©Copyright 2025
Yao-Fei Cheng

Beyond Memorization: Evaluating Length-Generalization in Transformer-based Language Models

Yao-Fei Cheng

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2025

Committee:

Shane Steinert-Threlkeld

Fei Xia

Program Authorized to Offer Degree:
Computational Linguistics

University of Washington

Abstract

Beyond Memorization: Evaluating Length-Generalization in Transformer-based Language Models

Yao-Fei Cheng

Chair of the Supervisory Committee:
Shane Steinert-Threlkeld
Department of Linguistics

Transformer-based large language models have made substantial progress in the NLP community. However, transformers have trouble with length generalization (i.e., extrapolating on different lengths than seen during training). Recently, (Zhou et al., 2024a) proposed the RASP-Generalization Conjecture to predict what tasks are length-generalizable using a few carefully handcrafted tasks in the mathematical domain. This work examines this conjecture by generating hundreds of synthetic tasks written in the shortest RASP-L programs. Our investigation does not support the conjecture. The tasks written in the shortest RASP-L programs are not length-generalizable. Furthermore, our analysis reveals that some unlength generalizable tasks are due to models not stopping generating. It can be easily fixed by using oracle length during evaluation as suggested in previous literature. Additionally, the analysis rejects induction head as the key factor for failure of length generalization as claimed in the previous findings. Despite our work not providing a precise explanation of transformers' length generalization capability, we show that previous claims cannot extend to other tasks rather than carefully handcrafted tasks.

TABLE OF CONTENTS

Page

List of Figures . . . ii

List of Tables . . . iii

Chapter 1: Introduction . . . 1

Chapter 2: Related Work . . . 3
- 2.1 Transformers . . . 3
- 2.2 Restrictred Access Sequence Processing Language (RASP) . . . 3
- 2.3 Induction Head . . . 4
- 2.4 Length Generalization . . . 5

Chapter 3: Methods . . . 6
- 3.1 RASP-L . . . 6
- 3.2 Data Generation . . . 7

Chapter 4: Experiments . . . 9
- 4.1 Data . . . 9
- 4.2 Experiment Details . . . 11

Chapter 5: Discussion . . . 13
- 5.1 Do shortest RASP-L written tasks length generalizable? . . . 13
- 5.2 Are transformers length-generalizable in the oracle setting? . . . 15
- 5.3 What about other tasks that cannot be length-generalizable in oracle settings? . . . 18

Chapter 6: Conclusion . . . 20

Appendix A: RASP-L grammars . . . 1

LIST OF FIGURES

Figure Number — Page

3.1 The flow chart for data generation. Synthesizing shortest RASP-L programs by combining RASP-L grammars (See details in Appendix A) arbitrarily via ULTK. . . . 7

3.2 The example of a generated task. We fed binary sequences into the program to generate output. Finally, we used produced output with corresponding input to train language models. . . . 8

4.1 The examples of tasks in different depths. . . . 11

4.2 Entropy histogram for tasks. . . . 12

5.1 The exact match accuracy in different depths for all tasks. The accuracy along the depth is the average accuracy of tasks in that depth. . . . 14

5.2 The accuracy in different depths with oracle length for all tasks. The accuracy along the depth is the average accuracy of tasks in that depth. . . . 16

5.3 Entropy in output sequences per depth. The black line indicates the tasks in depth six with lower entropy. . . . 17

5.4 The example of Vanilla and Mnemonics (Ebrahimi et al., 2024) sequences. There are N anchors inserted in an N-length sequence in the mnemonics setting. . . . 17

5.5 The exact match accuracy in different depths for all tasks with Mnemonics applied. The accuracy along the depth is the average accuracy of tasks in that depth. . . . 18

LIST OF TABLES

Table Number Page

4.1 Distribution of generated tasks based on depth. 10
4.2 Experimental hyperparameters. All experiments use AdamW optimizer. 10
5.1 The performance comparison between our implementation and (Zhou et al., 2024a). 15

ACKNOWLEDGMENTS

I would like to express my profound appreciation to my advisor, Prof. Shane Steinert-Threlkeld, at the University of Washington, for all the support and advice throughout my master's. He has always been inspiring whenever I talk to him, whether in a meeting or casual conversation. His research philosophy has motivated me to pursue a deeper understanding of research questions.

In the two years of study, I have been fortunate enough to work closely with Jeongyeob Hong, a classmate, a close friend, and a teammate. We have collaborated on several projects, including one workshop paper, which brought us to Bangkok, Thailand. His presentation skills and a keen eye for detail always amaze me. I believe our collaboration will bring us to many other places.

Finally, I thank my family for their warm company and full support throughout my studies. I would like to thank the friends I met in UW and Seattle. You enrich my life outside of the lab.

DEDICATION

to my grandparents in heaven, Huang, Shu-Ying, and Cheng, Jui-Tang

Chapter 1
INTRODUCTION

Transformer-based models have made substantial progress across various fields recently (Vaswani et al., 2017). In the natural language processing (NLP) community, transformer-based large language models (LLMs) have dominated the field due to their superior performance across diverse tasks (Touvron et al., 2023; Team et al., 2024; Brown et al., 2020). Furthermore, LLMs demonstrate tremendous potential to improve reasoning capacity by scaling up model parameter size and training data (Kaplan et al., 2020). However, it remains unclear whether transformer-based models exhibit their unprecedented improvements in reasoning skills through task learning or by memorizing training data patterns. Transformers' notoriously black-box characteristics make it challenging to directly answer questions such as *What do transformers learn?* or *Why do transformers fail on simple tasks like copy and paste in some cases?*

Recent work by Zhou et al. (Zhou et al., 2024a) proposed the RASP generalization conjecture: transformers learn length-generalizing[1] solutions when the target task can be expressed as a short program in RASP-L, a programming language that models transformer computations. They demonstrated this conjecture on several mathematical tasks, but testing has been limited to carefully handcrafted problems in narrow domains.

Our central research question is: *How does the RASP generalization conjecture perform when tested on a broader range of tasks, and what factors beyond program length influence generalization capability?* We conduct a systematic empirical study using binary sequence tasks—sequences of 0s and 1s—to isolate algorithmic complexity from vocabulary effects. While this constraint limits direct applicability to natural language, it enables controlled analysis of the relationship between RASP-L program length and generalization performance. We ensure all tasks achieve perfect accuracy on training lengths using six-layer transformers, establishing that models have learned the target

[1] We refer length-generalization to a capability on models can extrapolate on longer or shorter sequences than seen during training.

behavior.

Our findings reveal important nuances in transformer length generalization. While exact match evaluation shows 0% accuracy on out-of-distribution lengths, detailed analysis indicates that 13% of cases demonstrate successful algorithmic learning—models generate correct outputs but fail at appropriate termination (e.g., producing [A] [A] [B] [B] [B] . . . instead of [A] [A] [B]). This suggests that generation control and algorithmic learning represent separate challenges in length generalization.

We investigated failure cases that Zhou et al.'s induction head hypothesis cannot explain, where a substantial portion represents 87% of tasks in our evaluation set. According to Zhou et al., transformers fail at certain tasks because they lack "induction heads"—specialized attention mechanisms that identify where a current token appeared previously in the sequence and then use that information to predict what should come next. Our preliminary analysis suggests this explanation may be incomplete. We found several cases where the attention patterns showed successful token matching (indicating functional induction heads), yet the model still failed to generate the correct next token. These observations point to additional failure mechanisms beyond the simple presence or absence of induction heads. While these findings hint at additional contributing factors beyond induction head presence, further controlled experiments are needed to identify and validate alternative failure mechanisms.

Chapter 2
RELATED WORK

2.1 Transformers

Transformers (Vaswani et al., 2017) was introduced in 2017 to improve the training efficiency of recurrent neural networks (RNNs) (Elman, 1990; Hochreiter and Schmidhuber, 1997). Since then, the encoder-decoder architecture (Sutskever et al., 2014) with attention with attention mechanism has dominated the natural language processing (NLP) community. It extends from machine translation (MT) to other downstream tasks like question answering (QA). The following year, (Devlin et al., 2018) introduced a pre-training paradigm using a transformer encoder. Researchers found that pre-training with massive unlabeled data can help models acquire foundation knowledge. As a result, researchers can fine-tune downstream tasks with limited labeled data. Most recently, the pre-training paradigm has shifted from a pre-training transformer encoder to a decoder (Radford et al., 2018) because of its superior performance on various NLP benchmarks. Furthermore, researchers found that scaling up the model size and dataset for decoder-only performance will guarantee improved (Kaplan et al., 2020). This discovery has led to the recent trend of large language models (LLMs) (Brown et al., 2020; Team et al., 2024; Touvron et al., 2023).

2.2 Restrictred Access Sequence Processing Language (RASP)

Unlike traditional probability models, the mathematical explanation behind neural network models is challenging to interpret. Transformers are no exception. Its deeper layers and non-linear structures make it even harder to interpret from a mathematic perspective (Geshkovski et al., 2023). Previous literature on transformer interpretation is separated into different components. For example, attention scores (Vig, 2019), skip-connection (Voita et al., 2019; Chefer et al., 2021), and word embeddings (Geva et al., 2022; Dar et al., 2023).

Yet, it is still unclear what transformers learn only to rely on understanding components locally. In the parallel work, (Weiss et al., 2021) proposed the Restricted Access Sequence Processing Language

(RASP) language, a programming language that maps the basic components of a transformer-encoder—attention and feed-forward computation into simple primitives. In other words, it is a human-readable language that can be used to understand transformer computation. Specifically, RASP provides built-in functions such as *select* (creating selection matrices called selectors), *aggregate* (collapsing selectors and s-ops into a new s-ops), and *selector_width* (creating an s-op from a selector). Along with several elementwise operators reflecting the feed-forward sublayers of a transformer. Weiss et al. further demonstrated that some RASP programs performed similarly to the 'natural' solution found by trained transformers (Shown in Figures 4 and 5 in (Weiss et al., 2021)).

It has led researchers to further dive deeper into what algorithm transformers learn. For example, (Yang and Chiang, 2024) proposed a RASP variety, C-RASP, to understand soft attention computation in transformers. On the other hand, (Zhou et al., 2024a) proposed another RASP variety, RASP-L, to understand length generalization capability in transformers (see more detail in Section 2.4).

2.3 Induction Head

Besides scaling law brought by LLMs, researchers also observed in-context learning (Dong et al., 2024) in LLMs. Unlike previous training schemes, models require a certain amount of training data for acquiring knowledge; in-context learning only needs a few examples as references for LLMs to generate correct responses. Advanced methods such as chain-of-thought (CoT) prompting (Wei et al., 2022) decompose answers step by step, and it has been proved empirically it can improve accuracy for various benchmarks. It motivates an interesting research question: *Why does in-context learning work?* To response it, (Olsson et al., 2022) proposed an interesting observation. There is an attention head in charge of pattern matching, where the induction head will search previous tokens to try to find a similar pattern and copy the token to the next token. For example, [B] is highly likely to be predicted to be the next token in the sequence [A] [B] . . . [A] $\rightarrow$ because the subsequence [A] [B] has appeared in the early position of the sequence. The induction head figures out the subsequence pattern and applies this pattern to the next token prediction. Another interesting takeaway is that the induction head is based on context and does not memorize training data. It enables transformers to extrapolate well on out-of-distribution data (Song et al., 2024).

2.4 Length Generalization

Length generalization was first introduced in the context of neural networks (Lake and Baroni, 2018), where this task examinate if the model can extrapolate on longer sequences than seen during training in SCAN benchmark. It is an important task as it demonstrates the difference between symbolic and neural computation (Fodor and Pylyshyn, 1988). The initial result showed that this task is extremely challenging for end-to-end recurrent neural networks (Hochreiter and Schmidhuber, 1997). Similarly, it is an important yet challenging reasoning task for transformers (Xiao and Liu, 2023; Dziri et al., 2023; Zhou et al., 2024b; Lee et al., 2024; Qian et al., 2023). Previous research has attempted to perform systematic analyses of length generalization from various perspectives. For example, the decoding strategy (Newman et al., 2020), positional encoding (Kazemnejad et al., 2023; Shen et al., 2023), attention (Dubois et al., 2020; Press et al., 2022), and data format (Bueno et al., 2022). Most recently, (Zhou et al., 2024a; Huang et al., 2025) turned this generalization task into a chance to understand what algorithms transformers learn. Specifically, if transformers cannot extrapolate well on unseen length sets on the task, transformers might memorize the training data, not learning the task. They argue that tasks can be written in RASP-L, a RASP variety language, in a next token prediction way, which is length generalizable (see detail in Section 3.1.1). On the other hand, tasks such as addition, which is hard to implement in RASP-L, are not length generalizable. However, they only examined a few handcrafted tasks. In this work, we aim to examine a wide range of tasks that can be written in RASP-L, as previous work only examined a few carefully handcrafted tasks. We follow (Zhou et al., 2024a)'s core library[1] strictly to compose our grammar and generate hundreds of synthetic tasks for examining (see more detail in Section 3.2).

[1]https://github.com/apple/ml-np-rasp

Chapter 3
METHODS

3.1 RASP-L

RASP-L is a RASP variety language proposed by (Zhou et al., 2024a), where it was designed to represent auto-regressive decoder-only transformers with stricter rules compared to RASP. More specifically, RASP-L programs can only produce output sequences of the same length as the input sequence. For example, the task of *Mode*, where the task is to identify the most frequent element in the sequence: [A] [B] [B] [A] [B] [B] $\rightarrow$ [B] [B] [B] [B] [B] [B][1]. Additionally, arbitrary arithmetic involving indices is not allowed in RASP-L; only simple operations such as order comparison, predecessor, and successor are allowed. The RASP-L core library[2] consists of built-in functions to represent causal transformers with the restrictions mentioned above (See more details in Appendix A). For instance, the KQV function simulates the computation of causal attention in a transformer decoder layer. With these built-in functions, one is able to understand the transformers' internal mechanism by analyzing arbitrary RASP-L written tasks.

3.1.1 RASP-Generalization Conjecture

(Zhou et al., 2024a) provided a conjecture to predict a decoder-only transformer is likely to length-generalize if the following condition holds. We introduce the conjecture in this subsection and explain how we follow and generate data in the next subsection.

1. **Realizability.** If a decoder-only transformer on all input lengths can represent the task. This means that transformers perform perfectly when in-distribution data splits.

2. **Simplicity.** If the task can be implemented in RASP-L.

[1] It is an important detail as the original analysis in (Zhou et al., 2024a)'s work did not follow this restriction.

[2] https://github.com/apple/ml-np-rasp

3. **Diversity.** If no shorter RASP-L program exists, it agrees to the task in-distribution but not out-of-distribution. This means the RASP-L implementation should be the shortest and work on both in-/out-of-distribution.

3.2 Data Generation

This work aims to generate synthetic tasks using RASP-L for examination. In the following sections, we introduce our data generation pipeline.

3.2.1 Data Generation Pipeline

To fulfill the conjecture, we first generate tasks that are implemented in the shortest RASP-L program (**Diversity**), present in Figure 3.1. We then produce the input and output sequences using the shortest RASP-L in the arbitrary length (**Realizability**), shown in Figure 3.2. Furthermore, we ensure models have perfect in-distribution results (**Realizability**, see details in Section 5.1).

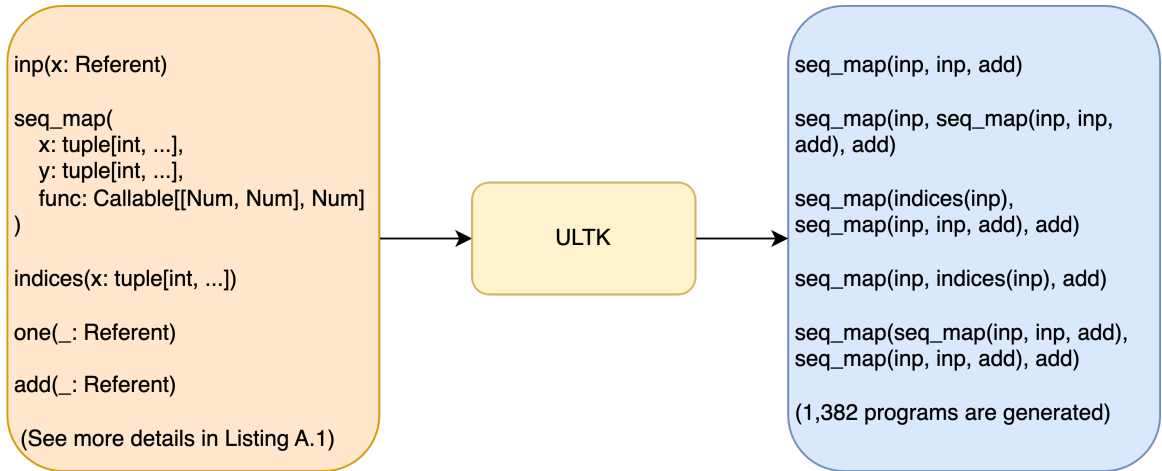


Figure 3.1: The flow chart for data generation. Synthesizing shortest RASP-L programs by combining RASP-L grammars (See details in Appendix A) arbitrarily via ULTK.

As shown in Figure 3.1, we synthesize RASP-L programs (blue block on the right-hand side) by combining the existing RASP-L grammars (orange block on the left-hand side) arbitrarily. For

instance, a typical RASP-L function such as “seq_map(x, y, func)" applies an operation “func" with an array “y” to each element in an input array “x," producing a transformed output sequence.

However, it is not guaranteed to be the shortest one. To address this issue, we used an in-house toolkit, ULTK[3] (Imel et al., 2025) to achieve it. ULTK evaluates the generated sequences across the entire space of relevant input sequences. Expressions that produce the same outputs across all sequences are considered equivalent; only the shortest expression is retained among these. This process results in a minimal representation of the input-output behavior, analogous to Boolean minimization in human concept learning (Feldman, 2000).

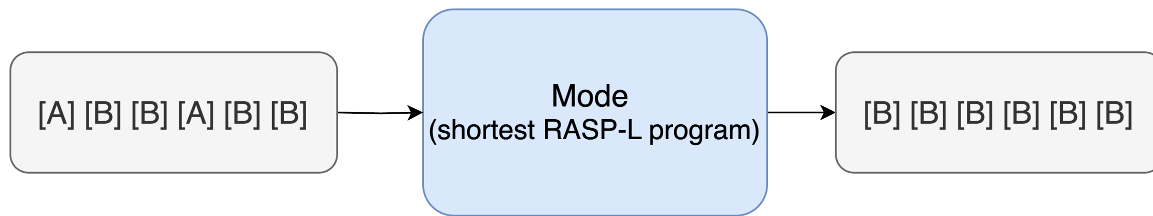


Figure 3.2: The example of a generated task. We fed binary sequences into the program to generate output. Finally, we used produced output with corresponding input to train language models.

Once we have synthesized tasks, we fed binary sequences into the programs to generate corresponding output to train language models, shown in Figure 3.2. One difference between our work and Zhou et al. is that we limit our input sequence to binary symbols as it is common in theoretical computer science and formal analysis (Strobl et al., 2024). We believe that simple binary symbols are enough for the task, as we do not want our analysis to depend on the complexity of input symbols.

[3]https://github.com/CLMBRs/ultk

Chapter 4
EXPERIMENTS

4.1 Data

As mentioned in Section 3.2, we generated binary synthetic tasks[1] using RASP-L and ULTK. As a result, $1,382$ tasks are generated. Additionally, we consider expressions with different depths, as illustrated in Figure 4.1. For example, a depth three expression, *sel_width(select(inp, inp, intersection))*, illustrated in Figure 4.1a, takes a parameter, which is a depth two expression, *select(inp, inp, intersection)*, where it takes three depth one expression. We posit that depth may be a factor that influences the generalization capabilities as previous studies have suggested models' inductive biases influence its capability to handle sequences in different structures (Hewitt et al., 2020; McCoy et al., 2020). Due to the computing resource constraint, we only sampled some tasks among all the functions according to depths. Specifically, we sampled up to 30 tasks per depth (depths range from one to seven). Table 4.1 shows the distribution of these tasks based on depth before sampling.

To produce input-output sequences for model training, we uniformly sampled input sequences from all of the possible combinations within the range of length. For example, in a range of 2, the possible inputs are: 0, 1, 00, 11, 01, and 10. We sampled some of the input from these combinations to form our training and testing sets.

Once we have input sequences, we then feed the input into RASP-L written programs to generate output sequences. Specifically, we generated 10k input and output sequences for each task, uniformly ranging from twenty-one to thirty, as the training data. For in-distribution testing, we generated 1k samples under the same setting. Similarly, we generated 1k samples for different length splits in the out-of-distribution setting. To be specific, we consider sequences that are shorter and longer than training data, namely L1-10, L11-20, L41-50, and L91-100, where the sequences are limited from one to ten, eleven to twenty, forty-one to fifty, and ninety-one to a hundred.

Additionally, to avoid potential memorization, we filtered out tasks whose generated sequences

[1] We use tasks to refer to the final generated expressions while expression as the function in RASP-L.

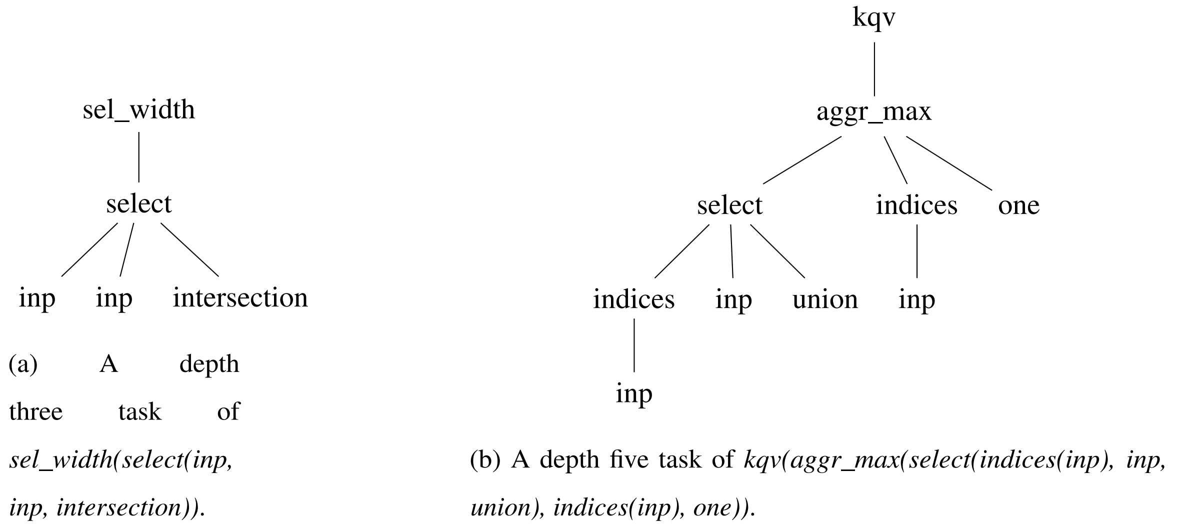
Depth	# of tasks
1	1
2	5
3	17
4	51
5	462
6	182
7	664
Total	1,382

Table 4.1: Distribution of generated tasks based on depth.

Model Size	Train Iter	Context Len	Batch Size	Learning Rate
6 layer; 8 head; 512 emb	30k	512	64	0.001

Table 4.2: Experimental hyperparameters. All experiments use AdamW optimizer.

contained too many repeated output sequences. As we can see in Figure 4.2, about six hundred tasks have low entropy in their output sequences. This means that identical output sequences appear in the corpus often. As a result, we sorted all the tasks according to entropy and only selected the top 30 tasks with the highest entropy for each task, resulting in 137 tasks being used in this work. The difference between our work and (Zhou et al., 2024a) is that our data is generated via raw sequence-to-sequence RASP-L program so that the output sequences are always the same length as the input sequences. On the other hand, (Zhou et al., 2024a) generated data via their handcrafted RASP-L program in the next prediction generation fashion.



(a) A depth three task of *sel_width(select(inp, inp, intersection))*.

(b) A depth five task of *kqv(aggr_max(select(indices(inp), inp, union), indices(inp), one))*.

Figure 4.1: The examples of tasks in different depths.

4.2 Experiment Details

We followed (Zhou et al., 2024a)'s implementation configuration in our study. Unlike Zhou et al., we did not use different hyperparameters for each task. Instead, we used one set of hyperparameters for all the tasks, shown in Table 4.2. One important detail we followed from (Huang et al., 2025) is that they tried to mimic the computations in the current LLMs'. An input x of length $|x| = k \leq N$ using positional encodings $p_{1+o}, \ldots, p_{k+o}$ where o is an offset such that $k + o \leq N$. This way, transformers were required to correctly perform the task independently of the offset $o \geq 0$ during the training. We trained the models as LMs on the generated data, using cross-entropy loss at each position. We implemented our custom LMs using HuggingFace (Wolf et al., 2020) and open pre-trained transformer (OPT) (Zhang et al., 2022) language model.[2]

[2]All of the models that we used are trained from scratch. We used OPT implementation as the default setting.

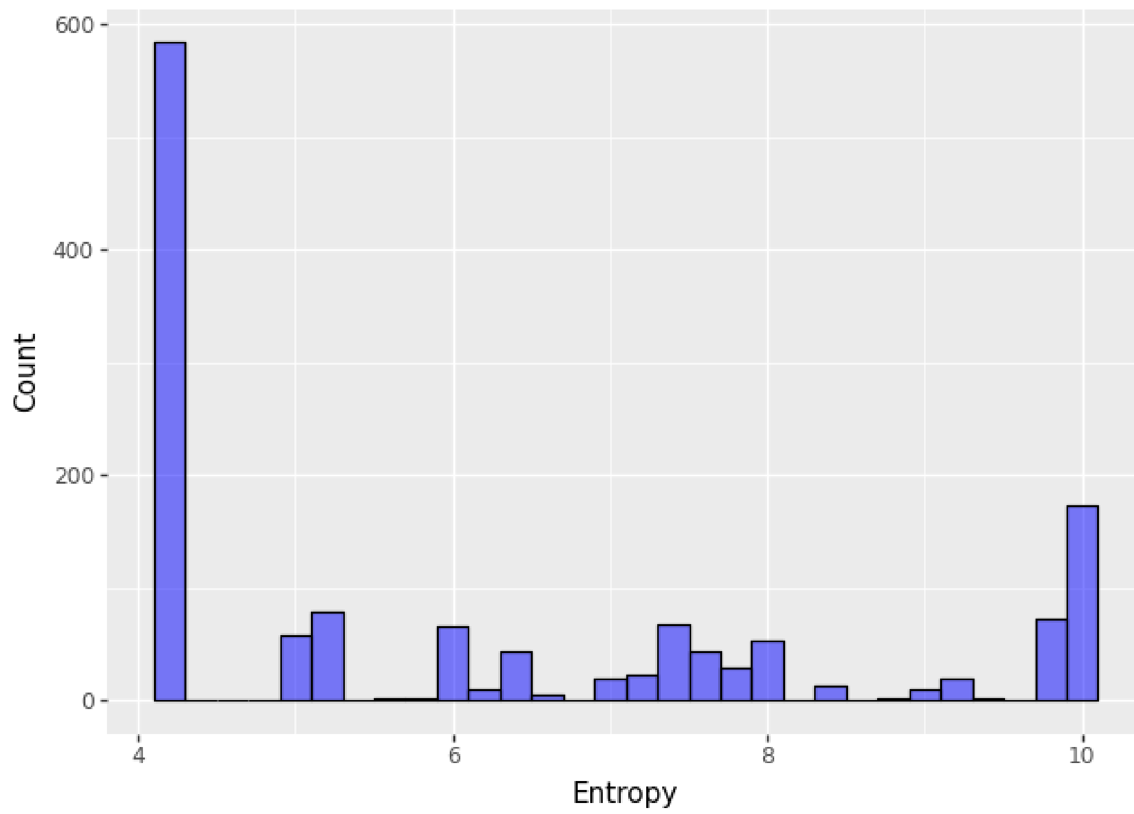


Figure 4.2: Entropy histogram for tasks.

Chapter 5

DISCUSSION

5.1 Do shortest RASP-L written tasks length generalizable?

Transformers perform perfectly in the in-distribution setting, shown in Figure 5.1, where models almost achieve accuracy 100% on L1-10 and L1-20. Now, we examine if tasks written in the shortest RASP-L are length generalizable. The short answer is NO. The tasks that we examine include simple tasks (e.g., copying and pasting) and some really complicated tasks that even humans cannot understand (usually with a deep depth). Those tasks are not length generalizable, as shown in Figure 5.1. We examine models in various length splits and calculate the average accuracy (i.e., y-axis) of tasks in each depth (i.e., x-axis); however, as we can see, the results are completely failed, achieving almost zero average across all tasks in splits of L21-30, L41-50, and L91-100.

Because the results contradicted the (Zhou et al., 2024a)'s claim, we performed another examination on tasks they mentioned in their work. We picked the task of *Mode* (i.e., identify the most frequent element in the input sequence) and summarized our reproduction in Table 5.1. We chose this task because we found an interesting implementation choice from Zhou et al.. As mentioned in Section 3.1, RASP-L programs task the input sequence and generate an output sequence of the same length as the input sequence. However, Zhou et al. implemented it in another way. Their implementation only generates a token as the output sequence. To be specific, their choice of implementation is: [A] [B] [B] [A] [B] [B] $\rightarrow$ [B]. On the other hand, we follow the restriction and implement it in this way: [A] [B] [B] [A] [B] [B] $\rightarrow$ [B] [B] [B] [B] [B] [B]. As a result, different implementation of the task of *Mode* influences the length generalization capabilities (in Section 5.2).

We show the performance difference between ours and Zhou et al. (implemented by our own) in Table 5.1. It shows that data formats influence the results. Transformers have a hard time generating repeated tokens in arbitrary lengths. On the other hand, always generate one token as the output can be extrapolated despite the input length being out-of-distribution (OOD). We posit that this issue is caused by transformers not knowing when to stop generating tokens. To verify our assumption, we

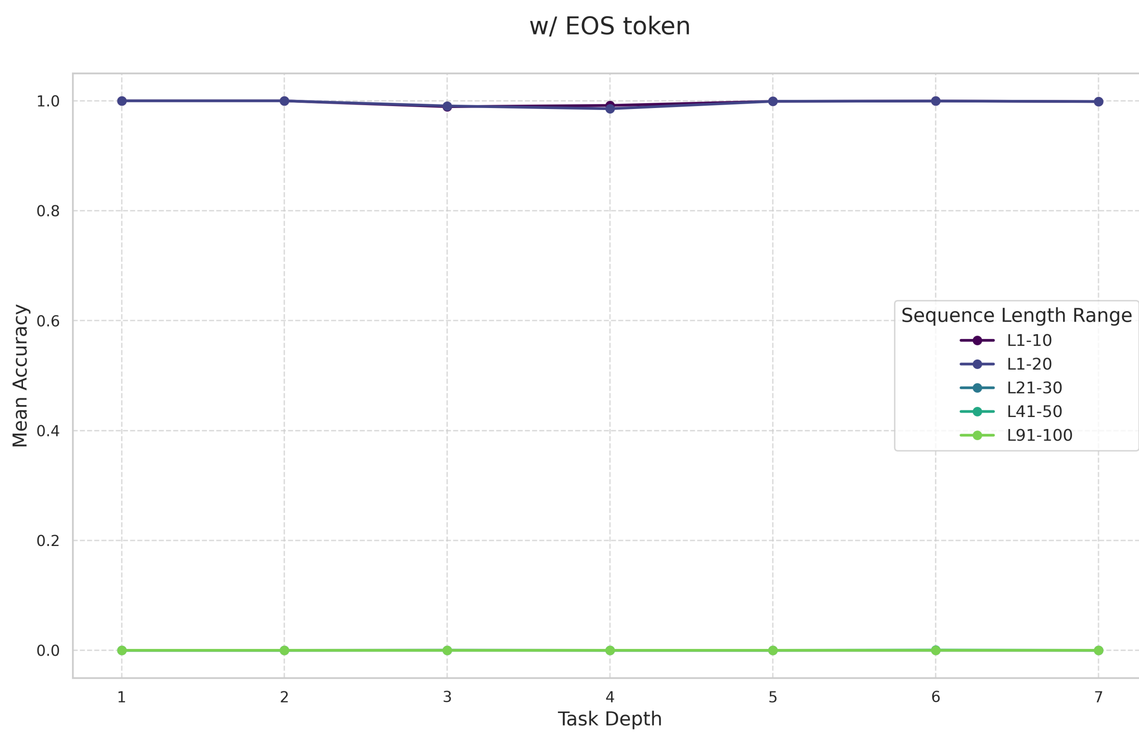


Figure 5.1: The exact match accuracy in different depths for all tasks. The accuracy along the depth is the average accuracy of tasks in that depth.

Task	L1-10	41-50	91-100
Mode (Zhou et al.)	1.00	0.98	0.92
Mode (Ours)	1.00	0.00	0.00
Mode (Ours) + Oracle	1.00	0.93	0.90

Table 5.1: The performance comparison between our implementation and (Zhou et al., 2024a).

calculate the accuracy of our *Mode* implementation with oracle length, where we forced the output sequences to be the same input length. The result in the final row of Table 5.1 demonstrates strong evidence that models can generate correct content, but it has trouble stopping.

5.2 Are transformers length-generalizable in the oracle setting?

Our result in Section 5.1 aligns with previous finding (Newman et al., 2020). Transformers have trouble generating end-of-sentence (EOS) tokens; therefore, transformers would generate repeated arbitrary tokens after a certain length (usually the answer's length). We further ask a research question: are transformers' length-generalizable in the oracle setting?

We follow (Newman et al., 2020) to train decoder-only transformers without EOS token using the same tasks we used in Section 5.1. In the evaluation phase, we used the oracle length to calculate the exact match accuracy, present in Figure 5.2. The in-distribution results follow the previous trend that performs perfectly, achieving almost 100% in all depths. Similarly, the results on OOD splits do not change much and remain at zero accuracies, except for tasks in depth six. To explore further why only tasks in depth six have performed differently, we tried to distinguish the difference between depths. As a result, we found that the entropy of output sequences in depth six is particularly low compared to other depths, shown in Figure 5.3. Meaning that there are more repeated tokens in the output sequences. For example, the output would be more likely to be all the same repeated tokens, such as [A] [A] [A] [A] [A] [A]. In this case, using oracle length in evaluation can vastly improve the exact match accuracy, as mentioned in previous sections.

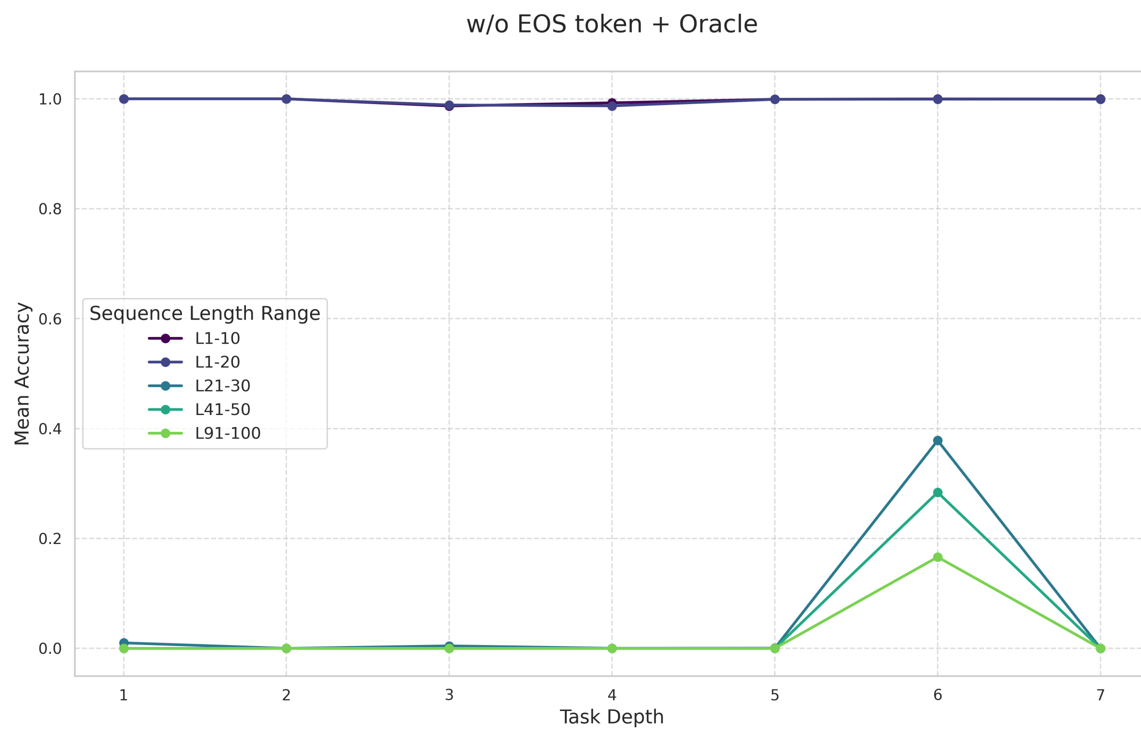


Figure 5.2: The accuracy in different depths with oracle length for all tasks. The accuracy along the depth is the average accuracy of tasks in that depth.

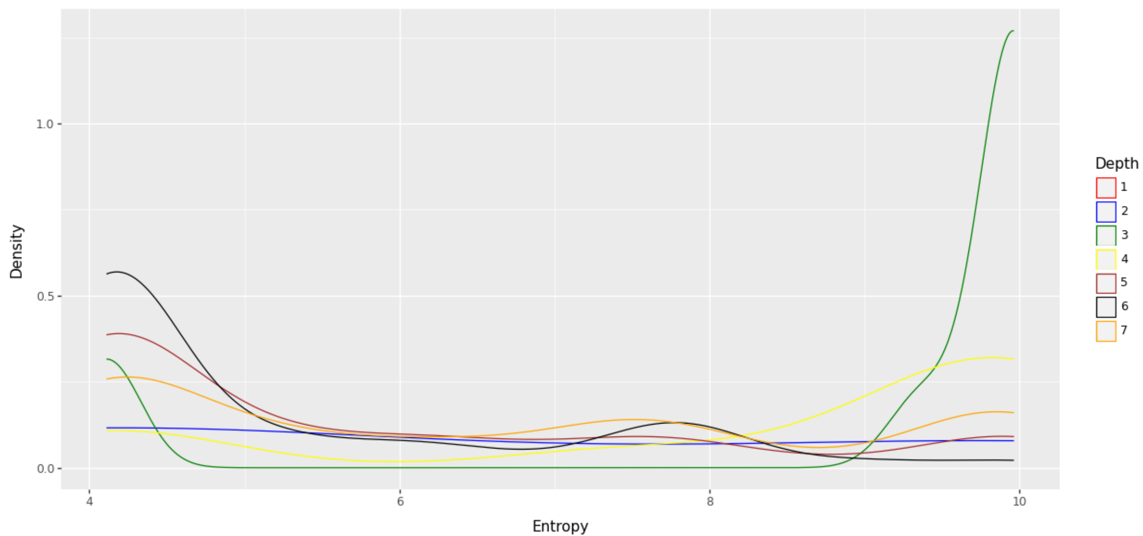


Figure 5.3: Entropy in output sequences per depth. The black line indicates the tasks in depth six with lower entropy.

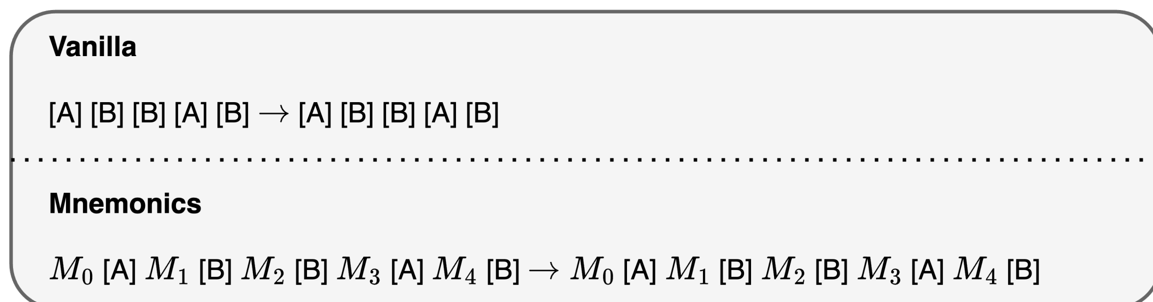


Figure 5.4: The example of Vanilla and Mnemonics (Ebrahimi et al., 2024) sequences. There are N anchors inserted in an N-length sequence in the mnemonics setting.

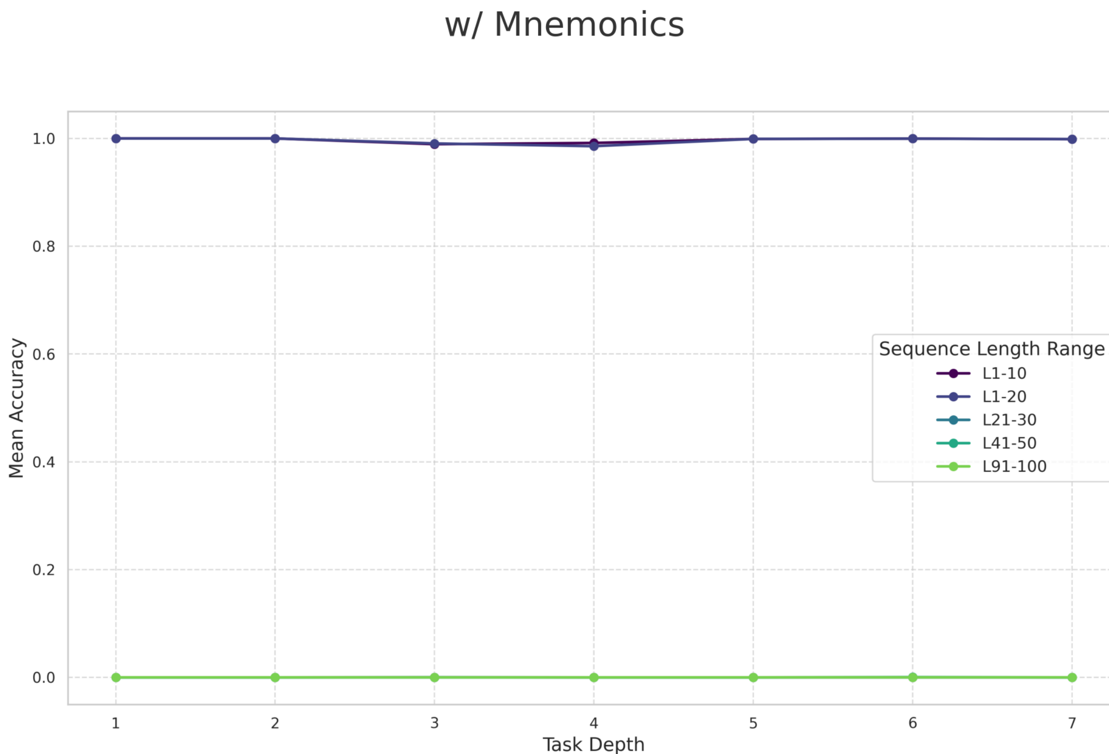


Figure 5.5: The exact match accuracy in different depths for all tasks with Mnemonics applied. The accuracy along the depth is the average accuracy of tasks in that depth.

5.3 What about other tasks that cannot be length-generalizable in oracle settings?

Although entropy in output sequences sheds light on why some tasks cannot be length-generalizable, some tasks remain unexplained (tasks in depths other than six in our case). As mentioned in (Zhou et al., 2024a), they posit tasks such as copying and pasting with repeated tokens (e.g., [A] [B] [C] [B] $\rightarrow$ [A] [B] [C] [B]), failed on length-generalization is due to induction head (Olsson et al., 2022). The repeated tokens cause transformers difficulty finding similar patterns in the previous sequences. For example, in the task of copy and paste (only binary symbol), [A] [B] [A] [B] [B] [B] $\rightarrow$ [A] [B] [A] [B] [B] [B]. The subsequence [A] [B] has appeared in the early position of the sequences many times; transformers get confused about what patterns need to be copied and use it to predict the next token.

(Ebrahimi et al., 2024) found this particular issue can be remedied by adding **anchors** to the sequences so that transformers can locate the patterns using these anchors, as illustrated in Figure 5.4, inserting N anchors, $M_0, \ldots, M_N$, to a length-N sequence. We follow this method, Mnemonics, by randomly inserting unique tokens that we sampled from a nltk English word corpus[1] to all of the tasks that we used in previous sections. To ensure there is no out-of-vocabulary (OOV) happening during evaluation on longer splits (e.g., L91-100), we first sampled a hundred unique English words from nltk before we applied Mnemonics to each task. We then follow the previous setting (see detail in Chapter 4) to re-train transformers with Mnemonics. However, the results remain unchanged, as we observed in Figure 5.5. Mnemonics do not help to improve poor length generalization capability in our case. The main difference between our experiment and Ebrahimi et al. is that they fine-tuned pre-trained transformers with Mnemonics; however, our transformers were trained from scratch. Additionally, they only tested Mnemonics on a few arithmetic tasks. These differences might influence the final results. Yet, our empirical evidence shows that there are other factors that influence length generalization capability besides induction head. We plan to leave further exploration of our synthetic tasks in our future study.

---

[1] https://www.nltk.org/howto/corpus.html

Chapter 6
CONCLUSION

In this work, we argued against the RASP-Generalization Conjecture. Our results demonstrate that transformers struggle with tasks written in the shortest RASP-L program. We first proposed to generate a massive amount of synthetic tasks written in the shortest RASP-L programs. Then examine decoder-only transformers on length-generalization capability with these synthetic data. We found some tasks failed on length generalization due to transformers do not know how to stop generating. This issue can be fixed by oracle length. However, the reason why other tasks are not length generalizable remains unknown. We attempted to equip induction head capability in transformers by adding anchors in sequences. However, it does not work as expected. We will leave the further investigation in the future work.

LIMITATION

This work is limited in several ways. The main limitation stems from the dependency on the RASP-L framework; therefore, the possible tasks are limited to combining the existing core library. Furthermore, in previous work (Zhou et al., 2024a), they handcrafted the next token prediction for every task, which is impossible to implement in a general form of implementation in the pipeline of data generation. Finally, unlike the previous implementation, it ignores the Python arguments and uses arbitrary arguments to generate expressions. Our implementation can only generate limited expressions due to this restriction. We plan to increase the flexibility in the next version to have more expressions for analysis.

Appendix A

RASP-L GRAMMARS

In this section, we provide the RASP-L grammars that we used to generate tasks in this work, shown in listing A.1.

```
# Type annotated RASP_L program, modified from https://github.com/apple/
    ml-np-rasp
# Original Liscense from Apple

from jaxtyping import Num
from typing import Callable
from ultk.language.semantics import Referent


global start
start = tuple[int, ...]


def inp(x: Referent) -> tuple[int, ...]:
    """
    Return 1d tuple of input sequence of ints.

    Main input function that transforms ultk vocab to valid
    input for sequence operations or selectors

    Parameter: x is any Referent based on SYMBOLS value in plug.py of
        len <= MAX_LEN
    Precondition: must be a Referent object as defined in ultk.langauge.
        semantics
    """
```

```
23      result= tuple(int(value) for value in x.name)
24      return result
25      #return np.array(result, dtype=int)
26
27
28  def zero(_: Referent) -> Num:
29      """
30      Return the int 0
31
32      Allows for the int 0 to be passed to sequence operations
33      such as full() and tok_map().
34
35      Parameter _: any sequence of symbols
36      Precondition: Referent
37      """
38      return 0
39
40
41  def one(_: Referent) -> Num:
42      """
43      Return the int 1
44
45      Allows for the int 1 to be passed to sequence operations
46      such as full() and tok_map().
47
48      Parameter _: any sequence of symbols
49      Precondition: Referent
50      """
51      return 1
52
53
54  def add(_: Referent) -> Callable[[Num, Num], Num]:
55      """
```

```
56      Return a Callable function that adds two ints.
57  
58      Initializes a 2-ary addition function to
59      be passed as an argument for func parameter
60      in tok_map, seq_map_int, kqv
61  
62      Parameter _: any sequence of symbols
63      Precondition: Referent
64      """
65      return lambda n, m: n + m
66  
67  
68  def subtract(_: Referent) -> Callable[[Num, Num], Num]:
69      '''
70      Return a Callable function that subtracts two ints.
71  
72      Initializes a 2-ary subtraction function to
73      be passed as an argument for func parameter
74      in tok_map, seq_map_int, kqv
75  
76      Parameter _: any sequence of symbols
77      Precondition: Referent
78      '''
79      return lambda n, m: n - m
80  
81  
82  def multiply(_: Referent) -> Callable[[Num, Num], Num]:
83      '''
84      Return a Callable function that multiplies two ints.
85  
86      Initializes a 2-ary multiplication function to
87      be passed as an argument for func parameter
88      in tok_map, seq_map_int, kqv
```

```
89
90      Parameter _: any sequence of symbols
91      Precondition: Referent
92      '''
93      return lambda n, m: n * m
94
95
96  def is_equal(_: Referent) -> Callable[[Num, Num], bool]:
97      """
98      Return a Callable function that detects if two ints are equal.
99
100     Initializes a 2-ary equal operator to
101     be passed as an argument for func parameter
102     in tok_map, seq_map_int, kqv
103
104     Parameter _: any sequence of symbols
105     Precondition: Referent
106     """
107     return lambda n, m: (n == m)
108
109
110 def less_than(_: Referent) -> Callable[[Num, Num], bool]:
111     """
112     Return a Callable function that detects
113     if one int is smaller than another.
114
115     Initializes a 2-ary less than operator to
116     be passed as an argument for func parameter
117     in tok_map, seq_map_int, kqv
118
119     Parameter _: any sequence of symbols
120     Precondition: Referent
121     """
```

```
    return lambda n, m: (n < m)


def intersection(_: Referent) -> Callable[[Num, Num], bool]:
    """
    Return a Callable function that functions as an
    'and' operator on ints.

    Initializes a 2-ary and operator to
    be passed as an argument for func parameter
    in tok_map, seq_map_int, kqv

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return lambda n, m: (n and m)


def union(_: Referent) -> Callable[[Num, Num], bool]:
    """
    Return a Callable function that functions as an
    'or' operator on ints.

    Initializes a 2-ary or operator to
    be passed as an argument for func parameter
    in tok_map, seq_map_int, kqv

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return lambda n, m: (n or m)


```

```
155 def xunion(_: Referent) -> Callable[[Num, Num], bool]:
156     """
157     Return a Callable function that functions as an
158     'xor' operator on ints.
159 
160     Initializes a 2-ary exclusive or operator to
161     be passed as an argument for func parameter
162     in tok_map, seq_map_int, kqv
163 
164     Parameter _: any sequence of symbols
165     Precondition: Referent
166     """
167     return lambda n, m: ((n or m) and not (n and m))
168 
169 
170 ## np-rasp core
171 def indices(x: tuple[int, ...]) -> tuple[int, ...]:
172     """
173     Return indices of the input.
174 
175     Implements sequence operation that returns
176     an array representing the input array's indices.
177 
178     Parameter x: 1d tuple of ints
179     Precondition: 1d tuple of at least one int
180     """
181     return tuple(range(len(x)))
182 
183 
184 # An array of shape x filled with const
185 def full(x: tuple[int, ...], const: Num) -> tuple[int, ...]:
186     """
187     Return a 1d len(x)-item tuple of const values.
```

```
188
189     Implements sequence operation that takes an input
190     1d tuple of length n and returns a 1d tuple of
191     length = n where each value = const.
192
193     Parameter x: a 1d tuple of ints
194     Precondition: 1d tuple of ints
195
196     Parameter const: an int value instantiated by zero(), one(), two()
197     Precondition: must be an int
198     """
199     return tuple(const for _ in range(len(x)))
200
201
202 # Apply func into elements of array x and return into integer array
203 def tok_map(x: tuple[int, ...],
204     const: Num,
205     func: Callable[[Num, Num], Num]
206 ) -> tuple[int, ...]:
207     """
208     Return a 1d tuple of int outputs of a 2-ary Callable
209
210     Implements sequence operation that takes a 1d tuple of ints
211     and applies the same function with the same second parameter
212     value to each element.
213
214     Parameter x: 1d tuple of ints
215     Precondition: 1d tuple of ints
216
217     Parameter const: an int value instantiated by zero(), one(), two()
218     Precondition: must be an int
219
220     Parameter func: a function that takes two ints and returns an int,
```

```
        typically add, subtract,
221     Precondition: a Callable with two int parameters
222     """
223     return tuple(func(xi, const) for xi in x)
224
225
226 # Apply func into elements pair of array x and y, and return into an
      integer array
227 def seq_map(
228     x: tuple[int, ...],
229     y: tuple[int, ...],
230     func: Callable[[Num, Num], Num]
231 ) -> tuple[int, ...]:
232     """
233     Return a 1d tuple of int outputs of a 2-ary Callable
234
235     Takes 2 int tuples, applies an element wise operation, and
236     returns the output, a tuple of ints.
237
238     Parameter x: 1d tuple of ints
239     Precondition: a tuple of at least one int
240
241     Parameter y: 1d tuple of ints
242     Precondition: a tuple of at least one int
243
244     Parameter func: a 2-ary int operation
245     Precondition: a Callable with two int parameters
246     """
247     return tuple(func(xi, yi) for xi, yi in zip(x, y))
248
249
250 # Creates selection matrix A, applies pred to compare the elements of k
      and q
```

```
251 # If causal, compare the ealier indice
252 # def select(
253 #    k: tuple[int, ...], q: tuple[int, ...], pred: Callable[[int, int],
        int], causal: bool = True
254 # ) -> tuple[tuple[bool, ...], ...]:
255 def select(k: tuple[int, ...],
256     q: tuple[int, ...],
257     pred: Callable[[Num, Num], bool]
258 ) -> tuple[tuple[bool, ...], ...]:
259     """
260     Return a 2d tuple of the element-wise outputs of a 2-ary Callable
261 
262     Takes 2 1d int tuples, applies an element wise operation, and
263     returns the output as a tuple of int-cast bools of shape (len(q),len
           (k)).
264 
265     Parameter x: any 1d tuple of ints
266     Precondition: 1d tuple of at least one int, len(x) == len(y)
267 
268     Parameter y: a 1d tuple of ints
269     Precondition: a 1d tuple of at least one int, len(x) == len(y)
270 
271     Parameter func: a 2-ary comparison, returns a bool
272     Precondition: a Callable with two int parameters that returns a bool
273     """
274     s = len(k)
275     A= [[0] * s for _ in range(s)]
276     for qi in range(s):
277         for kj in range(s):  # k_index <= q_index if causal
278             A[qi][kj] = bool(pred(k[kj], q[qi]))
279     return tuple(tuple(item) for item in A)
280 
281 
```

```
282 def sel_width(A: tuple[tuple[bool, ...], ...]) -> tuple[int, ...]:
283     """
284     Return a 1d tuple of ints, where the ith value represents the
285     ith row's width in 2d tuple A
286 
287     The return is a 1d tuple of ints where the ith value represents
288     the A's ith row's width. Width here is the number of tuple
289     positions where the value is 1 (int cast T).
290 
291     Parameter A: a 2d torch tuple of int cast bools with n rows and m
        columns
292     Precondition: a 2d torch tuple, values must be int 0 or int 1
293     """
294     result= []
295     for row in range(len(A)):
296         newval= sum(A[row])
297         result.append(newval)
298     return tuple(result)
299 
300 
301 # calculate the mean of selected values
302 def aggr_mean(_: Referent) -> Callable[[tuple[tuple[bool, ...], ...],
    tuple[int, ...], Num], tuple[int, ...]]:
303     """
304     Return a Callable function that performs a mean reduction similar to
         aggr_mean.
305 
306     This function is initialized to be passed as an argument for the `
        reduction` parameter
307     in functions like aggr and kqv.
308 
309     Parameter _: any sequence of symbols
310     Precondition: Referent
```

```
311
312         Return a 1d tuple of ints, where the ith value is the mean of Av
                's ith row.
313
314         Takes the dot product of 2d tuple A with the 1d tuple v, then
                divides
315         the ith value in the dot product's output array by the width
316         of the ith row of A. If the ith row of A has width 0, the
317         mean is replaced by the default value.
318
319         Parameter A: a 2d tuple of int cast bools with n rows and m
                columns
320         Precondition: a 2d torch tuple, values must be int 0 or int 1
321
322         Parameter v: a 1d tuple of ints, len(v) == m
323         Precondition: a 1d tuple of ints where len(v) == A.shape[1]
324
325         Parameter default: the value used to replace any instances of
                divide by zero
326         Precondition: an int
327     """
328     return lambda A, v, default=0: tuple(
329         (sum(row[i] * v[i] for i in range(len(v))) // sel_width(A)[idx]
                if sel_width(A)[idx] != 0 else default)
330         for idx, row in enumerate(A)
331     )
332
333
334 # calculate the maximum of selected values
335 def aggr_max(_: Referent) -> Callable[[tuple[tuple[bool, ...], ...],
        tuple[int, ...], Num], tuple[int, ...]]:
336     """
337     Return a Callable function that performs a max reduction over a 2d
```

```
        tuple.
338
339     This function is initialized to be passed as an argument for the
340     `reduction` parameter in functions like aggr and kqv.
341
342     Parameter _: any sequence of symbols
343     Precondition: Referent
344
345         The callable takes a 2d tuple A and a 1d tuple v as parameters
346         and returns a 1d tuple of ints representing the desired
              reduction
347         of A's rows' with v. This reduction Ttakes the element-wise
              product
348         of each row in A with the column v, then finds the resulting
              array's
349         max value. The ith value in the output tuple is the max value of
               A's
350         ith row's element-wise product with v. If the ith row of A has
              width
351         0 (i.e. no 1 values), the default value is used.
352
353         Parameter A: a torch tuple of int cast bools with n rows and m
              columns
354         Precondition: a 2d torch tuple, values must be int 0 or int 1
355
356         Parameter v: a 1d tuple of ints, len(v) == m
357         Precondition: a 1d tuple of ints where len(v) == A.shape[1]
358
359         Parameter default: the value used to replace any of A's zero-
              width rows
360         Precondition: an int
361     """
362     return lambda A, v, default=0: tuple(
```

```
363            max(row[i] * v[i] if row[i] else default for i in range(len(v)))
364            for row in A
365        )
366
367
368    def aggr_min(_: Referent) -> Callable[[tuple[tuple[bool, ...], ...],
           tuple[int, ...], Num], tuple[int, ...]]:
369        """
370        Return a Callable function that performs a min reduction similar to
               aggr_min.
371
372        This function is initialized to be passed as an argument for the `
               reduction` parameter
373        in functions like aggr and kqv.
374
375        Parameter _: any sequence of symbols
376        Precondition: Referent
377
378            Return a 1d tuple of ints, where the ith value is the min value
                   of Av's ith row.
379
380            Finds the min by calling the aggr_max function, after
                   mutliplying
381            each element of v by -1, the default by -1. So aggr_max will
                   return
382            the greatest value (the least negative) for each row of Av,
                   which
383            is then multiplied by -1 to get the min value.
384            Takes the element-wise product of each row in A with the column
                   v,
385            then finds the resulting vector's min value. The ith value in
                   the
386            output tuple is the min value of A's ith row's element-wise
```

```
            product
387         with v. If the ith row of A has width 0 (i.e. no 1 values),
388         the default value is used.
389 
390         Parameter A: a tuple of int cast bools with n rows and m columns
391         Precondition: a 2d tuple, values must be int 0 or int 1
392 
393         Parameter v: a 1d tuple of ints, len(v) == m
394         Precondition: a 1d tuple of ints where len(v) == A.shape[1]
395 
396         Parameter default: the value used to replace any of A's zero-
            width rows
397         Precondition: an int
398     """
399     return lambda A, v, default=0: tuple(
400         -max(row[i] * (-v[i]) if row[i] else -default for i in range(len
            (v)))
401         for row in A
402     )
403 
404 def kqv(
405     k: tuple[int, ...],
406     q: tuple[int, ...],
407     v: tuple[int, ...],
408     pred: Callable[[Num, Num], bool],
409     default: Num = 0,
410     reduction: Callable[[tuple[tuple[bool, ...], ...], tuple[int, ...],
        Num], tuple[int, ...]] = aggr_mean,
411 ) -> tuple[int, ...]:
412     """
413     Return a 1d tuple of ints representing the desired reduction of a
414     (q,k) matrix's rows' with v.
415 
```

```
416     Creates a 2d tuple table A of int cast bools by comparing q and k
            using
417     some input pred, then applies the input reduction function
418     aggr_mean or aggr_mean to A and v to find the mean, min, or max
419     value of each row of A multiplied element-wise by v.
420
421     Parameter k: a 1d tuple of ints
422     Precondition: a 1d tuple of ints, len(k) == len(q)
423
424     Parameter q: a 1d tuple of ints
425     Precondition: a 1d tuple of ints, len(q) == len(k)
426
427     Parameter v: a 1d tuple, len(v) == len(k)
428     Precondition: a 1d tuple of ints, len(v) == len(k)
429
430     Parameter default: value replacing any of the (q,k) matrix's zero-
            width rows
431     Precondition: an int
432
433     Parameter reduction: a function that reduces a vector of values to a
             single, representative value
434     Precondition: a Callable in [aggr_mean(), aggr_max(), aggr_min()]
435     """
436     return reduction(select(k, q, pred), v, default=default)
```

Listing A.1: The RASP-L grammars.

BIBLIOGRAPHY

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *arXiv preprint arXiv:2005.14165*, 2020.

Mirelle Candida Bueno, Carlos Gemmell, Jeff Dalton, Roberto Lotufo, and Rodrigo Nogueira. Induced Natural Language Rationales and Interleaved Markup Tokens Enable Extrapolation in Large Language Models. In Deborah Ferreira, Marco Valentino, Andre Freitas, Sean Welleck, and Moritz Schubotz, editors, *Proceedings of the 1st Workshop on Mathematical Natural Language Processing (MathNLP)*, 2022.

Hila Chefer, Shir Gur, and Lior Wolf. Transformer Interpretability Beyond Attention Visualization. In *Proceedings of the 2021 Computer Vision and Pattern Recognition (CVPR)*, 2021.

Guy Dar, Mor Geva, Ankit Gupta, and Jonathan Berant. Analyzing Transformers in Embedding Space. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *arXiv preprint arXiv.1810.04805*, 2018.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. A Survey on In-context Learning. In *arXiv preprint arXiv:2301.00234*, 2024.

Yann Dubois, Gautier Dagan, Dieuwke Hupkes, and Elia Bruni. Location Attention for Extrapolation to Longer Sequences. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and Fate: Limits of Transformers on Compositionality. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

M Reza Ebrahimi, Sunny Panchal, and Roland Memisevic. Your Context Is Not an Array: Unveiling Random Access Limitations in Transformers. In *Proceedings of the 1st Conference of Language Modeling (COLM)*, 2024.

Jeffrey L. Elman. Finding Structure in Time. *Journal of Cognitive Science*, 14(2):179–211, 1990.

Jacob Feldman. Minimization of Boolean complexity in human concept learning. *Nature*, 407(6804): 630–633, 2000. URL `https://doi.org/10.1038/35036586`.

Jerry A. Fodor and Zenon W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1):3–71, 1988.

Borjan Geshkovski, Cyril Letrouit, Yury Polyanskiy, and Philippe Rigollet. A mathematical perspective on Transformers. In *arXiv preprint arXiv:2312.10794*, 2023.

Mor Geva, Avi Caciularu, Kevin Wang, and Yoav Goldberg. Transformer Feed-Forward Layers Build Predictions by Promoting Concepts in the Vocabulary Space. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

John Hewitt, Michael Hahn, Surya Ganguli, Percy Liang, and Christopher D. Manning. RNNs can generate bounded hierarchical languages with optimal memory. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1978–2010, 2020.

Sepp Hochreiter and Jürge Schmidhuber. Long Short-Term Memory. *Journal of Neural Computation*, 9(8):1735–1780, 1997.

Xinting Huang, Andy Yang, Satwik Bhattamishra, Yash Sarrof, Andreas Krebs, Hattie Zhou, Preetum Nakkiran, and Michael Hahn. A Formal Framework for Understanding Length Generalization in Transformers. In *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, 2025.

Nathaniel Imel, Chris Haberland, and Shane Steinert-Threlkeld. The Unnatural Language ToolKit (ULTK): a software library for research in computational semantic typology. 2025.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. In *arXiv preprint arXiv:2001.08361*, 2020.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The Impact of Positional Encoding on Length Generalization in Transformers. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Brenden M. Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

Nayoung Lee, Kartik Sreenivasan, Jason D Lee, Kangwook Lee, and Dimitris Papailiopoulos. Teaching Arithmetic to Small Transformers. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

R. Thomas McCoy, Robert Frank, and Tal Linzen. Does Syntax Need to Grow on Trees? Sources of Hierarchical Inductive Bias in Sequence-to-Sequence Networks. *Transactions of the Association for Computational Linguistics*, 8:125–140, 2020.

Benjamin Newman, John Hewitt, Percy Liang, and Christopher D. Manning. The EOS Decision and Length Extrapolation. In *Proceedings of the 3rd BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, 2020.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context Learning and Induction Heads, 2022. URL `http://arxiv.org/abs/2209.11895`.

Ofir Press, Noah A. Smith, and Mike Lewis. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. In *Proceedings of the 10th International Conference on Learning Representations (ICLR)*, 2022.

Jing Qian, Hong Wang, Zekun Li, Shiyang Li, and Xifeng Yan. Limitations of Language Models in Arithmetic and Symbolic Induction. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. 2018. URL `https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf`.

Ruoqi Shen, Sébastien Bubeck, Ronen Eldan, Yin Tat Lee, Yuanzhi Li, and Yi Zhang. Positional Description Matters for Transformers Arithmetic. In *arXiv preprint arXiv:2311.14737*, 2023.

Jiajun Song, Zhuoyan Xu, and Yiqiao Zhong. Out-of-distribution generalization via composition: a lens through induction heads in Transformers. In *arXiv preprint arXiv:2408.09503*, 2024.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What Formal Languages Can Transformers Express? A Survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 2024.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2014.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Jack Krawczyk, Cosmo Du, Ed Chi, Heng-Tze Cheng, Eric Ni, Purvi Shah, Patrick Kane, Betty Chan, Manaal Faruqui, Aliaksei Severyn, Hanzhao Lin, YaGuang Li, Yong Cheng, Abe Ittycheriah, Mahdis Mahdieh, Mia Chen, Pei Sun, Dustin Tran, Sumit Bagri, Balaji Lakshminarayanan, Jeremiah Liu, Andras Orban, Fabian Güra, Hao Zhou, Xinying Song, Aurelien Boffy, Harish Ganapathy, Steven Zheng, HyunJeong Choe, Ágoston Weisz, Tao Zhu, Yifeng Lu, Siddharth Gopal, Jarrod Kahn, Maciej Kula, Jeff Pitman, Rushin Shah, Emanuel Taropa, Majd Al Merey, Martin Baeuml, Zhifeng Chen, Laurent El Shafey, Yujing Zhang, Olcan Sercinoglu, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, Alexandre Frechette, Charlotte Smith, Laura Culp, Lev Proleev, Yi Luan, Xi Chen, James Lottes, Nathan Schucher, Federico Lebron, Alban Rrustemi, Natalie Clay, Phil Crone, Tomas Kocisky, Jeffrey Zhao, Bartek Perz, Dian Yu, Heidi Howard, Adam Bloniarz, Jack W. Rae, Han Lu, Laurent Sifre, Marcello Maggioni, Fred Alcober, Dan Garrette, Megan Barnes, Shantanu Thakoor, Jacob Austin, Gabriel Barth-Maron, William Wong, Rishabh Joshi, Rahma Chaabouni, Deeni Fatiha, Arun Ahuja, Gaurav Singh Tomar, Evan Senter, Martin Chadwick, Ilya Kornakov, Nithya Attaluri, Iñaki Iturrate, Ruibo Liu, Yunxuan Li, Sarah Cogan, Jeremy Chen, Chao Jia, Chenjie Gu, Qiao Zhang, Jordan Grimstad, Ale Jakse Hartman, Xavier Garcia, Thanumalayan Sankaranarayana Pillai, Jacob Devlin, Michael Laskin, Diego de Las Casas, Dasha Valter, Connie Tao, Lorenzo Blanco, Adrià Puigdomènech Badia, David Reitter, Mianna Chen, Jenny Brennan, Clara Rivera, Sergey Brin, Shariq Iqbal, Gabriela Surita, Jane Labanowski, Abhi Rao, Stephanie Winkler, Emilio Parisotto, Yiming Gu, Kate Olszewska, Ravi Addanki, Antoine Miech, Annie Louis, Denis Teplyashin, Geoff Brown, Elliot Catt, Jan Balaguer, Jackie Xiang, Pidong Wang, Zoe Ashwood, Anton Briukhov, Albert Webson, Sanjay Ganapathy, Smit Sanghavi, Ajay Kannan, Ming-Wei Chang, Axel Stjerngren, Josip Djolonga, Yuting Sun,

Ankur Bapna, Matthew Aitchison, Pedram Pejman, Henryk Michalewski, Tianhe Yu, Cindy Wang, Juliette Love, Junwhan Ahn, Dawn Bloxwich, Kehang Han, Peter Humphreys, Thibault Sellam, James Bradbury, Varun Godbole, Sina Samangooei, Bogdan Damoc, Alex Kaskasoli, Sébastien M. R. Arnold, Vijay Vasudevan, Shubham Agrawal, Jason Riesa, Dmitry Lepikhin, Richard Tanburn, Srivatsan Srinivasan, Hyeontaek Lim, Sarah Hodkinson, Pranav Shyam, Johan Ferret, Steven Hand, Ankush Garg, Tom Le Paine, Jian Li, Yujia Li, Minh Giang, Alexander Neitz, Zaheer Abbas, Sarah York, Machel Reid, Elizabeth Cole, Aakanksha Chowdhery, Dipanjan Das, Dominika Rogozińska, Vitaliy Nikolaev, Pablo Sprechmann, Zachary Nado, Lukas Zilka, Flavien Prost, Luheng He, Marianne Monteiro, Gaurav Mishra, Chris Welty, Josh Newlan, Dawei Jia, Miltiadis Allamanis, Clara Huiyi Hu, Raoul de Liedekerke, Justin Gilmer, Carl Saroufim, Shruti Rijhwani, Shaobo Hou, Disha Shrivastava, Anirudh Baddepudi, Alex Goldin, Adnan Ozturel, Albin Cassirer, Yunhan Xu, Daniel Sohn, Devendra Sachan, Reinald Kim Amplayo, Craig Swanson, Dessie Petrova, Shashi Narayan, Arthur Guez, Siddhartha Brahma, Jessica Landon, Miteyan Patel, Ruizhe Zhao, Kevin Villela, Luyu Wang, Wenhao Jia, Matthew Rahtz, Mai Giménez, Legg Yeung, James Keeling, Petko Georgiev, Diana Mincu, Boxi Wu, Salem Haykal, Rachel Saputro, Kiran Vodrahalli, James Qin, Zeynep Cankara, Abhanshu Sharma, Nick Fernando, Will Hawkins, Behnam Neyshabur, Solomon Kim, Adrian Hutter, Priyanka Agrawal, Alex Castro-Ros, George van den Driessche, Tao Wang, Fan Yang, Shuo-yiin Chang, Paul Komarek, Ross McIlroy, Mario Lučić, Guodong Zhang, Wael Farhan, Michael Sharman, Paul Natsev, Paul Michel, Yamini Bansal, Siyuan Qiao, Kris Cao, Siamak Shakeri, Christina Butterfield, Justin Chung, Paul Kishan Rubenstein, Shivani Agrawal, Arthur Mensch, Kedar Soparkar, Karel Lenc, Timothy Chung, Aedan Pope, Loren Maggiore, Jackie Kay, Priya Jhakra, Shibo Wang, Joshua Maynez, Mary Phuong, Taylor Tobin, Andrea Tacchetti, Maja Trebacz, Kevin Robinson, Yash Katariya, Sebastian Riedel, Paige Bailey, Kefan Xiao, Nimesh Ghelani, Lora Aroyo, Ambrose Slone, Neil Houlsby, Xuehan Xiong, Zhen Yang, Elena Gribovskaya, Jonas Adler, Mateo Wirth, Lisa Lee, Music Li, Thais Kagohara, Jay Pavagadhi, Sophie Bridgers, Anna Bortsova, Sanjay Ghemawat, Zafarali Ahmed, Tianqi Liu, Richard Powell, Vijay Bolina, Mariko Iinuma, Polina Zablotskaia, James Besley, Da-Woon Chung, Timothy Dozat, Ramona Comanescu, Xiance Si, Jeremy Greer, Guolong Su, Martin Polacek, Raphaël Lopez Kaufman, Simon Tokumine, Hexiang Hu, Elena Buchatskaya, Yingjie Miao, Mohamed Elhawaty, Aditya Siddhant, Nenad Tomasev, Jinwei Xing,

Christina Greer, Helen Miller, Shereen Ashraf, Aurko Roy, Zizhao Zhang, Ada Ma, Angelos Filos, Milos Besta, Rory Blevins, Ted Klimenko, Chih-Kuan Yeh, Soravit Changpinyo, Jiaqi Mu, Oscar Chang, Mantas Pajarskas, Carrie Muir, Vered Cohen, Charline Le Lan, Krishna Haridasan, Amit Marathe, Steven Hansen, Sholto Douglas, Rajkumar Samuel, Mingqiu Wang, Sophia Austin, Chang Lan, Jiepu Jiang, Justin Chiu, Jaime Alonso Lorenzo, Lars Lowe Sjösund, Sébastien Cevey, Zach Gleicher, Thi Avrahami, Anudhyan Boral, Hansa Srinivasan, Vittorio Selo, Rhys May, Konstantinos Aisopos, Léonard Hussenot, Livio Baldini Soares, Kate Baumli, Michael B. Chang, Adrià Recasens, Ben Caine, Alexander Pritzel, Filip Pavetic, Fabio Pardo, Anita Gergely, Justin Frye, Vinay Ramasesh, Dan Horgan, Kartikeya Badola, Nora Kassner, Subhrajit Roy, Ethan Dyer, Víctor Campos Campos, Alex Tomala, Yunhao Tang, Dalia El Badawy, Elspeth White, Basil Mustafa, Oran Lang, Abhishek Jindal, Sharad Vikram, Zhitao Gong, Sergi Caelles, Ross Hemsley, Gregory Thornton, Fangxiaoyu Feng, Wojciech Stokowiec, Ce Zheng, Phoebe Thacker, Çağlar Ünlü, Zhishuai Zhang, Mohammad Saleh, James Svensson, Max Bileschi, Piyush Patil, Ankesh Anand, Roman Ring, Katerina Tsihlas, Arpi Vezer, Marco Selvi, Toby Shevlane, Mikel Rodriguez, Tom Kwiatkowski, Samira Daruki, Keran Rong, Allan Dafoe, Nicholas FitzGerald, Keren Gu-Lemberg, Mina Khan, Lisa Anne Hendricks, Marie Pellat, Vladimir Feinberg, James Cobon-Kerr, Tara Sainath, Maribeth Rauh, Sayed Hadi Hashemi, Richard Ives, Yana Hasson, Eric Noland, Yuan Cao, Nathan Byrd, Le Hou, Qingze Wang, Thibault Sottiaux, Michela Paganini, Jean-Baptiste Lespiau, Alexandre Moufarek, Samer Hassan, Kaushik Shivakumar, Joost van Amersfoort, Amol Mandhane, Pratik Joshi, Anirudh Goyal, Matthew Tung, Andrew Brock, Hannah Sheahan, Vedant Misra, Cheng Li, Nemanja Rakićević, Mostafa Dehghani, Fangyu Liu, Sid Mittal, Junhyuk Oh, Seb Noury, Eren Sezener, Fantine Huot, Matthew Lamm, Nicola De Cao, Charlie Chen, Sidharth Mudgal, Romina Stella, Kevin Brooks, Gautam Vasudevan, Chenxi Liu, Mainak Chain, Nivedita Melinkeri, Aaron Cohen, Venus Wang, Kristie Seymore, Sergey Zubkov, Rahul Goel, Summer Yue, Sai Krishnakumaran, Brian Albert, Nate Hurley, Motoki Sano, Anhad Mohananey, Jonah Joughin, Egor Filonov, Tomasz Kępa, Yomna Eldawy, Jiawern Lim, Rahul Rishi, Shirin Badiezadegan, Taylor Bos, Jerry Chang, Sanil Jain, Sri Gayatri Sundara Padmanabhan, Subha Puttagunta, Kalpesh Krishna, Leslie Baker, Norbert Kalb, Vamsi Bedapudi, Adam Kurzrok, Shuntong Lei, Anthony Yu, Oren Litvin, Xiang Zhou, Zhichun Wu, Sam Sobell, Andrea Siciliano, Alan Papir, Robby Neale, Jonas Bragagnolo, Tej Toor, Tina Chen, Valentin Anklin, Feiran Wang, Richie Feng, Milad

Gholami, Kevin Ling, Lijuan Liu, Jules Walter, Hamid Moghaddam, Arun Kishore, Jakub Adamek, Tyler Mercado, Jonathan Mallinson, Siddhinita Wandekar, Stephen Cagle, Eran Ofek, Guillermo Garrido, Clemens Lombriser, Maksim Mukha, Botu Sun, Hafeezul Rahman Mohammad, Josip Matak, Yadi Qian, Vikas Peswani, Pawel Janus, Quan Yuan, Leif Schelin, Oana David, Ankur Garg, Yifan He, Oleksii Duzhyi, Anton Älgmyr, Timothée Lottaz, Qi Li, Vikas Yadav, Luyao Xu, Alex Chinien, Rakesh Shivanna, Aleksandr Chuklin, Josie Li, Carrie Spadine, Travis Wolfe, Kareem Mohamed, Subhabrata Das, Zihang Dai, Kyle He, Daniel von Dincklage, Shyam Upadhyay, Akanksha Maurya, Luyan Chi, Sebastian Krause, Khalid Salama, Pam G. Rabinovitch, Pavan Kumar Reddy M, Aarush Selvan, Mikhail Dektiarev, Golnaz Ghiasi, Erdem Guven, Himanshu Gupta, Boyi Liu, Deepak Sharma, Idan Heimlich Shtacher, Shachi Paul, Oscar Akerlund, François-Xavier Aubet, Terry Huang, Chen Zhu, Eric Zhu, Elico Teixeira, Matthew Fritze, Francesco Bertolini, Liana-Eleonora Marinescu, Martin Bölle, Dominik Paulus, Khyatti Gupta, Tejasi Latkar, Max Chang, Jason Sanders, Roopa Wilson, Xuewei Wu, Yi-Xuan Tan, Lam Nguyen Thiet, Tulsee Doshi, Sid Lall, Swaroop Mishra, Wanming Chen, Thang Luong, Seth Benjamin, Jasmine Lee, Ewa Andrejczuk, Dominik Rabiej, Vipul Ranjan, Krzysztof Styrc, Pengcheng Yin, Jon Simon, Malcolm Rose Harriott, Mudit Bansal, Alexei Robsky, Geoff Bacon, David Greene, Daniil Mirylenka, Chen Zhou, Obaid Sarvana, Abhimanyu Goyal, Samuel Andermatt, Patrick Siegler, Ben Horn, Assaf Israel, Francesco Pongetti, Chih-Wei "Louis" Chen, Marco Selvatici, Pedro Silva, Kathie Wang, Jackson Tolins, Kelvin Guu, Roey Yogev, Xiaochen Cai, Alessandro Agostini, Maulik Shah, Hung Nguyen, Noah Ó Donnaile, Sébastien Pereira, Linda Friso, Adam Stambler, Adam Kurzrok, Chenkai Kuang, Yan Romanikhin, Mark Geller, Z. J. Yan, Kane Jang, Cheng-Chun Lee, Wojciech Fica, Eric Malmi, Qijun Tan, Dan Banica, Daniel Balle, Ryan Pham, Yanping Huang, Diana Avram, Hongzhi Shi, Jasjot Singh, Chris Hidey, Niharika Ahuja, Pranab Saxena, Dan Dooley, Srividya Pranavi Potharaju, Eileen O'Neill, Anand Gokulchandran, Ryan Foley, Kai Zhao, Mike Dusenberry, Yuan Liu, Pulkit Mehta, Ragha Kotikalapudi, Chalence Safranek-Shrader, Andrew Goodman, Joshua Kessinger, Eran Globen, Prateek Kolhar, Chris Gorgolewski, Ali Ibrahim, Yang Song, Ali Eichenbaum, Thomas Brovelli, Sahitya Potluri, Preethi Lahoti, Cip Baetu, Ali Ghorbani, Charles Chen, Andy Crawford, Shalini Pal, Mukund Sridhar, Petru Gurita, Asier Mujika, Igor Petrovski, Pierre-Louis Cedoz, Chenmei Li, Shiyuan Chen, Niccolò Dal Santo, Siddharth Goyal, Jitesh Punjabi, Karthik Kappaganthu, Chester Kwak, Pallavi LV, Sarmishta

Velury, Himadri Choudhury, Jamie Hall, Premal Shah, Ricardo Figueira, Matt Thomas, Minjie Lu, Ting Zhou, Chintu Kumar, Thomas Jurdi, Sharat Chikkerur, Yenai Ma, Adams Yu, Soo Kwak, Victor Ähdel, Sujeevan Rajayogam, Travis Choma, Fei Liu, Aditya Barua, Colin Ji, Ji Ho Park, Vincent Hellendoorn, Alex Bailey, Taylan Bilal, Huanjie Zhou, Mehrdad Khatir, Charles Sutton, Wojciech Rzadkowski, Fiona Macintosh, Konstantin Shagin, Paul Medina, Chen Liang, Jinjing Zhou, Pararth Shah, Yingying Bi, Attila Dankovics, Shipra Banga, Sabine Lehmann, Marissa Bredesen, Zifan Lin, John Eric Hoffmann, Jonathan Lai, Raynald Chung, Kai Yang, Nihal Balani, Arthur Bražinskas, Andrei Sozanschi, Matthew Hayes, Héctor Fernández Alcalde, Peter Makarov, Will Chen, Antonio Stella, Liselotte Snijders, Michael Mandl, Ante Kärrman, Paweł Nowak, Xinyi Wu, Alex Dyck, Krishnan Vaidyanathan, Raghavender R, Jessica Mallet, Mitch Rudominer, Eric Johnston, Sushil Mittal, Akhil Udathu, Janara Christensen, Vishal Verma, Zach Irving, Andreas Santucci, Gamaleldin Elsayed, Elnaz Davoodi, Marin Georgiev, Ian Tenney, Nan Hua, Geoffrey Cideron, Edouard Leurent, Mahmoud Alnahlawi, Ionut Georgescu, Nan Wei, Ivy Zheng, Dylan Scandinaro, Heinrich Jiang, Jasper Snoek, Mukund Sundararajan, Xuezhi Wang, Zack Ontiveros, Itay Karo, Jeremy Cole, Vinu Rajashekhar, Lara Tumeh, Eyal Ben-David, Rishub Jain, Jonathan Uesato, Romina Datta, Oskar Bunyan, Shimu Wu, John Zhang, Piotr Stanczyk, Ye Zhang, David Steiner, Subhajit Naskar, Michael Azzam, Matthew Johnson, Adam Paszke, Chung-Cheng Chiu, Jaume Sanchez Elias, Afroz Mohiuddin, Faizan Muhammad, Jin Miao, Andrew Lee, Nino Vieillard, Jane Park, Jiageng Zhang, Jeff Stanway, Drew Garmon, Abhijit Karmarkar, Zhe Dong, Jong Lee, Aviral Kumar, Luowei Zhou, Jonathan Evens, William Isaac, Geoffrey Irving, Edward Loper, Michael Fink, Isha Arkatkar, Nanxin Chen, Izhak Shafran, Ivan Petrychenko, Zhe Chen, Johnson Jia, Anselm Levskaya, Zhenkai Zhu, Peter Grabowski, Yu Mao, Alberto Magni, Kaisheng Yao, Javier Snaider, Norman Casagrande, Evan Palmer, Paul Suganthan, Alfonso Castaño, Irene Giannoumis, Wooyeol Kim, Mikołaj Rybiński, Ashwin Sreevatsa, Jennifer Prendki, David Soergel, Adrian Goedeckemeyer, Willi Gierke, Mohsen Jafari, Meenu Gaba, Jeremy Wiesner, Diana Gage Wright, Yawen Wei, Harsha Vashisht, Yana Kulizhskaya, Jay Hoover, Maigo Le, Lu Li, Chimezie Iwuanyanwu, Lu Liu, Kevin Ramirez, Andrey Khorlin, Albert Cui, Tian LIN, Marcus Wu, Ricardo Aguilar, Keith Pallo, Abhishek Chakladar, Ginger Perng, Elena Allica Abellan, Mingyang Zhang, Ishita Dasgupta, Nate Kushman, Ivo Penchev, Alena Repina, Xihui Wu, Tom van der Weide, Priya Ponnapalli, Caroline Kaplan, Jiri Simsa, Shuangfeng Li, Olivier Dousse,

Fan Yang, Jeff Piper, Nathan Ie, Rama Pasumarthi, Nathan Lintz, Anitha Vijayakumar, Daniel Andor, Pedro Valenzuela, Minnie Lui, Cosmin Paduraru, Daiyi Peng, Katherine Lee, Shuyuan Zhang, Somer Greene, Duc Dung Nguyen, Paula Kurylowicz, Cassidy Hardin, Lucas Dixon, Lili Janzer, Kiam Choo, Ziqiang Feng, Biao Zhang, Achintya Singhal, Dayou Du, Dan McKinnon, Natasha Antropova, Tolga Bolukbasi, Orgad Keller, David Reid, Daniel Finchelstein, Maria Abi Raad, Remi Crocker, Peter Hawkins, Robert Dadashi, Colin Gaffney, Ken Franko, Anna Bulanova, Rémi Leblond, Shirley Chung, Harry Askham, Luis C. Cobo, Kelvin Xu, Felix Fischer, Jun Xu, Christina Sorokin, Chris Alberti, Chu-Cheng Lin, Colin Evans, Alek Dimitriev, Hannah Forbes, Dylan Banarse, Zora Tung, Mark Omernick, Colton Bishop, Rachel Sterneck, Rohan Jain, Jiawei Xia, Ehsan Amid, Francesco Piccinno, Xingyu Wang, Praseem Banzal, Daniel J. Mankowitz, Alex Polozov, Victoria Krakovna, Sasha Brown, MohammadHossein Bateni, Dennis Duan, Vlad Firoiu, Meghana Thotakuri, Tom Natan, Matthieu Geist, Ser tan Girgin, Hui Li, Jiayu Ye, Ofir Roval, Reiko Tojo, Michael Kwong, James Lee-Thorp, Christopher Yew, Danila Sinopalnikov, Sabela Ramos, John Mellor, Abhishek Sharma, Kathy Wu, David Miller, Nicolas Sonnerat, Denis Vnukov, Rory Greig, Jennifer Beattie, Emily Caveness, Libin Bai, Julian Eisenschlos, Alex Korchemniy, Tomy Tsai, Mimi Jasarevic, Weize Kong, Phuong Dao, Zeyu Zheng, Frederick Liu, Fan Yang, Rui Zhu, Tian Huey Teh, Jason Sanmiya, Evgeny Gladchenko, Nejc Trdin, Daniel Toyama, Evan Rosen, Sasan Tavakkol, Linting Xue, Chen Elkind, Oliver Woodman, John Carpenter, George Papamakarios, Rupert Kemp, Sushant Kafle, Tanya Grunina, Rishika Sinha, Alice Talbert, Diane Wu, Denese Owusu-Afriyie, Cosmo Du, Chloe Thornton, Jordi Pont-Tuset, Pradyumna Narayana, Jing Li, Saaber Fatehi, John Wieting, Omar Ajmeri, Benigno Uria, Yeongil Ko, Laura Knight, Amélie Héliou, Ning Niu, Shane Gu, Chenxi Pang, Yeqing Li, Nir Levine, Ariel Stolovich, Rebeca Santamaria-Fernandez, Sonam Goenka, Wenny Yustalim, Robin Strudel, Ali Elqursh, Charlie Deck, Hyo Lee, Zonglin Li, Kyle Levin, Raphael Hoffmann, Dan Holtmann-Rice, Olivier Bachem, Sho Arora, Christy Koh, Soheil Hassas Yeganeh, Siim Põder, Mukarram Tariq, Yanhua Sun, Lucian Ionita, Mojtaba Seyedhosseini, Pouya Tafti, Zhiyu Liu, Anmol Gulati, Jasmine Liu, Xinyu Ye, Bart Chrzaszcz, Lily Wang, Nikhil Sethi, Tianrun Li, Ben Brown, Shreya Singh, Wei Fan, Aaron Parisi, Joe Stanton, Vinod Koverkathu, Christopher A. Choquette-Choo, Yunjie Li, T. J. Lu, Abe Ittycheriah, Prakash Shroff, Mani Varadarajan, Sanaz Bahargam, Rob Willoughby, David Gaddy, Guillaume Desjardins, Marco Cornero, Brona Robenek, Bhavishya Mittal, Ben Albrecht,

Ashish Shenoy, Fedor Moiseev, Henrik Jacobsson, Alireza Ghaffarkhah, Morgane Rivière, Alanna Walton, Clément Crepy, Alicia Parrish, Zongwei Zhou, Clement Farabet, Carey Radebaugh, Praveen Srinivasan, Claudia van der Salm, Andreas Fidjeland, Salvatore Scellato, Eri Latorre-Chimoto, Hanna Klimczak-Plucińska, David Bridson, Dario de Cesare, Tom Hudson, Piermaria Mendolicchio, Lexi Walker, Alex Morris, Matthew Mauger, Alexey Guseynov, Alison Reid, Seth Odoom, Lucia Loher, Victor Cotruta, Madhavi Yenugula, Dominik Grewe, Anastasia Petrushkina, Tom Duerig, Antonio Sanchez, Steve Yadlowsky, Amy Shen, Amir Globerson, Lynette Webb, Sahil Dua, Dong Li, Surya Bhupatiraju, Dan Hurt, Haroon Qureshi, Ananth Agarwal, Tomer Shani, Matan Eyal, Anuj Khare, Shreyas Rammohan Belle, Lei Wang, Chetan Tekur, Mihir Sanjay Kale, Jinliang Wei, Ruoxin Sang, Brennan Saeta, Tyler Liechty, Yi Sun, Yao Zhao, Stephan Lee, Pandu Nayak, Doug Fritz, Manish Reddy Vuyyuru, John Aslanides, Nidhi Vyas, Martin Wicke, Xiao Ma, Evgenii Eltyshev, Nina Martin, Hardie Cate, James Manyika, Keyvan Amiri, Yelin Kim, Xi Xiong, Kai Kang, Florian Luisier, Nilesh Tripuraneni, David Madras, Mandy Guo, Austin Waters, Oliver Wang, Joshua Ainslie, Jason Baldridge, Han Zhang, Garima Pruthi, Jakob Bauer, Feng Yang, Riham Mansour, Jason Gelman, Yang Xu, George Polovets, Ji Liu, Honglong Cai, Warren Chen, XiangHai Sheng, Emily Xue, Sherjil Ozair, Christof Angermueller, Xiaowei Li, Anoop Sinha, Weiren Wang, Julia Wiesinger, Emmanouil Koukoumidis, Yuan Tian, Anand Iyer, Madhu Gurumurthy, Mark Goldenson, Parashar Shah, M. K. Blake, Hongkun Yu, Anthony Urbanowicz, Jennimaria Palomaki, Chrisantha Fernando, Ken Durden, Harsh Mehta, Nikola Momchev, Elahe Rahimtoroghi, Maria Georgaki, Amit Raul, Sebastian Ruder, Morgan Redshaw, Jinhyuk Lee, Denny Zhou, Komal Jalan, Dinghua Li, Blake Hechtman, Parker Schuh, Milad Nasr, Kieran Milan, Vladimir Mikulik, Juliana Franco, Tim Green, Nam Nguyen, Joe Kelley, Aroma Mahendru, Andrea Hu, Joshua Howland, Ben Vargas, Jeffrey Hui, Kshitij Bansal, Vikram Rao, Rakesh Ghiya, Emma Wang, Ke Ye, Jean Michel Sarr, Melanie Moranski Preston, Madeleine Elish, Steve Li, Aakash Kaku, Jigar Gupta, Ice Pasupat, Da-Cheng Juan, Milan Someswar, Tejvi M, Xinyun Chen, Aida Amini, Alex Fabrikant, Eric Chu, Xuanyi Dong, Amruta Muthal, Senaka Buthpitiya, Sarthak Jauhari, Nan Hua, Urvashi Khandelwal, Ayal Hitron, Jie Ren, Larissa Rinaldi, Shahar Drath, Avigail Dabush, Nan-Jiang Jiang, Harshal Godhia, Uli Sachs, Anthony Chen, Yicheng Fan, Hagai Taitelbaum, Hila Noga, Zhuyun Dai, James Wang, Chen Liang, Jenny Hamer, Chun-Sung Ferng, Chenel Elkind, Aviel Atias, Paulina Lee, Vít Listík, Mathias Carlen, Jan van de

Kerkhof, Marcin Pikus, Krunoslav Zaher, Paul Müller, Sasha Zykova, Richard Stefanec, Vitaly Gatsko, Christoph Hirnschall, Ashwin Sethi, Xingyu Federico Xu, Chetan Ahuja, Beth Tsai, Anca Stefanoiu, Bo Feng, Keshav Dhandhania, Manish Katyal, Akshay Gupta, Atharva Parulekar, Divya Pitta, Jing Zhao, Vivaan Bhatia, Yashodha Bhavnani, Omar Alhadlaq, Xiaolin Li, Peter Danenberg, Dennis Tu, Alex Pine, Vera Filippova, Abhipso Ghosh, Ben Limonchik, Bhargava Urala, Chaitanya Krishna Lanka, Derik Clive, Yi Sun, Edward Li, Hao Wu, Kevin Hongtongsak, Ianna Li, Kalind Thakkar, Kuanysh Omarov, Kushal Majmundar, Michael Alverson, Michael Kucharski, Mohak Patel, Mudit Jain, Maksim Zabelin, Paolo Pelagatti, Rohan Kohli, Saurabh Kumar, Joseph Kim, Swetha Sankar, Vineet Shah, Lakshmi Ramachandruni, Xiangkai Zeng, Ben Bariach, Laura Weidinger, Tu Vu, Alek Andreev, Antoine He, Kevin Hui, Sheleem Kashem, Amar Subramanya, Sissie Hsiao, Demis Hassabis, Koray Kavukcuoglu, Adam Sadovsky, Quoc Le, Trevor Strohman, Yonghui Wu, Slav Petrov, Jeffrey Dean, and Oriol Vinyals. Gemini: A Family of Highly Capable Multimodal Models. In *arXiv preprint arXiv:2312.11805*, 2024.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. In *arXiv preprint arXiv:2302.13971*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

Jesse Vig. A Multiscale Visualization of Attention in the Transformer Model. In Marta R. Costa-jussà and Enrique Alfonseca, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In

Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS), 2022.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking Like Transformers. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, 2021.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. HuggingFace's Transformers: State-of-the-art Natural Language Processing. In *arXiv preprint arXiv:1910.03771*, 2020.

Changnan Xiao and Bing Liu. Conditions for Length Generalization in Learning Reasoning Skills. In *arXiv preprint arXiv:2311.16173*, 2023.

Andy Yang and David Chiang. Counting Like Transformers: Compiling Temporal Counting Logic Into Softmax Transformers. In *Proceedings of the 1st Conference of Language Modeling (COLM)*, 2024.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. In *arXiv preprint arXiv:2205.01068*, 2022.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. What Algorithms Can Transformers Learn? A Study in Length Generalization. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024a.

Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. Transformers Can Achieve Length Generalization But Not Robustly. 2024b.