DEEP LEARNING FOR SYMBOLIC MATHEMATICS

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural networks have a reputation for being better at solving statistical or approximate problems than at performing calculations or working with symbolic data. In this paper, we show that they can be surprisingly good at more elaborated tasks in mathematics, such as symbolic integration and solving differential equations. We propose a syntax for representing these mathematical problems, and methods for generating large datasets that can be used to train sequence-to-sequence models. We achieve results that outperform commercial Computer Algebra Systems such as Matlab or Mathematica.

1 INTRODUCTION

A longstanding tradition in machine learning opposes rule-based inference to statistical learning (Rumelhart et al., 1986), and neural networks clearly stand on the statistical side. They have proven to be extremely effective in statistical pattern recognition and now achieve state-of-the-art performance on a wide range of problems in computer vision, speech recognition, natural language processing (NLP), etc. However, the success of neural networks in symbolic computation is still extremely limited: combining symbolic reasoning with continuous representations is now one of the challenges of machine learning.

Only a few studies investigated the capacity of neural network to work with mathematical objects, and apart from some a few exceptions (Zaremba et al., 2014; Loos et al., 2017; Allamanis et al., 2017), the majority of these works focus on arithmetic tasks like integer addition and multiplication (Zaremba & Sutskever, 2014; Kaiser & Sutskever, 2015; Trask et al., 2018). On these tasks, neural approaches tend to perform poorly, and require the introduction of components biased towards the task at hand (Kaiser & Sutskever, 2015; Trask et al., 2018).

In this paper, we consider mathematics, and particularly symbolic calculations, as a target for NLP models. More precisely, we use sequence-to-sequence models (seq2seq) on two problems of symbolic mathematics: function integration and differential equations. Both are difficult, even for trained humans. Humans typically learn a set of rules, techniques and tricks (integration by parts, change of variable, etc.), that are not guaranteed to succeed, while Computer Algebra Systems solve equations using complex algorithms (Geddes et al., 1992). For instance, the complete description of the Risch algorithm (Risch, 1970) for function integration is composed of more than 100 pages.

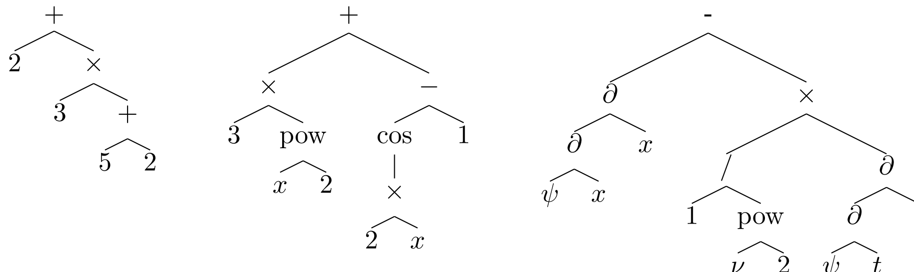
Yet, function integration is actually an example where pattern recognition should be useful: detecting that an expression is of the form $2yy'(y^2+1)^{-1/2}$ suggests that its primitive will contain $\sqrt{y^2+1}$. Detecting this pattern may be easy for small expressions y, but becomes more difficult as the number of operators in y increases. However, to the best of our knowledge, no study has investigated the ability of neural networks to detect patterns in mathematical expressions.

We first propose a representation of mathematical expressions and problems that can be used by sequence to sequence models, and discuss the size and structure of the resulting problem space. Then we show how to generate datasets for supervised learning of integration and first and second order differential equations. Finally, we apply transformer networks to these datasets, and achieve a better performance than state-of-the-art computer algebra programs, namely Matlab and Mathematica.

2 MATHEMATICS AS A NATURAL LANGUAGE

2.1 EXPRESSIONS AS TREES

Mathematical expressions can be represented as trees, with operators and functions as internal nodes, operands as children, and numbers, constants and variables as leaves. The following trees represent expressions $2 + 3 \times (5 + 2)$, $3x^2 + \cos(2x) - 1$, and $\frac{\partial^2 \psi}{\partial x^2} - \frac{1}{\nu^2}\frac{\partial^2 \psi}{\partial t^2}$:



Trees disambiguate the order of operations, take care of precedence and associativity and eliminate the need for parentheses. Up to the addition of meaningless symbols like spaces, punctuation or redundant parentheses, different expressions result in different trees. With few assumptions, discussed in Section A of the appendix, there is a one-to-one mapping between expressions and trees.

We consider expressions from a syntactical perspective, as meaningless sequences of mathematical symbols. $2 + 3$ and $3 + 2$ are different expressions, as are $\sqrt{4}x$ and $2x$: they will be represented by different trees. $x\ /\ 0$, $\sqrt{-2}$ or $\log(0)$ are also legitimate expressions, even though they do not necessarily make mathematical sense. In practice, expressions represent meaningful mathematical objects. Since there is a one-to-one correspondence between trees and expressions, equality between expressions will be reflected over their associated trees, as an equivalence: since $2 + 3 = 5 = 12 - 7 = 1 \times 5$, the four trees corresponding to these expressions are equivalent. For instance, expression simplification, a standard problem in formal mathematics, amounts to finding a shorter equivalent representation of a tree.

In this paper, we consider two problems: symbolic integration and differential equations. Both these problems boil down to transforming an expression into another, e.g. mapping the tree of an equation to the tree of a solution that satisfies it. We regard this as a particular instance of machine translation.

2.2 TREES AS SEQUENCES

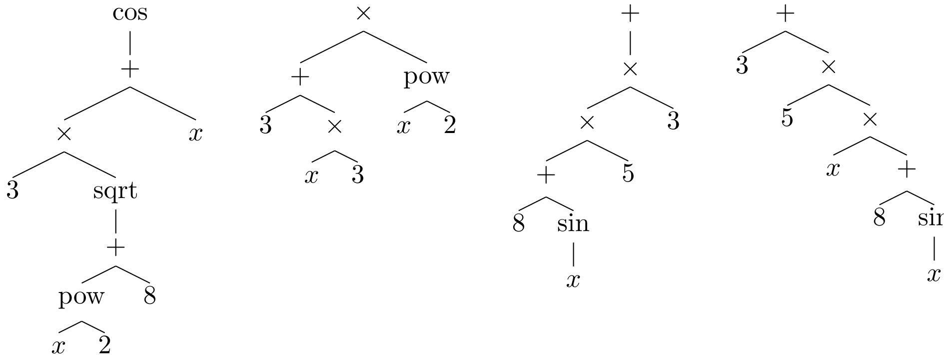
Machine translation systems typically operate on sequences (Sutskever et al., 2014; Bahdanau et al., 2015). Alternative approaches have been proposed to generate trees, such as Tree-LSTM (Tai et al., 2015) or Recurrent Neural Network Grammars (RNNG) (Dyer et al., 2016; Eriguchi et al., 2017). However, tree-to-tree models are more involved and much slower than their seq2seq counterparts, both at training and at inference. For the sake of simplicity, we use seq2seq models, which were shown to be effective at generating trees, e.g. in the context of constituency parsing (Vinyals et al., 2015), where the task is to predict a syntactic parse tree of input sentences.

Using seq2seq models to generate trees requires to map trees to sequences. To this effect, we use the prefix notation (also known as normal Polish notation), writing each node before its children, listed from left to right. For instance, the arithmetic expression $2 + 3 * (5 + 2)$ is represented as the sequence $[+\ 2\ *\ 3\ +\ 5\ 2]$. In contrast to the common infix notation $2+3*(5+2)$, prefix sequences need no parentheses and are therefore shorter. Inside sequences, operators, functions or variables are represented by specific tokens, and integers by sequences of digits preceded by a sign. As in the case between expressions and trees, there exists a one-to-one mapping between trees and prefix sequences.

2.3 GENERATING RANDOM EXPRESSIONS

In this work, we want to generate a training set of mathematical expressions. However, sampling uniformly expressions with n internal nodes is not a simple task. Naive algorithms (such as recursive methods or techniques using fixed probabilities for nodes to be leaves, unary, or binary) tend to favour

deep trees over broad trees, or left-leaning over right leaning trees. Here are examples of different trees that we want to generate with the same probability.



In Section B of the appendix, we present an algorithm to generate random trees and expressions representative of the whole problem space, and where the four expression trees above all generated with the same probability. In the next section, we investigate the number of such possible expressions.

2.4 Counting expressions

Expressions are built from a list of operators, variables (i.e. literals) and integers sampled over a finite set. Based on the task of interest, more involved operators can be used, like differentiation or integration, etc. More precisely, we define our problem space as:

- trees with up to n internal nodes
- a set of p_1 unary operators (e.g. $\cos, \sin, \exp, \log$)
- a set of p_2 binary operators (e.g. $+, -, \times, \mathrm{pow}$)
- a set of L leaf values containing variables (e.g. x, y, z), constants (e.g. e, π), integers (e.g. $\{-10, \ldots, 10\}$)

If $p_1 = 0$, expressions are represented by binary trees. The number of binary trees with n internal nodes is given by the n-th Catalan numbers C_n (Sloane, 1996). A binary tree with n internal nodes has exactly $n + 1$ leaves. Each node and leaf can take respectively p_2 and L different values. As a result, the number of expressions with n binary operators can be expressed by:

$$E_n = C_n p_2^n L^{n+1} \approx \frac{4^n}{n\sqrt{\pi n}} (p_2)^n L^{n+1} \quad \text{with} \quad C_n = \frac{1}{n+1}\binom{2n}{n}$$

If $p_1 > 0$, the number of trees with n internal nodes is the n-th large Schroeder number S_n (Sloane, 1996). It can be computed by recurrence using the following equation:

$$(n + 1)S_n = 3(2n - 1)S_{n-1} - (n - 2)S_{n-2} \tag{1}$$

Finally, the number E_n of expressions with n internal nodes, p_1 unary operator, p_2 operators and L leaves is recursively computed as

$$(n + 1)E_n = (p_1 + 2Lp_2)(2n - 1)E_{n-1} - p_1(n - 2)E_{n-2} \tag{2}$$

For $p_1 = p_2 = L = 1$, Equation 2 boils down to Equation 1. With $p_2 = L = 1, p_1 = 0$, we have $(n + 1)E_n = 2(2n - 1)E_{n-1}$ which is the recurrence relation satisfied by Catalan numbers. The derivations and properties of all these formulas are provided in Section B of the appendix.

Having defined a syntax for mathematical problems and techniques to randomly generate expressions, we are now in a position to build the data sets our models will use.

3 Generating datasets

To train networks on mathematical subjects, we generate datasets of problems and their solutions. Ideally, we would like datasets to be representative of the problem space. Unfortunately, some

functions do not have an integral which can be expressed with usual functions (e.g. $f(x) = \exp(x^2)$ or $f(x) = \log(\log(x))$), and solutions to arbitrary differential equations cannot always be expressed with usual functions. To address this issue, we propose other approaches to generate large training sets for integration and first and second order differential equations.

3.1 INTEGRATION

For integration, we generate random functions with up to n operators, using methods from Section 2, and calculate their derivative with a computer algebra system. The derivative becomes the problem, and the original sample the solution. In that setting, we simply train the model to predict the generated function given its derivative.

3.2 FIRST ORDER DIFFERENTIAL EQUATION

We now present a method to generate first order differential equations with their solutions. We start from a bivariate function $F(x, y)$ such that the equation $F(x, y) = c$ (where c is a constant) can be analytically solved in y. In other words, there exists a bivariate function f that satisfies $\forall(x, c), F\big(x, f(x, c)\big) = c$. By differentiation with respect to x, we have that $\forall x, c$:

$$\frac{\partial F\big(x, f_c(x)\big)}{\partial x} + f_c'(x)\frac{\partial F\big(x, f_c(x)\big)}{\partial y} = 0$$

where $f_c = x \mapsto f(x, c)$. As a result, for any constant c, f_c is solution of the first order differential equation:

$$\frac{\partial F\big(x, y\big)}{\partial x} + y'\frac{\partial F\big(x, y\big)}{\partial y} = 0 \tag{3}$$

With this approach, we can use the method described in Section C of the appendix to generate arbitrary functions $F(x, y)$ analytically solvable in y, and create a dataset of differential equations with their solutions.

Alternatively, instead of generating a random function F, we propose to directly generate a function $f(x, c)$ solution of a differential equation that has to be determined. If $f(x, c)$ is solvable in c, we can simply compute the function F that satisfies $F\big(x, f(x, c)\big) = c$. By applying the approach above, we have that for any constant c, $x \mapsto f(x, c)$ is solution of the differential Equation 3. Finally, the resulting differential equation is factorized, and we remove all positive factors from the equation.

A necessary condition for this approach to work is the solvability in c of generated functions $f(x, c)$. For instance, the function $f(x, c) = c \times \log(x + c)$ cannot be analytically solved in c, i.e. the function F that satisfies $F\big(x, f(x, c)\big) = c$ cannot be written with usual functions. Since all the operators and functions we use are invertible, a simple condition to ensure the solvability in c is to guarantee that c only appears once in the leaves of the tree representation of $f(x, c)$. A straightforward way to generate a suitable $f(x, c)$ is to sample a random function $f(x)$ by the methods described in Section C of the appendix, and to replace one of the leaves in its tree representation by c. For instance:

Generate a random function	$f(x) = x\log(c \,/\, x)$
Solve in c	$c = xe^{\frac{f(x)}{x}} = F(x, f(x))$
Differentiate in x	$e^{\frac{f(x)}{x}}\big(1 + f'(x) - \frac{f(x)}{x})\big) = 0$
Simplify	$xy' - y + x = 0$

3.3 SECOND ORDER DIFFERENTIAL EQUATION

Our method for generating first order equations can be extended to the second order, by considering functions of three variables $f(x, c_1, c_2)$ that can be solved in c_2. As before, we derive a function of

three variables F such that $F\big(x, f(x, c_1, c_2), c_1\big) = c_2$. Differentiation with respect to x yields a first order differential equation:

$$\frac{\partial F(x, y, c_1)}{\partial x} + f'_{c_1,c_2}(x) \frac{\partial F\big(x, y, c_1\big)}{\partial y}\Bigg|_{y=f_{c_1,c_2}(x)} = 0$$

where $f_{c_1,c_2} = x \mapsto f(x, c_1, c_2)$. If this equation can be solved in c_1, we can infer another three-variable function G satisfying $\forall x, G\big(x, f_{c_1,c_2}(x), f'_{c_1,c_2}(x)\big) = c_1$. Differentiating with respect to x a second time yields the following equation:

$$\frac{\partial G(x, y, z)}{\partial x} + f'_{c_1,c_2}(x) \frac{\partial G(x, y, z)}{\partial y} + f''_{c_1,c_2}(x) \frac{\partial G(x, y, z)}{\partial z}\Bigg|_{\substack{y=f_{c_1,c_2}(x)\\ z=f'_{c_1,c_2}(x)}} = 0$$

Therefore, for any constants c_1 and c_2, f_{c_1,c_2} is solution of the second order differential equation:

$$\frac{\partial G(x, y, y')}{\partial x} + y' \frac{\partial G(x, y, y')}{\partial y} + y'' \frac{\partial G(x, y, y')}{\partial z} = 0$$

Using this approach, we can create pairs of second order differential equations and solutions, provided we can generate $f(x, c_1, c_2)$ is solvable in c_2, and that the corresponding first order differential equation is solvable in c_1. To ensure the solvability in c_2, we can use the same approach as for first order differential equation, e.g. we create f_{c_1,c_2} so that c_2 has exactly one leaf in its tree representation. For c_1, we employ a simple approach where we simply skip the current equation if we cannot solve it in c_1. Although naive, we found that the differentiation equation can be solved in c_1 about 50% the time. Another strategy to ensure the solvability in c_1 is also proposed in Section D of the appendix. As an example:

Generate a random function	$f(x) = c_1 e^x + c_2 e^{-x}$
Solve in c_2	$c_2 = f(x)e^x - c_1 e^{2x} = F(x, f(x), c_1)$
Differentiate in x	$e^x\big(f'(x) + f(x)\big) - 2c_1 e^{2x} = 0$
Solve in c_1	$c_1 = \frac{1}{2} e^{-x}\big(f'(x) + f(x)\big) = G(x, f(x), f'(x))$
Differentiate in x	$\frac{1}{2} e^{-x}\big(f''(x) - f(x)\big)$
Simplify	$y'' - y = 0$

3.4 Dataset cleaning

Equation simplification In practice, we simplify generated expressions to reduce the number of unique possible equations in the training set, and to reduce the length of sequences. Also, we do not want to train our model to predict $x + 1 + 1 + 1 + 1 + 1$ when it can simply predict $x + 5$. As a result, sequences $[+\ 2\ +\ x\ 3]$ and $[+\ 3\ +\ 2\ x]$ will both be simplified to $[+\ x\ 5]$ as they both represent the expression $x + 5$. Similarly, expression $\log(e^{x+3})$ will be simplified to $x + 3$, $\cos x^2 + \sin x^2$ to 1, etc. On the other hand, $\sqrt{(x-1)^2}$ will not be simplified to $x - 1$ as we do not make any assumption on the sign of $x - 1$.

Coefficients simplification In the case of first order differential equations, we modify generated expressions by equivalent expressions up to a change of variable. For instance, $x + x\tan(3) + cx + 1$ will be simplified to $cx + 1$, as a particular choice of the constant c makes these two expressions identical. Similarly, $\log(x^2) + c\log(x)$ becomes $c\log(x)$.

We apply a similar technique for second order differential equations, although simplification is sometimes a bit more involved because there are two constants c_1 and c_2. For instance, $c_1 - c_2 x/5 + c_2 + 1$ is simplified to $c_1 x + c_2$, while $c_2 e^{c_1} e^{c_1 x e^{-1}}$ can be expressed with $c_2 e^{c_1 x}$, etc.

We also perform transformations that are not strictly equivalent, as long as they hold under specific assumptions. For instance, we simplify $\tan(\sqrt{c_2}x) + \cosh(c_1 + 1) + 4$ to $c_1 + \tan(c_2x)$, although the constant term can be negative in the second expression, but not the first one. Similarly $e^3 e^{c_1 x} e^{c_1 \log(c_2)}$ is transformed to $c_2 e^{c_1 x}$.

Invalid expressions Finally, we also remove invalid expressions from our dataset. For instance, expressions like $\log(0)$ or $\sqrt{-2}$. To detect them, we evaluate in an expression tree the values of subtrees that do not depend on x. If a subtree is not evaluated to a finite real number (e.g. $-\infty$, $+\infty$ or a complex number), we discard the expression.

4 EXPERIMENTS

4.1 DATASET

For all considered tasks, we generate datasets using the method presented in Section 3, with:

- expressions with up to $n = 15$ internal nodes
- $L = 12$ leaf values in $\{x\} \cup \{-5, \dots, 5\}$
- $p_2 = 4$ binary operators: $+, -, \times, /$
- $p_1 = 15$ unary operators: exp, log, sqrt, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, $\sinh^{-1}$, $\cosh^{-1}$, $\tanh^{-1}$

To produce meaningful mathematical expressions, when generating expressions, binary operators are sampled with higher probabilities than unary functions. Additions and multiplications represent on average half the nodes in our expressions.

4.2 MODEL

For all our experiments, we train a seq2seq model to predict the solutions of given equations, i.e. to predict a primitive given a function, or predict a solution given a differential equation. We use a transformer model (Vaswani et al., 2017) with 8 attention heads, 6 layers, and a dimensionality of 512. In our experiences, using larger models did not improve the performance. We train our models with the Adam optimizer (Kingma & Ba, 2014), with a learning rate of 10^{-4}. We remove expressions with more than 512 tokens, and train our model with 256 equations per batch.

At inference, expressions are generated by a beam search (Koehn, 2004), with early stopping. We normalize the log-likelihood scores of hypotheses in the beam by their sequence length. We report results with beam widths of 1 (i.e. greedy decoding), 10 and 50.

During decoding, nothing prevents the model from generating an invalid prefix expression, e.g. $[+\ 2\ *\ 3]$. To address this issue, Dyer et al. (2016) use constraints during decoding, to ensure that generated sequences can always be converted to valid expression trees. In our case, we found that model generations are almost always valid and we do not use any constraint. When an invalid expression is generated, we simply consider it as an incorrect solution and ignore it.

4.3 EVALUATION

At the end of each epoch, we evaluate the ability of the model to predict the solutions of given equations. In machine translation, hypotheses given by the model are compare to references written by human translators, typically with metrics like the BLEU score (Papineni et al., 2002) that measure the overlap between hypotheses and references. Evaluating the quality of translations is a very difficult problem, and many studies showed that a better BLEU score does not necessarily correlate with a better performance according to human evaluation. However, unlike in machine translation, we can easily verify the correctness of our model by simply comparing generated expressions to their reference solutions.

For instance, for the given differential equation $xy' - y + x$ with a reference solution $x\log(c\ /\ x)$ (where c is a constant), our model may generate $x\log(c) - x\log(x)$. We can check that these

two solutions are equal, although they are written differently, using a symbolic framework like SymPy (Meurer et al., 2017).

However, our model may also generate $xc - x\log(x)$ which is also a valid solution, that is actually equivalent to the previous one for a different choice of constant c. In that case, we replace y in the differential equation by the model hypothesis. If $xy' - y + x = 0$, we conclude that the hypothesis is a valid solution. In the case of integral computation, we can simply differentiate the model hypothesis, and compare it with the function to integrate. For the three problems, we measure the accuracy of our model on equations from the test set.

4.4 RESULTS

In Table 1, we report the accuracy of our models on the three different tasks, on a holdout test set composed of 5000 equations. We observe that the model is extremely efficient at function integration, with an accuracy of almost 100%, even with a beam size of 1. Greedy decoding does not work as well on differential equations. In particular, we observe an improvement in accuracy of almost 40% when using a large beam size of 50 for second order differential equations. Unlike in machine translation, where increasing the beam size does not necessarily increase the performance (Ott et al., 2018), in our case, we always observe significant improvements with wider beams. Typically, using a beam size of 50 provides an improvement of 8% accuracy over a beam size of 10. This makes sense, as increasing the beam size will provide more hypotheses, although a wider beam may displace a valid hypothesis to consider invalid ones with better log-probabilities.

	Integration	Differential equation (order 1)	Differential equation (order 2)
Beam size 1	98.8	77.6	43.0
Beam size 10	99.7	90.5	73.0
Beam size 50	99.7	94.0	81.2

Table 1: **Accuracy of our model on the three different tasks.** Results are reported on a holdout test set of 5000 equations. Using beam search decoding significantly improves the accuracy of the model.

4.5 COMPARISON WITH MATHEMATICAL FRAMEWORKS

We compare our model with two popular mathematical frameworks: Mathematica (Wolfram-Research, 2019) and Matlab (MathWorks, 2019).[1] Prefix sequences in our test set are converted back to their infix representations, and given as input to Mathematica. For an input equation, Mathematica either returns a solution, rewrites the equation in a different way (meaning it was not able to solve it) or times out after a preset delay. When Mathematica times out, we conclude that it is not able to compute a solution (although it might have found a solution given more time).

In Table 2, we present accuracy for our model with different beam sizes, and for Mathematica with timeout delays of 10 and 30 seconds. Because Mathematica times out on a non-negligible number of equations, the evaluation would take too long to run on our 5000 test equations, and we only evaluate on a smaller test subset of 500 equations, on which we also re-evaluate our model.

On all tasks, we observe that our model significantly outperforms Mathematica. On function integration, our model obtains close to 100% accuracy, while Mathematica does not reach 80%. On first order differential equations, Mathematica is on par with our model when it uses a beam size of 1, i.e. with greedy decoding. However, using a beam search of size 50 our model accuracy goes from 81.2% to 97.0%, largely surpassing Mathematica. Similar observations can be made for second order differential equations, where beam search is even more critical since the number of equivalent solutions is larger. For Mathematica, the performance naturally increases with the timeout duration. For each task, using a timeout of 30 seconds improves the test accuracy by about 4% over a timeout of 10 seconds. Our model, on the other hand, typically finds a solution in less than a second, even with a large beam size. On average, Matlab has a slightly lower performance than Mathematica on the problems we tested.

[1] All experiments were run with Mathematica 12.0.0.0, and Matlab R2019a.

	Integration	Differential equation (order 1)	Differential equation (order 2)
Mathematica (10s)	76.4	73.2	57.0
Mathematica (30s)	80.2	77.2	61.6
Matlab	65.2	-	-
Beam size 1	98.4	81.2	40.8
Beam size 10	99.6	94.0	73.2
Beam size 50	99.6	97.0	81.0

Table 2: **Comparison of our model with Mathematica and Matlab on a test set of 500 equations.** For Mathematica we report results by setting a timeout of 10 or 30 seconds per equation. On a given equation, our model typically finds the solution in less than a second.

Equation	Solution
$y' = \frac{16x^3 - 42x^2 + 2x}{(-16x^8 + 112x^7 - 204x^6 + 28x^5 - x^4 + 1)^{1/2}}$	$y = \sin^{-1}(4x^4 - 14x^3 + x^2)$
$3xy\cos(x) - \sqrt{9x^2\sin(x)^2 + 1}y' + 3y\sin(x) = 0$	$y = c\exp\left(\sinh^{-1}(3x\sin(x))\right)$
$4x^4yy'' - 8x^4y'^2 - 8x^3yy' - 3x^3y'' - 8x^2y^2 - 6x^2y' - 3x^2y'' - 9xy' - 3y = 0$	$y = \frac{c_1 + 3x + 3\log(x)}{x(c_2 + 4x)}$

Table 3: Examples of problems that our model is able to solve, on which Mathematica and Matlab were not able to find a solution. For each equation, our model finds a valid solution with greedy decoding.

Table 3 shows examples of functions that our model was able to solve, on which Mathematica and Matlab did not find a solution. The denominator of the function to integrate, $-16x^8 + 112x^7 - 204x^6 + 28x^5 - x^4 + 1$, can actually be rewritten as $1 - (4x^4 - 14x^3 + x^2)^2$. With the simplified input:

$$\frac{16x^3 - 42x^2 + 2x}{1 - (4x^4 - 14x^3 + x^2)^2)^{1/2}}$$

integration becomes easy and Mathematica is able to find the solution, but not with the expanded denominator.

When comparing with Matlab and Mathematica, we work from a data set generated for our model and use their standard differential equation solvers. Different sets of equations, and advanced techniques for solving them (e.g. transforming them before introducing them in the solver) would probably result in smaller performance gaps.

4.6 Equivalent solutions

An interesting property of our model is that it is able to generate solutions that are exactly equivalent, but written in different ways. For instance, we consider the following first order differential equation, along with one of its solutions:

$$162x\log(x)y' + 2y^3\log(x)^2 - 81y\log(x) + 81y = 0 \qquad y = \frac{9\sqrt{x}\sqrt{\frac{1}{\log(x)}}}{\sqrt{c + 2x}}$$

In Table 4, we report the top 10 hypotheses returned by our model for this equation. We observe that all generations are actually valid solutions, although they are expressed very differently. They are however not all equal: merging the square roots within the first and third equations would give the same expression except that the third one would contain a factor 2 in front of the constant c, but up to a change of variable, these two solutions are actually equivalent.

The ability of the model to recover equivalent expressions, without having been trained to do so, is very intriguing. This suggest that some deeper understanding of mathematics has been achieved by the model.

Hypothesis	Score	Hypothesis	Score
$\frac{9\sqrt{x}\sqrt{\frac{1}{\log(x)}}}{\sqrt{c+2x}}$	-0.04660	$\frac{9}{\sqrt{\frac{c\log(x)}{x}+2\log(x)}}$	-0.12374
$\frac{9\sqrt{x}}{\sqrt{c+2x}\sqrt{\log(x)}}$	-0.05557	$\frac{9\sqrt{x}}{\sqrt{c\log(x)+2x\log(x)}}$	-0.13949
$\frac{9\sqrt{2}\sqrt{x}\sqrt{\frac{1}{\log(x)}}}{2\sqrt{c+x}}$	-0.11532	$\frac{9}{\sqrt{\frac{c}{x}+2}\sqrt{\log(x)}}$	-0.14374
$9\sqrt{x}\sqrt{\frac{1}{c\log(x)+2x\log(x)}}$	-0.11740	$9\sqrt{\frac{1}{\frac{c\log(x)}{x}+2\log(x)}}$	-0.20465
$\frac{9\sqrt{2}\sqrt{x}}{2\sqrt{c+x}\sqrt{\log(x)}}$	-0.12373	$9\sqrt{x}\sqrt{\frac{1}{c\log(x)+2x\log(x)+\log(x)}}$	-0.23228

Table 4: Top 10 generations of our model for the first order differential equation $162x\log(x)y'+2y^3\log(x)^2-81y\log(x)+81y=0$, generated with a beam search. All hypotheses are valid solutions, and are equivalent up to a change of the variable c. Scores are log-probabilities normalized by sequence lengths.

5 Related work

Computers were used for symbolic mathematics since the late 1960s (Moses, 1974). Computer algebra systems (CAS), such as Matlab, Mathematica, Maple, PARI and SAGE, are used for a variety of mathematical tasks (Gathen & Gerhard, 2013). Modern methods for symbolic integration are based on Risch algorithm (Risch, 1970). Implementations can be found in Bronstein (2005) and Geddes et al. (1992). However, the complete description of the Risch takes more than 100 pages, and is not fully implemented in any mathematical framework.

Deep learning networks have been used to simplify treelike expressions. Zaremba et al. (2014) use recursive neural networks to simplify complex symbolic expressions. They use tree representations for expressions, but provide the model with problem related information: possible rules for simplification. The neural network is trained to select the best rule. Allamanis et al. (2017) propose a framework called *neural equivalence networks* to learn semantic representations of algebraic expressions. Typically, a model is trained to map different but equivalent expressions (like the 10 expressions proposed in Table 4) to the same representation. However, they only consider Boolean and polynomial expressions.

Most attempts to use deep networks for mathematics have focused on arithmetic over integers (sometimes over polynomials with integer coefficients). For instance, Kaiser & Sutskever (2015) proposed the Neural-GPU architecture, and train networks to perform additions and multiplications of numbers given in their binary representations. They show that a model trained on numbers with up-to 20 bits can be applied to much larger numbers at test time, while preserving a perfect accuracy. Freivalds & Liepins (2017) proposed an improved version of the Neural-GPU by using hard non-linear activation functions, and a diagonal gating mechanism. Saxton et al. (2019) use LSTMs (Hochreiter & Schmidhuber, 1997) and transformers on a wide range of problems, from arithmetic to simplification of formal expressions. However, they only consider polynomial functions, and the task of differentiation, which is significantly easier than integration. Trask et al. (2018) propose the Neural arithmetic logic units, a new module designed to learn systematic numerical computation, and that can be used within any neural network. Like Kaiser & Sutskever (2015), they show that at inference their model can extrapolate on numbers orders of magnitude larger than the ones seen during training.

6 Conclusion

In this paper, we show that standard seq2seq models can be applied to difficult tasks like function integration, or differential equation. Because there was no existing dataset to train such models, we propose an approach to generate arbitrarily large datasets of equations, with their associated solutions. We show that a simple transformer model trained on these datasets can perform extremely well both at computing function integrals, and solving differential equations, outperforming state-of-the-art mathematical frameworks like Matlab or Mathematica that rely on a large number of algorithms and heuristics, and a complex implementation (Risch, 1970). Results also show that the model is able to write identical expressions in very different ways.

These results are surprising given the incapacity of neural models to perform simpler tasks like integer addition or multiplication. However, proposed hypotheses are sometimes incorrect, and considering multiple beam hypotheses is often necessary to obtain a valid solution. The validity of a solution itself is not provided by the model, but by an external symbolic framework (Meurer et al., 2017). These results suggest that in the future, standard mathematical frameworks may benefit from integrating neural components in their solvers.

References

Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. Learning continuous semantic representations of symbolic expressions. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 80–88. JMLR.org, 2017.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2015.

M. Bronstein. *Symbolic Integration I: Transcendental Functions*. Algorithms and combinatorics. Springer, 2005. ISBN 978-3-540-21493-9.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 199–209, 2016.

Akiko Eriguchi, Yoshimasa Tsuruoka, and Kyunghyun Cho. Learning to parse and translate improves neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 72–78, 2017.

Philippe Flajolet and Andrew M. Odlyzko. Singularity analysis of generating functions. *SIAM J. Discrete Math.*, 3(2):216–240, 1990.

Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 edition, 2009. ISBN 0521898064, 9780521898065.

Karlis Freivalds and Renars Liepins. Improving the neural gpu architecture for algorithm learning. *ArXiv*, abs/1702.08727, 2017.

Joachim von zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 3rd edition, 2013. ISBN 1107039037, 9781107039032.

Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Norwell, MA, USA, 1992. ISBN 0-7923-9259-0.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *CoRR*, abs/1511.08228, 2015.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. ISBN 0-201-89683-4.

Philipp Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*, pp. 115–124. Springer, 2004.

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

MathWorks. Matlab optimization toolbox (r2019a), 2019. The MathWorks, Natick, MA, USA.

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`.

Joel Moses. Macsyma - the fifth year. *SIGSAM Bull.*, 8(3):105–110, August 1974. ISSN 0163-5824.

Myle Ott, Michael Auli, David Grangier, et al. Analyzing uncertainty in neural machine translation. In *International Conference on Machine Learning*, pp. 3953–3962, 2018.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pp. 311–318. Association for Computational Linguistics, 2002.

Robert H. Risch. The solution of the problem of integration in finite terms. *Bull. Amer. Math. Soc.*, 76(3):605–608, 05 1970.

David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group (eds.). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*, 2019.

N. J. A. Sloane. The encyclopedia of integer sequences, 1996.

Richard P. Stanley. *Enumerative Combinatorics: Volume 1*. Cambridge University Press, New York, NY, USA, 2nd edition, 2011. ISBN 1107602629, 9781107602625.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1556–1566, 2015.

Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems*, pp. 8035–8044, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 6000–6010, 2017.

Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Advances in neural information processing systems*, pp. 2773–2781, 2015.

H.S. Wilf. *generatingfunctionology: Third Edition*. CRC Press, 2005. ISBN 978-1-4398-6439-5. URL `https://www.math.upenn.edu/ wilf/gfologyLinked2.pdf`.

Wolfram-Research. Mathematica, version 12.0, 2019. Champaign, IL, 2019.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

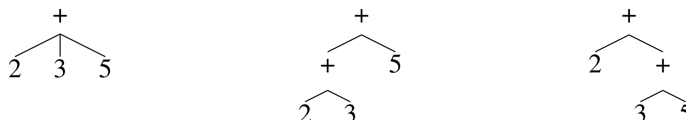
Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pp. 1278–1286, Cambridge, MA, USA, 2014. MIT Press.

A A SYNTAX FOR MATHEMATICAL EXPRESSIONS : SOME FINE POINTS

We represent mathematical expressions as trees with operators as internal nodes, and numbers, constants or variables, as leaves. By enumerating nodes in prefix order, we transform trees into sequences suitable for NLP models.

For this representation to be efficient, we want expressions, trees and sequences to be in a one to one correspondence. It is plain that different expressions will result in different trees and sequences. For the reverse to hold, we need to take care of a few special cases.

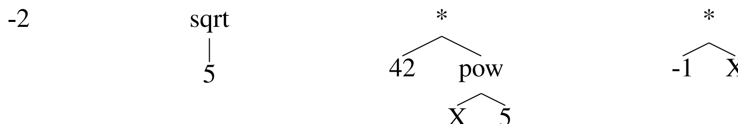
First, expressions like sums and products may correspond to several trees. For instance, expression $2 + 3 + 5$ can be represented as any one of those trees:



We will assume that all operators have at most two operands, and that, in case of doubt, they are associative to the right. $2 + 3 + 5$ would then correspond to the rightmost tree.

Second, the distinction between internal nodes (operators) and leaves (mathematical primitive objects) is somewhat arbitrary. Should number -2 be the basic object -2 or a unary minus operator, applied to number 2? What about $\sqrt{5}$, $42X^5$, or function $\log_{10}$? To make things simple , we only consider numbers, constants and variables as possible leaves, and avoid using a unary minus. In particular, expressions like $-x$ are represented as $-1 * x$.

Here are the trees for -2, $\sqrt{5}$, $42X^5$ and $-x$



Integers are represented in positional notation, as a sign followed by a sequence of digits (from 0 to 9 in base 10). 2354 and -34 would then be represented as $+2\ 3\ 5\ 4$ and $-\ 3\ 4$. For zero a unique representation is chosen ($+0$ or -0).

B MATHEMATICAL DERIVATIONS OF PROBLEM SPACE SIZE

All derivations make use of generating functions (Flajolet & Sedgewick, 2009) and (Wilf, 2005).

We deal first with the simpler binary case ($p_1 = 0$), then proceed to unary-binary trees and expressions. In each case, we calculate a generating function, from which we derive a closed formula or a recurrence to calculate the sequence, and an asymptotic expansion.

B.1 BINARY TREES AND EXPRESSIONS

The main part of this derivation follows (Knuth, 1997) (pages 388-389).

B.1.1 FORMULA AND GENERATING FUNCTION

Let b_n be the number of binary trees with n internal nodes. We have $b_0 = 1$ and $b_1 = 1$.

Any binary tree with n internal nodes can be generated by concatenating a left and a right subtree with k and $n - 1 - k$ internal nodes respectively. Summing over all possible values of k,

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + ... + b_{n-2} b_1 + b_{n-1} b_0$$

Let $B(z)$ be the generating function of b_n, $B(z) = b_0 + b_1 z + b_2 z^2 + b_3 z^3 + ...$

$$\begin{aligned}
B(z)^2 &= {b_0}^2 + (b_0 b_1 + b_1 b_0)z + (b_0 b_2 + b_1 b_1 + b2 b_0)z^2 + ... \\
&= b_1 + b_2 z + b_3 z^2 + ... \\
&= \frac{B(z) - b_0}{z} \\
zB(z)^2 - B(z) + 1 &= 0
\end{aligned}$$

Solving for $B(z)$

$$B(z) = \frac{1 \pm \sqrt{1-4z}}{2z}$$

and since $B(0) = b_0 = 1$, we derived the generating function for sequence b_n

$$B(z) = \frac{1 - \sqrt{1-4z}}{2z}$$

Let us derive a closed formula for b_n. By the binomial theorem

$$\begin{aligned}
B(z) &= \frac{1}{2z}\left(1 - \sum_{k=0}^{\infty}\binom{1/2}{k}(-4z)^k\right) \\
&= \frac{1}{2z}\left(1 + \sum_{k=0}^{\infty}\frac{1}{2k-1}\binom{2k}{k}z^k\right) \\
&= \frac{1}{2z}\sum_{k=1}^{\infty}\frac{1}{2k-1}\binom{2k}{k}z^k \\
&= \sum_{k=1}^{\infty}\frac{1}{2(2k-1)}\binom{2k}{k}z^{k-1} \\
&= \sum_{k=0}^{\infty}\frac{1}{2(2k+1)}\binom{2k+2}{k+1}z^k \\
&= \sum_{k=0}^{\infty}\frac{1}{k+1}\binom{2k}{k}z^k
\end{aligned}$$

Therefore

$$b_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

These are the Catalan numbers.

For expressions, we note that binary trees with n internal nodes have $n+1$ leaves, that each node has one of p_2 operators, and each leaf one of L leaves. For a tree with n nodes, this means $p_2^n L^{n+1}$ possible combinations of operators and leaves. And so, the number of binary expressions with n operators is

$$E_n = \frac{(2n)!}{(n+1)!n!}p_2^n L^{n+1}$$

B.1.2 Asymptotics

To derive an asymptotic approximation of b_n, we apply Stirling formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

to binomials

$$\binom{2n}{n} \approx \frac{4^n}{\sqrt{\pi n}}$$

$$b_n \approx \frac{4^n}{n\sqrt{\pi n}}$$

And since binary trees with n internal nodes have $n+1$ leaves, we have the following formulas for the number of expressions with n internal nodes:

$$E_n \approx \frac{1}{n\sqrt{\pi n}}(4p_2)^n L^{n+1}$$

B.2 Unary-binary trees

B.2.1 Generating function

Let s_n be the number of unary-binary trees (i.e. trees where internal nodes can have one or two children) with n internal nodes. We have $s_0 = 1$ and $s_1 = 2$ (the only internal node is either unary or binary).

Any tree with n internal nodes is obtained either by adding a unary internal node at the root of a tree with $n-1$ internal nodes, or by concatenating with a binary operator a left and a right subtree with k and $n-1-k$ internal nodes respectively.

Summing up as before, we have

$$s_n = s_{n-1} + s_0 s_{n-1} + s_1 s_{n-2} + ... + s_{n-1} s_0$$

Let $S(z)$ be the generating function of the s_n, the above formula translates into

$$S(z)^2 = \frac{S(z) - s_0}{z} - S(z)$$
$$zS(z)^2 + (z-1)S(z) + 1 = 0$$

solving and taking into account the fact that $S(0) = 1$, we obtain the generating function of the s_n

$$S(z) = \frac{1 - z - \sqrt{1 - 6z + z^2}}{2z}$$

The numbers s_n generated by $S(z)$ are known as the Schroeder numbers (OEIS A006318) (Sloane, 1996). They appear in different combinatorial problems (Stanley, 2011). Notably they correspond to the number of paths from $(0, 0)$ to (n, n) on an integer $(n \times n)$ grid, moving north, east or northeast, and never rising above the diagonal.

B.2.2 Calculation

Schroeder numbers do not have a simple closed formula, but a recurrence allowing for their calculation can be derived from their generating function.

Rewriting $S(z)$ as

$$2zS(z) + z - 1 = -\sqrt{1 - 6z + z^2}$$

and differentiating, we have

$$2zS'(z) + 2S(z) + 1 = \frac{3 - z}{\sqrt{1 - 6z + z^2}} = \frac{3 - z}{1 - 6z + z^2}(1 - z - 2zS(z))$$
$$2zS'(z) + 2S(z)\left(1 + \frac{3z - z^2}{1 - 6z + z^2}\right) = \frac{(3 - z)(1 - z)}{1 - 6z + z^2} - 1$$
$$2zS'(z) + 2S(z)\frac{1 - 3z}{1 - 6z + z^2} = \frac{2 + 2z}{1 - 6z + z^2}$$
$$z(1 - 6z + z^2)S'(z) + (1 - 3z)S(z) = 1 + z$$

Replacing $S(z)$ and $S'(z)$ with their n-th coefficient yields, for $n > 1$

$$ns_n - 6(n-1)s_{n-1} + (n-2)s_{n-2} + s_n - 3s_{n-1} = 0$$
$$(n+1)s_n = 3(2n-1)s_{n-1} - (n-2)s_{n-2}$$

Together with $s_0 = 1$ and $s_1 = 2$, this allows for fast ($O(n)$) calculation of Schroeder numbers.

B.2.3 ASYMPTOTICS

To derive an asymptotic formula of s_n, we develop the generating function around its smallest singularity (Flajolet & Odlyzko, 1990), i.e. the radius of convergence of the power series.

Since
$$1 - 6z - z^2 = (1 - (3 - \sqrt{8})z)(1 - (3 + \sqrt{8})z)$$
The smallest singular value is
$$r_1 = \frac{1}{(3 + \sqrt{8})}$$
and the asymptotic formula will have the exponential term
$$r_1^{-n} = (3 + \sqrt{8})^n = (1 + \sqrt{2})^{2n}$$
In a neighborhood of r_1, the generating function can be rewritten as
$$S(z) \approx (1 + \sqrt{2}) \left(1 - 2^{1/4}\sqrt{1 - (3 + \sqrt{8})z}\right) + O(1 - (3 + \sqrt{8})z)^{3/2}$$
since
$$[z_n]\sqrt{1 - az} \approx -\frac{a^n}{\sqrt{4\pi n^3}}$$
denoting by $[z_n]F(z)$ the n-th coefficient in the formal series of F
$$s_n \approx \frac{(1 + \sqrt{2})(3 + \sqrt{8})^n}{2^{3/4}\sqrt{\pi n^3}} = \frac{(1 + \sqrt{2})^{2n+1}}{2^{3/4}\sqrt{\pi n^3}}$$
Comparing with the number of binary trees, we have
$$s_n \approx 1.44(1.46)^n b_n$$

B.3 UNARY-BINARY EXPRESSIONS

In the binary case, the number of expressions can be derived from the number of trees. This cannot de done in the unary-binary case, because the number of leaves in a tree with n internal nodes depends on the number of binary operators only ($n_2 + 1$).

B.3.1 GENERATING FUNCTION

The number of trees with n internal nodes and n_2 binary operators can be derived from the following observation. Any unary-binary tree with n_2 binary internal nodes can be generated from a binary tree by adding unary internal nodes. Each node in the binary tree can receive one or several unary parents.

Since the binary tree has $2n_2 + 1$ nodes and the number of unary internal nodes to be added is $n - n_2$, the number of unary-binary trees that can be created from a specific binary tree is the number of multisets with $2n_2 + 1$ elements on $n - n_2$ symbols, that is
$$\binom{n + n_2}{n - n_2} = \binom{n + n_2}{2n_2}$$

If b_q denotes the q-th Catalan number, the number of trees with n_2 binary operators among n is
$$\binom{n + n_2}{2n_2} b_{n_2}$$

Since such trees have $n_2 + 1$ leaves, with L leaves, p_2 binary and p_1 unary operators to choose from, the number of expressions is
$$E(n, n_2) = \binom{n + n_2}{2n_2} b_{n_2} p_2^{n_2} p_1^{n-n_2} L^{n_2+1}$$

Summing over all values of n_2 (from 0 to n) yields the number of different expressions

$$E_n = \sum_{n_2=0}^{n} \binom{n+n_2}{2n_2} b_{n_2} p_2^{n_2} p_1^{n-n_2} L^{n_2+1} z^n$$

Let $E(z)$ be the corresponding generating function.

$$\begin{aligned} E(z) &= \sum_{n=0}^{\infty} E_n z^n \\ &= \sum_{n=0}^{\infty} \sum_{n_2=0}^{n} \binom{n+n_2}{2n_2} b_{n_2} p_2^{n_2} p_1^{n-n_2} L^{n_2+1} z^n \\ &= L \sum_{n=0}^{\infty} \sum_{n_2=0}^{n} \binom{n+n_2}{2n_2} b_{n_2} \left(\frac{Lp_2}{p_1}\right)^{n_2} p_1^n z^n \\ &= L \sum_{n=0}^{\infty} \sum_{n_2=0}^{\infty} \binom{n+n_2}{2n_2} b_{n_2} \left(\frac{Lp_2}{p_1}\right)^{n_2} (p_1 z)^n \end{aligned}$$

since $\binom{n+n_2}{2n_2} = 0$ when $n > n_2$

$$\begin{aligned} E(z) &= L \sum_{n_2=0}^{\infty} b_{n_2} \left(\frac{Lp_2}{p_1}\right)^{n_2} \sum_{n=0}^{\infty} \binom{n+n_2}{2n_2} (p_1 z)^n \\ &= L \sum_{n_2=0}^{\infty} b_{n_2} \left(\frac{Lp_2}{p_1}\right)^{n_2} \sum_{n=0}^{\infty} \binom{n+2n_2}{2n_2} (p_1 z)^{n+n_2} \\ &= L \sum_{n_2=0}^{\infty} b_{n_2} (Lp_2 z)^{n_2} \sum_{n=0}^{\infty} \binom{n+2n_2}{2n_2} (p_1 z)^n \end{aligned}$$

applying the binomial formula

$$\begin{aligned} E(z) &= L \sum_{n_2=0}^{\infty} b_{n_2} (Lp_2 z)^{n_2} \frac{1}{(1-p_1 z)^{2n_2+1}} \\ &= \frac{L}{1-p_1 z} \sum_{n_2=0}^{\infty} b_{n_2} \left(\frac{Lp_2 z}{(1-p_1 z)^2}\right)^{n_2} \end{aligned}$$

applying the generating function for binary trees

$$\begin{aligned} E(z) &= \frac{L}{1-p_1 z} \left(\frac{1 - \sqrt{1 - 4\frac{Lp_2 z}{(1-p_1 z)^2}}}{2\frac{Lp_2 z}{(1-p_1 z)^2}} \right) \\ &= \frac{1-p_1 z}{2p_2 z} \left(1 - \sqrt{1 - 4\frac{Lp_2 z}{(1-p_1 z)^2}} \right) \\ &= \frac{1 - p_1 z - \sqrt{(1-p_1 z)^2 - 4Lp_2 z}}{2p_2 z} \end{aligned}$$

Reducing, we have

$$E(z) = \frac{1 - p_1 z - \sqrt{1 - 2(p_1 + 2Lp_2 k)z + p_1 z^2}}{2p_2 z}$$

B.3.2 CALCULATION

As before, there is no closed simple formula for E_n, but we can derive a recurrence formula by differentiating the generating function, rewritten as

$$2p_2zE(z) + p_1z - 1 = -\sqrt{1 - 2(p_1 + 2p_2L)z + p_1z^2}$$

$$2p_2zE'(z) + 2p_2E(z) + p_1 = \frac{p_1 + 2p_2L - p_1z}{\sqrt{1 - 2(p_1 + 2p_2L)z + p_1z^2}}$$

$$2p_2zE'(z) + 2p_2E(z) + p_1 = \frac{(p_1 + 2p_2L - p_1z)(1 - p_1z - 2p_2zE(z))}{1 - 2(p_1 + 2p_2L)z + p_1z^2}$$

$$2p_2zE'(z) + 2p_2E(z)\left(1 + \frac{z(p_1 + 2p_2L - p_1z)}{1 - 2(p_1 + 2p_2L)z + p_1z^2}\right) = \frac{(p_1 + 2p_2L - p_1z)(1 - p_1z)}{1 - 2(p_1 + 2p_2L)z + p_1z^2} - p_1$$

$$2p_2zE'(z) + 2p_2E(z)\left(\frac{1 - (p_1 + 2p_2L)z}{1 - 2(p_1 + 2p_2L)z + p_1z^2}\right) = \frac{2p_2L(1 + p_1z) + p_1(p_1 - 1)z}{1 - 2(p_1 + 2p_2L)z + p_1z^2}$$

$$2p_2zE'(z)(1 - 2(p_1 + 2p_2L)z + p_1z^2) + 2p_2E(z)(1 - (p_1 + 2p_2L)z) = (2p_2L(1 + p_1z) + p_1(p_1 - 1)z)$$

replacing $E(z)$ and $E'(z)$ with their coefficients

$$2p_2(nE_n - 2(p_1 + 2p_2L)(n-1)E_{n-1} + p_1(n-2)E(n-2)) + 2p_2(E_n - (p_1 + 2p_2L)E_{n-1}) = 0$$

$$(n+1)E_n - (p_1 + 2p_2L)(2n-1)E_{n-1} + p_1(n-2)E_{n-2} = 0$$

$$(n+1)E_n = (p_1 + 2p_2L)(2n-1)E_{n-1} - p_1(n-2)E_{n-2}$$

which together with

$$E(0) = L$$
$$E(1) = (p_1 + p_2L)L$$

provides a formula for calculating E_n.

B.3.3 ASYMPTOTICS

As before, approximations of E_n for large n can be found by developing $E(z)$ in the neighbourhood of the root with the smallest module of

$$1 - 2(p_1 + 2p_2L)z + p_1z^2$$

The roots are

$$r_1 = \frac{p_1}{p_1 + 2p_2L - \sqrt{p_1^2 + 4p_2^2L^2 + 4p_2p_1L - p_1}}$$
$$r_2 = \frac{p_1}{p_1 + 2p_2L + \sqrt{p_1^2 + 4p_2^2L^2 + 4p_2p_1L - p_1}}$$

both are positive and the smallest one is r_2

To alleviate notation, let

$$\delta = \sqrt{p_1^2 + 4p_2^2L^2 + 4p_2p_1L - p_1}$$
$$r_2 = \frac{p_1}{p_1 + 2p_2L + \delta}$$

developping $E(z)$ near r_2,

$$E(z) \approx \frac{1 - p_1r_2 - \sqrt{1 - r_2(\frac{p_1 + 2p_2L - \delta}{p_1})}\sqrt{1 - \frac{z}{r_2}}}{2p_2r_2} + O(1 - \frac{z}{r_2})^{3/2}$$

$$E(z) \approx \frac{p_1 + 2p_2L + \delta - p_1^2 - \sqrt{p_1 + 2p_2L + \delta}\sqrt{2\delta}\sqrt{1 - \frac{z}{r_2}}}{2p_2p_1} + O(1 - \frac{z}{r_2})^{3/2}$$

and therefore

$$E_n \approx \frac{\sqrt{\delta}r_2^{-n-\frac{1}{2}}}{2p_2\sqrt{2\pi p_1n^3}} = \frac{\sqrt{\delta}}{2p_2\sqrt{2\pi n^3}}\frac{(p_1 + 2p_2L + \delta)^{n+\frac{1}{2}}}{p_1^{n+1}}$$

C GENERATING RANDOM EXPRESSIONS

Here are algorithms to generate random expressions with n internal nodes. We separate the binary ($p_1 = 0$) and general case. The generation algorithms are similar, but the probability distributions they use are different.

C.1 BINARY TREES

To generate random binary trees with n internal nodes, we use the following one-pass procedure. At all steps e denotes the number of empty nodes, and n the number of operator to be generated.

1. start with an empty node, set $e = 1$
2. sample a number k from $K(e, n)$, the probability distribution that the next internal node is the k-th, among e empty nodes, with n internal nodes left to allocate
3. for the k next empty nodes, sample a leaf and let $e = e - 1$
4. sample an operator, create two empty children, let $e = e + 1$, $n = n - 1$
5. if $n = 0$, we are done, return the tree, else goto step 2

To make this work, we need to calculate, for all e and n, the distribution $K(e, n)$ of probability of the next internal node, among e empty nodes.

Let $D(p, n)$ be the number of different binary subtrees that can be generated from p empty elements, with n internal nodes to generate. Denoting by b_n the Catalan numbers, we have

$$D(0, n) = 0$$

$$D(1, n) = b_n$$

$$D(2, n) = b_{n+1}$$

$$D(p, n) = D(p - 1, n + 1) - D(p - 2, n + 1)$$

To derive the recurrence, consider the case where $n + 1$ internal nodes remain, and we have $p - 1$ empty nodes ($p > 1$), the first empty node is either an internal node or a leaf. If it is an internal node, it has two empty children, and $D(p, n)$ different subtrees can then be generated. If it is a leaf, there are $D(p - 2, n + 1)$ remaining subtrees. As a result

$$D(p - 1, n + 1) = D(p, n) + D(p - 2, n + 1)$$

The recurrence formula allow us to calculate $D(n, p)$, for all interesting values of n and p.

From e empty nodes, $D(e, n)$ subtrees that can be generated. Consider the first empty node, it will be a leaf for $D(e - 1, n)$ trees, and an internal node for $D(e, n) - D(e - 1, n)$.

Consider now thre case where the first node is a leaf, there are $D(e - 1, n)$ such trees. $D(e - 2, n)$ have a leaf as their second node, and $D(e - 1, n) - D(e - 2, n)$ have an internal node.

This provides us with a formula for the probability distribution $K(e, n)$. For k between 1 and e

$$Prob(K(e, n) = k) = \frac{D(e - k + 1, n) - D(e - k, n)}{D(e, n)}$$

C.2 UNARY-BINARY TREES

For unary-binary trees, we use an extended version of the previous algorithm, which decides at step two whether the next operator is unary or binary, and uses a different distributions $L(e, n)$ in the unary case.

1. start with an empty node, set $e = 1$ and $n = n$
2. decide if the next internal node is unary or binary (according to probability p_b, usually $p_b = p_2/(p_1 + p_2)$)
3. if binary, sample a number v from $K(e, n)$

4. if unary, sample a number v from $L(e, n)$
5. for the $v-1$ next empty nodes, sample a leaf and let $e = e - 1$
6. sample an operator (unary or binary according to step 2), create one or two empty children, if binary let $e = e + 1$, let $n = n - 1$,
7. if $n = 0$, we are done, return the tree, else goto step 2

As before, to calculate K and L, we need to derive $D(p, n)$, the number of subtrees that can be generated from p empty nodes, with n internal nodes to generate, for unary-binary trees. The following three equations hold.

$$D(p, 1) = 2p$$
$$D(1, n) = D(1, n-1) + D(2, n-1)$$
$$D(p, n) = D(p-1, n) + D(p, n-1) + D(p+1, n-1)$$

The first one states that since we can build 2 one operator trees from one empty element (one binary, one unary), we can build $2p$ from p empty elements. The second states that if $n > 0$ operators remain to allocate, a single empty element must accommodate either a unary operator (with $D(1, n-1)$ possible subtrees) or a binary operator. The last one is the general case, for $p > 1$. Then the first empty node is either a leaf, a unary or a binary operator.

These three equations allow us to tabulate $D(p, n)$ for any value of n and p. For $K(p, n)$ and $L(p, n)$, we reason as follows. Starting with p empty nodes, and having an unary operator to allocate and n remaining, we have $D(p, n-1)$ trees with an operator on the first empty node, $D(p-1, n-1)$ with an operator on the second, and $D(p-k, n-1)$ with an operator on the k-th node. Overall,

$$Prob(L(p, n) = k) = \frac{D(p-k, n-1)}{\sum_{i=0}^{p-1} D(p-i, n-1)}$$

For $K(p, n)$, similar calculations yield

$$Prob(K(p, n) = k) = \frac{D(p+1-k, n-1)}{\sum_{i=0}^{p-1} D(p+1-i, n-1)}$$

D Generating second order differential equations - an alternative strategy

The generating method described at the end of section 3.3 generates an equation from a possible solution $y(x, c_1, c_2)$ by first inverting y with respect to c_2, yielding $F(x, y, c_1)$ such that $F(x, y, c_1) = c_2$. Differentiating F produces a first order differential equation that we need to solve in c_1. The resulting equation $G(x, y, y') = c_1$ is differentiated to provide a second order equation solved by y.

To guarantee that y can be inverted in c_2, we generate it with only one c_2 leaf. This does not guarantee that the differential equation derived from F can be solved in c_1, and a first version of our generating method relies on trials and error to generate suitable $y(x, c_1, c2)$.

We present an improved method for generating y, which guarantees solvability in c_1 and c_2, at the price of additional constraints in the solutions generated.

To guarantee that $y(x, c_1, c_2)$ solvable in c_2, we sample a unique c_2 in the tree representation of y. Under which additional conditions will $y'F'_y + F'_x = 0$ (with $F(x, y, c_1) = c_2$) be solvable in c_1?

It will be the case if both F'_x and F'_y are linear in c_1 or in a unique function of c_1 only. That is if there is $k(c_1)$ such that

$$F'_x = a(x, y)k(c_1) + b(x, y)$$
$$F'_y = c(x, y)k(c_1) + d(x, y)$$

This happens if $F(x, y, c_1)$ has one of the two forms

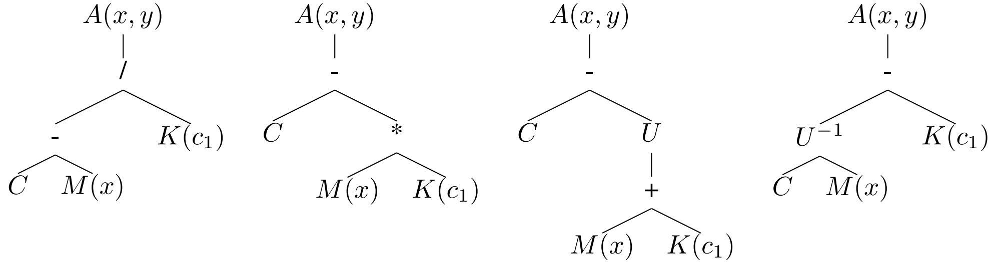
$$F_1(x, y, c_1) = A(x, y)K(c_1) + M(x, y) + P(c_1)$$
$$F_2(x, y, c_1) = U(A(x, y) + K(c_1) + M(x, y) + P(c_1)$$

with U an invertible function.

Since, F is obtained by inverting $y(x, c_1, c_2)$, which has only one instance of c_2, the expression of F will feature only one instance of y (cf section E of the appendix). And so F will in fact have one of those four forms (with only one instance of y):

$$F(x, y, c_1) = A(x, y)K(c_1) + M(x) + P(c_1) = C$$
$$F(x, y, c_1) = M(x)K(c_1) + A(x, y) + P(c_1) = C$$
$$F(x, y, c_1) = U(M(x) + K(c_1)) + A(x, y) + P(c_1) = C$$
$$F(x, y, c_1) = U(A(x, y) + K(c_1) + M(x) + P(c_1) = C$$

Since $P(c_1)$ will vanish when F is derived to provide the first order equation, we can set $P(c_1) = 0$ without loss of generality. By inverting the tree of F, we get that of $y(x, c_1, c_2)$, which will have one of the following structures (U a unary operator, A an expression in x and y, M and expression in x, and K an expression in c_1).



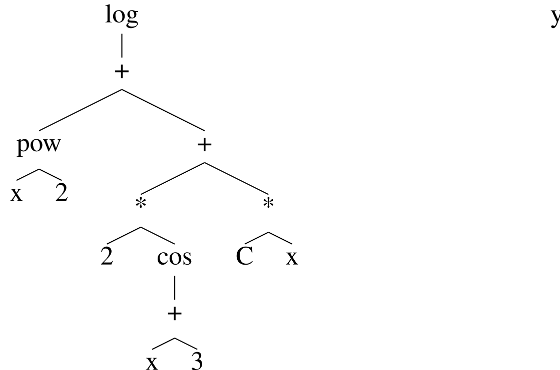
This results in the following method for generating solvable second order differential equations from their solutions.

1. Generate two random functions $M(x)$ and $K(c_1)$
2. Select one of the four subtrees, and build it from $M(x)$ and $K(c_1)$
3. Generate a random function $A(x)$
4. Select a node at random and replace it by the subtree, we haves $y(x, c_1, c_2)$
5. Solve in c_2 to get $F(x, y, c_1) = c_2$
6. Differentiate and solve in c_1 the first order equation
7. Differentiate to get a second order equation solved by $y(x, c_1, c_2)$

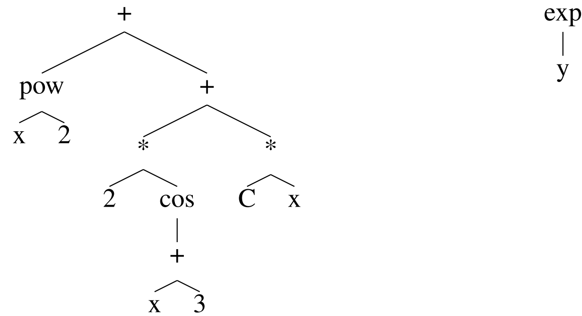
E EQUATION SOLVING BY TREE INVERSION

In our method for generating differential equations, several equations must be solved symbolically. For instance, we derive, from function $y(x, C)$, the bivariate function $F(x, y)$ such that $F(x, y) = C$. Provided C appears only once in the tree if y, and that all operators are invertible, this always can be done, and tree representation make it easy. Here, we provide an example of this procedure.

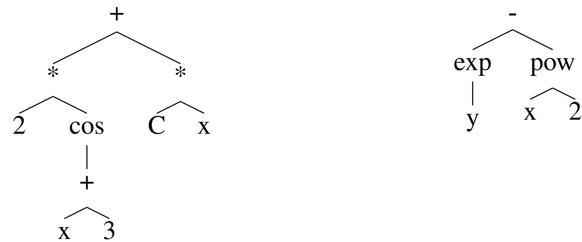
Let $y(x, C) = \log(x^2 + 2\cos(x + 3) + Cx)$. To find F, we build the tree corresponding to $y(x, C)$ and a second tree with only y at the root.



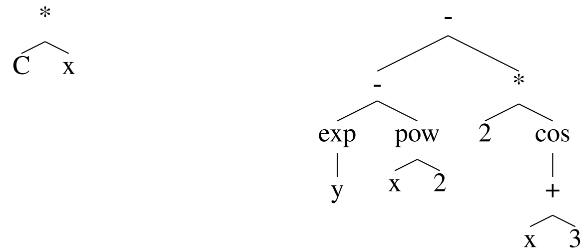
If the root of the left tree is unary, remove it, and add its inverse at the root of the right tree.



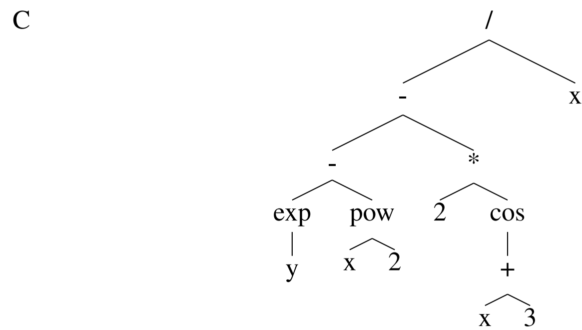
If the root of the left tree is binary, only one of its subtree contains C, move the other to the right tree, by inverting the top node.



Continue until only C remains in the left tree



The right tree is the function $F(x, y)$ sought.



The number of steps in this algorithm is equal to the depth of C in the tree representing $y(x, C)$).