
Technical Report

Empirical evaluation of low-rank
adaptation for efficient fine-tuning of
large language models

Tomas Lazauskas

July 2025

Report number 9

© The Alan Turing Institute 2025

This work is licensed under Creative Commons licence CC BY-SA 4.0. To view a copy of this licence, visit <http://creativecommons.org/licenses/by-sa/4.0/>

The Alan Turing Institute is a charity incorporated and registered in England and Wales with company number 09512457 and charity number 1162533 whose registered office is at British Library, 96 Euston Road, London, England, NW1 2DB, United Kingdom.

<https://doi.org/10.5281/zenodo.16417805>

Abstract

This study evaluates Low-Rank Adaptation (LoRA) as a parameter-efficient method for fine-tuning large language models (LLMs), with a focus on the Qwen2.5-3B-Instruct model and the S1.1 dataset. Traditional full fine-tuning of LLMs demands substantial memory and computation, whereas LoRA introduces trainable low-rank matrices into selected layers, significantly reducing the number of trainable parameters while keeping the base model weights fixed.

A series of experiments was conducted on the Baskerville HPC cluster to analyse LoRA’s effects on memory usage, training time, and model performance under various configurations and GPU settings. Results indicate that LoRA reduces trainable parameters by over 99% and lowers memory usage by up to 34% on a single GPU, with further reductions observed in multi-GPU environments. Although residual memory usage increases due to LoRA’s adapter layers, the overall memory footprint remains considerably lower than that of full fine-tuning.

Performance comparisons reveal that carefully chosen LoRA configurations can approach baseline accuracy, especially when higher rank values and optimised learning rates are used. However, slower convergence and early loss plateaus were observed in some settings. These findings highlight LoRA as an effective solution for enabling scalable fine-tuning of LLMs in resource-constrained environments.

1 Introduction

LLMs have demonstrated strong performance across a wide range of natural language processing tasks. However, adapting these pre-trained models to specific downstream tasks typically requires fine-tuning, a process that is computationally expensive and memory-intensive, especially for models with billions of parameters.

In many practical scenarios, such as academic research or experimentation under constrained budgets, full fine-tuning is not feasible due to hardware limitations. This has motivated the development of parameter-efficient fine-tuning (PEFT) methods, which aim to adapt large models using a small subset of trainable parameters while preserving most of the original model’s capabilities.

One such method is LoRA [2], which introduces trainable low-rank matrices into specific layers of the model, typically in the attention and feedforward modules, while keeping the original model weights frozen. This approach drastically reduces the number of trainable parameters, offering a promising alternative to full fine-tuning.

This study is motivated by the need to fine-tune LLMs effectively in resource-constrained environments, particularly for research and development purposes. The analysis focuses on evaluating how LoRA performs in terms of memory efficiency, training time, and model performance when applied to the Qwen2.5-3B-Instruct model using the S1.1 dataset from the s1: Simple test-time scaling paper [3], which

demonstrates that LLM performance can be improved by fine-tuning on a small subset of the S1.1 dataset and applying additional computation at inference time.

2 Background

2.1 LoRA

The core idea behind LoRA is to reduce the number of trainable parameters during fine-tuning by introducing low-rank updates to the model's weight matrices.

Instead of updating the full weight matrix W with shape $[d \times k]$, LoRA applies a low-rank decomposition using two smaller trainable matrices:

$$\Delta W = A \times B,$$

where A has shape $[d \times r]$, B has shape $[r \times k]$, and $r \ll \min(d, k)$.

The updated weight becomes:

$$W' = W + \alpha \times \Delta W$$

Here, α is a scaling factor that controls the strength of the adaptation. During fine-tuning, only the matrices A and B are updated, while the original weights W remain frozen.

This strategy reduces the number of trainable parameters from $d \times k$ to $r \times (d + k)$, which can be a substantial improvement when r is small relative to d and k .

2.2 LoRA in Practice

This study uses an implementation of LoRA provided by the Hugging Face PEFT library¹, which exposes several configurable parameters:

- **r** (*default: 8*): The rank of the low-rank matrices. Smaller values reduce memory and computation costs, but may limit model capacity.
- **lora_alpha** (*default: 8*): A scaling factor applied to the low-rank update. Larger values increase the impact of the adaptation.
- **lora_dropout** (*default: 0*): Dropout probability applied to the LoRA layers during training, serving as a form of regularisation.
- **bias** (*default: "none"*): Controls the inclusion of bias terms in LoRA layers. Valid options are "none", "all", and "lora_only".

¹Hugging Face's implementation of LoRA: https://huggingface.co/docs/peft/en/package_reference/lora

- **target_modules**: A list of submodules within the model (e.g., `q_proj`, `v_proj`) where LoRA adapters are applied.
- **task_type** (*inferred or specified*): Specifies the downstream task (e.g., "CAUSAL_LM", "SEQ_CLS") and determines how LoRA is configured for that task. In this study, the task type is set to "CAUSAL_LM" for causal language modelling.

Assumptions and rules of thumb:

- The ratio of `lora_alpha` to `r` is 2:1.

2.3 Experimental Setup

This study mainly uses **Qwen2.5-3B-Instruct**², an instruction-tuned language model with approximately 3 billion parameters. It is part of the Qwen2.5 model family, designed for strong general-purpose performance with support for a wide range of tasks.

The model is instruction-aligned, meaning it has been fine-tuned to follow user prompts more effectively, making it suitable for applications involving question answering, summarisation, and reasoning tasks.

The **s1.1** dataset, introduced in the work *s1: Simple test-time scaling* [3], is a collection of 1,000 diverse and challenging questions designed to evaluate and enhance the reasoning capabilities of large language models. Each question is paired with a detailed reasoning trace from the **DeepSeek-r1** model[1] that guides the model through a step-by-step solution process. The dataset emphasises complex problem-solving skills, especially in mathematical and logical domains, and serves as a high-quality training and evaluation resource for instruction-tuned models.

The experiments were conducted on **Baskerville**³, a GPU cluster at the University of Birmingham. Each node contains four NVIDIA A100 GPUs (80GB) connected via NVLink 3.0, and nodes are interconnected with InfiniBand HDR.

All experiments were run using the **Hugging Face Transformers**⁴ and Hugging Face’s implementation of LoRA⁵ libraries, with PyTorch as the underlying framework. The training scripts⁶ were adapted from the *s1: Simple test-time scaling* work, which also makes use of the native Fast Distributed Data Parallel (FSDP) implementation in PyTorch for efficient multi-GPU training.

²Qwen2.5-3B-Instruct is an instruction-tuned language model available on Hugging Face: <https://huggingface.co/Qwen/Qwen2.5-3B-Instruct>

³Baskerville is an EPSRC Tier 2 HPC facility managed by the University of Birmingham. See: <https://www.baskerville.ac.uk/>

⁴The Hugging Face Transformers library provides state-of-the-art general-purpose architectures for natural language understanding and generation: <https://huggingface.co/docs/transformers/index>

⁵Hugging Face’s implementation of LoRA: https://huggingface.co/docs/peft/en/package_reference/lora

⁶Repository for the training script: https://github.com/alan-turing-institute/t0-1/blob/main/train/s1_31a10f2/train/sft.py

3 Results

3.1 Single-GPU Results

Table 1: Qwen2.5-3B-Instruct Fine-Tuning Experiment LoRA Parameters for Single-GPU Experiments

Experiment	r	lora_alpha	lora_dropout	bias	target_modules
0 (Baseline)	-	-	-	-	-
1	8	16	0.05	none	all-linear
2	16	32	0.05	none	all-linear
3	32	64	0.05	none	all-linear
4	64	128	0.05	none	all-linear
5	128	256	0.05	none	all-linear
6	16	32	0.05	none	q_proj, v_proj, k_proj, o_proj, gate_proj, down_proj, up_proj
7	16	32	0.05	none	q_proj, v_proj, k_proj, o_proj
8	16	32	0.05	none	q_proj, v_proj
9	16	32	0.05	none	gate_proj, down_proj, up_proj
10	16	32	0.2	none	all-linear
11	16	32	0	none	all-linear
12	16	32	0.05	all	all-linear

This section focuses on findings from all experiments conducted using a single NVIDIA A100 80GB GPU, using short fine-tuning runs—0.2 epochs on 20% of the S1.1 dataset (approximately 200 samples).

Table 1 summarises the LoRA configurations used in the experiments conducted on a single GPU. These experiments were designed to analyse the impact of various LoRA parameters on memory usage, training time, and model performance. The experiments vary in terms of rank (**r**), scaling factor (**lora_alpha**), dropout rate (**lora_dropout**), **bias** configuration, and the **target_modules** for LoRA adaptation.

Experiment 0 serves as the **baseline**, representing fine-tuning of the Qwen2.5-3B-Instruct model without LoRA.

Table 2 presents the total number of model parameters and the subset that are trainable for each LoRA configuration. These values illustrate how LoRA significantly reduces the proportion of trainable parameters.

Observations based on Table 2

- **Increasing the rank **r** and **lora_alpha** increases both the number of trainable parameters and the overall model size.** Table 2 shows that the number of trainable parameters remains relatively small compared to the full model size, ranging from 0.1% to 7.2%, depending on the configuration. This highlights the efficiency of LoRA-based fine-tuning, particularly when targeting specific modules rather than all linear layers.

Table 2: Qwen2.5-3B-Instruct Fine-Tuning Experiment Trainable Parameters

Experiment	All Params	Trainable Params	Trainable %
0 (Baseline)	3.09E+09	3.09E+09	100%
1	3.10E+09	1.50E+07	0.5%
2	3.12E+09	2.99E+07	1.0%
3	3.15E+09	5.99E+07	1.9%
4	3.21E+09	1.20E+08	3.7%
5	3.33E+09	2.39E+08	7.2%
6	3.12E+09	2.99E+07	1.0%
7	3.09E+09	7.37E+06	0.2%
8	3.09E+09	3.69E+06	0.1%
9	3.11E+09	2.26E+07	0.7%
10	3.12E+09	2.99E+07	1.0%
11	3.12E+09	2.99E+07	1.0%
12	3.12E+09	3.00E+07	1.0%

- The `gate_proj`, `down_proj`, and `up_proj` linear layers in the feedforward (MLP) module of the `Qwen2.5-3B-Instruct` model are more memory-consuming when included in the LoRA configuration than the linear layers in the attention module (`q_proj`, `k_proj`, `v_proj`, and `o_proj`). This is evident from experiments 6, 7, and 8.
- `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `down_proj`, and `up_proj` are all the linear layers in the attention module and the feedforward (MLP) module of the `Qwen2.5-3B-Instruct` model, and match the `all-linear` target module in the LoRA configuration. This is evident from experiments 1, 2, and 3.
- As expected, `lora_dropout` does not affect the number of trainable parameters, since it is a regularisation technique that does not alter the model architecture (see experiments 2, 10, and 11). However, it does influence training time and loss, as shown in Table 5.
- Similarly, the `bias` parameter has little effect on the number of trainable parameters, as it is a hyperparameter that governs whether bias terms are included in the adaptation layers (see experiments 2 and 12 in Table 2). Nonetheless, it impacts training dynamics, including runtime and final loss, as later seen in Table 5.

Table 3 reports GPU memory usage during fine-tuning, as measured by tooling provided by PyTorch. Memory consumption is broken down into four components: `model parameters`, `optimizer states`, `gradients`, and `residual`, providing insight into how LoRA affects memory efficiency compared to full fine-tuning. **Peak GPU**

memory usage is also reported, along with the reduction in memory usage compared to the baseline (experiment 0).

In this context, **residual** memory refers to GPU memory used for storing intermediate activations and temporary tensors generated during the forward and backward passes. It may also include overhead from memory fragmentation, padding, or other runtime operations. While not directly attributed to model parameters, optimizer states, or gradients, residual memory is still significantly influenced by model architecture and training dynamics.

Table 3: Qwen2.5-3B-Instruct Fine-Tuning Experiment Memory Usage Reported by PyTorch

Experiment	Model Params (GB)	Optimizer States (GB)	Gradients (GB)	Residual (GB)	Peak GPU Mem (GB)	Reduction (GB)	Reduction (%)
0 (Baseline)	11.50	22.99	11.50	21.99	67.97	—	—
1	11.55	0.11	0.06	33.29	45.01	22.96	34
2	11.61	0.22	0.11	33.24	45.18	22.79	34
3	11.72	0.45	0.22	33.13	45.52	22.45	33
4	11.94	0.89	0.45	32.90	46.18	21.79	32
5	12.39	1.78	0.89	32.46	47.52	20.45	30
6	11.61	0.22	0.11	33.24	45.18	22.79	34
7	11.52	0.05	0.03	33.32	44.93	23.04	34
8	11.51	0.03	0.01	33.34	44.89	23.08	34
9	11.58	0.17	0.08	33.27	45.10	22.87	34
10	11.61	0.22	0.11	33.24	45.18	22.79	34
11	11.61	0.22	0.11	33.24	45.18	22.79	34
12	11.61	0.22	0.11	33.24	45.18	22.79	34

Observations based on Table 3

- **LoRA significantly reduces the training memory footprint, especially for optimizer states and gradients.** This reduction is directly tied to the number of trainable parameters. For instance, in experiment 0 (full fine-tuning), optimizer states consume 22.99 GB and gradients 11.5 GB. In experiment 1 (LoRA with $r = 8$, $\text{lora_alpha} = 16$), these values drop to 0.11 GB and 0.06 GB—only 0.5% of the full model’s trainable parameters—representing 0.48% and 0.52% of the baseline memory, respectively.
- Memory usage for optimizer states and gradients scales roughly proportionally with the number of trainable parameters, as seen across experiments 1 to 5.
- Even at high ranks (e.g., experiment 5 with $r = 128$, $\text{lora_alpha} = 256$), LoRA achieves a substantial reduction in peak memory usage: 47.52 GB vs. 67.97 GB in full fine-tuning ($\sim 30\%$ lower).
- Surprisingly, **residual memory is higher with LoRA** than with full fine-tuning. For example, experiment 0 (no LoRA) uses 21.99 GB of residual

memory, while experiment 1 uses 33.29 GB. This increase is potentially due to additional activations introduced by LoRA’s adapter layers.

- Residual memory shows a slight decreasing trend as LoRA rank increases, though this may reflect non-deterministic factors such as memory fragmentation, PyTorch kernel selection, or interaction with dropout layers.
- A major contributor to the residual memory is the block size of 32,768 tokens, which was kept consistent with the original s1 study[3]. This large block size leads to significant memory usage from intermediate activations during the forward and backward passes, and is a key driver of the residual memory footprint.

Table 4 shows GPU memory usage during fine-tuning as reported by `nvidia-smi`, which reflects total memory allocation from the hardware perspective. Unlike PyTorch’s tooling, `nvidia-smi` captures all memory usage, including CUDA runtime buffers, library overhead, and system-level allocations, providing a more holistic view of the GPU memory footprint. The **sm %** column indicates the percentage of GPU memory used, while **GB** converts this to gigabytes. The **Diff (GB)** column represents the difference in memory usage compared to the peak memory usage in Table 3, allowing for a direct comparison between PyTorch’s and `nvidia-smi`’s memory reports.

Table 4: Qwen2.5-3B-Instruct Fine-Tuning Experiment Memory Usage Reported by `nvidia-smi`

Experiment	sm %	GB	Diff (GB)
0 (Baseline)	87	69.06	1.09
1	70	55.57	10.56
2	71	56.36	11.18
3	60	47.63	2.11
4	65	51.60	18.70
5	68	53.98	21.52
6	66	52.39	7.21
7	68	53.98	9.05
8	62	49.22	4.33
9	59	46.83	1.73
10	59	46.83	1.65
11	67	53.18	8.00
12	68	53.98	8.80

Observations based on Table 4

- The memory usage reported by `nvidia-smi` is consistently higher than that reported by PyTorch in Table 3. This discrepancy is expected, as `nvidia-smi` reflects the total GPU memory allocation, including not only model-related components (parameters, optimizer states, gradients, and residuals) but also:
 - CUDA context and runtime buffers
 - Memory allocated by third-party libraries such as NCCL or cuBLAS
 - Overhead from Python processes and system-level operations
- In contrast, PyTorch’s internal profiler reports only memory explicitly allocated and tracked by its tensor operations, often underestimating the full hardware-level memory footprint. For instance, in experiment 0 (full fine-tuning), PyTorch reports a peak memory usage of 67.97 GB, whereas `nvidia-smi` reports 69.06 GB, a difference of approximately 1.1 GB, likely attributable to CUDA runtime and system overhead.
- Notably, in LoRA experiments (e.g., experiments 1–5), this gap appears to widen, possibly due to the additional adapter layers and associated kernel launches, which introduce more runtime complexity and memory fragmentation.

Table 5 summarises the training runtime and final training loss for each fine-tuning experiment. These metrics provide insight into the computational efficiency and convergence behavior of LoRA configurations compared to full fine-tuning. By examining how different LoRA settings affect training duration and loss, it becomes possible to assess the trade-offs between parameter efficiency and model performance.

Table 5: Qwen2.5-3B-Instruct Fine-Tuning Experiment Training Time and Loss

Experiment	train_runtime (s)	train_loss
0 (Baseline)	104.16	1.04
1	98.43	1.17
2	105.50	1.15
3	100.56	1.13
4	101.44	1.11
5	105.29	1.09
6	104.41	1.15
7	78.13	1.17
8	70.51	1.17
9	85.67	1.16
10	100.45	1.15
11	95.73	1.15
12	113.12	1.15

Observations based on Table 5

- **Baseline fine-tuning (experiment 0) yields the lowest training loss (1.04)** but also incurs a longer runtime (104.16 minutes), illustrating the trade-off between optimisation quality and computational cost when all parameters are updated.
- **LoRA configurations generally result in higher training loss (1.09–1.17)** when trained for the same duration as the baseline. This is expected, as LoRA updates only a small subset of model parameters. However, certain setups, particularly experiments 4 and 5 with higher `r` and `lora_alpha`, achieve competitive loss (1.11 and 1.09), demonstrating that LoRA could approach baseline fine-tuning performance with far fewer trainable parameters.
- **Training time is generally reduced for LoRA**, especially when fewer modules are targeted (e.g., experiments 7 and 8), which show the shortest runtimes (78.13 and 70.51 minutes). However, this comes at the cost of higher loss, highlighting a trade-off between efficiency and convergence.
- LoRA effectiveness is sensitive to parameter size: once the number of trainable parameters exceeds $\sim 1\%$ of the total model size (e.g., experiments 2–5), training time becomes comparable to baseline fine-tuning. This suggests that **the greatest efficiency gains from LoRA occur when the trainable parameter count is kept low**.
- Experiment 12 (`bias = "all"`) shows the longest runtime (113.12 minutes) despite a moderate loss (1.15), implying that enabling bias adaptation may introduce non-trivial overhead without clear performance benefits in short training runs.

3.2 Multi-GPU Experiments

While previous experiments focused on single-GPU fine-tuning with LoRA, this section investigates how memory usage and training dynamics evolve when scaling to multiple GPUs. The objective is to assess whether LoRA maintains its memory efficiency in a multi-GPU environment—a common approach in large-scale model training.

Experiments were conducted using 1, 2, 4, 8, and 16 NVIDIA A100 80GB GPUs, with 4 GPUs available per node on the Baskerville cluster. The same Qwen2.5-3B-Instruct model was fine-tuned on the full S1.1 dataset for up to 8 epochs. This configuration enables analysis of how key memory components—such as model parameters, optimizer states, gradients, and residuals—scale with the number of GPUs.

Table 6, similarly to Table 1 for single-GPU experiments, lists the LoRA configurations used for extended fine-tuning experiments on the full S1.1 dataset and for multiple epochs.

Table 6: Qwen2.5-3B-Instruct Fine-Tuning Experiment LoRA Parameters for Multi-GPU Experiments

Experiment	r	lora_alpha	lora_dropout	bias	target_modules
0 (Baseline)	NA	NA	NA	NA	NA
LoRA 1	64	128	0.05	none	q_proj, v_proj, k_proj, o_proj
LoRA 2	64	128	0.05	none	gate_proj, down_proj, up_proj
LoRA 3	64	128	0.05	all	all-linear
LoRA 4	64	128	0.05	none	all-linear
LoRA 5	64	128	0.1	none	all-linear
LoRA 6	32	64	0.05	none	all-linear
LoRA 7	16	32	0.05	none	all-linear
LoRA 8	16	32	0	all	gate_proj, down_proj, up_proj
LoRA 9	4	16	0	all	up_proj, down_proj
LoRA 10	8	16	0	all	q_proj, v_proj
LoRA 11	16	32	0	all	q_proj, v_proj
LoRA 12	32	64	0	all	q_proj, v_proj
LoRA 13	16	32	0	all	q_proj, v_proj, up_proj, down_proj
LoRA 14	16	32	0	all	q_proj, v_proj, gate_proj, down_proj, up_proj
LoRA 15	16	32	0	none	gate_proj, down_proj, up_proj

Tables 7 and 8 present GPU memory usage during baseline (no LoRA) and LoRA 4 fine-tuning of the Qwen2.5-3B-Instruct model across various multi-GPU configurations. The data is collected using PyTorch’s memory profiling tools. As in Table 3, memory usage is broken down into four components: **model parameters**, **optimizer states**, **gradients**, and **residual**. The **Peak GPU Memory** column indicates the maximum memory usage observed per GPU during training, while the **Reduction (GB)** and **Reduction (%)** columns show the memory savings compared to the single-GPU baseline. For multi-GPU configurations, memory values are averaged across all GPUs. **Reduction (%) per GPU** indicates the average memory reduction per GPU when using multiple GPUs.

Table 7: Qwen2.5-3B-Instruct Baseline (no LoRA) Fine-Tuning Experiment Multi-GPU Memory Usage Reported by PyTorch

# GPUs	Model (GB)	Opt. (GB)	Grad. (GB)	Residual (GB)	Peak (GB)	Red. (GB)	Red. (%)	Red. (%) / GPU
1	11.50	22.99	11.50	21.99	67.97	–	–	–
2	11.50	11.50	5.75	21.31	50.05	17.92	26.4	13.2
4	11.50	5.75	2.87	20.39	40.51	27.46	40.4	10.1
8	11.50	2.87	1.44	14.04	29.85	38.12	56.1	7.0
16	11.50	1.44	0.72	13.56	27.22	40.75	60.0	3.7

LoRA 4 was chosen for comparison because it represents a balanced configuration and achieves competitive performance in terms of training loss while maintaining reasonable memory efficiency.

Table 8: Qwen2.5-3B-Instruct LoRA 4 Fine-Tuning Experiment Multi-GPU Memory Usage Reported by PyTorch

# GPUs	Model (GB)	Opt. (GB)	Grad. (GB)	Residual (GB)	Peak (GB)	Red. (GB)	Red. (%)	Red. (%) / GPU
1	11.94	0.89	0.45	32.90	46.18	–	–	–
2	11.94	0.45	0.22	26.48	39.09	7.09	15.4	7.7
4	11.94	0.22	0.11	22.92	35.19	10.99	23.8	5.9
8	11.94	0.11	0.06	15.06	27.17	19.01	41.2	5.1
16	11.94	0.06	0.03	14.08	26.11	20.07	43.4	2.7

Observations Based on Tables 7 and 8

- The memory footprint for model parameters remains constant (~ 11.5 – 11.94 GB) across all GPU configurations. This is expected due to data parallelism, which replicates model weights on each GPU. Reducing this memory per GPU requires model parallelism, possible through frameworks such as DeepSpeed, Megatron-LM, or via parameter sharding using Fully Sharded Data Parallel (FSDP).
- Memory usage for optimizer states and gradients decreases approximately by half with each doubling of GPU count, demonstrating expected scaling behaviour under data parallelism.
- **Residual memory decreases as more GPUs are used**, dropping from 21.99 GB (1 GPU) to 14.04 GB (8 GPUs) for the baseline model, and from 32.90 GB (1 GPU) to 15.06 GB (8 GPUs) for LoRA 4. This reduction is significant, indicating that distributing the workload across multiple GPUs allows for more efficient memory usage of intermediate activations and temporary tensors.
- **The rate of residual memory reduction is non-linear.**
 - For the baseline model:
 - * The drop from 21.99 GB (1 GPU) to 20.39 GB (4 GPUs) is modest.
 - * The reduction becomes more substantial between 4 GPUs (20.39 GB) and 8 GPUs (14.04 GB).
 - * The decrease from 8 GPUs (14.04 GB) to 16 GPUs (13.56 GB) is minimal.
 - For the LoRA 4 model:
 - * The drop from 32.90 GB (1 GPU) to 26.48 GB (2 GPUs) is significant.
 - * The reduction continues but slows, from 26.48 GB (2 GPUs) to 22.92 GB (4 GPUs), and further to 15.06 GB (8 GPUs).
 - * The decrease from 8 GPUs (15.06 GB) to 16 GPUs (14.08 GB) is again minimal.

- Per-GPU memory savings plateau beyond 8 GPUs, especially for LoRA, where residual memory becomes dominant and does not scale down as efficiently.
- **This suggests diminishing returns in residual memory reduction beyond a certain point as more GPUs are added.** The trend indicates that some memory components may become bottlenecks or remain unchanged despite GPU scaling, highlighting the need for further investigation into memory allocation behaviour, potentially involving factors such as buffer allocations or non-shared caches.

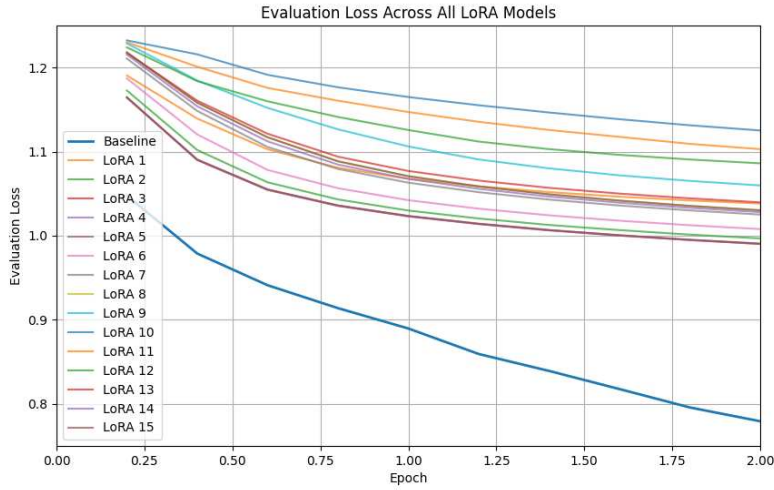


Figure 1: Evaluation loss curves for all fine-tuning experiments up to 2 epochs, comparing LoRA configurations and full fine-tuning on the S1 dataset. Lower loss indicates better model performance.

Figure 1 illustrates the evaluation loss curves for fine-tuning experiments conducted on the S1.1 dataset over two training epochs. The figure compares different LoRA configurations against a baseline, as described in Table 6. The curves show how the evaluation loss evolves over time for each configuration, with lower loss indicating potentially better model performance.

Observations based on Figure 1

In Figure 1, the baseline model consistently achieves the lowest evaluation loss during the fine-tuning process, demonstrating the best overall generalisation. In contrast, all LoRA models exhibit higher evaluation loss than the baseline, though their performance varies. Some configurations, such as LoRA 3, LoRA 4, and LoRA 5, perform relatively better, achieving losses closer to the baseline. These configurations

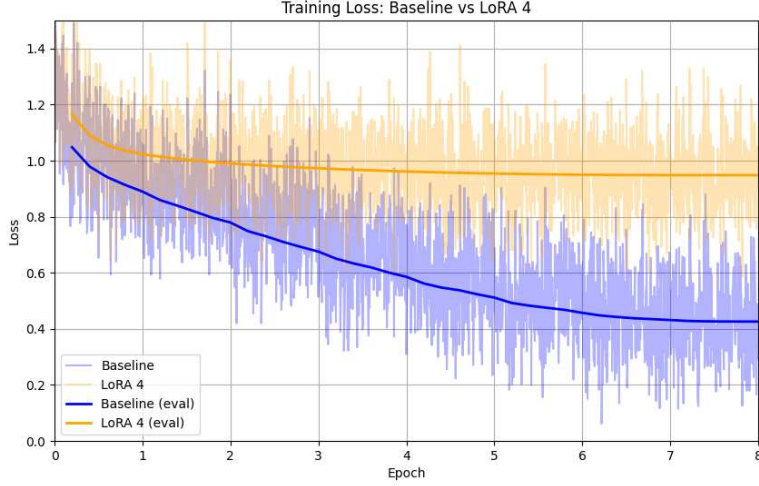


Figure 2: Training and evaluation loss comparison between Baseline and LoRA 4 models over 8 epochs.

share similar LoRA parameters ($r=64$, `lora_alpha=128`), with `lora_dropout` and `bias` having only minor impact on final performance.

Figure 2 presents a comparison of training and evaluation loss between the Baseline model and the LoRA 4 configuration over the course of 8 training epochs.

Observations based on Figure 2

As shown in Figure 2, the Baseline model (blue) demonstrates a steady and consistent decrease in both training and evaluation loss, indicating effective learning and generalisation throughout the 8 epochs. In contrast, the LoRA 4 model (orange) shows a much slower rate of improvement. Its loss begins to plateau after just 2–3 epochs, suggesting limited capacity to further minimise the loss and adapt to the dataset.

This plateau is likely due to the restricted number of trainable parameters in the LoRA 4 setup, which represents only a small fraction of the model’s total parameter count. As a result, LoRA 4 may require more training epochs to achieve convergence or may benefit from increased trainable capacity (e.g., higher rank r) or a more aggressive learning rate to accelerate adaptation.

The LoRA 4 configuration was selected as the best-performing LoRA setup thus far. To further investigate its potential, additional experiments were conducted to explore the impact of varying learning rates on the model’s convergence behaviour. Details of these experiments are provided in Table 9, while the corresponding evaluation loss curves are shown in Figure 3. In these experiments, the learning rate is varied while keeping all other LoRA parameters constant, allowing for a focused analysis of its effect on model performance.

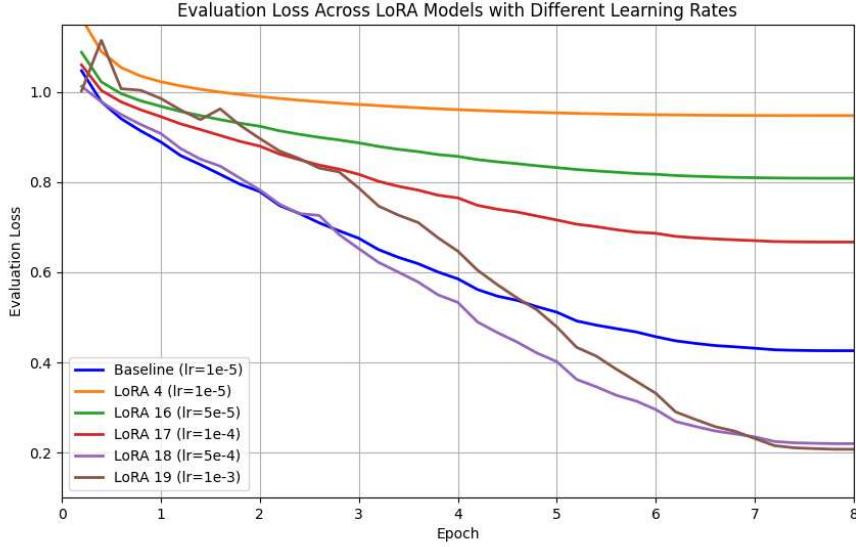


Figure 3: Evaluation loss across different learning rates over 8 training epochs.

Table 9: Qwen2.5-3B-Instruct Fine-Tuning Experiment LoRA Parameters for Multi-GPU Experiments with Varying Learning Rates

Experiment	r	lora_alpha	lora_dropout	bias	target_modules	learning_rate
0 (Baseline)	NA	NA	NA	NA	NA	1×10^{-5}
LoRA 4	64	128	0.05	none	all-linear	1×10^{-5}
LoRA 16	64	128	0.05	none	all-linear	5×10^{-5}
LoRA 17	64	128	0.05	none	all-linear	1×10^{-4}
LoRA 18	64	128	0.05	none	all-linear	5×10^{-4}
LoRA 19	64	128	0.05	none	all-linear	1×10^{-3}

Observations based on Figure 3

- The Baseline and LoRA 4 models use a learning rate of 1×10^{-5} , while LoRA 16, LoRA 17, LoRA 18, and LoRA 19 use learning rates of 5×10^{-5} , 1×10^{-4} , 5×10^{-4} , and 1×10^{-3} , respectively.
- Until around epoch 3, the Baseline and LoRA 18 models exhibit similar evaluation loss. However, after epoch 3, the LoRA 18 model continues to improve with a steady decrease in evaluation loss, while the Baseline model begins to plateau. This suggests that LoRA 18, with a learning rate of 5×10^{-4} , is learning more effectively.
- The LoRA 19 model initially (up to epoch 3) shows unstable performance, with evaluation loss increasing — likely due to an excessively high learning rate (1×10^{-3}) causing convergence issues. However, after epoch 3, its performance improves steadily, and by around epoch 7, it matches the evaluation loss of

LoRA 18. By epoch 8, LoRA 19 slightly outperforms LoRA 18, indicating eventual successful convergence despite the rough start.

- Overall, models using higher learning rates 5×10^{-4} and 1×10^{-3} (LoRA 18 and LoRA 19 respectively) demonstrate stable and effective convergence, making them strong candidates for optimal performance.

The final set of experiments evaluates the performance of the LoRA 18 and LoRA 19 models in comparison to the **Baseline** model using the `aime24_nofigures` benchmark from the s1: Simple test-time scaling paper[3]. This benchmark consists of a subset of questions from the AIME 2024 mathematics competition that do not require visual interpretation, making it appropriate for evaluating large language models on symbolic and logical reasoning tasks. The `aime24_nofigures` benchmark offers meaningful insights but may not be the most representative choice, since larger models, like 14B or 32B, typically excel in these tasks. It is included here to illustrate performance trends.

The evaluation tools used, covering formatting, inference, and scoring, are taken directly from the original s1 paper, ensuring comparability between the results reported here and those of prior work.

Table 10 below presents the proportion of correctly solved problems on the `aime24_nofigures` benchmark. Each score represents the proportion of questions correctly solved by the model (maximum = 1.0).

Table 10: Evaluation of LoRA 18 and LoRA 19 models with Baseline on `aime24_nofigures`

Model Name	<code>aime24_nofigures</code> Score
simplescaling/s1.1-3B	0.0667
TomasLaz/t0-s1.1-3B-3.2e	0.0667
TomasLaz/t0-s1.1-3B-LoRA18-3.2e	0.0333
TomasLaz/t0-s1.1-3B-LoRA19-3.2e	0
TomasLaz/t0-s1.1-3B-8e	0.0667
TomasLaz/t0-s1.1-3B-LoRA18-8e	0.0667
TomasLaz/t0-s1.1-3B-LoRA19-8e	0

Observations based on Table 10

- The **Baseline** model achieves the same performance (0.0667) as the **S1 reference model**, confirming the validity and consistency of the fine-tuning and evaluation process.
- **LoRA 18** performs moderately well after 3.2 epochs (0.0333) and matches baseline performance after 8 epochs (0.0667). This indicates that LoRA-based

fine-tuning can achieve comparable generalisation to full fine-tuning, though it may require extended training time to compensate for its lower parameter capacity.

- **LoRA 19** fails to solve any problems in both 3.2 and 8 epoch configurations. Despite its initially promising training loss trajectory, the high learning rate (1×10^{-3}) used in this configuration likely led to poor generalisation. This suggests that overly aggressive learning rates can cause unstable optimisation and convergence to suboptimal minima.
- Overall, **LoRA 18** emerges as a more reliable configuration for parameter-efficient fine-tuning, provided that training is allowed to continue for sufficient epochs. **LoRA 19** highlights the risks of using overly high learning rates in resource-efficient fine-tuning setups, particularly when the target tasks involve complex reasoning.

4 Conclusions

4.1 Memory Usage

- LoRA significantly reduces the memory footprint of fine-tuning large language models, especially in the optimizer states and gradients.
- The reduction in memory usage correlates with the number of trainable parameters, which remain small due to the use of low-rank matrices.
- However, residual memory usage is consistently higher with LoRA than with full fine-tuning. This increase is likely caused by additional operations and activations from LoRA’s adapter layers and should be considered when planning GPU resource allocation.
- The most influential parameters for memory usage are `r` (the rank of the low-rank matrices), `lora_alpha` (scaling factor), and the chosen `target_modules`.
- Memory consumption decreases as more GPUs are used. While optimizer state and gradient memory scale down almost linearly with GPU count (due to data parallelism), residual memory decreases non-linearly, showing diminishing returns beyond 8 GPUs (2 nodes).
- Since residual memory becomes the dominant contributor to total memory usage in LoRA-based fine-tuning, its behaviour must be better understood for efficient scaling.
- LoRA is highly effective for single- and few-GPU setups, but its relative advantage in memory savings narrows in large multi-GPU environments. This

suggests that LoRA is best leveraged when hardware is scarce, and alternative strategies may be more beneficial at larger scales.

- One of the main contributors to high residual memory in this study is the large `block_size` of 32,768 tokens, retained from the original `sl` setup. This choice increases the volume of intermediate activations during training, substantially impacting the residual memory footprint.

4.2 Training Time

- LoRA-based fine-tuning typically requires less training time than full fine-tuning, particularly when the trainable parameter count remains below $\sim 1\%$ of total model size.
- The number of trainable parameters, and thus the training time, is primarily affected by `r`, `lora_alpha`, and the selected `target_modules`.
- While `lora_dropout` does not directly impact training time, it may affect convergence rate and model generalisation.
- The `bias` parameter, when set to "all", can increase training time significantly without obvious short-term performance gains—suggesting overhead from additional gradient updates.

4.3 Performance

- LoRA fine-tuning generally results in slightly higher training and evaluation loss than full fine-tuning under short training runs. However, increased rank (`r`) and carefully tuned learning rates can help LoRA achieve comparable performance, especially over longer epochs.
- The `lora_dropout` and `bias` parameters do not affect parameter count, but can influence convergence stability and final loss. Their effects are subtle and likely task and model-dependent.
- Learning rate is a critical hyperparameter for LoRA performance. Higher learning rates (e.g., 5×10^{-4}) can lead to faster convergence and strong results (as seen with LoRA 18), while excessively high learning rates (e.g., 1×10^{-3}) may cause instability or poor generalisation (as seen with LoRA 19).
- LoRA 18, using `r=64`, `lora_alpha=128`, and learning rate 5×10^{-4} , matched full fine-tuning performance on the `aime24_nofigures` benchmark after 8 epochs, demonstrating the practical viability of LoRA in generalisation-heavy tasks.

- LoRA 19, despite showing promising loss curves during training, failed on benchmark evaluation, highlighting that low training loss does not guarantee real-world task performance without appropriate regularisation or hyperparameter tuning.
- With LoRA, it is feasible to fine-tune a 3B-parameter model on a single 80GB A100 GPU, which is not practical with full fine-tuning—making LoRA a compelling choice for researchers and developers with limited hardware resources.

5 Future Work

Key areas for future research include:

- **Residual memory analysis:** investigate the causes of increased residual memory usage in LoRA-based training, especially under multi-GPU settings, to improve memory efficiency and scalability.
- **Hybrid fine-tuning approaches:** explore combining LoRA with other parameter-efficient techniques to enhance performance while maintaining low memory costs.
- **Scalability with model parallelism:** integrate LoRA with advanced parallelism strategies such as FSDP or DeepSpeed to enable efficient fine-tuning of larger models (7B+).
- **Task and domain generalisation:** evaluate LoRA across diverse tasks and datasets to better understand its strengths and limitations beyond reasoning benchmarks.
- **Optimization strategies:** study the impact of adaptive learning rates, schedulers, and optimizer variants to improve convergence and stability in LoRA training.

6 Acknowledgements

This work was partially supported by **Baskerville**, a national accelerated compute resource, under the EPSRC Grant EP/T022221/1. Use of Baskerville’s high-performance computing infrastructure was essential to the completion of this research.

References

- [1] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [2] 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>.
- [3] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling, 2025. URL <https://arxiv.org/abs/2501.19393>.