W}_{tt'} u_{t'}, \qquad \mathsf{W} \in \mathbb{R}^{T \times T}$$

$$\text{(causal) conv.} \qquad y_t = \sum_{t'=1}^{t} \mathsf{W}_{t-t'} u_{t'}, \qquad \mathsf{W} \in \mathbb{R}^{T}.$$

It is often useful to differentiate between explicitly and implicitly parametrized layers, depending on whether the entries $\mathsf{W}_{tt'}$ are the learnable parameters of the layer or are themself parametric functions of positional encodings or of the input, i.e. $(t, t', u) \mapsto \mathsf{W}_{tt'}(u)$ [28]. Implicit parametrizations disentangle the number of model parameters and dimensionality T of the inputs. Further, they can be leveraged to create complex dependencies on the inputs in the entries of $\mathsf{W}(u)$ such as in self-attention, $\mathsf{W}_{tt'}(u) = \sigma(\langle Qu_t, Ku_{t'} \rangle)$. This ensures the layer can be applied to inputs with large T without a prohibitive parameter and memory cost. We often refer to the implicit parametrization for an implicit layer as its *featurization path*.

On nonlinearities in architecture design Linear primitives are typically interconnected via nonlinearities and residuals. Common nonlinearities are applied elementwise or to some specific dimension (e.g., the softmax used in attention). [20, 38]. Another commonly employed nonlinearity is gating, resulting in a polynomial function of the input. While other lines of work investigate choice and placement of nonlinearities in a layer to optimize quality, efficiency, or to minimize the emergence of activation outliers [34], these quality improvements are smaller compared to other layer and topology changes[2] and are thus outside the scope of this work.

Implicit primitives Implicitly parametrized computational primitives are the backbone of most model architectures of practical interest. An important class of implicit layers can be described starting from so-called *linear attention* [18, 31, 15][3], in its simplest (single-channel, unnormal-

[2] Many tweaks to activation choice, placement and presence of biases are carried out to improve numerical stability and reduce the presence of large outliers in activations, rather than improve scaling performance.

[3] We use t for consistency, although in practice these layers can be applied to both "sequence" dimension, as well as "width" dimension.

ized[4]) form

$$\begin{aligned} \text{recurrence} \quad & x_{t+1} = x_t + k_t(u)v_t(u) \\ \text{readout} \quad & y_t = q_t(u)x_t \end{aligned} \tag{2.1}$$

where $q, k, v : \mathbb{R}^T \to \mathbb{R}^T$ are the featurization path of the layer. Linear attention is a linear *recurrent neural network* (RNN) or *state-space model* (SSM) with constant identity state-to-state dynamics, and implicitly-parametrized input-to-state and state-to-output mappings. Linear attention can be evaluated in parallel during training or inference prefilling using its parallel form $y_t = q_t \sum_{t'=1}^{t} k_{t'} v_{t'}$, without materializing the state x. Notably, the class of subquadratic implicit models [28, 11, 41] emerges as generalizations of (2.1) with a few key differences.

2.2. State, cache, and memory

In autoregressive tasks, such as text generation, recurrent models enable lower latency and constant memory generation, since the fixed state x_t replaces the cache required in other generic nonlinear blocks such as attention e.g., the *kv-cache*. Indeed, kv-caches can be seen as a state of dynamic size, by reformulating attention as a recurrence with state size T, see [23]. For this reason, we use fixed states and dynamic states to refer to states and kv-caches in hybrid architectures.

Nonparametric state expansion tricks The size of the state and its utilization play a central role in the taxonomy, analysis, and design of efficient architectures. State size, as well as the parametrization of a block, determine memorization and recall capabilities of a layer, as well as inference efficiency. For this reason, different approaches have been developed to expand the state dimension without prohibitive parameter cost. The main ones are the *outer-product head trick*:

$$\begin{aligned} x_{t+1} &= x_t + (k_t \otimes I_M)v_t, & k_t, v_t, q_t &\in \mathbb{R}^M \\ y_t &= (I_M \otimes q_t)x_t, & x_t &\in \mathbb{R}^{M^2}. \end{aligned}$$

Note that we have used a vectorized notation instead of the commonly employed matrix notation for models using the state expansion trick. This configuration linearly increases the state size from a head dimension M to a total of M^2, and is employed in most linear attention variants [18], Hyena and RWKV variants [23, 27] as well as GLA [41].

The second method to expand the total number of states per layer is achieved via the *multi single-input single-output* (mSISO) layer configuration, which is equivalent to applying multiple independent recurrences with M states in parallel.

Given the importance of the total state dimension in determining the capacity of a layer, we find **model comparisons in an *iso-state* setting** – normalizing for the total number of states regardless of the specifics of the layer – to be required to ensure architecture improvements measured on smaller scale synthetic tasks can transfer to pretraining results at scale.

Manipulating the state Beyond state expansion techniques, efficient layers can be taxonomized based on their parametrization of state-to-state dynamics and their implicit parameters. For example, an input-varying layer introduces additional featurization path to extend input-variance to state-to-state transitions e.g., $x_{t+1} = g_t(u)x_t + k_t(u)v_t(u)$. We choose three state-of-the-art approaches spanning different possible combinations shown in Table 2.1.

The layers also vary slightly in their featurization paths e.g., GLA uses a low-rank elementwise implicit state-to-state transition, whereas Mamba uses a different low-rank parametrization and weight-tying.

2.3. Topology

Beyond the specifics of the layer itself, designing architectures involves arranging these computational primitives into blocks, interconnected with a particular topology, for example, sequential, parallel, or hybrid (as illustrated in Fig. 1.1). In this work, we explore sequential striped topologies i.e., where different computational primitives are applied sequentially, as well as sparse parallel topologies i.e., mixture of experts.

3. Mechanistic Architecture Design

In the ideal case, we would have access to an oracle capable of quantifying how changes in model design at the microscopic level – choice of computational primitives, parametrization, topology – propagate to the macroscopic scale i.e., scaling laws. Indeed, a key challenge in architecture design is predicting whether new designs will match or improve quality of existing baselines at scale.

Our working hypothesis is that the performance of an architecture primarily stems from its efficiency in performing an array of smaller token manipulation tasks well. We show that by probing the performance of architectures in

[4]For simplicity we detail unnormalized layers, as normalization simply redefines the operator as the ratio of two recurrences.

[6]Only the input-to-state and output-to-state maps are input-varying.

[6]Input-to-state and state-to-output maps are shared across channels.

Hyena [28]	*weakly* input-varying[5]	mSISO
Multi-Head Hyena [23]	*weakly* input-varying	mSISO with heads
Gated Linear Attention [41]	input-varying	heads
Mamba [11]	input-varying	mSISO and weight sharing[6]

Table 2.1: Taxonomy of layers based on recurrence class.

each of these individual tasks at a small scale, one can recover relative model rankings matching those obtained via scaling laws analysis in quantities of interest such as compute-optimal perplexity. We call this process of capability identification and evaluation, with the goal of architecture prototyping, *mechanistic architecture design* (in short "MAD"). Beyond approximating scaling performance, MAD provides a means to probe the compositionality of model skills.

3.1. Synthetic tasks to probe model skills

MAD utilizes synthetic tasks to probe model skills and inform model design, building on recent works [8, 28, 2] considering only a single or subset of these tasks. We provide a schematic for each task, with x representing the input, y the target sequence, and `prompt` the evaluation sequence.

3.1.1. In-context recall

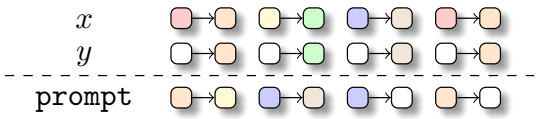

Figure 3.1: Schematic of in-context recall. White tokens are masked; y represents target sequences during training. At test time, the model is evaluated on recall of all key-value pairs that were already presented in the sequence.

To answer a prompt well, language models must be able to understand and learn from new information presented in the prompt (so-called in-context learning [6]).

A wealth of empirical work has demonstrated that the associative recall task, as studied in [8, 28], is well-suited to test a specific subset of in-context learning ability. Here, we are using a multi-query variant of this task, as proposed by [2]: given an input sequence of key-value pairs, models are tasked with retrieving all values associated with keys that were already shown in the input sequence.

To solve this task, a model thereby does not need to learn any information external to the prompt it is provided with at test time.

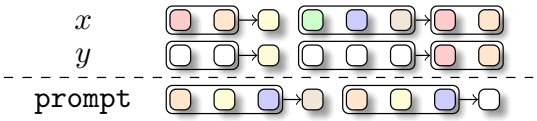

Figure 3.2: Fuzzy in-context recall. Boxes indicate adjacent tokens that form a key/value.

3.1.2. Fuzzy in-context recall

In language, semantic units are often spread out over multiple adjacent tokens (e.g., "blue sky" vs "gray sky"). To test how capable a model is of semantically grouping together adjacent tokens, we utilize a variant of in-context recall, in which keys and values are composed of a variable number of adjacent tokens.

For each sequence, variable length keys and values are randomly drawn from the vocabulary and then assigned into pairs. Since the structure of key/value lengths in a sequence, as well as the mapping from keys to values, change between sequences, fuzzy recall can be regarded as a more challenging variant of in-context recall.

3.1.3. Noisy in-context recall

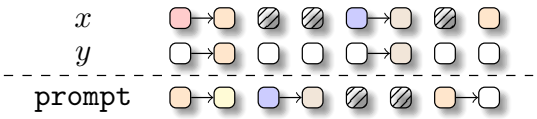

Figure 3.3: Schematic of noisy in-context recall.

To answer a prompt well, language models must be able to ignore irrelevant information of the input.

We test this ability with another modification to standard in-context recall. Here, irrelevant information, represented by *noise* tokens from a special subset of the vocabulary, is added in an arbitrary and variable pattern in between the key-value pairs. Since the noise tokens are sampled from a fixed dictionary, this task requires the model to implement a specific type of memory, in addition to the recall circuits required for in-context recall. In particular, the model needs to remember which tokens belong to the set of *noise* tokens, as these do not carry relevant information for the task.

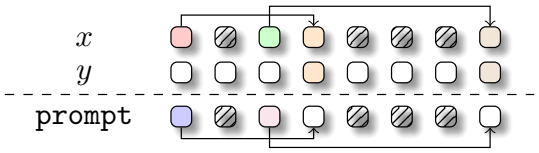

Figure 3.4: Schematic of the selective copy task. Grayed-out tokens are noise.

3.1.4. Selective Copying

In addition to ignoring irrelevant information of an input, language models must be able to selectively remember relevant information of an input.

In the selective copying task, models are tasked with copying tokens from one position of an input sequence to a later position of the sequence, while ignoring irrelevant noise tokens that are inserted into the sequence. Tokens are always copied in their order of occurrence. Models thereby need to not just remember the tokens that are to be copied but also their specific order of occurrence in the sequence. The copy positions are gleaned from the structure of each sample, while the contents change between samples and must be inferred in-context.

3.1.5. Compression

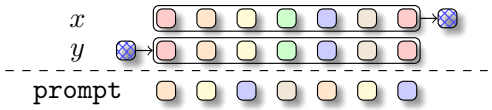

Figure 3.5: Schematic of the compression task. A sequence is encoded into a single token, and then decoded to reconstruct the original sequence.

Recent findings in the mechanistic interpretability literature [24] indicate that language models are often required to perform "token concatenation", where early sequence-mixing layers (e.g., attention) assemble information that is spread across multiple tokens in an input onto another token so that the assembled information can then be decoded well by subsequent channel-mixing layers (e.g., MLPs).

To test this capability we use a compression task, in which models are tasked with compressing a random sequence of input tokens into a single aggregation token, in a way that enables reconstruction via an MLP. In other words, the compression task tests the ability of a model to compress token embeddings into a single one with the least amount of information loss.

3.1.6. Memorization

In addition to manipulating and retrieving information from an input sequence, language modeling requires the memorization of factual knowledge.

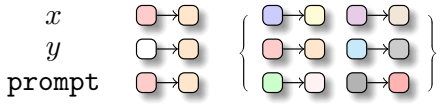

Figure 3.6: Schematic of the memorization task. The model is tasked with learning a fixed map between tokens (i.e., a set of "facts").

To test this skill, we utilize a memorization task, in which models are tasked with learning a fixed key-value mapping (resembling facts in language) from the training data. Unlike recall, the mapping requires no in-context computation as the ground-truth mapping is constant across samples.

3.2. MAD Protocol

MAD follows a two-step procedure, starting from the design of a new candidate architecture, followed by its systematic evaluation according to the following key principles:

i. Each MAD score is obtained by averaging architecture performances across a range of task difficulty levels. To manipulate difficulty, we independently vary a set of relevant experimental variables: length of the input sequence, size of the vocabulary, and size of the training set. Some tasks have additional variables such as the ratio of noise tokens in the noisy recall and selective copying tasks (Appendix B.1 and B.5).

ii. Fixed-state architectures are normalized to an iso-state and iso-parameter setting, including models featuring sparsely activated layers such as *mixtures of experts* (MoEs) [33]. Here, we normalize all fixed-state architectures to a common total state dimension of 4096 to control for any differences in model performance driven primarily by mismatch in model state dimension (Appendix B.3).

iii. To ensure that model performance estimates are not dependent on a specific training setting, we sweep each architecture in each task setting over a grid of learning rate and weight decay values. We only include the best runs in our final analysis (Appendix B.4).

iv. Model performances are always evaluated in an independent evaluation dataset, specific to each task setting.

3.3. Candidate architecture designs

We apply MAD to a set of small two-blocks architectures built from a collection of common primitives such as attention, SwiGLU [32], and variants of efficient implicit recurrent and convolutional layers described in Sec. 2.2. We build different types of architectures with these primitives: sequential, striped, and sparse parallel (mixtures).

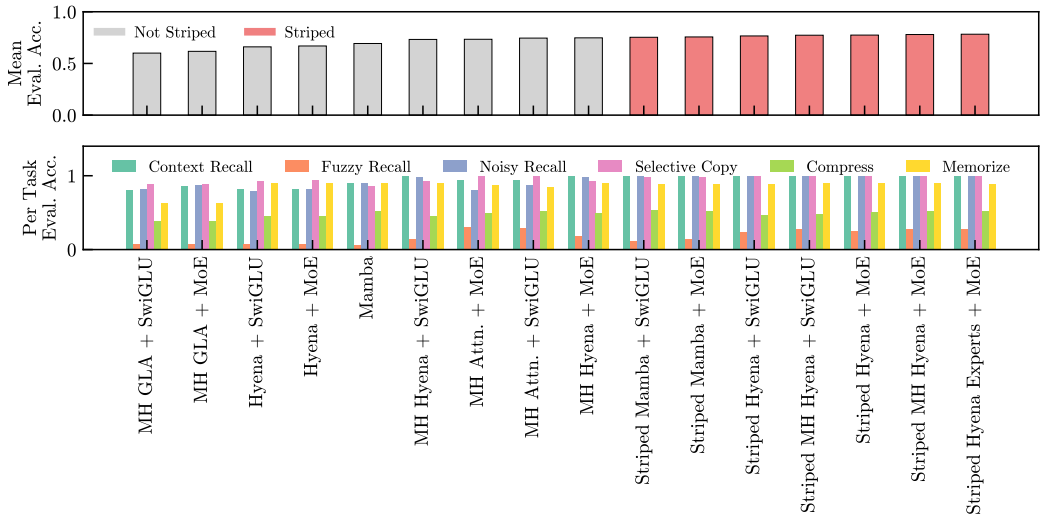

Figure 3.7: MAD analysis: An extensive evaluation of a suite of model architectures, built from common sequence- and channel-mixing layer types, across six synthetic tasks, each designed to probe a specific skill relevant for sequence modeling at scale.

In total, we evaluate 21 distinct architectures, including combinations of the primitives described in Sec. 2. Additional architecture details are provided in (Appendix B).

Mixture of sequence experts We further introduce to our MAD analysis a layer inspired by sparsely gated channel mixers, the *Hyena experts* layer. In a Hyena experts layer with E experts and K active experts, a router selects from a set of smaller Hyena mixers, using a router $G(u) : u \mapsto s$ from input sequence $u \in R^{T \times D}$ to scores $s \in \mathbb{R}^{T \times K}$:

$$s_t = \mathsf{softmax}(\mathsf{top}_K(u_t \mathsf{W}_g)), \quad \mathsf{W}_g \in \mathbb{R}^{D \times E}$$

$$\mathsf{HyenaExperts}(u)_t = \sum_{k'=1}^{k} s_{tk'} \mathsf{Hyena}(u)_{tk'}.$$

An advantage of the Hyena experts layer is that only a subset of the total state dimension is used to compose the output at each time step. We note that sparsely gated recurrences have also been explored for recurrences in [29], and that other similar schemes for sparse gating at the state level are also possible using input-varying recurrent primitives.

3.4. Results

We test a suite of architectures in the MAD protocol. In addition to ranking overall model performances across the synthetic tasks (Fig. 3.7), we take a high-level view on general patterns in model performances related to their design, including the presence of specific computational primitives in an architecture and the architecture's topology. We indicate a model's performance by its accuracy in correctly predicting tokens in the synthetic tasks. Note that model performances in MAD can likewise be measured through their evaluation loss (see Appendix B.1). Both performance metrics yield similar model rankings.

Hybridization to combine specialized layers Inspecting the performance on individual tasks via a stratified analysis (Appendix B.5) reveals specialization of architectures built with a single type of primitive, such as Mamba excelling at compression and Hyena at fuzzy recall.

> **Finding** 1: Striped architectures outperform all non-striped architectures on composite metrics, with an average gain in accuracy of 8.1% across the MAD synthetic tasks (Fig. 3.7).

We further find MAD performance to increases with models' total fixed state dimension, underscoring the importance of normalizing state dimensions when comparing model capabilities, further motivating a state-optimal scaling law analysis (Fig. D.2).

Head expansion trick It is beneficial to arrange the fixed state dimension into larger heads with fewer states instead of smaller heads with additional states (in the limit case, in a mSISO configuration).

> **Finding** 2: Architectures that expand their total state dimension through heads (see Sec. 2.2) outperform architectures without heads, with an average gain of 2.3% in accuracy across the MAD synthetic tasks (Fig. 3.7).

We note that the head expansion trick also linearly increases the computation in the layer, and for this reason it intro-

duces a trade-off between compute-optimality and state-optimality.

Sparse layers We find sparsely gated layers to outperform dense layers in MAD synthetics, in line with the literature on mixture of experts and their benefits.

> **Finding** 3: MAD performance improves with the addition of sparsely activated mixture of expert channel-mixing layers, when compared to architectures using SwiGLU channel mixers, with an average gain in accuracy of 1.7% across tasks (Fig. 3.7).

In our later analyses, we connect the performance of architectures on MAD to their performance at scale on The Pile [9] (Fig. 4.1).

4. Scaling Analysis

We seek to verify the connection between mechanistic design tasks and performance at scale. For this reason, we execute an extensive scaling law analysis on language pretraining, expanding on the framework of [17, 14]. We train more than 500 models of different architectures.

Let $\mathcal{M}_{w,\xi}$ be a model with parameters w and architecture ξ. Denote with $N = |w|$ the number of parameters, with D the total number of training tokens, and the training cost (in *floating point operations*, FLOPS) with $c_\xi(N, D)$. Let $\mathcal{A}_\xi(C)$ be the set of tuples (N, D) such that the training cost is exactly C, $\mathcal{A}_\xi(C) \coloneqq \{(N, D) \mid c_\xi(N, D) = C\}$. Given a tuple $(N, D) \in \mathcal{A}_\xi(C)$ one can evaluate $\mathcal{L}_\xi(N, D)$, the loss achievable for that combination of parameters/tokens. A point $(C, \ell(C))$ in the locus of the *compute-optimal* frontier in the loss-compute plane is defined as

$$(C, \ell(C)) \;:\; \ell(C) = \min_{(N,D)\in\mathcal{A}_\xi(C)} \mathcal{L}_\xi(N, D)$$

with $\ell(C)$ indicating the best loss achievable by training $\mathcal{M}_{\theta,\xi}$ at compute budget C, optimizing the allocation of compute to model size N and training tokens D, for architecture ξ. Relatedly, one may seek the functional form of the compute-optimal frontier in the parameter-compute or token-compute planes, composed of tuples (C, N^*) and (C, D^*), where D^*, N^* represent the optimal i.e., achieving lowest loss, allocation subject to the $(N^*, D^*) \in \mathcal{A}_\xi(\mathcal{C})$ constraint.

A primary objective of scaling law analyses is to determine such optimal allocation of the computational budget. To estimate efficient frontiers, we use an IsoFLOP approach, which explores different allocation ratios of model parameters and number of tokens at each compute budget. The loss optimum is then estimated via a quadratic fit (see Fig E.2).

4.1. Compute-optimal frontier for new architectures

Our first set of findings is related to the efficient frontier of the baseline Transformer++ [37] in relation to other architectures. [14] finds that when ξ is a standard Transformer architecture (combining attention and MLP), the optimal ratios between the number or model parameters, training tokens, and compute budget, are explained by a linear relationship in log-log space, i.e., $\log N^* \propto a \log C$ and $\log D^* \propto b \log C$.

> **Finding** 5: Let $a_{\mathtt{H}}, a_{\mathtt{T}}, b_{\mathtt{H}}, b_{\mathtt{T}}$ be the parameter size and data allocation coefficients for striped and Transformer models, respectively. We estimate $a_{\mathtt{T}} > a_H$ and $b_{\mathtt{T}} < b_H$ (Fig. E.1).

Optimal allocation of tokens and parameters is relatively stable under striping, with marginal differences. One notable difference is that optimal compute allocation in emerging efficient architectures is skewed towards additional data i.e., training smaller models for longer.

Beyond the efficient frontier Next, we look at optimality gaps when training outside the efficient frontier. By optimality gap, we refer to the increase in loss by training outside the compute-optimal frontier i.e., $\mathcal{L}(C(\tilde{N}, \tilde{D}, \xi))$ where $\tilde{N} = N^* + \delta N^*$ and the number of tokens $\tilde{D}$ is adjusted to preserve the total compute cost.

> **Finding** 6: The off compute-optimal perplexity gap is proportional to the hybridization ratio (Fig.E.2), for all IsoFLOP groups.

Intuitively, models with "flatter" IsoFLOP perplexity curves are preferred for overtraining smaller models, a setting particularly common in practice, as it results in smaller models with faster inference. Interestingly, the suboptimality gap in hybrids is smaller than Transformers, meaning they are better suited to training outside the optimal frontier.

Striping schedule and topology We study compute-optimal ratio and allocation of attention operators in striped architectures, as well as their overall topology (Fig. E.1).

> **Finding** 7: The compute-optimal hybridization ratio for striped models is 25% across all IsoFLOP groups[7] (Fig.E.2 and Table E.1).

4.2. State-optimal scaling

Beyond driving MAD synthetics performance, the total state size in a model is also an important factor in determining inference latency and memory cost. We explore *state-optimal* scaling, aiming to provide a coarse estimate of state utiliza-

[7] Accounting for state-optimality shifts the optimal ratio to 10%.

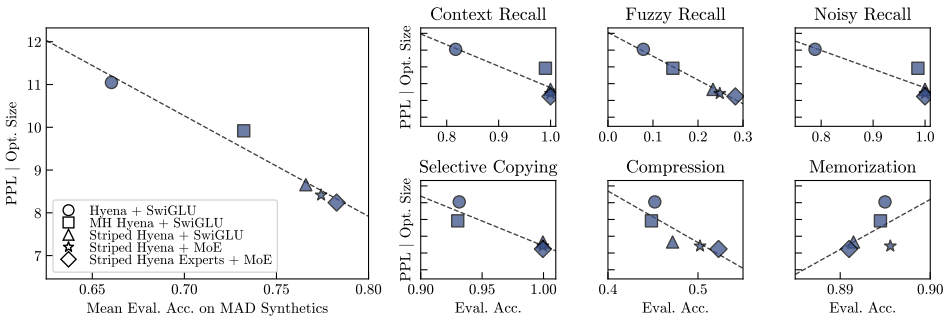

Figure 4.1: Improved performance on MAD synthetics correlates with better compute-optimal perplexity on The Pile. We highlight progressively improved versions of Hyena that were designed with the MAD pipeline, which translated to improved perplexity on the Pile (shown for $2e19$ FLOPs; see Appendix B.8 for an analysis across IsoFLOP groups).

tion by measuring scaling in perplexity over state dimension (Fig. D.2, right).

> **Finding** 8: There exists a relation of the type $P^* \propto M^c$ between compute-optimal perplexity P^* and total state size M, with $c \approx -0.28$ in our scaling experimental setup, consistent across all model architectures. The model class determines the offset of the state-optimal curve.

Concretely, state-optimal scaling indicates that one may reach any target perplexity (up to saturation of compute-optimal scaling laws) with fixed-state architectures, by paying a FLOP cost multiplier that depends on the model class – training longer to maximize state utilization. Input-varying recurrences, multihead and striped hybrid architectures achieve a favourable trade-off between metrics, with comparable or improved compute-optimal perplexity to Transformers++ and a reduced total state dimension.

5. Connecting MAD to scaling metrics

The goal of MAD is to provide a framework that can accelerate the architecture design process by using small synthetic tasks, which can be evaluated quickly and with little compute, to estimate whether improvements to an existing architecture, or a new candidate architecture, will perform well at scale. To gauge this hypothesis, we study the correlation between MAD scores and scaling properties of interest.

Correlation to compute-optimal perplexity We start with a case study using the Hyena [28] architecture. MAD has indicated that the performance of Hyena can be cumulatively improved by i) adding heads to the Hyena sequence mixer, ii) interleaving Hyena and attention layers, iii) using a sparse MoE channel mixer instead of SwiGLU, and iv) integrating a sparse routing mechanism into the Hyena sequence mixer (Fig. 3.7). Using the results of our scaling analysis (Sec. 4), we can investigate the correlation between the MAD scores of these architectures, as indicated by their average accuracy across the synthetic tasks, and their compute-optimal performance on The Pile (Fig. 4.1 left). We also consider perplexity on MAD tasks as an additional metric (Appendix B.5).

> **Finding** 9: Aggregate MAD scores are linearly correlated with compute-optimal perplexity at scale for all compute budgets (Fig. 4.1 left, Appendix B.8).

This result suggests that smaller, shallower models unit tested on MAD synthetics can be used to predict compute-optimal scaling, as well as to iterate on improvements to a base architecture. To better understand the contribution of each MAD task to the predictive power of the scores, we also report correlation for single-task performances and compute-optimal perplexity at scale (Fig. 4.1 right).

6. Conclusion

This work explores architecture optimization, from synthetic tasks designed to probe specific model capabilities to scaling laws. We introduce *mechanistic architecture design* (MAD), a methodology for fast prototyping and verification of new deep learning architectures based on key token manipulation tasks such as recall and compression. With MAD, we identify hybridization and new configurations to improve compute-optimal scaling of new architectures. We carry out an extensive scaling law analysis of new architectures, training over 500 models between parameter sizes of 70M to 7B, verifying the improvements found via MAD, and derive a collection of novel insights on the optimal scaling of new architectures. Finally, we show how MAD results are correlated with perplexity in a compute-optimal regime, paving the way for faster and cheaper architecture prototyping. Overall, this work provides evidence of correlation between scaling and a selection of synthetic token manipulation tasks, as well as of the existence of a variety of hybrid architectures improving over Transformers at scale.

Impact Statement

This paper introduces *mechanistic architecture design* (MAD), a methodology for improving the scaling performance of deep learning models, and presents several improved architectures. As a consequence of this line of work, we expect training and inference of large models to become more efficient, less expensive, and thus more readily available. Societal consequences related to the existence of large foundation models based on Transformers also apply when discussing new improved architectures.

Acknowledgments

We are grateful to the Hessian.AISC Service Center, funded by the Federal Ministry of Education and Research (BMBF), for the collaboration and joint use of their supercomputer forty-two.

References

[1] Akyürek, E., Wang, B., Kim, Y., and Andreas, J. In-context language learning: Architectures and algorithms. *arXiv preprint arXiv:2401.12973*, 2024.

[2] Arora, S., Eyuboglu, S., Timalsina, A., Johnson, I., Poli, M., Zou, J., Rudra, A., and Ré, C. Zoology: Measuring and improving recall in efficient language models. *arXiv preprint arXiv:2312.04927*, 2023.

[3] Bhattamishra, S., Patel, A., Blunsom, P., and Kanade, V. Understanding in-context learning in transformers and llms by learning to learn discrete functions. *arXiv preprint arXiv:2310.03016*, 2023.

[4] Bi, X., Chen, D., Chen, G., Chen, S., Dai, D., Deng, C., Ding, H., Dong, K., Du, Q., Fu, Z., et al. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*, 2024.

[5] Dupont, E., Doucet, A., and Teh, Y. W. Augmented neural odes. *Advances in neural information processing systems*, 32, 2019.

[6] Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., et al. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 1, 2021.

[7] Fathi, M., Pilault, J., Bacon, P.-L., Pal, C., Firat, O., and Goroshin, R. Block-state transformer. *arXiv preprint arXiv:2306.09539*, 2023.

[8] Fu, D. Y., Dao, T., Saab, K. K., Thomas, A. W., Rudra, A., and Ré, C. Hungry hungry hippos: Towards language modeling with state space models. *arXiv preprint arXiv:2212.14052*, 2022.

[9] Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[10] Geva, M., Bastings, J., Filippova, K., and Globerson, A. Dissecting recall of factual associations in auto-regressive language models. *arXiv preprint arXiv:2304.14767*, 2023.

[11] Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[12] Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

[13] Hewitt, J., Hahn, M., Ganguli, S., Liang, P., and Manning, C. D. Rnns can generate bounded hierarchical languages with optimal memory. *arXiv preprint arXiv:2010.07515*, 2020.

[14] Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[15] Hua, W., Dai, Z., Liu, H., and Le, Q. Transformer quality in linear time. In *International Conference on Machine Learning*, pp. 9099–9117. PMLR, 2022.

[16] Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[17] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[18] Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

[19] Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[20] Lin, Z., Feng, M., Santos, C. N. d., Yu, M., Xiang, B., Zhou, B., and Bengio, Y. A structured

self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.

[21] Ma, X., Zhou, C., Kong, X., He, J., Gui, L., Neubig, G., May, J., and Zettlemoyer, L. Mega: moving average equipped gated attention. *arXiv preprint arXiv:2209.10655*, 2022.

[22] Massaroli, S., Poli, M., Park, J., Yamashita, A., and Asama, H. Dissecting neural odes. *Advances in Neural Information Processing Systems*, 33:3952–3963, 2020.

[23] Massaroli, S., Poli, M., Fu, D. Y., Kumbong, H., Parnichkun, R. N., Timalsina, A., Romero, D. W., McIntyre, Q., Chen, B., Rudra, A., et al. Laughing hyena distillery: Extracting compact recurrences from convolutions. *arXiv preprint arXiv:2310.18780*, 2023.

[24] Nanda, N., Rajamanoharan, S., Kramár, J., and Shah, R. Fact finding: Attempting to reverse-engineer factual recall on the neuron level. *Alignment Forum*, 2023.

[25] Nguyen, E., Poli, M., Faizi, M., Thomas, A., Birch-Sykes, C., Wornow, M., Patel, A., Rabideau, C., Massaroli, S., Bengio, Y., et al. Hyenadna: Long-range genomic sequence modeling at single nucleotide resolution. *arXiv preprint arXiv:2306.15794*, 2023.

[26] Olsson, C., Elhage, N., Nanda, N., Joseph, N., DasSarma, N., Henighan, T., Mann, B., Askell, A., Bai, Y., Chen, A., et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

[27] Peng, B., Alcaide, E., Anthony, Q., Albalak, A., Arcadinho, S., Cao, H., Cheng, X., Chung, M., Grella, M., GV, K. K., et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

[28] Poli, M., Massaroli, S., Nguyen, E., Fu, D. Y., Dao, T., Baccus, S., Bengio, Y., Ermon, S., and Ré, C. Hyena hierarchy: Towards larger convolutional language models. *arXiv preprint arXiv:2302.10866*, 2023.

[29] Ren, L., Liu, Y., Wang, S., Xu, Y., Zhu, C., and Zhai, C. X. Sparse modular activation for efficient sequence modeling. *Advances in Neural Information Processing Systems*, 36, 2024.

[30] Sardana, N. and Frankle, J. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws. *arXiv preprint arXiv:2401.00448*, 2023.

[31] Schlag, I., Irie, K., and Schmidhuber, J. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pp. 9355–9366. PMLR, 2021.

[32] Shazeer, N. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[33] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[34] So, D. R., Mańke, W., Liu, H., Dai, Z., Shazeer, N., and Le, Q. V. Primer: Searching for efficient transformers for language modeling. *arXiv preprint arXiv:2109.08668*, 2021.

[35] Stanić, A., Ashley, D., Serikov, O., Kirsch, L., Faccio, F., Schmidhuber, J., Hofmann, T., and Schlag, I. The languini kitchen: Enabling language modelling research at different scales of compute. *arXiv preprint arXiv:2309.11197*, 2023.

[36] Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[37] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[38] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[39] Weiss, G., Goldberg, Y., and Yahav, E. On the practical computatifonal power of finite precision rnns for language recognition. *arXiv preprint arXiv:1805.04908*, 2018.

[40] White, C., Safari, M., Sukthanker, R., Ru, B., Elsken, T., Zela, A., Dey, D., and Hutter, F. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

[41] Yang, S., Wang, B., Shen, Y., Panda, R., and Kim, Y. Gated linear attention transformers with hardware-efficient training. *arXiv preprint arXiv:2312.06635*, 2023.

[42] Zhang, B. and Sennrich, R. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[43] Zhang, M., Saab, K. K., Poli, M., Dao, T., Goel, K., and Ré, C. Effectively modeling time series with simple discrete state spaces. *arXiv preprint arXiv:2303.09489*, 2023.

MECHANISTIC DESIGN AND SCALING OF HYBRID ARCHITECTURES

Supplementary Material

Contents

A. Additional Related Work

Synthetics for analysis and design The MAD framework builds on work on synthetic tasks for mechanistic interpretability of RNNs and Transformers, including associative recall, reasoning tasks, compression. [26] and a number of follow up in mechanistic interpretability use an induction task to probe into the internals of Transformer model. There is a large body of work [39, 13] studying the expressivity of recurrent models, either theoretically or empirically, using formal languages and other token manipulation tasks.

Smaller scale synthetics have been used during the iterative design procedure of new layers and primitives, particularly in the context of emerging deep signal processing architecture. [5, 22, 12, 8, 43, 28, 2]. Notably, [8] uses associative recall to identify a key capability gap in previous gated state-space models, and proposes a modification to the layer. [28] extend associative recall procedure to longer sequences, introducing new synthetic tasks such as counting. However, the pretraining results only involve smaller models, and are not obtained via compute-optimal scaling.

There exists a long line of work on neural architecture search methods (see [40] for a review). MAD provides a different approach based on synthetic tasks. MAD metrics are in principle compatible with various search methods.

Synthetics for evaluation Synthetics have also been leveraged to evaluate models and model classes [2, 3, 1]. [28] shows correlation between synthetics and pretraining results on The Pile. [2] maps associative recall accuracy gaps to a perplexity gap between pretrained models. A variety of other analyses on synthetics for emerging architectures finds certain classes of efficient architectures to be on par or outperform Transformers on most tasks, with gaps on tasks involving heavy recall or copying of tokens. With MAD, we aim to leverage tasks as unit tests with a quantitative connection to scaling properties, instead of using smaller-scale experiments to only build intuition on potential model differences.

Scaling laws We extend the compute-optimal scaling law analysis protocol of [17, 14] performed on Transformers to deep signal processing architectures, including hybrids and sparsely gated architectures. We base the scaling analysis in this work on the compute-optimal protocol, in order to evaluate relative performance and to identify optimal hybridization ratios. Moreover, we consider extensions such as state-optimal scaling and performance in overtrained regimes (outside the compute-optimal frontier), both of which have implications for efficient inference.

Other work on evaluation of new architectures experiments in parameter-matched and data-matched regimes, which can result in a mismatch with scaling results due to different FLOP costs per iteration. Other notable examples of compute-matched evaluations for new models are provided in [28, 11]. Previous evaluations are not carried out at compute-optimal model sizes which can vary significantly across architectures see e.g., Figures E.1 and E.6).

B. Mechanistic Architecture Design

B.1. Tasks

B.1.1. In-Context Recall

The in-context recall task is comprised of sequences of key-value pairs (with separate vocabularies for keys and values). Models are tasked with predicting all values for those keys that were already presented in the sequence:

input example: a b d e f g | a b f g

In this example, keys are drawn from the vocabulary {a, d, f} and values from the {b, e, g} vocabulary. Importantly, the mapping from keys to values is randomly shuffled between sequences. Models are tasked with autoregressively predicting all underlined value in this example.

In the baseline setting of this task, we use a vocabulary of 16 tokens and $12,800$ training sequences with a length of 128 tokens. The vocabulary is equally divided into keys and values.

B.1.2. Fuzzy In-Context Recall

The fuzzy in-context recall tasks adapts the in-context recall task by representing keys and values by a variable number of adjacent tokens:

input example: `(a d) (b) (d a f) (e g) | (d a f) (e g)`

In this example, keys are drawn from the vocabulary {a, d, f} and values are drawn from the vocabulary {b, e, g}. We use brackets for illustrative purposes to indicate adjacent tokens that together represent a key or value but they are not part of the actual input to the model. In sequential order, the presented keys are 'a d' and 'd a f', with associated values 'b' and 'e g'. For each sequence, keys and values are randomly drawn form the key and value dictionaries, with randomly drawn lengths (ranging from 1 to 3 tokens in our analyses). We always evaluate with keys of length 3 (the longest length used in our analyses), to disambiguate whenever a key token appears in two keys of different values. We pad sequences with a separate pad token if necessary to ensure that all sequences of a dataset are of the exact same length. As for the in-context recall task, models are tasked with autoregressively predicting all underlined values in this example.

In the baseline setting of this task, we use a vocabulary of 16 tokens and $12,800$ training sequences with a length of 128 tokens. The vocabulary is equally divided into key and value tokens.

B.1.3. Noisy In-Context Recall

The noisy in-context recall task represents another variation of in-context recall, in which noise tokens, from a separate vocabulary, are randomly inserted into the input sequences:

input example: `a b h d e f g | i a b f g`

In this example, keys and values are respectively drawn from the vocabularies {a, d, f} and {b, e, g}, while noise is drawn form the vocabulary {h, i}. As for in-context recall, models are tasked with autoregressively predicting the underlined values in this example.

In the baseline setting of this task, we use a vocabulary of 16 tokens, which are equally divided into keys and values, $12,800$ training sequences with a length of 128 tokens, and a share of 20% noise tokens in the input from a separate noise vocabulary of size 16.

B.1.4. Selective Copying

The selective copying task comprises sequences of randomly sampled tokens, with randomly inserted "blank" and "insert" tokens:

input example: `a c [b] t [b] [i] [i] [b] [i] | a c [b] t [b] a c [b] t`

In this example, tokens are drawn from the vocabulary {a,c,t}, while [b] and [i] indicate the blank and insert token. Given this example, the task of the model is to copy all non-special tokens to the positions of the insert tokens, in the order they were presented in the sequence. The purpose of the randomly inserted blank tokens is to force models to learn to selectively memorize or ignore information from the input.

In the baseline setting of this task, models are tasked with copying 16 randomly drawn tokens from a vocabulary of 16 tokens, and are provided with $12,800$ training sequences with a length of 256 tokens.

B.1.5. Compression

The compression task consists of random token sequences, each ending with a dedicated "compression token":

input example: `a e c b h g i [c] | [c] + [`pos_0`] -> a`

In this example, tokens are randomly drawn from the vocabulary {a, b, c, e, g, h, i}, while [c] indicates the compression token. Given this input, models are tasked with compressing all relevant sequence information into the compression token [c], such that a subsequent two-layer MLP can fully recover each token of the input sequence, given the model's output for the compression token. To indicate the position i that is to be recovered from the input, we add a non-learnable sin-cos position embedding (indicated by $[pos_i]$) to the models output for the compression token before feeding it to the MLP decoder.

In the baseline setting of this task, we use a vocabulary of 16 tokens and $12,800$ training sequences with a length of 32 tokens.

B.1.6. MEMORIZATION

The memorization task uses a fixed key-value dictionary, representing the facts to be learned:

key-value dictionary example: `{a:b, c:d, e:f}`

Input sequences comprise key-value pairs that are randomly sampled form this dictionary. Importantly, all values are masked out form the input sequences with a dedicated "insert token":

input example: `a [i] c [i] e [i] a [i] | a b c d e f a b`

In this example, the values that are to be inserted at the positions of the insert tokens are: 'b', 'd', 'f', and 'b'. Models are then tasked with correctly inserting the masked-out values at the positions of the insert tokens. As the values are never part of the input sequences, models need to learn the mapping from keys to values over the course of their training.

In the baseline setting of this task, we use a vocabulary of 256 tokens, equally divided into keys and values, and 256 training sequences with a length of 32 tokens (such that each fact is on average presented 32 times in the training data).

B.2. Manipulating Task Difficulty

For each MAD task, we evaluate model performances across several levels of difficulty. We manipulate task difficulty by i) increasing the length of the input sequences, ii) reducing the training dataset size, and iii) increasing the vocabulary size. In addition, we increase the share of noise in the inputs for the noisy in-context recall task as well as the number of tokens that are to be copied in the selective copying task. Importantly, we only change one task variable at a time, while keeping all others at their baseline level.

For all variants of in-context recall, we evaluate input sequence lengths of 128, 256, 512, and 1024 tokens, training dataset sizes with $12,800$, $6,400$, $3,200$, $1,600$ and 800 samples, and vocabulary sizes, which are equally divided into keys and values, of 16, 32, 64, and 128 tokens.

For noisy in-context recall, we additionally evaluate shares of 20%, 40%, 60%, and 80% noise tokens in the inputs.

For the selective copying task, we evaluate sequence lengths of 256, 512, and 1024 tokens, training dataset sizes with $12,800$, $6,400$, $3,200$, $1,600$ and 800 samples, vocabulary sizes of 16, 32, 64, and 128 tokens, and 16, 32, 64, and 96 tokens of a the input that are to be copied.

For the compression task, we evaluate input sequence lengths of 32, 64, 128 and 256 tokens, vocabulary sizes of 16, 32, 64, and 128 tokens, and training dataset sizes of $12,800$, $6,400$, $3,200$, $1,600$ and 800 samples.

For the memorization task, we evaluate vocabulary sizes of 256, 512, $1,024$, $2,048$, $4,096$, and $8,192$ tokens, while keeping the training dataset fixed at 256 samples with an input length of 32 (thereby effectively varying the rate at which each fact appears in the training data, with average rates of 32, 16, 8, 4, 2, and 1).

B.3. Architectures

We build architectures from a set of common channel- and sequence-mixing layer primitives. Each architecture is composed of 2 blocks with a total of 4 layers. In general, blocks combine a sequence mixing layer with a subsequent channel mixing layer, with the exception of Mamba layers, which combine sequence and channel mixing into a single layer [11]. All layers are set to a width of 128 for our main analysis (if not stated otherwise), with all other architecture settings given below.

Common architecture primitives are composed of two identical blocks combining each sequence-mixing layer with each of the two channel-mixing layers. Striped hybrid architectures combine each unique block of the common architecture primitives with a second block composed of multi-headed attention and one of the two channel mixers.

B.3.1. CHANNEL-MIXING LAYERS

- **SwiGLU MLP [32]:** inner width: 512

- **Mixture of Experts MLP [19]:** number of experts: 8, expert width: 16, number of active experts: 2

B.3.2. SEQUENCE-MIXING LAYERS

We normalize the (fixed) state dimension of all sequence mixers, before running the MAD pipeline. Whenever possible, we prioritize keeping the shape of the layer fixed, over the state dimension (e.g., reducing state dimension before expansion factors, or reducting state dimension before number of heads).

- **Hyena [28]:** filter order: 2, short filter order: 3, filter featurization is implemented following [23].
- **Mamba [11]:** state dimension: 4, convolution dimension: 4, width expansion: 2, no bias for linear and convolution layers.
- **Multi-head Gated Linear Attention [41]:** number of heads: 8, head dimension: 16
- **Multi-Head Attention [38]:** number of heads: 16, head dimension: 8, no bias for linear layers
- **Multi-Head Hyena [23]:** number of heads: 16, state dimension of heads: 2, filter order: 2, short filter order: 3.
- **Hyena Experts:** number of experts: 8, expert width: 16, number of active experts: 2. All other parameters are shared with standard Hyena.

At these settings, all evaluated architectures that do not include attention layers are normalized to a total state dimension of $4,096$.

B.4. Training

For each MAD task, we train models according to the setting described in Table B.1, using a standard cross-entropy loss objective. Note that we sweep all evaluated architectures over a 3×2 grid of learning rate and weight decay values (see Table B.1) and only include the best runs in our final analysis (as determined by their evaluation accuracy).

Table B.1: MAD training setting.

OPTIMIZER	ADAMW
OPTIMIZER MOMENTUM	$\beta_1, \beta_2 = 0.9, 0.98$
DROPOUT	NONE
BATCH SIZE	128
TRAINING EPOCHS	200
LEARNING RATE SCHEDULE	COSINE DECAY
NUMBER OF LAYERS	4
NUMBER OF EVALUATION SAMPLES	1,280
BASE LEARNING RATE	$[0.0001, 0.0005, 0.001]$
WEIGHT DECAY	$[0.0, 0.1]$

B.5. Results

B.5.1. TASK PERFORMANCES

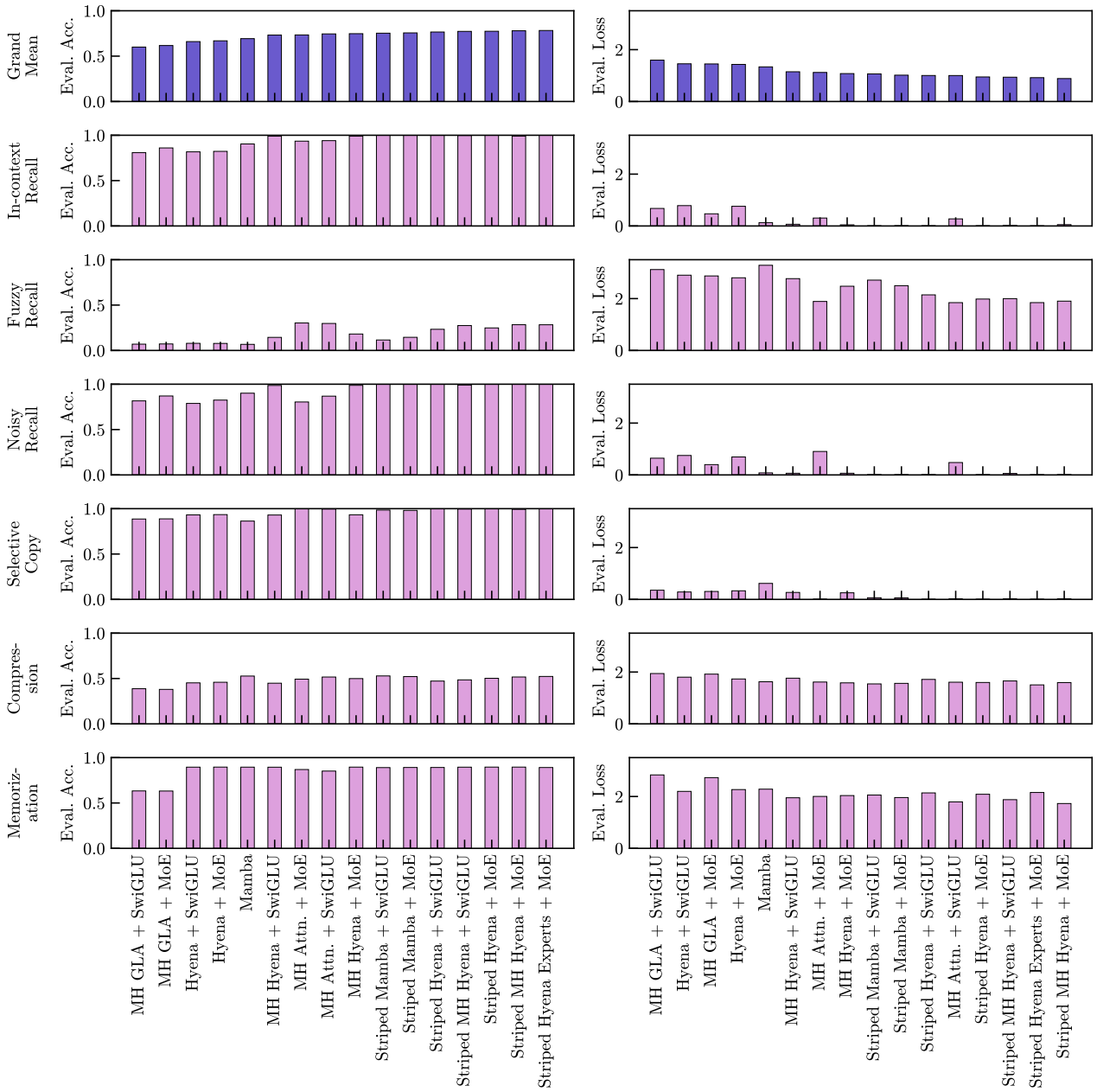

Figure B.1: Architecture performances within and across the MAD synthetic tasks, when using evaluation accuracy as a performance metric (left) or evaluation loss (right).

B.5.2. PERFORMANCE ON INDIVIDUAL TASKS

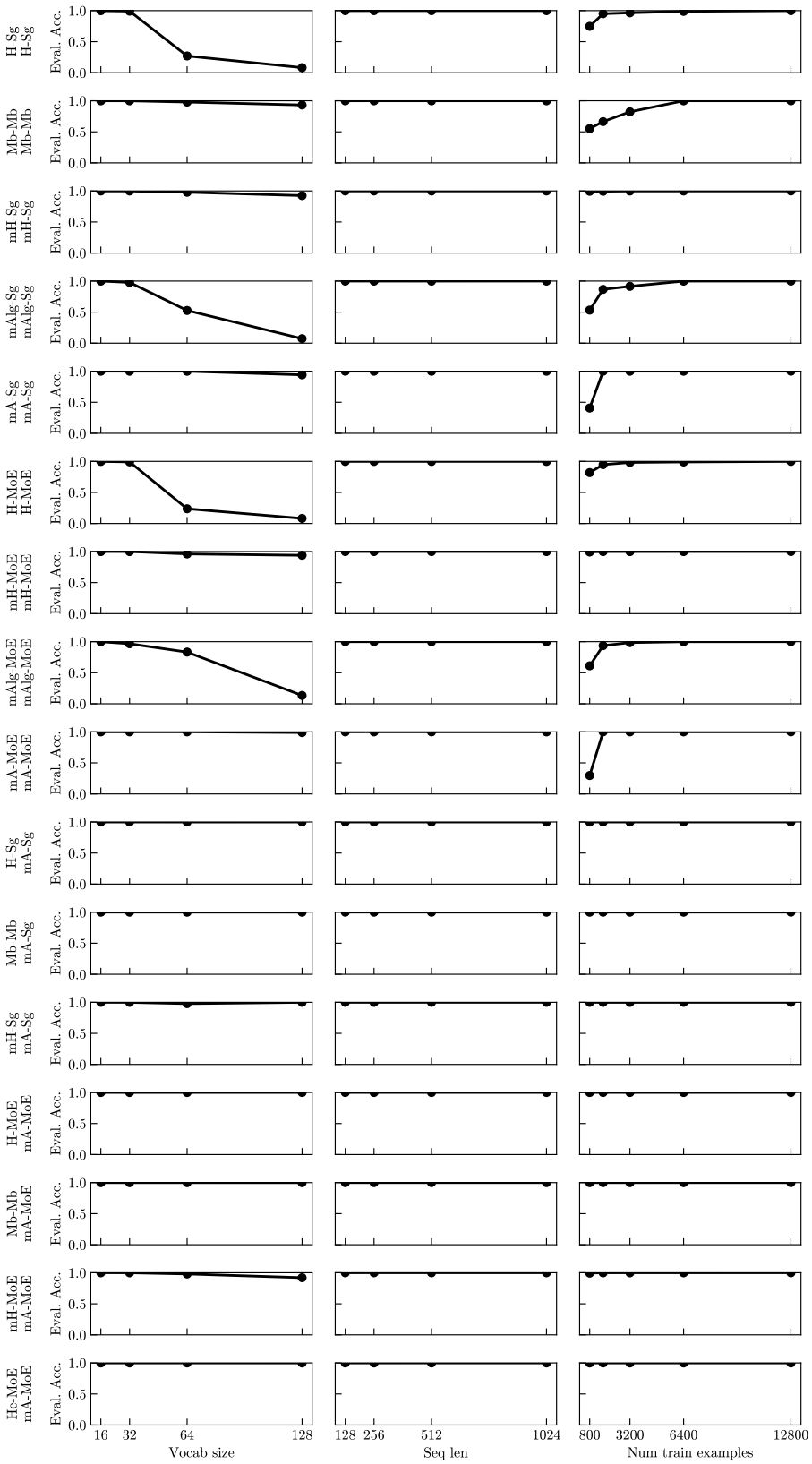

Figure B.2: **In-context recall task** model performances. `H`: Hyena, `Mb`: Mamba, `Alg`: Gated Lin. Attention, `A`: Attention, `He`: Hyena Experts, `Sg`: SwiGLU, `MoE`: Mixture of Experts MLP, `m{H,A,Alg}`: multi-headed model variants.

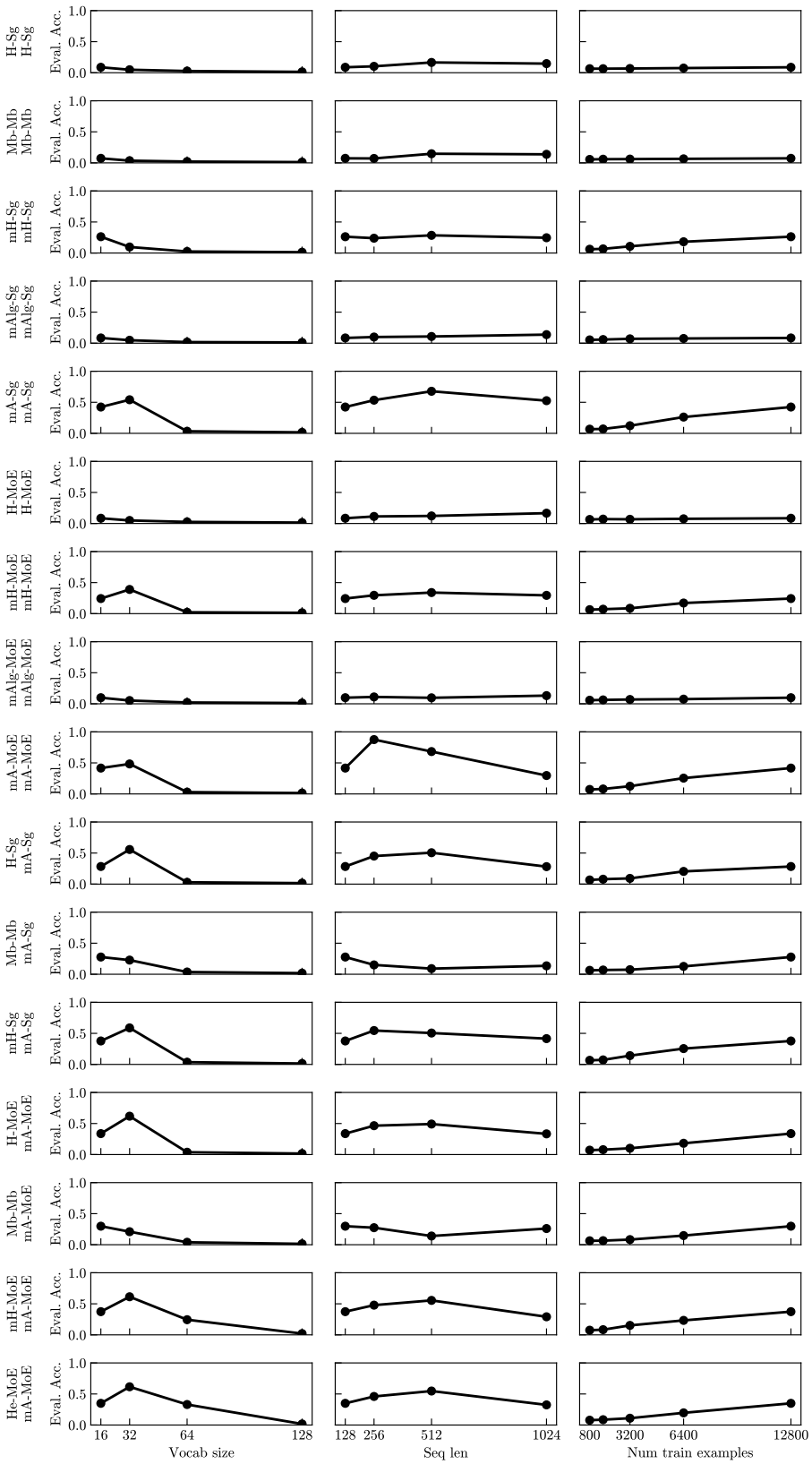

Figure B.3: **Fuzzy in-context recall task** model performances. `H`: Hyena, `Mb`: Mamba, `Alg`: Gated Lin. Attention, `A`: Attention, `He`: Hyena Experts, `Sg`: SwiGLU, `MoE`: Mixture of Experts MLP, `m{H,A,Alg}`: multi-headed model variants.

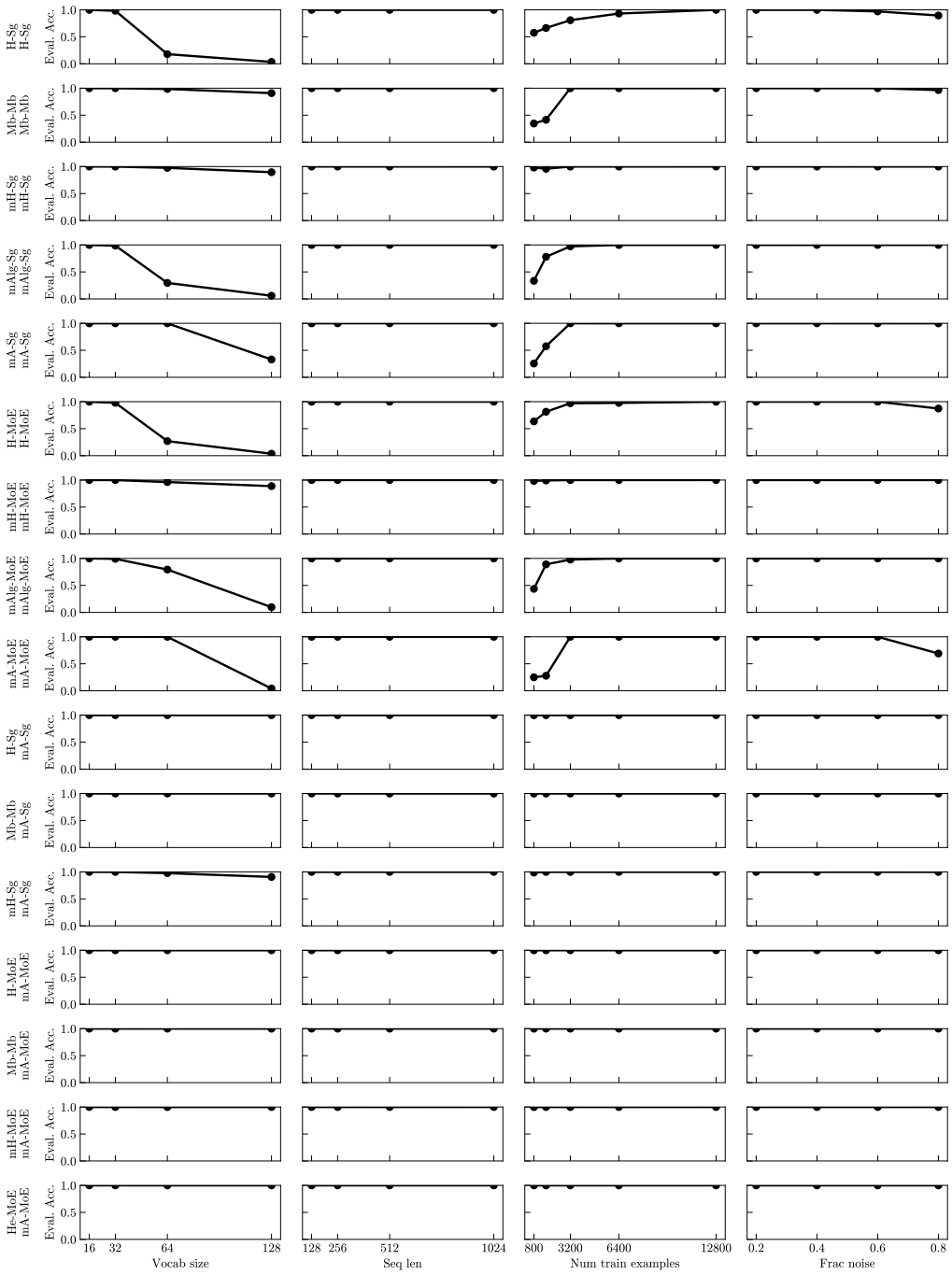

Figure B.4: **Noisy in-context recall task** model performances. `H`: Hyena, `Mb`: Mamba, `Alg`: Gated Lin. Attention, `A`: Attention, `He`: Hyena Experts, `Sg`: SwiGLU, `MoE`: Mixture of Experts MLP, `m{H,A,Alg}`: multi-headed model variants.

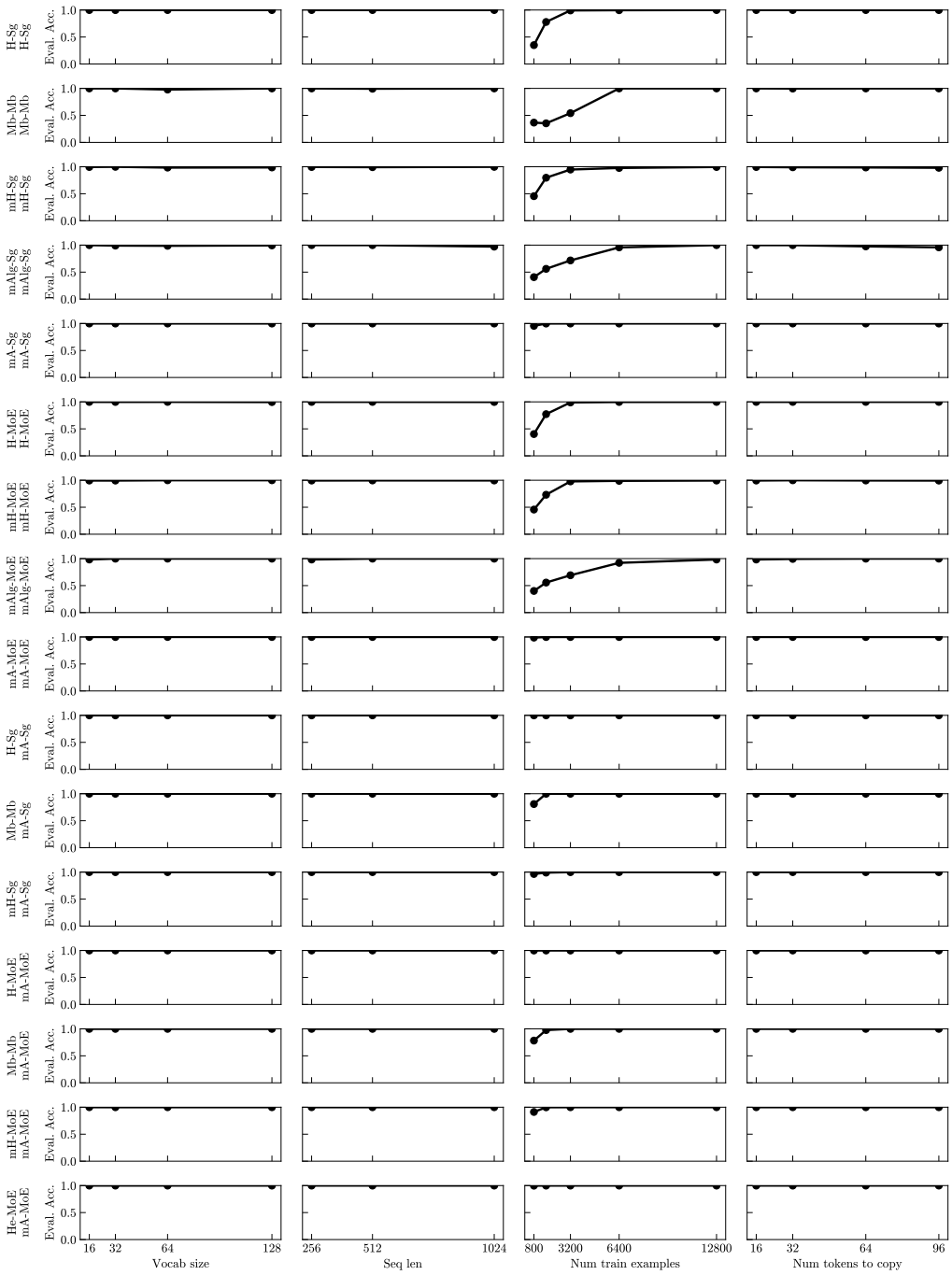

Figure B.5: **Selective Copying** model performances. `H`: Hyena, `Mb`: Mamba, `Alg`: Gated Lin. Attention, `A`: Attention, `He`: Hyena Experts, `Sg`: SwiGLU, `MoE`: Mixture of Experts MLP, `m{H,A,Alg}`: multi-headed model variants.

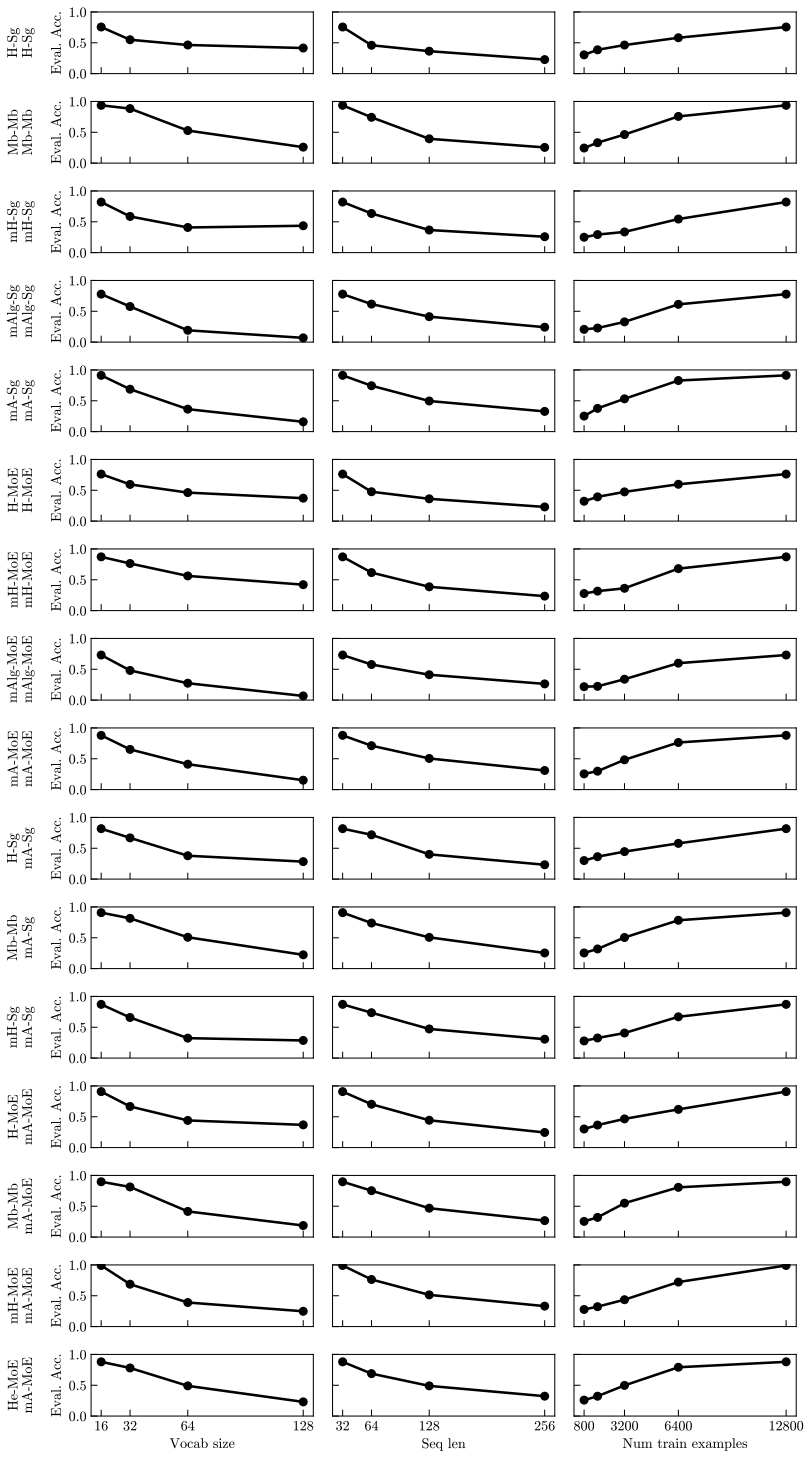

Figure B.6: **Compression** model performances. `H`: Hyena, `Mb`: Mamba, `Alg`: Gated Lin. Attention, `A`: Attention, `He`: Hyena Experts, `Sg`: SwiGLU, `MoE`: Mixture of Experts MLP, `m{H,A,Alg}`: multi-headed model variants.

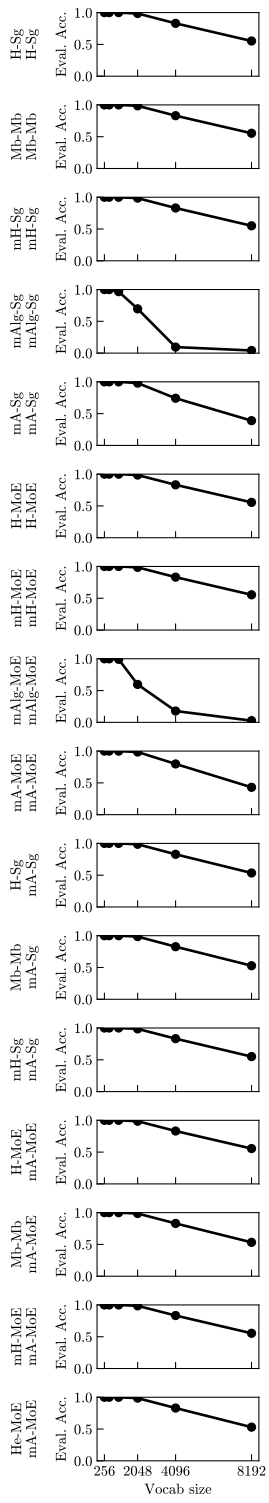

Figure B.7: **Memorization** model performances. `H`: Hyena, `Mb`: Mamba, `Alg`: Gated Lin. Attention, `A`: Attention, `He`: Hyena Experts, `Sg`: SwiGLU, `MoE`: Mixture of Experts MLP, `m{H,A,Alg}`: multi-headed model variants.

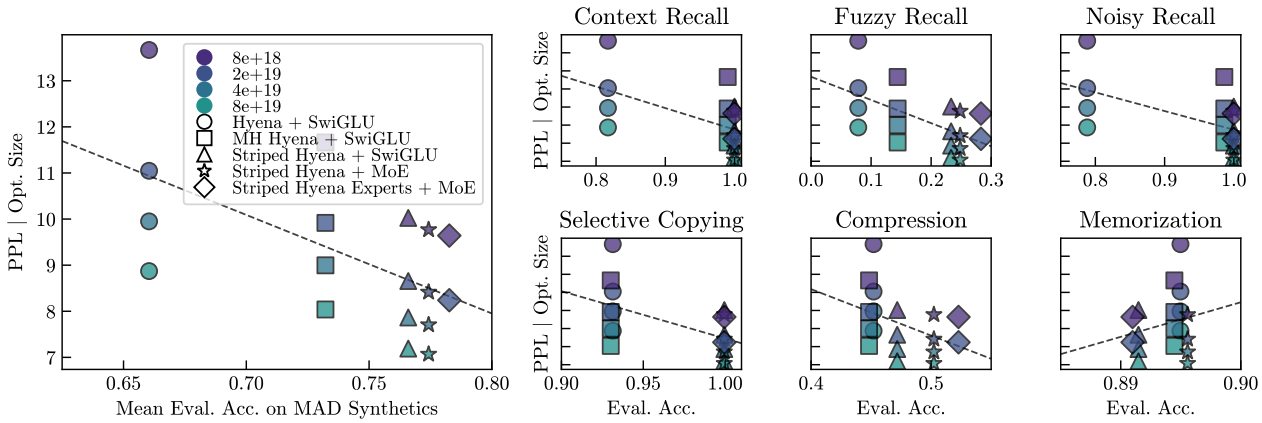

Figure B.8: Improved performance on MAD synthetics correlates with better compute-optimal perplexity on The Pile across IsoFLOP groups. We highlight progressively improved versions of Hyena that were designed with the MAD pipeline.

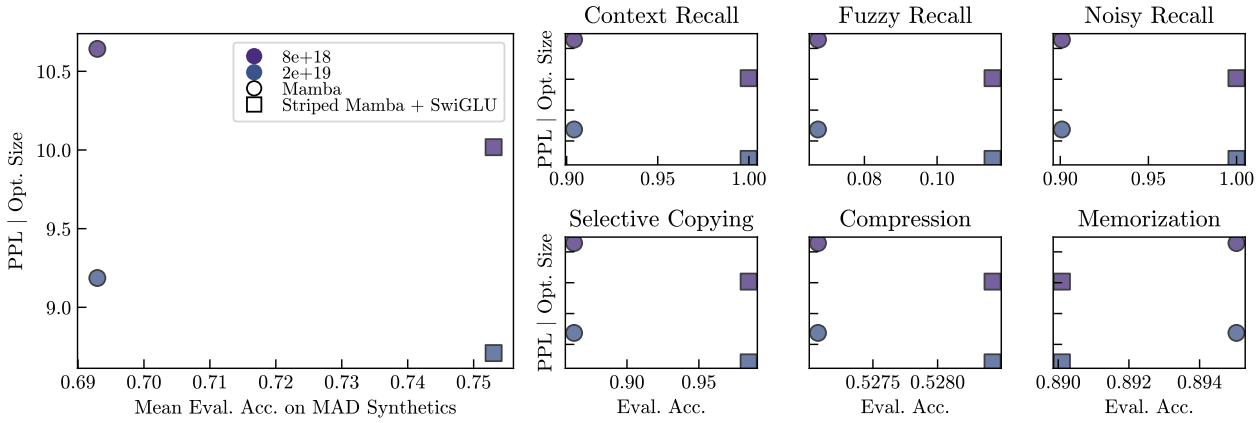

Figure B.9: Replication of Fig. B.8 for the Mamba and Striped Mamba architectures and IsoFLOP groups 8e18 and 2e19.

B.6. Extensions and Limitations of MAD

The MAD evaluation framework relies on extrapolating performance from smaller (e.g., 2-block) models to deeper models trained at scale. As such, the framework has not yet been applied to sophisticated topologies requiring small-scale testing with a larger number of blocks e.g., hybrid models with more than two sequence mixer primitives, or alternative interconnection topologies that span multiple layers.

In principle, MAD can be used to design architectures to optimize other quantities of interest, beyond perplexity or downstream benchmarks e.g., throughput. In this work, we focus on investigating correlation with compute-optimal scaling metrics, and leave other analyses to future work.

C. Scaling Laws

We design our model topologies starting from previous compute-optimal scaling results for Transformers [30], and selecting the number of layers (depth) and width to cover a range of parameters from $8e6$ to $7e9$ parameters (see Table D.2). The depth and width are generally fixed across models, which result in minor parameter count differences (except for the mixture of experts models where a distinction between total and active parameters must be made, see Tables D.4 and D.3). To compare how each model scales, we control for several compute budgets (IsoFLOP groups): 4e18, 8e18, 2e19, 4e19, 8e19, 2e20, 5e20, 2e21. We linearly interpolate learning rates from common settings at $150e6, 350e6, 1.3e9, 3e9$ and $7e9$ model sizes, obtaining a linearly inverse relationship with model size. Batch size is scaled (increased) in discrete steps, with larger training FLOPs using larger batch sizes.

For state-optimal scaling results, we obtain the optimal model size from the compute-optimal frontier, then compute the dynamic and fixed state dimensions of the closest model size available in the set of results.

C.1. Training Details

We control for key hyperparameters across all models, including batch size (Table D.1), learning rate (Table D.2) and scheduler. Most models were trained on a single node. For larger IsoFLOP groups, we trained in a multinode distributed training with tensor parallelism. We used a cosine decay learning rate scheduler, with warm up using 1% the number of training steps, and the minimum decay to reach 10% of the max learning rate.

Table C.1: Batch sizes by IsoFLOP group. For very small models (<54M) parameters, batch size 262k is used.

IsoFLOP	Batch Size
4.0E+18	524k
8.0E+18	524k
2.0E+19	524k
4.0E+19	524k
8.0E+19	524k
2.0E+20	1M
5.0E+20	1M
2.0E+21	2M

Batch sizes and hyperparameters Batch size and learning rate are two high-impact hyperparameters for scaling laws, as they visibly shift the compute-efficient frontier. We find that scaling the batch size with FLOP budgets, thus keeping it fixed within each IsoFLOP group, to be a simple and robust approach. Fig. D.1 provides an example of potential issues arising from incorrect batch scaling. These results are in line with recent findings [4].

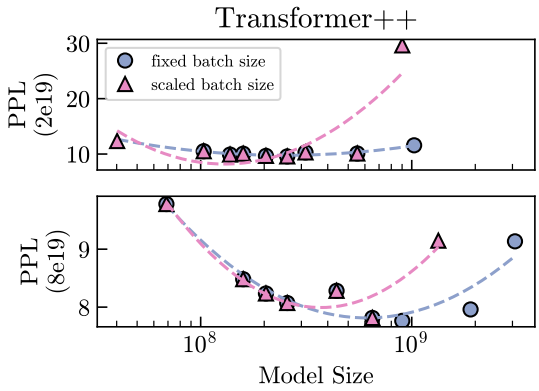

Figure C.1: Increasing batch size with compute FLOPS can shift the compute-efficient frontier. When increasing batch size after 10^9 parameters (red), the IsoFLOP curve underestimates the performance of larger models, when compared to a fixed batch size (blue), shifting the optimum estimation towards smaller models.

C.2. Model architectures

We describe shared architecture details first, followed by model specific designs below. All models use a modern SwiGLU unit as the channel mixer, except for Mamba and StripedMamba (which merges the GLU block with the sequence mixer layer, resulting in twice the number of sequence mixers). We use RMSNorm [42] for normalization. All models tie the embedding layers. All sparsely activated layers use learned argmax routing.

Transformer++ Transformer++ is state-of-the-art Transformer model, with rotary positional embeddings [36], SwiGLU and RMSNorm.

Hyena We use the original architecture [28] with some improvements. The channel mixer is replaced with SwiGLU, we use RMSNorm, set weight decay to 0 to all Hyena layer parameters.

Multi-Head Hyena We use a Hyena layer with heads as described by [23]. We sweep across different head dimensions at the IsoFLOP group 2e19 to find an optimal head dimension (8), and use the same number for all other experiments.

StripedHyena We use 3 striping schedule ratios: 1`A`:1`H`, 1`A`:3`H`, 1`A`:11`H`, where `A`=Attention and `H`=Hyena along model depth. In instances where the number of layers is not a multiple of the schedule, the ratio is repeated until the target depth is reached.

Mamba Mamba doubles the number of sequence mixers, replacing the dedicated channel mixer, and uses a custom input-varying recurrence. Hyperparameters (state dimension 16, expansion factor 2, conv projection length 4 and width of implicit network are sourced from the original implementation [11])

StripedMamba Similar to StripedHyena, we use the 3 striping ratio schedules to interleave attention at specified intervals along model depth.

StripedHyena-MoE The StripedHyena-MoE replaces SwiGLU with a total of 8 experts and 2 active experts. We keep the same depth and model width in the mixer layer as baseline models, and adjust the MoE widths to match active parameters.

StripedHyena Experts-MoE This model introduces expert in the Hyena sequence mixer at the output level, as described in the main text. We use a StripedHyena with striping ratio 1:11, and the following expert counts: total experts = 8, active experts = 2, total mixer experts = 8, active mixer experts = 2.

C.3. Model sizes and training hyperparameters

We show common model settings across all architectures by size in Table D.2. We use Adam optimizer betas [0.9, 0.95], weight decay 0.1, and no dropout. All models are trained in mixed precision: `bfloat16` with full precision for Hyena and Mamba convolution and recurrences.

C.4. FLOP calculation

We provide FLOP calculators for each model architecture explored in this study. Notation is provided in D.5.

C.4.1. TRANSFORMER++

- Embedding layers: $4LDV$
- MHA
  - **projections**: $6LD^2$
  - **attention**: $4L^2D + 2HL^2$
  - **out layer**: $2LD^2$
- GLU
  - $6LDD_{\texttt{glu}}$

C.4.2. HYENA

GLU and embedding calculation is the same as Transformer++.

- Sequence Mixer
  - **projections**: $6LD^2$
  - **convs on projections**: $18LD$
  - **featurization**: $S_{\texttt{hyena}}LD$[8]
  - **convolution and gates**: $10L\log_2(L)D + 4LD$
  - **out layer**: $2LD^2$

C.4.3. MULTI-HEAD HYENA

- Sequence Mixer
  - **projections**: $6LD^2$
  - **convs on projections**: $18LD$
  - **featurization**: $S_{\texttt{hyena}}LH$
  - **convolution and gates**: $10L\log_2(L)D^2/H + 4LD^2/H$
  - **out layer**: $2LD^2$

C.4.4. STRIPEDHYENA

FLOPS of StripedHyena are determined by summing the FLOPS of Hyena-GLU and MHA-GLU, with the mixing ratios specified by the particular instance of the model.

C.4.5. MAMBA

- Sequence Mixer
  - **projections**: $4LD^2E$
  - **short conv**: $6LDE$
  - **featurization**: $2LDE(D_{\texttt{dt}} + 2S_{\texttt{mamba}}) + 2LDED_{\texttt{dt}}$
  - **associative scan**: $2LDES_{\texttt{mamba}}$[9]
  - **out layer**: $2LD^2E$
- No separate GLU block (2x the sequence mixers).

C.4.6. STRIPEDMAMBA

FLOPS of StripedMamba are determined by summing the FLOPS of Mamba-Mamba and MHA-GLU, with the mixing ratios specified by the particular instance of the model.

C.4.7. STRIPEDHYENA-MOE

- Sequence mixer
  - Same as StripedHyena
- SwiGLU MoE (replaces MLP block)
  - **router**: $LDA_{\texttt{moe}}$
  - **up projections** $4DD_{\texttt{moe}}A_{\texttt{moe}}$
  - **down projection (sparse)** $2DD_{\texttt{moe}}G_{\texttt{moe}}$

[8] Other filter parametrizations e.g., canonical via rational functions, scale with the order $\mathcal{O}(S_{\texttt{hyena}}DL\log_2(L))$.

[9] Estimate assumes "most efficient" scan algorithm in terms of FLOPS (but not latency). In practice, the constant may be larger.

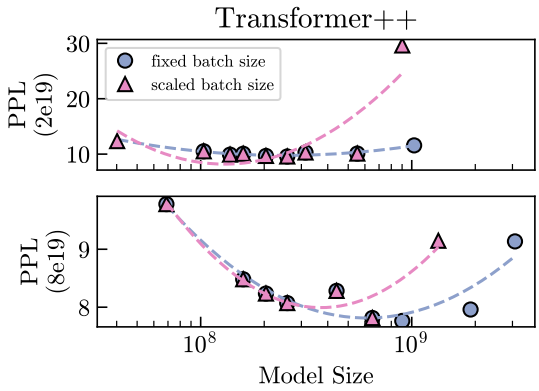

Figure D.1: Increasing batch size with compute FLOPS can shift the compute-efficient frontier. When increasing batch size after 10^9 parameters (red), the IsoFLOP curve underestimates the performance of larger models, when compared to a fixed batch size (blue), shifting the optimum estimation towards smaller models.

C.4.8. StripedHyena Experts + MoE

Model has experts in both sequence mixers (Hyena) and GLU layers. In attention layers, Transformer++ sequence mixer (MHA) FLOPS are used. The idea of Hyena experts is to select via a router (softmax - argmax selection) $G_{\texttt{moh}}$ smaller Hyena experts, and run computation only on those dimensions. Equivalently, this can be seen as adaptively choosing a subset of states, using the input sequence.

- Hyena experts
  - **router**: $LDA_{\texttt{moh}}$
  - **projections**: $6LD^2$
  - **convs on projections**: $18LD$
  - **featurization**: $S_{\texttt{hyena}}LD_{\texttt{moh}}G_{\texttt{moh}}$
  - **convolution and gates**: $10L\log_2(L)D_{\texttt{moh}}G_{\texttt{moh}} + 4LD_{\texttt{moh}}G_{\texttt{moh}}$
  - **out layer**: $2LD_{\texttt{moh}}D$

D. Scaling Laws

We design our model topologies starting from previous compute-optimal scaling results for Transformers [30], and selecting the number of layers (depth) and width to cover a range of parameters from $8e6$ to $7e9$ parameters (see Table D.2). The depth and width are generally fixed across models, which result in minor parameter count differences (except for the mixture of experts models where a distinction between total and active parameters must be made, see Tables D.4 and D.3). To compare how each model scales, we control for several compute budgets (IsoFLOP groups): 4e18, 8e18, 2e19, 4e19, 8e19, 2e20, 5e20, 2e21. We linearly interpolate learning rates from common settings at $150e6, 350e6, 1.3e9, 3e9$ and $7e9$ model sizes, obtaining a linearly inverse relationship with model size. Batch size is scaled (increased) in discrete steps, with larger training FLOPs using larger batch sizes.

For state-optimal scaling results, we obtain the optimal model size from the compute-optimal frontier, then compute the dynamic and fixed state dimensions of the closest model size available in the set of results.

D.1. Training Details

We control for key hyperparameters across all models, including batch size (Table D.1), learning rate (Table D.2) and scheduler. Most models were trained on a single node. For larger IsoFLOP groups, we trained in a multinode distributed training with tensor parallelism. We used a cosine decay learning rate scheduler, with warm up using 1% the number of training steps, and the minimum decay to reach 10% of the max learning rate.

D.2. Model architectures

We describe shared architecture details first, followed by model specific designs below. All models use a modern SwiGLU unit as the channel mixer, except for Mamba and StripedMamba (which merges the GLU block with the sequence mixer layer, resulting in twice the number of sequence mixers). We use RMSNorm [42] for normalization. All models tie the embedding layers. All sparsely activated layers use learned argmax routing.

Transformer++ Transformer++ is state-of-the-art Transformer model, with rotary positional embeddings [36], SwiGLU and RMSNorm.

Hyena We use the original architecture [28] with some improvements. The channel mixer is replaced with SwiGLU, we use RMSNorm, set weight decay to 0 to all Hyena layer parameters.

Multi-Head Hyena We use a Hyena layer with heads as described by [23]. We sweep across different head dimensions at the IsoFLOP group 2e19 to find an optimal head dimension (8), and use the same number for all other experiments.

StripedHyena We use 3 striping schedule ratios: 1`A`:1`H`, 1`A`:3`H`, 1`A`:11`H`, where `A`=Attention and `H`=Hyena along model depth. In instances where the number of layers is not a multiple of the schedule, the ratio is repeated until the target depth is reached.

Mamba Mamba doubles the number of sequence mixers, replacing the dedicated channel mixer, and uses a custom input-varying recurrence. Hyperparameters (state dimension 16, expansion factor 2, conv projection length 4 and width of implicit network are sourced from the original implementation [11])

StripedMamba Similar to StripedHyena, we use the 3 striping ratio schedules to interleave attention at specified intervals along model depth.

StripedHyena-MoE The StripedHyena-MoE replaces SwiGLU with a total of 8 experts and 2 active experts. We keep the same depth and model width in the mixer layer as baseline models, and adjust the MoE widths to match active parameters.

StripedHyena Experts-MoE This model introduces expert in the Hyena sequence mixer at the output level, as described in the main text. We use a StripedHyena with striping ratio 1:11, and the following expert counts: total experts = 8, active experts = 2, total mixer experts = 8, active mixer experts = 2.

D.3. Model sizes and training hyperparameters

We show common model settings across all architectures by size in Table D.2. We use Adam optimizer betas [0.9, 0.95], weight decay 0.1, and no dropout. All models are trained in mixed precision: `bfloat16` with full precision for Hyena and Mamba convolution and recurrences.

D.4. FLOP calculation

We provide FLOP calculators for each model architecture explored in this study. Notation is provided in D.5.

D.4.1. TRANSFORMER++

- Embedding layers: $4LDV$
- MHA
  - **projections**: $6LD^2$
  - **attention**: $4L^2D + 2HL^2$
  - **out layer**: $2LD^2$
- GLU
  - $6LDD_{\texttt{glu}}$

D.4.2. HYENA

GLU and embedding calculation is the same as Transformer++.

- Sequence Mixer
  - **projections**: $6LD^2$
  - **convs on projections**: $18LD$
  - **featurization**: $S_{\texttt{hyena}}LD$[10]
  - **convolution and gates**: $10L\log_2(L)D + 4LD$
  - **out layer**: $2LD^2$

D.4.3. MULTI-HEAD HYENA

- Sequence Mixer
  - **projections**: $6LD^2$
  - **convs on projections**: $18LD$
  - **featurization**: $S_{\texttt{hyena}}LH$
  - **convolution and gates**: $10L\log_2(L)D^2/H + 4LD^2/H$
  - **out layer**: $2LD^2$

D.4.4. STRIPEDHYENA

FLOPS of StripedHyena are determined by summing the FLOPS of Hyena-GLU and MHA-GLU, with the mixing ratios specified by the particular instance of the model.

D.4.5. MAMBA

- Sequence Mixer
  - **projections**: $4LD^2E$
  - **short conv**: $6LDE$
  - **featurization**: $2LDE(D_{\texttt{dt}} + 2S_{\texttt{mamba}}) + 2LDED_{\texttt{dt}}$
  - **associative scan**: $2LDES_{\texttt{mamba}}$[11]
  - **out layer**: $2LD^2E$
- No separate GLU block (2x the sequence mixers).

D.4.6. STRIPEDMAMBA

FLOPS of StripedMamba are determined by summing the FLOPS of Mamba-Mamba and MHA-GLU, with the mixing ratios specified by the particular instance of the model.

D.4.7. STRIPEDHYENA-MOE

- Sequence mixer
  - Same as StripedHyena
- SwiGLU MoE (replaces MLP block)
  - **router**: $LDA_{\texttt{moe}}$
  - **up projections** $4DD_{\texttt{moe}}A_{\texttt{moe}}$
  - **down projection (sparse)** $2DD_{\texttt{moe}}G_{\texttt{moe}}$

[10] Other filter parametrizations e.g., canonical via rational functions, scale with the order $\mathcal{O}(S_{\texttt{hyena}}DL\log_2(L))$.

[11] Estimate assumes "most efficient" scan algorithm in terms of FLOPS (but not latency). In practice, the constant may be larger.

D.4.8. STRIPEDHYENA EXPERTS + MOE

Model has experts in both sequence mixers (Hyena) and GLU layers. In attention layers, Transformer++ sequence mixer (MHA) FLOPS are used. The idea of Hyena experts is to select via a router (softmax - argmax selection) $G_{\texttt{moh}}$ smaller Hyena experts, and run computation only on those dimensions. Equivalently, this can be seen as adaptively choosing a subset of states, using the input sequence.

- Hyena experts
  - **router**: $LDA_{\texttt{moh}}$
  - **projections**: $6LD^2$
  - **convs on projections**: $18LD$
  - **featurization**: $S_{\texttt{hyena}}LD_{\texttt{moh}}G_{\texttt{moh}}$
  - **convolution and gates**: $10L\log_2(L)D_{\texttt{moh}}G_{\texttt{moh}} + 4LD_{\texttt{moh}}G_{\texttt{moh}}$
  - **out layer**: $2LD_{\texttt{moh}}D$

D.5. State-optimal scaling

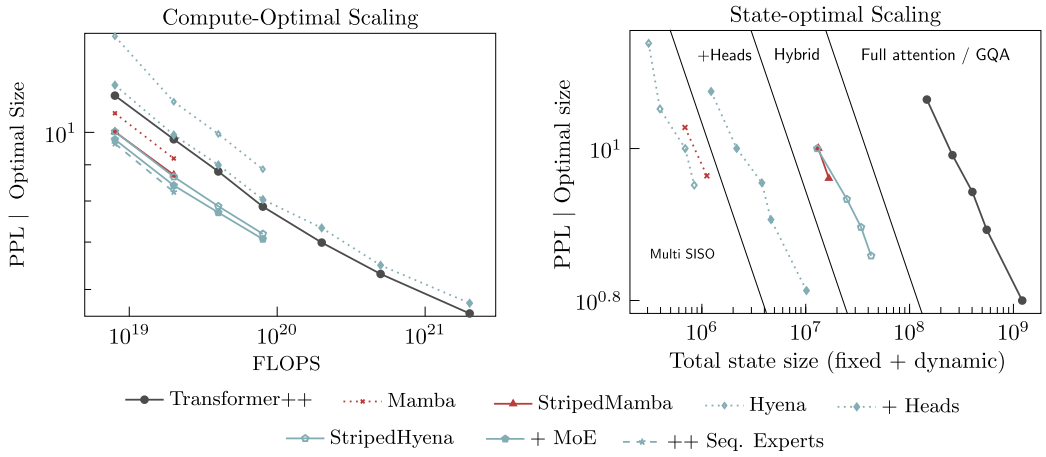

Figure D.2: Compute-optimal and state-optimal scaling on The Pile. We report total state dimension, fixed (recurrences) and dynamic (attention). All models are trained at sequence length 8k. We identify distinct regions in the state-optimal frontier, indicating that one may pay an additional FLOP cost to obtain the same perplexity with a state of smaller dimension, by using other classes of architectures.

E. Extended Scaling Results

E.1. Optimal hybridization topologies

We observe the topology of hybrid architectures to have significant effect on their downstream performance. In MAD tests, interleaving schedules for StripedHyena, with gated convolution followed by attention, outperform schedules attention followed by gated convolution.

Table E.1 provides ablations on the perplexity at larger scales. A variety of topologies achieve best perplexity, including chunked interleaving (6H:6A) and an *encoder-decoder* topology (6H:12A:6H), where Hyena layers surround a block of attention layers.

For all other experiments in the paper, including scaling laws, we adopt a simple 1H:1A topology for simplicity, as that is already seen to outperform other architectures in compute-optimal and state-optimal scaling.

E.2. Byte-level scaling laws

Scaling laws analysis primarily focus on sub-word level tokenization. With a new range of architectural options, we also explore compute-optimal scaling of a subset of architectures (Transformer++, Mamba, Hyena and StripedHyena) at byte

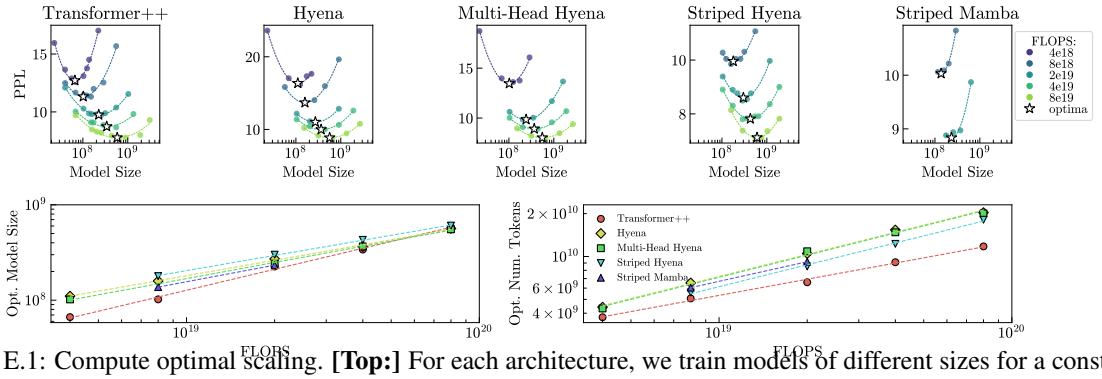

Figure E.1: Compute optimal scaling. **[Top:]** For each architecture, we train models of different sizes for a constant number of FLOPs (so-called IsoFLOP groups). For each of these IsoFLOP groups, we determine an optimum model size based on a polynomial fit to the observed training perplexities. **[Bottom:]** Using these estimates, we predict optimal model sizes and number of training tokens for each architecture.

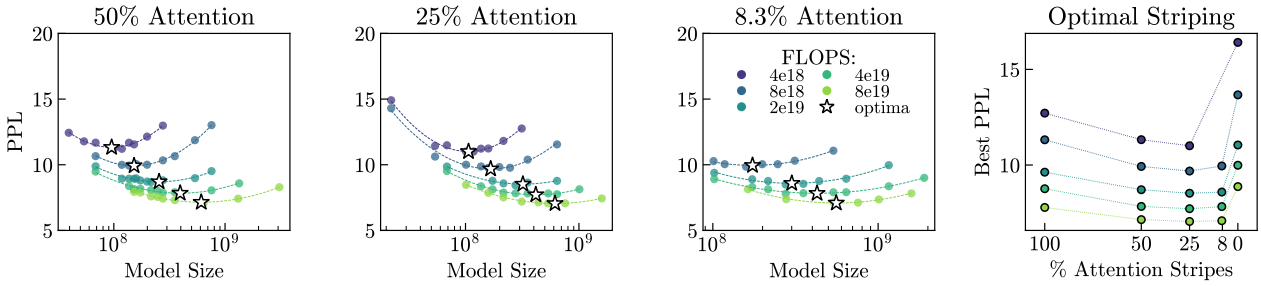

Figure E.2: Optimal striping ratio. We find that StripedHyena architectures outperform non-striped Hyena (0% Attention) and Transformer++ (100% Attention) architectures across all evaluated FLOPS groups. In particular, we find a ratio of 25% to be optimal.

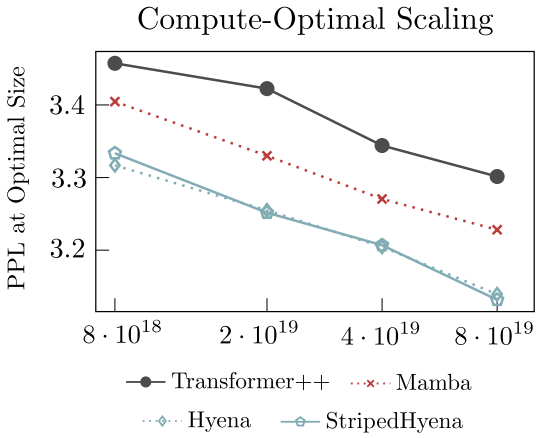

Figure E.3: Compute-optimal scaling at byte resolution.

resolution. We scale the models across FLOP budgets from 8e18 to 8e19 with model sizes from 6M to 1B parameters. The compute-optimal frontier is obtained using a similar protocol as outlined in Sec. D.

We find attention-based models to yield significantly higher perplexity at all IsoFLOP groups, with alternative architectures outperforming Transformer++, including non-striped variants (Figure E.3). These results show that model ranking varies significantly across domains and tokenization strategies.

Additional results We report additional results for scaling laws on DNA sequences. We trained all models on 8k sequence length, using model hyperparameters detailed in D.2. The model rankings are different from subword tokenized language data. We also compare architecture performance outside the compute-optimal frontier, namely with allocations of the computational budget are suboptimal but common in practice, such as overtraining smaller models E.5.

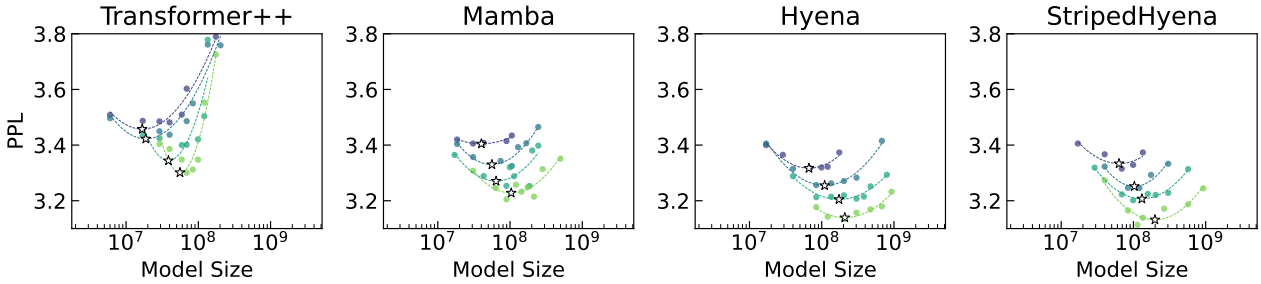

Figure E.4: Pretraining compute-optimal scaling on DNA sequences, with byte-level tokenization (nucleotide resolution).

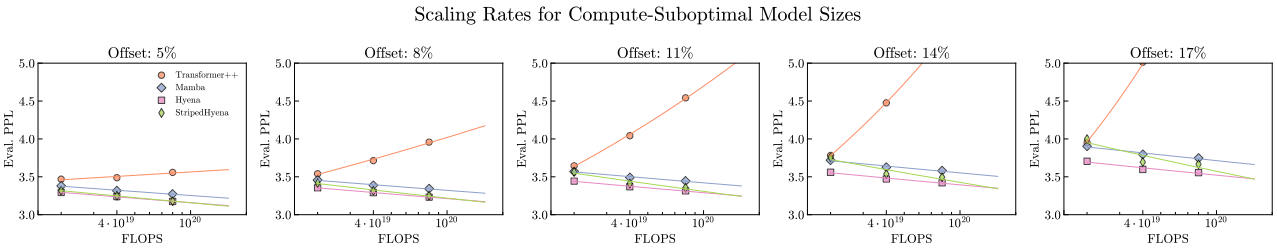

Figure E.5: Scaling off the compute-optimal frontier on DNA data. We verify the perplexity scaling at model sizes with a percentage offset from the optimal model size at each FLOP budget. In particular, we train a % offset smaller model, for longer. Transformers do not scale well to the overtraining regime.

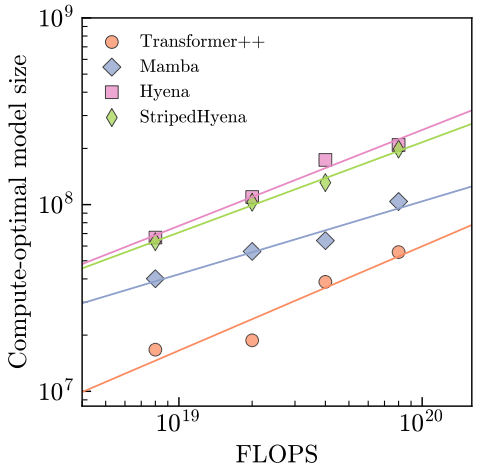

Figure E.6: Optimal model size vs FLOPS.

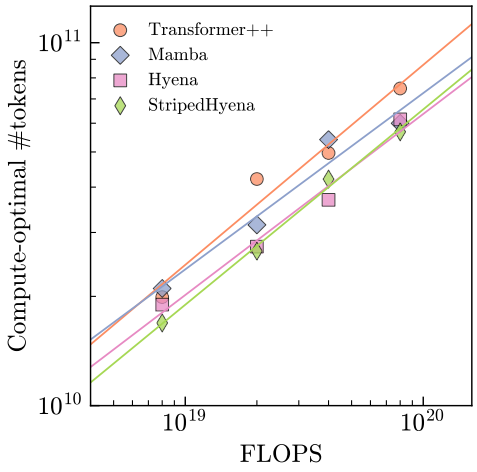

Figure E.7: Optimal number of tokens vs FLOPS.

Figure E.8: Comparison of optimal model size and number of tokens for each FLOP budget.

Table C.2: Common settings across all architectures. For Mamba, we use the layer structure of `Mb-Mb` following [11]. Actual parameter counts vary slightly for each architecture..

PARAMS (M)	D_MODEL	FFW_SIZE	KV_SIZE	N_HEADS	N_LAYERS	LEARNING RATE
8	128	336	64	2	4	9.77E-04
22	320	848	64	5	5	9.57E-04
38	448	1200	64	7	7	9.36E-04
54	512	1360	64	8	9	9.15E-04
70	576	1536	64	8	10	8.95E-04
102	640	1712	64	10	14	8.56E-04
118	704	1872	64	11	14	8.37E-04
134	768	2048	64	12	14	8.18E-04
150	768	2048	64	12	16	8.00E-04
163	768	2048	64	12	17	7.75E-04
175	768	2048	64	12	19	7.50E-04
196	832	2224	64	13	19	7.25E-04
217	832	2224	64	13	21	7.00E-04
251	896	2384	64	14	21	6.75E-04
278	896	2384	64	14	24	6.50E-04
306	960	2560	64	15	24	6.25E-04
350	1024	2736	64	16	24	6.00E-04
440	1152	3072	64	18	24	5.66E-04
536	1280	3408	64	20	24	5.33E-04
641	1408	3760	128	11	24	5.00E-04
756	1536	4096	128	12	24	4.75E-04
881	1664	4432	128	13	24	4.55E-04
1010	1792	4784	128	14	24	4.33E-04
1160	1920	5120	128	15	24	4.15E-04
1200	1920	5120	128	15	25	4.11E-04
1300	2048	5456	128	16	24	4.00E-04
1600	2176	5808	128	17	26	3.84E-04
1900	2304	6144	128	18	28	3.67E-04
2250	2432	6480	128	19	30	3.47E-04
2400	2560	6832	128	20	29	3.39E-04
2640	2560	6832	128	20	32	3.25E-04
3100	2688	7168	128	21	34	3.00E-04
4200	3072	8192	128	24	36	2.72E-04
5200	3328	8880	128	26	38	2.46E-04
7000	3712	9904	128	29	41	2.00E-04

Table C.3: MoE model sizes for StripedHyena. All MoE models use 8 total experts and 2 active experts. Other model settings for corresponding **active parameter** counts follow Table D.2, including d_model, n_heads, n_layers, ffw_size, kv_size, and learning rate.

Total Params (M)	Active Params	MoE Width
194	102	1728
228	118	1856
270	134	2048
303	150	2048
319	163	2048
352	175	2048
404	196	2176
452	217	2240
512	251	2368
580	278	2368
667	306	2560
761	350	2752
950	440	2752
1160	536	3392
1390	641	3712
1660	756	4096
1940	881	4416
2230	1010	4736
2550	1160	5056
2910	1300	5440

Table C.4: StripedHyena Expert model sizes, which all use 8 total experts and 2 active experts for both sequence mixing and GLU experts. Other model settings for corresponding **active parameter** counts follow Table D.2, including d_model, n_heads, n_layers, ffw_size, kv_size, and learning rate.

Total Params (M)	Active Params	Expert Width	Expert Total Width	MoE Width
241	101	80	640	2368
290	119	88	704	2624
337	137	96	768	2816
386	153	96	768	2880
408	160	96	768	2880
452	174	96	768	2880
520	199	104	832	3072
570	215	104	832	3072
661	248	112	896	3328
749	277	112	896	3328
860	315	120	960	3584
965	352	128	1024	3776
1220	441	144	1152	4288
1500	535	160	1280	4736
1810	641	176	1408	5216
2140	757	192	1536	5696
2510	882	208	1664	6176
2880	1010	224	1792	6592
3320	1160	240	1920	7104
3790	1310	256	2048	7616

Table C.5: Notation for FLOP calculation.

Notation	Description
C	Model FLOP cost per token
N	Number of layers
L	Sequence length
D	Model width
V	Vocabulary size
H	Number of heads
$D_{\texttt{glu}}$	Width of GLU (reverse bottleneck)
$D_{\texttt{moe}}$	Width of MoE expert
$D_{\texttt{dt}}$	Width of bottleneck in Mamba featurization
$D_{\texttt{moh}}$	Width of Hyena expert
$D_{\texttt{glu}}$	Width of GLU (reverse bottleneck)
$A_{\texttt{moe}}$	Number of MoE experts
$A_{\texttt{moh}}$	Number of Hyena experts
$G_{\texttt{moe}}$	Number of active MoE experts
$G_{\texttt{moh}}$	Number of active Hyena experts
$S_{\texttt{hyena}}$	filter order
$S_{\texttt{mamba}}$	state dimension
E	projection expansion factor

Table D.1: Batch sizes by IsoFLOP group. For very small models (<54M) parameters, batch size 262k is used.

IsoFLOP	Batch Size
4.0E+18	524K
8.0E+18	524K
2.0E+19	524K
4.0E+19	524K
8.0E+19	524K
2.0E+20	1M
5.0E+20	1M
2.0E+21	2M

Table D.2: Common settings across all architectures. For Mamba, we use the layer structure of `Mb-Mb` following [11]. Actual parameter counts vary slightly for each architecture..

PARAMS (M)	D_MODEL	FFW_SIZE	KV_SIZE	N_HEADS	N_LAYERS	LEARNING RATE
8	128	336	64	2	4	9.77E-04
22	320	848	64	5	5	9.57E-04
38	448	1200	64	7	7	9.36E-04
54	512	1360	64	8	9	9.15E-04
70	576	1536	64	8	10	8.95E-04
102	640	1712	64	10	14	8.56E-04
118	704	1872	64	11	14	8.37E-04
134	768	2048	64	12	14	8.18E-04
150	768	2048	64	12	16	8.00E-04
163	768	2048	64	12	17	7.75E-04
175	768	2048	64	12	19	7.50E-04
196	832	2224	64	13	19	7.25E-04
217	832	2224	64	13	21	7.00E-04
251	896	2384	64	14	21	6.75E-04
278	896	2384	64	14	24	6.50E-04
306	960	2560	64	15	24	6.25E-04
350	1024	2736	64	16	24	6.00E-04
440	1152	3072	64	18	24	5.66E-04
536	1280	3408	64	20	24	5.33E-04
641	1408	3760	128	11	24	5.00E-04
756	1536	4096	128	12	24	4.75E-04
881	1664	4432	128	13	24	4.55E-04
1010	1792	4784	128	14	24	4.33E-04
1160	1920	5120	128	15	24	4.15E-04
1200	1920	5120	128	15	25	4.11E-04
1300	2048	5456	128	16	24	4.00E-04
1600	2176	5808	128	17	26	3.84E-04
1900	2304	6144	128	18	28	3.67E-04
2250	2432	6480	128	19	30	3.47E-04
2400	2560	6832	128	20	29	3.39E-04
2640	2560	6832	128	20	32	3.25E-04
3100	2688	7168	128	21	34	3.00E-04
4200	3072	8192	128	24	36	2.72E-04
5200	3328	8880	128	26	38	2.46E-04
7000	3712	9904	128	29	41	2.00E-04

Table D.3: MoE model sizes for StripedHyena. All MoE models use 8 total experts and 2 active experts. Other model settings for corresponding **active parameter** counts follow Table D.2, including d_model, n_heads, n_layers, ffw_size, kv_size, and learning rate.

Total Params (M)	Active Params	MoE Width
194	102	1728
228	118	1856
270	134	2048
303	150	2048
319	163	2048
352	175	2048
404	196	2176
452	217	2240
512	251	2368
580	278	2368
667	306	2560
761	350	2752
950	440	2752
1160	536	3392
1390	641	3712
1660	756	4096
1940	881	4416
2230	1010	4736
2550	1160	5056
2910	1300	5440

Table D.4: StripedHyena Expert model sizes, which all use 8 total experts and 2 active experts for both sequence mixing and GLU experts. Other model settings for corresponding **active parameter** counts follow Table D.2, including d_model, n_heads, n_layers, ffw_size, kv_size, and learning rate.

Total Params (M)	Active Params	Expert Width	Expert Total Width	MoE Width
241	101	80	640	2368
290	119	88	704	2624
337	137	96	768	2816
386	153	96	768	2880
408	160	96	768	2880
452	174	96	768	2880
520	199	104	832	3072
570	215	104	832	3072
661	248	112	896	3328
749	277	112	896	3328
860	315	120	960	3584
965	352	128	1024	3776
1220	441	144	1152	4288
1500	535	160	1280	4736
1810	641	176	1408	5216
2140	757	192	1536	5696
2510	882	208	1664	6176
2880	1010	224	1792	6592
3320	1160	240	1920	7104
3790	1310	256	2048	7616

Table D.5: Notation for FLOP calculation.

Notation	Description
C	Model FLOP cost per token
N	Number of layers
L	Sequence length
D	Model width
V	Vocabulary size
H	Number of heads
$D_{\texttt{glu}}$	Width of GLU (reverse bottleneck)
$D_{\texttt{moe}}$	Width of MoE expert
$D_{\texttt{dt}}$	Width of bottleneck in Mamba featurization
$D_{\texttt{moh}}$	Width of Hyena expert
$D_{\texttt{glu}}$	Width of GLU (reverse bottleneck)
$A_{\texttt{moe}}$	Number of MoE experts
$A_{\texttt{moh}}$	Number of Hyena experts
$G_{\texttt{moe}}$	Number of active MoE experts
$G_{\texttt{moh}}$	Number of active Hyena experts
$S_{\texttt{hyena}}$	filter order
$S_{\texttt{mamba}}$	state dimension
E	projection expansion factor

Table E.1: Topology ablation for StripedHyena (750M at 2e19 FLOPS on The Pile). `H` and `A` indicate Hyena and MHA layers, respectively.

TOPOLOGY	PERPLEXITY
(1H:1A) $\times 12$	9.52
(2H:2A) $\times 6$	9.32
(3H:3A) $\times 4$	9.33
(4H:4A) $\times 3$	9.37
(6H:6A) $\times 2$	9.28
12H:12A	9.41
(2H:4A:4H:2A) $\times 2$	9.25
(H:5A:5H:A) $\times 2$	9.31
4H:10A:8H:2A	9.33
4H:12A:8H	9.31
6H:12A:6H	9.30
8H:12A:4A	9.35