Using LLMs to Identify Properties and Uses of a Molecule

Author(s): Hamid Shah and Damien Kula
Date: April 25, 2025

Problem Definition

Our goal of this project is to explore how a large language model can be trained to generate detailed approaches to a given research problem. The assignment is to provide a research method that is around 1 page in length. We were given a choice of any NLP problem we wanted to solve. We chose to focus on the following research question:

"How can we use an LLM to identify properties and uses of a molecule?"

To address this question, we use a dataset from hugging face, composed of short textual inputs representing molecule and general chemistry related queries and longer, more detailed outputs that describe the properties and applications of those molecules.

Dataset Curation

Dataset: https://huggingface.co/datasets/mlfoundations-dev/stackexchange_chemistry

Size: ~50K Rows

Dataset Structure

Each row in the dataset consists of:

Instruction: A user's original chemistry-related question or problem, often informal or context-heavy.

Completion: A textual response attempting to solve or explain the question in detail.

Conversations: The dialog history between "human" and "AI", typically showing two turns which consist of the question and the answer. This mainly serves as a more machine-readable format of the chat history with the model. This will not be utilized in our model training.

Reasoning for Dataset Choice:

This dataset is more general than some other ones we could have chosen, however, because we are only designing research approaches it is not entirely necessary to know everything about the topic of molecules. Besides this, most natural language data on molecules is in a format that requires advanced decoders to process. Given the size of the model, this is not a very realistic choice as the model itself does not have strong reasoning skills and we are working with limited hardware.

Dataset Splitting and Formatting:

In order to evaluate our model we went with a 90/10 split so we could evaluate a subset of the data. We also changed the format of our data for the finetuning process. For one, to prevent the model from rambling forever in training output we need to add an End-Of-Sequence token to each line of our dataset. In our data, we also did not care for the conversations column as it does not contribute to model understanding. So we filled the column with a set of null strings. We then recombine all sections of our edited dataset by calling a function formatting_prompts_func (this helps manage performance by only changing the dataset when necessary). The code below describes this process:

```
EOS_TOKEN = tokenizer.eos_token
if EOS_TOKEN is None:
    print("Warning: EOS_TOKEN is None. Adding a default one: '<|end_of_text|>'.")
    tokenizer.add_special_tokens({'eos_token': '<|end_of_text|>'})
    EOS_TOKEN = tokenizer.eos_token
    model.resize_token_embeddings(len(tokenizer))

# Load the raw dataset split
dataset = load_dataset("mlfoundations-dev/stackexchange_chemistry", split="train")

# --- Split the dataset BEFORE formatting ---
dataset_split = dataset.train_test_split(test_size=0.1, seed=42)  # 90% train, 10% eval
train_dataset = dataset_split['train']
eval_dataset = dataset_split['test']

print("Dataset loaded and split.")
print(f"Training examples: {len(train_dataset)}")
print(f"Evaluation examples: {len(eval_dataset)}")

def formatting_prompts_func(example):
    instructions = example["instruction"]
    inputs       = "" * len(instructions) #Here we are setting each position for the conversations co
    outputs      = example["completion"]
    texts = []
    for instruction, input, output in zip(instructions, inputs, outputs):
        # Must add EOS_TOKEN, otherwise your generation will go on forever!
        text = alpaca_prompt.format(instruction, input, output) + EOS_TOKEN
        texts.append(text)
    return { "text" : texts, }
```

Because we are dealing with a language model we also have a prompt associated with this formatted dataset as follows:

```
Below is an instruction that describes a task, paired with an input that
provides further context. Write a response that appropriately completes
the request.

### Instruction:
```

```
{instruction}

### Response:

{output}"""
```

LLM Selection and Training

Model: LLaMa 3 8B parameter model
Fine-tuning Framework: Unsloth

Why use LLaMa 3:

LLaMa 3 is a partially open-source model and not very resource-intensive compared with larger models meaning it is suited well for smaller setups and training at the cost of overall model accuracy. The 8 billion parameter model is the most straightforward to run as models get exponentially more expensive as we try to add more parameters. While the 8B model can be run on a single GPU, going to even the mid-tier model(40B) could require multiple high-end GPUs for finetuning, something we do not have the resources or budget for currently. As a result, LLaMa 3 tends to be more accessible and has more information on it than other much larger models and is a good starting point for experimentation. Also, from a time constraint standpoint, the fewer parameters allow us to make more adjustments to the training process as the total training time is shorter.

Unsloth Framework:

The main reason for using the Unsloth framework was its accessibility and immense documentation. Unsloth is an open-source project with lots of existing built-in compatibility with a variety of smaller-sized models. Since it is open source there is lots of documentation and existing code built on the framework which streamlines the coding and debugging process for our research problem. Most importantly, Unsloth made it very easy to run the LLaMa model without having to setup compatibility with our own hardware and downloading the model locally. This can be seen through its implementation in the code when creating a tokenizer.

```python
from unsloth import FastLanguageModel
import torch
from datasets import load_dataset, DatasetDict
from trl import SFTTrainer
from transformers import TrainingArguments, TextStreamer
from unsloth import is_bfloat16_supported
import math # Import math for perplexity calculation

# Configuration
max_seq_length = 2048
dtype = None # None for auto detection.
load_in_4bit = True # Use 4bit quantization to reduce memory usage. Can be False.


selected_model_name = "unsloth/Meta-Llama-3.1-8B"
print(f"Loading model: {selected_model_name}")

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = selected_model_name,
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)
```

Training Procedure:

The hardware utilized was a remote access Tesla 4 NVIDIA GPU which ended up handling the resource load of the model extremely efficiently, however, a major downside was the limited training time we could use on this GPU which prevented us from further runs and experimentation. In our training, LoRA adapters were used which speeds up fine-tuning by tuning the most significant parameters for our given model. This is in the range of 1-10% of all the parameters found within our model and these adapters are built-in with the FastLanguageModel module of Unsloth. Built into hugging face is a supervised fine-tuning trainer which we used to train our model. We trained the model for only 1 epoch due to both resource and time constraints. While this may be short the model does perform quite well with ever decreasing training loss.

Supervised Finetuner:

The finetuner uses AdamW as an optimizer. While very similar to Adam, the main difference is that AdamW decouples weight decay from the gradient step, instead applying it directly to the learning rate parameter. This finetuner utilizes the tokenizer and format established earlier when adjusting weights in the model.

Evaluation

Prompting and Human Evaluation:

One of the most important parts of model generation is making sure the prompt led to a desired output in the model. An issue with the small size of LLaMa 3-8B is that it loses context more quickly, meaning that generated text and the size of the text can quickly dwindle. Finetuning the model is meant to help this, but can only do so much. As a result, we ran multiple different prompts as an attempt to increase the length of the output and its accuracy. Using keywords such as “verbose” in our prompt were meant to push for higher word counts in the model. However, even so the model struggled to reach a 1-page length(like we wanted in the final experiments) likely because of the struggle to retain context. Our eventual chosen prompt is the one we felt had the most well-constructed answer.

Determining how good a language model is can be difficult due to the complexity and nuances in language. While negative log-likelihood and perplexity help quantify a models capabilities both of these methods are still flawed, relying on how well words are distributed against test data as opposed to the coherence of an output. So, looking at the output to see if it aligns with our goals was a major part of editing the model. Due to the limited compute prompt adjustments were the primary way we tried to change model output to better align with our goals.

Perplexity:

Perplexity is a common evaluation for finetuned models built off of the concept of crossentropy loss. The goal of this evaluation is to compare the distribution of our data with the distribution of our output to see if the model tends to be similar or produces a very randomized output. When testing our model for perplexity we got a value of ~6.0. While we didn’t measure the perplexity of our model without data initially, looking online we find that the perplexity of LLaMa 3-8B models range from a perplexity of 6-10 meaning our model falls in line with these averages. There are a variety of reasons why the model may not have a better perplexity score much of which can be attributed to model complexity and the amount of training. Unlike experimenting with prompting, to try and decrease the perplexity further than just with prompts it would likely require more training time or more model complexity which were severely limiting factors on the project.

Reflection

What We Learned:

This assignment was a very grounding experience in the amount of compute required to run an LLM successfully. However, besides the long training time and lack of more powerful hardware, we were able to learn an in-depth process of how to leverage LLMs for research purposes. The assignment itself has helped us realize that LLMs have many useful applications in research, beyond simply collecting sources.

Challenges and Improvements:

The two most significant challenges were model complexity constraints and hardware constraints. In the future, we should utilize more powerful hardware possibly requesting supercomputer access to help run and finetune the model. Model complexity is directly connected to hardware constraints. With more powerful hardware we can use more powerful models like LLaMa 3-70B which will likely produce longer and more relevant outputs from the finetuning process. To add on to this, fewer hardware constraints could allow us to explore more advanced methods of improving accuracy such as test-time compute to further align the model with our goals. Advanced prompt optimization could further help reduce overall model perplexity. It is possible to also improve model evaluation as well. The primary way to do this would be to improve human evaluation by having more people review the model itself. However, when accounting for the many other improvements that can be made with stronger hardware improved human evaluation may be unnecessary.