Learning to Solve Symbolic Mathematical Equations using RL

Amitayush Thakur [1] **Garvit Mohata** [1]

1. Problem Statement and Motivation

1.1. Motivation.

The challenge of solving symbolic equations, traditionally managed by Computer Algebra Systems (CAS) such as SymPy (Meurer et al., 2017) and Mathematica (Inc.), hinges on applying a vast array of pre-established mathematical rules. Despite their effectiveness, these systems do not evolve by learning new methods or elucidating the reasoning behind their solutions, particularly limiting them in educational contexts where understanding the solving process is invaluable. This reveals a compelling niche for innovation: could we devise algorithms that not only apply basic algebraic principles but also dynamically learn and improve their problem-solving strategies?

Our research proposes leveraging Reinforcement Learning (RL) to bridge this gap. By treating the solution of symbolic equations as a series of strategic decisions, RL offers a unique avenue to not only master established algebraic manipulations but also to potentially discover more efficient solving techniques. This approach serves as a proof of concept for the viability of learning-based algorithms in computational mathematics. Starting with simple linear equations, our experiments aim to validate the hypothesis that RL algorithms can indeed learn to solve symbolic equations, paving the way for more advanced explorations and applications in the field.

1.2. MDP Formulation.

In this section, we formally state our problem of how to solve linear equations for one specific variable, "x". Sometimes, these equations might look more complicated at first because they include other variables. However, as we work through them with algebra, those extra variables end up not affecting the solution—they just cancel out. This setup isn't just about finding the solution to these equations. It's also a great opportunity for us to learn different ways to work with equations, going beyond the basics of solving for " x".

[1]The University of Texas at Austin. Correspondence to: Amitayush Thakur <amitayush@utexas.edu>, Garvit Mohata <garvit.mohata@utexas.edu>.

This formulation allows us to test our hypothesis if we can learn the algorithms for doing algebraic manipulation given an expression.

1.2.1. States

We define the state space S to include all configurations of linear equations that, through algebraic manipulations, are ultimately reducible to the canonical form $a \cdot x + b = 0$. Formally, each state $s \in S$ encapsulates an equation initially presented as:

$$s : \sum_{i=1}^{k_1} a_i \cdot l_{\alpha_i} = \sum_{j=1}^{k_2} b_j \cdot l_{\beta_j}$$

where:

- $l_{\alpha_i}, l_{\beta_j}$ represent variable literals, taking one of 27 possible values, specifically the alphabet $a - z$ and the numeral 1. In other words, for all i, j, $l_{\alpha_i}, l_{\beta_j} \in \{1, a, b, ..., z\}$. The literal '1' allows us to incorporate constant terms. Multiplication with the literal '1' is often simplified in the state and is represented by the absence of any variable.
- $a_i, b_j \in \mathbb{R}$ are the real number coefficients,
- The equation is subject to the constraint that the sum of all coefficients for variables other than x equals zero, effectively ensuring that these variables do not contribute to the final form of the equation. This constraint is formalized as:

$$map(\sum_{\substack{i=1 \\ l_{\alpha_i} \neq x}}^{k_1} a_i \cdot l_{\alpha_i} - \sum_{\substack{j=1 \\ l_{\beta_j} \neq x}}^{k_2} b_j \cdot l_{\beta_j}, \vec{v_l}) = 0. \quad (1)$$

  where the map function maps the literal variables to their values as described by the vector $\vec{v_l} \in \mathbb{R}^{27}$ vector. To make the equation constrain the coefficient of every literal except 'x' to zero we restrict $\vec{v_l}$ to just one-hot vectors in $\{0, 1\}^{27}$. The Equation (1) holds for all such one-hot vectors $\vec{v_l}$.

This state space S thus encompasses a broad spectrum of linear equations, ranging from those already in or near the

target simplification to more complex configurations that initially obscure their underlying linearity. The end goal of manipulation within this space is to simplify or transform any given equation into the streamlined form $a \cdot x + b = 0$, underscoring the intrinsic linearity of the equation and the process of revealing it. Figure 1 shows some of the states which are part of MDP.

$$2x + 3y = 3y - 4 \tag{2}$$
$$5x - 2 = 0 \tag{3}$$
$$x + 0.5 - 0.5x = 0 \tag{4}$$

Figure 1: Examples of states in the defined MDP framework showcasing various stages of simplification towards the form $a \cdot x + b = 0$.

With this formulation of states, we can also represent unreduced higher-order equations in 'x' which ultimately simplifies and cancels out any higher-order terms of 'x' by renaming the higher powers of 'x' with different literals (see Equation (5)).

$$\begin{aligned} x^2 + x + 2 &= x^2 - x \\ &\iff y + x + 2 = y - x \\ &\qquad \text{where } y = x^2 \end{aligned} \tag{5}$$

1.2.2. Actions and Transisitions

Given the defined state space S, the set of actions A that can be applied to any state $s \in S$ to progressively transform it towards the canonical form $a \cdot x + b = 0$ includes:

1. **CollectVariables**(l): Groups terms involving the variable l, simplifying the structure. This action is expressed as:

$$Given\ s : \sum_{i=1}^{k_1} a_i \cdot l_{\alpha_i} = \sum_{j=1}^{k_2} b_j \cdot l_{\beta_j}$$

$$CollectVariables(s, l) :=$$

$$s' : a_l \cdot l + \sum_{\substack{i=1 \\ l_{\alpha_i} \neq l}}^{k_1'} a_i \cdot l_{\alpha_i} = b_l \cdot l + \sum_{\substack{j=1 \\ l_{\beta_j} \neq l}}^{k_2'} b_j \cdot l_{\beta_j}$$

$$where\ a_l,\ b_l \in \mathbb{R} \wedge k_1' < k_1 \wedge k_2' < k_2$$

   This action aims to arrive at a concise representation of variable l's influence. The action **CollectVariables**(1) simply collects the constant terms i.e. it does arithmetic simplification involving constant terms on both sides.

2. **MoveTerms(l)**: Moves terms which have literals to isolate x, through symmetric addition or subtraction:

$$Given\ s : \sum_{i=1}^{k_1} a_i \cdot l_{\alpha_i} = \sum_{j=1}^{k_2} b_j \cdot l_{\beta_j}$$

$$MoveLiteralTerms(s, l) :=$$

$$s' : \sum_{\substack{i=1 \\ l_{\alpha_i} \neq l}}^{k_1 - 1} a_i \cdot l_{\alpha_i} = \sum_{\substack{j=1 \\ l_{\beta_j} \neq l}}^{k_2} b_j \cdot l_{\beta_j} + b_l \cdot l$$

   where $b_l \in \mathbb{R}$ and $-b_l$ was coefficient of l on RHS in s.

   The action **MoveTerms**(1) simply moves the constant terms to the LHS.

3. **DivideByNonZeroCoeffOfX**: Normalizes x's coefficient to 1 by division, ensuring $a \neq 0$:

$$a \cdot x = b \to x = \frac{b}{a}.$$

   This action does not change the state of the equation unless it is in its reduced form of $a \cdot x = b$ or when $a = 0$.

4. **IdentityAction(l)**: There are three identity operations which maintains the equation's structure unchanged:

$$l + 0 = l, \quad l \cdot 1 = l, \quad l \cdot 0 = 0.$$

An emphasis is placed on avoiding division by zero, as such actions do not change the equation's state and ensure mathematical validity throughout the transformation process.

1.2.3. Sparse Reward Strategy for Equation Solving

In our approach to employing reinforcement learning (RL) for algebraic equation solving, we adopt a sparse reward strategy that focuses on the ultimate goal: successfully solving the equation. This method provides a binary reward signal:

$$R(s, a) = \begin{cases} 1 & \text{if the equation is successfully solved,} \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

rewarding the agent only upon reaching a solution. This approach not only simplifies the reward structure but also promotes an unbiased exploration of algebraic manipulation strategies.

Sparse rewards inherently avoid embedding human biases into the learning process, a common risk in environments with more granular, step-by-step reward systems. By refraining from dictating the 'correctness' of intermediate steps

based on human preconceptions, our RL model is free to discover and utilize potentially novel and efficient strategies for solving equations. This not only aligns the learning process with the problem's fundamental objective but also encourages a broader investigation into the space of possible solutions. The adoption of a sparse reward system represents a deliberate choice to foster innovation and efficiency in automated equation solving.

2. Implementation Details

2.0.1. Tools Needed to Build Environment.

We develop an environment for exploring algebraic manipulations using SymPy (Meurer et al., 2017), a Python library for symbolic mathematics, to enable reinforcement learning algorithms to learn and apply algebraic actions effectively. The environment represents equations as states, defines a set of algebraic actions, and utilizes SymPy's functionalities for action application and state transition.

State Representation

Each state in the environment corresponds to a symbolic equation represented using SymPy's expression objects, enabling flexible manipulation and analysis of equations.

Example:

$$\text{state} = \text{Eq}(x + 2 * x + y, y - 3x + 6) \tag{7}$$

Action Set

A limited set of algebraic actions is defined, leveraging SymPy functions to perform transformations on the equations. This selection is deliberately constrained to prevent solving the equation in a single step, thereby encouraging the reinforcement learning algorithm to learn a sequence of strategic actions toward simplification and solution. These actions are carefully selected to promote a deeper understanding of algebraic manipulation through incremental learning and decision-making processes, rather than immediate resolution.

Transition Function

The transition function applies the selected action to the current state, generating a new state through the corresponding SymPy operation.

2.0.2. RL Algorithms and Baseline

The exploration of algebraic manipulations within our SymPy-based environment leverages Reinforcement Learning (RL) algorithms to iteratively learn the most effective strategies for equation simplification and solution. A key approach we employ is *Linear Function Approximation* for the policy learning mechanism. This method approximates the value function or policy function as a linear combination of features derived from the state representation, offering a balance between simplicity and the capability to generalize across different states.

Linear Function Approximation

Linear Function Approximation is characterized as follows:

$$V(s) \approx w^T \cdot \phi(s), \tag{8}$$

where $V(s)$ estimates the value of state s, w is the weight vector, and $\phi(s)$ is the feature vector extracted from state s. The choice of features, $\phi(s)$, is critical for capturing the essence of the state of the task. For our application, the features include:

- The number of operators (e.g., $+$, $-$, $*$, $/$) on the left-hand side (LHS) and right-hand side (RHS) of the equation. These features capture the equation's complexity and the balance of operations between sides.
- The number of distinct variables present on the LHS and RHS. This reflects the equation's variable diversity and helps gauge the effort required to isolate the target variable x.

These features aim to provide a comprehensive yet concise representation of the equation's structure, facilitating the RL algorithm's learning of effective manipulation strategies.

Baseline for Comparison

As a baseline, we propose employing a *Random Action Selection* strategy, where the algorithm selects actions from the available set at random, without any learning or adaptation. This baseline provides a foundational comparison point to evaluate the learning efficiency and effectiveness of the Linear Function Approximation approach. By quantifying the improvement in performance over random action selection, we can assess the RL algorithm's capability to learn and apply meaningful algebraic manipulations toward solving equations.

We also want to compare against a *hand-crafted strategies* of trying various actions like `CollectVariables(l)`, `MoveVariables(l)`, etc. depending on the size of LHS and RHS. Some ideas like first isolating variables, then collection, and homogenization as proposed by Sterling et al. (1982) can be used for creating hand-crafted strategies for solving symbolic linear equations.

3. Stretch Goals

3.1. More Complex Equations

For our project, we are initially limiting the scope to only linear equations. However, we would love to expand this

work to more complicated non-linear equations, or equations that use trigonometric functions.

3.2. No Manual Feature Engineering

We initially plan to start with the approach described in Linear Function Approximation (Section 2.0.2). However, we plan to test it with more complex function approximation strategies that don't need any feature engineering like using LLMs to represent the state i.e. mathematical equations. We can try approaches like DQN or Policy Optimization with LLM as a function approximation or policy network.

References

Inc., W. R. Mathematica, Version 14.0. URL https://www.wolfram.com/mathematica. Champaign, IL, 2024.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. 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.

Sterling, L., Bundy, A., Byrd, L., O'Keefe, R., and Silver, B. Solving symbolic equations with press. In *Computer Algebra: EUROCAM'82, European Computer Algebra Conference Marseille, France 5–7 April 1982*, pp. 109–116. Springer, 1982.