Optimizing BERT-base Fake News Classification with LoRA and Mixed Precision

Dawei Sun , Yikai Xu
New York University, Tandon School of Engineering
{ds8092, yx3845}@nyu.edu

***Abstract*—This project investigates high-performance training strategies for BERT-based fake news classification on the LIAR dataset. We compare three configurations: full-precision fine-tuning, automatic mixed precision (AMP), and LoRA-based parameter-efficient fine-tuning combined with AMP. Our results show that AMP reduces training time and activation memory substantially without degrading accuracy, while LoRA+AMP provides the strongest efficiency gains by reducing GEMM computational load and trainable parameter size. Threshold optimization further improves macro-F1 under label imbalance. Operator-level profiling reveals that AMP accelerates both GEMM and attention kernels, whereas LoRA reduces projection-layer computation but leaves attention workloads largely unchanged. Overall, LoRA+AMP achieves the best trade-off between performance and computational efficiency, offering a practical approach for scalable Transformer training under limited resources.**

***Index Terms*—Fake news detection, BERT, LoRA, mixed precision, GPU profiling**

I. Introduction

Automatic fact-checking and fake news detection have become important applications of natural language processing, especially in high-stakes domains such as politics and public policy. In this project, we study high-performance training of Bidirectional Encoder Representations from Transformers (BERT) models on the LIAR dataset, a widely used benchmark for political fact-checking.

Instead of focusing solely on accuracy, we aim to (i) achieve competitive or improved macro-F1 on our data, while (ii) reducing training time and GPU memory usage through mixed precision and parameter-efficient fine-tuning.

The detailed problem we focus on are:

- How much speedup and memory reduction can mixed-precision training provide on a modern GPU without sacrificing accuracy?
- Can Low-Rank Adaptation (LoRA) parameter fine-tuning match full fine-tuning in terms of macro-F1, while substantially reducing the number of trainable parameters and memory footprint?
- How do different combinations of techniques (baseline, baseline+AMP, LoRA+AMP) trade off speed, memory, and predictive performance?

To answer these questions, we build a unified training framework based on HuggingFace Transformers and PEFT, and we systematically compare:

1) a full-precision BERT baseline,
2) BERT with automatic mixed precision (AMP),
3) BERT with LoRA-based parameter-efficient fine-tuning with AMP.

All experiments are conducted on the official LIAR train/validation/test split, with class reweighting to address label imbalance and a validation-based threshold optimization scheme to improve macro-F1. We further use the PyTorch Profiler to capture operator-level CPU and CUDA time, memory usage, and chrome traces for a subset of batches, which allows us to analyze performance characteristics across different configurations.

II. Literature Survey

A. Pre-trained Transformer Encoders and BERT Fine-tuning

BERT introduced a pre-training and fine-tuning paradigm that enables strong downstream performance with minimal task-specific architectural changes. Devlin et al. proposed masked language modeling (MLM) and next sentence prediction (NSP) objectives to learn deep bidirectional contextual representations, which can be adapted to classification tasks by attaching a lightweight prediction head and fine-tuning the entire network end-to-end [1]. In our work, we adopt `bert-base-uncased` as the backbone and fine-tune it for fake-news detection, using macro-F1 as a primary metric due to label imbalance.

B. Fake News Detection Benchmark: LIAR

A major obstacle for automatic fake news detection is the lack of reliable, large-scale labeled benchmarks. Wang introduced the LIAR dataset, containing 12.8K short political statements labeled into six fine-grained veracity classes, accompanied by rich speaker and contextual metadata [2]. LIAR has become a standard benchmark for studying fact-checking and deception detection models. Motivated by the dataset's metadata richness, we construct a structured textual input by concatenating statement and metadata fields into a single sequence, enabling BERT to jointly encode both the claim and its contextual cues. We further binarize the six-way labels to a fake-vs-true task for clearer precision/recall trade-off analysis.

C. Parameter-Efficient Adaptation with LoRA

While full fine-tuning updates all model parameters, this becomes costly for large Transformers in both memory and

optimization overhead. Hu et al. proposed Low-Rank Adaptation (LoRA), which freezes the pre-trained weights and injects trainable low-rank update matrices into selected linear layers (e.g., attention projections), substantially reducing the number of trainable parameters while maintaining competitive accuracy [3]. A key advantage of LoRA is that it does not introduce additional inference latency, since the low-rank updates can be merged into the original weights after training. Our LoRA setup follows this paradigm by applying LoRA to the `query` and `value` projection matrices, optimizing only the low-rank parameters plus the classification head.

D. Mixed Precision Training and Hardware Efficiency

Mixed precision training aims to accelerate training and reduce memory usage by performing most computations in reduced precision while preserving model quality via selective FP32 operations. Micikevicius et al. systematically demonstrated that using half precision for activations/gradients with appropriate numerical handling can significantly improve throughput and reduce memory footprint without accuracy degradation [4]. Modern GPUs (e.g., NVIDIA A100) provide specialized tensor cores that further amplify these benefits for matrix multiplications. In our experiments, we employ PyTorch AMP to execute most GEMMs in BF16 while keeping numerically sensitive operations in FP32, and we combine AMP with LoRA to jointly reduce activation memory and trainable-state memory.

E. Positioning of Our Work

Existing literature establishes strong modeling foundations (BERT), benchmark data (LIAR), and efficiency techniques (LoRA, mixed precision). Our report focuses on an end-to-end, empirical, and systems-oriented comparison of these techniques on the LIAR task: (i) we quantify accuracy/macro-F1 trade-offs under class imbalance and apply validation-based threshold tuning; (ii) we measure training time and peak GPU memory across FP32, AMP, and LoRA+AMP; and (iii) we use operator-level profiling to explain *where* speedups and memory savings come from (e.g., GEMM kernels vs. attention kernels). This connects algorithmic choices directly to GPU execution behavior, clarifying practical efficiency limits and opportunities for future work (e.g., more efficient attention).

III. Technical Details

This section describes in detail our dataset processing, model architectures, learning objectives, optimization setup, and performance-oriented techniques. All experiments are implemented in PyTorch using HuggingFace Transformers and the PEFT library, and run on a single NVIDIA A100 GPU.

A. Task and Dataset

We use the LIAR dataset, which consists of short political statements labeled into six fact-checking categories: `pants-fire`, `false`, `barely-true`, `half-true`, `mostly-true`, and `true`. Each statement is accompanied by rich metadata, including the subject, speaker, speaker job, state, party affiliation, and a free-text context.

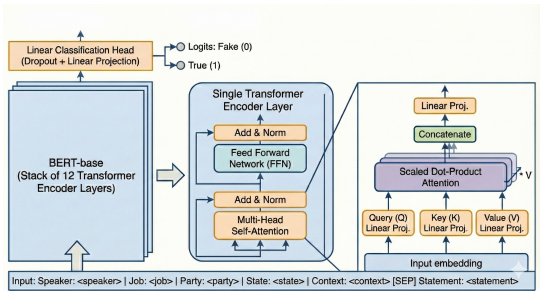

Fig. 1: Architecture and input statement of BERT classifier.

To simplify evaluation, we convert the original six-way classification into a binary fake-vs-true task:

$$\text{Fake} = \{\text{pants-fire},\ \text{false},\ \text{barely-true}\} \to 0,$$

$$\text{True} = \{\text{half-true},\ \text{mostly-true},\ \text{true}\} \to 1.$$

Rather than feeding only the bare statement into BERT, we construct a structured textual input that concatenates metadata and the statement in a consistent template:

```
Speaker: <speaker> | Job: <job> |
Party: <party> | State: <state> |
Context: <context> | Subject: <subject>
[SEP] Statement: <statement>
```

Any missing metadata fields (e.g., `speaker_job`, `party`) are imputed with the token `Unknown`. We use the `bert-base-uncased` tokenizer with a maximum sequence length of 256 tokens. For each sample, we obtain input IDs and an attention mask, along with an integer label in $\{0, 1\}$.

B. Model Architectures

Our main backbone is `bert-base-uncased` from HuggingFace, configured for sequence classification. The model is `AutoModelForSequenceClassification` instantiated with:

- 12 Transformer layers,
- hidden size 768,
- 12 self-attention heads,
- approximately 110M parameters,
- num_labels $= 2$ for binary classification.

We use the standard pooled `[CLS]` representation followed by a linear classification head:

$$h_{\text{CLS}} \in \mathbb{R}^{768}, \quad \text{logits} = W h_{\text{CLS}} + b \in \mathbb{R}^2.$$

We also increase the dropout rates in the BERT configuration to improve regularization: hidden_dropout_prob $= 0.3$, attention_probs_dropout_prob $= 0.3$.

C. Fine-Tuning Strategies

We evaluate three fine-tuning strategies to study the trade-offs between performance, efficiency, and hardware utilization on the LIAR fake-news classification task. These include full-precision fine-tuning, mixed-precision training, and parameter-efficient fine-tuning with LoRA combined with AMP.

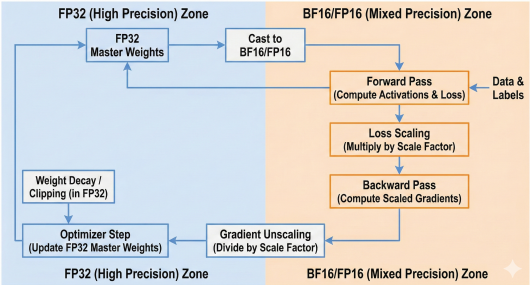

Fig. 2: Mixed precision training process with AMP.

1) Full-Precision Fine-Tuning (FP32): In the baseline configuration, all parameters of the `bert-base-uncased` backbone and the classification head are updated during training. This full-precision setting provides a strong performance reference, but has the largest memory footprint and the slowest training throughput. All forward activations, gradients, and optimizer states remain in FP32.

2) Mixed-Precision Fine-Tuning (Baseline + AMP): To improve training efficiency, we apply PyTorch Automatic Mixed Precision (AMP). Under AMP, most matrix multiplications and activations are computed in BF16, while numerically sensitive operations (e.g., layernorm, softmax, loss) remain in FP32. Optimizer states also remain in FP32 to ensure stability. This significantly reduces memory usage and accelerates training while maintaining predictive performance comparable to the FP32 baseline.

3) LoRA with Mixed Precision (LoRA + AMP): To further improve efficiency, we adopt Low-Rank Adaptation (LoRA), which introduces trainable low-rank matrices into the `query` and `value` projection weights of the self-attention layers:

$$W_{\text{eff}} = W + BA.$$

The original pre-trained weights W remain frozen, while A and B are learnable low-rank matrices with rank $r \ll d$.

Only the LoRA parameters and the classification head are updated, reducing the number of trainable parameters. Combining LoRA with AMP yields the most efficient configuration, as LoRA reduces gradient size and AMP reduces activation memory and compute cost.

4) Ablation Setup and Summary: The final ablation study compares the following three configurations:

- **Baseline (FP32)**: full fine-tuning in FP32.
- **Baseline + AMP**: full fine-tuning with automatic mixed precision.
- **LoRA + AMP**: parameter-efficient fine-tuning combined with AMP.

D. Loss Function and Class Reweighting

The LIAR dataset exhibits noticeable class imbalance after binarization, with the "fake" and "true" classes not equally represented. To mitigate this, we use a weighted cross-entropy loss:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} w_{y_i} \log p_\theta(y_i \mid x_i),$$

where w_0 and w_1 are class-specific weights, $y_i \in \{0, 1\}$ is the label, and p_θ is the model-predicted probability.

We compute the class weights as:

$$w_c = \frac{N}{C \cdot N_c},$$

where N is the total number of samples, $C = 2$ is the number of classes, and N_c is the number of samples of class c. This increases the penalty on misclassified minority-class examples, improving recall for the fake class and macro-F1.

E. Optimization and Training Strategy

We use the AdamW optimizer, linear learning rate warmup followed by decay, and global gradient norm clipping.

1) Optimizer: The optimizer is AdamW with default β-parameters, learning rate $\eta_{\max} = 2 \times 10^{-5}$, and weight decay 0.01. AdamW decouples L2 regularization from the adaptive moment updates, which is widely regarded as best practice for Transformer training.

2) Warmup and Linear Decay Scheduler: Let T denote the total number of training steps (number of batches times the number of epochs). We allocate the first 10% of steps to learning rate warmup and linearly decay the learning rate to zero over the remaining 90%:

$$\eta_t = \begin{cases} \dfrac{t}{0.1T}\,\eta_{\max}, & \text{if } t < 0.1T, \\ \eta_{\max}\left(1 - \dfrac{t - 0.1T}{0.9T}\right), & \text{if } t \geq 0.1T, \end{cases}$$

where t is the current step index.

3) Batching, Epochs, and Hardware: We use:

- batch size $= 128$,
- maximum sequence length $= 256$,
- number of epochs $= 10$.

Experiments are performed on a single NVIDIA A100 GPU with 40G memory.

F. Gradient Clipping

To prevent gradient explosion, we apply global gradient norm clipping with a maximum norm of $\tau = 1.0$. If g is the concatenation of all parameter gradients and $\|g\|_2$ its Euclidean norm, then:

$$\hat{g} = \begin{cases} g, & \text{if } \|g\|_2 \leq \tau, \\ g \cdot \dfrac{\tau}{\|g\|_2}, & \text{otherwise.} \end{cases}$$

G. LoRA Grid Search and Final Configuration

Before training the final LoRA models, we perform a grid search over key hyperparameters of the LoRA module:

- LoRA rank $r \in \{4, 8, 16\}$,
- LoRA scaling $\alpha \in \{16, 32\}$,
- LoRA dropout $\in \{0.0, 0.1\}$,
- learning rate $\in \{1 \times 10^{-5}, 2 \times 10^{-5}, 3 \times 10^{-5}\}$,
- weight decay fixed at 0.01.

For each configuration, we instantiate a fresh BERT backbone with LoRA, train for two epochs, and select the best configuration by validation macro-F1 to use for the full 10-epoch LoRA+AMP runs.

TABLE I: Test performance without threshold calibration.

Method	Accuracy	Macro F1	Fake Rec.	True Rec.
Baseline (FP32)	0.66	0.63	0.40	0.87
Baseline+AMP	0.66	0.62	0.41	0.85
LoRA+AMP	0.61	0.60	0.52	0.68

TABLE II: Test performance with threshold calibration.

Method	Thres.	Acc.	Macro F1	Fake Rec.	True Rec.
Baseline (FP32)	0.60	0.66	0.65	0.54	0.76
Baseline+AMP	0.65	0.65	0.64	0.59	0.70
LoRA+AMP	0.50	0.61	0.60	0.52	0.68

H. Threshold Optimization on Validation Set

In real-world fake news detection, missing fake statements is typically more costly than misclassifying true statements as fake. By default, a binary classifier with logits (z_0, z_1) predicts:

$$\hat{y} = \arg\max_{c \in \{0,1\}} z_c.$$

However, this rule can favor the majority class. We instead compute p_1 via softmax,

$$p_1 = \frac{\exp(z_1)}{\exp(z_0) + \exp(z_1)},$$

and apply a tunable threshold τ:

$$\hat{y}_\tau = \begin{cases} 1, & \text{if } p_1 > \tau, \\ 0, & \text{otherwise.} \end{cases}$$

We sweep τ from 0.10 to 0.90 on the validation set and select the best $\tau^\star$ by macro-F1, then fix $\tau^\star$ on the test set.

I. Evaluation Metrics

We consider accuracy, macro-F1 and fake-class recall.

Accuracy:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}.$$

Macro-F1:

$$F1_{\text{macro}} = \frac{1}{2}\left(F1_{\text{Fake}} + F1_{\text{True}}\right),$$

with

$$F1_c = 2 \cdot \frac{\text{Precision}_c \cdot \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}.$$

Fake-class recall:

$$\text{Recall}_{\text{Fake}} = \frac{\text{TP}_{\text{Fake}}}{\text{TP}_{\text{Fake}} + \text{FN}_{\text{Fake}}}.$$

J. Profiling Methodology

To analyze the computational behavior of different training configurations, we use the PyTorch Profiler on the first five batches of the second epoch for each setting, capturing operator-level CUDA time and memory usage.

IV. Results

A. Overall Classification Performance

Table I summarizes the test performance of all three configurations using the default decision rule $\hat{y} = \arg\max_k p(y = k \mid x)$.

The full baseline and Baseline+AMP achieve the same accuracy and very similar macro-F1. LoRA+AMP slightly lags in accuracy and macro-F1 but improves fake-class recall from about 0.40 to 0.52.

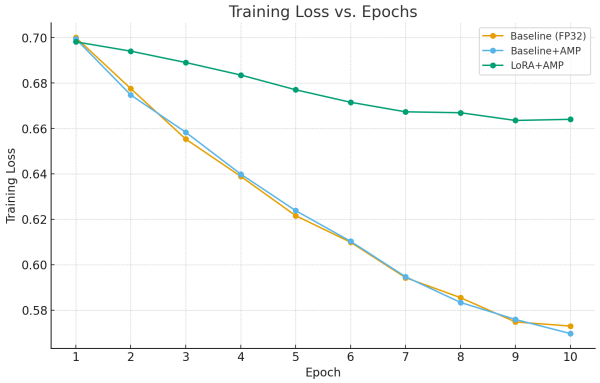

Fig. 3: Training loss vs. epoch for all three configurations.

B. Threshold Calibration on Validation Set

We sweep $\tau \in [0.10, 0.90]$ (step 0.05) over the predicted probability of the *True* class and select the best τ by validation macro-F1. The corresponding test results are shown in Table II.

Threshold calibration substantially improves macro-F1 and fake recall for the full and AMP baselines, with minimal impact on accuracy.

C. Training Dynamics and Convergence

Fig. 3 shows the training loss curves for all three configurations over 10 epochs. Baseline and Baseline+AMP converge faster and to a lower loss than LoRA+AMP, consistent with their higher test accuracy.

D. Training Efficiency and Memory Usage

Table III compares trainable parameters, average epoch time, total training time, and peak GPU memory usage.

AMP yields more than $5\times$ speedup and reduces peak memory by about 4.8 GB relative to FP32. LoRA+AMP further reduces memory and training time while updating only 0.27% of the parameters.

E. Operator-Level Profiling Results

To understand how AMP and LoRA affect the computational behavior of BERT during training, we profile the second epoch for each configuration after the first epoch has warmed up the model and data pipeline. This setup captures steady-state CUDA kernel execution while reflecting realistic end-to-end training costs.

Across all three configurations, the dominant compute hotspot is the set of matrix multiplications in BERT. These operators cover the Q/K/V projections, the feed-forward network, and other dense layers, so their behavior provides a direct view into how AMP and LoRA reshape the training workload.

TABLE III: Training efficiency and resource usage.

Method	Trainable Params	Avg Epoch (s)	Total (s)	Peak Mem (MB)
Baseline (FP32)	109.5M	78.5	785.0	20,878
Baseline+AMP	109.5M	14.7	147.4	16,057
LoRA+AMP	0.30M	13.5	135.3	11,494

TABLE IV: CUDA time (seconds) spent on matrix multiplication operators and total CUDA time.

Category	FP32	AMP	LoRA+AMP
`aten::mm`	49.25 s	3.44 s	2.32 s
`aten::addmm`	23.93 s	1.91 s	1.90 s
Total CUDA time	83.78 s	13.42 s	11.77 s

a) Matrix operators.: Table IV summarizes the CUDA time spent in `aten::mm` and `aten::addmm`, as well as the total CUDA time reported by the profiler, for the three training configurations.

`aten::mm` and `aten::addmm` are operator for matrix multiplication and addition. In the FP32 baseline, `aten::mm` and `aten::addmm` together consume about $49.25 + 23.93 \approx 73.18$ seconds of CUDA time, accounting for the majority of the 83.78 seconds total CUDA time in the profiled epoch.

With AMP enabled, the time spent in `aten::mm` and `aten::addmm` drops dramatically to $3.44 + 1.91 \approx 5.34$ seconds, which is $\approx 13.7\times$ speedup relative to the FP32 baseline. Because the computational graph is unchanged, this speedup comes entirely from executing the same GEMMs in a lower-precision. The end-to-end CUDA time for the epoch falls from 83.78 s to 13.42 s, consistent with the operator-level reduction in matmul cost.

Adding LoRA on top of AMP yields a further but smaller improvement. `aten::mm` time decreases from 3.44 s to 2.32 s, while `aten::addmm` remains essentially unchanged (1.91 s versus 1.90 s). Overall, the combined matmul time drops from 5.34 s to 4.22 s ($\approx 1.27\times$ speedup), and the total CUDA time decreases from 13.42 s to 11.77 s. This pattern is consistent with LoRA's design: only the small low-rank matrices A and B are trainable, so backward-pass computations for large dense weight matrices are skipped.

b) Memory profiling.: Table V reports the peak CUDA memory usage broken down by operator category. The results reflect two complementary effects of AMP and LoRA on memory footprint.

TABLE V: CUDA memory usage by operator category.

Category	Baseline FP32	Baseline+AMP	LoRA+AMP
Activations (MM + Linear)	10.8 GB	6.1 GB	6.0 GB
Attention activations	5.7 GB	3.9 GB	3.9 GB
Gradients (non-LoRA)	3.4 GB	3.4 GB	0.21 GB
Optimizer states	5.2 GB	5.2 GB	0.08 GB
LoRA-specific activations	–	–	0.14 GB
Peak GPU memory	20.8 GB	16.1 GB	11.5 GB

First, AMP substantially reduces activation memory, particularly in the linear projection and attention layers. Because BF16 activations occupy half the space of FP32, the activation footprint decreases from **10.8 GB** in the baseline to **6.1 GB** under AMP. Attention activations show a similar reduction (from **5.7 GB** to **3.9 GB**), consistent with the fact that the sequence-length-dependent tensors (Q, K, V, attention scores, and intermediate softmax buffers) become smaller when stored in BF16.

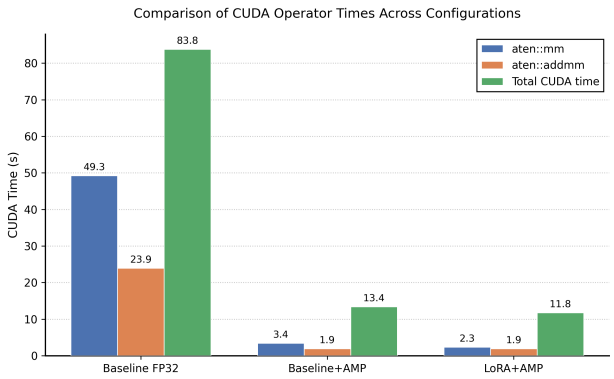

Fig. 4: CUDA time for matrix multiplication operators and total CUDA time.

Second, LoRA reduces the memory required for gradients and optimizer states. In the FP32 and AMP baselines, all parameters of the BERT backbone must store FP32 gradients and AdamW moment estimates, totaling **3.4 GB** for gradients and **5.2 GB** for optimizer states. Under LoRA, only the small low-rank matrices A and B are trainable, so the gradient footprint shrinks to **0.21 GB** and optimizer states to only **0.08 GB**. This accounts for most of the 4.6 GB reduction in memory usage relative to the AMP baseline.

V. Discussion

A. A. Interpretation of Results

The operator-level profiling reveals how AMP and LoRA improve training efficiency through distinct but complementary mechanisms. AMP provides the dominant speedup: by converting FP32 matrix multiplications into BF16 Tensor Core kernels, AMP reduces the cost of `aten::mm` and `aten::addmm` from 73.2 seconds in the baseline to only 5.35 seconds. This transformation explains the majority of the end-to-end training time reduction. LoRA introduces a more modest improvement. Because only its low-rank matrices are updated during backpropagation, LoRA reduces the number of gradient computations associated with large projection layers. This decreases `aten::mm` time to 2.32 seconds, though LoRA leaves attention computation largely unchanged because the size of Q/K/V tensors does not change. Attention kernels therefore exhibit similar runtimes across AMP and LoRA+AMP. Overall, AMP accelerates the existing computation through hardware-level optimizations, while LoRA reduces the amount of computation required for specific layers.

B. B. Challenges and Limitations

Despite the improvements gained from AMP and LoRA, several challenges remain.

First, attention kernels continue to dominate runtime even under LoRA+AMP. Because attention complexity scales quadratically with sequence length, optimizing linear projections alone cannot fully alleviate the GPU workload. Future techniques such as FlashAttention v2, long-sequence attention approximations, or block-sparse attention mechanisms may be necessary to reduce this bottleneck.

Second, LoRA introduces an additional set of design choices (rank, scaling factor, dropout), and its benefits depend on proper configuration. The grid search we performed provides a reasonable estimate of effective hyperparameters, but the search process itself incurs computational overhead and may require task-specific tuning.

Third, while AMP accelerates most kernels, it may introduce numerical instabilities in certain scenarios, especially during early training when gradients are large. Although BF16 is generally more stable than FP16, convergence sensitivity remains a potential concern for tasks with high gradient variance or noisy labels.

Finally, our experiments are limited by the size of the underlying model. Due to constraints on GPU memory and total training time, we restrict our study to `bert-base` (110M parameters). While this provides a reasonable baseline, larger Transformer architectures—such as `bert-large`, DeBERTa, or recent LLM-scale encoder models—exhibit different computational profiles and may interact with AMP and LoRA in more complex ways. Scaling to larger models could further highlight the benefits of low-rank adaptation and mixed-precision training, but would require substantially greater computational resources.

Overall, these limitations suggest that while AMP and LoRA significantly improve compute efficiency, further architectural and algorithmic strategies are needed to reduce the cost of attention and to ensure robust performance across a wider range of tasks and model scales.

VI. Conclusion

A. A. Summary of Findings

This work compared three training strategies—full-precision fine-tuning, AMP, and LoRA+AMP—for BERT-based fake news classification. AMP provides substantial speed and memory reductions with no meaningful loss in accuracy, while LoRA+AMP further lowers the computational cost by reducing backward propagation of attention matrix. Threshold tuning improves macro-F1 under class imbalance. Although LoRA reduces linear-layer computation, attention kernels remain the dominant runtime bottleneck.

B. B. Contributions

We provide: (i) a unified empirical comparison of efficiency-oriented fine-tuning strategies on the LIAR dataset, (ii) operator-level profiling that clarifies how AMP and LoRA affect GPU memory usage and attention kernels, and (iii) a validation-based threshold optimization procedure that improves macro-F1 and fake-statement recall.

C. C. Recommendations for Future Work

Future work should explore more efficient attention mechanisms (e.g., FlashAttention v2 or sparse attention), apply LoRA+AMP to larger encoder models beyond `bert-base`, and automate LoRA hyperparameter selection. These extensions may yield further efficiency and accuracy gains under limited compute budgets.

VII. Code Repository

A publicly accessible link to our source code is provided below:

https://github.com/cccchengyu/HPML-Project

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423/

[2] W. Y. Wang, ""liar, liar pants on fire": A new benchmark dataset for fake news detection," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 422–426. [Online]. Available: https://aclanthology.org/P17-2067/

[3] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[4] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017. [Online]. Available: https://arxiv.org/abs/1710.03740