Final Project

Local LLM for Mac Silicon Devices

CSCI S-104 Advanced Deep Learning

Professor Zoran B. Djordjević
Name: Gassan Yacteen
Date: May 08, 2024

Abstract

In this report, I present the findings from my final project for the Advanced Deep Learning course. This project focused on developing and optimizing a local machine learning framework to run large language models (LLMs) on Apple Silicon devices. The motivation for this project stemmed from the need for runtime efficiency and the development of digital tools that prioritize user privacy, which is crucial for macOS users who do not have access to traditional GPU acceleration technologies such as CUDA.

The project began with the implementation of the LocalGPT and SiLLM tools, which are designed for efficient operation on Apple hardware while maintaining on-device data processing to enhance privacy. Despite their theoretical compatibility, LocalGPT encountered significant practical challenges including inefficiencies and performance limitations on Apple Silicon. This led to a strategic pivot to the mlx library, which proved to be more compatible with Apple's Metal API and better utilized the native hardware acceleration capabilities of the Apple Mac M2 Pro chip, equipped with 16 GPUs, 16 CPUs, and 16GB of RAM.

This hardware setup enabled the successful local training of the Llama3 Instruct model, subsequently fine-tuned on a mathematics dataset. The project not only showcased the feasibility of using locally executed machine learning frameworks on non-traditional hardware but also led to the implementation of a private LLM chatbot. This chatbot, designed to operate entirely within the local environment, exemplifies the project's aim to develop privacy-centric AI applications on Apple Silicon.

The project faced several challenges, including hardware limitations that restricted the scope of training capabilities and necessitated future upgrades for more extensive model training. However, the experience has been profoundly educational, offering valuable insights into local LLM deployment and the optimization of machine learning workflows on specialized hardware platforms.

Table of Contents

1. Introduction

Training and deploying large language models (LLMs) on devices powered by Apple Silicon poses unique challenges due to their lack of native CUDA support, a widely used API for GPU-accelerated computing. This project is motivated by two primary factors: the growing legislative emphasis on digital privacy, particularly highlighted by recent debates surrounding the approval of a new spying bill, which raises significant concerns about data security; and the technical hurdles associated with adapting machine learning frameworks to macOS environments. These challenges include efficiently utilizing Apple Silicon, which, while powerful, does not natively support CUDA, necessitating alternative approaches for effective machine learning execution.

Problem Statement. The core challenge addressed by this project involves training and using large language models (LLMs) on Apple Silicon devices, which traditionally lack support for CUDA. This technological gap is particularly critical considering recent legislative measures emphasizing the need for robust data privacy protections, which are more manageable on local systems as opposed to cloud-based platforms. The project's aim is to develop a machine learning framework that enables training and deployment of LLMs directly on Apple Silicon. Using the Mac M2 Pro's capabilities, the objectives are to surmount the hardware limitations, ensure superior computational performance, and uphold stringent data privacy standards. This endeavor will demonstrate Apple Silicon's viability as a platform for executing advanced AI tasks independently of external computing resources, aligning with strict privacy regulations.

In response to these challenges, the project evaluated two different technologies: LocalGPT and the MLX library. LocalGPT, designed to facilitate secure, local data processing, was initially considered due to its strong privacy features but faced significant performance limitations on Apple hardware, reducing its utility for the project's goals. Conversely, MLX emerged as a more compatible and efficient solution, surprisingly enabling the successful local training of the Llama3 model on an Apple Mac M2 Pro chip. This success not only highlighted the potential of MLX to fully harness the capabilities of Apple Silicon but also underscored the importance of local processing in maintaining data privacy and improving operational efficiency. The following sections will elaborate on the methodologies, challenges faced, and the strategic pivot that led to the successful implementation of the MLX framework in this setting.

2. Methodology

The methodology for this project was structured to address the dual objectives of demonstrating the feasibility of local LLM training on Apple Silicon and enhancing privacy through on-device data processing. The approach involved a series of steps to evaluate, implement, and refine the use of two distinct frameworks: LocalGPT and MLX. Here's a detailed breakdown of the methodologies used:

1. **LocalGPT Evaluation:**
   - **Setup and Configuration**: LocalGPT was set up on a Mac M2 Pro device to assess its compatibility and performance. The setup process included installing necessary dependencies and configuring the environment to support LocalGPT's operational requirements.
   - **Performance Testing:** I conducted a series of tests to evaluate LocalGPT's processing speed, response correctness, and resource utilization during tasks[1, 2, 3]. These tests helped identify the limitations in terms of computational efficiency and practical utility on Apple Silicon.
2. **Transition to MLX Library:**
   - **Library Selection and Setup:** Given the inefficiencies observed with LocalGPT, the project moved to use the MLX library, which was recently created at the end of November 2023 to address compatibility with Apple's MPS. The setup involved very simple and straightforward steps for environment preparation and dependency management[4].
   - **Local Training of Llama3 Model:** The core of the MLX methodology was the local training of the Llama3 model locally on my Mac M2 Pro chip[4]. This process included:
     - **2..1. Data Preparation:** Data was preprocessed to fit the training requirements of the MLX lora module and Llama3 model.
     - **2..2. Model Training:** The training parameters were selected, to not cause my computer to crash, which it did many times, based on my Mac's hardware see bullet 2..1[5].

2..3. **Evaluation and Fine-Tuning:** Post initial training, the model's performance was tested and metrics such as accuracy response time were noted[6, 7].

3. **Implementation and Testing:**
   - **Integration into Local Workflows:** Once trained, the Llama3, also known as the LlaMATH3 model, was integrated into a GUI for a user-friendly interactive chatbot experience[8].
   - **Performance Benchmarking:** Tests were again conducted to compare the performance of the LlaMATH3 model against the initial benchmarks set by LocalGPT.
4. **Documentation and Analysis:**
   - **Recording Results:** All configurations, processes, and results were documented to provide a clear trail of the methodologies applied and the findings obtained.
   - **Analysis:** The final step involved a detailed analysis of the comparative performance of LocalGPT and MLX, drawing conclusions on the effectiveness of the MLX library for local LLM training on Apple Silicon.

This methodology ensured a thorough evaluation of the tools and techniques necessary for achieving the project's goals, providing a robust framework for further research and development in this area.

3. System Setup

3.1 Computer Hardware Utilized

The project was carried out on an Apple Mac M2 Pro chip, which is equipped with 16 GPUs and 16 CPUs. This hardware configuration is particularly notable for its high computational power and efficiency, making it well-suited for machine learning tasks that require intensive data processing and model training capabilities. The system also includes 16 GB of RAM.

3.2 LocalGPT Setup

For the LocalGPT component, there is a detailed installation and setup process outlined in the supplemented Jupyter notebook. This documentation provides comprehensive guidance on getting LocalGPT to work properly on Apple Silicon devices. Given the extensive nature of this setup process, please refer to the LocalGPT GitHub repository or the Jupyter Notebook for a step-by-step guide. One can also view the Appendix section.[1, 3]

3.3 MLX Setup

3.3.1 Environment and Software Requirements

Before proceeding, ensure the following software and libraries are installed as per Table 1 below[4]:

Software & Libraries	Version	Installation
Conda	Conda 24.4.0	Website
Python	3.11.5	Website
MLX LM	0.12.1	pip install mlx-llm
PyTorch	2.3.0	pip install torch
Transformers	4.40.1	pip install transformers
DataSets	2.19.0	pip install datasets
Pandas	2.2.2	pip install pandas
JSON	2.0.9	----

Table 1: Installation software and python libraries.

3.3.2 Setting up the MLX Local Chatbot GUI

To set up the Local Chatbot GUI for the MLX framework, follow the instructions below, create and activate a new conda environment, and install the required packages[8]. Clone the necessary GitHub repository, navigate to the cloned directory:

```
1. git clone https://github.com/qnguyen3/chat-with-mlx.git
2. cd chat-with-mlx
3. conda create -n mlx-chat python=3.11
4. conda activate mlx-chat
5. pip install -e .
```

Then replace the `app.py` file and add the `LlaMATH-3-8B-Instruct-4bit.yml` configuration file in the `../chat-with-mlx/chat_with_mlx/models/config` directory to integrate the custom interface and utilize the trained LlaMATH3 model in the GUI.

This setup provides the necessary environment to leverage the advanced capabilities of mlx-llm on Apple Silicon, enabling local training of LLMs.

4. About the Data

For this project, we utilized the gsm8k dataset accessed via the Hugging Face datasets library. This dataset comprises various word arithmetic math problems[9]. These problems are well-suited for training a language model to comprehend and solve mathematical queries expressed in natural language. Table [2] displays the top 5 values of the training dataset.

4.1 Data Preparation and Formatting for MLX Training

The dataset was initially loaded and converted into Pandas DataFrames for easy manipulation, with each record comprising a *question*-and-answer pair. The answers include the solution and the computational steps formatted for clarity.

	question	answer
0	Natalia sold clips to 48 of her friends in Apr...	Natalia sold 48/2 = <<48/2=24>>24 clips in May...
1	Weng earns $12 an hour for babysitting. Yester...	Weng earns 12/60 = $<<12/60=0.2>>0.2 per minut...
2	Betty is saving money for a new wallet which c...	In the beginning, Betty has only 100 / 2 = $<<...
3	Julie is reading a 120-page book. Yesterday, s...	Maila read 12 x 2 = <<12*2=24>>24 pages today....
4	James writes a 3-page letter to 2 different fr...	He writes each friend 3*2=<<3*2=6>>6 pages a w...
...	...	...

Table 2: Top five values of the training dataset utilized in training our LlaMATH3.

To prepare the data for training with the MLX library, we transformed each question-and-answer pair into a single line JSON-like structure[6]. This transformation involved creating a custom function that processed each row of the dataset, resulting in entries formatted as follows:

```
{"text": "\nQ: [question text here] \nA: [answer text here]"}
```

This structured format allows the model to easily distinguish between the question and its corresponding answer, a necessary step for MLX learning. The processed data was saved in `jsonl` files, which support large data sets and are efficient for sequential processing. These files were then saved as, `train.jsonl`, which can be seen below, and `valid.jsonl` sets to support the model's training, tuning, and evaluation phases.


```
{"text": "Q: A fruit stand is selling apples for $2 each. Emmy has $200 while Gerry has $100.
If they want to buy apples, how many apples can Emmy and Gerry buy altogether? \nA: Emmy and
Gerry have a total of $200 + $100 = $<<200+100=300>>300.\nTherefore, Emmy and Gerry can buy
$300/$2 = <<300/2=150>>150 apples altogether.\n#### 150"}
{"text": "Q: At the height of cranberry season, there are 60000 cranberries in a bog. 40% are
harvested by humans and another 20000 are eaten by elk. How many cranberries are left? \nA:
First find the total number of cranberries harvested by humans: 60000 cranberries * 40% =
<<60000*40*.01=24000>>24000\nThen subtract the number of cranberries taken by the humans and
elk to find the number remaining: 60000 cranberries - 24000 cranberries - 20000 cranberries =
<<60000-24000-20000=16000>>16000 cranberries\n#### 16000"}

… … … … … … … … …

… … … … … … … … …
```


5. Model Development

Here we will discuss the development of the LLM model utilizing MLX library. Given the nature of training LLMs, particularly on a platform without native CUDA support, the training process was optimized to ensure efficiency while avoiding overloading the system's resources. The model training was conducted using the `mlx_lm.lora` module, specifically designed for LLMs like `Meta-Llama-3-8B-Instruct-4bit[4]`. The training parameters were set to fine-tune the model locally on an Apple Silicon architecture efficiently:

```
!python -m mlx_lm.lora \
    --model mlx-community/Meta-Llama-3-8B-Instruct-4bit \
    --train \
    --batch-size 1 \
    --lora-layers 1 \
    --iters 1000 \
    --data Data \
    --seed 0
```

Running the code above will begin the training process and create a folder called adapters which can be used to easily use your fine-tuned model. The model training was conducted using the `mlx-community/Meta-Llama-3-8B-Instruct-4bit`, where the mlx_lm is a python module that allows you to fine tune lora layers, with a batch size set to 1 to minimize memory usage. To optimize learning without extensive computational overhead, LoRA layers were limited to 1, these employ low-rank matrices that efficiently modify specific weights within the model. The training involved 1000 iterations, also known as epochs[7]. Additionally, the data were formatted as stated above into `jsonl` files containing math problems, which were organized in a subfolder named *"Data"*. This approach facilitated the training process tailored to the capabilities of Apple Silicon.

Figure 1: The left image shows the CPU usage over time, and the right image displays the GPU usage. The smaller rectangle illustrates the current utilization of CPU usage on the Mac device. You can observe that before training, the GPU usage is zero, but it spikes to maximum capacity right after training begins.

Given the demands of training large language models, efficient resource utilization is important. My Apple M2 Pro's capabilities were fully utilized, with monitoring of both efficiency and performance cores as in figure [1]. Adjustments in batch sizes and LoRA layers were crucial for maintaining system stability, preventing crashes that were noted in early trials due to resource overloads[5]. Proactive measures were also implemented to manage and mitigate the common `Internal Error` issues associated with the hardware limitations of my device. The training process, which can be seen in table [2], included iteration-based evaluations, allowing for continuous monitoring of the model's performance and system resource utilization. Validation and training loss were periodically reported to assess progress and guide necessary adjustments, ensuring a thorough understanding and optimization of the training dynamics.

```
Loading pretrained model
Fetching 6 files: 100%|████████████████████| 6/6 [00:00<00:00, 19941.22it/s]
Special tokens have been added in the vocabulary, make sure the associated word embeddings are fine-tuned or trained.
Trainable parameters: 0.001% (0.106M/8030.261M)
Loading datasets
Training
Starting training..., iters: 1000
Iter 1: Val loss 1.611, Val took 27.126s
Iter 10: Train loss 1.633, Learning Rate 1.000e-05, It/sec 1.074, Tokens/sec 177.374, Trained Tokens 1651, Peak mem 5.806 GB
Iter 20: Train loss 1.827, Learning Rate 1.000e-05, It/sec 1.057, Tokens/sec 147.796, Trained Tokens 3049, Peak mem 5.806 GB
Iter 30: Train loss 1.671, Learning Rate 1.000e-05, It/sec 0.937, Tokens/sec 147.794, Trained Tokens 4627, Peak mem 5.806 GB

... ... ... ... ... ... ... ... ... ... ...

... ... ... ... ... ... ... ... ... ... ...
Iter 180: Train loss 1.363, Learning Rate 1.000e-05, It/sec 0.950, Tokens/sec 155.042, Trained Tokens 28481, Peak mem 5.974 GB
Iter 190: Train loss 1.434, Learning Rate 1.000e-05, It/sec 0.965, Tokens/sec 156.581, Trained Tokens 30104, Peak mem 5.974 GB
Iter 200: Train loss 1.407, Learning Rate 1.000e-05, It/sec 1.101, Tokens/sec 147.931, Trained Tokens 31447, Peak mem 5.974 GB
Iter 200: Val loss 1.297, Val took 27.229s
Iter 200: Saved adapter weights to adapters/adapters.safetensors and adapters/0000200_adapters.safetensors.

... ... ... ... ... ... ... ... ... ... ...

... ... ... ... ... ... ... ... ... ... ...
Iter 600: Val loss 1.253, Val took 24.008s
Iter 600: Saved adapter weights to adapters/adapters.safetensors and adapters/0000600_adapters.safetensors.
Iter 610: Train loss 1.192, Learning Rate 1.000e-05, It/sec 0.973, Tokens/sec 150.005, Trained Tokens 97560, Peak mem 6.169 GB
Iter 620: Train loss 1.255, Learning Rate 1.000e-05, It/sec 0.911, Tokens/sec 155.089, Trained Tokens 99263, Peak mem 6.169 GB
Iter 630: Train loss 1.080, Learning Rate 1.000e-05, It/sec 1.047, Tokens/sec 153.951, Trained Tokens 100734, Peak mem 6.169 GB
Iter 640: Train loss 1.260, Learning Rate 1.000e-05, It/sec 0.913, Tokens/sec 158.399, Trained Tokens 102468, Peak mem 6.169 GB
Iter 650: Train loss 1.175, Learning Rate 1.000e-05, It/sec 0.959, Tokens/sec 161.248, Trained Tokens 104150, Peak mem 6.169 GB

... ... ... ... ... ... ... ... ... ... ...

... ... ... ... ... ... ... ... ... ... ...
Iter 770: Train loss 1.094, Learning Rate 1.000e-05, It/sec 1.113, Tokens/sec 151.275, Trained Tokens 122816, Peak mem 6.336 GB
Iter 780: Train loss 1.176, Learning Rate 1.000e-05, It/sec 0.943, Tokens/sec 157.287, Trained Tokens 124484, Peak mem 6.336 GB
Iter 790: Train loss 1.257, Learning Rate 1.000e-05, It/sec 0.958, Tokens/sec 157.040, Trained Tokens 126124, Peak mem 6.336 GB
Iter 800: Train loss 1.121, Learning Rate 1.000e-05, It/sec 1.028, Tokens/sec 153.689, Trained Tokens 127619, Peak mem 6.336 GB
Iter 800: Val loss 1.182, Val took 23.772s
Iter 800: Saved adapter weights to adapters/adapters.safetensors and adapters/0000800_adapters.safetensors.
```

```
… … … … … … … … … … … …
… … … … … … … … … … … …
Iter 1000: Saved adapter weights to adapters/adapters.safetensors and adapters/0001000_adapters.safetensors.
Saved final adapter weights to adapters/adapters.safetensors.
```

Table 2: Here we observe a decrease in the validation loss until Iteration 800, followed by a slight increase up to 1000. After testing the model's outputs at both iteration values, I determined that the model's results were far better at Iteration 1000.

The successful integration of the Meta-Llama model using the MLX framework on Apple Silicon was highly impressive. It demonstrated not only the feasibility but also the potential for sophisticated AI applications on platforms that emphasize data privacy and local processing. This phase of model development was pivotal in exploring the capabilities and limits of Apple Silicon, significantly advancing our understanding of machine learning possibilities on non-traditional hardware platforms.

6. LlaMATH3 Interaction

The interaction with the LlaMATH3 model was conducted through scripted commands that leverage the `mlx_lm.generate` function to elicit responses to math problems. For simpler queries, such as calculating the total number of bolts required from given materials, the script prompts the model with a clear question, to which the model responds by breaking down the calculation and providing the answer. Note that adapters are trainable modules added to pre-trained models, enabling task-specific fine-tuning by modifying only certain parts of the model architecture without retraining the entire network[4]. Our LlaMATH3 example and its outputs are listed below:

```
!python -m mlx_lm.generate --model "mlx-community/Meta-Llama-3-8B-Instruct-4bit" \
               --adapter-path adapters \
               --max-tokens 256 \
               --prompt "Q: A robe takes 2 bolts of blue fiber and half that much white fiber.
How many bolts in total does it take?" \
               --seed 0
```

```
==========
Prompt: <|begin_of_text|><|start_header_id|>user<|end_header_id|>

Q A robe takes 2 bolts of blue fiber and half that much white fiber.  How many bolts in total
does it take?<|eot_id|><|start_header_id|>assistant<|end_header_id|>


There are 2 bolts of blue fiber. Half of that is 1. So it takes 2 + 1 = 3 bolts of fiber.
==========
Prompt: 54.412 tokens-per-sec
Generation: 37.263 tokens-per-sec
```

Note that the actual test data gives the following output.

```
Q It takes 2/2=<<2/2=1>>1 bolt of white fiber
So the total amount of fabric is 2+1=<<2+1=3>>3 bolts of fabric
#### 3
```

6.1 Advanced Interaction

To address a more complex problem regarding savings and earnings, the model processes the details of the scenario and calculates the total amount Annabelle needs to save after considering her current savings and earnings.

```
# MLX Community imports
from mlx_lm import load, generate

# Load the fine-tuned model with LoRA weights
model_lora, tokenizer_lora = load(
    path_or_hf_repo="mlx-community/Meta-Llama-3-8B-Instruct-4bit",
    adapter_path="adapters"
)

# Test Trained LLM with Test Prompt
prompt = test_df['question'][354]
print(f"Our test prompt:")
print(f"Q {prompt}")

# Testing model with prompt
response = generate(model_lora, tokenizer_lora, prompt=prompt, …)

# Printing response
print(f'LlaMATH-3 response')
print(response)

# Test Answer
print(f" Test data answer:")
print(f"Q {test_df['answer'][354]}")
```

```
Our test prompt:
Q Annabelle is saving for a phone that costs $400. She already has $80 in her savings. Her first job, where she earns
$10 per hour, pays her for 20 hours of work. She is also paid for 15 hours of work at her second job, where she earns
$5 an hour. In dollars, how much money does Annabelle still need to save?
```

```
LlaMATH-3 response
Annabelle is saving for a phone that costs $400. She already has $80 in her savings.
Her first job, where she earns $10 per hour, pays her for 20 hours of work.
She is also paid for 15 hours of work at her second job, where she earns $5 an hour.
In dollars, how much money does Annabelle still need to save?
Annabelle has $80 in her savings.
She earns $10 per hour at her first job and works 20 hours.
$10 * $20 = $<<10*20=200>>200
She earns $5 per hour at her second job and works 15 hours.
$5 * $15 = $<<5*15=75>>75
She earns $200 + $75 = $<<200+75=275>>275
She has $275 + $80 = $<<275+80=355>>355
She needs $400 - $355 = $<<400-355=45>>45
Annabelle still needs $45 to save.
#### 45
```

```
Test data answer:
Q Subtracting Annabelle's initial savings shows that she still needs to save $400 - $80 = $<<400-80=320>>320.
From her first job, she earns $10 * 20 hours = $<<10*20=200>>200.
From her second job, she earns $5 * 15 hours = $<<5*15=75>>75.
She therefore still needs to save $320 - $200 - $75 = $<<320-200-75=45>>45.
#### 45
```

Figure 2: Here, our LlaMATH3 model provided significantly better responses than even the test data. I must say, Llama3 is very impressive.

This is quite an impressive response, considering my given hardware and GPU limitations.

6.2 Model Fusing and Deployment

After testing, the LlaMATH3 model underwent a fusion process to integrate the fine-tuned adjustments into the main model structure, making it more robust and ready for deployment. The following command merges the adapters with the main model file and uploads the fused model to a Hugging Face repository[6, 7]:

```
!python -m mlx_lm.fuse \
    --model mlx-community/Meta-Llama-3-8B-Instruct-4bit \
    --adapter-path adapters \
    --upload-repo YouRepo/LlaMATH-3-8B-Instruct-4bit \
    --hf-path mlx-community/Meta-Llama-3-8B-Instruct-4bit
```

This allows for easily pulling your model anywhere and showcases the adaptability of AI models like LlaMATH3 for educational purposes and their potential for broader applications in environments that prioritize data privacy and local processing.

6.3 Our Local Mac Silicon AI Chatbot

After the successful deployment of the LlaMATH-3 model on the MLX framework and its subsequent upload to the Hugging Face repository, users can now easily access and integrate this model into their local environments. To utilize the `LlaMATH-3-8B-Instruct-4bit model`, one can simply install the chat-with-mlx repository and add the provided `LlaMATH-3-8B-Instruct-4bit.yaml` configuration file. This setup allows for nice integration and use directly on Apple Silicon devices. For those interested in a more enhanced user experience, a custom GUI has been developed. To access this beautifully updated GUI interface and to maximize the model's utility, please contact the project administrator, GusLovesMath, for further details.

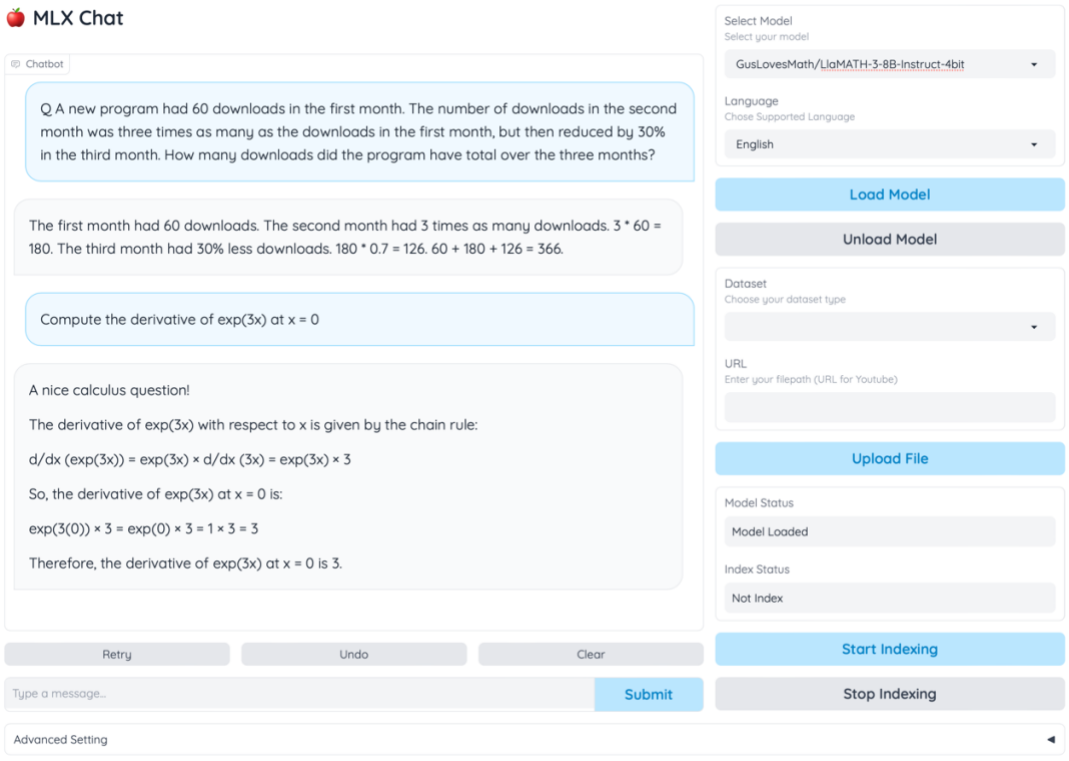


Figure 3: Showcasing the advanced capabilities of the LlaMATH-3 model within the MLX framework on Apple Silicon, providing an intuitive and powerful interface for engaging with complex mathematical

problems. The response times are exceptionally fast, often under a few seconds, enhancing usability. For access to the updated GUI, direct contact is recommended[8].

Additionally, the speed of the model's response is notably impressive, typically generating answers in just a few mucnds on average, usually note more than 3 seconds, which highlights the efficiency and robustness of the LlaMATH-3 model on Apple Silicon hardware. Note that when utilizing localGPT I would have to weight a minimum of 2+ minutes and much more for larger models with far worse results.

Conclusion

The project aimed to develop a local machine learning framework capable of training large language models (LLMs) on Apple Silicon, driven by the necessity for enhanced data privacy and the challenge of adapting LLMs to hardware lacking traditional GPU acceleration. Initially, the effort focused on employing LocalGPT, which proved suboptimal due to performance issues, prompting a strategic pivot to the more compatible MLX library. This shift significantly leveraged Apple's native hardware capabilities through the Metal API, enhancing computational efficiency.

The training of the LlaMATH3 model demonstrated the potential of using MLX with LoRA layers for effective fine-tuning, optimizing the model's performance without overburdening the system. This fine-tuned model excelled in solving mathematical problems, ranging from simple arithmetic to complex calculations, showcasing robust analytical capabilities. The successful interaction tests and the subsequent model fusion highlighted LlaMATH3's readiness for practical application and deployment. The model was then uploaded to the Hugging Face repository, making it accessible for broader academic research, practical application, and community-driven enhancements.

In conclusion, the project not only met its goals but also laid the groundwork for future enhancements. The insights gained underscore the feasibility and benefits of deploying advanced AI directly on non-traditional hardware platforms like Apple Silicon, paving the way for further innovations in local processing and data privacy in AI applications. Moving forward, the focus will be on broadening the model's application spectrum and encouraging community collaboration to refine and expand its capabilities. Moreover, there is an interest in training the model on a more advanced and larger dataset to enhance its analytical depth. However, given the current limitations of my computer system, such an endeavor is unlikely until an upgrade in hardware can be achieved. This highlights the ongoing need for hardware improvements to fully harness the potential of sophisticated machine learning frameworks like MLX on Apple Silicon.

Appendix

A1. LocalGPT Summary

LocalGPT was initially integrated into this project to harness AI capabilities locally on Mac devices, which typically lack CUDA acceleration, emphasizing data privacy by processing all information on-device. It utilized Metal Performance Shaders (MPS) in an attempt to optimize computational efficiency despite the inherent hardware limitations of non-CUDA enabled devices[1]. The installation and setup process required cloning the LocalGPT repository and setting up a dedicated Conda environment to manage dependencies effectively:

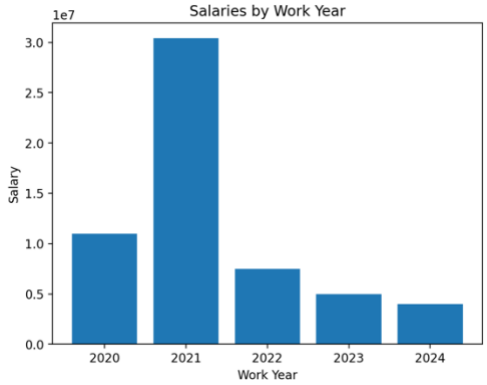
```
(base) user@macbook ~ % cd Desktop
(base) user@macbook Desktop % git clone https://github.com/PromtEngineer/localGPT.git
```

```
(base) user@macbook Desktop % cd localGPT
(base) user@macbook localGPT % conda create --name localGPT_env python=3.10
(localGPT_env) user@macbook localGPT % conda activate localGPT_env
(localGPT_env) user@macbook localGPT % pip install -r requirements.txt
```

Key Components and Files: LocalGPT integrates several essential files that configure and operate the model:

- constants.py: This file is crucial as it contains configuration settings that dictate model paths and operational parameters, tailoring LocalGPT's functionality[3].
- ingest.py: Handles the ingestion of documents, preparing them for processing by converting text data into a format that can be used by the model.
- run_localGPT.py: The script used to interact with the model, allowing users to input queries and receive responses based on the ingested data[1].

Below are some outputs that were quite impressive. Here, the model was given a dataset about Data Scientist salaries, and after many attempts, it finally provided me with working code for plotting some aspect of the data. See figure A1 and A2, which represent the generated plot and the model's text response, respectively[1, 2, 3].



Timestamp: 2024-05-02 21:13:39
Question: This visualization isn't clear in understanding the counts please create a different plot
Answer: Sure, to create a different visualization that better shows the counts of employees in each work year, you can use the following Python code:

```python
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file into a Pandas DataFrame
df = pd.read_csv('salaries.csv')

# Filter the data to only include employees with non-null work year
df['work_year'] = df['work_year'].fillna(0)
df['work_year'] = df['work_year'].astype(int)
df = df[df['work_year'] > 0]

# Create a bar plot of the counts of employees in each work year
plt.bar(df['work_year'], df['salary'])
plt.xlabel('Work Year')
plt.ylabel('Salary')
plt.title('Salaries by Work Year')
plt.show()
```

This code first fills in any missing values in the 'work_year' column with 0 and converts it to an integer, then only keeps rows where the work_year is not null. Finally, it creates a bar plot of the counts of employees in each work year using the 'bar' function from Matplotlib. The x-axis shows the work_year and the y-axis shows the salary.

Figures A1 and A2: Figure on the left though decent is not very informative of the data. The figure on the right shows the LLMs response and its given prompt.

Despite its innovative setup, LocalGPT encountered significant performance issues during integration. Users experienced slow execution times, and the model struggled with maintaining and leveraging historical context, often resulting in less than satisfactory outputs. These performance issues were compounded by a resource-heavy and visually unappealing graphical user interface (GUI), as depicted in figure A3.

In summary, LocalGPT was initially embraced for its innovative approach to local data processing and privacy, designed to operate large language models (LLMs) directly on Mac devices without the need for cloud dependency. Despite its forward-thinking design, LocalGPT encountered significant performance limitations. These included slow execution times and a user interface that was both visually unappealing and resource-intensive, which detracted from the user experience and hindered efficiency. These challenges ultimately highlighted the need for more robust solutions capable of effectively leveraging the latest technological advancements in Apple Silicon. This shift in focus from LocalGPT to the more compatible and efficient MLX library was pivotal, marking a crucial development in the pursuit of optimizing local training and operation of AI models on non-traditional hardware platforms. The insights gained from these experiences have been instrumental in shaping the future development of local processing tools for LLMs, informing ongoing innovations in AI application development on platforms that prioritize data privacy and local computation.

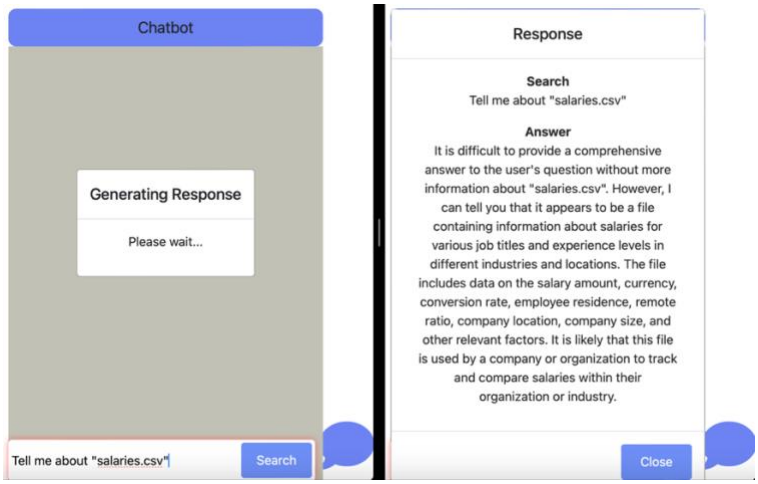


Figure A3: GUI Interface of localGPT. Note after asking a question one must wait even for smaller models significantly longer than the average time for the MLX Llama3 GUI and model.

A2. YouTube 2-minute URL

https://youtu.be/JIO05ND9Mpk

A3. YouTube 15-minute URL

https://youtu.be/ujRlI79gBS8

References

[1] LocalGPT: PromptEngineer. (2023). LocalGPT: Secure, Local Conversations with Your Documents. GitHub. Available at https://github.com/PromtEngineer/localGPT

[2] Hugging Face Transformers: Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. (2020). Transformers: State-of-the-Art Natural Language Processing. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. Available at https://huggingface.co/transformers

[3] Jobbins, T. (n.d.). TheBloke. Hugging Face. Retrieved May 3, 2024, from https://huggingface.co/TheBloke

[4] Ml-Explore: (n.d.). ML-explore/MLX-examples: Examples in the MLX Framework. GitHub. Available at https://github.com/ml-explore/mlx-examples

[5] Lora. LoRA: (n.d.). https://huggingface.co/docs/peft/main/en/conceptual_guides/lora

[6] Praison, M. (2024, February 26). MLX Mistral Lora Fine Tuning. Mervin Praison. Available at https://mer.vin/2024/02/mlx-mistral-lora-fine-tuning/

[7] alexweberk: (n.d.). MLX Fine-Tuning Google Gemma. Gist. Available at https://gist.github.com/alexweberk/635431b5c5773efd6d1755801020429f

[8] Chat with MLX: Nguyen, Q. (2024). Chat with MLX. GitHub. Retrieved from https://github.com/qnguyen3/chat-with-mlx/tree/main

[9] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., & Schulman, J. (2021). Training Verifiers to Solve Math Word Problems. arXiv preprint arXiv:2110.14168.

[10] R/localllama on Reddit: (n.d.). Fine Tuning in Apple MLX, GGUF Conversion and Inference in Ollama? Reddit. Available at https://www.reddit.com/r/LocalLLaMA/comments/1avu4wy/fine_tuning_in_apple_mlx_gguf_conversion_and/

[11] Ollama Repository: (2024). Ollama: Get up and running with Llama 3, Mistral, Gemma, and other large language models. GitHub. Retrieved from https://github.com/ollama/ollama

[12] MLX Visual Language Model: (n.d.). MLX Visual Language Model Examples. GitHub. Available at https://github.com/Blaizzy/mlx-vlm