SIRD: Symbolic Integration Rules Dataset

Vaibhav Sharma[*]
sharmavaibhav1729@gmail.com

Abhinav Nagpal [*†]
abhinav.nagpal1@aexp.com

Muhammed Fatih Balın[‡§]
balin@gatech.edu

Abstract

Advancements in neural networks and computer hardware lead to new use cases for deep learning in the natural sciences every day. Even though symbolic mathematics tasks have been explored, symbolic integration only has a few studies using black box models and currently lacks explainability. Symbolic integration is a challenging search problem and the final result is obtained by applying different integration rules at each step. We propose a novel and interpretable approach to perform symbolic integration using deep learning through integral rule prediction to speed up the search. We introduce the first-of-its-kind symbolic integration rules dataset comprising two million distinct functions and integration rule pairs. For complex rules such as u-substitution and integration by parts, it also includes the expression needed for rule application. We also train a transformer model on our proposed dataset and incorporate it into SymPy's *integral_steps* function to get *guided_integral_steps*, resulting in $6\times$ fewer branches explored by allowing our model to guide the depth-first-search procedure.

1 Introduction

We utilize symbolic mathematics and computer algebra systems to help us solve mathematical problems. By working with operators, symbols, and sequences, computers can tackle everything from simple equations to complex calculus, including differentiation, differential equations, and integration. The use of computers for mathematics has become essential in fields such as computer science, telecommunications systems, engineering, medicine, and many more. Humans possess impressive abilities for abstract mathematical and logical thinking. Our understanding and ability to solve mathematical problems not only rely on inference, utilizing laws, axioms, and symbol manipulation rules but also on our experience.

For many years, scientists have aimed to create machines that learn and reason independently. Previous studies (Huang et al., 2023; Piotrowski et al., 2019; Zaremba et al., 2014; Loos et al., 2017) have explored the potential of deep learning for mathematical reasoning. They focus on using deep learning for arithmetic tasks such as integer addition and multiplication (Kaiser & Sutskever, 2015; Zaremba et al., 2014), solving word problems (Huang et al., 2023; Wang et al., 2018; Chatterjee et al., 2022), and performing calculus (Lample & Charton, 2020; Panju & Ghodsi, 2020). However. the current attempts remain far from exceeding the performance of a human expert.

Recently, with the release of large language models (LLMs), testing their math-solving capability became evident in the community (Ji & Gao, 2023; Tang et al., 2023; Frieder et al., 2023). Primarily,

[*]Equal contribution
[†]American Express, AI Labs
[‡]School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA
[§]Corrosponding author.

37th Conference on Neural Information Processing Systems (NeurIPS 2023) Workshop on MATH-AI.

they investigate mathematical problems expressed in natural language involving numbers. On the other hand, calculus is free of words and is expressed via a combination of various symbols and expressions. These problems are either not explored enough or explored in a more end-to-end black-box fashion (Lample & Charton, 2020; Noorbakhsh et al., 2021). There are some previous works (Rich et al.; Meurer et al., 2017) that have also attempted to create parsing rules for different mathematical functions and to apply calculus rules to solve the problems in heuristic ways. Calculus problems can be formulated as search problems, where the final solution is multiple mathematical steps away.

With all the above considerations, we introduce the first-of-its-kind Symbolic Integration Rules Dataset (SIRD-2M), comprising 2 million function-integration rule pairs to attack the interpretable symbolic integration problem. The dataset includes examples of what integration rule needs to be applied at each step of integration for different functions. Moreover, it also includes an expression along with the integration rules, for the more complicated u-substitution and integration by-parts rules. We automate human-like step-by-step integration using integral rules predicted by a model trained on the SIRD dataset. This process is more efficient and interpretable compared to the existing studies that tend to predict the integral expression directly. SIRD can be employed to train a highly generalizable model, as demonstrated by our strong benchmarks. Our goal is to encourage more research in symbolic integration and other symbolic math problems. We think that general computer algebra systems can be made much more capable by utilizing an approach similar to ours.

To summarise, the major contributions of our work are as follows:

- Introduce SIRD-2M[5], a large-scale dataset comprising 2 million functions corresponding to 24 integration rules with additional expressions for certain integration rules.
- Benchmark three integration-related tasks - Complete Rule Prediction, Rule Prediction, and Integral Prediction.
- Modify *integral_steps* method of SymPy (Meurer et al., 2017), a symbolic maths package, and propose *guided_integral_steps* by looking at the outputs of a transformer model trained on SIRD at each integration step and dynamically guiding the depth-first-search procedure. We show that $6\times$ fewer branches are explored compared to *integral_steps*, whose search procedure depends on predefined static heuristics.

2 Symbolic Integration Rules Dataset (SIRD)

SIRD-2M consists of 2 million functions and the corresponding integration rules. The dataset includes samples for 24 different integration rules (see Appendix E), including the *u-substitution* and the *integration by parts* which require a function expression when applied. For such cases, we also provide the expression to be used along with the rule.

2.1 Data Source

The recent work by Lample & Charton (2020) introduced three datasets for symbolic integration - Forward (FWD), Backward (BWD), and Integration by Parts (IBP). These datasets comprise function-integral pairs and are produced using three different methods (see Appendix A for more information). For our study, we used functions from the FWD dataset to generate samples for SIRD. To this end, we modified the SymPy library's *integral_steps* function to produce the function and integration rule pairs during each step of integration. It is worth noting that integrals of the functions from FWD were originally generated using SymPy's *integrate* function. See Appendix B for more details on the mentioned SymPy functions and modified *integral_steps* to generate SIRD samples.

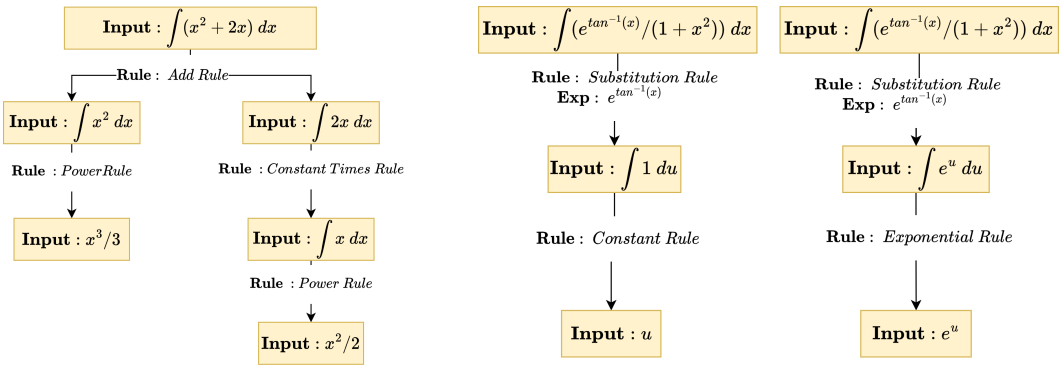
2.2 Data Generation, Validation & Processing

Data generation: As mentioned above, the FWD dataset consists of pairs of functions and corresponding integrals. When solving an integration problem, a human typically follows a step-by-step process. Given a function, a human contemplates which rule to apply to the function, applies it to obtain an expression, and repeats the process until it arrives at the final integral. We created

[5]Publicly available at `https://github.com/mfbalin/SIRD-Symbolic-Integration-Rules-Dataset`.

Table 1: SIRD examples.

Function	Rule	Transformation Expression	Model Input Sequence	Model Output Sequence
$x^2 + 2x$	add_rule	-	[add, pow, x, INT+, 2, mul, INT+, 2, x]	[add_rule]
$\exp(tan^{-1}(x))/(1+x^2)$	substitution_rule	$\exp(tan^{-1}(x))$	[mul, pow, add, INT+, 1, pow, x, INT+, 2, INT-, 1, exp, atan, x]	[substitution_rule, exp, $atan$, x]



(a) $x^2 + 2x$ (b) $\frac{\exp(tan^{-1}(x))}{1+x^2}$

Figure 1: A tree showing the rules chosen by the model while performing a search.

the samples for SIRD by extracting all pairs of expression and integration rules that appear in the intermediate steps of this process. For this purpose, we used the *integral_steps* function available in SymPy, which can provide all the intermediate steps associated with various rules applied to the subproblems of the function to obtain its integral. However, the syntax of the output generated by *integral_steps* is not straightforward and is not directly usable for our purposes. That is why, to obtain the exact expression-integration rule pairs for SIRD, we modified the individual integration rule functions from SymPy and the flow of the *integral_steps* function, creating a data generation script. As a result, one sample from the FWD dataset contributed multiple samples in SIRD - one for each step towards solving the integral of a function, after which we removed the duplicate entries. Appendix B.2 provides examples and a detailed explanation of the output syntax and step-by-step data generation, and Appendix C details how we ensured correctness.

Data processing: After generating the function and the corresponding rules, each subproblem-rule pair is separated, forming a sample for SIRD. There are two types of rules in the dataset: *a)* rules that come with an accompanying expression, requiring an additional expression to be applied (such as *u-substitution*), and *b)* simple rules like *add rule*, *multiplication rule*, etc. For a single SIRD sample, the expression is the input sequence to the model, and the integration rule along with possibly the additional expression is the output sequence. Similar to Lample & Charton (2020), we transform the input expressions and expressions accompanying rule name in certain output sequences to prefix (polish) tree notation to train our model, see Table 1 for details.

3 Experiments

We trained a sequence-to-sequence model that predicts the name of the next rule to be applied along with an expression if applicable. We used the transformer model from Vaswani et al. (2017); Lample & Charton (2020), as it has been empirically tested to perform relatively well for the symbolic integration task so we kept the same configuration, see Appendix D for details. We have only used input expressions with less than 384 tokens in prefix form (see Appendix F) to train our model on 1.6 million samples (80% of SIRD). We compare the predicted value and true label for various tasks, so our evaluation metric is accuracy. We define the following tasks and their evaluation criteria:

- **Complete Rule Prediction:** Evaluating if both the predicted rule and the expression (if applicable) are the same as ground truth to consider a prediction correct.

Table 2: Results for different tasks.

Task	Dataset	Number of Samples	Approach	Additional Granularity	Accuracy (%)
Complete Rule Prediction	SIRD-2M	210976	Our model	–	81.35
Rule Prediction	SIRD-2M	210976	Our model	–	82.87
		82832		Add Rule	99.99
		43190		Multiplication Rule	97.99
		16759		Substitution Rule	57.40
		9262		Parts Rule	66.76
		58933		Other Rules	57.47
Integral Prediction	FWD	7000	integral_steps	–	95.47
			guided_integral_steps	–	95.93
			Lample & Charton (2020)	–	97.38

- **Rule Prediction:** Evaluating if only the predicted rule is the same as the ground truth to consider a prediction correct.
- **Integral Prediction:** For this task we created *guided_integral_steps* by incorporating our model into the *integral_steps* function of SymPy. So instead of searching heuristically, *integral_steps* makes inference calls to our model at every intermediary step and explores different rules in an order based on our model's predictions. For correctness, we differentiate the final result and compare it with the original function.

Test set for different tasks: We divide SIRD into 80% for training, 10% for validation and 10% for testing. We benchmark our model's performance and generalizability using different test sets for different tasks. For Complete Rule Prediction and Rule Prediction tasks, we used the 10% test set of SIRD. We also measure the accuracy of individual rules on the same test set. Note that for the rule prediction tasks, there may be multiple correct rules to apply at any point, and our dataset includes only one correct rule as the label. For **Integral Prediction**, we used the test set of the FWD dataset for evaluation. However, we need to mention that the current version of SIRD does not support functions having Hyperbolic Trigonometric Functions (e.g. *sinh(x)* and *cosh(x)*) as the *integral_steps* function is not capable handling these types of functions. Therefore, we removed these functions from the test set of the FWD dataset before evaluation.

In the implementation of *integral_steps*, the SymPy function corresponding to each integration rule is called sequentially in a defined order. At each intermediary step, the applicability of the integration rule is checked before advancing to the next one until the integral of the function is found. In essence, it performs a depth-first search through all the defined rules at each intermediate step when integrating a given function. In the *guided_integral_steps*, we modified and removed the predefined order of integration rules. Instead, we run an inference through our model with the current expression as input, which outputs probability scores for each integration rule, based on which they are ranked. This way, the order of integration rules explored during depth-first-search becomes dynamic as it depends on the current expression.

4 Results and observations

4.1 Accuracy Comparison

Table 2 shows that *guided_integral_steps* outperforms the original *integral_steps* function, i.e. the dataset generator itself on the test set from the FWD dataset. This shows the generalizability of our approach over the dataset generator itself, which we discuss in Section 4.3. It should be noted that the *integrate* function from SymPy was used to generate the FWD dataset, and it can integrate a larger class of functions by using the Risch-Norman algorithm (Risch, 1969). In contrast, *integral_steps* is essentially based on hand-crafted patterns and heuristics-based integration rule search. Hence, *integral_steps* can not integrate all the functions which *integrate* can, and this limitation extends to *guided_integral_steps*.

4.2 Efficiency Comparison

The average runtime of *guided_integral_steps* was **0.21s** compared to **0.48s** for *integral_steps*, resulting in a **2.28**$\times$ improvement. *guided_integral_steps* explored an average number of **16.7**

branches before reaching the integral compared to **94.2** branches for *integral_steps*, making the depth-first search procedure around **6**$\times$ more efficient. See Section 4.4 for more details.

4.3 Generalization Beyond the Generator - SymPy *integral_steps*

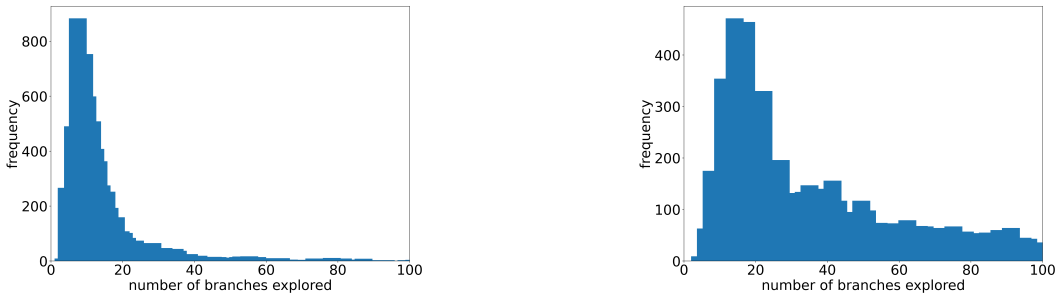
We have observed that *guided_integral_steps* can integrate some functions that *integral_steps* can not, presented in Table 3. This indicates the generalizability of our approach over the original data generation process.

Table 3: Examples where the original *integral_steps* fails but *guided_integral_steps* succeeds.

Function	**Integral**
$e^x + (\cos^{-1}(x))^2$	$x(\cos^{-1}(x))^2 - 2x - 2\sqrt{1-x^2}(\cos^{-1}(x))^2 + e^x$
$x + (\sin^{-1}(x))^2$	$\frac{x^2}{2} + x(\sin^{-1}(x))^2 + 3x + 2\sqrt{1-x^2}\sin^{-1}(x)$
$\sin(\sqrt{x}\tan(5))$	$\frac{2(-\sqrt{(x)}\cos(\sqrt{(x)}\tan(5))+\frac{\sin(\sqrt{(x)}\tan(5))}{\tan(5)}}{\tan(5)}$
$(\cos^{-1}(x))^2 + \cos^{-1}(x) + \frac{1}{4}$	$x(\cos^{-1}(x))^2 + x\cos^{-1}(x) - \frac{7x}{4} - 2\sqrt{1-x^2}\cos^{-1}(x) - \sqrt{1-x^2}$

4.4 Depth-First-Search - # Branches Explored

We have measured the number of branches explored by *integral_steps* and *guided_integral_steps* during their search procedures on the FWD test set. Fig. 2 shows that *guided_integral_steps* explores fewer than 20 branches for most of the functions in contrast to 100 branches for *integral_steps*.



(a) *guided_integral_steps* (b) *integral_steps*

Figure 2: Histograms of # branches explored during depth-first-search on the FWD test set.

5 Conclusion

We present a novel approach to the symbolic integration problem via deep learning. We frame integration as a search problem and accelerate it using AI, resulting in a fast, accurate, and interpretable approach. We introduce a new dataset called SIRD, where the task is to predict the integration rule that should be applied to a given function to find its integral. We show that a model trained on a portion of SIRD can be used to guide the search for the integral, outperforming heuristics-based search and showing superior generalization ability. Our work is a preliminary exploration of using deep learning for step-by-step symbolic integration, leading the way for further research on the topic.

6 Acknowledgements

We thank the Fatima Fellowship[6] and Hugging Face for organizing and sponsoring the Fatima Research Fellowship program.

References

Oishik Chatterjee, Isha Pandey, Aashish Waikar, Vishwajeet Kumar, and Ganesh Ramakrishnan. Warm: A weakly (+ semi) supervised math word problem solver. In *Proceedings of the 29th International Conference on Computational Linguistics*, pp. 4753–4764, 2022.

Simon Frieder, Luca Pinchetti, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Christian Petersen, Alexis Chevalier, and J J Berner. Mathematical capabilities of chatgpt. *ArXiv*, abs/2301.13867, 2023. URL https://api.semanticscholar.org/CorpusID:256415984.

Zeyu Huang, Xiaofeng Zhang, Jun Bai, Wenge Rong, Yuanxin Ouyang, and Zhang Xiong. Solving math word problems following logically consistent template. In *2023 International Joint Conference on Neural Networks (IJCNN)*, pp. 01–08. IEEE, 2023.

Yu Ji and Song Gao. Evaluating the effectiveness of large language models in representing textual descriptions of geometry and spatial relations. *ArXiv*, abs/2307.03678, 2023. URL https://api.semanticscholar.org/CorpusID:259375953.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL https://arxiv.org/abs/1412.6980.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=S1eZYeHFDS.

Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL https://doi.org/10.7717/peerj-cs.103.

Kimia Noorbakhsh, Modar Sulaiman, Mahdi Sharifi, Kallol Roy, and Pooyan Jamshidi. Pretrained language models are symbolic mathematics solvers too! *ArXiv*, abs/2110.03501, 2021. URL https://api.semanticscholar.org/CorpusID:238419670.

Maysum Panju and Ali Ghodsi. A neuro-symbolic method for solving differential and functional equations. *arXiv preprint arXiv:2011.02415*, 2020.

Bartosz Piotrowski, Josef Urban, Chad E Brown, and Cezary Kaliszyk. Can neural networks learn symbolic rewriting? *arXiv preprint arXiv:1911.04873*, 2019.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. URL https://arxiv.org/abs/2009.03393.

Albert Rich, Patrick Scheibe, and Nasser Abbasi. Rule-based integration: An extensive system of symbolic integration rules. *Journal of Open Source Software*, 3(32):1073. doi: 10.21105/joss.01073.

[6] cf. https://www.fatimafellowship.com/

Robert H Risch. The problem of integration in finite terms. *Transactions of the American Mathematical Society*, 139:167–189, 1969.

Xiaojuan Tang, Zilong Zheng, Jiaqi Li, Fanxu Meng, Song-Chun Zhu, Yitao Liang, and Muhan Zhang. Large language models are in-context semantic reasoners rather than symbolic reasoners. *ArXiv*, abs/2305.14825, 2023. URL https://api.semanticscholar.org/CorpusID:258865899.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Lei Wang, Dongxiang Zhang, Lianli Gao, Jingkuan Song, Long Guo, and Heng Tao Shen. Mathdqn: Solving arithmetic word problems via deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. *Advances in Neural Information Processing Systems*, 27, 2014.

A Symbolic Integration Datasets from Lample & Charton (2020)

Lample & Charton (2020) released the following three symbolic integration datasets:

- **Forward Dataset (FWD):** For this dataset, *integrate* function from SymPy (Meurer et al., 2017) was used. The direction of generation was from generated functions to integrals.
- **Backward Dataset (BWD):** Fot this dataset, randomly generated functions were differentiated. Hence, the function becomes integral of its differential. The direction of generation was from generated integrals to functions(differential of integrals).
- **Backward generation with integration by parts (IBP):** For this dataset, Integration by parts was used to generate the function-integration pair. For two randomly generated functions u and v:

  $$\int udv = uv - \int vdu$$

  Based on which of $\int udv$ and $\int vdu$ is already present in the generated data another one can be generated.

The major difference between these methods is that for BWD samples, integrals are shorter than the functions, while for the FWD samples, it is the other way around.

In this work, we have used the FWD dataset. As our problems deal with the intermediary steps' integration rules while solving integral of the functions rather than actual integrals, the ideal source dataset would be with diverse functions and one with short length functions so that *integral_steps* can complete heuristics search faster on a function. Because of the nature of the generation of the FWD dataset and that it was generated using *integrate* function from SymPy, *integral_steps* would be able to solve for functions relatively faster while generating samples for SIRD and hence relatively more number of samples compared to other two datasets.

B SymPy Functions & SIRD Sample Generation Script

B.1 Functions from SymPy

For the generation of FWD, Lample & Charton (2020) used *integrate* function from SymPy and we used the modified version of *integral_steps* function to generate samples for SIRD utilizing functions from FWD. Here, we provide a brief explanation for both the functions and on what grounds these differ.

- ***Integrate*** Function: This is the principal method to integrate functions in SymPy. This function uses the Risch-Norman algorithm (Risch, 1969) and can solve both definite and indefinite integrals, though it doesn't provide the steps of integration.
- ***Integral_steps*** Function:
  - This function imitates how a human would solve an integral problem step by step. It outputs all the intermediate steps (expressions and integration rules) required to solve the integral of a given function. Further, these intermediate steps can be input to another function ***_manualintegrate*** which can apply the steps to generate the final integral of the function.
  - To calculate output of *integral_steps*, various integration rules' are implemented in SymPy. Whenever it is called with an input, it heuristically searches which integral rule to apply based on a defined order in the codebase. There are certain functions such as the *add_rule* which recursively again calls *integral_steps* to solve the subproblems. It tries all the potential rules in a predefined order to either output steps to solve the integration of a function or fallback to *DontKnowRule*.

It is clear based on the description, that *integrate* function is far more capable to solve for an end-to-end integration problem compared to *integral_steps*. However, for our study, we rely on the latter as we require integration rules instead of just the final integral for a function. Also currently, *integral_steps* is not very capable of generating integration rules for complex functions involving hyperbolic trigonometric functions.

Given the nature of *integral_steps* implementation for many functions, it might go into an infinite search loop which either can be broken by *timeouts* or *Maximum Call Stack Exceeded*.

B.2 Modified *integral_steps* to Generate SIRD Samples

As described in section B.1, *integral_steps* function generates all the intermediary steps' expressions and integration rules while solving for a function using heuristically searching for correct rules to apply. But it generates its output in certain syntax which can be highly dynamic for different types of inputs. Following are the examples of its output:

Function: $x^2 + 2x$
Output from *integral_steps*: *AddRule(substeps=[PowerRule(base=x, exp=2, context=x**2, symbol=x), ConstantTimesRule(constant=2, other=x, substep=PowerRule(base=x, exp=1, context=x, symbol=x), context=2*x, symbol=x)], context=x**2 + 2*x, symbol=x)*

Function: $e^{tan^{-1}(x)}/(1 + x^2)$
Output from *integral_steps*: *AlternativeRule(alternatives=[URule(u_var=_u, u_func=exp(atan(x)), constant=1, substep=ConstantRule(constant=1, context=1, symbol=_u), context=exp(atan(x))/(x**2 + 1), symbol=x), URule(u_var=_u, u_func=atan(x), constant=1, substep=ExpRule(base=E, exp=_u, context=exp(_u), symbol=_u), context=exp(atan(x))/(x**2 + 1), symbol=x)], context=exp(atan(x))/(x**2 + 1), symbol=x)*

Writing parsing rules for the above output can make things unnecessarily complex and there can be many exceptions given the highly dynamic nature of output syntax. Hence, to get exact subexpression-integration rule pairs for a function, we created a data generation script by doing the following:

- Modified each integration rule function in SymPy to output a tuple of subexpression and rule name.
- For rules like *substitution_rule* which require to transform a subexpression of a function to get applied we also added a subexpression to be transformed along with the rule name in the output tuple.
- Modified the flow of *integral_steps* to accommodate this extra output along with the original.

Following are examples of output from our data generation script for the same functions:

Function: $x^2 + 2x$
Output from Our Script: *(AddRule(substeps=[PowerRule(base=x, exp=2, context=x**2, symbol=x), ConstantTimesRule(constant=2, other=x, substep=PowerRule(base=x, exp=1, context=x, symbol=x), context=2*x, symbol=x)], context=x**2 + 2*x, symbol=x),*
*[(IntegralInfo(integrand=**x**2 + 2*x**, symbol=x), **'add_rule'**),*
*(IntegralInfo(integrand=**x**2**, symbol=x), **'power_rule'**),*
*IntegralInfo(integrand=**2*x**, symbol=x), **'mul_rule'**,*
*(IntegralInfo(integrand=**x**, symbol=x), **'power_rule'**)]).*

Function: $e^{tan^{-1}(x)}/(1 + x^2)$
Output from Our Script: *(AlternativeRule(alternatives=[URule(u_var=_u, u_func=exp(atan(x)), constant=1, substep=ConstantRule(constant=1, context=1, symbol=_u), context=exp(atan(x))/(x**2 + 1), symbol=x), URule(u_var=_u, u_func=atan(x), constant=1, substep=ExpRule(base=E, exp=_u, context=exp(_u), symbol=_u), context=exp(atan(x))/(x**2 + 1), symbol=x)], context=exp(atan(x))/(x**2 + 1), symbol=x),*
*[(IntegralInfo(integrand=**exp(atan(x))/(x**2 + 1)**, symbol=x), **'substitution_rule', exp(atan(x))**),*
*(IntegralInfo(integrand=**1**, symbol=_u), **'constant_rule'**),*
*(IntegralInfo(integrand=**exp(atan(x))/(x**2 + 1)**, symbol=x), **'substitution_rule', atan(x)**),*
*(IntegralInfo(integrand=**exp(_u)**, symbol=_u), **'exp_rule'**)]).*

This way it becomes straightforward to parse the expression-integration rule pairs for a function constituting SIRD samples.

C Data Validation

Our data generation script generates a list of expression-integration rule pairs and the original output of the *integral_steps* function. We validated the correctness of the data generation script using two methods. Firstly, we compared the outputs of *guided_integral_steps* and *integral_steps*. Secondly, we applied the generated rules to obtain the integral and then differentiated it back to compare it with the original function. This ensured that the data generation script generated integration rules correctly.

D Architecture and Hyperparameters

We used transformer architecture with the following hyperparameters: 8 attention heads, 6 layers, and an embedding size of 512. We have used Adam optimizer (Kingma & Ba, 2014) for training our model with a learning rate of $4 * 10^{-5}$. We trained the model on batches of 256 samples. We limited the number of tokens in input expressions to 384 while processing data for training the model.

E SIRD-2M: Dataset Statistics

Symbolic Integration Rules Dataset (SIRD) consists of 2 million+ samples including 24 integration rules. Table 4 lists the frequency of different integration rules in SIRD.

Table 4: Integration rules and their frequencies in SIRD-2M.

Rule Name	**No. of Samples**
add_rule	827305
mul_rule	431507
partial_fractions_rule	224271
substitution_rule	169110
cancel_rule	166081
distribute_expand_rule	124024
parts_rule	93199
sqrt_linear_rule	27499
quadratic_denom_rule	19418
constant_rule	12928
trig_rule	4780
trig_expand_rule	4134
sqrt_quadratic_rule	3400
trig_sincos_rule	785
inverse_trig_rule	589
power_rule	339
trig_sindouble_rule	273
trig_tansec_rule	172
special_function_rule	24
trig_cotcsc_rule	15
trig_substitution_rule	7
hyperbolic_rule	6
exp_rule	2
trig_product_rule	1

F Input Expression Length

Before training a sequence-to-sequence model on SIRD-2M, we filtered the training data samples by the following criteria: the input expression prefix form should not exceed 384 tokens in length, and the output sequence, which includes both the rule name and the accompanying expression for eligible rules should be no longer than 29 tokens. Fig. 3 displays the distribution of input sequence lengths for all samples in SIRD-2M.

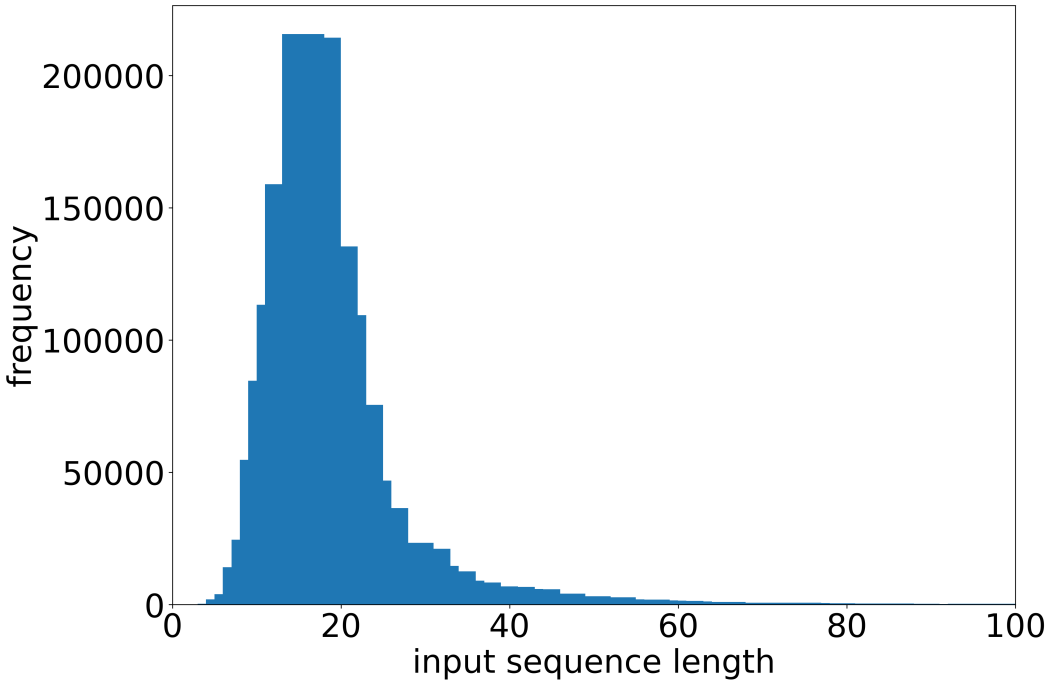


Figure 3: The histogram of the sequence lengths for functions in SIRD-2M.