Decoder-Only Transformer for Math Word Problem to Computational Graph Machine Translation

Joshua Negreanu
Oregon State University
negreanj@oregonstate.edu

Jose Sanchez
Oregon State University
sanchej7@oregonstate.edu

Carson Nitta
Oregon State University
nittac@oregonstate.edu

Abstract

The hardest part of solving a math word problem is often setting it up properly. From there, computation can easily solve for the answer. Bridging the gap between natural language and the logic of math may result in more intuitive methods of translating real-world problems to precise mathematical equations and solutions. We explore seq2seq machine translation of math word problems to computational graphs by leveraging a decoder-only architecture we call DoM, experimenting with English token embeddings, and evaluating over three key metrics on the MathQA [2] dataset. We find our model to perform well on test and validation samples, and propose using such small-scale auxiliary models to perform domain-specific reasoning tasks, offloading the needs of mathematical computation from an LLM.

1. Introduction

1.1. Problem Setting

Deriving mathematical reasoning and logic from written language context has and continues to be a prime area of research within natural language processing. Such an area of research deserves attention as it attempts to bridge a gap between problem descriptions and mathematical deduction, which could provide more intuitive methods of solving real-world problems.

However, this translation is not easy. Mathematics is built on rigor and logic, while natural language is often variable and subjective. Bridging the gap means not only taking in a written problem and reaching a solution, but providing logical steps towards the conclusion, often in the form of equations. This "reasoning" is crucial to ensure that a language model has properly set up the problem given sufficient context.

Modern large-scale methods, as will be discussed in section 2, often rely on an LLM providing internal reasoning generation as context for reaching a solution. However, LLMs are purpose-built for dealing with the stochastic nature of language generation, not the precise formulations of math. Other methods, ours included, involve the construction of algorithms that can be run externally to produce a solution. The structure of these algorithms illustrates the reasoning of the model.

1.2. Our Approach

We train and evaluate a seq2seq decoder-only transformer architecture (see section 3.2) on the MathQA dataset, which, among other features, provides written math word problems in English and corresponding linear constructions of computational graphs (see section 3.1). With this model, we note high scores in token-per-token generation as well as sequence accuracy, BLEU score, and graph edit distance (GED) in section 4. Our code is open source and publicly available via our Github repository[1].

2. Related Work

Our approach is heavily inspired by Amini et al. [2], which addressed mathematical reasoning of word problems. They created a new dataset of 37k+ math problems annotated with mathematical operations, a "linear formula," their reasoning approach, and the correct answer. These advancements in the sector brought to light LLM's challenges in mathematical reasoning tasks within word problem settings, which require both proper interpretation of the problem and accurate computation of it. These challenges in the field have brought about two camps of research: "fine-tuning" and "prompting" [1].

A "fine-tuning" methodology involves retraining models. This can include training with a more detailed dataset that includes step-by-step instructions to allow the model to capture more complex patterns in data [7]. Another approach is knowledge distillation [5], a process by which a "teacher" model, typically over-parameterized and large, distills its reasoning capabilities onto a much smaller student model. It does so by matching the "student" model

[1] https://github.com/joses2017smjh/OMR_LLM

with the teacher's softmax distributions during the training phase.

Current "prompting" approaches work on frozen LLMs that look to steer their generation during inference. Advances in the sector have involved the development of "chain-of-thought" [6] (CoT), "program-of-thought" [3] (PoT), "self-verification" [8], and "few-shot learning" [6]. In CoT, great improvement has been seen in models' accuracy in multi-step/logical problems with the inclusion of a "think out loud" prompt after the problem. PoT explores the integration of code generation and execution used for calculations to provide accurate answers. Self-verification re-prompts the LLM with its previous generation and has it self-assess, and a few-shot approach looks to provide the LLM with x amount of examples to provide a structure/pattern for the LLM to mimic.

These methods still deal with the limitations of LLMs, their probabilistic nature, which conflicts with the logical and precise rigor of mathematics. It underlines the challenges of tokenizing information in math problems and the degeneralization that occurs due to fine-tuning. We look at tackling this problem by enhancing the tokenization and proposing a specialized submodel for translating math problems into valid computational graphs.

3. Methodology

We explore the math word problem to computational graph machine translation problem with custom source and target vocabularies, a decoder-only transformer architecture, and a method of sampling that guarantees the proper construction of directed acyclic graphs (DAGs).

3.1. Vocabularies

For the purposes of machine translation, we consider the source vocabulary as English words along with five extra tokens (`<PAD>`, `<SOS>`, `<EOS>`, `<UNK>`, and `<NUM>`). Padding is used primarily during training to ensure batch elements have the same sequence length and are disregarded during loss calculation. All numbers are tokenized to `<NUM>`, encouraging the model to remain agnostic to the actual value of the number and rather focus on the fact that it is a number that can be referenced.

Our target vocabulary allows for the construction of computational graphs, taken from the MathQA linear formulations. Modeling our target language in such a way makes training and testing with the MathQA dataset easier. We provide operator (relation) nodes such as `add`, `log`, etc. We also consider constants, number references, and output references (all entities) that act as inputs to operator nodes. Constants include `const_2` and `const_pi`. Number references are references to numbers within the original word problem (`n0`, `n1`, etc.). Output references are references to previous operator node outputs (`#0`, `#1`, etc.).

The target language constructs DAGs linearly. Note that this functionally corresponds with mathematical equations (up to certain re-orderings). Consider the following equation:

$$(x_0 + x_1) \times 3 \tag{1}$$

This would be constructed sequentially in the target language as:

```
add n0 n1 multiply #0 const_3
```
(2)

A translational model, therefore, takes in tokens of the source language and predicts the tokens of the target language to be interpreted as steps towards constructing a computational graph.

We built a custom dataloader for the MathQA dataset to construct our respective languages in this manner.

3.2. Decoder-Only Transformer

We explore a decoder-only transformer architecture for seq2seq autoregression. Refer to Figure 1 for a diagram of the model DoM (Decoder-only Math). We have two versions of DoM, one with learned token embeddings with a vocabulary stripped from the MathQA dataset, and one with the GloVe word vector embeddings. We experimented with the usage of pretrained token embeddings to help generalize our model to more possible contexts. By forcing the model to accept pretrained word vectors, similar words will induce similar behavior from the model.

We start by embedding source tokens into 256-dimensional vectors for DoM and 300-dimensional vectors for DoM + GloVe. Normal DoM embeddings are learned through training, while GloVe embeddings are frozen. We add our custom tokens to the beginning of the vocabulary and set these to be learned during training. Target tokens are embedded by their own learned embedding layer into the same dimensionality. Each sequence is wrapped by `<SOS>` and `<EOS>` (embedded differently for the separate languages). Sinusoidal positional encoding is added to each sequence separately as well, inspired by the approach of Fu et al. [4]. The sequences are concatenated together to produce input to the transformer.

The concatenated sequence is then ran through an 8-layer transformer, each layer containing 8 heads of attention for DoM and 10 heads of attention for DoM + GloVe. The embedding dimension remains the same throughout the layers. Interestingly, we had originally experimented with a 12-layer transformer, but found training difficult. The problem may be small enough to warrant a less complex model.

We then split the output and discard those pertaining to the source sequence. Because we disregard source sequence outputs, the bottom half of our causal mask does not affect training or inference. We use a prefix causal mask to allow for future versions of DoM that predict over the source vocabulary as well.

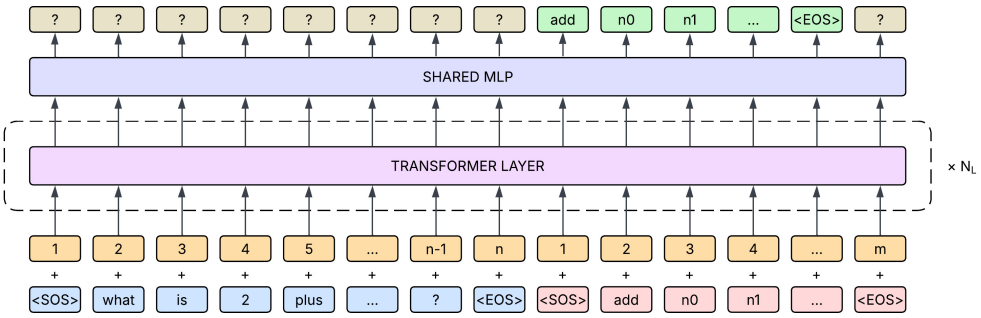


Figure 1. DoM decoder-only seq2seq transformer architecture.

The remaining target sequence output vectors are passed through a two-layer MLP (256/300, 1024, $|V_T|$)[2] with ReLU non-linearity. This classifies over the target vocabulary to predict next tokens per current tokens.

3.3. Training

The MathQA training dataset has 29,837 samples. The testing set has 2,985 samples and the validation set has 4,475 samples. We trained both models on the OSU DGX system for 30 epochs with a learning rate of 0.001 and weight decay regularization of 0.00001. Learning rate was modulated by a linear warmup for 10% of the epochs and cosine annealing cooldown for the remaining 90%.

During training, we logged training token-per-token loss and accuracy per batch (see Figures 2a and 3a). We also logged average validation token-per-token loss and accuracy per epoch (see Figures 2b and 3b).

Convergence of DoM was faster that DoM + GloVe, primarily because DoM + GloVe must learn around the frozen GloVe token embeddings. Once, however, the model learns to deal with these embeddings, it can leverage them properly.

3.4. Sampling Methods

Generative autoregressive models sample given input and append their predictions to form new input. We explore two methods of sampling, `argmax` and `smart_sample`. Some generative language models may see poor generations by sampling the highest probable word, and instead opt for a temperature-controlled nucleus sampling method. For our problem domain, sampling the most likely outcome is ideal, as we actually want the model to be deterministic in its output given an input.

`argmax` sampling simply chooses the most likely sample for each step, terminating at `<EOS>`. This works fine for many seq2seq problems, but does not guarantee the construction of valid DAGs.

`smart_sample` exploits the prior knowledge of graph construction to modify logit sampling during inference by masking specific token values. For example, if we start with a `<SOS>` token, we know as a prior to only generate operation nodes or the `<EOS>` token. If we, for example, generate `add`, we reference a lookup-dictionary to determine how many inputs the operator has (2 in this case), and then mask out all but entity tokens for two subsequent generations. We also parse through the input word problem and count how many numbers there are, and mask our vocabulary to ensure we do not reference numbers that were not given. Similarly, we decrease the mask of our output references as we generate more operation nodes in our DAG. `smart_sample` ensures that no matter what, our generation will at least be applicable to the given problem and can be followed through with computation. We will discuss performance differences between these two sampling methods in section 4.

Our unique sampling method for the construction of computational graphs is an artifact of the choice to generate over a combined vocabulary of relation and entity nodes. Original ideas for the architecture involved multiple heads that would be swapped during generation. However, we believed this would over-complicate the problem for the model and would lead to a harder time learning the sequential relation between nodes.

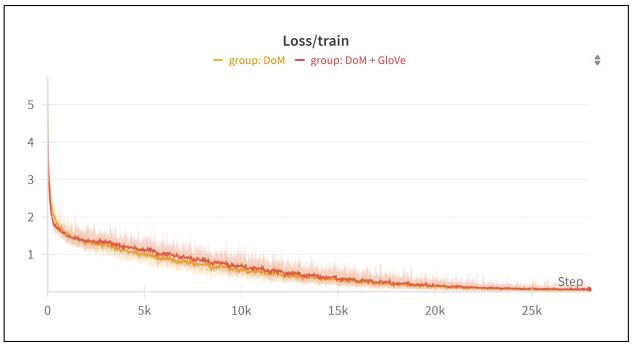
[2] $|V_T|$ is the size of the target vocabulary.

Model	Accuracy	BLEU	GED
DoM	76.18%	83.81%	—
DoM + GloVe	76.25%	83.79%	—

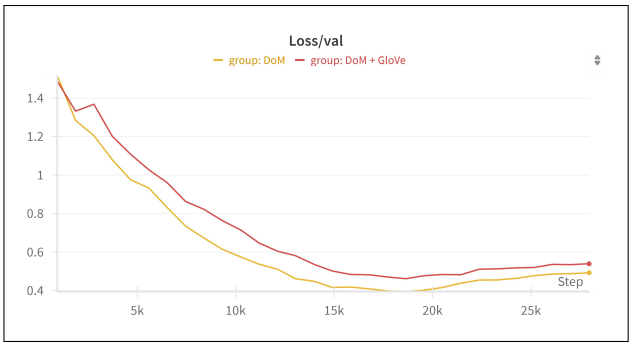
Table 1. Test dataset metrics with `argmax`.

4. Results

Our end-goal is, indeed, to ensure the correctness of the solution. However, the model's value comes from its ability to decipher computational structure from the word problem and construct the necessary equations to reach the solution. Correct graph constructions will result in correct answers guaranteed. Similar graphs may lead to slightly different

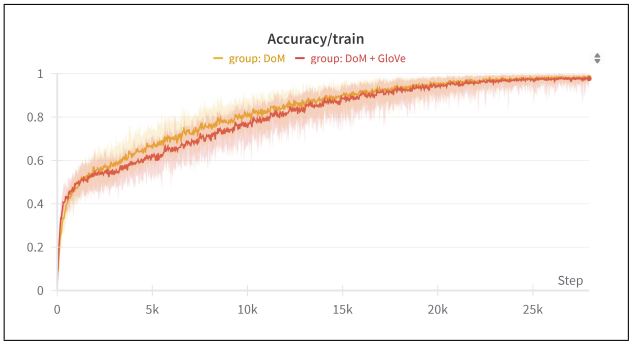


(a) Training set loss.

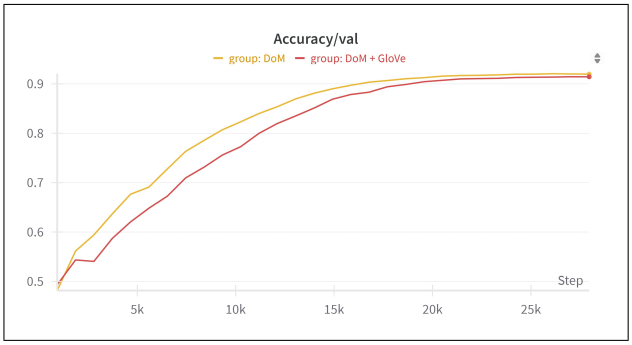


(b) Validation set loss.

Figure 2. Token-per-token loss during model training.



(a) Training set accuracy.



(b) Validation set accuracy.

Figure 3. Token-per-token accuracy during model training.

Model	Accuracy	BLEU	GED
DoM	77.45%	84.78%	2.72
DoM + GloVe	76.72%	84.06%	2.97

Table 2. Test dataset metrics with `smart_sample`.

Model	Accuracy	BLEU	GED
DoM	76.18%	83.81%	—
DoM + GloVe	75.96%	83.04%	—

Table 3. Validation dataset metrics with `argmax`.

Model	Accuracy	BLEU	GED
DoM	77.45%	84.78%	2.72
DoM + GloVe	76.45%	83.33%	3.11

Table 4. Validation dataset metrics with `smart_sample`.

answers, but reasoning may still be apparent. Therefore, comparing graph structure and generation is more insightful than analyzing the correctness of the answer.

4.1. Metrics

We evaluate our model on the MathQA test (see Tables 1 and 2) and validation (see Tables 3 and 4) data using three key benchmarks: pure accuracy, 2-gram BLEU score, and graph edit distance (GED). By pure accuracy, we mean if the model predicted the same complete DAG construction, it is correct, and if it produced even a single token wrong, it is incorrect. BLEU score is more forgiving, awarding the model for coming close to the same generation. Finally, GED (the number of changes needed to modify one graph to another) allows us to understand the similarity between generated and ground-truth graphs once constructed.

4.2. Evaluation Methods

We ran through each sample in the test and validation datasets, autoregressively generated target sequences, and compared those sequences with the ground-truth for each benchmark. `smart_sample` appears to perform better on our benchmarks than `argmax`, but only by a small margin. This would indicate that our models actually learned the sequential construction of DAGs well and can often construct them properly without guidance. Note that we only calculate GED with `smart_sample` as it guarantees proper

graph structures.

DoM and DoM + GloVe performed comparably on our benchmarks, but DoM often did slightly better. This is somewhat to be expected. DoM's source vocabulary is constructed from the MathQA dataset, and embeddings are learned to properly fit the dataset. DoM + GloVe uses the pretrained GloVe embeddings for a larger vocabulary. This helps the model generalize to a wider range of math problem contexts, but it loses the specificity of the MathQA question style. External testing with custom user-inputted prompts yielded more reasonable-sounding computational graphs from DoM + GloVe as opposed to DoM, though this is still to be explored. DoM + GloVe also must conform to the embeddings of the GloVe tokens, which may hold different semantic meanings within the context of a math word problem. As such, allowing slight modifications to the embeddings through training might help fine-tune the vector representations for the problem space.

5. Conclusion

Through this project, we familiarized ourselves with the application of decoder-only transformer frameworks for solving seq2seq problems. We specifically tackled English math word problems to computational graph machine translation, and constructed a custom dataloader and two models, evaluating them on a key metrics for their computational graph construction capabilities. Our goal from the beginning was never to create a model that competes with corporate fine-tuned LLMs, but rather to experiment and explore this area, building everything from scratch and learning in the process. We also aimed to suggest the use of small models, such as DoM, to be leveraged by an LLM for the offloading of domain-specific reasoning tasks.

Immediate future work would suggest experimenting with other pretrained embeddings, such as Word2Vec and BERT. For our current implementation, we would also like to train by allowing slight modification to pretrained embedding weights, as discussed in section 4. Later work includes the exploration of other seq2seq architectures, such as encoder-decoder transformers and the PALM framework [4]. Keeping models small in size is ideal, but we are also interested in the fine-tuning of a small-scale pretrained LLM for this generation task.

The largest limitation of this approach to mathematical reasoning is the scarcity of datasets. MathQA is simply the only large enough dataset for this specific seq2seq problem. In light of this, we consider experimenting with a multistage training cycle, switching between MathQA token-per-token DAG construction and then answer-informed training on the many other available math word problem datasets. In this way, the model would simultaneously ensure proper reasoning while also attempting to reach the correct solution on new problems.

References

[1] Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. Large language models for mathematical reasoning: Progresses and challenges, 2024. 1

[2] 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. *CoRR*, abs/1905.13319, 2019. 1

[3] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Trans. Mach. Learn. Res.*, 2023. 2

[4] Zihao Fu, Wai Lam, Qian Yu, Anthony Man-Cho So, Shengding Hu, Zhiyuan Liu, and Nigel Collier. Decoder-only or encoder-decoder? interpreting language model as a regularized encoder-decoder, 2023. 2, 5

[5] Zhenwen Liang, Wenhao Yu, Tanmay Rajpurohit, Peter Clark, Xiangliang Zhang, and Ashwin Kaylan. Let gpt be a math tutor: Teaching math word problem solvers with customized exercise generation, 2023. 1

[6] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. 2

[7] Mengxue Zhang, Zichao Wang, Zhichao Yang, Weiqi Feng, and Andrew Lan. Interpretable math word problem solution generation via step-by-step planning, 2023. 1

[8] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification, 2023. 2