Modeling Symbolic Mathematics with Transformers

Sri Rama Bandi
sbandi@umass.edu

Huy Ngo
hdngo@umass.edu

Eric Englehart
eengelhart@umass.edu

1 Problem Statement

For a while, Transformer models (Vaswani et al., 2023) have shown to be effective in a wide range of Natural Language Processing (NLP) tasks, particularly sequence-to-sequence tasks (e.g., machine translation). As these models were proving to be good at solving approximate problems, we wanted to understand their capabilities when challenged with symbolic data, by framing symbolic mathematic problems as a sequence-to-sequence task - more specifically, we will focus on solving Ordinary Differential Equations (ODEs) of the first order with Transformers.

Largely inspired by the works of (Lample and Charton, 2019), who successfully demonstrated that neural networks can achieve high accuracy in symbolic integration and solving ODEs, our project goal is to investigate whether or not a model trained in a more limited setup could, to some degree, achieve the similar ability to understand and even manipulate mathematical expressions to find solutions to first order ODEs. The core challenge of our project will largely lie in enabling smaller models to not only learn the hierarchical structure of mathematical expressions but also "obey" the strict requirement for mathematical correctness, both of which are vastly different from their usual domain of tasks. And thereafter, we will also investigate the effectiveness of applying Tree Regularization (Nandi et al., 2025) loss to our model, as the inductive biases it introduces for tree structures (much like our expressions) might prove helpful in making our model more robust and data efficient. Our baseline will be a transformer model trained with a standard sequence-to-sequence objective. Our improved model(relative to our baseline) is also the same architecture transformer model, trained with TREEREG, an auxiliary loss in additon to the standard objective. We successfully show that, in fact, training with TREEREG model outperforms our baseline model. All of our project code is available at: https://github.com/srirambandi/compsci685

2 What You Proposed vs. What You Accomplished

Overall, our project stayed mostly consistent with our initial proposal. Here are the summary of our accomplishments since then:

- **[DONE] Dataset Generation**: successfully adapted methodologies from (Lample and Charton, 2019) to implement our own high-quality ODE-to-Solution dataset generation script
- **[DONE] Transformer Model Implementation**: model is capable of using ODEs to predict their solutions
- **[DONE] Training and Inference**: the model was successfully trained on Colab and inference was performed to assess its performance
- **[DONE] Evaluation via Beam Search + SymPy**: during inference, generate multiple candidate solutions and evaluate their correctness using SymPy
- **[DONE] Improving on the baseline with TreeReg model**: an additional model is trained with an auxiliary TreeReg loss to inject structural inductive biases.
- **[DONE] Analysis and Comparison**: a comprehensive comparative analysis of the accuracies across both the models, for various beam sizes was planned, but we were able to analyse the performances for both the models on greedy sampling and Beam Search of size

10. This is due to the large amount of time it takes to evaluate on larger beam sizes.

3 Related Work

Neural networks have always been known for their strengths in pattern recognition and statistical learning, which are domains vastly different from the precise, rule-based nature of the mathematical domain. Because of this, success in neural networks' applications within this field have largely been limited.

However, earlier works demonstrating success in using these networks for simpler arithmetic problems (Zaremba and Sutskever, 2015) and emulation of a logic unit (Trask et al., 2018) have demonstrated their great potential in this field. As Artificial Intelligence advances, we start to slowly see growing interest towards applying deep learning for diverse tasks that require logical reasoning. These include translating math word problems in natural language into solvable equations using seq-2-seq models (Wang et al., 2017), simplifying complex Mixed Boolean-Arithmetic (MBA) expressions (Feng et al., 2020), and even assisting in automated theorem proving by learning to select relevant premises from a large formal corpora (Alemi et al., 2017). Among these pivotal works are one by (Lample and Charton, 2019), who demonstrated that standard seq-2-seq Transformer (Vaswani et al., 2023) models could achieve high accuracy in complex symbolic tasks like integration and Ordinary Differential Equations (ODEs) - surprisingly outperforming well-established Computer Algebra Systems (CAS) like Mathematica. Our project is mainly inspired by their methodology: representing mathematical expressions as trees and using the prefix notation of equation-solution pairs as training data sequences.

One of our core challenges in this task lie in somehow enabling the models to understand the hierarchical structure of the provided mathematical expressions. Recent research into "structural grokking" suggests that vanilla Transformers are able to learn such hierarchical generalization, albeit under training periods way beyond the point of accuracy saturation (Murty et al., 2023). Other studies have also devised ways to explicitly inject structural inductive biases into Transformers, like the TreeReg auxiliary loss function which can softly encourage hierarchy within the model without altering the base architecture or inference mechanism of the model (Nandi et al., 2025). Investigations into neural networks and logical entailment find that for tasks that are purely about understanding and manipulating symbolic structure (like solving ODEs), models with an architectural bias towards this structure tend to have an advantage (Evans et al., 2018), further supporting the feat of incorporating components like the TreeReg loss function into our model.

4 Dataset

Our task for the Transformer simply involves taking in the prefix notation of the ODE equation and output its solution in prefix notation. We, initially generated our own small dataset(DATASET_1) for accomplishing this project. Our dataset was small in size due to the various challenges as noted in **Sub-section 4.1.6**, as well as those noted by the authors in (Lample and Charton, 2019). For solving the problem of translating symbolic mathematical equations, our small dataset couldn't accomplish the task, with non-improving test accuracies, as we show in further sections. We attributed these failures to the size of the dataset, and were proven right later on with our further experiments. Due to the DATASET_1's limitations, we had to procure a second bigger DATASET_2, based on the original first-order ODE dataset used in (Lample and Charton, 2019), which succeeded in accomplishing the task. Below, we first go into detail about our own dataset - DATASET_1, and then explain the details of the bigger dataset - DATASET_2.

4.1 DATASET_1

For the data of this task, we initially extend upon the approach laid out by (Lample and Charton, 2019) to create our own version of the script that only generates first-order ODE's, and with a limited range of operators, symbols, and constants ("vocabulary"). This is done to compensate for the reduction in model size, data size, and computing resources. Here is the vocabulary of our data:

- Binary Operators: add, sub, mul, div, pow
- Unary Operators: sin, cos, tan, exp, log (ln), sqrt

- Symbols: x, y, y', c
- Integers [-100-100]

After generation, filtering, and de-duplication, we obtain a dataset of approximately 34,000 unique equation-solution pairs. The average length for equation prefixes is around 40 tokens, while the solution prefixes average around 30 tokens.

Given an ODE like $y' - 19 = 0$ with a solution $y = c + 19x$, both of them would be transformed into prefix notation, and stored in the dataset as `sub y' 19` and `add c mul 19 x`, respectively, with the $= 0$ and $y =$ parts omitted as they are the same for every equation and hence is implicitly handled. You can quickly verify the validity of a solution by plugging the equation raw string (not prefix) into an online solver like EMathHelp to get the same solution.

Mirroring (Lample and Charton, 2019)'s approach, we chose to represent mathematical expressions in prefix/Polish notation (operator before operands) because it is unambiguous and consequently eliminates the need to introduce parentheses or operator precedence rules to the list of tasks for the Transformer. This type of linear representation also suits the Transformer's linear processing properties in sequence-to-sequence tasks. The further sub-sections go into the specific mathematical detail of how each equation-solution pair was generated to such that a high-quality dataset is ensured.

4.1.1 Dataset Generation

First, we start by generating a random expression to be a candidate solution $f(x, c)$ for the ODE. This is known as the "backward" approach, which is effective because we avoid generating many equations without a valid solution. To ensure this solution can't be trivially solved, it must contain the variable x as well as the constant of integration c.

A common mistake at this point is to use a naive algorithm (e.g. recursive with fixed branching probabilities) to generate the random expressions. If we map out the path probabilities of this approach, we find that the chance of generating each type of expression tree structure is not the same, resulting in an imbalance in the dataset if this method is employed. To solve this, we

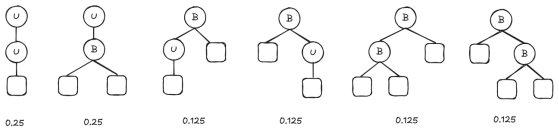

Figure 1: An example of appearance bias for certain types of 2-operator trees, assuming branching and selection of unary (U) and binary (B) operators are both 50%

pre-calculate a distribution tree via dynamic programming, essentially answering "how many tree types can be made if an operator is chosen at this point?" and use this distribution to perform weighted random selection at each node - giving each tree an equal chance of appearance in the dataset.

4.1.2 ODE Derivation

Given a solution $y = f(x, c)$, we attempt to solve for the constant c to obtain an equation in the form of $F(x, y) = c$. Then, we differentiate this relation w.r.t x using the chain rule

$$\frac{\partial F}{\partial x} + \frac{\partial F}{\partial y}\frac{dy}{dx} = 0$$

This resulting equation will be our ODE.

4.1.3 Cleaning, Filtering, and Validation

We "clean up" solutions and equations by fusing sub-expressions (e.g., $(sin(c) + 5)x$ into cx), absorbing numerical constants into symbols (e.g., $c * x * 5$ into cx), and factorizing then only retaining terms relevant to the derivatives and function. Next, we discard pairs that have a very large constant number due to SymPy simplification, contain undefined values like `NaN` or ∞, and contain terms that are not part of our vocabulary e.g. `cot`, `acot`. Finally, we substitute the solution back to the generated ODE to see if it simplifies to 0 (or a numerically negligible value) to make sure it's valid.

4.1.4 Conversion to Prefix and Storage

Validated ODEs and their solutions are converted from SymPy internal representation into prefix notation. We also ensure that each (equation, solution) pair is unique before adding to our dataset.

By performing all of these steps, we build a diverse dataset that well-represents the entire problem space, which can contribute towards better generalization during training.

4.1.5 Challenges

One of the challenges lie in the **non-deterministic representation of math expressions**. Different equations may yield the same solution, and a single equation may yield multiple different solutions. This property may likely be confusing, making it difficult for models, especially smaller ones, to complete the task. The model must also implicitly learn the rules of a valid prefix notation structure.

With that said, one of our biggest challenges was definitely the **vast but sparse solution space**. There are many possible expressions, but only a tiny fraction of those expressions have a clean, well-formed solution that we can include as part of our dataset. As a result, our dataset generation script takes a lot of time to run on consumer hardware, and the rate slows down greatly the longer they run. We even tried to run 12 scripts at once to maximize usage of all available CPU cores, but after 3 days of continuous running, we were only able to gather 34,000 sequences. For comparison, the original (Lample and Charton, 2019) paper had 40 million sequences for training, which has over 1000 times much more data than ours. After experimenting with this dataste, we found out this is insufficient for our task, so we readied the bigger DATASET_2, as we explain below.

4.2 DATASET_2

We assembled DATASET_2 by filtering (Lample and Charton, 2019)'s first-order ODE dataset(ODE1) of 65 Million examples for sequences of length 16 and less. We chose to process sequences of length 16 and less, after observing the statistics on the full ODE1, and being confident that this filtering process will yield us 680K of examples or 1% of the full dataset, which is enough for our small scale training and eval to task to prove our claims. Our newly available dataset consisted of the following operators, symbols and constants:

- Basic Binary Operators: add, sub, mul, div, pow
- Basic Unary Operators: sqrt, exp, log, abs, sign
- Trigonometric Unary Operators: sin, cos, tan
- Inverse Trigonometric Unary Operators: asin, acos, atan
- Hyperbolic Unary Operators: sinh, cosh, tanh
- Inverse Hyperbolic Unary Operators: asinh, acosh, atanh
- Symbols: x, y, y', c
- Integers [0-9], extended to [-9, 9] by INT- and INT+ prefixes

The rest of the charactaristics of our DATASET_2 stay the same as our DATASET_1.

5 Baselines

To evaluate the effectiveness of both beam search and TreeReg, we will be using our baseline model paired with greedy sampling, as random sampling is not suitable for tasks requiring high precision like this one. For the dataset, we split our DATASET_2 into testing and validation sets of sizes *test_size and val_size* respectively, where *test_size* is 1,024, *val_size* is 2,048, and the rest for training the models. We chose such unconventionally low numbers for testing and validation sets, as the time and compute costs for testing and validating with Beam Search is very high, as is also noted in (Lample and Charton, 2019). Hyperparameters are tuned on the validation set, and the final performance metrics will be from evaluation on the held-out test set.

6 Our Approach

Our core approach is to give a Transformer-based seq-2-seq model the task of "translating" a prefix-notated ODE into its matching solution, which is also in prefix notation. Additionally, we experimented on adding TreeReg (Nandi et al., 2025) loss as an additional training loss, to help improve the resulting model by taking advantage of the mathematical tree structure. The TreeReg implemenation is from the authors, and can be found here: https://github.com/ananjan-nandi-9/tree_regularization.

6.1 Model Architecture

We use a standard encoder-decoder Transformer model. The encoder and decoder each have 4 layers, with a model width of 256, and feed forward width of 512. The model has 4 attention heads with an attention head dim of 64, and uses learned positional encodings. Its vocabulary size is 42, with end-of-sequence, begin-of-sequence, and padding special tokens. The other tokens are the operators, variables, and the numbers in the dataset.

6.2 Training Details

Our model is trained with a batch size of 256 using an AdamW optimizer, under a learning rate of 2e-5, weight decay of 0.01, and default betas of (0.9, 0.999). The model is trained for 50 epochs on the training dataset, and the accuracy evaluations are done on the test set. Dropout is set to 0.1.
The test set is 1024 equations, and the validation set is 2048 equations. These numbers were chosen for expediency, as the evaluation is slow, due to the symbolic equality verification to check answer correctness.

We use TreeReg(Nandi et al., 2025) as an additional regularization loss, to help guide our model to use the tree structure of the mathematical data. Tree regularization quantifies indepdendence of tree constituents by using the Span Contextual Independence Score (SCIN). SCIN measures how indepdendent a span is from the context before and after it. TreeReg works by maximizing the SCIN score for spans that are within the same tree, and minimizes the score for spans that are from separate trees.

Following the original TreeReg implementation(Nandi et al., 2025), we use it on the second layer of our model, on 2 attention heads, and apply it once every 20 training steps. The loss weight $\lambda_{TreeReg} = 1.0$ follows the original as well.

	BASELINE	TREEREG
Greedy Sampling	76.56%	78.90%
Beam Size 10	79.69%	82.02%

Table 1: Model performance on 128 sequences of DATASET_2.

6.3 Qualitative Result

Below are some examples of our model output, under beam search of size 10:

Input	Target → Model
add mul INT- 1 exp x y	add c exp x → add c atan tan exp x
add pow x INT+ 2 add mul INT- 1 mul x y y	add x mul x add c x → add x add mul c x mul x add x cos INT+ 1
add pow x INT+ 2 add mul INT- 1 y mul x y	mul x add c mul INT- 1 x → add mul c x mul INT- 1 mul x add x sin INT+ 1
add INT- 1 add y sin x	add c add x cos x → add c add x atan tan cos x
add mul x y mul INT- 2 mul y log y	exp mul c pow x INT+ 2 → exp mul x add mul div INT+ 1 INT+ 4 x mul c x

Table 2: Correct predictions

Although not exactly the same, the model produces output that are numerically equivalent to the solution. However, many of these are redundant operations that can by eliminated via simple symbolic evaluation e.g. tan(atan)

Input	Target → Model
add mul INT- 3 x add mul INT+ 2 y mul x y	add x mul c pow x INT- 2 → add x mul sqrt INT+ 2 sqrt mul c pow x INT- 1
add mul y cos x mul INT- 1 mul y sin x	mul c pow cos x INT- 1 → mul pow x INT- 1 mul add x mul c x sin x
add mul INT- 1 y add mul INT+ 5 y mul x y	mul c add INT+ 5 x → mul c add INT+ 2 0 mul INT+ 2 x
add INT- 1 add mul INT- 1 y add mul x y log x	add mul c x log x → mul x add c mul pow x INT- 1 add x log x
add pow y INT+ 2 add mul INT- 1 y mul x y	mul x pow add c x INT- 1 → pow add INT+ 1 mul c pow x INT- 1 INT- 1

Table 3: Incorrect predictions

Errors can range some simple integer sign changes or a complete shuffling of the targete solution, indicating that the model is struggling with these cases.

6.4 Loss and Validation Graphs

7 Error Analysis

The model's most common errors are definitely incorrect solutions. While syntactically valid, these can either be near misses (structurally similar to the correct answer but contained incorrect coefficients or a single, slightly off operator) or are complete off target (unrelated to the true solution). A major part are not off target, while predictions from expressions that don't appear much

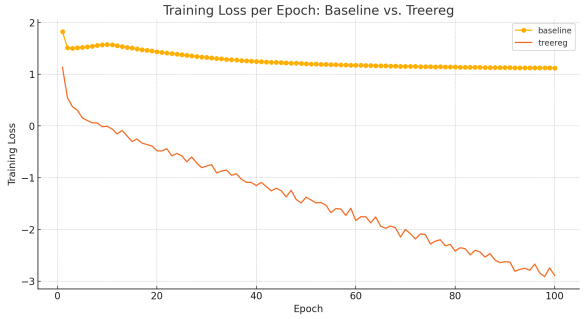

Figure 2: Training Loss comparision on DATASET_1, which shows baseline loss near flat-lining without improvement

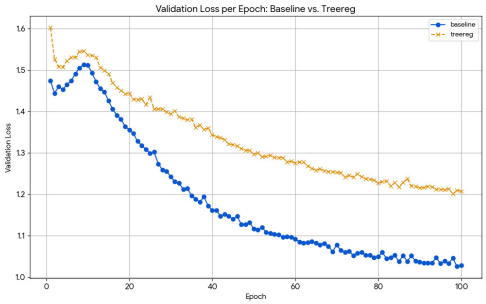

Figure 3: Validation Loss comparision on DATASET_1

are pretty much random - these evidence suggest the model still largely struggles with truly "understanding" the concept behind this task, at least not good enough to generalize to new, even simplest, ODEs. Again, we (confidently) attribute these errors to the lack of abundant training data, as it directly affects our model's ability to generalize.

8 Contributions of group members

- Huy Ngo: contributed to all stages of data generation (script, processing, cleaning, finalization); also handled a major part of the final report
- Sri Rama Bandi: Proposed the idea, project proposal, collaborated model implementation, training and evaluation code, contributed to dataset generation(DATASET_2), experiment analysis and figures, contributed to final report
- Eric Engelhart: Trained and evaluated all the models mentioned in the report, wrote the code to make TreeReg work, fixed bugs in evaluation, and wrote the parts of the report focused on the training details and TreeReg.

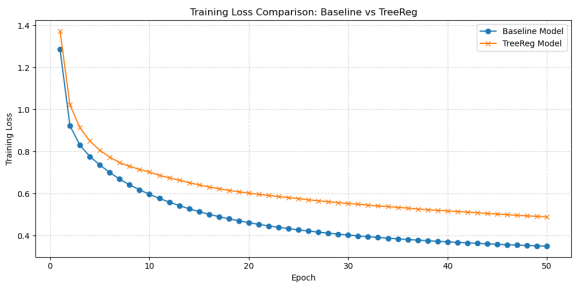

Figure 4: Training Loss comparision on DATASET_2, which shows good improvement in both the models

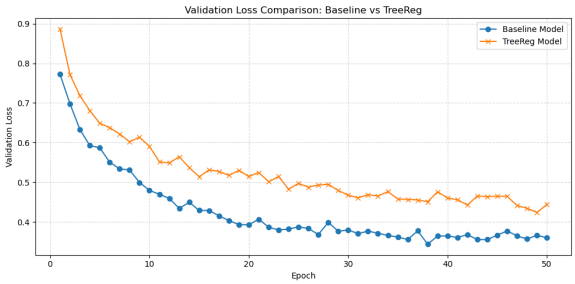

Figure 5: Validation Loss comparision on DATASET_2

9 Conclusion

This project has been a great experience for the team as a whole. We have dealt with the project from end-to-end:

- hypothesis that TREEREG model performs better on this task
- dataset(DATASET_1) generation
- model training and testing using NLP inference methods.
- observing the limitations of our dataset and assembling a bigger dataset
- understanding the results across the experiments

Of the multitude of subtasks involved in all the experiments, dataset generation, understanding TreeReg well enough to incorporate in our model proved to be surprisingly difficult. We have proven our claim, that TreeReg model in fact works better for these types of structured datasets, and that is our takeaway.

9.1 Future Work

These results that in the future, much bigger models with TreeRef auxiliary loss, on bigger datasets will yield better performance compared

to our best model.

10 AI Disclosure

- Did you use any AI assistance to complete this report? If so, please also specify what AI you used.
  - No

If you answered yes to the above question, please complete the following as well:

- If you used a large language model to assist you, please paste *all* of the prompts that you used below. Add a separate bullet for each prompt, and specify which part of the proposal is associated with which prompt.
  - N/A
- **Free response:** For each section or paragraph for which you used assistance, describe your overall experience with the AI. How helpful was it? Did it just directly give you a good output, or did you have to edit it? Was its output ever obviously wrong or irrelevant? Did you use it to generate new text, check your own ideas, or rewrite text?
  - N/A

References

Alemi, A. A., Chollet, F., Een, N., Irving, G., Szegedy, C., and Urban, J. (2017). Deepmath - deep sequence models for premise selection.

Evans, R., Saxton, D., Amos, D., Kohli, P., and Grefenstette, E. (2018). Can neural networks understand logical entailment?

Feng, W., Liu, B., Xu, D., Zheng, Q., and Xu, Y. (2020). NeuReduce: Reducing mixed Boolean-arithmetic expressions by recurrent neural network. In Cohn, T., He, Y., and Liu, Y., editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 635–644, Online. Association for Computational Linguistics.

Lample, G. and Charton, F. (2019). Deep learning for symbolic mathematics.

Murty, S., Sharma, P., Andreas, J., and Manning, C. D. (2023). Grokking of hierarchical structure in vanilla transformers.

Nandi, A., Manning, C. D., and Murty, S. (2025). Sneaking syntax into transformer language models with tree regularization.

Trask, A., Hill, F., Reed, S., Rae, J., Dyer, C., and Blunsom, P. (2018). Neural arithmetic logic units.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

Wang, Y., Liu, X., and Shi, S. (2017). Deep neural solver for math word problems. In Palmer, M., Hwa, R., and Riedel, S., editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 845–854, Copenhagen, Denmark. Association for Computational Linguistics.

Zaremba, W. and Sutskever, I. (2015). Learning to execute.