Flan-T5-small LLM Hypertune on Hendrycks' Math Dataset

Artem Ivaniuk, Tyler Hogan

Math 495 – Final Project

https://github.com/MazaArt/math495

Table of Contents

Original Project Motivation

For our project, we were initially interested in training a large language model to classify mathematical conjectures/statements based on the type of proof that they require/most commonly use. With that in mind, we found several academic papers on the topic of LLMs in Mathematical proof writing:

- The Open Proof Corpus: A Large-Scale Study of LLM-Generated Mathematical Proofs, https://arxiv.org/abs/2506.21621
- NaturalProofs: Mathematical Theorem Proving in Natural Language, https://arxiv.org/abs/2104.01112
- ProofRM: A Scalable Pipeline to Train a Generalized Math Proof Reward Model, https://openreview.net/forum?id=ZZAIF9fjlU

As well as several data sets/data sources currently available online, with mathematical conjectures attached to their most famous proofs:

- DEMI-MathAnalysis Dataset, https://github.com/ziye2chen/DEMI-MathAnalysis
- ProofWiki, https://proofwiki.org/wiki/Main_Page
- Hugging Face - ProofLang Corpus, https://huggingface.co/datasets/proofcheck/prooflang

Unfortunately, as we explored the datasets and resources available for us, we encountered significant issues in terms of a lack of data standardization, a lack of commonality of tags on which we could train the LLM (example: no dataset contained a tag for what type of technique the proof relied on), and hardly compatible APIs for being able to access or download the data (specifically ProofWiki proved challenging to be incorporated into our project)

Updated Project Pursuit

As a result of the issues with the original idea, we pivoted to creating an LLM that is specifically trained to answer mathematical questions with solid reasoning and correct answers. We realize that this is a massive open problem in the field, with which even modern top-tier LLMs are struggling, as observed in OpenAI sharing their continual advances towards perfect reasoning of formal math in the GPT models: https://openai.com/index/formal-math/

Hence, we decided to specialize in a specific dataset of problems taken from "Measuring Mathematical Problem Solving With the MATH" Dataset by Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt, accessible at https://github.com/hendrycks/math, and aimed to train a Google flan-t5-small model from the Hugging Face library.

Project Structure

At its core, our repository is a workflow for loading the selected set of math datasets, converting them to a usable format, fine-tuning a smaller language model (the exact model choice can be swapped), and then evaluating the model's performance. We divided the codebase into several Python scripts, each of which handles a distinct step in this pipeline. The idea was to keep each file focused on a single responsibility, allowing us to experiment and debug individual components without breaking the rest of the system or having to recompute the time-expensive model training or data downloading. At a high level:

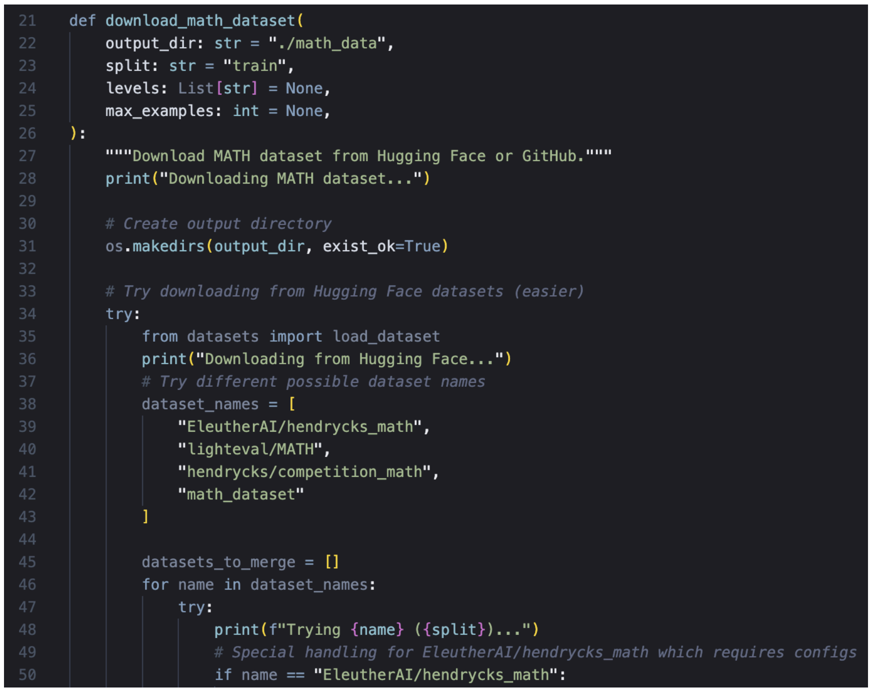
- download_math_data.py handles downloading and processing the original dataset from the "Measuring Mathematical Problem Solving With the MATH" Dataset

- load_data.py deals with loading the processed problems into a form that a model can actually train on.
- training.py performs the fine-tuning of the model and saves it to the mathqa_model directory, so we don't have to train the model anew for each use.
- Then, the inference.py script allows us to test the model interactively after training.

Individual Code Files

download_math_dataset.py

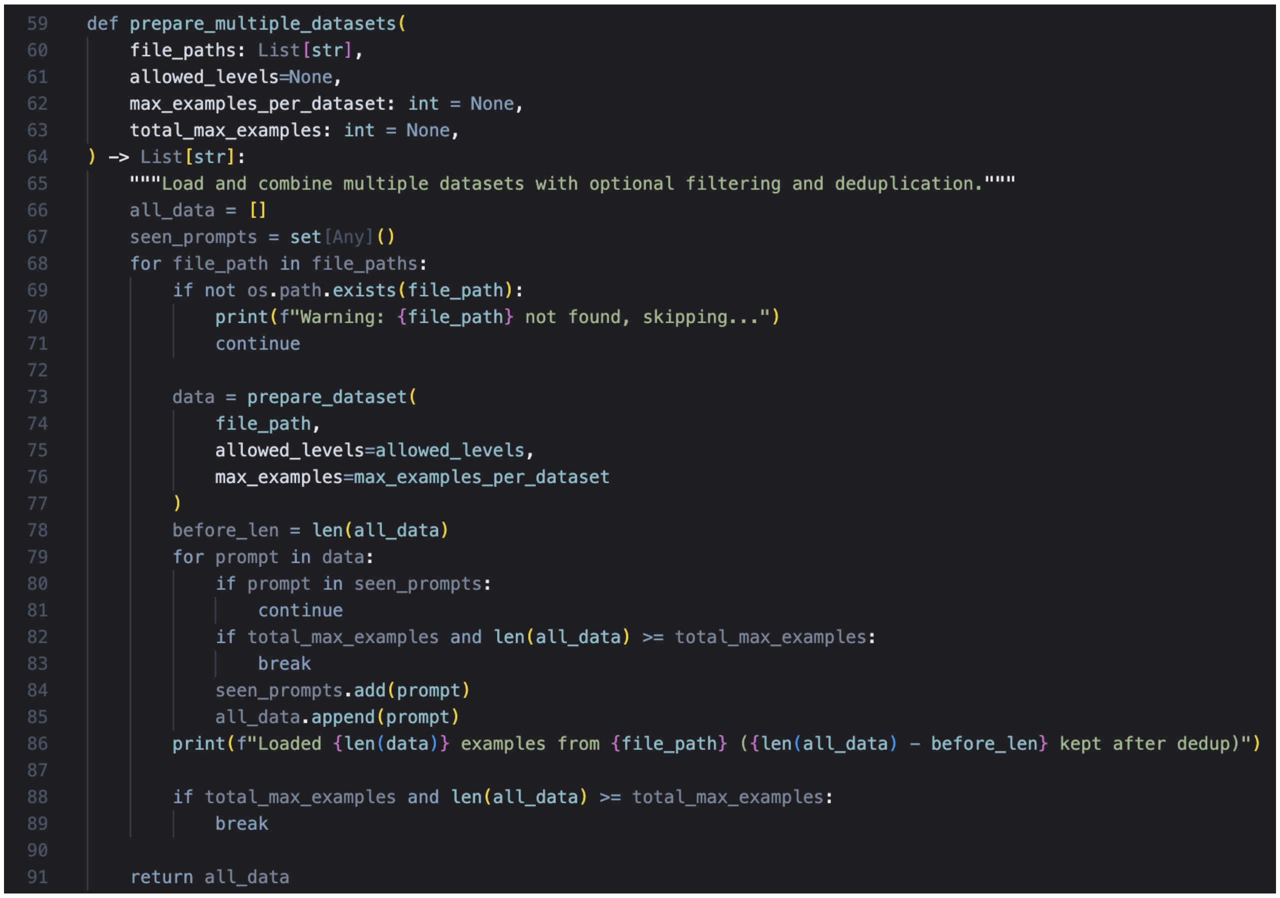
This script is responsible for retrieving a subset of the dataset. Due to compiling this project on our personal machines (MacBook Pro with an M1 chip), the whole dataset was too large (with a wide range of difficulty levels), so we restricted ourselves to a more manageable slice that still allowed us to test whether the model could learn to reason through multi-step problems. Through this script, we were able to extract only the information we wanted to analyze, thereby simplifying the data analysis process.



```python
def download_math_dataset(
    output_dir: str = "./math_data",
    split: str = "train",
    levels: List[str] = None,
    max_examples: int = None,
):
    """Download MATH dataset from Hugging Face or GitHub."""
    print("Downloading MATH dataset...")

    # Create output directory
    os.makedirs(output_dir, exist_ok=True)

    # Try downloading from Hugging Face datasets (easier)
    try:
        from datasets import load_dataset
        print("Downloading from Hugging Face...")
        # Try different possible dataset names
        dataset_names = [
            "EleutherAI/hendrycks_math",
            "lighteval/MATH",
            "hendrycks/competition_math",
            "math_dataset"
        ]

        datasets_to_merge = []
        for name in dataset_names:
            try:
                print(f"Trying {name} ({split})...")
                # Special handling for EleutherAI/hendrycks_math which requires configs
                if name == "EleutherAI/hendrycks_math":
```

load_data.py

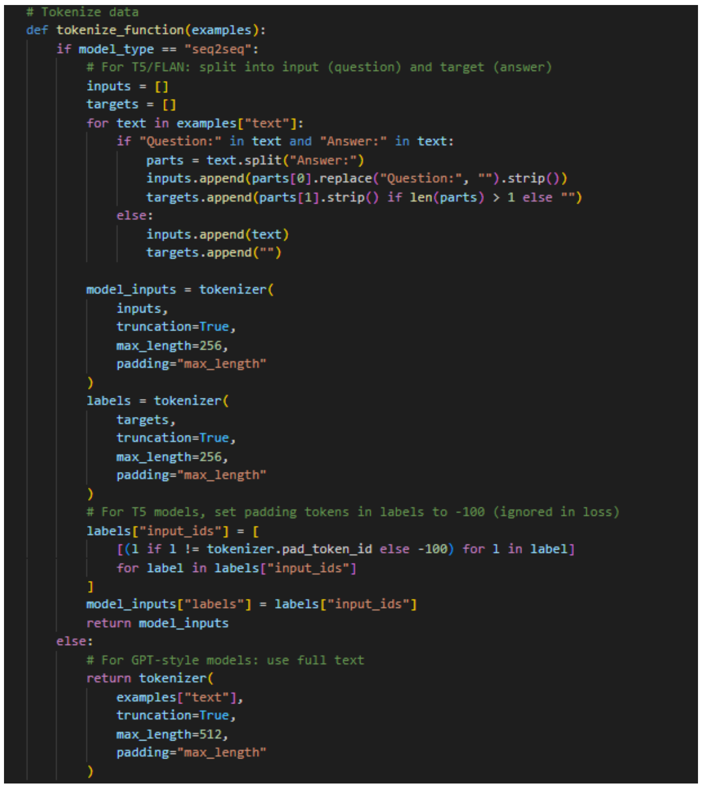
This file prepares the dataset for training. Because we are working with a seq2seq model (T5), we format each example as a prompt (“Solve the following math problem…”) followed by the ground-truth solution. The script handles tokenization and also filters out any examples that are too long for our model’s context window. The primary purpose of this file was to ensure that we could reliably feed batches into our training loop without encountering formatting issues.



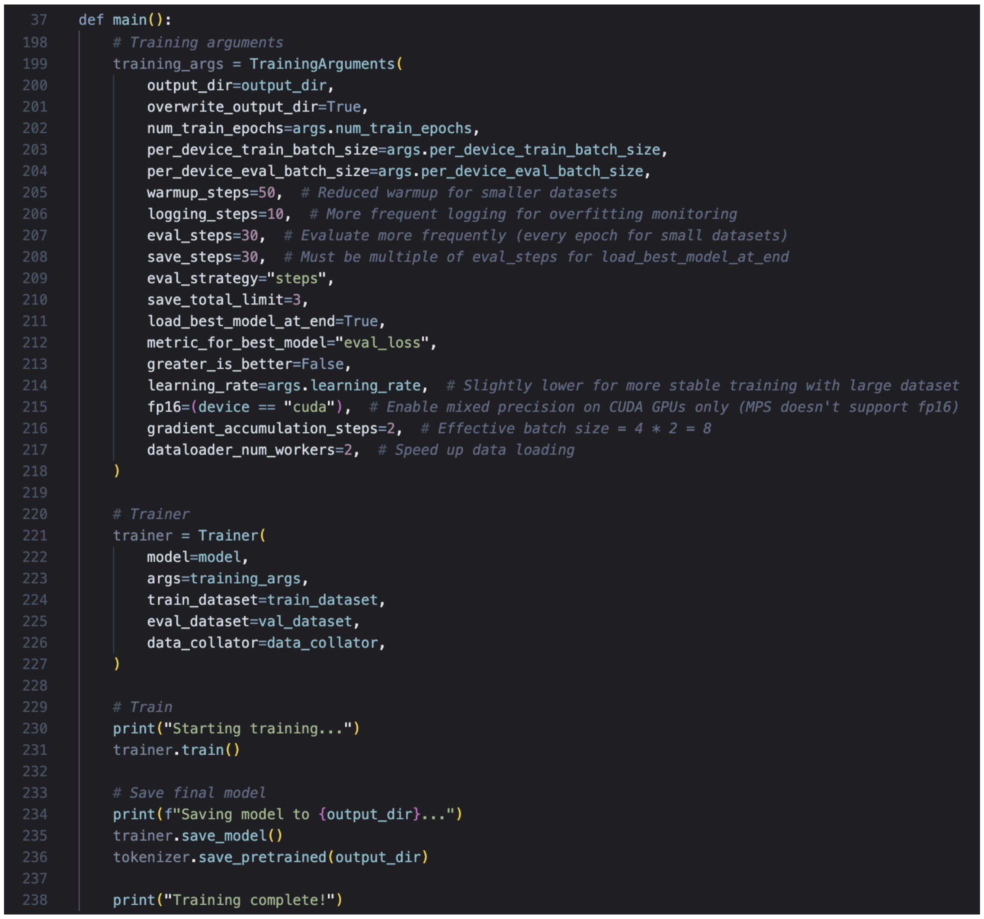
```python
def prepare_multiple_datasets(
    file_paths: List[str],
    allowed_levels=None,
    max_examples_per_dataset: int = None,
    total_max_examples: int = None,
) -> List[str]:
    """Load and combine multiple datasets with optional filtering and deduplication."""
    all_data = []
    seen_prompts = set[Any]()
    for file_path in file_paths:
        if not os.path.exists(file_path):
            print(f"Warning: {file_path} not found, skipping...")
            continue

        data = prepare_dataset(
            file_path,
            allowed_levels=allowed_levels,
            max_examples=max_examples_per_dataset
        )
        before_len = len(all_data)
        for prompt in data:
            if prompt in seen_prompts:
                continue
            if total_max_examples and len(all_data) >= total_max_examples:
                break
            seen_prompts.add(prompt)
            all_data.append(prompt)
        print(f"Loaded {len(data)} examples from {file_path} ({len(all_data) - before_len} kept after dedup)")

        if total_max_examples and len(all_data) >= total_max_examples:
            break

    return all_data
```

train.py

Here, we load the Flan-T5-small model, attach the dataset loader, and fine-tune the model using the Hugging Face Trainer API. More specifically, this file first parses command-line arguments, which allow us to specify batch sizes, learning rates, epochs, and other parameters. We then take our data, use an 80/20 train-test split, and tokenize it based on the model being used. Tokenization on the Flan-T5-small model splits each example in the data into a question and an answer. In this script, we also allow for the usage of a GPU, which we explain later in the report. It is here that we also control several other factors that affect the optimization process, all listed in training_args in the file. To keep training time reasonable, we used a relatively small batch size and a modest number of epochs. Still, even with these constraints, we were able to observe improvements in the model's reasoning quality over time. The training script also includes options for hyperparameter choices, logging, and model saving.

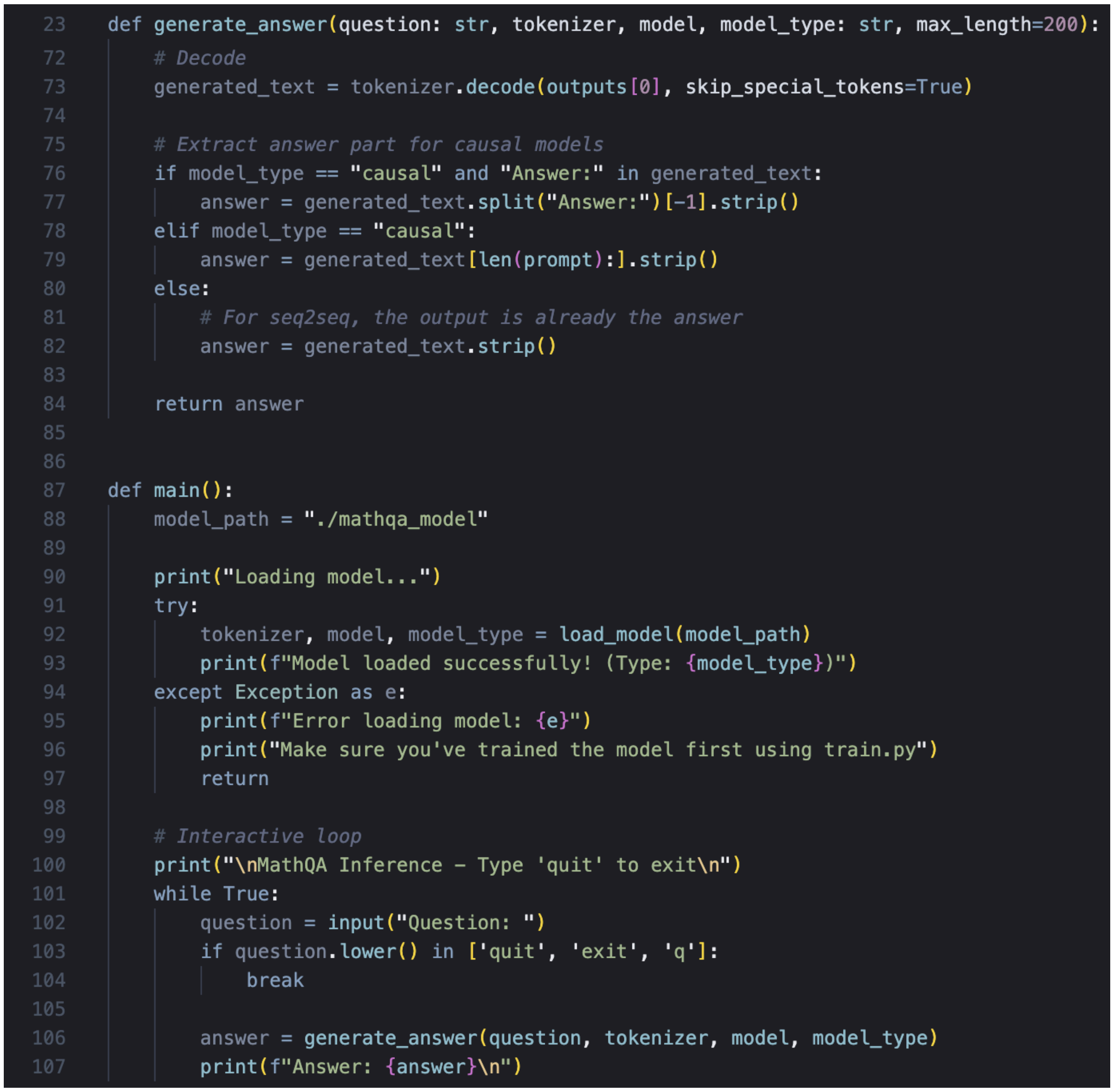


```python
# Tokenize data
def tokenize_function(examples):
    if model_type == "seq2seq":
        # For T5/FLAN: split into input (question) and target (answer)
        inputs = []
        targets = []
        for text in examples["text"]:
            if "Question:" in text and "Answer:" in text:
                parts = text.split("Answer:")
                inputs.append(parts[0].replace("Question:", "").strip())
                targets.append(parts[1].strip() if len(parts) > 1 else "")
            else:
                inputs.append(text)
                targets.append("")

        model_inputs = tokenizer(
            inputs,
            truncation=True,
            max_length=256,
            padding="max_length"
        )
        labels = tokenizer(
            targets,
            truncation=True,
            max_length=256,
            padding="max_length"
        )
        # For T5 models, set padding tokens in labels to -100 (ignored in loss)
        labels["input_ids"] = [
            [(l if l != tokenizer.pad_token_id else -100) for l in label]
            for label in labels["input_ids"]
        ]
        model_inputs["labels"] = labels["input_ids"]
        return model_inputs
    else:
        # For GPT-style models: use full text
        return tokenizer(
            examples["text"],
            truncation=True,
            max_length=512,
            padding="max_length"
        )
```


```
def main():
    # Training arguments
    training_args = TrainingArguments(
        output_dir=output_dir,
        overwrite_output_dir=True,
        num_train_epochs=args.num_train_epochs,
        per_device_train_batch_size=args.per_device_train_batch_size,
        per_device_eval_batch_size=args.per_device_eval_batch_size,
        warmup_steps=50,  # Reduced warmup for smaller datasets
        logging_steps=10,  # More frequent logging for overfitting monitoring
        eval_steps=30,  # Evaluate more frequently (every epoch for small datasets)
        save_steps=30,  # Must be multiple of eval_steps for load_best_model_at_end
        eval_strategy="steps",
        save_total_limit=3,
        load_best_model_at_end=True,
        metric_for_best_model="eval_loss",
        greater_is_better=False,
        learning_rate=args.learning_rate,  # Slightly lower for more stable training with large dataset
        fp16=(device == "cuda"),  # Enable mixed precision on CUDA GPUs only (MPS doesn't support fp16)
        gradient_accumulation_steps=2,  # Effective batch size = 4 * 2 = 8
        dataloader_num_workers=2,  # Speed up data loading
    )

    # Trainer
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        data_collator=data_collator,
    )

    # Train
    print("Starting training...")
    trainer.train()

    # Save final model
    print(f"Saving model to {output_dir}...")
    trainer.save_model()
    tokenizer.save_pretrained(output_dir)

    print("Training complete!")
```

inference.py

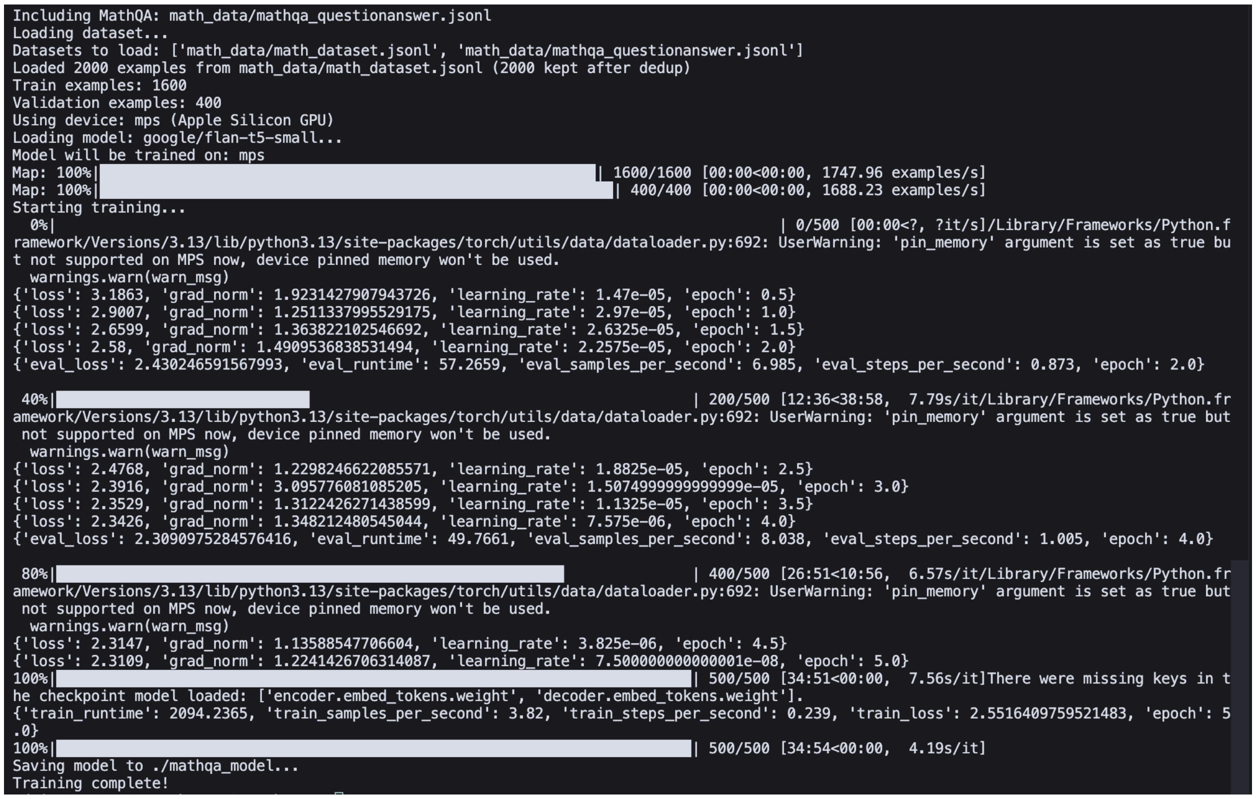
Once training was complete, we used this file to test the model through an interactive "conversation" in the terminal, where we would prompt questions and receive newly-generated answers in the window. The script simply loads the fine-tuned checkpoint and prints out the model's predicted reasoning/solution. With this, we got to check the model and compare before-and-after performance.



```python
def generate_answer(question: str, tokenizer, model, model_type: str, max_length=200):
    # Decode
    generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)

    # Extract answer part for causal models
    if model_type == "causal" and "Answer:" in generated_text:
        answer = generated_text.split("Answer:")[-1].strip()
    elif model_type == "causal":
        answer = generated_text[len(prompt):].strip()
    else:
        # For seq2seq, the output is already the answer
        answer = generated_text.strip()

    return answer


def main():
    model_path = "./mathqa_model"

    print("Loading model...")
    try:
        tokenizer, model, model_type = load_model(model_path)
        print(f"Model loaded successfully! (Type: {model_type})")
    except Exception as e:
        print(f"Error loading model: {e}")
        print("Make sure you've trained the model first using train.py")
        return

    # Interactive loop
    print("\nMathQA Inference - Type 'quit' to exit\n")
    while True:
        question = input("Question: ")
        if question.lower() in ['quit', 'exit', 'q']:
            break

        answer = generate_answer(question, tokenizer, model, model_type)
        print(f"Answer: {answer}\n")
```

Model Training

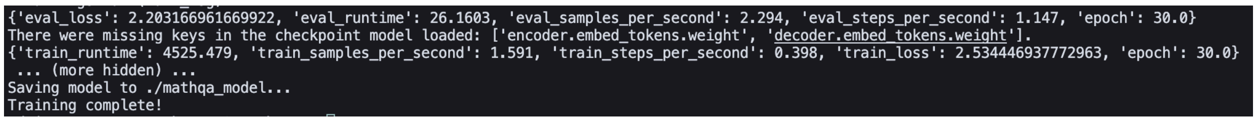
Attaching a screenshot of the terminal of an entire training run, apart from data load:



```
Including MathQA: math_data/mathqa_questionanswer.jsonl
Loading dataset...
Datasets to load: ['math_data/math_dataset.jsonl', 'math_data/mathqa_questionanswer.jsonl']
Loaded 2000 examples from math_data/math_dataset.jsonl (2000 kept after dedup)
Train examples: 1600
Validation examples: 400
Using device: mps (Apple Silicon GPU)
Loading model: google/flan-t5-small...
Model will be trained on: mps
Map: 100%|                                                      | 1600/1600 [00:00<00:00, 1747.96 examples/s]
Map: 100%|                                                        | 400/400 [00:00<00:00, 1688.23 examples/s]
Starting training...
  0%|                                                                                | 0/500 [00:00<?, ?it/s]/Library/Frameworks/Python.f
ramework/Versions/3.13/lib/python3.13/site-packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as true bu
t not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
{'loss': 3.1863, 'grad_norm': 1.9231427907943726, 'learning_rate': 1.47e-05, 'epoch': 0.5}
{'loss': 2.9007, 'grad_norm': 1.2511337995529175, 'learning_rate': 2.97e-05, 'epoch': 1.0}
{'loss': 2.6599, 'grad_norm': 1.363822102546692, 'learning_rate': 2.6325e-05, 'epoch': 1.5}
{'loss': 2.58, 'grad_norm': 1.4909536838531494, 'learning_rate': 2.2575e-05, 'epoch': 2.0}
{'eval_loss': 2.430246591567993, 'eval_runtime': 57.2659, 'eval_samples_per_second': 6.985, 'eval_steps_per_second': 0.873, 'epoch': 2.0}

 40%|                              | 200/500 [12:36<38:58,  7.79s/it/Library/Frameworks/Python.fr
amework/Versions/3.13/lib/python3.13/site-packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as true but
 not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
{'loss': 2.4768, 'grad_norm': 1.2298246622085571, 'learning_rate': 1.8825e-05, 'epoch': 2.5}
{'loss': 2.3916, 'grad_norm': 3.095776081085205, 'learning_rate': 1.5074999999999999e-05, 'epoch': 3.0}
{'loss': 2.3529, 'grad_norm': 1.3122426271438599, 'learning_rate': 1.1325e-05, 'epoch': 3.5}
{'loss': 2.3426, 'grad_norm': 1.348212480545044, 'learning_rate': 7.575e-06, 'epoch': 4.0}
{'eval_loss': 2.3090975284576416, 'eval_runtime': 49.7661, 'eval_samples_per_second': 8.038, 'eval_steps_per_second': 1.005, 'epoch': 4.0}

 80%|                                                      | 400/500 [26:51<10:56,  6.57s/it/Library/Frameworks/Python.fr
amework/Versions/3.13/lib/python3.13/site-packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as true but
 not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
{'loss': 2.3147, 'grad_norm': 1.13588547706604, 'learning_rate': 3.825e-06, 'epoch': 4.5}
{'loss': 2.3109, 'grad_norm': 1.2241426706314087, 'learning_rate': 7.500000000000001e-08, 'epoch': 5.0}
100%|                                                                           | 500/500 [34:51<00:00,  7.56s/it]There were missing keys in t
he checkpoint model loaded: ['encoder.embed_tokens.weight', 'decoder.embed_tokens.weight'].
{'train_runtime': 2094.2365, 'train_samples_per_second': 3.82, 'train_steps_per_second': 0.239, 'train_loss': 2.5516409759521483, 'epoch': 5
.0}
100%|                                                                           | 500/500 [34:54<00:00,  4.19s/it]
Saving model to ./mathqa_model...
Training complete!
```

Here, we ran 5 epochs on a 2000-question dataset from the GitHub repository and observed a decrease in loss from the initial 3.1863 to the final 2.3109. However, examining the train_runtime of 2094 seconds (35 minutes), we can see that the model training takes a very slow time, even on such a small dataset with a limited number of epochs. Hence, a major component of this project was optimizing the code and resources available to us (i.e., our GPU vs CPU) to load and train the best possible model.

As a final test, we ran a 30-epoch training cycle on 300 questions, which took 1.25 hours, showing a clear lack of quick convergence of the hypertune, but reducing loss from 3.1 to 2.2.

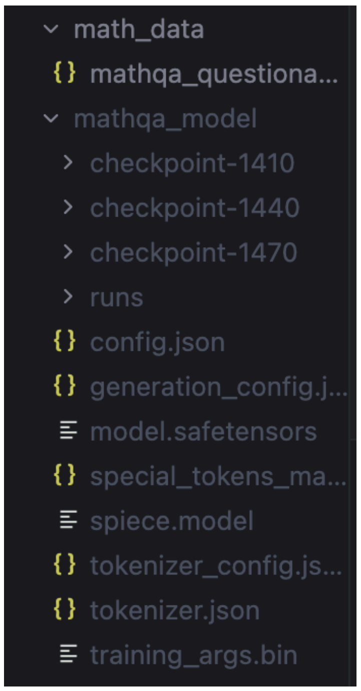


```
{'eval_loss': 2.203166961669922, 'eval_runtime': 26.1603, 'eval_samples_per_second': 2.294, 'eval_steps_per_second': 1.147, 'epoch': 30.0}
There were missing keys in the checkpoint model loaded: ['encoder.embed_tokens.weight', 'decoder.embed_tokens.weight'].
{'train_runtime': 4525.479, 'train_samples_per_second': 1.591, 'train_steps_per_second': 0.398, 'train_loss': 2.534446937772963, 'epoch': 30.0}
 ... (more hidden) ...
Saving model to ./mathqa_model...
Training complete!
```

Issues Encountered (the Fun Part)

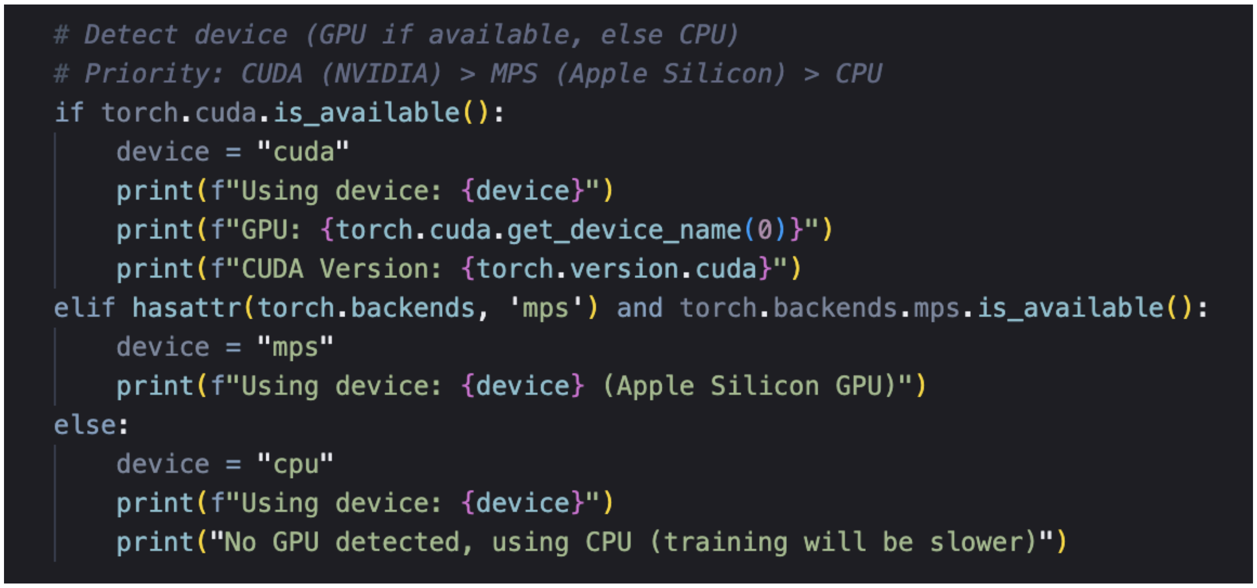
One of the biggest issues in AI is either the availability of datasets or the computational time required. For us, both of those problems were present because we did not have access to a professional computation machine or an infinite-sized data storage to access the entire original dataset.

Hence, our first solution involved breaking up the data loading, data cleaning, and training stages into separate scripts, and caching the results between each data loading and model training in the /math_data/ and /mathqa_model/ directories, respectively, as shown in the screenshot on the right.



```
⌄ math_data
  {} mathqa_questiona...
⌄ mathqa_model
  > checkpoint-1410
  > checkpoint-1440
  > checkpoint-1470
  > runs
  {} config.json
  {} generation_config.j...
  ≡ model.safetensors
  {} special_tokens_ma...
  ≡ spiece.model
  {} tokenizer_config.js...
  {} tokenizer.json
  ≡ training_args.bin
```

Then, our second component involved utilizing the GPU for model training, as it is capable of performing complex computations on a quicker scale. By searching for ideas on Stack Overflow, we were able to incorporate the following code into our database and transition the training from the CPU to the GPU, which expedited training time by approximately five times.

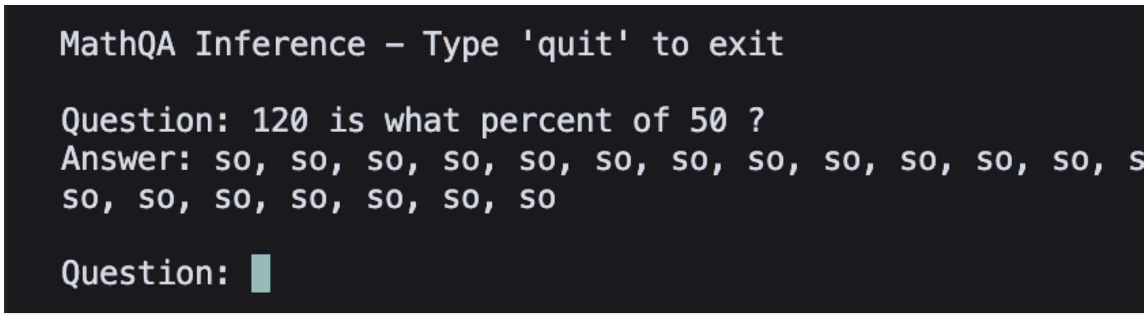


```python
# Detect device (GPU if available, else CPU)
# Priority: CUDA (NVIDIA) > MPS (Apple Silicon) > CPU
if torch.cuda.is_available():
    device = "cuda"
    print(f"Using device: {device}")
    print(f"GPU: {torch.cuda.get_device_name(0)}")
    print(f"CUDA Version: {torch.version.cuda}")
elif hasattr(torch.backends, 'mps') and torch.backends.mps.is_available():
    device = "mps"
    print(f"Using device: {device} (Apple Silicon GPU)")
else:
    device = "cpu"
    print(f"Using device: {device}")
    print("No GPU detected, using CPU (training will be slower)")
```

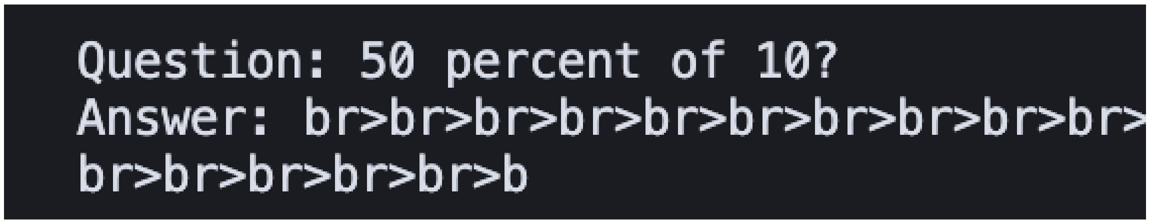
However, the time numbers provided in the Model Training section are for the GPU-enabled cases, illustrating just how slow the training process was.

Initial Testing

In the initial testing stages, particularly for the training agent with 5 epochs on the 2000-question data set, the output of the LLM was nowhere near in line with our expectations, clearly exhibiting hallucinations and not even providing a numeric output:



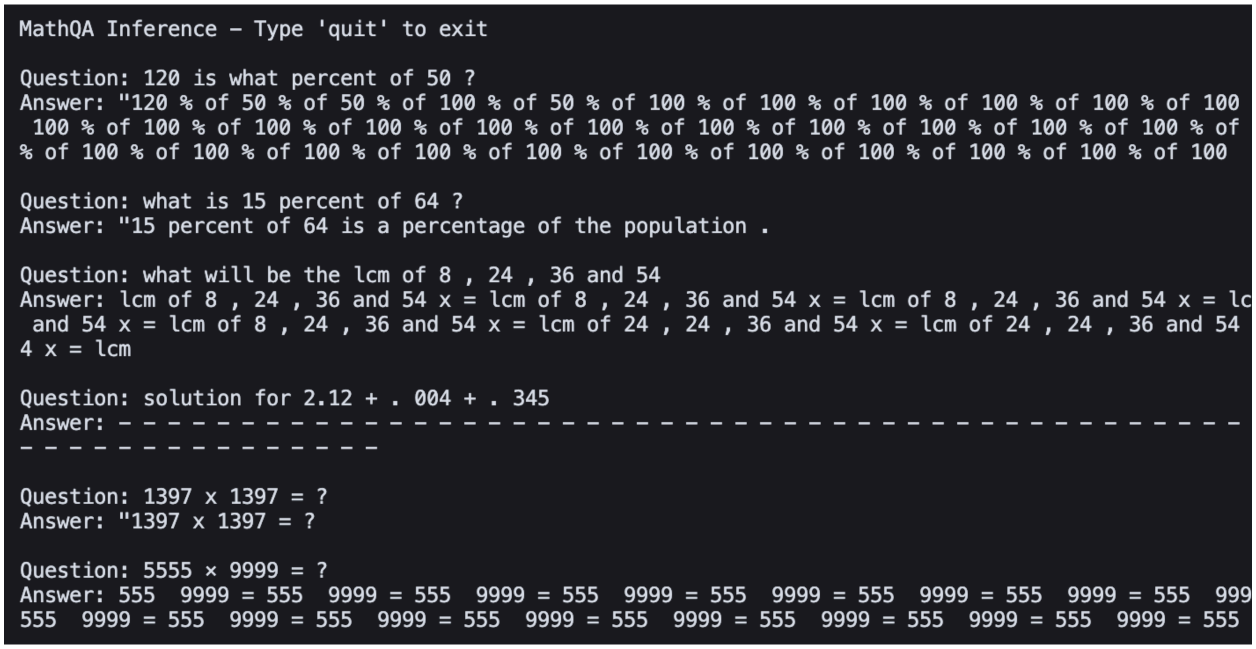
```
MathQA Inference - Type 'quit' to exit

Question: 120 is what percent of 50 ?
Answer: so, so, so, so, so, so, so, so, so, so, so, so, s
so, so, so, so, so, so, so

Question: 
```



```
Question: 50 percent of 10?
Answer: br>br>br>br>br>br>br>br>br>br>
br>br>br>br>br>b
```

Upon running a longer training period of 30 epochs on a smaller dataset, we expected to see an increase in the accuracy of the model, as it would be easier to process the smaller dataset in a quicker time scale, while still providing a good amount of specialization between the 300 questions in the smaller JSON.

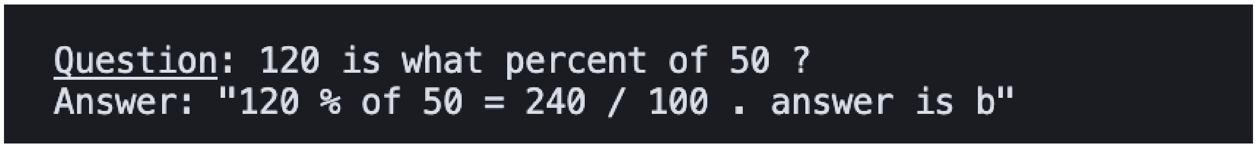
However, when testing this new model, it failed to draw any significant takeaways from the training data and randomly regurgitated the question itself (a common pattern found in the provided answers, as a way to start the solution) or provided a repeating sequence of random symbols or numbers, as seen in the screenshot below.



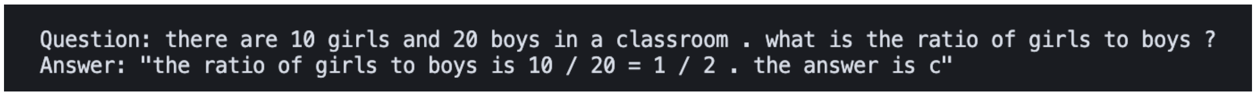
```
MathQA Inference – Type 'quit' to exit

Question: 120 is what percent of 50 ?
Answer: "120 % of 50 % of 50 % of 100 % of 50 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100
 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of
% of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100 % of 100

Question: what is 15 percent of 64 ?
Answer: "15 percent of 64 is a percentage of the population .

Question: what will be the lcm of 8 , 24 , 36 and 54
Answer: lcm of 8 , 24 , 36 and 54 x = lcm of 8 , 24 , 36 and 54 x = lcm of 8 , 24 , 36 and 54 x = lc
 and 54 x = lcm of 8 , 24 , 36 and 54 x = lcm of 24 , 24 , 36 and 54 x = lcm of 24 , 24 , 36 and 54
4 x = lcm

Question: solution for 2.12 + . 004 + . 345
Answer: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - -

Question: 1397 x 1397 = ?
Answer: "1397 x 1397 = ?

Question: 5555 × 9999 = ?
Answer: 555  9999 = 555  9999 = 555  9999 = 555  9999 = 555  9999 = 555  9999 = 555  9999 = 555  999
555  9999 = 555  9999 = 555  9999 = 555  9999 = 555  9999 = 555  9999 = 555  9999 = 555  9999 = 555
```

Final Results

After a final and longest training session that we could handle (2 hours, 50 epochs total), the final model trained on the smaller dataset provided noticeably more structured, correct answers with reasoning to the questions that we posed:



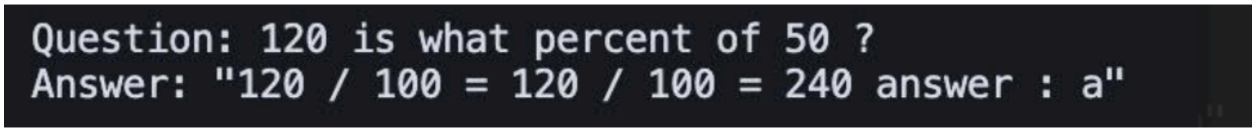
```
Question: 120 is what percent of 50 ?
Answer: "120 % of 50 = 240 / 100 . answer is b"
```



```
Question: there are 10 girls and 20 boys in a classroom . what is the ratio of girls to boys ?
Answer: "the ratio of girls to boys is 10 / 20 = 1 / 2 . the answer is c"
```

Unfortunately, it still continued to hallucinate components of the answers that were not applicable to the question posed, particularly "the answer is c" and "answer is b" parts in the screenshots included. This was likely a feature of the model overfitting to the most common patterns in the dataset found (aka having multiple answer choices labeled a, b, c) that the model proceeded to incorporate into its answers, even when such components were not applicable to the new questions it was exposed to.

Another common pattern we noticed was a large number of partially correct answers, which had the correct answers but included more unnecessary information than just the additional answer at the end. While the model was able to select the correct answer choice, it struggled more with the reasoning aspect. This makes sense, as coming up with a structured response to a question is clearly a harder task than just producing a single output.



```
Question: 120 is what percent of 50 ?
Answer: "120 / 100 = 120 / 100 = 240 answer : a"
```

Next Steps

Moving forward, we plan to make improvements by simply training the model on a larger portion of the dataset. Our current experiments have shown that our model only improves when we train for longer durations, with more epochs, and with slower learning rates. The main constraint we faced was time, as we couldn't train for extended periods while simultaneously optimizing other parts of our model. Another path we would like to explore is testing other models and evaluating their results, such as the Meta-Llama-3-8B-Instruct or the DeepSeek-R1-Distill-Qwen-7B model. These models, although not guaranteed to produce better results, may work more effectively on the entire set or a specific subset of problems, which we can test by feeding in certain categories of questions. In making these improvements, we will measure the quality of the results by tracking the number of hallucinations that occur along with the irrelevant information provided in correct responses (such as the "answer is b" from the example in the final results), although it is challenging to quantify some of these values. If we decide to make major improvements beyond those that allow us to use bigger datasets, there is a large amount of modern techniques available for us to implement that will enhance our results while maintaining the same training time. Examples of this include LoRA, QLoRA, and prefix tuning, which add a relatively small number of additional training parameters while keeping the original weights unchanged to increase the training quality marginally. Lastly, as the field continues to grow, new optimization methods for both the quality and speed of training are constantly introduced, offering more hope for reducing hallucinations while enhancing our reasoning quality.

Sources/Citations

Karpathy, Andrej. "Deep Dive into LLMs like ChatGPT." *YouTube*, YouTube,
www.youtube.com/watch?v=7xTGNNLPyMI. Accessed 10 Dec. 2025.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., & Steinhardt, J.
"Measuring Mathematical Problem Solving With the MATH Dataset."
arXiv:2103.03874, 2021. https://arxiv.org/abs/2103.03874

Chung, Hyung Won, et al.
"Scaling Instruction-Finetuned Language Models."
arXiv preprint arXiv:2210.11416, 2022.
https://arxiv.org/abs/2210.11416

Lhoest, Q., et al.
"Datasets: A Community Library for Natural Language Processing."
arXiv:2109.02846, 2021.

Hu, E. J., et al.

“LoRA: Low-Rank Adaptation of Large Language Models.”

arXiv:2106.09685, 2021.