In-Context Language Learning: Architectures and Algorithms

Ekin Akyürek [1] **Bailin Wang** [1] **Yoon Kim** [1] **Jacob Andreas** [1]

Abstract

Some neural language models (LMs) exhibit a remarkable capacity for *in-context learning (ICL)*: they can fit predictors to datasets provided as input. While the mechanisms underlying ICL are well-studied in the context of synthetic problems like in-context linear regression, there is still some divergence between these model problems and the "real" ICL exhibited by LMs trained on large text corpora. In this paper, we study ICL through the lens of a new family of model problems we term **in context language learning (ICLL)**. In ICLL, LMs are presented with a set of strings from a formal language, and must generate additional strings from the same language. We focus on in-context learning of regular languages generated by random finite automata. We evaluate a diverse set of neural sequence models on regular ICLL tasks. We first show that Transformers significantly outperform neural sequence models with recurrent or convolutional representations on ICLL tasks. Next, we provide evidence that they do so by computing in-context n-gram statistics using specialized attention heads. Finally, we show that hard-wiring these heads into neural models improves performance not just on synthetic ICLL, but natural language modeling, reducing the perplexity of 340M-parameter Transformers by up to 1.14 points (6.7%) on the SlimPajama dataset. Our results highlight the usefulness of in-context formal language learning as a tool for understanding ICL in models of natural text.

1. Introduction

One of the most striking features of modern neural language models is their capacity for **in-context learning** (ICL)—the ability to infer a conditional or unconditional distribution over natural language strings simply by performing next-token prediction following a sequence of examples from the distribution of interest. ICL is a crucial tool for steering large pre-trained language models (LMs), and a growing body of work aims to understand *when* and *how* these LMs perform ICL. Because of the complexity of large-scale LMs trained on natural text (and the lack of public information about many LMs' training data), almost all work on understanding ICL has focused on smaller LMs trained on simple **model problems** like in-context linear regression (Garg et al., 2022), character classification (Chan et al., 2022), and associative recall (Fu et al., 2023). Despite their simplicity, these model problems have played a key role in identifying properties (and limitations) of ICL in current LMs.

However, there remains a significant gap between these model problems and the capabilities exhibited by large-scale LMs. In particular, most model problems require relatively simple forms of learning: computing a fixed function of the entire training set (Akyürek et al., 2023; von Oswald et al., 2023a;b), or retrieving a single example relevant to the current input (Fu et al., 2023). In contrast, natural LMs exhibit richer and much more varied forms of ICL—in some cases producing structured generative models of text or code from a handful of inputs (Shin & Van Durme, 2022; Drozdov et al., 2023).

How can we systematically study these more complex forms of ICL? In this paper, we introduce a new family of model ICL problems that we collectively term **in-context language learning (ICLL)**. In ICLL, LMs are prompted with a finite collection of strings from an unknown formal language, and must infer the distribution over strings corresponding to the full language (Figure 1). ICLL exercises essential features of ICL in natural models: it involves structured outputs, probabilistic predictions, and algorithmic reasoning about input data. In this paper, we present a focused study of ICLL in **regular languages**—the class of formal languages generated by finite automata.

We begin by providing general background about neural sequence models, ICL and formal languages in Section 2, then define the ICLL task in Section 3. Next, we explore three questions about in-context language learning in neural sequence models:[1]

[1]MIT CSAIL. Correspondence to: Ekin Akyürek <akyurek@mit.edu>.

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

[1]Code & data are released at github.com/berlino/seq_icl

Q1: Which model classes can learn to perform ICLL accurately? (Section 4)

- We find that Transformers significantly outperform recurrent and convolutional LMs at in-context language learning, even when these different architectures perform comparably on other problems.
- Models with efficient convolutional parameterizations perform especially poorly on ICLL tasks.

Q2: What algorithmic solutions do successful in-context language learners implement? (Section 5)

- Transformer predictions on ICLL with regular languages are well approximated by smoothed n-gram models.
- Transformers develop "n-gram heads": higher-order variants of induction heads previously described in LMs (Olsson et al., 2022).
- Compared to other model architectures, Transformers better encode in-context n-gram counts in their hidden representations.

Q3: Can we improve neural models using our understanding of how they perform ICLL? (Section 6)

- Hard-wiring Transformers, RNNs and convolutional models with n-gram heads improves their performance on ICLL.
- These heads are not just useful for ICLL: when equipped with n-gram heads, neural sequence models of all classes exhibit perplexity improvements of up to 6.7% on natural language modeling tasks.

Our results highlight the usefulness of ICLL as a model problem—not only as a tool for research on ICL, but as a source of insight about architectural features that can improve language modeling in the real world. Many aspects of ICLL, even with regular languages, remain to be understood (e.g. learning dynamics and out-of-distribution generalization). Beyond these, future work might study ICLL in more expressive languages (e.g. context-free or context-sensitive languages), offering a path toward understanding of even more complex behaviors in natural LMs.

2. Background

2.1. Neural sequence modeling

Much of modern machine learning for natural language processing is concerned with building general-purpose tools for sequence prediction, in which we wish to place a distribution over strings $\boldsymbol{x}$. Very often this is done via a product of conditional distributions over **tokens**: $p(\boldsymbol{x}) = \prod_i p(x_i \mid \boldsymbol{x}_{<i})$. In practice the distribution $p(x_i \mid \boldsymbol{x}_{<i})$ is typically parameterized as a neural network, in which each input token x_i (a symbol, word piece, or word) is assigned an **embedding** $\boldsymbol{e}_i$, which is used to compute a sequence of **hidden representations** $\boldsymbol{h}_i^{(\ell)}$ (one in each layer ℓ of the network). The last of these representations is ultimately used to predict the distribution over next tokens. A wide variety of architectural choices are available for computing each $\boldsymbol{h}^{(\ell)}$ from $\boldsymbol{h}^{(\ell-1)}$.

Attentional Networks Today, the most widely used architecture for neural sequence modeling is the **Transformer** (Vaswani et al., 2017). Hidden representations in Transformers are computed via an attention mechanism: to obtain $\boldsymbol{h}_i^{\ell}$, models compute weighted averages of previous-layer hidden representations $\boldsymbol{h}_{<i}^{(\ell-1)}$ followed by a feed-forward layer. Letting $\boldsymbol{x} = \boldsymbol{h}^{(\ell-1)}$ and $\boldsymbol{h} = \boldsymbol{h}^{(\ell)}$ for readability, each Transformer layer may be expressed as:

$$\boldsymbol{h}'_i = \mathrm{softmax}(\boldsymbol{W}_k \boldsymbol{x}_{<i} (\boldsymbol{W}_q \boldsymbol{x}_i)^\top) \boldsymbol{W}_v \boldsymbol{x}_{<i} \,, \quad (1)$$

$$\boldsymbol{h} = \mathrm{FFN}(\boldsymbol{W}_o \boldsymbol{h}') \,, \quad (2)$$

where FFN denotes a feed-forward network[2].

Recurrent and Convolutional Networks Sequence models other than the attention networks can be characterized as recurrent and/or convolutional:

$$\text{recurrent} \qquad \boldsymbol{h}'_i = f(\boldsymbol{A}\boldsymbol{h}'_{i-1}, \boldsymbol{B}\boldsymbol{x}_i), \quad (3)$$

$$\text{convolutional} \qquad \boldsymbol{h}'_i = \boldsymbol{l} * \boldsymbol{x}_{<i}, \quad (4)$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are recurrence parameters, f is a learned transformation, $*$ denotes convolution and $\boldsymbol{l}$ is a convolutional filter. Many sequence models, like state-space models (Gu et al., 2022b) and RWKV (Peng et al., 2023) may be equivalently expressed in recurrent, convolutional, or even attentional forms. Hence, our experiments classify models based on *time-invariance* and *linearity*. Time-invariant networks only have parameters that do not depend on the input $\boldsymbol{x}$, whereas time-variant networks have input-dependent parameters. As a middle ground between them, *weakly time-invariant* networks have a mixture of input-dependent and input-independent parameters. In addition, we use *linear* and *non-linear* to characterize recurrent models, based on whether there are non-linear dependencies among hidden states (i.e., whether f is non-linear). See Appendix A for a more detailed discussion of attentional, recurrent and convolutional models, and relations between them.

2.2. In-context learning

One feature that has been observed in all model classes above (to an extent) is their capacity for **in-context learning**.

[2]Layer norms and residual connections are omitted for brevity.

When trained appropriately, sampling from these models given a context of the form:

$$p_{\text{LM}}(\cdot \mid [\boldsymbol{d}_1, f(\boldsymbol{d}_1), \bullet, \boldsymbol{d}_2, f(\boldsymbol{d}_2), \bullet, \ldots, \bullet, \boldsymbol{d}_k]),$$

yields an accurate estimate of $f(\boldsymbol{d}_k)$. Here $\bullet$ is a delimiter token, each $\boldsymbol{d}_j$ is an input datum, and $f(\boldsymbol{d}_j)$ is its associated output. In practice, $f(\boldsymbol{d})$ may be stochastic and compositional, and both inputs and outputs may be structured (e.g. natural language strings with multiple tokens). In addition to this *function-learning* view of ICL, we may understand it more generally as a problem of learning a context-dependent next-token distribution (i.e *in-context language learning or modeling*), for some distribution over strings p_f. Here, sampling from an LM given a context with examples from p_f:

$$p_{\text{LM}}(\cdot \mid [\underbrace{x_{1,1}, \ldots, x_{1,n_1}}_{\boldsymbol{d}_1 \sim p_f}, \bullet, \ldots, \bullet, x_{k,1}, \ldots, x_{k,n_k-1})]),$$

yields an estimate of x_{k,n_k}. Iteratively sampling tokens from these conditional distributions, we may sample from p_f.

While some work on understanding ICL has studied LMs trained on natural text (Olsson et al., 2022; Dai et al., 2023), most interpretability-oriented research has studied LMs trained to solve simpler problems like regression (Akyürek et al., 2023; von Oswald et al., 2023a), associative recall (Fu et al., 2023) or few-shot classification (Chan et al., 2022). Work in this family studies ICL from several perspectives: as task identification (Xie et al., 2022; Min et al., 2022), string manipulation (Olsson et al., 2022), or a form of learned "mesa-optimization" within a trained LM (Akyürek et al., 2023; von Oswald et al., 2023a; Dai et al., 2023).

2.3. Formal Languages

We study ICL in the context of formal language learning, in which models condition not on (input, output) pairs, but instead on a collection of strings sampled from a randomly generated language. Related language-learning problems were also studied by Xie et al. (2022) and Hahn & Goyal (2023); here we study models' ability to learn *new* languages rather than recognizing languages from a predetermined set.

Strings, languages, and automata In the context of formal language theory, a **language** L is defined as a set of strings over a finite alphabet Σ. A **probabilistic language** additionally defines an (optionally normalized) distribution $P(x)$ over the strings $x \in L$. An **automaton** is an abstract machine that defines a procedure for testing membership in some language, or generating strings from that language.

Our experiments focus on **regular languages**. These are standardly defined as the set of languages recognized by **deterministic finite automata** (DFAs). A DFA, in turn, is defined by an **alphabet** Σ, a set of **states** $\mathcal{S}$, an **initial state** $S_0 \in \mathcal{S}$, a subset of **accepting states** $\mathcal{S}_a \subseteq \mathcal{S}$, and a state **transition function** $T : \mathcal{S} \times \Sigma \to \mathcal{S}$. To determine whether a DFA accepts some string $\boldsymbol{x} = x_1 x_2 x_3 \ldots x_n$, we begin in S_0, set $S_i = T(S_{i-1}, x_i)$, and finally test if $S_n \in \mathcal{S}_a$.

To extend this definition to probabilistic languages, we may generalize DFAs to probabilistic finite automata (PFAs) by redefining transition functions, initial states, and final states respectively as distributions $T : \mathcal{S} \times \Sigma \times \mathcal{S} \to [0, 1]$, $I : \mathcal{S} \to [0, 1]$, and $A : \mathcal{S} \to [0, 1]$, satisfying $\Sigma_S A(S) = 1$, $\Sigma_S I(S) = 1$ and $A(S) + \Sigma_{x,S'} T(S, x, S') = 1$. Under mild conditions $p_{\text{PFA}}(x) = \sum_{S_0,\ldots,S_n} I(S^0) \prod_{i=1}^{n} T(S_{i-1}, x_i, S_i) A(S_n)$ is a distribution over strings, where the sum is over all state sequences.

In this work, we will use PFAs with a single initial state and without any terminal states (sometimes referred to as NFPAs[3]). We assign probabilities $p_{\text{PFA}}(x) = \sum_{s_0,\ldots,s_n} \prod_{i=1}^{n} T(s_{i-1}, x_i, s_i)$, where the sum is over all possible state sequences. This is proven to be a proper distribution for each sequence length (Dupont et al., 2005).

Formal languages and deep networks A large body of previous work has used formal languages to probe the limits of neural sequence models (Elman, 1990; Gers & Schmidhuber, 2001; Bhattamishra et al., 2020; Suzgun et al., 2019; Hewitt et al., 2020; Finlayson et al., 2022). Notably, Merrill & Sabharwal (2023) show that under standard hardness assumptions, bounded-precision Transformers cannot recognize important formal language classes, including some regular languages. In contrast to this past work, our study focuses not on whether sequence models can be trained to generate or recognize strings in a *fixed* formal language, but instead whether they can be "meta-trained" to adapt on the fly to new languages provided in context. Closest to this goal, the "associative recall" task studied by Fu et al. (2023) and Arora et al. (2023) may also be viewed as a special case of in-context language learning for an extremely restricted sub-class of regular languages; we compare the behavior of models on this class and general ICLL in Section 4.

3. REGBENCH: A Benchmark Dataset for In-Context Language Learning

What does it mean to *learn* a language of the kind described in Section 2? Classical formal language theory has studied a number of different learnability criteria, including exact identification (Gold, 1967), sometimes with stochastic samples (Angluin, 1988) and probabilistic success criteria (Pitt, 1989). But if our goal is to understand the behavior of LMs,

[3]Refer to Dupont et al. (2005) and Appendix C.2 more discussion of the relationship between NFPAs and their equivalance to HMMs.

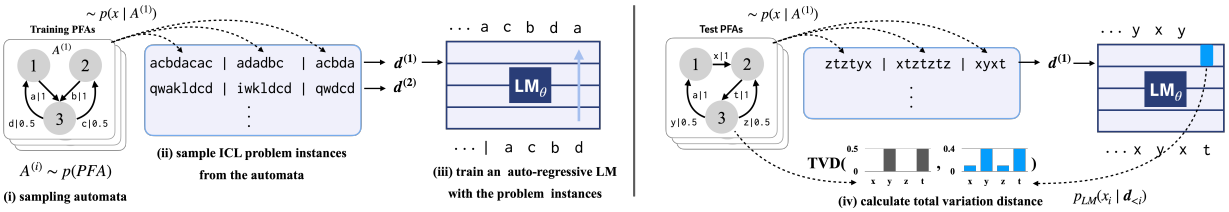


Figure 1. **ICLL Benchmark:** We randomly generate probabilistic finite automata (PFAs), then generate problem instances that include multiple samples from each PFA. We train and evaluate models on disjoint PFAs for in-context language learning.

we wish to characterize models' ability to *approximately* predict the next-token distribution given a *finite* set of samples, as in work on PAC learning (Valiant, 1984). Unlike the PAC setting (but like in natural language modeling), our evaluation assumes a *fixed* prior distribution over languages.

To do so, we introduce a new dataset called **REGBENCH**. REGBENCH consists of a set of **problem instances** $\boldsymbol{d}^{(i)}$, each comprising a sequence of **examples** $[\boldsymbol{d}_1^{(i)}, \bullet, \boldsymbol{d}_2^{(i)}, \bullet, \ldots, \bullet, \boldsymbol{d}_k^{(i)}]$ drawn from the same probabilistic language $L^{(i)}$. REGBENCH is related to other synthetic language learning datasets, especially the MLRegTest benchmark of van der Poel et al. (2023), but focuses on generation rather than membership testing. To describe how REGBENCH is constructed, we must specify **(1)** how languages are sampled, **(2)** how strings are sampled from these languages, and **(3)** how learners are evaluated.

3.1. Sampling languages

REGBENCH is built using probabilistic automata, themselves sampled from a probabilistic generative process. This process is defined formally as:

1. Choose a number of states uniformly from $|\mathcal{S}| \in [4, 12]$. Given this value, define a set of automaton states $\mathcal{S} = \{S_1, \ldots, S_{|\mathcal{S}|}\} \cup \{S_0\}$. Define the set of accepting states $\mathcal{S}_a = \{S_1, \ldots, S_{|\mathcal{S}|}\}$ (excluding S_0).

2. Choose an alphabet size uniformly from $|V| \in [4, 18]$. Sample a language-specific alphabet V, containing $|V|$ symbols, uniformly (without replacement) from a shared symbol set $\mathcal{V}$ (with $|\mathcal{V}| = 18$).

3. For each S_i, choose a number of outgoing edges uniformly from $o_i \in [1, 4]$. Then, construct a set of edges (S_i, x_j, S_j), where all x_j are sampled uniformly without replacement from V, and all S_j are sampled uniformly without replacement from $\{S_1, \ldots, S_{|S|}\} \setminus S_i$. For every symbol x' not sampled in this step, construct an edge (S_i, x', S_0). Together, these edges determine the transition distribution for a (non-probabilistic) DFA A.[4]

4. Construct a new DFA A' by minimizing A (Hopcroft, 1971).

5. Finally, turn A' into a probabilistic automaton without final states by defining each $T(S_i, x_j, S_j) = 1/o_j$ for edges generated above (excluding edges into S_0), and $T(S_i, x', S') = 0$ for all other x', S'.

This procedure may be run repeatedly to obtain a collection of PFAs A', each with corresponding DFA A, and associated with a stochastic language L.

3.2. Sampling strings

Given a PFA A, sampling from the associated language is straightforward:

(1) Sample a sequence length uniformly from $n \in [1, 50]$.

(2) Initialize the sampling procedure with S_0.

(3) For each $i \in 1 \ldots n$, sample a transition $(x_{i+1}, S_{i+1}) \sim T(S_i, \cdot, \cdot)$.

(4) Return $x_1 \cdots x_n$.

3.3. REGBENCH Dataset

Using these two sampling procedures, we construct REGBENCH as follows:

(1) Sample a collection of $N_{\text{train}} + N_{\text{test}}$ *distinct* automata $A^{(i)}$ using the procedure in Section 3.1.

(2) From each automaton, choose a number of strings uniformly from $k \in [10, 20]$.

(3) Sample k strings $(\boldsymbol{d}_{1\ldots k}^{(i)})$ from the automata $A^{(i)}$ with $n \in [1, 50]$ symbols each (making the average length of a problem instance $\approx L = 382$) to obtain the problem instance $\boldsymbol{d}^{(i)} = [\boldsymbol{d}_1^{(i)}, \bullet, \boldsymbol{d}_2^{(i)}, \bullet, \ldots, \bullet, \boldsymbol{d}_k^{(i)}]$.

(4) Finally, divide this collection of instances into training and test sets.

[4]This particular choice of transition function ensures that each accepted input corresponds to a unique state sequence. This makes it possible to calculate conditional next token probabilities without needing to marginalize over state sequences

Associative Recall Dataset We also experiment with the associative recall (AR) dataset (Fu et al., 2023) consists of strings in the form of $k_1 v_1, ..., k_n v_n k_q$, where each unique key k is followed by a corresponding unique value v, and the model needs to complete last query k_q with its matching value v_q that is presented at least one time in the context. AR is a simpler case of REGBENCH with deterministic languages[5] and the evaluation is based on accuracy of the last symbol.

4. Which Model Classes Learn to Perform ICLL Efficiently?

In this section, we use REGBENCH to analyze the behavior of neural sequence models on ICLL tasks. These experiments aim to characterize the relationship between REGBENCH and related existing evaluations of ICL, and to determine whether there are meaningful differences between different neural sequence models in their ability to perform ICLL. See Lee et al. (2023) for a study of ICL across models on a large collection of alternative tasks.

4.1. Setup

We train models on the REGBENCH dataset to maximize the likelihood:

$$\mathcal{L}(\theta) = \sum_{\boldsymbol{d} \in \mathcal{D}_{\text{train}}} \sum_{i} \log p_\theta(x_i \mid \boldsymbol{d}_{<i}). \tag{5}$$

$p_\theta(\cdot \mid \boldsymbol{d}_{<i})$ is the model's probability of generating a symbol following the context $\boldsymbol{d}_{<i}$ where this context consists of i symbols comprising of zero or more full examples $\boldsymbol{d}$ followed by a partial example[6]. For comparison, we also train models on the associative recall (AR) task introduced by Poli et al. (2023), using a vocabulary size of $\mathcal{V} = 40$ (based on Poli et al., 2023's "hard" setting) and an input sequence length of $L = 382$ (which matches REGBENCH's average sequence length). As with REGBENCH, we use a test set of size 500. For both datasets, we use training subsets of sizes from 150 examples to 40000 examples to evaluate scaling behavior of models.

4.2. Neural sequence models

We evaluate 10 neural sequence models: Transformers (Vaswani et al., 2017; Touvron et al., 2023); two Transformer variants with linear attention (RetNet, Sun et al., 2023 and Linear Transformer, Katharopoulos et al., 2020); four recurrent models (LSTM, Hochreiter & Schmidhuber, 1997, RWKV, Peng et al., 2023, GLA, Yang et al., 2023, and Mamba, Gu & Dao, 2023); and three models with convolutional representations (S4, Gu et al., 2022b, H3, Fu et al., 2023, and Hyena, Poli et al., 2023).

4.3. Baseline learning algorithms

We also compare to two classical procedures for generative sequence modeling. Given the procedure for sampling languages described in Section 3, the Bayes optimal predictor has the form:

$$p(x_i \mid \boldsymbol{d}_{<i}) = \sum_{L} p(x_i \mid L, \boldsymbol{d}_{\text{last}}) p(L \mid \boldsymbol{d}_{<i}) \tag{6}$$

$$\propto \sum_{L} p(x_i \mid L, \boldsymbol{d}_{\text{last}}) p(\boldsymbol{d}_{<i} \mid L) p(L), \tag{7}$$

where $\boldsymbol{d}_{\text{last}}$ is the last partial example in the $\boldsymbol{d}_{<i}$, $p(L)$ is the prior that a given language is produced by the sampling process, and $p(x_i \mid L, \boldsymbol{d}_{\text{last}})$ is the probability of the emitting x_i after $\boldsymbol{d}_{\text{last}}$ under the language L.

In contrast to model problems like linear regression (Garg et al., 2022), there are no known algorithms for efficiently computing the Bayes-optimal next-token predictive distribution for the data-generating process in Section 3. However, several classical approaches based on maximum likelihood estimation often perform well. We compare to:

- **In-context n-gram models**, which consider a fixed-sized context window of $n - 1$ symbols, locate all matching windows within the problem instance, and simply count the number of occurrences of each possible next token across those matches. Our experiments use a variant with backoff (Chen & Goodman, 1996)—see Appendix C.1 for details.
- **In-context HMMs**, which explicitly attempt to infer the probabilistic automaton that generated the context using the **Baum–Welch (BW)** algorithm (Rabiner, 1989). Given strings generated by a PFA (or hidden Markov model), BW performs maximum likelihood estimation of the transition distribution via Expectation–Maximization (Dempster et al., 1977), then uses the forward algorithm to infer the next-token distribution for a given context—see Appendix C.2 for details.

Note that both of these baselines use only the information available *within* an individual problem instance; unlike the neural models, they cannot pool information about the language-generating process across examples.

4.4. Metrics

We evaluate models using two quantities. The first is greedy-decoding **accuracy**—whether each next token predicted by

[5] Each AR string can be viewed as multiple samples from specific set of regular languages that only accepts strings in the form of kv with a finite set keys and values.

[6] In our convention, subscript ranges e.g. $\boldsymbol{d}_{<i}$ on problem instances correspond to symbol indices not the sample example indices.

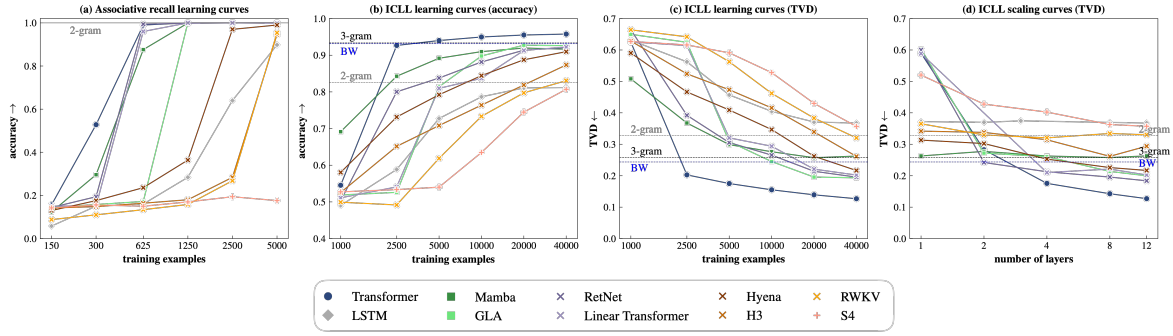


Figure 2. **REGBENCH results:** REGBENCH (b, c) yields greater contrast between models comparing to associative recall (a), and enables probabilistic evaluation (c). We find that Transformers are significantly more data-efficient than recent neural sequence models on in-context learning of regular languages. Transformers also show monotonically increasing scaling curves w.r.t. the number of layers (d).

the model is valid under the current language:

$$\text{accuracy}(p_\theta, L) = \frac{1}{\text{NT}} \sum_{\boldsymbol{d} \in \mathcal{D}_{\text{test}}} \sum_{i} \left[\mathbf{1}[\text{argmax}\, p_\theta(x \mid \boldsymbol{d}_{<i}) \in \text{supp}(L(x \mid \boldsymbol{d}_{\text{last}}))] \right]. \quad (8)$$

Here NT is number of total symbols in the test set. $L(x \mid \boldsymbol{d}_{\text{last}})$ is the short hand for $p(x \mid \boldsymbol{d}_{\text{last}}, L)$ used in Equation (6). We additionally compute **total variation distance** between each predicted next-token distribution and the distribution under the true language L:

$$\text{tvd}(p_\theta, L) = \frac{1}{2\text{NT}} \sum_{\boldsymbol{d} \in \mathcal{D}_{\text{test}}} \sum_{i} \sum_{x \in \mathcal{V}} \left| p_\theta(x \mid \boldsymbol{d}_{<i}) - L(x \mid \boldsymbol{d}_{\text{last}}) \right|. \quad (9)$$

4.5. Results

ICLL on REGBENCH shows clear differences across models On REGBENCH (Figure 2b–c), we find that Transformer models significantly outperform models in all other classes, across evaluation metrics and training set sizes. Indeed, most non-Transformer models underperform simple n-gram and Baum–Welch baselines, except in the high-data regime. In contrast, models are less clearly differentiated by the associative recall task (Figure 2a).

Depth is necessary but not sufficient for ICLL In Figure 2d, we find that no architecture achieves non-trivial performance on ICLL with only a single layer. Transformer models monotonically improve their accuracy as the number of layers increases; other models start to overfit to the training set with increasing depth.

5. What Algorithmic Solutions do In-Context Language Learners Implement?

We have seen that Transformers significantly outperform other neural sequence models at regular ICLL. Can we understand *why* these differences occur? We next analyze the behavior of Transformers trained for ICLL tasks to characterize the computations they perform and the features they represent. Our analysis uses three complementary strategies: attention visualization, probing, and black-box input–output analysis. While these methods all have limitations as interpretability tools (Wen et al., 2023; Bolukbasi et al., 2021; Belinkov, 2022), they offer convergent evidence that n-gram statistics play a key role in Transformer ICLL.

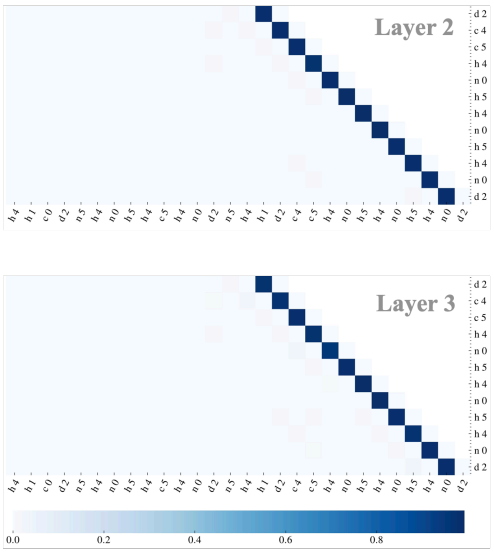
5.1. Transformers form in-context n-gram heads

In Figure 3, we visualize the attention of an (8-layer, 1-head) Transformer. The layer 2 and 3 attention heads each attend to the previous token. When composed, these heads enable each hidden representation (starting in layer 3) to incorporate information about the identities of the two tokens that precede it. The layer 5 head then attends to tokens *following* 2-grams matching the most recent 2-gram in the input. In Figure 3, for example, the input ends in `nh`, and the layer 5 head attends to all tokens X in contexts `nh`X. Notably, these heads do not selectively attend to tokens generated from the same DFA state, as might be expected if LMs inferred the true data-generating process. The pattern shown in Figure 3 is a higher-order analog of the "induction head" motif previously described by Olsson et al. (2022).

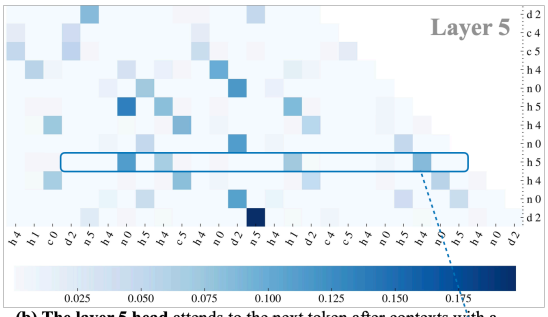
5.2. Transformers represent in-context n-gram counts better than other models

Next, we probe this model (Shi et al., 2016) to determine whether the n-gram statistics associated with these attention patterns are in fact *encoded* in hidden representations.

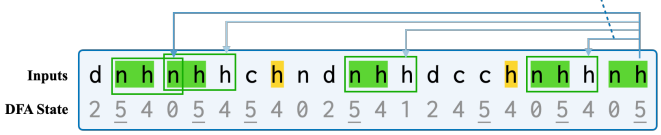
Setup To train **n-gram probes**, we extract the intermediate layer outputs $\boldsymbol{h}$ from the models as they process sequences from the training set. For varying values of n, we construct an MLP-based probe that takes as input a representation $\boldsymbol{h}_i$ at time step i and a query token x. We train this probe to predict the (unnormalized) **count** of times x occurs follow-



(a) The layer 2 and layer 3 heads both attend to the previous token. Enables the token information to be moved two-steps ahead.



(b) The layer 5 head attends to the next token after contexts with a matching 2-gram (implementing a higher-order "induction head").



(c) Layer 5 attentions zoomed in: at the final token **h**, this layer only attends to tokens following the final 2-gram nh. Notably, it does not attend to tokens following a matching 1-gram h, or to tokens following the matching state 5.

Figure 3. **N-gram heads in Transformers.** We plot the attention weights of an 8-layer, 1-head Transformer model trained on ICLL with $N = 2500$ training examples. Each row shows which tokens the label in that row attends to and the corresponding weights. We display the current PFA state next to each token label on the y and x axes. Heads in early layers attend to previous tokens (a, b), while the attention head in layer 5 selectively attends to tokens based on their 2-gram prefix rather than the 1-gram prefix or the PFA state.

ing the same $n-1$ tokens that appear at position i in the input. We then train similar probes to predict the (normalized) **frequency** $p(x \mid \boldsymbol{d}_{i-n+1:i}) = \frac{\text{count}(\boldsymbol{d}_{i-n+1:i}x)}{\text{count}(\boldsymbol{d}_{i-n+1:i})}$ and the binary **existence** $\mathbf{1}[\text{count}(\boldsymbol{d}_{i-n+1:i}x) > 0]$.

We additionally probe models for latent PFA states. Because the labeling of states is arbitrary, we train **state equivalence probes** to take two representations from different timesteps, and predict whether they were generated by the same underlying PFA state. Details are provided in Appendix E.

Metrics We evaluate count and frequency probes according to relative error ($\frac{|\hat{y}-y|}{y}$; lower is better). We evaluate existence and state equivalence probes according to binary classification accuracy (higher is better). We train separate probes per layer, per model and per task. In Figure 4, we display the result for each task and model at the **best** layer.

Results As seen in Figure 4, frequency probes on Transformers substantially outperform probes trained on other models. Interestingly, however, these results do not carry over to unnormalized counts, for which Transformer encodings do not seem to be meaningfully different from other architectures. In addition to counts, we can decode n-gram existence (Figure 4b) more accurately from Transformers than other models, as well as equivalence of underlying automaton states (Figure 4c). Supplementary results for models trained under high-resource conditions are presented in Figure 6.

5.3. Transformer predictions resemble n-gram models with learned reweighting

The previous two analyses show that Transformer ICLL computes n-gram statistics, but do not explain how this information is used to make predictions. Next, we show that Transformer predictions are well approximated by simpler models with access to the context *only* via n-gram statistics.

Setup We previously observed that Transformers, trained on enough data, more accurately predict the distribution over next tokens than n-gram language models. Here, we compare these different models' predictions *to each other*, in addition to the ground-truth language, to reveal similarities in prediction strategies across predictors of different classes.

We introduce one additional model for comparison: a **learned n-gram reweighting** model. Given an input sequence, this model first represents the input by computing the empirical next-token distribution in contexts matching the last 1 and 2 tokens of the input, as well as the empirical unigram distribution. It then concatenates these distributions and passes them through an MLP with one hidden layer. This model is trained using the same language modeling objective as other models. We evaluate two variants: one in which inputs contain unnormalized counts of matching n-grams, and another with normalized distributions. Both models are related to counter automata (Merrill, 2020).

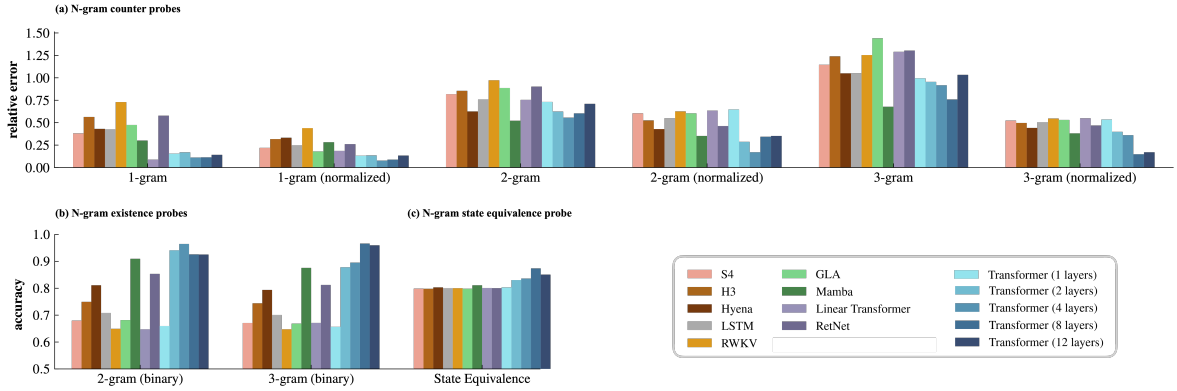


Figure 4. **Probing for n-grams in neural sequence models** (trained with $N_{\text{train}} = 2500$ examples): (a) Probes are trained to predict the counts and normalized counts (frequencies) of the most recent $(n-1)$-gram + a next query character from the model's hidden state at that time step. Results indicate that Transformer architectures more effectively encode frequencies of higher-order n-grams (bi-grams and tri-grams) compared to other models, with larger Transformer models exhibiting improved performance for higher n-grams. Additional probes show that Transformers better encode n-gram existence (b) and DFA state (c). Please refer to Appendix E for the details.

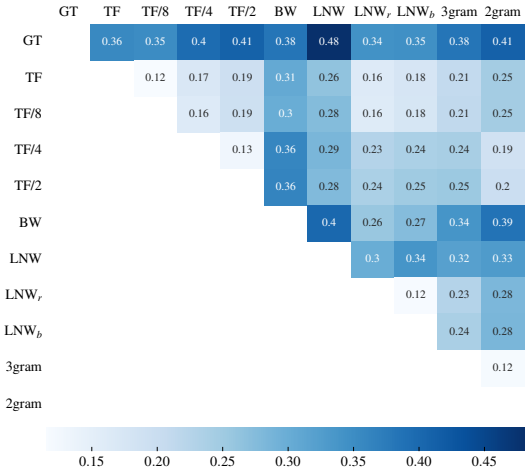


Figure 5. **Pairwise total variation distance (TVD):** We measure total variation distance between pairs of models (trained with $N = 2500$ examples) across the first 100 tokens of each string in the REGBENCH test set. The 12-layer Transformer model (TF) is closest to the learned MLP reweighting model with normalized in-context n-gram distributions as input (LNW$_r$). The 2-layer Transformer (TF/2) is closer to a 2-gram than a 3-gram model. For reference, the mean TVD between the TF and others TFs initialized with different random seed is 0.11.

Results Comparisons are shown in Figure 5. Large Transformers with many layers tend to produce next-token distributions more similar to those of n-gram models than to the ground-truth DFA or the Baum-Welch algorithm. Moreover, the learned n-gram reweighting model (LNW$_r$) best approximates the distributions from the large Transformer. Interestingly (and in line with the findings in Section 4) the shallow 2-layer Transformer more closely matches predictions from the 2-gram baseline than the 3-gram baseline.

6. How Can Findings About ICLL Inform the Design of Neural Sequence Models?

The preceding results suggest that Transformers' success stems in part from computation of in-context n-gram statistics. Can we use this information to improve other models, or to make Transformers more efficient?

The attention pattern associated with "n-gram heads" (illustrated in layer 5 of Figure 3) may be parameterized as:

$$A(n)_{ij} \propto \mathbf{1}[\underbrace{(\wedge_{k=1}^{n} x_{i-k} = x_{j-k-1})}_{\text{n-gram matching}} \wedge \underbrace{(i > j)}_{\text{causal mask}}], \quad (10)$$

where A_{ij} denotes the weight with which the ith token attends to the jth token. Building on this observation, we propose to equip models with special **static n-gram attention heads**, which attend in a fixed fashion, but apply a learned transformation of the attended-to representations:

$$\texttt{NGH}^n(h^{(l-1)})_i = W_1 \boldsymbol{h}_i^{(l-1)} + W_2 A(n)_i^\top \boldsymbol{h}^{(l-1)}, \quad (11)$$

For a model with a hidden state of size d, such a head has $2d^2$ parameters. To improve a model, we can simply insert it as a standalone new layer between the existing layers.[7]

Unlike standard self-attention, n-gram heads do not have to store the entire input in memory. Instead, in-context ngrams can be stored in a trie (Pauls & Klein, 2011) and queried efficiently. Thus, for models with a recurrent form, $\texttt{NGH}^n$ introduces little overhead during inference.

ICLL Our experiments take an existing architecture (for example RetNet), and insert a sequence of three NGH heads

[7] Arora et al. (2023)'s sparse attention layers resembles our $\texttt{NGH}^1$, but with learned attention weights; our implementation attends uniformly to all matching contexts.

Table 1. **N-gram layers bring other models to the Transormer level on ICLL**: We train RetNet and GLA models with n-gram heads on ICLL with $N = 2500$ training examples. In TVD metric, adding n-gram layers, brings model performance to the Transformer level trained on the same data without n-gram heads.. In accuracy, hybrid models can outperform Transformer models in the accuracy metric.

Model	TVD (↓)	Accuracy (↑)
RetNet (Sun et al., 2023)	0.392	0.800
└ $\texttt{NGH}_1^{(1)}$	0.310	0.814
└ $\texttt{NGH}_1^{(1,2)}$	0.229	0.925
└ $\texttt{NGH}_1^{(1,2,3)}$	0.217	0.94
GLA (Yang et al., 2023)	0.624	0.526
└ $\texttt{NGH}_1^{(1)}$	0.302	0.819
└ $\texttt{NGH}_1^{(1,2)}$	0.211	0.929
└ $\texttt{NGH}_1^{(1,2,3)}$	0.207	**0.946**
Transformer (Vaswani et al., 2017)	**0.203**	0.926

Table 2. **N-gram layers improve language models**: We train equal sized (340M parameters) language models with and without n-gram heads on 7B tokens from the SlimPajama dataset (Soboleva et al., 2023). Adding n-gram layers improves performance regardless of the model, reducing test set perplexities by up to 1.14.

Model	Perplexity (↓)
RetNet (Sun et al., 2023)	16.55
└ $\texttt{NGH}_1^{(1,2,3)}$ + $\texttt{NGH}_{-2}^{(1,2,3)}$	15.86 (**+4.2%**)
GLA (Yang et al., 2023)	15.65
└ $\texttt{NGH}_1^{(1,1,1)}$ + $\texttt{NGH}_{-2}^{(1,1,1)}$	15.54 (**+0.7%**)
└ $\texttt{NGH}_1^{(1,2,3)}$ + $\texttt{NGH}_{-2}^{(1,2,3)}$	15.24 (**+2.6%**)
Transformer (Llama) (Touvron et al., 2023)	16.96
└ $\texttt{NGH}_1^{(1,2,3)}$+ $\texttt{NGH}_{-2}^{(1,2,3)}$	**15.82** (**+6.7%**)

with increasing context size $[\texttt{NGH}^1, \texttt{NGH}^2, \texttt{NGH}^3]$ sequentially. We denote this whole bundle $\texttt{NGH}_{\text{m}}^{(1,2,3)}$, where m specifies the original layer after which the NGH heads are added, and the whole architecture as RetNet + $\texttt{NGH}_{\text{m}}^{(1,2,3)}$ (with negative m denoting an offset from the output layer rather than the input layer).

We insert n-gram layers into RetNet and GLA models. In Table 1, the addition of ordinary induction heads ($\texttt{NGH}^{(1)}$) improves GLA more than 50% in TVD, and improves RetNet slightly. With second-order n-gram heads ($\texttt{NGH}^{(1,2)}$), both RetNet and GLA match Transformer performance. Third-order heads improve performance even further, enabling other models to outperform Transformers in accuracy.

Language Modeling We next test if these improvements transfer to language modeling on real data. We add layer normalization and MLP networks after each NGH layer such that one $\texttt{NGH}_{\text{m}}^{(i)}$ plus MLP makes $4d^2$ parameters, and collectively three NGH layers (i.e., $i \in [1, 2, 3]$) make $12d^2$ parameters, matching the number of parameters of a standard Transformer layer.[8] For each base model, we insert two n-gram head sequences: one replacing the second layer ($\texttt{NGH}_1^{(1,2,3)}$), and one replacing the second-to-last ($\texttt{NGH}_{-2}^{(1,2,3)}$).

In Section 6, we observe that n-gram heads consistently improve LMs. Indeed, the best improvement appears in Transformers, with a 6.7% decrease in perplexity. Higher-order heads are important: using all n-grams decreases perplexity 4 times more than using just 1-gram layers.

7. Conclusion

This paper has investigated in-context language learning (ICLL) in neural sequence models. We identified key differences among model classes, with Transformers emerging as particularly adept at ICLL. Further investigation revealed that Transformers succeed by implementing in-context n-gram heads (higher-order induction heads). Inspired by these findings, we demonstrated that inserting simple n-gram heads into neural architectures significantly improves their ICLL performance and fit to natural data.

Impact Statement

This paper presents experiments on a synthetically generated dataset and a web corpus, with the goal of analyzing and improving in-context learning and language modeling. Like many models trained on web text, LMs trained on the SlimPajama dataset may produce biased or harmful output. We do not currently have reason to believe that our proposed modeling improvements affect the prevalence of these outputs. Our code release does not include a trained model.

Acknowledgements

Thanks to William Merill and Jon Rawski for valuable feedback on an early draft of this paper. Ekin Akyürek and Jacob Andreas are supported by Intel and the National Science Foundation under the PPoSS program (CCF-2217064) as well as the OpenPhilanthropy Foundation. Yoon Kim and Bailin Wang were supported by MIT-IBM Watson AI.

References

Akyürek, E., Schuurmans, D., Andreas, J., Ma, T., and Zhou, D. What learning algorithm is in-context learning? Investigations with linear models. In *Proceedings of the International Conference on Learning Representations*,

[8] See Appendix F for a sample implementation.

2023.

Angluin, D. *Identifying languages from stochastic examples*. Yale University. Department of Computer Science, 1988.

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*, abs/2312.04927, 2023.

Belinkov, Y. Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics*, 48(1), 2022.

Bhattamishra, S., Ahuja, K., and Goyal, N. On the ability and limitations of Transformers to recognize formal languages. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2020.

Bolukbasi, T., Pearce, A., Yuan, A., Coenen, A., Reif, E., Viégas, F., and Wattenberg, M. An interpretability illusion for BERT. *ArXiv preprint*, abs/2104.07143, 2021.

Chan, S., Santoro, A., Lampinen, A., Wang, J., Singh, A., Richemond, P., McClelland, J., and Hill, F. Data distributional properties drive emergent in-context learning in Transformers. *Advances in Neural Information Processing Systems*, 2022.

Chen, S. F. and Goodman, J. An empirical study of smoothing techniques for language modeling. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, California, USA, 1996.

Dai, D., Sun, Y., Dong, L., Hao, Y., Ma, S., Sui, Z., and Wei, F. Why can GPT learn in-context? Language models secretly perform gradient descent as meta-optimizers. In *Findings of the Association for Computational Linguistics*, 2023.

Dempster, A. P., Laird, N. M., and Rubin, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1), 1977.

Drozdov, A., Schärli, N., Akyürek, E., Scales, N., Song, X., Chen, X., Bousquet, O., and Zhou, D. Compositional semantic parsing with large language models. In *Proceedings of the International Conference on Learning Representations*, 2023.

Dupont, P., Denis, F., and Esposito, Y. Links between probabilistic automata and hidden Markov models: Probability distributions, learning models and induction algorithms. *Pattern Recognition*, 38(9), 2005.

Elman, J. L. Finding structure in time. *Cognitive science*, 14(2), 1990.

Finlayson, M., Richardson, K., Sabharwal, A., and Clark, P. What makes instruction learning hard? An investigation and a new challenge in a synthetic environment. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2022.

Fu, D. Y., Dao, T., Saab, K. K., Thomas, A. W., Rudra, A., and Re, C. Hungry hungry hippos: Towards language modeling with state space models. In *Proceedings of the International Conference on Learning Representations*, 2023.

Garg, S., Tsipras, D., Liang, P. S., and Valiant, G. What can Transformers learn in-context? A case study of simple function classes. *Advances in Neural Information Processing Systems*, 2022.

Gers, F. A. and Schmidhuber, E. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6), 2001.

Gold, E. M. Language identification in the limit. *Information and Control*, 10(5), 1967.

Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. *ArXiv preprint*, abs/2312.00752, 2023.

Gu, A., Goel, K., Gupta, A., and Ré, C. On the parameterization and initialization of diagonal state space models. *Advances in Neural Information Processing Systems*, 35, 2022a.

Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. In *Proceedings of the International Conference on Learning Representations*, 2022b.

Hahn, M. and Goyal, N. A theory of emergent in-context learning as implicit structure induction. *ArXiv preprint*, abs/2303.07971, 2023.

Hewitt, J., Hahn, M., Ganguli, S., Liang, P., and Manning, C. D. RNNs can generate bounded hierarchical languages with optimal memory. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2020.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8), 1997.

Hopcroft, J. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Mchines and Computations*. Elsevier, 1971.

Hua, W., Dai, Z., Liu, H., and Le, Q. V. Transformer quality in linear time. In *Procedings of the International Conference on Machine Learning*, 2022.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are RNNs: Fast autoregressive Transformers with linear attention. In *Proceedings of the International Conference on Machine Learning*, 2020.

Lee, I., Jiang, N., and Berg-Kirkpatrick, T. Exploring the relationship between model architecture and in-context learning ability. *ArXiv preprint*, abs/2310.08049, 2023.

Mehta, H., Gupta, A., Cutkosky, A., and Neyshabur, B. Long range language modeling via gated state spaces. *ArXiv preprint*, abs/2206.13947, 2022.

Merrill, W. On the linguistic capacity of real-time counter automata. *ArXiv preprint*, abs/2004.06866, 2020.

Merrill, W. and Sabharwal, A. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11, 2023.

Min, S., Lyu, X., Holtzman, A., Artetxe, M., Lewis, M., Hajishirzi, H., and Zettlemoyer, L. Rethinking the role of demonstrations: What makes in-context learning work? *ArXiv preprint*, abs/2202.12837, 2022.

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*, abs/2209.11895, 2022.

Pauls, A. and Klein, D. Faster and smaller n-gram language models. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, 2011.

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*, abs/2305.13048, 2023.

Pitt, L. Probabilistic inductive inference. *Journal of the ACM (JACM)*, 36(2), 1989.

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*, abs/2302.10866, 2023.

Qin, Z., Han, X., Sun, W., Li, D., Kong, L., Barnes, N., and Zhong, Y. The devil in linear transformer. *ArXiv preprint*, abs/2210.10340, 2022.

Rabiner, L. R. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), 1989.

Shazeer, N. GLU variants improve Transformer. *ArXiv preprint*, abs/2002.05202, 2020.

Shi, X., Padhi, I., and Knight, K. Does string-based neural MT learn source syntax? In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Austin, Texas, 2016.

Shin, R. and Van Durme, B. Few-shot semantic parsing with language models trained on code. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Seattle, United States, 2022.

Soboleva, D., Al-Khateeb, F., Myers, R., Steeves, J. R., Hestness, J., and Dey, N. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama, 2023. URL https://huggingface.co/datasets/cerebras/SlimPajama-627B.

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

Sun, Y., Dong, L., Huang, S., Ma, S., Xia, Y., Xue, J., Wang, J., and Wei, F. Retentive network: A successor to transformer for large language models. *ArXiv preprint*, abs/2307.08621, 2023.

Suzgun, M., Gehrmann, S., Belinkov, Y., and Shieber, S. M. Memory-augmented recurrent neural networks can learn generalized Dyck languages. *ArXiv preprint*, abs/1911.03329, 2019.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *ArXiv preprint*, abs/2302.13971, 2023.

Valiant, L. G. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

van der Poel, S., Lambert, D., Kostyszyn, K., Gao, T., Verma, R., Andersen, D., Chau, J., Peterson, E., Clair, C. S., Fodor, P., et al. MLRegTest: A benchmark for the machine learning of regular languages. *ArXiv preprint*, abs/2304.07687, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

von Oswald, J., Niklasson, E., Randazzo, E., Sacramento, J., Mordvintsev, A., Zhmoginov, A., and Vladymyrov, M. Transformers learn in-context by gradient descent. In *Proceedings of the International Conference on Machine Learning*, 2023a.

von Oswald, J., Niklasson, E., Schlegel, M., Kobayashi, S., Zucchet, N., Scherrer, N., Miller, N., Sandler, M.,

Vladymyrov, M., Pascanu, R., et al. Uncovering mesa-optimization algorithms in Transformers. *ArXiv preprint*, abs/2309.05858, 2023b.

Wen, K., Li, Y., Liu, B., and Risteski, A. Transformers are uninterpretable with myopic methods: A case study with bounded Dyck grammars. *ArXiv preprint*, abs/2312.01429, 2023. URL https://arxiv.org/abs/2312.01429.

Xie, S. M., Raghunathan, A., Liang, P., and Ma, T. An explanation of in-context learning as implicit Bayesian inference. In *Proceedings of the International Conference on Learning Representations*, 2022.

Yang, S., Wang, B., Shen, Y., Panda, R., and Kim, Y. Gated linear attention Transformers with hardware-efficient training. *ArXiv preprint*, abs/2312.06635, 2023.

Zhai, S., Talbott, W., Srivastava, N., Huang, C., Goh, H., Zhang, R., and Susskind, J. An attention free Transformer. *ArXiv preprint*, abs/2105.14103, 2021.

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

A. Model Architectures

A.1. Overview

Linear Time Invariant (LTI) Models Linear and time invariant models usually have both recurrent and convolutional forms. Concretely, their recurrences are linear and the parameters do not change over time, as in $\boldsymbol{h}'_i = \boldsymbol{A}\boldsymbol{h}'_{i-1} + \boldsymbol{B}\boldsymbol{x}_i$, where $\boldsymbol{A}, \boldsymbol{B}$ are learnable matrices and σ is an identity function and thus omitted. A representative example is the **S4** model (Gu et al., 2022b, along with its many variants, e.g., Gu et al., 2022a; Mehta et al., 2022), which demonstrates competitive performance of such forms on sequence modeling.

Extending the LTI principle, models with weak linear time invariance (WLTI) allow for a time-varying component that can still be captured convolutionally after a suitable transformation. When $\boldsymbol{h}'_i = \boldsymbol{A}\boldsymbol{h}'_{i-1} + \boldsymbol{B}(\boldsymbol{x}_i)\boldsymbol{x}_i$, the $\boldsymbol{B}$ is not time-invariant any more, but if it can be written in the form $\boldsymbol{h}'_i = \boldsymbol{A}\boldsymbol{h}'_{i-1} + \boldsymbol{B}\phi(\boldsymbol{x}_i)$, it may still be viewed as LTI over the transformed input $\phi(\boldsymbol{x}_i)$. Next, we list two kinds of WLTI networks from the literature.

Linear Attention as Weak Time-Invariance Linear attention networks such as **Linear Transformers** (Katharopoulos et al., 2020) and **Retentive Networks** (RetNets, Sun et al., 2023) can be represented by recurrent dynamics wherein hidden states are updated through the accumulation of key-value outer products, as in $\mathbf{S}_i = \lambda\mathbf{S}_{i-1} + \boldsymbol{k}_i^\intercal \boldsymbol{v}_i$, where $\mathbf{S}$ denotes 2-D hidden states, and $\boldsymbol{k}, \boldsymbol{v}$ denote key/value vectors (as in attentional networks) respectively. Since the input of the recurrence (i.e, $\boldsymbol{k}, \boldsymbol{v}$) is dependent on $\boldsymbol{x}$ and λ is fixed, we characterize them as WLTI networks. Moreover, this recurrent perspective reveals that the effective capacity of the hidden states scales quadratically with their dimensionality, offering a potential explanation for the performance of these models.

Weak Linear Time Invariant Convolutional Models The **H3** model (Fu et al., 2023) combines linear attention with a convolutional S4-like layer, resulting in a more complex recurrent form than linear attention with similar input-dependent construction for input (see Appendix A.10). The **RWKV** model (Peng et al., 2023), though not originally presented in a convolutional form, adheres to the LTI principle and can be decomposed into a pair of state space models (SSMs), suggesting a potential for convolutional reformulation (Gu & Dao, 2023; see Appendix A.8). The **Hyena** model (Poli et al., 2023) is a fully convolutional model with data-dependent gating, as in $\boldsymbol{h}'_i = \phi(\boldsymbol{x}_i)(\boldsymbol{l} * \boldsymbol{x}_{<i})$, where $\phi(\boldsymbol{x}_i)$ denotes a non-linear mapping of $\boldsymbol{x}_i$. We also characterize Hyena as WLTI considering the additional input-dependent mapping $\phi(\boldsymbol{x}_i)$, compared to vanilla LTI convolutional models.

Linear Time Variant Models Recent state-of-the-art models, **Mamba** (Gu & Dao, 2023) and **GLA Transformer** (Yang et al., 2023), feature fully time-dependent parameterization in both the recurrent and input transformations, as in $\boldsymbol{h}'_i = \boldsymbol{A}(\boldsymbol{x}_i)\boldsymbol{h}'_{i-1} + \boldsymbol{B}(\boldsymbol{x}_i)\boldsymbol{x}_i$. Their time-variant nature allows for a more flexible and adaptive response to the input sequence, potentially more expressive in terms of sequence modeling. However, these models represent a departure from the LTI framework and pose new challenges for efficient training, as they do not conform to convolutional structures.

Non-Linear Time Variant Models Finally, we also include **LSTM**s, which were the most widely used sequence models in the pre-Transformer era. LSTMs (Hochreiter & Schmidhuber, 1997) feature a complex mapping (σ) based on gating and intermediate cell states in the recurrence, apart from input-dependent parameterizations, as in $\boldsymbol{h}'_i = \sigma\big(\boldsymbol{A}(\boldsymbol{x}_i)\boldsymbol{h}'_{i-1} + \boldsymbol{B}(\boldsymbol{x}_i)\boldsymbol{x}_i\big)$ (see Appendix A.7). Such non-linear time variant models are incompatible with efficient parallel training due to complex dependencies among hidden states. Nevertheless, we include LSTMs to examine their expressivity in comparison to recent sequence models, especially other recurrent models.

A.2. Modeling Overview

We experiment with a set of neural autoregressive sequence models that generate sequences via a product of conditional distributions $\Pi_{i=1}^{T} p_m\left(\boldsymbol{x}_{i+1} \mid \boldsymbol{x}_{1:i}\right)$. We characterize each model with three common consecutive modules: (i) an **embedding layer** that maps input tokens to vectors, (ii) a stack of **backbone layers**, (iii) and an **output projection** layer that maps the final outputs from the backbone to the distribution over the token space:

$$p_m\left(\boldsymbol{x}_{i+1} \mid \boldsymbol{x}_{1:i}\right) = \mathrm{softmax}(m_{\text{output}} \circ m_{\text{backbone}} \circ m_{\text{embed}}\left(\boldsymbol{x}_{1:i}\right)) \tag{12}$$

Collectively, the three modules form a mapping from one-hot vectors to pre-softmax logits, i.e., $\{0,1\}^{T\times|V|} \to \mathbb{R}^{T\times|V|}$ where T denotes the sequence length and $|V|$ denotes the vocabulary size.

Embedding Layer (m_{embed}) The embedding layer is a single projection matrix which converts one-hot input vectors to dense vectors:

$$m_{\text{embed}}(\boldsymbol{x}) = \boldsymbol{W}_e\, \boldsymbol{x}. \tag{13}$$

In almost all models, positional information need not be provided in the embedding layer. Instead, it may either be explicitly incorporated into attention mechanisms (as in rotary positional embeddings; Su et al., 2024), or implicitly provided via recurrent / convolutional updates. The only exception is the original Transformer, in which positional information is added via learned positional embeddings:

$$m_{\text{embed}}(\boldsymbol{x})_i = \boldsymbol{W}_e\, \boldsymbol{x}_i + \boldsymbol{W}_p\, \boldsymbol{i}, \tag{14}$$

where $\boldsymbol{i}$ is one-hot representation of the time-step i. We use the original Transformer in our synthetic experiments, and use the improved Transformer with Rope in our language modelling experiments on real data.

Backbone Layers ($m^{(l)}_{\text{backbone}}$) Each backbone layer updates the previous layer's hidden outputs ($\boldsymbol{h}^{(l-1)}$) in two sequential steps. First, a **token mixer** that models the token-level interactions, which may be implemented as any *causal* network mapping hidden states of previous tokens to a new hidden state for the current token:

$$\boldsymbol{a}^{(l)} = m^{(l)}_{\text{mixer}}\left(\boldsymbol{h}^{(l-1)}\right), \tag{15}$$

where $m^{(l)}_{\text{mixer}}\left(\boldsymbol{h}^{(l-1)}\right)_i = m^{(l)}_{\text{mixer}}\left(\boldsymbol{h}^{(l-1)}_{1:i}\right)_t$. Then, a **feed-forward** network, $m^{(l)}_{\text{FF}}$, applied to the final outputs of the layer with a residual connection:

$$m^{(l)}_{\text{backbone}}(\boldsymbol{h}^{(l-1)}) = m^{(l)}_{\text{FF}}\left(\boldsymbol{a}^{(l)}\right) + \boldsymbol{a}^{(l)}. \tag{16}$$

or alternatively in gated attention unit (GAU, Hua et al., 2022):

$$m^{(l)}_{\text{backbone}}(\boldsymbol{h}^{(l-1)}) = m^{(l)}_{\text{GAU}}\left(\boldsymbol{h}^{(l-1)}, \boldsymbol{a}^{(l)}\right) + \boldsymbol{a}^{(l)}. \tag{17}$$

where $m_{\text{GAU}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{W}_3(\boldsymbol{W}_1\boldsymbol{x}) \odot (\boldsymbol{W}_2\boldsymbol{y})$, and $\odot$ denotes element-wise product. [9] In contrast to the token mixer, the feed-forward network is applied individually for each token, i.e., there is no token-wise interaction.

Output Projection (m_{output}) The projection layer consists of a single fully-connected projection that maps outputs of the backbone $\boldsymbol{h}^{(L)}$ to the output space:

$$m_{\text{output}}(\boldsymbol{h}^{(L)}) = \mathbf{W}_o\, \boldsymbol{h}^{(L)} \tag{18}$$

In the following sections, we review the model architectures studied in this work. They share the common skeleton presented above, and mainly differ in the design of the token-mixing module m_{mixer}.[10] We intend to present a unified view of all models by using shared notation and equivalent forms (when possible).

We will denote input/output of a token mixer as $\boldsymbol{x} \in \mathbb{R}^{L\times d}$ and $\boldsymbol{y} \in \mathbb{R}^{L\times d}$, respectively. When possible, we will present all the possible forms (i.e., attention-style form, recurrent form and convolutional form) of a model. Generally, attentional or convolutional forms are useful for training, whereas recurrent forms are useful for efficient inference.

A.3. Transformers with Self Attention (Vaswani et al., 2017)

Standard self-attention performs token mixing in the following way:

$$\boldsymbol{q}_i, \boldsymbol{k}_j = \boldsymbol{W}_q\boldsymbol{x}_i, \boldsymbol{W}_k\boldsymbol{x}_j \in \mathbb{R}^{d_k} \tag{19}$$

$$\boldsymbol{A}_{ij} \propto \exp(\langle \boldsymbol{q}_i, \boldsymbol{k}_j\rangle) \in (0,1) \qquad \text{softmax attention} \tag{20}$$

$$\boldsymbol{v}_j = \boldsymbol{W}_v\boldsymbol{x}_j \in \mathbb{R}^{d_v} \tag{21}$$

$$\boldsymbol{z}_i = \sum_{j=1}^{i} \boldsymbol{A}_{ij}\boldsymbol{v}_j \in \mathbb{R}^{d_v} \tag{22}$$

$$\boldsymbol{y}_i = \boldsymbol{W}_o\boldsymbol{z}_i \in \mathbb{R}^{d} \tag{23}$$

[9] Among all the models presented, only Mamba employs the GAU architecture.

[10] For brevity, we omit the normalization layers which are applied before each token mixer and feed-forward layer.

where d_k, d_v denote the dimension for query/key and value vectors, respectively. The attention scores are computed based on the pairwise dot-product between the query vector of the current token and key vectors from the context. In the multi-head attention, attention output $\boldsymbol{z}_i$ is independently computed in each head; all outputs are concatenated as the final attention output, which will then be fed into the output projection $\boldsymbol{W}_o$.

A.4. Transformers with Linear Attention (Katharopoulos et al., 2020)

Linear attention (Katharopoulos et al., 2020) simplifies standard attention by replacing $\exp(\langle \boldsymbol{q}_i, \boldsymbol{k}_j \rangle)$ with a kernel map $k(\boldsymbol{q}_i, \boldsymbol{k}_j)$ with an associative feature map (i.e., $k(\boldsymbol{q}_i, \boldsymbol{k}_j) = \phi(\boldsymbol{q}_i)\phi(\boldsymbol{k}_j)$). In this work, we consider a simple feature map of identity function (i.e., $\phi(\boldsymbol{q}_i) = \boldsymbol{q}_i$), which yields surprisingly good performance for language model on real data in recent works (Qin et al., 2022; Yang et al., 2023). With this feature map, the token mixing process is very similar to standard attention, except that attention scores are not normalized.

$$\boldsymbol{q}_i, \boldsymbol{k}_j = \boldsymbol{W}_q \boldsymbol{x}_i, \boldsymbol{W}_k \boldsymbol{x}_j \in \mathbb{R}^{d_k} \tag{24}$$

$$\boldsymbol{A}_{ij} = \langle \boldsymbol{q}_i, \boldsymbol{k}_j \rangle \qquad \text{linear attention} \tag{25}$$

$$\boldsymbol{v}_j = \boldsymbol{W}_v \boldsymbol{x}_j \in \mathbb{R}^{d_v} \tag{26}$$

$$\boldsymbol{z}_i = \sum_{j=1}^{i} \boldsymbol{A}_{ij} \boldsymbol{v}_j \in \mathbb{R}^{d_v} \tag{27}$$

$$\boldsymbol{y}_i = \boldsymbol{W}_o \boldsymbol{z}_i \in \mathbb{R}^{d} \tag{28}$$

The linear attention has an equivalent recurrent form as follows.

$$\boldsymbol{q}_i, \boldsymbol{k}_j = \boldsymbol{W}_q \boldsymbol{x}_i, \boldsymbol{W}_k \boldsymbol{x}_j \in \mathbb{R}^{d_k} \tag{29}$$

$$\mathbf{S}_i = \mathbf{S}_{i-1} + \boldsymbol{k}_i^\intercal \boldsymbol{v}_i \in \mathbb{R}^{d_k \times d_v} \tag{30}$$

$$\boldsymbol{z}_i = \boldsymbol{q}_i^\intercal \mathbf{S}_i \in \mathbb{R}^{d_v} \tag{31}$$

(the rest is the same as the attentional form)

where $\mathbf{S}$ is the 2-D hidden state of the linear recurrence.

A.5. RetNet (Sun et al., 2023; Qin et al., 2022)

Based on linear attention, RetNet [11] further incorporates rotary positional embeddings(Su et al., 2024) and a fixed decay rate λ. The resulting token mixer, called *retention*, has the following form:

$$\boldsymbol{q}_i, \boldsymbol{k}_j = \boldsymbol{W}_q \boldsymbol{x}_i, \boldsymbol{W}_k \boldsymbol{x}_j \in \mathbb{R}^{d_k} \tag{32}$$

$$\tilde{\boldsymbol{q}}_i, \tilde{\boldsymbol{k}}_j = \text{RoPE}(\boldsymbol{q}_{1:i}), \text{RoPE}(\boldsymbol{k}_{1:j}) \in \mathbb{R}^{d_k} \tag{33}$$

$$\boldsymbol{A}_{ij} = \lambda^{i-j} \langle \tilde{\boldsymbol{q}}_i, \tilde{\boldsymbol{k}}_j \rangle \tag{34}$$

$$\boldsymbol{v}_j = \boldsymbol{W}_v \boldsymbol{x}_j \in \mathbb{R}^{d_v} \tag{35}$$

$$\boldsymbol{z}_i = \text{retention}(\boldsymbol{x}_{1:i}) = \sum_{j=1}^{i} \boldsymbol{A}_{ij} \boldsymbol{v}_j \in \mathbb{R}^{d_v} \tag{36}$$

$$\boldsymbol{r}_i = \boldsymbol{W}_r \boldsymbol{x}_i \in \mathbb{R}^{d_v} \tag{37}$$

$$\boldsymbol{y}_i = \boldsymbol{W}_o \big(\text{swish}(\boldsymbol{r}_i) \odot \boldsymbol{z}_i \big) \in \mathbb{R}^{d} \tag{38}$$

While the addition of rotary positional embedding to query/key vectors is straightforward in this attention-style form, the additional decay term λ is easier to understand in this equivalent recurrent form. The linear attention has an equivalent

[11] TransNormer proposed in Qin et al. (2022) has almost the same architecture as RetNet.

recurrent form as follows:

$$\boldsymbol{q}_i, \boldsymbol{k}_j = \boldsymbol{W}_q \boldsymbol{x}_i, \boldsymbol{W}_k \boldsymbol{x}_j \in \mathbb{R}^{d_k} \tag{39}$$

$$\tilde{\boldsymbol{q}}_i, \tilde{\boldsymbol{k}}_j = \text{RoPE}(\boldsymbol{q}_{1:i}), \text{RoPE}(\boldsymbol{k}_{1:j}) \in \mathbb{R}^{d_k} \tag{40}$$

$$\mathbf{S}_i = \lambda \mathbf{S}_{i-1} + \tilde{\boldsymbol{k}}_i^\intercal \boldsymbol{v}_i \in \mathbb{R}^{d_k \times d_v} \tag{41}$$

$$\boldsymbol{z}_i = \tilde{\boldsymbol{q}}_i^\intercal \mathbf{S}_i \in \mathbb{R}^{d_v} \tag{42}$$

(the rest is the same as the attentional form)

A.6. GLA (Yang et al., 2023)

Compared with RetNet, the GLA architecture incorporates more fine-grained data-dependent gating. Instead of using rotary positional embeddings, these gates can implicitly capture positional information. For the ease of understanding, we first show the recurrent form of GLA, and then its attention-style form.

For each token, GLA additionally relies on two data dependent decay vectors $\boldsymbol{\alpha}_i \in \mathbb{R}^{d_k}$ and $\boldsymbol{\beta}_i \in \mathbb{R}^{d_v}$. The outer-product of these vectors (i.e., $\boldsymbol{\alpha}_i^\intercal \boldsymbol{\beta}_i$) decides how much information to preserve from previous hidden state $\mathbf{S}_{i-1}$.

$$\boldsymbol{q}_i, \boldsymbol{k}_j = \boldsymbol{W}_q \boldsymbol{x}_i, \boldsymbol{W}_k \boldsymbol{x}_j \in \mathbb{R}^{d_k} \tag{43}$$

$$\boldsymbol{\alpha}_i = \sigma(\boldsymbol{W}_\alpha \boldsymbol{x}_i) \in \mathbb{R}^{d_k} \quad \boldsymbol{\beta}_j = \sigma(\boldsymbol{W}_{\boldsymbol{\beta}} \boldsymbol{x}_j) \in \mathbb{R}^{d_v} \tag{44}$$

$$\boldsymbol{v}_i = \boldsymbol{W}_v \boldsymbol{x}_i \in \mathbb{R}^{d_v} \tag{45}$$

$$\mathbf{S}_i = \boldsymbol{\alpha}_i^\intercal \boldsymbol{\beta}_i \odot \mathbf{S}_{i-1} + \boldsymbol{k}_i^\intercal \boldsymbol{v}_i \in \mathbb{R}^{d_k \times d_v} \tag{46}$$

$$\boldsymbol{z}_i = \boldsymbol{q}_i^\intercal \mathbf{S}_i \in \mathbb{R}^{d_v} \tag{47}$$

$$\boldsymbol{r}_i = \boldsymbol{W}_r \boldsymbol{x}_i \in \mathbb{R}^{d_v} \tag{48}$$

$$\boldsymbol{y}_i = \boldsymbol{W}_o\big(\text{swish}(\boldsymbol{r}_i) \odot \boldsymbol{z}_i\big) \in \mathbb{R}^{d} \tag{49}$$

Like linear attention and RetNet, GLA also has the following attentional form.[12]

$$\boldsymbol{q}_i, \boldsymbol{k}_j = \boldsymbol{W}_q \boldsymbol{x}_i, \boldsymbol{W}_k \boldsymbol{x}_j \in \mathbb{R}^{d_k} \tag{50}$$

$$\boldsymbol{\alpha}_i = \sigma(\boldsymbol{W}_\alpha \boldsymbol{x}_i) \in \mathbb{R}^{d_k} \quad \boldsymbol{\beta}_j = \sigma(\boldsymbol{W}_{\boldsymbol{\beta}} \boldsymbol{x}_j) \in \mathbb{R}^{d_v} \tag{51}$$

$$\boldsymbol{a}_i = \prod_i \boldsymbol{\alpha}_{1:i} \in \mathbb{R}^{d_k} \quad \boldsymbol{b}_j = \prod_j \boldsymbol{\beta}_{1:j} \in \mathbb{R}^{d_v} \tag{52}$$

$$\boldsymbol{v}_j = \boldsymbol{W}_v \boldsymbol{x}_j \in \mathbb{R}^{d_v} \tag{53}$$

$$\tilde{\boldsymbol{q}}_i = \boldsymbol{q}_i \odot \boldsymbol{a}_i \in \mathbb{R}^{d_k} \quad \tilde{\boldsymbol{k}}_j = \boldsymbol{k}_j / \boldsymbol{a}_j \in \mathbb{R}^{d_k} \quad \tilde{\boldsymbol{v}}_j = \boldsymbol{v}_j \odot \boldsymbol{b}_j \in \mathbb{R}^{d_v} \tag{54}$$

$$\boldsymbol{z}_i = \text{gla}(\boldsymbol{x}_{1:i}) = \big(\sum_{j=1}^{i} \boldsymbol{A}_{ij} \boldsymbol{v}_j\big) / \boldsymbol{b}_i \in \mathbb{R}^{d_v} \tag{55}$$

(the rest is the same as the recurrent form)

Here $\odot$ and $/$ denote element-wise multiplication and division; σ denotes a sigmoid function.

Connections among Linear Attention, RetNet, GLA RetNet and GLA both inherit the basic linear recurrence with 2-D hidden states from linear attention. GLA and RetNet mainly differ in the introduction of a decay term to their recurrent forms: GLA incorporates fine-grained data-dependent gates ($\boldsymbol{\alpha}, \boldsymbol{\beta}$) whereas RetNet uses a single fixed decay λ that is shared across all tokens and hidden dimensions. Moreover, RetNet and GLA incorporate an additional output gate $\boldsymbol{r}_i$ before the output projection $\boldsymbol{W}_o$. Such output gating is also used in the LSTM and RWKV models described next.

A.7. LSTM (Hochreiter & Schmidhuber, 1997)

All the recurrent models we have presented so fare are *linear*, in that the there are no non-linear dependencies between adjacent hidden states. We also consider LSTMs (Hochreiter & Schmidhuber, 1997), a widely used class of non-linear

[12] Please refer to the original paper for the derivation.

recurrent models with the form:

$$\boldsymbol{f}_i = \sigma(\mathbf{W}_f \boldsymbol{x}_i + \mathbf{U}_f \boldsymbol{h}_{i-1}) \in \mathbb{R}^d \tag{56}$$
$$\boldsymbol{i}_i = \sigma(\mathbf{W}_i \boldsymbol{x}_i + \mathbf{U}_i \boldsymbol{h}_{i-1}) \in \mathbb{R}^d \tag{57}$$
$$\boldsymbol{o}_i = \sigma(\mathbf{W}_o \boldsymbol{x}_i + \mathbf{U}_o \boldsymbol{h}_{i-1}) \in \mathbb{R}^d \tag{58}$$
$$\tilde{\boldsymbol{c}}_i = \tanh(\mathbf{W}_c \boldsymbol{x}_i + \mathbf{U}_c \boldsymbol{h}_{i-1}) \in \mathbb{R}^d \tag{59}$$
$$\boldsymbol{c}_i = \boldsymbol{f}_i \odot \boldsymbol{c}_{i-1} + \boldsymbol{i}_i \odot \tilde{\boldsymbol{c}}_i \in \mathbb{R}^d \tag{60}$$
$$\boldsymbol{y}_i = \boldsymbol{o}_i \odot \tanh(\boldsymbol{c}_i) \in \mathbb{R}^d \tag{61}$$

where $\boldsymbol{f}, \boldsymbol{i}, \boldsymbol{o}$ denote forget, input and output gates respectively. To strictly follow the architecture of traditional multi-layer LSTM, we do not use the feed-forward in-between LSTM layers, i.e., the input of layer l is directly the output from layer $l-1$.

A.8. RWKV (Peng et al., 2023)

The recurrence of RWKV is motivated by attention-free networks (Zhai et al., 2021). In contrast to linear attention models, this architecture uses a one-dimensional hidden state:

$$\boldsymbol{k}_i = \boldsymbol{W}_k \boldsymbol{x}_i \in \mathbb{R}^{d_v} \quad \boldsymbol{v}_i = \boldsymbol{W}_v \boldsymbol{x}_i \in \mathbb{R}^{d_v} \tag{62}$$
$$\boldsymbol{a}_i = \exp(-\boldsymbol{w}) \odot \boldsymbol{a}_{i-1} + \exp(\boldsymbol{k}_i) \odot \boldsymbol{v}_i \in \mathbb{R}^{d_v} \tag{63}$$
$$\boldsymbol{b}_i = \exp(-\boldsymbol{w}) \odot \boldsymbol{b}_{i-1} + \exp(\boldsymbol{k}_i) \in \mathbb{R}^{d_v} \tag{64}$$
$$\boldsymbol{z}_i = \mathrm{wkv}(\boldsymbol{x}_{1:i}) = \frac{\boldsymbol{a}_{i-1} + \exp(\boldsymbol{k}_i + \boldsymbol{u}) \odot \boldsymbol{v}_i}{\boldsymbol{b}_{i-1} + \exp(\boldsymbol{k}_i + \boldsymbol{u})} \in \mathbb{R}^{d_v} \tag{65}$$
$$\boldsymbol{r}_i = \boldsymbol{W}_r \boldsymbol{x}_i \in \mathbb{R}^{d_v} \tag{66}$$
$$\boldsymbol{y}_i = \boldsymbol{W}_o\big(\sigma(\boldsymbol{r}_i) \odot \boldsymbol{z}_i\big) \in \mathbb{R}^{d_v} \tag{67}$$

where $\boldsymbol{w}, \boldsymbol{u} \in \mathbb{R}^{d_v}$ are learnable parameters and σ is an activation function. The WKV operators maintain a recurrence with a pair of states $(\boldsymbol{a}_i, \boldsymbol{b}_i)$. Unlike linear attention, in which the 2D hidden state is constructed via an outer-product $\boldsymbol{k}_i^\intercal \boldsymbol{v}_i$, WKV uses an element-wise dot-product $\exp(\boldsymbol{k}_i) \odot \boldsymbol{v}_i$, so key and value vectors have the same shape.

Since the decay term $\boldsymbol{w}$ is not data-dependent, WKV also has the following equivalent convolutional form:

$$\boldsymbol{k}_i = \boldsymbol{W}_k \boldsymbol{x}_i \in \mathbb{R}^{d_v} \quad \boldsymbol{v}_i = \boldsymbol{W}_v \boldsymbol{x}_i \in \mathbb{R}^{d_v} \tag{68}$$
$$\tilde{\boldsymbol{kv}}_i = \exp(\boldsymbol{k}_i) \odot \boldsymbol{v}_i \in \mathbb{R}^{d_v} \quad \tilde{\boldsymbol{k}}_i = \exp(\boldsymbol{k}_i) \in \mathbb{R}^{d_v} \tag{69}$$
$$\boldsymbol{l}_i = \exp(-i\boldsymbol{w}) \in \mathbb{R}^{d_v} \tag{70}$$
$$\boldsymbol{a} = \boldsymbol{l} * \tilde{\boldsymbol{kv}} \in \mathbb{R}^{L \times d_v} \tag{71}$$
$$\boldsymbol{b} = \boldsymbol{l} * \tilde{\boldsymbol{k}} \in \mathbb{R}^{L \times d_v} \tag{72}$$
(the rest is the same as the recurrent form)

where $*$ denotes batched long convolution operator, i.e., one dimension of the filter $\boldsymbol{h}[:, i] \in \mathbb{R}^{L \times 1}$ handles one corresponding dimension $\boldsymbol{a}[:, i], \boldsymbol{b}[:, i] \in \mathbb{R}^{L \times 1}$.

A.9. S4 (Gu et al., 2022b)

Structured state space models (S4) are a family of sequence models defined with four parameters $(\Delta, \mathbf{A}, \mathbf{B}, \mathbf{C})$. S4 models are typically represented as a sequence mapping $\mathbb{R}^{L \times 1} \to \mathbb{R}^{L \times 1}$, wherein the input and output are both scalars (i.e. $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^1$). In this case, S4 has the following recurrent form.

$$\boldsymbol{h}_i = \bar{\mathbf{A}} \boldsymbol{h}_{i-1} + \bar{\mathbf{B}} \boldsymbol{x}_i \in \mathbb{R}^{d_k} \tag{73}$$
$$\boldsymbol{y}_i = \mathbf{C} \boldsymbol{h}_i \in \mathbb{R}^1 \tag{74}$$

where d_{inner} denotes the dimension of hidden states $\boldsymbol{h}_i$, $\bar{\mathbf{A}} \in \mathbb{R}^{d_{\text{inner}} \times d_{\text{inner}}}$, $\bar{\mathbf{B}} \in \mathbb{R}^{d_{\text{inner}} \times 1}$ are transformed parameters for discrete sequence data according to a certain discretization rule, and $\mathbf{C} \in \mathbb{R}^{1 \times d_{\text{inner}}}$. Equivalently, S4 has the following convolutional form:

$$\bar{\mathbf{K}} = [\mathbf{C}\bar{\mathbf{B}}, \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{B}}, \dots \mathbf{C}\bar{\mathbf{A}}^{L-1}\bar{\mathbf{B}}] \tag{75}$$

$$\boldsymbol{y} = \boldsymbol{x} * \bar{\mathbf{K}} \tag{76}$$

where $*$ denotes the convolution operator and $\bar{\mathbf{K}}$ denotes a convolutional kernel. This form is critical for enabling efficient parallel training via Fast Fourier Transform (FFT) algorithms.

Since the recurrent forms of other models are usually presented with vector input/output (i.e., $\boldsymbol{x}_i, \boldsymbol{y}_i \in \mathbb{R}^d$), we may present S4's equivalent batched recurrent form as follows:

$$\mathbf{S}_i = \bar{\mathbf{A}} \circ \mathbf{S}_{i-1} + \bar{\mathbf{B}} \circ \boldsymbol{x}_i \in \mathbb{R}^{d \times d_{\text{inner}}} \tag{77}$$

$$\boldsymbol{y}_i = \mathbf{C} \circ \mathbf{S}_i \in \mathbb{R}^d \tag{78}$$

where $\mathbf{S}_i$ denotes a 2-D hidden state, $\bar{\mathbf{A}} \in \mathbb{R}^{d \times d_{\text{inner}} \times d_{\text{inner}}}$, $\bar{\mathbf{B}} \in \mathbb{R}^{d \times d_{\text{inner}} \times 1}$, $\mathbf{C} \in \mathbb{R}^{d \times 1 \times d_{\text{inner}}}$, $\circ$ denotes batched matrix multiplication.[13] In this batched form, d numbers of independent SSM run in parallel, each responsible for a dimension of input $\boldsymbol{x}$.

Connections to Linear Attention With this batched form in Equation (78), it becomes clear that S4, like linear attention, enjoys a large effective hidden states for recurrence. We can also draw a rough parallel between ($\bar{\mathbf{B}}$, $\mathbf{C}$) in S4 and ($\boldsymbol{q}_i, \boldsymbol{k}_j$) in linear attention as they handle the input and output for the recurrences, respectively. This parallel reveals the difference between S4 and linear attention. In S4, the input and output mapping is not data-dependent, i.e., ($\bar{\mathbf{B}}$, $\mathbf{C}$) does not depend on input $\boldsymbol{x}$. In comparison, $\boldsymbol{q}_i$ and $\boldsymbol{k}_j$ are linear mappings of the input $\boldsymbol{x}_i$.

A.10. H3 (Fu et al., 2023)

H3 is a mixture of state-space models and linear attention. In particular, it employs the outer-product structure from linear attention (i.e., $\boldsymbol{k}_i^T \boldsymbol{v}_i$) to construct the input of a state-space model:

$$\boldsymbol{k}_i = \boldsymbol{W}_k \boldsymbol{x}_i \in \mathbb{R}^{d_k} \quad \boldsymbol{v}_i = \boldsymbol{W}_v \boldsymbol{x}_i \in \mathbb{R}^{d_v} \tag{79}$$

$$\boldsymbol{x}'_i = \boldsymbol{k}_i^\intercal \boldsymbol{v}_i \in \mathbb{R}^{d_k \times d_v} \tag{80}$$

$$\mathbf{S}_i = \bar{\mathbf{A}} \odot \mathbf{S}_{i-1} + \bar{\mathbf{B}} \odot \boldsymbol{x}'_i \in \mathbb{R}^{d_k \times d_v \times d_{\text{inner}}} \tag{81}$$

$$\boldsymbol{z}'_i = \mathbf{C} \circ \mathbf{S}_i \in \mathbb{R}^{d_k \times d_v} \tag{82}$$

$$\boldsymbol{q}_i = \boldsymbol{W}_q \boldsymbol{x}_i \in \mathbb{R}^{d_k} \tag{83}$$

$$\boldsymbol{z}_i = \boldsymbol{q}_i \boldsymbol{z}'_i \in \mathbb{R}^{d_v} \tag{84}$$

$$\boldsymbol{y}_i = \boldsymbol{z}'_i \boldsymbol{W}_o \in \mathbb{R}^d \tag{85}$$

where $\bar{\mathbf{A}}$, $\bar{\mathbf{B}}$, $\mathbf{C} \in \mathbb{R}^{d_{\text{inner}}}$ are parameters of the state-space models, $\odot$ denotes element-wise product with broadcasting, $\circ$ denotes batched matrix-vector product. The SSM is diagonally parameterized (i.e., $\bar{\mathbf{A}}$ is a vector) and the original H3 paper additionally uses another shift-SSM (Fu et al., 2023) to further refine the key vector $\boldsymbol{k}_i$.

A.11. Hyena (Poli et al., 2023)

Hyena is a purely convolutional model that does not have an equivalent recurrent form, unlike S4. However, it recursively applies the convolution operator at the sequence level for N times. In practice, N is usually set to be 2, and the resulting

[13] To differentiate the size of hidden states in recurrences, we use $\boldsymbol{h}_i$ in the 1-D case and $\mathbf{S}_i$ in the 2-D case.

form is as follows:

$$\boldsymbol{v}^n = \boldsymbol{W}_n \boldsymbol{x} \in \mathbb{R}^{L\times d} \tag{86}$$

$$\boldsymbol{z}^0 = \boldsymbol{v}^0 \in \mathbb{R}^{L\times d} \tag{87}$$

$$\left.\begin{aligned} \boldsymbol{l}_i^n &= \mathrm{FFN}(\boldsymbol{i}) \in \mathbb{R}^{L\times d} \\ \boldsymbol{z}^n &= \boldsymbol{v}^{n-1} \odot (\boldsymbol{l}^n * \boldsymbol{z}^{n-1}) \in \mathbb{R}^{L\times d} \end{aligned}\right\} \text{ recursion } n = 1 \dots N \tag{88}$$

$$\boldsymbol{y} = \boldsymbol{z}^N \in \mathbb{R}^{L\times d} \tag{89}$$

where $*$ denotes batched convolution. In practice, the filter is padded to the size of $(2L-1)\times d$ so that the convolution operator becomes a circular convolution for efficient training using FFT. [14] Note that the resulting kernels $\boldsymbol{l}^1, \boldsymbol{l}^2$ do not depend on the input $\boldsymbol{x}$, but the convolution output is controlled by the data-dependent gate $\boldsymbol{v}^1, \boldsymbol{v}^2$ in Equation (88).

A.12. Mamba (Gu & Dao, 2023)

Mamba has the same recurrent form as S4, and uses data-dependent parameterization for $\bar{\mathbf{A}}$, $\bar{\mathbf{B}}$, $\mathbf{C}$:

$$\boldsymbol{v}_i = \boldsymbol{W}_v \boldsymbol{x}_i \in \mathbb{R}^{d_v} \tag{90}$$

$$\mathbf{S}_i = \bar{\mathbf{A}}_i \odot \mathbf{S}_{i-1} + \bar{\mathbf{B}}_i \odot \boldsymbol{v}_i \in \mathbb{R}^{d_k \times d_v} \quad (\odot \text{ with broadcast}) \tag{91}$$

$$\boldsymbol{y}_i = \mathbf{C}_i \mathbf{S}_i \in \mathbb{R}^{d_v} \tag{92}$$

where $\bar{\mathbf{A}}_i, \bar{\mathbf{B}}_i \in \mathbb{R}^{d_k \times d_v}$, $\mathbf{C}_i \in \mathbb{R}^{d_v}$ are data-dependently parameterized, i.e., computed based on $\boldsymbol{x}_i / \boldsymbol{v}_i$. However, due to the data-dependence, this recurrent form no longer has an equivalent convolutional form for efficient training. The original paper handles this issue with customized hardware-efficient training algorithms based on the recurrent form.

B. Optimization & Hyperparameter Search

Hyper Parameter	Search
hidden size	[64, 128, 256, 512, 1024]
number of layers	[1, 2, 4, 8, 12]
number of heads	[1, 2, 4]
epochs	[200, 400]
batch size	32
optimizer	[AdamW]
└ learning rate	[1e-4, 2.5e-4]
└ weight decay	[0.01, 0.1]
└ βs	[(0.9, 0.99)]
scheduler	Cosine Scheduler with Warmup
└ minimum learning rate	2.5e-5
└ warm-up start learning rate	1e-7
└ warm-up steps	25000

Table 3. Hyper-parameter search space for neural models.

We perform exhaustive search over the grid of hyper-parameters in Table 3 and pick the best setting best on validation set on ICLL and AR seperately. In AR we search through hidden sizes up to 256. In ICLL, we search first up to a hidden size of 256; if the best performing hidden size is 256, we try 512, and then 1024. We also use only the best-performing weight decay of 0.1 and learning rate of 2.5e-4 in the additional search runs.

[14] Please refer to Section 2 and 3 of (Poli et al., 2023) for details.

C. Algorithms

In this section, we share the details of implemented algorithms in Section 4. We use $\boldsymbol{x}$ to denote problem instances $\boldsymbol{d}$ for simplicity.

C.1. In-context N-gram Language Model

Algorithm 1 In-context n-gram language model with back-off

1: **Input:** Current prefix of $s = \boldsymbol{x}_{1:i-1}$ input, n-gram order n
2: **Output:** Next token distribution
3: *// Create in-context corpus from the prefix*
4: $\mathcal{D} \leftarrow \text{map}(\text{add\_padding\_character}, \text{s.split(' • ')})$
5: *// Build all n-gram token counts up to order n*
6: $c \leftarrow \text{count\_all\_n\_grams}(\mathcal{D}; n)$
7: *// Apply smoothing*
8: $c^* \leftarrow \text{smoothing}(c)$
9: *// To compute $P(x_i \mid \boldsymbol{x}_{i-N+1}^{i-1})$ as :*
10: **if** $c(\boldsymbol{x}_{i-N+1}^{i}) > 0$ **then**
11: return $P(x_i \mid \boldsymbol{x}_{i-N+1}^{i-1}) = \frac{c^*(\boldsymbol{x}_{i-N+1}^{i})}{c^*(\boldsymbol{x}_{i-N+1}^{i-1})}$
12: **else**
13: *// Backoff to lower order model*
14: return $P(x_i \mid \boldsymbol{x}_{i-N+1}^{i-1}) = \alpha(\boldsymbol{x}_{i-N+1}^{i-1}) P(x_i \mid \boldsymbol{x}_{i-N+2}^{i-1})$
15: **end if**

In Algorithm 1, we present an **in-context** applied n-gram model that incorporates a back-off mechanism (Chen & Goodman, 1996). This model differs from the standard n-gram approach that is trained on a fixed training set. Instead, here we train a unique n-gram for each example at each time step to predict the subsequent word. The back-off strategy allows for the assignment of non-zero probabilities to unseen n-grams by utilizing information from lower-order n-grams, as shown in line 14 of Algorithm 1. For each n-gram context $\boldsymbol{x}_{i-N+1}^{i-1}$, the back-off weight $\beta(\boldsymbol{x}_{i-N+1}^{i-1})$ can be computed as follows:

$$\beta(\boldsymbol{x}_{i-N+1}^{i-1}) = 1 - \sum_{\{w \mid c(\boldsymbol{x}_{i-N+1}^{i-1} w) > 0\}} \frac{c^*(\boldsymbol{x}_{i-N+1}^{i-1} w)}{c^*(\boldsymbol{x}_{i-N+1}^{i-1})} \tag{93}$$

In the absence of smoothing, the summation is expected to equal 1, resulting in β being 0. Smoothing techniques, such as Laplace smoothing, modify the counts and allocate a probability mass for unseen n-grams. Alternatively, by excluding the probability corresponding to the padding token w, we can reserve probability mass for back-off without explicit smoothing. This approach is employed in our n-gram model implementation and it worked slightly better than add-one smoothing.

Finally, the back-off weights α are calculated by normalizing beta for the lower-order n-gram probabilities for the unseen current n-gram sequences:

$$\alpha(\boldsymbol{x}_{i-N+1}^{i-1}) = \frac{\beta(\boldsymbol{x}_{i-N+1}^{i-1})}{\sum_{\{w \mid c(\boldsymbol{x}_{i-N+1}^{i-1} w) = 0\}} P(w \mid \boldsymbol{x}_{i-N+2}^{i-1})} \tag{94}$$

It is important to note that the normalization ensures that the probabilities of all potential continuations of a given context sum to one.

C.2. In-context Baum-Welch HMM Language Model

Given a NFPA from REGBENCH, we can construct a Hidden Markov Model (HMM) that assigns the same probabilities to

Algorithm 2 In-context Baum-Welch HMM language model

Input: Current prefix of $s = \boldsymbol{x}_{1:i-1}$ input, number of states $|S|$, a vocabulary $\mathcal{V}$, a maximum number of iterations N.
Output: Next token distribution
// initialize the corpus from the prefix
$O \leftarrow$ s.split(' • ')
// initialize the HMM parameters given the number of states and vocabulary
$\lambda \leftarrow (A, B, \pi)$
for N times **do**
  // expectation step (E-step)
  // mask A to not have self transitions (see Appendix C.3)
  $A \leftarrow$ mask_transitions(A)
  // mask π to start at state 0 (see Appendix C.4)
  $\pi \leftarrow$ mask_pi(π)
  $\xi, \gamma, b \leftarrow \mathbf{0}, \mathbf{0}, \mathbf{0}$
  for each observation sequence O^n **do**
    // run forward-backward to get α and β
    $\alpha, \beta \leftarrow$ forward-backward$(O^n \mid \lambda)$
    *// accumulate expected values **for all** time steps t, states l, next states m, tokens k*
    $P(q_t = s_l \mid O^n, \lambda) \propto \alpha_t(l)\beta_t(l)$
    $P(q_t = s_l, q_{t+1} = s_m \mid O^n, \lambda) \propto \alpha_t(l)A_{lm}\beta_{t+1}(m)B_{mk}$
    // expected states
    $\gamma_t(l) \leftarrow \gamma_t(l) + P(q_t = s_l \mid O^n, \lambda)$
    // expected transitions
    $\xi_t(l, m) \leftarrow \xi_t(l, m) + P(q_t = s_l, q_{t+1} = s_m \mid O^n, \lambda)$
    // expected emissions
    $b(m, k) \leftarrow b(m, k) + 1_{O_t^n = k} P(q_t = s_m \mid O^n, \lambda)$
  end for
  // maximization step (M-step)
  // update state transition probabilities
  $A_{lm} \leftarrow \frac{\sum_{t=1}^{T-1} \xi_t(l,m)}{\sum_{t=1}^{T-1} \gamma_t(l)}$
  // update emission probabilities
  $B_{mk} \leftarrow \frac{b_{mk}}{\sum_{t=1}^{T} \gamma_t(m)}$
  // update initial state probabilities
  $\pi_l \leftarrow \gamma_1(l)/|O|$
end for
// Prediction Step
// Run forward algorithm on the last observation to get α
$\alpha \leftarrow$ forward$(O^{\text{last}} \mid \lambda)$
$p(x_i s) \propto \alpha_{|O^{\text{last}}|}(l) A_{lm} B_{mx_i}$
return $p(x_i)$

any given string[15]. The construction can be done as:

- For each pair of states $S_i, S_j \in \mathcal{S}$, create a corresponding HMM state $H_{(S_i,S_j)}$.
- Define the transition probabilities $A(H_{(S_i,S_j)}, H_{(S_l,S_m)}) \propto 1[j = l]\, T_{\text{PFA}}(S_i, w, S_j)$, where each character w transitions to a unique state in the PFA.
- Set the emission probabilities $B(H_{(S_i,S_j)}, w) = 1[T_{\text{PFA}}(S_i, w, S_j) > 0]$, and 0 otherwise.

[15] Please refer to Dupont et al. (2005) for equivalence of NFPAs and hidden Markov models.

- Set initial state probabilities 1 for the start states and 0 for the others: $\pi(H_{(S_i,S_j)}) = 1[i = 1]$.

The number of states in the constructed HMM is the square of the number of states in the probabilistic automaton. Therefore, we fit an HMM to the examples in REGBENCH with a maximum of $|S| = 12^2 = 144$ states. Algorithm 2 details the in-context Baum-Welch predictor. We begin by constructing a list of observations from the current prefix $x_{1:i-1}$, and then fit an HMM given the global vocabulary $\mathcal{V}$ and number of states $|S| = 144$ using an improved Baum-Welch algorithm that is consistent with the structure of probabilistic automata in REGBENCH. We incorporate two pieces of prior information about the dataset:

C.3. Masking A to enforce state transitions

In our construction, we assume $A_{H_{(S_i,S_j)}H_{(S_l,S_m)}} = 0$ if $j \neq l$. We enforce this constraint in each iteration by masking the corresponding entries in A. Additionally, as our PFA sampling schema in Section 3.1 does not include self-transitions, we set all $A_{H_{(S_i,S_i)}H_{(S_i,S_l)}} = 0$.

C.4. Masking π to start at the initial state

All our PFAs have a single start state, which we denote as $H_{(S_0,S_i)}$ for all i, without loss of generality. We mask all other initial state probabilities such that $\pi(H_{(S_0,S_i)}) = 0$ for $i \neq 0$.

In our experiments, these masking strategies significantly improved the accuracy of the Baum-Welch algorithm. The results presented are based on this modified BW algorithm.

D. Learned MLP Reweighting

We provide the training details of MLP n-gram reweighting models' used in Section 5.3, namely LNW, LNW$_r$, and LNW$_b$.

Count Features (LNW Model) Given a sequence $\boldsymbol{x}$, we first extract n-gram features for each position i and for each n-gram length n:

$$\text{gram}(i; n) = \left[\text{count}_{\boldsymbol{x}_{1:i}}(\boldsymbol{x}_{i-n}^{i-1} w) - 1 \quad \forall w \in \mathcal{V}\right] \in \mathcal{Z}^{|\mathcal{V}|} \tag{95}$$

The full set of n-gram features is the concatenation of n-gram features for $n = 1$ to $n = 3$:

$$\text{gram}(i; \leq 3) = \text{concatenate}(\text{gram}(i; 1), \text{gram}(i; 2), \text{gram}(i; 3)) \in \mathcal{Z}^{3|\mathcal{V}|} \tag{96}$$

We then train a sequence model that takes $\text{gram}(i; \leq 3)$ as input and applies a 2-layer MLP with GeLU activation to produce the unnormalized scores for the next token distribution. We use the same language modeling loss as used to train sequence models. Our hyper-parameters for MLP training are as follows:

hyper parameter	value
hidden size	1024
epochs	50
batch size	32
optimizer	Adam
└ learning rate	1e-3
└ βs	(0.9, 0.99)
scheduler	reduce on plateu
└ patiance	5 epochs
└ factor	0.5
└ minimum learning rate	1e-5

Frequency Features (LNW$_r$ Model) The frequency features model uses normalized n-gram features $\frac{\text{gram}(i;n)}{\sum_{\mathcal{V}} \text{gram}(i;n)}$ instead of the raw n-gram counts described above.

Binary Features (LNW$_b$ Model) The binary features model uses n-gram existence features, where $\text{gram}(i; n) = 1$ if the n-gram exists at position i and 0 otherwise, instead of raw n-gram features.

E. Probing Experiments

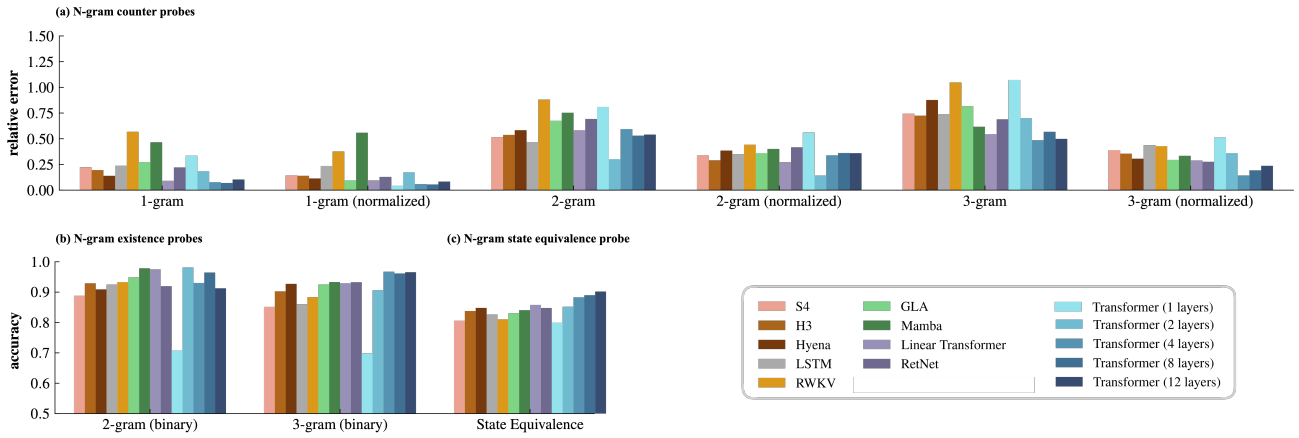


Figure 6. Additional results on probing analysis of n-gram representations with neural sequence models trained with $N_{\text{train}} = 40000$ examples. See Figure 4 for details.

Model and Objective We train a 2-layer Multilayer Perceptron (MLP) as our probe model in two configurations: **(1)** $f_{\text{n-gram}}(\boldsymbol{h}, \boldsymbol{c})$, where $\boldsymbol{h}$ represents a hidden state at a specific time step and $\boldsymbol{c}$ denotes the query n-gram; and **(2)** $f_{\text{equal}}(\boldsymbol{h}_i, \boldsymbol{h}_j)$, which is employed in state equivalence probes. The formulation of our $f_{\text{n-gram}}(\boldsymbol{h}, \boldsymbol{c})$ for an n-gram probe is as follows:

$$\boldsymbol{e}_c = \boldsymbol{W}_{\text{embed}}\boldsymbol{c} \in \mathbb{R}^{n \times d/2} \tag{97}$$

$$\boldsymbol{e}_c = \text{flatten}(\boldsymbol{e}_c) \in \mathbb{R}^{nd/2} \tag{98}$$

$$\boldsymbol{e}_h = \boldsymbol{W}_{\text{proj}}\boldsymbol{h} \in \mathbb{R}^{nd/2} \tag{99}$$

$$\boldsymbol{x} = \text{concatenate}([\boldsymbol{e}_c, \boldsymbol{e}_h, \boldsymbol{e}_c \odot \boldsymbol{e}_h]) \tag{100}$$

$$\boldsymbol{y} = \boldsymbol{W}_2\,\text{GeLU}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \tag{101}$$

where n is the order of the n-gram and d is the dimensionality of the hidden state used by the probe. For regression tasks, we employ the mean squared error loss on the output and our targets are corresponding n-gram counts, $\text{count}(\boldsymbol{x}^i_{t-n+1}c)$, or normalized counts (frequencies): $p(c \mid \boldsymbol{x}^i_{i-n+1}) = \frac{\text{count}(\boldsymbol{x}^i_{i-n+1}c)}{\text{count}(\boldsymbol{x}^i_{i-n+1})}$. For classification tasks, we utilize binary cross-entropy loss with the logits being $\boldsymbol{y}$, and targets are whether the corresponding n-gram exists or not.

Data We train the probe using hidden states extracted from the actual training set of the models. Specifically, we randomly select an example from the training set, randomly choose a time step within that example, and then create a query n-gram by appending a random next character to the last $n-1$ characters at the chosen time step. For regression tasks, we only consider n-grams that appear at least once in the prefix. Each epoch involves iterating over each example once. For testing, we apply the same sampling procedure using hidden states from the test set.

Model and Objective The state equivalence probe $f_{\text{equal}}(\boldsymbol{h}_i, \boldsymbol{h}_j)$ is defined as:

$$\boldsymbol{e}_i = \boldsymbol{W}_{\text{proj}}\boldsymbol{h}_i \in \mathbb{R}^d \tag{102}$$

$$\boldsymbol{e}_j = \boldsymbol{W}_{\text{proj}}\boldsymbol{h}_j \in \mathbb{R}^d \tag{103}$$

$$\boldsymbol{x} = \text{concatenate}([\boldsymbol{e}_i, \boldsymbol{e}_j, \boldsymbol{e}_i \odot \boldsymbol{e}_j]) \in \mathbb{R}^{3d} \tag{104}$$

$$\boldsymbol{y} = \boldsymbol{W}_2\,\text{GeLU}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \tag{105}$$

where d is the dimensionality of the hidden state used by the probe. We use the cross-entropy loss in the classification of whether two states are equivalent.

Data Similar to the n-gram probe, we train the state equivalence probe using hidden states from the model's training set. We randomly select an example and then sample two time steps within it, ensuring that in 50% of cases the probe receives

identical states, and in the remaining 50%, it receives different states. The testing procedure mirrors that of the training phase.

We employ the following hyperparameters for all probe training. We train separate probes for each layer and present the best results in Figure 4 and Figure 6.

hyper parameter	**value**
hidden size (d)	128
epochs	1000
batch size	64
optimizer	Adam
└ learning rate	3e-4
└ βs	(0.9, 0.99)
scheduler	Cosine Annealing
└ minimum learning rate	1e-4

F. Implementations of N-gram Layers

In Figure 7 and Figure 8, we provide a Python implementation for n-gram layers that we use in our experiments.

```python
def ngram_head(x, hidden_state, shift_step=1, ngram=1):
    """
    Args:
        x: bsz * input_len
        hidden_state: bsz * input_len * d_model
        ngram: 1 means bigram, 2 means trigram
        shift_step: which token to attend to after the matching ngram
    Output:
        bsz * input_len * d_model
    """
    bsz, seq_len = x.shape

    # bsz * L * L, match unigram as the first step
    mask_0 = x[:, None, :] == x[:, :, None]
    causal_mask = torch.tril(torch.ones(seq_len, seq_len,
        dtype=torch.bool, device=x.device), diagonal=-1)
    mask_0 = torch.logical_and(mask_0, causal_mask)

    masks = [mask_0.long()]
    for _ in range(1, ngram):
        # mask_0[i, j] = True means token i-1 and token j-1 is matched
        mask_0 = F.pad(mask_0, (1, -1, 1, -1), "constant", False)
        masks.append(mask_0.long())
    ngram_mask = torch.stack(masks, dim=-1).sum(dim=-1) >= ngram
    if shift_step > 0:
        ngram_mask = F.pad(ngram_mask,
            (shift_step, -shift_step), "constant", False)
    ngram_mask = torch.logical_and(ngram_mask, causal_mask)

    # form a uniform distribution for matched tokens
    ngram_mask_norm = ngram_mask / ngram_mask.sum(dim=2, keepdim=True)
    ngram_mask_norm = torch.nan_to_num(ngram_mask_norm, 0)
    ngram_mask_norm = ngram_mask_norm.to(hidden_state.dtype)
    output = torch.einsum("bmn,bnz->bmz", ngram_mask_norm, hidden_state)
    return output

class Ngram(nn.Module):
    def __init__(self, d_model, ngram=1):
        super().__init__()
        self.d_model = d_model
        self.ngram = ngram
        self.t0 = nn.Linear(self.d_model, self.d_model)
        self.t1 = nn.Linear(self.d_model, self.d_model)

    def forward(self, x, input_ids):
        bsz, seq_len, _ = x.shape
        h0 = ngram_head(input_ids, x, ngram=self.ngram)
        h1 = x
        y = self.t0(h0) + self.t1(h1)
        return y
```

Figure 7. Python implementation of n-gram layers.

```python
class NgramBlock(nn.Module):
    def __init__(self, config, ngram):
        """
        Args:
            ngram: 1, 2, or 3

        Note: parameter size 4d^2
        """
        super().__init__()
        self.ln_1 = RMSNorm(config.d_model, eps=1e-5)

        self.attn = Ngram(config, ngram)
        self.ln_2 = RMSNorm(config.d_model, eps=1e-5)

        mlp_hidden = config.d_model
        self.mlp = nn.Sequential(
            nn.Linear(config.d_model, mlp_hidden),
            nn.SiLU(),
            nn.Linear(mlp_hidden, config.d_model),
        )

    def forward(self, x, input_ids):
        x_att = self.attn(self.ln_1(x), input_ids)
        x = x + x_att
        x_mlp = self.mlp(self.ln_2(x))
        x = x + x_mlp
        return x
```

Figure 8. Python implementation of n-gram blocks with SwiGLU-MLP (Shazeer, 2020) and RMSNorm (Zhang & Sennrich, 2019).

G. Language Model Experiments

In the language model experiments, all models share the same following training hyperparameters. We plan to extend the experiment setting to larger models trained with more tokens in the future.

hyper parameter	**value**
hidden size (d)	1024
number of training tokens	7e9
number of warm-up tokens	5e8
batch size (number of tokens)	5e5
optimizer	AdamW
weight decay	0.01
└ learning rate	3e-4
└ βs	(0.9, 0.95)
scheduler	Cosine Annealing
└ minimum learning rate	3e-5