ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 6 June 2024

Math-101 for Neural Networks

[1] Mohd Nomaan, [2] Dr. Philemon Daniel

[1] [2] ECE Department NIT Hamirpur, India
Corresponding Author Email: [1] 194577@nith.ac.in, [2] phildani7@nith.ac.in

Abstract*— With the role of AI becoming more and more decisive in technological progress and human civilization's trajectory, we must look into this very critical limitation of Neural Networks – their inability to reason and comprehend mathematics. In this research, we investigate the mathematical capabilities of neural networks, focusing on symbolic integration—a pivotal mathematical task with diverse practical applications. We approach integration as neural translation task, using transformers. A novel digit encoding method has also been introduced, which significantly improves the performance of our basic integrator model. The study includes discussions on dataset generation, preprocessing procedures for handling symbolic mathematics data, detailed model architecture, and a comparative analysis of achieved results with mathematical software solutions.

I. INTRODUCTION

AI can be seen as the driving force behind the ever-changing technologies and evolution of human civilization. Neural networks are constantly improving at tasks like recognizing faces and predicting trends. Large Language Models (LLMs), such as ChatGPT, are great at generating and understanding natural language. However, despite their skill in crafting sonnets, identifying faces, and creating art, these advanced models face challenges with basic arithmetic, like multiplication. This limitation of neural networks becomes evident as we move into the age of AI. Addressing this shortfall is crucial if we aim to achieve Artificial General Intelligence (AGI). To enable these systems to think like humans and handle a variety of tasks, including mathematics, we must find ways to improve their abilities in this area.

Several studies have explored how well neural networks can handle mathematics. Google's Minerva project [8] trained an LLM to solve basic high-school math problems. David et al. in [2] created a dataset of math problems and evaluated the performance of LSTM and Transformer models in solving them. The Deep Neural Solver [8] by tencent was an RNN-based model to automatically solve math word problems by translating them into mathematical equations. Francois Charton conducted extensive research on neural networks' ability in tackling calculus [3], linear algebra [4], and recurrent sequences [7].

In this paper, we explore into how effectively neural networks can perform symbolic integration—a fundamental mathematical task with many practical applications. Symbolic integration presents itself as an ideal domain for investigating the mathematical capabilities of neural networks. We propose treating symbolic integration as a natural language translation task, wherein a model learns to translate input functions into their respective anti-derivatives. Our approach begins with an explanation of how we generated a dataset for this task, outlining input and output sequences. We then provide a detailed description of the model architecture used. Finally, we evaluate the model's performance and compare it with existing mathematical software to weigh its effectiveness. Through this exploration, we aim to illuminate the potential of neural networks in handling complex mathematical tasks and pave the way for further advancements in AI-driven mathematical problem-solving.

II. DATASET GENERATION

To prepare a seq-to-seq model for an integration task, we need to convert mathematical functions into a format that an NLP model can understand. The input sequences to the model consist of symbolic functions, while the output sequences contain their corresponding antiderivatives. Creating a dataset of function-integral pairs by manually integrating each function would be time-consuming. Therefore, we use the inverse method of differentiation to generate these pairs. As described in [3], we convert these symbolic expressions into prefix form, simplifying parsing and ensuring uniform expression representation.

For representing the symbolic expressions, we employ functional trees, where each node represents an operator or an operand, as shown in fig. 1. To generate an expression, we start from the topmost node of the tree and iteratively select operators/operands for nodes until the tree reaches its maximum depth or all branches terminate with an operand.

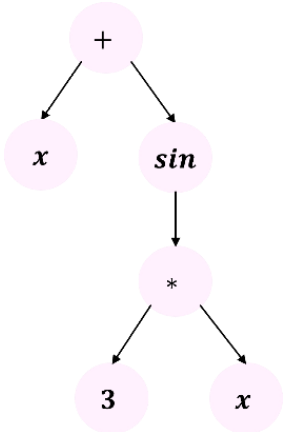

Fig. 1. Functional Tree for f(x) = x + sin(3x)

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 6 June 2024

The algorithm used to generate symbolic expressions is described here:

Algorithm 1 Algorithm for function generation

Input: Maximum number of internal nodes (max_nodes)
Output: A random function tree

```
Initialize a function tree
Generate a node recursively:
  Decrement max_nodes by 1
  if (max_nodes <= 0) then
    Create a leaf node with a random operand
    return node
  Choose a random operator for the node
  if (operator is binary) then
    Generate two child nodes recursively
    return node
  else
    Generate a child node recursively
    return node
Set the returned node as root node
end
```

where,

- Operators can be any unary or binary mathematical operator like +, -, *, /, ^, sine, cosine, tangent, etc. For this paper, considering the computing constraints, the operators are chosen from the set – {+, -, *, ^}
- Operands can be any variable x, y, z, etc. or any real number. For this paper, we consider only univariate functions in x and constants (coefficients or operands) are in the range [-3000,3000]

The functions generated using this algorithm are then converted into prefix notation using Sympy [6] and stored as function-integral pairs. For this study, we collected a dataset of 20,000 function-integral pairs, which are saved in text files labeled p_expr.txt and p_intg.txt respectively. The prefix 'p' indicates that the equations are in polish notation. This method simplifies expression format and ensures consistency in representation.

III. MODEL ARCHITECTURE

We selected the task of mathematical reasoning - Integration of univariate functions, to assess the capability of neural networks. We tackle this challenge as a neural translation task, wherein a sequence-to-sequence model learns to translate input functions into their corresponding integrals. Fig. 2 provides a simple illustration of this process. The sequence-to-sequence model utilized in this study is based on the Transformer architecture, as described in [1]. Before feeding the equations into the transformer for neural translation, preprocessing is necessary. Initially, the equations are segmented into individual tokens, and special tokens like [SOS], [EOS], [UNK] are added to them. Additionally, each tokenized equation is standardized to a fixed size (seq_len) by adding [PAD] tokens. This results in source and target equations in tokenized form. Subsequently, these equations are encoded from text to numeric representation using a vocabulary derived from the tokens. Each token is encoded into a vector with dimensions (1, vocab_size), thereby transforming each equation into the format (seq_len, vocab_size).

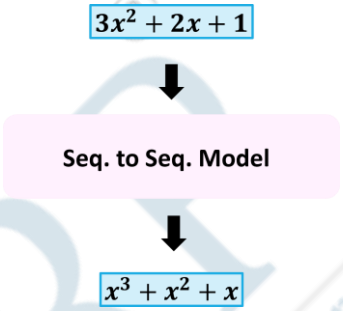

Fig. 2. Symbolic integration using a seq. to seq. model.

Then, we proceed to generate input embeddings from these equations. This process involves converting each token into a multi-dimensional vector with d_model dimensions. These transformed inputs are now prepared to be fed into the transformer model to generate integrals for the input equations. As the equations traverse through the layers of the transformer, they are translated into their corresponding integrals. However, they remain in the vector format (seq_len, d_model). To obtain the final output integrals, a projection layer is used to convert the d_model dimensional vectors into the corresponding tokens. These output integrals are subsequently compared token by token with the actual integrals (referred as labels), and loss is computed using a loss function. This loss is then backpropagated through the layers of the transformer, and finally weights and biases get updated according to that.

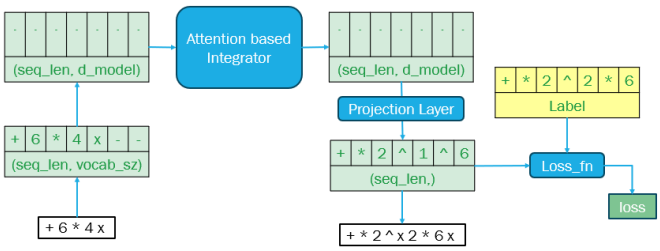

Fig. 3. A basic attention based integrator

This formed the foundational architecture for translating equations into their respective integrals. While this model effectively handled basic functions, we needed to enhance its capabilities to integrate more complex functions with greater precision. Therefore, we implemented several modifications to fine tune the model according to our needs. One significant limitation of the initial version was its treatment of numbers in a manner identical to mathematical symbols. Consequently, a significant portion of the vocabulary was dedicated to numbers, with each possible number considered as a separate

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 6 June 2024

token. To address this issue and retain the numeric essence of coefficients and constants in equations, we devised a solution. We encoded digits as special digit tokens and incorporated the values of these tokens into the input embeddings through concatenation. This refinement enables us to accurately process the numeric information conveyed by the coefficients and constants, a consideration overlooked in the previous model.

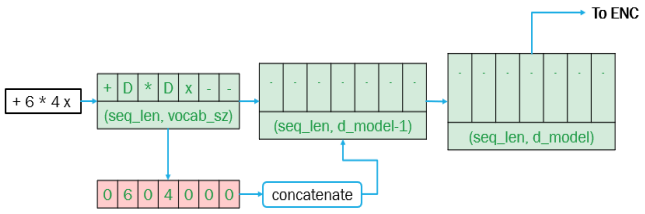

Fig. 4. Digit encoding of input expressions

However, this alternative encoding scheme poses a challenge when it comes to decoding it back into equation form. The output from the transformer is in the format (seq_len, d_model). Therefore, we must decode it into tokens and decode digit tokens into valid numeric values. To address this, we implemented two parallel decoding mechanisms within the projection layer. In the first part, a dense linear layer projects the transformer output (seq_len, d_model) into tokens (seq_len,). Simultaneously, in the second part, a sequential regression layer is employed to project the values of digit tokens. This is achieved through a series of dense layers, allowing us to accurately decode digit tokens into their corresponding numeric values.

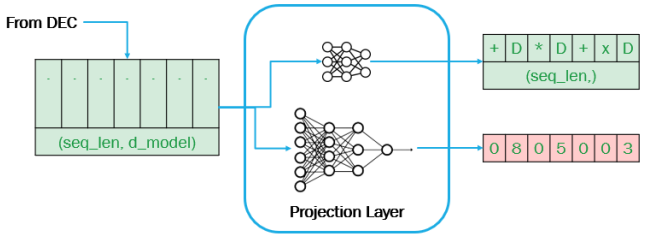

Fig. 5. Projection layer to decode transformer output

The next challenge was to compute the loss based on this output. To tackle this, we devised a custom loss function designed to handle token class mismatches and slight shifts in the numeric values of coefficients differently. In our earlier setup, where all numbers were treated as distinct tokens, even a minor error in the numeric value of a predicted coefficient was treated same as predicting a different variable or operator. In this new function, we employed cross-entropy loss to quantify the error in token prediction and mean square error to quantify the error in the value of digit tokens. The total loss is computed as the weighted sum of these two errors, ensuring a better estimation of model performance.

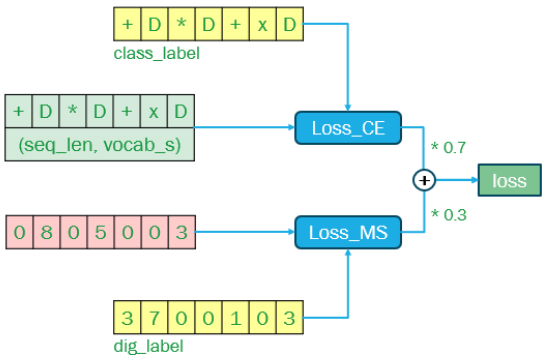

Fig. 6. Loss calculation

IV. TRAINING AND EVALUATION

For neural translation, we employed an encoder-decoder transformer at the core of our Integrator model. We utilized the build_transformer () function to instantiate a transformer, and its details are elaborated here. Initially, both the source and target embeddings undergo positional encoding to integrate order information into the embeddings. The encoder comprises a multi-head self-attention block, which facilitates the transformation of the embeddings to incorporate the relationship of each token with every other token in the input sequence. Subsequently, after the attention layer, a fully connected feedforward layer is added that aids in the learning process. On the other hand, the decoder consists of a multi-head self-attention block, followed by a cross-attention block connecting it to the encoder, and finally a feedforward block. The decoder output is then passed into the projection layer, which yields the final output. The configuration of the model is described in Table 1 providing a comprehensive overview of its architecture.

Table I: Transformer network configuration

N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}
3	256	2048	4	64	64	0.1

where,

N : No. of encoder-decoder blocks

d_{model} : Embedding size

d_{ff} : Feed forward layer upward projection size

h : No of attention heads

d_k : Query and Key vector dimension

d_v : Value vector dimension

P_{drop} : Dropout probability

The dataset comprises 20,000 equation-integral pairs, which are divided into training and validation data in a 9:1 ratio. During training, the data is loaded in batch sizes of 32, while for validation, a batch size of 1 is utilized. The source and target sequence lengths (seq_len) for the equations are set to 80. For weight updates, we employ the Adam optimizer with a learning rate of 1e-4. The model is trained for 50 epochs, and after each epoch, we execute the validation

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 6 June 2024

function to assess the progress. During validation, an equation is passed through the model, and once its integral is obtained in embedding form as output, we further decode it into symbolic form using either greedy decode or beam search technique. Fig. 7 illustrates the distinction between these two decoding methods. This model was implemented using the PyTorch framework, and training was conducted on a T4 GPU in Google Colab.

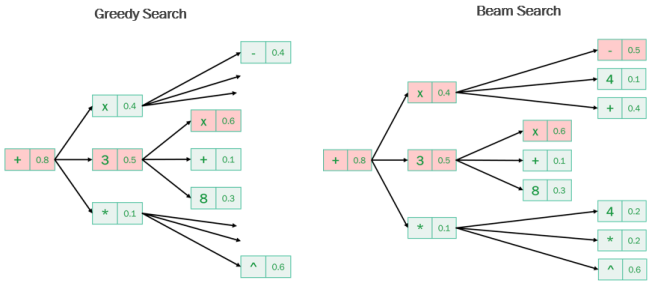

Fig. 7. Greedy and beam search decoding techniques

In neural translation tasks, metrics like BLEU score are commonly used. However, due to the unique nature of our problem and dataset, BLEU scores were not particularly helpful. Therefore, in this paper, we rely on basic token comparison to verify the correctness of our solution. It's important to note that there can be multiple ways to represent an equation, resulting in multiple valid outputs for the same input. To address this issue, we simplify the output integrals before comparing them to the actual integrals using SymPy [6]. After simplification, the accuracy of the model is calculated by comparing the tokens of the output integral with those of the actual integral. This token comparison provides a straightforward and effective means of evaluating the model's performance in our context.

V. RESULTS

The outcomes of this research, along with a comparison to existing mathematical software, are presented in Table 2. The training dataset comprised 18,000 equation-integral pairs, while the remaining 2,000 were allocated for validation. To provide a comparative analysis, the performance of mathematical software such as Mathematica, Matlab, and Maple in symbolic integration was referenced from [3]. Initially, the base integrator model succeeded in integrating around half (52%) of the input equations accurately. However, after incorporating digit encoding and our custom loss function into the base model, a notable improvement in the accuracy was observed. Furthermore, employing beam search (with a beam size of 3) instead of greedy search (with a beam size of 1) resulted in an additional optimization, enhancing the model's performance by 3%. Overall, we could achieve promising performance, comparable to that of existing mathematical software using neural machine translation.

Tabel II: Training and validation results

		Train Accuracy	**Valid. Accuracy**
Mathematical Software	Mathematica	-	84 %
	Matlab	-	65 %
	Maple	-	67 %
Base Integrator Model		53 %	52 %
Integrator with digit encoding	Greedy Search	70 %	64 %
	Beam Search (beam size = 3)	**72 %**	**67 %**

VI. CONCLUSION

In this study, we investigated the effectiveness of neural networks in mathematical reasoning, specifically focusing on the task of symbolic integration. We showcased the potential of neural translation models, particularly encoder-decoder transformers in this domain. Through techniques like digit encoding and custom loss functions, we achieved accuracy levels comparable to traditional mathematical software. While our study examined a limited subset of mathematical problems, it serves as a foundation for addressing the broader challenge of mathematical reasoning in AI models. Continued efforts in this direction hold promise for the development of robust and mathematically aware models capable of reasoning through problems with human-like proficiency.

REFERENCES

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017b). Attention is All you Need. arXiv (Cornell University), 30, 5998–6008. https://arxiv.org/pdf/1706.03762v5

[2] Saxton, D., Grefenstette, E., Hill, F., & Kohli, P. (2019, April 2). Analysing mathematical reasoning abilities of neural models. arXiv.org. https://arxiv.org/abs/1904.01557

[3] Lample, G., & Charton, F. (2019, December 2). Deep learning for symbolic mathematics. arXiv.org. https://arxiv.org/abs/1912.01412

[4] Charton, F. (2021, December 3). Linear algebra with transformers. arXiv.org. https://arxiv.org/abs/2112.01898

[5] Testolin, A. (2024). Can neural networks do arithmetic? A survey on the elementary numerical skills of State-of-the-Art Deep Learning models. Applied Sciences, 14(2), 744. https://doi.org/10.3390/app14020744

[6] SymPy. (n.d.). https://www.sympy.org/en/index.html

[7] D'Ascoli, S., Kamienny, P., Lample, G., & Charton, F. (2022, January 12). Deep symbolic regression for recurrent sequences. arXiv.org. https://arxiv.org/abs/2201.04600

[8] Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., Wu, Y., Neyshabur, B., Gur-Ari, G., & Misra, V. (2022, June 29). Solving Quantitative Reasoning Problems with Language Models. arXiv.org. https://arxiv.org/abs/2206.14858

[9] Lu, P., Qiu, L., Yu, W., Welleck, S., & Chang, K. (2022, December 20). A survey of deep learning for mathematical

reasoning. arXiv.org. https://arxiv.org/abs/2212.10535

[10] Wang, Y., Liu, X., & Shi, S. (2017). Deep Neural Solver for Math Word Problems. In Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (pp. 845–854). Copenhagen, Denmark: Association for Computational Linguistics.

[11] Zhang, D. (2022). Deep learning in automatic Math Word Problem solvers. In Springer eBooks (pp. 233–246). https://doi.org/10.1007/978-3-031-09687-7_14

[12] Thawani, A., Pujara, J., Ilievski, F., & Szekely, P. (2021). Representing Numbers in NLP: A Survey and a Vision. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (pp. 644–656). Online: Association for Computational Linguistics