You are viewing *main* version, which requires installation from source. If you'd like regular pip install, checkout the latest stable version (v0.25.1).

LoRA Without Regret

Copy page

Recent research from the team at Thinking Machines Lab (Schulman et al., 2025) shows that **LoRA can match full fine-tuning performance** when configured correctly, while using only ~67% of the compute. These findings are exciting to TRL users because they're straightforward to implement and can improve model performance on smaller budgets.

This guide provides simple instructions to reproduce the results of the blog post in TRL.

> It is recommended to read the blog post before following this guide, or to consult both resources in parallel for best results.

Benefits of LoRA over full fine-tuning

First of all, let's remind ourselves of the benefits of LoRA over full fine-tuning.

LoRA adds adapter layers on top of the base model, which contains significantly fewer parameters than the base model itself. This design reduces GPU memory requirements and enables more efficient training. As described in the blog, this approach was originally thought to involve a performance trade-off, although careful configuration can overcome this trade-off and match full fine-tuning performance.

Examples with TRL

Let's implement and train LoRA adapters in TRL scripts based on the core findings of the blog post. Afterwards, we'll revisit each finding in light of the TRL results.

Supervised Fine-Tuning (SFT)

The blog post performs SFT on a range of models and datasets from the Hub, which we can reproduce in TRL.

Model	Dataset
Llama-3.2-1B-Instruct	allenai/tulu-3-sft-mixture
Llama-3.2-1B-Instruct	open-thoughts/OpenThoughts-114k
Llama-3.1-8B-Instruct	allenai/tulu-3-sft-mixture
Llama-3.1-8B-Instruct	open-thoughts/OpenThoughts-114k

python jobs local

We can integrate these findings with the TRL Python API like so:

```python
from datasets import load_dataset
from peft import LoraConfig
from trl import SFTTrainer, SFTConfig

dataset = load_dataset("open-thoughts/OpenThoughts-114k", split="train")

peft_config = LoraConfig(r=256, lora_alpha=16, target_modules="all-linear")

training_args = SFTConfig(
    learning_rate=2e-4,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=4,
    num_train_epochs=1,
    report_to=["trackio"],
)

trainer = SFTTrainer(
    model="Qwen/Qwen2.5-3B-Instruct",
    train_dataset=dataset,
    peft_config=peft_config,
    args=training_args,
)

trainer.train()
```

Once training starts, you can monitor the progress in Trackio, which will log the URL.

Reinforcement Learning (GRPO)

The blog post performs GRPO on a range of models and datasets from the Hub, and once again we can reproduce the results in TRL.

Model	Dataset
Llama-3.1-8B-Base	GSM8k
Llama-3.1-8B-Base	DeepMath-103K
Qwen3-8b-base	DeepMath-103K

For reinforcement learning, the blog uses a math reasoning task that we can reproduce as a Python function.

python jobs local

We can implement these recommendations with the TRL Python API like so:

```
from datasets import load_dataset
from peft import LoraConfig
from trl import GRPOConfig, GRPOTrainer
from trl.rewards import reasoning_accuracy_reward

dataset = load_dataset("HuggingFaceH4/OpenR1-Math-220k-default-verified", split="tra
peft_config = LoraConfig(
    r=1,
    lora_alpha=32,
    target_modules="all-linear"
)

training_args = GRPOConfig(
    learning_rate=5e-5,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=4,
    num_train_epochs=1,
    num_generations=8,
    generation_batch_size=8,
    report_to=["trackio"],
)

trainer = GRPOTrainer(
    model="Qwen/Qwen3-0.6B",
    reward_funcs=reasoning_accuracy_reward,
    args=training_args,
    train_dataset=dataset,
    peft_config=peft_config,
)

trainer.train()
```

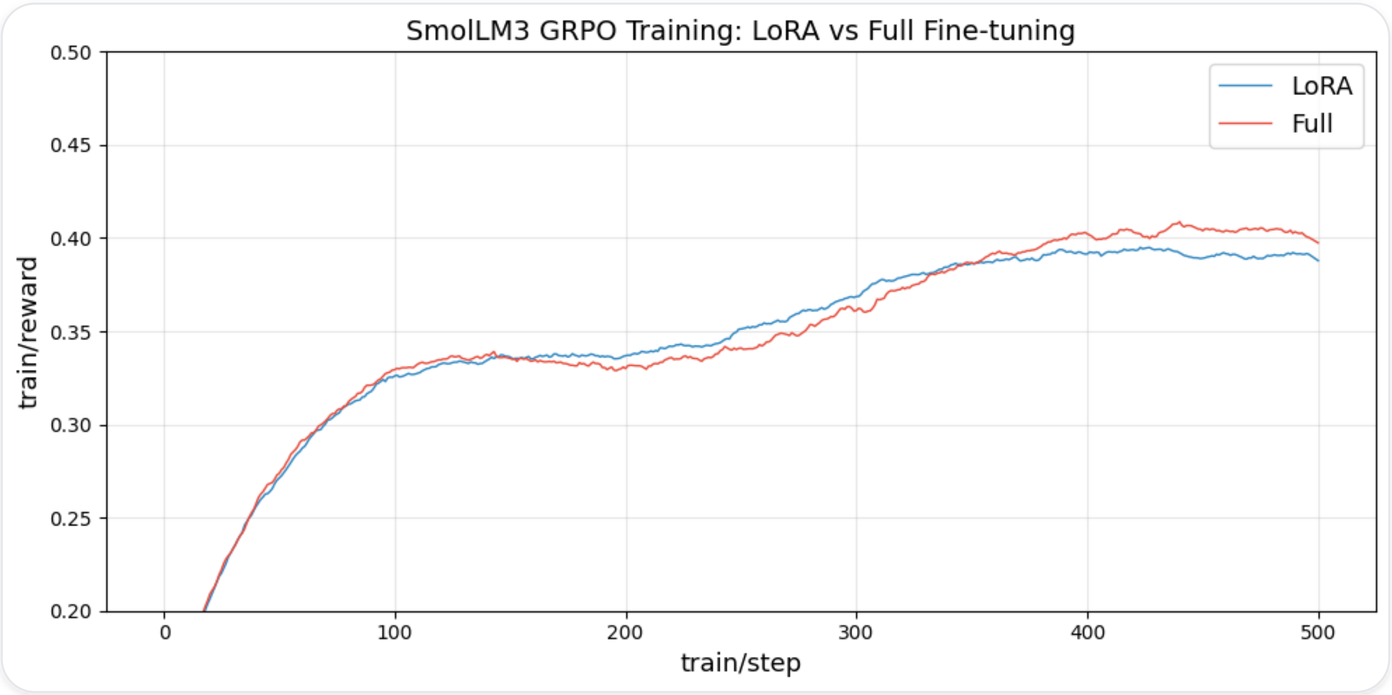
> This snippet skips the reward function which is defined above to keep the example concise.

The reinforcement learning script with GRPO is implemented as a custom script in TRL, which uses the reward function shown above. You can review it at `grpo.py` - Reinforcement learning with LoRA best practices

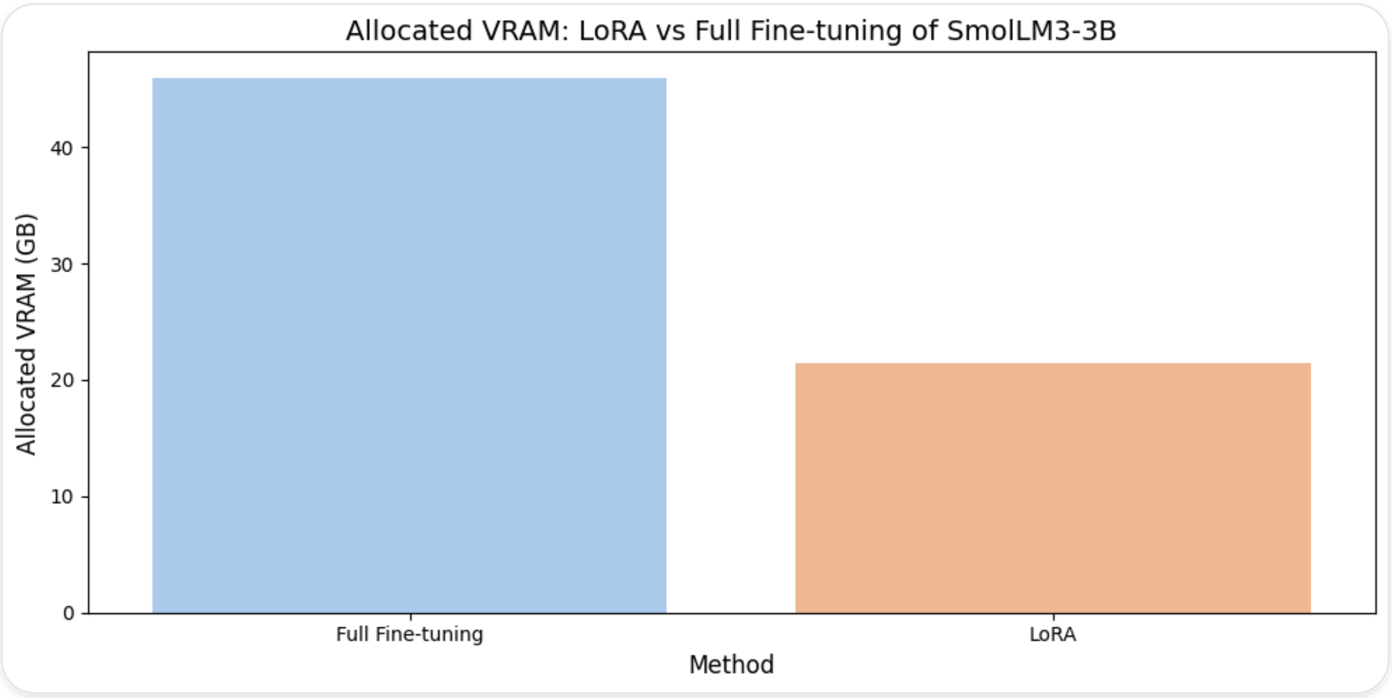
Key findings in optimizing LoRA

The authors recommend applying LoRA to all weight matrices rather than limiting it to attention layers, as increasing the rank does not compensate for this restriction. In TRL, this can be configured using `--lora_target_modules all-linear` to apply LoRA to all weight matrices.

We were able to reproduce the results of the blog post using TRL and the SmolLM3 model. We trained the model for 500 steps on the Math 220k dataset with the reward function and configuration above. As you can see in the figure below, the LoRA model's average train reward curve matches the full fine-tuning curve.



And most importantly, the LoRA model uses significantly less memory than the full fine-tuning model, as we can see in the figure below.



Here are the parameters we used to train the above models

Parameter	LoRA	Full FT
`--model_name_or_path`	HuggingFaceTB/SmolLM3-3B	HuggingFaceTB/SmolLM3-3B
`--dataset_name`	HuggingFaceH4/OpenR1-Math-220k-default-verified	HuggingFaceH4/OpenR1-Math-220k-default-verified
`--learning_rate`	1.0e-5	1.0e-6
`--max_prompt_length`	1024	1024
`--max_completion_length`	4096	4096
`--lora_r`	1	-
`--lora_alpha`	32	-
`--lora_dropout`	0.0	-
`--lora_target_modules`	all-linear	-

Let's break down the key findings of the blog post and how we were able to reproduce them.

1. LoRA performs better when applied to all weight matrices

The authors recommend applying LoRA to all weight matrices rather than limiting it to attention layers, as increasing the rank does not compensate for this restriction.

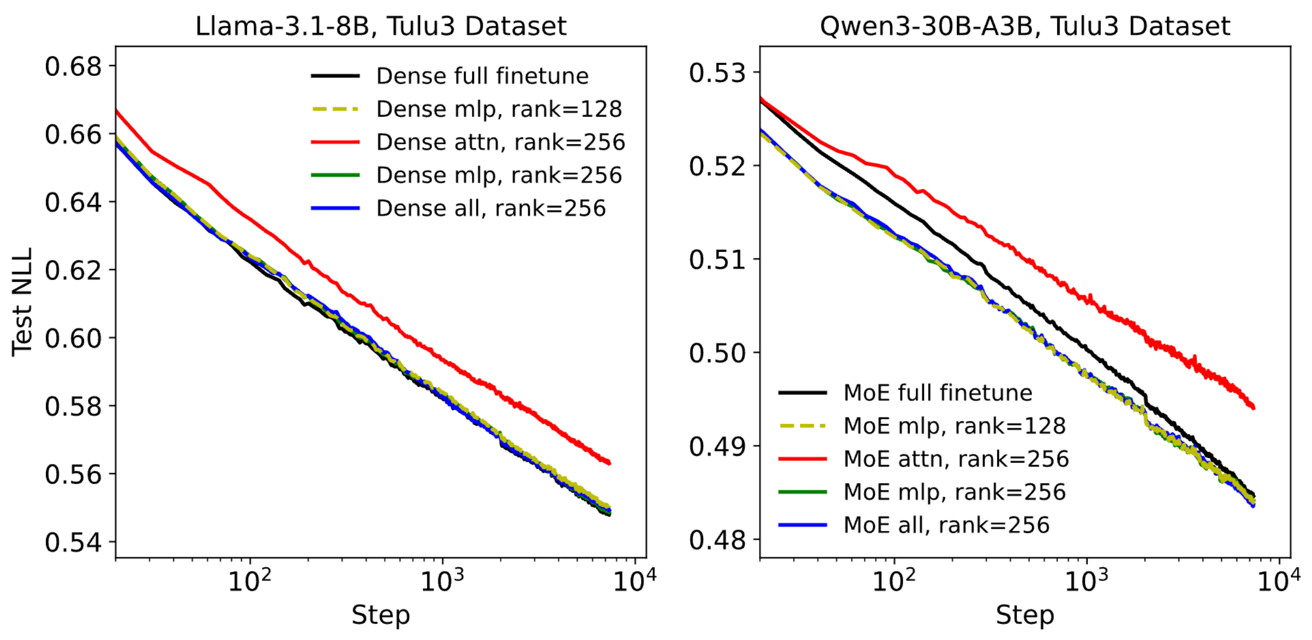


Figure 4: Attention-only LoRA significantly underperforms MLP-only LoRA, and does not further improve performance on top of LoRA-on-MLP. This effect holds for a dense model (Llama-3.1-8B) and a sparse MoE (Qwen3-30B-A3B-Base).

Attention-only LoRA underperforms even when using a higher rank to match parameter count. In TRL, this can be configured using `--lora_target_modules all-linear` to apply LoRA to all weight matrices. In Python, we can do this like so:

```python
from peft import LoraConfig

peft_config = LoraConfig(target_modules="all-linear")
```

2. The adapter needs sufficient capacity to learn from the dataset

The blog post recommends using a sufficient LoRA rank to learn from the dataset. The rank determines the number of trainable parameters in the LoRA adapter. Therefore, “For datasets that exceed LoRA capacity, LoRA underperforms FullFT”.

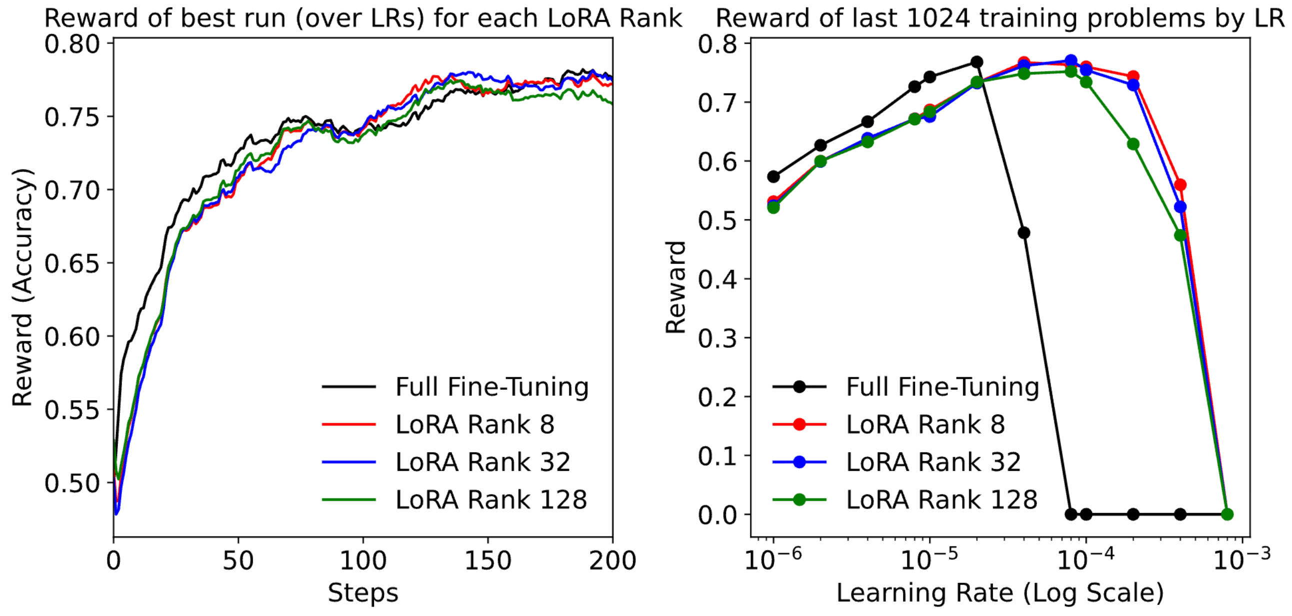


Figure 7: Experiments on the DeepMath dataset with Qwen3-8b-base. In the left plot, we show the learning curve for different ranks and full fine-tuning. For each of these settings, we show the best learning rate, which results in the highest final performance. On the right, we plot learning rate vs final performance. As in our previous math experiments, LoRA seems to have a wider peak of near-optimal learning rates.

In the TRL script, we could use `--lora_r` to set the rank and adapt it based on the task and dataset we’re training on. The blog post recommends the following ranks based on the task and dataset size:

Reinforcement learning tasks typically require lower capacity, so smaller LoRA ranks can be used. This is because policy gradient algorithms extract roughly ~1 bit of information per episode, demanding minimal parameter capacity.

The blog post defines the ideal dataset size for LoRA to match full fine-tuning as “Post-training scale”. Which we can use to determine the recommended rank for SFT and RL LoRAs as:

Task Type	Dataset Size	Recommended Rank
SFT	Post-training scale	256

RL	Any size	1-32

3. “FullFT and high-rank LoRAs have similar learning curves”

Counterintuitively, the blog post recommends using a higher learning rate than for full fine-tuning. In the table above, we used 1.0e-5 for LoRA and 1.0e-6 for full fine-tuning. In the TRL script, we could use `--learning_rate` to set the learning rate. The $\frac{1}{r}$ scaling in LoRA makes the optimal learning rate approximately rank-independent.

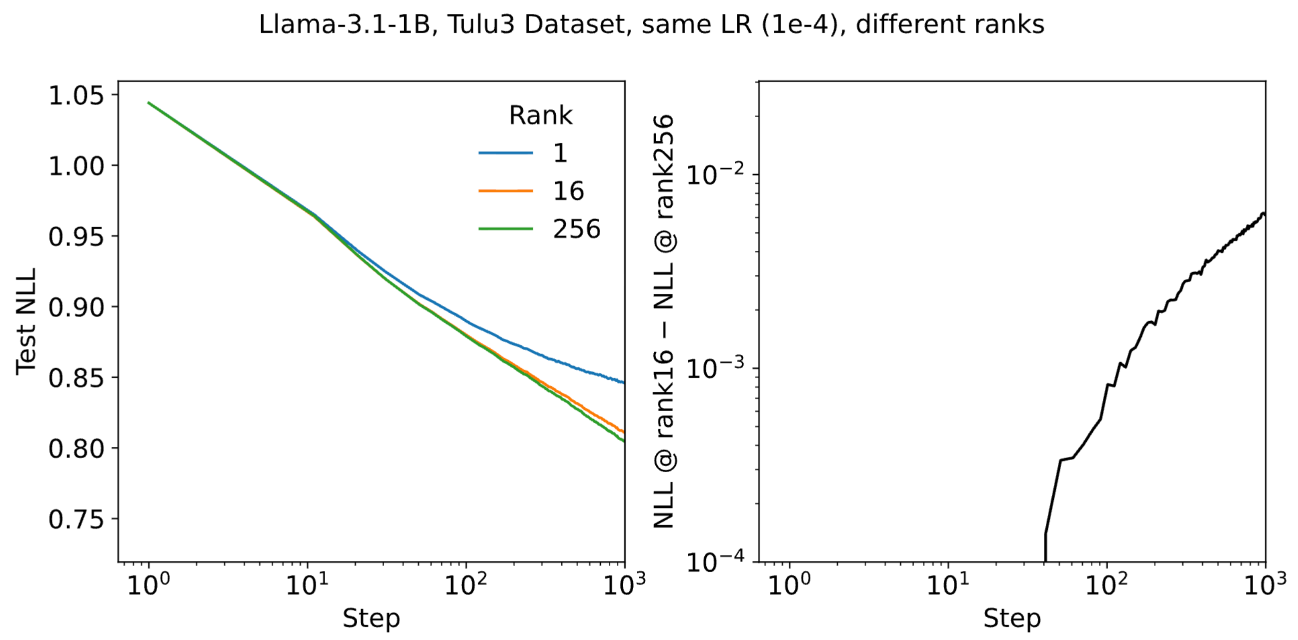


Figure 9: These plots look at the differences in the learning curves, early in training, for different ranks with the same learning rate. On the left, we show the learning curves. The right shows the difference between rank 16 and 256, which grows over time. Strangely, it is negative (though tiny) for the first few steps, so that part of the curve is missing from the plot.

4. “In some scenarios, LoRA is less tolerant of large batch sizes than full fine-tuning.”

The blog post recommends using an effective batch size < 32 because the authors found LoRA to be less tolerant of large batch sizes. This could not be mitigated by increasing the LoRA rank. In the TRL script, we could use `--per_device_train_batch_size` and `--gradient_accumulation_steps` to set the batch size.

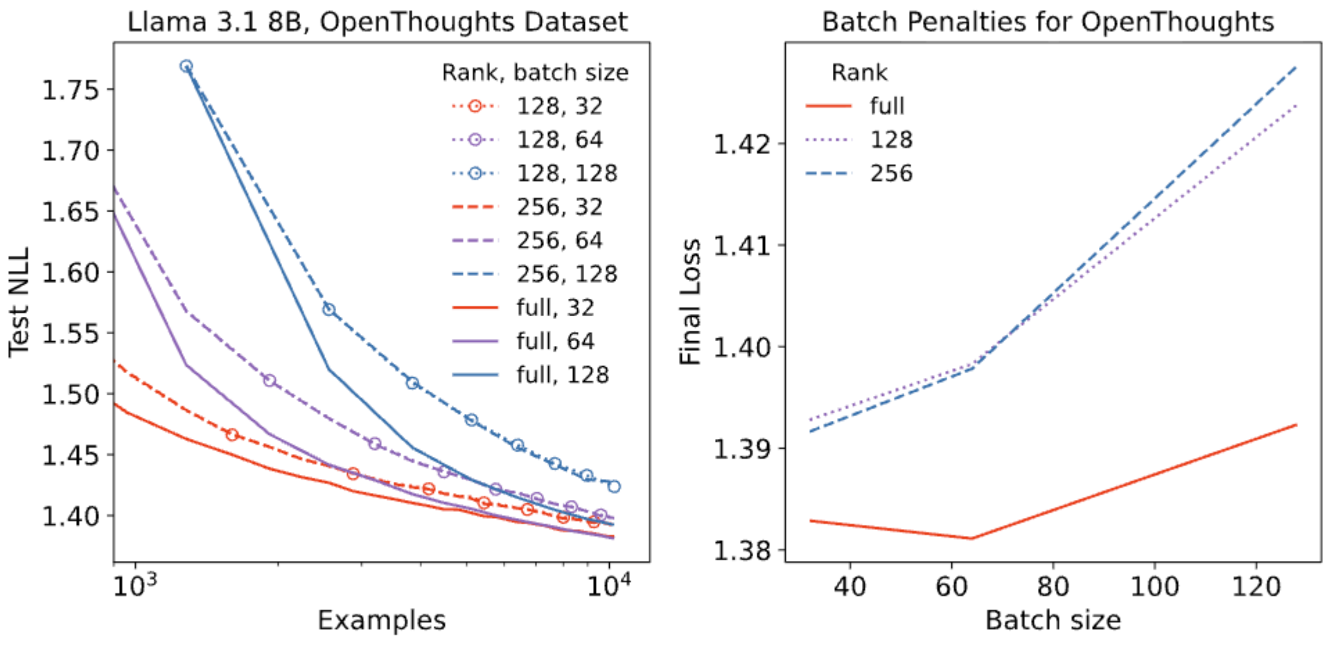


Figure 3: Batch size effects on LoRA vs FullFT performance. Left: Learning curves for different batch sizes show a persistent gap between LoRA (dashed) and FullFT (solid) at large batch sizes. Right: Final loss as a function of batch size shows LoRA pays a larger penalty for increased batch size.

Takeaways

Using TRL, you can efficiently implement LoRA adapters to match full fine-tuning performance, applying the core insights (targeting all weight matrices, choosing the right rank, and managing batch size and learning rate) without the heavy compute cost of FullFT.

Citation

```
@article{schulman2025lora,
    title        = {{LoRA Without Regret}},
    author       = {John Schulman and Thinking Machines Lab},
    year         = 2025,
    journal      = {Thinking Machines Lab: Connectionism},
    doi          = {10.64434/tml.20250929},
    note         = {https://thinkingmachines.ai/blog/lora/}
}
```

</> Update on GitHub

← Community Tutorials

Model Classes →