Structure-Unified M-Tree Coding Solver for Math Word Problem

Bin Wang, Jiangzhou Ju, Yang Fan, Xinyu Dai*, Shujian Huang, Jiajun Chen
National Key Laboratory for Novel Software Technology, Nanjing University
Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing
{wangbin, jujiangzhou, fanyang}@smail.nju.edu.cn
{daixinyu, huangsj, chenjj}@nju.edu.cn

Abstract

As one of the challenging NLP tasks, designing math word problem (MWP) solvers has attracted increasing research attention for the past few years. In previous work, models designed by taking into account the properties of the binary tree structure of mathematical expressions at the output side have achieved better performance. However, the expressions corresponding to a MWP are often diverse (e.g., $n_1+n_2\times n_3-n_4$, $n_3\times n_2-n_4+n_1$, etc.), and so are the corresponding binary trees, which creates difficulties in model learning due to the non-deterministic output space. In this paper, we propose the Structure-Unified M-Tree Coding Solver (SUMC-Solver), which applies a tree with any M branches (M-tree) to unify the output structures. To learn the M-tree, we use a mapping to convert the M-tree into the M-tree codes, where codes store the information of the paths from tree root to leaf nodes and the information of leaf nodes themselves, and then devise a Sequence-to-Code (seq2code) model to generate the codes. Experimental results on the widely used MAWPS and Math23K datasets have demonstrated that SUMC-Solver not only outperforms several state-of-the-art models under similar experimental settings but also performs much better under low-resource conditions[1].

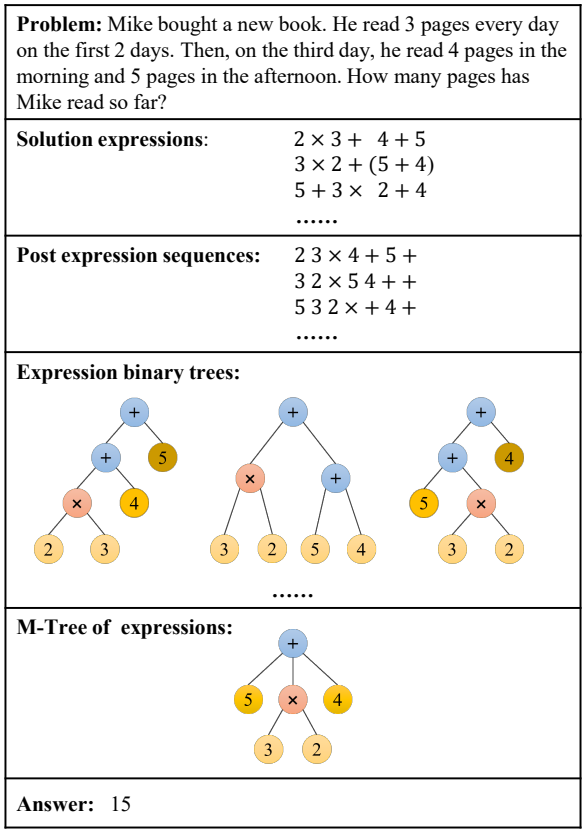

Figure 1: An example of math word problems, which has multiple solution expressions and binary trees, but only one M-tree output.

1 Introduction

Given the description text of a MWP, an automatic solver needs to output an expression for solving the unknown variable asked in the problem that consists of mathematical operands (numerical values) and operation symbols ($+, -, \times, \div$), as shown in Fig. 1. It requires that the solver not only understand the natural-language problem but also be able to model the relationships between the numerical values to perform arithmetic reasoning. These challenges mean MWP solvers are often broadly considered good test beds for evaluating the intelligence level of agents (Lin et al., 2021).

In recent years, the research on designing automatic MWP solvers has also made great progress due to the success of neural network models on NLP. Wang et al. (2017) first apply a Sequence-to-Sequence (seq2seq) model to solve the MWP, and since then more methods (Wang et al., 2018; Chiang and Chen, 2019; Wang et al., 2019) based on seq2seq models have been proposed for further improvement. To use the structural information from expressions more effectively, other methods (Liu

*Corresponding author
[1]Code and data are available at https://github.com/devWangBin/SUMC-Solver

Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 8122 - 8132
December 7-11, 2022

et al., 2019a; Xie and Sun, 2019) use Sequence-to-Tree (seq2tree) models with a well-designed tree-structured decoder to generate the pre-order sequence of a binary tree in a top-down manner and have achieved better performance. Zhang et al. (2020b) combines the merits of the graph-based encoder and tree-based decoder (graph2tree) to generate better solution expressions.

Promising results have been achieved in solving MWP, but the existing methods learn to output only one expression sequence or binary tree, ignoring that there is likely to be far more than one that can obtain the correct answer. For example, in Fig. 1, to answer the question "How many pages has Mike read so far?", we can add up the number of pages that Mike read each day to get the expression "$2 \times 3 + 4 + 5$"; we can also calculate the first two days followed by the sum of pages read on the third day to get the expression "$3 \times 2 + (5 + 4)$"; or we could even calculate the problem in a less obvious way with the expression "$5 + 3 \times 2 + 4$". The number of different expression sequences or binary trees can grow very large with different combinations, which results in a large non-deterministic output space and creates difficulties in model learning. Specifically, when a problem has multiple correct outputs, but the solver only obtains one of them, the knowledge learned by the model will be incomplete, and the demand for data will also increase, making most data-driven methods perform poorly under low-resource conditions.

In previous work, to overcome these limitations, Wang et al. (2018, 2019) used the equation normalization method, which normalizes the output sequence by restricting the order of operands and has only a limited effect. Zhang et al. (2020a) proposed to use multiple decoders to learn different expression sequences simultaneously. However, the large and varying number of sequences for MWPS makes the strategy less adaptable. For the models that learn the binary-tree output (Xie and Sun, 2019; Wu et al., 2020; Zhang et al., 2020b), they generally use a tree decoder to perform top-down and left-to-right generation that only generate one binary tree at a time, which dose not propose a solution to these limitations.

To address the challenge that the output diversity in MWP increases the difficulty of model learning, we analyzed the causes for the diversity, which can be summarized as the following:

- Uncertainty of computation order of the mathematical operations: This is caused by 1) giving the same priority to the same or different mathematical operations. For example, in the expression $n_1 + n_2 + n_3 - n_4$, three operations have the same priority. Consequently, the calculations in any order can obtain the correct answer, which leads to many equivalent expressions and binary trees. And 2) brackets can also lead to many equivalent outputs with different forms. For example, $n_1 + n_2 - n_3$, $n_1 - (n_3 - n_2)$ and $(n_1 + n_2) - n_3$ are equivalent expressions and can be represented as different binary trees.

- The uncertainty caused by the exchange of operands or sub-expressions: Among the four basic mathematical operations $\{+, -, \times, \div\}$, addition "$+$" and multiplication "$\times$" have the property that the operands or sub-expressions of both sides are allowed to be swapped. For example, the expression $n_1 + n_2 \times n_3$ can be transformed to get: $n_1 + n_3 \times n_2$, $n_2 \times n_3 + n_1$, etc.

In this paper, to account for the aforementioned challenge, we propose SUMC-Solver for solving math word problems. The following describes the main contents of our work:

We designed the M-tree to unify the diverse output. Existing work (Xie and Sun, 2019; Wu et al., 2020, 2021b) has demonstrated through extensive experiments that taking advantage of the tree structure information of MWP expressions can achieve better performance. We retain the use of a tree structure but further develop on top of the binary tree with an M-tree which contains any M branches. The ability of the M-tree to unify output structures is reflected in both horizontal and vertical directions:

- To deal with the uncertainty of computation orders for mathematical operations, we set the root to a specific operation and allow any number of branches for internal nodes in the M-tree, reducing the diversity of the tree structure in the vertical direction.

- To deal with the uncertainty caused by the exchange between the left and right sibling nodes in original binary trees, we redefine the operations in the M-tree to make sure that the exchange between any sibling nodes will not affect the calculation process and treat M-trees that differ only in the left-to-right order of their sibling nodes as the same. Like the M-tree example shown in

Fig. 1. The exchange between node "5", "×", and "4" will neither affect the calculation process nor form a new tree. With this method, the structural diversity in the horizontal direction is also reduced.

We designed the M-tree codes and a seq2code framework for the M-tree learning. We abandoned the top-down and left-to-right autoregressive generation used for binary trees in previous methods. The reason is that the generation can not avoid the diversity caused by the generation order of sibling nodes. Instead, we encode the M-tree into M-tree codes that can be restored to the original M-tree, where the codes store the information of the paths from the root to leaf nodes and leaf nodes themselves. And inspired by the sequence labeling methods used in studies mentioned in 2.2, we innovatively use a seq2code framework to generate the M-tree codes in a non-autoregressive way, which takes the problem text as the input sequence and outputs the M-tree codes of the numbers (numerical values) in the math word problem. Then we restore the codes to a M-tree that can represent the calculation logic between the numbers and finally calculate the answer.

Our contributions can be summarized as follows:

- We analyze the causes of output diversity in MWP and design a novel M-tree-based solution to unify the output.
- We design the M-tree codes to represent the M-tree and propose a seq2code model to generate the codes in a non-autoregressive fashion. To the best of our knowledge, this is the first work to analyze mathematical expressions with M-tree codes and seq2code.
- Experimental results on MAWPS (Koncel-Kedziorski et al., 2016) and Math23K datasets (Wang et al., 2017) show that SUMC-Solver outperforms previous methods with similar settings. This is especially the case in low-resource scenarios, where our solver achieves superior performance.

2 Related Work

2.1 Math Word Problem Solver

With the success of deep learning (DL) in various NLP tasks, designing a DL-Based MWP solver has become a major research focus lately. Wang et al. (2017) first addresses the MWP with a seq2seq model, which implements the solver as a generative model from problem text sequence to expression sequence. By utilizing the semantic meanings of operation symbols, Chiang and Chen (2019) apply a stack to help generate expressions. To better utilize expression structure information, other methods (Liu et al., 2019a; Xie and Sun, 2019; Li et al., 2020) transform expressions into binary-tree-structured representations and learn the tree output. Zhang et al. (2020b) additionally introduces a graph-based encoder to enrich the representation of problems.

There are also approaches that explore the use of more extensive networks and external knowledge to help solve the MWP. Li et al. (2019) builds several special attention mechanisms to extract the critical information of the input sequence, and Zhang et al. (2020a) propose using teacher-student networks that combine two solvers to solve the MWP. Wu et al. (2020) utilizes external knowledge through the use of an entity graph extracted from the problem sequence and Lin et al. (2021) proposes a hierarchical encoder with a dependency parsing module and hierarchical attention mechanism to make better use of the input information. Following the previous work, Wu et al. (2021b) continues to use external knowledge to enrich the problem representations and further explicitly incorporate numerical value information encoded by an external network into solving math word problems. Based on the graph2tree framework, Wu et al. (2021a) uses external knowledge to further enrich the input graph information. Yu et al. (2021) uses both a pre-trained knowledge encoder and a hierarchical reasoning encoder to encode the input problems. Qin et al. (2021) constructed different auxiliary tasks using supervised or self-supervised methods and external knowledge (common sense and problem's part-of-speech) to help the model learn the solution of MWPs.

Different from the above methods that mainly focused on the input side and directly generated expression sequences or binary trees, we designed the structure-unified M-tree and the M-tree codes on the output side. Also, we design a simple model to test the advances of the M-tree and M-tree codes by comparing with methods under similar experimental settings, which means that methods using additional knowledge or multiple encoders will not become our baselines.

2.2 Sequence Labeling Parsing

Our method of converting the M-tree into the M-tree codes has similarities with that of sequence labeling parsing, both of which convert complex structural information into a collection of equivalently expressed codes or labels. Constituent parsing is an NLP task where the goal is to obtain the syntactic structure of sentences expressed as a phrase structure tree. As the tree can be represented by a sequence of labels of the input sentence, Gómez-Rodríguez and Vilares (2018) propose transforming constituent parsing into a sequence labeling task and significantly reduce the time required for constituent parsing. Vilares et al. (2019) modify the labeling scheme to avoid certain types of errors. They predict three parts of the label separately to reduce data sparsity and then combine various strategies to alleviate the error transmission problem in the original method. For discontinuous constituent parsing, the experiments (Vilares and Gómez-Rodríguez, 2020) show that despite the architectural simplicity, under the suitable representation, the sequence labeling can also be fast and accurate. Strzyz et al. (2019b) propose using a similar sequence labeling method for dependent parsing, and Strzyz et al. (2019a) combine constituent parsing labeling and dependent parsing labeling with training a multi-task learning model that can perform both parsing tasks with higher accuracy.

In the above work, the labels are used for the classification task, where the output is a one-hot vector, and each token in the input sequence corresponds to a single label. In contrast, our model only learns the codes of the numbers in the input sequence, where the codes are represented as non-one-hot vectors because each number may have multiple codes. Also, these codes cannot be obtained directly from the problem definition, making the design of the M-tree codes challenging.

3 The Design of SUMC-Solver

In this section, we present the design and implementation details regarding our proposed SUMC-Solver, including the problem definition in Section 3.1, the design of the M-tree in Section 3.2, and the detailed description of the M-tree codes and the seq2code model in Section 3.3.

3.1 Problem Definition

A math word problem is represented by a sequence of tokens, where each token can be either a word (e.g., "Mike" and "candies" in Fig. 2) or a numerical value (e.g., "9" and "8"). Some external constants, including 1 and π, which are required to solve the math word problem, but not mentioned in the problem text, are also added to the sequence to get the final input sequence $X = (x_1, x_2, ..., x_n)$. All the numerical values (including added constants) that appear in X are denoted as a set $V = \{v_1, v_2, ..., v_m\}$, and our goal is to generate a set $C = \{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_m\}$, where $\mathbf{c}_i$ is a target code vector for v_i.

3.2 M-Tree

Data Pre-processing For the input sequence, we add additional constants (e.g., 1 and π) that may be used to the front of the input sequence and replace each numerical value v_i with a special symbol. For the given expression in dataset, we try to remove all the brackets of the expression by using the SymPy[2] Python package to prepare for the conversion of expression to the M-tree. For example, $n_1 + (n_2 \pm n_3)$ is converted to $n_1 + n_2 \pm n_3$ and $n_1 \times (n_2 \pm n_3)$ is converted to $n_1 \times n_2 \pm n_1 \times n_3$. For mathematical operations other than $\{+, -, \times, /\}$, such as a^b, we convert it to a product of multiple operands, which allows the M-tree to be extended to solve more complex mathematical problems.

The Design of M-Tree We define the M-tree as follows: M-tree is a tree with only two kinds of nodes: internal nodes and leaf nodes, and each internal node has any M branches, where M is an integer greater than or equal to 1. There are four types of leaf nodes, corresponding to four forms of the numerical value: $\{v, -v, \frac{1}{v}, -\frac{1}{v}\}$, which denote the original value v in the problem X, the opposite of v, the reciprocal of v, and the opposite of the reciprocal of v, respectively. There are four types of internal nodes, corresponding to four redefined operations $\{+, \times, \times-, +/\}$ that ensure sibling nodes are structurally equivalent in the M-tree and two M-trees that differ only in the order of their sibling nodes will be treated as the same. The root of the M-tree is set as a "+" node to unify the structure (operators can have only 1 operand, so $n_1 \times n_2$ will be represented as a unique subtree of the root node). For an internal node that has

[2] https://www.sympy.org/

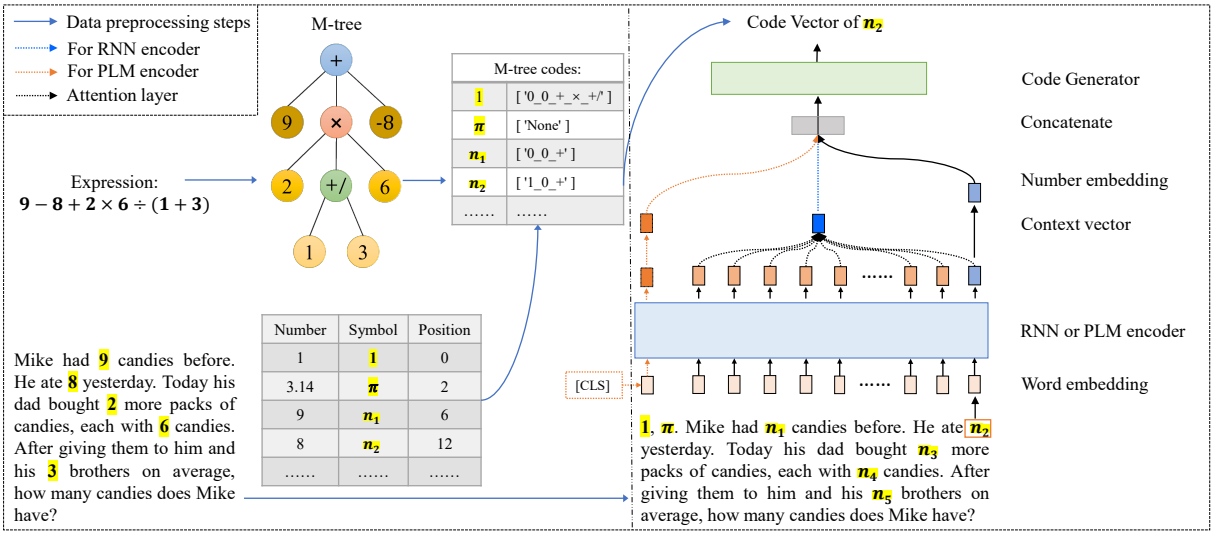

Figure 2: The left half is an example of MWP with the M-tree and M-tree codes, and the right half is the main architecture of our seq2code model (see Section 3.2 and Section 3.3 for more details).

k children $\{v_1, v_2, ..., v_k\}$, where k is an integer greater than or equal to 1:

- The node of "+" ("×") means to sum (multiply) the values of all its child nodes: $v_1 + v_2+, ..., +v_k$ $(v_1 \times v_2\times, ..., \times v_k)$.
- The node of "×−"("+/") means to get the opposite (reciprocal) of the product (sum) value of all its child nodes: $-v_1 \times v_2\times, ..., \times v_k$ $(\frac{1}{v_1+v_2+,...,+v_k})$.

The implementation details of the M-tree are provided in Section A in the Appendix.

3.3 M-Tree Codes and Seq2code Model

3.3.1 The Design of M-Tree Codes

Since the nodes in the M-tree can have any number of branches and sibling nodes are structurally equivalent, autoregressive-based generation cannot avoid the diversity caused by the sequential order of sibling nodes at the output side. To address this challenge, we encode the structure information of the M-tree into each leaf node, forming a mapping between the M-tree and the codes set of leaf nodes so that the model can generate the codes in a non-autoregressive way. Details about M-tree codes are as follows:

Components of M-tree Codes The M-tree code of each leaf node consists of two parts: one part describes the numerical value, and the other part is formed by the path from the root node to the current leaf node. A specific example is shown in Fig. 2. The first part of the code uses two binary bits to distinguish the four forms (mentioned in 3.2) of numerical values. Specifically, for a leaf node in the M-tree represented as $v_i^{'}$, where v_i is the numerical value in the input sequence, the first part of the M-tree code of $v_i^{'}$ will be set according to the following rules:

- If $v_i^{'} = v_i$, the code is set as "0_0";
- If $v_i^{'} = -v_i$, the code is set as "1_0";
- If $v_i^{'} = \frac{1}{v_i}$, the code is set as "0_1";
- If $v_i^{'} = -\frac{1}{v_i}$, the code is set as "1_1";

The second part is set as the sequential operation symbols of all internal nodes on the path from the root to the current leaf node $v_i^{'}$, so leaf nodes with the same parent node will share the same second part code. For example, the second part of the M-tree code of "−8" in the example showing in Fig. 2 is "+", and the code of "1" or "3" is "+_ × _ + /". In some special cases, if the internal nodes that are siblings have the same type (e.g., all "×" nodes), they need to be marked with a special symbol added to the end to distinguish them from each other in order to restore the correct M-tree from the codes.

After converting all M-trees in the training dataset to M-tree codes, a set of M-tree codes will be obtained. The final set of M-tree codes is denoted as $B = \{b_1, b_2, ..., b_l\}$, which has l different codes in total. For example, in the example of Fig. 2, the M-tree code "1_0_+" of "−8" is an element of B.

Vector Representation of M-tree Codes The final code vector c_i for model learning will be obtained based on B. Considering that the value v_i

that appears only once in the input problem text may appear multiple times in the M-tree. For example, in "$v_i \times v_j \pm v_i \times v_k$", v_i will appear in two leaf nodes and have two identical or different M-tree codes. Consequently, the set of numerical values $V = \{v_1, v_2, ..., v_m\}$ is mapping to a set of l-dimensional non-one-hot vectors: $\mathbf{C} = \{\mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_m}\}$, where $\mathbf{c_i}$ is the code vector of the corresponding v_i and the value of $\mathbf{c_i}$ in the k-th dimension indicates how many codes of b_k that v_i has. For example, the final code vector of the value "π" in the example showing in Fig. 2 will be set as $[1, 0, ..., 0]^\top$, where only the first dimension has the value of 1 indicating that "π" has only one M-tree code "None", which means that it does not appear in the M-tree.

Reducing M-tree codes to M-tree The process of converting M-tree to M-tree codes is reversible. Briefly, a code vector is generated for each number in the text and mapped to one or more M-tree codes at first. Then, the number is formatted according to the first part of the M-tree code. Finally, all the numbers are merged by scanning the second part of the M-tree code from back to front, while the M-tree is generated bottom-up.

3.3.2 Sequence-to-Code Model

To verify the advances of the M-tree and M-tree codes, we design a simple seq2code model to tackle the MWP task, which takes the problem sequence as its input and then outputs the corresponding codes (represented as vectors) for numerical values in the problem. After combining all the codes to restore the M-tree, we can calculate the final answer for the problem. Next, we introduce the two core parts of the model: the problem encoder and the code generator.

Problem Encoder We use an encoder to transform the words of a MWP into vector representations. There are two kinds of encoders used in our experiments: a Recurrent Neural Network (RNN) encoder or a pre-trained language model (PLM) encoder.

For the RNN encoder, we use a bidirectional LSTM (BiLSTM) (Hochreiter and Schmidhuber, 1997) network. Formally, given the input sequence $X = (x_1, x_2, ..., x_n)$ and the numerical values set $V = \{v_1, v_2, ..., v_m\}$, we denote the positions of the numerical values as $Q = \{q_1, q_2, ..., q_m\}$, in which q_i is the position of v_i in X. The encoder encodes the input sequence into a sequence of hidden states $\mathbf{H} = \{\mathbf{h}_1^x, \mathbf{h}_2^x, ..., \mathbf{h}_n^x\} \in \mathrm{R}^{n \times 2d}$ as follows:

$$\begin{aligned}
\mathbf{h}_t^x &= \left[\overrightarrow{\mathbf{h}_t^x}, \overleftarrow{\mathbf{h}_t^x}\right], \\
\overrightarrow{\mathbf{h}_t^x}, \overrightarrow{\mathbf{c}_t^x} &= BiLSTM\left(\mathbf{e}_t^x, \overrightarrow{\mathbf{c}_{t-1}^x}, \overrightarrow{\mathbf{h}_{t-1}^x}\right), \\
\overleftarrow{\mathbf{h}_t^x}, \overleftarrow{\mathbf{c}_t^x} &= BiLSTM\left(\mathbf{e}_t^x, \overleftarrow{\mathbf{c}_{t-1}^x}, \overleftarrow{\mathbf{h}_{t-1}^x}\right).
\end{aligned} \tag{1}$$

Where $\mathbf{e}_t^x$ is the word embedding vector for x_t, n is the size of input sequence X, d is the size of the LSTM hidden state, and $\mathbf{h}_t^x$ is the concatenation of the forward and backward hidden states.

And then for the numerical value v_i in the problem X, its semantic representation $\mathbf{e}_i^c$ is modeled by the corresponding BiLSTM output vector:

$$\mathbf{e}_i^c = \mathbf{h}_{q_i}^x. \tag{2}$$

In order to better capture the relationship between different numerical values and the relationship between v_i and the unknown value to be solved (answer of the problem), we use an attention layer to derive a context vector $\mathbf{E}_i$ for v_i, which is expected to summarize the key information of the input problem and help generate the final target code for v_i. The context vector $\mathbf{E}_i$ is calculated as a weighted representation of the source tokens:

$$\mathbf{E}_i = \sum_t \alpha_{it} \mathbf{h}_t^x, \tag{3}$$

where

$$\alpha_{it} = \frac{\exp\left(\mathrm{score}\left(\mathbf{e}_i^c, \mathbf{h}_t^x\right)\right)}{\sum_t \exp\left(\mathrm{score}\left(\mathbf{e}_i^c, \mathbf{h}_t^x\right)\right)}$$

and

$$\mathrm{score}\left(\mathbf{e}_i^c, \mathbf{h}_t^x\right) = \mathbf{U}^\top \tanh\left(\mathbf{W}\left[\mathbf{e}_i^c, \mathbf{h}_t^x\right]\right).$$

where $\mathbf{U}$ and $\mathbf{W}$ are trainable parameters. Finally, we concatenate context vector $\mathbf{E}_i$ and $\mathbf{e}_i^c$ to obtain $\mathbf{z}_i^c$ as the input of the generator:

$$\mathbf{z}_i^c = \left[\mathbf{E}_i, \mathbf{e}_i^c\right]. \tag{4}$$

For the PLM encoder, we use RoBERTa-base (Liu et al., 2019b) or BERT-base (Devlin et al., 2019) to encode the input sequence X to get the token embeddings $Ems = \{em_t^x\}_{t=1}^n$ and get the semantic representation $\mathbf{e}_i^c$ in the same way as the RNN encoder, but for the context vector $\mathbf{e}_i^c$ we use

the output embedding of the special token [CLS] in RoBERTa.

$$\mathbf{e}_i^c = \mathbf{em}_{q_i}^x, \tag{5}$$
$$\mathbf{E}_i = \mathbf{em}_{cls}^x. \tag{6}$$

Code Generator We use a simple three-layer Feedforward Neural Network (FFNN) to implement the generator. With the input $\mathbf{z}_i^c$, the final code vector $\mathbf{c}_i^{'}$ is generated as follows:

$$\begin{aligned} \mathbf{z}_{i1}^c &= \sigma\left(\mathbf{z}_i^{c\top}\mathbf{W}_1 + \mathbf{B}_1\right), \\ \mathbf{z}_{i2}^c &= \sigma\left(\mathbf{z}_{i1}^{c\top}\mathbf{W}_2 + \mathbf{B}_2\right), \\ \mathbf{c}_i^{'} &= \mathbf{z}_{i2}^{c\top}\mathbf{W}_3 + \mathbf{B}_3. \end{aligned} \tag{7}$$

Where σ is an activation function, $\mathbf{W}_i$ and $\mathbf{B}_i$ are the parameters of the FFNN.

Training Objective Given the training dataset $\mathbf{D} = \{(X^i, C^i) : 1 \leq i \leq N\}$, where C^i is the set of all the code vectors corresponding to the numerical values appearing in X^i, we minimize the following loss function:

$$\mathcal{L} = \sum_{(X^i,C^i)\in\mathbf{D}} \sum_{\mathbf{c}_i\in C^i} \mathcal{L}_{MSE}(\mathbf{c}_i, \mathbf{c}_i^{'}), \tag{8}$$

where

$$\mathcal{L}_{MSE}(\mathbf{c}_i, \mathbf{c}_i^{'}) = \frac{1}{l}\sum_{j=1}^{l}\left(\mathbf{c}_{ij} - \mathbf{c}_{ij}^{'}\right)^2. \tag{9}$$

where l is the dimensionality of code vectors.

4 Experiments

4.1 Datasets

We evaluate our SUMC-Solver on two commonly used MWP datasets, MAWPS (Koncel-Kedziorski et al., 2016) with 2,373 problems and Math23K[3] with 23,162 problems. For Math23K, we use the public test set. For MAWPS, we evaluate the performance via five-fold cross-validation and improved the pre-processing method in the previous work (Xie and Sun, 2019; Zhang et al., 2020b) to avoid coarsely filtering out too much data, and the final amount of available data was 2,373 (previously 1,921). We use answer accuracy as the evaluation metric: if the value predicted by the solver equals the true answer, it is thought of as correct.

[3]Available from https://ai.tencent.com/ailab/nlp/dialogue/#Dataset/

4.2 Implementation Details

The parameter settings are as follows: 1) For the RNN encoder, the dimensionality of word embedding and hidden states are 128 and 512, respectively. We select nearly 2500 words that appear most frequently in the training set as the vocabulary and replace the remaining words with a unique token UNK. The global learning rates are initialized to 0.002 for Math23K and 0.008 for MAWPS. 2) For the PLM encoder, we use RoBERTa-base and BERT-base for Math23K and MAWPS, respectively. The initial global learning rate for both datasets is 2×10^{-5}. 3) For the code generator, the dimension of the FFNN is (2048, 1024, $|c_i|$), where c_i is the code vector and its dimensionality is 153 for Math23K and 28 for MAWPS, respectively.

4.3 Compared Methods

Considering SUMC-Solver with one traditional sequence encoder without any other external knowledge as input and one simple generator, we only compare methods with similar settings: **T-RNN** Wang et al. (2019) applied a seq2seq model to predict a tree-structure template, which includes inferred numbers and unknown operators. Then, They used a RNN to obtain unknown operator nodes in a bottom-up manner. **StackDecoder** Chiang and Chen (2019) used the RNN to understand the semantics of problems, and a stack was applied to generate post expressions. **GTS** Xie and Sun (2019) utilized a RNN to encode the input and another RNN to generate the expression based on top-down decomposition and bottom-up subtree embedding. **GTS-PLM** replaces the encoder with a pre-trained language model compared to the original GTS. **SAU-Solver** Qin et al. (2020) devised Universal Expression Trees to handle MWPs with multiple unknowns and equations. Then a RNN encodes the input and a well-designed decoder considering the semantic transformation between equations obtains the expression. **Graph2Tree** (Zhang et al., 2020b) is a graph-to-tree model that leverages an external graph-based encoder to enrich the quantity representations in the problem. **UniLM-Solver** UNIfied Pre-trained Language Model (UniLM) (Dong et al., 2019) have achieved superior performance on natural language understanding and generation tasks, which can be used to model the generation process from the input text to the output expression.

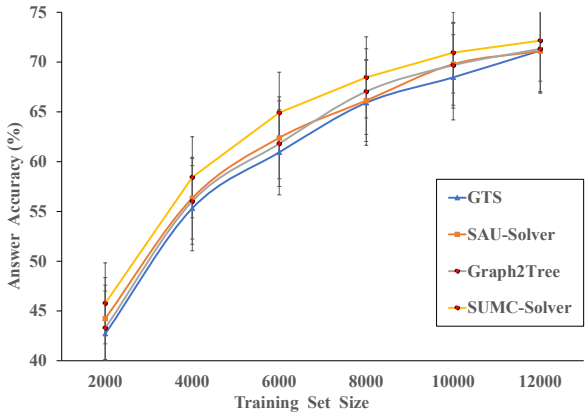

Figure 3: Answer accuracy in different low-resource conditions

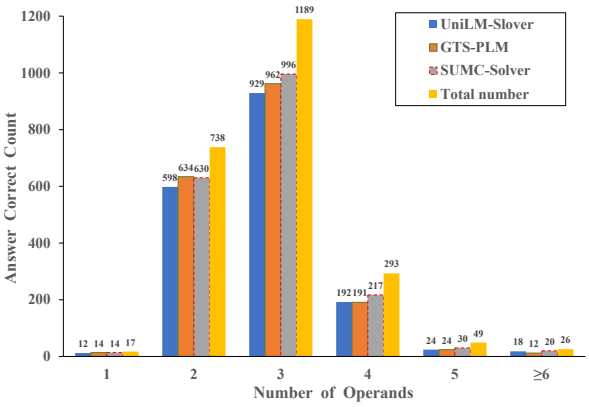

Figure 4: Comparison of the number of problems answered correctly by different models on test data, where the test data are classified according to the number of operands they require.

4.4 Results and Analyses

Answer Accuracy The experiment results are shown in Table 1. We observe that SUMC-Solver outperforms all baselines in the two MWP datasets. When using an RNN as the encoder, SUMC-Solver surpasses StackDecoder and T-RNN that learn the sequence output by 9-10 percent. For methods that learn the binary-tree output, SUMC-Solver also achieves better results than GTS, SAU-Solver and Graph2Tree, although these methods used a well-designed tree decoder or an external graph-based encoder to enrich the representations. When using a PLM as the encoder, SUMC-Solver achieves an accuracy of 82.5%, a significant improvement (3 and 5 percent, respectively) over GTS-PLM and UniLM-Solver. In conclusion, the two different encoder settings above both show that the design of the M-tree and M-tree codes is reasonable and advanced, which allows us to achieve better performance using only a simple seq2code model.

	Model	Math23K	MAWPS*
RNN	T-RNN	66.9	66.8
	StackDecoder	67.8	-
	GTS	75.6	75.2†
	SAU-Solver	76.2†	75.5†
	Graph2Tree	76.6†	78.1†
	SUMC-Solver	**77.4**	**79.9**
PLM	UniLM-Solver	77.5†	78.0†
	GTS-PLM	79.5†	79.8†
	SUMC-Solver	**82.5**	**82.0**

Table 1: Answer accuracy of SUMC-Solver and various baselines. Math23K denotes results on the public test set, MAWPS* denotes 5-fold cross-validation and the results with † are obtained by our reproduction. We reproduced the results for: 1) Getting new results, such as the results of SAU-Solver on the public test set of Math23K and the results of GTS-PLM; 2) Using improved data preprocessing method for MAWPS.

Comparison in Low-resource Situations The annotation cost for MWPs is high, so it is desirable for the model to perform well in lower resource settings. Therefore, we evaluate our model performance with GTS, SAU-Solver and Graph2Tree on training sets of different sizes. The test set contains 2,312 randomly sampled instances. Detailed results can be found in Fig. 3. Tt can be observed that SUMC-Solver consistently outperforms other models irrespective of the size of the training set. Firstly, when the size of the training set is less than 6000, the performance of SAU-Slover is better than that of GTS; when the number exceeds 6000, these two models perform similarly. In terms of overall performance, the results of SAU-Solver and Graph2Tree are better than those of the GTS when resources are constrained. Secondly, with a 6000-sample training set, the most significant performance gap between SUMC-Solver and other models occurs, where our model approximately obtains an additional 5% on accuracy. This shows that SUMC-Solver has the most prominent advantages in low-resource situations.

Performance on Different Numbers of Operands We divide the test set (2,312 randomly sampled instances) into different levels according to the number of operands (numerical values in problems) required to calculate the answer and evaluate the model performance on these different data. The

Codes	Code set size	Test set coverage (%)
M-tree	153	100.0
Binary-tree	1290	93.5

Table 2: Comparison of Binary-Tree and M-Tree Codes.

details are shown in Fig. 4. From the results, we can see that most of the MWPs require between 2 and 4 operands, and SUMC-Slover performs better than the baseline models on data requiring more operands, which shows that our solver has the potential to solve more complex problems.

Comparison of Binary-Tree and M-Tree Codes The seq2code framework can also be applied to the binary-tree structure if choosing one binary tree for each MWP and converting it to the codes in the same way. We transformed the data of Math23K's training set and compared the binary-tree codes and M-tree codes, which is shown in the Table 2. It can be observed that applying the M-tree structure can greatly reduce the size of the code set and ensure that the obtained codes can cover the data in the test set, which shows the effect of the M-tree on unifying the output structure is very significant.

5 Conclusion

In this paper, we proposed SUMC-Slover to solve math word problems, which applies the M-tree to unify the diverse output and the seq2code model to learn the M-tree. The experimental results on the widely used MAWPS and Math23K datasets demonstrated that SUMC-Solver outperforms several state-of-the-art models under similar settings and performs much better under low-resource conditions.

Limitations

Some discussions on the limitations of SUMC-Solver are as follows: 1) The M-tree corresponding to the output of each MWP is unique. However, as mentioned in Section 3.3.1, some special M-trees need to be distinguished by introducing special symbols randomly when converting them into M-tree codes, which makes the M-tree codes correspond to the MWP may not be unique. Through the statistics of the datasets, we found that about 90% of the data do not belong to this particular case. At the same time, for the remaining 10%, despite the increased difficulty, they are still learnable based on previous work experience, which makes SUMC-Solver still achieve a significant performance improvement. 2) The network structure is relatively simple for the seq2code framework used in SUMC-Solver. In previous work, the use of graph-based encoders and the introduction of external knowledge to enrich the representation of the input problem text have been shown to further improve the performance of the solver, and seq2code can be naturally integrated with these improved approaches to try to achieve better results.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments. This work was supported by the National Natural Science Foundation of China (No. 61936012 and 61976114).

References

Ting-Rui Chiang and Yun-Nung Chen. 2019. Semantically-aligned equation generation for solving and reasoning math word problems. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 2656–2668. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13042–13054.

Carlos Gómez-Rodríguez and David Vilares. 2018. Constituent parsing as sequence labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 1314–1324. Association for Computational Linguistics.

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

Rik Koncel-Kedziorski, Subhro Roy, Aida Amini, Nate Kushman, and Hannaneh Hajishirzi. 2016. MAWPS: A math word problem repository. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pages 1152–1157. The Association for Computational Linguistics.

Jierui Li, Lei Wang, Jipeng Zhang, Yan Wang, Bing Tian Dai, and Dongxiang Zhang. 2019. Modeling intra-relation in math word problems with different functional multi-head attentions. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 6162–6167. Association for Computational Linguistics.

Shucheng Li, Lingfei Wu, Shiwei Feng, Fangli Xu, Fengyuan Xu, and Sheng Zhong. 2020. Graph-to-tree neural networks for learning structured input-output translation with applications to semantic parsing and math word problem. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 2841–2852. Association for Computational Linguistics.

Xin Lin, Zhenya Huang, Hongke Zhao, Enhong Chen, Qi Liu, Hao Wang, and Shijin Wang. 2021. HMS: A hierarchical solver with dependency-enhanced understanding for math word problem. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 4232–4240. AAAI Press.

Qianying Liu, Wenyv Guan, Sujian Li, and Daisuke Kawahara. 2019a. Tree-structured decoding for solving math word problems. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 2370–2379. Association for Computational Linguistics.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019b. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Jinghui Qin, Xiaodan Liang, Yining Hong, Jianheng Tang, and Liang Lin. 2021. Neural-symbolic solver for math word problems with auxiliary tasks. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 5870–5881. Association for Computational Linguistics.

Jinghui Qin, Lihui Lin, Xiaodan Liang, Rumin Zhang, and Liang Lin. 2020. Semantically-aligned universal tree-structured solver for math word problems. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 3780–3789. Association for Computational Linguistics.

Michalina Strzyz, David Vilares, and Carlos Gómez-Rodríguez. 2019a. Sequence labeling parsing by learning across representations. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 5350–5357. Association for Computational Linguistics.

Michalina Strzyz, David Vilares, and Carlos Gómez-Rodríguez. 2019b. Viable dependency parsing as sequence labeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 717–723. Association for Computational Linguistics.

David Vilares, Mostafa Abdou, and Anders Søgaard. 2019. Better, faster, stronger sequence tagging constituent parsers. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 3372–3383. Association for Computational Linguistics.

David Vilares and Carlos Gómez-Rodríguez. 2020. Discontinuous constituent parsing as sequence labeling. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 2771–2785. Association for Computational Linguistics.

Lei Wang, Yan Wang, Deng Cai, Dongxiang Zhang, and Xiaojiang Liu. 2018. Translating a math word problem to an expression tree. *CoRR*, abs/1811.05632.

Lei Wang, Dongxiang Zhang, Jipeng Zhang, Xing Xu, Lianli Gao, Bing Tian Dai, and Heng Tao Shen. 2019. Template-based math word problem solvers with recursive neural networks. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 7144–7151. AAAI Press.

Yan Wang, Xiaojiang Liu, and Shuming Shi. 2017. Deep neural solver for math word problems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 845–854. Association for Computational Linguistics.

Qinzhuo Wu, Qi Zhang, Jinlan Fu, and Xuanjing Huang. 2020. A knowledge-aware sequence-to-tree network for math word problem solving. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 7137–7146. Association for Computational Linguistics.

Qinzhuo Wu, Qi Zhang, and Zhongyu Wei. 2021a. An edge-enhanced hierarchical graph-to-tree network for math word problem solving. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 1473–1482. Association for Computational Linguistics.

Qinzhuo Wu, Qi Zhang, Zhongyu Wei, and Xuanjing Huang. 2021b. Math word problem solving with explicit numerical values. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 5859–5869. Association for Computational Linguistics.

Zhipeng Xie and Shichao Sun. 2019. A goal-driven tree-structured neural model for math word problems. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 5299–5305. ijcai.org.

Weijiang Yu, Yingpeng Wen, Fudan Zheng, and Nong Xiao. 2021. Improving math word problems with pre-trained knowledge and hierarchical reasoning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 3384–3394. Association for Computational Linguistics.

Jipeng Zhang, Roy Ka-Wei Lee, Ee-Peng Lim, Wei Qin, Lei Wang, Jie Shao, and Qianru Sun. 2020a. Teacher-student networks with multiple decoders for solving math word problem. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4011–4017. ijcai.org.

Jipeng Zhang, Lei Wang, Roy Ka-Wei Lee, Yi Bin, Yan Wang, Jie Shao, and Ee-Peng Lim. 2020b. Graph-to-tree learning for solving math word problems. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 3928–3937. Association for Computational Linguistics.

A Implementation Details of the M-tree

After the data pre-processing for expression mentioned in 3.2, We can easily convert it into a M-tree based on the following steps:

(1) By following the order of priority for operations: (operations in brackets) $>$ $(\times = \div)$ $>$ $(+ = -)$, Converting the operations one-by-one in the expression as follows:

1. For $v_1 \div v_2$, it is converted to $v_1 \times v_2^{'}$, where $v_2^{'}$ is the reciprocal of v_2.

2. For $v_1 - v_2$, it is converted to $v_1 + v_2^{'}$, where $v_2^{'}$ is the opposite of v_2.

3. For $v_1 - v_2 \times v_3$, it is converted to $v_1 + v_2(\times -)v_3$, where $v_2(\times -)v_3$ means the opposite of $v_2 \times v_3$.

4. For $v_1 \div (v_2 + v_3)$, it is converted to $v_1 \times v_2(+/)v_3$, where $v_2(+/)v_3$ means the reciprocal of $v_2 + v_3$.

After the conversion, only four operations we defined in the M-tree will be left in the new expression, and they all have the property that the computation is not affected by the left-right order between child nodes, which can be used to reduce the structural diversity in the horizontal direction.

(2) After obtaining the new expression, we convert it to a binary tree and then reduce it from top to bottom to get the final M-tree. Let the parent node be v_p and the child node be v_c, and the details are as follows:

1. If it is one of the 4 cases: 1) "$v_p = v_c = +$", 2) "$v_p = v_c = \times$", 3) "$v_p = +/$ and $v_c = +$", 4) "$v_p = \times -$ and $v_c = \times$", then merge directly, delete the child node v_c and assign its children (if has) to v_p and continue checking down.

2. If "$v_p = \times$ and $v_c = \times -$", then make "$v_p = \times -$" and do the same as 1.

3. If "$v_p = \times -$ and $v_c = \times -$", then make "$v_p = \times$" and do the same as 1.

After merging the nodes from top to bottom, the height of the tree will be minimized, and the tree structure will be unified in the vertical direction. And we obtain a structure-unified M-Tree for the origin solution expression.