Fine-Tuning on Mac: LoRA & QLoRA with MLX

February 26, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Install mlx-lm, point it at a HuggingFace model, give it a JSONL file with 200-500 examples, and run mlx_lm.lora --train. QLoRA on a 16GB MacBook fine-tunes an 8B model in about an hour. A 32GB Mac handles 14B. A 64GB Mac handles 32B. The big advantage: unified memory means no VRAM wall. A 32GB M2 Pro can fine-tune models that would crash an RTX 3090 (24GB). The tradeoff is speed -- NVIDIA trains 2-4x faster on models that fit in VRAM. After training, fuse the adapters and export to GGUF for Ollama.

📚 **More on this topic:** Fine-Tuning on Consumer Hardware (NVIDIA) · Best Local LLMs for Mac · Running LLMs on Mac M-Series · VRAM Requirements · Ollama on Mac

We already have a general LoRA/QLoRA guide that covers fine-tuning on NVIDIA GPUs with Unsloth. This is the Mac version. Different framework, different constraints, different advantages.

The short version: Apple's MLX framework lets you fine-tune models on Apple Silicon using LoRA and QLoRA. The unified memory architecture means your entire RAM pool is available for training – no separate VRAM limit. A 32GB MacBook Pro can fine-tune models that would crash a 24GB RTX 3090. The tradeoff is speed. NVIDIA hardware trains 2-4x faster when the model fits in VRAM. But if the model doesn't fit in VRAM, NVIDIA can't train it at all without multi-GPU setups. That's where Mac wins.

Why Mac works for fine-tuning

On a PC, fine-tuning lives or dies by VRAM. An RTX 3090 has 24GB. A 7B model at full precision needs ~28GB for training – it doesn't fit. QLoRA drops that to ~7GB, making it possible, but you're always fighting the VRAM ceiling.

On Mac, there's no separate VRAM. The GPU shares the same memory pool as the CPU. A MacBook Pro with 32GB of unified memory can use all 32GB (minus what macOS needs) for training. No copying data between CPU and GPU memory. No out-of-memory crashes because the GPU buffer ran out while system RAM sat idle.

This matters most for larger models. A 14B model needs ~14-18GB for LoRA training. That's tight on a 24GB GPU but comfortable on a 32GB Mac. A 32B model at QLoRA needs ~20-25GB. Impossible on any single consumer GPU. Straightforward on a 48GB Mac.

The catch: Apple Silicon's memory bandwidth is lower than dedicated GPUs. An M3 Max pushes 400 GB/s. An RTX 4090 pushes 1,008 GB/s. Training is a bandwidth-heavy operation, so the Mac is slower per step. You get there – it just takes longer.

What you need

Hardware

Any Apple Silicon Mac works. The question is how large a model you can fine-tune:

Unified memory	LoRA (full precision base)	QLoRA (4-bit base)	Training speed
8 GB	1B-3B (tight)	3B	Slow, limited batch size
16 GB	3B-7B	7B-8B	Workable
24 GB	7B-8B	8B-14B	Comfortable
32 GB	8B-14B	14B-32B (tight)	Good
48 GB	14B-32B	32B	Good
64 GB	32B	32B-70B (tight)	Fast
96 GB	70B (tight)	70B	Fast
128 GB	70B	70B+ comfortably	Fast

For comparison, here's what the same tasks require on NVIDIA:

Mac (unified)	Equivalent NVIDIA setup
16 GB Mac (QLoRA 8B)	RTX 3060 12GB or RTX 4060 Ti 16GB
32 GB Mac (QLoRA 14B)	RTX 3090 24GB (tight)
48 GB Mac (QLoRA 32B)	No single consumer GPU – need dual GPUs
64 GB Mac (LoRA 32B)	RTX A6000 48GB ($4,000+)
96 GB Mac (LoRA 70B)	Multi-GPU setup ($3,000+)

The Mac doesn't train faster. It trains models that don't fit elsewhere.

Software

Python 3.10+ and pip. That's it.

```
pip install "mlx-lm[train]"
```

This installs `mlx-lm` with training dependencies, including the `mlx_lm.lora` command. Verify it works:

```
python -c "import mlx; print(mlx.__version__)"
```

You should see version 0.30+ (current is 0.30.6 as of early 2026).

Choosing a base model

MLX fine-tuning requires HuggingFace-format models. GGUF files won't work – you need the original safetensors weights.

Supported architectures: Llama, Mistral, Qwen2, Phi, Gemma, Mixtral, OLMo, MiniCPM, InternLM2.

Model	Size	HuggingFace ID	Best for
Llama 3.1 8B Instruct	8B	`meta-llama/Llama-3.1-8B-Instruct`	General fine-tuning, good baseline
Qwen 2.5 7B Instruct	7B	`Qwen/Qwen2.5-7B-Instruct`	Multilingual, strong coding
Mistral 7B Instruct v0.3	7B	`mistralai/Mistral-7B-Instruct-v0.3`	Instruction following
Phi-4 Mini	3.8B	`microsoft/phi-4-mini-instruct`	Reasoning on low memory
Qwen 2.5 14B Instruct	14B	`Qwen/Qwen2.5-14B-Instruct`	Best quality for 32GB Macs
	70B		96GB+ Macs only

Model	Size	HuggingFace ID	Best for
Llama 3.3 70B Instruct		`meta-llama/Llama-3.3-70B-Instruct`	

Start with an 8B model. It's the sweet spot for learning the workflow and iterating fast. Move to 14B or 32B once you've nailed your dataset and training config.

For quantized models (QLoRA), use MLX-format quantized versions from the `mlx-community` org on HuggingFace, or quantize yourself:

```
mlx_lm.convert --hf-path meta-llama/Llama-3.1-8B-Instruct -q
```

This creates a 4-bit quantized version in MLX format. Point `mlx_lm.lora` at the quantized model and it automatically uses QLoRA.

Preparing training data

mlx-lm accepts three JSONL formats. Pick the one that matches your task.

Chat format (recommended for instruction tuning)

```
{"messages": [{"role": "system", "content": "You are a helpful legal assistant."}, {"role": "use
```

Completions format (for prompt-response pairs)

```
{"prompt": "Summarize this contract clause:", "completion": "This clause establishes..."}
```

Text format (for continued pretraining)

```
{"text": "The Treaty of Westphalia (1648) established the principle of..."}
```

Put your data in a folder with these files:

- `train.jsonl` (required)
- `valid.jsonl` (optional but recommended – 10-15% of your data)
- `test.jsonl` (optional – for evaluation after training)

Each example must be a single line. No multi-line JSON.

How much data you need

Less than you think:

Examples	What to expect
50-100	Enough to shift tone and format. Won't learn new knowledge.
200-500	The practical sweet spot. Enough to specialize behavior on a well-defined task.
500-1,000	Good for complex tasks with varied inputs. Diminishing returns beyond this for most LoRA fine-tunes.
1,000+	Only if your task has high variability. More data doesn't always help – 200 clean examples beat 2,000 sloppy ones.

Quality matters more than quantity. Every example should represent exactly the behavior you want. If you're fine-tuning a model to write SQL from natural language, every example should be a clean natural-language-to-SQL pair. No filler, no duplicates, no contradictory examples.

Running LoRA fine-tuning

The basic command:

```
mlx_lm.lora \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --train \
  --data ./my-dataset \
  --iters 500
```

This downloads the model from HuggingFace (first run only), trains LoRA adapters for 500 iterations, and saves them to `./adapters/`.

Key parameters

Parameter	Default	What it does
`--iters`	1000	Training iterations. 300-600 is usually enough for 200-500 examples.
`--batch-size`	4	Reduce to 2 or 1 if you hit memory pressure.
`--num-layers`	16	Number of layers to apply LoRA to. Reduce to 8 or 4 to save memory.
`--learning-rate`	1e-5	Default works for most cases.
`--fine-tune-type`	lora	Options: `lora`, `dora`, `full`. Stick with `lora` unless you know why you need the others.
`--grad-checkpoint`	off	Enable with `--grad-checkpoint` to trade speed for memory.
`--mask-prompt`	off	Enable to compute loss only on the assistant's response, not the prompt. Recommended for chat format.

A realistic example

Fine-tuning Llama 3.1 8B on a 32GB Mac to write SQL:

```
mlx_lm.lora \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --train \
  --data ./sql-dataset \
  --iters 500 \
  --batch-size 2 \
  --num-layers 16 \
  --mask-prompt
```

During training you'll see output like:

```
Iter 10: Train loss 2.431, It/sec 0.89, Tokens/sec 285
Iter 20: Train loss 1.872, It/sec 0.91, Tokens/sec 291
Iter 100: Train loss 0.843, It/sec 0.92, Tokens/sec 294
...
Iter 500: Train loss 0.312, It/sec 0.93, Tokens/sec 296
```

Loss should drop steeply in the first 100 iterations and flatten by 300-500. If it's still dropping at 500, increase `--iters`. If it flatlines at a high value, your data may have issues.

Training speed depends on your chip. Rough numbers for LoRA on an 8B model:

Chip	Tokens/sec (training)
M1 Max 32GB	~250
M2 Pro 16GB	~200
M2 Ultra 192GB	~475
M3 Max 48GB	~320
M4 Max 128GB	~380

A 500-iteration run on 300 examples takes roughly 15-45 minutes depending on your hardware and sequence length.

QLoRA: when memory is tight

QLoRA quantizes the base model to 4-bit and trains LoRA adapters on top. The base model weights stay frozen at 4-bit. Only the small LoRA adapter trains at full precision.

The memory savings are dramatic:

Model	LoRA memory	QLoRA memory	Savings
Llama 3.1 8B	~14 GB	~7 GB	50%
Qwen 2.5 14B	~24 GB	~12 GB	50%
Llama 3.3 70B	~130 GB	~45 GB	65%

To use QLoRA, point `mlx_lm.lora` at a quantized model. If the model is already quantized, it uses QLoRA automatically:

```
# First, quantize the model (one-time)
mlx_lm.convert \
  --hf-path meta-llama/Llama-3.1-8B-Instruct \
  -q \
  --q-bits 4

# Then train with QLoRA (automatic when model is quantized)
mlx_lm.lora \
  --model mlx_model \
```

```
  --train \
  --data ./my-dataset \
  --iters 500
```

Or use pre-quantized models from `mlx-community` on HuggingFace:

```
mlx_lm.lora \
  --model mlx-community/Llama-3.1-8B-Instruct-4bit \
  --train \
  --data ./my-dataset \
  --iters 500
```

When to use QLoRA vs LoRA

- **16GB Mac + 8B model:** QLoRA (LoRA won't fit)
- **32GB Mac + 8B model:** LoRA (better quality, you have the memory)
- **32GB Mac + 14B model:** QLoRA (LoRA is too tight)
- **64GB Mac + 32B model:** Either works. QLoRA for speed, LoRA for quality.

QLoRA produces slightly lower quality than full-precision LoRA because the base weights are approximated at 4-bit. For most practical tasks the difference is small. If you're on the edge of fitting, use QLoRA and don't worry about it.

Memory-saving tricks

If you're still running out of memory:

1. **Reduce batch size:** `--batch-size 1`
2. **Fewer LoRA layers:** `--num-layers 8` or `--num-layers 4`
3. **Enable gradient checkpointing:** `--grad-checkpoint`
4. **Close other apps.** Safari with 30 tabs eats 4-8GB. Close everything non-essential.
5. **Watch Activity Monitor.** Green memory pressure means you're fine. Yellow means tight. Red means swap – your training will crawl.

Testing your fine-tuned model

After training completes, test with the adapters loaded:

```
mlx_lm.generate \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --adapter-path ./adapters \
  --prompt "Write a SQL query to find the top 5 customers by total spend"
```

Compare the output with and without the adapter to see what changed. Run without `--adapter-path` to see the base model's response.

For quantitative evaluation, use the test set:

```
mlx_lm.lora \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --adapter-path ./adapters \
  --data ./my-dataset \
  --test
```

This reports test loss. Lower is better, but the real test is whether the model's output matches what you want. Read the actual generations, not just the loss number.

Deploying with Ollama

Your fine-tuned adapters are only useful inside mlx-lm until you export them. To use the model with Ollama, you need to fuse the adapters into the base model and convert to GGUF.

Step 1: Fuse adapters

```
mlx_lm.fuse \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --adapter-path ./adapters \
  --save-path ./fused-model \
  --de-quantize
```

The `--de-quantize` flag is important if you trained with QLoRA. It converts the fused model back to fp16, which is needed for clean GGUF conversion.

Step 2: Export to GGUF

```
mlx_lm.fuse \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --adapter-path ./adapters \
  --export-gguf
```

This produces a GGUF file (fp16) in the output directory. Note: GGUF export currently works with Llama, Mistral, and Mixtral architectures. For other architectures, use llama.cpp's `convert_hf_to_gguf.py` on the fused safetensors model.

Step 3: Create an Ollama model

Write a `Modelfile`:

```
FROM ./fused-model/ggml-model-f16.gguf

TEMPLATE """{{ if .System }}<|start_header_id|>system<|end_header_id|>
{{ .System }}<|eot_id|>{{ end }}<|start_header_id|>user<|end_header_id|>
{{ .Prompt }}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
{{ .Response }}<|eot_id|>"""

PARAMETER stop "<|eot_id|>"
PARAMETER temperature 0.7
```

Then create and run:

```
ollama create my-sql-model -f Modelfile
ollama run my-sql-model
```

Your fine-tuned model is now available through Ollama's API, accessible by any tool that speaks the Ollama or OpenAI-compatible protocol.

Alternative: quantize the GGUF

The fp16 GGUF is large (~16GB for an 8B model). For daily use, quantize it:

```
# Using llama.cpp's quantize tool
./quantize fused-model/ggml-model-f16.gguf fused-model/model-q4_k_m.gguf q4_k_m
```

Then point your Modelfile at the quantized version.

Mac vs NVIDIA: honest comparison

	Mac (MLX)	NVIDIA (CUDA)
Max model for LoRA (single device)	70B on 128GB Mac	70B only with multi-GPU ($5,000+)
Training speed (8B LoRA)	~250-400 tok/s	~800-1,500 tok/s
Memory efficiency	Unified – all RAM available	VRAM-limited (24GB max consumer)
Ecosystem	mlx-lm only	Unsloth, Axolotl, HuggingFace, dozens of tools
Multi-GPU training	Not supported	Standard with FSDP, DeepSpeed
Setup complexity	pip install, done	CUDA, cuDNN, driver versions, venv management
Cost for 8B fine-tuning	$0 (use existing Mac)	$700-900 (used RTX 3090) + PC

Where Mac is genuinely better

- **Models that don't fit in 24GB VRAM.** Fine-tuning a 14B model with full LoRA needs ~24GB. That's the absolute ceiling for an RTX 3090 and it'll likely OOM with any real batch size. A 32GB Mac handles it without drama.
- **No driver/CUDA hassle.** Install mlx-lm, run the command. No CUDA toolkit version conflicts, no cuDNN mismatches, no "which PyTorch build do I need" headaches.
- **Silent training.** A multi-hour training run on Mac produces zero fan noise. An RTX 3090 under training load sounds like a desk fan on high.
- **You already own the hardware.** If you have a 32GB+ Mac, fine-tuning is free to try. No GPU purchase, no separate PC.

Where Mac falls short

- **Raw speed.** An RTX 3090 trains 2-4x faster than an M2 Max on the same model. For production workflows where you're iterating on datasets and hyperparameters, this adds up.
- **Ecosystem.** Unsloth (2-5x speedup on NVIDIA) doesn't work on Mac. Axolotl, a popular fine-tuning framework, is CUDA-only. HuggingFace Trainer works on Mac via MPS but is slower and less tested than CUDA.
- **Multi-GPU scaling.** If you need to train 70B models fast, NVIDIA scales to 2, 4, 8 GPUs. MLX has no multi-device training support.
- **Community.** Most fine-tuning tutorials, guides, and debugging resources assume CUDA. When something goes wrong on MLX, you're searching through GitHub issues.

The practical verdict

For most people reading this article, the answer is straightforward: you own a Mac with 32GB+, you want to fine-tune a 7B-14B model for a specific task, and MLX is the path of least resistance. Install one package, run one command, get a fine-tuned model. No hardware purchase needed.

The calculus changes if fine-tuning is a core part of your workflow and you're iterating on datasets daily. The 2-4x speed advantage of NVIDIA hardware adds up across dozens of training runs, and the broader ecosystem (Unsloth, Axolotl) saves real time. But if the model you want to fine-tune doesn't fit on any single consumer GPU, Mac is your only option short of cloud rentals.

Troubleshooting

Out-of-memory / system becomes unresponsive

Your model is too large for available memory. Fixes in order:

1. Switch to QLoRA (4-bit base model)
2. `--batch-size 1`
3. `--num-layers 4`
4. `--grad-checkpoint`
5. Close all other apps
6. Use a smaller model

“Model type not supported”

MLX LoRA supports: Llama, Mistral, Qwen2, Phi, Gemma, Mixtral, OLMo, MiniCPM, InternLM2. If your model isn’t one of these architectures, it won’t work. Check the model card on HuggingFace for the architecture.

Training loss doesn’t decrease

- Check your data format. Each line must be valid JSON with the correct keys.
- Make sure examples are consistent. Contradictory examples confuse training.
- Try increasing `--learning-rate` slightly (e.g., 3e-5 instead of 1e-5).
- If loss is extremely high from the start (>10), the model may not match your data format. Try a different chat template.

“You must use HuggingFace format models”

GGUF files don’t work with mlx-lm training. You need the original safetensors weights from HuggingFace. Download with:

```
mlx_lm.convert --hf-path <model-name>
```

Or let `mlx_lm.lora` download automatically by passing the HuggingFace model ID.

GGUF export fails for non-Llama models

The built-in `--export-gguf` flag in mlx_lm.fuse only supports Llama, Mistral, and Mixtral. For Qwen, Phi, or Gemma:

1. Fuse the model: `mlx_lm.fuse --model <model> --save-path ./fused`
2. Convert using llama.cpp: `python convert_hf_to_gguf.py ./fused --outtype f16`
3. Quantize: `./quantize fused.gguf fused-q4.gguf q4_k_m`

Training is extremely slow

- Check Activity Monitor for memory pressure. Yellow/red means the system is swapping, which destroys training speed.
- Make sure examples are under 3,500 tokens each. Very long sequences slow training and eat memory.

- `--grad-checkpoint` saves memory but adds ~30% training time. Only use it if you need to.

Bottom line

MLX fine-tuning on Mac is the simplest path to a custom model if you already own the hardware. The entire workflow is four commands: install, train, fuse, deploy. No GPU purchase, no CUDA setup, no driver conflicts.

The practical setup: a 32GB Mac, 200-500 clean examples in JSONL, QLoRA on an 8B model, 300-500 iterations. You'll have a fine-tuned model running in Ollama within a couple of hours.

It's not as fast as NVIDIA. The ecosystem isn't as mature. But for a first fine-tune, or for models that don't fit in GPU VRAM, it's the easiest way to get there.

Get notified when we publish new guides.

Subscribe — free, no spam
