SANTA CLARA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Date: June 13, 2024

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Darren Inouye
Lucas Lindo
Robin Lee
Edmund Allen

ENTITLED

Applied Auto-tuning on LoRA Hyperparameters

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

Younghyun Cho

Thesis Advisor

Department Chair

Applied Auto-tuning on LoRA Hyperparameters

by

Darren Inouye
Lucas Lindo
Robin Lee
Edmund Allen

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 13, 2024

Applied Auto-tuning on LoRA Hyperparameters

Darren Inouye
Lucas Lindo
Robin Lee
Edmund Allen

Department of Computer Science and Engineering
Santa Clara University
June 13, 2024

ABSTRACT

This senior design project explores the application of Bayesian optimization-based auto-tuning techniques on the low-rank adaptation (LoRA) fine-tuning of large language models (LLMs), demonstrating how fine-tuning methods can reduce training times and costs, albeit with a slight trade-off in accuracy. However, little is known about the optimal hyperparameters on LoRA and its variants for those methods. This project addresses this lack of knowledge by analyzing data gathered from auto-tuning LoRA hyperparameters to determine the most optimal parameter configurations for a model's accuracy and training efficiency.

The team has implemented a pipeline utilizing many different technologies. The main technology driving the Bayesian optimization-based auto-tuning is GPTune, a performance auto-tuner built on Gaussian Process regression. The team selected Llama3 as the base model for testing due to its high-performance capability and relevancy. These models are fine-tuned using the QLoRA framework and evaluated on their training time and loss. GPTune uses Bayesian optimization to efficiently explore the Pareto front of training time and loss for varying QLoRA parameters.

This research has contributed to LLMs by demonstrating the efficacy of Bayesian optimization-based auto-tuning techniques in fine-tuning LoRA. By applying these techniques, we have shown that it is possible to optimize hyperparameters more efficiently, reducing computational resource requirements and improving model performance. Our work provides a proof of concept for integrating advanced auto-tuning frameworks like GPTune with existing fine-tuning tools, paving the way for more accessible and cost-effective use of LLMs in various applications.

ACKNOWLEDGEMENTS

We would like to thank our advisor, Dr. Younghyun Cho, for providing access to the Google Cloud GPUs used in our experiments and for his continuous support throughout the project. Dr. Younghyun Cho's assistance has been useful in guiding the direction of the project and in teaching us how to operate GPTune, which is a vital component of the project. We would also like to thank Santa Clara University for providing the facilities to run the WAVE HPC, since it was our major computing resource for running our experiments. The WAVE HPC's computing power was helpful for running a majority of our computationally intensive experiments.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Large Language Models (LLMs) are artificial intelligence models trained to understand and generate human-like language [27]. These models handle various natural language processing tasks, such as text completion, language translation, summarization, and question answering. Notable applications of LLMs include OpenAI's ChatGPT [25], GitHub Copilot, and Character.AI. LLMs typically use a neural network architecture based on the transformer model, which employs an "attention mechanism" to focus on different parts of the input sequence during output generation.

Other neural network architectures, such as recurrent neural networks (RNNs) and long short-term memory (LSTM) networks, have been used for natural language processing (NLP) tasks. However, the transformer architecture has demonstrated superior performance due to its attention mechanism.

Though there are other neural network architectures, such as recurrent neural networks (RNNs) or long short-term memory (LSTMs), none of them have been as successful as the transformer architecture for natural language processing (NLP). The primary reason for this lies in the "attention mechanism" concept in the transformer, enabling the model to concentrate on distinct segments of the input sequence while generating each output token.

1.2 Problem & Motivation

Training and fine-tuning LLMs present significant computational challenges due to the exponential increase in resource demands as model sizes grow. For example, GPT-2 has 1.5 billion parameters [2], GPT-3 has 175 billion parameters [2], and GPT-4 has an estimated 1.7 trillion parameters [1]. Efficient hyperparameter tuning using auto-tuners is essential to manage these demands. This project aims to apply Bayesian optimization-based auto-tuning techniques to LoRA fine-tuning, using an auto-tuner to optimize LoRA for LLMs quickly. This approach makes these models more accessible for real-world applications by maximizing accuracy within a given system's resources.

1.3 Literature Review

Applying a LoRA (Low-Rank Adaptation) to an LLM to address the high cost of fine-tuning is an established concept. However, this research aims to explore new aspects of this approach. Prior work in the field has informed our research and provided valuable insights.

LoRA: Low-Rank Adaptation of Large Language Models

Researchers at Microsoft introduced the initial proposition and detailed methodology for LoRA. Their development aimed to efficiently fine-tune large language models like GPT-3 by significantly reducing the number of trainable parameters [16]. They reduced the trainable parameters in GPT-3 by nearly 10,000 times and decreased the GPU load by three times [16]. This solution provided flexibility and compatibility, improving training efficiency while maintaining performance comparable to fully fine-tuned models.

LoRA+: Efficient Low-Rank Adaptation of Large Models

Building on the initial proposal of LoRA, researchers at Berkeley developed methods to tune LoRA itself and adjust its learning rates [15]. They demonstrated a 1-2% increase in model accuracy compared to an untuned LoRA while maintaining the same low resource demand [15]. This project aims to achieve similar improvements using GPTune, an auto-tuning framework.

Hyperparameter Optimization for Large Language Model Instruction-Tuning

Researchers at Polytechnique Montreal developed an optimization framework for LoRA hyperparameters using NOMAD and NNI-TPE for black-box optimization methods. They applied this to Llama 2 [30] models and evaluated them using InstructEval [3]. While similar to our framework, our approach also considers training time as an optimization objective. Additionally, we aim to improve their design by integrating evaluation into the optimization process [20].

1.4 Solution

This project explores automated search techniques, such as Bayesian optimization, on LLM fine-tuning hyperparameters. Given the large set of hyperparameters and substantial computational resources required for training an LLM, we designed experiments with a manageable scope. We began auto-tuning experiments using models like Llama3 [22] and leveraged the SCU's WAVE HPC cluster resources. Instead of tuning the entire LLM, we focused on a "fine-tuning" scenario, which involves retraining parts of the model using previously trained parameters (frozen model weights). We utilized existing fine-tuning tools (e.g., QLoRA [7]) to reduce the search space and potentially test other auto-tuning

techniques. Additionally, we investigated the integration of an open-source auto-tuner, GPTune, to optimize the hyperparameters of these QLoRAs. We combined these efforts to create an overall design for the auto-tuning, which depicted in figure 1.1. These methods aim to achieve optimal hyperparameter configurations for training time and loss, providing a proof of concept for larger LLM applications.

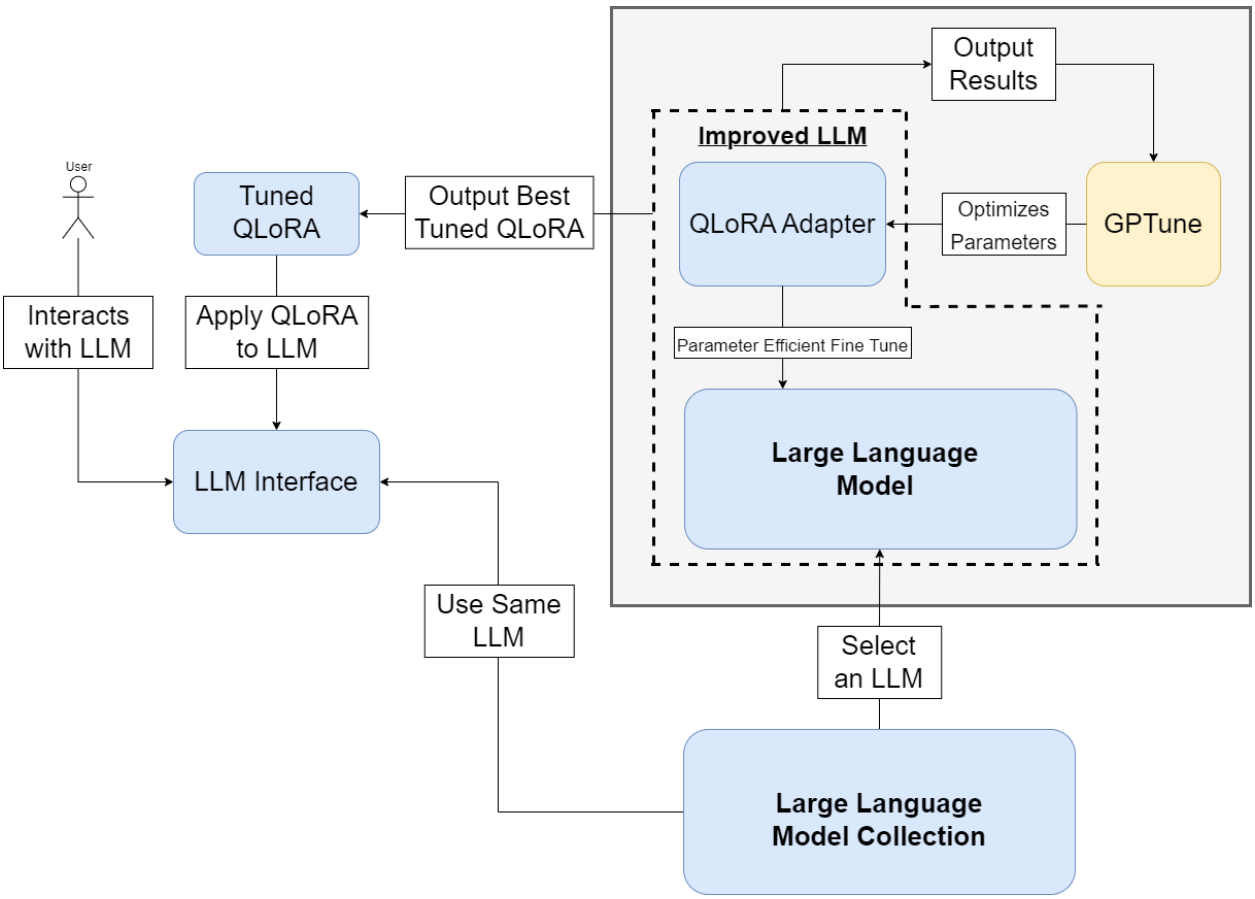

Figure 1.1: Architecture Diagram

Chapter 2

Design Considerations

2.1 Computing Infrastructure

Our experiments require a lot of computation power and storage. Our initial assessments revealed that personal computers lack sufficient computational power for fine-tuning within a reasonable time frame. This led us to explore alternative computing resources, each with pros and cons.

SCU's WAVE High Performance Computer (HPC)

Santa Clara University's WAVE HPC [31] offers free access to students, offering a cost-effective solution for our computational needs. A significant concern with this platform was the competitive nature of resource allocation among students and the use of older hardware [31], which may not optimally support the latest computational demands required for our project.

SCU's WAVE HPC GPU Hardware Capabilities: [31]

- WAVE HPC GPU Nodes have up to 2 NVIDIA Tesla V100 GPUs.
- WAVE HPC AMD Node has one AMD Instinct MI 100 GPU.

Other Resources on the WAVE HPC exist, however, they are not as applicable to our needs. A GPU is necessary for us to efficiently and effectively fine-tune our large language model.

Amazon Web Services (AWS) EC2/GPU Clusters

AWS EC2/GPU clusters offer high performance and scalability, suitable for the intensive computations required for fine-tuning large language models[28]. The main obstacle with this option is the requirement for substantial funding to cover the costs associated with these high-performance computing services.

AWS GPU Instances Hardware Capabilities: [28]

- Amazon EC2 P3 Instances have up to 8 NVIDIA Tesla V100 GPUs.

- Amazon EC2 P4 Instances have up to 8 NVIDIA Tesla A100 GPUs.

After further research, we concluded that the costs of utilizing AWS services would be too high.

Google Cloud

Google Cloud [4] provides reasonably priced GPU resources. However, high-end GPU resources like A100s and V100s are often limited in availability.

The Google Cloud offers many configurations for their GPU options.

- NVIDIA T4 (up to 4x)
- NVIDIA P4 (up to 4x)
- NVIDIA V100s (up to 8x)
- NVIDIA A100s (up to 4x)

Selected Computing Approach

Our selected approach used the most readily available option, SCU's WAVE HPC [31]. Since many LLM frameworks are built with libraries that rely on NVIDIA GPUS, we used the WAVE HPC GPU Nodes with NVIDIA Tesla V100 GPUs.

We used Google Cloud as an alternative computational resource. Specifically, we used nodes that included an NVIDIA Tesla 2xV100 GPU setup. However, we did seek to use more powerful or more GPU configurations.

2.2 Large Language Model Selection

We selected OpenAI's GPT-3 [25], the engine behind ChatGPT [25], as a model of interest because it is trained on 175 billion parameters. GPT-3 [25], short for "Generative Pre-trained Transformer 3," is trained on 175 billion machine learning parameters. As a result of its large scale, GPT-3 can generate more coherent and contextually relevant sentences over longer passages to generate text that closely resembles human writing.

Unfortunately, GPT-3's source code, training dataset, and other parameters have not been released to the public. This proprietary aspect of the model restricts its use in our project.

picoGPT and LittleGPT

GPT-2 [27], or "Generative Pre-trained Transformer 2," is a large-scale, unsupervised language model developed by OpenAI. It is a transformer-based model, which means it uses self-attention mechanisms to understand the context and semantics of a sentence. As a result, it can generate coherent and contextually relevant sentences by predicting subsequent words within a given piece of text.

One key aspect of GPT-2 is that its code, model weights, and detailed technical analysis have been publicly available. picoGPT and LittleGPT represent refactored and modified versions of GPT-2.

picoGPT [18] is a minimal implementation of GPT-2 using plain NumPy, with the forward pass code consisting of only 40 lines. The entire forward pass code is 40 lines of code. As a result, its implementation results in slow inference speeds, limited customization, and overall subpar performance. We mainly used picoGPT as a resource to learn about the transformer architecture and inner workings of large language models.

LittleGPT [9] includes a collection of transformer models based on GPT2-Small, GPT2-Medium, and GPT2-XL with the following architecture modifications to speed up training and inference. The caveat to this project is that the latest commit to this project was more than 2 years ago, indicating that the project has been abandoned. This makes sense as better, faster, and stronger models and supporting libraries have been released since then. Similar to picoGPT, we utilized LittleGPT as a learning resource.

Llama Models

Llama2 [30] and Llama3 [22] are families of open-sourced, pre-trained, and fine-tuned large language models (LLMs) released by Meta AI. Llama2, released in 2023, offers models with seven billion (7B), 13 billion (13B), or 70 billion parameters (70B). It performed well on external benchmarks compared to other open-source language models and was considered the de facto open-source model of its time.

Llama3, the most recent release in 2024, continues the legacy of Llama2 with models available in eight billion (8B) or 70 billion parameters (70B). These models stand out among other open-source language models of similar size in performance on benchmarks such as MMLU and Human-Eval. Given its high relative performance, Llama3 is currently regarded as the leading open-source model.

Mistral 7B

Mistral [19] is another series of publicly viewable and pre-trained competitive large language models released by Mistral AI in 2023. The Mistral model has open-weights but is not open-sourced.

In comparison to Llama2, on the same parameter count of 7B, Mistral performs significantly better on many metrics. This impact has solidified Mistral's models as an alternative to Llama2. This performance is largely because the Mistral project has focused on training their models on higher quality training data. This training data is proprietary.

Selected Model

We selected Llama3 for its strong performance and relevance. Specifically, we will use the 8 billion parameter version for balanced performance and resource requirements. In addition, Llama3 has a large amount of documentation surrounding its implementation and use.

2.3 Fine-tuning Framework

Adapting large pre-trained models for specific tasks requires efficient and flexible fine-tuning frameworks. Our investigation into available tools identified several potential candidates [11; 21; 24; 34; 7; 23].

Parameter-Efficient Fine-Tuning (PEFT) Library

The PEFT library was initially presented as a viable solution due to its specialized methods for fine-tuning [21]. However, its low-level operations and broad methodological scope posed substantial challenges to our specific requirements, leading us to consider alternative solutions.

Llama.cpp

Llama.cpp [11] is a comprehensive framework for LLM inference, designed for minimal setup and optimized performance on various hardware. Additionally, llama.cpp provides tooling needed for LLM customization, including fine-tuning via LoRA, model format conversions, and model quantization. Llama.cpp is implemented in plain C/C++ without any dependencies to be platform and hardware-agnostic.

Axolotl and Llama Factory

Axolotl [24] is a comprehensive tool for fine-tuning various AI model architectures. It supports multiple fine-tuning techniques, including full fine-tuning, LoRA, QLoRA, ReLoRA, and GPTQ, and integrates with technologies like xFormers, FlashAttention, RoPE Scaling, and multipacking.

Llama Factory [34] is another modern framework for fine-tuning large language models. It offers 32-bit full-tuning, 16-bit freeze-tuning, 16-bit LoRA, and 2/4/8-bit QLoRA via AQLM/AWQ/GPTQ/LLM.int8. It also employs technologies such as FlashAttention-2, Unsloth, RoPE scaling, NEFTune, and rsLoRA for enhanced performance.

Axolotl and Llama Factory rely on Flash Attention [5], a library optimized for fast and memory-efficient training, leveraging the NVIDIA Ampere Architecture, primarily used in NVIDIA A100 Tesla GPUs. Despite their robust capabilities, Axolotl and Llama Factory were determined to be less compatible with our project due to their reliance on specific hardware advancements and technologies, limiting broader hardware compatibility.

QLoRA and QLoRA Fine-tune

QLoRA [7], Quantized Low-Rank Adaptation, is an extension of LoRA. QLoRA applies quantization to the weights of the LoRA adapters. Quantization involves converting a continuous range of values into a finite set, effectively reducing the precision of the weights. In the case of QLoRA, the weights are quantized to a lower precision, such as 4-bit instead of the typical 8-bit.

This is the key advantage of QLoRA. By quantizing the weights, QLoRA further reduces the memory footprint and storage requirements compared to LoRA. Even with reduced precision, QLoRA maintains similar effectiveness to LoRA in terms of performance. This combination of efficiency and performance makes QLoRA a good choice for LLM fine-tuning.

QLoRA Fine-tune [23] is a framework that allows easy training of QLoRA adapters and fine-tuning with custom datasets.

Selected Fine-tuning Framework

We selected QLoRA Fine-tune [23] for its support of QLoRA [7], simplicity, and comprehensive documentation.

In our research, QLoRA was preferred over LoRA because of our limitations in computational power and need for memory efficiency.

In addition, because QLoRA Fine-tune has a simple implementation, it uses a limited number of external libraries. As a result, it has a high amount of compatibility with many GPUs and machines. This is beneficial as it allows us to explore many more options for our hardware selection.

2.4 Hyperparameter Search and Auto-tuning

Identifying an auto-tuning solution necessitates a preliminary decision on the hyperparameter search methodology. Our review of the available options highlighted the merits and drawbacks of each, ultimately guiding our selection.

Random Search

Random Search selects hyperparameter combinations randomly from a specified range or distribution. Unlike methods that systematically go through combinations of hyperparameters, Random Search jumps across the space, allowing for a broader exploration of the hyperparameter space. This method is particularly beneficial when the number of hyperparameters is large, making exhaustive searches impractical. Although Random Search may miss the optimal hyperparameter combination, it is computationally less expensive and can often find a good combination more quickly than exhaustive methods. This is also its greatest disadvantage, as it disregards any patterns or relationships inherent among hyperparameters.

Grid Search

Grid Search is a brute-force method for hyperparameter tuning that systematically iterates through a grid of multiple combinations of hyperparameter values. It exhaustively searches through the specified subset of the hyperparameter space, evaluating and comparing the model performance for each combination to identify the best one. While

Grid Search is simple to understand and implement, it can be computationally intensive, especially as the number of hyperparameters and their values increases.

Bayesian Optimization

Bayesian Optimization is a more sophisticated and efficient approach to hyperparameter tuning. It builds a probabilistic model that maps hyperparameters to a probability of a score on the objective function. The method uses past evaluation results to form a probabilistic model mapping hyperparameters to a likelihood of objective scores. It then uses this model to select the most viable hyperparameters to evaluate the true objective function. Bayesian optimization is most useful for optimizing the performance of models that are expensive to train, as it aims to find the best hyperparameters in fewer iterations by focusing on exploring areas with the highest potential for improvement. It includes well-known implementations such as Gaussian Processes [14].

Selected Method and Auto-Tuner

We selected Bayesian optimization for hyperparameter search due to its theoretical foundation, efficiency in search space navigation, and effectiveness in optimizing costly functions. Despite its complexity in implementation, we decided that the benefits of using such a method would largely overshadow its drawbacks.

There are three additional reasons we chose Bayesian optimization. First, the search space we wanted to explore was unknown and possibly complex. Bayesian optimization is best suited for problems with complex behaviors since it can optimize in those situations better than gradient descent and other basic search algorithms. Secondly, our research could be used for transfer learning on larger-scale projects and models. By researching good hyperparameter configurations on smaller models, we can apply our findings to larger models, saving training time and costs. Lastly, we could perform hyperparameter analysis in the future. This would entail hyperparameter sensitivity, which is important but not as relevant for our use case.

We selected GPTune [20], a performance auto-tuner built on Gaussian Process regression, to implement Bayesian optimization. Specifically designed for HPC applications with high evaluation costs, GPTune aligns with the demands of our project. Its capability to efficiently optimize hyperparameters using Bayesian optimization and accomplish multi-objective tuning makes it an essential component of our design strategy.

2.5 Technologies Used

The integration of SCU's WAVE HPC [31] and Google Cloud [4] for computing, QLoRA [7] and QLoRA Fine-Tune [23] for the fine-tuning framework, and GPTune [20] for hyperparameter auto-tuning forms the backbone of our approach to optimizing the Quantized Low-Rank Adaptation of Large Language Models. These design considerations are pivotal to our goal of achieving optimal performance in LoRA [21] adapted fine-tuned models.

2.6 Use Cases

Personalized Language Models

By adapting large language models to specific domains or individuals, we can create personalized language models that better understand and respond to the unique needs of different users.

Efficient Fine-tuning

Optimal hyperparameters can significantly reduce the time and resources required for fine-tuning large language models on new tasks, making it more accessible and cost-effective to deploy these models in real-world applications.

Improved Model Performance

By identifying the optimal hyperparameters for QLoRA, we can improve the performance of fine-tuned large language models on varying tasks, including natural language processing, computer vision, and robotics.

Novel Applications

The combination of QLoRA and optimal hyperparameters opens up new possibilities for developing innovative applications that leverage the power of large language models, such as personalized chatbots, intelligent search engines, and language-based decision-making systems.

Chapter 3

Implementation

Algorithm 1 Driver Algorithm

```
1: while samples < max_samples do
2:     Use GPTune on all previously recorded results to generate new sets of values for hyperparameters
3:     for set in sets do
4:         Load values from the set as inputs for hyperparameters
5:         Run QLoRA training with input values for hyperparameters
6:         Record the resulting training loss and training time
7:     end for
8:     Transfer recorded results into the data file GPTune that reads
9:     samples ← samples + number_of_sets
10: end while
```

3.1 Architecture Overview

The current architecture for our project utilized two other projects, GPTune [20] and QLoRA Fine-tune [23]. GPTune is the tuning framework that will change the hyperparameters for training the QLoRAs [7]. QLoRA Fine-tune is the training framework for creating QLoRAs. Algorithm 1 demonstrates the general workflow for our experiments. GPTune uses samples, which are recorded results from previous training runs, to generate the next sets of hyperparameter values to test. Each set of values is the input for each training run's hyperparameters. We use QLoRA Fine-tune with a given set of hyperparameter values for a training run. Each training run will record its results into a separate folder containing a JSON file with the training loss and training time alongside the respective QLoRA file. After all the sets of values have been tested, the resulting training loss and training time are transferred to the JSON file GPTune reads for samples. This cycle repeats until we've generated enough samples defined in max_samples.

3.2 Evaluation Metrics

Objective Hyperparameters

We need to quantify the results of our experiments with some resulting value. Our current objectives are the QLoRA's training loss and training time. These two hyperparameters were chosen because they are simple to understand and easy to evaluate, especially since they are values directly produced from training QLoRAs.

Training loss is a value that measures how close a model, the produced QLoRA, fits the dataset it was trained on. The lower the value, the more identical it performs at repeating the dataset. For instance, if the dataset asks "What color is gold?" and "Yellow" is the answer within the dataset, then the model could answer "Red" and increase training loss. The model could also answer "Yellow" instead and decrease the overall training loss. This can go down to zero, where the model perfectly fits the dataset. However, this tends to lead to the issue of overfitting, where the model becomes less capable of correctly responding to information outside of the dataset. Despite this potential issue, we will use training loss as an objective because it is a direct way of evaluating our experiments.

Training time is the time it takes for the QLoRA to be trained. This value allows us to measure resource cost. Since we train for a set amount of steps rather than time, we can determine that we use more resources the longer it takes to train. We can use this value to check if certain hyperparameters or combinations of hyperparameters will cause a reduced or increased amount of resource usage.

Multi-Objective Tuning

Since we seek to optimize more than one objective for our fine-tuning, we want a set of non-dominated solutions that form a Pareto front rather than one dominant solution. The Pareto front is the set of solutions that are non-dominated by each other but are better than all other known solutions. For instance, one solution on the Pareto front may have the lowest training loss but another has the lowest training time with a higher training loss. Depending on your preferences, you may choose one of these solutions over the other or a solution that strikes a middle ground between the two. You can see a clear depiction of this example within figure 4.4.

We will also need to modify Bayesian optimization to tune for multiple objectives. Fortunately, GPTune can apply the NSGA-II algorithm[6], which solves this issue. The NSGA-II algorithm works over the optimization process to find the hyperparameter configurations that approach the Pareto front. It is an evolutionary algorithm based on the elitist principle, meaning the best configurations carry over to the next generation. However, it also preserves diversity by implementing crowding distance to avoid overcrowding the same configurations and to prevent staying at a sub-optimal solution that is a local optimum.

3.3 Experimental Setup

The setup of this project for experiments will require at least two components and potentially some manual modification. The first component is the language model. The second component is a dataset.

A pre-trained language model is the basis for training a QLoRA [7]. The QLoRA's weights only exist to alter the original weights within the language model it was created from. A majority of the regular function still relies on the language model. Thus, the quality of the base language model will affect the final results produced.

Datasets are a major part of the evaluation process. Datasets for LLMs tend to contain 3 different "splits" within them. They are the training data split, validation data split, and the test data split. The training data split is used for training a language model. The validation split is usually used as the immediate test for the newly trained model. This split creates a buffer between the training and test data split to prevent overfitting for both data splits. The test data split is used as the final performance test for the model. This data would also need to be relevant to the topic you want to train the QLoRA for, otherwise, the QLoRA would likely be unhelpful in completing the task you want. We will only need the training data split because we only evaluate training loss and training time. However, you will have to manually specify the delimiter that separates each entry in the dataset according to its file type and format.

3.4 Integration of GPTune and QLoRA

We utilize a script to format data files for GPTune [20]. The input for the script is the data produced by QLoRA. This formatted file is used by another script that calls GPTune and references the file for GPTune to analyze. GPTune produces the new hyperparameters we need to input into QLoRA. Currently, this is a manual process.

3.5 Tuneable Hyperparameters

QLoRA refers to some of its hyperparameters with LoRA in the name, since the internals of QLoRA are LoRA. As such, we will refer to those hyperparameters similarly in this section. Initially, we tuned only the LoRA rank and LoRA alpha. In a later experiment, we added learning rate to the hyperparameters GPTune [20] was allowed to tune.

LoRA Rank

LoRA rank is the size of the matrices that represent a portion of the weights in the model. Essentially, a higher LoRA rank increases the number of trainable weights for a model, which causes the LoRA to be more specifically tuned for what it is trained on. Conversely, a lower LoRA rank will not affect the model as much, so the model remains more generalized.

LoRA Alpha

LoRA alpha acts as a scaling factor for the weights in LoRAs. LoRA alpha determines how much of the original model's weights will change with the addition of the LoRA weights. It effectively works such that when LoRA alpha decreases, the impact of the LoRA's weights on the original model's weights increases. When LoRA alpha increases, the influence of the LoRA's weights on the original model's weights decreases.

Learning Rate

Learning rate modifies the speed at which the weights change toward a theoretical optimal value. A high learning rate may cause the weights to reach points near their optimal value quickly but never settle at their optimal value because they constantly overshoot that target after getting near it. A low learning rate may be too slow to let the weight reach its optimal value within the set amount of given trials.

Potential Future Hyperparameter

Another hyperparameter that could be added in future work would be LoRA dropout. LoRA dropout reduces overfitting by dropping a set amount of randomly selected weights during training, but will likely lower training loss. LoRA dropout has not been added, since we wanted to keep finishing our initial experiments before expanding our scope to include LoRA dropout.

3.6 Methodology

Training QLoRAs remains resource-intensive, even with reduced costs. As such, we devised a general method to save on resources while testing our program. The general idea is to train a pilot set of QLoRAs based on random hyperparameter configurations. Afterward, we would train new sets of QLoRAs using the information from the foundational set of QLoRAs.

For instance, our first experiment trained 20 pilot pseudo-random QLoRAs. The results of these 20 pilot pseudo-random QLoRAs were input into GPTune [20]. Then, we trained 5 QLoRAs using Bayesian optimization through GPTune. We also train an additional 5 pseudo-random QLoRAs to create a baseline. This lets us compare the effectiveness of the pseudo-random QLoRAs and the GPTune LoRAs.

This method massively reduces the amount of training time for our experiments. It normally takes 3-4 hours to train one LoRA on only one-fifth of the self-instruct dataset [33]. However, it technically does not allow us to compare completely fairly between training algorithms. Overall, it is a necessary trade-off because we have limited resources.

3.7 Challenges

Datasets

Dataset preparation requires foundational work and knowledge. First, it requires knowledge about how the dataset is formatted and how it should be formatted for an LLM to understand the task. Secondly, you need to know what the dataset is about, otherwise, you won't be able to select an appropriate evaluation task. Another extraneous problem includes datasets using unfamiliar file types such as parquet or multiple bin files.

Manual Work

Currently, the program requires some manual input to complete its tasks. For instance, converting QLoRA results into GPTune-compatible files requires the user to initiate the program. The user would need to follow a few procedures over the entire process to get results.

Shell Scripting

We are inexperienced with shell scripting, so programming functional shell scripts to automate the pipeline has been arduous. It has been hard to find resources on problems specific to this project.

Chapter 4

Results

4.1 Experiment 1

4.1.1 Pseudo-Random Search

In our first experiment, we tested a constrained version of random search that we will call pseudo-random search because it mimics the recommended defaults. Pseudo-random search uses two arrays of the seven numbers 4, 8, 16, 32, 64, 128, and 256. One array is for LoRA rank, and the other is for LoRA alpha. Each run randomly selects one of the seven numbers for LoRA rank and another number selected from the seven numbers for LoRA alpha. These seven numbers were specifically chosen because they follow the pattern of recommended configurations, which are numbers that are 2^n. These numbers are also believed to create ideally sized matrices for LoRA rank.

4.1.2 Experimental Setup

Fine-tuning Problem	
Pre-trained model	Llama 3 (8B model)
Fine-tuning dataset	Self-instruct (0.19 epoch)
QLoRA Configuration Hyperparameters	
LoRA rank (r)	Number of training parameters in adaptation layers
LoRA alpha (a)	Scale factor in which the adaptation layers affect the output
Objectives	
Training time	Time taken by the training
Training loss	Fitness to the training data

Table 4.1: Experiment 1 Configurations

We wanted to analyze the difference between a selection of recommended choices of rank and alpha values versus what GPTune [20] could produce. GPTune searched for numbers between 4 and 256 with a step size of 4 for rank and alpha. GPTune was also set up to run once and generate a batch of 5 runs to test. Therefore, we conducted experiments to calculate the Pareto front for the pseudo-random search and the Bayesian optimized approach.

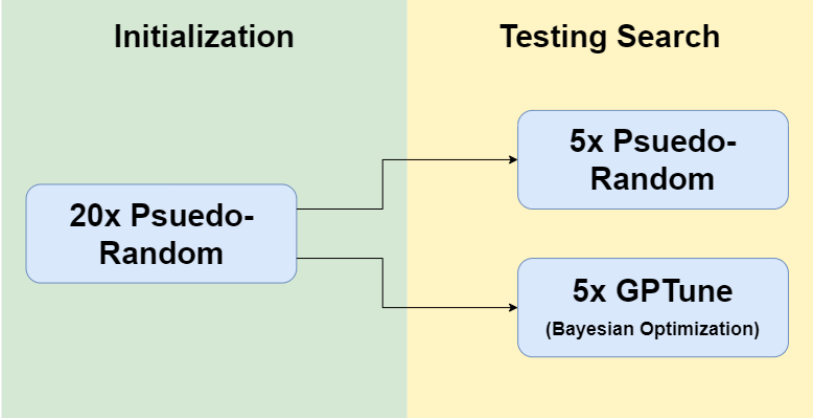

Figure 4.1: Diagram of Experiment 1's Structure

20 Pilot Pseudo-Random runs were conducted to prime the Bayesian optimization algorithm.

5 Pseudo-Random runs were conducted for testing.

5 Bayesian Optimized runs were conducted for testing.

4.1.3 Experiment Results

For our first experiment, we calculated the Pareto front for the results of our pure pseudo-random configurations and our Bayesian optimized configurations (4.2). From our Pareto fronts, we could try to find the configurations of runs that are most optimized for training loss and training time.

In the graph shown below, each data point's configuration is represented as **(rank, alpha)**

We calculated the Pareto fronts for our 20 pilot pseudo-random runs, 20 pilot pseudo-random runs with 5 pseudo-random runs, and 20 pilot pseudo-random runs with 5 Bayesian optimized runs. Within those Pareto fronts, we can see that the pseudo-random search, Bayesian optimized search, and the 20 pilot pseudo-random runs have the same Pareto front. Configurations such as (8, 32), (16, 8), and (64, 4) lie on that shared Pareto front. They all come from the 20 pilot pseudo-random runs.

We conclude this first experiment with the knowledge that our selection of rank and alpha hyperparameters is optimal for balancing training loss and time. From both sets of experiments, configurations that were in powers of 2 such as (8, 32), (16, 8), and (64, 4) were found to be best for this case.

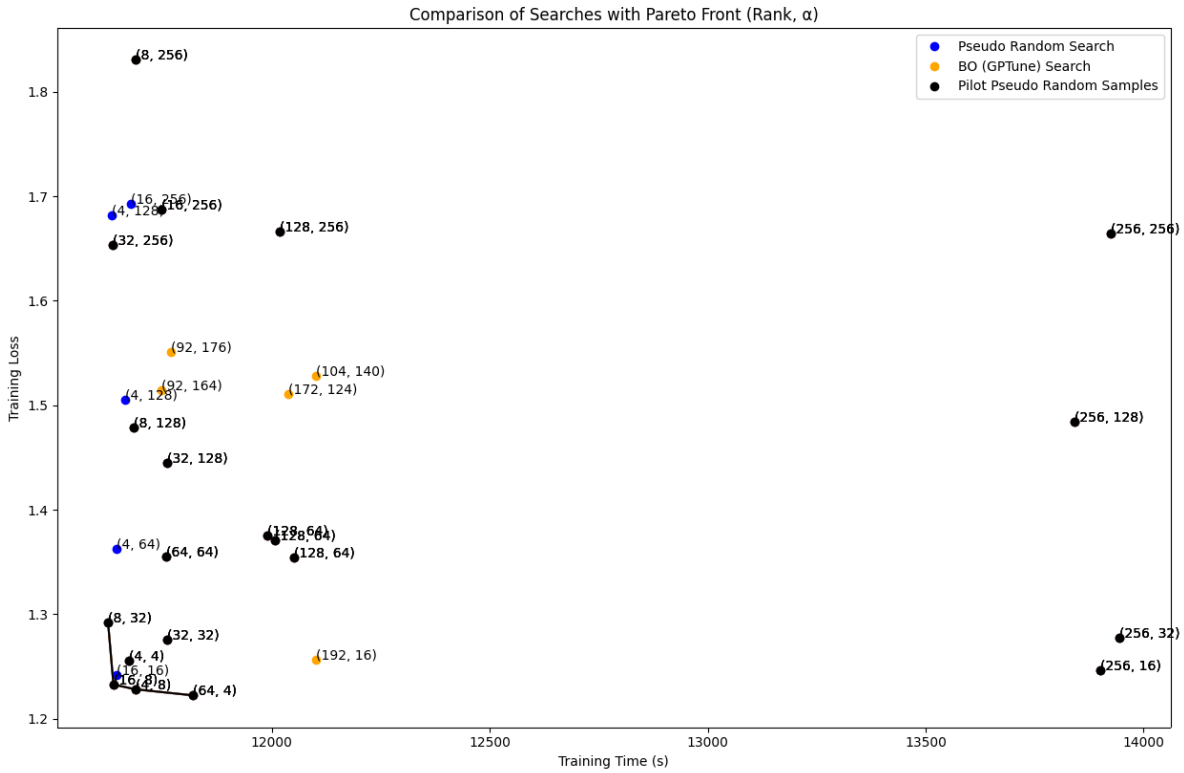

Figure 4.2: Data Plot for Experiment 1's Training Runs

4.2 Experiment 2

4.2.1 Experimental Setup

Fine-tuning Problem	
Pre-trained model	Llama 3 (8B model)
Fine-tuning dataset	Self-instruct (0.19 epoch)
QLoRA Configuration Hyperparameters	
LoRA rank (r)	Number of training parameters in adaptation layers
Objectives	
Training time	Time taken by the training
Training loss	Fitness to the training data

Table 4.2: Experiment 2 Configurations

In our second experiment, we sought to analyze naive choices of rank with a fixed alpha value. Our first experiment did not show a difference in effectiveness between pseudo-random runs and Bayesian optimized runs since the pilot pseudo-random runs already found the optimal configurations. So, we made rank capable of going between 4 and 256

with a step size of 4 for random search, which is the same search space as GPTune's [20] Bayesian optimized search. We also simplified the experiment by fixing alpha to 16. GPTune was set up to run once and generate a batch of 3 runs to test. To this end, we conducted experiments to calculate the Pareto front for the random search and the Bayesian optimized approach.

5 Pilot Random runs were conducted to prime the Bayesian optimization algorithm.

3 Random runs were conducted for testing.

3 Bayesian Optimized runs were conducted for testing.

4.2.2 Experiment Results

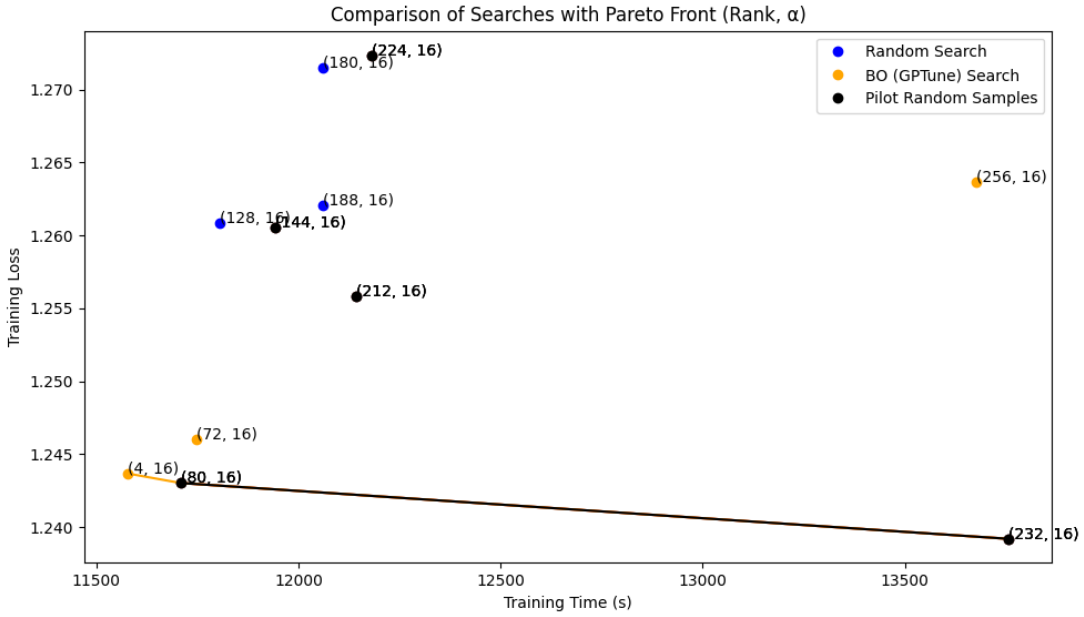

Figure 4.3: Data Plot for Experiment 2's Training Runs

It is to be noted that the Pareto fronts for this experiment are quite different from the one seen in Experiment 1. This is expected as random selections of configurations would not find an optimal configuration efficiently. However, our three runs of Bayesian optimization extended and changed the Pareto front. This means our approach efficiently found hyperparameter configurations that matched our multi-objective goal.

If our Bayesian optimization algorithm continued to run many times, we would see the Pareto front converge to something similar to what we found in our first experiment.

This finding also cements that Bayesian optimized methods can efficiently search for hyperparameters better than naive random search. For problems like ours, the trade-off of algorithmic complexity and extra computational demands is worth the price for optimal results.

4.3 Experiment 3

4.3.1 Experimental Setup

Fine-tuning Problem	
Pre-trained model	Llama 3 (8B model)
Fine-tuning dataset	Self-instruct (1.05 epoch)
QLoRA Configuration Hyperparameters	
LoRA rank (r)	Number of training parameters in adaptation layers
Objectives	
Training time	Time taken by the training
Training loss	Fitness to the training data

Table 4.3: Experiment 3 Configurations

In our third experiment, we wanted to observe if there would be any change from Experiment 2 by increasing the epoch and changing GPTune [20] to output one run at a time. Normally, it would be ideal to train for at least 1 epoch for a dataset, otherwise, some training data will be missed. However, we initially kept the epoch low for Experiments 1 and 2 because it kept the training time for each run low, allowing us to complete some initial experiments in time. As such, we increased the epoch from 0.19 to 1.05 because we had more time to afford and had some results. The number 1.05 results from a miscalculation due to the training hyperparameter using the number of samples instead of epochs as the measurement that dictates how much of a dataset is used. The difference between 1 and 1.05 epochs was unlikely to harm the experiment, so we continued with that number.

GPTune was also set up to run more iteratively. It would generate only one Bayesian optimized run and include the results from that run into the data that would help GPTune generate the next Bayesian optimized run. This means that GPTune could gauge what it should test next based on its previous tries, unlike previous experiments. To this end, we conducted the following experiments and calculated the Pareto front for the Random search and the Bayesian optimized approach.

10 Pilot Random runs were conducted to prime the Bayesian optimization algorithm.

5 Random runs were conducted for testing.

5 Bayesian Optimized runs were conducted for testing.

4.3.2 Experiment Results

The Pareto fronts for this experiment are similar to those in Experiment 2. One of the five Bayesian optimized runs extended the Pareto front. Once again, the random search didn't improve the Pareto front. This repeated result adds some more certainty that Bayesian optimized methods are a better search method than a random search.

Also, the training losses across the runs were generally about 0.11 lower than in Experiment 2. The reduction in training loss was expected from an increased epoch since the QLoRA [7] would have more chances to be tuned

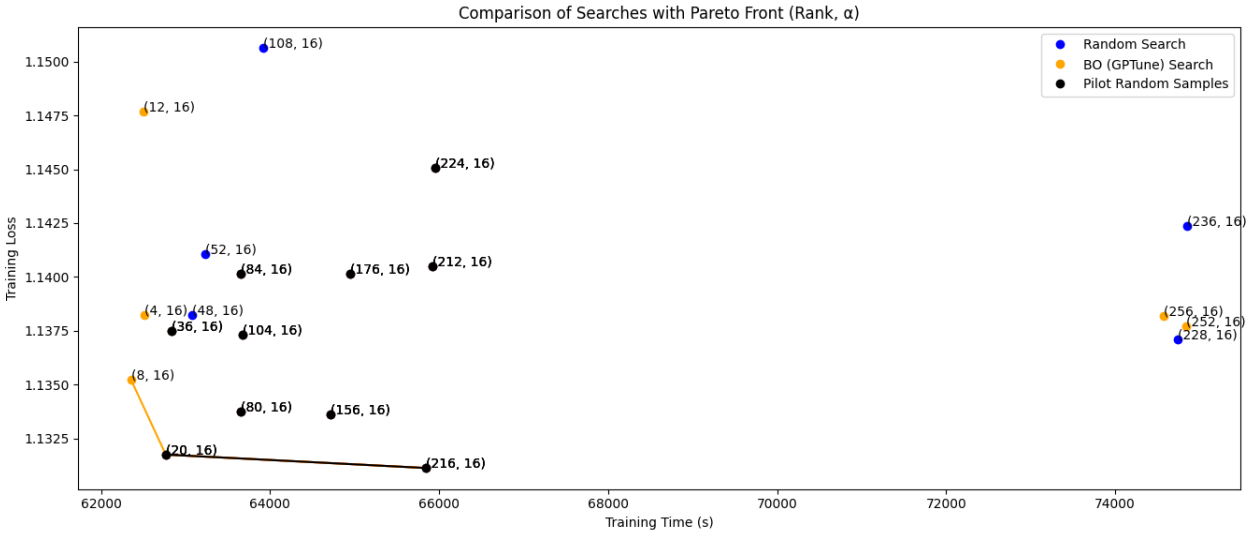

Figure 4.4: Data Plot for Experiment 3's Training Runs

towards the general concept behind the data and consequently suffer fewer errors. Increasing the epoch from 0.19 to 1.05 also increased the approximate median of the training time per run from about 3 hours and 20 minutes to about 17 hours and 40 minutes.

Overall, this finding helps support that Bayesian optimized methods can search for hyperparameters better than naive random search and that increasing the epoch leads to benefits.

4.4 Experiment 4

4.4.1 Experimental Setup

Fine-tuning Problem	
Pre-trained model	Llama 3 (8B model)
Fine-tuning dataset	LAMBADA (1.00 epoch)
QLoRA Configuration Hyperparameters	
LoRA rank (r)	Number of training parameters in adaptation layers
LoRA alpha (a)	Scale factor in which the adaptation layers affect the output
Learning rate (l)	Speed that weights change towards a theoretical optimal value
Objectives	
Training time	Time taken by the training
Training loss	Fitness to the training data

Table 4.4: Experiment 4 Configurations

In our fourth experiment, we wanted to expand our scope by adding a new hyperparameter and running a full test on a different platform. We wanted to stay at 1 epoch but couldn't repeat the same settings as Experiment 3 because it took a long time to complete. So, we decided to switch to a different but smaller dataset called LAMBADA [26] to maintain 1 epoch while reducing each training run to about 2 hours and 15 minutes.

Both rank and alpha had the same search space of numbers between 4 and 256 with a step size of 4 for random search and the Bayesian optimization search. Learning rate was added as the new hyperparameter to test with a search space between 0.00001 and 0.01 with a step size of 0.00001. Adding learning rate as a new hyperparameter would increase the search space, making the multi-objective tuning problem more complex.

GPTune [20] was also set up to run more iteratively in the same way as Experiment 3. It would generate only one Bayesian optimized run and use those results in GPTune for the next Bayesian optimized run. GPTune would gauge what it should test next based on its previous tries.

We also ran this same experiment Google Cloud [4] to test that it functions on a platform other than the SCU's WAVE HPC [31]. The experiment run on the HPC is displayed in figure 4.5 and the experiment run on the Google Cloud is displayed in figure 4.6

20 Pilot Random runs were conducted to prime the Bayesian optimization algorithm.

5 Random runs were conducted for testing.

5 Bayesian Optimized runs were conducted for testing.

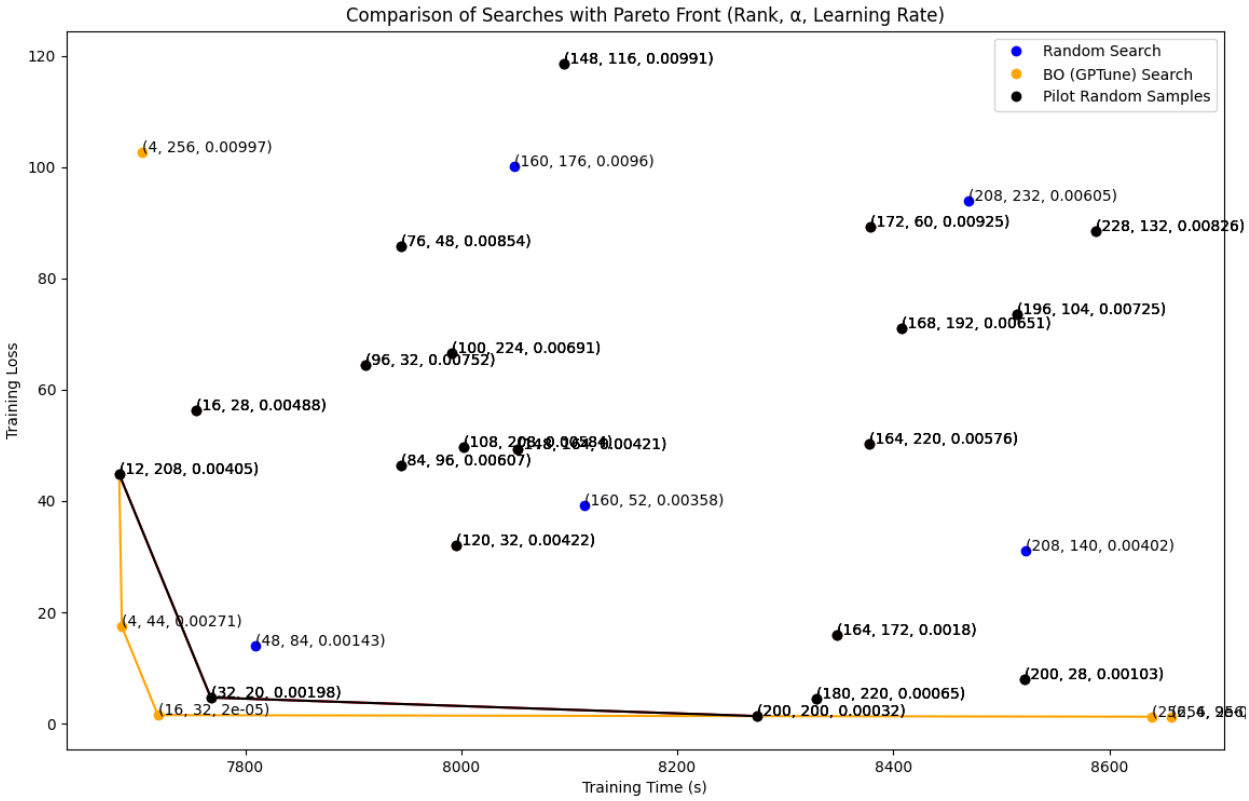

Figure 4.5: Data Plot for Experiment 4's Training Runs on the WAVE HPC

4.4.2 Experiment Results

The Pareto fronts for this experiment are more differentiated from each other in comparison to all the previous experiments. Three of the five Bayesian optimized runs extended the Pareto front for the experiments on both platforms. The Pareto fronts for the Bayesian optimized search outperformed the Pareto fronts for random search, but it still contained some points from the pilot random samples. This larger gap between the Pareto front of the Bayesian optimized search and random search is likely due to the increase in the complexity of the problem from the inclusion of learning rate. As the search space increases, it is more unlikely that random searching would yield good results.

As a result of using a different dataset, the training time and loss were different from previous experiments. However, there is a noticeable difference in the training times between the platforms. This is expected since the hardware differs between the WAVE HPC and Google Cloud.

Overall, the results help confirm that a Bayesian optimized search will more efficiently search for optimal hyperparameters than a random search in a complex problem. This difference is more apparent in this experiment than in the previous experiments.

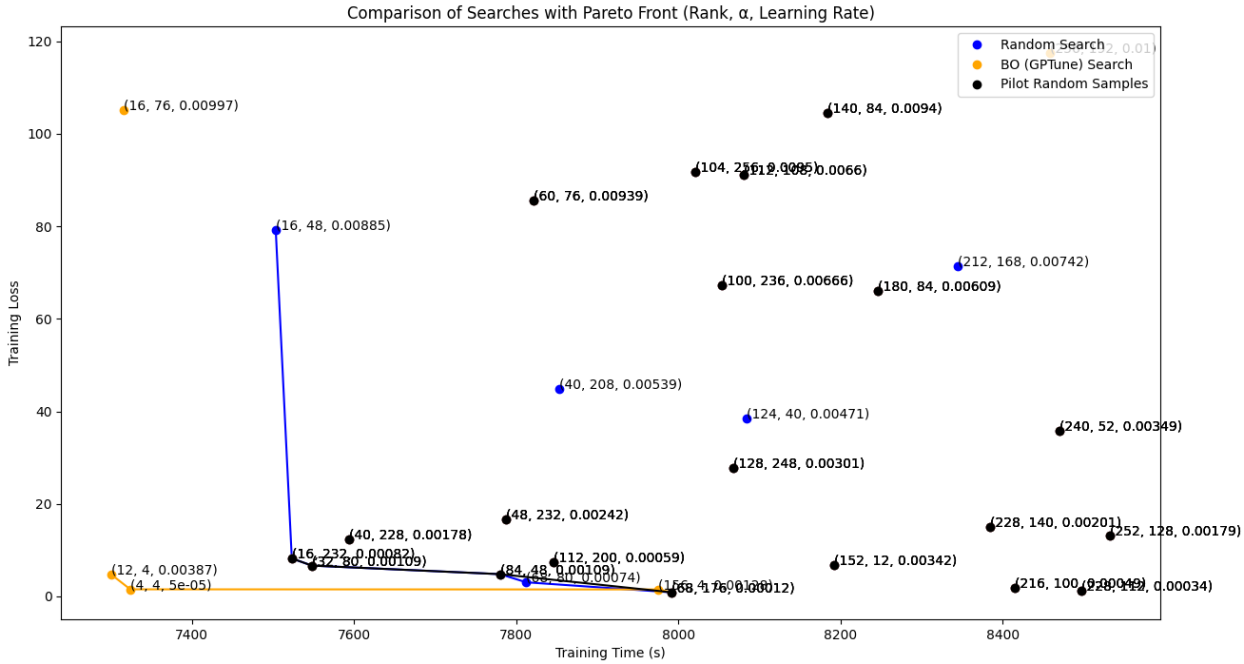

Figure 4.6: Data Plot for Experiment 4's Training Runs on the Google Cloud

4.5 Concluding Findings

Our first experiment showed minimal differences between the Pareto fronts of pseudo-random and the Pareto fronts of Bayesian optimized configurations of rank and alpha. We pinned this similarity in the Pareto front to the de-facto industry standard of choosing rank and alpha configurations in powers of 2.

Initially, it seemed that these selections of hyperparameters in powers of 2 would not largely affect the fine-tuned adapter. However, it is realized in the second experiment that this specific selection of hyperparameters is pertinent to the adapter's performance. Rank and Alpha may often be chosen in powers of 2 because they correspond to the size and training options of the frozen model weights.

Another observation is that excessive values of rank and alpha such as 128 and 256 seem to lead to relatively large training loss and training time. This large deficit in training loss may be because the adapter was not fully trained before its evaluation.

Meanwhile, we can also see that lower ranks generally have a decreased training time compared to higher ranks. This is likely because a lower rank decreases the number of parameters to train within a QLoRA [7]. Less parameters means less time needed to train the entire QLoRA at each step.

Lastly, we observed the confirmation of our hypothesis that Bayesian optimization is better than naive random search in efficiently searching for hyperparameters within search spaces, especially in larger search spaces.

Chapter 5

Constraints and Standards

5.1 Constraints

Several constraints impacted our development process.

Scope vs. Time vs. Cost

Our project's scope changed multiple times over the year. This affected how much time we could spend implementing our project and what costs to account for when researching technologies requiring a budget. Once we reached a point where we could conduct training on models, the time we had left was much smaller.

Our project has a trade-off between time and cost; optimally, the more money we spend on better hardware from cloud resources, the less time it takes to train LoRAs and conduct testing. However, utilizing cloud resources produces on-demand costs. Thus, we need to account for our limited budget for this project. Not only that, we can only request as many resources as our limit/quota provides.

Knowledge and Third-Party Software

Our project utilizes multiple third-party software in its design. Many third-party software we use are not mainstream and provide little specific documentation, making learning and using them more time-consuming. Furthermore, any bugs or issues that we encounter with this software will lead to delays since our circumstances are so specific (using GPTune [20] for LLM LoRA tuning).

Our team spent significant time learning about LLM technology throughout the project. Though this meant having a high degree of knowledge in particular domains of LLMs, we constrained our time in deciding on our scope and implementing our project.

Specifically, we looked into a few areas of research. We researched hyperparameter tuning frameworks (GPTune), LLM models, LLM parameter-efficient fine-tuning, running LLMs in resource-constrained environments, and LLM evaluation techniques.

In addition, our team had to learn specific skills to support our means to implement our project. We learned basic shell scripting, terminal and shell usage, WAVE HPC [31] usage, and Google Cloud [4] provisioning.

Hardware Availability

The SCU WAVE HPC [31] is a shared resource used by students and faculty. It is more often in use than not.

We also used Google Cloud GPUs for experiments. However, you have quotas on how many resources you can use at a time and those resources aren't always available unless you have reserved them.

Hardware Performance

Due to the shared usage of SCU's WAVE HPC cluster, we would not have sufficient computational resources to fine-tune a full-sized LLM model effectively without using a large portion of the cluster. This means we cannot train large LLMs in a reasonable amount of time because we have fewer computational resources. Because of this, we are limited to using smaller LLMs that may be less powerful and optimal.

We used the WAVE HPC platform for most experiments but found that its shared performance among other users constrained our usage. It seemed infeasible to continue moving forward with our plans while relying entirely on its capabilities, thus necessitating an attempt to use a second platform, Google Cloud [4].

Having chosen to test with Google Cloud, we utilized two 16-GB V100 GPUs alongside the 32-GB V100 GPU that WAVE HPC had. We were considering other cheaper solutions, such as RunPod, but dismissed them in favor of more trusted, reliable solutions like Google Cloud. We eventually got QLoRA [7] functioning within Google Cloud for a portion of our project.

Running tests took a considerable amount of time. Each run took upwards of a few hours to complete, which was a heavy constraint to take into account since we had a restricted timeline. Thus, it was necessary to begin our experiments ahead of our self-imposed deadlines to elaborate upon our findings effectively in graphical and verbal form.

5.2 Standards

Version Control

Our project uses Git [29] for revision control as a standard. Git is useful for storing code, collaborating, and managing revisions. It allows developers to have a local repository on their machines while swapping and updating multiple remote repositories. We use Git in our project to implement third-party technologies, as much of the code is stored on GitHub. In future work, we plan on utilizing Git for storing our framework publicly. This will also help our code be portable between machines.

Python

Because Python is an open-source language frequently updated with new features, it is not a formally approved standard programming language. However, many machine-learning technologies revolve around the use of Python. Thus, we use Python Language Reference [32] as a standard for Python syntax and code semantics to ensure readability, maintainability, and future collaboration. We utilize Python in both fine-tuning and GPTune's driver code. By standardizing our Python code, we can ensure that future work on this project is maintainable and easier to follow.

Database

Following GPTune's database scheme, we use JSON (ISO21778) [17] to store the run information from GPTune. JSON is a good standard for databases due to being simplistic yet interchangeable between programming languages. JSON is not restricted to a singularly defined digit format, provides a simple notation for collections of pairs, and can represent complex data structures through ordered lists of values. Because of its simplicity, JSON grammar will likely never change, making it very stable as a notation. Using JSON for our data, we can more easily store and analyze our results in a standardized format.

LLM Terminology

LLM terminology can be confusing since many of the terms seem to overlap. Thus, we use Google's Machine Learning Glossary [12] to properly standardize specific LLM terms used in this project.

Licensing

Many of our third-party technologies contain different licenses in their code repositories. GPTune [20] contains a BSD license, QLoRA [7] contains an MIT license, and self-instruct [33] and LAMBADA [26] datasets contain an Apache-2.0 license. Any redistributed code that utilizes these third-party software must also include the same licenses. The original creators cannot be held liable for any damages made. We will comply with these licenses for any published code repositories that include these third-party software.

Chapter 6

Societal Issues

Ethical Questions

Reviewing ACM's Software Engineering Code [13], there are some issues with ethics that we need to consider. Most LLMs [27] are trained on inherently biased data, as it originates from human-generated content. Those human products are subject to human biases. Thus, the LLM will inherit that bias. Not only that, LoRAs are also trained on that same type of human data, so the LoRA will also likely perpetuate those biases. In future work, we can try to mitigate the propagation of these biases in our QLoRA [7] by using evaluation tests that test against those biases. This method will not eliminate the bias issue but will reduce it.

There is also a privacy concern. Some datasets that train LLMs are created with invasive mechanisms. This could mean introducing to the dataset confidential information, copyrighted content, or information that was gained without consent. This violates code 3.13 which states to "be careful to use only accurate data derived by ethical and lawful means, and use it only in ways properly authorized."[13] LLMs trained on this information can be probed to retrieve or produce results that breach the privacy of their owners. We could attempt to reduce this issue by rejecting results for protected content, however, it would be an unfruitful solution since there is no blanket-wide way to identify what type of content breaches privacy.

Social Implications

Our research primarily benefits a niche community within society. If fine-tuning technology becomes more complex, our results can positively impact efficiently finding optimal fine-tuning hyperparameters.

Politics

As an auto-tuning framework for LLMs, our research has no direct political concerns.

Economic Considerations

Our initial research costs were minimal as we mainly utilized free or open-source resources. Some costs were spent on testing Google Cloud [4] GPUs to see if they were viable for current and future work (our thesis advisor provided the access to the Google Cloud resources). We ran some experiments on the Google Cloud GPUs and obtained results. Future work may need a budget for utilizing Google Cloud services to implement the full auto-tuning framework with evaluation.

Health and Safety

As an auto-tuning framework for LLMs, our research has no direct health concerns.

Manufacturability

Because our project focuses more on research and less on building a product, there are no issues regarding manufacturability.

Sustainability

Our project faces sustainability challenges due to reliance on a specific LLM library. The technological landscape of LLMs is continuing to grow, and current resources are constantly being improved and becoming outdated. However, our project can become sustainable by allowing other LLM frameworks and libraries to be used.

Environmental Impact

Training LLMs demand significant power, especially with current frameworks leveraging GPU acceleration for speed. Our results reduce the amount of energy consumed by finding optimal hyperparameters more efficiently. Furthermore, we can learn optimal configurations on smaller models and problems that consume less energy and apply them to larger models that consume more energy. While the reduced amount of training may have a positive impact by lessening power consumption, we must also consider that heavy power consumption is unavoidable in this realm of technology. This inherent consumption of power may negatively impact the environment and is something that should not be ignored.

Usability

The usage of GPTune [20] in the LLM context requires prior knowledge of using both LLM frameworks and GPTune. The project is built on open-source code and datasets. However, running our project can be time-consuming or impossible without a proper computing cluster.

Lifelong Learning

Given the novelty of LLMs in research, formal documentation is limited, necessitating independent learning. Hence, much of our work for this project involved learning independently. We faced many consecutive roadblocks, and by overcoming those roadblocks we were able to grow and learn new skills along the way.

Compassion

Our project focuses on LLM efficiency and optimization and does not directly address compassion-related societal issues.

Chapter 7

Future Work

7.1 Next Steps

Comparison to Other Algorithms

In our experiments, we only compared Bayesian optimization to either pseudo-random search or random search. While they are fine for a minimal baseline, comparing Bayesian optimization with other search algorithms such as grid search would be better. This would generate a better baseline to compare against and show the difference in performance between multiple potential algorithms.

Large Language Model Evaluation

An initial goal was to use LLM evaluations as an objective rather than training loss. LLM evaluations are better than training loss because they are one step removed from the training process. The data used in the evaluation is not used in the training. This helps prevent overfitting because the trained QLoRA [7] will not know what it will be tested on. Essentially, this is an improved way of validating the real-world usage of the produced QLoRA. The test datasets used in LLM evaluations are usually a test split from the dataset they were created from. You could also use other datasets in the evaluation to test for different forms of performance. However, LLM evaluation uses extra time and resources to evaluate a model, which we could not afford to use at the time.

Initially, we went through multiple LLM evaluation frameworks. If we add an LLM evaluation framework in future work, we will use EleutherAI's Model Evaluation Harness [10] because it provides minimal setup and easy integration of QLoRA. In comparison, other evaluation frameworks were often too complex or difficult to use.

7.2 Other Potential Work

One interesting direction that our research could go is finding the best task-specific inference hyperparameters for large language models. Finding the best inference hyperparameters in large language models, such as temperature, top k, and top p, is challenging due to the high-dimensional search space, non-intuitive relationships between parameters and

performance, time-consuming experimentation, noisy and varying results, and inherent trade-offs. These challenges make it difficult for practitioners to identify the optimal combination of hyperparameters that cater to specific use cases or application requirements. Bayesian optimization can help address these issues by intelligently exploring the search space and efficiently identifying the best hyperparameter settings with minimal trial and error. In addition, the work presented in *Ollama-grid-search: Optimizing large language models with grid search* [8] by dezoito provides a helpful starting point for automating the process of selecting the best model hyperparameters, given an LLM model and a prompt, by iterating over the possible combinations and letting the user visually inspect the results. Integrating this interactive tooling with the Bayesian optimization framework could create a powerful end-to-end system for rapidly exploring the space of LoRA and other adaptation techniques. Such a tool could be invaluable for tuning large language models to specific use cases.

Chapter 8

Conclusion

Our research focused on creating a pipeline to identify optimal hyperparameter configurations (rank, alpha) for LoRA and its variants such as QLoRA [7] and LoRA [16]. The primary objectives were to optimize for both training loss and training time. Initially, we followed conventional guidelines by selecting hyperparameters in powers of 2. We aimed to explore the relationship between rank and alpha (e.g., whether rank = 1/2 alpha results in a better adapter) and their impact on training loss and training time. Additionally, we questioned the efficacy of our hyperparameter selection method by comparing it with naively random values.

Our initial experiments showed minimal differences between the Pareto fronts of pseudo-randomly found and Bayesian-optimized configurations of rank and alpha. This similarity was due to the pilot pseudo-random samples performing significantly well. This result supported the standard practice of choosing Rank and Alpha configurations in powers of 2.

We confirmed in the second experiment that selecting hyperparameters in powers of 2 is crucial for the adapter's performance. This is because Rank and Alpha values correspond to the size and training options of the frozen model weights. Additionally, we validated that Bayesian optimization is more effective than naive random search in narrowing the hyperparameter search space.

In the third experiment, we increased the number of epochs from 0.19 to 1.05 and changed GPTune to produce the Bayesian optimized runs iteratively. Bayesian optimization outperformed random search by identifying more efficient hyperparameter configurations and extending the Pareto front. Additionally, the results indicated that increasing the number of epochs led to a lower training loss and significantly increased training time. This confirmed that more epochs can enhance model performance but at the cost of greater computational resources.

In the final iteration of our experimentation, we introduced learning rate as an additional hyperparameter and conducted experiments on both SCU's WAVE HPC [31] and Google Cloud [4]. The results showed that incorporating the learning rate into the optimization process showed the effectiveness of Bayesian optimized searches versus random search. The comparison between Wave HPC and Google Cloud also highlighted the training time difference caused by

using different hardware, reinforcing the importance of selecting appropriate computing resources for training LLMs.

Our solution offers several key advantages. It applies to all LoRA variants, making it versatile and broadly useful. Moreover, it reinforces current standards through empirical studies, providing concrete evidence to support established practices. Two limitations of our current approach are the use of training loss as a metric, which can be susceptible to overfitting and may compromise the reliability of the results, and the lack of other search algorithms for comparison, which makes it difficult to know if this is the best search method.

Future work could address these limitations by optimizing configuration hyperparameters for accuracy and evaluation loss instead of training loss, and the additional usage of other search algorithms would help create better baseline comparisons. These adjustments are expected to provide more robust and grounded results.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] Yew Ken Chia, Pengfei Hong, Lidong Bing, and Soujanya Poria. INSTRUCTEVAL: Towards Holistic Evaluation of Instruction-Tuned Large Language Models, 2023.

[4] Google Cloud. GPU platforms. `https://cloud.google.com/compute/docs/gpus#v100-gpus`, 2024.

[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[7] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient Finetuning of Quantized LLMs. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 10088–10115. Curran Associates, Inc., 2023.

[8] dezoito. ollama-grid-search: Optimizing large language models with grid search. `https://github.com/dezoito/ollama-grid-search`, 2023.

[9] Ben Fattori. Little-GPT. `https://github.com/fattorib/Little-GPT`, 2022.

[10] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A framework for few-shot language model evaluation. *Version v0. 0.1. Sept*, page 8, 2021.

[11] Georgi Gerganov. llama.cpp. https://github.com/ggerganov/llama.cpp, 2024.

[12] Google. Machine Learning Glossary. https://developers.google.com/machine-learning/glossary, 2024.

[13] Don Gotterbarn, Keith Miller, and Simon Rogerson. Software engineering code of ethics. *Commun. ACM*, 40(11):110–118, 1997.

[14] Robert B Gramacy. *Surrogates: Gaussian Process Modeling, Design, and Optimization for the Applied Sciences*. CRC Press, 2020.

[15] Soufiane Hayou, Nikhil Ghosh, and Bin Yu. LoRA+: Efficient Low Rank Adaptation of Large Models. *arXiv preprint arXiv:2402.12354*, 2024.

[16] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[17] Information technology — The JSON data interchange syntax. Standard, International Organization for Standardization, Switzerland, CH, 2017.

[18] jaymody. picogpt. https://github.com/jaymody/picoGPT/, 2023.

[19] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023.

[20] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. GPTune: Multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 234–246, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft, 2022.

[22] AI Meta. Introducing meta llama 3: The most capable openly available llm to date. *Meta AI.*, 2024.

[23] Anchen. Fine-Tuning Open-source LLaMa with QLoRa Scripts Repository. https://github.com/mzbac/qlora-fine-tune, 2023.

[24] OpenAccess-AI-Collective. Axolotl. https://github.com/OpenAccess-AI-Collective/axolotl/, 2023.

[25] OpenAI. ChatGPT. https://openai.com/blog/chatgpt/, 2022.

[26] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernandez. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1525–1534, Berlin, Germany, 2016. Association for Computational Linguistics.

[27] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[28] Amazon Web Services. Deep Learning AMI User Guide. https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html, 2024.

[29] Diomidis Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.

[30] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[31] Santa Clara University. SCU HPC: High Performance Computing. https://www.scu.edu/wave/wave-hpc/, 2024.

[32] Guido Van Rossum and Fred L Drake Jr. The python language reference. *Python Software Foundation: Wilmington, DE, USA*, 2014.

[33] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-Instruct: Aligning Language Model with Self Generated Instructions, 2022.

[34] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, and Yongqiang Ma. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. *arXiv preprint arXiv:2403.13372*, 2024.