Deep Learning

Treinamento de LLMs: do Pré-treinamento ao Alinhamento

Lucas Silveira Kupssinskü

Machine Learning Theory and Applications Laboratory
PUCRS

agosto de 2026

Visão Geral

1. Pré-treinamento
2. Fine-tuning Supervisionado (SFT)
3. RLHF com PPO
4. Direct Preference Optimization (DPO)
5. Outros Métodos de Alinhamento
6. Otimizações de Treinamento
7. Ajuste Eficiente em Parâmetros (PEFT)

Visão Geral

Objetivo: *Next-Token Prediction*

O pré-treinamento de LLMs se baseia em **modelagem de linguagem causal**: dado um prefixo de tokens, prever o próximo.

Loss Function: Cross Entropy

$$\mathcal{L}_{\text{LM}}(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log P_\theta(x_t \mid x_1, \ldots, x_{t-1})$$

Contexto	→	Alvo
`O gato`	→	`sentou`
`O gato sentou`	→	`no`
`O gato sentou no`	→	`tapete`

Uma única sequência de comprimento T gera $T-1$ pares (contexto, alvo).

Propriedade chave: o corpus de texto é a própria supervisão: não há necessidade de anotação humana.

Datasets de Pré-treinamento

Grandes corpora públicos

- **Common Crawl**: rastreamento da web, bruto e ruidoso
- **C4** (*Colossal Clean Crawled Corpus*): versão filtrada do CC
- **The Pile**: 825 GB de texto diversificado (livros, código, arXiv, GitHub)
- **ROOTS** (BigScience): corpus multilíngue (46 idiomas, 1.6 TB)
- **FineWeb** (HuggingFace): 15 T de tokens filtrados da web

Etapas de processamento

1. **Filtragem**: remoção de conteúdo ruim e spam (heurísticas + classificadores)
2. **Deduplicação**: remoção de documentos duplicados ou quase-duplicados
3. **Tokenização**: conversão para tokens
4. **Shuffling**: evitar correlações temporais no gradiente

Regra geral: qualidade > quantidade — corpus menor e limpo supera corpus bruto.

Tokenização e Vocabulário

A tokenização converte texto bruto em sequências de *tokens*.

Tamanho do vocabulário e impacto

- Vocabulário menor $\Rightarrow$ sequências mais longas, mais computação
- Vocabulário maior $\Rightarrow$ matriz de embedding maior, cobertura melhor
- Modelos modernos: 32 k – 200 k tokens (BPE ou Unigram)

Comparação de vocabulários

Modelo	Vocab	Algoritmo
GPT-2	50 257	BPE
LLaMA-2	32 000	BPE (sentencepiece)
LLaMA-3	128 256	BPE
BERT	30 522	WordPiece
T5	32 100	Unigram

O *tokenizer* é **treinado antes** do modelo, no mesmo corpus, e permanece fixo durante o treinamento.

Pré-treinamento: *Warm-up* do LR

1. *Warm-up* do LR

- LR cresce linearmente de 0 até $\eta_{\max}$
- Dura tipicamente 1–2% do treino total
- Evita explosão de gradiente no início e estabiliza a norma dos parâmetros
- **Por quê:** os estimadores de momento do Adam começam em zero, sem *warm-up* as primeiras atualizações seriam enormes

Otimizador padrão: AdamW com β_1=0.9, β_2=0.95, ϵ=10^{-8}, *weight decay* λ=0.1, *gradient clipping* em 1.0.

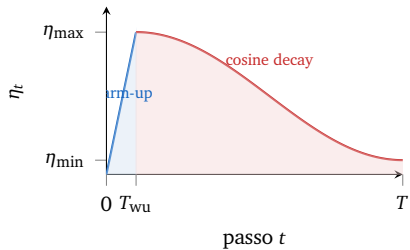
Pré-treinamento: *Cosine Decay* do LR

2. *Cosine Decay*

- LR decai de $\eta_{\max}$ a $\eta_{\min}$

$$\eta_t = \eta_{\min} + \frac{\eta_{\max} - \eta_{\min}}{2}\left(1 + \cos\frac{\pi t}{T}\right)$$

- Transição suave evita instabilidades



Pré-treinamento: *Batch Ramp-up*

3. *Batch Ramp-up*

- *Batch size* cresce ao longo do treino
- Início: mini-batches pequenos para convergência rápida
- Fim: batches grandes para melhor generalização

Visão Geral

1. Pré-treinamento
2. **Fine-tuning Supervisionado (SFT)**
3. RLHF com PPO
4. Direct Preference Optimization (DPO)
5. Outros Métodos de Alinhamento
6. Otimizações de Treinamento
7. Ajuste Eficiente em Parâmetros (PEFT)

SFT: Ajuste para Seguir Instruções

Após o pré-treinamento, o modelo sabe "completar texto", mas não necessariamente **seguir instruções** de forma útil.

O *Supervised Fine-Tuning* (SFT), ou *Instruction Tuning*, ajusta o modelo em pares `(instrução, resposta)` de alta qualidade.

Função de perda do SFT

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{t \in \mathcal{R}} \log P_\theta(x_t \mid x_{<t})$$

onde $\mathcal{R}$ são **apenas os tokens da resposta** (não do prompt/instrução).

É a **mesma perda do pré-treino**, mas mascarada: calculá-la também no prompt faria o modelo memorizar instruções em vez de aprender a responder.

Exemplo de amostra de SFT

[INSTRUÇÃO: mascarado]
Explique o que é retropropagação.

[RESPOSTA: perda calculada aqui]
Retropropagação é o algoritmo que calcula gradientes…

Datasets para Instruction Tuning

Datasets clássicos

- **FLAN** [2]: 1.8k tarefas NLP; diversidade > volume
- **ShareGPT**: conversas reais de usuários do ChatGPT; qualidade variável
- **LIMA** [11]: 1.000 exemplos **muito bem curados** ≈ 52k do Alpaca
- **OpenHermes**: mistura de ShareGPT, GPT-4 e outros; qualidade alta e consistente

Quantidade vs. Qualidade

Dataset	Exemplos	Qualidade
Alpaca (GPT-3.5)	52 000	Baixa–Média
ShareGPT	90 000	Média
LIMA	1 000	Alta
OpenHermes-2.5	1 000 000	Alta

Lição do LIMA: "1000 exemplos cuidadosamente selecionados superam 52000 gerados automaticamente."

Chat Templates e Tokens Especiais

Modelos de chat usam **tokens especiais** para delimitar turnos da conversa, permitindo multi-turno e controle de papéis. Tokens comuns

- `<|im_start|>`, `<|im_end|>` (ChatML / Qwen)
- `[INST]`, `[/INST]` (LLaMA-2 chat)
- `<|start_header_id|>` (LLaMA-3)
- `<s>` (BOS), `</s>` (EOS)

O template define **onde calcular a perda**: apenas nos tokens gerados pelo assistente.

Exemplo: formato ChatML

```
<|im_start|>system Você é um assistente útil. <|im_end|>
<|im_start|>user Explique redes neurais. <|im_end|>
<|im_start|>assistant [loss calculada aqui] Redes neurais são…<|im_end|>
```

SFT vs. Pré-treinamento: Diferenças de Otimização

Pré-treinamento

- LR alto: 10^{-4} – 3×10^{-4}
- Centenas de bilhões de tokens
- *Batch size* muito grande (milhões de tokens)
- *Weight decay*: 0.1
- Foco: **absorver conhecimento**

Fine-tuning (SFT)

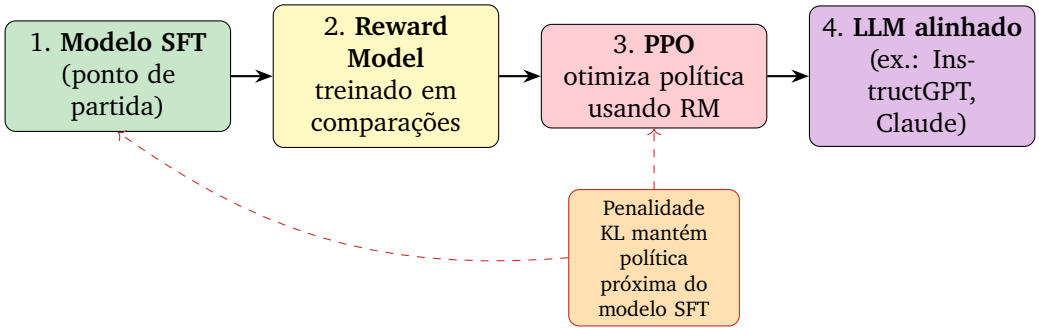
- LR baixo: 10^{-6} – 2×10^{-5}
- Milhares a milhões de exemplos
- *Batch size* menor (128–512 sequências)
- Poucas épocas (1–3) para evitar *overfitting*
- Foco: **adaptar comportamento**

Risco de *Catastrophic Forgetting*

LR alto ou muitas épocas podem fazer o modelo "esquecer" conhecimento adquirido no pré-treinamento. Use LR baixo e monitore *benchmarks* gerais ao longo do SFT.

Visão Geral

Pipeline Completo do RLHF [9]



Etapa 1: amostrar pares de respostas do modelo SFT

Etapa 2: humanos escolhem a resposta preferida

Etapa 3: treinar o RM com *Bradley-Terry*

Etapa 4: usar PPO para otimizar a política, maximizando recompensa do RM com penalidade KL em relação ao SFT

O passo PPO mantém **4 modelos** em memória (política, referência, *reward model*, *value function*) $\rightarrow$ $\sim$4$\times$ a VRAM do modelo base.

Dados de Preferência: Pares Escolhido/Rejeitado

Formato dos dados para RLHF, DPO e variantes

Prompt x: *"Explique o que é aprendizado de máquina de forma simples."*

Resposta escolhida y_w (✓)

"Aprendizado de máquina é um método pelo qual computadores aprendem padrões a partir de dados, sem serem explicitamente programados para cada tarefa."

Tripla: $(x,\ y_w,\ y_l)$ — usada por RLHF, DPO, GRPO

Resposta rejeitada y_l (×)

"ML é quando você faz o computador aprender coisas. É tipo uma coisa de dados. Basicamente é inteligência artificial mas diferente."

HH-RLHF [1], UltraFeedback (64k), OpenHermes

Treinamento do *Reward Model*

O *Reward Model* (RM) é treinado para **prever preferência humana** entre pares de respostas para o mesmo prompt.

Coleta de dados: para cada prompt x, amostram-se duas respostas (y_w, y_l) onde y_w é a preferida pelos anotadores.

Função de perda do RM (Bradley-Terry)

$$\mathcal{L}_{\text{RM}} = -\mathbb{E}_{(x,y_w,y_l)}\left[\log \sigma\big(r_\phi(x,y_w) - r_\phi(x,y_l)\big)\right]$$

onde $r_\phi(x,y)$ é o *scalar reward* predito.

É a verossimilhança de um modelo de *ranking*: equivale a **classificação binária** com *logit* $= r_\phi(y_w) - r_\phi(y_l)$.

Datasets de comparações humanas

Dataset	Pares	Fonte
HH-RLHF (Anthropic)	170k	Humanos
WebGPT Comparisons	19k	Humanos
SHP	385k	Reddit
UltraFeedback	64k	GPT-4

O RM geralmente parte do **mesmo modelo SFT** com uma cabeça de regressão escalar adicionada.

Exemplo Numérico: *Reward Model*

Prompt x**:** "Explique o que é uma rede neural." **Humanos preferem** y_w (detalhada) sobre y_l (vaga).

Cenário A: RM bem treinado

Resposta	$r_\phi(x, y)$
y_w (detalhada)	+2.1
y_l (vaga)	−0.8

$$\Delta r = (+2.1) - (-0.8) = \mathbf{+2.9}$$
$$\sigma(\Delta r) \approx 0.948$$
$$\mathcal{L}_{\text{RM}} = -\log(0.948) \approx \mathbf{0.053}$$

✓ RM concorda com o rótulo humano

Cenário B: RM mal treinado

Resposta	$r_\phi(x, y)$
y_w (detalhada)	−0.8
y_l (vaga)	+2.1

$$\Delta r = (-0.8) - (+2.1) = \mathbf{-2.9}$$
$$\sigma(\Delta r) \approx 0.052$$
$$\mathcal{L}_{\text{RM}} = -\log(0.052) \approx \mathbf{2.96}$$

× RM discorda; gradiente grande

Função Objetivo do PPO com Penalidade KL

O PPO otimiza a política π_θ para maximizar recompensa enquanto permanece próximo da política SFT de referência π_{ref}.

Objetivo RLHF-PPO

$$J(\theta) = \mathbb{E}_{x,y\sim\pi_\theta}\left[r_\phi(x,y) - \beta\,\mathrm{KL}(\pi_\theta\|\pi_{\text{ref}})\right]$$

Na prática a KL vira um **bônus por token**: subtrai-se $\beta \log \frac{\pi_\theta(t)}{\pi_{\text{ref}}(t)}$ do *reward* de cada token.

Clip Surrogate do PPO

$$L^{\text{CLIP}} = \mathbb{E}_t[\min(r_t A_t,\ \text{clip}(r_t, 1-\varepsilon, 1+\varepsilon)A_t)]$$

onde $r_t = \pi_\theta/\pi_{\text{old}}$ e A_t é a vantagem.

Papel do coeficiente KL β

β pequeno	Liberdade para maximizar reward; risco de *reward hacking*
β grande	Permanece próximo do SFT; pode não melhorar o alinhamento

Instabilidades típicas:

- *Reward hacking*: modelo "engana" o RM
- *KL explosion*: divergência excessiva da política

Visão Geral

1. Pré-treinamento
2. Fine-tuning Supervisionado (SFT)
3. RLHF com PPO
4. **Direct Preference Optimization (DPO)**
5. Outros Métodos de Alinhamento
6. Otimizações de Treinamento
7. Ajuste Eficiente em Parâmetros (PEFT)

DPO: Motivação

Problema com RLHF-PPO:

- Requer treinamento de um *Reward Model* separado
- Otimização RL é instável e sensível a hiperparâmetros
- Precisa manter 4 modelos em memória simultaneamente (política, referência, RM, *value function*)

Ideia do DPO (Rafailov et al., 2023): *eliminar o RM, reparametrizando o problema diretamente na política.*

O DPO usa a solução ótima do RLHF para dispensar o RM (derivação a seguir).

Comparação de infraestrutura

Componente	RLHF	DPO
Reward Model	✓	×
Value Function	✓	×
Política de referência	✓	✓
Geração online	✓	×
Dados de comparação	✓	✓

Online = **on-policy** (amostras frescas); DPO é **off-policy** (pares pré-coletados).

Derivação da Função de Perda do DPO [10]

Solução ótima do RLHF (restrição KL): $\pi^*(y|x) \propto \pi_{\text{ref}}(y|x)\exp(r^*(x,y)/\beta)$, com normalizador $Z(x)$ (*função de partição*). Isolando r^* e substituindo no Bradley-Terry, $Z(x)$ **cancela** na diferença:

Perda DPO

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x,y_w,y_l)}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right]$$

Interpretação intuitiva

A perda aumenta $\pi_\theta(y_w|x)$ e diminui $\pi_\theta(y_l|x)$, **ponderado pela mudança relativa em relação à referência**. Sem Reward Model explícito — o modelo de linguagem é o reward model.

Gradiente implícito

O DPO aumenta a probabilidade relativa de y_w mais agressivamente quando o RM implícito **subestima** y_w, e vice-versa.

Datasets: UltraFeedback (64k), Anthropic-HH (170k), TL;DR (93k) — formato `(prompt, chosen, rejected)`.

Visão Geral

GRPO: *Group Relative Policy Optimization* [5]

Proposto no DeepSeek-R1 (DeepSeek-AI, 2025), o GRPO elimina a *value function* do PPO substituindo-a por uma **estimativa de baseline por grupo**.

$$J_{\text{GRPO}} = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\min\left(r_i\hat{A}_i,\ \text{clip}(r_i, 1-\varepsilon, 1+\varepsilon)\,\hat{A}_i\right)\right]$$

onde $\hat{A}_i = \dfrac{r_i - \text{mean}(\{r_j\})}{\text{std}(\{r_j\})}$ e G é o tamanho do grupo de amostras.

- Sem *value function*: menos parâmetros, menos memória
- Estimativa de vantagem simples e estável (z-score; sem variância no grupo, $\hat{A}_i$=0 e não há gradiente)
- Funciona bem com *reward* esparso (verificação matemática)
- Permite raciocínio de *chain-of-thought* emergente

Resultado: DeepSeek-R1 alcança desempenho competitivo com OpenAI o1 em AIME e MATH usando GRPO com verificação automática de respostas.

Exemplo Numérico: GRPO

Problema: "Quanto é 12×7?" (resposta correta: 84) Grupo de $G = 4$ respostas amostradas.

i	Resposta y_i	r_i	$\hat{A}_i$
1	"84" (correto)	1	+1.0
2	"82" (errado)	0	−1.0
3	"84" (correto)	1	+1.0
4	"76" (errado)	0	−1.0

$$\bar{r} = \frac{1+0+1+0}{4} = 0.5, \qquad \sigma_r = 0.5$$

$$\hat{A}_i = \frac{r_i - 0.5}{0.5}$$

Efeito na política

- $\hat{A}_i > 0$: resposta **melhor que a média** $\Rightarrow$ política reforçada
- $\hat{A}_i < 0$: resposta **pior que a média** $\Rightarrow$ política penalizada

Sem *value function*: o baseline é a **média das recompensas do grupo**. Com recompensas binárias e metade do grupo acertando, $\hat{A}_i \in \{-1, +1\}$.

SimPO: *Reward* Sem Modelo de Referência [8]

SimPO

Motivação: DPO depende do modelo de referência, o que pode ser ineficiente.

$$\mathcal{L}_{\text{SimPO}} = -\log \sigma\left(\frac{\beta}{|y_w|}\log \pi_\theta(y_w|x) - \frac{\beta}{|y_l|}\log \pi_\theta(y_l|x) - \gamma\right)$$

- Sem modelo de referência (π_{ref})
- *Length-normalized reward*: divide pelo comprimento para evitar viés
- Margem γ garante separação entre escolhida e rejeitada

Tabela Comparativa dos Métodos de Alinhamento

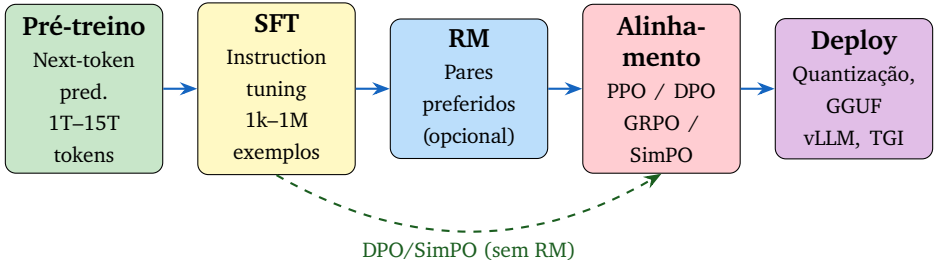
Método	Precisa de RM	Precisa de pares	*Online*	Sem ref.	Caso de uso típico
RLHF-PPO	✓	✓	✓	×	Alinhamento geral (InstructGPT)
DPO	×	✓	×	×	Alinhamento simples, eficiente
IPO	×	✓	×	×	DPO mais robusto a ruído
SimPO	×	✓	×	✓	Sem dependência de referência
GRPO	✓*	×	✓	×	Raciocínio matemático/código

* GRPO usa reward verificável automaticamente (ex.: checar resposta matemática), não um RM neural separado.

Tendência atual

Para a maioria dos casos, **SFT + DPO** é o pipeline padrão. PPO é reservado para cenários onde a geração *online* traz ganhos claros. GRPO cresce para tarefas de raciocínio com reward verificável.

Resumo: Pipeline Completo de Treinamento de LLMs



Pré-treino

GPT-3, LLaMA, Falcon
Datasets: The Pile, C4
Flash Attention, Precisão Mista

SFT + Alinhamento

InstructGPT, ChatGPT
LLaMA-Chat, Mistral-Instruct
Qwen-Chat, DeepSeek-R1

Pesquisa ativa

DPO e variantes
GRPO para raciocínio
Constitutional AI (Anthropic)

Visão Geral

Precisão Mista: Por que Usar Menos Bits?

Treinar em **precisão total** (fp32) consome 16 bytes por parâmetro (pesos + gradientes + estados Adam).

Motivação

Reduzir a precisão de fp32 para fp16/bf16 durante as operações principais permite:

- **2× menos memória** para pesos e ativações
- **2–8× throughput** em Tensor Cores modernos
- Treinamento de modelos maiores com o mesmo hardware

Atenção: precisão insuficiente causa *overflow/underflow* e instabilidade.

Throughput de GEMM por precisão

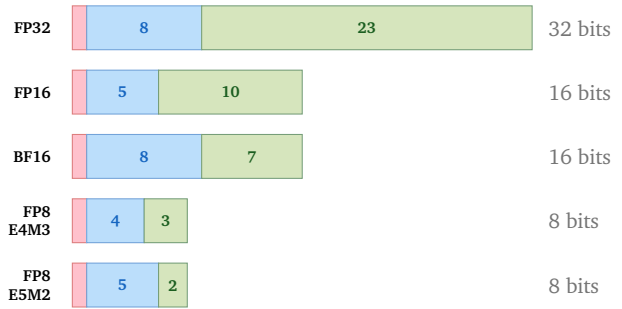
GPU	FP32	BF16/FP16
A100 SXM	312 TFLOPS	624 TFLOPS
H100 SXM	67 TFLOPS	1 979 TFLOPS
RTX 4090	83 TFLOPS	330 TFLOPS

GEMM = multiplicação de matrizes

Tensor Cores (GPUs Volta+) são unidades de hardware dedicadas à multiplicação de matrizes em fp16/bf16, daí o salto de throughput acima.

Representações de Ponto Flutuante em Deep Learning

Layout de bits



Impacto dos campos

- **Expoente**: **intervalo** dinâmico
- **Mantissa**: **precisão** entre valores

Alcance representável (valores positivos)

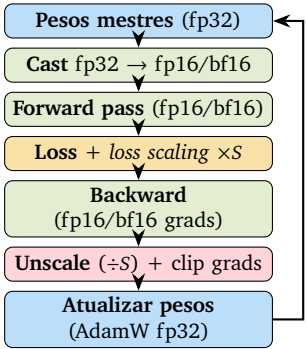
Formato	Mín. normal	Máx. finito
FP32	$\approx 1.18 \times 10^{-38}$	$\approx 3.4 \times 10^{38}$
FP16	$\approx 6.1 \times 10^{-5}$	65 504
BF16	$\approx 1.18 \times 10^{-38}$	$\approx 3.4 \times 10^{38}$
FP8 E4M3	$\approx 1.56 \times 10^{-2}$	448
FP8 E5M2	$\approx 6.1 \times 10^{-5}$	57 344

Comparação chave

BF16 = mesmo expoente do FP32 (8 bits), evita overflow de gradientes; **FP16** = mais mantissa (10 bits) mas range $\sim 10^{10}\times$ menor que FP32. **FP8 E5M2** (mais range): gradientes. **FP8 E4M3** (mais precisão): pesos/ativações.

Precisão Mista Automática (AMP)

Pesos mestres: cópia dos parâmetros em fp32 usada pelo otimizador. Forward/backward usam cópia em fp16/bf16 (rápido em *Tensor Cores*), mas o AdamW acumula na cópia mestre em fp32 para não descartar gradientes minúsculos.



Loss Scaling (necessário com FP16)

Gradientes em fp16 sofrem **underflow** (valores pequenos → zero). Solução: escalar a loss:

$$\tilde{\mathcal{L}} = S \cdot \mathcal{L}, \quad g_{\text{fp32}} = (\tilde{g}_{\text{fp16}})/S$$

S é ajustado dinamicamente: sobe se sem overflow, desce se overflow detectado.

BF16 dispensa *loss scaling*

Mesmo intervalo dinâmico do FP32 ⇒ sem risco de overflow em gradientes. **Preferir BF16** em GPUs Ampere+.

Flash Attention: O Gargalo de Memória [4]

O **mecanismo de atenção padrão** materializa a matriz $S = QK^\top \in \mathbb{R}^{N\times N}$ inteira na HBM:

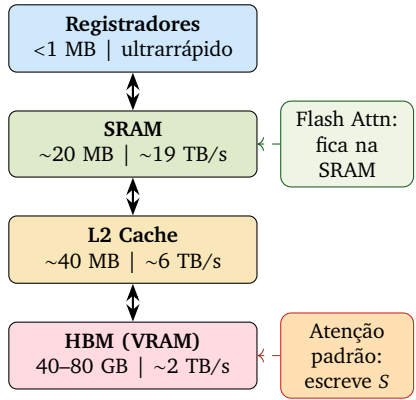
$$\text{Attn}(Q,K,V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

Problema de IO

Para $N = 4096$, fp16: $S \approx$ **32 MB/camada**. Com 32 camadas: **GBs de transfers HBM**, que é lenta.

O bottleneck **não é computação, é memória**: cada elemento de S deve ser escrito e relido da HBM pelo menos 2 vezes (forward + backward).

Hierarquia de memória da GPU



Flash Attention: Tiling na SRAM

Ideia central: dividir Q, K, V em **blocos** que cabem na SRAM e processar atenção bloco a bloco, sem materializar $S = QK^\top$ na HBM.

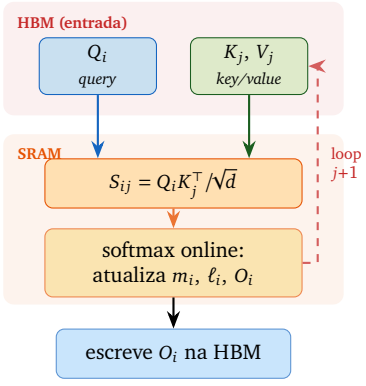
Online Softmax

O softmax requer ver todos os escores para normalizar. Solução: manter estatísticas acumuladas por bloco:

$$m_i^{\text{new}} = \max(m_i^{\text{old}}, \max_j s_{ij})$$

$$\ell_i^{\text{new}} = e^{m_i^{\text{old}} - m_i^{\text{new}}} \ell_i^{\text{old}} + \sum_j e^{s_{ij} - m_i^{\text{new}}}$$

Matematicamente equivalente ao softmax global.



Online Softmax: Exemplo Numérico

Scores: $s = [3, 1, 2, 4]$ em dois blocos: $B_1 = [3, 1]$, $B_2 = [2, 4]$

Processamento por blocos:

Bloco $B_1 = [3, 1]$

$$m^{(1)} = \max(3, 1) = 3$$

$$\ell^{(1)} = e^{3-3} + e^{1-3} = 1.000 + 0.135 = 1.135$$

Bloco $B_2 = [2, 4]$ $\quad (m^{(1)} = 3)$

$$m^{(2)} = \max(3, 4) = 4$$

$$\ell^{(2)} = \underbrace{e^{3-4}}_{\text{rescale}} \cdot \ell^{(1)} + e^{2-4} + e^{4-4}$$

$$= 0.368 \times 1.135 + 0.135 + 1.000$$

$$= \mathbf{1.553}$$

Softmax global (referência):

$$\ell = e^{3-4} + e^{1-4} + e^{2-4} + e^{4-4}$$

$$= 0.368 + 0.050 + 0.135 + 1.000$$

$$= \mathbf{1.553} \checkmark$$

Por que isso importa?

Sem subtração do máximo:

$$s = [1000, 1001] \Rightarrow e^{1000} = \infty$$

Online: $s - m = [-1, 0]$

$\Rightarrow e^{-1} + e^{0} = 1.368 \checkmark$ sem overflow

O fator $e^{m^{(1)} - m^{(2)}}$ rescala o acumulador anterior para o novo máximo global.

Flash Attention 2 e Versões Modernas [3]

Flash Attention 2 (Dao, 2023)

Melhorias sobre a v1:

- **Menos operações de rescale**: reduz FLOPs do softmax online em ~2×
- **Paralelismo na dimensão de sequência**: divide o trabalho tanto pelas heads quanto pelo comprimento
- **Causal masking eficiente**: pula a metade superior da matriz em geração autoregressiva
- **Resultado**: 2–4× mais rápido que FA1 em sequências ≥1k tokens

Flash Attention 3 (2024, H100)

Projetado para GPUs Hopper (H100):

- Usa **WGMMA** e **TMA**: primitivas de hardware do H100 para GEMMs e carregamentos assíncronos
- Pipeline: calcula GEMM enquanto carrega o próximo bloco em paralelo (sobreposição compute/IO)
- Suporte nativo a **FP8**: atenção em 8 bits
- ~1,5–2× mais rápido que FA2 no H100

Adoção: FA2 padrão em PyTorch 2.0+, vLLM e HuggingFace.

Visão Geral

Motivação: Custo do *Fine-tuning* Completo

Fazer *fine-tuning* completo de um LLM exige armazenar em GPU:

- **Pesos do modelo**: ~2 bytes/parâmetro (fp16)
- **Gradientes**: mesmo tamanho dos pesos
- **Estados do otimizador** (AdamW): ~12 bytes/parâmetro
- **Ativações**: proporcional ao *batch size*

Exemplo: modelo de 7B parâmetros

Pesos + gradientes + otimizador ≈ **112 GB**: fora do alcance de GPUs de consumo (RTX 4090: 24 GB).

Memória por parâmetro

Componente	fp32	fp16/bf16
Pesos	4 bytes	2 bytes
Gradientes	4 bytes	2 bytes
Adam m (1ª mom.)	4 bytes	—
Adam v (2ª mom.)	4 bytes	—
Total	**16 bytes**	**~16 bytes***

* estados Adam mantidos em fp32 mesmo com pesos em fp16

Solução PEFT: treinar apenas uma fração pequena dos parâmetros, mantendo o modelo base congelado.

LoRA: *Low-Rank Adaptation* [7]

Hu et al. (2022) decompõem a atualização de pesos como um produto de **baixo *rank***:

Decomposição LoRA

Para uma camada $W_0 \in \mathbb{R}^{d \times k}$ congelada, adicionamos um desvio treinável:

$$h = W_0 x + \frac{\alpha}{r} BAx$$

onde $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, $r \ll \min(d, k)$ e α é um fator de escala.

- A normal, B com **zeros**: ΔW=0 no início, o modelo parte do base
- Somente A e B são atualizados durante o treino
- Parâmetros treináveis: $r(d+k) \ll dk$

Configurações típicas

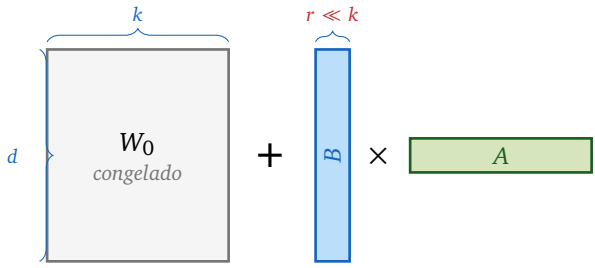
Hiperparâmetro	Valor típico	Notas
rank r	4–64	Menor = menos parâmetros
α	r–$2r$	Controla a escala
lora_dropout	0.05	Regularização
Camadas alvo	Q, V, ff	Projeções de atenção

Exemplo: LLaMA-7B com r=16, adaptando Q e V: apenas 0,17% dos parâmetros são treináveis, redução de memória de ~3×.

Adaptar também as projeções de saída (O, *gate*, *up*, *down*) melhora o desempenho ao custo de mais parâmetros.

LoRA: Decomposição de Baixo *Rank* — Visualização

Atualização de pesos: modelo congelado + adaptadores



Contagem de parâmetros

modelo 7B: d=k=4096, r=16

Matriz	Dim.	Params
W_0	$d \times k$	16.7M *(frozen)*
B	$d \times r$	65k
A	$r \times k$	65k
Total treinável		**130k**
fração		**≈0,8%**

Deploy: fundir $W_0 + \frac{\alpha}{r} BA$ antes de servir → zero overhead de inferência.

QLoRA: LoRA com Base Quantizada em 4 Bits [6]

Dettmers et al. (2023) combinam LoRA com quantização do modelo base, permitindo ajustar modelos de 65B em uma única GPU de 48 GB.

Componentes chave do QLoRA

1. **NF4** (*NormalFloat 4-bit*): minimiza o erro de quantização para pesos com distribuição normal (vs. INT4 uniforme)
2. **Dupla quantização**: quantizar também as constantes de quantização, economizando ≈0,37 bits/parâmetro extra
3. ***Paged Optimizers***: memória unificada CPU/GPU para absorver picos durante o treino

Comparação de memória (modelo 7B)

Método	VRAM (treino)	Precisão
Full FT (fp16)	~112 GB	fp16
LoRA (fp16)	~40 GB	fp16
QLoRA (NF4)	~10 GB	NF4 + fp16

Custo: pequena degradação de qualidade (≈0,1–0,2 pontos em benchmarks) em troca de acessibilidade em hardware de consumo.

Implementação: bibliotecas `bitsandbytes` + `peft` do HuggingFace.

Comparação dos Métodos PEFT

Método	Params. treináveis	VRAM (7B)	Desempenho	Complexidade	Caso de uso típico
Full FT	100%	~112 GB	Máximo	Baixa	Dados abundantes, hardware dedicado
LoRA	0,1–1%	~40 GB	Alto	Baixa	Ajuste geral, produção
QLoRA	0,1–1%	~10 GB	Alto*	Média	Hardware de consumo (GPU única)

* Pequena degradação vs. LoRA em fp16

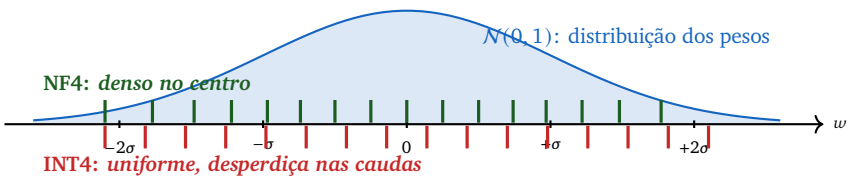
Recomendação prática

Para a maioria dos cenários, **QLoRA** é o padrão para ajuste em hardware limitado; **LoRA em bf16** quando há mais VRAM disponível. *Full fine-tuning* apenas quando há hardware dedicado e dados abundantes.

Quantização para PEFT: NF4 vs. INT4

NormalFloat 4-bit (NF4)

Tipo de dado de 4 bits projetado para pesos de redes neurais, que tipicamente seguem $\mathcal{N}(0, \sigma^2)$. Em vez de espaçamento uniforme (INT4), os 16 valores representáveis do NF4 são posicionados nos **quantis da distribuição normal**: $q_i = Q_F^{-1}\left(\frac{i}{15}\right)$, $i = 0, \ldots, 15$. Mais resolução perto de zero, onde há mais pesos; menos resolução nas caudas.



QLoRA: pesos base em NF4 (congelados); adaptadores LoRA em bf16 (treináveis). Menor erro de quantização que INT4 com os mesmos 4 bits.

Referências I

[1] Yuntao Bai et al. “Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback”. Em: arXiv preprint arXiv:2204.05862 (2022).

[2] Hyung Won Chung et al. “Scaling instruction-finetuned language models”. Em: JMLR 25 (2024), pp. 1–53.

[3] Tri Dao. “FlashAttention-2: Faster attention with better parallelism and work partitioning”. Em: ICLR. 2024.

[4] Tri Dao et al. “FlashAttention: Fast and memory-efficient exact attention with IO-awareness”. Em: NeurIPS. Vol. 35. 2022, pp. 16344–16359.

[5] DeepSeek-AI. “DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning”. Em: arXiv preprint arXiv:2501.12948 (2025).

[6] Tim Dettmers et al. “QLoRA: Efficient finetuning of quantized LLMs”. Em: NeurIPS. Vol. 36. 2023.

[7] Edward J Hu et al. “LoRA: Low-rank adaptation of large language models”. Em: ICLR. 2022.

[8] Yu Meng, Mengzhou Xia e Danqi Chen. “SimPO: Simple preference optimization with a reference-free reward”. Em: NeurIPS. Vol. 37. 2024.

[9] Long Ouyang et al. “Training language models to follow instructions with human feedback”. Em: NeurIPS. Vol. 35. 2022, pp. 27730–27744.

[10] Rafael Rafailov et al. “Direct preference optimization: Your language model is secretly a reward model”. Em: NeurIPS. Vol. 36. 2023.

[11] Chunting Zhou et al. “LIMA: Less is more for alignment”. Em: NeurIPS. Vol. 36. 2023.