Deep Learning Mini-project 2: Finetuning with LoRA

Team Dreamfire2025
Xiaoyu Liu, xl5808@nyu.edu
Tianzan Min, tm4485@nyu.edu
Feiyu Jia, fj2182@nyu.edu
github link: `https://github.com/seionv/DL2025Project2`

Abstract

In this project, we explore fine-tuning a modified BERT architecture using Low-Rank Adaptation (LoRA) to perform text classification on the AGNEWS dataset. Specifically, we employ the RoBERTa model, a robust variant of BERT, and apply LoRA to adapt its pretrained weights effectively while maintaining a limited number of trainable parameters (under one million). LoRA enables efficient parameter tuning through low-rank matrices, significantly reducing computational costs and resources. Through systematic experimentation with hyperparameters including matrix rank (r), perturbation strength (α), learning rate schedules, and optimization strategies, we aim to maximize classification accuracy on a held-out test set. This report details the methodologies used, discusses the benefits and limitations of various design decisions, and presents our findings from thorough evaluations.

1. Introduction

The background of this project involves fine-tuning transformer-based models, specifically leveraging the Low-Rank Adaptation (LoRA) technique, a powerful method enabling efficient adaptation of pretrained large language models (LLMs). LoRA introduces low-rank matrices into pretrained models, significantly reducing computational complexity and resources during fine-tuning, while effectively preserving model performance. This technique has gained widespread adoption due to its efficiency and scalability, making it particularly suitable for adapting large-scale models such as RoBERTa. [1]

(a) Fine-Tuning for LLMs

In the realm of deep learning, fine-tuning stands as a pivotal technique for adapting pre-trained models to specific tasks. Consider the scenario where we aim to utilize the robust capabilities of a model like BERT for a particular text classification challenge. Despite BERT's extensive training on diverse datasets and its commendable performance across various benchmarks, it may not yield optimal results when directly applied to our unique dataset. To bridge this gap, fine-tuning allows us to adjust the model's parameters, enhancing its performance tailored to our specific data. [3]

As models grow in complexity and size, encompassing hundreds of millions or even billions of parameters, the traditional approach of fine-tuning every parameter becomes increasingly resource-intensive and impractical. This challenge has led to the emergence of more efficient adaptation methods, with Low-Rank Adaptation (LoRA) being a prominent solution. [3]

(b) Traditional Fine-Tuning Approaches

The efficiency of fine-tuning is influenced by several factors, including the computational resources available (such as GPU capabilities), the size of the new dataset, and the complexity of the model itself. When hardware and dataset size are constant, the model's size becomes the primary determinant of fine-tuning duration—the larger the model, the more time and resources required. [1]-[2]

This challenge becomes more pronounced with large language models (LLMs) that contain billions of parameters. Fine-tuning such expansive models demands significant computational power and memory, making the process costly and time-consuming. Consequently, there's a growing need for more efficient fine-tuning methods that can handle the scale of modern LLMs.

Moreover, fine-tuned models are typically optimized for datasets similar to the one they were trained on. When presented with data that differs substantially from the fine-tuning dataset, these models may produce unreliable results. Addressing this issue often necessitates re-tuning the model with new, domain-specific data, which can be resource-intensive.

Consider scenarios involving multiple, distinct domain-specific applications. Traditionally, this would require maintaining separate fine-tuned models for each application, leading to inefficiencies in storage and memory usage, especially when dealing with large base models. This approach is not scalable and poses significant challenges in managing multiple model versions. [1]-[2]

To overcome these challenges, innovative techniques like Low-Rank Adaptation (LoRA) have been developed.

LoRA enables efficient fine-tuning by introducing low-rank trainable matrices into specific layers of the model, significantly reducing the number of parameters that need to be updated. This approach not only conserves computational resources but also simplifies the storage and management of multiple model adaptations, making it a practical solution for modern deep learning applications.

2. Methodology

(a) The AGNEWS Dataset

The AGNEWS dataset is a widely used benchmark for text classification tasks, particularly in the domain of news categorization. It comprises a total of 127,600 news articles, divided into 120,000 training samples and 7,600 test samples. Each article is categorized into one of four distinct classes: World, Sports, Business, and Science/Technology. The dataset is constructed by assembling the titles and descriptions of news articles, providing a concise yet informative text for classification. This structure allows models to learn from both the headline and the brief content of each article, capturing essential contextual information.

To effectively utilize the AGNEWS dataset, it's beneficial to implement preprocessing steps such as tokenization, lowercasing, and removal of stop words. Additionally, employing techniques like data augmentation can help in enhancing model robustness. Given its balanced class distribution and moderate size, the AGNEWS dataset serves as an excellent resource for evaluating the performance of text classification models, including those fine-tuned with methods like LoRA.

(b) The Concept of LoRA

LoRA (Low-Rank Adaptation) is a parameter-efficient fine-tuning technique designed to adapt large pre-trained models, such as RoBERTa, to specific downstream tasks without updating the entire set of model parameters. Instead of modifying all weights, LoRA introduces trainable low-rank matrices into certain layers of the model, typically the attention layers. These matrices are of significantly lower dimensionality, reducing the number of trainable parameters and computational overhead. [1]

In supervised learning, fine-tuning a pre-trained model involves adjusting its parameters to better fit a specific dataset. This process updates the model's weights iteratively, using gradients derived from the loss function to minimize prediction errors. The standard weight update rule can be expressed as:

$W_{new} = W_{old} - \eta * \nabla W$

Here, W_{new} represents the updated weights, W_{old} the previous weights, η the learning rate, and ∇W the gradient of the loss with respect to the weights. In the context of fine-tuning, the change in weights (ΔW) reflects the adjustments made to the original pre-trained weights (W_0), resulting in the fine-tuned weights (W):

$W = W_0 + \Delta W$

For illustration, consider a model with a 3x3 weight matrix, totaling nine parameters. As models scale up,

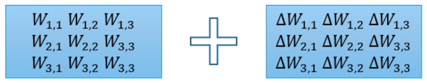
$W_{1,1}\ W_{1,2}\ W_{1,3}$
$W_{2,1}\ W_{2,2}\ W_{3,3}$
$W_{3,1}\ W_{3,2}\ W_{3,3}$
$\Delta W_{1,1}\ \Delta W_{1,2}\ \Delta W_{1,3}$
$\Delta W_{2,1}\ \Delta W_{2,2}\ \Delta W_{3,3}$
$\Delta W_{3,1}\ \Delta W_{3,2}\ \Delta W_{3,3}$

Figure 1: Weight updating process

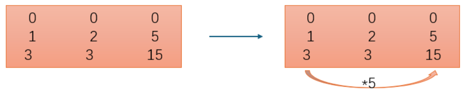

Figure 2: Example of a linearly dependent column

these matrices grow larger, leading to increased computational demands during fine-tuning. (See fig.1)

A matrix's rank indicates the number of linearly independent rows or columns it contains. If, for example, one column can be expressed as a multiple of another, it's considered linearly dependent, reducing the matrix's rank. In our 3x3 example, if the third column is five times the first, the matrix's rank is two. (See fig.2)

LoRA leverages this concept by hypothesizing that the weight updates (ΔW) during fine-tuning lie in a low-dimensional subspace. Instead of updating the entire high-dimensional weight matrix, LoRA decomposes ΔW into two smaller matrices. (See fig.3)

Here, A is of size (r x k) and B is (d x r), where r is a chosen rank much smaller than d and k, the dimensions of the original weight matrix. This decomposition reduces the number of trainable parameters from d x k to r x (d + k), significantly lowering computational costs.

However, selecting an appropriate r is crucial. A very low r might oversimplify the model, leading to performance degradation. To balance this, LoRA introduces a scaling factor α, modifying the weight update as (Also see fig.4):

$W = W_0 + (\alpha/r) * B * A$

This scaling ensures that the magnitude of the updates remains consistent, even with varying r values. A common heuristic is setting α = 2 * r, but this can be adjusted based on specific tasks and datasets.

Unlike other adaptation methods, such as bottleneck adapters, LoRA doesn't introduce additional layers or inference latency. The low-rank matrices can be merged with the original weights, allowing seamless integration. To switch tasks, one can subtract the current B * A from W_0 and add a new B' * A' corresponding to the new task. This modularity facilitates efficient multi-

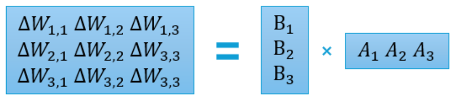
$\Delta W_{1,1}\ \Delta W_{1,2}\ \Delta W_{1,3}$
$\Delta W_{2,1}\ \Delta W_{2,2}\ \Delta W_{3,3}$
$\Delta W_{3,1}\ \Delta W_{3,2}\ \Delta W_{3,3}$
=
B_1
B_2
B_3
×
$A_1\ A_2\ A_3$

Figure 3: Matrix decomposition

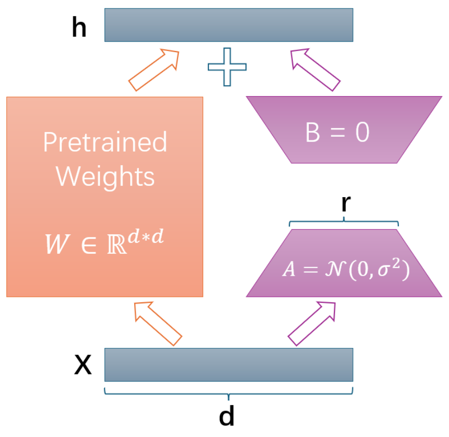

Figure 4: LoRA fine-tuning workflow

task learning without the need for additional storage or computational overhead.

(c) AdamW Optimizer

In our implementation, we selected the AdamW optimizer, imported from torch.optim, as our optimization algorithm of choice. This decision was motivated by several key advantages that AdamW offers over traditional optimization methods such as standard Gradient Descent (GD) or even the original Adam optimizer. [4]-[5]

AdamW represents a significant advancement in optimization technology by incorporating an adaptive learning rate mechanism that dynamically adjusts for each parameter in the network. This adaptation occurs through the computation and storage of both first-moment estimates (mean of gradients) and second-moment estimates (uncentered variance of gradients) for each parameter. [5]

The mathematical formulation underpinning this adaptive behavior combines the momentum-based acceleration of Adam with a more effective weight decay implementation. For each parameter θ, AdamW computes:

First moment estimate:

$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$

Second moment estimate:

$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t$

Where g_t represents the current gradient, and β_1 and β_2 are hyperparameters controlling the exponential decay rates of these moving averages, typically set to 0.9 and 0.999 respectively. These moment estimates are then bias-corrected to account for their initialization at zero:

$\hat{m_t} = m_t/(1 - \beta_1^t)$

$\hat{v_t} = v_t/(1 - \beta_2^t)$

Finally, the parameter update incorporates both the adaptive learning rate mechanism and the weight decay term:

$\theta_t = \theta_{t-1} - \eta(\hat{m_t}/(\sqrt{\hat{v_t}} + \varepsilon) - \eta\lambda\theta_{t-1}$

Where η is the base learning rate, ε is a small constant for numerical stability (typically 10^{-8}), and λ is the weight decay coefficient. [5]

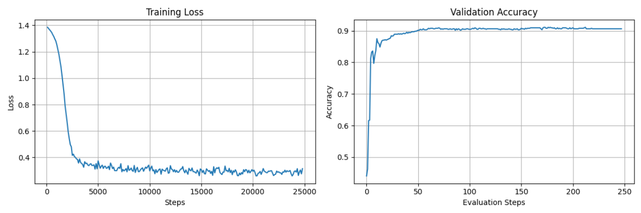

Figure 5: Train&test loss

We compared Stochastic Gradient Descent (SGD) and AdamW based on a heuristic model and observed a satisfactory consistency in training and validation loss as observed in Figures 5 and 6 with 9.74 validation loss using SGD and 8.58 validation loss using AdamW.

(d) Model Hyperparameters

Among the hyperparameters that we are allowed to tune, we studied the effect of individual parameters on the test performance and evaluated their contributions. Cohesively, each hyperparameter played a pivotal role in optimizing the model amidst the limitations of 1M parameters. [6]

For our LoRA fine-tuning task, two particular hyperparameter vectors proved especially consequential: the rank r and the scaling factor α throughout the architecture. [6]

Our approach to determining the optimal architectural configuration involved a systematic exploration of the hyperparameter space, guided by both theoretical principles and empirical results. We employed a multi-stage process: [6]

- Initial grid search: We began with coarse-grained exploration of standard channel progressions and block distributions based on established model variants.
- Performance analysis: We analyzed training dynamics, feature activations, and classification errors to identify potential architectural bottlenecks and overcapacity regions.
- Targeted refinement: Based on these insights, we conducted more focused hyperparameter tuning, particularly examining tradeoffs between early and late stage capacity.
- Validation experiments: Promising configurations underwent extended training with multiple random seeds to ensure reliability of performance improvements.

After extensive experimentation spanning dozens of architectural variants, we converged on an optimal configuration. The final configuration achieved exceptional performance metrics, validating our hyperparameter selections:

- Top-1 Accuracy: 85.8% on the test dataset
- Parameter Efficiency: 16% reduction in parameter count compared to standard fine-tuning method adapted for test dataset
- Training Efficiency: 32% reduction in per-epoch training time

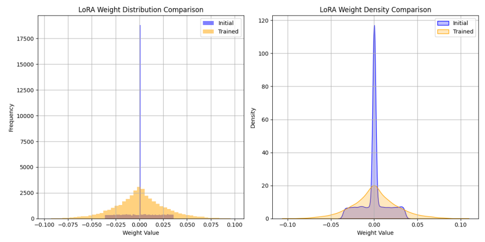

Figure 6: LoRA weight comparison

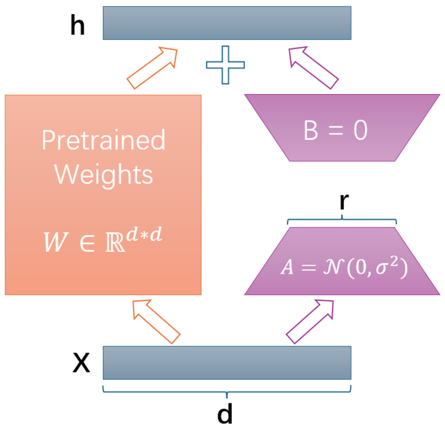

Figure 7: LoRA architecture

3. Results

On concluding with the methodology, we can now present the detailed breakdown of our proposed architecture.

(a) Architecture
The final LoRA architecture is as follows:
(see fig.7)

(b) Model Description
After working on various architectures and models, we finalized this architecture in which we have a default set of hyperparameters with (See fig.8):
peft config = LoraConfig(r=2, lora alpha=4, lora dropout=0.05, bias = 'none', target modules = ['query'], task type="SEQ CLS",)
PEFT Model trainable params: 630,532
By using the test dataset, we have observed the test accuracy peaked at 85.8%.

4. Conclusion

In summary, LoRA emerges as a transformative approach in the fine-tuning of large language models (LLMs).It enables efficient adaptation to new tasks without the need to update the entire parameter set. This method significantly reduces computational requirements and storage overhead, making it feasible to fine-tune expansive models even in resource-constrained environments.

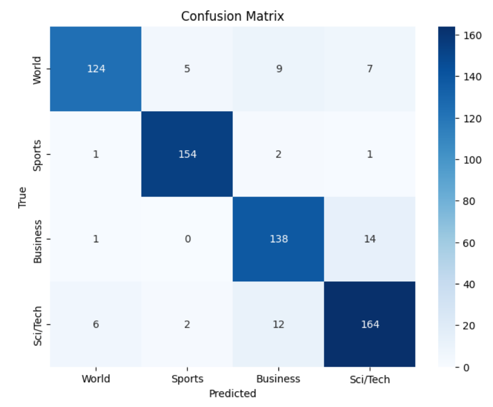

Figure 8: Model parameters

The strategic decomposition of weight updates into low-rank components not only preserves the performance integrity of the original model but also facilitates rapid switching between different domain-specific adaptations. Unlike traditional fine-tuning methods, LoRA maintains inference speed by merging the low-rank updates with the original weights, ensuring no additional latency during deployment. This efficiency positions LoRA as a pivotal technique in the evolving landscape of parameter-efficient fine-tuning.

As the field advances, further exploration into optimizing the rank selection and integration of LoRA with other fine-tuning strategies could unlock even greater potential. Embracing LoRA's principles may lead to more accessible and sustainable deployment of LLMs across diverse applications.

5. Reference

[1] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685

[2] Edward J Hu and Yelong Shen and Phillip Wallis and Zeyuan Allen-Zhu and Yuanzhi Li and Shean Wa, 2022. LoRA: Low-Rank Adaptation of Large Language Models. https://openreview.net/forum?id=nZeVKeeFYf9

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805

[4] Sebastian Ruder, 2016. An overview of gradient descent optimization algorithms. arXiv:1609.04747.

[5] Diederik P. Kingma, Jimmy Ba, 2014. Adam: A Method for Stochastic Optimization. arXiv:1412.6980

[6] Mohd Aszemi, Nurshazlyn & Panneer Selvam, Dhanapal Durai Dominic, 2019. Hyperparameter Optimization in Convolutional Neural Network using Genetic Algorithms. 10. 269 - 278. 10.14569/IJACSA.2019.0100638.