Learning Transformer Programs even better

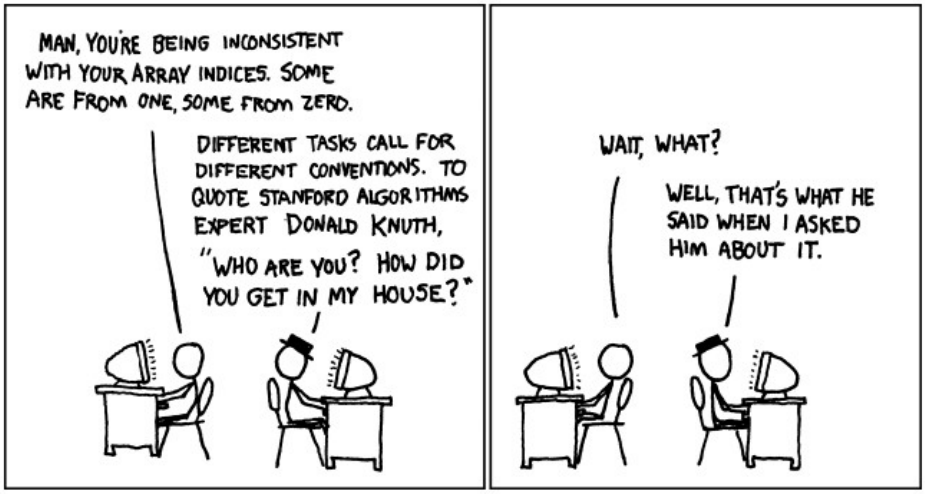
Group 8: Nina Oosterlaar (5092612 - nimoosterlaar@tudelft.nl), Sebastiaan Beekman (5885116 - jsbeekman@tudelft.nl), Joyce Sung (5011825 - wjsung@tudelft.nl), and Zoya van Meel (5114470 - hvanmeel@tudelft.nl)

Transformers have revolutionised the deep learning field. The next steps in deep learning research involve figuring out what these models are and aren't capable of. Traditionally, mechanistic interpretability was attempted via reverse engineering, such as inspecting the trained transformer weights or other post-hoc methods [4]. However, these methods are only partial explanations and can be misleading [5]. Friedman, Wettig, and Chen [1] have designed a transformer training procedure built on the RASP programming language. RASP is a simple language that can be compiled into transformer weights. Friedman *et al.* [1] modified a transformer such that the weights of a trained model can be extracted into human-readable program code. They call these programs Transformer Programs. Transformer Programs were learned for a variety of simple problems, including string reversal, sorting, and Dyck-language recognition. We attempt to reproduce the methodology they laid out and expand upon their work.

We performed a gridsearch for the parameters of all problems tested by the paper to see if the same parameters and testing performance were found. This gridsearch was also expanded upon to see if better performance is possible. Furthermore, we looked at length generalization concerning all problems. Length generalization is a method to see how well models perform on samples which are longer than they have ever seen in training. We also introduced the extra problem of positive integer addition to see if this was something a transformer could learn. Lastly, the code was refactored for a more intuitive experience and improved ease of use.

Refactoring

When multiple developers work on the same codebase, problems can occur. This is where refactoring comes into play. It turns redundant, unreadable, and/or incorrect code into clean code. Besides, refactoring increases the quality of the code itself *and the structure around it*. So maintaining code and functionalities becomes straightforward.



Despite that, refactoring is easier said than done. Creating clean code means understanding the codebase and its functionalities. This is where most of the problems arise. Developers are notoriously known for not writing (any) documentation. This lack of documentation can turn refactoring into a daunting task. Without proper documentation, understanding the codebase becomes similar to solving a complex puzzle without all the pieces.

In this case, we encountered these challenges firsthand. In the absence of documentation or explanations of the code's functionalities and how to interpret them, we were essentially operating in the dark. To tackle the task of refactoring, we came up with a systematic approach:

Firstly, we prioritized understanding the existing codebase. This involved going into the logic of the primary `run.py` file and annotating our findings with comments. By documenting the nuances of the code, we aimed to create a roadmap for future refactoring and maintenance.

Next, we identified areas of redundancy and code that needed rewriting. This covered everything from unnecessary libraries to convoluted functions and branching paths. By categorizing these elements, we could systematically address each one.

The third step involved testing the redundancies. Through debugging and testing, we ensured that the alterations did not change the functionality of the code before, as the final step, removing them.

However, not every aspect of the code underwent modification. Some passable parameters remained untouched, despite being obsolete. Additionally, fragments of duplicate code persisted. Even so, we deemed the refactored code as finished since cleaning up the code is not our primary goal. The code is now more clear, more comprehensible, and more adaptable, laying the groundwork for the foreseeable future.

Interpretability

As described before, the goal of the paper is to provide a procedure for training Transformer programs that are mechanistically interpretable by design. This is achieved by directly turning a modified Transformer into a discrete human-readable Python program using the RASP language. Different components of a Transformer are mapped deterministically to RASP primitives. For example, the central RASP operation, *select*, corresponds to attention in the Transformer. Other operations are *predicate*, which maps every combination of key and query to a boolean value in ${0, 1}$, and *aggregate*, which outputs a weighted average of values, where the weights are determined by the attention matrix resulting from the *select* operator.

For every component of the Transformer, a mapping is created using the RASP logic which, in turn, is written to the corresponding Python program. This results in a Python file containing massive amounts of code, encapsulating this (not always so logical) logic. For example, the authors looked at a Transformer with two layers and four attention heads trained on the sorting task. The code corresponding to this Transformer was 300 lines long and excluded the classifier weights. For more complex models, the number of lines increases even further. When we went through the same file, it was hard to understand the logic behind it. There were a multitude of functions, branching statements, and RASP components. It would of course be possible to use the debugger to get a sense of what is going on, but to truly understand the mechanics of the Transformer, a lot of effort must be put into understanding the code.

The authors of the paper touched upon the interpretability 'problem'. They argue that their methods are interpretable because the flow of information can be followed between different components of the model.

For more complex models it is possible to analyze the model as a collection of interpretable feature functions, instead of a single interpretable algorithm. This is all true. However, the authors also mention that the programs can still be complicated and non-intuitive. We agree with this sentiment.

When going through the results, which will be discussed later in this post, we tried to analyse the created RASP programs. it was, however, too difficult to understand the flow of information on a large scale to make conclusive statements. This might also be due to our lack of experience with the RASP program, and by deepening our knowledge in this field it may become easier to interpret. All in all, the paper is a good first step in the direction of interpreting Transformer models by mapping the program to a Python program which can be analyzed, but a lot can still be improved to make Transformer programs interpretable.

Grid search

In section 4 of the paper, the authors learned Transformer Programs on a variety of tasks: Reverse, Histogram, Double Histogram, Sort, Most Frequent, Dyck-1, and Dyck-2. These task were inspired by the original RASP paper [6]. The tasks, and their respective results, are a central point of their research. Hence, we first reproduce their results by also performing a grid search and then comparing them with our findings.

Methods

Grid search is a hyperparameter optimization technique used to systematically search through a predefined grid of hyperparameter values. The goal is to find an optimal combination of parameters that yields the best model performance. It is exceptionally effective, although computationally very expensive.

The authors performed a grid search on all the aforementioned tasks. They kept all the hyperparameters constant, except the number of layers, the number of attention heads, and the number of Multilayer Perceptron (MLP) layers. The values defined for these hyperparameters were (2, 3), (4, 8), and (2, 4) respectively. The authors trained five models with random seeds for each configuration and displayed the test set accuracy of the model with the best validation set accuracy. They also stated the best combination of hyperparameters they found in the table.

We wanted to reproduce their results but were also curious whether extending the grid search would yield better results. For the number of layers, we searched over the values (1, 2, 3, 4), for the heads (2, 4, 8, 16), and the MLPs (2, 4, 8). All the other hyperparameters were kept the same. We also trained every model five times and noted down the test accuracy of the model with the best validation set accuracy, just like the authors did.

Results

In the table below the results of the paper and our results can be seen.

	Best parameters according to Friedman *et al*. [2] and its performance according to the paper and the reproduction					Best expanded gridsearch parameters and increase w.r.t the reported accuracy			
Task	Layers	Heads	MLPs*	**Reported Accuracy (%)**	**Reproduced Accuracy (%)**	Layers	Heads	MLPs*	**Accuracy (%)**
String Reversal	3	8	2	99.79	99.89 ▲ 0.10%	4	16	2	99.99 ▲ 0.20%
Histogram	1	4	2	100.00	99.99 ▼ 0.01%	1	2	2	100.00 ▲ 0.00%
Double Histogram	3	4	2	98.40	99.67 ▲ 1.28%	2	8	8	99.99 ▲ 1.62%
Sorting	3	8	4	99.83	100.00 ▲ 0.17%	1	8	4	100.00 ▲ 0.17%
Most-Frequent	3	8	4	75.69	82.39 ▲ 8.85%	4	16	8	83.14 ▲ 9.84%
Dyck-1	3	8	2	99.30	99.65 ▲ 0.35%	3	16	8	99.94 ▲ 0.64%
Dyck-2	3	4	4	99.09	99.22 ▲ 0.13%	4	4	4	99.64 ▲ 0.56%

*Multilayer Perception

First, an important observation to make is that the authors denoted 1 layer as the optimal configuration for the Histogram method. However, as described in the methods, they performed a grid search only on layers 2 and 3. That is, at least, what they stated in their appendix. Thus, there seems to be an inconsistency in the paper.

An interesting observation is that the performance we found with the same hyperparameter settings as the author's best is actually in most cases better than the performance they found. Since this is the case for almost all tasks, it does not seem like a complete coincidence (even though it is still possible). However, we do not know what might have caused this otherwise.

The optimal hyperparameters we found differ from what the authors found for every task. In some cases, even expanding into the extended range. For example, two tasks had 4 layers as the optimal number. Moreover, for most tasks, the optimal number of heads and MLP layers is equal to 16 and 8 respectively. Again falling into the extended range. This implies that more complex models seem to yield better performance. Even so, the difference of performance is not significant and the extended grid search is computationally expensive. Thus, a trade-off between performance and efficiency should be taken into account.

Two interesting tasks are Histogram and Sort. In our grid search, they both resulted in a test accuracy of 100%. For the Histogram task, this is in line with the results of the authors since they also achieved an accuracy of 100%. Perfect performance on this task already occurs, in both cases, in the lowest possible configuration of layers, heads and MLPs (1, 2, 2). Thus, the histogram methods seem to be especially easy to learn.

On the other hand, the Sort task reaches perfect performance on the first layer already. However, the results in the paper never reach 100% accuracy for this task. The hyperparameters they state for Sort are also significantly higher than ours. This outcome is unusual to us. The Sort task has excellent performance in most runs, not requiring the large hyperparameter values the authors found.

All in all, our configurations and performances differ for all the tasks, but the differences are in most cases not that substantial.

Length generalization

The goal of the authors was to make Transformers more interpretable. Another paper which uses a version of RASP called RASP-L [2], introduces the concept that Transformers can length generalize on tasks of

which they learn the algorithm. Since this paper tries to explain Transformer code, we were interested in a length generalization experiment. Good performance would imply that the Transformer would have learned the underlying algorithm for the task and then the RASP code could be used to determine what algorithm was learned. To experiment with this it was first necessary to see if the Transformers were capable of length generalization at all. Length generalization is not an ability Transformers show consistently and some Transformers show none at all, the type of embedding used in the Transformer seems to have a large influence on its ability to length generalize [3].

Methods

For length generalization, the existing logic in the programs was altered. In the original code, a dataset was created, and this dataset was then split up into a train, test and validation set. Now, the test set is separately created from the train and validation set.

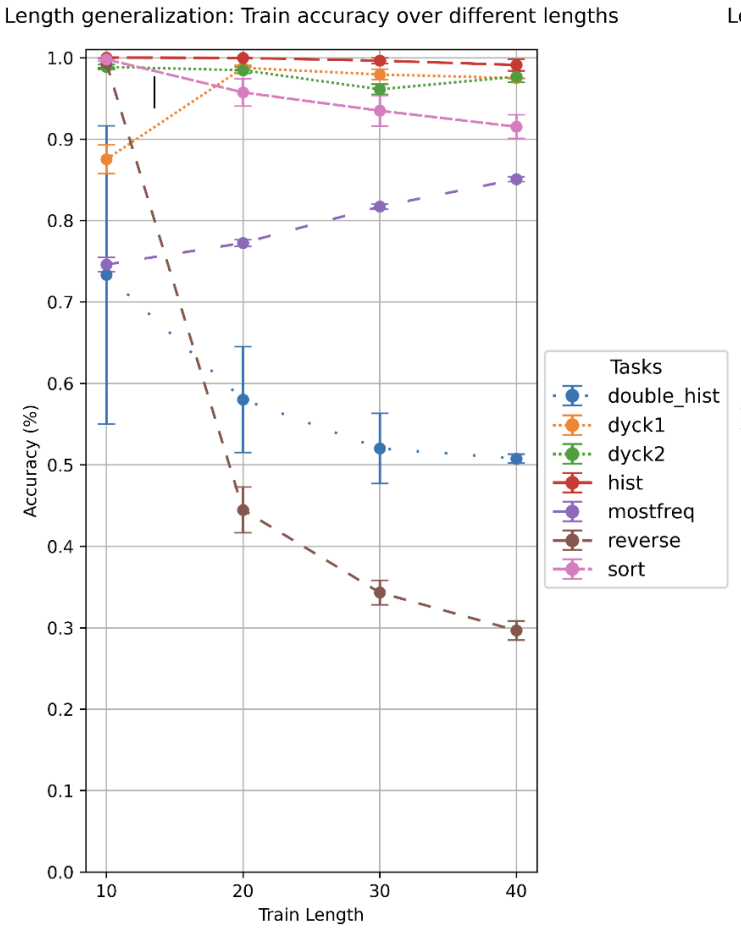
Generally, the train and validation set have a minimum length of 1 and a max length of n. The dataset the code then generates are instances of a problem with a length varying between 1 and n. The test has a minimum length of $n+1$ and a maximum length of $n+10$. Thus, the test set generates instances with varying length between $n+1$ and $n+10$. This ensures that all the instances in the test set are longer than the data the model has seen, thus testing if the model can length generalize to larger lengths. This approach corresponds with one of the general approaches of the aforementioned RASP-L paper [2]. The Transformer model is trained on $n = 10, 20, 30, 40$ for all the tasks previously mentioned tasks.

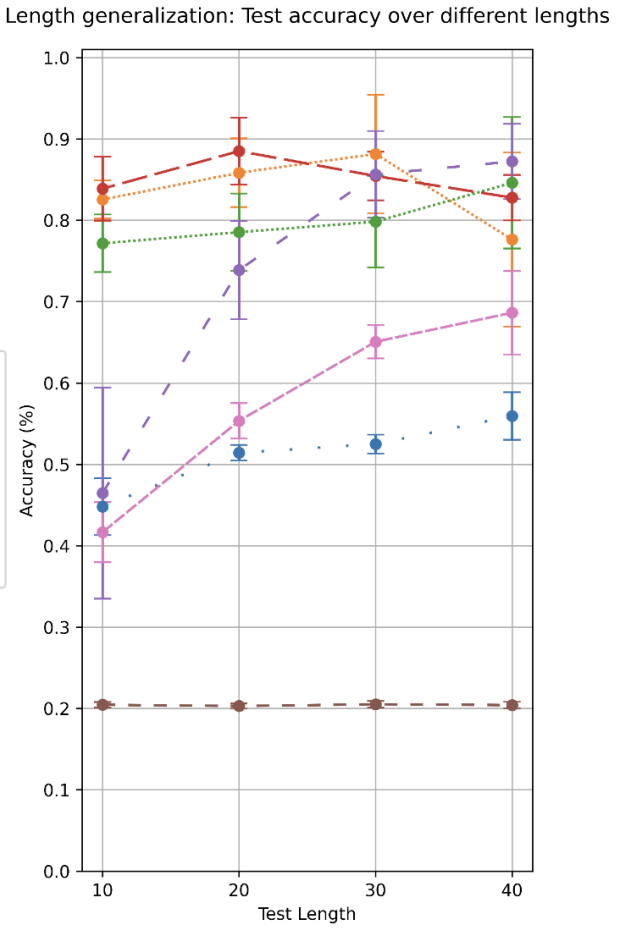
One important note is that the two Dyck tasks do not allow for varying lengths in the original code. The length of the Dyck train and validation set was set to n, and the length of the test was set to $2n$. The Transformer model was trained for $n = 10, 20, 30, 40$ as well, but the generated data does not vary in length. The hyperparameters of the Transformer models were chosen to be the same as the hyperparameters the authors reported to be optimal in Table 1 of their paper.

The model was trained five times for each task and each value of n. The average and standard deviation of the test accuracy and train accuracy of those five runs were computed. The accuracy of the tasks with larger lengths can then be compared to the accuracy of the training data.

Results

The results of our length generalization experiments are shown in the figure below.





We observe a trend that more training length diversity, meaning training with a length ranging from 1 to a larger value of n, results in better length generalization. This is in line with what the authors of the RASP-L paper found [2]. There they found that larger training length diversity ensures that the Transformer learns the underlying algorithm to solve the problem of interest, instead of for example learning heuristics to solve the problem. This might explain why in our results experiments with greater input diversity lead to better length generalization. To confirm this, further research can be done by analyzing the created RASP code and comparing this to the expected algorithm. We leave this to further research, as the code produced now is still not very interpretable

The aforementioned trend can be best seen in the following tasks: sort, most frequent, Dyck-1, Dyck-2, and double histogram. However, the performance increases to varying degrees. This might imply that some methods, like sort and double histogram, are harder to represent in Transformer architecture and thus harder to learn. It might be an interesting experiment to increase those training lengths even further to see if this leads to better performance on par with other tests. For Dyck-1 and Dyck-2 we do see an increase in performance when increasing the training length, however, they already perform relatively well for shorter training lengths as well. This might suggest that the algorithm is easier to learn, but it should be kept in mind that the length of these problems is constant, unlike the other problems, which also might have an effect.

The histogram task is an outlier to this trend. It seems like its length generalizes well for a small input diversity, thus it does not seem that adding more input diversity helps with length generalization. When we compare the performance on the test and the training dataset, it is very similar within an experiment. Thus, it seems that the length generalization ability stays the same, however, the performance seems to decrease slightly as the input length increases. This is the same for the training and test data. We think the slight

decrease in performance might be caused by the randomness in the training process, leading to a higher probability of a mistake for larger input lengths. One interesting further research topic might be to look into the cases where the Transformer fails to correctly implement the histogram task to see if there are certain trends.

The reverse task is another outlier, but not only in terms of length generalization, but it also fails to learn the task in the training dataset for increasing length. So we cannot speak about its length generalization ability, since it has not learnt the task itself for increasing training length. We are not sure why this is the case, especially since the reverse task has generally good performance in the grid search. This is a topic for further research.

One of the reasons that the Transformer seems to be able to length generalize to some extent is the way that the positional embeddings are created. The most commonly used positional embeddings, like for example absolute positional embeddings, are not well suited for length generalization [3]. This is due to this method being unable to generate a positional embedding for an unseen position. However, the model proposed in our paper does not use absolute positional embeddings, but the positional embeddings are created through one-hot encoding. This positional encoding technique is not as limited in handling unseen lengths. This can also be seen in our results. The model does never length generalize perfectly, since the performance of the test datasets does not come close to the performance of the training dataset. However, the relatively good performance some tasks seem to achieve on the test set does indicate that length generalization is possible to some extent.

All in all, the Transformer's ability to length generalize is task-dependent. Sometimes the model seems to be able to represent the algorithm of a task pretty well, enabling it to length generalize, while for other tasks this is not the case. Additionally, having a larger diversity of input lengths seems to aid its ability to length generalize. We leave interpreting the RASP code to determine whether the actual underlying algorithm of a task was learned for future work, as stated before we do not find the RASP

Addition

Like in the previous section we experiment with a task which is often not performed well by Transformers, being a simple arithmetic task, in this case, addition. Again it is interesting to know if a Transformer can complete this task and study the resulting RASP code from the learned Transformer to learn more about why Transformers can not perform these tasks. Lee *et al.* [7] and Zhou *et al.* [2] found that data formatting is important in Transformer performance on simple arithmetic, for this reason, we also try the addition task with additional formatting. To get the best possible performance a grid search is performed as well.

Methods

For the addition task, we generated a new data set. The input was a sequence which started with a beginning of sentence token, then a random sequence of digits, followed by a +, which was followed by another random sequence of digits. The two sequences have a minimum length of 1 and a maximum length of 2. The expected output was a sequence starting with a beginning of sentence token, a sequence of digits which was the sum of the two generated input numbers, followed by padding tokens to make the input and output length the same. During the data generation, unique data was generated and split into a training, test and validation set. One note on the uniqueness is that the sequence ['0', '1'] is different from the sequence ['1'], even while this denotes the same number.

For the additional formatting "index hints" as used by Zhou *et al.* [2] are used, which indicate the significance level of the digit. For index hints the alphabet was used such that `31+456` becomes `b3c1+a4b5c6`. So now in the input sequence within the random sequence of digits, its digit is preceded by its index hint, in the output sequence these hints are used as well. Further details of data generation are the same as for the addition task described in the above paragraph.

For both tasks: addition and addition with index hints a grid search was performed as described in previous sections.

Results

The results of our addition experiments are shown in the figure below.

Best parameters found with the gridsearch w.r.t. its performance on the validation set				
Task	Layers	Heads	MLPs*	**Accuracy (%)**
Addition	3	4	8	93.24
Addition Hints	3	16	8	92.00

*Multilayer Perception

When comparing the addition task with and without hints, without hints starts outperforming the addition task with hints for more layers and attention heads. This result is unexpected and might be worth exploring in further research.

We observe that the performance on the addition task is similar to the small Transformers used in Lee *et al.* [7] with a similar amount of training data. This shows that the Transformer used in this paper with its architecture and unique embeddings performs on addition similarly to other Transformers. This lays the groundwork for future work to further experiment with longer sequences and thus a relatively smaller training set as well as length generalization. These are tasks Transformers are often unable to do [7], and the algorithms which they learn when they fail or complete the task could be studied by using the Transformer model introduced in this paper and analyzing its corresponding RASP code.

Discussion

We performed a gridsearch similar to Table 1 in [1]. The accuracy we found using the reported best parameters was usually higher than in the paper. According to our gridsearch, equal or better performance was also possible in a few cases with a simpler network setup. Fewer layers or heads performed just as well. It is unclear to us why these configurations are not stated in the table. We would like to have seen more explanation on how they chose the configurations shown in the paper and a more extensive hyperparameter gridsearch. Especially since the training time of the model is not too long, this could easily have been expanded.

Our extended gridsearch found that adding more layers did not have a profound effect on performance, except on the most-frequent element task. However, increasing the number of attention heads resulted, more often than not, in better performance. Increasing the number of heads and multilayer perceptron layers and lowering the number of regular layers would usually increase performance. We attribute this to the fact that having more attention heads can enhance the model's ability to attend to diverse features,

leading to better representation learning. Adding more MLP layers can increase the model's capacity to learn intricate representations. It must be kept in mind that the increase in performance could also be a sign of overfitting. This is because the amount of training data stayed the same during all fits. Finding the right balance between attention heads, MLP layers, and regular layers is crucial and this should be explored further than the paper did and even further than we report.

Lastly, we observe a trend that more training length diversity, i.e. training with a length ranging from one to a larger value, results in better length generalization. This is in line with the findings of Zhou *et al*. [2], which introduced RASP-L. They found that larger training length diversity ensures that a transformer learns the underlying algorithm to solve the problem of interest. This could explain the results of our experiments with greater input diversity leading to better length generalization. To confirm this, further research should be done. Analyzing the created RASP code and comparing this to the expected algorithm could be the first step. We leave this to further research, as the code produced now is theoretically interpretable, but still needs a long way to become easily human-readable.

References

[1] D. Friedman, A. Wettig, and D. Chen, "Learning transformer programs," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[2] H. Zhou, A. Bradley, E. Littwin, et al., "What algorithms can transformers learn? a study in length generalization," *arXiv preprint arXiv:2310.16028*, 2023.

[3] A. Kazemnejad, I. Padhi, K. Natesan Ramamurthy, P. Das, and S. Reddy, "The impact of positional encoding on length generalization in transformers," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[4] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, "What does BERT look at? an analysis of BERT's attention," *arXiv preprint arXiv:1906.04341*, 2019.

[5] S. Jain and B. C. Wallace, "Attention is not explanation," *arXiv preprint arXiv:1902.10186*, 2019.

[6] G. Weiß, Y. Goldberg, and E. Yahav, "Thinking like transformers," in *International Conference on Machine Learning*, PMLR, 2021, pp. 11 080–11 090.

[7] N. Lee, K. Sreenivasan, J. D. Lee, K. Lee, and D. Papailiopoulos, "Teaching arithmetic to small transformers," *arXiv preprint arXiv:2307.03381*, 2023.

Contributions

Member	Contribution
Nina Oosterlaar	Length generalization and grid search implementation
Sebastiaan Beekman	Refactoring, gridsearch implementation, data collection, and debugging
Joyce Sung	Length generalization and addition implementation
Zoya van Meel	Addition implementation, data visualisation, and poster design

Everyone contributed equally to the final blog