LoRA Training on Consumer Hardware: Fine-Tune Models With 12GB VRAM

February 11, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** A 12GB GPU can QLoRA-train a 7B model in 2-4 hours. Unsloth handles the optimization automatically — 2x faster, 60% less VRAM than standard training. Use rank 16, learning rate 2e-4, batch size 2 with gradient accumulation 8. You need 200-500 quality examples for most tasks. After training, merge the adapter and export to GGUF to run in Ollama. Total adapter size: 50-200MB. A used RTX 3090 at $700 opens up 13-14B models and cuts training time in half.

📚 **Background reading:** LoRA and QLoRA Explained · What Can You Run on 12GB VRAM · Quantization Explained

The LoRA/QLoRA guide covers what fine-tuning is and when to use it. This article is the hands-on recipe: exact configs, VRAM limits, working code, and the full pipeline from training to running your model in Ollama. Everything tested against 12GB and 24GB consumer GPUs.

If you have an RTX 3060 12GB or better, you can fine-tune a 7B model this afternoon.

What Fits on 12GB

Not everything fits. The VRAM math depends on model size, quantization, LoRA rank, batch size, and sequence length. Here's what actually works.

QLoRA (4-bit) VRAM Usage

Model	Rank 16, Batch 1	Rank 16, Batch 2	Rank 32, Batch 2	Minimum GPU
3B	~3.5 GB	~4.5 GB	~5 GB	6 GB
7B	~6-7 GB	~8-9 GB	~9-10 GB	8 GB (tight) / 12 GB
8B (Llama 3.1)	~7-8 GB	~9-10 GB	~10-11 GB	12 GB
13B	~10-11 GB	~13-14 GB	~15 GB	16 GB / 24 GB
14B (Qwen 2.5)	~8.5-10 GB	~12-13 GB	~14 GB	12 GB (tight) / 16 GB

Model	Rank 16, Batch 1	Rank 16, Batch 2	Rank 32, Batch 2	Minimum GPU
32B	~20-22 GB	~26-28 GB	~30 GB	24 GB (tight)

All numbers assume gradient checkpointing enabled and Unsloth's memory optimizations. Without Unsloth, add 30-60% more VRAM.

What This Means for 12GB

- **7B/8B models:** Comfortable. Batch size 2, rank 16, 2048 sequence length. Room to spare.
- **14B models:** Tight but possible with batch size 1, rank 8, shorter sequences (1024). Unsloth's gradient checkpointing is mandatory.
- **13B+ dense models:** Won't fit at useful settings. Need 16GB+.

Standard LoRA (16-bit) for Comparison

Model	Rank 16, Batch 1	Minimum GPU
3B	~8 GB	8 GB
7B	~15-19 GB	24 GB
13B	~28-33 GB	2x 24 GB

Standard LoRA at 16-bit is 3-4x the VRAM of QLoRA. On consumer hardware, QLoRA is the only practical option for 7B+ models.

LoRA Rank Impact

Higher rank = more trainable parameters = better capacity to learn = more VRAM. The tradeoff matters on 12GB.

Rank	Trainable Params (7B)	VRAM Overhead	When to Use
8	~6.5M	+0.5 GB	Simple style/format tasks
16	~13M	+0.5-1 GB	General fine-tuning (start here)
32	~26M	+1-1.5 GB	Domain adaptation
64	~52M	+1.5-2.5 GB	Complex tasks, code, math
128	~104M	+3-4 GB	Near full fine-tuning quality

On 12GB, stay at rank 16-32 for 7B models. Rank 64+ pushes you into OOM territory with reasonable batch sizes.

The Training Stack

Three tools worth considering. Pick one.

	Unsloth	Axolotl	torchtune
Best for	Single GPU, max speed	Multi-GPU, complex setups	PyTorch-native minimalism
Speed	2-2.5x faster	Standard	Standard
VRAM savings	60-80% less	Standard	Standard
Config style	Python API	YAML file	YAML + recipes
Multi-GPU	Pro only (free = single GPU)	Yes, open source	Yes
Learning curve	Low (notebook-friendly)	Medium	Medium
RL/DPO support	Yes	Yes (DPO, GRPO, KTO, ORPO)	Yes
Vision models	Yes	Yes (+ audio)	Limited
Install	`pip install unsloth`	`pip install axolotl`	`pip install torchtune`

The recommendation: Unsloth for your first fine-tune and for any single-GPU setup. It's faster, uses less memory, and the Python API is easier to debug than YAML configs. Switch to Axolotl when you need multi-GPU training or features Unsloth doesn't support (audio models, some advanced RL methods).

Recipe 1: Fine-Tune a 7B Model (RTX 3060 12GB)

This trains Mistral 7B on a custom dataset using QLoRA + Unsloth. Runs on any 12GB GPU.

What you need:

- GPU with 12GB+ VRAM

- 32GB+ system RAM
- Python 3.10+, CUDA 12.x
- Your dataset in JSON (Alpaca or ChatML format)

Install

```
pip install unsloth
```

Full Training Script

```
from unsloth import FastLanguageModel
from transformers import TrainingArguments
from trl import SFTTrainer
from datasets import load_dataset

# ── Load model in 4-bit ──────────────────────────────
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/mistral-7b-instruct-v0.3-bnb-4bit",
    max_seq_length=2048,
    dtype=None,
    load_in_4bit=True,
)

# ── Attach LoRA adapters ─────────────────────────────
model = FastLanguageModel.get_peft_model(
    model,
    r=16,
    target_modules=[
        "q_proj", "k_proj", "v_proj", "o_proj",
        "gate_proj", "up_proj", "down_proj",
    ],
    lora_alpha=16,
    lora_dropout=0,
    bias="none",
    use_gradient_checkpointing="unsloth",
)

# ── Load dataset ─────────────────────────────────────
# Replace with your dataset path
dataset = load_dataset("json", data_files="my_data.json", split="train")

# Format for training (Alpaca style)
```

```
alpaca_prompt = """### Instruction:
{instruction}

### Input:
{input}

### Response:
{output}"""

def format_examples(examples):
    texts = []
    for inst, inp, out in zip(
        examples["instruction"], examples["input"], examples["output"]
    ):
        texts.append(alpaca_prompt.format(
            instruction=inst, input=inp or "", output=out
        ))
    return {"text": texts}

dataset = dataset.map(format_examples, batched=True)

# ── Train ──────────────────────────────────────────────────
trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=2048,
    packing=False,
    args=TrainingArguments(
        output_dir="./output-7b",
        per_device_train_batch_size=2,
        gradient_accumulation_steps=8,
        num_train_epochs=1,
        learning_rate=2e-4,
        fp16=True,
        logging_steps=10,
        save_steps=200,
        save_total_limit=2,
        warmup_steps=10,
        weight_decay=0.01,
        lr_scheduler_type="linear",
        optim="adamw_8bit",
        report_to="none",
    ),
)
trainer.train()

# ── Save adapter ──────────────────────────────────────────
```

```
model.save_pretrained("my-7b-adapter")
tokenizer.save_pretrained("my-7b-adapter")
```

Expected Results (RTX 3060 12GB)

Metric	Value
VRAM used	~8-9 GB
Training speed	~500-600 tokens/sec
Time (500 examples, 1 epoch)	~30-60 min
Time (5,000 examples, 1 epoch)	~2-4 hours
Adapter size on disk	~50-200 MB

The remaining 3-4 GB of VRAM headroom matters. It's what lets you run batch size 2 instead of 1, which improves training stability.

Recipe 2: Fine-Tune a 14B Model (RTX 3090 24GB)

With 24GB, you can train Qwen 2.5 14B — a much more capable base. Same Unsloth approach, larger model.

```
from unsloth import FastLanguageModel
from transformers import TrainingArguments
from trl import SFTTrainer
from datasets import load_dataset

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/Qwen2.5-14B-Instruct-bnb-4bit",
    max_seq_length=2048,
    dtype=None,
    load_in_4bit=True,
)

model = FastLanguageModel.get_peft_model(
    model,
    r=32,                # Higher rank for larger model
    target_modules=[
        "q_proj", "k_proj", "v_proj", "o_proj",
```

```
        "gate_proj", "up_proj", "down_proj",
    ],
    lora_alpha=32,
    lora_dropout=0,
    bias="none",
    use_gradient_checkpointing="unsloth",
)

dataset = load_dataset("json", data_files="my_data.json", split="train")
# ... same formatting as Recipe 1 ...

trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=2048,
    packing=False,
    args=TrainingArguments(
        output_dir="./output-14b",
        per_device_train_batch_size=2,
        gradient_accumulation_steps=8,
        num_train_epochs=1,
        learning_rate=2e-4,
        fp16=True,
        logging_steps=10,
        save_steps=200,
        warmup_steps=10,
        optim="adamw_8bit",
        report_to="none",
    ),
)
trainer.train()
model.save_pretrained("my-14b-adapter")
tokenizer.save_pretrained("my-14b-adapter")
```

Expected Results (RTX 3090 24GB)

Metric	Value
VRAM used	~14-16 GB
Training speed	~300-400 tokens/sec
Time (5,000 examples, 1 epoch)	~3-5 hours
Adapter size	~100-400 MB

Axolotl Alternative (YAML Config)

If you prefer config files over Python scripts, here's the Axolotl equivalent:

```
base_model: Qwen/Qwen2.5-14B-Instruct
model_type: AutoModelForCausalLM

load_in_4bit: true
adapter: qlora
lora_r: 32
lora_alpha: 32
lora_dropout: 0
lora_target_modules:
  - q_proj
  - k_proj
  - v_proj
  - o_proj
  - gate_proj
  - up_proj
  - down_proj

datasets:
  - path: my_data.json
    type: alpaca

sequence_len: 2048
micro_batch_size: 2
gradient_accumulation_steps: 8
num_epochs: 1
learning_rate: 2e-4
optimizer: paged_adamw_8bit
lr_scheduler: linear
warmup_steps: 10
gradient_checkpointing: true
bf16: true
output_dir: ./output-14b-axolotl
```

```
axolotl train config.yml
```

Same result, different workflow. Axolotl configs are easier to version-control and share with teammates.

Dataset Prep in 30 Minutes

The LoRA guide covers dataset theory in depth. Here's the shortcut.

Minimum Viable Dataset

Dataset Size	What to Expect
< 100 rows	Marginal. Use an instruct model as base.
200-500 rows	Good for style/format tasks. Start here.
1,000-5,000 rows	Strong results on most tasks.
10,000+ rows	Diminishing returns unless your task is complex.

Quality beats quantity every time. 500 carefully reviewed examples outperform 10,000 sloppy ones.

ChatML Format (Recommended for 2026 Models)

Most modern models (Qwen 2.5, Mistral, Llama 3.x) use ChatML. Structure your data like this:

```
[
  {
    "messages": [
      {"role": "system", "content": "You are a legal document summarizer."},
      {"role": "user", "content": "Summarize this contract clause: ..."},
      {"role": "assistant", "content": "This clause establishes that..."}
    ]
  },
  {
    "messages": [
      {"role": "system", "content": "You are a legal document summarizer."},
      {"role": "user", "content": "Summarize this contract clause: ..."},
      {"role": "assistant", "content": "The parties agree to..."}
    ]
  }
]
```

Quick Dataset Creation

1. Collect 50-100 real input/output pairs from your actual use case

2. Review each one manually — fix errors, standardize formatting
3. If you need more volume, use a larger model (GPT-4, Claude) to generate additional examples in the same format, then review and fix those too
4. Split 90/10 into train/validation

The red flag: if every example was generated by an LLM without human review, you're training your model to imitate that LLM's quirks. Always review.

Hyperparameters That Matter

Start with these values. Only change them if something goes wrong.

Parameter	Starting Value	Adjust When
LoRA rank (r)	16	Underfitting → increase to 32-64. VRAM pressure → decrease to 8.
LoRA alpha	Same as rank	Keep alpha = rank for a scaling factor of 1.
Learning rate	2e-4	Loss spikes → halve it. Loss plateaus → double it.
Batch size	2	OOM → reduce to 1.
Gradient accumulation	8	Keeps effective batch at 16 regardless of per-device batch.
Epochs	1-3	Val loss diverging → stop earlier. Loss still dropping → add an epoch.
Dropout	0	Overfitting → try 0.05. Research suggests limited benefit for short runs.
Warmup steps	5-10% of total	Prevents early instability.
Sequence length	2048	OOM → reduce to 1024. Need longer context → increase but check VRAM.

DoRA: The Newer Alternative

DoRA (Weight-Decomposed Low-Rank Adaptation) splits weight updates into magnitude and direction components. On LLaMA-7B, it improved commonsense reasoning accuracy by 3.7% over standard LoRA. Available in both Unsloth and PEFT:

```
model = FastLanguageModel.get_peft_model(
    model,
    r=16,
    use_dora=True,  # Enable DoRA
    # ... rest of config same as before
)
```

The VRAM cost is slightly higher than standard LoRA (~5-10% more). On 12GB that's enough to matter — test whether you can still fit your config before committing to a full training run.

RSLoRA for High Ranks

If you're using rank 64+, enable RSLoRA (Rank-Stabilized LoRA) for better training stability:

```
model = FastLanguageModel.get_peft_model(
    model,
    r=64,
    use_rslora=True,
    # ...
)
```

From LoRA to Ollama

Training produces a small adapter (50-200MB). To run it in Ollama or llama.cpp, you need to merge it into the base model and export to GGUF.

Option A: Unsloth Direct Export (Easiest)

```
# After training, export merged model as GGUF in one step
model.save_pretrained_gguf(
    "my-model-gguf",
    tokenizer,
    quantization_method="q4_k_m",
)
```

```
# Other quantization options: "q8_0", "q5_k_m", "f16"
```

This merges the LoRA adapter into the base model and quantizes to GGUF in a single call. Output: one `.gguf` file ready for Ollama.

Option B: Manual Merge + llama.cpp Convert

```
# 1. Merge adapter into base model (16-bit)
model.save_pretrained_merged(
    "merged-model",
    tokenizer,
    save_method="merged_16bit",
)
```

```
# 2. Convert to GGUF
cd llama.cpp
python convert_hf_to_gguf.py ../merged-model --outfile my-model.gguf

# 3. Quantize
./llama-quantize my-model.gguf my-model-q4_k_m.gguf q4_k_m
```

Option C: Keep LoRA Separate (No Permanent Merge)

```
# Convert adapter to GGUF format
python convert_lora_to_gguf.py my-7b-adapter

# Merge at runtime
llama-export-lora -m base.gguf -o merged.gguf --lora adapter.gguf
```

This is useful if you maintain multiple LoRA adapters for different tasks against the same base model.

Load in Ollama

Create a `Modelfile`:

```
FROM ./my-model-q4_k_m.gguf

PARAMETER temperature 0.7
PARAMETER num_ctx 2048

SYSTEM "You are a legal document summarizer."
```

```
ollama create my-fine-tune -f Modelfile
ollama run my-fine-tune "Summarize this contract clause: ..."
```

Your custom model is now running locally, same as any other Ollama model.

Training Time and Speed

Real benchmarks for QLoRA, rank 16, 1 epoch, gradient checkpointing on.

7B Model (e.g., Mistral 7B, Llama 3.1 8B)

GPU	500 Examples	5,000 Examples	10,000 Examples
RTX 3060 12GB	~30-60 min	~2-4 hours	~4-8 hours
RTX 3090 24GB	~15-30 min	~1-2 hours	~2-4 hours
RTX 4090 24GB	~10-20 min	~45-90 min	~1.5-3 hours

14B Model (e.g., Qwen 2.5 14B)

GPU	500 Examples	5,000 Examples	10,000 Examples
RTX 3060 12GB	Not feasible (OOM at useful batch)	—	—
RTX 3090 24GB	~30-60 min	~3-5 hours	~5-8 hours
RTX 4090 24GB	~20-40 min	~1.5-3 hours	~3-5 hours

The RTX 4090 is roughly 1.5-2x faster than the 3090, but both have 24GB so they run the same models. For pure training throughput, the 4090 is better. For value, a used RTX 3090 at $600-700 trains the same models at half the cost.

Context Length Impact

Longer sequences eat VRAM fast. On 12GB with a 7B model:

Sequence Length	VRAM (batch 2, rank 16)	Unsloth Max Context
1024	~7 GB	Comfortable
2048	~8-9 GB	Comfortable
4096	~11-12 GB	Tight
8192	OOM	Need 24GB

The difference between Unsloth and standard training is stark here: at 12GB, standard HuggingFace + Flash Attention 2 maxes out around 900 tokens. Unsloth pushes that to ~21,000.

When Training Goes Wrong

OOM on 12GB

The most common problem. Fixes in order of impact:

1. **Enable Unsloth gradient checkpointing** — `use_gradient_checkpointing="unsloth"`. Saves 30% VRAM with ~2% speed cost.
2. **Reduce batch size to 1** — Increase `gradient_accumulation_steps` to compensate (e.g., 16).
3. **Reduce sequence length** — Drop from 2048 to 1024. Only matters if your examples are long.
4. **Reduce LoRA rank** — Drop from 16 to 8. Slight quality cost.
5. **Close everything else** — Chrome, VS Code with GPU acceleration, Docker. Anything using VRAM.
6. **Use `paged_adamw_8bit` optimizer** — Saves ~1GB vs standard AdamW.

If none of this works, your model is too large for your GPU. Drop to a smaller base model.

Loss Spikes or NaN

Your learning rate is too high. Cut it in half and restart. If 2e-4 spikes, try 1e-4. If that spikes, try 5e-5. With QLoRA, 2e-4 works for most models, but some (especially larger ones) prefer 1e-4.

Also check your dataset for corrupt entries: empty fields, extremely long examples, or encoding issues can cause NaN.

Overfitting (Validation Loss Goes Up While Training Loss Drops)

The model is memorizing your training data instead of learning patterns. Signs: it quotes training examples verbatim, performs worse on new inputs than the base model.

Fixes:

- Stop training earlier (1 epoch instead of 3)
- Add more diverse examples
- Lower learning rate
- Try LoRA dropout at 0.05

Catastrophic Forgetting

After fine-tuning, the model loses general abilities it had before. It can do your task but can't hold a normal conversation anymore.

Fixes:

- Mix 10-20% general-purpose examples into your training data
- Train for fewer steps
- Start from an instruct model (already trained for general conversation) rather than a base model
- Lower the learning rate

Model Produces Garbage After Training

Usually means something went wrong with the chat template. The model was trained with one format but you're prompting it with another.

Fix: Make sure your inference prompt matches your training format exactly. If you trained with Alpaca format, prompt with Alpaca format. If you trained with ChatML, use ChatML at inference.

The Bottom Line

A 12GB GPU is enough to fine-tune 7B models with QLoRA. That's not a compromise — 7B models in 2026 are strong enough that a well-tuned one outperforms a generic 14B on your specific task.

The recipe:

1. Install Unsloth: `pip install unsloth`
2. Collect 200-500 examples of your task
3. Run the Recipe 1 script above (~2-4 hours on RTX 3060)
4. Export to GGUF, load in Ollama
5. Test, iterate on the dataset, retrain

If you have 24GB (used RTX 3090, RTX 4090), move up to 14B models for better baseline quality and faster training. The workflow is identical, just change the model name.

Article #100. The site started with "what GPU should I buy." Now you're training your own models on it.

Related Guides

- Fine-Tuning LLMs: LoRA and QLoRA Guide
- What Can You Run on 12GB VRAM?
- What Can You Run on 24GB VRAM?
- Used RTX 3090 Buying Guide
- Quantization Explained
- Local AI Planning Tool — VRAM Calculator

Sources: Unsloth GitHub, Unsloth Benchmarks, Sebastian Raschka — Practical Tips for LoRA, Axolotl GitHub, Modal — How Much VRAM for Fine-Tuning, NVIDIA — Introducing DoRA

Source: https://insiderllm.com/guides/lora-training-consumer-hardware/

Free guides for running AI locally