TINY REPRODUCTION: LORA

Anonymous authors
Paper under double-blind review

ABSTRACT

This paper presents a reproduction study of Low-Rank Adaptation (LoRA), a technique designed to adapt large pre-trained language models by injecting trainable low-rank matrices into frozen layers. Using RoBERTa-base evaluated on the GLUE benchmark, I verify the original authors' central claims: my implementation matches the performance of full fine-tuning within 2% while reducing the number of trainable parameters by 99.7% and introducing zero additional inference latency. Beyond standard replication, I extend the analysis to profile system efficiency on a smaller model architecture.

1 INTRODUCTION

The paradigm of natural language processing (NLP) has shifted significantly toward large-scale pre-training on general domain data followed by adaptation to downstream tasks. As models continue to scale, the standard approach of full fine-tuning, which involves retraining all model parameters, has become computationally prohibitive and inefficient.

To address these challenges, Parameter-Efficient Fine-Tuning (PEFT) methods were developed to adapt large models by updating only a small fraction of the parameters. However, prior to the introduction of Low-Rank Adaptation (LoRA) (Hu et al., 2021), the prevailing state-of-the-art PEFT approaches introduced new trade-offs. Adapter layers, while parameter-efficient, insert sequential layers between existing model modules, inevitably introducing inference latency. Alternatively, prefix-tuning optimizes continuous prompt vectors but reduces the available effective sequence length for the input, thereby limiting the model's context window.

In response to these limitations, LoRA is founded on the hypothesis that the change in weights during model adaptation has a low ”intrinsic rank”. Instead of updating the full weight matrices, LoRA freezes the pre-trained model weights and injects trainable low-rank decomposition matrices into the Transformer layers. This architecture allows the trainable matrices to be merged with the frozen weights during inference, eliminating the latency overhead associated with adapter layers while preserving the model's input sequence length.

This project serves as a ”Tiny Reproduction” of the original LoRA paper, aiming to verify its core claims regarding parameter efficiency, model performance, and inference latency. Due to the computational constraints of directly recreating the original experiments, this study scales down the experimental setup to a RoBERTa-base model evaluated on selected tasks from the GLUE benchmark.

Specifically, this report seeks to validate three core results from the original study. First, I demonstrate that LoRA drastically reduces the number of trainable parameters compared to full fine-tuning. Second, I confirm that the method achieves performance comparable to full fine-tuning on downstream GLUE tasks. Finally, I verify that LoRA introduces zero additional inference latency compared to the baseline model.

In addition to replicating these core findings, I extend the analysis by conducting a holistic evaluation of training efficiency, focusing on two key metrics that go beyond standard performance benchmarks. I examine GPU memory utilization to verify the tangible impact of improving parameter efficiency through LoRA. Furthermore, I profile energy consumption to determine whether these parameter-efficiency improvements translate into distinct energy-efficiency gains.

2 Related Work

A prominent approach to parameter-efficient transfer learning is the introduction of adapter modules (Houlsby et al., 2019). This method inserts small, trainable fully connected networks between the frozen layers of a pre-trained Transformer model. During fine-tuning, only these adapter modules are updated, while the original model parameters remain fixed. This approach significantly improves parameter efficiency; for example, adapters have achieved performance within 0.4% of full fine-tuning on the GLUE benchmark while training only 3.6% of the parameters per task. However, a primary drawback of adapter layers is the introduction of inference latency. Because adapters are additional layers processed sequentially within the network, they prevent the model from fully leveraging hardware parallelism, resulting in slower inference speeds compared to the base model.

An alternative lightweight adaptation method is Prefix-Tuning (Li & Liang, 2021), which focuses on optimizing the input rather than the model architecture. This technique freezes the language model parameters and optimizes a small, continuous vector called a ”prefix” that is prepended to the input tokens. These prefixes act as ”virtual tokens” to steer the model’s generation for specific tasks. Prefix-tuning has demonstrated performance comparable to full fine-tuning with as few as 0.1% of the parameters and can outperform full fine-tuning in low-data regimes. However, its reliance on prompt tokens reduces the usable input sequence length. Because a portion of the context window is reserved for the prefix, less space is available for the actual task data, which limits the model’s ability to process long sequences.

These methods highlight a critical trade-off in parameter-efficient fine-tuning: reducing trainable parameters often comes at the cost of either inference latency or effective context window size.

3 Proposed Method

3.1 LoRA Formulation

This project implements Low-Rank Adaptation (LoRA) based on the hypothesis that the change in weights during model adaptation possesses a low ”intrinsic rank” (Hu et al., 2021). In a standard neural network dense layer, a pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$ is typically updated via full fine-tuning such that $W = W_0 + \Delta W$. LoRA constrains this update ΔW by representing it as the product of two low-rank matrices $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, where the rank $r \ll \min(d, k)$.

During the training process, W_0 is frozen and receives no gradient updates. Only A and B, which have significantly fewer parameters than the original weight matrix, are treated as trainable parameters. The modified forward pass for an input x is expressed as:

$$h = W_0x + \Delta W x = W_0x + \frac{\alpha}{r}BAx$$

Here, $\Delta W x$ is scaled by a factor of $\frac{\alpha}{r}$, where α is a constant in r. This scaling factor is critical for tuning efficiency since it acts as a normalization term that reduces the need to retune the learning rate when the rank r is varied.

To ensure the training stability of the adaptation, LoRA employs a specific initialization strategy. The matrix B is initialized to zero, while A is initialized with random values (specifically Kaiming uniform initialization in my implementation). This ensures that at the start of training, $\Delta W = BA = 0$, meaning the model initially behaves exactly like the pre-trained model.

A key advantage of this formulation is that it has no inference latency overhead since the learned matrices can be directly combined with the original weights. For deployment, LoRA computes the explicit weight matrix $W' = W_0 + \frac{\alpha}{r}BA$. Inference is then performed using W', resulting in zero additional computational overhead compared to the base model.

3.2 LoRA Implementation

To replicate LoRA, I implemented a custom PyTorch module, ‘lora_linear‘, which wraps standard linear layers. The key implementation logic is summarized in Algorithm 1.

Algorithm 1 Custom torch.nn.Module class for LoRA

Require: Input $x \in \mathbb{R}^{b \times d_{in}}$, Weights $W \in \mathbb{R}^{d_{out} \times d_{in}}$, Bias β, Rank r, Scaling Hyperparameter α

$merge\_status \leftarrow$ **False**
$scale \leftarrow \alpha / r$
$A \in \mathbb{R}^{r \times d_{in}} \leftarrow \texttt{kaiming\_uniform\_}(a = \sqrt{5})$ ▷ Init A with Kaiming Uniform
$B \in \mathbb{R}^{d_{out} \times r} \leftarrow 0$ ▷ Init B to zeros
function ToggleMerge(merge) ▷ Merge weights for inference
    if $merge$ is **True then**
        $W \leftarrow W + (BA) \cdot scale$
    else
        $W \leftarrow W - (BA) \cdot scale$
    end if
    $merge\_status \leftarrow merge$
end function
function Forward(x) ▷ Forward pass logic
    if $merge\_status$ is **True then**
        return $xW^T + \beta$
    else
        $out \leftarrow xW^T + \beta$
        $lora \leftarrow (xA^T B^T) \cdot scale$
        return $out + lora$
    end if
end function

As detailed in the algorithm, the forward pass maintains two separate computational paths. The first path computes the standard linear projection using the frozen pre-trained weights (W_0). The second path computes the low-rank adaptation term. The input x is projected down to the rank dimension r by matrix A, and then projected back up to the output dimension by matrix B. These two outcomes are summed to produce the final activation.

To enable zero inference latency overhead, the implementation includes a 'toggle_merge' function. When enabled, this function performs the calculation $W_{merged} = W_{original} + (B \times A) \times \text{scale}$ and updates the layer's weight parameter in-place. This effectively "bakes" the learned low-rank features into the standard weight matrix. The flag 'merged_weight' ensures that the expensive matrix multiplication for the LoRA path is skipped during the forward pass when weights are merged, reverting the layer to a standard linear operation.

4 Experiments

4.1 Experimental Setup

Models and Datasets: To validate the effectiveness of LoRA, I employ the RoBERTa-base model (125M parameters) (Liu et al., 2019). Following the "Tiny Reproductions" track guidelines, I scale down the evaluation scope to five datasets from the GLUE benchmark (Wang et al., 2018): SST-2, MRPC, CoLA, RTE, and STS-B. These datasets were selected to provide a representative mix of single-sentence and sentence-pair tasks while remaining computationally feasible for the available hardware. All experiments were conducted on a single NVIDIA GeForce RTX 3080 Ti (12GB VRAM).

Implementation and Hyperparameters: I adhere closely to the hyperparameters reported in the original LoRA paper (Hu et al., 2021) and the RoBERTa paper (Liu et al., 2019). For the LoRA configuration, I adapt the query (W_q) and value (W_v) projection matrices in the self-attention modules. A detailed summary of the hyperparameters for both Full Fine-Tuning (FFT) and LoRA is provided in Table 7.

A notable modification was made regarding model initialization. The original LoRA experiments initialized the model weights for the MRPC, RTE, and STS-B tasks using a checkpoint that had already been adapted to the MNLI dataset (Hu et al., 2021). Due to the high computational cost of processing the large MNLI dataset, I omit this intermediate step and fine-tune these tasks directly from the pre-trained RoBERTa-base checkpoint.

Baselines: To provide a comprehensive evaluation, I compare the custom LoRA implementation against three distinct baselines. First, Full Fine-Tuning (FFT), where all model parameters are retrained, serves as the primary performance ceiling. Second, to validate the correctness of the custom implementation, I compare results against the authors' official `loralib` package. Finally, I reference the results directly reported in the original LoRA paper (Hu et al., 2021). Comparing against both the official package and reported numbers is essential because computational constraints prevented the full replication of the authors' MNLI-initialization setup.

Resource and Energy Analysis Methodology: In addition to predictive performance, I analyze the system efficiency of LoRA. I instrument the training loop using the NVIDIA Management Library (`pyNVML`) to log energy consumption, instantaneous power draw, and GPU utilization at 0.5-second intervals.

To ensure a fair comparison between FFT and the LoRA, I perform an iso-accuracy analysis. I utilize an early stopping mechanism where training terminates once the model reaches a performance threshold defined by $\min(\text{LoRA}_{\text{perf}}, \text{FFT}_{\text{perf}}) - 0.005$. This allows for the comparison of the energy required to reach a specific convergence target rather than simply comparing fixed epoch counts.

4.2 Parameter Efficiency

Table 1 illustrates the dramatic reduction in computational overhead achieved by LoRA. While full fine-tuning requires updating all 125 million parameters of the RoBERTa-base model, my LoRA implementation optimizes only 0.3 million parameters, a reduction of approximately 99.76%. This count is identical to the figure reported in the original LoRA paper, confirming the architectural correctness of the implementation. This massive reduction validates the core claim that large models can be adapted with drastically fewer parameters compared to full fine-tuning.

Table 1: Trainable parameter counts for Full Fine-Tuning versus LoRA.

Method	Trainable Params (M)
FFT	124.6
LoRA(paper)	0.3
LoRA(my)	0.3

4.3 Performance on Downstream Tasks

Table 2 benchmarks the predictive performance of my implementation (`LoRA(my)`) against Full Fine-Tuning (`FFT`), the original paper's reported results (`LoRA(paper)`), and the official reference code (`LoRA(ref)`). The reported metrics are Matthew's correlation for CoLA, Pearson correlation for STS-B, and accuracy for SST-2, MRPC, and RTE.

Table 2: Performance comparison of RoBERTa-base fine-tuned through various methods on GLUE benchmark tasks.

Method	SST-2	MRPC	CoLA	RTE	STS-B
FFT	94.4	90.2	62.1	75.5	90.7
LoRA(paper)	95.1	89.7	63.4	86.6	91.5
LoRA(ref)	94.3	88.5	63.4	77.6	90.2
LoRA(my)	94.4	88.2	62.6	76.5	89.9

My implementation achieves performance within 2% of the FFT baseline across the GLUE subset, and notably outperforms FFT on CoLA and RTE. This corroborates the finding that low-rank adaptation preserves model capacity and does not degrade downstream task performance compared to full model updates.

A divergence is observed on MRPC, RTE, and STS-B when comparing my results to `LoRA(paper)`. This is expected since the original paper initializes these specific tasks using a model previously fine-tuned on MNLI to exploit transfer learning (Hu et al., 2021). Due to computational constraints, I initialized directly from pre-trained weights. The impact of this is most visible on RTE (a 10.1% gap), which is a small dataset that benefits significantly from MNLI transfer.

Crucially, when compared to the `LoRA(ref)` baseline, which I ran under the same constraints without MNLI initialization, my implementation performs within 1.1%. This close alignment acts as a control, isolating the initialization strategy as the variable for the performance drop rather than any flaw in the reproduction code.

4.4 Inference Overhead

Table 3 presents the inference latency on the validation sets. The results show that `LoRA(my)` incurs no significant latency overhead compared to the baseline. This empirically validates the architectural benefit of LoRA. Since the learned low-rank matrices are algebraically merged with the frozen weights prior to inference, the deployment architecture remains identical to the base model.

Table 3: Inference latency (seconds) on GLUE validation sets comparison between baseline (FFT) and LoRA models.

Method	SST-2	MRPC	CoLA	RTE	STS-B
FFT	0.62	0.47	0.36	0.67	1.10
LoRA(my)	0.63	0.46	0.36	0.65	1.09

4.5 GPU VRAM Utilization

The original LoRA study reports a memory reduction of up to $2/3$ on GPT-3 (Hu et al., 2021) without performing similar analysis for other models. I aim to analyze how this claim translates to the significantly smaller RoBERTa-base model. Table 4 presents the average VRAM usage during training. On average, my LoRA implementation reduces memory usage by 38.0% compared to Full Fine-Tuning (FFT). While this reduction confirms LoRA lowers the hardware barrier to entry, enabling training on consumer GPUs, it falls short of the 66% reduction reported for GPT-3 (Hu et al., 2021).

Table 4: Average GPU memory utilization (MB) during training.

Method	SST-2	MRPC	CoLA	RTE	STS-B
FFT	3161.7	3539.0	3437.4	10564.1	3702.3
LoRA(my)	1809.6	2072.5	1981.5	6684.0	2704.8

The variance in reduction across tasks reveals that memory savings are heavily dependent on sequence length. Tasks with longer effective sequence lengths, such as RTE and STS-B, require significantly more memory for intermediate activations. Despite LoRA, these activations must still be stored to compute gradients. Consequently, the fixed memory savings from removing optimizer states are diluted by the large activation overhead in these tasks, resulting in lower relative reductions (e.g., 26.9% for STS-B) compared to short-sequence tasks like CoLA (42.3%).

Furthermore, the discrepancy between my 38% reduction and the 66% reported for GPT-3 highlights model size dependent behavior. In smaller models like RoBERTa, activation memory constitutes a much larger proportion of the total footprint than in massive models where parameter weights dominate. Finally, I note that my Python-based implementation lacks the fused kernel optimizations of production libraries, which may slightly underestimate the potential efficiency gains.

4.6 ENERGY CONSUMPTION AND EFFICIENCY

While the primary contribution of LoRA is memory reduction, the original study also notes a 25% speedup during GPT-3 training (Hu et al., 2021). I extend this analysis to RoBERTa to determine if parameter efficiency translates to energy efficiency, which would broaden LoRA's applicability to resource-constrained environments beyond just memory limitations. I employ an iso-accuracy stopping criterion to compare the total energy required to reach identical performance levels. Tables 5 and 6 detail the power, time, and energy metrics.

Table 5: Full Fine-Tuning efficiency metrics.

Metric	SST-2	MRPC	CoLA	RTE	STS-B
Power Avg (W)	327.9	290.7	299.6	321.0	316.3
Train Time (s)	644.9	53.8	112.5	194.2	64.8
Training Epochs	4	3	8	10	3
Total Energy (kJ)	191.0	14.2	30.5	56.5	18.6

Table 6: LoRA efficiency metrics.

Metric	SST-2	MRPC	CoLA	RTE	STS-B
Power Avg (W)	334.7	325.8	325.2	325.0	327.0
Train Time (s)	756.0	60.6	433.2	128.1	95.6
Training Epochs	8	6	56	9	7
Total Energy (kJ)	227.9	18.0	127.5	37.8	28.4

LoRA exhibits a slightly higher average power draw than Full Fine-Tuning (FFT) due to increased overheads for computing low-rank adaptations during training. Furthermore, RoBERTa's relatively small memory footprint likely shifts the workload from memory-bound to compute-bound, maintaining high GPU utilization.

Regarding total efficiency, LoRA epochs completed approximately 40% faster than FFT epochs due to the elimination of gradient calculations for frozen weights. However, this throughput gain failed to translate into net energy savings. On average, LoRA increased total energy consumption, primarily driven by tasks like CoLA which required significantly more epochs to converge (56 vs 8). However, on tasks where convergence rates were similar, such as RTE, LoRA actually reduced total energy consumption (37.8 kJ vs 56.5 kJ). This suggests that while LoRA decouples training from VRAM constraints, energy efficiency is not guaranteed and is highly sensitive to the specific optimization dynamics of the downstream task.

5 CONCLUSION

This reproduction successfully verifies the core value proposition of Low-Rank Adaptation (LoRA), demonstrating that it yields predictive performance comparable to full fine-tuning while reducing trainable parameters by 99.7% and introducing zero inference latency. However, my holistic efficiency analysis reveals that the benefits reported for massive models like GPT-3 do not strictly scale down to smaller architectures like RoBERTa-base. While LoRA lowers the hardware barrier to entry, my results indicate that memory savings are capped by the dominance of activation memory over parameter storage in smaller models, and that total energy consumption can increase due to slower convergence rates. Therefore, future work should prioritize optimizing LoRA implementations for regimes where activation memory is a greater bottleneck and investigating convergence acceleration techniques to ensure that parameter efficiency translates into energy efficiency.

REFERENCES

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2790–2799. PMLR, 09–15 Jun 2019. URL `https://proceedings.mlr.press/v97/houlsby19a.html`.

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, 2021. URL `https://arxiv.org/abs/2106.09685`.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation, 2021. URL `https://arxiv.org/abs/2101.00190`.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. URL `https://arxiv.org/abs/1907.11692`.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In Tal Linzen, Grzegorz Chrupała, and Afra Alishahi (eds.), *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pp. 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-5446. URL `https://aclanthology.org/W18-5446/`.

A APPENDIX

Method	Hyperparameter	SST-2	MRPC	CoLA	RTE	STS-B
	Optimizer	AdamW				
	LR Schedule	Linear				
RoBERTa base (FFT)	Batch Size	16	16	32	32	16
	# Epochs	5	5	10	10	10
	Learning Rate	2E-05	3E-05	3E-05	3E-05	3E-05
	Weight Decay	0.01				
RoBERTa base (LoRA)	Batch Size	16	16	32	32	16
	# Epochs	60	30	80	80	40
	Learning Rate	5E-04	4E-04	4E-04	5E-04	4E-04
	Weight Decay	0.06				
	LoRA Config.	$r_q = r_v = 8$				
	LoRA α	8				
	Max Seq. Len.	512				

Table 7: The hyperparameters used for RoBERTa on the GLUE benchmark.