A Hybrid Top-Down and Bottom-Up Neural Model for Mathematical Word Problems

Raghav Bansal[*] **Jeffrey Tsaw**[†]
Carnegie Mellon University
raghavb@andrew.cmu.edu, jtsaw@andrew.cmu.edu

1 Introduction

Solving mathematical world problems (MWP) is an important NLP task that has interested researchers since the 1960s. The goal is to develop models which take the text of a world problem as input and produce a numerical answer as output. Such models must be capable of both natural language understanding and mathematical reasoning. Solving MWPs has been a difficult task for a three reasons. First, there is a conceptual gap between the natural language form of the MWP and its corresponding mathematical equation. Second, MWPs can test a student's ability in a diverse set of concepts. Third, effectively representing the structure present in mathematical expressions is difficult.

Typically, MWPs are formulated as text with the problem description, a mathematical solution equation and the final answer based on evaluating this expression. As a simple example, consider Table 1. Here the model must: 1) understand what is meant by words such as *length*, *width* and *shorter* and 2) infer the correct equation to compute the final answer of 10.

Problem: The length of Bob's backyard is 5ft. The width is 2ft shorter than the length. What is the area of Bob's backyard?
Mathematical Equation: $5 \times (5 - 3)$
Answer: 10

Table 1: Sample MWP

Modern approaches to solving MWPs typically fall into 2 categories: top down and bottom up. Top down approaches involve constructing goal vectors, and decomposing the problem into sub-goals, where each sub-goal becomes either an operation or a number. A bottom up approach involves constructing small expressions, and combining useful expressions until the final solution expression is generated. However, these approaches fail to emulate human-like problem solving, which is often a *combination* of top down and bottom up processing. In this work, we present a novel hybrid top-down and bottom-up approach to solving MWPs.

We conduct experiments on Math23K, a Chinese MWP dataset with 23161 samples, and evaluate the performance according to equation accuracy, which measures if predicted and the ground-truth equations match, and answer accuracy, which directly measures if the predicted and ground-truth solutions match. Our experiments demonstrate that the hybrid model achieves slightly better performance compared to existing state of the art.

[*]Department of Statistics and Data Science
[†]Department of Electrical and Computer Engineering

Spring 2022: CMU 10-707 Project

2 Background

In order to evaluate the performance of our approach, we pick GTS as a baseline. GTS is a the simplest top-down model for solving MWPs. It uses a 2-layer Bidirectional GRU-based RNN as the problem encoder. The decoder involves recursively decoding a goal vector into tokens, and computing sub-goals if the token is an operator, until all decoded tokens are numbers. This process can be visualized in 1. Although there several models that have outperformed GTS since, we chose GTS as a baseline as many subsequent models are extensions of it. After performing 5-fold cross-validation on Math23k, GTS achieves a final test equation accuracy of 65.3, and an answer accuracy of 74.3.

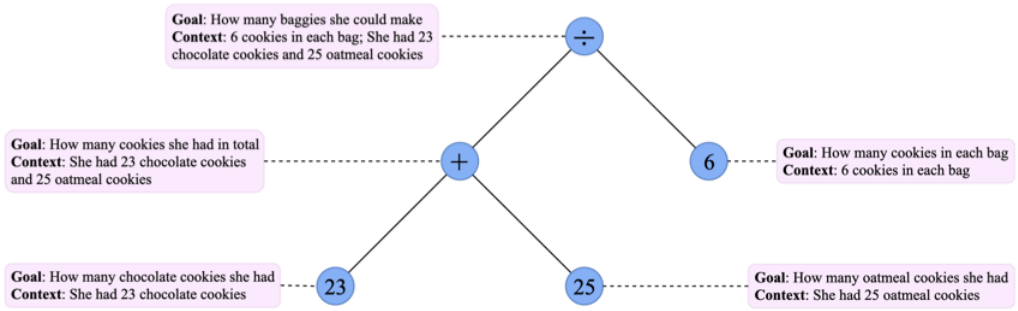


Figure 1: Expression tree from [1]

3 Related Work

3.1 Initial Approaches to MWPs

The initial methods for solving MWPs were rule-based approaches [2, 3], which use hand-crafted features to map problems to a fixed set of equation templates. Soon after, semantic parsing [4, 5] and statistical machine learning approaches [6, 7] became popular.

3.2 Neural Network Approaches to MWPs

Recently, deep neural networks have been shown to outperform all of these classical methods. In [8], the authors trained a deep seq2seq model to solve MWPs. The key feature of this model in an encoder-decoder framework, where the encoder maps the problem text to a hidden state and then the decoder maps the hidden state to a mathematical equation. Soon after, [9] improved the decoder so it produces an expression tree rather than a sequence. The final answer can be computed from the tree with postfix traversal. The use of trees normalizes the output, avoiding degradation in performance due to non-uniqueness of equations. [1] introduced GTS, which is discussed in Section 2. [10], [11] and [12] all have decoders that are top-down goal driven based, primarily inspired by GTS. [13] and [14] both use graphs to represent the relationships between words, quantities, and other quantities as the problem encoder, while maintaining a top-down decoding process. Alternatively, [15] solves MWPs in a bottom up process, building up likely expression trees until the final solution tree is reached. Finally, there has been success in fine tuning pre-trained language models, such as GPT-3, as in [16].

3.3 Top-Down Bottom-Up Processing

There have been several machine learning based models that have been similarly inspired by the top-down bottom-up nature of neural processing. In [17], the authors demonstrate that a combined top-down and bottom-up approach to single figure image segmentation outperform pure top-down and

pure bottom-up approaches. [18] explores RNN architectures that dynamically combine top-down and bottom-up signals using attention. Finally, [19] fuse a separate bottom-up and top-down neural networks for visual object recognition.

4 Methods

We present a hybrid top-down bottom-up neural model for solving MWPs. The solving process is as follows. The bottom-up network is inspired by [15] and is first trained to generate candidate trees and subtrees of the solution. When training the top-down tree decoder, the bottom-up network is first run to produce candidate solution trees and its subtrees. After each subgoal is calculated, a query network queries all the candidate trees and picks the tree that best solves this goal, if any. Finally, after that subgoal is realized, the subtree is replaced with the chosen candidate tree.

4.1 Bottom-Up Network

Like in [15] the bottom-up network is a DAG-LSTM structured network that works as below.

4.1.1 Problem Encoder

We use a 2 Layer Bidirectional GRU model as the problem encoder as in [1]. If $P = w_1 w_2 \dots w_n$ is the input problem text, we first append all potential constants (i.e. 1, 2, 3.14, 0.5) that might be used in the solution expression to the end of the problem text. The augmented input problem text is thus $P = w_1 w_2 \dots w_n c_1 c_2 \dots c_k$ if there are k constants. We embed each token w_i in the text by looking up an embedding matrix M_{embed} to produce embedding $\mathbf{w}_i$. The encoder takes in each the sequence $\mathbf{w_1}, \mathbf{w_2}, \dots \mathbf{w_{n+k}}$ and produces a forward direction sequence of hidden states $\overrightarrow{\mathbf{h}_1}, \overrightarrow{\mathbf{h}_2}, \dots, \overrightarrow{\mathbf{h}_{n+k}}$ and a backward direction sequence of hidden states $\overleftarrow{\mathbf{h}_1}, \overleftarrow{\mathbf{h}_2}, \dots, \overleftarrow{\mathbf{h}_{n+k}}$, where the forward and backward directions are calculated according to

$$\overrightarrow{\mathbf{h}_t} = \text{GRU}(\overrightarrow{\mathbf{h}_{t-1}}, \mathbf{w}_t)$$
$$\overleftarrow{\mathbf{h}_t} = \text{GRU}(\overleftarrow{\mathbf{h}_{t+1}}, \mathbf{w}_t)$$

The final hidden state at any time t is thus given by

$$\mathbf{h}_t = \overrightarrow{\mathbf{h}_t} + \overleftarrow{\mathbf{h}_t}$$

4.1.2 Bottom-Up Decoder

Once the problem and relevant quantities are encoded, the decoder will create a new tree using existing candidates and operations, and determine whether the new tree should be added to the list of candidates. More specifically, if C_s is the current list of candidates, and $i, j \in C_s$, and o is an operation, then (o, i, j) is a potential tree.

For a new tree (o, i, j), the decoder computes a hidden representation h based on the hidden representations h_i, h_j. This is done using the corresponding DAG-LSTM cell of operation o, or DAG-LSTM$_o$. DAG-LSTM cells are discussed in section 4.1.3. We use this hidden representation and calculate the attention c over the encoded input $\mathbf{H} = \mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_{n+k}$, according to the self-attention mechanism introduced in BERT [20]. More specifically, the query vector $q = W^q h$, the key vector is $K = W^k H$, and the value vector is $V = W^v H$. We use the attention output c, and the hidden representation of the new tree h to calculate the probability of being *positive*, where positive candidates are added to C_s. Concretely, the probability p that (o, i, j) is positive is given by

$$h = \text{DAG-LSTM}_o(h_i, h_j)$$
$$c = \text{Attention}(h, H)$$
$$p = \text{Softmax}(\text{Linear}([h; c]))$$

We add the candidate (o, i, j) to C_s if $p > 0.9$.

4.1.3 DAG-LSTM Cells

DAG-LSTM cells are introduced in [15] and are designed to encode trees in a way that captures the commutativity of the operation. The idea is that trees representing $a + b$ and $b + a$ should have the same representation since they represent the same value, but trees representing $a - b$ and $b - a$ should be different. Previous Seq2Seq models attempt to solve this problem by rearranging values, however, when a, b are subtrees, this problem becomes difficult with such a heuristic. For this reason, we adopt the same strategy as [15] and encode new trees with these cells. Next, we introduce the two types of DAG-LSTM cells.

DAG-LSTM for Commutative Operations: For new trees (o, l, r) where o is a commutative operation (i.e $\times$ or $+$) the hidden representation h for this tree should be the same for (o, l, r) and (o, r, l). Let h_l, h_r be the hidden representations for trees l, r respectively. We set $\tilde{h} = h_l + h_r$. Then, h can be computed according to

$$
\begin{aligned}
i &= \sigma(W^i \tilde{h} + U^i x + b^i) \\
f_k &= \sigma(W^f h_k + U^f x + b^f) \quad \text{for } k = l, r \\
o &= \sigma(W^o \tilde{h} + U^o x + b^o) \\
\hat{c} &= \tanh W^c \tilde{h} + U^c x + b^c \\
c &= f_l \odot c_l + f_r \odot c_r + i \odot \hat{c} \\
h &= \tanh c \odot o
\end{aligned}
$$

Where c_l, c_r are the cell states c of the l tree and r tree respectively, and x is the embedding of o, found by looking up an embedding matrix M_{op}.

DAG-LSTM for Non-Commutative Operations: For new trees (o, l, r) where o is not a commutative operation (i.e / or $-$) the hidden representation h for this tree should be the different for (o, l, r) and (o, r, l). Set $\tilde{h} = [h_l; h_r]$. Then, h can be computed according to

$$
\begin{aligned}
i &= \sigma(W^i \tilde{h} + U^i x + b^i) \\
f_k &= \sigma(W^f \tilde{h} + U^f x + b^f) \quad \text{for } k = l, r \\
o &= \sigma(W^o \tilde{h} + U^o x + b^o) \\
\hat{c} &= \tanh W^c \tilde{h} + U^c x + b^c \\
c &= f_l \odot c_l + f_r \odot c_r + i \odot \hat{c} \\
h &= \tanh c \odot o
\end{aligned}
$$

4.1.4 Training Procedure

Let O denote the list of operations and C_s be the list of candidate trees. During training, we initialise C_s as the list of constants C and the numbers in the problem N_P, or $C_s = C \cup N_P$. We then construct every possible candidate tree (o, l, r) for $l, r \in C_s$ and $o \in O$ as well as calculate p, the probability it is positive. If it is positive, we add it to C_s, or $C_s = C_s \cup (o, l, r)$. This process terminates if the length of the new tree (o, l, r) is greater than the maximum length of any tree in the dataset, or no tree (o, l, r) is positive. Given a dataset $D = \{(P_i, T_i) | 1 \leq i \leq n\}$, we define the loss function as a variant of binary cross entropy:

$$
L = \sum_{s \in D} \sum_{\substack{c \in C_s \text{ or} \\ c \in \text{Subtree}(s)}} y_c \log(p_c) + (1 - y_c) \log(1 - p_c)
$$

where $s = (P, T)$ is a data sample with problem text P, and solution tree T, Subtree(s) refers to all subtrees of the solution tree T, and y_c is the ground truth label of tree c. More specifically, $y_c = 1$ if c is a subtree of T, and 0 otherwise. Since we are using the bottom-up network to generate candidate subtrees, we encourage the network to generate more trees than less, and penalize false negatives more than false positives. We compute the accuracy of this network as $\frac{\#\text{correct subtrees in } C_s}{\#\text{total subtrees in T}}$.

Alternative loss functions were considered where only $c \in C_s$ trees were factored into the loss. However, this resulted in the loss being 0 if no trees are ever added to C_s, so there was little incentive for the network to assign trees a $p > 0$.

4.2 Top-Down Network

The top-down network is identical to that of GTS, with the addition of the query network that determines if a subtree should be replaced in the solution.

4.2.1 Problem Encoder

We use the same encoder structure detailed in [15] with one slight difference. In the top-down network, we no longer append the constants to the end of the problem statement, and instead, encode the problem statement as is.

We briefly experimented with other encoders and embeddings. Mainly, we experimented with more pre-trained BERT embeddings for Chinese characters. Additionally, we tried using an group attention based encoder detailed in [21]. Such an encoder uses four attention mechanisms to capture context in a MWP that may not be ordered sequentially and thus would not be picked up by a bidirectional GRU. It uses: 1) global attention among all tokens in the MWP, 2) local attention between a token and its neighboring words, 3) quantity attention between all numerical values, and 4) question attention between the question and the numerical values. Overall, we found these modifications failed to beat the baseline. The results are detailed in 5.

4.2.2 Top-Down Decoder

We use the same tree decoder as presented in GTS. Given a predicted token $\hat{y}$, during subgoal generation $\mathbf{q}_l$ and $\mathbf{q}_r$, these two vectors are also passed into the query network. For details of the top-down decoder, refer to [1].

4.2.3 Query Network

We present a novel query network that determines if, given an embedding of a subtree, it achieves a specified subgoal generated by the top-down decoder. Concretely, we first generate all candidate subtrees using the bottom-up network described in 4.1. We then encode the subtree (o, l, r) using the corresponding DAG-LSTM of type o, and the hidden representations of the left and right subtrees h_l, h_r respectively. Then, for a given left and right subgoal $\mathbf{q}_l, \mathbf{q}_r$, we calculate

$$
\begin{aligned}
h &= \text{DAG-LSTM}_o(h_l, h_r) \\
c_k &= W^c \tanh W^h[\mathbf{q}_k; h] + b^h \quad k \in (l, r) \\
a_k &= \text{ReLU}(W^a c_k + b^a) \quad k \in (l, r) \\
p_k &= \text{Softmax}(\text{Linear}(a_k)) \quad k \in (l, r)
\end{aligned}
$$

where p_l, p_r denote the probability that the subtree h satsfies the left subgoal and right subgoal $\mathbf{q}_l$ and $\mathbf{q}_r$ respectively.

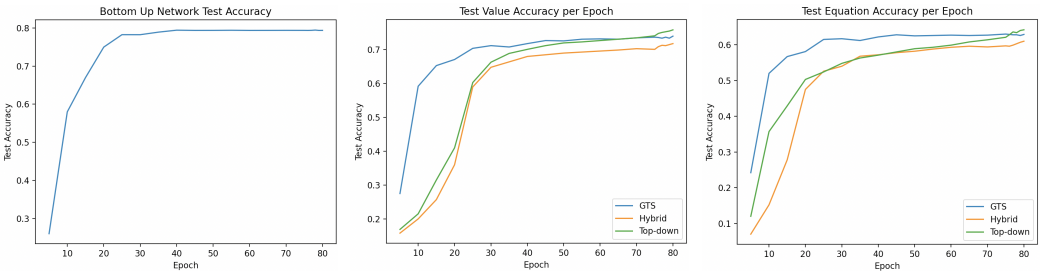
Note that each subgoal $\mathbf{q}_k$ queries all the subtree embeddings and finds, if any, the subtree that satisfies the subgoal with the highest probability.

During training, the top-down decoder and query network and trained in conjunction, although the query network does not factor into the top-down decoding process, and only factors in during test-time evaluation.

4.2.4 Training Procedure

The overall loss function for the top-down network is comprised of the GTS loss function, and the query network loss function. Formally, this is

$$L = L_{GTS} + L_{\text{query}}$$



(a) (b) (c)

Figure 2: (a) Test subtree accuracy per epoch of bottom-up network; (b) Test value accuracy per epoch of GTS, hybrid model, and top-down with real subtrees; (c) Test equation accuracy per epoch of the 3 models above.

The GTS loss function is simply to minimize the negative log-likelihood on the dataset $D = \{(P_i, T_i)|1 \leq i \leq n\}$ or

$$L_{GTS} = \sum_{P,T \in D} -\log(T|P)$$

The query loss function is to minimize the binary cross entropy between each subgoal and every subtree in the solution tree T. More specifically,

$$L_{\text{query}} = \sum_{P,T \in D} \frac{1}{|\text{Subtree}(T)|} \sum_{\mathbf{q}_l, \mathbf{q}_r \in T} \sum_{c \in \text{Subtree}(T)} y_c \log p_c + (1 - y_c) \log(1 - p_c)$$

where $y_c = 1$ if c is the subtree in T that realizes goal $\mathbf{q}_k$ for some $k \in (l, r)$, and p_c is calculated according to the query network described in 4.2.3. During training, we compute all possible subtrees directly using solution tree T, instead of generating them from the bottom-up network.

In addition to this approach, we also considered an end-to-end training procedure, where the bottom-up and top-down networks are trained in conjunction. However, preliminary experiments with this setup failed to learn anything. We believe there was too much variance in the network, and the bottom up network wasn't learning to generate subtrees, and as a result the top-down network wasn't learning to score them properly in the query network, and as a result the loss was always high.

4.3 Training and Evaluation of Hybrid Network

When training the hybrid network, we train the bottom-up network and top down network in parallel. This is possible since the query network takes in solution trees and subtrees directly from the solution expression T, instead of generating them using the bottom-up network.

During evaluation, the bottom-up network first generates candidate solution trees and subtrees and passes them to the top down network. During the top-down decoding process, each time a subgoal is generated, it queries each candidate subtree. If the query network returns a replacement probability $p_r > 0.9$, the corresponding candidate subtree is saved. Once the top-down network finishes decoding and produces solution tree T', all the found replacement subtrees then replace the corresponding subtree in T' to generate a final solution tree T.

5 Results

We first train the bottom-up network on Math23k using 5-fold cross validation. We train for 80 epochs, and use a batch size of 8. In each epoch, the dataset is first shuffled and then cut into mini-batches. We use a learning rate of 0.001 that is halved every 20 epochs, and optimize using Adam. Additionally, we set the dropout probability to 0.5. The resulting loss and accuracy plots are shown below.

We evaluate the performance on the test set after every 5 training epochs for the first 75 epochs, then after each epoch for the last 5. The results can be found in Figure 2(a). From this we, see that the bottom-up network finds the subtrees fairly well, with a final test accuracy of 79.39%.

During the pass through the test data, we also collected data on the average number of trees for each depth. The average number of each tree added to the candidate trees during the final test is shown in Table 2. We see that the network is extremely capable at finding trees below length 5, and there is a

Tree Length	Average Count
3	41.24
4	23.62
5	7.06
6	4.33
7	0.76
8	0.31
9	0.02
10	0.0056

Table 2: Average number of trees per length present in candidate set C_s in test data.

steep drop off in the network's ability to find these trees. This is likely due to the high variance, as finding trees of length 5 is dependent on finding the correct subtrees of length 4, which is dependent on finding the correct subtrees of length 3. This makes finding higher length subtrees a hard task. Furthermore, another reason would be that the average solution tree length was only 6.8, meaning there are only a few samples with subtrees of length > 5, so the network doesn't become as adept at finding these as it does trees of lower length.

We then combined the top down and bottom up network into the hybrid network and evaluated the performance of this compared to the baseline GTS model. We train both networks for 80 epochs using 5-fold cross validation on the Math23k dataset. As in the experiments with the bottom-up network, we use a batch size of 64 a learning rate of 0.001. We evaluate the performance on the test set every 5 epochs. We use these parameters as they are the same as the baseline, and this ensures an accurate comparison between the methods. We also compare the performance on the test set of both these models to our top down model with the subtrees from the solution being directly fed in as input, instead of being generated from the bottom up network. The test value accuracy per epoch can be seen in Figure 2(b), and the test equation accuracy can be seen in Figure 2(c). For reference, the final test accuracy, averaged across all 5 folds, for the 3 models are shown in Table 3. Although the

Model	Avg Test Value Accuracy $(\%)$	Avg Test Equation Accuracy $(\%)$
GTS	74.00	62.90
Hybrid (ours)	71.81	61.90
Top-down with real subtrees	75.86	64.20

Table 3: Average final test equation and value accuracies for each model.

hybrid model didn't outperform the baseline, the top down model with real subtrees outperformed the baseline on both equation and value accuracy after 80 epochs. The performance of the hybrid model can therefore most likely be explained by the bottom-up network not generating enough accurate candidate subtrees for the top-down network to replace.

Our experiments changing the encoder generally failed to show improvements over the GTS baseline. In fact, we generally saw degradations in test accuracy. In the first experiment, we replaced the basic GTS embedding with a Chinese-character based BERT embedding. Then, we replaced the encoder with a group-attention based encoder. Finally, we tried combining the BERT embedding with the group-attention based encoder. We trained these networks with the same settings as the experiments above. The final test accuracy for the three models is shown in Table 4. The worse performance is likely due to overfitting in the encoder.

Model	Avg Test Value Accuracy (%)	Avg Test Equation Accuracy (%)
GTS	74.00	62.90
GTS-BERT	70.40	60.11
GTS with Attention	72.51	61.63
GTS-BERT with Attention	71.00	60.90

Table 4: Average final test equation and value accuracies for each model.

6 Discussion and Analysis

Overall the hybrid model provided interesting and promising results. With the hybrid bottom-up and top-down approach, both the test value and equation accuracy came close to the baseline model. However, when generating the subtrees using the real solution tree, the top down network outperformed GTS and the hybrid model on both test equation and value accuracy. This points to the bottom-up network being the limiting factor in answer generation of the hybrid network. Based on the average number of trees being generated in Table 2, the bottom-up network is producing too many candidate trees that are embedded close together, meaning the query network is struggling to score each tree accurately. Additionally, when training the top-down model, the query network impacts the embeddings of the subgoals. When evaluating the top-down network without replacements (this is just vanilla GTS), the model fails to learn to solve the problems, and could not reach a test accuracy of 10% even after 80 epochs. It could be interesting to first train GTS, such that the embeddings of the goal are fixed and helpful in the top-down decoding process, and then train the query network, instead of training them in parallel. More investigation is also required on the effectiveness of the query network. The architecture of the network was primarily inspired by the score function in GTS.

On the encoder side, switching the embedding failed to produce better results. The reason could be that the BERT embeddings for Chinese characters were trained on individual characters, whereas in the preprocessing of the Math23k dataset, the characters are separated into *tokens*, sometimes containing multiple characters, and these are embedded separately. This mean the embedding matrix could produce a better representation of the *tokens*, compared to pretrained BERT embeddings. The GroupATT encoder also introduces a lot more parameters as it contains several Multihead Attention layers. This introduces a high amount of variance, and made training unstable, which is why it failed to perform as well as the more simple bidirectional GRU encoder.

Lastly, there are inherent limitations to the Math23k dataset. Occasionally, converting the solution expression to a solution tree resulted in an unknwon token (UNK) in the solution tree. This meant that even if the model was learning to output the correct token, the solution doesn't have the correct reference token, which inhibits the ability of the model to fully learn. Moving forward, a better way of handling UNK tokens needs to be used, rather than comparing the model output with itself.

To conclude, the hybrid model produced interesting results and had similar test equation and value accuracy to GTS. When using the real subtrees instead of ones generated by the bottom-up network, the model outperformed GTS on both test equation and value accuracy on Math23k. Potential extensions include a more rigorous handling of UNK tokens, a training approach that trains the query network separately, and improvements to the bottom-up network such that it is more precise, and has less false positives.

References

[1] Zhipeng Xie and Shichao Sun. A goal-driven tree-structured neural model for math word problems. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 5299–5305. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[2] Charles R. Fletcher. Understanding and solving arithmetic word problems: A computer simulation. *Behavior Research Methods*, 17(5):565–571, September 1985. Copyright: Copyright 2011 Elsevier B.V., All rights reserved.

[3] Ma Yuhui, Zhou Ying, Cui Guangzuo, Ren Yun, and Huang Ronghuai. Frame-based calculus of solving arithmetic multi-step addition and subtraction word problems. 2:476–479, 2010.

[4] Rik Koncel-Kedziorski, Hannaneh Hajishirzi, Ashish Sabharwal, Oren Etzioni, and Siena Dumas Ang. Parsing algebraic word problems into equations. *Transactions of the Association for Computational Linguistics*, 3:585–597, 2015.

[5] Shuming Shi, Yuehui Wang, Chin-Yew Lin, Xiaojiang Liu, and Yong Rui. Automatically solving number word problems by semantic parsing and reasoning. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1132–1142, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[6] Mohammad Javad Hosseini, Hannaneh Hajishirzi, Oren Etzioni, and Nate Kushman. Learning to solve arithmetic word problems with verb categorization, October 2014.

[7] Nate Kushman, Yoav Artzi, Luke Zettlemoyer, and Regina Barzilay. Learning to automatically solve algebra word problems. pages 271–281, June 2014.

[8] 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*, pages 845–854, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[9] Lei Wang, Yan Wang, Deng Cai, Dongxiang Zhang, and Xiaojiang Liu. Translating a math word problem to a expression tree. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1064–1069, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[10] Jipeng Zhang, Roy Ka-Wei Lee, Ee-Peng Lim, Wei Qin, Lei Wang, Jie Shao, and Qianru Sun. Teacher-student networks with multiple decoders for solving math word problem. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 4011–4017. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[11] Zhenwen Liang, Jipeng Zhang, Jie Shao, and Xiangliang Zhang. Mwp-bert: A strong baseline for math word problems. 2021.

[12] Jinghui Qin, Lihui Lin, Xiaodan Liang, Rumin Zhang, and Liang Lin. Semantically-aligned universal tree-structured solver for math word problems, 2020.

[13] 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*, pages 3928–3937, Online, July 2020. Association for Computational Linguistics.

[14] Yibin Shen and Cheqing Jin. Solving math word problems with multi-encoders and multi-decoders. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 2924–2934, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics.

[15] Yixuan Cao, Feng Hong, Hongwei Li, and Ping Luo. A bottom-up dag structure extraction model for math word problems. In *AAAI*, 2021.

[16] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021.

[17] E. Borenstein, E. Sharon, and S. Ullman. Combining top-down and bottom-up segmentation. pages 46–46, 2004.

[18] Sarthak Mittal, Alex Lamb, Anirudh Goyal, Vikram Voleti, Murray Shanahan, Guillaume Lajoie, Michael Mozer, and Yoshua Bengio. Learning to combine top-down and bottom-up signals in recurrent neural networks with attention over modules. *CoRR*, abs/2006.16981, 2020.

[19] Yuhua Zheng, Yan Meng, and Yaochu Jin. Fusing bottom-up and top-down pathways in neural networks for visual object recognition, 2010.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[21] Jierui Li, Lei Wang, Jipeng Zhang, Yan Wang, Bing Tian Dai, and Dongxiang Zhang. Modeling intra-relation in math word problems with different functional multi-head attentions. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6162–6167, Florence, Italy, July 2019. Association for Computational Linguistics.