Using DeepseekMath Model to Answer Complex Math Problems in LaTeX

Thomas Lawton

Abstract

This report explores the application of the DeepseekMath model to solve complex mathematical problems in LaTeX format. The model leverages advanced natural language processing and machine learning techniques to interpret and solve these math problems. By incorporating machine learning techniques such as self-consistency for best result selection, custom stopping criteria for iterative generation, prompt engineering, and time and debugging logs for resource and efficiency management, we can improve its accuracy and efficiency and then test the model on a series of complex mathematical problems. We will evaluate its performance on the AIMO Math Olympiad Competition test database.

Code -
https://drive.google.com/drive/folders/1yQb5a02QF4vfCzddICaaa1rJP03ndJei

Datasets -
Deepseek-Math:
https://github.com/deepseek-ai/DeepSeek-Math

AIMO:
https://www.kaggle.com/competitions/ai-mathematical-olympiad-prize/data

Introduction

In DeepseekMath, the concept of zero-shot learning is applied where the model uses its pre-trained knowledge to generate solutions for mathematical problems it has not encountered before, showing its capability to handle a wide range of problems without additional task-specific training.

DeepseekMath is used to interpret mathematical problem statements written in natural language, generate the corresponding code to solve these problems, and execute the code to obtain numeric answers. Let P be a mathematical problem expressed in natural language. The goal is to transform P into a solvable mathematical function or code C and then execute C to obtain the solution S,

i.e. given a problem P: P→ C→ S, where C is the code generated by the model to solve P and S is the solution. The transformation from P to C involves interpreting the natural language text, and generating the corresponding code with an engineered prompt, from which it can be executed and processed to return a numerical answer.

We have fine-tuned and applied the model employing several machine learning techniques, including self-consistency for best result selection, custom stopping criteria for iterative generation, prompt engineering, and time and debugging logs for resource and efficiency management to ensure the accuracy and efficiency of the solutions.

Methods

In this section, we delve into the methodologies employed in the project. Each subsection will provide an overview of the techniques and processes to illustrate how these methodologies were implemented and how they interact with each other.

Initialization and Environment

To initialize the model, the DeepseekMath model and tokenizer are loaded with a device map to distribute load across the GPUs. The notebook was run on the Kaggle Notebook environment, with the datasets mentioned above as the input to the environment. The notebook uses a main prediction function to iteratively call the model to generate multiple answers to each problem and select the most common answer. The notebook ran in 450 seconds total and outputs a submission.csv file for evaluation of results using the AIMO API.

Time-Tracking and Debugging

Time tracking is used throughout the notebook to monitor the duration of the notebook's runtime and ensure that the model does not exceed the maximum allowed. This is especially crucial when dealing with LLM's as they are computationally expensive. Additionally, debugging flags and messages are incorporated to identify and resolve issues during the model's testing and execution. These debugging tools help in tracing errors and ensure the correct functioning of the model.

Iterative Generation

When dealing with long and complex inputs such as the mathematical problems we are trying to solve, there is an increased chance of error or logical inconsistencies given the complexity of the input. Iterative generation is a crucial technique in our approach to handle these errors and inconsistencies. This method involves generating text and code in multiple stages, processing intermediate results, and including those results in subsequent generations. This is implemented using custom stopping criteria, in which the model will stop generating on certain stop words:

stop_words = [" ```output", " ```python", " ```\nOutput" , ")\n```" , " ``````output"]

Once the model reaches the stopping criteria, it will store the generated code to a cumulative code variable for continuing generation later. It will then process the code and execute it to return the numerical result. This result is then included in the prompt for continuing generation. The *old_values* (the key values from the previous generation) as well as the updated prompt (one of the model inputs), are used to continue generation as seen in figure 1.1.

```
# Continue generating with past key to prevent extra computation
if USE_PAST_KEY:
    old_values = generation_output.past_key_values
else:
    old_values = None

# Continue generation with old values stored from the first generation and with new prompt
generation_output = model.generate(**model_inputs,
                                   max_new_tokens=TOTAL_TOKENS-ALREADY_GEN,
                                   return_dict_in_generate=USE_PAST_KEY,
                                   past_key_values=old_values,
                                   do_sample = True,
                                   temperature = temperature_inner,
                                   top_p = top_p_inner,
                                   num_return_sequences=1, stopping_criteria = stopping_crit
```

figure 1.1

Iterative generation provides two main advantages. First, by iterating through the generation process, we can catch any errors in the generated code early and prevent unnecessary computations which are based on illogical reasoning from the model and will generate a suboptimal result. If the code execution fails during iterative generation, *code_output* is set to -1, and the code is not added to the *cumulative_code* variable so as to not include those results in the next generation, as shown in figure 1.2.

```
# Include output in prompt
if code_output!=-1:
    if (decoded_output[-len(")\n```"):]==")\n```"):
        prompt = decoded_output+'```output\n'+str(code_output)+'\n```\n'
    else:
        prompt = decoded_output+'\n'+str(code_output)+'\n```\n'
else:
    prompt = decoded_output
    cummulative_code=""
```

figure 1.2

Second, we can also improve accuracy by including the intermediate results so the model has a consistent memory, leading to more logical step by step explanations. As seen in figure 1.2, the *code_output*, which is derived from processing and executing the code generated by the model, is included in the next prompt. This immediate feedback allows the model to maintain consistency in its memory and explanations, leading to more accurate predictions.

Self-Consistency

To enhance reliability and accuracy of the generated answers, self-consistency is employed in the prediction function by generating multiple answers for each problem and selecting the most common result. By doing this, we can avoid the inherent variability of the model and filter out anomalies and outliers, leading to more dependable results.

```
# Best answer selection,take the most common answer and place in best
if len(outputs) > 0:
    occurances = Counter(outputs).most_common()
    print(occurances)
    if occurances[0][1] > best_count:
        print("GOOD ANSWER UPDATED!")
        best = occurances[0][0]
        best_count = occurances[0][1]
    if occurances[0][1] > 5:
        print("ANSWER FOUND!")
        break
```

figure 2.1

As shown in figure 2.1, after each repetition, the most common answer generated by the model is updated as the best answer and tracked with *best*. Once a specific answer is generated more than 5 times, it will return the answer as to avoid redundant

computation of similar results. Another way we avoid redundant computation is with a dynamic threshold.

```
# Early Exit Conditions (found best, over allowed runtime)
best, best_count = best_stats.get(i,(-1,-1))
if best_count>np.sqrt(jj):
    print("SKIPPING CAUSE ALREADY FOUND BEST")
    continue
```

figure 2.2

As shown in figure 2.2, if a result appears more times than the square root of the number of iterations, it is likely to be the most common answer, and the rest of the iterations are skipped. This allows enough repetitions to provide accurate results while limiting unnecessary computations.

Prompt Engineering

To provide a diverse set of answers for self-consistency, one of two separate prompts is chosen on each repetition.

```
# PROMPT ENGINEERING: 1 OF 2 SEPARATE PROMPTS WILL BE CHOSEN FOR EACH REPITION
code = """Below is a math problem you are to solve (positive numerical answer):
\"{}\"
To accomplish this, first determine a sympy-based approach for solving the problem by
listing each step to take and what functions need to be called in each step. Be clear
so even an idiot can follow your instructions, and remember, your final answer should
be positive integer, not an algebraic expression!
Write the entire script covering all the steps (use comments and document it well) and
\print the result. After solving the problem, output the final numerical answer within \\boxed

Approach:"""


cot = """Below is a math problem you are to solve (positive numerical answer!):
\"{}\"
Analyze this problem and think step by step to come to a solution with programs.
After solving the problem, output the final numerical answer within \\boxed{}.\n\n"""

promplt_options = [code,cot]
```

figure 3.1

Both prompts ask the model to think step by step to ensure logical reasoning and then generate code to solve the problem, placing the final answer within *\\boxed{}* for easy extraction of the numerical result using *process_text_output*. Using two separate prompts allows the model to give more diverse answers, which in combination with self-consistency for best result selection, can improve accuracy.

Fine-Tuning Hyperparameters

The *temperature* of the model is set relatively high to 0.9 to provide more diverse answers. Since self-consistency is employed to select the most common answer, a higher temperature helps as it allows a more diverse set of options for the self-consistency in the model to consider.

The number of repetitions, *n_reptitions*, represents the number of answers the model generates before selecting the best. Too many repetitions can be too computationally expensive and too little repetitions can lead to inaccurate results. *n_reptitions* is set to 17, as this is where the runtime was not too long.

Conclusion

The results of the test on the AIMO private database was 21/50. In this paper, we presented an approach to solving complex mathematical problems in LaTeX format using the DeepSeekMath model. Our methodology combines iterative generation, self-consistency, prompt engineering, and fine-tuning of hyperparameters to achieve higher accuracy and reliability in the predictions generated by the model.

Related Papers

Shao, Z. April 24, 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models arXiv:2402.03300v3 https://arxiv.org/pdf/2402.03300

Wang, X. March 7, 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models arXiv:2203.11171v4. https://arxiv.org/pdf/2203.11171

Acknowledgements

The notebook used for this paper was forked from: https://www.kaggle.com/code/suzchin/solution-baseline-deepseekmath-7b and fine-tuned.

Below are other Kaggle notebooks in which code was used or inspired from.

https://www.kaggle.com/code/alexandervc/deepseekmath-solution-explained-for-beginners/

https://www.kaggle.com/code/abdurrafae/improved-code-interpretation/

https://www.kaggle.com/code/olyatsimboy/aimo-zero-shot-sc-mmos-deepseekmath