Learning to Reason Mathematically

Arash Mehrjou [1 2] **Ryota Tomioka** [3] **Andrew Fitzgibbon** [3] **Simon Peyton Jones** [3]

Abstract

We introduce the simplification of mathematical expressions as a sequential task whose solution requires understanding the structure of the expressions. We do not assume any expert information and develop a curriculum learning algorithm that makes learning in a space with a highly sparse reward signal possible. Graph Neural Network is used to represent the expressions and we show via an intermediate task that it has sufficient expressive power to keep the necessary information for the simplification. The proposed algorithm is able to learn the simplifying sequence of actions from scratch by solving a curriculum of expressions with increasing complexity.

1. Introduction

Many different learning algorithms with almost the same principles (connectivity in architecture and backpropagation in learning) have achieved considerable success in areas such as object detection (Redmon et al., 2016), speech recognition (Hinton et al., 2012), and machine translation (Wu et al., 2016) in supervised learning and areas such as generative models (Goodfellow et al., 2014) and density estimation (Sohl-Dickstein et al., 2009; Saremi et al., 2018) in unsupervised learning. Reinforcement Learning (RL) is another major area that enjoys the connected hierarchical architecture of neural networks to scale the already known algorithms to more interesting problems (Silver et al., 2017b).

The major focus of machine learning in the past few years has been on problems which are so-called *intuitive* and their analysis is almost effortless for the human brain. Seeing, hearing, voice generation, etc are all natural products of our connectionist brain. However, we need to educate ourselves and think carefully to be able to perform even the simplest *logical* tasks such as algebraic operations. One reason for this difficulty would be the way objects are represented in our brain. Despite the efficacy of our neural circuits in representing natural signals, our brain seems to lack an easy way to represent abstract logical objects such as geometric concepts, mathematical expressions, and etc. It is commonly believed that such a representation will be necessary for Artificial General Intelligence (AGI) when processing natural sensory information and logical thinking are supposed to occur by the same computational framework (Legg & Hutter, 2007; Garnelo et al., 2016).

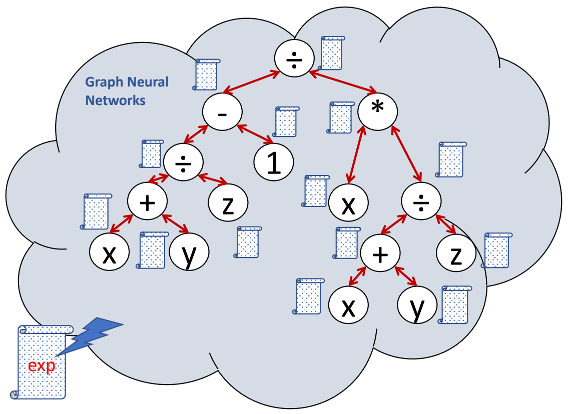


Figure 1. The expression tree of $e := \frac{\left(\frac{x+y}{z}\right)-1}{x*\left(\frac{x+y}{z}\right)}$. The scrolls next to each node represents $\boldsymbol{h}_v$ which is initialized by the node labels and computed after passing through the propagation dynamics function for T steps. The output function aggregates these scrolls and gives an overall representation of the expression tree denoted by the scroll marked with red "exp".

In this work, we propose a representation for mathematical expressions based on Graph Neural Networks (GNN) and use it in an RL task whose goal is to optimize an inherent complexity measure of expressions. We see this simplification as a surrogate task toward logical reasoning whose solution can shed light on RL algorithms in symbolic domains.

Related works— Using machine learning in logical problems have recently received a fair amount of attention, for example, in Automatic Theorem Provers (ATP) (Robinson & Voronkov, 2001). Closest to our work but in a different domain is (Kaliszyk et al., 2018) where

[*]Equal contribution [1]Max Planck Institute for Intelligent Systems, Tübingen, Germany [2]ETH, Zürich, Switzerland [3]Microsoft Research, Cambridge, UK. Correspondence to: Arash Mehrjou <amehrjou@tue.mpg.de>.

Presented at the ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data Copyright 2019 by the author(s).

MCTS (Abramson, 2014) is applied as a search strategy to guide ATP toward more efficient proofs given a large corpus of previous proofs which are translated to a formal language. In the same direction, an RL environment is designed by Huang et al. based on a proof assistant with a large library of supervised data. Other forms of using ML to improve current theorem provers have shown promising results (Gonthier et al., 2013; Loos et al., 2017) suggesting the value of deeper understanding of the learning-based automated logical reasoning. In a different application but with the same flavor of self-taught learning in sparse rewarding state space, McAleer et al. proposed a method to learn the sequence of actions to solve a randomly initialized Rubik cube.

In the following, after an introduction to GNN as necessary background information, the proposed representation is introduced. An intermediate classification task is presented in Section. 3.1 to show the efficacy of the representation. The simplification task which is the main goal of this paper is then presented in Section. 3.2.

2. Background

2.1. Graph Neural Networks

In this section, we briefly review GNN (Scarselli et al., 2009) and the version we employed for this work (Li et al., 2015). GNN is a dynamical message passing algorithm that condenses the information of a graph in an m-dimensional vector. The algorithm can focus on a particular node of the graph or the whole graph as an entirety. Assume $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is the graph of interest and $v \in \mathcal{V}$ a certain node. $\mathrm{GNN}(\mathcal{G}, v) \in \mathbb{R}^m$ gives the representation that preserves the graph structure. Recent theoretical results show that the discriminative power of GNN is comparable to Weisfeiler-Lehman (Weisfeiler & Lehman, 1968) graph isomorphism test (Xu et al., 2018). Assume the *node vector* for node v is represented by $\boldsymbol{h}_v \in \mathbb{R}^D$. Graphs may also have node and edge *labels* represented by $\boldsymbol{l}_v \in \{1, \ldots, L_{|\mathcal{V}|}\}$ and $\boldsymbol{l}_e \in \{1, \ldots, L_{|\mathcal{I}|}\}$ respectively. Let $\mathrm{NBR}(v)$ be the adjacent nodes to v and $\mathrm{Co}(v)$ be the set of edges with v as one end. (See Appendix. A for more detailed definitions.)

To obtain the vectorial representation, information is *propagated* along the edges of the graph by the dynamics

$$\boldsymbol{h}_v^{(t)} = f\left(\boldsymbol{l}_v, \boldsymbol{l}_{\mathrm{Co}(v)}, \boldsymbol{l}_{\mathrm{NBR}(v)}, \boldsymbol{h}_{\mathrm{NBR}(v)}^{(t-1)}\right). \tag{1}$$

that run for T steps to give node vectors $\boldsymbol{h}_v^{(T)}$.

The final representation is formed by aggregating node vectors and initial node labels via a function called *output function* which is defined as $\boldsymbol{o}_v = g(\boldsymbol{h}_v^{(T)}, \boldsymbol{l}_v)$ for each node or $\boldsymbol{o}_\mathcal{G} = g(\boldsymbol{h}_0^{(T)}, \boldsymbol{l}_0, \ldots, \boldsymbol{h}_\nu^{(T)}, \boldsymbol{l}_\nu)$ for the whole graph. The error is backpropagated through T steps to learn the parameters of f and g. In this work, to gain more expressive power, we use the architecture proposed by Li et al. in which the propagation function is implemented by GRU (Cho et al., 2014) and the dynamics does not continue until convergence ($T < \infty$). See Appendix. B for more details.

2.2. Expression Tree

The main purpose of this work is to learn to do simple manipulations on mathematical expressions which are represented by GNN to allow gradient-based learning. By mathematical expressions, we refer to any expression $E(\mathbf{V}) = x \odot y \odot \ldots \odot s \odot t$ where $\mathbf{V} = \{x, y, \ldots, s, t\}$ is a set of variables (symbols) and $\odot \in \{+, -, *, \div, (,)\}$. It is possible to represent every mathematical expression of this form as a tree called *expression tree* (See Fig. 1).

3. Method

In this section, we describe the components of our tasks and the proposed algorithm to solve them. First, we present the way GNN of Section. 2.1 is employed to represent expression trees of Section. 2.2. An expression tree is a graph whose nodes are either variables or mathematical operators. The edges of this graph represent the relative position of variables with respect to operators. GNN does not discriminate between variables and operators and considers both as graph nodes. The initial node vector $\boldsymbol{h}(v) \in \mathbb{R}^D$ is a concatenation of a one-hot vector (indexing each operator and variable) as the node label $\boldsymbol{l}_v \in \mathbb{R}^{|\mathcal{V}|}$ and a zero vector with dimension $D - |\mathcal{V}|$ to allow extra capacity for the representation of nodes. Even though the edges in an expression tree are uni-directional, we found it technically useful to have more edge types in the GNN architecture. This gives the learning algorithm more expressive power if it is necessary for the downstream task.

In the following, we define two tasks: The first task is a binary classification that is a diagnostic step toward the second task. The second task is a self-taught RL agent that learns to apply logical modifications to simplify mathematical expressions only from self-experience. To keep the presentation concise and clear, we deferred most of the details to the appendix. The reader is encouraged to consult the appendix at any point for more details.

3.1. Classification task

We define a classification task to detect if an expression contains a special subexpression. The idea behind this experiment is to make sure that the necessary information for more sophisticated tasks won't be washed out through the dynamics of the propagation function of GNN. For example, applying the rules of algebra on mathematical expressions requires knowing which subexpressions are present in the

Table 1. Binary classification result for detecting whether a particular subexpression exists within host expressions or not

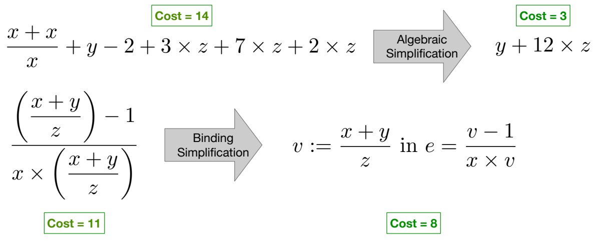
SubExp	$1-x$	$x+y$	$\frac{x}{x+y}$	$\frac{z}{x}+\frac{y}{z}$
Acc	$81 \pm 7\%$	$79 \pm 9\%$	$77 \pm 5\%$	$83 \pm 11\%$

expression to know the legal algebraic manipulations. We developed an algorithm called Host-Virus to synthesize data for this experiment (See Appendix. C for details). The general idea is to synthesize a set of so-called *host* expressions e by growing expression trees with a prior probability over operations, variables, and the depth of the tree. Then, *contaminate* the host expressions by planting a specific expression so-called *virus* in them. The classifier is supposed to detect contamination. The accuracies reported in Table. 3.1 for different virus expressions show that the GNN is able to detect if an expression contains a particular subexpression. For comparison, we did the same experiment for a model that consists of an ordinary embedding layer followed by a dense layer (see Appendix. E). We call this approach Ordinary Embedding Network (OEN). It was observed that OEN easily finds shortcuts in the dataset to reduce its loss which can be seen as over-fitting to the dataset. For example, in the case where $x+y$ is the virus, it achieves a decent accuracy of 78% but it was so certain (0.998%) that the virus expression $x+y$ itself belongs to the non-contaminated class which is obviously wrong. Further investigations showed that OEN had become sensitive to the length of the expression which normally increases when an expression is contaminated with another expression. We observed that the GNN representation is more robust to this effect and it leads to correct classification even for fairly short contaminated expressions.

3.2. Expression simplification task

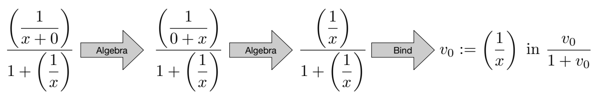
We now move on to a more challenging task which can be seen as a small step toward machines that learn mathematical reasoning such as *algebra* from scratch.

Mathematical reasoning is a sequential task that starts from a set of premises (initial state) and applies a set of logical rules (action space) that are applicable to the current state of the problem. The set of legal moves is often enormous if we do not restrict ourselves to a subset of mathematics. In the terminology of RL, a reward for a logical reasoning task is issued when the theorem is proved or some predetermined goal is achieved. Here, as an example of mathematical reasoning, we focus on the simplification of mathematical expressions and demonstrate it as an RL task. The expressions will be the states of the Markov Decision Process (MDP). The set of possible rules that form the action space can be accurately written. Even this seemingly simple problem turns out to be difficult for an RL agent mainly due to the sparsity of the rewarding states.



Cost = 14
$\frac{x+x}{x}+y-2+3\times z+7\times z+2\times z$
Algebraic Simplification
Cost = 3
$y+12\times z$
$\frac{\left(\frac{x+y}{z}\right)-1}{x\times\left(\frac{x+y}{z}\right)}$
Binding Simplification
$v:=\frac{x+y}{z}$ in $e=\frac{v-1}{x\times v}$
Cost = 11
Cost = 8

Figure 2. Two examples of the cost of expressions and the reduction in their cost by simplification via algebraic rules and binding rules.



$\frac{\left(\frac{1}{x+0}\right)}{1+\left(\frac{1}{x}\right)}$
Algebra
$\frac{\left(\frac{1}{0+x}\right)}{1+\left(\frac{1}{x}\right)}$
Algebra
$\frac{\left(\frac{1}{x}\right)}{1+\left(\frac{1}{x}\right)}$
Bind
$v_0:=\left(\frac{1}{x}\right)$ in $\frac{v_0}{1+v_0}$

Figure 3. An example of the simplification process when both algebraic and binding rules are used.

Let's consider an MDP defined as $\mathcal{M}=(\mathcal{S},\mathcal{A},c)$. Every state of this MDP is a mathematical expression $e\in\mathcal{S}$ similar to Fig. 1. The action space $\mathcal{A}$ is the set of applicable rules on the current state of the MDP, hence can be written as a function of the expression $\mathcal{A}(e)$. The cost function (negative reward function) $c(e)$ is defined as a measure of the complexity of the expression e. Every mathematical operation contributes a certain amount of complexity to the cost of an expression, roughly reflecting the time it takes a computer to compute that operation. We consider the following atomic costs for the operations $\{+:1,-:1,*:2,/:2\}$ implying that multiplication and division are twice as costly as addition and subtraction which is a plausible assumption in computer engineering. The cost of an expression is then defined as the sum of the cost of all its operations. The ultimate purpose of the algorithm is to start from initial expression e_{init} and apply a sequence of logical actions chosen from $\mathcal{A}(e)$ on each expression e to find the simplest possible mathematical expression e_{final} which is logically equivalent to e_{init} but with lower cost, i.e., $c(e_{\text{final}})\leq c(e_{\text{init}})$. We call this task *expression simplification*. Mastering this sequential task requires the agent to know about the expression and its subexpressions which will be reflected in the GNN representation. The earlier task in Section. 3.1 was designed to test such capability in GNN representation. In other words, if the GNN representation of an expression degenerates footprints of its subexpressions, there is no way to simplify expressions such as $(c*e)/e\to c$ that requires detecting the common subexpression e in the nominator and denominator of the ratio. We leave the details of the task to the Appendix. H and give the overall description of the task here. The RL agent starts with an initial expression e_{init} as the ini-

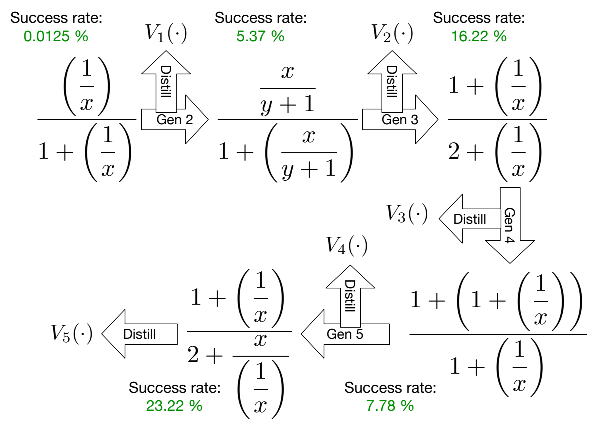


Figure 4. The progress of generation-by-generation curriculum learning strategy that makes the learning possible in an environment with highly sparse rewarding states.

tial state of the MDP. A set of logical manipulations (rules) are possible for e_{init}. We define two sets of logical manipulations (algebraic, binding). Algebraic rules correspond to a set of standard algebraic actions such as commutativity, addition with 0, etc (See Table. 2 in the appendix for details). Binding rules correspond to the *change of variables* when a frequent costly subexpression is assigned to a new variable which takes the role of the subexpression. An example of both classes of manipulations can be seen in Fig. 2. There are occasions when both sets of actions are required to achieve a simpler expression as shown in Fig. 3. The RL agent applies an action chosen from the possible logical manipulations for the expression e_{init} and transits to the resultant expression as the next state of the MDP. The value function $V(\cdot, \boldsymbol{\theta}_V)$ (Schulman et al., 2015) is learned from $(e_{\text{init}}, c(e(t)) - c(e_{\text{init}}))$ pairs and guides the logical manipulations. Roughly speaking, the actions that result in expressions with a higher value are more likely to be explored by the agent. See Appendix. G for further details. Learning this function requires rewarding states which are unfortunately hard to obtain in this task. In the next section, we present a curriculum learning approach to deal with this issue.

3.2.1. Curriculum Learning

At the beginning of the training procedure, the value function is untrained and it has no preference for any action on any expression. Hence, the strategy will be applying a sequence of random legal actions and choosing the sequence that gives the best value (simplest expression). However, the number of legal manipulations for expression e grows rapidly with the complexity of e. In other words, a deeper expression tree allows much more algebraic and binding manipulations. Consequently, the number of distinct episodes starting from expression e becomes enormously large such that exploring all of them for a rewarding state (an expression simpler than e_{init}) becomes quickly infeasible. This is largely due to the fact that the value function is unbiased at the beginning and does not downweight the actions which are not so promising. To combat this issue, we manually design a curriculum of expressions $[e^{(0)}, e^{(1)}, \ldots, e^{(g)}, \ldots]$ with increasing complexity such that $e^{(g)}$ shows the expression of g^{th} *generation*. The first generation expression $e^{(0)}$ of the curriculum must be simple enough such that by choosing uniformly random actions, a couple of rewarding states (simpler expressions) can be found among a reasonable number of simulated episodes. Once a simpler expression is found in an episode, the value function is updated for all expressions of that episode. In other words, the knowledge about solving $e^{(0)}$ is distilled in $V_1(\cdot)$. For the expression of the next generation $e^{(1)}$, the actions are chosen by a mixture of uniform distribution and the Boltzmann distribution derived from V_1 (See Appendix. F). This means that we partially rely on the already accumulated knowledge in the value function but still leave some room for purely random simulations. Fig. 4 shows the expressions of a designed curriculum and also the success rate of simulations. The success rate is the ratio of episodes that have led to an expression with the least possible cost over the number of all explored episodes. As expected, the success rate of the initial agent that takes purely random actions for $e^{(0)}$ is very small but still enough to bootstrap $V_1(\cdot)$ and guide the simulations for the next generation to achieve a significant increase in the success rate. Moving from one generation to the next, the information of the successful episodes will be cumulatively distilled in the value function as detailed in Appendix. G.

3.3. Conclusion

We showed the efficacy of GNN to represent mathematical expressions. We designed a task called *Expression Simplification* whose solution requires a good representation of expressions. As a middle step toward this task, we also designed a supervised learning task that classifies expressions according to the presence of a particular subexpression in them. This intermediate step gives us some assurance that GNN can be a good representation for the simplification task. We see the classification step also as a diagnostic tool that helps us choose crucial hyperparameters of GNN such as the number of propagation steps T and the number of edge types. Viewing the introduced RL task as a step toward goal-driven logical manipulation of symbolic objects, a possible direction in the future would be extending the set of expressions to other symbolic objects such as incorporating unary functions in the expressions. This would be useful in learning the exact equations that govern the observed data rather than uninterpretable neural network counterparts.

References

Abramson, B. *The expected-outcome model of two-player games*. Morgan Kaufmann, 2014.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Chollet, F. keras. https://github.com/fchollet/keras, 2015.

Garnelo, M., Arulkumaran, K., and Shanahan, M. Towards deep symbolic reinforcement learning. *arXiv preprint arXiv:1609.05518*, 2016.

Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., OConnor, R., Biha, S. O., et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pp. 163–179. Springer, 2013.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680, 2014.

Hinton, G., Deng, L., Yu, D., Dahl, G., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Kingsbury, B., et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

Huang, D., Dhariwal, P., Song, D., and Sutskever, I. Gamepad: A learning environment for theorem proving. *arXiv preprint arXiv:1806.00608*, 2018.

Kaliszyk, C., Urban, J., Michalewski, H., and Olšák, M. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems*, pp. 8822–8833, 2018.

Legg, S. and Hutter, M. Universal intelligence: A definition of machine intelligence. *Minds and machines*, 17(4): 391–444, 2007.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Loos, S., Irving, G., Szegedy, C., and Kaliszyk, C. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

McAleer, S., Agostinelli, F., Shmakov, A., and Baldi, P. Solving the rubik's cube without human knowledge. *arXiv preprint arXiv:1805.07470*, 2018.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

Robinson, A. J. and Voronkov, A. *Handbook of automated reasoning*, volume 1. Gulf Professional Publishing, 2001.

Saremi, S., Mehrjou, A., Schölkopf, B., and Hyvärinen, A. Deep energy estimator networks. *arXiv preprint arXiv:1805.08306*, 2018.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017a.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017b.

Sohl-Dickstein, J., Battaglino, P., and DeWeese, M. R. Minimum probability flow learning. *arXiv preprint arXiv:0906.4779*, 2009.

Weisfeiler, B. and Lehman, A. A. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9): 12–16, 1968.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

Learning to Reason Mathematically

Supplementary

A. Definitions

The predecessor and successor nodes of v are respectively defined as $\mathrm{IN}(v) = \{v' | \, (v', v) \in \mathcal{I}\}$ and $\mathrm{OUT}(v) = \{v' | \, (v, v') \in \mathcal{I}\}$ for a directed graph. $\mathrm{NBR}(v) = \mathrm{IN}(v) \cup \mathrm{OUT}(v)$ is the set of all adjacent nodes to v and $\mathrm{Co}(v) = \{(v', v'') \in \mathcal{I} | v = v' \vee v = v''\}$ is the set of all edges connected to the node v.

B. Architectural Details

In this sections, we present the details of the GNN architecture which is used in this paper. As mentioned in the main text, we used the architecture introduced in (Li et al., 2015) that extends the original GNN (Scarselli et al., 2009) in two main lines: Firstly, It allows nonlinear propagation function via a recurrent unit called GRU. Secondly, it runs the dynamics of the propagation function for certain time steps T in contrast to (Scarselli et al., 2009) where the dynamics runs until convergence. Setting $T = 10$ works well for a wide variety of experiments including those introduced in this paper. We observed that too small or too large T both deteriorates the expressive power of GNN. We set 5 edge types for the GNN to gain sufficient expressive power. Notice that edge types can be seen as generalized directions. Even though each edge can take one of two directions, it can take several types allowing higher representational power as each edge type corresponds to a new adjacency matrix. The latent space of the GRU unit is set to 200 in our experiments. The output (aggregation) function is implemented by an MLP with one hidden layer with the dimension of 200.

C. Host - Virus algorithm

To create the dataset required for the classification task, we developed an algorithm inspired by the way viruses act in nature. We first generate a set of random expressions by growing the expression trees as Fig.1. To grow a tree, the synthesizer alternately chooses between a list of available variables such as $\{x, y, z\}$ and a list of available operands $\{+, -, *, \div\}$ while sampling the members of each set with some prior probability (uniform in our setting). The depth of the tree is also randomly chosen in the range $[1, 4]$. We put these expressions in a set called *hosts*. Then we define the expression e', called *virus*, where detecting its presence in the other expressions is the goal of the classification task. To this aim, we check every member of the set *hosts* for variables in the layers of the expression tree deeper than 2. The virus then contaminates the host with a fixed probability (0.5 in our experiment) by replacing one of these variables located at inner layers of the expression tree of the host. By exposing each host expression to the virus, we will have a set of contaminated expressions that are considered as *positive* samples. Other expressions which are not contaminated by the virus are considered as *negative* samples for the learning process. We make sure that in the final dataset which is used for training the classifier, the number of positive and negative samples are approximately equal.

D. Details of the classification experiment

For the classification task, we employed a GNN architecture similar to Appendix. B and used the Host-Virus algorithm as detailed in Appendix. C to create $100,000$ positive and $100,000$ negative examples. 80% of this dataset is used for training and the rest 20% for testing. We train the algorithm end-to-end with batch-size = 16 and learning-rate = 0.0001 with Adam optimizer and repeated the entire process for 1000 times to make sure the observations are not dependent on the initialization and other random effects. The test accuracy in Table. 3.1 for four different virus subexpressions is computed over the left-out portion of the dataset.

E. Details of the classification baseline

As a baseline for the classification experiment, we treated mathematical expressions as textual data and employed ordinary neural networks on top of an embedding layer (OEN model). The textual inline representation of the mathematical expression is first tokenized. For the dataset generated by the Host-Virus algorithm of Section. 3.1, the vocabulary becomes

$$\{`\ `, `(`, `)`, `x`, `y`, `z`, `1`, `2`, `3`, `+`, `-, `*`, `/`\} \tag{2}$$

where that tokenization is character-level. Each expression is mapped to a sequence of integers using (2) and zero-padded to an array of length 100 which is larger than the length of any expression in the dataset. Integer-encoded expressions are then fed to a Keras's embedding layer (Chollet, 2015) that outputs 10-dimensional latent vectors. The embedding layer maps every input expression to a 10×100 matrix that is flattened into a 1000-dimensional vector. A single dense layer is used on top of this latent layer that outputs the scalar class as a real in the range $[0, 1]$. The Keras's 100-to-10 embedding layer has 120 parameters and the 1000-to-1 dense layer has 1001 parameters resulting in 1121 number of parameters in total.

F. Details of random / semi-random simulations

Reward signal is necessary for any learning in MDPs. The goal of the RL task of this paper is to apply a sequence of mathematical modifications on some mathematical expressions to reduce the cost of that expression. Unfortunately, the reward space is not dense here. Many sequences of modifications may not change the cost (or even increase the cost) of the expressions for a couple of steps before the expression with the minimum cost is obtained. For example, Fig. 5 shows a sequence of actions applied on the expression e_{init} that first increases the cost but finally results in reduced cost which is indeed minimum. This makes the learning process immensely difficult because the unchanged or increased cost in intermediate steps may result in early disappointment. To mitigate this problem, the initial (input, output) pairs to train the value function is created by applying several sequences of purely random (but of course legal) actions on the initial expression. This process is sometimes called *simulation* phase in the literature and has turned out to be effective when the model of the environment is given (e.g. in games like Chess (Silver et al., 2017a)). Assume few episodes out of many randomly simulated episodes arrive at expressions with reduced cost. We then update the value function to regress from all expressions in a successful episode to the achieved advantage signal $c(e_{\text{init}}) - c(e_{\text{init}})$. With a slight abuse of terminology, we have used *value* instead of *advantage* throughout the paper even though the way it is computed as the difference between the cost of the current and the final expressions is more aligned with the definition of the *advantage* function. This process gives the value function some idea about the goodness of expressions in the process of simplification. This knowledge is then used to bias the exploration when the agent wants to solve the expression of the second generation. We observed that the success rate (ratio of successful episodes) increased considerably in this case showing that the initial knowledge distilled in the value function of the previous generation favors the random episodes in the right direction. The bias is imposed by a mixture of uniform and Boltzmann distributions where the energy function of the Boltzmann distribution is proportional to the negative value function. This becomes an alternating process of cumulatively learning the value function, and use it to make the exploration more efficient and generate more data points to improve the value function further. Continuing this process results in a curriculum-learning approach in the sense that we need to start with simpler expressions such that it is feasible to arrive at some rewarding states by applying random actions on them for a reasonable number of times. The resultant few successful episodes are distilled in the value function and indirectly helps to increase the ratio of successful episodes for the expressions of subsequent generations. We call it semi-random since the choice of actions at each step is biased by the knowledge that the value function has accumulated so far.

G. Details of the Value function

For the RL task, we implement the value function as a regressor from states (expressions) to reals, i.e., $V : \mathcal{E} \to \mathbb{R}$ where $\mathcal{E}$ is the space of all expressions. Similar to other value function-based RL, the value function must capture the goodness of each state in terms of the long-run return (accumulated discounted reward). The target for the parameterized value function is determined by the simulations as described in Section. F. The regressor function $\hat{V}(\cdot; \boldsymbol{\theta}_V)$ maps the GNN representation of the current state (expression) $\boldsymbol{h}_e$ to the maximum advantage achieved in this episode. In practice, the result of several simulations is saved in a dataset $\{(e_n, r_n)\}_{n=1}^{N}$ consisting of (expression, success rate) pairs and the following loss function is optimized to update the parameters of the value function

$$\boldsymbol{\theta}_V^* = \underset{\boldsymbol{\theta}_V}{\operatorname{argmin}} \, \|\hat{V}(\boldsymbol{h}_{e_n}; \boldsymbol{\theta}_V) - r_n\|_2^2 \tag{3}$$

where $\boldsymbol{h}_{e_n}$ is the representation of the expression e_n produced by the trained GNN up to this point in the algorithm. Notice that the value function is updated for all the expressions that appear in the episode.

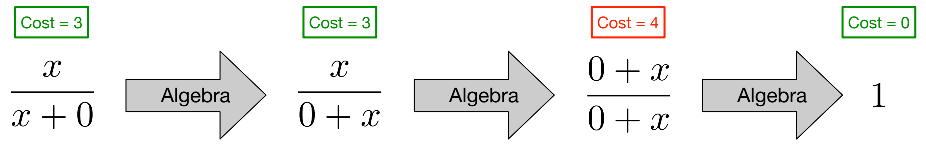


Figure 5. An example of the simplification process with temporary increase in cost. Notice that there exist another sequence of action that does not result in the increase in cost but we can easily think of scenarios that simplification of mathematical expressions requires adding and subtracting some terms that leads to temporary increase in the cost.

H. Details of the RL task

Currently, we don't have a method to encode the actions (rules) in our learning algorithm. Therefore, learning value-action function $Q(e, a)$ instead of the value function $V(e)$ requires considerably more samples. This is due to the fact that not being able to encode the actions, makes it impossible to transfer knowledge from a pair (e, a_1) to pair (e, a_2). Simply speaking, if we need N samples to approximate $V(e)$, we need roughly $|\mathcal{A}(e)|N$ samples to approximate $Q(e, a)$ with comparable accuracy. Therefore, to decide the action to take at state (expression) e, the outcome of every action is produced first and the value function is evaluated for each of the products. The action is then taken according to the mixture distribution $\mathrm{Uniform(a) + p(a|e)}$ where $p(a|e)$ is the Boltzmann distribution defined as

$$p(a|e) = \frac{e^{\frac{V(e')}{\tau}}}{\sum\limits_{e'' \in \mathcal{S}(e,a)} e^{\frac{V(e'')}{\tau}}} \tag{4}$$

where $\mathcal{S}(e, a) = \{\bar{s}|\bar{e} = M(e, a),\ \forall a \in \mathcal{A}(e)\}$ and τ is a scale hyperparameter which is set to 1 in our experiments. Notice that the environment model $M(e, a)$ is a deterministic function $M : \mathcal{E} \times \mathcal{A} \rightarrow \mathcal{E}$ which applies action a on the expression e and produces the resultant expression. M simply encodes the rules of mathematics. In the terminology of RL, M is the dynamical model of the environment.

H.1. Mathematical Rules

In the following, we describe two sets of rules as the manipulations on mathematical expressions which is the action space of the RL task.

Table 2. The set of algebraic rules to modify mathematical expressions

Algebraic Rules

Name	Description
$\frac{a}{a} \equiv 1$	Self-Division
$a * 1 \equiv a$	Multiplication by 1
$(a+b) - c \equiv (a-c) + b$	Threefold commutativity
$a \pm 0 \equiv a$	Addition / Subtraction by 0
$5 * 4 \equiv 20$	Evaluating numeric values
$a \odot b \equiv b \odot a; \odot \in \{+, -, *\}$	Commutativity of addition, subtraction, and multiplication
$a * b + a * c \equiv a * (b + c)$	Distributive property of multiplication to addition

Table 3. The set of binding rules to modify mathematical expressions.

Binding Rules

LHS	RHS	Description
$\mathrm{op}(f(x), g(x))$	let $a := f(x)$ in $op(a, g(x))$	Extracting a subexpression out of an operator
$\mathrm{op}(x,$ let $a := f(x)$ in $g(a))$	let $a := f(x)$ in $op(x, g(a))$	Bind-Extracting a subexpression out of an operation
let $a := f(x)$ and $b := f(x)$ in $\mathrm{op}(a, b, g(x))$	$a := f(x)$ in $g(a, a, g(x))$	Binding two equivalent subexpressions

In the following, two sample episodes of simplification by means of the above rules are presented.

H.2. An example of the simplification by binding rules

$$
\begin{aligned}
&[(1.0/x)/((1.0/x) + (1.0/x)),\\
&((1.0/x)/((1.0/x) + (\mathrm{var0} := (1.0/x) \text{ in } \mathrm{var0}))),\\
&((1.0/x)/(\mathrm{var0} := (1.0/x) \text{ in } ((1.0/x) + \mathrm{var0}))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } ((1.0/x)/((1.0/x) + \mathrm{var0}))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } ((\mathrm{var1} := (1.0/x) \text{ in } \mathrm{var1})/((1.0/x) + \mathrm{var0}))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var1} := (1.0/x) \text{ in } (\mathrm{var1}/((1.0/x) + \mathrm{var0})))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var0}/((1.0/x) + \mathrm{var0}))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var0}/((\mathrm{var1} := (1.0/x) \text{ in } \mathrm{var1}) + \mathrm{var0}))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var0}/(\mathrm{var1} := (1.0/x) \text{ in } (\mathrm{var1} + \mathrm{var0})))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var1} := (1.0/x) \text{ in } (\mathrm{var0}/(\mathrm{var1} + \mathrm{var0})))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var0}/(\mathrm{var0} + \mathrm{var0})))]
\end{aligned}
$$

H.3. An example of the simplification by algebraic + binding rules

$$
\begin{aligned}
&[((1.0/x)/(1.0 + (1.0/(x + 0.0)))),\\
&((\mathrm{var0} := (1.0/x) \text{ in } \mathrm{var0})/(1.0 + (1.0/(x + 0.0)))),\\
&((\mathrm{var0} := (1.0/x) \text{ in } \mathrm{var0})/(1.0 + (1.0/(0.0 + x)))),\\
&((\mathrm{var0} := (1.0/x) \text{ in } \mathrm{var0})/(1.0 + (1.0/x))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var0}/(1.0 + (1.0/x)))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var0}/(1.0 + (\mathrm{var1} := (1.0/x) \text{ in } \mathrm{var1})))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var0}/(\mathrm{var1} := (1.0/x) \text{ in } (1.0 + \mathrm{var1})))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var1} := (1.0/x) \text{ in } (\mathrm{var0}/(1.0 + \mathrm{var1})))),\\
&(\mathrm{var0} := (1.0/x) \text{ in } (\mathrm{var0}/(1.0 + \mathrm{var0})))]
\end{aligned}
$$