MILE: MEMORY-INTERACTIVE LEARNING ENGINE FOR SOLVING MATHEMATICAL PROBLEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mathematical problem solving is a task that examines the capacity of machine learning models for performing quantitative and logical reasoning. Existing work employed formulas as intermediate labels in this task to formulate the computing and reasoning processes and achieved remarkable performance. However, we are questioning the limitations of existing methods from two perspectives: the expressive capacity of formulas and the learning capacity of existing models. In this work, we proposed Memory-Interactive Learning Engine (MILE), a new framework for solving mathematical problems. Our main contribution in this work includes a new formula representing technique and a new decoding method. In our experiment on Math23K dataset, MILE outperformed existing methods on not only question answering accuracy but also robustness and generalization capacity. [1]

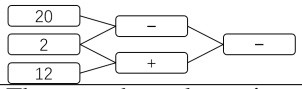
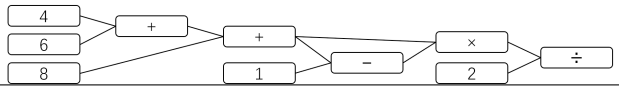
1 INTRODUCTION

With the rapid development of deep learning in recent years, super-human level performance has been achieved in more and more fields starting with computer vision and natural language processing. Not satisfied by these achievements, researchers have also begun to question the capacity of machine learning models for performing abstract and logical reasoning. Mathematical reasoning is a field quite suitable for this study because of its high-level demand for analytical thinking. In this field, early work tried to solve mathematical problems with language models by performing end-to-end sequence-to-sequence prediction from question texts to answers. However, language models such as LSTM (Hochreiter & Schmidhuber, 1997) and Transformer (Vaswani et al., 2017) did not show desirable performance when there came problems that required relatively complicated computations such as finding the greatest common divisor (gcd) and base conversion (Saxton et al., 2019). Another approach to solving mathematical problems is to make use of some intermediate labels known as formulas, templates, or equations (Wang et al., 2017; 2019; Liu et al., 2019; Xie & Sun, 2019; Zhang et al., 2020). Some examples for mathematical problems and the formulas are presented in Table 1. Under this approach, machine learning models are only expected to predict formulas from questions, and these formulas can be calculated in a rule-based manner to acquire the final answers. These formulas endow machine learning models with an outstanding capacity for formulating and organizing abstract computing processes. Recent work also benefited from this practice to achieve remarkable performance on solving mathematical problems.

However, in our study, we are also alert to some limitations exposed in existing work. The first defect we are concerned about is the expressive capacity of existing formula representations. Most existing work represents formulas with infix notation or prefix notation, while these representations are under the assumption that the computations are tree-structured with algebraic operations at non-leaf nodes and raw numbers at leaf nodes. However, these tree-structured computations are not competent for solving all mathematical problems, and a simple exception is finding the gcd. The underlying reason for this defect is that tree-structured algebraic computation is not Turing-complete, and this representation does not truly support variables that can be referenced multiple times. This limits the expressive capacity of the formulas. Besides, another issue that raised our concerns throughout our study is that it seems that the existing formula predicting models may not truly understand the underlying concepts and natures of the formulas. Their behavior is more similar to merely fitting

[1] We will release the software after this paper is published.

Table 1: Two examples of mathematical problems together with their corresponding formulas and computing graph. Note that the number and constant tokens in the formulas have been replaced with corresponding values for better readability. In the second example, prefix notation is not applicable because the computation is not tree-structured.

question	Andy has 12 apples. Bob has 20 apples. Bob gives 2 apples to Andy. How many more apples does Bob have than Andy now?
answer	4
f_p	[$-$, $-$, 20, 2, $+$, 12, 2]
f_s	[($-$, 20, 2), ($+$, 12, 2), ($-$, M_0, M_1)]
computing graph	
question	There are three classes in a school. Class one has 4 students. Class two has 6 students. Class three has 8 students. If every student shakes hands with each other, how many times of handshakes happen in total?
answer	153
f_p	N/A
f_s	[($+$, 4, 6), ($+$, M_0, 8), ($-$, M_1, 1), ($\times$, M_1, M_2), ($\div$, M_3, 2)]
computing graph	

the formulas they saw in the training stage. Some evidence for this proposition is uncovered in our experiment, and more discussion is going to be made in Section 5.

With an eye to these limitations of existing work, in this work, we are motivated to put forward a new paradigm for formula representation and prediction for solving mathematical problems. We first introduced a new approach for formula representation named segmented representation, and then proposed Memory-Interactive Learning Engine (MILE), a new framework for formula learning that cooperates with our segmented representation. Our major contributions in this work can be summarized as follows:

- We introduced a new formula representing technique that supports non-tree-structured formulas.
- We proposed a new decoding method that is compatible with this new formula representation and outperforms the formula predicting models developed in existing work.
- We illustrated that our proposed method is more robust and has better generalization capacity than existing approaches.

2 Related Work

2.1 Mathematical Reasoning Tasks

Mathematical reasoning is a field that examines the capacity of machine learning models for performing abstract, quantitative, and logical reasoning on mathematical problems. There are various specific tasks within this field focusing on different reasoning abilities, while the solution to mathematical problems described in natural languages is most widely studied in recent years, and many relevant datasets have been published. Math23K (Wang et al., 2017) is a dataset crawled from a couple of online education websites consisting of 23,162 mathematical problems with formula annotations. MathQA (Amini et al., 2019) is a dataset consisting of 37,200 mathematical problems collected from another former dataset AQuA (Ling et al., 2017). However, the formula annotations for MathQA is imperfect and contains noise (Chen et al., 2020; Wu & Nakayama, 2022). This influenced the effectiveness of formula learning. Mathematics (Saxton et al., 2019) is a comprehensive dataset providing problems generated in various mathematical areas including polynomial and calculus. However, this dataset is proposed to study the end-to-end reasoning capacity of language models, so no formula annotations are involved. GSM8K (Cobbe et al., 2021) is a dataset that contains 8,792 mathematical problems together with their solutions in natural languages. This dataset has been used to study the chain-of-thought approach for solving mathematical problems in recent work (Chowdhery et al., 2022; Wei et al., 2022).

2.2 MATHEMATICAL PROBLEM SOLVERS

Generally, the existing approaches for solving mathematical problems can be categorized into three families: end-to-end methods, formula-based methods, and chain-of-thought methods. The characteristics of them are as follows.

As a basic and naive approach, end-to-end methods simply regard both the questions and answers as sequences of alphabets, digits, and symbols, and conduct a sequence-to-sequence prediction (Saxton et al., 2019). These methods are easier to be implemented and can be trained in an end-to-end manner. However, the performance achieved by these methods on some problem categories requiring relatively complicated computations, such as finding the gcd and base conversion, is not satisfying. This exposed the insufficiency of language models in performing end-to-end logical reasoning.

On the other hand, formula-based methods employ formulas to formulate the computing and reasoning processes. Under this methodology, language models are only responsible for predicting formulas from questions. These formulas are then calculated under pre-defined rules to acquire the final answers. This approach was first proposed by Wang et al. (2017) and then get improved by later work (Wang et al., 2019; Liu et al., 2019; Xie & Sun, 2019; Zhang et al., 2020). Formula-based methods have achieved remarkable performance on challenging mathematical reasoning dataset represented by Math23K.

Chain-of-thought methods are another novel approach that devotes to solving mathematical problems step-by-step. These methods acquire solutions to problems by generating each of its intermediate reasoning steps with language models continuously. Recent work succeeded in implementing this method by fine-tuning pretrained language models or performing few-shot prompting (Cobbe et al., 2021; Chowdhery et al., 2022; Wei et al., 2022). Some work has also shown that language models pretrained on a huge corpus can even be zero-shot learner (Shwartz et al., 2020; Reynolds & McDonell, 2021; Kojima et al., 2022). However, the performance of these methods still may not exceed specialized models working on predicting formulas and performing precise rule-based calculations. As a result, we still consider formula-based approaches valuable for this task.

2.3 NEURAL NETWORKS WITH EXTERNAL MEMORY

Neural Networks with external memory, or what is known as Neural Turing Machine, is an approach that endows Recurrent Neural Networks with the ability to write to or read from external memory (Graves et al., 2014; 2016). This approach is proposed with the motivation to strengthen the capacity of models for maintaining long-term information. The basic architecture of these models can be described as a recurrent controller network along with an external memory space, which inspired our idea of introducing memory mechanisms to MILE.

3 FORMULA REPRESENTATION

In formula-based methods, given a question denoted by q, the machine learning model is expected to predict a formula f for q. The formula f can be formulated in multiple different fashions, while the infix notation and prefix notation are most commonly adopted in existing work Wang et al. (2017; 2019). In the following discussion, we take prefix notation as an example, while infix notation shares a similar idea. We let f_p denote the formula f represented in prefix notation to distinguish it from our representations. Generally, f_p is made up of a sequence of tokens [t_0, ..., t_n]. Among them, each token t_i is an element from the union of three sets: T_{op}, T_{num}, T_{con}. Here, T_{op} is the set of operator tokens, of which the elements are relevant to specific tasks and datasets. A simple and typical example of T_{op} is the set of the four basic arithmetic operations $\{+, -, \times, \div\}$. T_{num} is the set of number tokens $\{N_0, ..., N_m\}$ that establish reference to the numbers appearing in the question text. During data preprocessing, the numbers in the question text are extracted as [n_0, ..., n_m]. In the calculation of formulas, N_i will be replaced by the i-th extracted number n_i. T_{con} is the set of constant tokens that refer to the constants that are crucial for solving some problems but may not explicitly appear in the question text. The composition of T_{con} is also relevent to specific tasks and datasets. An example can be $\{1, \pi, 60, 100\}$.

Nevertheless, as we indicated above, this prefix notation f_p can only be utilized to represent tree-structured computations, which limited the expressive capacity of formulas. To address this problem,

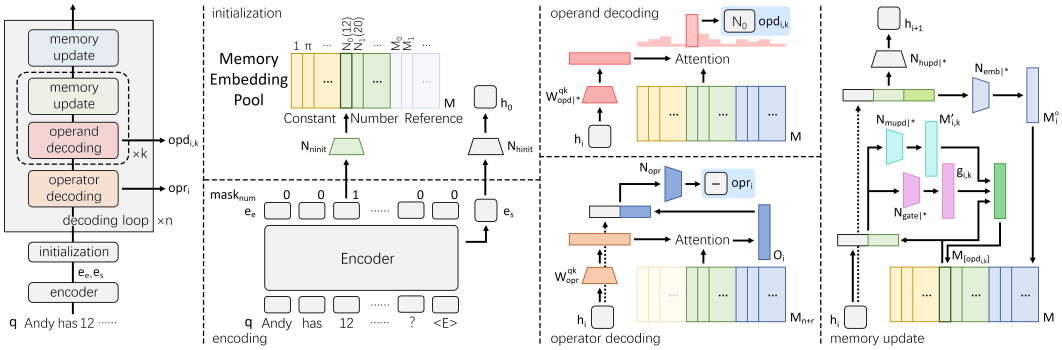


Figure 1: The overview of our Memory-Interactive Learning Engine.

we proposed the segmented representation f_s, a new practice for formula representation that is not restricted to tree-sturctured computations.

Generally, the basic form of this representation is a sequence of terms [S_0, ..., S_n]. Among them, each term S_i contains one operator and a certain number of operands. That is, with opr denotes the operator and opd denotes the operands, $S_i = (opr, opd_1, ..., opd_k)$. Here, the number of operands k is specific to the operator. For example, arithmetic operations take two operands and the logarithm operation takes one. Compared to the prefix notation, the operator opr in f_s is generated from the same operator set T_{op}, while the operand opd is generated from the union of T_{num}, T_{con}, and another particular set T_{ref}. Here, $T_{ref} = \{M_0, ..., M_{n-1}\}$ is the set of reference tokens that establish reference to the intermediate computing results of the previous terms. By the time of calculation, each term of f_s is calculated in order from S_0 to S_n, while the last term S_n raises the final result of the whole formula.

4 Memory-Interactive Learning Engine

Generally, MILE follows the encoder–decoder architecture as shown in Figure 1. Its computation can be formulated with Equations 1 and 2.

$$e_e, e_s = Encoder(q) \tag{1}$$

$$f = Decoder(e_e, e_s) \tag{2}$$

MILE does not have selectivity to the encoder, as long as it encodes the question q to two tensors: encoder embedding e_e and encoder summary e_s. Here, $e_e \in \mathbb{R}^{L \times d_e}$, where L denotes the input sequence length and d_e denotes the embedding dimension. $e_s \in \mathbb{R}^{d_s}$, where d_s denotes the summary dimension. Due to its compatibility, MILE can work on any common encoders such as LSTM (Hochreiter & Schmidhuber, 1997), BERT (Devlin et al., 2019), or GNN (Scarselli et al., 2008; Zhang et al., 2020).

4.1 Memory Embedding Pool

As its name indicates, the most distinctive design of MILE is a latent space of memory that we call Memory Embedding Pool. This design is inspired by existing work that extended Recurrent Neural Networks with external memory (Graves et al., 2016) and implemented Neural Turing Machine (Graves et al., 2014). Following a similar principle, our Memory Embedding Pool is developed to assist the model to maintain long-term information better.

Generally, our Memory Embedding Pool is composed of three types of different embeddings corresponding to the three types of available operands: constant embedding, number embedding, and reference embedding. It established a unified embedding space for all these available operands, of which the underlying consideration is that these operands should share equal status throughout the decoding even though their origins are different. With M denotes the Memory Embedding Pool, $M \in \mathbb{R}^{[W_c+W_n+W_r] \times d_m}$. Here, d_m denotes the memory embedding dimension. W_c, W_n, and W_r are the length of T_{con}, T_{num}, and T_{ref}, respectively. They are also referred to as the widths of memory embeddings. We generated the three types of embedding under the following principles.

Constant Embedding We simply treat constant embeddings as trainable embedding vectors shared across all questions. That is, with M_c denotes the constant embedding, $M_c \in \mathbb{R}^{W_c \times H_m}$ is directly a parameter to be optimized in our model.

Number Embedding We generate the number embeddings specific to each question basing on its encoder embedding e_e. They are calculated after the encoding is finished and before the decoding is started. Details are presented in the *Initialization* paragraph in Section 4.2.

Reference Embedding We generate the reference embeddings continuously after the decoding of each formula term. That is, its width W_r starts with zero and grows as the decoding goes on. Details are presented in the *Memory Update* paragraph in Section 4.2.

4.2 Formula Decoding

As shown in Figure 1, the decoding of a formula is a recurrent process with each formula term S_i = (opr_i, $opd_{i,1}$, ..., $opd_{i,k}$) as basic decoding unit. In this process, each decoding loop contains three subprocedures: operator decoding, operand decoding, and memory update. Their working principles are presented as follows.

Initialization We first initialize the hidden state $h \in \mathbb{R}^{d_h}$ and number embedding $M_n \in \mathbb{R}^{W_n \times d_m}$ at the beginning of decoding. Their calculations are described by Equations 3 to 6.

$$h_0 = N_{hinit}(e_s) \tag{3}$$

$$mask_{num} = [\mathbf{1}\ if\ \text{is\_num}(t_i)\ else\ \mathbf{0}\ for\ t_i\ in\ q] \tag{4}$$

$$e_{e|num} = e_e.\text{gather}(mask_{num}) \tag{5}$$

$$M_n = N_{ninit}(e_{e|num}) \tag{6}$$

Here, N_{hinit} and N_{ninit} are multilayer perceptrons (MLPs). $\text{is\_num}()$ judges whether a token t_i in q is a number token. $\text{gather}()$ gathers the embedding vectors on number tokens along the axis of sequence length so that $e_{e|num} \in \mathbb{R}^{W_n \times d_e}$. Note that $e_{e|num}$ is padded to W_n on the axis of sequence length for the convenience of following calculations, even though there can be less than W_n number tokens in some input sequences.

Operator Decoding The decoding process of operators is described by Equations 7 to 9. Here, M_{n+r} is the variable part of the memory embedding, which contains the number embedding and reference embedding and starts from the end of the constant embedding. $O_i \in \mathbb{R}^{d_m}$ is an observation across M_{n+r} and is obtained simply with the attention mechanism, where $W^{qk}_{opr} \in \mathbb{R}^{d_h \times d_m}$ denote the product of query and key matrixes. At last, O_i is concatenated to h_i to be passed through a MLP classifier N_{opr} to predict the operator opr_i. For short, We abbreviate opr_i to $\hat{o}$ in the following discussion.

$$M_{n+r} = M_{[W_c:]} \tag{7}$$

$$\begin{aligned} O_i &= \text{Attention}(h_i, M_{n+r}) \\ &= softmax(h_i W^{qk}_{opr} M^\top_{n+r}) M_{n+r} \end{aligned} \tag{8}$$

$$opr_i = \text{argmax}_{T_{op}} N_{opr}([h_i, O_i]) \tag{9}$$

Operand Decoding The decoding process of operands is described by Equations 10 and 11. Note that this process is repeated k times in each decoding loop because each operator is associated with k operands, while for most arithmetic operations $k = 2$.

$$P_{i,k} = softmax(h_i W^{qk}_{opd|\hat{o},k} M^\top) \tag{10}$$

$$opd_{i,k} = \text{argmax}_{T_{con} \cup T_{num} \cup T_{ref}} P_{i,k} \tag{11}$$

Here, $P_{i,k}$ is the likelihood on each memory embedding terms of being the desirable k-th operand for operator $\hat{o}$. Again, $W^{qk}_{opd|\hat{o},k} \in \mathbb{R}^{d_h \times d_m}$ denote the product of query and key matrices. However, unlike W^{qk}_{opr}, which is a single parameter shared throughout the whole decoding process, there is a set of $W^{qk}_{opd|*} = \{W^{qk}_{opd|opr_0,1}, ..., W^{qk}_{opd|opr_n,k}\}$ corresponding to each operand position of each

operator. An exception here is that for the commutative operations like addition and multiplication, a single $W^{qk}_{opd|*}$ is shared by the symmetric operands so that $W^{qk}_{opd|+,1} \equiv W^{qk}_{opd|+,2}$. This design helps the model to better learn the underlying concepts of mathematical operations and prevent overfitting.

Memory Update After the operator and operands in each formula term S_i is decoded, we update the Memory Embedding Pool by two means. First, we update the embeddings of the memory terms that are selected as operands. Then, we generate an embedding for the current formula term S_i and append it to the reference embedding. The memory update for operands follows Equations 12 to 14.

$$M'_{i,k} = N_{mupd|\hat{o},k}([h_i, M_{[opd_{i,k}]}]) \tag{12}$$

$$g_{i,k} = N_{gate|\hat{o},k}([h_i, M_{[opd_{i,k}]}]) \tag{13}$$

$$M_{[opd_{i,k}]} = g_{i,k} M'_{i,k} + (1 - g_{i,k}) M_{[opd_{i,k}]} \tag{14}$$

Here, M is updated solely on its $opd_{i,k}$ slice $M_{[opd_{i,k}]}$. Its update follows a gate mechanism similar to the one in GRU Chung et al. (2014). $N_{mupd|\hat{o},k}$ and $N_{gate|\hat{o},k}$ are the MLPs that calculate the update and gate vectors, respectively, with the original $M_{[opd_{i,k}]}$ concatenated to h_i as inputs. Similar to $W^{qk}_{opd|*}$, there are also two sets of $N_{mupd|*}$ and $N_{gate|*}$ corresponding to each operand position of each operator, and the ones for symmetric operands are shared.

The generation of the new memory embedding for S_i follows Equations 15 and 16.

$$M^{\circ}_i = N_{emb|\hat{o}}([h_i, M_{[opd_{i,1}]}, ..., M_{[opd_{i,k}]}]) \tag{15}$$

$$M = [M, M^{\circ}_i] \tag{16}$$

Here, $N_{emb|\hat{o}}$ is the MLP coming from set $N_{emb|*}$ that calculates the new embedding vector M°_i. It takes as input the concatenation of h_i and the k associated original memory embedding $M_{[opd_{i,1}]}$ to $M_{[opd_{i,k}]}$. M°_i is concatenated to M then, which makes the width of reference embedding W_r grow by one in each decoding loop. At last, the hidden state is updated following Equation 17.

$$h_{i+1} = N_{hupd|\hat{o}}([h_i, M_{[opd_{i,1}]}, ..., M_{[opd_{i,k}]}]) \tag{17}$$

As a summary, all the trainable parameters and modules in the decoding side of MILE include M_c, N_{hinit}, N_{ninit}, W^{qk}_{opr}, N_{opr}, $W^{qk}_{opd|*}$, $N_{mupd|*}$, $N_{gate|*}$, $N_{emb|*}$, and $N_{hupd|*}$.

4.3 Formula Mutation

Considering that the decoding mechanism of MILE is relatively more complex than that in existing approaches like RNN decoder and Tree decoder Zhang et al. (2020) and MILE has more parameters, the training of MILE is more challenging. Moreover, due to the cost of crafting formula annotations, most publically available mathematical reasoning datasets with formula annotations only contain from several thousands to tens of thousands of training samples. In our experiment, we also found that such a limited amount of data may not satisfy the training demand on MILE. As a result, we proposed formula mutation, a data augmentation approach to expand the scale of training data.

The idea of formula mutation is quite straightforward—generating functionally equivalent formulas. This can also be easily implemented on formulas in segmented representation, for which we just need to swap the symmetric operands in each formula term performing commutative operations. To be more formally, with $f_s = [S_0, ..., S_n]$, $S_i = (opr_i, opd_{i,1}, ..., opd_{i,k})$, $T_{op|c}$ denotes the set of commutative operators, and $\prod$ performs the Cartesian product, the mutation on f_s is given by Equations 18 and 19. An example of the mutation result on a formula with commutative operations is provided in Appendix A.

$$S'_i = \begin{cases} \{S_i\}, & \text{if } opr_i \notin T_{op|c} \\ \{(opr_i, opd_{i,1}, opd_{i,2}), (opr_i, opd_{i,2}, opd_{i,1})\}, & \text{if } opr_i \in T_{op|c} \\ \qquad \text{(in this case } k = 2 \text{ for certain)} \end{cases} \tag{18}$$

$$\text{Mut}(f_s) = \prod_{i=0}^{n} S'_i \tag{19}$$

Table 2: Question answering accuracy of baselines and MILE on Math23K dataset under 5-fold cross validation and on a public test set.

	5-fold	public
DNS (Wang et al., 2017)	58.1%	-
T-RNN (Wang et al., 2019)	66.9%	-
TreeDecoder (Liu et al., 2019)	-	69.0%
GTS (Xie & Sun, 2019)	74.3%	75.6%
Graph2Tree (Zhang et al., 2020)	75.5%	77.4%
LSTM / LSTM	65.2%	65.9%
LSTM / Tree	**68.9%**	**69.7%**
MILE (LSTM)	65.3%	66.1%
Graph / LSTM	71.2%	72.8%
Graph / Tree	**75.1%**	**77.2%**
MILE (Graph)	74.9%	77.0%
BERT / LSTM	78.0%	79.5%
BERT / Tree	81.7%	82.9%
MILE (BERT)	**82.5%**	**83.6%**

Formula mutation is utilized by default in MILE, while it can also be applied to other decoding methods by transforming the formula in segmented representation f_s back to prefix notation f_p. We conducted contrast experiments on the influence of formula mutation and report our interesting findings in Section 5.2.

5 EXPERIMENTS

In our experiments, we studied the performance of MILE and compared MILE with existing decoding methods on the Math23K dataset (Wang et al., 2017). We selected Math23K as the dataset for our experiments because it is the largest officially published mathematical reasoning dataset with complete formula annotation. Math23K is also the dataset on which most existing work reported their result, which allows convenient comparisons. We show the experimental setup and primary result in Section 5.1 and 5.2, and present our findings in further studies in Section 5.3 and 5.4.

5.1 EXPERIMENTAL SETUP

As we indicated above, MILE has an encoder–decoder architecture. For the encoder, we compared the LSTM encoder Hochreiter & Schmidhuber (1997), the graph encoder introduced by Zhang et al. (2020), and BERT Devlin et al. (2019). For the decoder, we compared our method to the LSTM decoder and the tree decoder introduced by Zhang et al. (2020). Note that MILE predicts formulas in segmented representation f_s while other decoders predict formulas in prefix notation f_p. The implementation details are presented in Appendix B. We evaluate the performance of models with the accuracy of final answers acquired from predicted formulas. Note that this accuracy is different from and is not directly comparable with the formula matching accuracy applied in some existing work, which does not tolerate the functionally equivalent formulas that lead to the same answers.

5.2 PRIMARY RESULT

Table 2 shows the question answering accuracy of baselines and MILE on Math23K dataset under 5-fold cross validation and on a public test set. Here, DNS is the original RNN-based learning model proposed by Wang et al. (2017). T-RNN (Wang et al., 2019) predicts equation templates in tree structure with recursive neural networks. TreeDecoder (Liu et al., 2019) implements a tree-structured decoder to predict prefix templates. GTS (Xie & Sun, 2019) employs tree-structured neural networks to predict expression trees in a goal-driven manner. Graph2Tree (Zhang et al., 2020) introduces a novel approach with graph-based encoder and tree-based decoder to further improve the performance. What follows are the nine model implementations in our experiment combining the three encoders and three decoders mentioned in Section 5.1.

Table 3: Question answering accuracy of different model implementations [without / with] formula mutation under 5-fold cross validation.

	LSTM	Tree	MILE
LSTM	65.2% / **65.5%**	**68.9%** / 68.0%	63.1% / **65.3%**
Graph	**71.2%** / 69.6%	**75.1%** / 72.0%	73.3% / **74.9%**
BERT	**78.0%** / 75.9%	**81.7%** / 81.3%	81.6% / **82.5%**

From this result, it can first be noticed that when MILE is working on the LSTM encoder or graph encoder, its performance is even worse than the combination of these encoders and tree decoder. However, when MILE is working with BERT, which is a much more powerful encoder, it shows better performance than the tree decoder. We explain this result with the design of MILE. Note that the decoding mechanism of MILE is relatively more complex, and MILE also contains components, such as $W^{qk}_{opd|*}$, that are specific to every single valid operator. These characteristics make the training of MILE be faced with more difficulty and can get overfitted more easily. These challenges can be further magnified by the deficiency of training data, which is the very case of Math23K—there are only about 18k training samples under 5-fold cross validation. However, these problems can be largely relieved by BERT, which is pretrained on a large enough corpus. A well-pretrained encoder can make MILE focus on the training of the decoding side and thus achieve better performance.

Nevertheless, we want to clarify that because BERT included external training data in its pretraining stage, the accuracies achieved by models involving BERT as encoders are not directly comparable with the others above. However, considering that employing large-scale pretrained language models has become a common practice in the solutions to NLP tasks nowadays, we still consider the results on BERT meaningful and having reference value.

On the other hand, we studied the impact of the formula mutation technique introduced in Section 4.3. The result is reported in Table 3. From this result, an interesting finding we obtained is that MILE is the only model that stably benefits from the formula mutation. In the view of human beings, the formula mutation is a quite natural practice, and the functionally equivalent formulas would not injure our learning of arithmetic. Conversely, the variety of formulas imparts us a better understanding of the nature of mathematics. However, this property does not hold for the existing formula predicting models, which further indicates that these models may not truly grasp the underlying concepts and natures of formulas and are simply fitting the formulas in the training set. In contrast, the capacity of MILE for learning from mutated formulas suggests that MILE learns and predicts formulas in a way more natural and close to human beings, which explained the reason that MILE outperforms other decoding techniques from another aspect. To provide more evidence for this proposition, we conducted some further studies and report the results in following sections.

5.3 Generalization Capacity

The first further question we want to investigate is the generalization capacity of different formula predicting models. For this goal, we conducted experiments on different train–test set splits. In the primary experiment presented in Section 5.2 under 5-fold cross validation and on the public test set, the split of the training and test sets is conducted randomly. Whereas, in this experiment, we split these two sets with two different metrics: *number count* and *formula length*. For *number count* split, we sort the (q, f) pairs by the count of number tokens in q. Then with proportion p, we use p of samples with more number tokens in q as test data, and the rest (1-p) of samples with less number tokens in q as training data. This split is denoted by *number-p* for short. For *formula length* split, we sort the (q, f) pairs by the length of f (i.e., the count of terms S_i in f). Then with proportion p, we use p of samples with longer f as test data, and the rest (1-p) of samples with shorter f as training data. This split is denoted by *formula-p* for short. All the other experiment settings except for the train–test set split are the same as the primary experiment. The result is reported in Table 4.

From this result, it can first be noticed that there is a huge gap between the accuracies achieved in 5-fold cross validation and our experimental splits. This is because our splits produce a highly challenging test environment, where the models have to generalize to problems that have more numbers involved and need longer calculating flows to solve than all the problems seen in the training stage. However, even in this challenging environment, MILE shows the best performance among the compared methods, which verified its superior generalization capacity. More detailed analyses of this result and a case study are presented in Appendix C and D.

Table 4: Question answering accuracy of different inference models under different train–test set splits.

split	BERT / LSTM	BERT / Tree	MILE (BERT)
5-fold	78.0%	81.7%	**82.5%**
number-0.2	13.4%	23.5%	**24.9%**
number-0.3	31.1%	39.6%	**41.5%**
number-0.4	39.3%	46.0%	**47.7%**
formula-0.2	21.4%	24.0%	**25.4%**
formula-0.3	4.0%	3.8%	**4.1%**
formula-0.4	17.9%	19.2%	**19.8%**

Table 5: Question answering accuracy of different inference models with token replacement.

	BERT / LSTM	MILE (BERT)
original	78.0%	**82.5%**
replace-2x	30.6%	**74.1%**
replace-3x	10.3%	**70.9%**

5.4 Robustness

The robustness of different formula predicting models is another question that raised our concerns. To shed light on this question, we designed an experiment in which the word tokens in question texts can be randomly replaced by number tokens. That is, with $q = [t_0, ..., t_n]$, we have $\{t^{\neg num}\} = \{t_i \ for \ t_i \ in \ q \ if \ not \ \mathrm{is\_num}(t_i)\}$ denotes the set of non-number tokens. For each q, we randomly sample r tokens from $\{t^{\neg num}\}$ and replace them with r random tokens sampled from $\{t^{num}\}$. Here, $\{t^{num}\}$ denotes the collection of all number tokens from all q in the dataset. That is, $\{t^{num}\} = \cup_{q_i} \{t_j \ for \ t_j \ in \ q_i \ if \ \mathrm{is\_num}(t_j)\}$. In the experiment, we decide r for each q independently in line with m, the count of number tokens in q. As for the two experiment settings *replace*-2x and *replace*-3x, we let $r = 2m$ and $r = 3m$, respectively. An additional restriction here is that at least half of the word tokens in each q will be kept unchanged. After this replacement, the formulas are also updated correspondingly to keep the number references accurate. The other settings in this experiment are the same as the primary experiment. The result is reported in Table 5. [2]

From this result, it is shown that the performance of the LSTM decoder is influenced by the token replacement significantly. The reason here is that the number tokens T_{num} distinguish numbers appearing in question texts by their indexes in a hard-coded way, which is highly sensitive to the absolute indexes of numbers instead of their relative contexts. As a result, this coding fails when some word tokens are replaced by number tokens in the question texts, which changes the absolute indexes of number tokens. This problem can be relieved to some extent by performing cross attention mechanism with encoder (Xie & Sun, 2019; Zhang et al., 2020). Nevertheless, we go one step further in the design of MILE, where the decoding of operands is fully powered by the attention calculation on the Memory Embedding Pool. Owing to this characteristic, the influence brought by token replacement is much less on MILE.

6 Conclusion

In this work, we discussed the limitations of existing techniques for solving mathematical problems from two aspects: the expressive capacity and the learning effectiveness. To address these problems, we first introduced the segmented representation technique to support non-tree-structured formulas, and then proposed MILE to implement reasonable formula predictions with the segmented representation. Our experiment on Math23K dataset verified the effectiveness of our proposed method and illustrated its superiority in generalization capacity and robustness. In view of this, we consider our proposals a rational approach toward the mathematical problem solving.

[2] We failed to conduct this experiment on the tree decoder because the modification of the tokens in questions influences the relation between entities, which further changes some labels essential for the decoding named number group. However, these labels were pre-generated and loaded in the published code of Graph2Tree, which made it impossible for us to reproduce.

REFERENCES

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. MathQA: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2357–2367, 2019.

Kezhen Chen, Qiuyuan Huang, Hamid Palangi, Paul Smolensky, Kenneth D Forbus, and Jianfeng Gao. Mapping natural-language problems to formal-language solutions using structured neural representations. In *International Conference on Machine Learning*, pp. 1566–1575, 2020.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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, Volume 1 (Long and Short Papers)*, pp. 4171–4186, June 2019.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, 2016.

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

HuggingFace. bert-base-chinese. `https://huggingface.co/bert-base-chinese`, 2018. Accessed: 2022-09-28.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations*, 2015.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 158–167, 2017.

Qianying Liu, Wenyv Guan, Sujian Li, and Daisuke Kawahara. 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)*, pp. 2370–2379, 2019.

Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–7, 2021.

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

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

Vered Shwartz, Peter West, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Unsupervised commonsense question answering with self-talk. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4615–4629, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Lei Wang, Dongxiang Zhang, Jipeng Zhang, Xing Xu, Lianli Gao, Bing Tian Dai, and Heng Tao Shen. Template-based math word problem solvers with recursive neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 7144–7151, 2019.

Yan Wang, Xiaojiang Liu, and Shuming Shi. Deep neural solver for math word problems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 845–854, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Yuxuan Wu and Hideki Nakayama. Weakly supervised formula learner for solving mathematical problems. In *Proceedings of the 29th International Conference on Computational Linguistics*, 2022.

Zhipeng Xie and Shichao Sun. A goal-driven tree-structured neural model for math word problems. In *IJCAI*, pp. 5299–5305, 2019.

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

A EXAMPLE OF FORMULA MUTATION

Table 6: Example of the mutation results on a formula.

original formula	$[(+, N_0, N_1), (-, N_2, M_0), (\times, M_1, N_3)]$
mutation results	$[(+, N_0, N_1), (-, N_2, M_0), (\times, M_1, N_3)]$
	$[(+, N_1, N_0), (-, N_2, M_0), (\times, M_1, N_3)]$
	$[(+, N_0, N_1), (-, N_2, M_0), (\times, N_3, M_1)]$
	$[(+, N_1, N_0), (-, N_2, M_0), (\times, N_3, M_1)]$

B IMPLEMENTATION DETAILS

For the LSTM encoder, we employed a two-layer Bidirectional LSTM Schuster & Paliwal (1997) with hidden state size 256. For the LSTM decoder, we employed a two-layer LSTM with hidden state size 512. For the graph encoder and the tree decoder, we followed the default implementations and settings in the published code of Zhang et al. (2020). For BERT, we used the pretrained Chinese BERT (HuggingFace, 2018) because all the questions in Math23K are written in Chinese. For MILE, we have the memory embedding dimension $d_m = 128$ and the dimension of hidden state $d_h = 512$. We employed two-layer MLPs for N_{hinit}, N_{ninit}, N_{opr}, $N_{mupd|*}$, and $N_{emb|*}$ with hidden layer dimension the same as output layers. We employed single layer fully connected networks for $N_{gate|*}$ and $N_{hupd|*}$. The dimension of the query and key vectors of W^{qk}_{opr} and $W^{qk}_{opd|*}$ is 128. We optimized all the models with Adam (Kingma & Ba, 2015). The learning rate is 10^{-4} for all the parameters in MILE's decoding side, and 10^{-3} for parameters in all other parts of models.

As for the decoding, we let $T_{op} = \{+, -, \times, \div, =\}$ and $T_{con} = \{1, 2, \pi, 100\}$ in line with the demand of Math23K. Note that the operator $=$ in T_{op} is a special token for supporting the case that the answers to some questions can directly be a number appearing in the question texts. There is no arithmetic computation required in these cases, so we formulate f_s with $[(=, N_i)]$.

C DETAILED RESULT ANALYSES

An abnormal phenomenon observed in the experiment result in Section 5.3 is that, in *number count* split, the accuracy achieved on *number*-0.3 and *number*-0.4, where 30% or 40% of samples are utilized as test data, is even higher than *number*-0.2, where 20% of samples are utilized as test data. In *formula length* split, these variations are not even monotonic. However, this phenomenon is not incomprehensible. We analyzed the sliding window average accuracy on test samples in these test sets and show the results in Figure 2 and 3. In these two figures, the solid gray lines show the count of numbers and the length of formulas of test samples, which are also the evidence for our sample sorting and set split. In both figures, the accuracy drops significantly when the count of numbers or length of formulas grows. This illustrates the difficulty of generalizing to mathematical problems that have more numbers involved and need longer calculating flows to solve. However, compared with *number*-0.2, the 10% or 20% additional test data in *number*-0.3 and *number*-0.4 is of lower difficulty and have higher accuracy. This is because the numbers in these samples are fewer and there are also problems with three numbers in the training set, which makes the generalization easier. As a result, including these additional samples into the test set thins the total difficulty of testing and makes the final accuracy higher. Nevertheless, if we focus on the overlapping part of test samples in *number*-0.2, *number*-0.3 and *number*-0.4 on the right side, the accuracy achieved on *number*-0.2 is still higher than *number*-0.3 and *number*-0.4, which is a reasonable outcome. The result on *formula length* split can be explained similarly, while things are a little more radical. The train–test set split on *formula*-0.3 is almost right on the edge between problems with formula lengths less than and not less than three. This makes the generalization extremely hard and leads to very low accuracy. In contrast, a part of the problems with formula length three is split to the training set in *formula*-0.2, so this makes the generalization much easier and the accuracy higher.

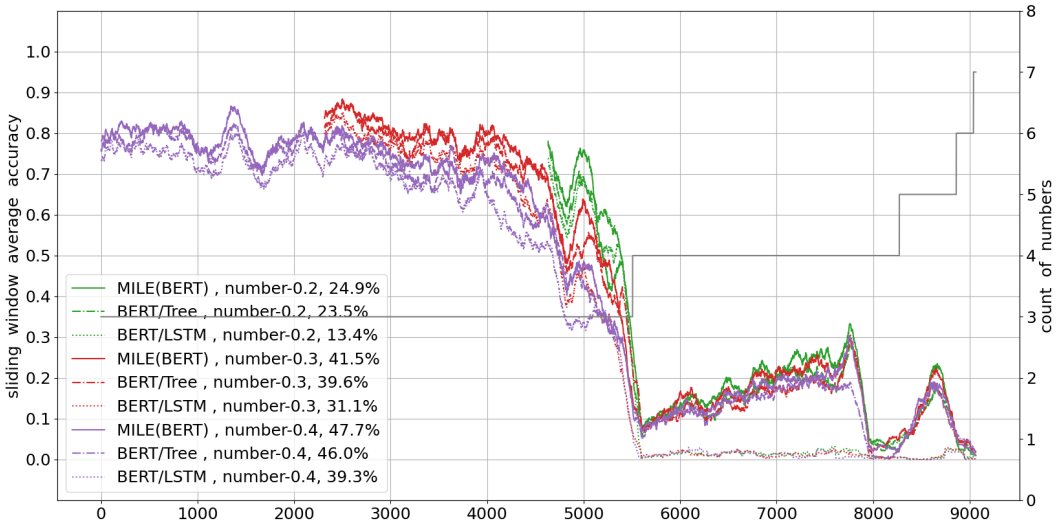


Figure 2: The sliding window average accuracy on *number-p* with window width 200.

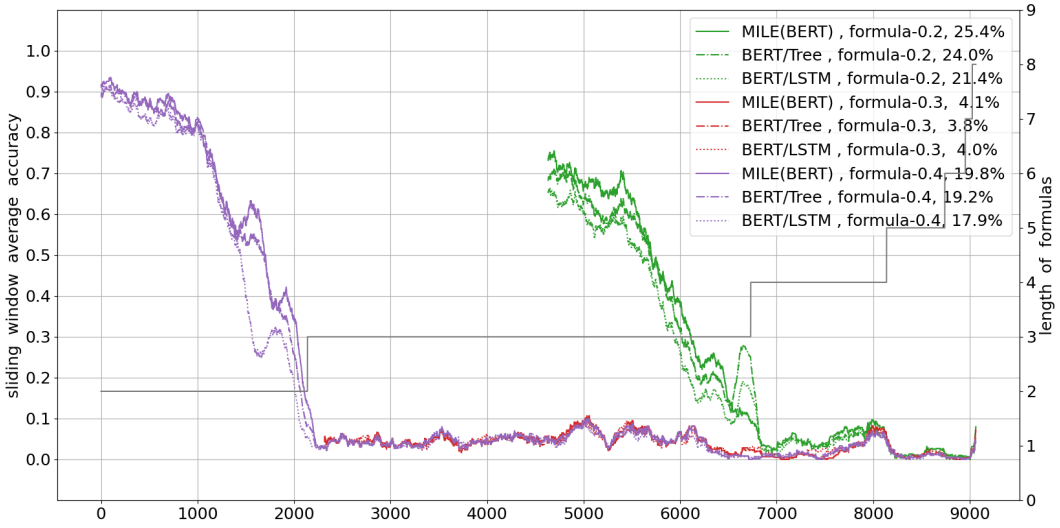


Figure 3: The sliding window average accuracy on *formula-p* with window width 200.

D CASE STUDY

Table 7: Examples of the formulas predicted by different inference models on some problems. Note that the questions in Math23K dataset are written in Chinese, we translate them into English only for this demonstration.

test set	*number*-0.2
question	Lee deposits 400 dollar in the bank for 4 years. The annual interest rate is 2.80%. How much after-tax interest can Lee get when it is due? (Deposit interest tax is 20%)
answer	35.84
BERT / LSTM	(raw) $[\times, \times, N_0, N_1, -, 1, N_2]$ (flatten) $(400 \times 4) \times (1 - 2.80\%)$ **(wrong)**
BERT / Tree	(raw) $[\times, \times, N_0, N_2, N_1]$ (flatten) $(400 \times 2.80\%) \times 4$ **(wrong)**
MILE (BERT)	(raw) $[(\times, N_0, N_2), (\times, M_0, N_1), (-, 1, N_3), (\times, M_1, M_2)]$ (flatten) $((400 \times 2.80\%) \times 4) \times (1 - 20\%)$
test set	*number*-0.4
question	The distance from Lily's house to the shopping mall is 2400 meters. She ran at a speed of 150 meters per minute for 5 minutes, and then walked at a speed of 60 meters per minute for 16 minutes. How many meters does she still need to walk to reach the shopping mall?
answer	690
BERT / LSTM	(raw) $[+, N_0, \times, N_1, N_2]$ (flatten) $2400 + (150 \times 5)$ **(wrong)**
BERT / Tree	(raw) $[-, N_0, \times, N_1, N_4]$ (flatten) $2400 - (150 \times 16)$ **(wrong)**
MILE (BERT)	(raw) $[(\times, N_1, N_2), (-, N_0, M_0), (\times, N_3, N_4), (-, M_1, M_3)]$ (flatten) $2400 - (150 \times 5) - (60 \times 16)$
test set	*formula*-0.2
question	Students in a school go to camp. There are 28 students in the lower grades. There are 17 times as many as lower grades and 16 more students in the higher grades. If every 13 students share a tent, how many tents do the lower grades students and higher grades students need in total?
answer	40
BERT / LSTM	(raw) $[+, +, N_0, \div, N_0, N_1, N_2]$ (flatten) $(28 + (28 \div 17)) + 16$ **(wrong)**
BERT / Tree	(raw) $[\div, +, \times, N_0, N_1, N_2, N_3]$ (flatten) $((28 \times 17) + 16) \div 13$ **(wrong)**
MILE (BERT)	(raw) $[(\times, N_0, N_1), (+, N_2, M_0), (+, N_0, M_1), (\div, M_2, N_2)]$ (flatten) $(28 + (16 + (28 \times 17))) \div 13$