

Zoology: Measuring and Improving Recall in Efficient Language Models

Simran Arora^{*†}, Sabri Eyuboglu^{*†}, Aman Timalsina[△], Isys Johnson[‡], Michael Poli[†], James Zou[†], Atri Rudra[‡], and Christopher Ré[†]

[†]Stanford University

[‡]University at Buffalo

[△]Purdue University

[‡]{simran, eyuboglu, poli, jamesz, chrismre}@cs.stanford.edu

[‡]{isysjohn,atri}@buffalo.edu

[△]{atimalsi}@purdue.edu

March 5, 2024

Abstract

Attention-free language models that combine *gating* and *convolutions* are growing in popularity due to their efficiency and increasingly competitive performance. To better understand these architectures, we pretrain a suite of 17 attention and *gated-convolution* language models, finding that SoTA gated-convolution architectures still underperform attention by up to 2.1 perplexity points on the Pile. In fine-grained analysis, we find 82% of the gap is explained by each model’s ability to recall information that is previously mentioned in-context, *e.g.* *Hakuna Matata means no worries* *Hakuna Matata it means no* → *??*. On this task, termed *associative recall*, we find that attention outperforms gated-convolutions by a large margin: a 70M parameter attention model outperforms a 1.4 billion parameter gated-convolution model on associative recall. This is surprising because prior work shows gated convolutions can perfectly solve synthetic tests for AR capability. To close the gap between synthetics and real language, we develop a new formalization of the task called multi-query associative recall (MQAR) that better reflects actual language. We perform an empirical and theoretical study of MQAR that elucidates differences in the parameter-efficiency of attention and gated-convolution recall. Informed by our analysis, we evaluate simple convolution-attention hybrids and show that hybrids with input-dependent sparse attention patterns can close 97.4% of the gap to attention, while maintaining sub-quadratic scaling. Our code is accessible at: <https://github.com/HazyResearch/zooology>.

1 Introduction

Two advances – gating and long convolutions – have catalyzed a wave of excitement around *gated-convolution* language models [Fu et al., 2023a, Ma et al., 2022, Wang et al., 2022, Poli et al., 2023a, *inter alia*.]. These architectures combine gating (*i.e.* element-wise multiplication) with *long* convolutional filters (*i.e.* the length of the sequence) to enable interactions between distant tokens [Dauphin et al., 2017, Gu et al., 2021]. Recent work suggests that these models, which exhibit better asymptotic scaling in input sequence length than attention, can match attention in language modeling quality [Poli et al., 2023a, Peng et al., 2023, Fu et al., 2023b].

We pretrain and evaluate 17 language models across 4 scales (70M - 1.4Bn) and 5 architectures on the same data and infrastructure setup. Surprisingly, we find that there is still a perplexity gap of up to 2.1 points between state-of-the-art convolution-based architectures and strong Transformer baselines in language modeling on the Pile (Table 1). Through fine-grained analysis, we find a single, simple capability is responsible

^{*}Equal contribution, Random ordering by coin toss.

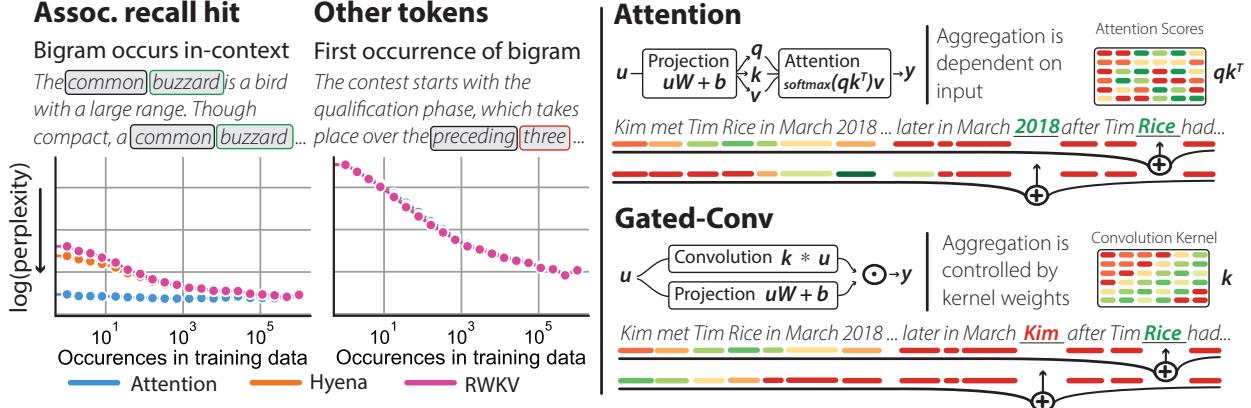


Figure 1: **The associative recall gap.** We stratify Pile validation data for models from each architecture class by whether or not the predicted token is a previously seen bigram in the example context. We plot validation perplexity versus the bigram’s frequency in the training data. We can clearly see that the gap is localized to examples where bigrams occur, and are seen rarely during in training.

for much of the gap: recalling information seen in-context. Consider the example below, where some tokens can be predicted by recalling an earlier association:

$\underbrace{\text{Hakuna Matata!}}_{\text{Key-Value}}$ It means no worries for the rest of your days! $\underbrace{\text{Hakuna Matata}}_{\text{Query}}$ means no \rightarrow worries $\underbrace{\text{AR Hit}}_{\text{Query}}$ $\underbrace{\text{AR Hit}}_{\text{AR Hit}}$

We find that errors on “AR Hits” (e.g. *worries* above) account for 82% of the perplexity gap to attention on average, despite only representing 6.4% of all tokens in the Pile dataset.¹ A 70M parameter Transformer can predict AR Hits better than a 1.4Bn parameter Hyena gated convolution model (20× larger) (Table 1, Table 5). The AR gap persists at the 7Bn parameter scale when comparing RWKV and Llama-2 (Appendix G).

This task, *associative recall* (AR), has a long history in machine learning [Graves et al., 2014, Ba et al., 2016, *inter alia.*] (Appendix A.1). Prior work argues that a model’s ability to perform AR is predictive of in-context learning quality [Elhage et al., 2021, Olsson et al., 2022]. As a result, AR has been adopted as a tool in designing new architectures and prior work has shown that gated convolution architectures match attention on synthetic AR tasks used as proxies for real language modeling [Fu et al., 2023a, Poli et al., 2023a, Lutati et al., 2023]. For this reason, the downstream AR perplexity gaps are surprising.

Through measuring recall on real data, we learn the key disparity is that prior synthetic formulations assume there is *one query* per input, at a *fixed position* in the sequence, where tokens come from a small vocabulary size (e.g. $|V| < 50$, less than model dimension). Yet, language modeling often requires performing *multiple recalls* (e.g. for both “*Hakuna Matata*” and “*no worries*” above, in a single forward pass), at *varying positions*, with tokens from a large vocabulary (larger than model dimension). We thus propose the study of *multi-query* AR (MQAR). Compared to the prior AR formulations, MQAR better captures the persisting quality gaps on synthetic and real world data (Section 4). However, it is not clear why MQAR elucidates the gap.

We formalize the the Mqar gap across the architecture classes. Gated-convolutions process variable sequences using *fixed filters* defined by the model weights rather than as functions of the input data (See Figure 1). We find it is inefficient for gated-convolutions to perform the variable distance token-to-token interactions (e.g. at a distance of 10 tokens for *Hakuna Matata* and 9 for *no worries*) in a parameter and FLOPs efficient way compared to attention. Attention achieves input-dependence since it computes all token-to-token interactions when determining how to mix information in the sequence. We formally describe the limitation of gated convolutions via theory and experiments in the rest of the paper.

We first introduce a simple operator, BASECONV, that can provably simulate the class of architectures built from gating and convolution primitives. This includes architectures such as H3, Hyena, RWKV, and RetNet.

¹We measure AR on real data using a simple heuristic: n -gram tokens that are repeated in context.

Model	Param (M)	TFLOPs	Overall		Slices		% of gap due to AR Hits
			AR Hits	Other Tokens	AR Hits	AR Hits	
Attention	125	2.46	11.01 (2.40)	2.16 (0.77)	12.45 (2.52)	—	—
Long Conv	128	1.74	16.98 (2.83)	25.62 (3.24)	16.46 (2.80)	40.1%	
H3	168	2.55	12.06 (2.49)	6.75 (1.91)	12.60 (2.53)	88.4%	
Hyena	158	2.41	11.60 (2.45)	5.00 (1.61)	12.28 (2.51)	100.0%	
RWKV	169	2.08	11.64 (2.45)	5.70 (1.74)	12.29 (2.51)	100.0%	
Attention	360	6.23	9.44 (2.25)	1.98 (0.69)	10.62 (2.36)	—	—
Long Conv	360	4.08	13.13 (2.57)	13.27 (2.59)	13.12 (2.57)	40.5%	
H3	357	4.85	10.38 (2.34)	4.81 (1.57)	11.00 (2.40)	65.8%	
Hyena	358	5.03	10.07 (2.31)	3.83 (1.34)	10.75 (2.38)	98.2%	
RWKV	351	4.31	9.79 (2.28)	3.82 (1.34)	10.51 (2.35)	100.0%	

Table 1: **Language modeling validation perplexity on the Pile.** After pretraining on 10B tokens of Pile data, we report log perplexity with negative log-likelihood in parentheses. We report overall scores, and for the AR vs. non-AR token slices defined in Section 3. FLOPs are computed for inputs of 2048 tokens based on the equations in Appendix C. A table with additional results is in Table 4.

We show with theory and experiments that the model dimension for BASECONV (and thus the aforementioned architectures) to solve MQAR grows with the input sequence length (Theorem 4.4) while attention can solve MQAR with model dimension independent of sequence length (Proposition 4.3, Section 4.3).² In practice, gated-convolutions appear to encode approximate solutions to AR that support only a subset of token-interaction distances (which affects the ability to identify matching keys given MQAR queries). While theoretically analyzing deep learning architectures is challenging [Hahn, 2020, Merrill et al., 2022, Keles et al., 2023], the fact that gating and convolutions are polynomial operations facilitates our precise analysis.

We show that input-dependent sequence mixing is important to solve Mqar efficiently. The scaling for gated convolutions with input-independent convolutions is undesirable so we next ask which architectural choices close the gap. We show that data-dependent sequence mixing helps an architecture solve MQAR efficiently (Theorem 4.5). The model needs to adapt the sequence mixing weights based on the token-interaction distances required for each new example.

Several architectural modifications could satisfy the input-dependence property. Based on our analysis, we evaluate *minimal modifications* that only add input-dependent operations on exact-match repeated bigram tokens (our heuristic for measuring tokens that require recall). Note that language models often need to perform recall between fuzzier substrings (e.g. synonymous bigrams) or higher-dimensional concepts. However, we show that simply inserting input-dependent operator — e.g., a convolution filter that shifts the sequence based on the bigram positions or sparse attention placed only on repeated bigram positions — to the BASECONV architecture at < 10% of layers suffices to outperform the Transformer baseline on Pile language modeling (Section 5) and succeed on MQAR synthetic tasks (Section 4.3). Moreover, this closes > 80% of the MQAR perplexity gap on the Pile validation data. Finally, we prototype solutions that *learn* the positions at which to use input-dependent operations, validating that they also close the gap to attention on the Pile.

In this work, we analyze the increasingly-popular convolution-based architectures and identify fundamental limitations. We hope the MQAR framework and our analysis of the role of input-dependence for MQAR inform the design of future architectures. We release our code for reproducibility: <https://github.com/HazyResearch/zoology>.

2 Background and Preliminaries

In this section, we describe the setting, introduce notation, and discuss important related work. See Appendix A for a broader discussion of related work.

Language modeling. We study auto-regressive language models trained on the task of next token

²Note that the *runtime* of Attention is quadratic, but gated convolutions is near linear.

prediction [Gao et al., 2020]. Given a sequence of N tokens $\mathbf{x} = \{x_0, \dots, x_{N-1}\}$ drawn from a vocabulary C , the model outputs a probability distribution over C for each of x_i given the preceding tokens $\text{P}(x_i|x_0, \dots, x_{i-1})$. The language models in this work share the same high-level architecture. First, each token x_i in the input is embedded in d -dimensional space yielding a matrix $\mathbf{u} \in \mathbb{R}^{N \times d}$. Next, \mathbf{u} is passed through a stack of L layers, with layer ℓ outputting $\mathbf{u}^\ell \in \mathbb{R}^{N \times d}$. Finally, the embeddings \mathbf{u}^L output by the last layer are mapped back to logits over C with a linear projection. Each layer transforms \mathbf{u} with a *sequence mixer* (e.g. attention) followed by a *state mixer* (e.g. MLP). Unless specified, our models adhere to the implementation details of the LLaMA architecture (except the sequence mixer, which we vary throughout) [Touvron et al., 2023].

Sequence mixers. Our work evaluates how the choice of sequence mixer affects the quality and behavior of language models. Most sequence mixers aggregate the token embeddings in a sequence via a weighted sum. For example, $\mathbf{y}[i, :] = \sum_{j=0}^{N-1} \omega(i, j) \mathbf{u}[j, :]$, where ω is a function outputting scalar weights. We study the differences between two classes of sequence mixers, discussed next.

Attention. [Vaswani et al., 2017] We review attention, the *de facto* language model sequence mixer. An attention layer is parameterized by three learnable projection matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$. To compute the output \mathbf{y} given inputs \mathbf{u} , attention applies the projections to the input: $\mathbf{q} = \mathbf{Q}\mathbf{u}$, $\mathbf{k} = \mathbf{K}\mathbf{u}$, $\mathbf{v} = \mathbf{V}\mathbf{u}$. The projected embeddings are aggregated according to: $\mathbf{y} = \text{softmax}(\frac{1}{\sqrt{d}} \mathbf{q} \mathbf{k}^\top) \mathbf{v}$ (shown in Fig. 1) in $\mathcal{O}(N^2 d)$ time, which is expensive for long sequences (large N).

Gated-convolutions. A more efficient alternative to attention is the convolution, which is defined as $\mathbf{y}[i, :] = \sum_{j=0}^{N-1} \mathbf{k}[j, :] \odot \mathbf{u}[i - j, :]$ where the kernel $\mathbf{k} \in \mathbb{R}^{N \times d}$ is a learnable weight matrix. Convolutions can be computed in time $\mathcal{O}(Nd \log N)$ using the Fast Fourier Transform (FFT) and the convolution theorem: $\mathbf{y} = \mathbf{u} * \mathbf{k} = \text{FFT}^{-1}(\text{FFT}(\mathbf{u}) \odot \text{FFT}(\mathbf{k}))$ [Cooley and Tukey, 1965]. While purely convolutional architectures match or outperform attention in certain domains (e.g. vision [Tay et al., 2022, Gu et al., 2021], audio [Goel et al., 2022], and time-series [Zhang et al., 2023]), they trail by a large margin on language [Fu et al., 2023a].

Recent work has closed much of this gap by combining convolutions with *gating* (i.e. elementwise multiplication of the input with a transformed version of itself). *Gating* was first proposed in the context of convolutions by Dauphin et al. [2017], but more recently it has played a central role in state-of-the-art sub-quadratic language models, some of which claim attention-level quality [Fu et al., 2023a, Wang et al., 2022, Poli et al., 2023a, Peng et al., 2023, Zhai et al., 2021, Fu et al., 2023b, *inter alia*]. Though they may appear different on the surface, these sub-quadratic architectures can all be expressed in terms of convolutions and gating. See Appendix H.2 for detailed descriptions of these architectures and their similarities. This work analyzes the differences between attention and the broad class of *gated convolution* mixers.

3 Identifying the associative recall problem

In this section, we measure the perplexity gap between gated convolutions and attention and show that single skill termed *associative recall* accounts for 82% of the gap on average. This is surprising because prior work shows gated convolutions solve a synthetic version of associative recall perfectly. Informed by our analysis, we define a new synthetic formulation that better reflects real data. This task facilitates our analysis of why the gap occurs (Section 4) and how to fix it (Section 5).

3.1 Fine-grained analysis of downstream quality

Perplexity Gap We pretrain a suite of large language models with different sequence mixers across 3 scales (70M-360M) for 10B tokens on the standard Pile language modeling setting using the EleutherAI GPT-NeoX training infrastructure [Gao et al., 2020, Andonian et al., 2023]. In the main paper, we compare attention to three state-of-the-art gated convolution sequence mixers: H3, Hyena, and RWKV [Fu et al., 2023a, Poli et al., 2023a, Peng et al., 2023].³ We further include a pure long-convolution model to underscore the importance of gating. In Appendix F, we include results on additional sequence mixers [Hasani et al., 2022, Sun et al., 2023]. We use a strong Transformer attention baseline with rotary embeddings and SwiGLU MLPs following the Llama architecture [Touvron et al., 2023]. We also use this strong training recipe when training

³RWKV is commonly referred to as an RNN. We show it can be viewed as a convolution (Appendix H.2.2).

the attention-free sequence mixers. For experimental details and hyperparameters for all architectures see Appendix C.

Across scales, we find that attention outperforms the gated convolutions by at least a third of a perplexity point on average: the minimum gaps are +2.14, +0.59, +0.35 PPL at 70M, 160M, and 360M parameter scales, respectively. We report overall test perplexity in Table 1. Though these gaps are relatively small on average, models may still perform very differently on different subsets of data [Eyuboglu et al., 2022].

Associative Recall Perplexity To better understand the differences between attention and gated convolutions, we perform a fine-grained analysis of next token predictions and observe that convolution-based models struggle to recall associations previously seen in context. For example, in Fig. 1, the model must recall the association between “*Tim*” and the last name “*Rice*”. Following a long line of prior work, we call this skill *associative recall* (AR) [Willshaw et al., 1969, Hopfield, 1982] (see Appendix A for an extended discussion of AR’s history in machine learning). In Appendix D.1.1, we provide annotated Pile examples to demonstrate the phenomenon qualitatively.

Quantifying AR performance. It is challenging to derive a quantitative measure of associative recall performance on the Pile because we don’t know which next token predictions in raw text require associative recall. We use a simple heuristic to identify these tokens, which we refer to as *AR Hits*. An AR Hit is the last token of an n -gram repeated in context (e.g. the second occurrence of “*Rice*” in Fig. 1). However, some common n -grams (e.g. “*of the*”) could have been memorized during training, so we factor in the frequency with which an n -grams appeared in the training data. This heuristic enables us to scale our analysis to over 10 million tokens of Pile validation data.

We stratify Pile tokens into two slices based on this heuristic and report perplexity on each in Table 1:

1. **AR Hits:** (6.4% of tokens) Tokens in the final position of a bigram (a pair of consecutive tokens) which previously appeared in context, but $\leq 1250 \times$ during training.
2. **Other tokens:** (93.6% of tokens) Tokens in the final position of a bigram which did not previously appear in context or it appeared $> 1,250$ times during training.

In Figure 1, we visualize these slices by plotting log-perplexity against the frequency with which bigrams appear during training. Strikingly, the gap between attention and gated convolutions is the largest on AR hits with the fewest occurrences. On the other tokens, there is no gap.

In Table 1, we also compute the percentage of the difference in perplexity between attention and each model that is due to AR tokens: $\frac{\Delta \log(\phi_{\text{AR}}) \cdot |T_{\text{AR}}|}{\Delta \log(\phi) \cdot |T|}$, where ϕ is the perplexity and T is the set of tokens in the test set. This quantity can also be interpreted as the fraction of the overall gap that would close if a model matched attention on the AR slice. We find that the AR slice accounts for 82% of the average quality gap between the gated convolutions and attention.

To evaluate the AR capacity of larger models, we train two 1.4 billion parameter attention and Hyena models for 50 billion tokens on the Pile and repeat this analysis (see Table 5). Strikingly, a 70 million parameter attention model is a full perplexity point better in the AR slice than this 1.4B Hyena model that is $20 \times$ its size (2.41 vs. 3.43 ppl.). In Appendix G, we also evaluate open-source RWKV and attention models trained up to 7 billion parameters. We use a controlled semi-synthetic dataset to measure AR capacity and show that RWKV’s performance degrades sharply as we increase the number of queries in an example while attention performs consistently well Appendix G.

3.2 Formalizing the problem: Multi-Query Associative Recall

This gap in associative recall perplexity is very surprising because prior work shows gated-convolutions can perfectly solve a formalized version of the task [Fu et al., 2023a, Poli et al., 2023a, Olsson et al., 2022]. In this synthetic task, the input \mathbf{x} contains a sequence of bigrams representing *key-value* pairs from a random dictionary followed by a *single* query token. For example, the correct output for the input below would be 3:

$$\text{A 4 B } \underbrace{\text{3 C 6 E 2 F 1 C 6 G 8}}_{\text{Key-Value}} \rightarrow \underbrace{\text{B ?}}_{\text{Query}}$$

Gated convolutions (*e.g.* H3, Hyena, RWKV) can solve this task perfectly for most sequence lengths.

These conclusions are inconsistent with our findings on the Pile, as described above, so we ask how this formulation of AR differs from the way AR manifests in real language. We identify a major difference. In real world inputs, the language model often needs to perform multiple associative recalls in a single forward pass, at varying positions in the sequence (*e.g.* “Tim Rice” and “March 2018” in Fig. 1). We refer to this as Multi-Query AR (MQAR). We formally define the MQAR problem as follows:⁴

Definition 3.1 (Multi-Query-AR (MQAR)). We are given an input sequence $\mathbf{x} = \{x_0, \dots, x_{N-1}\}$ where each $x_i \in C$ is a token drawn from a vocabulary of size $c = |C|$. The task is to check, for every query $1 \leq i < N$, whether there exists a $0 \leq j < i$ such that $\mathbf{u}_i \equiv \mathbf{u}_j$. If so, output \mathbf{u}_{j+1} .

For example, the correct output for input below would be 4, 6, 1, 2, 3:

$$\begin{array}{ccccccccc} \text{A} & 4 & \text{B} & 3 & \text{C} & 6 & \underbrace{\text{F} & 1}_{\text{Key-Value}} & \text{E} & 2 \rightarrow \text{A} & ? & \text{C} & ? & \underbrace{\text{F} & ?}_{\text{Query}} & \text{E} & ? & \text{B} & ? \end{array}$$

In Section 4, we use MQAR to explain the quality gap between gated convolutions and attention.

4 Explaining the associative recall problem

In this section, we provide an explanation for the gap in associative recall performance by analyzing the formal MQAR task theoretically and empirically. In Section 4.1, we define a simple gated-convolution architecture, called BASECONV, which we show can simulate a broad class of architectures built from gating and convolutions. This allows us to make general statements that apply to popular gated-convolution architectures like Hyena, RWKV, or H3. In Section 4.2, we show that there exist theoretical solutions to MQAR that could in principle be learned by BASECONV, and we analyze their complexity in terms of model width and depth. In Section 4.3, we use experiments on synthetic data to show that solving MQAR with BASECONV (and other gated-convolution architectures) requires model dimension to scale linearly with the sequence length. In contrast, attention solves MQAR consistently in our experiments with model dimension scaling independently of sequence length. These empirical scaling laws provide a potential explanation for the AR gap and, alongside our theoretical analysis, point to the potential solutions discussed in Section 5.

4.1 BaseConv: a minimal gated convolution operator

In this section, we define our minimal gated-convolution architecture, called BASECONV. Given a function, we would like to know the most efficient model (*e.g.* parameters, FLOPs) that can represent the solution. In this work, we show we can precisely reason about this question for representing *polynomial functions* with gated convolutions as gating and convolutions are both polynomial operations. The standard model defining computational complexity for polynomials is by the size of the smallest arithmetic circuit that can compute the polynomial. We define the BASECONV gated convolution operator which is exciting because (1) it is universal in that it can simulate any arithmetic circuit \mathcal{C} (with only a poly-log blowup in the corresponding parameters) and (2) it is simple to implement efficiently (19 lines of pure PyTorch including imports, see Appendix B).

Definition 4.1 (BASECONV Operator). Given an input $\mathbf{u} \in \mathbb{R}^{N \times d}$, the BASECONV operator for layer ℓ is defined as:

$$\mathbf{y} := \underbrace{(\mathbf{u} \cdot \mathbf{W}^\ell + \mathbf{b}_1^\ell)}_{\text{Linear Projection}} \odot \underbrace{(\mathbf{h}^\ell * \mathbf{u} + \mathbf{b}_2^\ell)}_{\text{Convolution}} \quad (1)$$

where the layer is parameterized by learnable filters $\mathbf{h} \in \mathbb{R}^{N \times d}$, a linear projection $\mathbf{W}^\ell \in \mathbb{R}^{d \times d}$, and ‘bias’ matrices $\mathbf{b}_1, \mathbf{b}_2 \in \mathbb{R}^{N \times d}$. The \odot is component-wise product and convolution of two matrices is computed as convolution of the corresponding columns.

⁴In Appendix H.7.1 we define a formal, general form of this definition, used in the theoretical analysis.

In our experiments, each BASECONV layer uses $\tilde{\mathcal{O}}(Nd + d^2)$ parameters⁵ and can be computed in $\tilde{\mathcal{O}}(Nd^2)$ operations. For our theoretical results, we can assume the weight matrix \mathbf{W}^ℓ is restricted to a class of matrices that support near-linear time matrix multiplication (e.g. Kaleidoscope matrices, see Definition H.3). Under this assumption, BASECONV uses $\tilde{\mathcal{O}}(Nd)$ parameters and $\tilde{\mathcal{O}}(Nd)$ FLOPs (Proposition H.6). We now state the equivalency result between arithmetic circuits and BASECONV a “canonical” representation of arithmetic circuits (Theorem H.21 in Appendix H.5):

Theorem 4.2 (Equivalency to Arithmetic Circuits). *For an arithmetic circuit \mathcal{C} of size s and depth Δ that takes $\mathbf{u} \in \mathbb{R}^{N \times d}$ as input, there exists an equivalent BASECONV operator that uses $\tilde{\mathcal{O}}(s\Delta)$ parameters and $\tilde{\mathcal{O}}(\Delta)$ layers.⁶*

In other words, any gated convolution model with small number of layers can be simulated by BASECONV with only a (poly)logarithmic blowup in parameters and layers. We note that arithmetic circuits are a very well studied computation model in computational complexity Bürgisser et al. [1996]. Many well-known efficient algorithms on matrices (e.g. the FFT or the current best known matrix-matrix multiplication algorithm) in fact give small arithmetic circuits. However, arithmetic circuits are inherently discrete objects – we cannot learn them via gradient descent. Theorem 4.2 shows that (up to poly-log loss in parameters), we can *instead* learn over BASECONV models. This result generalizes a similar result from Dao et al. [2020] for the special class of linear functions: we generalize the earlier result to the class of *all* polynomials.

For specific gated convolution layers, we can get rid of the poly-logarithmic factor blowup—we observe in the appendix that BASECONV and Hyena models can simulate each other with only a small constant blowup in parameters (Proposition H.12 in Appendix H.5).

4.2 Theoretical analysis of gated convolution capacity and associative recall

In this section, we provide theoretical MQAR solutions that could in principle be learned by each architecture and analyze their complexity in terms of model width and depth. First, we note that attention solves MQAR with parameters independent of sequence length (Proposition H.27).

Proposition 4.3 (Attention). *Given an input $\mathbf{u} \in \{0, 1\}^{N \times 3c}$, Attention (even without using soft-max) solves MQAR for \mathbf{u} using $\mathcal{O}(c^2)$ parameters, $\mathcal{O}(Nc^2 + N^2c)$ time complexity and $\mathcal{O}(1)$ layers.*

It is natural to wonder then if *all* pairwise comparisons among tokens are necessary to solve MQAR. Indeed, in the RAM setting, a sequential algorithm can simply utilize N logarithmic insertion and membership queries to solve MQAR in subquadratic time. Unfortunately, any model attempting to emulate this would require $\Omega(N)$ layers. Instead, we observe that we can parallelize this algorithm using dyadic intervals and achieve a depth of $\tilde{\mathcal{O}}(1)$ (Proposition H.30). We then convert this algorithm into an arithmetic circuit and apply Theorem 4.2 to derive an equivalent BASECONV model. This allows us to prove new upper bounds for BASECONV models applied to MQAR, which improves upon the quadratic time complexity of attention to near-linear runtime at the cost of using poly-log layers (Theorem H.37 in Appendix H.7).

Theorem 4.4 (Data-Independent Filters⁷). *Given an input $\mathbf{u} \in \{0, 1\}^{N \times \mathcal{O}(\log c)}$ to MQAR (where we assume that distinct tokens are embedded into distinct vectors in $\{0, 1\}^{\mathcal{O}(\log c)}$), there exists a BASECONV operator that solves MQAR for \mathbf{u} using $\tilde{\mathcal{O}}(N \log c)$ parameters as well as time complexity and $\tilde{\mathcal{O}}(1)$ layers.*

Nevertheless, the poly-logarithmic number of layers in the above result is undesirable in practice. But, we show that using *input-dependent* convolution filters, one can get constant many layers (for a sub-class of inputs). Towards that end, we define the interaction distance between a query \mathbf{q}_i and the matching key \mathbf{k}_j as $i - j$. This then allows us to present the corresponding upper bound for data-dependent mixing (Theorem H.38 in Appendix H.8).

Theorem 4.5 (Input-Dependent Filters). *Given an input $\mathbf{u} \in \{0, 1\}^{N \times c}$ to MQAR (where we assume that the*

⁵We use $\tilde{\mathcal{O}}(\cdot)$ to hide poly-log factors.

⁶The formal statement in the Appendix has a sharper version of this result in terms of the circuit ‘width’.

⁷We note here that existing architectures also use data-independent convolution filters, meaning the filter is defined as a function of the model parameters, independent of the input.

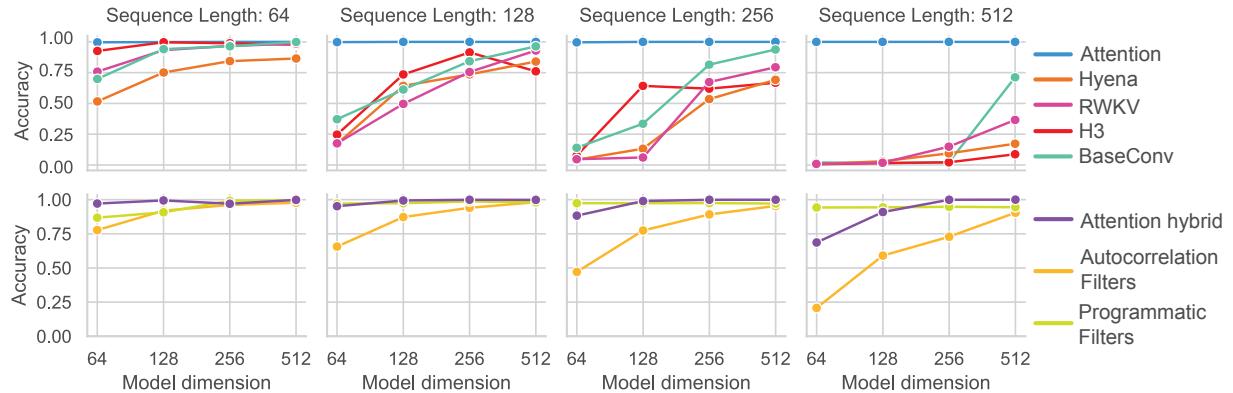


Figure 2: The x-axis is the model dimension and the y-axis is accuracy on MQAR. Increasing the sequence length correlates with increased task difficulty. Both rows use the same experimental setup, but evaluate different sequence mixers. **(Top, Claim 1) Gated convolutions and attention.** We observe the gated convolutions require larger dimensionality than attention to solve the task. **(Bottom, Claim 2) Input-dependent filters.** Models with input-independent aggregation achieve improved scaling over the gated-convolutions with input-independent aggregation.

tokens are embedded as one-hot encoding in $\{0, 1\}^c$ and there exists at most t distinct interaction distances),⁸ there exists a BASECONV operator that uses input-dependent kernels to solve the above case of MQAR using $\mathcal{O}(t \cdot Nc)$ parameters and $\mathcal{O}(1)$ layers.

4.3 Empirical analysis of gated convolution capacity and associative recall

In this section, we measure empirically how model dimension must scale in order for different sequence mixers to solve MQAR.

Setup We train and evaluate models on a synthetic MQAR with vocabulary size 8,192, varying model dimension and sequence length from 64 to 512. Appendix E provides further details on the formulation and construction of this synthetic task. Following Olsson et al. [2022], we train two layer models with a Transformer backbone that interleaves sequence mixing and state mixing (MLPs). For each architecture, we sweep four learning rates from $\log(-4)$ to $\log(-2)$ for each architecture, and report maximum test accuracy.

Our results, which are summarized in Figure 2, support two main claims:

Claim 1 (Gated-convolutions and attention). *Gated-convolution models with two layers require model dimension to scale at least linearly in sequence length in order to solve associative recall, while attention models can solve it with near-constant dimensionality.* We compare attention and BASECONV as well as three popular instantiations of gated-convolution architectures: RWKV, H3, and Hyena [Peng et al., 2023, Fu et al., 2023c, Poli et al., 2023a]. In the top row of Fig. 2, attention solves MQAR perfectly at all sequence lengths using a constant model dimension of 64. In contrast, MQAR does not achieve accuracy > 0.9 unless $d \geq N$.

Claim 2 (Input-dependent filters). *Using input-dependent filters in gated-convolution models can close some of the gap to attention.* In Theorem 4.5, we show that BASECONV with input-dependent filters could solve MQAR with improved scaling. In this solution, we construct a filter that spikes at position j if matching keys are separated by j tokens. We evaluate two approaches for constructing this filter: (1) programmatically (*i.e.* hard-coded comparisons between token ids) or (2) with autocorrelation, which could learn to perform fuzzy-matches (see Appendix H.8). In the bottom row of Fig. 2, we see that BASECONV with programmatic input-dependent filters achieves near-constant scaling in model dimension and that BASECONV with autocorrelation input-dependent filters achieves improved scaling over BASECONV with input-independent filters.

⁸Note that the interaction distances can be arbitrary: there is just a bounded number of distinct distances.

These input-dependent filters cannot easily be made to satisfy causality and using an $O(N \log N)$ filter per gap could be expensive if each gap applies only to a small number of bigrams. A simpler and perhaps more efficient way to solve the problem would be to introduce a small amount of attention to an otherwise BASECONV model [Fu et al., 2023a]. As the first natural baseline, we evaluate an attention hybrid on synthetic MQAR and show that it achieves improved scaling in Figure 2. Next, we put these empirical and theoretical insights into practice on Pile language modeling.

5 Closing the Associative Recall Gap

In this section, we evaluate hybrid BASECONV-Attention models that leverage different sparsity patterns. We show that hybrids with input-dependent sparsity patterns can close most of the gap to attention, while maintaining sub-quadratic scaling. We also show that BASECONV hybrids can outperform attention-only models by up to a full perplexity point, all while being dramatically simpler to implement and analyze than prior hybrids [Fu et al., 2023a] (see implementation in Appendix B). We describe the architectures in Section 5 and results on Pile language modeling in Section 5.

Sparse BaseConv-Attention Hybrids We evaluate hybrids composed primarily of BASECONV layers and three attention layers (6.3% of layers at 354M parameters and 10% at 168M parameters). We augment the attention layers with operators that *selectively* apply attention to some tokens based on a selection function $f : \mathbb{R}^{N \times d} \rightarrow \{0, 1\}^N$. They take as input $\mathbf{u} \in \mathbb{R}^{N \times d}$ and output $\mathbf{y} \in \mathbb{R}^{N \times d}$:

$$\mathbf{y}[i, :] = \text{softmax}\left(\frac{1}{\sqrt{d}} \mathbf{q}[i, :] \mathbf{k}^\top\right) \mathbf{v} \cdot f(\mathbf{u})[i] \quad (2)$$

where $\mathbf{q}, \mathbf{k}, \mathbf{v}$ are query, key, and value projections, as in attention. We evaluate four choices for f :

(1) *Full attention*. First, we evaluate the performance of full attention hybrids. These hybrids correspond to fixing $f(\mathbf{u})[i] = 1$ for all i in Eq. (2). Prior work has shown full attention hybrids to be effective, but they do not explain why supplementing gated convolutions with attention is necessary [Fu et al., 2023a]. Based on our findings in Section 3, we hypothesize that sparse attention applied only to associative recall hits may suffice.

(2) *Random selection*. As a control, we evaluate sparse attention randomly applied to tokens. This corresponds to a stochastic selection function where $f(\mathbf{u})[i]$ is drawn from a Bernoulli. Many prior works have employed sparse attention patterns like this one, which are independent of the input [Zaheer et al., 2020, Child et al., 2019a, Beltagy et al., 2020, *inter alia*].

(3) *Programmatic selection*. Next, to evaluate our hypothesis that attention is needed for associative recall, we prototype a programmatic selection function that selects only those tokens that might be associative recall hits. Specifically, $f(\mathbf{x}[i, :])$ is 1 if the token x_i previously occurred in the sequence. In practice, we compare raw token ids, not token embeddings.

$$f(\mathbf{x})[i] = \begin{cases} 1 & \text{if there exists } j < i \text{ such that } x_i = x_j \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

(4) *Learned selection*. Finally, we prototype a learned selection function $f(\mathbf{u})[i] = \sigma(\mathbf{u}[i, :] \cdot \mathbf{W})$ parameterized as a simple linear layer with sigmoid activation. We fix a hyperparameter k and select the top- k tokens with the highest score in each batch. This allows us to compute attention in $\mathcal{O}(ndk)$ time. During training, we add small amount of Gaussian noise to the top- k calculation to encourage exploration and use an auxiliary loss to encourage sparse selection: $\ell_f(\mathbf{u}) = \frac{1}{N} \max(0, \sum_{i=1}^N f(\mathbf{u})[i] - k)$. This approach is most similar to the recently proposed SeqBoat architecture [Ren et al., 2023]. We discuss the differences in Appendix A.

Downstream Evaluations We evaluate the prototypes on Pile language modeling. We take BASECONV architectures at the 150M and 360M parameter scales and add input-dependent selection to three layers (details in Appendix C.2). Results are in Table 2. We validate that the prototypes close the overall and AR quality gaps to attention when added to the BASECONV backbone.

Model	Param (M)	TFLOPs	Overall		Slices	
			AR Hits	Other Tokens	AR Hits	Other Tokens
Attention	125	2.46	11.01 (2.40)	2.16 (0.77)	12.45 (2.52)	
BASECONV	168	2.46	12.90 (2.56)	8.68 (2.16)	13.29 (2.59)	
+ Random selection	162	2.44	12.13 (2.50)	5.33 (1.67)	12.83 (2.55)	
+ Programmatic selection	162	2.44	11.06 (2.40)	2.70 (0.99)	12.19 (2.50)	
+ Learned selection	166	2.44	11.06 (2.40)	3.20 (1.16)	12.14 (2.50)	
+ Full attention	166	2.58	9.57 (2.26)	2.01 (0.70)	10.76 (2.38)	
Attention	360	6.23	9.44 (2.25)	1.98 (0.69)	10.62 (2.36)	
BASECONV	354	4.81	11.01 (2.40)	5.98 (1.79)	11.52 (2.44)	
+ Random selection	365	5.06	12.94 (2.56)	6.17 (1.82)	13.62 (2.61)	
+ Programmatic selection	365	5.06	9.54 (2.26)	2.35 (0.86)	10.50 (2.35)	
+ Learned selection	351	5.06	9.59 (2.26)	2.61 (0.96)	10.58 (2.36)	
+ Full attention	351	5.10	8.59 (2.15)	1.95 (0.67)	8.91 (2.19)	

Table 2: **Language model perplexity on slices of the PILE.** We evaluate Hyena and BASECONV with Hybridization and Selective look-up at 160 and 355M parameters. We validate that the methods enable the gated convolutions to outperform attention.

At 360M parameters, BASECONV with just 3 attention layers can outperform the Transformer, while requiring fewer FLOPs. Our hybrid attention-BASECONV models outperform attention only models by 0.85 perplexity points while enabling an 18% reduction in total FLOPs vs attention. However, this uses full quadratic attention. We next show that sparse attention localized to potential AR tokens, is also sufficient to close the gap, validating our insights on the role of input-dependence for MQAR. At 360M parameters, programmatic selection closes 85% of the gap between pure BASECONV and attention on the AR slice, in contrast to the random selection control. Learned selection closes 72% a of the gap using just $k = 256$ (sub-quadratic) attention positions per example.

6 Discussion and Conclusion

We present an extensive analysis of gated convolution architectures in light of their recent popularity. We identify a persisting quality gap between efficient convolution and inefficient attention based architectures, largely due to a single failure mode associative recall. We design a new multi-query associative recall (MQAR) analysis tool, which correlates with downstream AR quality. We theoretically and empirically explain the gap is due to insufficient data-dependent mixing in gated convolutions and we show minimal architectures that close the gap on the Pile.

Our results in analyzing gated convolutions go beyond the conventional wisdom that attention is the “*right*” model. A significant amount of work focuses on improving the efficiency of attention [Dao et al., 2022, Dao, 2023, Katharopoulos et al., 2020a] and theoretically studying the exact power of attention [Hahn, 2020, Merrill et al., 2022, Keles et al., 2023]. Attention is often used as the goalpost for what is needed downstream. We hope our contributions highlight the value of MQAR, and more broadly tasks tied to real language modeling, as a proxy to study.

Acknowledgments

We thank Tri Dao, Daniel Fu, Neel Guha, Stefano Massaroli, Eric Nguyen, and Michael Zhang for helpful feedback and discussion during this work. We are grateful to Together Computer for making this work possible. We gratefully acknowledge the support of DARPA under Nos. FA86501827865 (SDH) and FA86501827882 (ASED); NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); ONR under No. N000141712266 (Unifying Weak Supervision); the Moore Foundation, NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba,

TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, the Okawa Foundation, American Family Insurance, Google Cloud, Microsoft Azure, Swiss Re, Brown Institute for Media Innovation, Department of Defense (DoD) through the National Defense Science and Engineering Graduate Fellowship (NDSEG) Program, Fannie and John Hertz Foundation, National Science Foundation Graduate Research Fellowship Program, Texas Instruments Stanford Graduate Fellowship in Science and Engineering, and members of the Stanford DAWN project: Teradata, Facebook, Google, Ant Financial, NEC, VMWare, and Infosys. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of DARPA, NIH, ONR, or the U.S. Government. AR's work is supported by NSF grant# CCF-2247014. IJ's work is supported by an NSF Graduate Fellowship.

References

Daniel Y. Fu, Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. Hungry Hungry Hippos: Towards language modeling with state space models. In *International Conference on Learning Representations*, 2023a. [1](#), [1](#), [2](#), [3.1](#), [3.2](#), [4.3](#), [5](#), [5](#), [A.3](#), [A.3](#), [C.2](#), [E.1](#)

Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Zettlemoyer Luke. Mega: Moving average equipped gated attention. *arXiv preprint arXiv:2209.10655*, 2022. [1](#), [A.3](#), [A.3](#)

Junxiong Wang, Jing Nathan Yan, Albert Gu, and Alexander M Rush. Pretraining without attention. *arXiv preprint arXiv:2212.10544*, 2022. [1](#), [2](#)

Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. *arXiv preprint arXiv:2302.10866*, 2023a. [1](#), [1](#), [2](#), [3.1](#), [3.2](#), [4.3](#), [A.2](#), [A.3](#), [B](#), [C.2](#), [E.1](#), [2](#), [H.2](#), [8](#)

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941. PMLR, 2017. [1](#), [2](#)

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021. [1](#), [2](#), [A.2](#), [A.3](#), [3](#), [H.2.2](#)

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, and Jiaming et al. Kong. Rwkv: Reinventing rnns for the transformer era. *arXiv:2305.13048*, 2023. [1](#), [2](#), [3.1](#), [4.3](#), [A.3](#), [C.2](#), [2](#), [G](#), [H.2](#), [H.2.2](#), [11](#)

Daniel Y. Fu, Simran Arora, Jessica Grogan, Isys Johnson, Sabri Eyuboglu, Armin W. Thomas, Benjamin Spector, Michael Poli, Atri Rudra, and Christopher Ré. Monarch mixer: A simple sub-quadratic gemm-based architecture, 2023b. [1](#), [2](#), [A.3](#)

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014. [1](#), [A.1](#)

Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. *Advances in neural information processing systems*, 29, 2016. [1](#), [A.1](#), [H.7.1](#)

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, et al. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 1, 2021. [1](#), [A.1](#)

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022. [1](#), [3.2](#), [4.3](#), [A.1](#), [E.1](#)

Shahar Lutati, Itamar Zimerman, and Lior Wolf. Focus your attention (with adaptive iir filters), 2023. [1](#)

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. In *Transactions of the Association for Computational Linguistics*, volume 8, 2020. 1, 6

William Merrill, Ashish Sabharwal, and Noah A. Smith. Saturated transformers are constant-depth threshold circuits. In *Transactions of the Association for Computational Linguistics*, volume 10, 2022. 1, 6

Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. On the computational complexity of self-attention. In *34th International Conference on Algorithmic Learning Theory*, volume 201, page 1–23, 2023. 1, 6

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020. 2, 3.1, D, D.1

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, and Shruti Bhosale. Llama 2: Open foundation and fine-tuned chat models. *arXiv:2307.09288*, 2023. 2, 3.1, C.2, G

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. volume 30, 2017. 2, A.3, C.2

James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. 2, A.2

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022. 2, 4

Karan Goel, Albert Gu, Chris Donahue, and Christopher Ré. It’s raw! audio generation with state-space models. *Proceedings of the 39 th International Conference on Machine Learning*, 2022. 2

Michael Zhang, Khaled Saab, Michael Poli, Tri Dao, Karan Goel, and Christopher Ré. Effectively modeling time series with simple discrete state spaces. *International Conference on Learning Representations*, 2023. 2

Shuangfei Zhai, Walter Talbott, Nitish Srivastava, Chen Huang, Hanlin Goh, Ruixiang Zhang, and Josh Susskind. An attention free transformer. *arXiv preprint arXiv:2105.14103*, 2021. 2

Alex Andonian, Quentin Anthony, Stella Biderman, Sid Black, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Michael Pieler, Jason Phang, Shivanshu Purohit, Hailey Schoelkopf, Dashiell Stander, Tri Songz, Curt Tigges, Benjamin Thérien, Phil Wang, and Samuel Weinbach. GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch, 9 2023. URL <https://www.github.com/eleutherai/gpt-neox>. 3.1

Ramin Hasani, Mathias Lechner, Tsun-Huang Wang, Makram Chahine, Alexander Amini, and Daniela Rus. Liquid structural state-space models. *arXiv preprint arXiv:2209.12951*, 2022. 3.1, A.3

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023. 3.1, A.3, D.1.2, 3, 6, H.2, H.2.3, 12

Sabri Eyuboglu, Maya Varma, Khaled Saab, Jean-Benoit Delbrouck, Christopher Lee-Messer, Jared Dunnmon, James Zou, and Christopher Ré. Domino: Discovering systematic errors with cross-modal embeddings. In *International Conference on Learning Representations*, March 2022. 3.1

David J Willshaw, O Peter Buneman, and Hugh Christopher Longuet-Higgins. Non-holographic associative memory. *Nature*, 222(5197):960–962, 1969. 3.1, A.1

John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982. 3.1, A.1

P. Bürgisser, T. Lickteig, M. Clausen, and A. Shokrollahi. *Algebraic Complexity Theory*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 1996. ISBN 9783540605829. URL <https://books.google.com/books?id=dYcgjfXsYk8C>. 4.1, H.13

Tri Dao, Nimit S Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. *arXiv preprint arXiv:2012.14966*, 2020. 4.1, H.3, H.4, H.14, H.17, H.4, H.19

Daniel Y. Fu, Elliot L. Epstein, Eric Nguyen, Armin W. Thomas, Michael Zhang, Tri Dao, Atri Rudra, and Christopher Ré. Simple hardware-efficient long convolutions for sequence modeling. *arXiv preprint arXiv:2302.06646*, 2023c. 4.3, A.3, C.2, 10, 13

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and et al. Big bird: Transformers for longer sequences. *Proceedings of NeurIPS*, 2020. 5, 4

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019a. 5

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020. 5, D.1.2, 4

Liliang Ren, Yang Liu, Shuohang Wang, Yichong Xu, Chenguang Zhu, and ChengXiang Zhai. Sparse modular activation for efficient sequence modeling. *Sparse Modular Activation for Efficient Sequence Modeling*, 2023. 5, A.3

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022. 6

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023. 6

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020a. URL <https://arxiv.org/abs/2006.16236>. 6

Chris Olah. Mechanistic interpretability, variables, and the importance of interpretable bases: An informal note on some intuitions related to mechanistic interpretability, 2022. URL <https://transformer-circuits.pub/2022/mech-interp-essay/index.html>. A.1, A.3

Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv:2201.02177*, 2022. A.1

Nick Cammarata, Shan Carter, Gabriel Goh, Chris Olah, Michael Petrov, Ludwig Schubert, Chelsea Voss, Ben Egan, and Swee Kiat Lim. Thread: circuits. *Distill*, 5(3):e24, 2020. A.1

Dingquan Wang and Jason Eisner. The galactic dependencies treebanks: Getting more data by synthesizing new languages. *Transactions of the Association for Computational Linguistics*, 4:491–505, 2016. A.1

Jennifer C White and Ryan Cotterell. Examining the inductive bias of neural language models with artificial languages. *arXiv preprint arXiv:2106.01044*, 2021. A.1

Zeyuan Allen-Zhu and Yuanzhi Li. Physics of language models: Part 1, context-free grammar. *arXiv preprint arXiv:2305.13673*, 2023. A.1

Shauli Ravfogel, Yoav Goldberg, and Tal Linzen. Studying the inductive biases of rnns with synthetic variations of natural languages. *arXiv preprint arXiv:1903.06400*, 2019. A.1

Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. *arXiv preprint arXiv:2111.02080*, 2021. A.1, G

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020. A.1

JA Feldman, GE Hinton, and JA Anderson. Parallel models of associative memory, 1981. [A.1](#)

Wei Zhang and Bowen Zhou. Learning to update auto-associative memory in recurrent neural networks for improving sequence memorization. *arXiv preprint arXiv:1709.06493*, 2017. [A.1](#)

Stefano Massaroli, Michael Poli, Daniel Y Fu, Hermann Kumbong, David Romero, Rom Parnichukun, Aman Timalsina, Quinn McIntyre, Beidi Chen, Atri Rudra, Ce Zhang, Christopher Ré, Stefano Ermon, and Yoshua Bengio. Laughing hyena distillery: Extracting compact recurrences from convolutions. 2023. [A.2](#)

David W. Romero, Anna Kuzina, Erik J. Bekkers, Jakub M. Tomczak, and Mark Hoogendoorn. Ckconv: Continuous kernel convolution for sequential data. 2022. [A.3](#)

Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces, 2022. [A.3](#)

Albert Gu, Ankit Gupta, Karan Goel, and Christopher Ré. On the parameterization and initialization of diagonal state space models, 2022. [A.3](#)

Harsh Mehta, Ankit Gupta, Ashok Cutkosky, and Behnam Neyshabur. Long range language modeling via gated state spaces, 2022. [A.3](#)

Jimmy T. H. Smith, Andrew Warrington, and Scott W. Linderman. Simplified state space layers for sequence modeling, 2023. [A.3](#)

Eric Nguyen, Michael Poli, Marjan Faizi, Armin Thomas, Callum Birch-Sykes, Michael Wornow, Aman Patel, Clayton Rabideau, Stefano Massaroli, Yoshua Bengio, Stefano Ermon, Stephen A. Baccus, and Chris Ré. Hyenadna: Long-range genomic sequence modeling at single nucleotide resolution, 2023. [A.3](#)

Transformers revolutionized ai. what will replace them?, 2023. URL <https://www.forbes.com/sites/robtoews/2023/09/03/transformers-revolutionized-ai-what-will-replace-them/?sh=6ed698269c1f>. [A.3](#)

Jue Wang, Wentao Zhu, Pichao Wang, Xiang Yu, Linda Liu, Mohamed Omar, and Raffay Hamid. Selective structured state-spaces for long-form video understanding. In *CVPR*, 2023. [A.3](#)

Brandon Yang, Gabriel Bender, Quoc V. Le, and Jiquan Ngiam. Condconv: Conditionally parametrized convolutions for efficient inference. In *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 2020. [A.3](#)

Chrysoula Kosma, Giannis Nikolentzos, and Michalis Vazirgiannis. Time-parameterized convolutional neural networks for irregularly sampled time series. 2023. [A.3](#)

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention, 2020b. [A.3](#)

Together. Redpajama: An open source recipe to reproduce llama training dataset, 2023. URL <https://github.com/togethercomputer/RedPajama-Data>. [D](#), [D.2](#), [4](#)

Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. *Association of Computational Linguistics*, 2019. [D.1.2](#)

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. [1](#)

Anonymous. Mamba: Linear-time sequence modeling with selective state spaces. 2023a. URL <https://openreview.net/forum?id=AL1fq05o7H>. [3](#), [F.1](#)

Anonymous. Gateloop: Fully data-controlled linear recurrence for sequence modeling. 2023b. URL <https://openreview.net/pdf?id=02Ug9N8DCI>. [3](#), [F.1](#)

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019b. [4](#)

Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. *arXiv preprint arXiv:1911.02972*, 2019. 4

Michael T Heideman and C Sidney Burrus. *Multiplicative complexity, convolution, and the DFT*. Springer, 1988. H.1.1

Ilya Volkovich. A guide to learning arithmetic circuits. In *Conference on Learning Theory*, pages 1540–1561. PMLR, 2016. H.1.1

Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. *Proceedings of the 40th International Conference on Machine Learning*, 2023b. H.2.1

Thathachar S Jayram, Ravi Kumar, and Dandapani Sivakumar. The one-way communication complexity of hamming distance. *Theory of Computing*, 4(1):129–135, 2008. H.23

Ofir Press, Noah A Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*, 2021. H.7.2

Selim G Akl and Henk Meijer. Parallel binary search. *IEEE Transactions on Parallel & Distributed Systems*, 1(02):247–250, 1990. H.7.4, H.7.4

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022. H.32

Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, 1983. H.35

Chris Chatfield. The analysis of time series: An introduction, fifth edition. 1995. H.8.1

Appendix

The appendix includes the following content:

1. Appendix A provides an extended discussion of related work and concepts.
2. Appendix B provides a code implementation of the BASECONV architecture.
3. Appendix C gives details for the experiments, including model architectures and hyperparameters.
4. Appendix D provides additional analysis of how MQAR appears in real data across a variety of language distributions.
5. Appendix E provides a formal definition of the MQAR problem, synthetic construction procedure, and experimental details.
6. Appendix F provides additional synthetic experiments and analysis.
7. Appendix G provides experiments and analysis for how the MQAR gap changes as we scale the gated convolution and attention architectures.
8. Appendix H gives proofs and additional discussion for the theoretical analysis in our work.

A Extended Related Work

We are inspired by and build on prior work in mechanistic interpretability (Appendix A.1), efficient architecture design (Appendix A.2), and input-dependent architectures (Appendix A.3).

A.1 Mechanistic Interpretability, Synthetic Languages, and Associative Recall

Work in mechanistic interpretability aims to decompose the capabilities of a neural network into human-understandable algorithms that can be attributed to specific parameters in the model [Olah, 2022, Power et al., 2022, Elhage et al., 2021, Cammarata et al., 2020]. Some of these works, use synthetic data to validate mechanistic interpretations of neural networks [Olsson et al., 2022]. This relates to a broader line of work using synthetic languages to study language model architectures [Wang and Eisner, 2016, White and Cotterell, 2021, Allen-Zhu and Li, 2023, Ravfogel et al., 2019, Xie et al., 2021]. *Mechanistic design* puts mechanistic interpretations to use in designing new architectures and learning algorithms. Several works in architecture research have used synthetic tasks to validate architecture designs [Kitaev et al., 2020].

Our work is focused on one particular synthetic task: *associative recall*. Motivated by psychological models of how humans associate and retrieve information, work in the early days of neural network research focused on developing systems capable of associative recall [Willshaw et al., 1969, Feldman et al., 1981, Hopfield, 1982]. For example, Hopfield networks, proposed in 1982, are a recurrent neural network explicitly designed to support associative (“content-addressable”) memory [Hopfield, 1982]. More recently, several notable recurrent neural network mechanisms (*e.g.* neural turing machines, LSTMs, and RNNs with fast weights) were evaluated on a synthetic version of associative recall [Graves et al., 2014, Ba et al., 2016, Zhang and Zhou, 2017, *inter alia*]. These works use a formulation of associative recall very similar to the single-query associative recall in our work. Since the rise of large language models, several works have argued that LLMs ability to perform *in-context* learning is due, at least in part, to the associative recall capabilities of attention [Elhage et al., 2021, Olsson et al., 2022].

A.2 Efficient Language Modeling Architectures

We first briefly review the efficiency motivations for the recent excitement around gated convolution architectures. While attention requires compute that scales as $\mathcal{O}(N^2)$ in sequence length N , convolutions scale as $\mathcal{O}(N \log N)$ [Cooley and Tukey, 1965]. Ideal, inference complexity is $\mathcal{O}(1)$ in sequence length, as provided by recurrent neural networks. State-space models can be computed either as a convolution or recurrence, to achieve both sub-quadratic training and constant inference complexity in sequence length [Gu et al., 2021].

Architectures that use *implicit* convolutional filters [Poli et al., 2023a], can be converted to an SSM via a simple distillation step [Massaroli et al., 2023].

A.3 Input-Dependence in Sequence Modeling Architectures

In an input-dependent sequence model, the way tokens are aggregated across the sequence is controlled by the data, not just the model parameters. We highlight several prior works related to our study that explore input-dependent sequence models.

- **Attention** [Vaswani et al., 2017] achieves input-dependent sequence mixing since the Q , K , and V terms are achieved via linear combinations of the input x . For instance, $Q = xW_Q$ for some learned weight matrix W_Q .
- **Input-Independent Convolution Architectures.** Given the quadratic scaling of attention, Gu et al. [2021] propose S4, an architecture that uses long convolutions to process the input. CKConv [Romero et al., 2022] uses a convolution filter that is implicitly parametrized by an MLP of the relative positions of the observations in a time-series. A line of subsequent work improved upon S4 by changing the parametrization [Gupta et al., 2022, Gu et al., 2022, Mehta et al., 2022, Ma et al., 2022, Fu et al., 2023c] or changing the SISO convolutions to MIMO convolutions [Smith et al., 2023]. The convolution filter in each of these architectures is input-independent. The linear convolution layer alone cannot perform MQAR [Fu et al., 2023a].
- **Input-Dependence via Gating.** Since pure convolution architectures with input-independent filters struggle to perform associative recall, an important task in in-context learning [Olah, 2022], subsequent work proposed to introduce input-dependence by adding a *gating* or element-wise multiplication operation to the architecture, where both the inputs to the operator are defined in terms of the input [Fu et al., 2023a, Poli et al., 2023a]. The multiplication is generally $y = \sigma(Wx) \odot x$. The filters remain input-independent. These architectures demonstrate promising results on the associative recall synthetics proposed in prior work [Fu et al., 2023a], [Poli et al., 2023a] and provide large downstream improvements over S4.

Prior work suggests the models match attention in quality [Poli et al., 2023a, Fu et al., 2023b, Peng et al., 2023] and this has led to increasing use of these architectures [Nguyen et al., 2023, toe, 2023], however our results suggest there is still a sizable gap to attention. We show the gap is largely due to the models’ recall abilities and capture this behavior in a novel synthetic task. We theoretically show that dimension scales in sequence length when solving MQAR with gated convolutions.

Selective S4 Wang et al. [2023] learns which positions to mask prior to passing the input to the next long convolution (S4) layer. This resembles gating (which also controls which information flows forwards to the next layer).

- **Input-Dependent Convolution Architectures** Prior work introduces convolution architectures with input-dependent filters [Yang et al., 2020, Kosma et al., 2023]. For instance, Yang et al. [2020] parametrizes the convolution filter as a linear combination of n projections of the input. Liquid S4 [Hasani et al., 2022] is also motivated by introducing input-dependence to S4. To accomplish this, the work proposes including correlation terms between the input tokens during state mixing. We evaluate Liquid S4 and find the architecture lags attention on the Pile by 4.4 perplexity points in Table 3.

Model	Param (M)	Overall		Slices		% of gap due to
				AR Hits	Other Tokens	
Attention	125	12.37 (2.52)	2.32 (0.84)	14.04 (2.64)	—	—
Liquid S4	145	16.80 (2.82)	22.75 (3.12)	16.42 (2.80)	52.6%	—

Table 3: Validation perplexity on the Pile after pretraining on 5B tokens. We report log perplexity with negative log-likelihood in parentheses. AR vs. non-AR token slices are defined in Section 3.

- **Recurrent Architectures** The pure long-convolution architectures can also be computed as recurrences. Letting s_i represent the hidden state at time i , u_i be the input at time i , and A, B, C be projection matrices, the recurrence computes:

$$s_{i+1} = f(As_i + Bu_i)$$

$$y_i = Cs_i$$

In a linear RNN like S4, the A , B , and C matrices are input-independent and f is an identity function. However, recent work proposes input-dependent RNNs (e.g. RetNet [Sun et al. \[2023\]](#)), where a subset of these matrices is input-dependent (e.g. $A = xW_a$). Note that linear attention mechanisms can also be expressed as input-dependent RNNs [\[Katharopoulos et al., 2020b\]](#). While these architectures improve over gated convolutions in MQAR synthetic experiments and downstream quality on the Pile (Appendix F), we observe and prove that the required RNN hidden state dimensionality grows with the number of key-value pairs that the model needs to recall in the sequence (Appendix H.6). In contrast, attention complexity does not scale with the number of key-value pairs to recall.

- **Hybrid Architectures** Finally, we can combine architectural components that provide different capabilities. Prior work proposes architectures that hybridize sub-quadratic layers and attention [\[Fu et al., 2023a, Ma et al., 2022, Ren et al., 2023\]](#), but does not motivate this choice from a mechanistic design perspective. Further, H3 without attention [\[Fu et al. 2023a\]](#) and MEGA [\[Ma et al., 2022\]](#), which uses blocked attention, underperform attention on language modeling.

SeqBoat [\[Ren et al., 2023\]](#) shares closest resemblance to our learned selection module. SeqBoat introduces a different module that learns where to use attention however, their model can end up using full attention at a layer, providing quadratic scaling in the worst case. Different from SeqBoat, we use an auxiliary loss to encourage sparsity in the selected attention positions and select the top- k positions to remain sub-quadratic.

Overall, our work finds that previously proposed sub-quadratic models do not efficiently solve the MQAR task.

B Code Listing

In this section, we include code listings for the BASECONV operator. We begin with the a standard BASECONV implementation that uses an explicitly-parameterized long convolution. Below, we also provide the implementation for BASECONVwith an implicitly-parameterized long convolution.

```

1 import torch
2
3 def fft_conv(u: torch.Tensor, k: torch.Tensor):
4     """
5         Args:
6             u (torch.Tensor): (batch_size, d_model, seq_len)
7             k (torch.Tensor): (d_model, l_max)
8         Returns:
9             y (torch.Tensor): (batch_size, d_model, seq_len)
10    """
11    seqlen = u.shape[-1]
12    fft_size = 2 * seqlen
13    k_f = torch.fft.rfft(k, n=fft_size) / fft_size
14    u_f = torch.fft.rfft(u.to(dtype=k.dtype), n=fft_size)
15    y = torch.fft.irfft(u_f * k_f, n=fft_size, norm="forward")[..., :seqlen]
16    return y
17
18 class BaseConv(torch.nn.Module):
19
20     def __init__(self, d_model: int, l_max: int, **kwargs):
21         super().__init__()
22         self.d_model, l_max = d_model, l_max,
23         self.projection = torch.nn.Linear(self.d_model, self.d_model)
24         self.filter = torch.nn.Parameter(torch.randn(self.d_model, l_max), requires_grad=True)

```

```

25
26     def forward(self, u: torch.Tensor):
27         """
28             Args:
29                 u (torch.Tensor): (batch_size, d_model, seq_len)
30             Returns:
31                 y (torch.Tensor): (batch_size, d_model, seq_len)
32         """
33         u_conv = fft_conv(u.transpose(1, 2), self.filter).transpose(1, 2)
34         u_proj = self.projection(u)
35         y = u_conv * u_proj
36         return y + u

```

Listing 1: **Explicit BaseConv implementation.** Implementation of the BaseConv layer with explicitly-parameterized long convolutions. The implementation is 19 lines excluding comments and whitespace.

In some of our experiments, we interleave BASECONV layers that use explicit short convolution filters (like those in `torch.nn.Conv1d`) and with BASECONV layers that use implicit long convolution filters (like those described in Poli et al. [2023a]).

Below we include the code for BASECONV with an implicit convolution.

```

1  class PositionalEmbedding(nn.Module):
2      def __init__(self, emb_dim: int, seq_len: int, **kwargs):
3          """Complex exponential positional embeddings for implicit long convolution filters.
4          """
5          super().__init__()
6          t = torch.linspace(0, 1, seq_len)[None, :, None]  # 1, L, 1
7          bands = (emb_dim - 1) // 2
8          t_rescaled = torch.linspace(0, seq_len - 1, seq_len)[None, :, None]
9          w = 2 * math.pi * t_rescaled / seq_len  # 1, L, 1
10         f = torch.linspace(1e-4, bands - 1, bands)[None, None]
11         z = torch.exp(-1j * f * w)
12         z = torch.cat([t, z.real, z.imag], dim=-1)
13         self.z = nn.Parameter(z, requires_grad=False)
14
15     def forward(self, L):
16         return self.z[:, :L]
17
18 class BaseImplicitConv(nn.Module):
19     """
20         BaseConv with implicit filter parameterized by an MLP.
21
22     Args:
23         d_model (int): Number of expected features in input and output.
24         l_max (int): The maximum sequence length.
25         d_emb (int, optional): Dimension of the positional embeddings. Must be odd and $\geq$ 3 (time, sine and cosine). Defaults to 3.
26         d_hidden (int, optional): The number of features in the hidden layer of the MLP.
27         Defaults to 16.
28     """
29
30     def __init__(self, d_model: int, l_max: int, d_emb: int=3, d_hidden: int = 16,):
31         """
32             Long convolution with implicit filter parameterized by an MLP.
33         """
34         super().__init__()
35         self.pos_emb = PositionalEmbedding(d_emb, l_max)
36         self.filter_mlp = nn.Sequential(nn.Linear(d_emb, d_hidden), torch.nn.ReLU(), nn.
37             Linear(d_hidden, d_model))
38         self.projection = torch.nn.Linear(d_model, d_model)
39
40     def forward(self, u: torch.Tensor, *args, **kwargs):
41         """
42             Args:
43                 u (torch.Tensor): (batch_size, seq_len, d_model)
44             Returns:
45

```

```

43     y (torch.Tensor): (batch_size, seq_len, d_model)
44     """
45     filter = self.filter_mlp(self.pos_emb(u.shape[1])).transpose(1, 2)
46     u_conv = fft_conv(u.transpose(1, 2), filter).transpose(1, 2).to(dtype=u.dtype)
47     u_proj = self.projection(u)
48     y = u_conv * u_proj
49     return y + u

```

Listing 2: **Implicit BaseConv implementation.** Implementation of the BASECONV layer with implicitly-parameterized long convolutions. (The implementation is 34 lines excluding comments and whitespace).

C Downstream Experimental Details

We use A100 80GB Nvidia GPUs to run all experiments. We use the reference training infrastructure from <https://github.com/EleutherAI/gpt-neox> for all pretraining runs. The Pile data is tokenized using the GPT2BPETokenizer and all models see the data in the same order.

Below we provide details on the hyperparameters and settings for training each architecture studied in the paper on the real-world Pile data. In Appendix C.1 we summarize and justify our method for measuring the quality gap due to associative recall between the convolution architectures and attention. In Appendix C.2, we provide details on the pure gated convolution architectures, attention baseline, and hybrid architectures studied in the paper.

C.1 Measuring the Mqar Gap on Real Language Data

Here we summarize our method for computing the amount of quality gap between the convolution and attention models that is ascribed to associative recall capability (e.g., in Table 1):

1. Given an input sequence, we identify recurring bigrams (i.e. bigrams that have already appeared in the sequence at a prior position). Since bigrams that appear frequently during training may be memorized by the model, rather than requiring the model to perform recall at inference-time, we only measure AR log-probabilities with respect to bigrams that are seen fewer than a threshold number of times during training. The threshold used in all the experiments in our submission is 1,250 training occurrences in the 10B tokens of pretraining data.
2. We measure the log-probability assigned to the true bigram completion. This bigram completion is referred to as an AR Hit in our work. This protocol assumes that the model can produce the completion by recalling the prior occurrence of the bigram in the sequence.
3. For the model being evaluated m , and the attention model M , we measure the % of the quality gap between m and M ascribed to associative recall capability as follows. Let the average log-probability for all AR Hits across validation sequences be l_H^m and l_H^M for m and M respectively. Let the average log-probabilities of *all tokens* in the validation sequences be l^m and l^M respectively. Let p_H be the proportion of AR Hit tokens in the validation data. As the final gap ascribed to AR, we report:

$$\min\left(\frac{(l_H^m - l_H^M)p_H}{l^m - l^M}, 1.0\right)$$

Shown above, if m is better than attention (M) overall and M is better than m at AR, we ascribe 100% of the gap to AR.

We briefly discuss two important decisions in this protocol. First, we only measure **explicit bigrams**, i.e. bigrams are identified based on token ids in the sequence. However, intuitively, models may also perform associative recall between related *concepts* produced by a contextual language model. For instance, language may contain bigrams in which one word is swapped by a synonym. As another example, a model may see a sentence such as “The iPhone is outside my budget so I instead purchased an Android phone. ... It was much ...” and predict “cheaper” for the blank, recalling that the sequence is discussing cost. Our work does not measure such fuzzy (more abstract) recall instances.

Next, we measure the gap based on **log-probabilities** rather than perplexity. This is simply because we want to make our metrics independent of the number of tokens in each of the slices of the validation set. Approximately 6.4% of validation tokens are AR Hits with the threshold set to consider bigrams seen less than 1,250× during training.

C.2 Gated Convolution Downstream Architectures

We evaluate over 4 previously proposed architectures as well as BASECONV, the theoretically “canonical” representation for gated convolutions, introduced in Section 4, for a total of 14 training runs. Here we provide details on the hyperparameters and configurations used for training each architecture. We also provide details on the FLOPs computation.

- **Attention** [Vaswani et al., 2017, Touvron et al., 2023] We train using the the specifications in Table 6. The parameters are sourced from the Transformer implementation in <https://github.com/EleutherAI/gpt-neox>.
- **Hyena** [Poli et al., 2023a] We train using the specifications in Table 8. The parameters are sourced from the Appendix of Poli et al. [2023a] and the implementation is sourced from the provided reference at <https://github.com/HazyResearch/safari>.
- **H3** [Fu et al., 2023a] We train using the specifications in Table 10. The hyperparameters and implementation use the reference at <https://github.com/HazyResearch/H3>.
- **RWKV** [Peng et al., 2023] We train using the specifications in Table 11. The parameters are sourced from the Appendix of Peng et al. [2023] and the details provided in the reference implementation at <https://github.com/BlinkDL/RWKV-LM>. We specifically evaluate RWKV-V4.
- **Pure long convolution** We train using the specifications in Table 13. We evaluate a simple long convolution based model with *no gating* as a reference point. While this is a generic architecture, we use the reference implementation and initializations/regularizations from recent work Fu et al. [2023c]. The implementation is provided at <https://github.com/HazyResearch/safari>.
- **BaseConv** We train using the specifications in Table 15. The implementation, amounting to 19 lines of PyTorch, is shown in Appendix B.

We provide the equations used to compute the FLOPs for each model, letting D be the model width, H the head dimension, L the depth, N the sequence length, V the vocabulary size, and B the batch size. FLOPs equations for different architecture types are provided in Table 7 (attention), Table 9 (Hyena and adapted for H3), Table 14 (pure long convolution), and Table 16 (BASECONV). We compute FLOPs for RWKV as in the Appendix of [Peng et al., 2023], based on the number of linear layer parameters, plus input and language modeling head FLOPs.

Hybrid Architectures In the main paper and appendix, we evaluate a series of architectures with a hybrid of gated convolution and non gated convolution layers. For these architectures, we use the same number of layers as the pure gated convolution architecture (as reported in Appendix C.2). The hybridized (non gated convolution) layers are inserted as replacements of the gated convolution layer. For each architecture, we simply evenly intersperse the two types of layers. We evaluate with the following replacement layers in this work:

- **Full attention** following the specification of attention in Appendix C.2.
- **Random selection** We evaluate sparse attention randomly applied to tokens as introduced in Section 5.
- **Programmatic selection** We apply sparse attention on AR hit tokens as introduced in Section 5. When processing the input sequence, we can causally determine if a bigram of raw token ids is repeated in the sequence to construct the attention pattern.
- **Learned selection** We *learn* the positions on which to use attention by introducing a simple linear layer with sigmoid activation to the architecture that is trained to output high scores if attention should

be applied to the token. We can take the top- k positions to control the computational complexity of the layer. This approach is introduced in Section 5.

We use these protocols to validate that input-dependence suffices to address the associative recall gap and highlight that there are varied approaches to incorporating input-dependence in an architecture.

D Extended Downstream Analysis of Mqar

In this section, we provide additional analysis of how MQAR manifests in real language data. We extend our discussion of associative recall on the Pile [Gao et al. \[2020\]](#). We also perform analysis on sources in RedPajama including ArXiv and StackOverflow [Together \[2023\]](#).

D.1 Additional Discussion of Mqar in the Pile

The Pile is a widely popular language modeling corpus [Gao et al. \[2020\]](#).

1. First, in Appendix D.1.1, we provide several real examples of *how* associative recall occurs in the Pile to demonstrate it’s role in language modeling. We color code the tokens to highlight differences in the next token predictions from Attention, RWKV and Hyena models.
2. Next, in Appendix D.1.2, we explain *where* associative recall hits tend to occur in sequences. This is useful to guide the design of sequence mixers — if associative recall tends to occur in specific places, the input-dependent sequence mixer may not need to compute all N^2 token-to-token interactions for every sequence.

D.1.1 Pile Examples

For the examples below, the legend is: tokens are colored as **both correct**, **both incorrect**, **Attention correct** and **Hyena incorrect**, **Attention incorrect** and **Hyena correct**. For tokens where the models disagree, the predictions of each model are provided in parentheses.

```
while lunch -ing at the Ma -ison Ber -gey b -ist -ro near his apartment : he had been mus
-ing about(rwkv= about, attn= on) the ... (723 tokens)... the young waitress -'s sigh at
the Ma -ison Ber(rwkv= Bl, attn= Ber) -gey(rwkv=-nd, attn=-gey)
```

Example D.1: Comparing next token predictions of a 350M parameter Attention model and a 350M parameter RWKV model. In this example, the models need to perform associative recall to correctly predict “Bergey” in the 4-gram Ma-ison Ber-gey. The previous mention of the 4-gram was more than 700 tokens earlier in the passage.

```
The second(rwkv= first, attn= second) section is all about Pixar Fest , and the
(rwkv= third, attn= the) final section is all(rwkv= about, attn= all) about Pixar Pier
. ... (480 tokens)... -If(rwkv=-Disney, attn=-If) there wasn -âG -t enough Pixar at
Disneyland , Pixar Fest(rwkv= would, attn= Fest) is(rwkv= at, attn= is) coming to the
Disneyland Resort on April 13 , 2018 .
```

Example D.2: Comparing next token predictions of a 350M parameter Attention model and a 350M parameter RWKV model. In this example, the models need to perform associative recall to correctly predict “Fest” in the bigram Pixar Fest.

David Mus -a P -id -cock is an indigenous English revert to(rwkv=-ant, attn= to) Islam ,(rwkv=, attn= who) who formed the Islamic Party of Britain in 1989, at London Central Mosque(rwkv= Mosque, attn= University) .(rwkv=, attn= in) ... (711 tokens)... -I have just found out that David Mus(rwkv= Cameron, attn= Mus) -a P(rwkv= (, attn= P) -id(rwkv=-asha, attn=-id) -cock(rwkv=-ham, attn=-cock) is(rwkv= (, attn= is) speaking at a Yorkshire Forum debate today in Bradford with

Example D.3: Comparing next token predictions of a 350M parameter Attention model and a 350M parameter RWKV model. In this example, the models need to perform associative recall to correctly predict a middle and last name. Note that the middle and last names span several tokens.

-The most common map elements(hyena= in, attn= elements) in Sub -Space are prizes , or ” -g(hyena==points, attn==g) -reens ”(hyena=”,, attn=") (-for their green color -). Pri -zes allow players to upgrade their ships and gain(hyena= other, attn= gain) special weapons or abilities(hyena= abilities, attn= upgrades) . While prizes are generally plent -ifuly scattered throughout the map(hyena= map, attn= game) , the upgrades or abilities(hyena= bonuses, attn= abilities) they award are randomly selected by the zone . -Energy -Rather than dealing with ammunition counts and hit points separately , Sub -Space combines both of these elements into a single unit of measure: energy . Each ship is equipped with a certain amount of energy , from which(hyena= which, attn= the) it must draw its health as well as its weapons power (1,526 tokens)... Speed Zone proved to be less popular than the Jack -pot -/ -Running , Chaos , or ” -flag ” zone games and support was discontinued shortly after Sub -Space(hyena==space, attn==Space) went to retail .

Example D.4: Comparing next token predictions of a 350M parameter Attention model and a 355M parameter Hyena model. In this example, the models need to perform associative recall to correctly predict the name of the game “SubSpace”. Note that both models correctly perform the recall when there is a short gap between tokens, but only Attention correctly performs recall when the gap is greater than 1,000 tokens.

Thus far , no systematic study has been published on is -olation(hyena=-olation, attn=-ot) via sequential centrif -ug -ation , and no systematic analysis is ... (491 tokens)... Fig . 1(hyena= 1, attn=-A_1) -is -olation via sequential(hyena= centrif, attn= sequential) centrif -ug -ation .

Example D.5: Comparing next token predictions of a 350M parameter Attention model and a 355M parameter Hyena model. In this example, the models need to perform associative recall to correctly predict the technique “sequential centrifugation”.

Miss -ouri(hyena=-ouri, attn=-iss) Southern has had 14 Major League Base- ball draft selections since the draft began in 1965 (149 tokens)... Fred G . Hughes Stadium(hyena= Stadium, attn= Field) (-opened in 1975) is named(hyena= the, attn= named) after former(hyena= the, attn= former) J -op -lin Globe publisher and(hyena= and, attn= Fred) Missouri South- ern(hyena= State, attn= Southern) board of reg -ents member

Example D.6: Comparing next token predictions of a 350M parameter Attention model and a 355M parameter Hyena model. In this example, the models need to perform associative recall to correctly predict the name of the university “Missouri Southern”.

D.1.2 Distribution of Mqar Hit Positions in Sequences

In Figure 3, we compute and plot the distances between AR hits and its prior bigram (key-value) occurrence in the sequence across the Pile training data. The distances follow a power law distribution where most AR hits are within 100 token positions from the prior bigram occurrence, and a long tail of AR hits requires long-range interactions.

Implications This suggests that architectures which compute token-to-token interactions within windows, where the window size is less than the full sequence length, may suffice to handle MQAR in real data. For instance, sliding window attention may help with AR [Beltagy et al., 2020, Sun et al., 2023, *inter alia*]. Prior work leverages the observation that local attention captures most of the token dependencies to improve efficiency Sukhbaatar et al. [2019].

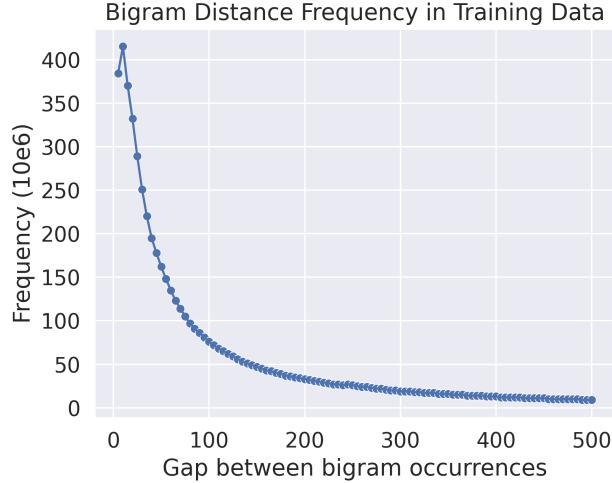


Figure 3: Across the Pile training data, we measure the distances between n -grams and their prior occurrences in the provided context. We plot the frequency across distances, finding it follows a power law distribution.

D.2 Analyzing Mqar across Varied Language Distributions

We use the HuggingFace sample of the popular RedPajama language modeling corpus Together [2023] to understand how the prevalence of MQAR hits and ways in which the hits appear vary across language distributions. We analyze text sequences from the following sources: ArXiv papers, Books, C4, Common Crawl, GitHub, Stack Exchange, and Wikipedia.

First we profile the prevalence of MQAR hits by computing the number of repeated bigram occurrences per sequence. Each sequence is 2,048 tokens in length and exclude bigrams containing NLTK stop words or punctuation from the computation. The prevalence of hits and the distances between hit tokens and the prior bigram occurrence in context vary across distributions as shown in Figure 4 and 5.

Arxiv, Github, and Stack Exchange contain relatively structured or richly formatted language data. We observe that the prevalence of MQAR hits is relatively high in these three sources in Figure 4. We inspect why these documents contain many bigrams. In Arxiv, we find domain-specific terminology, groups of related citations, Latex commands, and mathematical equations are frequently repeated within the same document. In GitHub and Stack Exchange, we find function and variable names are frequently reused. Meanwhile, the C4, CC, Books, and Wikipedia bigrams are highly variable. The topics and documents in these corpora are relatively diverse and unstructured.

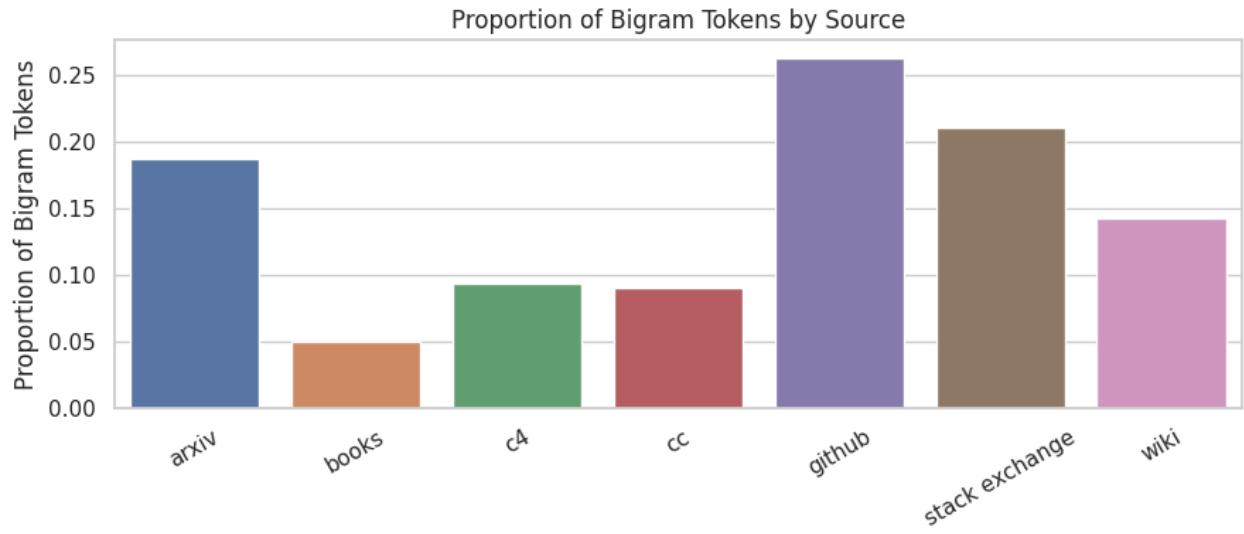


Figure 4: MQAR hits by sub-source of the RedPajama language corpus [Together \[2023\]](#). Hits are measured using the repeated bigrams in each sequence of length .

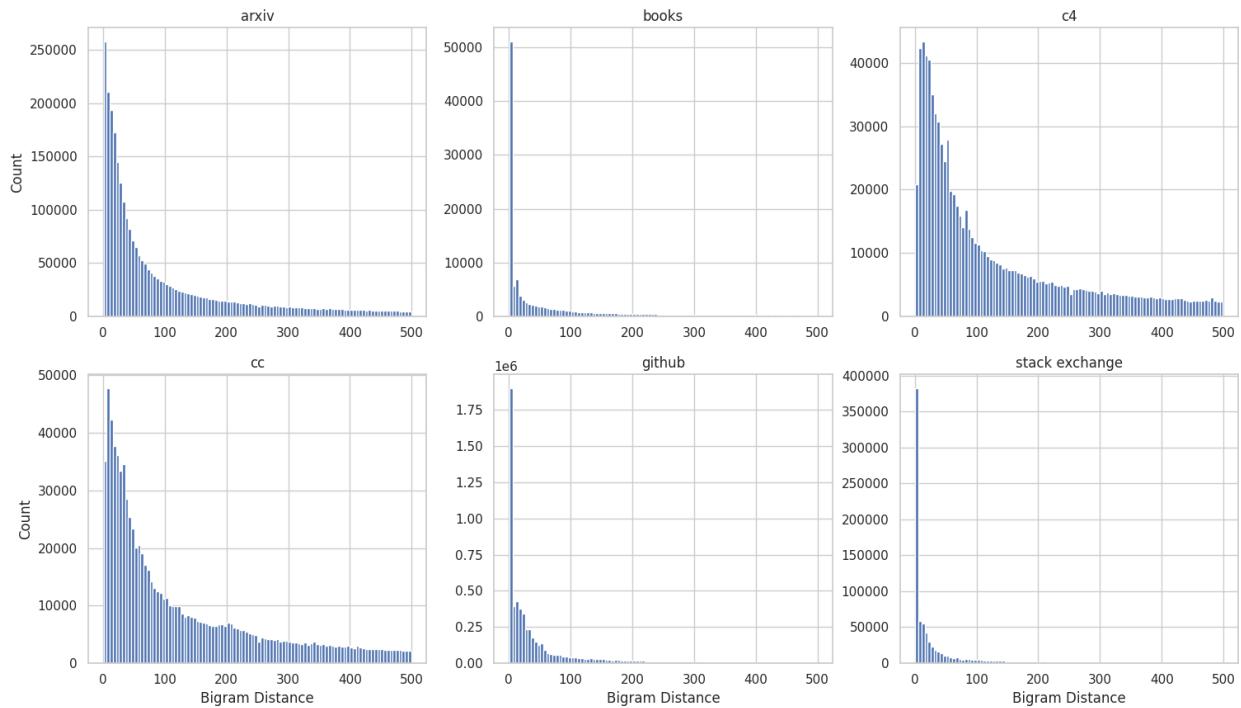


Figure 5: We measure the distance between n -grams and their prior occurrences in the provided context across sequences from each source. We plot the frequency across distances, finding it follows a power law distribution.

If the placement V of the hyperplanes is not generic, then the induced subdivision Σ is not a triangulation. However, the above bijection is unaffected and, in particular, the fine type of a vertex of CD_A can be read off the corresponding facet of Σ . We define the **crosscut complex** $ccut\Sigma \subseteq 2^{[n] \times [d]}$ to be the unique simplicial complex with the same vertices-in-facets incidences as the **polyhedral complex** Σ . The **crosscut complex** is a standard notion in combinatorial topology and can be defined in more generality (see Björner). The following observation is immediate from the definitions.

Proposition D.1. *For V an ordered sequence of n points in T^{d-1} let $A = A(V)$ be the corresponding tropical arrangement and let Σ be the regular subdivision of $\Delta_{n-1} \times \Delta_{d-1}$ induced by V . Then the fine cotype ideal fCI is Alexander dual to the Stanley-Reisner ideal of $ccut\Sigma$.*

The **crosscut complex** encodes the information of which collections of vertices lie in a common face. Hence, the **crosscut complex** is a purely combinatorial object and does not see the affine structure of the underlying **polyhedral complex**.

Example D.7: ArXiv sequence where the initial bigram occurrence is highlighted in red and the repeated bigram with the MQAR hit is highlighted in blue.

```

1  /**
2   * Adds a directed edge to the graph from pt1 to pt2. Precondition: Both
3   * GeographicPoints have already been added to the graph
4   *
5   * @param from The starting point of the edge
6   * @param to The ending point of the edge
7   * @param roadName The name of the road
8   * @param roadType The type of the road
9   * @param length The length of the road, in km
10  * @throws IllegalArgumentException If the points have not already been
11  * added as nodes to the graph, if any of the arguments is null, or if the
12  * length is less than 0.
13  */
14  public void addEdge(GeographicPoint from, GeographicPoint to,
15      String roadName, String roadType, double length)
16      throws IllegalArgumentException {
17
18      if (from == null || to == null || roadName == null
19          || roadType == null || length < 0 || !mapNodes.containsKey(from)
20          || !mapNodes.containsKey(to)) {
21          throw new IllegalArgumentException();
22      }
23
24      if (!mapNodes.get(from).hasEdge(to)) {
25          mapNodes.get(from).addNeighbor(to, roadName, roadType, length);
26          numEdges++;
27      }
28  }

```

Example D.8: GitHub sequence where substrings such as “IllegalArgumentException”, “roadType” (and other variable names), “GeographicPoint” (and other data types), “containsKey” (and other function calls) are repeated throughout.

E Synthetic Mqar Experimental Details

In this paper, we propose MQAR as a useful tool to help explain gaps between three popular sequence modeling layers — Transformers or Sparse Transformers, Convolutions, and Recurrences. In this section, we detail the motivation behind the design of MQAR and include extended experiments on additional architectures. We provide a procedure for generating MQAR synthetic data, which can help in the development of new architectures.

E.1 Mqar Generation Procedure

Here we provide additional discussion on the properties of our MQAR synthetic data. The objective of the synthetic analysis is to help explain the differences in language modeling behavior between different classes of language modeling architectures, as observed on real-world data Section 3. Synthetic recall tasks were used in the development of Hyena Poli et al. [2023a] and H3 Fu et al. [2023a], and Olsson et al. [2022] to study Transformer in-context learning behavior.

1. **Attention:** Attention can perform recall trivially (Proposition 4.3). The bound is independent of the sequence length N .
2. **Convolutions:** The gated convolution models use input-independent filters. We show in Section 4 that the dimensionality for solving recall grows with the input sequence length. Real language modeling can require performing $\mathcal{O}(N)$ recalls in one forward pass. Our intuition is that it is difficult to efficiently compute **all token-interaction distances** with such a filter.
3. **Recurrences:** Recurrent models include a single hidden state as an information bottleneck. We show in Appendix H.6 that the gated recurrence requires $\Omega(N)$ bits to solve MQAR for $d \leq \sqrt{N}$ for model dimension d . Our intuition is that it is difficult to store **large numbers of key-value pairs** seen in the sequence in low-dimensional hidden states.

Motivated by this analysis, we propose Procedure 1 for generating synthetic MQAR data. This procedure allows toggling two simple properties of the synthetic dataset, specified below, to reveal the differences between the sequence modeling architecture families. We also discuss deviations from the prior work that uses synthetic recall data.

1. **Size of the Key-Value Map:** The number of unique key-value pairs appearing in each example in the dataset. In Procedure 1, we use the input parameter D to toggle this value.
2. **Number of Gaps in the Data:** The number of unique token-interaction distances required to perform the recall task. Prior work assumes there is a single query token that requires recall per example, failing to test whether the architecture can support multiple token-interaction distances. In Procedure 1, we use the parameters N (sequence length), D (size of the key-value map per example), and α to toggle this property. We enforce that the token-interaction distances appearing in the synthetic examples follow a power-law distribution specified by α (based on Figure 3). Keeping α and D constant while varying N changes the number of unique token-interaction distances appearing in the example.

Finally, noting that the vocabulary size appears in the theoretical bounds for all three architecture families (Section 4), we increase the vocabulary size to be much larger than the model dimension as typically seen in language modeling (30k - 50k tokens). Prior work uses vocab sizes ≤ 40 tokens.

Algorithm 1 MQAR Synthetic Procedure

Input: Vocabulary C , Sequence length N , Power-law parameter α , Number of Key-Value Pairs D

Output: Synthetic sequence

- 1: Let the first half of C be keys K and the second half be values V .
- 2: Pair each key token $k \in K$ with a *random* value token $v \in V$.
- 3: Sub-select D random key-value pairs to include in the sequence.
- 4: Place the D key-value pairs at the start of the sequence (i.e. consuming the first $2D$ positions).
- 5: Place a second occurrence of each $d \in D$ at a distance from the first occurrence in the sequence. The distance for each $d \in D$ is selected at random from the positions $[2D..N]$, where the probability of choosing each position follows the power law distribution specified by α .
- 6: Output the synthetic sequence.

E.2 Training Details

We first describe the architectures evaluated on MQAR synthetic data and then the training hyperparameters. We evaluate the following four architecture categories in the experiments:

1. **Attention** Standard GPT-2 style multi-headed Transformer architecture with learned positional embeddings Brown et al. [2020]. Attention, the core building block of Transformers, is defined in Section 2. The architecture for synthetics has 1 attention head.
2. **Gated convolutions** (Hyena Poli et al. [2023a], RWKV Peng et al. [2023]). Gating, convolutions, and

the class of gated convolutions are defined in Section 2.

3. **Gated recurrences** (RetNet [Sun et al. \[2023\]](#)). RetNet proposes computing attention over chunks of the sequence and combining this with a recurrence over the sequence. We evaluate this architecture (“Chunked RNN”) with chunk sizes of 32 and 8 in Figure 6.

In RetNet, hidden states are updated as:

$$S_n = \gamma S_{n-1} + A_n^T V_n$$

Outputs at each timestep are defined as

$$O_n = C_n S_n$$

where $\gamma \in \mathbb{R}^1$, $A = xW_a$ for input $x \in \mathbb{N} \times$ and learned weight matrix $W_a \in \mathbb{R}^{d \times d}$, $C = xW_c$ for $W_c \in \mathbb{R}^{d \times d}$, $V = xW_v$ for $W_v \in \mathbb{R}^{d \times d}$.

RetNet similar to a state-space-model (SSM) [Gu et al. \[2021\]](#), except for that the matrices A and C are *input-dependent* (i.e. they are functions of the input x). We note that RetNet is a special case of the recently proposed Mamba [Anonymous \[2023a\]](#) and GateLoop [Anonymous \[2023b\]](#) architectures, which replace the γ term in RetNet with yet another input-dependent matrix.

The RetNet model is formally defined in Appendix H.2.3.

4. **Sparse attention** Several works have proposed methods for efficiently computing attention [Tay et al. \[2022\]](#). We evaluate two classical methods: sliding window attention [Beltagy et al. \[2020\]](#), [Zaheer et al. \[2020\]](#) and blocked window attention [Child et al. \[2019b\]](#), [Qiu et al. \[2019\]](#).

Consider a window size w . Sliding window attention permits each token to attend to the prior w tokens in the sequence. Blocked window attention first splits the sequence into blocks of size w tokens, and then computes standard causal attention within the block.

For all synthetic runs in the main paper and the appendix, we use the following training protocol:

- **Optimizer and schedule:** Weight decay 0.1, warmup duration 10%, linear warmup, AdamW optimizer. For each run, we sweep the learning rates in $\text{np.logspace}(-4, -2, 4)$. We train for 64 epochs.
- **Training duration:** The global batch size is 8 for input sequence length or model dimension ≥ 512 , 16 for input sequence length or model dimension ≥ 256 , and 64 otherwise.
- **Width and depth:** For all synthetic experiments, we use exactly two layers (each with one sequence mixer and one MLP, interleaved with layer normalization). The model dimension, sequence length, and number of KV pairs are varied based on the relevant experiment (see Appendix F).
- **Position information:** No position embeddings are used for the pure convolution runs (input-dependent nor input-independent filters). Position embeddings are used for the runs with any attention variant.
- **Data:** We train and evaluate each model on 100,000 and 3,000 data points respectively. The data and data order for all runs are constant.

F Extended Results on Mqar Across Architectures

In this section, we provide further validation that MQAR is a useful diagnostic tool to explain the behaviors of a wide variety of architecture families. We show:

1. **Gated convolutions** In the main paper, we claim the gated convolution models with input-independent filters use larger dimensionality than attention as the number of token-interaction distances required for the task increases (N , holding α and D constant in Procedure 1).
2. **Gated recurrences** In this section, we show gated recurrent models use larger dimensionality than attention to solve the task as the number of unique key-value pairs to store (D in Procedure 1) increases.
3. **Sparse attention** In this section, we validate that sliding window attention helps close the MQAR gap when the token-interaction distances are relatively short, and degrades on longer distances. Meanwhile, blocked window attention struggles to close the gap (intuitively tokens earlier in a block are able to interact with few other tokens in the sequence). Concretely, in Appendix D, we observe that repeated bigrams in the Pile and RedPajama corpora tend to occur within neighboring tokens in the context.

We finally show the architectures that perform well on MQAR also perform well on downstream real-world AR on the Pile. We use the experimental protocols outlined in Appendix E for the synthetic experiments and Appendix C for the downstream experiments in this section.

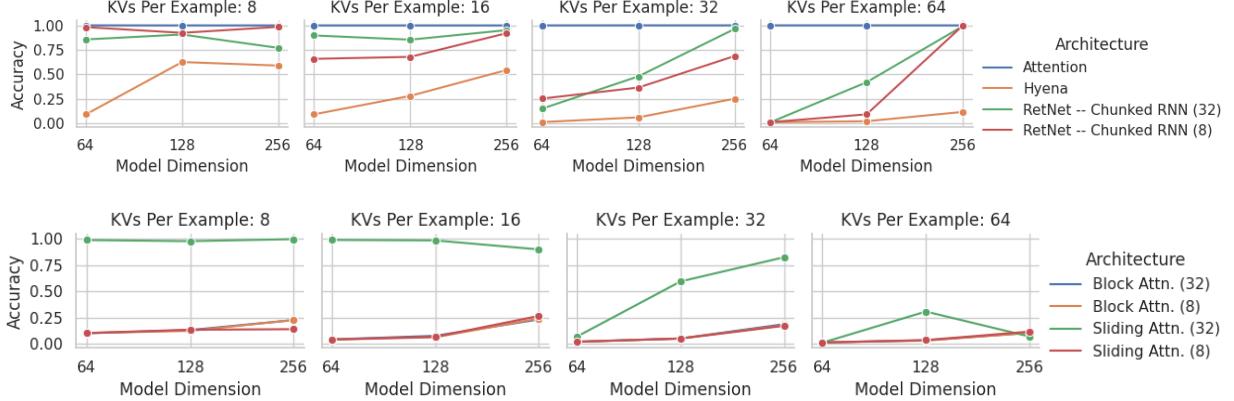


Figure 6: MQAR quality as we vary the model dimension for synthetic datasets that contain different numbers of key-value pairs per example. **(Top)** We focus on the evaluation of the input-dependent gated recurrent RetNet [Sun et al. \[2023\]](#) architecture. The architecture combines chunked attention with recurrence and we evaluate at two different chunk sizes (8 and 32). For reference, we include gated convolution (Hyena) and attention. **(Bottom)** We evaluate two efficient attention variants – sliding window and blocked window attention – at two window sizes (8 and 32).

F.1 Synthetic Experiments

Experiment 1: Increasing the number of token-interaction distances per example. To construct the synthetic dataset following Procedure 1, we set $\alpha = 0.1$ and $|C| = 8,192$ for all experiments. We vary $N \in \{64, 128, 256, 512\}$ to increase the number of token-interaction distances that will appear in the examples and we vary the model dimension $\in \{64, 128, 256, 512\}$. The results of this experiment are shown in Fig. 2, validating that the gated convolutions use larger dimensionality than attention as the number of token-interaction distances required for the task increases.

Experiment 2: Increasing the number of key-value pairs occurring per example. To construct the synthetic dataset following Procedure 1, we set $\alpha = 0.1$ and $|C| = 8,192$ for all experiments. We fix $N = 256$ and vary $D \in \{\frac{N}{32}, \frac{N}{16}, \frac{N}{8}, \frac{N}{4}\}$ to increase the number of key-value pairs occurring per example. For each number of key-value pairs per example, we additionally vary the model dimension $\in \{64, 128, 256\}$. The results are shown in Fig. 6. We validate that as the number of associative recall keys and values exceeds the window size, the model again uses increased dimensionality relative to full $O(N^2)$ attention to solve the task. While Fig. 6 highlights RetNet, we note that it is a special case of the recently released Mamba [Anonymous \[2023a\]](#) and GateLoop [Anonymous \[2023b\]](#) architectures. Corresponding to this experiment, we theoretically show in Appendix H.6 that the gated recurrence uses $\Omega(N)$ bits to solve MQAR for $d \leq \sqrt{N}$. Intuitively, it is difficult to store a large number of key-value pairs in low dimensional hidden states.

Experiment 3: Increasing the range of token-interaction distances per example. To construct the synthetic dataset following Procedure 1, we set $\alpha = 0.1$ and $|C| = 8,192$ for all experiments. We fix $N = 256$ and vary $D \in \{\frac{N}{32}, \frac{N}{16}, \frac{N}{8}, \frac{N}{4}\}$ to increase the number of key-value pairs occurring per example and the range of token-interaction distances per example. For each number of key-value pairs per example, we additionally vary the model dimension $\in \{64, 128, 256\}$.

The results are shown in Fig. 6, validating that sliding window attention closes the gap to attention when the number of KV pairs is within the window size. I.e., 16 KVs means that there are 16 keys and 16 values, for 32 total tokens. Sliding attention with window size 32 performs well up until 16 KVs and degrades beyond this point relative to attention models of the same dimensionality. Sliding window attention with window size 8 does not perform well on any setting, as expected. Blocked attention also not perform well – intuitively tokens at the beginning of a block lack sufficient prior context to attend to.

Model	Param (M)	TFLOPs	Overall		Slices		% gap due to AR Hits
			AR Hits	Other Tokens	AR Hits	AR Hits	
Attention	73	1.52	12.99 (2.56)	2.41 (0.88)	14.76 (2.69)	—	—
Long Conv	76	1.20	20.28 (3.01)	40.25 (3.70)	19.25 (2.96)	44.4%	44.4%
H3	72	1.33	15.78 (2.76)	13.86 (2.63)	15.94 (2.77)	63.2%	63.2%
Hyena	72	1.34	15.13 (2.72)	9.00 (2.20)	15.74 (2.76)	60.8%	60.8%
RWKV	72	1.89	16.10 (2.78)	14.11 (2.65)	16.26 (2.79)	57.9%	57.9%
Attention	125	2.46	11.01 (2.40)	2.16 (0.77)	12.45 (2.52)	—	—
Long Conv	128	1.74	16.98 (2.83)	25.62 (3.24)	16.46 (2.80)	40.1%	40.1%
H3	168	2.55	12.06 (2.49)	6.75 (1.91)	12.60 (2.53)	88.4%	88.4%
RWKV	169	2.08	11.64 (2.45)	5.70 (1.74)	12.29 (2.51)	100.0%	100.0%
RetNet (128)	152	2.60	11.15 (2.41)	3.01 (1.10)	12.45 (2.55)	100%	100%
Hyena	160	2.41	11.60 (2.45)	5.00 (1.61)	12.28 (2.51)	100%	100%
+ Slide Attn. (64)	159	2.28	11.57 (2.45)	5.23 (1.65)	12.30 (2.51)	100%	100%
+ Slide Attn. (256)	159	2.29	10.65 (2.37)	3.42 (1.23)	11.52 (2.45)	100%	100%
Attention	360	6.23	9.44 (2.25)	1.98 (0.69)	10.62 (2.36)	—	—
Long Conv	360	4.08	13.13 (2.57)	13.27 (2.59)	13.12 (2.57)	40.5%	40.5%
H3	357	4.85	10.38 (2.34)	4.81 (1.57)	11.00 (2.40)	65.8%	65.8%
Hyena	358	5.03	10.07 (2.31)	3.83 (1.34)	10.75 (2.38)	98.2%	98.2%
RWKV	351	4.31	9.79 (2.28)	3.82 (1.34)	10.51 (2.35)	100.0%	100.0%

Table 4: **Language modeling validation perplexity on the Pile.** After pretraining on 10B tokens of Pile data, we report log perplexity with negative log-likelihood in parentheses. We report overall scores, and for the AR vs. non-AR token slices defined in Section 3. FLOPs are computed for inputs of 2048 tokens based on the equations in Appendix C.

F.2 Demonstrating Mqar as a Tool for Architecture Development

In the main paper, we provide experiments showing that there is gap between the gated convolution and attention models on synthetic MQAR data and correspondingly in the downstream Pile results. In Table 4, we show that the trends of RetNet and sparse attention on MQAR also carry to the downstream results. We hope MQAR synthetic data may be useful in future architectural development.

G Mqar Perplexity Gap and Model Size

In this section, we investigate how the associative recall gap changes as we increase the model size to a billion parameters and beyond. Overall, the results suggest that the larger models improve in memorizing bigrams from the training data, but do not rapidly improve in in-context learning (perplexity on rare bigrams). Below, we pretrain at the 1.4Bn parameter scale and observe that the 70M parameter attention model is a full perplexity point better on AR than this 1.4Bn Hyena model (Table 1). We discuss these results further in (Section 3).

Model	Param (B)	TFLOPs	Overall		Slices		% of gap due to AR Hits
			AR Hits	Other Tokens	AR Hits	AR Hits	
Attention	1.4	1.52	8.19 (2.10)	1.91 (0.65)	9.86 (2.29)	—	—
Hyena	1.4	1.20	9.65 (2.27)	3.43 (1.23)	11.01 (2.40)	40.3%	40.3%

Table 5: **Large-scale language model validation perplexity on the Pile.** After pretraining on 50B tokens of Pile data, we report log perplexity with negative log-likelihood in parentheses. We report overall scores, and for the AR vs. non-AR token slices defined in Section 3. FLOPs are computed for inputs of 2048 tokens based on the equations in Appendix C.

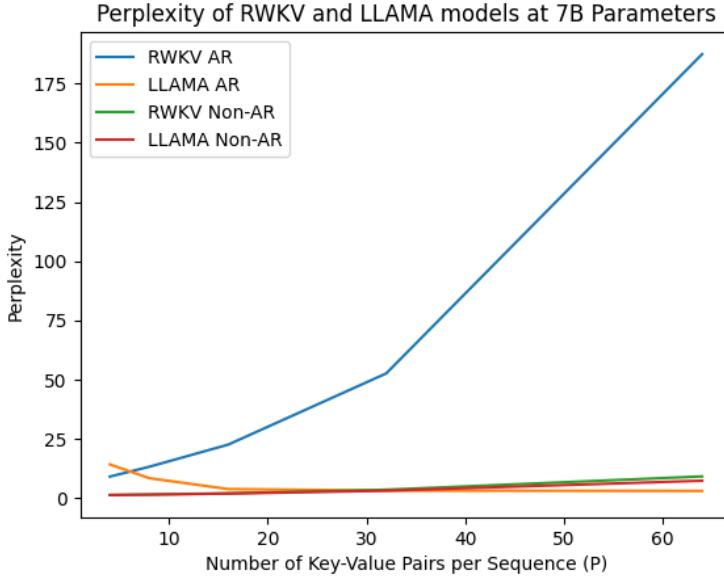


Figure 7: Perplexity of RWKV 7B and Llama2 7B parameter models on synthetic MQAR data as a function of the number of recalls required per input / number of key-value pairs per input (P). Inputs are constructed using each model’s full vocabulary.

Open-source 7B parameter models We next evaluate the RWKV (gated convolution) Peng et al. [2023] and Llama 2 (attention) pretrained models at the 7B parameter scale Touvron et al. [2023].⁹ These are both popular models that took a significant amount of effort to train, towards maximizing quality. We find that there is a gap between RWKV and attention at the 7B scale and that it increases as the model needs to conduct more recalls per sequence (P below).

We summarize the experimental protocol below.

1. **Justifying the experimental protocol.** Since frequent bigrams may just be memorized by the model and not require in-context recall, our work measures AR quality on infrequent bigrams in validation sequences (Figure 1, Appendix C.1). We do not have access to the custom training data mixtures used in training RWKV or Llama 2 to measure bigram frequencies, so we use a synthetic test to fairly measure the AR capabilities.
2. **Evaluation dataset.** We construct the synthetic dataset following Algorithm 1, where each sequence contains P token pairs, or “bigrams”. Each bigram, containing a “key” token and a “value” token, appears twice in the sequence. On the second occurrence of a “key” token, the model should look back to the prior occurrence to output the corresponding “value” token. We measure the AR perplexity of the model based on its ability to predict the correct “value” token as the next word for these repeated keys. The sequences are constructed using the models’ vocabulary tokens.
3. **Evaluation details.** We evaluate (inference only, no training) on sequence lengths of 1024 tokens. We measure the AR perplexity when the sequences contain $P \in \{16, 32, 64, 128, 256\}$ key-value pairs, using 1000 samples per P value. The tokens that do not contain a key or value are simply filled with a fixed token id (so this token is repeated frequently in the sequence). We plot perplexity for AR and non-AR tokens (fixed token) vs. P . We find RWKV quality degrades with P on the AR slice (blue line), while all other lines remain flat. MQAR remains problematic at scale.

Overall our results suggest that simply scaling the model size may not close the MQAR quality gap between the gated convolutions and attention.

Measuring AR Gap with Scale In the main paper, we measure the AR perplexity on downstream data based on bigrams that are seen $< 1,250\times$ during pretraining. However, we hypothesize that larger models

⁹We specifically evaluate the RWKV-Raven model downloaded from https://huggingface.co/docs/transformers/model_doc/rwkv and the Llama 2 model downloaded from <https://github.com/facebookresearch/llama>.

may memorize a larger number of bigram pairs seen in the training dataset, but do not rapidly gain the capability to perform associative recall as well as attention in-context. In-context learning is defined as learning from examples *provided in context* [Xie et al., 2021].

Concretely, the gap between the Hyena / RWKV and attention models at the 350m scale is 1.85 / 1.84 perplexity points when we focus on bigrams seen $< 1,250\times$ during pretraining (Table 1). If we instead focus on bigrams seen $1\times$ during pretraining, the gap to attention quality is 12.0 / 13.2 perplexity points respectively. The gated convolutions appear to struggle in the regime of rare bigrams that require the model to use the context.

H Details on Theoretical Analysis

This section provides proofs and extensions to the theoretical results in the main paper.

H.1 Preliminaries and Notation

H.1.1 Notation

We denote the all 1 row vector of size k , given by $[1 \ 1 \ \dots \ 1 \ 1]$, and the all 0 row vector of size k , given by $[0 \ 0 \ \dots \ 0 \ 0]$, as $\mathbf{1}^k$ and $\mathbf{0}^k$, respectively. We also construe the standard basis vector \mathbf{e}_i as a column vector in these notes, and adhere to the following matrix indexing convention: $\mathbf{M}[i, j]$ is the entry in the i th row and the j th column, $\mathbf{M}[i, :] \in \mathbb{F}^{1 \times n}$ denotes the i th row, and $\mathbf{M}[:, j] \in \mathbb{F}^{m \times 1}$ denotes the j th column of $\mathbf{M} \in \mathbb{F}^{m \times n}$. We then use $\mathbf{1}^{m \times n}, \mathbf{0}^{m \times n} \in \mathbb{F}^{m \times 1}$ to denote the matrix of all 1s and 0s, respectively.

Next, we denote the *Hadamard product* of vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$ as $\mathbf{u} \odot \mathbf{v}$; the operation can be extended to matrices by applying the Hadamard product column-wise across the matrices. This is commonly referred to as *(element-wise) gating*. For vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$, we also denote their *linear (or acyclic) convolution* as $\mathbf{u} * \mathbf{v}$ and *cyclic convolution* as $\mathbf{u} \circledast \mathbf{v}$.

Polynomial Notation. Because convolution is intimately tied to operations on polynomials, it is convenient to use them to discuss the inputs and outputs of gated convolution models. Let us define maps $\text{poly}, \text{poly}^* : \mathbb{F}^n \rightarrow \mathbb{F}[X]/(X^n)$ such that

$$\text{poly}(\mathbf{u}) = \sum_{i=0}^{n-1} \mathbf{u}[i]X^i, \text{ and } \text{poly}^*(\mathbf{u}) = \sum_{i=0}^{n-1} \mathbf{u}[i]X^{n-1-i}.$$

This allows us to map between vectors and polynomial. Accordingly, we also define $\text{coeff} : \mathbb{F}[X]/(X^{n+1}) \rightarrow \mathbb{F}^n$ as the map converting polynomials back to vectors: $\text{coeff}(\mathbf{u}(X)) = \mathbf{u}$ with $\mathbf{u}[i]$ defined as the coefficient in $\mathbf{u}(X)$ at degree i .

These operations allow us to interpret the convolution of vectors in terms of polynomial multiplication [Heideman and Burrus, 1988]. More specifically, we have

$$\begin{aligned} \mathbf{u} * \mathbf{v} &= \text{coeff}(\mathbf{u}(X) \cdot \mathbf{v}(X) \bmod X^n), \text{ and} \\ \mathbf{u} \circledast \mathbf{v} &= \text{coeff}(\mathbf{u}(X) \cdot \mathbf{v}(X) \bmod X^n - 1). \end{aligned}$$

We can similarly interpret the Hadamard product of vectors $\mathbf{u} \odot \mathbf{v}$ as the Hadamard product of polynomials $\mathbf{u}(X) \odot \mathbf{v}(X)$:

$$\mathbf{u} \odot \mathbf{v} = \text{coeff}(\mathbf{u}(X) \odot \mathbf{v}(X)) = \text{coeff}\left(\sum_{i=0}^{n-1} (\mathbf{u}[i] \cdot \mathbf{v}[i]) \cdot X^i\right).$$

Arithmetic Circuit Notation. We briefly introduce the notation of arithmetic circuits [Volkovich, 2016], the focus of Appendix H.5. An *arithmetic circuit* \mathcal{C} with variables $X \triangleq \{x_1, x_2, \dots, x_n\}$ over a field \mathbb{F} is interpreted as a directed acyclic graph, where the input nodes are labelled by either the variables from X or constants from \mathbb{F} and the internal nodes are labelled by $+$ or \times with the output being the polynomial computed at the output node.

We shall also refer to the *size* of the circuit as the number of nodes, the *depth* of the circuit as the length of the longest path between an input node and the output node, and the *width* of the circuit as the number of parallel operations in the circuit, or ‘wires’ which will be intersected by a horizontal ‘cut’ through the circuit. Moreover, the *degree* of a circuit is defined as the degree of the polynomial computed by the circuit. We summarize this with the following definition:

Definition H.1. An arithmetic circuit \mathcal{C} is an (n, s, Δ, w) -*circuit* if \mathcal{C} is an n -variate arithmetic circuit of size s and of depth at most Δ , and width w .

Model Notation. Now we introduce the notation we will be using for defining layers. In what follows, we denote $\mathbf{u} \in \mathbb{R}^{N \times d}$ as the model input; N as the sequence length; L as the number of stacked layers, indexed by ℓ ; and d as the input (embedding) dimension.

H.1.2 Summary of the Results

The outline of our results are as follows: In Appendix H.2 we introduce *gated convolution models* and define H3, Hyena, RWKV, RetNet, and BASECONV. In Appendix H.3 we introduce a set of primitive operations that BASECONV can implement. We use these as tools in the subsequent proofs. Then, in Appendix H.5, we show that a general arithmetic circuit of size s and degree at most Δ can be simulated by BASECONV. We use the above results to show all models built from gating and convolutions can be simulated by BASECONV. This helps us analyze the *class* of gated convolutions, beyond a specific proposal (e.g. Hyena).

Next, we study the representational power that gated convolutions and attention use to solve MQAR. In Appendix H.7, we derive a BASECONV model inspired from dyadic intervals to show the dimensionality for gated convolutions (with input-independent filters) solve MQAR. Next, we analyze the dimensionality for a BASECONV model with data-dependent kernels to solve MQAR in Appendix H.8.

H.2 Gated Attention-Free Models

We now present formal definitions of gated convolution and recurrence models with respect to the 5-tuple of parameters (N, L, d, N', d') .

Definition H.2. An (N, L, d, N', d') – Gated Convolution Model is a stacked sequence to sequence model with L layers such that:

1. Input and output are $N \times d$ matrices,
2. Each layer’s operations consist of element-wise gating, convolution, linear projection, and
3. All the individual gated convolution layers take in $N' \times d'$ matrices and output $N' \times d'$ matrices. We refer to the tuple (N', d') as the *inner dimension* of the model.

We define the Hyena, RWKV, RetNet, and BASECONV layers to make step 2 more concrete. We also assume that the input $\mathbf{u} \in \mathbb{R}^{N \times d}$ is embedded into $\mathbf{u}' \in \mathbb{R}^{N' \times d'}$ such that

$$\mathbf{u}'[n, t] = \begin{cases} \mathbf{u}[n, t] & \text{if } n < N, t < d \\ 0 & \text{otherwise.} \end{cases}$$

The output from the last layer $\mathbf{z} \in \mathbb{R}^{N' \times d'}$ is transformed into output $\mathbf{y} \in \mathbb{R}^{N \times d}$ by extracting the top left $N \times d$ entries in \mathbf{z} .

Next, we define the class of weight matrices that we will use in the linear projections in the models:

Definition H.3. The linear projection $\text{Linear}_{m,m}$ has its matrix representation as the weight matrix¹⁰ $\mathbf{W} \in \mathbb{R}^{m \times m}$ taken to be a **K-matrix** for $\mathbf{W} \in (\mathcal{B}\mathcal{B}^*)^{\text{poly-log } m}_{\text{poly-log } m}$ (Definition H.17). Consequently, each matrix \mathbf{W} has $\tilde{\mathcal{O}}(m)$ parameters and runtime for matrix vector multiplication, and allows us to represent general linear transformations with low-depth linear arithmetic circuits [Dao et al., 2020]. We will also need linear maps $\text{Linear}_{m,n}$ for $m < n$, where each take the corresponding square matrices from $\text{Linear}_{m,m}$ and note that such matrices has $\tilde{\mathcal{O}}(n)$ parameters and runtime for matrix vector multiplication.

Remark H.4. We note that the weight matrix $\mathbf{W} \in \mathbb{R}^{d \times d}$ above is taken to be a dense matrix in the experiments. However, for our theoretical results, we restrict the linear projection in our models as per

¹⁰I.e. $\text{Linear}_{m,m}(\mathbf{z}) = \mathbf{W} \cdot \mathbf{z}$.

Definition H.3. In the rest of the paper, unless mentioned otherwise all linear projections will follow Definition H.3.

Next, we define three popular gated convolution models that we study in our work: Hyena Poli et al. [2023a], RWKV Peng et al. [2023], and RetNet Sun et al. [2023].

H.2.1 The Hyena Layer

We will now outline the Hyena layer [Poli et al., 2023b]. Hyena takes a sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$ as input and produces $L + 1$ projections p^1, \dots, p^L, v by passing \mathbf{y} through a linear layer and applying a short convolution afterwards. The algorithm then recursively performs a point-wise multiplication of the projection with the convolution of the filter h^l with the previous output. We summarize this process in Algorithm 3.

Algorithm 2 Projection (\mathbf{u}, \mathbf{h})

Input: Input sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$, a short convolution filter $\mathbf{h} \in \mathbb{R}^N$.

- 1: In parallel for $0 \leq n < N$: $\hat{\mathbf{z}}[n, :] \leftarrow \text{Linear}_{d, (L+1)d}(\mathbf{u}[n, :])$ so that $\hat{\mathbf{z}} \in \mathbb{R}^{N \times (L+1)d}$
- 2: In parallel for $0 \leq t < (L+1)d$: $\mathbf{z}[:, t] \leftarrow \mathbf{h} * \hat{\mathbf{z}}[:, t]$
- 3: Reshape and split $\mathbf{z} \in \mathbb{R}^{N \times (L+1)d}$ into $\mathbf{p}^1, \dots, \mathbf{p}^L, \mathbf{v}$, where $\mathbf{p}^\ell, \mathbf{v} \in \mathbb{R}^{N \times d}$ for $\ell \in [L]$.
- 4: **return** $\mathbf{p}^1, \dots, \mathbf{p}^L, \mathbf{v}$.

Algorithm 3 Hyena ($\mathbf{u}, \mathbf{h}, \mathbf{h}_s$)

Input: Input sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$, set of convolution filters $\mathbf{h}^1, \dots, \mathbf{h}^L \in \mathbb{R}^{N \times d}$, short convolution filter $\mathbf{h}_s \in \mathbb{R}^N$.

- 1: $\mathbf{p}^1, \dots, \mathbf{p}^L, \mathbf{v} \leftarrow \text{Projection}(\mathbf{u}, \mathbf{h}_s)$.
- 2: $\mathbf{z}^0 \leftarrow \mathbf{v}$
- 3: **for** $\ell = 1, \dots, L$ **do**
- 4: In parallel for $0 \leq t < d$: $\mathbf{z}^\ell[:, t] \leftarrow \mathbf{p}^\ell[t, :] \odot (\mathbf{h}^\ell[:, t] * \mathbf{z}^{\ell-1}[:, t])$.
- 5: **return** \mathbf{z}^L

Remark H.5. In Algorithm 3, L is used to denote the number of recursive applications of the Hadamard product and convolutions, not the number of layers. Note that asymptotically, the recursive step does not make a difference.

Henceforth, we will refer to a model consisting of L Hyena layers as a gated convolution model with associated tuple $(N, L, d, N, (L+1)d)$ as $(N, L, d, N, (L+1)d)$ – Hyena.

H.2.2 The RWKV Layer

We will now describe the RWKV layer [Peng et al., 2023]. RWKV is typically referred to as an RNN, but, like some other recurrent models (e.g. S4 [Gu et al., 2021]), it can also be viewed as a convolution. Here, we present the convolutional view of RWKV. We will see that it is closely related to the Hyena layer.

RWKV takes a sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$ as input and applies a short convolution. Next, it produces three projections q, k, v by passing \mathbf{u} through a linear layer. Then, it performs a convolution sandwiched by element-wise multiplication. We summarize this process in Algorithm 5.

Algorithm 4 RWKVProjection (\mathbf{u}, \mathbf{h})

Input: Input sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$, a short convolution filter $\mathbf{h} \in \mathbb{R}^N$.

- 1: In parallel for $0 \leq t < d$: $\hat{\mathbf{z}}[:, t] \leftarrow \mathbf{h} * \mathbf{u}[:, t]$
- 2: In parallel for $0 \leq n < N$: $\mathbf{z}[n, :] \leftarrow \text{Linear}_{d, 3d}(\hat{\mathbf{z}}[n, :])$ so that $\mathbf{z} \in \mathbb{R}^{N \times 3d}$
- 3: Reshape and split $\mathbf{z} \in \mathbb{R}^{N \times 3d}$ into $\mathbf{q}, \mathbf{k}, \mathbf{v} \in \mathbb{R}^{N \times d}$.
- 4: **return** $\mathbf{q}, \mathbf{k}, \mathbf{v}$.

In practice, the short convolution filter \mathbf{h}_s is restricted to a length two filter with $\mathbf{h}_s[0] = \mu$ and $\mathbf{h}_s[1] = 1 - \mu$, where $\mu \in [0, 1]$ is learned parameter. In the RWKV paper, the short convolution is referred to as a “time shift”.

Algorithm 5 RWKV ($\mathbf{u}, \mathbf{h}, \mathbf{h}_s$)

Input: Input sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$, set of convolution filters $\mathbf{h}^1, \dots, \mathbf{h}^L \in \mathbb{R}^{N \times d}$, short convolution filter $\mathbf{h}_s \in \mathbb{R}^N$.

- 1: $\mathbf{q}, \mathbf{k}, \mathbf{v} \leftarrow \text{Projection}(\mathbf{u}, \mathbf{h}_s)$.
- 2: In parallel for $0 \leq t < d$: $\mathbf{z}^\ell[:, t] \leftarrow \sigma(\mathbf{q}[:, t]) \odot (\mathbf{h}^\ell[:, t] * (\text{softmax}(\mathbf{k}) \odot \mathbf{v})[:, t])$.
- 3: **return** \mathbf{z}^L

In practice, the long convolution filter \mathbf{h} is also restricted to $\mathbf{h}[:, t] = e^{w(t-1)}$, where $w \in \mathbb{R}$ is a learnable parameter that controls how quickly the magnitudes of the filter decreases as t grows.

To summarize, the differences between Hyena and RWKV are: (1) Hyena applies the short convolution *after* the linear projection whereas RWKV applies it *before*, (2) RWKV includes non-linear activations (sigmoid and softmax) while Hyena does not, (3) RWKV and Hyena use different parameterizations for the convolutional filters, and (4) Hyena recursively performs a point-wise multiplication of the projections with the convolution filter whereas RWKV performs this operation only once (though, in practice, Hyena uses a recurrence depth of one, making them equivalent in this regard).

H.2.3 The Retnet Layer

In this section, we introduce the RetNet layer [Sun et al. \[2023\]](#). To this end, we take an input sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$ and project it using learnable weight matrices. We then compute the states \mathbf{z}^n recurrently as in line 5 and the output $\text{Out}[n, :]$ as in line 6 for each $n \in [N]$ (see Algorithm 6 below).

Algorithm 6 RetNet ($\mathbf{u}, \mathbf{h}, \mathbf{h}_s$)

Input: Input sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$, a scalar $\gamma \in \mathbb{R}$, learnable weight matrices $\mathbf{W}_A, \mathbf{W}_C, \mathbf{W}_V \in \mathbb{R}^{d \times d}$.

- 1: $\mathbf{A}, \mathbf{C}, \mathbf{V} \leftarrow \mathbf{u} \mathbf{W}_A, \mathbf{u} \mathbf{W}_C, \mathbf{u} \mathbf{W}_V$ so that $\mathbf{A}, \mathbf{C}, \mathbf{V} \in \mathbb{R}^{N \times d}$.
- 2: Initialize the output $\text{Out} \in \mathbb{R}^{N \times d}$
- 3: Initialize the state $\mathbf{z}^0 \leftarrow (\mathbf{A}[0, :])^\top \mathbf{V}[0, :]$.
- 4: **for** $1 \leq n < N$ **do**
- 5: $\mathbf{z}^n \leftarrow \gamma \mathbf{z}^{n-1} + (\mathbf{A}[n, :])^\top \mathbf{V}[n, :]$ for $\mathbf{z}^n \in \mathbb{R}^{d \times d}$
- 6: $\text{Out}[n, :] \leftarrow \mathbf{C}[n, :] \mathbf{z}^n$
- 7: **return** Out

H.2.4 BaseConv

Finally, we introduce the BASECONV here as follows:

$$\mathbf{Y} = (\mathbf{u} \mathbf{W} + \mathbf{b}_1) \odot (\mathbf{u} * \mathbf{h} + \mathbf{b}_2), \quad (4)$$

with input $\mathbf{u} \in \mathbb{R}^{N' \times d'}$, weight matrix $\mathbf{W} \in \mathbb{R}^{d' \times d'}$ and bias matrices $\mathbf{b}_i \in \mathbb{R}^{N' \times d'}$ defining linear projections of the input sequence, and $\mathbf{h} \in \mathbb{R}^{N' \times d'}$ is the a set of the d' mixed length filters.

The corresponding pseudocode for BASECONV is as follows:

Algorithm 7 BASECONV($\mathbf{u}, \mathbf{W}, \mathbf{b}_1, \mathbf{h}, \mathbf{b}_2$)

Input: Input sequence $\mathbf{u} \in \mathbb{R}^{N' \times d'}$, linear mapping $\mathbf{W} \in \mathbb{R}^{d' \times d'}$ so that $\mathbf{W} \in (\mathcal{B}\mathcal{B}^*)^{\text{poly log } d'}_{\text{poly log } d'}$, convolution filter $\mathbf{h} \in \mathbb{R}^{N' \times d'}$, bias matrices $\mathbf{b}_1, \mathbf{b}_2 \in \mathbb{R}^{N' \times d'}$.

- 1: In parallel for $0 \leq n < N' : \mathbf{x}[n, :] = \text{Linear}_{d', d'}(\mathbf{u}[n, :])$
- 2: In parallel for $0 \leq t < d' : \mathbf{z}[:, t] = \mathbf{h}[:, t] * \mathbf{u}[:, t]$
- 3: In parallel for $0 \leq t < d' : \mathbf{y}[:, t] \leftarrow (\mathbf{x}[:, t] + \mathbf{b}_1[:, t]) \odot (\mathbf{z}[:, t] + \mathbf{b}_2[:, t]).$ ▷ See equation 4
- 4: **return** \mathbf{y}

Note that the convolution in Algorithm 7 is not limited to causal convolution and allows circular convolution as well. For simplicity, we use $*$ here and will disambiguate when required. We will start by specifying the parameter count and complexity of a single layer of BASECONV below.

Proposition H.6. *A single layer of BASECONV requires $\tilde{O}(N'd')$ parameters and runtime.*

Proof. We know the parameters and runtime for the linear part of BASECONV via Definition H.3 is $\tilde{O}(N'd')$. Further, each convolution operation requires $\mathcal{O}(N')$ parameters and $\mathcal{O}(N' \log N')$ runtime, and we employ d' such operations. Finally the Hadamard product step takes $O(nd)$ time. \square

Similar to Hyena, we will refer to a model consisting of L BASECONV layers as $(N, L, d, N', d') - \text{BASECONV}$. In our experiments, we extend BASECONV by adding an MLP after Algorithm 7. For simplicity we will denote BASECONV($\mathbf{u}, \mathbf{h}, \mathbf{b}_1, \mathbf{b}_2$) as BASECONV(\mathbf{u}, \mathbf{h}) when $\mathbf{b}_1 = \mathbf{b}_2 = \mathbf{0}$.

We will now show that there exists a BASECONV model that can emulate each of the basic operations in Algorithm 7.

Lemma H.7. *The functions $\text{Linear}_{d, d'}(\mathbf{u})$ (Definition H.3), with d, d' defined as in Algorithm 7, convolution with filter $\mathbf{h} \in \mathbb{R}^{N' \times d}$, and element-wise gating can be computed with Algorithm 7 via a $(N, 1, d, N, d') - \text{BASECONV}$.*

Proof. For each operation from Definition H.2 and Algorithm 7:

1. For any input $\mathbf{u} \in \mathbb{R}^{N' \times d'}$, $\text{Linear}_{d, d'}(\mathbf{u})$ with matrix representation $\mathbf{W} \in \mathbb{R}^{N' \times d'}$ can be performed by a single BASECONV layer computing BASECONV($\mathbf{y}, \mathbf{W}, \mathbf{h}, \mathbf{b}_1, \mathbf{b}_2$) with \mathbf{b}_1 and \mathbf{b}_2 being the matrix of all 0s and all 1s, respectively while the convolution with the zero filter. That is, we have

$$\mathbf{Y} = (\mathbf{u}\mathbf{W} + \mathbf{0}^{N' \times d'}) \odot (\mathbf{u} * \mathbf{0}^{N' \times d'} + \mathbf{1}^{N' \times d'}) = (\mathbf{u}\mathbf{W}) \odot \mathbf{1}^{N' \times d'} = \mathbf{u}\mathbf{W} = \text{Linear}_{d, d'}(\mathbf{u}).$$

2. For any input $\mathbf{u} \in \mathbb{R}^{N \times d}$, convolution with filter $\mathbf{h} \in \mathbb{R}^{N \times d}$ can be performed by a single BASECONV layer computing BASECONV($\mathbf{y}, \mathbf{W}, \mathbf{h}, \mathbf{b}_1, \mathbf{b}_2$) where \mathbf{W}, \mathbf{b}_2 are all zeroes, and \mathbf{b}_1 is the matrix of all 1s so that we get

$$\mathbf{Y} = (\mathbf{u}\mathbf{0}^{N' \times d'} + \mathbf{1}^{N' \times d'}) \odot (\mathbf{u} * \mathbf{h} + \mathbf{0}^{N' \times d'}) = \mathbf{1}^{N' \times d'} \odot (\mathbf{u} * \mathbf{h}) = \mathbf{u} * \mathbf{h}.$$

3. We may compute element-wise gating between matrices $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{N \times d}$, where \mathbf{v} is some fixed factor, with a single layer computing BASECONV($\mathbf{y}, \mathbf{0}^{N' \times d'}, \mathbf{e}_0, \mathbf{v}, \mathbf{0}^{N' \times d'}$ where \mathbf{e}_1 is the identity filter, respectively, by Definition H.2.

$$\mathbf{Y} = (\mathbf{u}\mathbf{0}^{N' \times d'} + \mathbf{v}) \odot (\mathbf{u} * \mathbf{e}_0 + \mathbf{0}^{N' \times d'}) = \mathbf{v} \odot \mathbf{u}.$$

\square

H.3 Primitives

In this section, we will establish some additional basic primitives that we expect a gated convolution model to emulate: `shift`, `remember` and `add`. We specify them below:

1. **Shift** an sequential input of length N up or down by s entries:

`shift_up(\mathbf{y}, s)`, `shift_down(\mathbf{y}, s)`

- **Input:** $\mathbf{y} \in \mathbb{R}^{N \times d}$, $s \geq 0$
- **Output:** $\mathbf{z} \in \mathbb{R}^{N \times d}$ where $\mathbf{z}^+ = \text{shift_down}(\mathbf{y}, s)$ and $\mathbf{z}^- = \text{shift_up}(\mathbf{y}, s)$

$$\mathbf{y} \equiv \begin{pmatrix} \leftarrow \mathbf{y}_0 \rightarrow \\ \vdots \\ \leftarrow \mathbf{y}_{i-1} \rightarrow \\ \leftarrow \mathbf{y}_i \rightarrow \\ \vdots \\ \leftarrow \mathbf{y}_{N-1} \rightarrow \end{pmatrix} \quad \mathbf{z}^+ \equiv \begin{pmatrix} \leftarrow \mathbf{0} \rightarrow \\ \vdots \\ \leftarrow \mathbf{0} \rightarrow \\ \leftarrow \mathbf{y}_0 \rightarrow \\ \vdots \\ \leftarrow \mathbf{y}_{N-1-s} \rightarrow \end{pmatrix} \quad \mathbf{z}^- \equiv \begin{pmatrix} \leftarrow \mathbf{y}_s \rightarrow \\ \vdots \\ \leftarrow \mathbf{y}_{N-1} \rightarrow \\ \leftarrow \mathbf{0} \rightarrow \\ \vdots \\ \leftarrow \mathbf{0} \rightarrow \end{pmatrix}$$

2. **Add** a sequence $\mathbf{x} \in \mathbb{R}^{n \times d}$ to a running sum $\mathbf{S} \in \mathbb{R}^{n \times d}$ for some $2n \leq N'$ with both \mathbf{x} and \mathbf{S} contained as subvectors in $\mathbf{y} \in \mathbb{R}^{N \times d}$.

`addn($\mathbf{y} : \mathbf{x}, \mathbf{S}$):`

- **Input:** sequence \mathbf{y} containing $\mathbf{x}, \mathbf{S} \in \mathbb{R}^{n \times d}$ for $2n \leq N$ such that $\mathbf{y}[0 : n-1] \equiv \mathbf{x}, \mathbf{y}[n : 2n-1] \equiv \mathbf{S}$ and $\mathbf{y}[2n : N-1] \equiv \mathbf{0}^{N-2n}$.
- **Output:** $\mathbf{z} \in \mathbb{R}^{N \times d}$ containing the sum $\mathbf{y} + \mathbf{S}$ such that $\mathbf{y}[0 : n-1] \equiv \mathbf{1}^n, \mathbf{y}[n : 2n-1] \equiv \mathbf{S} + \mathbf{x}$ and $\mathbf{z}[2n : N-1] \equiv \mathbf{0}^{N-2n}$.

$$\mathbf{y} \equiv \begin{pmatrix} \leftarrow \mathbf{x} \rightarrow \\ \leftarrow \mathbf{S} \rightarrow \\ \leftarrow \mathbf{0} \rightarrow \\ \vdots \\ \leftarrow \mathbf{0} \rightarrow \end{pmatrix} \quad \mathbf{z} \equiv \begin{pmatrix} \leftarrow \mathbf{1}^n \rightarrow \\ \leftarrow \mathbf{S} + \mathbf{x} \rightarrow \\ \leftarrow \mathbf{0} \rightarrow \\ \vdots \\ \leftarrow \mathbf{0} \rightarrow \end{pmatrix}$$

3. **Remember** $\mathbf{v} \in \mathbb{R}^{m \times d}$ as part of a sequence of input $\mathbf{y} \in \mathbb{R}^{N \times d}$ while performing gated convolution *only* on $\mathbf{x} \in \mathbb{R}^{n \times d}$ for some $m, n \leq N \times d$.

$\text{remember}_{n,m,s,t}(\mathbf{y} : \mathbf{x}, \mathbf{v}, \mathbf{h}, \mathbf{p})$:

- **Input:** sequence $\mathbf{y} \in \mathbb{R}^{N \times d}$ containing $\mathbf{x} \in \mathbb{R}^{n \times d}$, $\mathbf{v} \in \mathbb{R}^{m \times d}$, and modifiers $\mathbf{p}, \mathbf{h} \in \mathbb{R}^{n \times d}$ such that $\mathbf{y}[0 : n-1] \equiv \mathbf{x}$, $\mathbf{y}[n+s : n+s+m-1] \equiv \mathbf{v}$ and $\mathbf{y}[i] = 0$ otherwise with $\mathbf{x} * \mathbf{h} \in \mathbb{R}^{(n+s) \times d}$ and $\mathbf{v} * \mathbf{h} \in \mathbb{R}^{(m+t) \times d}$.
- **Output:** $\mathbf{z} \in \mathbb{R}^{N \times d}$ containing $(\mathbf{p} \odot (\mathbf{x} * \mathbf{h})) \in \mathbb{R}^{(n+s) \times d}$, such that:

$$\mathbf{y} \equiv \begin{pmatrix} \leftarrow \mathbf{x} \rightarrow \\ \hline \mathbf{0}^s \\ \hline \leftarrow \mathbf{v} \rightarrow \\ \hline \vdots \\ \hline \mathbf{0} \end{pmatrix} \quad \mathbf{z} \equiv \begin{pmatrix} \leftarrow \mathbf{p} \odot (\mathbf{x} * \mathbf{h}) \rightarrow \\ \hline \leftarrow \mathbf{v} \rightarrow \\ \hline \vdots \\ \hline \mathbf{0} \end{pmatrix}$$

These primitives are building blocks of our proofs in the sequel. We will show that each of these primitives can be solved by some $(N, L, d, N', d') - \text{BASECONV}$ model with a small constant L .

Proposition H.8 (The Shift Primitive). *For any $\mathbf{y} \in \mathbb{R}^{N \times d}$, there exist $(N, 1, d, N, d) - \text{BASECONV}$ and $(N, 3, d, N, d) - \text{BASECONV}$ that computes $\text{shift_down}(\mathbf{y}, s)$ and $\text{shift_up}(\mathbf{y}, s)$ for any $s \leq N$.*

Proof. Define the following kernel dependent on $s \leq N$

$$\mathbf{h}_s[n, :] \equiv \begin{cases} \mathbf{1}^d & \text{if } n = s + 1 \\ \mathbf{0}^d & \text{otherwise.} \end{cases}$$

We now deal with the down and up shifts separately:

1. We define $\mathbf{W} := \mathbf{0}^{N \times d}$, $\mathbf{b}_1 := \mathbf{1}^{N \times d}$, $\mathbf{b}_2 := \mathbf{0}^{N \times d}$. Then, for input $\mathbf{y} \in \mathbb{R}^{N \times d}$, $\text{BASECONV}(\mathbf{y}, \mathbf{0}^{N \times d}, \mathbf{h}_s, \mathbf{1}^{N \times d}, \mathbf{0}^{N \times d})$ for BASECONV in Algorithm 7 is given by equation 4 as

$$\mathbf{Y} \equiv \mathbf{y} * \mathbf{h}_s.$$

Now, to perform $\text{shift_down}(\mathbf{y}, s)$, we note that

$$\begin{aligned} \mathbf{Y}[:, j] &= \mathbf{y}[:, j] * \mathbf{h}_s[:, j] = \text{coeff}(\mathbf{y}[:, j](X) \cdot \mathbf{h}_s[:, j](X)) \\ &= \text{coeff} \left(\left(\sum_{i=0}^{N-1} \mathbf{y}[i, j] \cdot X^i \right) \cdot X^s \pmod{X^N} \right) \\ &= \text{coeff} \left(\sum_{i=0}^{N-1} \mathbf{y}[i, j] \cdot X^{i+s} \pmod{X^N} \right) \\ &= \text{coeff} \left(\sum_{i=s}^{N-1+s} \mathbf{y}[i-s, j] \cdot X^i \pmod{X^N} \right) \\ &= \text{coeff} \left(\sum_{i=s}^{N-1} \mathbf{y}[i-s, j] \cdot X^i \right), \end{aligned}$$

which implies that we exactly get what is specified in the output.

2. We again define $\mathbf{W} := \mathbf{0}^{N \times d}$, $\mathbf{b}_1 := \mathbf{1}^{N \times d}$, $\mathbf{b}_2 := \mathbf{0}^{N \times d}$. Then, for input $\mathbf{y} \in \mathbb{R}^{N \times d}$, $\text{BASECONV}(\mathbf{y}, \mathbf{0}^{N \times d}, \mathbf{e}_0, \mathbf{1}^{N \times d}, \mathbf{0}^{N \times d})$ for BASECONV in Algorithm 7 is given in equation 4 as

$$\mathbf{Y}_0 \equiv \mathbf{y} * \mathbf{e}_0.$$

Now, to perform $\text{shift_up}(\mathbf{y}, s)$, as before, we first apply the circular convolution to reverse the input

$$\mathbf{Y}_0[:, j] = \mathbf{y}[:, j] \circledast \mathbf{e}_0 = \text{coeff} \left(\sum_{i=0}^{N-1} \mathbf{y}[N-1-i, j] \cdot X^i \right),$$

We then apply $\mathbf{Y}_1 \equiv \text{shift_down}(\mathbf{Y}_0, s)$ to get

$$\begin{aligned} \mathbf{Y}_1[:, j] &\equiv \text{coeff} \left(\sum_{i=s}^{N-1} \mathbf{Y}_0[N-1-(i-s), j] \cdot X^i \right), \\ &\equiv \text{coeff} \left(\sum_{i=s}^{N-1} \mathbf{Y}_0[N-1-i+s, j] \cdot X^i \right). \end{aligned}$$

Finally, we apply another circular convolution with the identity filter to replace $N-1-i$ with i to get

$$\mathbf{Y}_2[:, j] = \mathbf{Y}_1[:, j] \circledast \mathbf{e}_0 = \text{coeff} \left(\sum_{i=0}^{N-1} \mathbf{y}[i+s, j] \cdot X^i \right),$$

Here, we note that we can compute both of these primitives in one and three layers, respectively (see Lemma H.11). \square

Now, we present a BASECONV model with two layers that implements the $\text{add}_n(\mathbf{y} : \mathbf{x}, \mathbf{S})$, the purpose of which is to add some window of computation \mathbf{x} to a running sum \mathbf{S} .

Proposition H.9 (The Running Sum Primitive). *For any $\mathbf{x}, \mathbf{S} \in \mathbb{R}^{n \times d}$ contained in some $\mathbf{y} \in \mathbb{R}^{N \times d}$, there exists a $(N, 2, d, N, d) - \text{BASECONV}$ that computes $\text{add}_n(\mathbf{y} : \mathbf{x}, \mathbf{S})$ for BASECONV as in Algorithm 7.*

Proof. We will show this for $d' = 1$ and the general case follows as we will explain at the end. We now specify the two layers that we use

$$\begin{aligned} \mathbf{z}^1 &\equiv \text{BASECONV}(\mathbf{y}, \mathbf{0}^{N \times 1}, \mathbf{h}^1, \mathbf{b}_1^1, \mathbf{0}^{N \times 1}) \equiv \mathbf{b}_1^1 \odot (\mathbf{h}^1 * \mathbf{y}) \\ \mathbf{z} &\equiv \text{BASECONV}(\mathbf{z}^1, \mathbf{0}^{N \times 1}, \mathbf{h}^2, \mathbf{b}_1^2, \mathbf{b}_2^2) \equiv \mathbf{b}_1^2 \odot (\mathbf{h}^2 * \mathbf{y} + \mathbf{b}_2^2), \end{aligned}$$

where we will specify the kernels as we go along. Let us start by defining the kernel and the bias for the first layer as

$$\mathbf{h}^1 \equiv \begin{pmatrix} \mathbf{e}_0 \\ \mathbf{e}_0 \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix}, \quad \mathbf{b}_1 \equiv \begin{pmatrix} \mathbf{0}^n \\ \mathbf{1}^n \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix}.$$

Let us first compute $\mathbf{h}^1 * \mathbf{y}$ as follows:

$$\begin{aligned} \mathbf{h}^1(X) \cdot \mathbf{y}(X) &= (X^n + 1) \cdot (\mathbf{S}(X) \cdot X^n + \mathbf{x}(X)) \\ &= \mathbf{S}(X) \cdot X^{2n} + (\mathbf{S} + \mathbf{x})(X) \cdot X^n + \mathbf{x}(X). \end{aligned}$$

We then have

$$z_1 \equiv b_1^1 \odot (h^1 * y) \equiv \begin{pmatrix} 0^n \\ 1^n \\ 0^n \\ \dots \\ 0^n \end{pmatrix} \odot \begin{pmatrix} x \\ S + x \\ S \\ \dots \\ 0^n \end{pmatrix} \equiv \begin{pmatrix} 0^n \\ S + x \\ 0^n \\ \dots \\ 0^n \end{pmatrix}$$

Resetting for Next Phase. We now use the next layer to reset for the next phase. Here, we need the first vector to be 1^n in order to start adding the next vector. We thus use the kernel and the biases h^2, b_1^2, b_2^2 defined as

$$h^2 \equiv \begin{pmatrix} e_0 \\ 0^n \\ 0^n \\ \dots \\ 0^n \end{pmatrix}, \quad b_1^2 \equiv \begin{pmatrix} 1^n \\ 1^n \\ 0^n \\ \dots \\ 0^n \end{pmatrix}, \quad b_2^2 \equiv \begin{pmatrix} 1^n \\ 0^n \\ 0^n \\ \dots \\ 0^n \end{pmatrix}.$$

Explicitly, for the second layer, we compute the result of the convolution in terms of polynomials as follows:

$$h^2(X) \cdot z^1(X) = 1 \cdot (S + x)(X) \cdot X^n = (S + x)(X) \cdot X^n.$$

Thus, the output for the second layer is given by

$$z \equiv b_1^2 \odot (h^2 * z^1 + b_2^2) \equiv \begin{pmatrix} 1^n \\ 1^n \\ 0^n \\ \dots \\ 0^n \end{pmatrix} \odot \left(\begin{pmatrix} 0^n \\ S + x \\ 0^n \\ \dots \\ 0^n \end{pmatrix} + \begin{pmatrix} 1^n \\ 0^n \\ \dots \\ 0^n \end{pmatrix} \right) \equiv \begin{pmatrix} 1^n \\ 0^n \\ \dots \\ 0^n \end{pmatrix} \odot \begin{pmatrix} 1^n \\ S + x \\ 0^n \\ \dots \\ 0^n \end{pmatrix} \equiv \begin{pmatrix} 1^n \\ S + x \\ 0^n \\ \dots \\ 0^n \end{pmatrix}.$$

Therefore, we have used two BASECONV layers to add x to the running sum S and reset for the next phase. Here, we note that the only operations we perform are convolutions and Hadamard product and they generalize in the obvious way to $d > 1$. \square

Next, we show that a five layer BASECONV model can perform gated convolution on windows of the input (without changing the rest of the input).

Proposition H.10 (The Remembering Primitive). *For any $x \in \mathbb{R}^{n \times d}, v \in \mathbb{R}^{m \times d}$ contained in some $y \in \mathbb{R}^{N \times d}$ for some $n + m + s + t \leq N$ so that for $h \in \mathbb{R}^{n \times d}$ and $p \in \mathbb{R}^{(n+s) \times d}$ with $x * h \in \mathbb{R}^{(n+s) \times d}$ and $v * h \in \mathbb{R}^{(m+t) \times d}$, there exists a $(N, 5, d, N, d)$ – BASECONV that computes $\text{remember}(y : x, v, h, p)$ for BASECONV as in Algorithm 7.*

Proof. We will again show this for $d' = 1$ and the general case should follow. We now specify the first two layers that we use

$$\begin{aligned} z^1 &\equiv \text{BASECONV}(y, \mathbf{0}^{N \times 1}, h^1, b_1^1, \mathbf{0}^{N \times d}) \equiv b_1^1 \odot (h^1 * y) \\ z^2 &\equiv \text{BASECONV}(z^1, \mathbf{0}^{N \times 1}, h^2, b_1^2, \mathbf{0}^{N \times d}) \equiv b_1^2 \odot (h^2 * y), \end{aligned}$$

The kernel \mathbf{h}^1 and the bias \mathbf{b}_1^1 for the first layer are then given by

$$\mathbf{h}^1 \equiv \begin{pmatrix} \mathbf{h} \\ \mathbf{0}^m \\ \vdots \\ \mathbf{e}_{s+t} \\ \mathbf{0}^n \\ \mathbf{0}^n \\ \dots \\ \mathbf{0}^n \end{pmatrix}, \quad \mathbf{b}_1^1 \equiv \begin{pmatrix} \mathbf{p} \\ \mathbf{0}^{m+t} \\ \vdots \\ \mathbf{0}^n \\ \mathbf{0}^s \\ \mathbf{1}^m \\ \dots \\ \mathbf{0}^n \end{pmatrix}.$$

where recall that $\mathbf{x} * \mathbf{h} \in \mathbb{R}^{(n+s) \times d}$ and $\mathbf{v} * \mathbf{h} \in \mathbb{R}^{(m+t) \times d}$.

We now want to first specify the result of applying the first kernel:

$$\begin{aligned} (\mathbf{h}^1 * \mathbf{y}) &= \text{coeff}((\mathbf{h}(X) + X^{n+m+s+t}) \cdot (\mathbf{v}(X) \cdot X^{n+s} + \mathbf{x}(X))) \\ &= \text{coeff}(\mathbf{h} * \mathbf{v}(X) \cdot X^{n+s} + \mathbf{h} * \mathbf{x}(X) + \mathbf{v}(X) \cdot X^{2n+2s+m+t} + \mathbf{x}(X) \cdot X^{n+m+s+t}) \end{aligned}$$

We then have

$$z_1 \equiv \mathbf{b}_1^1 \odot (\mathbf{h}^1 * \mathbf{y}) \equiv \begin{pmatrix} \mathbf{p} \\ \mathbf{0}^{m+t} \\ \vdots \\ \mathbf{0}^n \\ \mathbf{0}^s \\ \mathbf{1}^m \\ \dots \\ \mathbf{0}^n \end{pmatrix} \odot \begin{pmatrix} \mathbf{h} * \mathbf{x} \\ \mathbf{h} * \mathbf{v} \\ \vdots \\ \mathbf{x} \\ \mathbf{0}^s \\ \mathbf{v} \\ \dots \\ \mathbf{0}^n \end{pmatrix} \equiv \begin{pmatrix} \mathbf{p} \odot (\mathbf{h} * \mathbf{x}) \\ \mathbf{0}^{m+t} \\ \vdots \\ \mathbf{0}^n \\ \mathbf{0}^s \\ \mathbf{v} \\ \dots \\ \mathbf{0}^n \end{pmatrix}.$$

We now describe the second kernel \mathbf{h}^2 and the bias matrix \mathbf{b}_1^2 as follows:

$$\mathbf{h}^2 \equiv \begin{pmatrix} \mathbf{e}_0 \\ \mathbf{0}^{m+t} \\ \vdots \\ \mathbf{e}^0 \\ \mathbf{0}^{n+s} \\ \mathbf{0}^m \\ \dots \\ \mathbf{0} \end{pmatrix}, \quad \mathbf{b}_1^2 \equiv \begin{pmatrix} \mathbf{0}^{n+s} \\ \mathbf{0}^{m+t} \\ \vdots \\ \mathbf{0}^n \\ \mathbf{1}^{n+s} \\ \mathbf{1}^m \\ \dots \\ \mathbf{0} \end{pmatrix}$$

This yields the following convolution computation:

$$\begin{aligned}\mathbf{h}^2 \odot \mathbf{z}^1 &\equiv \text{coeff}((X^{m+n+s+t} + 1) \cdot (\mathbf{v}(X) \cdot X^{2n+2s+m+t} + (\mathbf{p} \odot (\mathbf{h} * \mathbf{x}))(X))) \\ &\equiv \text{coeff}(\mathbf{v}(X) \cdot X^{3n+3s+2m+2t} + \mathbf{v}(X) \cdot X^{2n+2s+m+t} \\ &\quad + (\mathbf{p} \odot (\mathbf{h} * \mathbf{x}))(X) \cdot X^{m+n+s+t} + (\mathbf{p} \odot (\mathbf{h} * \mathbf{x}))(X))\end{aligned}$$

Thus we have

$$\mathbf{z}^2 \equiv \mathbf{b}_2^1 \odot (\mathbf{h}^2 * \mathbf{z}^1) \equiv \begin{pmatrix} \mathbf{0}^{n+s} \\ \mathbf{0}^{m+t} \\ \mathbf{0}^n \\ \mathbf{1}^{n+s} \\ \mathbf{1}^m \\ \dots \\ \mathbf{0} \end{pmatrix} \odot \begin{pmatrix} \mathbf{p} \odot (\mathbf{h} * \mathbf{x}) \\ \mathbf{0}^{m+t} \\ \mathbf{0}^n \\ \mathbf{p} \odot (\mathbf{h} * \mathbf{x}) \\ \mathbf{v} \\ \dots \\ \mathbf{0} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{0}^{n+s} \\ \mathbf{0}^{m+t} \\ \mathbf{0}^n \\ \mathbf{p} \odot (\mathbf{h} * \mathbf{x}) \\ \mathbf{v} \\ \dots \\ \mathbf{0} \end{pmatrix}$$

We now shift this up by $2n + s + m + t$ entries using the primitive operation defined in Proposition H.8 that costs three additional layers so that we end up with

$$\mathbf{z} \equiv \begin{pmatrix} \mathbf{p} \odot (\mathbf{h} * \mathbf{u}) \\ \mathbf{v} \\ \dots \\ \mathbf{0} \end{pmatrix}$$

Again, we note that the only operations we perform are convolutions and Hadamard product and they generalize in the obvious way to $d > 1$. \square

Finally, we show that these primitives may be composed by 'stacking' models with matching inner dimension (N', d') .

Lemma H.11. *For $f, g : \mathbb{R}^{N \times d} \rightarrow \mathbb{R}^{N \times d}$ that have (N, L_1, d, N', d') and (N, L_2, d, N', d') BASECONV models then their composition $f \circ g$ has an $(N, L_1 + L_2, d, N', d')$ BASECONV model which can be computed by performing their models in succession, or 'stacking'.*

Proof. This result follows from noting that for any $f(\mathbf{u})$ which requires L_1 layers to compute and that we can compute $f \circ g(\mathbf{u}) = g(f(\mathbf{u}))$ using the BASECONV model with L_2 layers, yielding $L_1 + L_2$ layers in total. \square

H.3.1 BaseConv-Hyena Equivalence

We show that the equivalence between BASECONV and Hyena by showing that each layer can simulate the other's computation using a constant number of layers.

Proposition H.12. *For any input $\mathbf{u} \in \mathbb{R}^{N \times d}$ and (N, L, d, N', d) -Hyena such that $\mathbf{z}_{\text{Hyena}} \equiv \text{Hyena}(\mathbf{u})$ with a set of filters \mathbf{h}^ℓ and linear projections \mathbf{p}^ℓ as per Definition H.3 for $\ell \in [L]$, there exists a $(N, 5L, d, N' + N, d)$ -BASECONV model such that $\mathbf{z}_{\text{Hyena}} \equiv \text{BASECONV}(\mathbf{u})$.*

Similarly, for any input $\mathbf{u}_{\text{BASECONV}} \in \mathbb{R}^{N \times d}$ and (N, L, d, N', d) -Coyote such that $\mathbf{z}_{\text{BASECONV}} \equiv \text{BASECONV}(\mathbf{u})$ with a set of filters \mathbf{h}^ℓ for $\ell \in [L]$, there exists a series of Hyena layers such that we have

$$\underbrace{\text{Hyena}(\text{Hyena}(\dots \text{Hyena}(\mathbf{u}_{\text{BASECONV}, \mathbf{h}})))}_{L \text{ layers}} \equiv \mathbf{z}_{\text{BASECONV}}.$$

Proof. For the input $\mathbf{u}_{\text{Hyena}} \in \mathbb{R}^{N \times d}$, the output of the ℓ th layer $\mathbf{z}_{\text{Hyena}}^\ell \in \mathbb{R}^{N' \times d'}$ for Hyena is given by (see Algorithm 3)

$$\mathbf{z}_{\text{Hyena}}^\ell \equiv \mathbf{p}_{\text{Hyena}}^\ell \odot (\mathbf{h}^\ell * \mathbf{z}^{\ell-1}),$$

where $\mathbf{p}_{\text{Hyena}}^\ell \equiv \text{Linear}(\mathbf{u}_{\text{Hyena}}) \in \mathbb{R}^{N' \times d}$. Now, using the original input $\mathbf{u}_{\text{Hyena}} \in \mathbb{R}^{N \times d}$ to Hyena, we define the following input for BASECONV using one layer:

$$\mathbf{u}_{\text{BASECONV}} \equiv \begin{pmatrix} \mathbf{u}_{\text{Hyena}} \\ \hline \mathbf{0}^{(N'-N) \times d} \\ \hline \mathbf{u}_{\text{Hyena}} \end{pmatrix}$$

Then, we simply use the $\text{remember}_{N, N, N' - N, N' - N}(\mathbf{u}_{\text{BASECONV}} : \mathbf{u}_{\text{Hyena}}, \mathbf{u}_{\text{Hyena}}, \mathbf{h}_{\text{Hyena}}^\ell, \mathbf{p}_{\text{Hyena}}^\ell)$ primitive for BASECONV. Consequently, this allows us to “remember” the input $\mathbf{u}_{\text{Hyena}}$ in the output of the previous BASECONV layer $\mathbf{z}_{\text{BASECONV}}^{\ell-1}$. We then use this to retrieve $\mathbf{p}_{\text{Hyena}}^\ell \equiv \text{linear}(\mathbf{u}_{\text{Hyena}})$ with the projection used for BASECONV given by

$$p_{\text{BASECONV}}^\ell \equiv \text{Linear}(\mathbf{z}_{\text{BASECONV}}^{\ell-1}) \equiv \begin{pmatrix} \mathbf{1}^{N \times d} \\ \hline \mathbf{p}_{\text{Hyena}}^\ell. \end{pmatrix}$$

Overall, the output of the ℓ th layer for BASECONV is given by

$$\mathbf{z}_{\text{BASECONV}}^\ell \equiv \begin{pmatrix} p_{\text{Hyena}}^\ell \odot (\mathbf{h}_{\text{Hyena}}^\ell * \mathbf{u}_{\text{Hyena}}) \\ \hline \mathbf{0}^{(M-N) \times d} \\ \hline \mathbf{u}_{\text{Hyena}} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{z}_{\text{Hyena}}^\ell \\ \hline \mathbf{0}^{(M-N) \times d} \\ \hline \mathbf{u}_{\text{Hyena}} \end{pmatrix}$$

Hence, we can reproduce the output of the ℓ th layer of Hyena using five layers of BASECONV after augmenting the input and using the remembering primitive (Proposition H.10) with internal dimension $N' + N$.

Now, for the input $\mathbf{u}_{\text{BASECONV}} \in \mathbb{R}^{N \times d}$, the output of the ℓ th layer for BASECONV is given by

$$\mathbf{z}_{\text{BASECONV}}^\ell \equiv \text{Linear}(\mathbf{z}_{\text{BASECONV}}^{\ell-1}) \odot \text{conv}(\mathbf{h}^\ell, \mathbf{z}_{\text{BASECONV}}^{\ell-1}).$$

Here, we show inductively that simply using ℓ -many Hyena models recursively simulates $\mathbf{z}_{\text{BASECONV}}^\ell$. For $\ell = 1$, we have

$$\text{Hyena}(\mathbf{u}_{\text{BASECONV}}, \mathbf{h}) \equiv \text{Linear}(\mathbf{u}_{\text{BASECONV}}) \odot (\mathbf{h}^1 * \mathbf{u}_{\text{BASECONV}}) \equiv \mathbf{z}_{\text{BASECONV}}^1.$$

We now assume that $(\ell - 1)$ -many recursive Hyena models produce $\mathbf{z}_{\text{BASECONV}}^{(\ell-1)}$. For the ℓ th layer, we then have

$$\begin{aligned} & \text{Hyena}(\text{Hyena}(\dots \text{Hyena}(\mathbf{u}_{\text{BASECONV}}, \mathbf{h}))) \\ \equiv & \text{Hyena}(\mathbf{z}_{\text{BASECONV}}^{(\ell-1)}) \\ \equiv & \text{linear}(\mathbf{z}_{\text{BASECONV}}^{(\ell-1)}) \odot \text{conv}(\mathbf{h}^\ell, \mathbf{z}_{\text{BASECONV}}^{(\ell-1)}) \\ \equiv & \mathbf{z}_{\text{BASECONV}}^\ell. \end{aligned}$$

□

H.4 Linear Arithmetic Circuits

In this section we show the relation between linear arithmetic circuits and BASECONV. We recall a few definitions from [Dao et al., 2020].

Definition H.13 (Linear Arithmetic Circuit [Bürgisser et al., 1996]). An arithmetic circuit is called a *linear arithmetic circuit* if it only uses addition, subtraction and scalar multiplication. Further, every multiplication has a fixed constant from \mathbb{F} as at least one of its two inputs. In other words, all gates in the circuit are linear functions of their inputs (i.e. of the form $ax + by$ for fixed constants $a, b \in \mathbb{F}$).

Definition H.14 (Butterfly Matrices [Dao et al., 2020]). A *butterfly factor* of size $k \geq 2$ (denoted as \mathbf{B}_k) is a matrix of the form $\mathbf{B}_k = \begin{bmatrix} \mathbf{D}_1 & \mathbf{D}_2 \\ \mathbf{D}_3 & \mathbf{D}_4 \end{bmatrix}$ where each \mathbf{D}_i is a $\frac{k}{2} \times \frac{k}{2}$ diagonal matrix. We restrict k to be a power of 2.

A *butterfly factor matrix* of size n with block size k (denoted as $\mathbf{B}_k^{(n)}$) is a block diagonal matrix of $\frac{n}{k}$ (possibly different) butterfly factors of size k :

$$\mathbf{B}_k^{(n)} = \text{diag} \left([\mathbf{B}_k]_1, [\mathbf{B}_k]_2, \dots, [\mathbf{B}_k]_{\frac{n}{k}} \right)$$

Finally, a *butterfly matrix* of size n (denoted as $\mathbf{B}^{(n)}$) is a matrix that can be expressed as a product of butterfly factor matrices: $\mathbf{B}^{(n)} = \mathbf{B}_n^{(n)} \mathbf{B}_{\frac{n}{2}}^{(n)} \dots \mathbf{B}_2^{(n)}$. Equivalently, we may define $\mathbf{B}^{(n)}$ recursively as a matrix that can be expressed in the following form:

$$\mathbf{B}^{(n)} = \mathbf{B}_n^{(n)} \begin{bmatrix} [\mathbf{B}^{(\frac{n}{2})}]_1 & 0 \\ 0 & [\mathbf{B}^{(\frac{n}{2})}]_2 \end{bmatrix}$$

(Note that $[\mathbf{B}^{(\frac{n}{2})}]_1$ and $[\mathbf{B}^{(\frac{n}{2})}]_2$ may be different.)

From Definition H.14, we observe that size n butterfly factor is comprised of three vectors $\mathbf{d}, \mathbf{d}^+, \mathbf{d}^- \in \mathbb{R}^n$ such that

$$\begin{aligned} \mathbf{d} &= (\text{diag}^{-1}(\mathbf{D}_1), \text{diag}^{-1}(\mathbf{D}_4)), \\ \mathbf{d}^+ &= \left(\mathbf{0}^{\frac{n}{2}}, \text{diag}^{-1}(\mathbf{D}_2) \right), \text{ and} \\ \mathbf{d}^- &= \left(\text{diag}^{-1}(\mathbf{D}_3), \mathbf{0}^{\frac{n}{2}} \right), \end{aligned}$$

where $\text{diag}^{-1}(\mathbf{D}) : \mathbb{R}^{n \times n} \mapsto \mathbb{R}^n$ is the mapping from diagonal matrices to the vector of its diagonal entries. Let us define $\mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3 \in \mathbb{R}^{n \times n}$ as $\text{diag}(\mathbf{d})$, $\text{diag}(\mathbf{d}^+)$, and $\text{diag}(\mathbf{d}^-)$ respectively. Then we note that

$$\mathbf{D}_1 \equiv \begin{bmatrix} \mathbf{D}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_4 \end{bmatrix} \quad \mathbf{D}_2 \mathbf{S}^{\frac{n}{2}} \equiv \begin{bmatrix} \mathbf{0} & \mathbf{D}_2 \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \quad \mathbf{S}^{\frac{n}{2}} \mathbf{D}_3 \equiv \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{D}_3 & \mathbf{0} \end{bmatrix} \quad (5)$$

where $\mathbf{S}^k \in \mathbb{F}^{n \times n}$ is a shift matrix for $i \in [n/2]$. This gives us the following proposition:

Proposition H.15. For any powers of 2, $n = k \geq 2$, any butterfly factor matrix $\mathbf{B}_k^{(n)}$ is equivalent to

$$\mathbf{B}_k^{(n)} = \mathbf{S}^{\frac{k}{2}} \mathbf{D}_3 + \mathbf{D}_2 \mathbf{S}^{\frac{n}{2}} + \mathbf{D}_1$$

where $\mathbf{D}_3, \mathbf{D}_2, \mathbf{D}_1, \mathbf{S}^{\frac{n}{2}}$ are defined as in equation 5.

We use Proposition H.15 to show that butterfly matrices can easily be computed by BASECONV.

Lemma H.16. For any $n, d \geq 2$, $k \geq 1$, and arbitrary vector $\mathbf{x} \in \mathbb{R}^{nd}$:

- (1) there exists a (N, L, d, N', d') – BASECONV that can represent $\mathbf{B}_k^{(nd)} \cdot \mathbf{x}$ with $N = n, N' = \mathcal{O}(N)$, $L = \mathcal{O}(1)$, and $d' = \mathcal{O}(d)$, and
- (2) there exists a (N, L, d, N', d') – BASECONV that can represent $\mathbf{B}^{(nd)} \cdot \mathbf{x}$ with $N = n, N' = \mathcal{O}(N)$, $L = \mathcal{O}(\log nd)$, and $d' = \mathcal{O}(d)$.

Proof. (1) Given $\mathbf{x} \in \mathbb{R}^{nd}$, construct $\mathbf{u} \in \mathbb{R}^{n \times d}$ where \mathbf{x} is the row-major form of \mathbf{u} . We show that BASECONV can compute $\mathbf{B}_{nd} \cdot \mathbf{x}$ column by column.

Let $\mathbf{A} = \mathbf{S}^{\frac{k}{2}} \mathbf{D}_3, \mathbf{C} = \mathbf{D}_2' \mathbf{S}^{\frac{n}{2}}$, and $\mathbf{D} = \mathbf{D}_1$ for $\mathbf{D}_i, \mathbf{S}^{\frac{k}{2}} \in \mathbb{R}^{nd \times nd}$ for $1 \leq i \leq 3$ as defined in Proposition H.15. We take $\mathbf{d}_1 = \mathbf{1}^{nd} \mathbf{D}, \mathbf{d}_2 = \mathbf{1}^{nd} \mathbf{C}_2, \mathbf{d}_3 = \mathbf{1}^{nd} \mathbf{A}$, which extracts the diagonal entries of \mathbf{D}_i . With this we construct $\mathbf{D}'_i \in \mathbb{R}^{n \times d}$ where \mathbf{d}_i is the row major form of \mathbf{D}'_i . This implies that

$$\mathbf{D}_i \mathbf{x} \equiv \mathbf{D}'_i \odot \mathbf{u}.$$

Then we can decompose $\mathbf{B}_{nd} \cdot \mathbf{x}$ into

$$\mathbf{B}_{nd}\mathbf{x} \equiv \mathbf{D}_1 \odot \mathbf{u} + \mathbf{D}_2 \odot \mathbf{u} + \mathbf{D}_3 \odot \mathbf{u}.$$

By Lemma H.7, each Hadamard product $\mathbf{A} \odot \mathbf{u}, \mathbf{B} \odot \mathbf{u}, \mathbf{C} \odot \mathbf{u}$ can be trivially be performed with a single layer BASECONV model. Let each of these model outputs be denoted $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3$, respectively. Finally all that remains is to compute the $\mathbf{y}_1 + \mathbf{y}_2 + \mathbf{y}_3$. We achieve this using layers of add primitives¹¹:

$$\begin{aligned} & \text{add}_n(\mathbf{y}^1 : \mathbf{y}_1, \mathbf{0}) \\ & \text{add}_n(\mathbf{y}^2 : \mathbf{y}_2, \mathbf{y}_1) \\ & \text{add}_n(\mathbf{y}^3 : \mathbf{y}_3, \mathbf{y}_1 + \mathbf{y}_2), \end{aligned}$$

where using by Proposition H.9 and Lemma H.11, this requires six more layers, and we get

$$\mathbf{y}^3 \equiv \mathbf{y}_1 + \mathbf{y}_2 + \mathbf{y}_3 \equiv \mathbf{B}_{nd}\mathbf{x}.$$

Then we can construct the $(N, L, d, N', d') - \text{BASECONV}$ as desired with $L = O(1)$ layers.

(2) From Definition H.14, $\mathbf{B}^{(nd)} = \mathbf{B}_{nd}^{(nd)} \mathbf{B}_{\frac{nd}{2}}^{(nd)} \dots \mathbf{B}_2^{(nd)}$. From (1), BASECONV can compute any butterfly matrix by simulating the $\log(nd)$ butterfly factor matrices which comprise $\mathbf{B}^{(nd)}$. With Lemma H.11, this creates a BASECONV with $5 \cdot \log(nd) = \mathcal{O}(\log(nd))$ layers. Lemma H.11 □

Butterfly matrices comprise the kaleidoscope hierarchy, which we define below:

Definition H.17 (The Kaleidoscope Hierarchy [Dao et al., 2020]).

- Define \mathcal{B} as the set of all matrices that can be expressed in the form $\mathbf{B}^{(n)}$ (for some n).
- Define $(\mathcal{B}\mathcal{B}^*)$ as the set of matrices \mathbf{M} of the form $\mathbf{M} = \mathbf{M}_1 \mathbf{M}_2^*$ for some $\mathbf{M}_1, \mathbf{M}_2 \in \mathcal{B}$.
- Define $(\mathcal{B}\mathcal{B}^*)^w$ as the set of matrices \mathbf{M} that can be expressed as $\mathbf{M} = \mathbf{M}_w \dots \mathbf{M}_2 \mathbf{M}_1$, with each $\mathbf{M}_i \in (\mathcal{B}\mathcal{B}^*)$ ($1 \leq i \leq w$). (The notation w represents width.)
- Define $(\mathcal{B}\mathcal{B}^*)_e^w$ as the set of $n \times n$ matrices \mathbf{M} that can be expressed as $\mathbf{M} = \mathbf{S} \mathbf{E} \mathbf{S}^\top$ for some $en \times en$ matrix $\mathbf{E} \in (\mathcal{B}\mathcal{B}^*)^w$, where $\mathbf{S} \in \mathbb{F}^{n \times en} = [\mathbf{I}_n \ 0 \ \dots \ 0]$ (i.e. \mathbf{M} is the upper-left corner of \mathbf{E}). (The notation e represents expansion relative to n .)

We similarly show how BASECONV can simulate any kaleidoscope matrix.

Lemma H.18. *Given $n, d \geq 2, e > 0$ for any $nd \times nd$ matrix $\mathbf{M} \in (\mathcal{B}\mathcal{B}^*)_e^w$, and $\mathbf{x} \in \mathbb{R}^{nd}$ there exists a $(N, L, d, N', d') - \text{BASECONV}$ that can represent $\mathbf{M} \cdot \mathbf{x}$ with $N = n, L = \mathcal{O}(w \log(nd)), N' = en$, and $d' = d$.*

Proof. By Definition H.17, \mathbf{M} can be decomposed with respect to size $end \times end$ matrix

$$\mathbf{E} = \mathbf{E}_1 \cdot \mathbf{E}_2 \cdots \mathbf{E}_w.$$

Further, any $\mathbf{E}_i \in (\mathcal{B}\mathcal{B}^*)$ can be expressed as a product of $2 \log end$ butterfly factor matrices. Then by Lemma H.16 and Lemma H.11 we can compute $\mathbf{E}_i \mathbf{x}'$ in by stacking $2 \log end$ $(n, d, L, en, d) - \text{BASECONV}$ models each with $L = O(1)$. Because \mathbf{E} has width w , Lemma H.11 implies that composing with each \mathbf{E}_i for $1 \leq i \leq w$ constructs a final model with $\mathcal{O}(w \log(end))$ layers. □

Finally, the kaleidoscope hierarchy is related to linear arithmetic circuits via the following result. We note that in [Dao et al., 2020] it is assumed that $w = s$, yet inspection of the proof yields the following stronger result:

Theorem H.19 ([Dao et al., 2020]). *Let \mathbf{M} be an $n \times n$ matrix such that multiplication of \mathbf{M} times an arbitrary vector \mathbf{u} can be represented as (n, s, Δ, w) -linear arithmetic circuit \mathcal{C} . Then, $\mathbf{M} \in (\mathcal{B}\mathcal{B}^*)_{\mathcal{O}(w/n)}^{\mathcal{O}(\Delta)}$.* We combine Theorem H.19 and Lemma H.18 to show that BASECONV can compute any linear arithmetic circuit with polylogarithmic factors in Δ .

Corollary H.20. *For any (nd, s, Δ, w) -linear arithmetic circuit \mathcal{C} that can be represented by a matrix $\mathbf{M} \in \mathbb{R}^{nd \times nd}$ multiplied by a vector $\mathbf{x} \in \mathbb{R}^{nd}$, there exists an equivalent $(n, \Delta', d, w, d) - \text{BASECONV}$ with $\Delta' = \mathcal{O}(\Delta \log(w))$ such that $\mathbf{M}\mathbf{x} = \text{BASECONV}(\mathbf{u}, \mathbf{h})$ where \mathbf{x} is the row major form of $\mathbf{u} \in \mathbb{R}^{n \times d}$.*

¹¹Recall that $\text{add}_n(\mathbf{y} : \mathbf{x}, \mathbf{S})$ adds the subvector \mathbf{x} to \mathbf{S} for the input \mathbf{y} .

H.5 General Arithmetic Circuits

We are now ready to prove the result that yields the equivalency between arithmetic circuits and BASECONV.

Theorem H.21. *For any (nd, s, Δ, w) -arithmetic circuit \mathcal{C} , there exists an equivalent $(N, \Delta', d, N', d') - \text{BASECONV}$ with $N = n$, $\Delta' = \mathcal{O}(\Delta \log w)$, $N' = \mathcal{O}(w)$, $d' = d$ that simulates \mathcal{C} .*

For the reader's convenience, we begin with a proof sketch and then provide the details afterwards.

Proof Sketch of Theorem H.21. Let us layer \mathcal{C} so that each layer \mathcal{C}_ℓ for $\ell \in [L_\mathcal{C}]$ either only has linear gates or multiplication gates, where the number of such layers $L_\mathcal{C} = \mathcal{O}(\Delta)$. The composition of all \mathcal{C}_ℓ layers results in \mathcal{C} . We use $\mathbf{z}^{\ell+1} \in \mathbb{R}^w$ to denote the output of the ℓ -th layer \mathcal{C}_ℓ which feeds as the input to the $(\ell+1)$ -th layer $\mathcal{C}_{\ell+1}$. Here, we note that if we can simulate each \mathcal{C}_ℓ with BASECONV, then we can simulate the entire layered circuit \mathcal{C} due to Lemma H.11.

Now, if the layer $\mathcal{C}_\ell^{\text{lin}}$ is a linear layer (with only addition gates), then it can be represented by a matrix $\mathbf{M} \in \mathbb{R}^{w \times w}$ multiplied by $\mathbf{z}^\ell \in \mathbb{R}^w$ (We can append with 0s if necessary so that the input from the previous gates can be written as w -length vector). Thus, we can apply Corollary H.20 to simulate $\mathcal{C}_\ell^{\text{lin}}$ with an equivalent $(n, \log w, d, \mathcal{O}(\log w), d) - \text{BASECONV}$ model.

Next, if $\mathcal{C}_\ell^{\text{mult}}$ instead consists of only the multiplication gates. Then, we note here that the output \mathbf{z}^ℓ may not exactly equal the input to $\mathcal{C}_\ell^{\text{mult}}$. Nevertheless, we can apply a $\mathcal{O}(w)$ sparse linear map $\mathbf{R} \in \mathbb{R}^{w \times w}$ so that $\mathbf{R}\mathbf{z}^\ell$ yields vectors $\mathbf{v}^1, \mathbf{v}^2$, and \mathbf{v}^3 , where \mathbf{v}_1 constitutes the “first” input to all the multiplication gates and \mathbf{v}_2 constitutes all the “second” inputs while \mathbf{v}^3 consists of all entries needed as inputs in the subsequent layers. That is, for the i th gate in $\mathcal{C}_\ell^{\text{mult}}$, we compute $\mathbf{v}_i^1 \cdot \mathbf{v}_i^2$. This implies that for all the gates in $\mathcal{C}_\ell^{\text{mult}}$, we can simply compute $\mathbf{v}_1 \odot \mathbf{v}_2$. To this end, we use the `remember` primitive with constant number of layers from Proposition H.10 to define a $(n, \mathcal{O}(\log w), d, w, d) - \text{BASECONV}$ model that remembers \mathbf{v}^3 while performing the Hadamard product of \mathbf{v}^1 with \mathbf{v}^2 .

Overall, we can then collect all the resulting BASECONV layers and compose them as in Lemma H.11 to simulate \mathcal{C} . Overall, the number of layers used is given by $\mathcal{O}(\Delta \log w)$ while the internal dimension remains fixed at w . \square

Using the outline in the proof sketch, we now delve into a detailed analysis of the arithmetic circuit \mathcal{C} to an equivalent BASECONV model.

Proof of Theorem H.21. Let $\mathbf{u} \in \mathbb{R}^{nd}$ be the input to the arithmetic circuit \mathcal{C} with depth Δ and width w . We begin by rearranging the circuit into layers of addition and multiplication gates. That is, each layer \mathcal{C}^ℓ has either all addition gates or all multiplication gates. This allows us to readily apply the results from Appendix H.4. Note here that we can assert that the number of such layers $L_\mathcal{C} = \mathcal{O}(\Delta)$. Moreover, we construe the input to each such circuit as a vector of length w by appending with extra 0s if necessary so that the composition of the layers results in a circuit equivalent to \mathcal{C} . See Fig. 8 for an example of such decomposition

Let $\mathbf{z}^\ell \in \mathbb{R}^w$ denote the input to each layer \mathcal{C}_ℓ for $\ell \in [L_\mathcal{C}]$ with $\mathbf{z}^1 \equiv (\mathbf{u}, \mathbf{0}^{w-n^d})$. It is important to note here that we may *not* have $\mathbf{z}^{\ell+1} \equiv \mathcal{C}_\ell(\mathbf{z}^\ell)$ since the inputs for gates at the $(\ell+1)$ -th layer may come from any of the previous layers. We now handle the case of layers with addition gates, multiplication gates, and the intermediate stage separately.

Addition Gates. Let $\mathcal{C}_\ell^{\text{lin}}$ denote an arbitrary linear layer which only contains the addition gates that takes in \mathbf{z}^ℓ as input. We know that there exists a matrix $\mathbf{M} \in \mathbb{R}^{w \times w'}$ such that we have

$$\mathcal{C}_\ell^{\text{lin}}(\mathbf{z}^\ell) \equiv \mathbf{M}\mathbf{z}^\ell.$$

Here, note here that we may need entries from the vector \mathbf{z}^ℓ in subsequent layers. Let s be the total number of such entries, then for at each index for such entries $i \in [w]$, we have the corresponding index $i_s \in [s]$. We then append the i th standard row vector into the matrix \mathbf{M} to get the matrix $\mathbf{M}' \in \mathbb{R}^{w \times (w'+s)}$ so that we have

$$(\mathbf{M}'\mathbf{z}^\ell)[j] = \begin{cases} (\mathbf{M}\mathbf{z}^\ell)[j] & \text{if } j < w' \\ \mathbf{z}^\ell[i] & \text{if } j = w' + i_s \end{cases}$$

Here, note that we must have $w' + s \leq w$ as w is the width of the circuit. If needed, we then append the matrix \mathbf{M}' with all-zero rows $\mathbf{0}^w$ so that we get the matrix $\mathbf{M}'' \in \mathbb{R}^{w \times w}$. Note here that we have thus

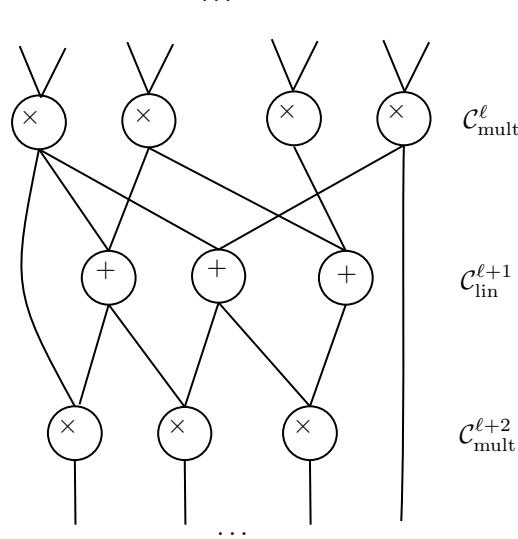


Figure 8: An example decomposition of an arithmetic circuit as layers of only addition or multiplication gates.

preserved all entries needed in subsequent layers and incorporated the output of the ℓ th layer with the output $\mathbf{M}'\mathbf{z}^\ell \equiv \mathbf{z}^{\ell+1}$ serving as the input of the $(\ell+1)$ -th layer. Applying Corollary H.20 then yields an equivalent $(n, \mathcal{O}(\log w), d, w, d)$ – BASECONV model.

Multiplication Gates. Next, we deal with the layer $\mathcal{C}_{\text{mult}}^\ell$ of multiplication gates by collecting the two inputs to each multiplication gates as $\mathbf{v}_1^\ell, \mathbf{v}_2^\ell$. Note that for the input $\mathbf{z}^\ell \in \mathbb{R}^w$ from the previous layer, we will again have entries that need to be used in the subsequent layers that we will denote as \mathbf{v}_3^ℓ . We thus need to permute the entries of \mathbf{z}^ℓ to get the vector $[\mathbf{v}_1^\ell :: \mathbf{v}_2^\ell :: \mathbf{v}_3^\ell]$ so that we can remember \mathbf{v}_3^ℓ while taking the Hadamard product $\mathbf{v}_1^\ell \odot \mathbf{v}_2^\ell$. To this end, we can achieve this permutation of entries using a $\mathcal{O}(w)$ -sparse linear matrix¹² $\mathbf{R}^\ell \in \mathbb{R}^{w \times w}$ with which is equivalently represented by an $(w, w, 1, w)$ -linear arithmetic circuit that simply moves the appropriate wires from the previous layer. This can again be achieved by a equivalent $(n, \mathcal{O}(\log w), d, w, d)$ – BASECONV model. That is, $\mathbf{R}^\ell \mathbf{z}^\ell$ has the following form:

$$\mathbf{R}^\ell \mathbf{z}^\ell \equiv \begin{pmatrix} \mathbf{v}_1^\ell \\ \mathbf{v}_2^\ell \\ \vdots \\ \mathbf{v}_3^\ell \end{pmatrix}.$$

Next, we can now define a $(n, 1, d, n, d)$ – BASECONV which extracts \mathbf{v}_1^ℓ as the projection with the input to the remember primitive given by $\mathbf{y}^\ell \equiv (\mathbf{v}_2^\ell, \mathbf{v}_3^\ell, \mathbf{0})$. We then specify the **remember** primitive as

¹²Each row of \mathbf{R}^ℓ has exactly one non-zero entry.

remember $(\mathbf{y}^\ell : \mathbf{v}_2^\ell, \mathbf{v}_3^\ell, \mathbf{e}_0, \mathbf{v}_1^\ell)$ which computes the following

$$\begin{aligned} \mathbf{z}^{\ell+1} &\equiv \begin{pmatrix} \mathbf{v}_1^\ell \odot (\mathbf{e}_0 * \mathbf{v}_2^\ell) \\ \mathbf{v}_3^\ell \\ \mathbf{0} \end{pmatrix} \\ &\equiv \begin{pmatrix} \mathbf{v}_1^\ell \odot \mathbf{v}_2^\ell \\ \mathbf{v}_3^\ell \\ \mathbf{0} \end{pmatrix} \\ &\equiv \mathcal{C}_{\text{mult}}^\ell(\mathbf{z}^\ell) \in \mathbb{R}^w. \end{aligned}$$

Using Proposition H.10, we know that this requires a $(n, \mathcal{O}(\log w), d, w, d) - \text{BASECONV}$ which remembers the entries that will serve as inputs in subsequent layers while performing the Hadamard product $\mathbf{v}^1 \odot \mathbf{v}^2$. Overall, we can now stack all the resulting BASECONV layers and compose them as in Lemma H.11 to simulate \mathcal{C} . Overall, the number of layers blows up by $\mathcal{O}(\Delta \log w)$ while the internal dimension remains fixed at w . \square

H.6 The Retnet Reduction and Space Complexity for AR

We are now in a position to show that there exists an equivalent BASECONV model that can simulate a Retnet layer from Algorithm 6. Recall that for an input sequence $\mathbf{u} \in \mathbb{R}^{N \times d}$, we project the input using weight matrices $\mathbf{W}_A, \mathbf{W}_C, \mathbf{W}_V \in \mathbb{R}^{d \times d}$ to $\mathbf{A}, \mathbf{C}, \mathbf{V} \in \mathbb{R}^{N \times d}$ and compute the following recurrence (see line 5 in Algorithm 6):

$$\mathbf{z}^n \equiv \gamma \mathbf{z}^{n-1} + (\mathbf{A}[n, :])^\top \mathbf{V}[n, :], \quad (6)$$

where $\mathbf{z}^n \in \mathbb{R}^{d \times d}$. We unroll this recurrence to express each state \mathbf{z}^n in terms of \mathbf{u} with coefficients given by γ and the weight matrices $\mathbf{W}_A, \mathbf{W}_C, \mathbf{W}_V$ as follows:

$$\begin{aligned} \mathbf{z}^0 &\equiv ((\mathbf{u} \mathbf{W}_A)[0, :])^\top (\mathbf{u} \mathbf{W}_V)[0, :] \\ &\equiv (\mathbf{u}[0, :] \mathbf{W}_A)^\top (\mathbf{u}[0, :]) \mathbf{W}_V \\ &\equiv \mathbf{W}_A^\top (\mathbf{u}[0, :])^\top (\mathbf{u}[0, :]) \mathbf{W}_V \\ &\equiv \mathbf{W}_A^\top (\mathbf{u}^\top[:, 0] \mathbf{u}[0, :]) \mathbf{W}_V. \end{aligned}$$

Similarly, we have

$$\begin{aligned} \mathbf{z}^1 &\equiv \gamma \mathbf{z}^0 + \mathbf{W}_A^\top (\mathbf{u}^\top[:, 1] \mathbf{u}[1, :]) \mathbf{W}_V \\ &\equiv \gamma (\mathbf{W}_A^\top (\mathbf{u}^\top[:, 1] \mathbf{u}[0, :]) \mathbf{W}_V) + \mathbf{W}_A^\top (\mathbf{u}^\top[:, 1] \mathbf{u}[1, :]) \mathbf{W}_V \\ &\equiv \mathbf{W}_A^\top (\gamma (\mathbf{u}^\top[:, 0] \mathbf{u}[0, :]) + \mathbf{u}^\top[:, 1] \mathbf{u}[1, :]) \mathbf{W}_V \end{aligned}$$

We can then generalize this to $n \in [N]$ to get

$$\mathbf{z}^n \equiv \mathbf{W}_A^\top \left(\sum_{i=0}^n \gamma^{n-i} \mathbf{u}^\top[:, i] \mathbf{u}[i, :] \right) \mathbf{W}_V. \quad (7)$$

The above helps us infer that \mathbf{z}^n can be expressed as a polynomial in $\{\mathbf{u}[0, 0], \mathbf{u}[0, 1], \dots, \mathbf{u}[N-1, d-1]\}$. For each polynomial, there exists an arithmetic circuit that computes the polynomial in a natural way, whence we can apply Theorem H.21 to get an equivalent BASECONV model.

Corollary H.22. *For a RetNet model with $\mathcal{O}(d^2)$ parameters and N layers, there exists an equivalent BASECONV model that uses $\mathcal{O}(Nd)$ parameters and $\mathcal{O}(N \log d)$ layers.*

Proof. We will start by describing the arithmetic circuit that computes the states $\mathbf{z}^n \in \mathbb{R}^{d \times d}$ here. Indeed, each state requires exactly two alternative layers with only multiplication gates and only addition gates, respectively. First, we note that for each $n \in N$, we have

$$\mathbf{z}^n \equiv \gamma \mathbf{z}^{n-1} + (\mathbf{A}[n, :])^\top \mathbf{V}[n, :].$$

Since we can compute \mathbf{z}^0 in $\mathcal{O}(1)$ layers and computing $\gamma \cdot \mathbf{z}^{n-1}$ needs exactly one layer with each entry from \mathbf{z}^{n-1} serving as an input to a multiplication gate along with γ , the depth of the circuit is $\mathcal{O}(N)$ with width d^2 . We can thus apply Theorem H.21 to get an $(\mathcal{O}(N), \mathcal{O}(N \log d), d^2, \mathcal{O}(d^2), \mathcal{O}(d^2))$ –BASECONV model that can compute \mathbf{z}^n for each $n \in N$. \square

The Space Complexity of AR In the context of the Retnet Model and associative recall, it is worth exploring the space complexity of the associative recall (AR) problem:

The AR problem takes key-value pairs $\{\mathbf{k}_i, \mathbf{v}_i\}_{i=0}^{n-1}$ along with a query \mathbf{q} appended at the end as input and the goal is to output \mathbf{v}_i if $\mathbf{q} = \mathbf{k}_i$ for some $i \in [0, N - 1]$.

Indeed, we will first provide a lower-bound for *any* model that purports to solve AR. To this end, we will require a randomized communication complexity lower bound result for the *index problem*:

The index problem has two agents, Alice and Bob, where Alice has a string $\mathbf{x} \in \{0, 1\}^n$ and Bob has an index $i \in [n]$, and the goal for the players is to output the i -th entry \mathbf{x}_i . Moreover, we also require the communication to be *one-way*: only Alice is allowed to send a single message to Bob and Bob needs to output the answer.

We will make use of the following lower-bound result.

Theorem H.23 ([Jayram et al., 2008]). *The one-way randomized communication complexity¹³ of the index problem for sending an n -length bit string is $\Omega(n)$.*

We now use Theorem H.23 to provide a lower bound on the number of bits required by the Retnet model to solve AR.

Corollary H.24. *The RetNet model requires $\Omega(N)$ -bits to solve AR for $d \leq \sqrt{N}$.*

Proof. Consider an instance (\mathbf{x}, i) of the index problem with $\mathbf{x} \in \{0, 1\}^N$. We now describe the corresponding instance of the AR problem:

$$\{i, \mathbf{x}_i\}_{i=0}^{N-1}, i. \quad (8)$$

Next, consider the following one-way protocol for solving the index problem using the RetNet model. Alice with their access of $\mathbf{x} \in \{0, 1\}^N$ generate an input for AR (without the query) as in equation 8. Alice then runs the RetNet model on $\{i, \mathbf{x}_i\}_{i=0}^{N-1}$ and sends the memory content of running the RetNet model to Bob. This should include the output \mathbf{z}^{N-1} as we can reasonably assume that both have access weight matrices $\mathbf{W}_A, \mathbf{W}_C, \mathbf{W}_V$ and the scalar γ . Since we assume that this model solves AR, the output $\mathbf{Out}[N, :] = \mathbf{x}_i$ should contain the associated value of i . Here, Bob can compute $\mathbf{Out}[N, :]$ by using the memory content sent by Alice along with the term $\gamma \mathbf{z}^{N-1}$:

$$\mathbf{x}_i = \mathbf{Out}[N, :] = \mathbf{C}[N, :] \mathbf{z}^N = \mathbf{C}[N, :] (\gamma \mathbf{z}^{N-1} + (\mathbf{A}[N, :])^\top \mathbf{V}[N, :]).$$

That is, the total number of bits that are communicated in this protocol is $O(d^2)$. For $d \leq \sqrt{N}$, have shown that a one-way communication protocol exists for solving the index problem exists that uses $o(N)$ communication complexity. This contradicts Theorem H.23 and hence, we conclude that the ReNet model solving AR also needs $\Omega(N)$ bits. \square

H.7 The Multiple-Query Associative Recall Problem

H.7.1 Introduction

In this section, we consider a general version of the associative recall problem [Ba et al., 2016].

¹³The randomized communication complexity of function f is defined as $\min_{\pi} \|\pi\|$, where π ranges over all randomized protocols that can solve f with probability of success at least $2/3$.

Setup. Next, we will redefine the *multiple-query associative recall* problem (MQAR) from Definition 3.1 to a slightly more general problem:

Suppose we are given an input sequence $\mathbf{u}[0 \cdots N-1] \triangleq \left\{ (\mathbf{k}_0, \mathbf{v}_0, \mathbf{q}_0), \dots, (\mathbf{k}_{\frac{N}{3}-1}, \mathbf{v}_{\frac{N}{3}-1}, \mathbf{q}_{\frac{N}{3}-1}) \right\}$

with each $\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i \in C$ is a token drawn from a vocabulary of size $c = |C|$. Our goal is then to check, for each $1 \leq i \leq \frac{N}{3} - 1$, whether there exists $0 \leq j < i$ such that $\mathbf{q}_i \equiv \mathbf{k}_j$, and if so, output \mathbf{v}_j .

Here, we note that it suffices to have $d \approx \log(c)$ so that $\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i$ is embedded in $\{0, 1\}^d$. However, we will specify the specific embedding being used for the results below. Here, we construe the tokens $\mathbf{k}_i, \mathbf{q}_i$ and \mathbf{v}_i to be the *keys*, the *queries*, and the *associated values*. Indeed, it might be helpful to think of the input \mathbf{u} as a streaming sequence of key-value pairs for which we sequentially employ standard associative recall for every key that shows up in the sequence so far.

To see that the above generalizes Definition 3.1, consider a sequence of length $\frac{N}{3}$: $\mathbf{u}[0 \cdots N-1] := \{\mathbf{x}_0, \dots, \mathbf{x}_{N-1}\}$, where each $\mathbf{x}_i \in C$. The goal of Definition 3.1 is then to check, for each $1 \leq i < N-1$, whether there exists $0 \leq j < i$ such that $\mathbf{x}_i \equiv \mathbf{x}_j$, and if so, output \mathbf{x}_{j+1} , and continue otherwise. We can reduce this problem to the above general formulation by taking the following sequence of tuples as the input $\{(\mathbf{x}_i, \mathbf{x}_{i+1}, \mathbf{x}_i)\}$.

Remark H.25. As noted above, this version is more general than Definition 3.1. Thus, the results proven in the sequel, which are proven for the above general MQAR can be ported (with constant blowup in parameters) to get the results corresponding results for Definition 3.1.

H.7.2 Mqar Solution via Attention

Before describing how BASECONV solves the multiple-query associative recall problem, we discuss how Attention solves it trivially using pairwise inner-products. To this end, we will specify how the input is presented to attention.

Remark H.26. We note that the input for the multiple-query associative recall problem $\mathbf{u} \in \{0, 1\}^{N \times d}$ has designated indices for the keys, queries, and values in the sequence. We gather these indices below:

$$\begin{aligned} \mathcal{K} &= \{i \in \{0, \dots, N-1\} \mid i \equiv 0 \pmod{3}\}, \\ \mathcal{V} &= \{i \in \{0, \dots, N-1\} \mid i \equiv 1 \pmod{3}\}, \\ \mathcal{Q} &= \{i \in \{0, \dots, N-1\} \mid i \equiv 2 \pmod{3}\}. \end{aligned} \tag{9}$$

The input $\mathbf{u} \in \mathbb{R}^{N \times d}$ to Attention for $d = 3c$ is then given by

$$\mathbf{u}[i, :] \equiv \begin{cases} [\mathbf{k}_i : \mathbf{0}^c : \mathbf{0}^c] & \text{if } i \in \mathcal{K} \\ [\mathbf{0}^c : \mathbf{v}_i : \mathbf{0}^c] & \text{if } i \in \mathcal{V} \\ [\mathbf{0}^c : \mathbf{0}^c : \mathbf{q}_i] & \text{if } i \in \mathcal{Q} \end{cases}$$

Here, each $\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i$ is embedded as a one-hot encoding in $\{0, 1\}^c$.

Without softmax, the output for an attention layer $\mathbf{O} \in \mathbb{R}^{N \times d}$ is given by

$$\mathbf{O} \equiv (\mathbf{QK}^\top) \mathbf{V}, \tag{10}$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ are defined as $\mathbf{uW}_Q, \mathbf{uW}_K, \mathbf{uW}_V$ for $\mathbf{u} \in \mathbb{R}^{N \times d}$. Instead of position embeddings, we use ALiBi, a popular technique that biases the attention scores \mathbf{QK}^\top with a lower-triangular Toeplitz matrix $\mathbf{B} \in \mathbb{R}^{N \times N}$ [Press et al., 2021]. The values in this matrix are controlled by a fixed hyperparameter so they do not count towards the number of parameters in the model.

Algorithm 8 ALiBi-without-softmax ($\mathbf{u}[0 \cdots N-1], \mathbf{O}_{\text{prev}}[0 \cdots N-1], \mathbf{B}$)

Input: Input sequence $\mathbf{u}[0 \cdots N-1] \triangleq \{(\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i)\}_{i=0}^{\frac{N}{3}-1}$ with each $\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i \in \{0, 1\}^{3c}$, previous layer's output $\mathbf{O}_{\text{prev}}[0 \cdots N-1] \in \mathbb{R}^{N \times 3c}$, and linear bias $\mathbf{B} \in \mathbb{R}^{N \times N}$.

- 1: Add $\mathbf{u}_{\text{curr}} \leftarrow \mathbf{u} + \mathbf{O}_{\text{prev}}$ as an input to this layer.
- 2: $\mathbf{K}, \mathbf{Q}, \mathbf{V} \leftarrow \mathbf{u}_{\text{curr}} \mathbf{W}_Q, \mathbf{u}_{\text{curr}} \mathbf{W}_K, \mathbf{u}_{\text{curr}} \mathbf{W}_V$.
- 3: $\mathbf{O} \leftarrow (\mathbf{QK}^\top + \mathbf{B}) \mathbf{V}$
- 4: **return** \mathbf{O} as the output of this layer.

Proposition H.27. *Given an input $\mathbf{u} \in \{0, 1\}^{N \times d}$ (encoded as in Remark H.26) where $d = 3c$, Attention with linear biases (even without using soft-max) solves MQAR for \mathbf{u} using $\mathcal{O}(c^2)$ parameters, $\mathcal{O}(Nc^2 + N^2c)$ time complexity and $\mathcal{O}(1)$ layers.*

Proof. We use two layers of attention. We will start by specifying the projection matrices for the first layer $\mathbf{W}_Q^1, \mathbf{W}_K^1, \mathbf{W}_V^1 \in \mathbb{R}^{d \times d}$ as:

$$\mathbf{W}_K^1 \equiv \mathbf{W}_Q^1 \equiv \mathbf{0}, \quad \mathbf{W}_V^1 \equiv \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{c \times c} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}$$

Above, \mathbf{W}_V^1 is meant to isolate the \mathbf{v}_i embeddings. For the first layer, we then have $\mathbf{Q}^1 \equiv \mathbf{K}^1 \equiv \mathbf{0}$, and

$$\mathbf{V}^1[i, :] := \mathbf{u}^1[i, :] \mathbf{W}_V^1 \equiv \begin{cases} [\mathbf{0}^c : \mathbf{v}_i : \mathbf{0}^c] & \text{if } i \in \mathcal{V}, \\ \mathbf{0}^d & \text{otherwise,} \end{cases}$$

where $\mathcal{K}, \mathcal{Q}, \mathcal{V}$ are defined as in equation 9. The output for the first layer is given by the following:

$$\begin{aligned} \mathbf{O}^1[i, :] &= ((\mathbf{Q}\mathbf{K}^\top + \mathbf{B}^1) \mathbf{V})[i, :] \\ &= \mathbf{B}^1 \mathbf{V}[i, :] \\ &= \begin{cases} [\mathbf{0}^c : \mathbf{v}_i : \mathbf{0}^c] & \text{if } i \in \mathcal{K} \\ \mathbf{0}^d & \text{otherwise,} \end{cases} \end{aligned}$$

The $\mathbf{Q}\mathbf{K}^\top$ is ignored ($\equiv \mathbf{0}$) and we isolate the shifted sequence by setting the bias matrix \mathbf{B} appropriately. In particular, the last equality follows from the fact that \mathbf{B}^1 is an up-shift matrix that shifts each row of \mathbf{V} by 1. We apply the residual connection at the end of the standard Transformer block to insert the \mathbf{k}_i adjacent to the \mathbf{v}_i . For the second layer, the input $\mathbf{u}^2 \in \mathbb{R}^{N \times d}$ is given by

$$\mathbf{u}^2[i, :] \equiv \begin{cases} [\mathbf{k}_i : \mathbf{v}_i : \mathbf{0}^c] & \text{if } i \in \mathcal{K} \\ [\mathbf{0}^c : \mathbf{v}_i : \mathbf{0}^c] & \text{if } i \in \mathcal{V}, \\ [\mathbf{0}^c : \mathbf{0}^c : \mathbf{q}_i] & \text{if } i \in \mathcal{Q} \end{cases}$$

Further, we take the following projection matrices:

$$\mathbf{W}_K^2 \equiv \begin{pmatrix} \mathbf{I}_{c \times c} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}, \quad \mathbf{W}_Q^2 \equiv \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{I}_{c \times c} & \mathbf{0} & \mathbf{0} \end{pmatrix}, \quad \mathbf{W}_V^2 \equiv \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{I}_{c \times c} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}$$

We then have the following matrices as input to attention after applying projection:

$$\begin{aligned} \mathbf{Q}^2[i, :] &:= \mathbf{u}^2[i, :] \mathbf{W}_K^2 \equiv \begin{cases} [\mathbf{q}_i : \mathbf{0}^{2c}] & \text{if } i \in \mathcal{Q}, \\ \mathbf{0}^d & \text{otherwise,} \end{cases} \\ \mathbf{K}^2[i, :] &:= \mathbf{u}^2[i, :] \mathbf{W}_Q^2 \equiv \begin{cases} [\mathbf{k}_i : \mathbf{0}^{2c}] & \text{if } i \in \mathcal{K}, \\ \mathbf{0}^d & \text{otherwise,} \end{cases} \\ \mathbf{V}^2[i, :] &:= \mathbf{u}^2[i, :] \mathbf{W}_V^2 \equiv \begin{cases} [\mathbf{v}_i : \mathbf{0}^{2c}] & \text{if } i \in \mathcal{K} \cup \mathcal{V}, \\ \mathbf{0}^d & \text{otherwise.} \end{cases} \end{aligned} \tag{11}$$

Here, note that the values have been shifted to the corresponding key position. Next, we compute the term in the parenthesis for the second layer as

$$\begin{aligned} (\mathbf{Q}^2 \mathbf{K}^{2\top})[i, j] &= (\mathbf{Q}^2 \mathbf{K}^{2\top})[i, j] = \langle \mathbf{Q}^2[i, :], \mathbf{K}^{2\top}[:, j] \rangle \\ &= \langle \mathbf{Q}^2[i, :], \mathbf{K}^2[j, :] \rangle \\ &= \begin{cases} \langle \mathbf{q}_i, \mathbf{k}_j \rangle & \text{if } i \in \mathcal{Q}, j \in \mathcal{K} \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} 1 & \text{if } i \in \mathcal{Q}, j \in \mathcal{K}, \mathbf{q}_i \equiv \mathbf{k}_j \equiv \mathbf{e}_k \text{ for some } k \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Finally, we compute the output as follows:

$$\begin{aligned}
\mathbf{O}^2[i, :] &= ((\mathbf{Q}^2 \mathbf{K}^{2\top} + \mathbf{B}^2) \mathbf{V}^2)[i, :] \\
&= ((\mathbf{Q}^2 \mathbf{K}^{2\top} + \mathbf{0}) \mathbf{V}^2)[i, :] \\
&= (\mathbf{Q}^2 \mathbf{K}^{2\top})[i, :] \cdot \mathbf{V}^2 \\
&= \sum_{j=0}^{N-1} (\mathbf{Q}^2 \mathbf{K}^{2\top})[i, j] \cdot \mathbf{V}^2[j, :] \\
&= \sum_{j \in \mathcal{K}} (\mathbf{Q}^2 \mathbf{K}^{2\top})[i, j] \cdot [\mathbf{v}_j : \mathbf{0}^{2c}] \\
&= \begin{cases} [\mathbf{v}_j : \mathbf{0}^{2c}] & \text{if } j \in \mathcal{K}, i \in \mathcal{Q}, \mathbf{q}_i \equiv \mathbf{k}_j \\ \mathbf{0}^d & \text{otherwise} \end{cases},
\end{aligned}$$

where we use the fact that for each index $j \in \mathcal{K}$, the matrix \mathbf{V}^2 contains the associated value from equation 11. Thus, for each query \mathbf{q}_i , we solve the associated value problem yielding a match for the j th key.

In total, we only need $\mathcal{O}(c^2)$ -many parameters to perform these multiplications and the linear bias in the first layer is a hyperparameter that is static and unlearned; the time complexity comes from the multiplication of $\mathbf{Q}\mathbf{K}^\top$ in $\mathcal{O}(N^2c)$, and projections in $\mathcal{O}(Nc^2)$. Finally, we only need $\mathcal{O}(1)$ layers for this solution. \square

In the sequel, we develop a parallel algorithm to solve the multiple-query associative recall problem with $\mathcal{O}(Nd \cdot \log^2 N)$ work complexity and $\mathcal{O}(d \cdot \log^2 N)$ time. We then convert the algorithm into a BASECONV model via the route of arithmetic circuits, which then solves the multiple-query associative recall problem with $\tilde{\mathcal{O}}(1)$ layers and $\tilde{\mathcal{O}}(Nd)$ parameters.

H.7.3 Initial Attempt: A Sequential Algorithm

We will first discuss the algorithm that simply uses an associative array to solve the multiple-query associative recall problem. Specifically, we want to use a data structure that allows for logarithmic insertion and membership query. Here, we do not specify a choice but data structures including self-balancing binary search trees which allow for $\mathcal{O}(\log N \cdot d)$ `insert` and `find` operations for d -bit entries should be sufficient.

Algorithm 9 Sequential-MQ-AR (\mathbf{u}) [$0 \cdots N - 1$]

Input: Input sequence $\mathbf{u}[0 \cdots N - 1] \triangleq \{(\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i)\}_{i=0}^{\frac{N}{3}-1}$ with each $\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i \in \{0, 1\}^d$.

- 1: Initialize an associative array with `insert` and `find` and an output array `out` $\leftarrow \emptyset$.
- 2: **for** $i \in \{0, \dots, \frac{N}{3} - 1\}$ **do**
- 3: $(\mathbf{k}_j, \mathbf{v}_j) \leftarrow \text{find}(\mathbf{q}_i)$ ▷ Query for \mathbf{q}_i in the data structure.
- 4: **if** \mathbf{k}_j is not null **then**
- 5: Add \mathbf{v}_j to `out`.
- 6: `insert`($\mathbf{k}_i, \mathbf{v}_i$) ▷ Add the key-value pair to the data structure.
- 7: **return** `out`.

Proposition H.28. *Algorithm 9 solves the multiple-query associative recall problem (MQAR) in $\mathcal{O}(dN \log N)$ time for an input sequence $\mathbf{u} \in \{0, 1\}^{N \times d}$.*

Proof. For any $i \in \{0, \dots, \frac{N}{3} - 1\}$, we know that both insertion and lookup operations take $\mathcal{O}(\log(i) \cdot d)$ time. Overall, the runtime of the algorithm is

$$\sum_{i=0}^{\frac{N}{3}-1} \mathcal{O}(\log(i) \cdot d) = \mathcal{O}(\log(N!) \cdot d) = \mathcal{O}(N \log N \cdot d).$$

\square

H.7.4 Algorithm via Parallel Binary Search

Our plan is to convert the algorithm for solving the multiple-query associative recall problem in the RAM model into an arithmetic circuit, which by Theorem H.21 will lead to a BASECONV model that solves the multiple-query associative recall problem. With respect to Algorithm 9, it may be the case that the arithmetic circuit has a large number of layers $\Omega(N)$. Unfortunately, this would imply that the resulting BASECONV model may have near quadratic complexity in N . Instead, we now initiate our effort into designing a BASECONV model with both small enough number of parameters and number of layers. Here, we will first subdivide the problem using dyadic intervals into $\mathcal{O}(N)$ subproblems and reduce each such subproblem into a *multiple search problem* [Akl and Meijer, 1990]. To this end, we briefly introduce the multiple search problem below.

Given two array of numbers $A \triangleq a_0 \leq \dots \leq a_{n-1}$ and $B \triangleq (b_0 \leq \dots \leq b_{m-1})$ with $n \leq m$, for each $a_j \in A$, the goal is to find the smallest element in B that is larger than or equal to a_j .

The multiple search problem is solved by a *parallel binary search* (pbs) algorithm in [Akl and Meijer, 1990] with work complexity $\mathcal{O}(n \cdot \log m)$ and time $\mathcal{O}(\log n \log m)$. Specifically, for sorted arrays $A[0 \dots n-1]$ and $B[0 \dots m-1]$, pbs constructs the array $C[0 \dots n-1]$ defined as

$$C[i] \triangleq \begin{cases} \min_{0 \leq j < m} \{j \mid A[i] \leq B[j]\} & \text{if } A[i] \leq B[m-1] \\ m & \text{otherwise.} \end{cases} \quad (12)$$

The algorithm itself runs in exclusive-read exclusive-write (EREW) PRAM model—no two processors are allowed to read from or write into the same memory location at the same time.

We now augment the algorithm copied from [Akl and Meijer, 1990] for our purposes below.

Algorithm 10 pbs-key-values ($\mathbf{q}[s \dots t]$, $\mathbf{k}[x \dots y]$, n, m)

Input: sorted arrays $\mathbf{q}[s \dots t] := \{\mathbf{q}_i\}_{i=s}^t$, $\mathbf{k}[x \dots y] := \{(j, \mathbf{k}_j)\}_{j=x}^y$.

- 1: Initialize n processors denoted P_0, P_1, \dots, P_{n-1} \triangleright {Sequential steps are assumed to be executed by P_s .}
- 2: Initialize the output array $C := [m]_{i=s}^t$.
- 3: **if** $s \leq t$ **then**
- 4: mid $\leftarrow \lfloor (s+t)/2 \rfloor$
- 5: **if** $\mathbf{q}[\text{mid}] \leq \mathbf{k}[x][1]$ **then**
- 6: **for** $i := s$ to mid in parallel **do**
- 7: $C[i] \leftarrow j$ \triangleright Step executed in parallel by P_i
- 8: pbs-key-values ($\mathbf{q}[\text{mid}+1 \dots t]$, $\mathbf{k}[x \dots y]$)
- 9: **else**
- 10: **if** $\mathbf{q}[\text{mid}] > \mathbf{k}[y][1]$ **then**
- 11: **for** $i := \text{mid}$ to t in parallel **do**
- 12: $C[i] \leftarrow y+1$ \triangleright Step executed in parallel by P_i
- 13: pbs-key-values ($\mathbf{q}[s \dots \text{mid}-1]$, $\mathbf{k}[x \dots y]$)
- 14: **else** \triangleright $C[\text{mid}]$ is determined using sequential binary search
- 15: $z \leftarrow \min_{x \leq j \leq y} \{j \mid \mathbf{q}[\text{mid}] \leq \mathbf{k}[j][1]\}$
- 16: $C[\text{mid}] \leftarrow z$
- 17: **do** steps 18 and 19 in parallel
- 18: pbs-key-values ($\mathbf{q}[s \dots \text{mid}-1]$, $\mathbf{k}[x \dots z-1]$)
- 19: pbs-key-values ($\mathbf{q}[\text{mid}+1 \dots t]$, $\mathbf{k}[z \dots y]$)
- 20: **return** C .

Let Σ be the set $\{0, 1\}$ and denote the set of binary strings of size n as Σ^n . We define $\text{prefix}(\mathbf{x})$ for n -bit strings as the set of all initial substrings of $\mathbf{x} \in \Sigma^n$ which includes the empty string and \mathbf{x} itself. Next, let $\text{dec} : \{0, 1\}^n \rightarrow \mathbb{N}$ be the decimal representation of an n -bit string \mathbf{x} with $\mathbf{x}[0]$ denoting the least significant bit. We also use $\text{sort}(A)$ as a procedure that sorts an array A . Finally, wlog, we assume that N is a power of 2. We are now ready to present a parallel algorithm that solves the multiple-query associative recall problem below.

Algorithm 11 Parallel-MQAR ($\mathbf{u}[0 \dots N-1]$)

Input: Input sequence $\mathbf{u}[0 \dots N-1] \triangleq \{(\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i)\}_{i=0}^{\frac{N}{3}-1}$ with each $\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i \in \{0, 1\}^d$.

- 1: Initialize $\frac{N}{3} \log(\frac{N}{3})$ processors denoted $P_0, \dots, P_{\frac{N}{3} \log(\frac{N}{3})-1}$.
- 2: Initialize the index and output array \mathbf{idx} , $\mathbf{val} \leftarrow []$.
- 3: **for** $k := \{0, \dots, \log(\frac{N}{3})-1\}$ **do**
- 4: **for** $\mathbf{x} := \{\mathbf{x} \in \Sigma^{\log(\frac{N}{3})-k} \mid \mathbf{x}[\log(\frac{N}{3})-k-1] = 0\}$ **do**
- 5: *{All the steps below are executed in parallel by $\{\{\{P_i^{\mathbf{x}, k}\}_{\mathbf{x}}\}_{i \in [0, 2k-1]}\}_k$ }*
- 6: $I_k^{\mathbf{x}} \leftarrow \{\mathbf{y} \in \Sigma^{\log(\frac{N}{3})} \mid \mathbf{x} \in \text{prefix}(\mathbf{y})\}$.
- 7: $\mathbf{k}_{\text{sorted}}, I_{\text{permuted}}^{\mathbf{x}} \leftarrow \text{sort}(\{\mathbf{k}_{\text{dec}(i)}\}_{i \in I_k^{\mathbf{x}}})$ ▷ $I_{\text{permuted}}^{\mathbf{x}} := \{(j, \text{dec}(i)) \mid \mathbf{k}_{\text{sorted}}[j] \equiv \mathbf{k}_{\text{dec}(i)}\}$
- 8: $\mathbf{x}[\log(\frac{N}{3})-k-1] \leftarrow 1$
- 9: $J_k^{\mathbf{x}} \leftarrow \{\mathbf{y} \in \Sigma^{\log(\frac{N}{3})} \mid \mathbf{x} \in \text{prefix}(\mathbf{y})\}$.
- 10: $\mathbf{q}_{\text{sorted}}, J_{\text{permuted}}^{\mathbf{x}} \leftarrow \text{sort}(\{\mathbf{q}_{\text{dec}(j)}\}_{j \in J_k^{\mathbf{x}}})$ ▷ $J_{\text{permuted}}^{\mathbf{x}} := \{(\text{dec}(j), k) \mid \mathbf{q}_{\text{sorted}}[k] \equiv \mathbf{q}_{\text{dec}(j)}\}$
- 11: $C_k \leftarrow \text{pbs-key-values}(\mathbf{q}_{\text{sorted}}, \mathbf{k}_{\text{sorted}}, 2^k, 2^k)$
- 12: **for** $j \in J_k^{\mathbf{x}}$ **do**
- 13: **if** $C_k[\text{dec}(j)] \neq 2^k$ **then**
- 14: $\mathbf{c}_j^{\mathbf{x}} \leftarrow C_k[J_{\text{permuted}}^{\mathbf{x}}(\text{dec}(j))]$
- 15: **if** $\mathbf{c}_j^{\mathbf{x}} \neq 2^k$ **then** ▷ cf. equation 12
- 16: Add $I_{\text{permuted}}^{\mathbf{x}}(\mathbf{c}_j^{\mathbf{x}})$ to $\mathbf{idx}[\text{dec}(j)]$.
- 17: **for** $i \in \{1, \dots, \frac{N}{3}-1\}$ **do**
- 18: *{Executed in parallel by P_i }*
- 19: **if** $\exists j \in \mathbf{idx}[i]$ **then**
- 20: Add \mathbf{v}_{j+1} to \mathbf{val}
- 21: **return** \mathbf{val} .

Remark H.29. In lines 7 and 10, we keep track of the sorted permutation of the indices of keys and queries, respectively. This helps us in the retrieval of the index of the matching key as in line 16.

H.7.5 Correctness and Complexity

Proposition H.30. *Algorithm 11 solves the multiple-query associative recall problem with work complexity $\mathcal{O}(Nd \cdot \log^2 N)$ and time $\mathcal{O}(d \cdot \log^2 N)$.*

Proof. The correctness of **pbs** implies the correctness of Algorithm 11 if we can show that, for each $1 \leq i < \frac{N}{3}$, we check for a match among the keys $\{\mathbf{k}_j\}_{j \in [i-1]}$. To this end, for each $1 \leq i < \frac{N}{3}$, let the set of all iterator indices associated with an index i be defined as $K_i \triangleq \{(k, x) \mid i \in J_k^{\mathbf{x}}\}$ with $J_k^{\mathbf{x}}$ as noted in line 9. Then, we define the corresponding set for keys as $\mathcal{I}_i \triangleq \bigcup_{(k, \mathbf{x}) \in K_i} I_k^{\mathbf{x}}$ with $I_k^{\mathbf{x}}$'s defined as in line 6. That is, for all the calls to **pbs-key-values** that i is part of (given by K_i) where the algorithm checks for a match among the keys in \mathcal{I}_i , it then suffices to show that $\mathcal{I}_i = [i-1]$.

Here, first note that if some index $j \in I_k^{\mathbf{x}} \subseteq \mathcal{I}_i$ for some $\mathbf{x} \in \Sigma^{\log(\frac{N}{3})-k}$, then, by definition, $\mathbf{x} \in \text{prefix}(\text{bin}(j))$. Here, let $\mathbf{x}^1 := \mathbf{x}|_{\mathbf{x}[\log(\frac{N}{3})-k]=1}$ where we set the $(\log(\frac{N}{3})-k)$ -th index of \mathbf{x} to be 1. Consequently, as we have $i \in J_k^{\mathbf{x}}$ for the same k and \mathbf{x} as in $I_k^{\mathbf{x}}$ (cf. line 6), we must have $\mathbf{x}^1 \in \text{prefix}(\text{bin}(i))$. Thus we get $j < i$, whence we can claim that $\mathcal{I}_i \subseteq [i-1]$.

For the other direction, for any i , let b denote the position of the most significant bit in $\text{bin}(i)$ which differs from $\text{bin}(j)$ for any $j \in [i-1]$. Then, there must exist a binary string that is in the prefix set of both $\text{bin}(i)$ and $\text{bin}(j)$. That is, there exists $\mathbf{x} \in \text{prefix}(\text{bin}(i)) \cap \text{prefix}(\text{bin}(j))$ with $\mathbf{x} \in \Sigma^b$. Thus, we then must have $\text{bin}(j) \in I_{\log(\frac{N}{3})-b}^{\mathbf{x}}$ and $\text{bin}(i) \in I_{\log(\frac{N}{3})-b}^{\mathbf{x}}$ with \mathbf{x} as the corresponding witness. Hence, we have $[i-1] \subseteq \mathcal{I}_i$.

Overall, we have shown that $\mathcal{I}_i = [i-1]$. Since this holds for all $1 \leq i < \frac{N}{3}$, we can conclude that Algorithm 11 solves the multiple-query associative recall problem.

Next, it is easy to see that we execute lines 6 to 16 $\sum_{k=0}^{\log(\frac{N}{3})-1} \frac{\frac{N}{3}}{2^{k+1}}$ -many times. We note that sorting n values each of size d can be done with work complexity $n \log n \cdot d$. We note that, at each instance, we are

sorting sorting 2^k values. Meanwhile, remembering 2^k sorted permutation of indices can be done in linear time using arrays. Moreover, each call to `pbs-key-values` has $n = m = 2^k$ which has work complexity $n \log m$. Finally, we know that the work complexity of lines 18 to 20 is $\mathcal{O}(N)$. Thus, the overall work complexity of Algorithm 11 is

$$d \cdot \sum_{k=0}^{\log(\frac{N}{3})-1} \frac{\frac{N}{3}}{2^{k+1}} \mathcal{O}(2^k \cdot \log 2^k) = d \cdot \mathcal{O}(N) \cdot \sum_{k=0}^{\log N/3-1} \mathcal{O}(k) = \mathcal{O}(Nd \cdot \log^2 N). \quad (13)$$

We will now analyze the depth of Algorithm 11. We know that the depth of computation for Algorithm 10 is $\mathcal{O}(\log n \log m)$ for input sizes. Moreover, we have $\mathcal{O}(1)$ depth for the computation in 18 to 20 as each entry in `idx` can have at most one entry. Since the nested for loops iterating over ks and the associated `xs` runs in parallel, the depth of Algorithm 11 is dominated by the largest depth among all calls to `pbs-key-values` and to `sort`. The largest such call to `pbs-key-values` is of size $n = m = 2^{\log(\frac{N}{3})-1} = N/6$ which yields a depth of $d \cdot \log^2 \frac{N}{3}$. Moreover, using sorting networks (Definition H.34), we know that the largest depth is for sorting $\frac{N}{6}$ values of size d given by $d \cdot \Theta(\log N)$ (Lemma H.35). Thus, we can conclude that Algorithm 11 takes $\mathcal{O}(d \cdot \log^2 N)$, time where N is the length of the input. \square

H.7.6 Conversion to Arithmetic Circuit

We will convert Algorithm 11 to an arithmetic circuits modularly. In particular, after writing out an explicit circuit for Algorithm 10, we will use this circuit as a black-box along with circuits for sorting networks.

Circuit for `pbs-key-values`. We will denote the corresponding arithmetic circuit for Algorithm 10 as `pbs-key-values` as well with the input gates comprising of each entry from $\mathbf{q}[s \dots t]$ and $\mathbf{k}[x \dots y]$ and the i -th output gate yielding the value $C[i]$ as in Algorithm 10.

Here, we first convert the comparisons for the `if` statements in Algorithm 10. To this end, we briefly introduce comparators.

Definition H.31 (Comparators). A *comparator* is a device with inputs \mathbf{x} and \mathbf{y} and outputs \mathbf{x}' and \mathbf{y}' , that performs the following function:

$$\begin{aligned} \mathbf{x}' &= \min(\mathbf{x}, \mathbf{y}), \\ \mathbf{y}' &= \max(\mathbf{x}, \mathbf{y}). \end{aligned}$$

Using comparators, we can use bit-wise `XOR` and `AND` to define the result of the comparisons in lines 5 and 10 as the following fixed variables:

$$\begin{aligned} \ell_x &:= \mathbb{1}\{\mathbf{q}[\text{mid}] \leq \mathbf{k}_x\}, \\ g_y &:= \mathbb{1}\{\mathbf{q}[\text{mid}] > \mathbf{k}_y\}, \\ z &:= \text{bin-search}(\{\mathbf{k}_i\}_{i=s}^t, \mathbf{q}[\text{mid}]). \end{aligned} \quad (14)$$

This then allows us to infer the index of the key array for each recursive call to `pbs-key-values` in lines 8, 13, 18, and 19 from Algorithm 10. Specifically, let z_s and $z_t - 1$ denote the *starting and ending indices* for the keys as inputs to the recursive calls in Algorithm 10) below:

$$\text{pbs-key-values}(\mathbf{q}[\text{mid} + 1 \dots t], \mathbf{k}[z_t \dots y]); \quad (\text{lines 8 and 19}) \quad (15)$$

$$\text{pbs-key-values}(\mathbf{q}[s \dots \text{mid} - 1], \mathbf{k}[x \dots z_s - 1]); \quad (\text{lines 13 and 18}) \quad (16)$$

Here, z_t and z_s can assume values dependent on the results of the comparisons in lines 5 and 10. Specifically, we have

$$\begin{aligned} z_t &= \ell_x \cdot x + (1 - \ell_x)(1 - g_y) \cdot z = \begin{cases} x & \text{if } \mathbf{q}[\text{mid}] \leq \mathbf{k}_x \text{ (line 8)} \\ z & \text{if } \mathbf{q}[\text{mid}] \in (\mathbf{k}_x, \mathbf{k}_y] \text{ (line 19)}, \\ 0 & \text{otherwise} \end{cases} \\ z_s &= g_y \cdot (y + 1) + (1 - \ell_x)(1 - g_y) \cdot z = \begin{cases} y + 1 & \text{if } \mathbf{q}[\text{mid}] > \mathbf{k}_y \text{ (line 13)} \\ z & \text{if } \mathbf{q}[\text{mid}] \in (\mathbf{k}_x, \mathbf{k}_y] \text{ (line 18)}, \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Here, z_s or z_t getting a value 0 signifies that the branch is dead, and we do not execute the recursive call. Finally, let the arrays $C_t[\text{mid} + 1 \dots t]$ and $C_s[s \dots \text{mid} - 1]$ denote the outputs to the recursive calls in equation 15 and equation 16, respectively. We can then succinctly express the outputs for each index of the output array C as

$$C[i] = \begin{cases} \ell_x \cdot x + z_s \cdot (C_1[i]) & i \in [s \dots \text{mid} - 1] \\ g_y \cdot (y + 1) + z_t \cdot (C_2[i]) & i \in [\text{mid} + 1 \dots t] \\ \ell_x \cdot x + (1 - \ell_x)(1 - g_y) \cdot z + g_y \cdot (y + 1) & i = \text{mid} \end{cases} \quad (17)$$

We can thus state the circuit schematically in Fig. 9.

Now, before accounting for the complexity of the circuit for Algorithm 10, we must first assert the complexity of the comparators that we use in Fig. 9.

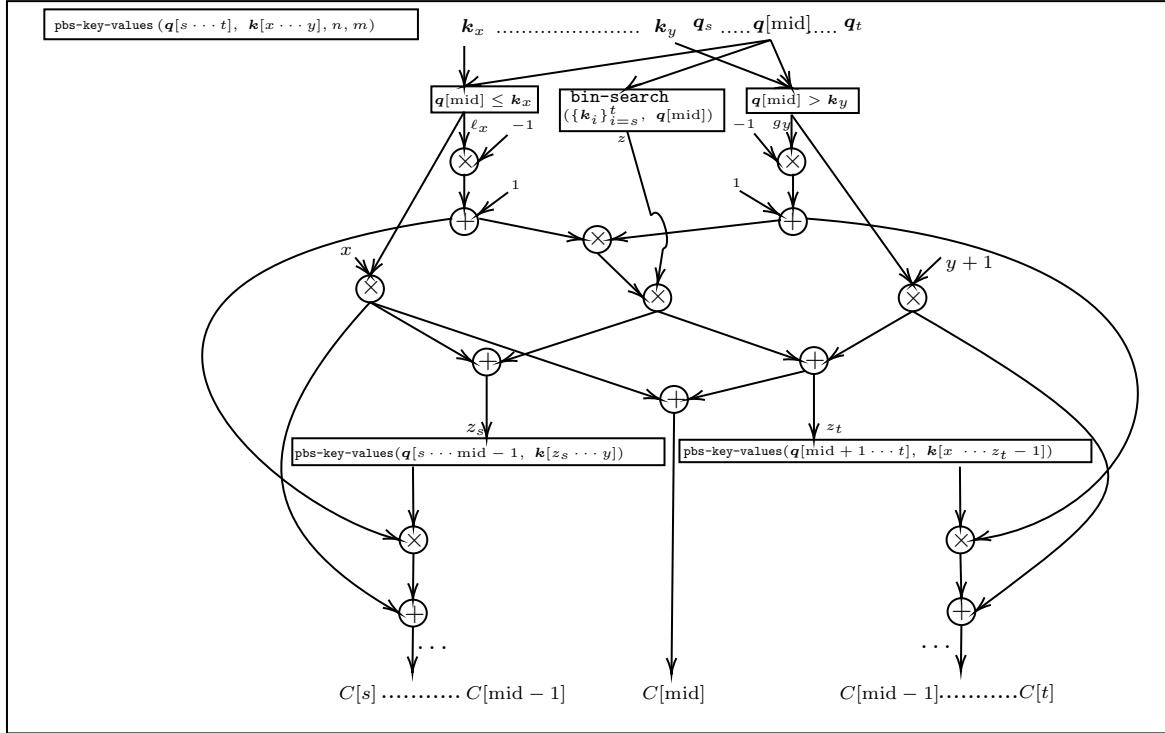


Figure 9: $\text{pbs-key-values}(\mathbf{q}[s \dots t], \mathbf{k}[x \dots y], n, m)$ as a circuit with recursive calls and subprocedures as “black boxes.”

Lemma H.32 (Cormen et al. [2022]). *For binary strings $\mathbf{x}, \mathbf{y} \in \Sigma^d$ of length d , there exists a comparison network of size $\mathcal{O}(d)$, width $\mathcal{O}(d)$, and depth $\mathcal{O}(\log d)$.*

We use this lemma to compute the complexity of the arithmetic circuit from Fig. 9.

Proposition H.33. *There exists an $(\mathcal{O}((n+m)d), \mathcal{O}(nd \cdot (\log m + \log n)), \mathcal{O}(\log n(\log m + \log n \log d)), \mathcal{O}(nd))$ -arithmetic circuit¹⁴ equivalent to pbs-key-values (Algorithm 10) with inputs \mathbf{q} and \mathbf{k} of lengths n and m with d -bit entries.*

Proof. The size of the circuit for pbs-key-values should equal the work-complexity of Algorithm 10 but we also need to account for the comparison gates in equation 14. Further, the circuit for binary search also has a size of $\mathcal{O}(d \cdot n)$ instead of $\mathcal{O}(d \cdot \log n)$ work as in Algorithm 10. Using Lemma H.32, along with the fact that the comparison gates and the binary search are used at most $\mathcal{O}(\log n)$ times, we deduce that we are adding $\mathcal{O}(nd \log n)$ size to the circuit in addition to the work complexity of the parallel algorithm. Thus, the overall size of the arithmetic circuit for pbs-key-values is $\mathcal{O}(dn \log m + nd \log n)$. Further, the depth of the circuit

¹⁴Recall that a (n, s, Δ, w) -arithmetic circuit is an n -variate circuit with size s , depth at most Δ , and width w .

here is determined by the runtime of Algorithm 10 along with the depth of the comparison gates and binary search $\mathcal{O}(\log n \log d)$, and finally, at most n processors are used for the parallel algorithm in Algorithm 10, which yields the width of the circuit. \square

Circuit for Parallel-MQAR. Now, we will call the circuit for Parallel-MQAR with the same name while the input gates contain the inputs of Algorithm 11. Indeed, we can directly “translate” Algorithm 11 to an arithmetic circuit as the values for $I_k^{\mathbf{x}}$ and $I_k^{\mathbf{p}}$ for each \mathbf{x} and k are predetermined from N . Thus, we start by placing the corresponding sorting networks which feeds into the **pbs-key-values** ($\mathbf{q}[s \dots t]$, $\mathbf{k}[x \dots y]$, n, m) circuit for Algorithm 10 in Fig. 9 so that the output values from the calls to **pbs-key-values** result in the checks as in line 16 of Algorithm 11. That is, we get outputs $C_k[J_{\text{permuted}}^{k\mathbf{x}}(\text{dec}(i))]$ from each call to **pbs-key-values**. We can then use a comparison gate to check if this value equals 2^k , and if not, we have found a match $C_k[J_{\text{permuted}}^{k\mathbf{x}}(\text{dec}(i))]$ for the query \mathbf{q}_i which results in the output of the associated value $\mathbf{v}_{C_k[J_{\text{permuted}}^{k\mathbf{x}}(\text{dec}(i))]+1}$, exactly as in Algorithm 11. That is, we first define the following variable as the output of the comparison gate:

$$\mathbf{c}_{\text{dec}(i)}^k := \mathbb{1}\{C_k[J_{\text{permuted}}^{k\mathbf{x}}(\text{dec}(i))] \neq 2^k\}. \quad (18)$$

Here, as $C_k[J_{\text{permuted}}^{k\mathbf{x}}(\text{dec}(i))] \neq 2^k$ implies that $I_{\text{permuted}}^{k\mathbf{x}}(C_k[J_{\text{permuted}}^{k\mathbf{x}}(\text{dec}(i))])$ equals the index of the matching key \mathbf{k}_j corresponding to the query \mathbf{q}_i , the i th output is then simply given by $\mathbf{c}_{\text{dec}(i)}^k \cdot I_{\text{permuted}}^{k\mathbf{x}}(C_k[J_{\text{permuted}}^{k\mathbf{x}}(\text{dec}(i))])$, where the 0 output implies that there does not exist a matching key.

Here, we also briefly introduce the the sorting networks that we use to sort the keys and queries:

Definition H.34 (Informal). *Sorting networks* are circuits with gates and wires where the gates of the circuit are comparators (Definition H.31) connecting two wires. Each such circuit can perform sorting on a fixed number of values.

We can then show the circuit schematically as in Fig. 10.

We now delineate the complexity of the circuit, starting with the complexity of the sorting networks.

Lemma H.35 (Ajtai et al. [1983]). *Let A be an array with d -bit entries of size n . Then, one can implement a sorting network to sort the array A with size $\mathcal{O}(d \cdot n \log n)$ and depth $\mathcal{O}(\log d \log n)$.*

Proposition H.36. *There exists an $(N \cdot d, \mathcal{O}(Nd \cdot \log^2 N), \mathcal{O}(\log d \log^2 N), \mathcal{O}(Nd \log N))$ -arithmetic circuit that solves the multiple-query associative recall problem.*

Proof. We note here that for each k , there are $\frac{N}{2^{k+1}}$ parallel problems of size 2^k for both the sorting networks and the **pbs – key – values** circuit. Using Lemma H.35, the cumulative size of these sorting networks is $\mathcal{O}(d \cdot N \log^2 N)$ (see equation 13) with overall depth $\mathcal{O}(\log d \log N)$.

Similarly, the next layer again runs $\sum_{k=0}^{\log \frac{N}{3}-1} \frac{N}{2^{k+1}}$ -many circuits for **pbs – key – values** each of which has size $\mathcal{O}(2^k d (\log 2^k + \log 2^k)) = \mathcal{O}(d \cdot 2^k \log 2^k)$, depth $\mathcal{O}(\log^2 2^k \log d)$ and width $\mathcal{O}(2^k)$ (Proposition H.33). Again, the cumulative size of this layer is given by $\mathcal{O}(Nd \cdot \log^2 N)$ (see equation 13). Since we run each of these circuits in parallel, the depth of this layer is again $\mathcal{O}(\log d \log^2(N))$ while the width is $\mathcal{O}(N \cdot \log N)$. Finally, we perform $\frac{N}{3} \log \frac{N}{3}$ comparisons at the end of d -bit strings in parallel which results in size $\mathcal{O}(N \log N \cdot d)$, depth $\mathcal{O}(\log d)$ and width $\mathcal{O}(N \log N \cdot d)$ (Lemma H.32). Therefore, the resulting arithmetic circuit has size $\mathcal{O}(d \cdot N \log^2 N + Nd \cdot \log^2 N + N \log N \cdot d) = \mathcal{O}(Nd \log^2 N)$, depth $\mathcal{O}(\log d \log^2 N)$ and width $\mathcal{O}(Nd \log N)$. \square

H.7.7 The Resulting BaseConv Model

As we have an arithmetic circuit for solving the multiple-query associative recall problem, we can now invoke Theorem H.21 to claim that there is a corresponding **BASECONV** model that solves the multiple-query associative recall problem with $\tilde{\mathcal{O}}(N \log c)$ parameters and $\tilde{\mathcal{O}}(1)$ layers.

Theorem H.37. *There exists a $(Nd, \tilde{\mathcal{O}}(1), \tilde{\mathcal{O}}(1), \tilde{\mathcal{O}}(Nd), \tilde{\mathcal{O}}(1))$ – **BASECONV** solves the multiple-query associative recall problem.*

Proof. Directly applying Theorem H.21 yields a **BASECONV** model with the number of layers $\mathcal{O}(\log d \cdot \log^2 N \cdot \log Nd \log N) = \mathcal{O}(1)$ layers while the claim on the input and inner dimensions follow trivially. \square

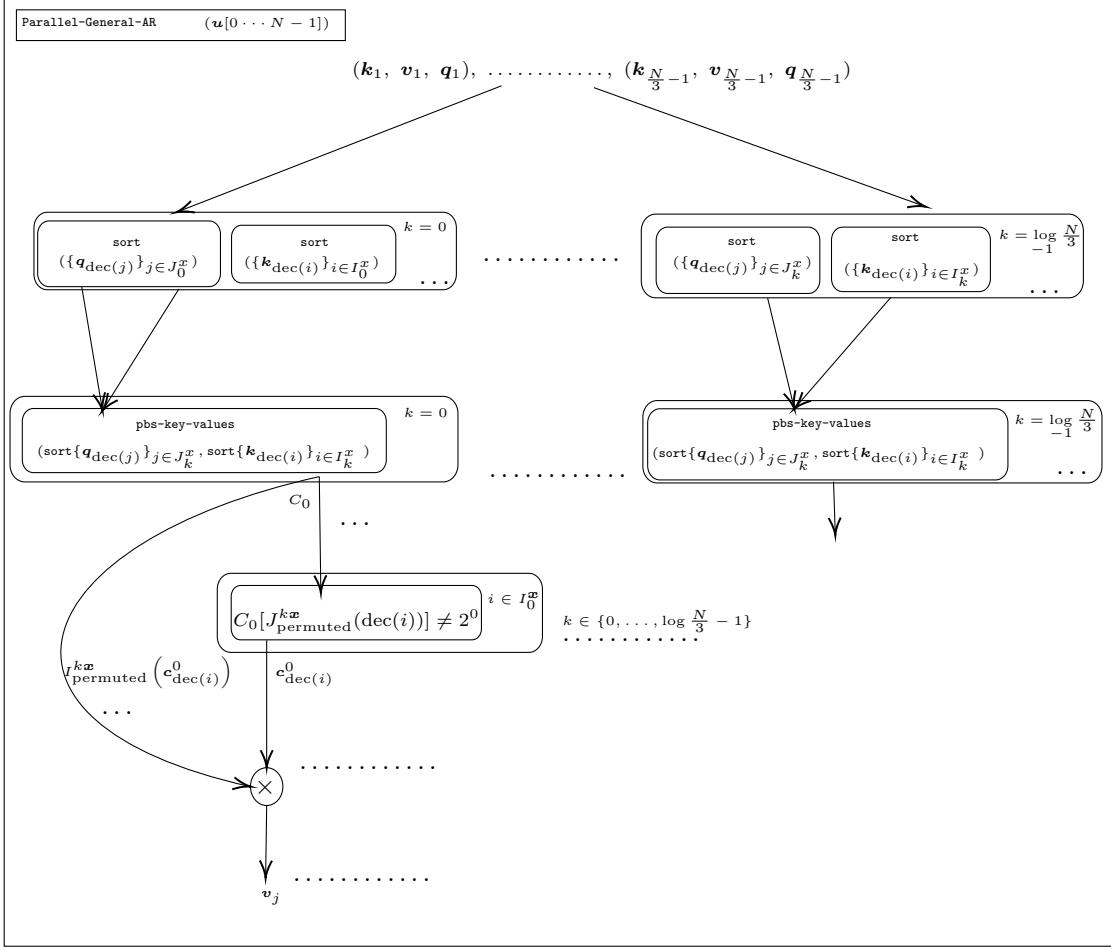


Figure 10: $\text{Parallel-MQAR}(u[0 \cdots N - 1])$ as a circuit that includes sorting networks and the circuit for pbs-key-values as subroutines.

H.8 Data-Dependent Convolutions

H.8.1 Introduction

In this section, we are again concerned with solving the multiple-query associative recall problem (Appendix H.7.1). However, in contrast to Appendix H.7.4, which yields a circuit that is unchanged and works for *all* inputs, we instead take the viewpoint of adapting the sequence mixing weights (Section 2) with respect to the particular sequence that the model receives as input. More specifically, we take the distance between the tokens in the sequence as a measure for designing data-dependent convolutions.

Setup. To formally setup the problem, as in our discussion of designing a parallel algorithm, we consider the following problem description of the multiple-query associative recall problem.

Suppose we are given an input $\mathbf{u}[0 \cdots N - 1] \triangleq \{(\mathbf{k}_0, \mathbf{v}_0, \mathbf{q}_0), \dots, (\mathbf{k}_{\frac{N}{3}-1}, \mathbf{v}_{\frac{N}{3}-1}, \mathbf{q}_{\frac{N}{3}-1})\}$ with each $\mathbf{k}_i, \mathbf{v}_i, \mathbf{q}_i \in C$. Here, each token is embedded using the standard one-hot encoding in $\{0, 1\}^c$ (i.e. we assume $d = c$).¹⁵ Our goal is again to check, for each $1 \leq i \leq \frac{N}{3} - 1$, whether there exists $0 \leq j < i$ such that $\mathbf{q}_i \equiv \mathbf{k}_j$, and if so, output \mathbf{v}_j .

Here, we define the interaction distance between the i th query \mathbf{q}_i and the matching key \mathbf{k}_j as $i - j$. We then also assume that *number of distinct interaction distances* is bounded by t .

¹⁵Our arguments do need $c = d$. However we do not need $d = N$ but we made this simplification for ease of presentation.

We can identify these distances using an autocorrelation [Chatfield, 1995], which has an elegant underlying formulation. We will briefly introduce the relevant mathematical machinery in the context of elucidating the data-dependent model that we seek to develop in the sequel.

Autocorrelations We introduce autocorrelation convolutions. Let $\tilde{\mathbf{u}}[t] := \mathbf{u}[-t]$, then the cross correlation of two vectors \mathbf{u} and \mathbf{v} is given by

$$\mathbf{u} \star \mathbf{v} \triangleq \tilde{\mathbf{u}} * \mathbf{v}.$$

The *autocorrelation* of a vector $\mathbf{u} \in \mathbb{R}^n$ is the cross correlation of \mathbf{u} with itself. Moreover, in terms of polynomials, we have $\tilde{\mathbf{u}}(X) = X^{n-1} \cdot \mathbf{u}(1/X)$. Thus, in analogy with our interpretation of convolution in terms of polynomial multiplication, we characterize the autocorrelation of a vector $\mathbf{u} \in \mathbb{R}^n$, given by $\mathbf{w} \in \mathbb{R}^n$ as follows:

$$\mathbf{w} = \text{coeff}(\mathbf{u}(X) \cdot \tilde{\mathbf{u}}(X) \bmod X^n - 1).$$

H.8.2 BaseConv with kernels generated using Auto-Correlation

We are now ready to describe the model that solves the multiple-query associative recall problem using data-dependent kernels derived using auto-correlations.

The Input-Dependent Kernels. There are several potential strategies to identify the best token-interaction distances for an input using a convolution. Here we will focus on an autocorrelation convolution for exposition. Autocorrelation allows us to identify the top t distinct shifts of the sequence that result in highly overlapping values (e.g. matches between query and keys in our associative recall setting). We can then construct convolution filters that perform each of these t shifts. That is, we define a function Top such that $\text{Top}(\mathbf{u} \star \mathbf{u}, t)$ returns a list of the top t shifts $\{s_\ell\}_{\ell \in [t]}$. We then use these top t distances $\{s_\ell\}_{\ell \in [t]}$ to define the following two kernels:

$$\begin{aligned} \mathbf{h}^k(X) &\equiv \sum_{\ell \in [t]} X^{s_\ell + (\ell-1) \cdot \frac{N}{3}}, \\ \mathbf{h}^v(X) &\equiv \sum_{\ell \in [t]} X^{s_\ell - 1 + (\ell-1) \cdot \frac{N}{3}}. \end{aligned} \tag{19}$$

Here, we note that we only have two kernels as we will assume $N' = t \frac{N}{3}$ and the shift will be done in "parallel." Obviously, one can instead define t distinct shift kernels but then there is a cost of $\mathcal{O}(t)$ in the number of layers.

Projections. We define the following projections $\mathbf{K}, \mathbf{Q}, \mathbf{V} \in \{0, 1\}^{N \times d}$ of the input that we shall use below using equation 9.

$$\begin{aligned} \mathbf{K}[i, :] &:= \begin{cases} \mathbf{u}[i, :] & \text{if } i \in \mathcal{K}, \\ \mathbf{0}^d & \text{otherwise,} \end{cases} \\ \mathbf{Q}[i, :] &:= \begin{cases} \mathbf{u}[i, :] & \text{if } i \in \mathcal{Q}, \\ \mathbf{0}^d & \text{otherwise,} \end{cases} \\ \mathbf{V}[i, :] &:= \begin{cases} \mathbf{u}[i, :] & \text{if } i \in \mathcal{V}, \\ \mathbf{0}^d & \text{otherwise,} \end{cases} \end{aligned} \tag{20}$$

Finally, we present the BASECONV model that solves the multiple-query associative recall problem with input-dependent kernels using $\mathcal{O}(1)$ layer and $\mathcal{O}(t \cdot Nd)$ -many parameters.

Theorem H.38. *There exists a BASECONV model with gated data-dependent convolutions that solves the multiple-query associative recall problem on inputs from $\{0, 1\}^{3N \times c}$ with the total number of distinct interaction distances bounded by t in $\mathcal{O}(1)$ layers and $\mathcal{O}(t \cdot Nc)$ total parameters.*

Proof. We note that we have the input dimension $d = c$. Now, for any input sequence $\mathbf{u} \in \{0, 1\}^{N \times d}$, we get the input-dependent kernels as in equation 19 using autocorrelation of the input. We will now outline the

following computations for the BASECONV layers:

$$\begin{aligned} \mathbf{y} &= \text{Linear}_{\mathbf{Q}}(\mathbf{u}) \odot (\mathbf{h}^K * \text{Linear}_{\mathbf{K}}(\mathbf{u})) \\ &= \mathbf{Q} \odot (\mathbf{h}^K * \mathbf{K}) \end{aligned} \quad (21)$$

$$\begin{aligned} \mathbf{z} &= \text{Linear}_{\mathbf{E}}(\mathbf{y}) \odot (\mathbf{h}^V * \text{Linear}_{\mathbf{V}}(\mathbf{u})) \\ &= \mathbf{E} \odot (\mathbf{h}^V * \mathbf{V}), \end{aligned} \quad (22)$$

where we have the linear projections $\text{Linear}_{\mathbf{Q}}(\mathbf{u}) = \mathbf{Q}$, $\text{Linear}_{\mathbf{K}}(\mathbf{u}) = \mathbf{K}$, $\text{Linear}_{\mathbf{V}}(\mathbf{u}) = \mathbf{V}$ and $\text{Linear}_{\mathbf{E}}(\mathbf{y}) = \mathbf{E}$ defined as

$$\mathbf{E}[i, :] := \text{Linear}_{\mathbf{E}}(\mathbf{y})[i, :] = \begin{cases} \mathbf{1}^d & \text{if } \exists j \in [d] \text{ such that } \mathbf{y}[i, j] = 1 \\ \mathbf{0}^d & \text{otherwise} \end{cases}. \quad (23)$$

Here, we will first present the argument for the special case when we have $t = 1$ as that will help us elucidate the general case. To this end, as the kernels from equation 19 for $t = 1$ are given by

$$\begin{aligned} \mathbf{h}^k(X) &\equiv X^{s_1}; \\ \mathbf{h}^v(X) &\equiv X^{s_1-1}, \end{aligned} \quad (24)$$

we observe that convolving with these kernels $\mathbf{h} * \mathbf{y}$ is equivalent to operating with the following primitives (Appendix H.3):

$$\begin{aligned} &\text{shift_down}(\mathbf{y}, s_1); \\ &\text{shift_down}(\mathbf{y}, s_1 - 1). \end{aligned} \quad (25)$$

We note that we shift down instead of shifting up as the index of the top-left entry is $(0, 0)$. We can then write down the computations performed in equation 21 and equation 22 as follows:

$$\mathbf{y} = \mathbf{Q} \odot \text{shift_down}(\mathbf{K}, s_1) \quad (26)$$

$$\mathbf{z} = \mathbf{E} \odot \text{shift_down}(\mathbf{V}, s_1 - 1), \quad (27)$$

We begin by examining \mathbf{y} below:

$$\begin{aligned} \mathbf{y}[i, :] &= (\mathbf{Q} \odot \text{shift_down}(\mathbf{K}, s_1))[i, :] \\ &= \mathbf{Q}[i, :] \odot \text{shift_down}(\mathbf{K}, s_1)[i, :] \end{aligned} \quad (28)$$

$$= \left(\begin{cases} \mathbf{u}[i, :] & \text{if } i \in \mathcal{Q} \\ \mathbf{0}^d & \text{otherwise} \end{cases} \right) \odot \left(\begin{cases} \mathbf{u}[i - s_1, :] & \text{if } i - s_1 \in \mathcal{K}, \\ \mathbf{0}^d & \text{otherwise} \end{cases} \right) \quad (29)$$

$$= \begin{cases} \mathbf{u}[i, :] \odot \mathbf{u}[i - s_1, :] & \text{if } i \in \mathcal{Q} \text{ and } i - s_1 \in \mathcal{K}, \\ \mathbf{0}^d & \text{otherwise} \end{cases}$$

Here, we use the fact that the Hadamard product is row-independent in equation 28, and the definitions of the projections from equation 20 in equation 29. Examining the j th entry, we get

$$\mathbf{u}[i, j] \odot \mathbf{u}[i - s_1, j] = \begin{cases} 1 & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K} \text{ and } \mathbf{q}_i \equiv \mathbf{k}_{i-s_1} \equiv \mathbf{e}_j \\ 0 & \text{otherwise.} \end{cases}$$

That is, we can express

$$\mathbf{y}[i, :] = \begin{cases} \mathbf{e}_j & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K} \text{ and } \mathbf{q}_i \equiv \mathbf{k}_{i-s_1} \equiv \mathbf{e}_j \\ \mathbf{0}^d & \text{otherwise} \end{cases}. \quad (30)$$

Consequently, as per the definition in equation 23, we get

$$\mathbf{E}[i, :] = \begin{cases} \mathbf{1}^d & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K} \text{ and } \mathbf{q}_i \equiv \mathbf{k}_{i-s_1} \\ \mathbf{0}^d & \text{otherwise} \end{cases} \quad (31)$$

We can now finally specify the output \mathbf{z} from equation 27 as follows:

$$\begin{aligned}\mathbf{z}[i,:] &= (\mathbf{E} \odot \text{shift_down}(\mathbf{V}, s_1 - 1)) [i,:] \\ &= \mathbf{E}[i,:] \odot \text{shift_down}(\mathbf{V}, s_1 - 1)[i,:]\end{aligned}\tag{32}$$

$$= \begin{cases} \mathbf{1}^d & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K} \text{ and } \mathbf{q}_i \equiv \mathbf{k}_{i-s_1} \\ \mathbf{0}^d & \text{otherwise} \end{cases} \odot \begin{cases} \mathbf{u}[i - s_1 + 1,:] & \text{if } i - s_1 + 1 \in \mathcal{V}, \\ \mathbf{0}^d & \text{otherwise} \end{cases} \tag{33}$$

$$\begin{aligned}&= \begin{cases} \mathbf{u}[i - s_1 + 1,:] & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K}, i - s_1 + 1 \in \mathcal{V} \text{ and } \mathbf{q}_i \equiv \mathbf{k}_{i-s_1} \\ \mathbf{0}^d & \text{otherwise} \end{cases} \\ &= \begin{cases} \mathbf{v}_{i-s_1} & \text{if } \mathbf{q}_i \equiv \mathbf{k}_{i-s_1} \\ \mathbf{0}^d & \text{otherwise} \end{cases}\end{aligned}\tag{34}$$

Again, we use the fact that the Hadamard product is row-independent in equation 32, and the definitions of the projections from equation 20 in equation 33. Overall, we have solved associative recall for all queries that have interaction distance exactly equal to s_1 .

In order to generalize this to arbitrary $t \leq \frac{N}{3}$, we first increase the internal dimension so that the input to the kernels $\mathbf{u}' \in \mathbb{R}^{(N \cdot t) \times d}$ in equation 19 and the projections $\mathbf{K}', \mathbf{Q}', \mathbf{V}' \in \mathbb{R}^{(N \cdot t) \times d}$ are given by

$$\mathbf{u}' \equiv \begin{pmatrix} \mathbf{0}^d \\ \vdots \\ \mathbf{0}^d \\ \hline \mathbf{u} \end{pmatrix}, \mathbf{K}' \equiv \begin{pmatrix} \mathbf{0}^d \\ \vdots \\ \mathbf{0}^d \\ \hline \mathbf{K} \end{pmatrix}, \mathbf{Q}' \equiv \begin{pmatrix} \mathbf{Q} \\ \vdots \\ \mathbf{Q} \\ \hline \mathbf{Q} \end{pmatrix}, \mathbf{V}' \equiv \begin{pmatrix} \mathbf{0}^d \\ \vdots \\ \mathbf{0}^d \\ \hline \mathbf{V} \end{pmatrix},$$

We then observe that for $\mathbf{h}_\ell^k(X) := X^{s_\ell}$ and $\mathbf{h}_\ell^v(X) := X^{s_\ell - 1}$, we have

$$\begin{aligned}\mathbf{h}^k(X) &\equiv \sum_{\ell \in [t]} \mathbf{h}_\ell^k(X) \cdot X^{(\ell-1) \cdot \frac{N}{3}}, \\ \mathbf{h}^v(X) &\equiv \sum_{\ell \in [t]} \mathbf{h}_\ell^v(X) \cdot X^{(\ell-1) \cdot \frac{N}{3}}.\end{aligned}$$

In analogy with equation 25, we can then equivalently write

$$\begin{aligned}
(\mathbf{h}^K * \mathbf{K}) &\equiv \begin{pmatrix} \mathbf{h}_t^K \\ \vdots \\ \mathbf{h}_2^K \\ \mathbf{h}_1^K \end{pmatrix} * \begin{pmatrix} \mathbf{0}^d \\ \vdots \\ \mathbf{0}^d \\ \mathbf{K} \end{pmatrix} \\
&\equiv \begin{pmatrix} \mathbf{h}_t^K * \mathbf{K} \\ \vdots \\ \mathbf{h}_2^K * \mathbf{K} \\ \mathbf{h}_1^K * \mathbf{K} \end{pmatrix} \\
&\equiv \begin{pmatrix} \text{shift_down}(\mathbf{K}, s_t) \\ \vdots \\ \text{shift_down}(\mathbf{K}, s_2) \\ \text{shift_down}(\mathbf{K}, s_1) \end{pmatrix}.
\end{aligned}$$

Similarly, we also have

$$(\mathbf{h}^V * \mathbf{V}') \equiv \begin{pmatrix} \text{shift_down}(\mathbf{V}, s_t - 1) \\ \vdots \\ \text{shift_down}(\mathbf{V}, s_2 - 1) \\ \text{shift_down}(\mathbf{V}, s_1 - 1) \end{pmatrix}.$$

That is, the argument for $t = 1$ now applies to each of the t shifts as we now have (cf. equation 21)

$$\begin{aligned}
\mathbf{y}' &\equiv \mathbf{Q}' \odot (\mathbf{h}^V * \mathbf{V}) \\
&\equiv \begin{pmatrix} \mathbf{Q} \\ \vdots \\ \mathbf{Q} \\ \mathbf{Q} \end{pmatrix} \odot \begin{pmatrix} \text{shift_down}(\mathbf{V}, s_t - 1) \\ \vdots \\ \text{shift_down}(\mathbf{V}, s_2 - 1) \\ \text{shift_down}(\mathbf{V}, s_1 - 1) \end{pmatrix} \\
&\equiv \begin{pmatrix} \mathbf{Q} \odot \text{shift_down}(\mathbf{V}, s_t - 1) \\ \vdots \\ \mathbf{Q} \odot \text{shift_down}(\mathbf{V}, s_2 - 1) \\ \mathbf{Q} \odot \text{shift_down}(\mathbf{V}, s_1 - 1) \end{pmatrix} \\
&\equiv \begin{pmatrix} \mathbf{y}_t \\ \vdots \\ \mathbf{y}_2 \\ \mathbf{y}_1 \end{pmatrix},
\end{aligned}$$

where, for each $\ell \in [t]$, we have (cf. equation 30)

$$\mathbf{y}_\ell[i, :] \equiv \begin{cases} \mathbf{e}_j & \text{if } i \in \mathcal{Q}, i - s_\ell \in \mathcal{K} \text{ and } \mathbf{q}_i \equiv \mathbf{k}_{i-s_\ell} \equiv \mathbf{e}_j, \\ \mathbf{0}^d & \text{otherwise} \end{cases}.$$

We then analogously get \mathbf{E}' as follows:

$$\mathbf{E}' \equiv \text{Linear}_{\mathbf{E}}(\mathbf{y}') \equiv \begin{pmatrix} \text{Linear}_{\mathbf{E}}(\mathbf{y}_t) \\ \vdots \\ \text{Linear}_{\mathbf{E}}(\mathbf{y}_2) \\ \text{Linear}_{\mathbf{E}}(\mathbf{y}_1) \end{pmatrix} \equiv \begin{pmatrix} \mathbf{E}_t \\ \vdots \\ \mathbf{E}_2 \\ \mathbf{E}_1 \end{pmatrix},$$

where, for each $\ell \in [t]$, we have (cf. equation 31)

$$\mathbf{E}_\ell[i, :] = \begin{cases} \mathbf{1}^d & \text{if } i \in \mathcal{Q}, i - s_\ell \in \mathcal{K} \text{ and } \mathbf{q}_i \equiv \mathbf{k}_{i-s_\ell} \\ \mathbf{0}^d & \text{otherwise} \end{cases}$$

The output in the general case is then given by

$$\begin{aligned}
\mathbf{z}' &\equiv \mathbf{E}' \odot (\mathbf{h}^V * \mathbf{V}') \\
&\equiv \begin{pmatrix} \mathbf{E}_t \\ \vdots \\ \mathbf{E}_2 \\ \mathbf{E}_1 \end{pmatrix} \odot \begin{pmatrix} \text{shift_down}(\mathbf{V}, s_t - 1) \\ \vdots \\ \text{shift_down}(\mathbf{V}, s_2 - 1) \\ \text{shift_down}(\mathbf{V}, s_1 - 1) \end{pmatrix} \\
&\equiv \begin{pmatrix} \mathbf{E}_t \odot \text{shift_down}(\mathbf{V}, s_t - 1) \\ \vdots \\ \mathbf{E}_2 \odot \text{shift_down}(\mathbf{V}, s_2 - 1) \\ \mathbf{E}_1 \odot \text{shift_down}(\mathbf{V}, s_1 - 1) \end{pmatrix} \\
&\equiv \begin{pmatrix} \mathbf{z}_t \\ \vdots \\ \mathbf{z}_2 \\ \mathbf{z}_1 \end{pmatrix},
\end{aligned}$$

where, for each $\ell \in [t]$, we have (cf. equation 34)

$$\mathbf{z}_\ell[i, :] \equiv \begin{cases} \mathbf{v}_{i-s_\ell} & \text{if } \mathbf{q}_i \equiv \mathbf{k}_{i-s_\ell} \\ \mathbf{0}^d & \text{otherwise} \end{cases}$$

Finally, we define the last output layer to compute $\mathbf{z}_{out} \equiv \text{Linear}_{\text{sum}}(\mathbf{z}') \equiv \sum_{\ell \in [t]} \mathbf{z}_\ell$ so that we have

$$\mathbf{z}_{out}[i, :] \equiv \begin{cases} \mathbf{v}_{i-s_\ell} & \text{if } \mathbf{q}_i \equiv \mathbf{k}_{i-s_\ell} \text{ for some } \ell \in [t] \\ \mathbf{0}^d & \text{otherwise} \end{cases}$$

To recall, we retrieved the top t interaction distances of the input \mathbf{u} using auto-correlation and defined the corresponding convolution kernels (equation 19). We then shifted down the keys \mathbf{K} using the first kernel and gated with the corresponding queries \mathbf{Q} so that we got a match exactly when there exists a key that is at s_ℓ interaction distance from the corresponding query. After “smearing” this match to get \mathbf{E} , we used it as a mask to retrieve the value in the next layer. Overall, since we have t atomic kernels that perform t shifts with each of these kernels using $\mathcal{O}(Nd)$ parameters, we can conclude that the output solves the associative recall problem for all queries with exactly ℓ interaction distance from the corresponding keys for all $\ell \in [t]$ using $\mathcal{O}(1)$ layers and $\mathcal{O}(t \cdot Nc)$ parameters as we have $d = c$. \square

Table 6: Attention Training Settings

	73M	125M	360M
Optimizer	Adam		
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$		
Optimizer eps	$1e - 8$		
Precision	BFloat16		
Learning rate decay	Cosine		
Learning rate (min, base)	8e-5, 8e-4		
Global batch size	256		
Training iterations	20000		
Warmup Duration (Linear)	0.01		
Weight decay	0.1		
Num Layers	6	12	24
Hidden Size	704	768	1024
FFN Width		4	
Position Embeddings		Rotary	
Weight Tying		True	
Number of Heads (H)	8	12	16

Table 7: Attention FLOPs Computation

	Equation
Input Layer	$B \times V \times N \times D$
Sequence Mixer $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ Projections	$B \times N \times D \times D \times 3$
Sequence Mixer Attention	$B \times H \times H \times D + H \times N \times N + B \times N \times N \times D$
Sequence Mixer Output Projection	$B \times N \times D \times D$
Channel Mixer (FFN Width 4)	$B \times D \times D \times 8 \times \frac{2}{3} \times N$
Language Modeling Head	$B \times V \times N \times D$

Table 8: Hyena [Poli et al., 2023a] Training Settings

	72M	158M	358M
Optimizer	Adam		
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.98$		
Precision	BFloat16		
Learning rate decay	Cosine		
Learning rate (min, base)	(1e-5, 8e-4)		
Global batch size	256		
Training iterations	20000		
Warmup Duration (Linear)	0.01		
Weight decay	0.1		
Num Layers	8	18	24
Hidden Size	768	864	1024
FFN Width		2	
Position Embeddings		None	
Weight Tying		True	
Short Conv. Filter Size		3	
Exp. Mod. Decay (Fast, Slow)		0.3, 1.2	
Filter Sine Freq. (w)		14	
Filter Order		64	
Filter Inner MLP		2	
Filter Weight Decay		0	

Table 9: Hyena FLOPs Computation

	Equation
Input Layer	$B \times V \times N \times D$
Sequence Mixer Input Projection	$B \times N \times D \times D \times 3 + B \times N \times 9 \times D$
Sequence Mixer Long Convolution	$10 \times N \times \log(N) \times D \times B$
Sequence Mixer Short Convolution	$3 \times B \times N \times D$
Sequence Mixer Implicit MLP (Order 64)	$D \times 64$
Sequence Mixer Output Projection	$B \times N \times D \times D$
Channel Mixer (FFN Width 2)	$B \times D \times D \times 2 \times 2 \times N$
Language Modeling Head	$B \times V \times N \times D$

Table 10: H3 [Fu et al., 2023c] Training Settings

	72M	168M	357M
Optimizer	Adam		
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$		
Precision	BFloat16		
Learning rate decay	Cosine		
Learning rate (min, base)	(8e-5, 8e-4)		
Global batch size	256		
Training iterations	20000		
Warmup Duration (Linear)	0.01		
Weight decay	0.1		
Weight Tying	True		
Num Layers	8	19	33
Hidden Size	720	864	1024
FFN Width	2		
Position Embeddings	None		
State Space Model State Size		64	

Table 11: RWKV Peng et al. [2023] Training Settings

	72M	169M	351M
Optimizer	Adam		
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.999$		
Optimizer eps	$1e-8$		
Precision	BFloat16		
Learning rate decay	Cosine		
Learning rate (min, base)	1e-5, 8e-4		
Global batch size	256		
Training iterations	20000		
Warmup Duration (Linear)	0.01		
Weight decay	0.1		
Weight Tying	False		
Num Layers	6	12	20
Hidden Size	624	768	984
Position Embeddings	None		
Initialization	From Reference Impl.		

Table 12: RetNet [Sun et al., 2023] Training Settings

152M	
Optimizer	Adam
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.98$
Optimizer eps	1.0e-8
Precision	BFloat16
Learning rate decay	Linear
Learning rate (min, base)	(1e-5, 8e-4)
Global batch size	256
Training iterations	20000
Warmup Duration (Linear)	0.01
Weight decay	0.05
Normalization	Layernorm
Weight Tying	True
Num Layers	12
Hidden Size	768
Value Hidden Size	1280
Window Size	128
Position Embeddings	xPos (relative position embeddings)
Initialization	DeepNet

Table 13: Simple Long Convolution Fu et al. [2023c] Training Settings

	76M	128M	360M
Optimizer	Adam		
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$		
Precision	BFloat16		
Learning rate decay	Linear		
Learning rate (min, base)	8e-5, 8e-4		
Global batch size	256		
Training iterations	20000		
Warmup Duration (Linear)	0.01		
Weight decay	0.1		
Num Layers	6	12	24
Hidden Size	704	864	1024
FFN Width	4		
Position Embeddings	-		
Weight Tying	True		
Channels	1		
Lam	0.001		
Kernel Dropout	0.1		
Kernel LR	5e - 5		
Activation	GeLU		
Exponential Modulation	True		

Table 14: Simple Long Convolution FLOPs Computation

Equation	
Input Layer	$B \times V \times N \times D$
Sequence Mixer Long Convolution	$10 \times N \times \log(N) \times D \times B$
Sequence Mixer Output Projection	$B \times N \times D \times D$
Channel Mixer (FFN Width 4)	$B \times D \times D \times 8 \times \frac{2}{3} \times N$
Language Modeling Head	$B \times V \times N \times D$

Table 15: BASECONV Training Settings

	168M	354M
Optimizer	Adam	
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$	
Precision	BFloat16	
Learning rate decay	Cosine	
Learning rate (min, base)	8e-5, 8e-4	
Global batch size	256	
Training iterations	20000	
Warmup Duration (Linear)	0.01	
Weight decay	0.1	
Num Layers	30	48
Hidden Size	852	1080
FFN Width	2	
Position Embeddings	-	
Weight Tying	True	
Short Conv. Filter Size	3	
Exp. Mod. Decay (Fast, Slow)	0.3, 1.2	
Filter Sine Freq. (w)	14	
Filter Order	64	
Filter Inner MLP	2	
Filter Weight Decay	0	

Table 16: BASECONV FLOPs Computation

Equation	
Input Layer	$B \times V \times N \times D$
Sequence Mixer Long Convolution	$10 \times N \times \log(N) \times 0.5(D) \times B$
Sequence Mixer Short Convolution	$B \times N \times 0.5(D)$
Sequence Mixer Implicit MLP (Order 64)	$0.5(D) \times 64$
Sequence Mixer Linear Projection	$B \times N \times D \times D$
Channel Mixer (FFN Width 2)	$B \times D \times D \times 2 \times 2 \times N$
Language Modeling Head	$B \times V \times N \times D$