Addestrare un LLM locale con SFT/LoRA: la procedura (ms-swift)

Ultimo aggiornamento: 2026-11-13

Obiettivo

Addestrare un piccolo modello locale (0,6B) a fare UNA cosa ben definita: leggere una mail di alert in italiano (o l'output di uno script di guardia) e rispondere SOLO con un JSON strutturato. Questo esercizio serve anche a capire in pratica come si addestra un LLM: dataset, LoRA, iperparametri, valutazione.

Nota da leggere prima di iniziare: l'addestramento NON aggiunge conoscenza generale al modello; insegna comportamento e formato (capire l'italiano operativo, rispettare lo schema JSON). La capacità complessiva resta quella della base scelta — per questo il perimetto è "decisore breve", non "esecutore di cron interi".

Stato attuale (già preparato, non rifare)

Tutto sta su pcsebus in `/localIA/training/` ed è già validato end-to-end da Hermes (micro-test completato il 13/11: training, merge, conversione, quantizzazione e inferenza eseguiti davvero):

- ambiente: `venv-swift/` (torch 2.6 cu124 + ms-swift 4.6.0)
- base già scaricata: `hf-cache/` (Qwen/Qwen3-0.6B)
- dataset v1: `dataset/train_openai.jsonl` — 311 esempi: 302 triage di mail NAS reali (dalla casella hermesgirl@busce.it, etichette = tua tabella di triage) + 3 verdetti price-watch reali + 6 auditor
- dataset v4 (quello definitivo): `train_v4.jsonl` — 190 esempi + `holdout_v4.jsonl` 28 famiglie mai viste: etichette da motore a regole deterministiche (Passo 2-bis) e regole di confine scritte nel system prompt (Passo 2-ter)
- micro-test in `output/microtest/` (1 epoca — serviva solo a validare i comandi)

Passo 1 — Aprire la shell

Dal browser: https://busce.it/term → login (utente sebus + password + codice 2FA) → si apre il terminale web su nas64. Poi:

```
ssh pcsebus
cd /localIA/training
source venv-swift/bin/activate
export HF_HOME=/localIA/training/hf-cache
export PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True
```

Verifica che la GPU libera sia la 3060: `nvidia-smi` (la 5060 Ti la lasciamo ai servizi).

Passo 2 — Capire il dataset (2 minuti, è il pezzo che conta)

Ogni riga di `dataset/train_openai.jsonl` è una conversazione:

```
{"messages": [
```

```
  {"role": "system", "content": "Sei un assistente di triage... Rispondi SOLO con JSON valido con chiavi:
gravita, categoria, sintesi_it, azione, notifica_sebus."},
  {"role": "user", "content": "Oggetto: [nas64.internal] SMART error (Temperature)...\n<corpo della mail>"},
  {"role": "assistant", "content": "{\"gravita\": \"critica\", \"categoria\": \"smart\", ...}"}
]}
```

Guardalo con i tuoi occhi:

```
head -2 dataset/train_openai.jsonl | python3 -m json.tool
wc -l dataset/train_openai.jsonl
```

Regole imparate sul campo, valide sempre:

- la qualità del dataset determina la qualità del risultato: 300 esempi ben etichettati battono 5000 sporchi
- le risposte `assistant` sono l'insegnamento: devono rispettare ESATTAMENTE lo schema che vuoi in produzione
- l'10% viene tenuto da parte (`--split_dataset_ratio 0.1`) per misurare i progressi senza barare

Passo 2-bis — Come si costruisce un dataset che insegna davvero

Il dataset v1 (311 esempi etichettati "a occhio") ha prodotto un modello che rispondeva JSON valido al 76% ma sbagliava le categorie, sempre nello stesso modo: tutte le mail diverse finivano nella stessa classe. Non era colpa delle epoche: era incoerenza di etichette (due mail quasi identiche con categorie diverse). Le tre regole che risolvono:

1. una famiglia, una etichetta: raggruppa le mail per oggetto normalizzato (numeri → #, giorni → GG). Le 1085 mail di alert reali sono solo ~108 soggetti unici: il modello deve imparare a livello di famiglia, non memorizzare le varianti.
2. etichette deterministiche, non a occhio: uno script (`build_v3.py`) assegna la categoria con regole a priorità (`if 'degradedarray' in oggetto → disk-raid`, ecc.) e calcola i contesti che il modello non può dedurre (es. il giorno della settimana dalla data reale della mail). Il modello impara le tue regole, senza ambiguità.
3. holdout onesto = intere famiglie escluse, non esempi sparsi: se lo split è per esempio, i quasi-duplicati finiscono in validation e il punteggio è gonfiato (il "99% di token accuracy" del v2 era una balla per questo motivo).

```
python3 build_v3.py    # genera train_v4.jsonl + holdout_v4.jsonl con le regole
```

Passo 2-ter — Il system prompt deve contenere le regole di confine

Il v3 (dataset pulito) era robusto (JSON 92%, gravità 92%) ma sbagliava ancora metà delle categorie: i confini fra classi vicine non erano scritti da nessuna parte. Le mail "Resource limit matched" di domenica alle 03:15 sono il carico ATTESO del backup notturno (monitoring-load), ma il prompt diceva solo "picco loadavg" e il modello le leggeva come transiente. Fix: il prompt enum diventa a trigger, elenca gli identificatori esatti dell'oggetto:

- `"disk-raid" = oggetto con degradedarray / "is now unavailable"`
- `"rear-sospetto" = SOLO se l'oggetto contiene SOSPETTO; "rear" + FAIL = backup-fail`

- "monitoring-load" = Monitoring alert con "Resource limit matched" (atteso 03:00-07:00 / la domenica)
- "security" = ... oppure "Your user account was used to log in" (NON è manutenzione)

Più due righe chiave: "la categoria deve corrispondere a ciò che dice LETTERALMENTE l'oggetto" e "la prima riga del messaggio contiene la data col giorno della settimana: usala". Risultato: categorie su famiglie mai viste 50% → 96%.

Passo 3 — Addestrare (QLoRA) — IL COMANDO DA LANCIARE

Attenzione: quasi tutti i tutorial in rete usano `--train_type`, che non esiste in ms-swift 4.x (ora si chiama `--tuner_type`). Questo è il comando validato sul tuo ambiente:

```
M=$(ls -d hf-cache/hub/models--Qwen--Qwen3-0.6B/snapshots/* | head -1)
CUDA_VISIBLE_DEVICES=1 swift sft \
  --model "$M" \
  --tuner_type lora \
  --quant_bits 4 \
  --dataset train_v4.jsonl \
  --split_dataset_ratio 0.1 \
  --torch_dtype bfloat16 \
  --num_train_epochs 5 \
  --per_device_train_batch_size 2 \
  --gradient_accumulation_steps 8 \
  --learning_rate 1e-4 \
  --lora_rank 16 \
  --lora_alpha 32 \
  --target_modules all-linear \
  --max_length 1536 \
  --eval_steps 25 \
  --save_steps 25 \
  --warmup_ratio 0.05 \
  --save_total_limit 2 \
  --output_dir /home/hermes/train-out-v4
```

Su 3060: ~5,7 GiB di VRAM con `max_length 1536` e batch 2, ~8 s a step, ~8 minuti per 5 epoche. Trucco anti-OOM: i server llama.cpp NON occupano solo la GPU dove li lanci — se non pinli con `CUDA_VISIBLE_DEVICES` toccano TUTTE le GPU (~600 MB ciascuna). Per il training: ferma i server o pinlì su un'altra GPU, e usa `PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True`.

La mappa concettuale per capire cosa stai muovendo:

parametro	cosa significa	come si tocca
`--model`	la BASE da cui parti (quella che "sa già parlare")	cambia base = cambia modello finale
`--tuner_type lora` + `--quant_bits 4`	QLoRA: congela i pesi in 4-bit e addestra solo due piccole matrici aggiunte (i "disturbi" LoRA). Per questo un 0,6B si addestra su una GPU da 12 GB	`lora_rank` = larghezza delle matrici (capacità di adattamento): 8 leggero, 16 medio, 32+ forte ma overfitting in agguato
`--dataset` + `--split_dataset_ratio`	gli esempi input→output; l'10% serve per misurare senza barare	è LA leva più importante, più di ogni parametro

parametro	cosa significa	come si tocca
`--learning_rate`	quanto grandi sono i passi di aggiornamento	1e-4 è l'intervallo sicuro per LoRA; 1e-3 destabilizza; 1e-5 non impara
`--num_train_epochs`	quante volte rivede tutto il dataset	con poche centinaia di esempi: 2–3; col dataset v3 a 3 epoche l'eval stava ancora scendendo (si può provare 5)
`per_device_train_batch_size` × `gradient_accumulation_steps`	batch effettivo = quante domande guarda prima di correggersi (4×4=16)	GPU libera? aumenta il primo; altrimenti il secondo
`--max_length`	lunghezza massima in token degli esempi	deve coprire le mail più lunghe del dataset
`--warmup_ratio`	all'inizio passi piccoli per non destabilizzare	0.05 standard, lascialo

Cosa guardare mentre gira: le righe `{'loss': ...}` — il loss deve scendere in modo irregolare ma deciso e poi appiattirsi (nel micro-test: 2,3 → 1,4 in un'epoca, `token_acc` 63%→74%: stava imparando). Se il loss di EVAL sale mentre quello di training scende = overfitting: fermati a un checkpoint precedente.

Passo 4 — Unire la LoRA alla base (merge)

L'output dell'addestramento contiene solo le matrici LoRA (poche decine di MB). Per un modello autonomo (comando validato):

```
CKPT=$(ls -d /home/hermes/train-out-v4/*/checkpoint-* | tail -1)
swift export --model "$M" --adapters "$CKPT" --merge_lora true --output_dir /home/hermes/train-out-v4/merged
```

Passo 5 — Convertire in GGUF e quantizzare

Il formato che llama.cpp usa per l'inferenza (comandi validati, ~2 minuti in tutto):

```
python3 /localIA/llama/llama.cpp/convert_hf_to_gguf.py /home/hermes/train-out-v4/merged --outfile /home/
hermes/triage-v4.f16.gguf --outtype f16
/localIA/llama/llama.cpp/build/bin/llama-quantize /home/hermes/triage-v4.f16.gguf /home/hermes/triage-v4-
Q4_K_M.gguf Q4_K_M
```

Risultato atteso: ~1,2 GB f16 → ~400 MB Q4_K_M.

Passo 6 — Valutare (dove si decide se ha imparato)

```
CUDA_VISIBLE_DEVICES=1 /localIA/llama/llama.cpp/build/bin/llama-server   -m /home/hermes/triage-v4-Q4_K_M.
gguf --jinja -c 4096 -np 1 -cram 0 --port 11436 &
```

(`CUDA_VISIBLE_DEVICES` punta la GPU giusta; senza, llama.cpp tocca tutte le GPU.)

Tre trappole che falsano la misura (verificate tutte):

- server di default = misura inaffidabile: llama-server con più slot e prompt-cache riusa stato fra richieste e a temperature 0 può dare output diversi su input identici. Per misurare usare sempre `-np 1 -cram 0`.

- Qwen3 "pensa" prima di rispondere: la risposta può contenere prima un blocco di ragionamento e poi il JSON. Con `max_tokens` basso il ragionamento si mangia tutto il budget e la risposta risulta vuota o troncata → sembra un modello rotto che non lo è. Nel client aggiungere `"chat_template_kwargs": {"enable_thinking": false}` con `max_tokens ≥ 900`.

Poi manda in batch l'holdout (famiglie mai viste) con lo stesso system prompt del training, e conta: JSON valido? chiavi esatte? gravità corretta? Confronta con lo stesso identico test sul modello base NON addestrato: il "guadagno dell'addestramento" è la differenza, non il punteggio assoluto — e i numeri sono confrontabili solo fra bench fatti con lo stesso holdout e lo stesso server deterministico.

Soglie per dire "funziona": JSON valido ≥ 95%, nessuna risposta vuota su input valido, gravità corretta ≥ 90%. Valori del modello finale v4 (28 famiglie unseen, config sopra, GPU 3060): JSON 100%, vuote 0, gravità 100%, categorie 96% — l'unico errore è una mail di test del canale mail ("[test] canale rear-rotation"), classificata backup-ok per il token "rear". Produzione attiva: `pcsebus:11436`, alias `triage-v4`.

Passo 7 — Il vero esperimento (dopo che il giro funziona)

1. stesso identico comando su un'altra base (es. `Qwen3-1.7B`): vince solo se categorie e gravità salgono sopra le soglie
2. variare UNA cosa alla volta per imparare: `lora_rank` 8 vs 16; 1 epoca vs 3; metà dataset vs tutto. Annota loss di eval e punteggio del bench: è tutto qui il metodo.

Appendice — Glossario rapido

- pretraining (lontanissimo da noi): il modello impara la lingua da trilioni di token. Costi da datacenter.
- SFT (quello che facciamo qui): insegnare compiti specifici con esempi domanda→risposta corretta.
- LoRA/QLoRA: addestrare solo piccoli "disturbi" sopra un modello congelato; i pesi originali non si toccano.
- loss: errore medio sulle risposte attese; il suo andamento dice se stai imparando o memorizzando.
- allucinazione: completamento plausibile ma falso. L'SFT su estrazione la riduce: il formato ammette solo dati presenti nella mail.