Measuring and Improving Recall in Convolutional Language Models

Simran Arora*[1], Sabri Eyuboglu*[1], Aman Timalsina[3], Isys Johnson[2], Michael Poli[1], James Zou[1], Atri Rudra[2], and Christopher Ré[1]

[1]Stanford University, [2]University at Buffalo, [3]Purdue University

Abstract

Convolution-based language models are asymptotically more efficient than Transformers as sequence length grows and are increasingly competitive in quality. To better understand the quality differences between these architectures, we pretrain a suite of 14 language models across attention and convolution-based architectures, finding that the SoTA *gated convolution* architectures still underperform Transformers by up to 2.1 perplexity points on the Pile. Our analysis shows that a single language modeling capability, termed *associative recall (AR)* accounts for 76% of the perplexity gap on average. The task requires recalling an association from earlier in the context, e.g. *Hakuna Matata means no worries...Hakuna Matata it means no* $\rightarrow$ `??`. We show via experiments and theory that the associative recall solution encoded by convolution-based models is less *parameter efficient* than the one encoded by attention. The issue arises because convolution-based models process sequences using fixed filters that do not depend on the input data. Finally, we propose two methods for constructing input-dependent convolution filters and show theoretically and empirically that they can solve AR with improved parameter-efficiency relative to input-independent filters.

1 Introduction

Two recent advances have catalyzed excitement around convolution-based language models [9, 14, 19, 16, inter alia.]: (1) gating (*i.e.* element-wise multiplication) and (2) *long* convolutional filters (*i.e.* filters the length of the sequence) to enable interactions between distant tokens [8, 12]. Language models based on gated-convolutions support sub-quadratic training and inference, unlike attention.

However, it remains unclear how differences in these architectures affect a model's behavior. To study this, we pretrain and evaluate 14 language models across 3 scales and 5 architectures on the same data and infrastructure setup. In spite of the recent progress, we find there is still a perplexity gap of up to 2.1 points between state-of-the-art convolution-based architectures and Transformers in language modeling on the Pile (Table 1). Through fine-grained analysis, we find that a single simple issue is responsible for much of the gap: recalling an association seen earlier in-context. For example:

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

We find that errors on "AR Hits" (*e.g. worries* in the example above) account for 76% of the perplexity gap to Attention on average, despite only representing 6.4% of all tokens in the Pile dataset. Scaling gated convolutions to $4\times$ the parameters of a Transformer baseline fails to close the gap (Table 1).

The associative recall gap is particularly surprising given that prior work demonstrates how gated-convolutions can solve a version of the very task [9, 16]. However, in the *synthetic* version studied in

*Equal contribution, order determined by coin toss. Contact: `{simran, eyuboglu}@stanford.edu`.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

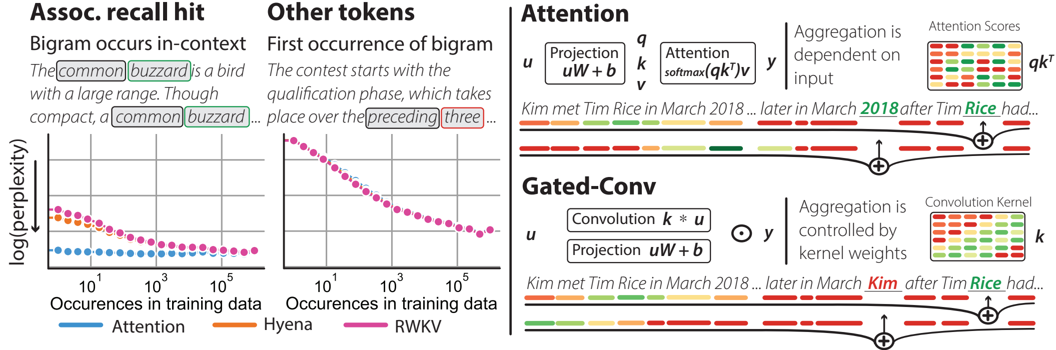


Figure 1: **The associative recall gap.** We stratify Pile validation data by whether or not the predicted token is an *associative recall hit* (see Section 2). We find the perplexity between gated convolutions and attention grows as an associative recall bigram was seen less frequently in the training data.

prior work, there is only one AR query per-example. Through our downstream analysis, we find this doesn't reflect the way AR manifests in real languages: e.g., above there are multiple AR hits in a single passage (*Matata* and *worries*). We formally study this Multi-query AR (MQAR) setting.

By analyzing MQAR, we provide the following explanation for the observed gap: the associative recall solution encoded by gated convolutions is less *parameter efficient* than the solution encoded by attention. To perform variable distance recalls (*e.g.* at a distance of 10 tokens for *Hakuna Matata* and 9 for *no worries*), convolution-based models require dimensionality to scale with sequence length.

To analyze the *class* of gated convolutions (beyond specific architectures like Hyena), we introduce a simple operator called BASECONV, which provably simulates any gated convolution architecture within poly-log factors (in number of parameters and layers). Using BASECONV as the canonical representation of gated convolutions, we prove the model dimension for BASECONV to solve MQAR grows with the input sequence length (Theorem 3.3) while attention can solve MQAR with model dimension independent of sequence length (Proposition 3.2). This scaling is undesirable. However, we show theoretically that if we extend BASECONV to include convolutions with filters that are dynamically defined as a function of the data (*input-dependent* convolutions), then the resulting class of architectures could solve MQAR with improved efficiency (Theorem 3.4).

In Section 3, we empirically validate these theoretical observations: (1) On MQAR, we show scaling the gated convolutions closes the MQAR gap to attention. In practice, with limited dimensionality, gated-convolutions appear to encode approximate solutions to MQAR that support only a subset of recall distances. (2) We propose using input-dependent convolutions and show they scale better than those with input-independent convolutions, matching attention with fewer parameters. To do this, we introduce two prototype approaches for constructing input-dependent convolutions (Figure 2).

Overall, we document fundamental differences between attention and gated-convolution language models. We prove and demonstrate that input-dependent convolutions can help close the gap.

2 Identifying the associative recall problem

In this section, we show that there is a perplexity gap between state-of-the-art gated convolutions and attention and identify that Associative Recall(AR) accounts for 76% of the quality gap on average.

2.1 Fine-grained analysis of downstream quality

Perplexity Gap We pretrain a suite of large language models across 3 different scales (70M-360M) for 10B tokens on the standard Pile language modeling setting [11]. We cover 4 different architectures (see Appendix A for experimental details). Across scales, we find that the Transformer outperforms the gated convolutions by at least half of a perplexity point: the minimum gaps are $+2.14$, $+0.63$, $+0.94$ PPL at 70M, 160M, and 360M parameter scales, respectively. We report overall test perplexity in the first column of Table 1. The extended results for 70M and 360M are in Table 2.

Associative Recall Perplexity We perform a fine-grained analysis of the next token predictions by dividing the overall Pile test split into two slices, AR and non-AR tokens, defined as follows:

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.

Model	Param (M)	TFLOPs	**Overall**	**Slices**		**% of gap due to**
				AR Hits	Other Tokens	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%
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%

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 2. Extended results in Table 2.

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 more than $1,000$ times during training.

In Table 1 and Table 2, we find that the AR slice accounts for 76% of the average perplexity gap between the gated convolutions and attention, measured as: $\frac{\Delta \log(\phi_{\text{AR}})\cdot|T_{\text{AR}}|}{\Delta \log(\phi)\cdot|T|}$, where ϕ is the perplexity and T is the set of tokens in the test set.

2.2 Defining the problem: Multi-Query Associative Recall

This gap in AR perplexity is surprising since gated convolution architectures (H3, Hyena [16, 9]) were explicitly evaluated on AR synthetic tasks in prior work and shown to match attention on the task. In our study, we find major properties of AR in real language that are missing from prior synthetic formulations. In real world inputs, the language model often needs to perform multiple associative recalls in a single forward pass, at different positions in the sequence. We refer to this as Multi-Query AR (MQAR). We formally define the MQAR problem as follows:

Definition 2.1 (Multi-Query-AR (MQAR))**.** Suppose we are given an input sequence $\boldsymbol{x} = \{x_0, \ldots, x_{N-1}\}$ where each $x_i \in C$ is a token drawn from a vocabulary of size $c = |C|$. The sequence consists of key, value, and query triplets $\boldsymbol{k}_i, \boldsymbol{v}_i, q_i \in C$ from: $\{(k_0, v_0, q_0), \ldots, (k_{N-1}, v_{N-1}, q_{N-1})\}$. The task is to check, for every query $\boldsymbol{q}_i$, whether there exists a key $\boldsymbol{k}_j$ at $0 \le j < i$ such that $\boldsymbol{q}_i \equiv \boldsymbol{k}_j$.

3 Explaining the associative recall problem

In this section, we analyze how the quality of gated convolution models on MQAR relates to the model's size, towards explaining the performance gaps discussed in Section 2.

3.1 BaseConv: a minimal gated convolution operator

First,we define a minimal gated-convolution architecture, called BaseConv, which we show can simulate a broad class of architectures built from gating and convolutions.

Definition 3.1 (BaseConv Operator)**.** Given an input $\boldsymbol{u} \in \mathbb{R}^{N\times d}$, layer ℓ of BaseConv is:

$$\boldsymbol{y} := \underbrace{\left(\boldsymbol{u}\cdot \boldsymbol{W}^{\ell} + \boldsymbol{b}_1^{\ell}\right)}_{\textbf{Linear Projection}} \odot \underbrace{\left(\boldsymbol{h}^{\ell} * \boldsymbol{u} + \boldsymbol{b}_2^{\ell}\right)}_{\textbf{Convolution}} \tag{1}$$

where the layer is parameterized by learnable filters $h \in \mathbb{R}^{N\times d}$, a linear projection $\boldsymbol{W}^{\ell} \in \mathbb{R}^{d\times d}$ that supports (near) linear time matrix-vector multiplication, and 'bias' matrices $\boldsymbol{b}_1, \boldsymbol{b}_2 \in \mathbb{R}^{N\times d}$. The $\odot$ is component-wise product (*i.e.* gating) and convolution of two matrices is computed column-wise.

We show any gated convolution model can be simulated by BaseConv with only a (poly)logarithmic blowup in parameters and layers (Theorem D.18 in Appendix D.5). Further, we observe that BaseConv and Hyena models can simulate each other with only a small constant blowup in parameters (Proposition D.9 in Appendix D.5). This suggests that simply re-combining the gating and convolution primitives does not change the representational capacity.

3.2 Theoretical analysis of gated convolution capacity and associative recall

In this section, we analyze the model complexity for attention and gated convolution architecture families to solve MQAR. First, we note attention can solve MQAR with number of parameters independent of sequence length (assuming $c^2 > N$, as it typically is in practice).

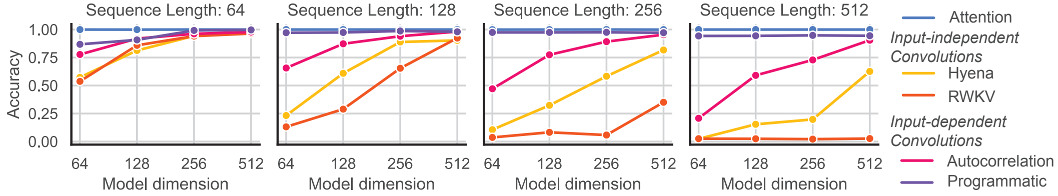


Figure 2: **Model dimension and accuracy on associative recall.** The x-axis is the model dimension and the y-axis is accuracy. Increasing the sequence length correlates with increased task difficulty.

Proposition 3.2 (Attention)**.** *Given an input* $\boldsymbol{u} \in \{0,1\}^{N\times c}$*, Attention solves* MQAR *for* $\boldsymbol{u}$ *using* $\mathcal{O}(\max(N, c^2))$ *parameters,* $\mathcal{O}(\min(Nc^2, N^2c))$ *time complexity and* $\mathcal{O}(1)$ *layers.*

Next, we prove that BASECONV can solve associative recall with number of parameters linear in sequence length and number of layers poly-logarithmic in sequence length. This improves upon the prior best known upper bound for gated convolutions to solve associative recall.

Theorem 3.3 (BASECONV)**.** *Given an input* $\boldsymbol{u} \in \{0,1\}^{N\times\mathcal{O}(\log c)}$ *to* MQAR *(where we assume that distinct tokens are embedded into vectors in* $\{0,1\}^{\mathcal{O}(\log c)}$*), there exists a* BASECONV *operator that solves* MQAR *for* $\boldsymbol{u}$ *using* $\tilde{\mathcal{O}}(N\log c)$ *parameters as well as time complexity and* $\tilde{\mathcal{O}}(1)$ *layers.*

Compared to attention's bounds, BASECONV's have a dependence on sequence length, which is undesirable for long sequence lengths encountered in practice. Thus, we analyze simple extensions of BASECONV that use *input-dependent* filters. In this class of models, the layer is defined as in Definition 3.1, except the convolutional filter is not a weight, but rather a function of the input itself: $\mathbf{h}^t = f(\mathbf{u})$. A natural choice for f is *autocorrelation*, a standard operation borrowed from signal processing where the input is convolved with itself [5]. 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 $\boldsymbol{q}_i$ and the matching key $\boldsymbol{k}_j$ as $i-j$. This allows us to present the upper bound for input-dependent mixing (Theorem D.29 in Appendix D.7).

Theorem 3.4 (Input-dependent BASECONV)**.** *Given an input* $\boldsymbol{u} \in \{0,1\}^{N\times c}$ *to* MQAR *(where we assume that the tokens are embedded as one-hot encoding in* $\{0,1\}^c$ *and there exists at most* t *distinct interaction distance), there exists a* BASECONV *operator that uses* data-dependent kernels *to solve the above case of* MQAR *using* $\mathcal{O}(t\cdot Nc)$ *parameters and* $\mathcal{O}(1)$ *layers.*

This improves upon the poly-logarithmic scaling incurred with input-independent convolutions.

3.3 Empirical analysis of gated convolution capacity and associative recall

In this section, we provide empirical evidence (1) that gated-convolution models (Hyena and RWKV) require larger model dimension than attention to solve MQAR on long sequences and (2) that introducing input-dependent convolutions enables convolution-only models to solve MQAR with improved parameter efficiency. Our results are summarized in Figure 2.

We train and evaluate models on synthetic MQAR with vocabulary size $8,192$, varying sequence lengths $\in \{32\text{ - }1,024\}$ and model dimension $\in \{64\text{-}1,024\}$. Appendix C provides further details on the formulation and construction of this synthetic task.

We find that the accuracy of gated-convolution models drops as we decrease the model dimension and increase the sequence length, while attention solves the task with near perfect accuracy at all sequence lengths. In Figure 2, we propose and validate two prototypes –autocorrelation and programmatic— for constructing input-dependent convolutions (defined in Appendix A) that demonstrate improved scaling vs. Hyena and RWKV. We hope these prototypes inspire future efficient architectures.

4 Conclusion

We (1) find a quality gap between efficient convolution and attention based architectures, finding 76.2% of the gap on average lies in tokens that require *recall*. (2) We show that the gap in associative recall performance is due to differences in the parameter-efficiency of the AR solutions of each architecture. (3) Finally, we present strategies for introducing input-dependent sequence aggregation. This work can guide future architectures that are both efficient and maintain high quality.

Acknowledgments

We would like to thank Albert Gu, Stefano Massaroli, Tri Dao, Michael Zhang, Dan Fu, Hermann Kumbong, Neel Guha, Kush Bhatia and Eric Nguyen for helpful discussions and feedback. We gratefully acknowledge the support of NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); US DEVCOM ARL under No. W911NF-21-2-0251 (Interactive Human-AI Teaming); ONR under No. N000141712266 (Unifying Weak Supervision); ONR N00014-20-1-2480: Understanding and Applying Non-Euclidean Geometry in Machine Learning; N000142012275 (NEPTUNE); NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, Google Cloud, Salesforce, Total, the HAI-GCP Cloud Credits for Research program, the Stanford Data Science Initiative (SDSI), National Science Foundation Graduate Research Fellowship, and members of the Stanford DAWN project: Facebook, Google, and VMWare. 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 NIH, ONR, or the U.S. Government.

References

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. An 0 (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, 1983. D.26

[2] Selim G Akl and Henk Meijer. Parallel binary search. *IEEE Transactions on Parallel & Distributed Systems*, 1(02):247–250, 1990. D.6.4, D.6.4

[3] 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. D.6.1

[4] P. Bürgisser, T. Lickteig, M. Clausen, and A. Shokrollahi. *Algebraic Complexity Theory*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 1996. D.10

[5] Chris Chatfield. The analysis of time series: An introduction, fifth edition. 1995. 3.2, D.7.1

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022. D.23

[7] 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. D.4, D.11, D.14, D.4, D.16

[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

[9] 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*, 2023. 1, 1, 2.2, C

[10] 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*, 2023. A.1, 8

[11] 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. 2.1

[12] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021. 1

[13] Michael T Heideman and C Sidney Burrus. *Multiplicative complexity, convolution, and the DFT*. Springer, 1988. D.1.1

[14] 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

[15] 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. A.1, 7

[16] 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*, 2023. 1, 1, 2.2, A.1, C, 3

[17] 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*, 2023. D.2.1, D.3

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

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

Appendix

Appendix A gives details for the experiments, including model architectures and hyperparameters. Appendix B provides additional details of the MQAR problem and synthetics used in our work. Appendix C provides additional details of the MQAR problem and synthetics used in our work. Appendix D gives proofs and additional discussion for the theoretical analysis in our work.

A Experimental Details

We use A100 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.

A.1 Models

We evaluate over 4 previously proposed architectures as well as BASECONV, the theoretically "canonical" representation for gated convolutions, introduced in Section 3, for a total of 14 training runs. Here we provide details on the hyperaparamters and configurations used for training each architecture. We also provide details on the FLOPs computation.

Notation: 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.

Hyena [16] We train and compute FLOPs using the specifications in Tables 3 and 4 respectively. The parameters are sourced from the Appendix of [16] and the implementation is sourced from the provided reference at https://github.com/HazyResearch/safari.

Attention We train and compute FLOPs using the the specifications in Tables 5 and Table 6 respectively. The parameters are sourced from the Transformer implementation in https://github.com/EleutherAI/gpt-neox.

RWKV [15] We train and compute FLOPs using the specifications in Table 7. The parameters are sourced from the Appendix of [15] and the details provided in the reference implementation at https://github.com/BlinkDL/RWKV-LM. We specifically focus on RWKV-V4. We compute FLOPs as in the Appendix of [15], based on the number of linear layer parameters, plus input and language modeling head FLOPs.

Long Convolution We train and compute FLOPs using the specifications in Tables 8 and 9 respectively. 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 initialziations/regularizations from recent work [10]. The implementation is provided at https://github.com/HazyResearch/safari.

BASECONV We train and compute FLOPs using the specifications in Tables 10 and 11 respectively.

A.2 Input-Dependence Implementations

The results in Section 3 highlight how the input-independent sequence aggregation of gated-convolutions is a fundamental limitation that affects their ability to solve associative recall. We theoretically show that introducing input-dependent aggregation into gated-convolution models can enable them to solve associative recall with improved parameter efficiency. In Figure 2, we show that input-dependent convolutions via autocorrelation or programmatic filters outperforms Hyena and RWKV across scales. Here we define the autocorrelation and programmatic filters baselines in additional detail.

First, we explore a simple convolution-only layer based on autocorrelation, a standard operation borrowed from signal processing where the input is convolved with itself (*i.e.* the convolutional filter is a function of the input). Autocorrelation can, in sub-quadratic time, identify the most important token interaction distances for the input u. See (Appendix D.7.2) for a formal definition of this operation.

Next, in the programmatic baseline, we mark the positions of potential associative recall hits as we causally process the input. Specifically, we mark positions (u_i) that correspond to tokens (u_j) that previously occurred in the sequence ($u_{<i}$) at some position j, since MQAR may be helpful to predict

the next word. We can then construct a filter with a 1 at position $i - j - 1$ and 0's elsewhere, to shift forward the value that is associated with the matching token. Ideally we would like learnable filters, however this baseline is illustrative and effective as shown in Figure 2.

B Extended Results

Below, we provide extended results beyond Table 1 across two additional parameter scales: 70M parameters and 350M parameters. The purpose is to demonstrate that AR remains an issue as we increase the model size.

Model	Param (M)	TFLOPs	Overall	Slices: AR Hits	Slices: Other Tokens	% of gap due to 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%
Hyena	72	1.34	15.13 (2.72)	9.00 (2.20)	15.74 (2.76)	60.8%
RWKV	72	1.89	16.10 (2.78)	14.11 (2.65)	16.26 (2.79)	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%
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%
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 2: **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 2.

C Extended Discussion of MQAR

Here we provide additional discussion on the properties of our MQAR synthetics, drawing contrast to the AR formulations in prior work [9, 16]. We hope MQAR can be a useful tool to stress test future architectures.

As discussed, the core difference is that language modeling can require performing $\mathcal{O}(n)$ recalls in one forward pass. Prior work assumes there is a single token that requires AR per example, we address this by designing a synthetic with multiple queries per example.

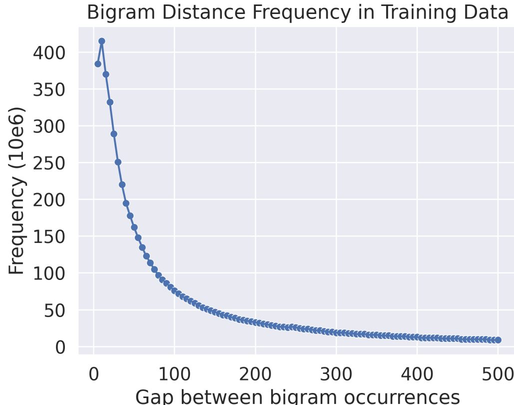


Figure 3: Across the Pile training data, we measure the distance between n-grams and their prior occurrences a provided context. We plot the frequency across differences, finding it follows a power law distribution.

The other modifications we find important are:

- *Vocab size.* We increase the vocabulary size to match the sizes typically used in language modeling (30k - 50k tokens). Prior work uses vocab sizes ≤ 40 tokens.
- *Gap distribution.* Prior work further assumes that the single AR token is always at a fixed position in the sequence. In real-world English text, the gaps between repeated n-gram occurrences follows a power-law (Figure 3). We setup our multi-query AR to contain a power-law distribution of lookup distances.
- *Limited repetition.* The prior AR task has each n-gram to be repeated numerous times per input. Our setup requires the model to identify a needle-in-the-haystack when recalling the n-gram.

D Details on Theoretical Analysis

D.1 Preliminaries and Notation

D.1.1 Notation

We denote the all 1 row vector of size k, given by $[1 \quad 1 \quad \dots \quad 1 \quad 1]$, and the all 0 row vector of size k, given by $[0 \quad 0 \quad \dots \quad 0 \quad 0]$, 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 ith row and the jth column, $\mathbf{M}[i, :] \in \mathbb{F}^{1\times n}$ denotes the ith row, and $\mathbf{M}[:, j] \in \mathbb{F}^{m\times 1}$ denotes the jth 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 $\text{poly}, \text{poly}^* : \mathbb{F}^n \to \mathbb{F}[X]/(X^n)$ such that

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

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

These operations allow us to interpret the convolution of vectors in terms of polynomial multiplication [13]. More specifically, we have

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

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

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

Arithmetic Circuit Notation. We briefly introduce the notation of arithmetic circuits [18], the focus of Appendix D.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 D.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 $\boldsymbol{u} \in \mathbb{R}^{N \times d}$ 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.

D.1.2 Summary of the Results

The outline of our results are as follows: In Appendix D.2 we introduce *gated convolution models* and define BASECONV and Hyena. In Appendix D.3 we introduce a set of primitive operations that BASECONV can implement. Then, in Appendix D.5, we show that general arithmetic circuit of size s and degree at most Δ can be simulated by BASECONVNext, in Appendix D.6, we derive a BASECONV model inspired from dyadic intervals, followed by a BASECONV model with data-dependent kernels, both of which can solve the multiple-query associative recall problem (MQAR).

D.2 Gated Convolution Models

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

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

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

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

D.2.1 The Hyena Layer

We will now outline the Hyena layer [17]. Hyena takes a sequence $\boldsymbol{u} \in \mathbb{R}^{N \times d}$ as input and produces $L + 1$ projections $p^1, \ldots, p^L, v$ by passing $\boldsymbol{y}$ though 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 2.

Algorithm 1 `Projection` $(\boldsymbol{u}, \boldsymbol{h})$

Require: Input sequence $\boldsymbol{u} \in \mathbb{R}^{N \times d}$, a short convolution filter $\boldsymbol{h} \in \mathbb{R}^N$.
1: In parallel for $0 \leq n < N : \hat{\boldsymbol{z}}[n, :] \leftarrow \texttt{Linear}_{d', (L+1)d}\,(\boldsymbol{u}[n, :])$ so that $\hat{\boldsymbol{z}} \in \mathbb{R}^{N \times (L+1)d}$
2: In parallel for $0 \leq t < (L+1)d : \boldsymbol{z}[:, t] \leftarrow \boldsymbol{h} * \hat{\boldsymbol{z}}[:, t]$
3: Reshape and split $\boldsymbol{z} \in \mathbb{R}^{N \times (L+1)d}$ into $\boldsymbol{p}^1, \ldots, \boldsymbol{p}^L, \boldsymbol{v}$, where $\boldsymbol{p}^\ell, \boldsymbol{v} \in \mathbb{R}^{N \times d}$ for $\ell \in [L]$.
4: **return** $\boldsymbol{p}^1, \ldots, \boldsymbol{p}^L, \boldsymbol{v}$.

Algorithm 2 `Hyena` $(\boldsymbol{u}, \boldsymbol{h}, \boldsymbol{h}_s)$

Require: Input sequence $\boldsymbol{u} \in \mathbb{R}^{N \times d}$, set of convolution filters $\boldsymbol{h}^1, \ldots, \boldsymbol{h}^L \in \mathbb{R}^{N \times d}$, short convolution filter $\boldsymbol{h}_s \in \mathbb{R}^N$.
1: $\boldsymbol{p}^1, \ldots, \boldsymbol{p}^L, \boldsymbol{v} \leftarrow \texttt{Projection}(\boldsymbol{u}, \boldsymbol{h}_s)$.
2: $\boldsymbol{z}^0 \leftarrow \boldsymbol{v}$
3: **for** $\ell = 1, \ldots, L$ **do**
4: In parallel for $0 \leq t < d : \boldsymbol{z}^\ell[:, t] \leftarrow \boldsymbol{p}^\ell[t, :] \odot \left(\boldsymbol{h}^\ell[:, t] * \boldsymbol{z}^{\ell-1}[:, t]\right)$.
5: **return** $\boldsymbol{z}^L$

Remark D.3. Algorithm 2 is not the model in [17] in terms of the number of layers L as we have focused on the recursive application of the Hadamard product and convolutions. However, asymptotically, this does not make a difference.

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

D.2.2 BASECONV

Finally, we introduce the BASECONV here as follows:

$$\mathbf{Y} = (\boldsymbol{u}\mathbf{W} + \boldsymbol{b}_1) \odot (\boldsymbol{u} * \boldsymbol{h} + \boldsymbol{b}_2), \tag{2}$$

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

Algorithm 3 BASECONV $(\boldsymbol{u}, \boldsymbol{W}, \mathrm{b}_1, \boldsymbol{h}, \mathrm{b}_2)$

Require: Input sequence $\boldsymbol{u} \in \mathbb{R}^{N' \times d'}$, linear mapping $\boldsymbol{W} \in \mathbb{R}^{d' \times d'}$, convolution filter $\boldsymbol{h} \in \mathbb{R}^{N' \times d'}$, bias matrices $\mathrm{b}_1, \mathrm{b}_2 \in \mathbb{R}^{N' \times d'}$.

1: In parallel for $0 \leq n < N'$: $\boldsymbol{x}[n, :] = \mathtt{Linear}_{d', d'}(\boldsymbol{u}[n, :])$

2: In parallel for $0 \leq t < d'$: $\boldsymbol{z}[:, t] = \boldsymbol{h}[:, t] * \boldsymbol{u}[:, t]$[2]

3: In parallel for $0 \leq t < d'$: $\boldsymbol{y}[:, t] \leftarrow (\boldsymbol{x}[:, t] + \boldsymbol{b}_1[:, t]) \odot (\boldsymbol{z}[:, t] + \boldsymbol{b}_2[:, t])$. ▷ See (2)

4: **return** $\boldsymbol{y}$

Similar to Hyena, we will refer to a model consisting of L BASECONV layers as $(N, L, d, N', d') -$ BASECONV. In our experiments, we extend BASECONV by adding an MLP after Algorithm 3. For simplicity we will denote BASECONV $(\boldsymbol{u}, \boldsymbol{h}, \boldsymbol{b}_1, \boldsymbol{b}_2)$ as BASECONV $(\boldsymbol{u}, \boldsymbol{h})$ when $\boldsymbol{b}_1 = \boldsymbol{b}_2 = \mathbf{0}$.

We will now show that there exists a BASECONV model that can emulate each of the basic operations in Algorithm 3.

Lemma D.4. *The functions* $\mathtt{Linear}_{d,d}(\boldsymbol{u})$*, with* d, d' *defined as in Algorithm 3, convolution with filter* $\boldsymbol{h} \in \mathbb{R}^{N \times d}$*, and element-wise gating can be computed with Algorithm 3 via a* $(N, 1, d, N, d') -$ BASECONV.

Proof. For each operation from Definition D.2 and Algorithm 3:

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

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

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

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

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

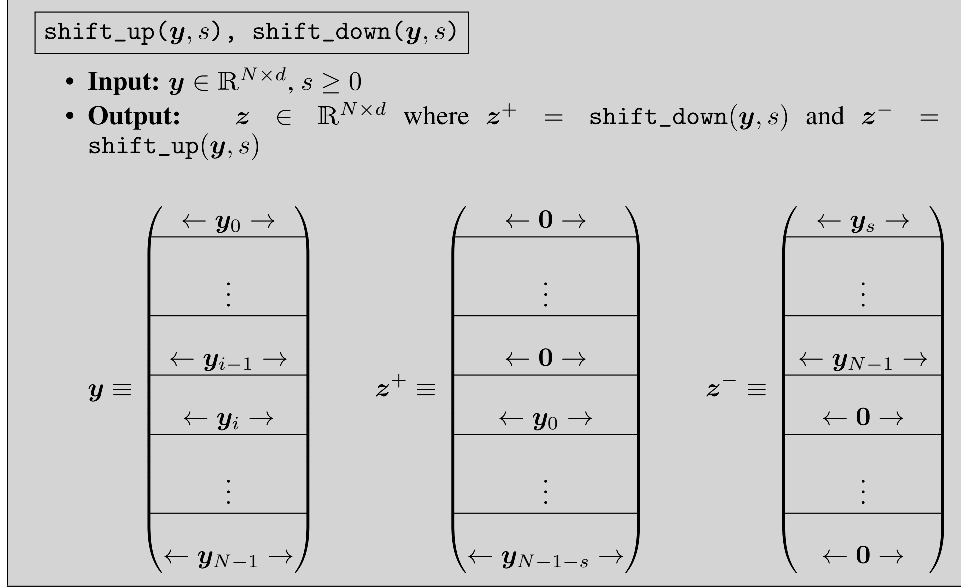
$$\mathbf{Y} = (\boldsymbol{u}\mathbf{0}^{N' \times d'} + \boldsymbol{v}) \odot (\boldsymbol{u} * \boldsymbol{e}_0 + \mathbf{0}^{N' \times d'}) = \boldsymbol{v} \odot \boldsymbol{u}.$$

□

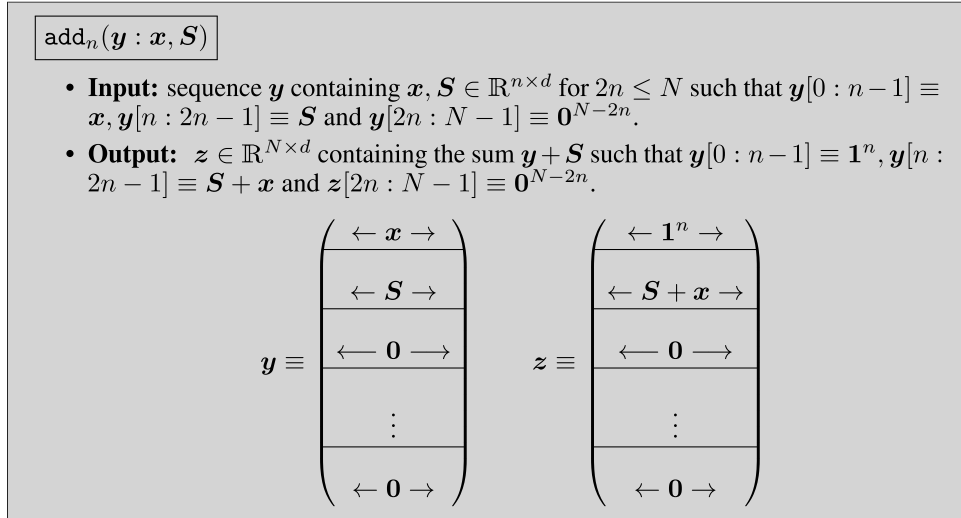
D.3 Primitives

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:



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



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

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

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

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 D.5 (The Shift Primitive)**.** *For any $\boldsymbol{y} \in \mathbb{R}^{N \times d}$, there exist $(N, 1, d, N, d) - \text{BASECONV}$ and $(N, 3, d, N, d) - \text{BASECONV}$ that computes $\texttt{shift_down}(\boldsymbol{y}, s)$ and $\texttt{shift_up}(\boldsymbol{y}, s)$ for any $s \leq N$.*

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

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

We now deal with the down and up shifts separately:

1. We define $\mathbf{W} := \mathbf{0}^{N \times d}, \boldsymbol{b}_1 := \mathbf{1}^{N \times d}, \boldsymbol{b}_2 := \mathbf{0}^{N \times d}$. Then, for input $\boldsymbol{y} \in \mathbb{R}^{N \times d}$, $\text{BASECONV}\left(\boldsymbol{y}, \mathbf{0}^{N \times d}, \boldsymbol{h}_s, \mathbf{1}^{N \times d}, \mathbf{0}^{N \times d}\right)$ for BASECONV in Algorithm 3 is given by (2) as

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

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

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

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

2. We again define $\mathbf{W} := \mathbf{0}^{N \times d}, \boldsymbol{b}_1 := \mathbf{1}^{N \times d}, \boldsymbol{b}_2 := \mathbf{0}^{N \times d}$. Then, for input $\boldsymbol{y} \in \mathbb{R}^{N \times d}$, $\text{BASECONV}\left(\boldsymbol{y}, \mathbf{0}^{N \times d}, \boldsymbol{e}_0, \mathbf{1}^{N \times d}, \mathbf{0}^{N \times d}\right)$ for BASECONV in Algorithm 3 is given in (2) as

   $$\mathbf{Y}_0 \equiv \boldsymbol{y} \circledast \boldsymbol{e}_0.$$

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

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

We then apply $\boldsymbol{Y}_1 \equiv \texttt{shift_down}(\boldsymbol{Y}_0, s)$ 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

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

Here, we note that we can compute both of these primitives in one and three layer, respectively (see Lemma D.8).

□

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

Proposition D.6 (The Running Sum Primitive)**.** *For any $\boldsymbol{x}, \boldsymbol{S} \in \mathbb{R}^{n \times d}$ contained in some $\boldsymbol{y} \in \mathbb{R}^{N \times d}$, there exists a $(N, 2, d, N, d) -$ BaseConv that computes $\texttt{add}_n(\boldsymbol{y} : \boldsymbol{x}, \boldsymbol{S})$ for BaseConv as in Algorithm 3.*

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

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

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

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

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

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

We then have

$$\boldsymbol{z}_1 \equiv \boldsymbol{b}_1^1 \odot \left(\boldsymbol{h}^1 * \boldsymbol{y}\right) \equiv \begin{pmatrix} \mathbf{0}^n \\ \hline \mathbf{1}^n \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix} \odot \begin{pmatrix} \boldsymbol{x} \\ \hline \boldsymbol{S}+\boldsymbol{x} \\ \hline \boldsymbol{S} \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix} \equiv \begin{pmatrix} \mathbf{0}^n \\ \hline \boldsymbol{S}+\boldsymbol{x} \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix}$$

Resetting for Next Phase. We now use the next layer to reset for the next phase. Here, we need the first vector to be $\mathbf{1}^n$ in order to start adding the next vector. We thus use the kernel and the biases $\boldsymbol{h}^2, \boldsymbol{b}_1^2, \boldsymbol{b}_2^2$ defined as

$$\boldsymbol{h}^2 \equiv \begin{pmatrix} \boldsymbol{e}_0 \\ \hline \mathbf{0}^n \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix}, \quad \boldsymbol{b}_1^2 \equiv \begin{pmatrix} \mathbf{1}^n \\ \hline \mathbf{1}^n \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix}, \quad \boldsymbol{b}_2^2 \equiv \begin{pmatrix} \mathbf{1}^n \\ \hline \mathbf{0}^n \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix}.$$

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

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

Thus, the output for the second layer is given by

$$\boldsymbol{z} \equiv \boldsymbol{b}_1^2 \odot \left(\boldsymbol{h}^2 * \boldsymbol{z}^1 + \boldsymbol{b}_2^2\right) \equiv \begin{pmatrix} \mathbf{1}^n \\ \hline \mathbf{1}^n \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix} \odot \left(\begin{pmatrix} \mathbf{0}^n \\ \hline \boldsymbol{S}+\boldsymbol{x} \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix} + \begin{pmatrix} \mathbf{1}^n \\ \hline \mathbf{0}^n \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix} \right) \equiv \begin{pmatrix} \mathbf{1}^n \\ \hline \mathbf{1}^n \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix} \odot \begin{pmatrix} \mathbf{1}^n \\ \hline \boldsymbol{S}+\boldsymbol{x} \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix} \equiv \begin{pmatrix} \mathbf{1}^n \\ \hline \boldsymbol{S}+\boldsymbol{x} \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix}.$$

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

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 D.7 (The Remembering Primitive)**.** *For any $\boldsymbol{x} \in \mathbb{R}^{n \times d}, \boldsymbol{v} \in \mathbb{R}^{m \times d}$ contained in some $\boldsymbol{y} \in \mathbb{R}^{N \times d}$ for some $n+m+s+t \leq N$ so that for $\boldsymbol{h} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{p} \in \mathbb{R}^{(n+s) \times d}$ with $\boldsymbol{x} * \boldsymbol{h} \in \mathbb{R}^{(n+s) \times d}$ and $\boldsymbol{v} * \boldsymbol{h} \in \mathbb{R}^{(m+t) \times d}$, there exists a $(N, 5, d, N, d)-$BASECONV that computes* $\texttt{remember}(\boldsymbol{y} : \boldsymbol{x}, \boldsymbol{v}, \boldsymbol{h}, \boldsymbol{p})$ *for* BASECONV *as in Algorithm 3.*

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

$$\boldsymbol{z}^1 \equiv \text{BASECONV}\left(\boldsymbol{y}, \mathbf{0}^{N \times 1}, \boldsymbol{h}^1, \boldsymbol{b}_1^1, \mathbf{0}^{N \times d}\right) \equiv \boldsymbol{b}_1^1 \odot \left(\boldsymbol{h}^1 * \boldsymbol{y}\right)$$
$$\boldsymbol{z}^2 \equiv \text{BASECONV}\left(\boldsymbol{z}^1, \mathbf{0}^{N \times 1}, \boldsymbol{h}^2, \boldsymbol{b}_1^2, \mathbf{0}^{N \times d}\right) \equiv \boldsymbol{b}_1^2 \odot \left(\boldsymbol{h}^2 * \boldsymbol{y}\right),$$

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

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

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

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

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

We then have

$$\boldsymbol{z}_1 \equiv \boldsymbol{b}_1^1 \odot \left(\boldsymbol{h}^1 * \boldsymbol{y}\right) \equiv \begin{pmatrix} \boldsymbol{p} \\ \hline \boldsymbol{0}^{m+t} \\ \hline \boldsymbol{0}^n \\ \hline \boldsymbol{0}^s \\ \hline \boldsymbol{1}^m \\ \hline \dots \\ \hline \boldsymbol{0}^n \end{pmatrix} \odot \begin{pmatrix} \boldsymbol{h} * \boldsymbol{x} \\ \hline \boldsymbol{h} * \boldsymbol{v} \\ \hline \boldsymbol{x} \\ \hline \boldsymbol{0}^s \\ \hline \boldsymbol{v} \\ \hline \dots \\ \hline \boldsymbol{0}^n \end{pmatrix} \equiv \begin{pmatrix} \boldsymbol{p} \odot (\boldsymbol{h} * \boldsymbol{x}) \\ \hline \boldsymbol{0}^{m+t} \\ \hline \boldsymbol{0}^n \\ \hline \boldsymbol{0}^s \\ \hline \boldsymbol{v} \\ \hline \dots \\ \hline \boldsymbol{0}^n \end{pmatrix}.$$

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

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

This yields the following convolution computation:

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

Thus we have

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

We now shift this up by $2n+s+m+t$ entries using the primitive operation defined in Proposition D.5 that costs three additional layers so that we end up with

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

Again, we note that the only operations we perform and 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 D.8. *For $f, g : \mathbb{R}^{N \times d} \to \mathbb{R}^{N \times d}$ 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 $f(\boldsymbol{u})$ which requires L_1 layers to compute and that we can compute $f \circ g(\boldsymbol{u}) = g(f(\boldsymbol{u}))$ using the BASECONV model with L_2 layers, yielding $L_1 + L_2$ layers in total. $\square$

D.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 D.9. *For any input $\boldsymbol{u} \in \mathbb{R}^{N \times d}$ and $(N, L, d, N', d)-$Hyena such that $\boldsymbol{z}_{\text{Hyena}} \equiv \text{Hyena}(\boldsymbol{u})$ with a set of filters $\boldsymbol{h}^{\ell}$ and projections $\boldsymbol{p}^{\ell}$ for $\ell \in [L]$, there exists a $(N, 5L, d, N' + N, d)$-BASECONVmodel such that $\boldsymbol{z}_{\text{Hyena}} \equiv \text{BASECONV}(\boldsymbol{u})$.*

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

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

Proof. For the input $\boldsymbol{u}_{\text{Hyena}} \in \mathbb{R}^{N \times d}$, the output of the ℓth layer $\boldsymbol{z}^{\ell}_{\text{Hyena}} \in \mathbb{R}^{N' \times d'}$ for Hyena is given by (see Algorithm 2)

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

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

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

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

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

Overall, the output of the ℓth layer for BASECONV is given by

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

Hence, we can reproduce the output of the ℓth layer of Hyena using five layers of BASECONV after augmenting the input and using the remembering primitive (Proposition D.7) with internal dimension $N' + N$.

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

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

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

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

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

$$\begin{aligned}
& \text{Hyena}\left(\text{Hyena}\left(\ldots \text{Hyena}(\boldsymbol{u}_{\text{BASECONV},\boldsymbol{h}})\right)\right) \\
\equiv\; & \text{Hyena}\left(z^{(l-1)}_{\text{BASECONV}}\right) \\
\equiv\; & \text{linear}\left(z^{(l-1)}_{\text{BASECONV}}\right) \odot \text{conv}\left(h^{l}, z^{(l-1)}_{\text{BASECONV}}\right) \equiv z^{l-1}_{\text{BASECONV}} \\
\equiv\; & z^{\ell}_{\text{BASECONV}}.
\end{aligned}$$

□

D.4 Linear Arithmetic Circuits

In this section we show the relation between linear arithmetic circuits and BASECONV. We recall a few definitions from [7].

Definition D.10 (Linear Arithmetic Circuit [4])**.** 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 D.11 (Butterfly Matrices [7])**.** 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 :

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

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: $\mathbf{B}^{(n)} = \mathbf{B}_n^{(n)} \mathbf{B}_{\frac{n}{2}}^{(n)} \ldots \mathbf{B}_2^{(n)}$. Equivalently, we may define $\mathbf{B}^{(n)}$ recursively as a matrix that can be expressed in the following form:

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

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

From Definition D.11, we observe that size n butterfly factor is comprised of three vectors $\boldsymbol{d}, \boldsymbol{d}^{+}, \boldsymbol{d}^{-} \in \mathbb{R}^n$ such that

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

where $\operatorname{diag}^{-1}(\mathbf{D}) : \mathbb{R}^{n \times n} \mapsto \mathbb{R}^n$ 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 $\operatorname{diag}(\boldsymbol{d}), \operatorname{diag}\left(\boldsymbol{d}^{+}\right)$, and $\operatorname{diag}\left(\boldsymbol{d}^{-}\right)$ respectively. Then we note that

$$\mathbf{D}_1 \equiv \begin{bmatrix} \mathbf{D}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_4 \end{bmatrix} \qquad \mathbf{D}_2 \boldsymbol{S}^{\frac{n}{2}} \equiv \begin{bmatrix} \mathbf{0} & \mathbf{D}_2 \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \qquad \boldsymbol{S}^{\frac{n}{2}} \mathbf{D}_3 \equiv \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{D}_3 & \mathbf{0} \end{bmatrix} \tag{3}$$

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 D.12. *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 (3).

We use Proposition D.12 to show that butterfly matrices can easily be computed by BASECONV .

Lemma D.13. *For any $n, d \geq 2$, $k \geq 1$, and arbitrary vector $\boldsymbol{x} \in \mathbb{R}^{nd}$:*

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

Proof. (1) Given $\boldsymbol{x} \in \mathbb{R}^{nd}$, construct $\boldsymbol{u} \in \mathbb{R}^{n \times d}$ where $\boldsymbol{x}$ is the row-major form of $\boldsymbol{u}$. We show that BASECONV can compute $\mathbf{B}_{nd} \cdot \boldsymbol{x}$ 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 D.12. We take $\boldsymbol{d}_1 = \mathbf{1}^{nd}\mathbf{D}$, $\boldsymbol{d}_2 = \mathbf{1}^{nd}\mathbf{C}_2, \boldsymbol{d}_3 = \mathbf{1}^{nd}\mathbf{A}$, which extracts the diagonal entries of $\mathbf{D}_i$. With this we construct $\mathbf{D}_i' \in \mathbb{R}^{n \times d}$ where $\boldsymbol{d}_i$ is the row major form of $\boldsymbol{D}_i'$. This implies that

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

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

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

By Lemma D.4, each Hadamard product $\mathbf{A} \odot \boldsymbol{u}, \mathbf{B} \odot \boldsymbol{u}, \mathbf{C} \odot \boldsymbol{u}$ can be trivially be performed with a single layer BASECONV model. Let each of these model outputs be denoted $\boldsymbol{y_1}, \boldsymbol{y_2}, \boldsymbol{y_3}$, respectively. Finally all that remains is to compute the $\boldsymbol{y_1} + \boldsymbol{y_2} + \boldsymbol{y_3}$. We achieve this using layers of add primitives[3]:

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

where using by Proposition D.6 and Lemma D.8, this requires six more layers, and we get

$$\boldsymbol{y}^3 \equiv \boldsymbol{y_1} + \boldsymbol{y_2} + \boldsymbol{y_3} \equiv \mathbf{B}_{nd}\boldsymbol{x}.$$

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

(2) From Definition D.11, $\mathbf{B}^{(nd)} = \mathbf{B}^{(nd)}_{nd}\mathbf{B}^{(nd)}_{\frac{nd}{2}} \dots \mathbf{B}^{(nd)}_{2}$. From (1), BASECONV can compute any butterfly matrix by simulating the $\log(nd)$ butterfly factor matrices which comprise $\mathbf{B}^{(nd)}$. With Lemma D.8, this creates a BASECONV with $5 \cdot \log(nd) = \mathcal{O}(\log(nd))$ layers. Lemma D.8

□

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

Definition D.14 (The Kaleidoscope Hierarchy [7])**.**

- Define $\mathcal{B}$ as the set of all matrices that can be expressed in the form $\mathbf{B}^{(n)}$ (for some n).
- Define $(\mathcal{BB}^*)$ 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 $(\mathcal{BB}^*)^w$ 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 $\mathbf{M}_i \in (\mathcal{BB}^*)$ $(1 \le i \le w)$. (The notation w represents width.)
- Define $(\mathcal{BB}^*)^w_e$ 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 $\mathbf{E} \in (\mathcal{BB}^*)^w$, where $\mathbf{S} \in \mathbb{F}^{n \times en} = [\mathbf{I}_n \quad 0 \quad \dots \quad 0]]$ (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 D.15. *Given $n, d \ge 2$, $e > 0$ for any $nd \times nd$ matrix $\mathbf{M} \in (\mathcal{BB}^*)^w_e$, and $\boldsymbol{x} \in \mathbb{R}^{nd}$ there exists a $(N, L, d, N', d') -$ BASECONV that can represent $\mathbf{M} \cdot \boldsymbol{x}$ with $N = n, L = \mathcal{O}(w \log(end)), N' = en$, and $d' = d$.*

Proof. By Definition D.14, $\mathbf{M}$ 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 $\mathbf{E}_i \in (\mathcal{BB}^*)$ can be expressed as a product of $2 \log end$ butterfly factor matrices. Then by Lemma D.13 and Lemma D.8 we can compute $\mathbf{E}_i\boldsymbol{x'}$ in by stacking $2 \log end$ $(n, d, L, en, d) -$ BASECONV models each with $L = \mathcal{O}(1)$. Because $\mathbf{E}$ has width w, Lemma D.8 implies that composing with each $\mathbf{E}_i$ for $1 \le i \le w$ 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 [7] it is assumed that $w = s$, yet inspection of the proof yields the following stronger result:

[3] Recall that $\texttt{add}_n(\boldsymbol{y} : \boldsymbol{x}, \boldsymbol{S})$ adds the subvector $\boldsymbol{x}$ to $\boldsymbol{S}$ for the input $\boldsymbol{y}$.

Theorem D.16 ([7]). *Let* $\mathbf{M}$ *be an* $n \times n$ *matrix such that multiplication of* $\mathbf{M}$ *times an arbitrary vector* $\boldsymbol{u}$ *can be be represented as* (n, s, Δ, w)*-linear arithmetic circuit* $\mathcal{C}$*. Then,* $\mathbf{M} \in (\mathcal{B}\mathcal{B}^*)^{\mathcal{O}(\Delta)}_{\mathcal{O}(w/n)}$.

We combine Theorem D.16 and Lemma D.15 to show that BASECONV can compute any linear arithmetic circuit with polylogarithmic factors in Δ.

Corollary D.17. *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* $\boldsymbol{x} \in \mathbb{R}^{nd}$*, there exists an equivalent* $(n, \Delta', d, w, d) -$ BASECONV *with* $\Delta' = \mathcal{O}(\Delta \log(nd))$ *such that* $\mathbf{M}\boldsymbol{x} = \text{BASECONV}(\boldsymbol{u}, \boldsymbol{h})$ *where* $\boldsymbol{x}$ *is the row major form of* $\boldsymbol{u} \in \mathbb{R}^{n \times d}$.

D.5 General Arithmetic Circuits

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

Theorem D.18. *For any* (nd, s, Δ, w)-arithmetic circuit $\mathcal{C}$*, there exists an equivalent* $(N, \Delta', d, N', d') -$ BASECONV *with* $N = n, \Delta' = \mathcal{O}(\Delta \log w)$*,* $N' = O(w), d' = d$ *which simulates* $\mathcal{C}$.

Proof. Let us layer $\mathcal{C}$ so that each layer $\mathcal{C}_\ell$ for $\ell \in L_{\mathcal{C}} = \mathcal{O}(\Delta)$ either only has linear gates or multiplication gates, and the composition of all $\mathcal{C}_\ell$ layers results in $\mathcal{C}$. We use $\boldsymbol{z}^\ell \in \mathbb{R}^w$ to denote the output of the ℓ-th layer $\mathcal{C}_\ell$ which feeds as the input to the $(\ell+1)$-th layer $\mathcal{C}_{\ell+1}$. Here, we note that if we can simulate each $\mathcal{C}_\ell$ with BASECONV, then we can simulate the entire layered circuit $\mathcal{C}$ due to Lemma D.8.

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 $\boldsymbol{M} \in \mathbb{R}^{w \times w}$ multiplied by $\boldsymbol{z}^{\ell-1} \in \mathbb{R}^w$ (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 D.17 to simulate $\mathcal{C}_\ell^{\text{lin}}$ with an equivalent $(n, \log w, d, w, d) -$ BASECONV model.

Next, if $\mathcal{C}_\ell^{\text{mult}}$ instead consists of only the multiplication gates. Then, we note here that the output $\boldsymbol{z}^{\ell-1}$ may not exactly equal the input to $\mathcal{C}_\ell^{\text{mult}}$. Nevertheless, we can apply a $O(w)$ sparse linear map $\mathbf{R} \in \mathbb{R}^{w \times w}$ so that $\mathbf{R}\boldsymbol{z}^{\ell-1}$ yields vectors $\boldsymbol{v}^1, \boldsymbol{v}^2$, and $\boldsymbol{v}^3$, where $\boldsymbol{v}_1$ constitutes the "first" input to all the multiplication gates and $\boldsymbol{v}_2$ constitutes all the "second" inputs while $\boldsymbol{v}^3$ consists of all entries needed as inputs in the subsequent layers. That is, for the ith gate in $\mathcal{C}_\ell^{\text{mult}}$, we compute $\boldsymbol{v}_i^1 \cdot \boldsymbol{v}_i^2$. This implies that for all the gates in $\mathcal{C}_\ell^{\text{mult}}$, we can simply compute $\boldsymbol{v}_1 \odot \boldsymbol{v}_2$. To this end, we can simply use the $\texttt{remember}(\boldsymbol{z}^{\ell-1} : \boldsymbol{v}_1, \boldsymbol{v}_3, \boldsymbol{e}_0, \boldsymbol{v}_2)$ primitive with constant number of layers from Proposition D.7 to define a $(n, O(\log w), d, w, d) -$ BASECONV model that remembers $\boldsymbol{v}^3$ while performing the Hadamard product of $\boldsymbol{v}^1$ with $\boldsymbol{v}^2$.

Overall, we can then collect all the resulting BASECONVlayers and compose them as in Lemma D.8 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. $\square$

D.6 The Multiple-Query Associative Recall Problem

D.6.1 Introduction

In this section, we consider a general version of the associative recall problem [3]. The original formulation of the associative recall problem considers an input sequence of key-value pairs with a query at the end. If there exists a key in the sequence that matches the query, we return its associated value. However, as we have shown empirically in Section 2, proficiency in this synthetic does not necessarily align to performance in downstream tasks. Therefore, we have proposed a new general synthetic version of this problem that helps in this regard.

Setup. We recall here the *multiple-query associative recall* problem (MQAR) from Definition 2.1.

> Suppose we are given an input sequence $\boldsymbol{u}[0 \cdots 3N-1] \triangleq \{(\boldsymbol{k}_0, \boldsymbol{v}_0, \boldsymbol{q}_0), \ldots, (\boldsymbol{k}_{N-1}, \boldsymbol{v}_{N-1}, \boldsymbol{q}_{N-1})\}$ with each $\boldsymbol{k}_i, \boldsymbol{v}_i, \boldsymbol{q}_i \in C$ is a token drawn from a vocabulary of size $c = |C|$. Our goal is then to check, for each $1 \leq i \leq N-1$, whether there exists $0 \leq j < i$ such that $\boldsymbol{q}_i \equiv \boldsymbol{k}_j$, and if so, output $\boldsymbol{v}_j$.

Here, we note that it suffices to have $d \approx \log(C)$ so that $\boldsymbol{k}_i, \boldsymbol{v}_i, \boldsymbol{q}_i$ is embedded in $\{0,1\}^d$. Here, we construe the tokens $\boldsymbol{k}_i, \boldsymbol{q}_i$ and $\boldsymbol{v}_i$ to be the *keys*, the *queries*, and the *associated values*. Indeed, it might be helpful to think of the input $\boldsymbol{u}$ 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. A slightly more specific problem considers a sequence $\boldsymbol{u}[0 \cdots N-1] := \{\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{N-1}\}$, where each $\boldsymbol{x}_i \in C$. The goal is then to check, for each $1 \le i < N-1$, whether there exists $0 \le j < i$ such that $\boldsymbol{x}_i \equiv \boldsymbol{x}_j$, and if so, output $\boldsymbol{x}_{j+1}$, and continue otherwise. We can reduce this problem to the general formulation by taking the following sequence of tuples as the input $\{(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}, \boldsymbol{x}_i)\}$.

D.6.2 Trivial 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.

Proposition D.19. *Given an input $\boldsymbol{u} \in \{0,1\}^{N \times c}$, Attention (even without using soft-max) solves* MQAR *for $\boldsymbol{u}$ using $\mathcal{O}(\max(N, c^2))$ parameters, $\mathcal{O}(\min(Nc^2, N^2c))$ time complexity and $\mathcal{O}(1)$ layers.*

Proof. Without softmax, the output for attention $\mathbf{O} \in \mathbb{R}^{3N-1 \times d}$ is given by

$$\mathbf{O} \equiv \left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V}^1, \tag{4}$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{3N-1 \times d}$ are positional embeddings learned from the input $\boldsymbol{u} \in \mathbb{R}^{3N-1 \times d}$ such that we have

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

where $\mathcal{K}, \mathcal{Q}, \mathcal{V}$ are defined as in (13). We can then use N parameters to shift up $\mathbf{V}^1$ so that we have

$$\mathbf{V}^1[i,:] := \begin{cases} \boldsymbol{u}[i-1,:] & \text{if } i \in \mathcal{V}, \\ \mathbf{0}^d & \text{otherwise} \end{cases} = \begin{cases} \boldsymbol{u}[i+1,:] & \text{if } i \in \mathcal{K}, \\ \mathbf{0}^d & \text{otherwise} \end{cases},$$

where note that $\boldsymbol{u}[i+1,:] \equiv \boldsymbol{v}_i$ for $i \in \mathcal{K}$. Next, we compute the term in the parenthesis as

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

Finally, we compute the output as follows:

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

That is, for each query $\boldsymbol{q}_i$, we solve the associated value problem yielding a match for the jth key.

In total, we only need $\mathcal{O}(c^2)$-many parameters to perform these multiplications with N for shifting values; the time complexity comes from the fact that we can either multiply $\mathbf{Q}\mathbf{K}^\top$ or $\mathbf{K}^\top\mathbf{V}^{\mathbf{1}}$, and finally, we only need $\mathcal{O}(1)$ layer to do so. □

In the sequel, we develop an 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.

D.6.3 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 4 `Sequential-MQ-AR` $(\boldsymbol{u})\,[0\cdots N-1]$

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

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

Proposition D.20. *Algorithm 4 solves the multiple-query associative recall problem* (MQAR) *in* $\mathcal{O}(dN\log N)$ *time for an input sequence* $\boldsymbol{u}\in\{0,1\}^{N\times d}$.

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

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

□

D.6.4 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 D.18 will lead to a BASECONV model that solves the multiple-query associative recall problem. With respect to Algorithm 4, 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. 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* [2]. To this end, we briefly introduce the multiple search problem below.

> Given two array of numbers $A \triangleq a_0 \leq \ldots \leq a_{n-1}$ and $B \triangleq (b_0 \leq \ldots \leq b_{m-1})$ 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 [2] with work complexity $\mathcal{O}(n \cdot \log m)$ and time $\mathcal{O}(\log n \log m)$. Specifically, for sorted arrays $A[0 \cdots n-1]$ and $B[0 \cdots m-1]$, `pbs` constructs the array $C[0 \cdots n-1]$ defined as

$$C[i] \triangleq \begin{cases} \min_{0 \leq j < m} \{j \mid A[i] \leq B[j]\} & \text{if } A[i] \leq B[m-1] \\ m & \text{otherwise.} \end{cases} \quad (5)$$

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 [2] for our purposes below.

Algorithm 5 `pbs-key-values` $(\boldsymbol{q}[s \cdots t],\ \boldsymbol{k}[x \cdots y], n, m)$

Require: sorted arrays $\boldsymbol{q}[s \cdots t] := \{\boldsymbol{q}_i\}_{i=s}^{t},\ \boldsymbol{k}[x \cdots y] := \{\boldsymbol{k}_j\}_{j=x}^{y}$.

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

Let Σ be the set $\{0,1\}$ and denote the set of binary strings of size n as Σ^n. We define $\text{prefix}(\boldsymbol{x})$ for n-bit strings as the set of all initial substrings of $\boldsymbol{x} \in \Sigma^n$ which includes the empty string and $\boldsymbol{x}$ itself. Next, let $\text{dec} : \{0,1\}^n \to \mathbb{N}$ be the decimal representation of an n-bit string $\boldsymbol{x}$ with $\boldsymbol{x}[0]$ denoting the least significant bit. We also use `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 6 `Parallel-MQAR` $(\boldsymbol{u}[0 \cdots N-1])$

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

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

D.6.5 Correctness and Complexity

Proposition D.21. *Algorithm 6 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 6 if we can show that, for each $1 \leq i < N$, we check for a match among the keys $\{\boldsymbol{k}_j\}_{j \in [i-1]}$. To this end, for each $1 \leq i < N$, let the set of all iterator indices associated with an index i be defined as $K_i \triangleq \{(k, x) | i \in J_k^{\boldsymbol{x}}\}$ with $J_k^{\boldsymbol{x}}$ as noted in line 9. Then, we define the corresponding set for keys as $\mathcal{I}_i \triangleq \bigcup_{(k,\boldsymbol{x}) \in K_i} I_k^{\boldsymbol{x}}$ with $I_k^{\boldsymbol{x}}$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 $j \in I_k^{\boldsymbol{x}} \subseteq \mathcal{I}_i$ for some $\boldsymbol{x} \in \Sigma^{\log N - k}$, then, by definition, $\boldsymbol{x} \in \text{prefix}(\text{bin}(j))$. Here, let $\boldsymbol{x}^1 := \boldsymbol{x}|_{\boldsymbol{x}[\log N - k] = 1}$ where we set the $(\log N - k)$-th index of $\boldsymbol{x}$ to be 1. Consequently, as we have $i \in J_k^{\boldsymbol{x}}$ for the same k and $\boldsymbol{x}$ as in $I_k^{\boldsymbol{x}}$ (*cf.* line 6), we must have $\boldsymbol{x}^1 \in \text{prefix}(\text{bin(i)})$. 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 $\text{bin}(i)$. That is, there exists $\boldsymbol{x} \in \text{prefix}(\text{bin}(i)) \cap \text{prefix}(\text{bin}(j))$ with $\boldsymbol{x} \in \Sigma^b$. Thus, we then must have $\text{bin}(j) \in I_{\log N - b}^{\boldsymbol{x}}$ and $\text{bin}(i) \in J_{\log N - b}^{\boldsymbol{x}}$ with $\boldsymbol{x}$ 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 $1 \leq i < N$, we can conclude that Algorithm 6 solves the multiple-query associative recall problem.

Next, it is easy to see that we execute lines 9 to 14 $\sum_{k=0}^{\log N - 1} \frac{N}{2^{k+1}}$-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. 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 15 to 17 is $\mathcal{O}(N)$.

Thus, the overall work complexity of Algorithm 6 is

$$d \cdot \sum_{k=0}^{\log N-1} \frac{N}{2^{k+1}} \mathcal{O}(2^k \cdot \log 2^k) = d \cdot \mathcal{O}(N) \cdot \sum_{k=0}^{\log N-1} \mathcal{O}(k) = \mathcal{O}(Nd \cdot \log^2 N). \tag{6}$$

We will now analyze the depth of Algorithm 6. We know that the depth of computation for Algorithm 5 is $\mathcal{O}(\log n \log m)$ for input sizes. Moreover, we have $\mathcal{O}(1)$ depth for the computation in 15 to 17 as each entry in `idx` can have at most one entry. Since the nested for loops iterating over ks and the associated $\boldsymbol{x}$s runs in parallel, the depth of Algorithm 6 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 $n = m = 2^{\log N-1} = N/2$ which yields a depth of $d \cdot \log^2 N$. Moreover, using sorting networks Definition D.25, we know that the largest depth is for sorting $N/2$ values of size d given by $d \cdot \Theta(\log N)$ (Lemma D.26). Thus, we can conclude that Algorithm 6 takes $\mathcal{O}(d \cdot \log^2 N)$, time where N is the length of the input. ☐

D.6.6 Conversion to Arithmetic Circuit

We will convert Algorithm 6 to an arithmetic circuits modularly. In particular, after writing out an explicit circuit for Algorithm 5, we will uses 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 5 as `pbs-key-values` as well with the input gates comprising of each entry from $\boldsymbol{q}[s \cdots t]$ and $\boldsymbol{k}[x \cdots y]$ and the i-th output gate yielding the value $C[i]$ as in Algorithm 5.

Here, we first convert the comparisons for the `if` statements in Algorithm 5. To this end, we briefly introduce comparators.

Definition D.22 (Comparators)**.** A *comparator* is a device with inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and outputs $\boldsymbol{x}'$ and $\boldsymbol{y}'$, that performs the following function:

$$\boldsymbol{x}' = \min(\boldsymbol{x}, \boldsymbol{y}),$$
$$\boldsymbol{y}' = \max(\boldsymbol{x}, \boldsymbol{y}).$$

Using comparators, we can use bit-wise `XOR` and `AND` to define the result of the comparisons in lines 6 and 11 as the following fixed variables:

$$\begin{aligned} \ell_x &:= \mathbb{1}\{\boldsymbol{q}[\text{mid}] \le \boldsymbol{k}_x\}, \\ g_y &:= \mathbb{1}\{\boldsymbol{q}[\text{mid}] > \boldsymbol{k}_y\}, \\ z &:= \texttt{bin-search}(\{\boldsymbol{k}_i\}_{i=s}^t,\ \boldsymbol{q}[\text{mid}]). \end{aligned} \tag{7}$$

This then allows us to infer the index of the key array for each recursive call to `pbs-key-values` in lines 9, 14, 19, and 20 from Algorithm 5. 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 5) below:

$$\texttt{pbs-key-values}(\boldsymbol{q}[\text{mid}+1 \cdots t],\ \boldsymbol{k}[z_t \cdots y]);\ \text{(lines 9 and 20)} \tag{8}$$
$$\texttt{pbs-key-values}(\boldsymbol{q}[s \cdots \text{mid}-1],\ \boldsymbol{k}[x\ \cdots z_s - 1]);\ \text{(lines 14 and 19)} \tag{9}$$

Here, z_t and z_s can assume values dependent on the results of the comparisons in lines 6 and 11. Specifically, we have

$$z_t = \ell_x \cdot x + (1-\ell_x)(1-g_y) \cdot z = \begin{cases} x & \text{if } \boldsymbol{q}[\text{mid}] \le \boldsymbol{k}_x \text{ (line 9)} \\ z & \text{if } \boldsymbol{q}[\text{mid}] \in (\boldsymbol{k}_x, \boldsymbol{k}_y] \text{ (line 20)} \\ 0 & \text{otherwise} \end{cases},$$

$$z_s = g_y \cdot (y+1) + (1-\ell_x)(1-g_y) \cdot z = \begin{cases} y+1 & \text{if } \boldsymbol{q}[\text{mid}] > \boldsymbol{k}_y \text{ (line 14)} \\ z & \text{if } \boldsymbol{q}[\text{mid}] \in (\boldsymbol{k}_x, \boldsymbol{k}_y] \text{ (line 19)} \\ 0 & \text{otherwise} \end{cases}.$$

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 $C_t[\mathrm{mid}+1\cdots t]$ and $C_s[s\cdots \mathrm{mid}-1]$ denote the outputs to the recursive calls in (8) and (9), respectively. We can then succinctly express the outputs for each index of the output array C as

$$C[i] = \begin{cases} \ell_x \cdot x + z_s \cdot (C_1[i]) & i \in [s \cdots \mathrm{mid}-1] \\ g_y \cdot (y+1) + z_t \cdot (C_2[i]) & i \in [\mathrm{mid}+1 \cdots t] \\ \ell_x \cdot x + (1-\ell_x)(1-g_y)\cdot z + g_y \cdot (y+1) & i = \mathrm{mid} \end{cases} \tag{10}$$

We can thus state the circuit schematically in Fig. 4.

Now, before accounting for the complexity of the circuit for Algorithm 5, we must first assert the complexity of the comparators that we use in Fig. 4.

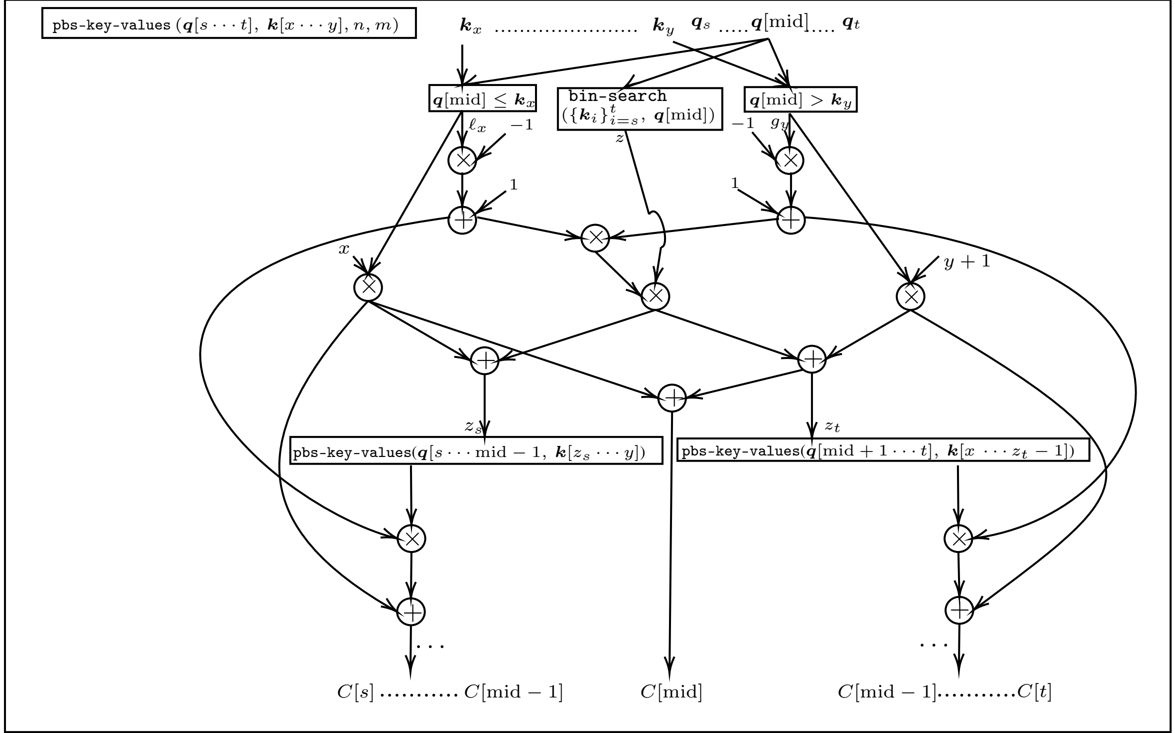


Figure 4: pbs-key-values ($\boldsymbol{q}[s\cdots t]$, $\boldsymbol{k}[x\cdots y], n, m$) as a circuit with recursive calls and subprocedures as "black boxes."

Lemma D.23 ([6])**.** *For binary strings $\boldsymbol{x}, \boldsymbol{y} \in \Sigma^d$ of length d, there exists a comparison network of size $\mathcal{O}(d)$, width $\mathcal{O}(d)$, and depth $\mathcal{O}(\log d)$ that computes the variables in (7).*

Proposition D.24. *There exists an $(\mathcal{O}(n+m), \mathcal{O}(nd\cdot(\log m + \log n)), \mathcal{O}(\log n(\log m + \log n \log d)), \mathcal{O}(n))$-arithmetic circuit[4] equivalent to* pbs-key-values *(Algorithm 5) with inputs $\boldsymbol{q}$ and $\boldsymbol{k}$ 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 5 but we also need to account for the comparison gates in (7). 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 5. Using Lemma D.23, 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 here is determined by the runtime of

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

Algorithm 5 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 5, which yields the width of the circuit. □

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 6. Indeed, we can direct "translate" Algorithm 6 to an arithmetic circuit as the values for $I_k^{\boldsymbol{x}}$ and $I_k^{\boldsymbol{x}}$ for each $\boldsymbol{x}$ and k are predetermined from N. Thus, we start by placing the corresponding sorting networks which feeds into the `pbs-key-values` $(\boldsymbol{q}[s \cdots t],\ \boldsymbol{k}[x \cdots y], n, m)$ circuit for Algorithm 5 in Fig. 4 so that the output values from the calls to `pbs-key-values` result in the checks as in line 14 of Algorithm 6. That is, we get outputs $C_k[\mathrm{dec}(i)]$ 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 $C_k[\mathrm{dec}(i)]$ for the query $\boldsymbol{q}_i$ which results in the output of the associated value $\boldsymbol{v}_{C_k[\mathrm{dec}(i)]+1}$, exactly as in Algorithm 6. That is, we first define the following variable as the output of the comparison gate:

$$\boldsymbol{c}^k_{\mathrm{dec}(i)} := \mathbb{1}\{C_k[\mathrm{dec}(i)] \neq 2^k\}. \tag{11}$$

Here, as $C_k[\mathrm{dec}(i)] \neq 2^k$ implies that $C_k[\mathrm{dec}(i)]$ equals the index of the matching key $\boldsymbol{k}_j$ corresponding to the query $\boldsymbol{q}_i$, the ith output is then simply given by $\boldsymbol{c}^k_{\mathrm{dec}(i)} \cdot C_k[\mathrm{dec}(i)]$, 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 D.25 (Informal)**.** *Sorting networks* are circuits with gates and wires where the gates of the circuit are comparators (Definition D.22) 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. 5.

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

Lemma D.26 ([1])**.** *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 D.27. *There exists an* $(N, \mathcal{O}(Nd \cdot \log^2 N), \mathcal{O}(\log d \log^2 N), \mathcal{O}(Nd \log N))$-arithmetic circuit *that solves the multiple-query associative recall problem.*

Proof. We note here that for each k, there are $N/2^{k+1}$ parallel problems of size 2^k for both the sorting networks and the $\texttt{pbs} - \texttt{key} - \texttt{values}$ circuit. Using Lemma D.26, the cumulative size of these sorting networks is $\mathcal{O}(d \cdot N \log^2 N)$ (see (6)) with overall depth $\mathcal{O}(\log d \log N)$.

Similarly, the next layer again runs $\sum_{k=0}^{\log N - 1} \frac{N}{2^{k+1}}$-many circuits for $\texttt{pbs} - \texttt{key} - \texttt{values}$ each of which has size $\mathcal{O}(2^k d(\log 2^k + \log 2^k)) = O(d \cdot 2^k \log 2^k)$, depth $\mathcal{O}(\log^2 2^k \log d)$ and width $\mathcal{O}(2^k)$ (Proposition D.24). Again, the cumulative size of this layer is given by $\mathcal{O}(Nd \cdot \log^2 N)$ (see (6)). 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 $N \log N$ 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 D.23). Therefore, the resulting arithmetic circuit has size $\mathcal{O}(d \cdot N \log^2 N + Nd \cdot \log^2 N + N \log N \cdot d) = O(Nd \log^2 N)$, depth $\mathcal{O}(\log d \log^2 N)$ and width $\mathcal{O}(Nd \log N)$. □

D.6.7 The Resulting BaseConv Model

As we have an arithemtic circuit for solving the multiple-query associative recall problem, we can now invoke Theorem D.18 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 D.28. *There exists a* $\left(N, \tilde{\mathcal{O}}(1), \tilde{\mathcal{O}}(1), \tilde{\mathcal{O}}(N), \tilde{\mathcal{O}}(1)\right)$ − BaseConv *solves the multiple-query associative recall problem.*

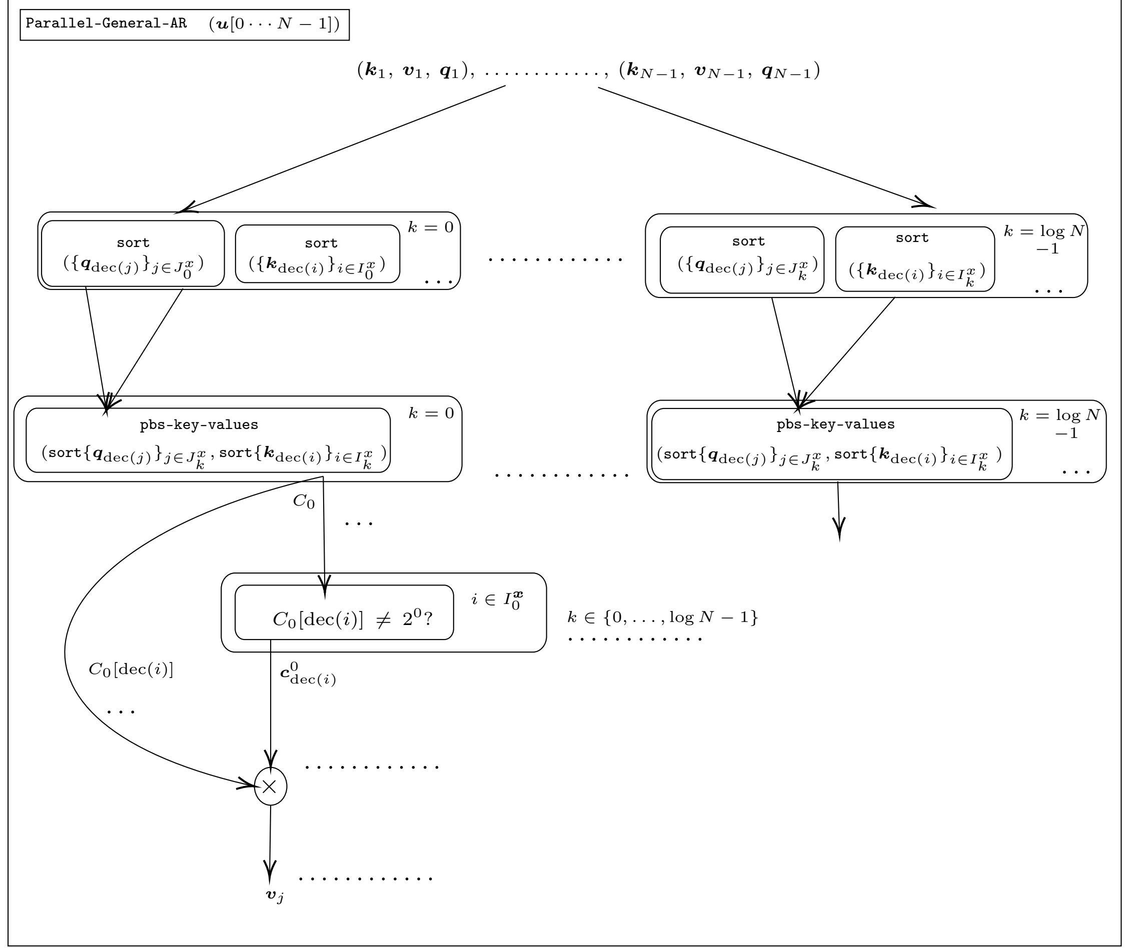


Figure 5: `Parallel-MQAR`$(\boldsymbol{u}[0 \cdots N-1])$ as a circuit that includes sorting networks and the circuit for `pbs-key-values` as subroutines.

Proof. Directly applying Theorem D.18 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. □

D.7 Data-Dependent Convolutions

D.7.1 Introduction

In this section, we are again concerned with solving the multiple-query associative recall problem (Appendix D.6.1). However, in contrast to Appendix D.6.4, which yields a circuit that is unchanged and works for *all* inputs, we instead take the viewpoint of adapting the model with respect to the particular sequence that the model gets 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 $\boldsymbol{u}[0 \cdots 3N - 1] \triangleq \{(\boldsymbol{k}_0, \boldsymbol{v}_0, \boldsymbol{q}_0), \ldots, (\boldsymbol{k}_{N-1}, \boldsymbol{v}_{N-1}, \boldsymbol{q}_{N-1})\}$ with each $\boldsymbol{k}_i, \boldsymbol{v}_i, \boldsymbol{q}_i \in C$. Here, each token is embedded using the standard one-hot encoding in $\{0,1\}^c$ (i.e. we

assume $d = c$).[5] Our goal is again to check, for each $1 \leq i \leq N-1$, whether there exists $0 \leq j < i$ such that $\boldsymbol{q}_i \equiv \boldsymbol{k}_j$, and if so, output $\boldsymbol{v}_j$.
Here, we define the interaction distance between the ith query $\boldsymbol{q}_i$ and the matching key $\boldsymbol{k}_j$ as $i - j$. We then also assume that *number of distinct interaction distances* is bounded by t.

It turns out that exploiting the information on these distances requires the use of auto-correlation [5], which has an elegant underlying formulation. We will instead briefly introduce the relevant mathematical machinery in the context of elucidating the data-dependent model that we seek to develop in the sequel.

Auto-Correlations We introduce auto-correlations in the context of convolutions. Let $\tilde{\boldsymbol{u}}[t] := \boldsymbol{u}[-t]$, then the cross correlation of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ is given by

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

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

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

D.7.2 Coyote 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 Data-Dependent Kernels. Auto-correlation allows us to pick the top t distinct shifts, and we then use this information to define the data-dependent kernels. That is, we assume there exists a function $\texttt{Top}$ such that $\texttt{Top}\,(u \star u, t)$ 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:

$$\begin{aligned} \mathbf{h}^k(X) &\equiv \sum_{\ell \in [t]} X^{s_\ell + (\ell-1)\cdot N}, \\ \mathbf{h}^v(X) &\equiv \sum_{\ell \in [t]} X^{s_\ell - 1 + (\ell-1)\cdot N}. \end{aligned} \tag{12}$$

Here, we note that we only only have two kernels as we will assume $N' = tN$ 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.

Indices and Projections. Now, note that the input for the multiple-query associative recall problem $\boldsymbol{u} \in \{0,1\}^{3N \times d}$ has designated indices for the keys, queries, and values in the sequence. We gather these indices below:

$$\begin{aligned} \mathcal{K} &= \{i \in \{0, \dots, 3N-1\} \mid i \equiv 0 \mod 3\}, \\ \mathcal{V} &= \{i \in \{0, \dots, 3N-1\} \mid i \equiv 1 \mod 3\}, \\ \mathcal{Q} &= \{i \in \{0, \dots, 3N-1\} \mid i \equiv 2 \mod 3\},. \end{aligned} \tag{13}$$

The above help us define the following projections $\mathbf{K}, \mathbf{Q}, \mathbf{V} \in \{0,1\}^{3N \times d}$ of the input that we shall use below.

$$\begin{aligned} \mathbf{K}[i,:] &:= \begin{cases} \boldsymbol{u}[i,:] & \text{if } i \in \mathcal{K}, \\ \mathbf{0}^d & \text{otherwise} \end{cases}, \\ \mathbf{Q}[i,:] &:= \begin{cases} \boldsymbol{u}[i,:] & \text{if } i \in \mathcal{Q}, \\ \mathbf{0}^d & \text{otherwise} \end{cases}, \\ \mathbf{V}[i,:] &:= \begin{cases} \boldsymbol{u}[i,:] & \text{if } i \in \mathcal{V}, \\ \mathbf{0}^d & \text{otherwise} \end{cases}, \end{aligned} \tag{14}$$

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

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

Theorem D.29. *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 $\boldsymbol{u} \in \{0,1\}^{3N\times d}$, we get the data-dependent kernels as in (12) using auto-correlation of the input. We will now outline the following computations for the BASECONV layers:

$$\begin{aligned}\boldsymbol{y} &= \texttt{Linear}_{\mathbf{Q}}(\boldsymbol{u}) \odot \left(\mathbf{h}^K * \texttt{Linear}_{\mathbf{K}}(\boldsymbol{u})\right)\\ &= \mathbf{Q} \odot \left(\mathbf{h}^K * \mathbf{K}\right) \qquad (15)\\ \boldsymbol{z} &= \texttt{Linear}_{\mathbf{E}}(\boldsymbol{y}) \odot \left(\mathbf{h}^V * \texttt{Linear}_{\mathbf{V}}(\boldsymbol{u})\right)\\ &= \mathbf{E} \odot \left(\mathbf{h}^V * \mathbf{V}\right), \qquad (16)\end{aligned}$$

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

$$\mathbf{E}[i,:] := \texttt{Linear}_{\mathbf{E}}(\boldsymbol{y})[i,:] = \begin{cases}\mathbf{1}^d & \text{if } \exists\, j \in [d] \text{ such that } \boldsymbol{y}[i,j] = 1\\ \mathbf{0}^d & \text{otherwise}\end{cases}. \qquad (17)$$

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 (12) for $t = 1$ are given by

$$\begin{aligned}\mathbf{h}^k(X) &\equiv X^{s_1};\\ \mathbf{h}^v(X) &\equiv X^{s_1-1},\end{aligned} \qquad (18)$$

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

$$\begin{aligned}&\texttt{shift_down}(\boldsymbol{y}, s_1);\\ &\texttt{shift_down}(\boldsymbol{y}, s_1 - 1).\end{aligned} \qquad (19)$$

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 (15) and (16) as follows:

$$\boldsymbol{y} = \mathbf{Q} \odot \texttt{shift_down}(\mathbf{K}, s_1) \qquad (20)$$

$$\boldsymbol{z} = \mathbf{E} \odot \texttt{shift_down}(\mathbf{V}, s_1 - 1), \qquad (21)$$

We begin by examining $\boldsymbol{y}$ below:

$$\begin{aligned}\boldsymbol{y}[i,:] &= \left(\mathbf{Q} \odot \texttt{shift_down}(\mathbf{K}, s_1)\right)[i,:]\\ &= \mathbf{Q}[i,:] \odot \texttt{shift_down}(\mathbf{K}, s_1)[i,:] \qquad (22)\\ &= \left(\begin{cases}\boldsymbol{u}[i,:] & \text{if } i \in \mathcal{Q}\\ \mathbf{0}^d & \text{otherwise}\end{cases}\right) \odot \left(\begin{cases}\boldsymbol{u}[i-s_1,:] & \text{if } i - s_1 \in \mathcal{K},\\ \mathbf{0}^d & \text{otherwise}\end{cases}\right) \qquad (23)\\ &= \begin{cases}\boldsymbol{u}[i,:] \odot \boldsymbol{u}[i-s_1,:] & \text{if } i \in \mathcal{Q} \text{ and } i - s_1 \in \mathcal{K},\\ \mathbf{0}^d & \text{otherwise}\end{cases}\end{aligned}$$

Here, we use the fact that the Hadamard product is row-independent in (22), and the definitions of the projections from (14) in (23). Examining the jth entry, we get

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

That is, we can express

$$\boldsymbol{y}[i,:] = \begin{cases}\mathbf{e}_j & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K} \text{ and } \boldsymbol{q}_i \equiv \boldsymbol{k}_{i-s_1} \equiv \mathbf{e}_j\\ \mathbf{0}^d & \text{otherwise}\end{cases}. \qquad (24)$$

Consequently, as per the definition in (17), we get

$$\mathbf{E}[i,:] = \begin{cases}\mathbf{1}^d & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K} \text{ and } \boldsymbol{q}_i \equiv \boldsymbol{k}_{i-s_1}\\ \mathbf{0}^d & \text{otherwise}\end{cases} \qquad (25)$$

We can now finally specify the output $\boldsymbol{z}$ from (21) as follows:

$$
\begin{aligned}
\boldsymbol{z}[i,:] &= \left(\mathbf{E} \odot \texttt{shift_down}(\mathbf{V}, s_1 - 1)\right)[i,:] \\
&= \mathbf{E}[i,:] \odot \texttt{shift_down}(\mathbf{V}, s_1 - 1)[i,:] && (26) \\
&= \left(\begin{cases} \mathbf{1}^d & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K} \text{ and } \boldsymbol{q}_i \equiv \boldsymbol{k}_{i-s_1} \\ \mathbf{0}^d & \text{otherwise} \end{cases}\right) \odot \left(\begin{cases} \boldsymbol{u}[i - s_1 + 1,:] & \text{if } i - s_1 + 1 \in \mathcal{V}, \\ \mathbf{0}^d & \text{otherwise} \end{cases}\right) \\
& && (27) \\
&= \begin{cases} \boldsymbol{u}[i - s_1 + 1,:] & \text{if } i \in \mathcal{Q}, i - s_1 \in \mathcal{K}, i - s_1 + 1 \in \mathcal{V} \text{ and } \boldsymbol{q}_i \equiv \boldsymbol{k}_{i-s_1} \\ \mathbf{0}^d & \text{otherwise} \end{cases} \\
&= \begin{cases} \boldsymbol{v}_{i-s_1} & \text{if } \boldsymbol{q}_i \equiv \boldsymbol{k}_{i-s_1} \\ \mathbf{0}^d & \text{otherwise} \end{cases} && (28)
\end{aligned}
$$

Again, we use the fact that the Hadamard product is row-independent in (26), and the definitions of the projections from (14) in (27). 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 $t \le N$, we first increase the internal dimension so that the input to the kernels $\boldsymbol{u}' \in \mathbb{R}^{(3N \cdot t) \times d}$ in (12) and the projections $\mathbf{K}', \mathbf{Q}', \mathbf{V}' \in \mathbb{R}^{(3N \cdot t) \times d}$ are given by

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

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

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

In analogy with (19), we can then equivalently write

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

Similarly, we also have

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

That is, the argument for $t = 1$ now applies to each of the t shifts as we now have (*cf.* (15))

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

where, for each $\ell \in [t]$, we have (*cf.* (24))

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

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

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

where, for each $\ell \in [t]$, we have (*cf.* (25))

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

The output in the general case is then given by

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

where, for each $\ell \in [t]$, we have (*cf.* (28))

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

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

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

To recall, we retrieved the top t interaction distances of the input $\boldsymbol{u}$ using auto-correlation and defined the corresponding convolution kernels ((12)). We then shifted djown 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 3: Hyena [16] 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 4: 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 5: 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 6: 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 7: RWKV [15] 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 8: Simple Long Convolution [10] 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 9: 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 10: 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 11: 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	$B \times V \times N \times D$