Recursive Transformer: A Novel Neural Architecture for Generalizable Mathematical Reasoning

Stanford CS224N Custom Project

Jerry Chen
Department of Computer Science
Stanford University
jerrychen@stanford.edu

Isabelle Lee
Department of Computer Science
Stanford University
isabelle.lee@stanford.edu

Rohan Deshpande
Department of Electrical Engineering & Computer Science
Stanford University
rohand@stanford.edu

Mentor: Dor Arad Hudson (dorarad@stanford.edu)

Abstract

There has been increasing interest in recent years on investigating whether neural models can learn mathematical reasoning. Previous work in this field revealed that models seem to perform shortcuts when generating an answer, causing them to fail to generalize to a larger number of arithmetic operators. In this paper, we introduce a recursive framework to the traditional transformer architecture. We explore two approaches in designing a recursive transformer 1) a strongly supervised variant which teacher forces each recursive step and 2) a weakly supervised approach which does not constrain the model's intermediate solutions. We also adopt a curriculum-like training procedure by gradually increasing the complexity of the data. The strongly supervised approach not only successfully learned complex addition and subtraction but also demonstrated its ability to generalize by performing well when the number of operators was increased. These early results are a testament to the promise of the recursive transformer approach.

1 Introduction

Although neural architectures have made large advances in recent years in natural language processing and understanding, these models often seem to be black boxes, their inner workings difficult to interpret. Because mathematics is often referred to as a universal language – digits, operators, and other symbols have fixed meaning independent of spoken language – training neural language models to learn mathematics may provide insight into the internal mechanisms by which they learn other languages. In this paper, we explore the ability of neural models to develop their own representations of and intuition for integer arithmetic.

2 Related Work

Arithmetic can be framed as a sequence-to-sequence language processing problem, where the resulting value is the output of the input expression. Accordingly, some prior research has been done investigating the application of neural architectures typically used for language tasks, such as LSTMs [1] and transformers [2], to mathematical problems. Wangperawong [3] trained a transformer model on addition, subtraction, and multiplication of two numbers and saw high accuracies in those tasks. More recently, Saxton et al. [4] analyzed the performance of both LSTM and transformer architectures in training a general neural model for various mathematical tasks, ranging from simple arithmetic

to factoring polynomials. Their study yielded promising results, achieving over 90% accuracy on arithmetic problems, including problems with multiple numbers. Another architecture that has been proposed for the purpose mathematical and algorithmic reasoning is Kaiser and Sutskever's Neural GPU [5], which is able to perform addition and multiplication on a pair of binary numbers. It has since been improved upon by Price et al. [6] as well as Freivalds and Liepins [7] to successfully learn arithmetic with decimal inputs.

However, none of these architectures have proven successful at extrapolating to problems larger and more complex than those seen in training. Saxton et al. yielded the most promising results, training and succeeding on arithmetic expressions with multiple operators, such as adding four or five numbers together, but their transformer model is unable to generalize to an expression with more than seven or eight numbers. Meanwhile, the Neural GPU is able to extrapolate to much larger numbers than it is trained on but completely fails at operations with more than two numbers.

3 Approach

The results from prior work motivates the need to design a novel approach which addresses the generalization issue. It is common to conjecture that each layer in the neural model corresponds to a reduction of the original problem. For example if the input is `5+4+3`, the first layer may yield `9+3` while the second could produce `12`. Notice, however, that the number of layers in the model is finite and so it would always be possible to produce a problem large enough such that it will be unsolvable by the model (a hundred `1`s added together, for example). Furthermore, the task at hand may be solved much more simply by leveraging recursion, which humans implicitly use to solve such problems.

We aim to solve these issues by proposing a new method – heavily inspired by the RNN and the Transformer – called the "Recursive Transformer," which allows the output of the transformer to be fed back to its input. This approach inherits the benefits of both existing models:

- Tokens that are not positionally close together can still interact with each other due the transformer's implementation of attention.
- The model uses the same parameter values (in the form of a transformer) at each "thinking step", similar to how an RNN's use the same weights when generating each token. We hypothesise that by sharing parameter values across thinking steps, the model will be able to perform well with a smaller number of parameters while also generalizing well.

We investigated both a strongly and weakly supervised approach to building and training the proposed architecture, described in the sections below.

3.1 Forced Recursive Transformer (FRT)

In the strongly supervised approach, we treat each recursive step independently, allowing us to supervise each intermediate step. For example, rather than providing `1 + 2 + 3` as a question and `6` as the answer, we instead split the solution into a series of reductions: `1 + 2 + 3` will reduce to `3 + 3`, and `3 + 3` will reduce to `6`. By re-framing the original problem into a series of single-step reductions, we are able to completely eliminate the recurrent loop when training the model, making the model highly efficient. This approach is can be thought of as teacher forcing abstracted to both the "thinking step" level and the sequence generation level.

Similar to how natural language models must produce an end token to signify the end of a sentence, the FRT model needs a way of communicating the end of its recursion. We introduce two special tokens, "Loop Continue" and "Loop End", which allow the model to communicate whether or not it should continue to recurse. The dataset was augmented such that the final step contains a "Loop End" token appended to the answer while all intermediate steps contain a "Loop Continue" token.

Sub-problem Marking. Central to the transformer design is the concept of cross-attention, which works by performing dot products between keys and queries between the encoder and decoder [2]. This particular mechanism – without augmenting the token space – may not be optimal for the task at hand; for example, consider the problem `11+33-9`. After the decoder produces the first two tokens (`44`) it could be difficult for the transformer to learn to attend to the `-` token. From human experience, we have observed that children frequently cross-off parts of the problem that they've already visited. In other words, children augment the generated sequence in order to better determine where they should focus their attention. Taking inspiration from this example, we try a secondary (strongly supervised)

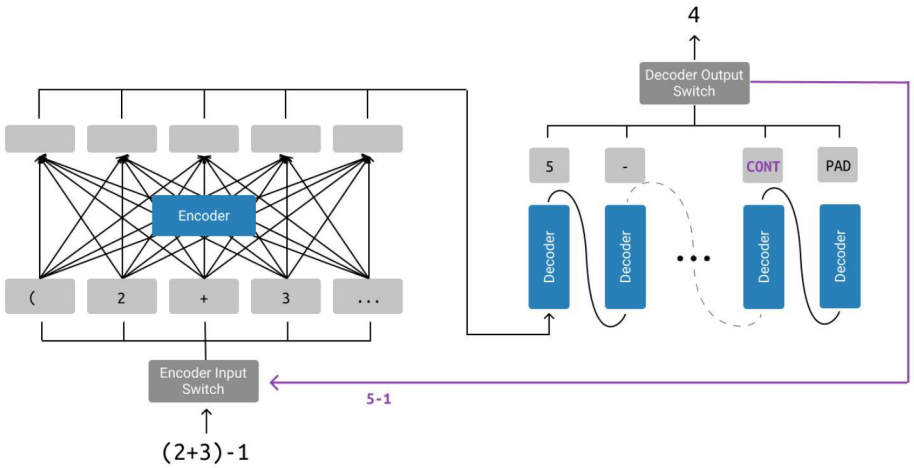


Figure 1: An abstracted diagram of the Forced Recursive Transformer model architecture. When fed the input `(2+3)-1`, the transformer produces `5-1` and the loop continue token. The output is fed back into the transformer's input because of the loop continue flag, as illustrated by the purple line. The second step produces `4` and the loop stop flag. Thus, the final answer produced by FRT is `4`.

approach to FRT which introduces a extra step. Rather than directly performing a single reduction on the input problem, the model now highlights the the sub-problem it will solve using two special tokens and then subsequently reduces the marked problem as a separate step. For example, given the problem `1 + 2 + 3`, the marking step would yield `$1 + 2# + 3`, where `$` and `#` mark the beginning and end of the sub-problem. The subsequent step reduces the problem to `3 + 3`. We hypothesize that the introduction of these two marking tokens will help the model distinguish between the part of the problem that must be solved and the part that should be copied.

3.2 Weakly Supervised Recursive Transformer (WSRT)

Our second approach does not supervise intermediate steps, but it does provide weak supervision in the form of quantifying the number of steps required to solve the problem. For an N step problem, WSRT must instead predict the intermediate representation one step at a time, and then use teacher forcing for the final step (unlike FRT which teacher forces each intermediate step). This methodology leads to multiple challenges which we attempted to address:

- The model must predict the intermediate sequence for the first $N-1$ thinking steps, which requires performing transformer inference. When performing inference, transformer models generally produce a probability distribution over the set of tokens before selecting the token with the highest probability. This discretization makes the loss non-differentiable, so we instead directly compute the embedded state of the new token directly by taking the weighted sum across the the embedding space using the probability distribution over tokens.
- Unlike FRT where each input problem leads to a single step reduction, each input problem for WSRT may require a different number of "thinking steps," which presents challenges to batched training. If we sort sort problems into batches by the number of steps, it is possible that the optimizer takes a step in a sub-optimal direction. On the other hand, collating samples of different step counts into the same batch may also have an adverse effect; it is possible that many batches using the same number of steps may cause the model to be highly dependent on this particular step count. In our implementation, we instead randomly select between these two strategies on a per-batch level with the hope of training the model without it being sensitive to step count.

4 Experiments

4.1 Data

Simple arithmetic data. In our preliminary experiments, we aimed to demonstrate a proof-of-concept version of our FRT model by training only on simple additions and subtractions between two numbers.

This "simple addition/subtraction" dataset consisted of 2 million data points with up to 9 digit numbers. The distribution of numbers was carefully tuned in order to guarantee an even mix in digit length.

Multi-step arithmetic data. In order to train and test on problems that required multi-step operations (problems consisting multiple arithmetic operators and parenthesis), we modified the dataset generation tool provided by Saxton et. al. [4]. We parsed each multi-step expression in order to generate the intermediate steps (since FRT requires supervision in intermediate steps) by adapting an open-source arithmetic parser [8]. This allowed us to generate increasingly "reduced" representations of mixed arithmetic expressions. As described in the approach section, a boolean flag which indicates whether the model should halt or not was appended to the expressions.

Notice, however, that if every intermediate step was included in the final dataset, the dataset would be skewed towards smaller problems since every N step problem can be reduced to an $N-1$, $N-2$, $\ldots, 1$ step problem. To control for the fact that the number of operators is unbalanced, we introduced the idea of "step corruption", where we randomly choose intermediate steps to include in the training data using a specially tuned probability distribution. This method was motivated by masked language modeling and span corruption introduced by the BERT [9] and T5 [10] models.

With this parsing method, we generate our multi-step training set by parsing the 666,666 `train-hard/arithmetic__add_sub_multiple` examples from DeepMind's dataset to obtain 2 million multi-step examples. In addition, we splice it with a subset of 500K examples from the simple addition/subtraction training data to ensure the model does not completely forget the previous task. Then, we repeat the process for their interpolation and extrapolation test sets to generate our multi-step interpolation and extrapolation tests.

Dataset for "sub-problem marking". As described in the approach section, another method for training FRT is to create a "marking step" where the model marks the sub-problem it will solve and a separate reduction step where the model actually reduces the problem. We modified the arithmetic parser code to include marked sub-problems and reductions and then repeated the same process as we did for the regular multi-step datasets. In addition to the previous multi-step test sets, we also generated adversarial test sets for marking and reducing the extrapolation examples; these consist of the first four marking and reductions for each example in the original extrapolation set, such that the vast majority of examples have more steps than what was seen in training.

Extra-long arithmetic. We wanted to ensure that the model would see some examples that filled out the entire input length, so we generated a small set of 100,000 examples with very large numbers (up to 30 digit) but with between one and three operators. This data was mixed into the original marked training set to create our training set for FRT_Gold. The test sets remained unchanged from the previous marking sets, as our goal was to observe the impact on the extrapolation results.

Recursive prediction test set. Because recursive prediction is meant to be an end-to-end prediction from initial question to final answer, we use the unprocessed DeepMind interpolation and extrapolation test sets for recursive prediction. This enables us to compare our FRT architectures side-by-side with the DeepMind transformer results.

4.2 Evaluation Method

For each test set, we feed the model a question from the dataset and compare whether the output is an exact character match with the dataset's answer, which is the evaluation metric used by the DeepMind group [4]. While this works well for the recursive predictions that output a single, final answer, limitations of this evaluation metric for testing intermediate steps include that it cannot differentiate between small off-by-one mistakes and completely wrong answers, and that it is quite unforgiving of carrying mistakes or a small discrepancy in a single digit place. In addition, we found that on some rare occasions, the model would perform a different reduction than the example expected or even do two or more reductions, such that the model was technically correct but did not match the example verbatim.

We considered a few alternatives, such as character-by-character similarity or a relative difference between the output and example answers had we parsed and evaluated a numerical output. However, we determined that it would be difficult to compare the results across curricula and test sets and much more challenging to standardize some overarching accuracy metric, so the exact match is sufficient for our purposes.

4.3 FRT Experimental Details and Results

We created many variants of the FRT model due to experimentation with different model configurations (see Table 1) and training datasets.

Table 1: Configurations details for various FRT models.

Variant	Features	Attention Heads	Enc/Dec Layers	Feed Forward Size	Learning Rate
FRT_Baseline	32	8	6	64	10^{-4}
FRT_Small	32	4	6	64	$5 \cdot 10^{-4}$
FRT_Small_Marked	32	4	6	64	$5 \cdot 10^{-4}$
FRT_Regular	128	8	6	256	$5 \cdot 10^{-4}$
FRT_Regular_Marked	128	8	6	256	$5 \cdot 10^{-4}$
FRT_Large	256	16	6	512	10^{-4}
FRT_Gold_Marked	128	8	6	256	$5 \cdot 10^{-4}$

FRT_Baseline was trained on simple one-operator addition or subtraction between two numbers of up to 9 digits. It achieved a 93.2% per-step accuracy on the held-out test set. All other models were pretrained on FRT_Baseline's training data. For fine-tuning, we used either the normal multi-step training data or the marked multi-step training data, described in the previous section, depending on the experiment.

We first tested the models' ability to perform correct intermediate single-step reductions (see Per-Step Accuracy column in Table 2). We then tested the models' ability to arrive at a correct fully-reduced final answer (see Total Accuracy column in Table 2) by implementing a recursive prediction function, which we discussed in Section 4.2. We used the unparsed, un-step-corrupted interpolate and extrapolate test set to evaluate only its final answer, with the hope that the model learned to perform the correct intermediate steps in between.

Table 2: Results of FRT model variants.

Test Data	Variant	Per-Step Accuracy	Total Accuracy
Simple addition/subtraction of numbers up to 9 digits	FRT_Baseline	93.20%	N/A
	FRT_Small	97.15%	N/A
Interpolation: Multi-step addition/subtraction (between 1 and 6 steps)	FRT_Small	96.65%	-
	FRT_Small_Marked	91.32%	16.65%
	FRT_Regular	97.14%	88.93%
	FRT_Regular_Marked	**99.99%**	**99.95%**
	FRT_Large	99.99%	99.75%
	FRT_Gold_Marked	**99.99%**	**99.95%**
Extrapolation: Multi-step addition/subtraction (between 9 and 11 steps)* *When parsed for per-step testing, only about half of the parsed examples remain above the training maximum of 6 steps	FRT_Small	51.13%	-
	FRT_Small_Marked	67.39%	-
	FRT_Regular	47.09%	-
	FRT_Regular_Marked	76.13%	-
	FRT_Large	70.25%	-
	FRT_Gold_Marked	**97.47%**	**64.55%**
Adversarial extrapolation: sub-problem marking only	FRT_Small_Marked	63.18%	N/A
	FRT_Regular_Marked	66.18%	N/A
	FRT_Large	51.38%	N/A
	FRT_Gold_Marked	**97.99%**	N/A
Adversarial extrapolation: reduction only	FRT_Small_Marked	22.23%	N/A
	FRT_Regular_Marked	50.00%	N/A
	FRT_Large	35.57%	N/A
	FRT_Gold_Marked	**84.11%**	N/A

4.4 WSRT Experimental Details and Results

Learning on single-step problems. WSRT was pre-trained on the simple addition/subtraction dataset. The model was randomly given 1 or 2 steps to solve these single-step problems in order to encourage it

to learn how to copy problems that are already full reduced. The model achieved an 84.86% accuracy on a test set containing 50,000 9-digit additions/subtractions. This performance is worse than FRT_Small which achieved 97.15% on the same dataset.

Learning on multi-step problems: Despite training a dozen model variants which varied the parameter size, learning rate, batch collation strategy (batches constructed using samples that require same/different number of steps) none of the models succeeded in converging.

Possible reasons for the model's inability to converge and degraded performance on single-step problems include:

- WSRT's design prevents padding from being masked since intermediate representations are never converted to indices. Although the loss function did include a re-weighting mechanism to account for the increased proportion of padding characters, we believe that the presence of padding can still negatively impact the model's performance.
- Stochasticity in the required number of steps within a batch and the number of steps allotted across different batches likely explains the model's instability during training.
- WSRT must run inference for $N - 1$ of the N "thinking steps" that the model is given. Although the dependency between thinking steps cannot be removed, we significantly improved WSRT's computational efficiency by using a causal transformer decoder mechanism which leverages caches to reduce repeated work [11]. Despite this, WSRT is an order of magnitude slower that FRT at the same parameter size, making it difficult to train the model to convergence in a single-GPU environment.

5 Analysis

5.1 Test Performance

Unmarked intermediate steps. Each of the model architectures performed well in the first step of the curriculum, learning simple addition and subtraction of two numbers. Similarly, all three models (FRT Small, Regular, and Large) that were trained on intermediate steps achieved extremely high accuracy on the DeepMind interpolation test set for multiple addition/subtraction, which consisted of similar patterns to those seen in training (no longer than six operators). However, our normal multi-step models performed far worse on extrapolation. Although the surface-level extrapolation accuracies appear better than expected, given that past work has completely failed to extrapolate, we must note that the extrapolation set we tested on was re-processed to include a distribution of intermediate steps, such that about half the test set fell below the threshold of true extrapolation (i.e. the model had seen problems of comparable size in training). Upon filtering the results of test examples solely larger than the training examples, we find that our initial models, too, completely fail to extrapolate.

Intermediate steps with sub-problem marking. Inspecting the extrapolation results, we find that the model is generally able to make the expected reduction at the beginning of the problem but fails to copy the rest of the expression, especially towards the end of longer problems. Past a certain point, the model tends to either end the output early or repeat the same few characters until it reaches the maximum target length. We guessed that training the model with the additional supervision of sub-problem marking would help mitigate some of these issues, and it did to some extent. Whereas the models trained only with the raw intermediate steps completely failed on extrapolation, FRT_Regular_Marked was able to successfully mark or reduce about half of the true extrapolation examples, for an overall extrapolation accuracy of 76.13%. When we test FRT_Regular_Marked on an adversarial test set almost entirely composed of examples larger than those seen in training, it correctly marks about two-thirds and correctly reduces about one-half of the examples, which is a significant improvement from the models trained without this extra supervision. However, noting that these results are on a step-by-step basis, we would expect the recursive prediction on the extrapolation test to nearly completely fail.

Sub-problem marking with exposure to long inputs. Despite the improved performance with sub-problem marking, we saw similar mistakes to those made before we introduced the extra marking step. The model was able to mark or reduce the sub-problem correctly in almost all cases, but it would often either drop characters or repeat the same characters towards the end of each input. One other possibility for why the model fails to extrapolate is that the source lengths in the extrapolation test set are much longer than those in the training data. Because the DeepMind dataset limits the numbers in the training set to be fairly small, no more than five or six digits in length, the model never sees training inputs longer than about 40 characters. However, the extrapolation data reaches lengths of up to 60

characters, and it might have been the case that the model fails because it has never had to work with those positions.

To test this hypothesis, we injected our training data with a small set of 100 thousand examples with very large numbers but only up to three operators, well within the operator complexity of the original training data. By doing so, we expose the model to examples that fill out the input length without compromising the integrity of the extrapolation test set. As shown in the results section, fine-tuning the original FRT_Regular_Marked model on this newly diversified training set led to immediate and significant improvements in extrapolation. The new model achieves a very high score on the adversarial marking test, though it performs somewhat worse on the reductions steps. Most notably, its end-to-end recursive predictions perform extremely well, with a nearly perfect score on the interpolation test and 64.55% on extrapolation. Given that past work on this topic has almost entirely failed to extrapolate, these results support our belief that neural language models learn arithmetic more effectively with the additional supervision of intermediate steps.

5.2 Common Mistakes Made

One of the advantages of our recursive architecture is the model's increased interpretability. Unlike prior work which acts more like a black-box by taking an input question and returning a final answer, our model is able to "show its work" by providing the intermediate steps. Aside from the most glaring issues with copying for extrapolation, some of the most common mistakes the model makes include:

- **Carrying mistakes**, leading to off-by-one errors in a couple digits
- **Dropping parentheses** that are unrelated to the current sub-problem
- **Double negatives** sometimes causing the model to subtract rather than add

While we expected the models to have trouble with carrying and double negatives, the parentheses issue was more surprising. One possible root cause could have been that we did not consider removing parentheses to be a separate step from the reduction inside the parentheses – for example, having `1 + (2 + 3)` $\rightarrow$ `1 + (5)` $\rightarrow$ `1 + 5` instead of skipping straight from `1 + (2 + 3)` $\rightarrow$ `1 + 5`. This may have caused the model to learn that parentheses, though necessary for determining the next sub-problem to mark, are not useful for the reduction themselves, leading to the issues we see.

5.3 Embeddings

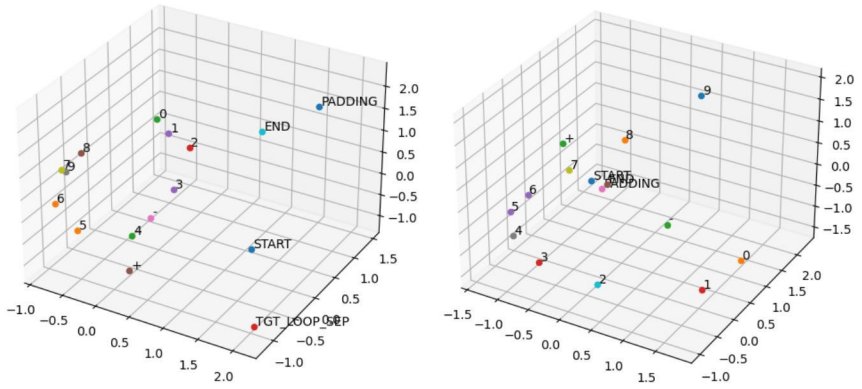
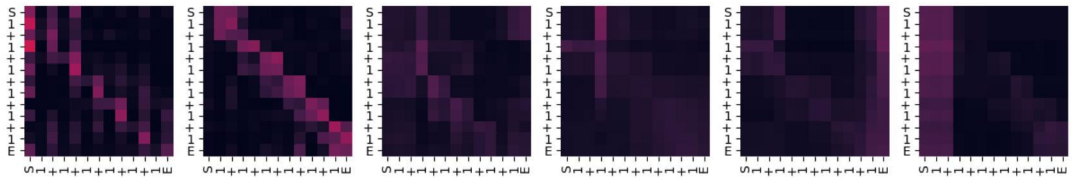


Figure 2: Embedding space of FRT_Baseline (left) and WSRT (right), reduced to 3 dimensions using PCA.

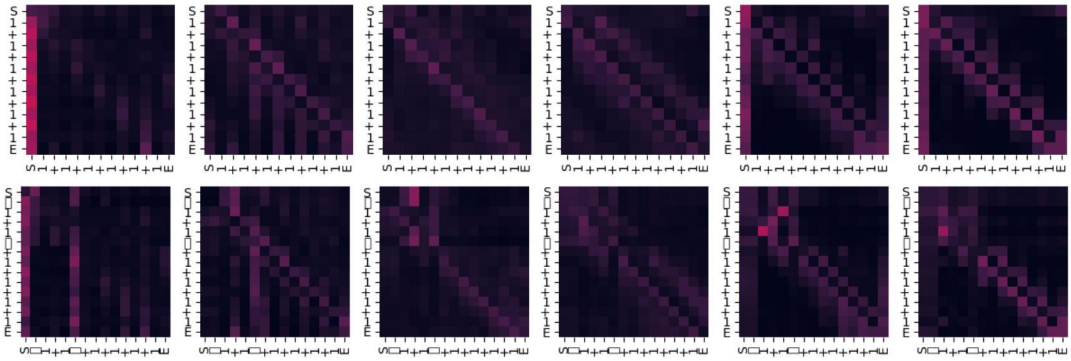
Aside from measuring quantitative performance, we hoped that the embedding vectors learned by the models would also demonstrate that the models had developed their own intuition for the language of arithmetic. Most important are the digit embeddings, as they would provide insight into the internal representation of numbers by each model. We applied PCA in order to visualize the principal components of embedding space in three dimensions. All the models clustered numerical characters, operators, and special tokens into distinct groups. Furthermore, two noteworthy representations arose from FRT_Baseline and WSRT.

The number line is a common representation of numbers by human understanding, so fundamental that it is taught in elementary school mathematics. However, considering only the single digits, we could also close the ends of the number line at 0 and 9 to form a circular representation, such that the one's digit wraps around (this is equivalent to base 10 modulus arithmetic). As shown in Figure 2, FRT_Baseline and WSRT seem to have developed similar intuitions in their digit embeddings, with the digits even appearing in the correct order along the circumference of the circle. Although we did not observe the same phenomenon in other variants of FRT/WSRT, these results show that it is indeed possible for neural models to develop a human-like understanding of numbers when trained on arithmetic.

5.4 Attention



(a) FRT_Regular encoder attention on `1+1+1+1+1+1`



(b) FRT_Gold_Marked encoder attention on marking (middle) and reduction (bottom) `1+1+1+1+1+1`

Figure 3: Encoder self-attention heatmaps for (a) FRT_Regular and (b) FRT_Gold_Marked.

Our motivation for introducing sub-problem marking was the concern that models without marking would struggle with multi-head attention, and we can see from Figure 3 that this is at least partially true. The first encoder layer of FRT_Regular has each operator attend to the relevant numbers, and the second layer shows each sub-problem attending within itself. However, attention seems to deteriorate in the rest of the layers. On the other hand, FRT_Gold_Marked maintains a clear attention pattern throughout all layers on marking steps, such that each number attends only to the relevant operators and vice versa. In the reduction step, we can see that the marking characters help partition the problem into sections that attend within themselves, and in the second-to-last layer, we even see that the two operands of the reduction attend very highly to each other.

6 Conclusion

We have developed and validated a strongly supervised recursive transformer model which performs mathematical reasoning on arithmetic problems of varying complexity. This recursive approach led to high accuracies while maintaining a relatively low number of parameters. We found that providing additional supervision by teaching the FRT model to mark sub-problems and exposing it to longer inputs granted the model greater capability to logically proceed through a problem. Our most promising results are shown by the FRT_Gold_Marked model: with an accuracy of 99.99% accuracy on interpolation and 64.55% on extrapolation, it significantly outperforms current work, which fails on extrapolation. Although the WSRT model failed to converge on multi-step problems, the insight we gained into its design challenges can serve as a solid foothold for future work.

References

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[3] Artit Wangperawong. Attending to mathematical language with transformers. *CoRR*, abs/1812.02825, 2018.

[4] D. Saxton, Edward Grefenstette, Felix Hill, and P. Kohli. Analysing mathematical reasoning abilities of neural models. *ArXiv*, 2019.

[5] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms, 2016.

[6] Eric Price, Wojciech Zaremba, and Ilya Sutskever. Extensions and limitations of the neural gpu, 2016.

[7] Karlis Freivalds and Renars Liepins. Improving the neural gpu architecture for algorithm learning, 2018.

[8] Georg Nebehay. parser. `https://github.com/gnebehay/parser`, 2020.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[10] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[11] Alexandre Matton and Adrian Lam. Making pytorch transformer twice as fast on sequence generation. 2020.