Exploring Length Generalizability of RASP Transformer Programs

Toward Revealing Transformer Program Potential

Authors:	***Student no.:***	***Contact information:***
Ming Da Yang	5122546	M.D.Yang@student.tudelft.nl
Ben Cornelisse	4956834	B.Cornelisse@student.tudelft.nl
Mohamed Elsayed	4725255	M.Elsayed-1@student.tudelft.nl
Kevin Tran	4904672	M.T.K.Tran@student.tudelft.nl

In this blog, we explore Transformer Programs, specifically focusing on their capacity to manage and interpret larger sequences of data. In recent years, Transformer models have emerged as powerful tools for language processing tasks. To better understand the function of these models, the Restricted Access Sequence Processing Language (RASP) (Weiss et al., 2021) assembly-like programming language for building transformers was developed. It focusses on mechanistic interpretability, allowing researchers to understand and structure Transformer architectures effectively.

To make the Transformers Programs, research previously has tried to reverse-engineer the Transformer Models by analyzing Weights, Biases and the Structure of existing Transformers. This approach requires a lot of manual work, as the networks are large and diverse for different tasks. In the paper by Dan Friedman, Alexander Wettig, and Danqi Chen, a procedure is proposed to design Transformer Models in such a way that they have inherent interpretability and efficiency. Consequently, these models can be transferred into Transformer Programs, e.g. Readable Code without the need of reverse engineering. For this work, Dan Friedman et al. (2023) built their code based upon RASP and we will in turn reproduce and build upon their code.

As part of our reproducibility project, we conduct experiments including training and testing on the “sort” and “most-frequent” RASP tasks as described in the paper by Friedman et al. (2023). We conduct this experiment again but also on larger datasets and also by assessing the length generalizability. Through these experiments, we aim to contribute to the understanding of Transformer Programs' capabilities across different data scales, verifying the results acquired by Friedman et al. and applying our own twist by applying length generalization. Being able to reproduce the results is valuable as it validates the findings of the researchers.

Introduction to Transformer Programs

Most large language models and specifically transformers act like a blackbox and cannot easily be read and interpreted by a human. Transformer Programming aims to bridge this gap between the program and humans. As stated in the paper by Friedman et al.: *“In this work, instead of attempting to explain black-box models, we aim to train Transformers that are mechanistically interpretable by design.”*

The authors of the paper “Learning Transformer Programs” build on the RASP program that was developed by Weiss et al. (2021). RASP is a big step in linking regular programming with transformers by translating the RASP language into transformer weights. This translation allows Transformers to interpret and execute human-readable instructions directly. Human-readable instructions are programming code or commands written in a way that is understandable. RASP translates these commands into a language that can then be understood by the transformers, enabling these transformers to perform the specified tasks.

Friedman et al. introduce a method called Transformer Programs, which aims to make transformers easier for humans to understand. It sets rules to constrain the Transformers and uses a technique to turn them into easy-to-read Python code. Tests show that these Transformer Programs work well in different tasks and are easier to understand than regular transformers. The aim of the paper is to showcase that Transformer Programs represent a promising step towards building inherently interpretable machine learning models.The goal of this blog is to validate the findings presented by Friedman et al., in their paper.

Motivation: Testing Length Generalizability

While the proposed Transformer models offer interpretability and reasonable performance on standard tasks, it's good to evaluate their generalizability on longer sequences of data. Real-world applications often involve processing extended texts or sequences, and the ability of a model to handle such inputs efficiently is important. Understanding how Transformer Programs perform with increased input lengths is important for using them effectively in practical scenarios.

Methodology

Friedman et. al (2023) introduced a method to obtain a deterministic mapping between Transformer components and RASP-like programming by defining constraints on Transformer weights. These constraints are:

Disentangled residual stream

Summarized, a Transformer can be seen as a sequence of modules (attention heads and MLP's) that read from the residual stream, apply a nonlinear function on it and then write to the residual stream. The difficulty here is due to the many calculations being done on the residual stream, it becomes unclear which modules are responsible for individual behaviors in a Transformer. Therefore the first constraint is to *disentangle* the residual stream.
This means that each attention head reads a fixed amount of tokens from the residual stream, and writes to a dedicated and fixed address. In practice this means that the output of each module gets concatenated to the residual stream. The following module then reeds from the last output of the residual stream, applies a nonlinear transformation on it and subsequently concatenates its output to the input. This way the individual behavior of a transformer module can be studied.
The final dimension of the output then becomes $2 + (L \times H) \times k$, where L is the number of layers, H is the number of attention heads and k is the cardinality of the categorical variable (two if it's binary, ten if it's in base 10).

Transformer Program modules

Secondly each module is constrained such that the mapping between input and output is interpretable and rule-based. The primary modules used in Transformer Programs are: Categorical attention heads. Which correspond to the 'select' and 'aggregate' function primitives in RASP.

To reproduce the experiment outlined in the paper, we first implement the Transformer Programs architecture as described on the github repository (Princeton-Nlp, n.d.) of the paper. We utilize the provided codebase to reconstruct the model architecture based on the methodology provided in the paper. We verify the compatibility of our model with the one described in the paper and verify whether the obtained accuracies for the "sort" and "most_frequent" tasks aligns with those reported in the study.

The sort task is about training the model to arrange numbers or letters in order, like sorting books by their size.

The most frequent task would involve teaching the same model to pick out the number that appears the most in a list.This task is about making the model learn to do that quickly and in a way that lets us understand how it figured it out.

- Example of sort task on a sequence:

- Example of most frequent task on a sequence

For these tasks we will perform the following experiments:

Table 1; Experiments that will be performed.

Experiment	**Motivation**
Train and test on sort and most_freq RASP tasks with the same parameters as Friedman et. al (2023)	We want to verify Friedman et. al (2023) results, based on accuracy
Train and test sort and most_freq RASP tasks on larger training and test input sequences	We want to verify if the model by Friedman et. al (2023) performs better or worse on larger input sequences compared to the original results
Train and test on sort and most_freq RASP tasks on length generalizability with the same parameters as Friedman et. al (2023), but with only larger test input sequences	As additional results we want to test how length generalizable the found models are.

To conduct the experiment to reproduce the paper, we extend the initial repository used by the paper. Our repository can be found on: https://github.com/mln9d4/TransformerPrograms_reproducibility

For the first experiment, we will try to reproduce the results that the authors found on the sort and most_freq task. This is done by forking and cloning their github (Princeton-Nlp, n.d.) and running the shell script. Using the predefined shell script rasp.sh in the scripts directory, we can immediately train and test the transformer programs found using their predefined parameters which they also used in the paper.

The goal of the second experiment is to see how training the transformer on longer sequences will affect the accuracy, given it is tested on the same sequence length it was trained on. Modifying the shell script, namely the *MAX_LENGTH* variable changes the length the sequences are generated and the transformer is trained and tested on.

For the final experiment we modified the shell script so that *MAX_LENGTH* does not also modify the *dvar* input argument, which defines the input space. Meaning that for testing we can input a larger sequence than the training sequence. We try to find out how length-generalizable the found transformer program is by training it on the sequence length used in the paper, which is 8, and test it on longer sequences. The initial embedding is a sum of the token embedding and the positional embedding, for which one-hot-encoding is used. The constraint of the disentangled residual stream - where each attention head reads

a fixed amount of tokens - means that the one-hot vector is of a fixed size, determined by *dvar*. Therefore, to test on longer sequences than present in the training set, the dimension of the one-hot vector must be larger than the training sequence length. Thus, the *dvar* parameter is changed to **100** and *MAX_LENGTH* is set to **8.** This ensures that the one-hot vector dimension is big enough to house the trained data sequence length of **8** and the test data sequence lengths of **10, 20, 30, 40, 50, 60, 70 ,80 and 90**.

For both tasks the original data is reproduced with **1413** training samples. Then the data is tested with **1413** test samples**.**

Table 2; table with results of the best-performing model found by Friedman et al (2023).

Dataset	Description	Example	k	L	H	M	*Acc.*
Reverse	Reverse a string.	`reverse`("abbc") = "cbba"	8	3	8	2	99.79
Histogram	For each token, the number of occurrences of that letter in the sequence.	`hist`("abbc") = "1221"	8	1	4	2	100.0
Double hist.	For each token, the number of unique tokens with the same histogram value.	`hist2`("abbc") = "2112"	8	3	4	2	98.40
Sort	Sort the input in lexicographical order.	`sort`("cbab") = "abbc"	8	3	8	4	99.83
Most-Freq	The unique input tokens in order of frequency, using position to break ties.	`most_freq`("abbc") = "bac"	8	3	8	4	75.69
Dyck-1	For each position i, is the input up until i a valid string in Dyck-1 (T); a valid prefix (P); or invalid (F).	`dyck1`("()())") = "PTPTF"	16	3	8	2	99.30
Dyck-2	The same as above, but in Dyck-2.	`dyck2`("({})(}") = "PPPTPF"	16	3	4	4	99.09

Results of the experiment of the authors of the paper are in Table 3. On the sort task their method is able to find a program that gets an accuracy of 99%.The Most-Freq gets an accuracy of 75.69%. These results show that, at least on short inputs, Transformer Programs can learn effective solutions to a variety of algorithmic tasks.

Results and Discussion

In this part, we will show and discuss the results that we obtained for the previously mentioned experiments.

Table 3; experiment results of reproduction of author results and of training and testing on larger sequence inputs.

Experiment	Sequence Length	Sort Accuracy	Most_freq Accuracy
Train and test on sort and most-freq RASP tasks with the same parameters as Friedman et. al (2023)	8	0.9996	0.7341
Train and test sort and most-freq RASP tasks on larger training and test input sequences	8	0.9996	0.7341
	12	0.9701	0.6787
	16	0.9609	0.7199
	32	0.9081	0.8166

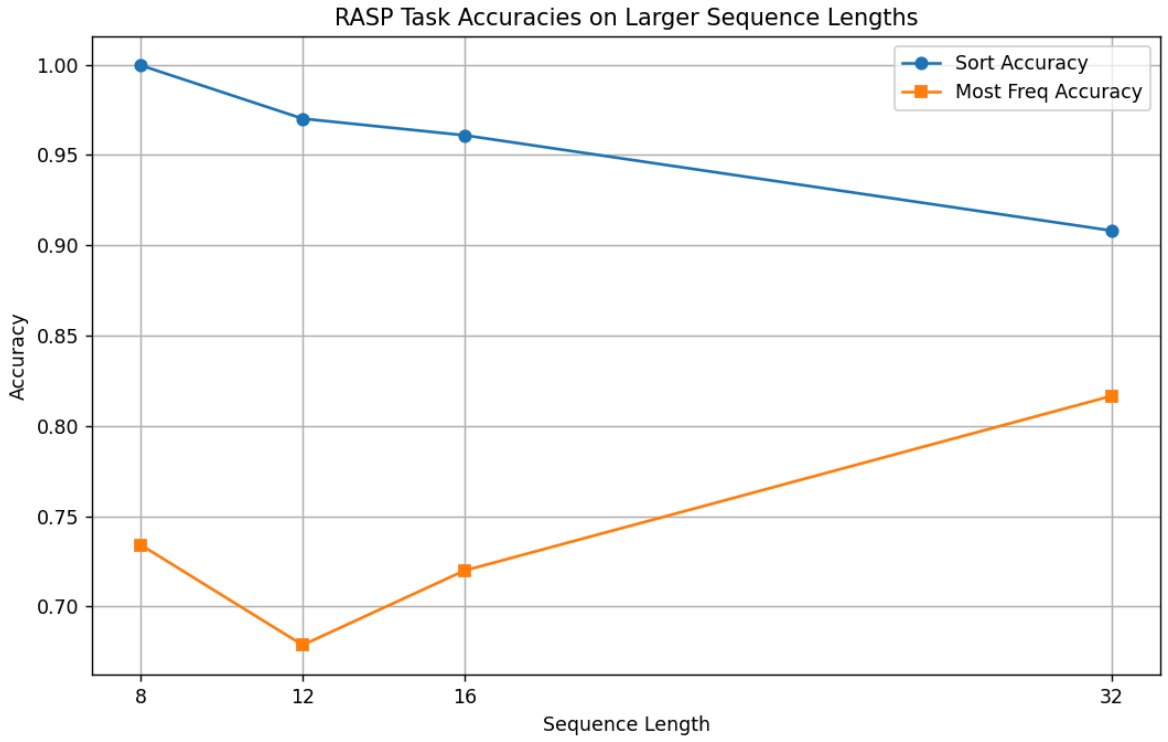

Figure 1; validation accuracies of RASP Tasks trained and tested on larger sequence lengths

When looking at table 2, we firstly trained the model using the same parameters as in the paper to check if the results are consistent. The results are within an acceptable margin of error which is caused by training randomness, thus we can conclude that the results are congruent with the results in the paper.

After this, the table shows the results of training the tasks on longer sequence lengths. In this case, we see that generally the testing accuracy drops, when the sequence is longer for the sort task. However looking at the most_freq task we can observe that unlike the sort task, it develops an upward trend. Suggesting it performed as well or better at longer sequence lengths. For a more definitive trend line, more variation in sequence lengths need to be tested.

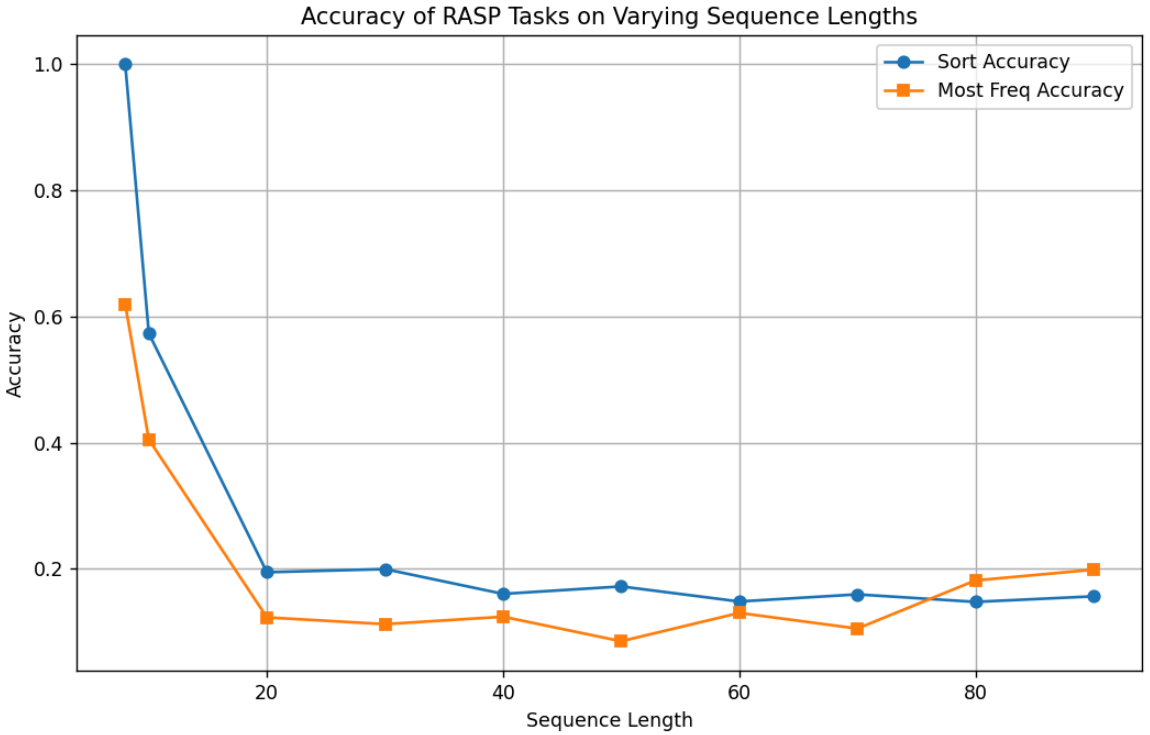

Figure 2; validation accuracy of RASP Tasks on various sequence input lengths testing length-generalizability

Table 3; table of results for length-generalizability.

Experiment	Sequence Length	Sort Accuracy	Most_freq Accuracy
Train and test on sort and most-freq RASP tasks on length generalizability with the same parameters as Friedman et. al (2023), but with only larger test input sequences	8	1.0	0.6190

	10	0.5729	0.4047
	20	0.1944	0.1228
	30	0.1994	0.1121
	40	0.1601	0.1239
	50	0.1719	0.0850
	60	0.1480	0.1299
	70	0.1593	0.1051
	80	0.1474	0.1814
	90	0.1562	0.1985

Table 2 contains the results of testing the model used in the paper on longer sequences. We see that the model does not length-generalize well and the token-accuracy drops steeply when the model is tested on longer sequences. This is both true for the sort and most frequent tasks.

It seems that generally it is easier for the Transformer to execute the sort task than the most frequent task, which is congruent with the findings in the paper. Furthermore, it seems that the models generally struggle to keep the accuracy when dealing with longer sequences.

We also observed that when training the transformer on the most_freq task with max len 8 and *dvar* 100 the accuracy it finds drops to 0.6190 compared to what we reproduced when we used a *dvar* of 8 which nets an accuracy of 0.7341. We found that *dvar* **does** influence the accuracy for the most_freq task, however for the sort task it does not.

The experimental results reveal insights into the length generalizability of Transformer Programs. We observe:

- Performance Trends: The performance of the Transformer Programs may degrade gradually as sequence length increases, indicating potential limitations in handling longer inputs.
- Interpretability: Despite variations in performance, the interpretability of the Transformer Programs remains intact, allowing researchers to analyze model behavior and identify underlying patterns or mechanisms.

Conclusion

In this study, we followed the methods used by Friedman et al. (2023) and were able to get similar results, which means their findings are reliable. For example, our sort task accuracy was 0.9996 for a sequence length of 8, just like in their study. However, for the most_freq task, our accuracy was slightly lower at 0.7341.

When we tried the tasks with longer sequences, the sort task accuracy reduced from 0.9996 for a sequence length of 8 to 0.9609 for a sequence length of 16, and to 0.9081 for a sequence length of 32. On the other hand, the most_freq task showed a decrease in accuracy from 0.7341 for a sequence length of 8 to 0.7199 for a sequence length of 12 but then improved to 0.8166 for a sequence length of 32. This means that the most_freq task might actually do better with longer sequences.

The biggest difference was seen when we tested the tasks with much longer sequences. For sorting, accuracy fell from perfect to 0.1474 when the sequence length was increased to 80. Meanwhile, the most_freq task's accuracy went up from 0.6190 to 0.1985 as the sequence length went from 8 to 90.

We also found out that changing the dvar parameter affected the most_freq task's accuracy, dropping it from 0.7341 to 0.6190, but it didn't really affect the sort task.

To wrap up, our research supports the original study and gives us new information on how Transformer models handle longer sequences. It seems that these models may struggle with longer sequences for some tasks but not for others. This shows that we need to adjust the models differently for each task, and it also suggests we should look more into why models react differently as sequences get longer.

Contribution and criteria

In this blog post we have reproduced Learning Transformer Programs and included an extra experiment that tested if the found programs are length-generalizable. We used four criteria to write this blog post which different authors were responsible for.

Ming Da, Ben, Mohamed and Kevin Tran all contributed to the **reproduction** of the code. As can be seen in the experiment result, we reproduced the results of the paper of the two RASP tasks, most_freq and sort. Special thanks to Kevin for discovering that the repository required a linux environment to run, hence we all started using WSL 2 to be able to run the training script.

Ming Da and Kevin Tran contributed to **hyperparameters check** and **length generalization check** of the models. We both checked the influence of dvar on the validation accuracy. Furthermore, by having increased the dvar we were able to also check the found model for how well it length generalized. Using dvar 100, we were able to test inputs up to length 100 while the model was trained on sequences of length 8.
Mohamed and Ben contributed to **longer sequence training and testing**. They ran training scripts on longer sequences, since the author only tested sequences of length 8. Furthermore, they also tested the models on the longer sequences and tabulated the accuracy that resulted from these models.

Appendix

Github with codebase used to run the experiments:
https://github.com/mln9d4/TransformerPrograms_reproducibility

Results can be found in output/rasp.
Our own written python files for testing length generalization can be found in the test folder. Reproducing the results were done with rasp.sh script which can be found in the scripts folder. Furthermore, in the output/rasp directory you can find the used input arguments for the training sessions.

Bibliography

Weiss, G., Goldberg, Y., & Yahav, E. (2021, 13 June). Thinking like transformers. arXiv.org. https://arxiv.org/abs/2106.06981

Friedman, D., Wettig, A., & Chen, D. (2023, 1 June). Learning transformer programs. arXiv.org. https://arxiv.org/abs/2306.01128

Princeton-Nlp. (n.d.). GitHub - princeton-nlp/TransformerPrograms: [NeurIPS 2023] Learning Transformer Programs. GitHub. https://github.com/princeton-nlp/TransformerPrograms