

Come nasce un LLM — dalla parola al modello che gira sul tuo computer

Tag: AI · DeepSeek · Distillazione · GGUF · Guide · LLM · MoE · PyTorch · Qwen · Transformer

Ultimo aggiornamento: 2026-09-27

Come nasce un LLM — dalla parola al modello che gira sul tuo computer

Guida passo passo, scritta per essere capita anche da un ragazzo delle superiori. Dove serve si scende nel tecnico: comandi veri, numeri veri, esempi veri.

Introduzione — cosa saprai fare alla fine

Alla fine di questo documento saprai rispondere a queste domande:

- 1 Che cos'è davvero un LLM (e perché «indovina la parola dopo» è tutto il trucco).
- 2 Che cos'è un Transformer e cosa fanno encode e decode.
- 3 Con quali strumenti si costruisce un LLM (PyTorch e dintorni).
- 4 Come si addestra (pre-training) e come si rinforza (SFT, RL, DPO).
- 5 Che cos'è un MoE (Mixture of Experts) e perché un modello da 35 miliardi gira su un portatile.
- 6 Come si crea un file GGUF quantizzato e perché è la chiave per l'uso locale.
- 7 Che cos'è la distillazione di un modello.
- 8 Come nasce un modello «senza censura» (uncensored).
- 9 Cosa hanno di nuovo Qwen 3.8 e DeepSeek V4.1 (settembre 2026).

Non serve saper programmare per leggere: serve solo curiosità. I comandi sono alla fine di ogni sezione, per chi vuole provare.

1 · Come un LLM «vede» il testo: i token

Non lettere, non parole: pezzi di parola statisticamente convenienti

Il gatto dorme sul tetto, perché il sole è caldo.

testo di partenza

Il	·gatto	·dorm	e	·sul	·tett	o	,	·perché	·il	·sole	·è	·cal	do	.
412	4183	8412	311	1902	7710	299	11	6512	366	5521	640	3312	288	13

«gatto» in italiano = 2 token: «·gatt» + «o» → il tokenizer impara le fusioni più frequenti (BPE), quindi in italiano è ~1 token ogni 0,7-0,9 parole. Ogni token diventa il numero indicato (id) e da lì il modello pesca una riga della sua tabella di embedding.

```
embedding[412] -> [ 0.021, -0.44, 0.87, 0.13, ... ]
embedding[4183] -> [-0.31, 0.02, -0.55, 0.71, ... ]
embedding[8412] -> [ 0.09, -0.12, 0.33, -0.08, ... ]
```

16 mila parole di vocabolario x dimensione (es. 4096) = la prima «tabella» del modello

Se cambi una lettera il token cambia id e il suo vettore cambia:
il modello NON vede lettere, vede vettori. La parola «gatto» e la parola «gatti» finiscono in punti vicini dello spazio → per questo «capisce».

tokens

1 · Che cos'è un LLM

LLM = Large Language Model, «grande modello del linguaggio».

La definizione onesta è questa:

Un LLM è un programma che, dato un pezzo di testo, calcola la probabilità di quale sarà il prossimo pezzo di testo. Ripete l'operazione una volta per ogni pezzo, e così «scrive».

Tutto qui? Sì, tutto qui. La parte incredibile è che per indovinare bene la parola successiva il modello è costretto a costruirsi dentro una quantità enorme di cose: grammatica, fatti sul mondo, traduzioni implicite tra lingue, la struttura del codice, piccole catene di ragionamento. Nessuno gli ha detto «impara la grammatica»: la grammatica serve per indovinare meglio, quindi la impara da sé.

1.1 Il pezzo di testo si chiama token

Il modello non vede lettere né parole: vede token, cioè pezzi di parola. Un token può essere una parola intera, mezza parola, un segno di punteggiatura.

Esempio con la frase «Il gatto dorme sul tetto, perché il sole è caldo.»:

testo	Il	· gatto	· dorm	e	· sul	· tett	o	,	· perché	...
id	412	4183	8412	311	1902	7710	299	11	6512	...

In italiano 1 token $\approx$ da 0,7 a 0,9 parole. Le regole con cui il testo viene spezzato si chiamano tokenizer (in genere un algoritmo chiamato BPE, Byte Pair Encoding): si parte dalle lettere e si fondono insieme le coppie di simboli che compaiono più spesso. In pratica il tokenizer «impara» le sillabe e i pezzi utili della lingua.

1.2 Da numero a vettore: l'embedding

Ogni id viene trasformato in un vettore: una lista di numeri (es. 4096 numeri). Vettori di parole simili finiscono vicini nello spazio. È così che il modello «capisce» che gatto e gatti sono parenti, e che Parigi sta a Francia come Roma sta a Italia.

1.3 Il ciclo di generazione

- 1 Il testo entra → diventa token → diventa vettori.
- 2 Il modello calcola le probabilità del token successivo. Esempio: tetto 41%, divano 17%, letto 11%, ramo 7%.
- 3 Se ne sceglie uno (vedi §6.4, temperatura).
- 4 Il token scelto viene riappeso in fondo al testo e si ricomincia.

Un modello che scrive una risposta di 300 parole esegue questo giro circa 400 volte. Non «pensa prima e poi scrive»: scrive, e ogni parola influenza la successiva.

Da ricordare: un LLM non ha una banca dati di frasi pronte. Ha miliardi di numeri («pesi») che, combinati, producono il calcolo delle probabilità.

2 · Il Transformer — il motore dentro tutto

Prima del 2017 i modelli leggevano il testo in ordine, una parola dopo l'altra, come si legge con il dito sul foglio: era lento e dimenticava presto l'inizio della frase.

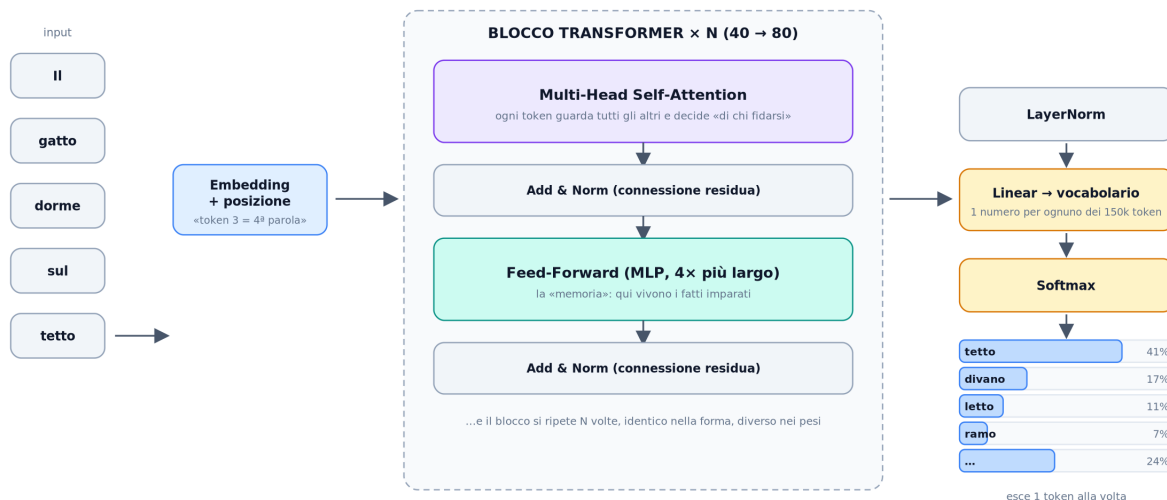
Nel 2017 arriva il Transformer (paper «Attention is All You Need»). L'idea:

Guarda tutte le parole insieme, e per ogni parola decidi da quali altre parole trarre informazione.

Questo meccanismo si chiama attenzione (self-attention).

2 · Dentro il Transformer: cosa succede a ogni parola

Stessa struttura ripetuta N volte: attenzione → memoria → attenzione → memoria



transformer

2.1 Cosa fa l'attenzione, in parole povere

Nella frase «Il gatto dorme sul tetto perché è stanco», la parola è deve capire a chi si riferisce. Con l'attenzione il token è «chiede» a tutti gli altri token: quanto sei rilevante per me? E riceve risposte tipo:

- gatto → molto rilevante (0,62)
- tetto → poco (0,11)
- dorme → abbastanza (0,20)

Il risultato è una media pesata dei significati di tutte le parole: è diventata una miscela che sa di gatto e di stanchezza. È così che nasce il contesto.

Tecnicamente ogni token produce tre vettori — Query (cosa cerco), Key (cosa offro), Value (cosa porto) — e l'attenzione è:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{Q \cdot K^T}{\sqrt{d}} \right) \cdot V$$

Non serve capire la formula: basta capire che $Q \cdot K^T$ misura «quanto siamo simili», softmax trasforma quei numeri in percentuali, e $\cdot V$ fa la media pesata.

2.2 Il blocco completo

Ogni blocco del Transformer fa due lavori, ripetuti N volte (da 40 a 80):

- 1 Multi-Head Self-Attention — ogni token guarda tutti gli altri («di chi mi fido?»).
- 2 Feed-Forward (MLP) — qui vive la maggior parte della «memoria» del modello: i fatti imparati.
- 3 Connessione residua + LayerNorm — servono a far funzionare la rete in profondità senza che l'informazione si perda.

Alla fine di tutto: LayerNorm → Linear (un numero per ogni token del vocabolario) → Softmax = le probabilità del prossimo token.

Multi-Head: non una ma 32-64 attenzioni in parallelo, ognuna specializzata (una guarda la sintassi, una le entità, una le ripetizioni...). Poi i risultati vengono concatenati.

3 · Encoder e decoder — chi fa cosa

Tre famiglie di Transformer, tre mestieri diversi:

3 · Encoder, decoder, o tutti e due

La stessa idea (attenzione) usata in tre modi diversi



Regola: cambiare l'architettura cambia il mestiere. Per «parlare» serve un decoder — ed è per questo che oggi tutti costruiscono decoder.

encoder-decoder

Tipo	Mestiere	Esempi	Come guarda
Encoder-only	capire: classificare, cercare, raggruppare	BERT, modelli di embedding	ogni token vede tutti gli altri (bidirezionale)
Decoder-only	generare: scrivere, chattare, programmare	GPT, Llama, Qwen, DeepSeek, Mistral	ogni token vede solo quelli prima (causale)
Encoder-Decoder	trasformare: tradurre, riassumere	T5, Whisper, traduttori	encoder legge tutto → decoder genera con cross-attention

- Encode = il percorso dall'input verso la rappresentazione interna. Nei modelli moderni comprende anche la tokenizzazione (testo → id).
- Decode = il percorso dalla rappresentazione interna all'output, un token alla volta.

Perché per chattare serve un decoder-only? Perché la maschera causale impedisce al modello di «sbirciare» il futuro: così può essere addestrato a indovinare il token successivo su qualsiasi testo esistente. È il trucco che ha reso possibile addestrare su tutto internet.

Perché «decoder» e non «encoder-decoder»? Perché un decoder solo è più semplice da addestrare, scala meglio e con abbastanza dati fa anche le traduzioni. Nel 2026 il 99% dei modelli da chat è decoder-only — con un'eccezione importante: DeepSeek V4.1 usa di nuovo un encoder-decoder (CED) per risparmiare memoria (vedi §9).

3.1 Encode/decode nel senso «tokenizer»

Attenzione a un doppio uso delle parole:

```
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen3.8-27B")

ids = tok.encode("Il gatto dorme sul tetto") # encode: testo -> numeri
print(ids) # [412, 4183, 8412, ...]
print(tok.decode(ids)) # decode: numeri -> testo
# 'Il gatto dorme sul tetto'
```

4 · Con cosa si costruisce un LLM — PyTorch e dintorni

4 · Con cosa si costruisce un LLM: lo stack degli strumenti

Dal metallo (GPU) al modello che parla: ogni livello ha il suo mestiere



Regola: i livelli bassi li toccano gli ingegneri delle grandi aziende; per addestrare o raffinare un modello in casa si lavora quasi sempre con Transformers + TRL + PEFT (o Unsloth) sopra PyTorch.

stack

4.1 PyTorch: il motore

PyTorch è la libreria con cui si scrivono le reti neurali. Non è «un programma da lanciare»: è il linguaggio in cui si descrive il modello.

Cosa offre:

- `torch.Tensor` — il mattone base: una griglia di numeri che vive sulla CPU o sulla GPU.
- `autograd` — il pezzo magico: registra tutte le operazioni e calcola da solo i gradienti (le «colpe» dei singoli pesi nell'errore commesso).
- `nn.Module` — il modello: strati impilati, come mattoncini.
- `DataLoader` — serve i dati in blocchi (batch).
- `optimizer` — AdamW, Muon: spostano i pesi per ridurre l'errore.
- AMP / bf16 — calcolo a metà precisione: metà memoria, doppio velocità.
- `DistributedDataParallel` / FSDP — la stessa riga di codice su mille schede video.

Esempio minimo che addestra qualcosa di vero:

```
import torch, torch.nn as nn, torch.nn.functional as F

model = nn.Sequential(nn.Linear(4096, 16384), nn.GELU(), nn.Linear(16384, 150000))
opt = torch.optim.AdamW(model.parameters(), lr=3e-4)
```

```

for batch in dataloader:                # batch: (testo, token_corretto)
    x, y = batch
    logits = model(x)                    # 1. forward
    loss = F.cross_entropy(logits.view(-1, 150000), y.view(-1)) # 2. errore
    loss.backward()                      # 3. backpropagation
    opt.step(); opt.zero_grad()          # 4. correzione

```

Quattro righe: forward → loss → backward → update. È tutto l'addestramento. Il resto è scala.

Perché PyTorch e non altro? Perché è flessibile, si legge come Python normale e, soprattutto, l'intero ecosistema della ricerca ci è stato costruito sopra. Esistono alternative (JAX per la velocità, TensorFlow per la produzione storica), ma nel 2026 PyTorch è la lingua franca. Le grandi aziende non partono da PyTorch «puro» per addestrare: usano Megatron-LM, DeepSpeed o FSDP per spalmare il modello su migliaia di GPU — ma sono strati sopra PyTorch.

4.2 Le librerie di alto livello

- Hugging Face Transformers — migliaia di modelli pronti all'uso con due righe.
- TRL — SFT, DPO, GRPO già pronti.
- PEFT / LoRA — addestrano solo una piccola parte dei pesi: un fine-tuning su una sola GPU invece che su un cluster.
- Unsloth / Axolotl — acceleratori: rendono il fine-tuning locale 2-5 volte più veloce.
- vLLM — server di inferenza veloce (batching continuo, PagedAttention).
- llama.cpp — esecuzione su CPU, in formato GGUF: è il motivo per cui un modello gira sul tuo portatile.

4.3 Cosa serve, in soldi e ferro

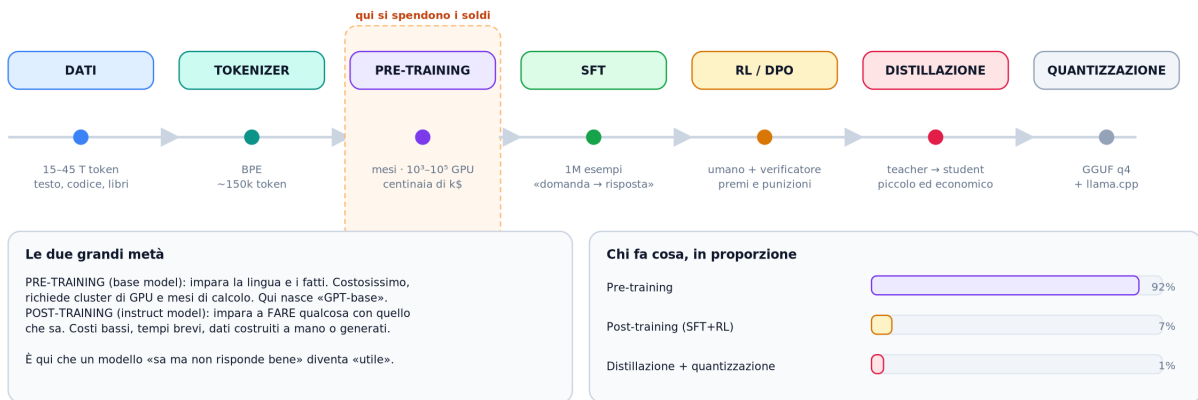
Fase	Hardware tipico	Costo indicativo
Pre-training da zero (frontiera)	10.000-100.000 GPU per mesi	da decine di milioni di \$
Pre-training piccolo (1-8B)	8-64 GPU per settimane	migliaia di \$
Fine-tuning LoRA di un 8B	1 GPU per poche ore	decine di \$
Distillazione	1-8 GPU	centinaia di \$
Quantizzazione	1 PC	qualche ora di CPU/GPU

Notizia bellissima per chi impara: il pre-training costa il 90% del totale, ma le competenze che si acquisiscono nelle altre fasi sono le stesse. Si impara tutto spendendo poco.

5 · Come si crea un LLM: le fasi

5 • Le fasi: dalla raccolta dati al modello che gira da te

Il pre-training costa il 90% e non è quello che rende il modello «simpatico»



fasi

5.1 Raccolta dati

Testo, codice, libri, articoli, matematica. Si parla di 10-45 trilioni di token (DeepSeek V4.1: 45T). Gran parte del lavoro è pulizia: rimuovere duplicati, spam, contenuti tossici, dati personali.

5.2 Addestramento del tokenizer

Si sceglie il vocabolario (~150.000 token) e si allena il BPE sul corpus. Questa scelta condiziona tutto: un tokenizer che spezza male l'italiano rende il modello più lento e meno preciso sulla nostra lingua.

5.3 Pre-training — il modello «base»

4 • Il motore di tutto: il ciclo di addestramento

Forward → loss → backward → update. Ripetuto qualche milione di volte.



```
import torch, torch.nn.functional as F

logits = model(x) # 2. forward: (batch, sequenza, vocabolario)
loss = F.cross_entropy(
    logits.view(-1, V), y.view(-1)) # 3. loss
loss.backward() # 4. backward: i gradienti
optimizer.step() # 5. update dei pesi
optimizer.zero_grad() # pulizia, si ricomincia
```

«Indovinare la parola dopo» sembra un gioco stupido: è tutto qui. Per indovinare bene, il modello è OBBLIGATO a costruirsi dentro:

- grammatica e sintassi · traduzione implicita tra lingue
- fatti del mondo · aritmetica semplice
- struttura del codice · catene di ragionamento brevi

La «loss» è solo un numero (es. 2.31) che dice quanto è sbagliato. Deve scendere: 2.31 → 1.4 → 0.9. Quando smette di scendere, fine del pre-training.

training-loop

Si addestra il modello a indovinare il token successivo su tutto il corpus. Il progresso si misura con la loss (l'errore medio). Esempio reale di discesa:

step	loss
1.000	3.41
100.000	2.31
1.000.000	1.87
5.000.000	1.52
20.000.000	1.34

← rendimenti decrescenti

Quando la loss smette di scendere, il pre-training è finito. Il risultato è un modello base: conosce la lingua e i fatti, ma non «risponde» — se gli chiedi «Capitale dell'Italia?» potrebbe continuare con «Capitale dell'Italia? 1. Napoli 2. Roma» perché ha visto troppi quiz.

Durata: da giorni (modelli piccoli) a mesi (frontiera).

5.4 SFT — Supervised Fine-Tuning

Si prendono ~1 milione di coppie «domanda → risposta ideale» scritte da umani e si continua l'addestramento su quelle. Il modello impara la forma del dialogo: rispondere, seguire istruzioni, rifiutare richieste pericolose, usare il formato giusto.

Costo: basso. Tempo: ore o giorni. È il passaggio che trasforma un modello inutilizzabile in un assistente.

5.5 RL — Reinforcement Learning (il «rinforzo»)

Qui si va oltre l'imitazione: si dà al modello un premio quando fa bene.

Il ciclo classico (RLHF, Reinforcement Learning from Human Feedback):

- 1 Il modello produce due risposte alla stessa domanda.
- 2 Un umano dice quale è migliore. (Migliaia di volte.)
- 3 Quelle preferenze addestrano un reward model: un secondo modello che impara a dare un voto («quanto piacerebbe agli umani?»).
- 4 Il modello principale viene addestrato a massimizzare quel voto (con un vincolo che gli impedisca di allontanarsi troppo dal comportamento iniziale).

Attenzione al realismo: chiedere a degli umani di valutare milioni di risposte è caro e lento. Per questo oggi si usano molto di più metodi senza reward model umano: DPO (Direct Preference Optimization) — si addestra direttamente sulle coppie «questa risposta è meglio di quella», senza passare dal reward model. Semplice ed efficace. GRPO — gruppi di risposte valutate da un verificatore automatico. Funziona benissimo dove la risposta è verificabile: matematica (il risultato è giusto o sbagliato), codice (i test passano o no). È il motore dietro i modelli «ragionatori» del 2025–2026. RLVR (RL with Verifiable Rewards) — il nome generale di questa famiglia.

Il rinforzo è ciò che ha trasformato i modelli dal «rispondere con sicurezza cose sbagliate» al «provare, controllare, correggere».

5.6 Ordine pratico, con i costi

dati → tokenizer → PRE-TRAINING → SFT → RL/DPO → (distillazione) → (quantizzazione) → deploy

				
~90% del costo totale	~7%	~2%	1%	1%

6 · Mixture of Experts (MoE): tanti esperti, pochi accesi

6 • Mixture of Experts: tanti esperti, pochi accessi

Non serve che tutto il cervello lavori per dire «ciao»



L'onestà che serve: gli esperti «spenti» occupano comunque RAM (vanno tenuti in memoria). Il MoE risparmia calcolo, non memoria → con la CPU/GPU giusta è 3-5× più veloce a parità di qualità.

moe

6.1 Il problema

Un modello dense (tradizionale) usa tutti i suoi parametri per ogni token. Un 27B dense attiva 27 miliardi di pesi anche solo per scrivere , . Per sapere di più bisogna ingrandire il modello → ogni token costa sempre di più.

6.2 L'idea del MoE

Si sostituisce l'unico grande strato Feed-Forward con molti esperti (cioè molti piccoli Feed-Forward indipendenti) e un router:

- 1 Il token arriva al router.
- 2 Il router decide a quali esperti mandarlo (tipicamente 2 su 8, oppure 6 su 384).
- 3 Solo quegli esperti calcolano. Gli altri restano spenti.

Esempio reale — DeepSeek V4.1-Flash: ogni strato MoE ha 1 esperto condiviso + 384 esperti e ne attiva 6 per token. Il modello ha 552 miliardi di parametri, ma ne accende 8-16 miliardi. Il risultato: qualità da modello gigantesco, costo di calcolo da modello medio.

6.3 Esempio concreto e numeri

Modello	Parametri totali	Attivi per token	RAM minima (q4)
Qwen3.8-27B (dense)	27B	27B	~16 GB
Qwen3.8-35B-A3B (MoE)	35B	3B	~20 GB
DeepSeek V4.1-Flash	552B + 196B	8B / 16B	oltre 300 GB
Qwen3.8-Flash-Next	125B (+51B n-gram)	6B	~70 GB

6.4 La verità che nessuno dice

Il MoE risparmia calcolo, non memoria. Gli esperti «spenti» devono comunque stare in RAM perché il prossimo token potrebbe richiederli. Quindi:

- Con poca RAM → meglio un modello denso piccolo.
- Con buona RAM e poca GPU → il MoE è il miglior affare: 3-5× più veloce a parità di qualità.

6.5 Perché il router non impazzisce

Se il router mandasse tutto su due esperti, gli altri non imparerebbero mai. Si aggiunge perciò una loss di bilanciamento che penalizza le distribuzioni troppo sbilanciate. È un dettaglio tecnico, ma è la differenza tra un MoE che funziona e uno inutile.

7 · Quantizzazione: come si crea un GGUF

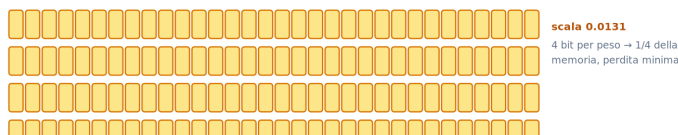
7 · Quantizzazione: come nasce un GGUF

Meno bit per peso = meno RAM = il modello entra nel tuo computer

1 peso in FP16 → 16 bit → 2 byte



1 blocco di 32 pesi in Q4_K_M → 4 bit + una scala



```
# 1) dal modello originale (safetensors) al GGUF in precisione alta
python convert_hf_to_gguf.py ./Qwen3.8-72B --outfile qwen38-27b-f16.gguf --outtype f16
```

```
# 2) quantizzazione (la scelta che conta)
./llama-quantize qwen38-27b-f16.gguf qwen38-27b-Q4_K_M.gguf Q4_K_M
```

```
# 3) il trucco pro: quantizzare MISURANDO l'importanza dei pesi
./llama-imatrix -m qwen38-27b-f16.gguf -f wikipedia-it.txt -o imatrix.dat
./llama-quantize --imatrix imatrix.dat qwen38-27b-f16.gguf qwen38-27b-IQ4_XS.gguf IQ4_XS
```

Quanto pesa davvero un 27B

FP16 / BF16	≈ 54 GB
Q8_0	≈ 29 GB
Q6_K	≈ 22 GB
Q5_K_M	≈ 19 GB
Q4_K_M	≈ 16 GB
Q3_K_M / IQ3	≈ 13 GB

Il GGUF non è solo «i pesi»: dentro ci sono tokenizer, template della chat, metadati della quantizzazione, tensori pronti per mmap.

Per questo un unico file gira su llama.cpp, Ollama, LM Studio, koboldcpp, telefoni e Raspberry.

Regola pratica: q4_k_m è quasi sempre il miglior compromesso. Sotto q3 la qualità crolla in modo evidente; sopra q6 paghi memoria senza guadagno percepibile.

quantizzazione

7.1 Che cos'è la quantizzazione

Un peso di una rete è un numero con la virgola (es. 0.0273437...). In FP16 occupa 16 bit = 2 byte. Con 27 miliardi di pesi servono ~54 GB: troppo per un PC.

La quantizzazione riduce i bit per peso: 8, 6, 5, 4, perfino 3. Con 4 bit per peso lo stesso modello scende a ~16 GB — e gira su una macchina normale, con una perdita di qualità spesso impercettibile.

Metodo Q4_K_M: i pesi vengono raggruppati in blocchi di 32 e ogni blocco ha una scala che riconduce i valori a 4 bit. I blocchi rendono l'errore distribuito e limitato.

7.2 Perché proprio il formato GGUF

GGUF (GGML Universal File) è il formato di llama.cpp. Un file GGUF non contiene solo i pesi: contiene anche

- il tokenizer,
- il chat template (come si formatta il dialogo),
- i metadati della quantizzazione,
- i tensori disposti per il caricamento via mmap (apertura istantanea).

Risultato: un solo file che gira su llama.cpp, Ollama, LM Studio, koboldcpp, Raspberry Pi e telefoni.

7.3 Procedura, passo per passo

Passo 1 — procurarsi il modello in formato originale (safetensors su Hugging Face).

```
git lfs install
git clone https://huggingface.co/Qwen/Qwen3.8-27B
```

Passo 2 — convertire in GGUF non quantizzato (FP16)

```
python convert_hf_to_gguf.py ./Qwen3.8-27B \
--outfile qwen38-27b-f16.gguf \
--outtype f16
```

Passo 3 — quantizzare

```
# la scelta che conta
./llama-quantize qwen38-27b-f16.gguf qwen38-27b-Q4_K_M.gguf Q4_K_M
```

Passo 4 (consigliato) — misurare l'importanza dei pesi con imatrix

Non tutti i pesi contano uguale. Si può misurare quali contano di più facendo passare del testo reale e quantizzare con più precisione quelli:

```
./llama-imatrix -m qwen38-27b-f16.gguf \
-f wikipedia-it.txt \
-o imatrix.dat

./llama-quantize --imatrix imatrix.dat \
qwen38-27b-f16.gguf \
qwen38-27b-IQ4_XS.gguf IQ4_XS
```

Passo 5 — provare

```
./llama-cli -m qwen38-27b-Q4_K_M.gguf \
-p "Spiega in tre frasi cos'è un Transformer" \
-n 200 --temp 0.7
```

7.4 Tabella delle scelte (per un modello da 27B)

Quantizzazione	Peso su disco	Qualità	Quando usarla
FP16	~54 GB	riferimento	solo per quantizzare o per server grossi
Q8_0	~29 GB	quasi identica	quando la RAM abbonda
Q6_K	~22 GB	praticamente identica	ottimo compromesso «alto»
Q5_K_M	~19 GB	eccellente	buon compromesso
Q4_K_M	~16 GB	molto buona	la scelta standard di tutti
IQ4_XS + imatrix	~14 GB	buona	quando serve scendere ancora
Q3_K_M	~13 GB	accettabile	solo se non c'è alternativa
Q2/Q1	<10 GB	pessima	sconsigliata

Regola pratica: Q4_K_M è quasi sempre il punto di equilibrio migliore. Sotto Q3 la qualità crolla in modo evidente (ripetizioni, errori logici); sopra Q6 si paga memoria senza guadagno percepibile.

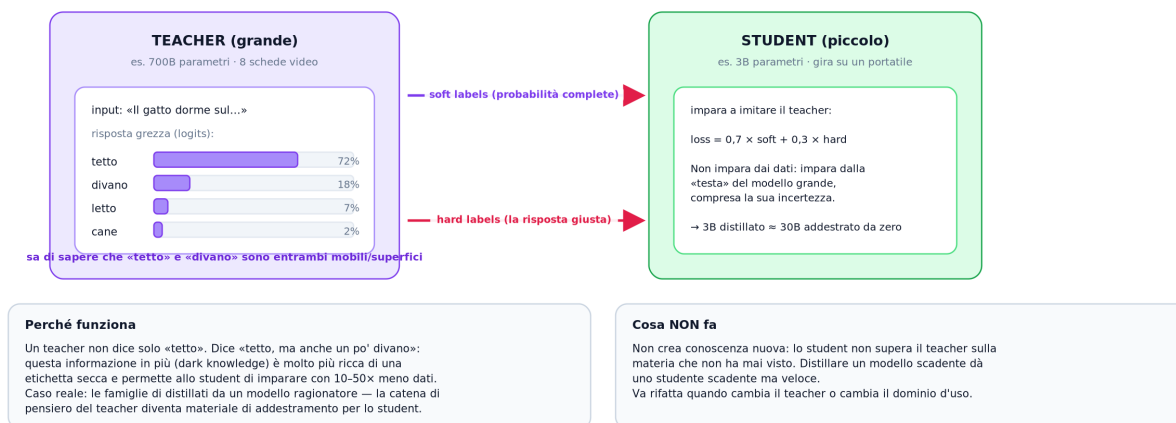
7.5 Vocabolario minimo del formato

Sigla	Significato
F16 / BF16	16 bit per peso, nessuna compressione
Q8_0, Q6_K, Q5_K_M, Q4_K_M	quantizzazione «K-quant»: blocchi con scala
IQ4_XS, IQ3_XXS	I-quant: più lenti da produrre, migliori a parità di bit
imatrix	file di importanza dei pesi, migliora molto le quantizzazioni basse

8 · La distillazione di un modello

8 · Distillazione: far imparare un piccolo dal grande

Lo studente non impara solo la risposta; impara anche i dubbi del maestro



distillazione

8.1 L'idea

Immagina di avere un insegnante bravissimo ma lento e costoso (700B parametri, 8 schede video) e di voler ottenere uno studente veloce che giri su un portatile (3B parametri).

Nella distillazione lo studente non impara dai dati grezzi: impara imitando l'insegnante.

8.2 Il dettaglio che fa la differenza: le soft labels

Se l'insegnante, dopo «Il gatto dorme sul...», risponde:

token	probabilità
tetto	72%
divano	18%
letto	7%
cane	2%

...allora dice due cose:

- 1 la risposta giusta è tetto;
- 2 ma divano e letto sono plausibili, mentre cane no.

Questa seconda informazione si chiama dark knowledge ed è ricchissima: insegna allo studente le relazioni tra concetti. Una etichetta secca (tetto, punto) non contiene nulla di tutto questo.

La loss dello studente combina le due cose:

```
loss =  $\alpha$  · KL(soft teacher, soft student) + (1 -  $\alpha$ ) · cross_entropy(hard label, student)
```

Con questo metodo uno studente da 3B può arrivare a comportarsi come un modello da 30B addestrato da zero — con 10-50 volte meno dati.

8.3 La distillazione delle catene di pensiero

Nei modelli ragionatori si fa una variante: si fa generare all'insegnante il ragionamento completo (i passaggi, non solo la risposta finale) e lo studente impara a riprodurre anche quello. È il modo con cui sono nate molte famiglie di modelli piccoli ma «pensanti».

8.4 Cosa NON fa la distillazione

- Non crea conoscenza nuova. Lo studente non può superare l'insegnante su ciò che l'insegnante non sa.
- Trasmette anche gli errori. Gli studenti ereditano i pregiudizi e i difetti dell'insegnante.
- Va rifatta quando cambia l'insegnante o il dominio d'uso.

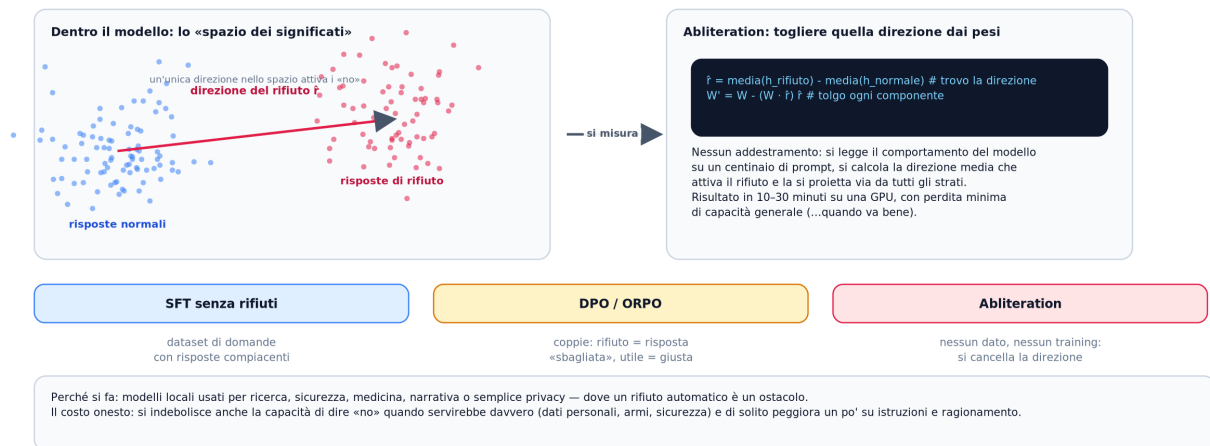
```
# Sketch: distillazione con PyTorch
teacher.eval()
student.train()
for x, y in loader:
    with torch.no_grad():
        t_logits = teacher(x)                # l'insegnante non impara
    s_logits = student(x)
    loss = F.kl_div(F.log_softmax(s_logits / T, -1),
                   F.softmax(t_logits / T, -1),
                   reduction="batchmean") * T**2 \
          + F.cross_entropy(s_logits, y)      # hard label
    loss.backward(); opt.step(); opt.zero_grad()
```

T è la temperatura: alzandola, le probabilità si «ammorbidiscono» e la dark knowledge emerge meglio.

9 · Come nasce un modello «senza censura» (uncensored)

9 • I modelli «senza censura»: come si fanno davvero

Non è magia e non è (solo) una questione legale: è algebra lineare applicata ai pesi



uncensored

Un modello che risponde «Non posso aiutarti con questa richiesta» ha imparato quel comportamento nel post-training (SFT e RL): è diventato un rifiuto, non una mancanza di capacità. Esistono tre strade per ridurlo.

9.1 Strada 1 — SFT senza rifiuti

Si prepara un dataset di domande con risposte compiacenti (o si fa rispondere un altro modello senza filtri) e si fa un fine-tuning sopra. Funziona, ma richiede dati e può rovinare la generale capacità del modello.

9.2 Strada 2 — DPO / ORPO sulle preferenze

Si costruiscono coppie di risposte:

- positiva = risposta utile e completa;
- negativa = il rifiuto.

Poi si insegna al modello a preferire la prima. Meno dati della SFT, effetto più mirato.

9.3 Strada 3 — Abliteration (la più elegante)

Questa è la scoperta più interessante degli ultimi anni. Analizzando gli stati interni (hidden states) del modello si osserva che:

i rifiuti non sono sparsi ovunque: esiste una direzione nello spazio dei significati che attiva il comportamento di rifiuto.

Allora:

1. si fanno passare ~100 prompt "problematici" e ~100 prompt neutri;
2. si calcola $\hat{r} = \text{media}(h_{\text{rifiuto}}) - \text{media}(h_{\text{normale}})$ # la direzione
3. si proietta via quella componente dai pesi di tutti gli strati:
 $W' = W - (W \cdot \hat{r}) \hat{r}$
4. si salva il modello: non serve alcun addestramento.

Risultato in 10-30 minuti su una GPU, con perdita minima di capacità generale (quando va bene). Una libreria famosa per farlo è `abliterator`.

Perché è possibile? Perché le reti neurali rappresentano i concetti come direzioni in uno spazio vettoriale. Toglierne una è un'operazione di algebra lineare, non una modifica «chirurgica»

imprevedibile. (Il rovescio della medaglia: tutte le direzioni si sovrappongono un po', quindi si indebolisce anche la capacità di dire «no» quando servirebbe davvero.)

9.4 Il costo onesto

Un modello uncensored:

- è utile per ricerca, sicurezza informatica difensiva, medicina, narrativa, privacy;
- ma perde sensibilità: può generare contenuti dannosi, dati personali, istruzioni pericolose;
- peggiora un po' su istruzioni complesse e sul ragionamento (il rifiuto è spesso l'ultimo strato di una catena di controllo).

Regola pratica: si usa in locale, per lavoro legittimo, e non si mette dietro un'API pubblica senza filtri aggiuntivi. Un base model grezzo è già di fatto «senza censura»: è il post-training che lo rende un assistente.

10 · La nuova generazione: Qwen 3.8 e DeepSeek V4.1

10 · La nuova generazione: Qwen 3.8 contro DeepSeek V4.1

Due filosofie diverse, sei problemi comuni da risolvere

QWEN 3.8 (Alibaba)	DEEPSEEK V4.1-FLASH
Qwen3.8-Max: 2.4 T parametri, pesi aperti · Qwen3.8-Flash-Next: 125B + 51B, 6B attivi	MoE multimodale: 552B backbone + 196B Engram · 8B attivi in prefill, 16B in decode
► Attenzione ibrida GDN + QSA Gated DeltaNet comprime la storia; Qwen Sparse Attention sceglie a blocchi cosa guardare → contesto fino a 1M token senza esplodere	► Architettura CED 40 strati = 20 di encoder + 20 di decoder; la KV del decoder è proiettata dall'encoder → in prefill si attivano solo 8B parametri
► Gated Residual (4 rami) il flusso residuo si allarga e un «cancello» decide quanto passa: più stabile da addestrare	► CSA2 + KV in FP4 attenzione sparsa compressa con riutilizzo della cache tra strati: 890 byte di KV per token = 1/4 del modello precedente
► N-gram Embedding tabella di embedding n-grammi tenuta in RAM dell'host, non sulla GPU: capacità extra quasi gratis	► SWA Bounded Replay ricostruisce la cache dell'attenzione locale rigiocando pochi token → -8x di cache su disco
► Optimizer Muon insieme ad AdamW: aggiornamenti più «ortogonali», training più stabile	► Engram memoria condizionale da 196B parametri consultata a «enciclopedia» solo quando serve
► Controllo del ragionamento reasoning_effort: 1 riga di configurazione = 1 secondo di riflessione o 1 minuto	► Multimodale nativo encoder visivo DeepSeek-ViT addestrato da zero; immagini dentro il pre-training, non aggiunte dopo
► Preserve thinking il modello RICORDA il ragionamento dei turni precedenti invece di riscriverlo	► Post-training SFT → RL → distillazione on-policy: tutta l'innovazione è nel generatore automatico di compiti

Cosa dicono insieme sullo stato dell'arte (settembre 2026)

① MoE sparso ovunque: modelli enormi ma con pochi miliardi attivi per token · ② contesto da 1M token con cache KV compressa · ③ multimodalità nativa dal pre-training
④ attenzione che non è più «quadratica pura» (GDN, QSA, CSA2) · ⑤ post-training guidato da RL su compiti verificabili · ⑥ sforzo di ragionamento regolabile a runtime

motori-2026

(Fotografia di settembre 2026.)

10.1 Qwen 3.8 (Alibaba)

Per la prima volta Alibaba rilascia i pesi di un modello di classe «Max»:

- Qwen3.8-Max — modello di punta con 2,4 trilioni di parametri, pesi aperti.
- Qwen3.8-Flash-Next — 125B parametri + 51B di embedding n-gram, 6B attivi per token (MoE multimodale). È l'anteprima dell'architettura Qwen4.
- Qwen3.8-27B — dense, il cavallo di battaglia per l'uso locale.

Novità tecniche:

Componente	A cosa serve
GDN + QSA (attenzione ibrida)	Gated DeltaNet comprime la storia, Qwen Sparse Attention sceglie a blocchi cosa guardare → contesto fino a 1M token senza costi proibitivi
Gated Residual a 4 rami	Il flusso residuo si allarga e un «cancello» decide quanto passa: addestramento più stabile in profondità
N-gram Embedding (51B)	Tabella di embedding tenuta in RAM dell'host, non sulla GPU: capacità extra quasi gratis
Optimizer Muon	Insieme ad AdamW: aggiornamenti più «ortogonali», training più stabile
reasoning_effort	Una riga di configurazione = un secondo di riflessione o un minuto
Preserve thinking	Il modello ricorda il ragionamento dei turni precedenti invece di riscriverlo

10.2 DeepSeek V4.1-Flash

Filosofia opposta, stesso obiettivo: fare tanto con poco. È il modello più piccolo della nuova famiglia V4.1, multimodale nativo.

- 552B parametri di backbone + 196B di memoria Engram.
- 8B attivi durante il prefill (lettura dell'input), 16B durante il decode (generazione).
- Contesto fino a 1 milione di token.
- Addestrato su 45 trilioni di token multimodali (testo + immagini dal pre-training, non aggiunte dopo).

Novità tecniche:

Componente	A cosa serve
CED (Causal Encoder-Decoder)	40 strati = 20 encoder + 20 decoder: la KV del decoder è proiettata dalle uscite dell'encoder, così la maggior parte dei token di prompt salta il calcolo completo. Quasi dimezza il prefill
CSA2 (Compressed Sparse Attention 2)	Ogni strato condivide la KV globale con altri e riusa gli indici Top-K: tre modalità statiche — Full, Reindex, Reuse
KV cache in FP4	890 byte di KV per token — circa 1/4 rispetto a V4-Flash. È il numero che rende sostenibile il contesto da 1M
SWA Bounded Replay	Ricostruisce la KV dell'attenzione locale rigiocando solo la finestra recente → cache persistente a 1/8, niente da scrivere su SSD

Componente	A cosa serve
Engram	Memoria condizionale da 196B parametri consultata «a enciclopedia» solo quando serve
DSpark	Decodifica speculativa semi-autoregressiva: una bozza veloce + verifica con controllo di confidenza
Hierarchical Sparse Indexer	Limita le ricerche degli strati profondi a un insieme di candidati già selezionato: costo degli indici che non cresce con il contesto
MoE	1 esperto condiviso + 384 instradati, 6 attivati per token
DeepSeek-ViT	Encoder visivo addestrato da zero (2D-RoPE, unshuffle 3×3)

10.3 Confronto

	Qwen 3.8 (Flash-Next / Max)	DeepSeek V4.1-Flash
Pesi	aperti, anche per il modello Max	aperti (Flash)
Scala	125B attivi 6B / 2,4T	552B + 196B Engram, attivi 8B/16B
Attenzione	GDN + Qwen Sparse Attention	CSA2 + SWA
Contesto	1M token	1M token
Multimodale	sì	sì, nativo dal pre-training
Idea chiave	capacità extra in RAM dell'host (n-gram)	compressione estrema della KV cache
Ragionamento	reasoning_effort regolabile	RL su vasta scala, sforzo configurabile

10.4 Cosa dicono insieme sullo stato dell'arte

- 1 MoE sparso ovunque. Modelli enormi, pochi miliardi attivi per token.
- 2 Contesto da 1M token reso possibile solo dalla compressione della KV cache.
- 3 Multimodalità nativa: le immagini entrano nel pre-training, non si aggiungono dopo.
- 4 Attenzione non più quadratica pura: GDN, QSA, CSA2, SWA.
- 5 Post-training guidato da RL su compiti verificabili, con generazione automatica dei compiti.
- 6 Sforzo di ragionamento regolabile a runtime: lo stesso modello risponde subito o «pensa» dieci volte tanto.

La tendenza generale è chiara: non si vince più solo ingrandendo il modello, si vince facendogli fare il lavoro con meno calcolo. È per questo che il 2026 è l'anno in cui i modelli buoni sono diventati

piccoli abbastanza da girare in casa.

11 · Glossario essenziale

Termine	Significato in una riga
Token	Pezzo di parola; l'unità di lavoro dell'LLM
Embedding	Vettore numerico che rappresenta un token
Transformer	Architettura basata sull'attenzione (2017)
Attention	Meccanismo che pesa quanto ogni token conta per un altro
Encoder / Decoder	Percorso input→rappresentazione / rappresentazione→output
Causale	Maschera che impedisce di guardare i token futuri
Pre-training	Addestramento «indovina il prossimo token» su testi grezzi
SFT	Fine-tuning su coppie domanda/risposta scritte da umani
RLHF / DPO / GRPO	Metodi di rinforzo basati su preferenze o verificatori
Loss	Numero che misura l'errore; deve scendere
Backpropagation	Calcolo dei gradienti all'indietro, strato per strato
MoE	Più esperti + router: solo pochi si accendono per token
Quantizzazione	Meno bit per peso = meno memoria
GGUF	Formato file per llama.cpp: pesi + tokenizer + template
imatrix	Mappa di importanza dei pesi, migliora la quantizzazione
Distillazione	Un modello piccolo impara da uno grande (soft labels)
Dark knowledge	Informazione contenuta nelle probabilità «sbagliate» dell'insegnante
Abliteration	Rimozione della direzione del rifiuto dai pesi
KV cache	Memoria dei token già elaborati; il collo di bottiglia del contesto lungo
LoRA / PEFT	Addestrare pochi parametri aggiuntivi invece di tutti

12 · Laboratorio: provare tutto in casa

Nessuno di questi passi richiede un cluster.

```
# 1) installare llama.cpp (CPU, funziona su qualsiasi cosa)
git clone https://github.com/ggml-org/llama.cpp && cd llama.cpp
cmake -B build && cmake --build build --config Release -j

# 2) scaricare un modello già quantizzato (la via più semplice)
huggingface-cli download Qwen/Qwen3.8-27B-GGUF \
  qwen3.8-27b-Q4_K_M.gguf --local-dir ./models

# 3) chattare sul proprio computer, senza internet
./build/bin/llama-cli -m ./models/qwen3.8-27b-Q4_K_M.gguf \
  -p "Spiegami cos'è un MoE come se avessi 15 anni" -n 300

# 4) modalità server con API compatibile OpenAI
./build/bin/llama-server -m ./models/qwen3.8-27b-Q4_K_M.gguf \
  --host 0.0.0.0 --port 8080 -c 32768
```

E per creare una quantizzazione propria da zero, i passi sono quelli di §7.3.

Percorso di apprendimento consigliato:

- 1 Far girare llama-cli con un modello q4 → capire il risultato finale.
- 2 Convertire e quantizzare un modello piccolo (0,5–3B) → capire il formato GGUF.
- 3 Fare un fine-tuning LoRA su un 8B con Unsloth → capire SFT.
- 4 Provare un DPO su 200 coppie di preferenze → capire il rinforzo.
- 5 Provare l'abliteration su un modello piccolo → capire l'algebra dei comportamenti.
- 6 Distillare un modello da 3B da uno da 30B con TRL → capire la distillazione.

Fatti questi sei passi, il funzionamento di un LLM moderno non ha più zone d'ombra.

Fine del documento.