RecT: A Recursive Transformer Architecture for Generalizable Mathematical Reasoning

Rohan Deshpande[1], Jerry Chen[1] and Isabelle Lee[1]

[1]Stanford University, 450 Jane Stanford Way, Stanford, CA 94305–2004, United States of America

Abstract

There has been increasing interest in recent years on investigating whether neural models can learn mathematical reasoning. Previous approaches have attempted to train models to derive the final answer directly from the question in a single step. However, they have typically failed to generalize to problems more complex than those in the training set. In this paper, we posit that these failures can be circumvented by introducing a strongly supervised recursive framework to the traditional transformer architecture. Rather than having the model output the answer directly in a single shot, we reduce each problem into a sequence of intermediate steps that are teacher forced during training. During inference, the autoregressive model recursively generates each intermediate step until it arrives at the final solution. We validate our method by training models to solve a popular mathematical reasoning task: complex addition and subtraction with parentheses. Our model not only attains a near perfect accuracy on problems of similar difficulty to the train set but also showcases generalization capabilities: while current state-of-the-art neural architectures completely fail to extrapolate to more complex arithmetic problems, we achieve a 66.26% accuracy.

Keywords

Neural mathematical reasoning, recursive transformer, transformer, generalizable machine learning, interpretable machine learning

1. Introduction

Although neural architectures have made large advances 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 and intuition for integer arithmetic.

Past sequence-to-sequence approaches have trained models to take the full problem statement as input and produce the final answer as output. These methods perform well when evaluating the models on test samples from the same distribution as the training samples, but even the most promising models, which use transformers [1], have lacked the ability to extrapolate.

We tackle the generalization issues encountered by existing models by introducing an alternative approach that trains the model to perform a single computational step rather than

International Joint Conference on Learning & Reasoning (IJCLR) '21: International Workshop on Neural-Symbolic Learning and Reasoning (NeSy) 2020-2021, October 25–27, 2021, Virtual

rohand@stanford.edu (R. Deshpande); jerrychen@stanford.edu (J. Chen); isabelle.lee@stanford.edu (I. Lee)

to predict the final answer in one shot. By feeding the reduced output back into the model as input, the model eventually reduces the original problem into the final numerical answer. We propose an architecture for this method that we call the Recursive Transformer (RecT), which builds upon a standard transformer by allowing the output to be recursively fed back into its input, similarly to an RNN.

Training on arithmetic addition and subtraction of up to six operators, we show that RecT performs exceedingly well on test problems of the same structure (i.e. up to six operators), scoring nearly perfectly on that test set. Notably, RecT achieves over 66% accuracy on a more difficult dataset, consisting of problems with nine to eleven operators, which state-of-the-art methods have entirely failed. In addition to the improved performance on the extrapolation dataset, RecT has far fewer parameters than other state-of-the-art models and also "shows its work" by computing the answer step-by-step, so we can analyze its mistakes for insight into its understanding of 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 [2] and transformers [3], to mathematical problems. Wangperawong [4] trained a transformer model on addition, subtraction, and multiplication of two numbers and saw high accuracies in those tasks. More recently, Saxton et al. [1] 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. Likewise, 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

RecT inherits the benefits of both recursive and transformer based models:

- Tokens that are not positionally close together can still interact with each other through the transformer's attention mechanism.

- The model uses the same parameter values (in the form of a transformer) at each "computation step", similar to how RNNs use the same weights when generating each token. We hypothesise that by sharing parameter values across computation steps, the model will be able to perform well with a smaller number of parameters while also generalizing well because of its recursive formulation.

RecT treats 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 can be thought of as teacher forcing abstracted to both the "thinking step" level and the character generation level.

Similar to how natural language models must produce an end token to signify the end of a sentence, the RecT 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 signal whether or not it should continue to recurse. By adding one of the two tokens to the output, the model is taught during training whether it has reached the final answer or needs to recurse.

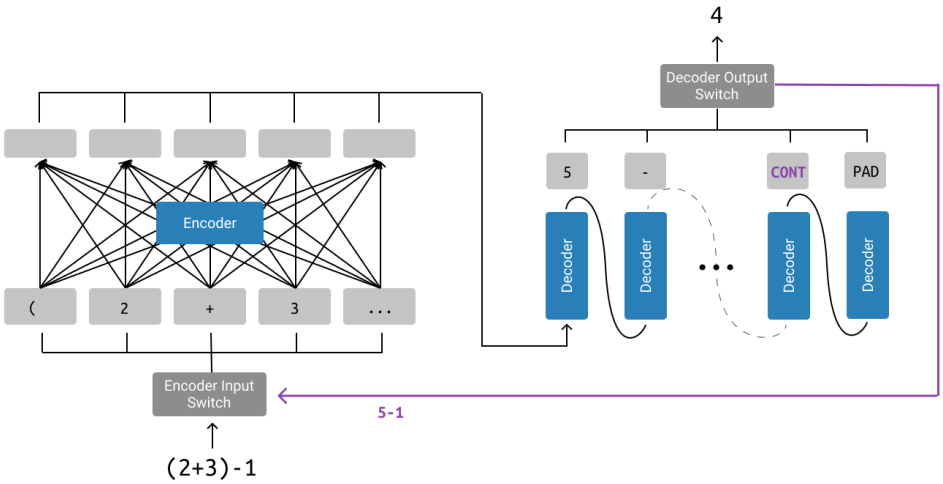


Figure 1: An abstracted diagram of the RecT model architecture, sans marking. When fed the input `(2+3)-1`, the transformer produces `5-1` and the loop continue token. The output is fed back into the transformer's input because of the loop continue flag, as illustrated by the purple line. The second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

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 [3]. 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 introduce a new strategy to RecT which adds an additional step. Rather than directly performing a single reduction on the input problem, the model is now taught to highlight the sub-problem it will solve using two special tokens and then subsequently reduce 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`. As we show in Section 6.3, the introduction of these two marking tokens helps the model distinguish between the part of the problem that must be solved and the part that should be copied.

4. Experiment Setup

Our experimentation consists of two phases. In the first phase, we examine the relative performance of various strategies, such as sub-problem marking and randomized padding (4.1). In the second phase, we apply the best performing strategies on three variants of the RecT model and compare the results to a baseline single-shot transformer model.

4.1. Data

Simple arithmetic data. Due to the complexity of the task at hand, our experimentation takes a curriculum based approach where the models are first pre-trained on simple arithmetic. The pretraining dataset consists of 2 million data points, each being an addition or subtraction between two numbers of up to 9 digits each. The distribution of numbers is 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 require multi-step operations (problems consisting multiple arithmetic operators and parentheses), we leverage the DeepMind Mathematics Dataset [1]. We parse each multi-step expression in order to generate the intermediate steps by adapting an open-source arithmetic parser [8]. This allows us to generate increasingly "reduced" representations of the questions. As described in the approach section, a boolean flag which indicates whether the model should halt or not is appended to the expressions.

Notice, however, that if every intermediate step is included in the final dataset, the dataset skews towards smaller problems since every N step problem can be reduced to problems of size $N-1$, $N-2$, and down to 1 step. To control for the fact that the number of operators is unbalanced, we introduce the idea of "step corruption", where we randomly choose which intermediate steps to include in the training data using a calibrated probability distribution. This method is 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. Then, we randomly sample 1.5 million of the multi-step examples and combine them with 500 thousand randomly selected datapoints from the simple addition/subtraction training data to ensure the model does not completely forget the previous task. We

repeat the process on DeepMind's interpolation and extrapolation test sets in order to generate our multi-step interpolation and extrapolation tests.

Sub-problem marking. As described in the approach section, another method for training RecT is to simulate 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 modify the arithmetic parser code to include marked sub-problems and reductions and then repeat the same process as described for the regular multi-step datasets.

Randomized padding. A key component of the transformer is the positional encoder, which allows the model to capture the ordering of the characters. Since we are testing the generalization of the model, the problem lengths in the extrapolation test set are longer than those in the training set. Consequently, at test time, the model may see characters at positions that it has never seen or used before. As a result, the model runs the risk of unintentionally overfitting its interpretation of different positions that it saw during training. For the task at hand, the model simply needs to know the relative positions of characters - not the absolute positions. Equivalently, the model should be robust enough to produce the same result regardless of whether the problem starts at the 0^{th} position or the n^{th} position. This insight motivates our randomized padding technique: a random amount of padding is inserted before the problem, ensuring that the model "sees" all character positions being used although the problem lengths during training are shorter than those in extrapolation.

End-to-end testing. RecT predicts the result of a single computational step rather than the final answer. In order to generate the final answer, the result from each computation step is fed back into the model – recursively making predictions – until it produces a halt token. For this end-to-end evaluation, we use the unprocessed DeepMind interpolation and extrapolation test sets for recursive prediction. This allows us to accurately compare our RecT architectures side-by-side with the baseline single-shot transformer model.

4.2. Evaluation Method

We evaluate the model by feeding it a question from the test set and comparing whether the output is an exact character match with the dataset's answer, the same method as described in Saxton et. al [1]. Although this metric is quite unforgiving towards carrying mistakes and small discrepancies in the single digit place, we choose not to be lenient with any inaccuracies produced by the model, regardless of their proximity to the correct answer. However, we qualitatively examine a selection of such near-misses in Section 6.2.

We measure two types of accuracy: the first is the per-step accuracy, which describes the model's ability to perform a single computational step such as 1+2+3 → 3+3; and the second is the end-to-end accuracy, which denotes the model's accuracy in recursively reducing the initial problems into their final numerical answers.

4.3. RecT Models

We experiment with three sizes of the RecT model, which we refer to as RecT_Small, RecT_Medium, and RecT_Large (hyperparameters detailed in Table 1). In addition to the training method above

using both sub-problem marking and randomized padding, we train variants that skip either the marking or the padding to demonstrate the benefits of the two techniques.

Table 1
Configurations details for RecT Small, Medium, Large, and Baseline models.

Variant	Features	Attention Heads	Enc/Dec Layers	Feed Forward Size	Learning Rate	Parameters
RecT_Small	32	4	6	64	$5 \cdot 10^{-4}$	130,464
RecT_Medium	128	8	6	256	$5 \cdot 10^{-4}$	1,996,320
RecT_Large	256	16	6	512	10^{-4}	7,924,768
Baseline	256	16	6	512	10^{-4}	7,924,768

4.4. Baseline Model

We believe that the proposed method will generalize better to more complex problems when compared to prior work. Though it is well-known that previous attempts completely fail to generalize, we seek to experimentally confirm these results by training and evaluating a baseline model. Doing so enables us to accurately determine the benefits of the proposed method.

The baseline model, a vanilla transformer, attempts to solve the math problems in a single shot – without recursion – as described by prior work such as Saxton et al. [1]. Since solving the problem in a single shot is a harder task than single step reductions, we use the same model configuration as RecT_Large, which has ~8M parameters.

5. Results

We find an average accuracy improvement of 22.65% on the extrapolation task when the models are trained with sub-problem marking. Results from our experimentation show an additional 20.53% improvement in extrapolation accuracy when randomized padding is also included. Hence, we decide to utilize sub-problem marked data and randomized padding in subsequent experimentation.

All three models achieve near perfect single step accuracy on the interpolation set, and consequently, the end-to-end interpolation accuracies of the three models are also extremely high, with RecT_Small performing the worst at 98.890%. Notably, RecT_Medium achieves a perfect score on the interpolation set. On the extrapolation set, the per-step accuracies are fairly high, with RecT_Medium scoring 96.426% and the other two just under 90%. However, due to the recursive architecture, errors propagate through each step, causing the end-to-end accuracy to decrease exponentially as the size of the problem increases. RecT_Medium performs the highest across the board and scores decently on the extrapolation set with 66.260% accuracy, but the other two models fall off significantly.

Table 2
Results of RecT model variants.

Test Data	Variant	Per-Step Accuracy	End-to-End Accuracy
Simple addition/subtraction of two numbers, each up to 9 digits	RecT_Small	N/A	97.210%
	RecT_Medium	N/A	99.220%
	RecT_Large	N/A	99.820%
Interpolation: Multi-step addition/subtraction (between 3 and 6 operators)	RecT_Small	99.875%	98.890%
	RecT_Medium	100%	100%
	RecT_Large	99.998%	99.980%
Extrapolation: Multi-step addition/subtraction (between 9 and 11 operators)	RecT_Small	88.153%	37.350%
	RecT_Medium	96.426%	66.260%
	RecT_Large	88.372%	23.110%

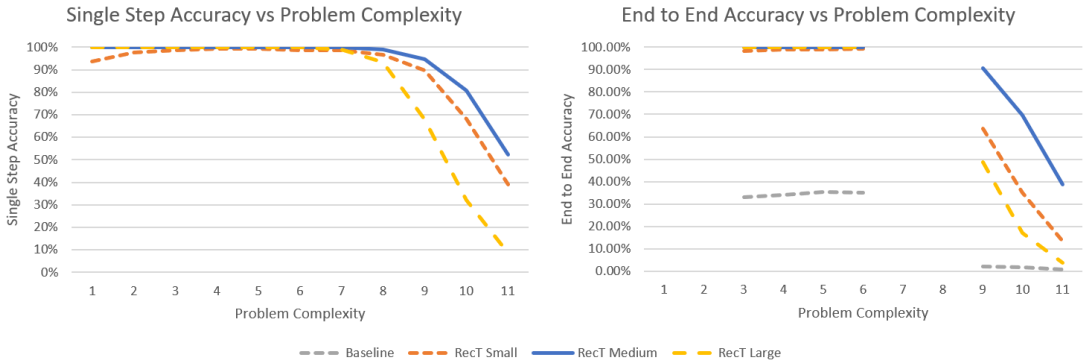


Figure 2: Accuracy comparisons between the three RecT variants and the baseline single-shot model at different problem complexities. Note that the baseline single-shot model does not produce intermediate values as it is not recursive. For this reason, it is not included in the single step accuracy analysis.

6. Analysis

6.1. Generalization

Our results show that all three RecT models significantly outperform than the baseline single-shot model. Though the baseline model performs slightly worse than expected on interpolation due to slow convergence, it performs poorly on extrapolation, which we find consistent with results in prior work. This is contrasted with RecT_Medium which achieves 90% end to end accuracy on 9-operator problems despite not having seen problems with more than 6 operators during training. We infer that this is because the recursive pattern inherently builds in generalization capability. Our analysis also shows that RecT_Large performs worse than Rect_Small on extrapolation across the board, despite the fact that it has ~ 60x more parameters. We believe this phenomenon is a result of over-fitting, which is known to be more likely for larger models.

Despite the general success from RecT models, we see a significant drop-off in accuracy for 11-operator problems. Perhaps increasing diversity in problem complexity during training –

e.g. training on data with 3 to 10 operators and extrapolating on data with 13 to 16 operators – while holding the number of parameters fixed would reinforce the model's need to generalize and in turn yield better results.

6.2. 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. Table 3 in the appendix shows an example proof of work that was produced by RecT_Medium in the extrapolate set. It demonstrates the model's ability to solve an 9 operator problem recursively by marking the appropriate sub-problem with the `$` and `#` symbols, reduce the marked problem while gracefully handling double negatives and parentheses, and correctly output loop continue/stop tokens. The baseline model, on the other hand, incorrectly produces the answer `-100` in a single shot, with no indication of what caused the error.

Through the models' proof of work, we notice that some of the most common mistakes the model makes include:

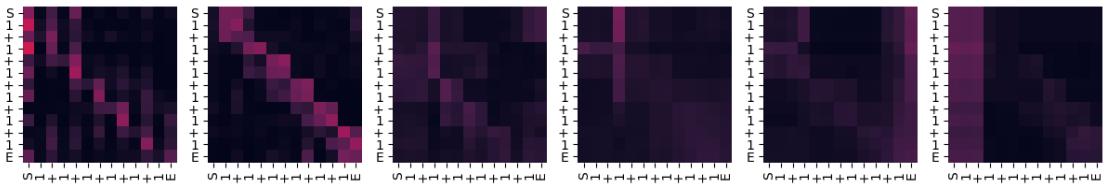
- **Carrying mistakes** leading to off-by-one errors in a couple digits
- **Dropping parentheses** that are unrelated to the current sub-problem
- **Double negatives** sometimes causing the model to subtract rather than add

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)` → `1 + (5)` → `1 + 5` instead of skipping straight from `1 + (2 + 3)` → `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.

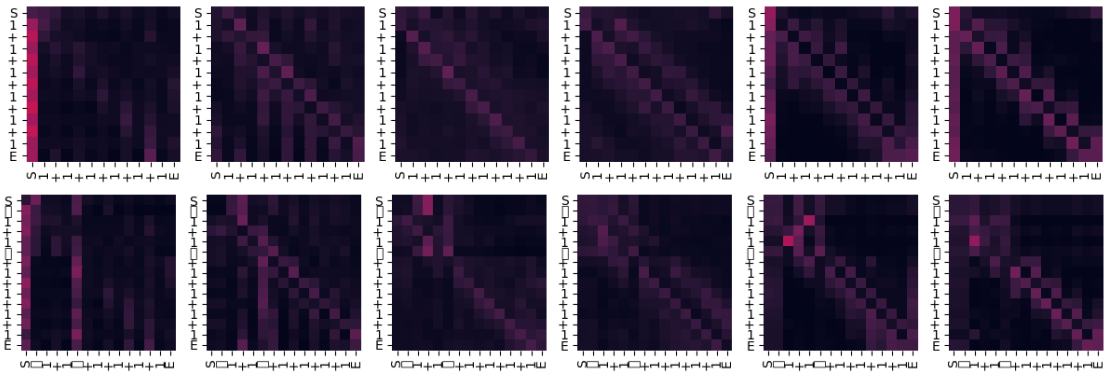
6.3. Comparing marking and non-marking RecT

When we train a model without sub-problem marking and evaluate on the extrapolation test set, 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 output length. We find that training the model with the additional supervision of sub-problem marking significantly mitigate these issues.

One explanation, supported by the attention visualizations in Figure 3, may be that the presence of the sub-problem markers helps the model refine the multi-headed attention weights. We can see that for a RecT model trained without marking, the first encoder layer has each operator attend to the relevant numbers, and the second layer shows each sub-problem attending within itself. However, attention begins to degenerate in the rest of the layers, which are less distinctly interpretable. On the other hand, RecT trained with marking maintains a clear attention pattern throughout all layers on marking steps, such that each number attends only



(a) RecT encoder attention for 1+1+1+1+1+1 without subproblem marking



(b) RecT encoder attention on marking (first row) and reduction (second row) 1+1+1+1+1+1. The rectangles in the reduction step denote the "marks" that the model places.

Figure 3: Encoder self-attention heatmaps for RecT without (a) and with (b) subproblem marking. The images in each row represent the attentions for each encoding layer of the model.

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.

6.4. Embeddings

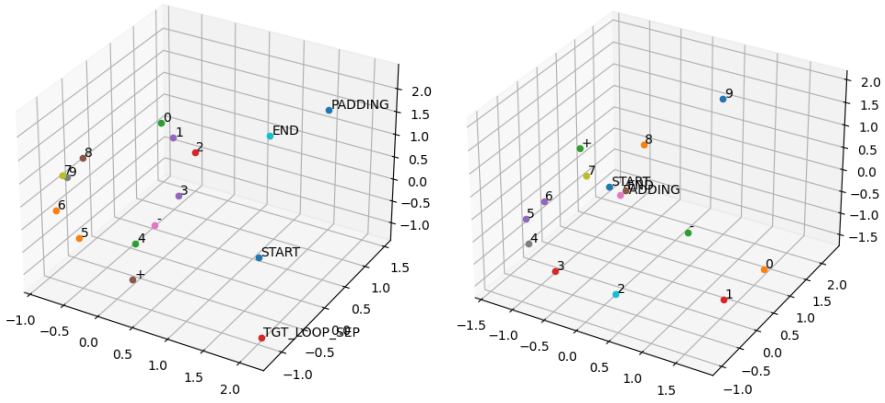


Figure 4: Embedding visualizations of two RecT models with circular representations of digits.

Aside from measuring quantitative performance, we also study the learned embedding vectors to see if the representations learned by the models may demonstrate that that they have developed an interpretable intuition for the language of arithmetic. The digit embeddings, especially, would provide insight into the internal representation of numbers by each model. We apply principal component analysis in order to visualize the embedding space in three dimensions. All the models learn to cluster numerical characters, operators, and special tokens into distinct groups. Furthermore, two of the models produce noteworthy representations.

The number line is a common representation of numbers by human understanding, but 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 modulo arithmetic). As shown in Figure 4, two of the models 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 consistently observe the same phenomenon in other variants, these results show that it is indeed possible for neural models to develop a human-like understanding of numbers when trained on arithmetic.

7. Future work

Although we have shown that the RecT approach has significant benefits, the method relies on strong supervision. The ability to generate ground truth for the intermediate steps may be a limiting factor preventing this method from being applied to other problems and domains. As a result, approaches where intermediate steps are weakly supervised or unsupervised may be desirable. Currently, the RecT framework determines whether to recurse or stop based on a special token in the output. An alternative approach involves making a continuous valued prediction on whether to recurse or not via a classifier (rather than outputting a discrete token). This enables the ability to train the model using a Markovian approach: each time the model recurses, it computes the stopping probability and loss if it were to stop at the current step. The total loss would be the sum of the individual losses weighted by the stopping probabilities. Unlike the strongly supervised method, however, this method would likely suffer from unstable training and slower computation speeds.

8. 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 leads to high accuracies while maintaining a relatively low number of parameters. We find that providing additional supervision by teaching the RecT model to mark sub-problems and applying randomized padding granted the model greater capability to logically proceed through a problem. With an accuracy of 100% on interpolation and 66.26% on extrapolation, our method significantly outperforms current work, which fails on extrapolation. In addition, RecT provides increased interpretability by outputting a proof of work, allowing a better understanding of its inner workings to serve as a basis for future improvements.

References

[1] D. Saxton, E. Grefenstette, F. Hill, P. Kohli, Analysing mathematical reasoning abilities of neural models, ArXiv (2019).

[2] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (1997) 1735–1780. URL: https://doi.org/10.1162/neco.1997.9.8.1735. doi:10.1162/neco.1997.9.8.1735.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, 2017. arXiv:1706.03762.

[4] A. Wangperawong, Attending to mathematical language with transformers, CoRR abs/1812.02825 (2018). URL: http://arxiv.org/abs/1812.02825. arXiv:1812.02825.

[5] Łukasz Kaiser, I. Sutskever, Neural gpus learn algorithms, 2016. arXiv:1511.08228.

[6] E. Price, W. Zaremba, I. Sutskever, Extensions and limitations of the neural gpu, 2016. arXiv:1611.00736.

[7] K. Freivalds, R. Liepins, Improving the neural gpu architecture for algorithm learning, 2018. arXiv:1702.08727.

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

[9] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, CoRR abs/1810.04805 (2018). URL: http://arxiv.org/abs/1810.04805. arXiv:1810.04805.

[10] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, CoRR abs/1910.10683 (2019). URL: http://arxiv.org/abs/1910.10683. arXiv:1910.10683.

9. Appendix

Table 3
Reduction steps performed by RecT_Medium on the extrapolate set.

Q: -6+8+(-3-1)+0+((148-1)+(-317-320)-3), A: 153 STOP

Step 1: $-6+8#+(-3-1)+0+((148-1)+(-317-320)-3) CONT	Step 10: 0+(147+(-317-320)-3) CONT
Step 2: 2+(-3-1)+0+((148-1)+(-317-320)-3) CONT	Step 11: 0+(147+$(-317-320)#-3) CONT
Step 3: 2+$(-3-1)#+0+((148-1)+(-317-320)-3) CONT	Step 12: 0+(147+3-3) CONT
Step 4: 2+-2+0+((148-1)+(-317-320)-3) CONT	Step 13: 0+($147+3#-3) CONT
Step 5: $2+-2#+0+((148-1)+(-317-320)-3) CONT	Step 14: 0+(150-3) CONT
Step 6: 0+0+((148-1)+(-317-320)3) CONT	Step 15: 0+$(150-3)# CONT
Step 7: $0+0#+((148-1)+(-317-320)-3) CONT	Step 16: 0+153 CONT
Step 8: 0+((148-1)+(-317-320)-3) CONT	Step 17: $0+153# CONT
Step 9: 0+($(148-1)#+(-317-320)-3) CONT	Step 18: 153 STOP