Deep Learning

Modelos de Linguagem Decoder-Only

Lucas Silveira Kupssinskü

Machine Learning Theory and Applications Laboratory
PUCRS

agosto de 2026

Visão Geral

Visão Geral

O que Encoder e Decoder fazem?

Encoder: compreensão

- Processa **toda a entrada** de uma vez
- Atenção bidirecional
- Produz vetor de contexto rico
- Exemplos: BERT, RoBERTa

Decoder: geração

- Gera **um token por vez**
- Atenção causal (só vê o passado)
- Recebe contexto via cross-attention
- Exemplos: GPT, T5-decoder

Para LLMs, o encoder é desnecessário.

O decoder processa o *prompt* como parte da sequência.

Máscara Causal vs. Atenção Bidirecional

Bidirecional (BERT/Encoder):
cada posição vê *toda* a sequência.

	t_1	t_2	t_3	t_4
t_1	✓	✓	✓	✓
t_2	✓	✓	✓	✓
t_3	✓	✓	✓	✓
t_4	✓	✓	✓	✓

Causal (GPT/Decoder-Only):
posição i só vê $j \leq i$.

	t_1	t_2	t_3	t_4
t_1	✓	×	×	×
t_2	✓	✓	×	×
t_3	✓	✓	✓	×
t_4	✓	✓	✓	✓

Garante que t_4 não “veja o futuro” durante o treino.

Máscara Causal: Exemplo Numérico

Sequência de 3 tokens; scores $S = \dfrac{QK^\top}{\sqrt{d_k}}$, máscara M somada antes do softmax (por linha):

$$\underbrace{\begin{bmatrix} 2 & 1 & 3 \\ 0 & 2 & 1 \\ 1 & 0 & 2 \end{bmatrix}}_{S} + \underbrace{\begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}}_{M} = \begin{bmatrix} 2 & -\infty & -\infty \\ 0 & 2 & -\infty \\ 1 & 0 & 2 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} 1.00 & 0 & 0 \\ 0.12 & 0.88 & 0 \\ 0.24 & 0.09 & 0.67 \end{bmatrix}$$

- $e^{-\infty} = 0$: os pesos acima da diagonal **zeram** após o softmax
- Linha 2: $\dfrac{e^0}{e^0 + e^2} = 0.12$, $\dfrac{e^2}{e^0 + e^2} = 0.88$

Leitura

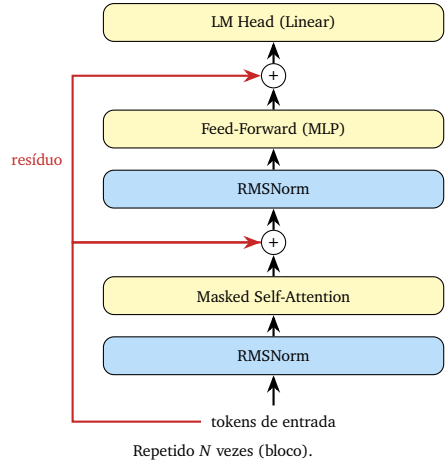
O token 1 só atende a si mesmo; o token 3 distribui atenção pelos três tokens do passado (inclusive ele próprio).

Do Encoder-Decoder para o Decoder-Only

Ideia central: treine apenas o decoder.

O modelo recebe o **prompt** e continua gerando autorregressivamente.

- Sem cross-attention
- Sem encoder separado
- Arquitetura mais simples



LM Head: do Vetor aos Logits

Após os N blocos, cada posição t carrega um vetor $h_t \in \mathbb{R}^{d_{\text{model}}}$.

Projeção no vocabulário

$$z = h_t W_{\text{LM}}, \qquad W_{\text{LM}} \in \mathbb{R}^{d_{\text{model}} \times |V|}$$

Cada logit é um **produto interno**, $z_v = h_t \cdot w_v$, onde w_v é a coluna de W_{LM} associada ao token v: mede o alinhamento entre o estado atual e cada token do vocabulário.

- **Weight tying**: muitos modelos reutilizam a matriz de embedding ($W_{\text{LM}} = E^\top$)

Exemplo: $d_{\text{model}} = 3$, $|V| = 5$, uma coluna por token:

$$\underbrace{\begin{bmatrix} 1.0 & 0.5 & -0.5 \end{bmatrix}}_{h_t} \underbrace{\begin{bmatrix} 2.0 & 1.5 & 0.8 & 0.3 & -0.3 \\ 1.6 & 1.0 & 0.2 & 0.6 & 0.0 \\ -0.4 & -0.2 & -0.2 & 0.2 & 0.4 \end{bmatrix}}_{W_{\text{LM}}}$$

$$= \underbrace{\begin{bmatrix} \texttt{rolha} & \texttt{roupa} & \texttt{rua} & \texttt{rainha} & \texttt{rei} \\ \mathbf{3.0} & 2.1 & 1.0 & 0.5 & -0.5 \end{bmatrix}}_{z}$$

Cálculo de $z_{\texttt{rolha}}$ (primeira coluna)

$1.0 \times 2.0 + 0.5 \times 1.6 + (-0.5) \times (-0.4) =$
$2.0 + 0.8 + 0.2 = 3.0$

LM Head: dos Logits à Distribuição

O softmax converte os logits em uma distribuição de probabilidade sobre o vocabulário:

Da pontuação à probabilidade

$$P(v \mid x_{\leq t}) = \text{softmax}(z)_v = \frac{e^{z_v}}{\sum_{u \in V} e^{z_u}}$$

- z: **logits**, um score por token do vocabulário
- A exponenciação amplia as diferenças entre scores; a soma normaliza o total para 1

Exemplo: *"O rato roeu a ___"*, logits do slide anterior:

Token	**Logit** z_v	$P(v \mid x_{\leq t})$
`rolha`	3.0	**0.60**
`roupa`	2.1	0.25
`rua`	1.0	0.08
`rainha`	0.5	0.05
`rei`	−0.5	0.02

Cálculo

$$P(\texttt{rolha}) = \frac{e^{3.0}}{e^{3.0} + \cdots + e^{-0.5}} = \frac{20.1}{33.2} \approx 0.60$$

A **estratégia de decodificação** (seção adiante) decide qual token emitir a partir dessa distribuição.

Treinamento: Todas as Posições em Paralelo

A máscara causal permite computar a perda de **todas as posições em um único passe**:

Posição	1	2	3	4	5
Entrada	`<s>`	`O`	`rato`	`roeu`	`a`
Alvo	`O`	`rato`	`roeu`	`a`	`rolha`
Perda	$-\log P(\texttt{O})$	$-\log P(\texttt{rato})$	$-\log P(\texttt{roeu})$	$-\log P(\texttt{a})$	$-\log P(\texttt{rolha})$

- Alvo = entrada deslocada uma posição (*shift-by-one*)
- Uma sequência de n tokens fornece n exemplos de treino simultâneos
- A máscara causal garante que a posição t não "veja" o próprio alvo

Por que o decoder-only venceu

Treino massivamente paralelo (ao contrário das RNNs, que processam token a token) com um objetivo simples, aplicável a qualquer texto bruto, sem anotação.

Visão Geral

GPT-1: Pré-Treino + Fine-Tuning[1]

Paradigma

Pré-treino generativo em texto bruto; depois, *fine-tuning* supervisionado para cada tarefa.

Objetivo de pré-treino: modelagem de linguagem causal

$$\mathcal{L} = -\sum_{t} \log P(x_t \mid x_1, \ldots, x_{t-1}; \Theta)$$

Arquitetura e treino:

- 117M parâmetros
- 12 camadas, $d = 768$, 12 cabeças
- Contexto: 512 tokens
- Treino: BooksCorpus (≈1B tokens)

Contribuição

Mostrou que pré-treino auto-supervisionado em texto melhora fortemente o fine-tuning.

[1] Alec Radford et al. Improving Language Understanding by Generative Pre-Training. Rel. técn. OpenAI, 2018. URL: https://openai.com/blog/language-unsupervised

GPT-2: Multitarefa Zero-Shot, sem Fine-Tuning[1]

Paradigma

Zero-shot: um LM suficientemente grande realiza tarefas **sem fine-tuning** e sem exemplos.

"Language models are unsupervised multitask learners"

A tarefa é induzida pelo próprio prompt: por exemplo, `TL;DR:` ao final de um texto induz sumarização.

Arquitetura e treino:

- 1.5B parâmetros (XL; versões de 117M a 1.5B)
- 48 camadas, $d = 1600$, 25 cabeças; Pre-Norm
- Contexto: 1024 tokens
- Treino: WebText ($\approx$10B tokens de texto web filtrado)

Contribuição

Um único LM realiza múltiplas tarefas zero-shot.

[1] Alec Radford et al. Language Models are Unsupervised Multitask Learners. Rel. técn. OpenAI, 2019. URL: https://openai.com/blog/better-language-models

GPT-3: Few-Shot In-Context Learning[1]

Paradigma

Few-shot: exemplos fornecidos *no contexto*, sem atualização de pesos.

Modos de uso em contexto:

- **Zero-shot**: só a instrução
- **One-shot**: 1 exemplo no prompt
- **Few-shot**: K exemplos no prompt

Arquitetura e treino:

- 175B parâmetros
- 96 camadas, $d = 12288$, 96 cabeças
- Contexto: 2048 tokens
- Treino: 300B tokens (Common Crawl filtrado, livros, Wikipedia)

Contribuição

Capacidades emergentes com escala (aritmética, tradução, raciocínio); o fine-tuning por tarefa dá lugar à engenharia de *prompts*.

[1] Tom B. Brown et al. "Language Models are Few-Shot Learners". Em: NeurIPS. vol. 33. 2020, pp. 1877–1901

Visão Geral

Bloco Transformer Decoder-Only em Detalhe

O bloco básico repete-se N vezes:

1. **Normalização** (Pre-Norm)
2. **Masked Self-Attention**
3. Conexão residual: $x \mathrel{+}= \text{Attn}(\cdot)$
4. **Normalização**
5. **Feed-Forward** (MLP)
6. Conexão residual: $x \mathrel{+}= \text{FFN}(\cdot)$

Sem cross-attention. Toda informação está nas n posições anteriores da sequência.

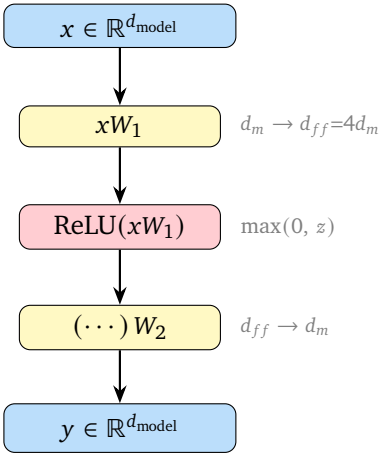
O que mudou desde 2017?

- ReLU-FFN → **SwiGLU**
- LayerNorm → **RMSNorm**
- Post-Norm → **Pre-Norm**
- Codificação posicional: RoPE

As três primeiras mudanças serão detalhadas a seguir.

FFN Padrão: Exemplo Numérico (ReLU)

$$\mathrm{FFN}(x) = \mathrm{ReLU}(xW_1)\,W_2 = \max(0,\, xW_1)\,W_2$$



Exemplo numérico

$d_{\text{model}} = 3, \quad d_{ff} = 4 \times 3 = 12$

$x = [\ 1.0,\ 0.5,\ -0.5\ ]$

$xW_1 = [\ 2.0,\ -1.0,\ 0.5,\ \ldots\ ]$ *(dim 12)*

$\mathrm{ReLU}(xW_1) = [\ 2.0,\ \mathbf{0.0},\ 0.5,\ \ldots\ ]$

$\Rightarrow$ ReLU **zera** o valor negativo -1.0!

$[\ \cdots\ ]W_2 \ \approx\ y = [\ 0.9,\ -0.1,\ 0.3\ ]$

SwiGLU: A FFN com *Gate* das LLMs mais atuais

FFN clássica usa **ReLU**:

ReLU-FFN (Vaswani 2017)

$$\text{FFN}(x) = \max(0,\, xW_1)\, W_2$$

Llama 2/3, PaLM, Mistral usam **SwiGLU** [18]:

SwiGLU-FFN

$$\text{FFN}(x) = \big(\text{SiLU}(xW_1) \odot xW_3\big)\, W_2$$

onde $\text{SiLU}(z) = z \cdot \sigma(z)$ (*Swish*).

O **portão** xW_3 filtra dinamicamente a informação.

Por que $\frac{8}{3}\, d_{\text{model}}$?

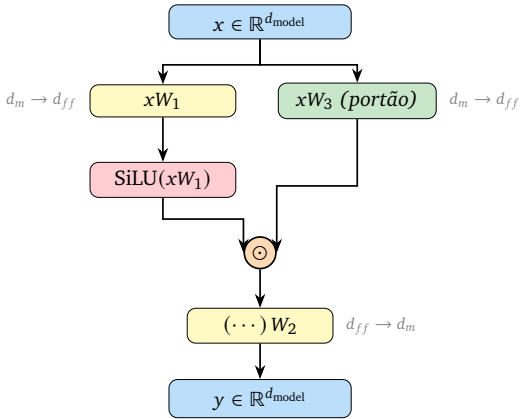
SwiGLU usa **3 matrizes** em vez de 2. Para manter o mesmo número de parâmetros:

$3 \times d_{\text{ff}} \cdot d = 2 \times 4d \cdot d$

$\Rightarrow d_{\text{ff}} = \frac{8}{3}\, d_{\text{model}}$

- SwiGLU supera ReLU e GELU empiricamente
- Motivação teórica: portão aprende *quais* dimensões ativar
- Alternativas: GeGLU, ReGLU

SwiGLU: Diagrama Computacional e Dimensões



Exemplo numérico

$d_{\text{model}} = 3, \quad d_{ff} = \frac{8}{3} \times 3 = 8$

$x = [\ 1.0,\ 0.5,\ -0.5\ ]$

$xW_1 = [\ 2.0,\ -1.0,\ 0.5,\ 1.2,\ \ldots\ ]$

$\text{SiLU}(xW_1) \approx [\ 1.76,\ -0.27,\ 0.31,\ 0.92,\ \ldots\ ]$

$xW_3 = [\ 0.8,\ \mathbf{0.0},\ 1.2,\ 0.5,\ \ldots\ ]$ *(portão)*

$[\ 1.76,\ -0.27,\ 0.31,\ 0.92,\ \ldots\ ]$
$\odot\ [\ 0.8,\ \mathbf{0.0},\ 1.2,\ 0.5,\ \ldots\ ]$
$= [\ 1.41,\ \mathbf{0.00},\ 0.37,\ 0.46,\ \ldots\ ]$
$\Rightarrow$ portão **0.0** suprime a dim. 2!

$[\cdots]W_2 \approx y = [\ 0.7,\ -0.2,\ 0.4\ ]$

Visão Geral

Por que Normalizar as Ativações?

Durante o treino de redes profundas:

- Distribuição das ativações muda a cada camada (*internal covariate shift*)
- Gradientes explodem ou desaparecem
- Taxas de aprendizado devem ser muito pequenas

Batch Norm: normaliza sobre o batch, problemático para sequências de tamanho variável.

Solução para Transformers

Layer Normalization [2]: normaliza sobre as *features* de cada token, sem dependência do batch.

Permite treino estável de Transformers muito profundos com learning rates maiores.

Layer Normalization [2]

Dado o vetor de ativações $\mathbf{x} \in \mathbb{R}^d$ de um token:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i, \qquad \sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\mathrm{LN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \hat{\mathbf{x}} + \boldsymbol{\beta}$$

$\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^d$: parâmetros aprendidos.

- Independente do tamanho do batch
- Funciona com sequências de tamanho variável
- Aplicada **por token** (ao longo da dimensão d)

Nota

A normalização ocorre ao longo da dimensão de features d de cada token, não ao longo da sequência.

RMS Norm: Simplificação do Layer Norm[1]

O re-centramento (subtração de μ) é desnecessário quando a invariância de escala é suficiente.

Normaliza apenas pela raiz do valor quadrático médio:

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$$

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} \odot \boldsymbol{\gamma}$$

Sem μ, **sem** $\boldsymbol{\beta}$, mais simples e rápido.

- Mais rápido: sem cálculo de média
- Mesma qualidade empírica que LayerNorm
- Menos parâmetros (sem $\boldsymbol{\beta}$)

Adotado por

Llama 1/2/3, Mistral, Mixtral, Gemma, Falcon, PaLM, …

Citado em [20] como padrão atual para LLMs eficientes.

[1] Biao Zhang e Rico Sennrich. "Root Mean Square Layer Normalization". Em: NeurIPS. vol. 32. 2019

Layer Norm, RMS Norm e Batch Norm: Exemplo Numérico

Lote de 2 tokens, cada um com $d = 4$ *features*:

$$X = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 9 & 7 & 5 & 3 \end{bmatrix} \qquad \rightarrow \text{LN/RMS: por linha (token)} \quad \downarrow \text{BN: por coluna (feature)}$$

LayerNorm — token 1 (eixo d)

$\mu = 4, \quad \sigma^2 = 5$

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2}} = [-1.34,\ -0.45,\ 0.45,\ 1.34]$$

média 0 e variância 1, usando só as features do token

RMSNorm — token 1 (eixo d)

$\text{RMS} = \sqrt{\frac{1}{4}\sum_i x_i^2} = 4.58$

$\mathbf{x}/\text{RMS} = [0.22,\ 0.66,\ 1.09,\ 1.53]$ (só reescala)

BatchNorm — feature 1 (lote)

coluna 1 = [1, 9]: $\mu = 5,\ \sigma^2 = 16$

$\hat{\mathbf{x}} = [-1,\ 1]$

usa estatísticas **do lote**, depende do batch

Conclusão

LN/RMS normalizam **por token**; BN **por feature**. Transformers usam LN/RMS por independerem do batch.

Layer Norm, RMS Norm e Batch Norm: Exemplo Numérico

Lote de 2 tokens, cada um com $d = 4$ *features*:

$$X = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 9 & 7 & 5 & 3 \end{bmatrix} \qquad \rightarrow \text{LN/RMS: por linha (token)} \quad \downarrow \text{BN: por coluna (feature)}$$

LayerNorm — token 1 (eixo d)

$\mu = 4, \quad \sigma^2 = 5$

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2}} = [-1.34,\ -0.45,\ 0.45,\ 1.34]$$

média 0 e variância 1, usando só as features do token

RMSNorm — token 1 (eixo d)

$$\text{RMS} = \sqrt{\tfrac{1}{4}\textstyle\sum_i x_i^2} = 4.58$$

$\mathbf{x}/\text{RMS} = [0.22,\ 0.66,\ 1.09,\ 1.53]$ (só reescala)

BatchNorm — feature 1 (lote)

coluna 1 = [1, 9]: $\mu = 5,\ \sigma^2 = 16$

$\hat{\mathbf{x}} = [-1,\ 1]$

usa estatísticas **do lote**, depende do batch

Conclusão

LN/RMS normalizam **por token**; BN **por feature**. Transformers usam LN/RMS por independerem do batch.

Layer Norm, RMS Norm e Batch Norm: Exemplo Numérico

Lote de 2 tokens, cada um com $d = 4$ *features*:

$$X = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 9 & 7 & 5 & 3 \end{bmatrix} \qquad \rightarrow \text{LN/RMS: por linha (token)} \quad \downarrow \text{BN: por coluna (feature)}$$

LayerNorm — token 1 (eixo d)

$\mu = 4, \quad \sigma^2 = 5$

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2}} = [-1.34,\ -0.45,\ 0.45,\ 1.34]$$

média 0 e variância 1, usando só as features do token

RMSNorm — token 1 (eixo d)

$$\text{RMS} = \sqrt{\tfrac{1}{4}\textstyle\sum_i x_i^2} = 4.58$$

$\mathbf{x}/\text{RMS} = [0.22,\ 0.66,\ 1.09,\ 1.53]$ (só reescala)

BatchNorm — feature 1 (lote)

coluna 1 = [1, 9]: $\mu = 5,\ \sigma^2 = 16$

$\hat{\mathbf{x}} = [-1,\ 1]$

usa estatísticas **do lote**, depende do batch

Conclusão

LN/RMS normalizam **por token**; BN **por feature**. Transformers usam LN/RMS por independerem do batch.

Layer Norm, RMS Norm e Batch Norm: Exemplo Numérico

Lote de 2 tokens, cada um com $d = 4$ *features*:

$$X = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 9 & 7 & 5 & 3 \end{bmatrix} \qquad \rightarrow \text{LN/RMS: por linha (token)} \quad \downarrow \text{BN: por coluna (feature)}$$

LayerNorm — token 1 (eixo d)

$\mu = 4, \quad \sigma^2 = 5$

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2}} = [-1.34,\ -0.45,\ 0.45,\ 1.34]$$

média 0 e variância 1, usando só as features do token

RMSNorm — token 1 (eixo d)

$$\text{RMS} = \sqrt{\tfrac{1}{4}\textstyle\sum_i x_i^2} = 4.58$$

$\mathbf{x}/\text{RMS} = [0.22,\ 0.66,\ 1.09,\ 1.53]$ (só reescala)

BatchNorm — feature 1 (lote)

coluna 1 = [1, 9]: $\mu = 5,\ \sigma^2 = 16$

$\hat{\mathbf{x}} = [-1,\ 1]$

usa estatísticas **do lote**, depende do batch

Conclusão

LN/RMS normalizam **por token**; BN **por feature**. Transformers usam LN/RMS por independerem do batch.

Pre-Norm vs. Post-Norm

Post-Norm (Vaswani 2017)

$x \leftarrow \mathrm{LN}(x + \mathrm{SubLayer}(x))$

Normalização **depois** da operação.
→ gradientes instáveis com N grande; requer *warm-up* lento.

Pre-Norm (GPT-2 em diante)

$x \leftarrow x + \mathrm{SubLayer}(\mathrm{LN}(x))$

Normalização **antes** da operação.
→ treino mais estável; gradientes fluem melhor pela conexão residual.

Hoje, quase todos os LLMs usam **Pre-Norm**.
A conexão residual "bypassa" a normalização, preservando o sinal ao longo das camadas.

Visão Geral

Decodificação Greedy

Após o LM produzir a distribuição $P(\cdot \mid x_{<t})$:

Greedy

$x_t = \arg\max_v P(v \mid x_{<t})$

- Rápido: nenhuma busca adicional
- Determinístico: mesma saída para o mesmo prompt
- Pode ser subótimo: maximiza cada passo isoladamente, não a sequência

Exemplo

Para *"O rato roeu a ___"*:

$P(\texttt{rolha}) = 0.60$, $P(\texttt{roupa}) = 0.25$, …

Greedy emite `rolha` **sempre**, mesmo que a continuação a partir de `roupa` fosse melhor.

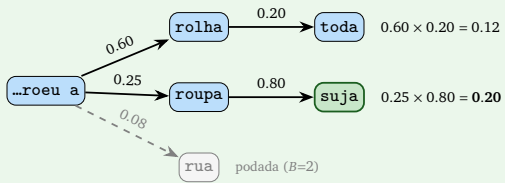
Beam Search

Beam Search

Mantém B hipóteses parciais; a cada passo expande todas e retém as B de maior **probabilidade conjunta**.

- $O(B)$ mais lento que greedy
- Encontra sequências de maior probabilidade total que greedy perde
- Também determinístico

Exemplo com $B = 2$



O beam escolhe `roupa suja`; greedy teria travado em `rolha`.

Amostragem

Amostragem

$x_t \sim P(\cdot \mid x_{<t})$

- Estocástico: saídas variam entre execuções
- Necessário para criatividade e diversidade
- Risco: tokens raros da cauda geram incoerências

Os parâmetros **temperature**, **top-k** e **top-p** (próximos slides) controlam essa distribuição.

Exemplo

Em 100 gerações de *"O rato roeu a ___"*:

≈60 emitem `rolha`, ≈25 `roupa`, ≈8 `rua`, …

Até `rei` ($P = 0.02$) aparece ocasionalmente, coerente ou não.

Por que Não Usar Sempre Greedy ou Beam? [10]

Holtzman et al. (2020): em geração aberta, maximizar a probabilidade produz **degeneração**.

Saída típica de beam search (GPT-2)

"Eu adoro música. Eu adoro música. Eu adoro música. Eu adoro música. Eu adoro…"

- Repetição em loop: repetir aumenta a probabilidade da própria repetição
- Texto genérico e previsível, sem informação nova

Observação central

Texto humano **não** é o texto mais provável: humanos alternam tokens esperados e surpreendentes.

- Amostragem pura resolve a repetição, mas a **cauda** da distribuição gera incoerências
- Solução: **amostragem truncada** (top-k, top-p), corta a cauda e amostra do resto

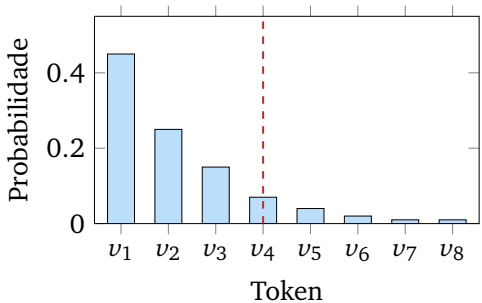
Temperature, Top-k e Top-p

Temperature:

$$P_T(x) = \frac{\exp(z_x/T)}{\sum_v \exp(z_v/T)}$$

- $T \to 0$: greedy
- $T = 1$: distribuição original
- $T > 1$: distribuição mais uniforme

Top-p: selecione o menor conjunto de tokens cuja probabilidade acumulada $\geq p$ (ex: $p = 0.9$), então amostre dentro desse conjunto.



Temperature, Top-k e Top-p: Exemplo Numérico

Logits para *"O rato roeu a ___"* (mesmo exemplo da LM head):

Token	Logit	$P_{T=0.5}$	$P_{T=1}$	$P_{T=2}$
`rolha`	3.0	**0.84**	**0.64**	**0.46**
`roupa`	2.1	0.14	0.26	0.29
`rua`	1.0	0.02	0.09	0.17
`rei`	−0.5	0.00	0.02	0.08

- T=0.5 **afia**: quase greedy
- T=2 **achata**: mais diversidade, mais risco

Truncamento sobre $P_{T=1}$:

Top-k com $k = 2$

Mantém `rolha`, `roupa`; renormaliza:

$$P(\texttt{rolha}) = \frac{0.64}{0.64 + 0.26} = 0.71$$

Top-p com $p = 0.9$

Acumulada: $0.64 \rightarrow 0.90$; o conjunto {`rolha`, `roupa`} já atinge p, os demais são cortados.

Top-k fixa o **tamanho** do conjunto; top-p fixa a **massa** de probabilidade, adaptando-se à incerteza do modelo.

Quando Parar de Gerar?

A geração autorregressiva não termina sozinha; três critérios de parada usuais:

1. **Token** `<eos>` (*end of sequence*): o modelo emite um token especial de fim
2. **Limite de tokens**: teto de custo e de janela de contexto
3. **Stop sequences**: strings definidas pela aplicação (ex.: `"\n\n"`, fim de bloco de código)

De onde vem o <eos>?

No pré-treino, documentos são concatenados separados por `<eos>`. O modelo aprende $P(\texttt{<eos>} \mid \text{contexto})$ como qualquer outro token.

Na prática

Amostrar `<eos>` encerra a resposta; atingir o limite de tokens **trunca** a resposta no meio, situação que as APIs sinalizam (ex.: `finish_reason`).

Decodificação Guiada (*Constrained Decoding*) [22]

Problema: LLMs podem gerar texto que não segue formato específico (JSON, SQL, código).

Solução: restringir a distribuição de saída a tokens válidos segundo uma **gramática** ou **esquema**.

Willard & Louf (2023): pré-compila um DFA para cada gramática, o mascaramento de tokens inválidos é $O(1)$ por token.

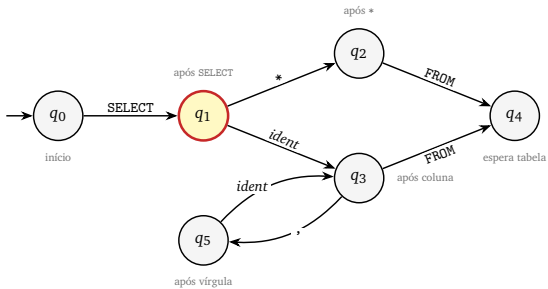
Exemplos de uso

- Geração de JSON estruturado
- SQL sem erros de sintaxe
- Extração em formato fixo
- Chamadas de API com parâmetros corretos

Implementações: `outlines`, `guidance`, `llama.cpp` grammar mode, vLLM structured outputs.

Decodificação Guiada: DFA para um Fragmento de SQL

Gramática simplificada: `SELECT`, depois $*$ ou lista de colunas separadas por `,`, depois `FROM` e uma tabela.



Estado atual: q_1

O modelo já gerou `SELECT`. De q_1 só saem transições que consomem $*$ ou um identificador de coluna; qualquer outro token é bloqueado (próximo slide).

Decodificação Guiada: Máscara sobre os Logits em q_1

O DFA define o conjunto válido $V_{q_1} = \{*,\ \textit{identificadores}\}$; a máscara soma $-\infty$ ao logit dos demais antes do softmax:

$$P(v \mid x_{\le t}) = \text{softmax}\big(z + M_{q_1}\big)_v, \qquad M_{q_1}[v] = 0 \text{ se } v \in V_{q_1},\ -\infty \text{ caso contrário}$$

Token	**Logit** z_v	$v \in V_{q_1}$?	$z_v + M_{q_1}$	P
`FROM`	2.5	×	$-\infty$	0
`*`	2.1	✓	2.1	**0.49**
`nome`	1.8	✓	1.8	0.36
`WHERE`	1.2	×	$-\infty$	0
`idade`	0.9	✓	0.9	0.15
`;`	0.3	×	$-\infty$	0

$$P(*) = \frac{e^{2.1}}{e^{2.1} + e^{1.8} + e^{0.9}} = \frac{8.2}{16.7} \approx 0.49$$

Leitura

Sem a máscara, `FROM` seria o token mais provável ($P \approx 0.36$) e produziria `SELECT FROM`, inválido. Com a máscara, o softmax renormaliza a probabilidade apenas entre os tokens válidos.

Por que $O(1)$ por token? A tabela estado → máscara é pré-compilada a partir da gramática; a cada passo basta indexá-la pelo estado atual [22].

Nomes de coluna fora do vocabulário são gerados em vários subtokens; o DFA é definido sobre o vocabulário de subtokens do modelo.

Decodificação Especulativa: Ideia Geral [14]

Gargalo: LLMs grandes são lentos, cada token exige um passe completo.

Observação: a maioria dos tokens é fácil de prever; tokens difíceis são exceção.

Algoritmo:

1. **Draft model** pequeno gera γ tokens
2. **Target model** verifica *em paralelo*
3. No primeiro rejeitado: re-amostra o token e **descarta o restante** do rascunho
4. Todos aceitos: amostra um token extra do passe do alvo, ganhando $\gamma + 1$ tokens

Propriedade chave

Saída **idêntica em distribuição** ao modelo alvo, sem perda de qualidade por construção.

Speedup típico

2–3× com draft 10× menor.
Ex: Llama 3 70B + Llama 3 8B como draft.

Algoritmo Especulativo: Critério de Aceitação

Draft model M_q, target model M_p:

Para cada token $\tilde{x}_t$ proposto pelo draft:

$$\text{Aceitar com prob. } \min\left(1,\ \frac{p(\tilde{x}_t \mid x_{<t})}{q(\tilde{x}_t \mid x_{<t})}\right)$$

Se rejeitado: re-amostre de

$$p'(x) \propto \max\big(0,\ p(x) - q(x)\big)$$

Verifique todos os γ tokens **em um único passe** do modelo alvo.

O tamanho do rascunho γ é um hiperparâmetro de inferência; valores típicos ficam entre 3 e 8, equilibrando taxa de aceitação e custo do draft.

- $p \geq q$: token **sempre aceito**
- Draft e target concordam → aceleração máxima
- Tokens difíceis: target corrige
- Rejeição na posição t: as posições $t{+}1, \ldots, \gamma$ do rascunho são **descartadas**

Complexidade

$\gamma + 1$ passos de M_q + 1 passe de M_p → até $\gamma + 1$ tokens gerados (γ aceitos + 1 extra amostrado de M_p).
Sem especulação: $\gamma + 1$ passes de M_p.

Decodificação Especulativa: Exemplo Passo a Passo

Contexto: completar "O gato subiu na ___" com $\gamma = 3$.

Draft M_q propõe: `árvore`, `e`, `dormiu`, verificados em **um único passe** de M_p.

Token	q	p	$\min\left(1, \frac{p}{q}\right)$	Decisão
`árvore`	0.6	0.9	**1,00**	sempre aceito
`e`	0.7	0.4	0.57	aceito c/ prob. 57%
`dormiu`	0.8	0.2	0.25	aceito c/ prob. 25%

Cenário A (todos aceitos)

3 aceitos + 1 extra de M_p:
4 tokens em 1 passe.
Sem especulação: 4 passes.

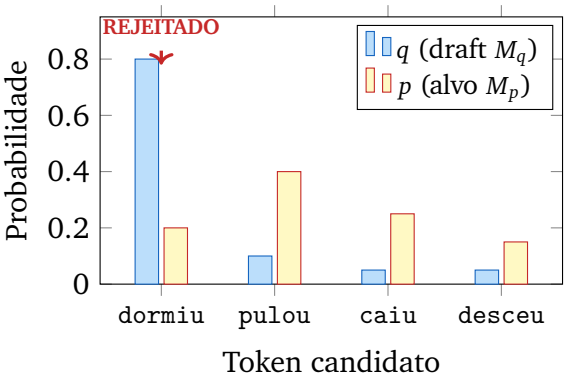
Cenário B (`e` rejeitado)

Re-amostra `e` de $p' \propto \max(0, p-q)$ e descarta `dormiu`;
`árvore` + token corrigido, ainda 1 passe.

Rejeição: Distribuições q e p para `dormiu`

Contexto: “O gato subiu na árvore e ___” (`árvore` e `e` já aceitos).

Draft M_q propõe `dormiu` com q=0.8; modelo alvo tem p=0.2.



Por que `dormiu` é rejeitado?

$p(\texttt{dormiu}) = 0.2 < q(\texttt{dormiu}) = 0.8$

Aceitar com probabilidade:

$$\min\left(1,\ \frac{p}{q}\right) = \frac{0.2}{0.8} = 0.25$$

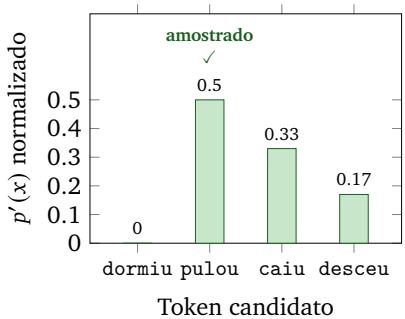
Neste cenário, `dormiu` **é rejeitado.**

Observação

M_p prefere `pulou` (p=0.4), o draft superestimou a certeza em `dormiu`.

Re-amostragem: $p'(x) \propto \max\big(0,\ p(x) - q(x)\big)$

Token	p	q	$p-q$	p' (norm.)
`dormiu`	0.20	0.80	−0.60	0.00
`pulou`	0.40	0.10	+0.30	**0.50**
`caiu`	0.25	0.05	+0.20	0.33
`desceu`	0.15	0.05	+0.10	0.17



Token selecionado de p'

`pulou` (p'=0.50)
"O gato subiu na árvore e `pulou`..."

Garantia de correção

p' concentra massa onde M_p discorda de M_q, garantindo distribuição **idêntica a amostrar de M_p**.

Decodificação Especulativa: Variantes e Resultados

Leviathan et al. (2023):

- Testado: T5-XXL (11B) + T5-small (60M) como draft
- Speedup 2–3× dependendo da tarefa
- HumanEval (código): maior speedup (tokens repetitivos)
- Sem degradação de qualidade

Variantes modernas:

- **Self-speculative**: camadas rascunho do próprio modelo
- **Medusa**: múltiplas cabeças de predição em paralelo
- **Eagle**: alinha representações de draft e target

Requisito

Benefício depende da taxa de aceitação α: quanto mais similares os modelos, maior α e maior o speedup.

Visão Geral

KV-Cache: Como Funciona na Prática

Gerando a sequência *"O rato roeu a rolha"*:

1. **Prefill**: alimenta o *prompt* inteiro de uma vez
   → computa e *armazena* K, V de cada token
2. **Decode passo 1**: prediz próximo token
   → computa K, V *só* do token novo
   → reutiliza K, V dos tokens anteriores do cache
3. **Decode passo 2, 3, ...**: idem
   → cache cresce a cada passo gerado

Sem cache (naïve)

Para o n-ésimo token:
Recalcula K, V de todos os n tokens
Custo total: $O(n^2 d)$

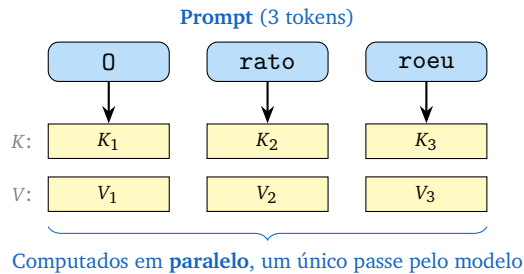
Com cache

Computa K, V apenas do token novo
Concatena com o cache existente
Custo por token: $O(nd)$

Memória do cache $\propto n \cdot L \cdot d_k$, cresce com a sequência e número de camadas.

KV-Cache: Visualização
Prefill

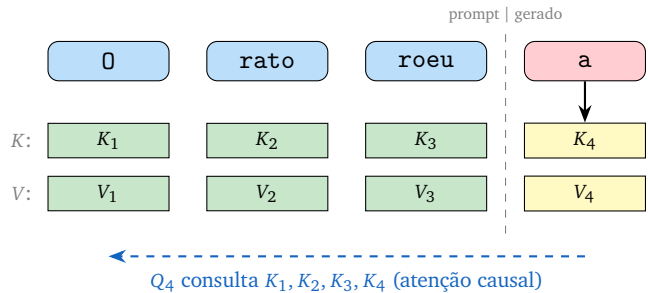
Exemplo: *“O rato roeu a rolha”*, todos os K,V do prompt são computados de uma vez.



= token = computando agora

KV-Cache: Visualização
Decode (Passo 1)

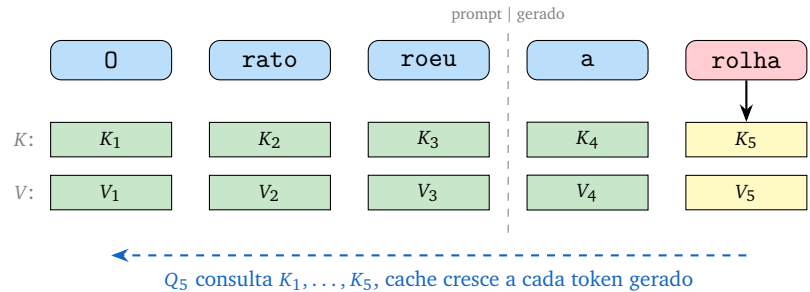
Gerando `"a"`: apenas K_4, V_4 são computados; K_1, K_2, K_3 vêm do cache.



= cache (reutilizado) = computando agora

KV-Cache: Visualização
Decode (Passo 2)

Gerando `"rolha"`: apenas K_5, V_5 são computados; cache cresce para 4 entradas.



= cache (reutilizado) = computando agora

Multi-Head Attention (MHA)
Recapitulação

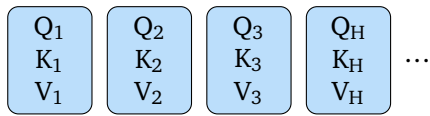
Com H cabeças, $d_k = d_{\text{model}}/H$:

- H projeções de Q, K, V independentes
- H pares de matrizes KV em cache por camada
- Custo de KV-cache por token: $2 \cdot H \cdot d_k \cdot L$

Para GPT-3 ($H = 96$, $d_k = 128$, $L = 96$):

Cache = $2 \times 96 \times 128 \times 96 \approx 2.4$M floats/token

Em fp16 (2 bytes/float): ≈4.7 MB por token; um contexto de 2048 tokens ocupa ≈**9.7 GB por requisição**.



KV-cache: H pares por camada

Gargalo em inferência: KV-cache cresce com n, H, L.

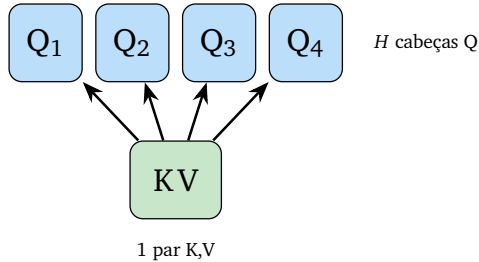
Multi-Query Attention (MQA)[1]

Reduza K e V para **uma única cabeça** compartilhada; mantenha H cabeças de Q.

$$\text{head}_h = \text{Attn}(QW_h^Q,\ KW^K,\ VW^V)$$

Redução de cache: $H\times \to 1\times$ nos tensores K, V.

Cache por token: $2 \cdot d_k \cdot L$ (independente de H).



- KV-cache $H\times$ menor
- Inferência muito mais rápida
- Leve queda de qualidade

[1] Noam Shazeer. "Fast Transformer Decoding: One Write-Head is All You Need". Em: arXiv preprint arXiv:1911.02150 (2019)

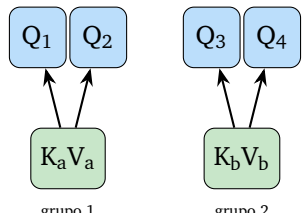
Grouped Query Attention (GQA)[1]

Um intermediário entre MHA e MQA.

Divida as H cabeças de Q em G grupos. Cada grupo compartilha um par K,V.

$$G = 1 \implies \text{MQA} \qquad G = H \implies \text{MHA}$$

Llama 2 (34B e 70B), Llama 3, Mistral e Mixtral usam GQA; as variantes 7B e 13B do Llama 2 ainda usam MHA.



- $G = 8$ típico (Llama 2 70B: $H = 64$, $G = 8$)
- Cache reduzido $H/G\times$
- Qualidade quase idêntica ao MHA

[1] Joshua Ainslie et al. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints". Em: EMNLP. 2023, pp. 4895–4901

Comparação: MHA, MQA e GQA

Variante	Pares K,V	Cache KV	Qualidade	Exemplos
MHA	H	$H \cdot d_k$	Referência	GPT-2, GPT-3, BERT
MQA	1	d_k	$\downarrow$ levemente	PaLM, Falcon
GQA	G $(1 < G < H)$	$G \cdot d_k$	$\approx$ MHA	Llama 2 70B, Llama 3, Mistral

GQA na prática

Llama 2 70B: $H = 64$ cabeças Q, $G = 8$ grupos KV.
Cache 8× menor que MHA com qualidade semelhante.

Por que importa?

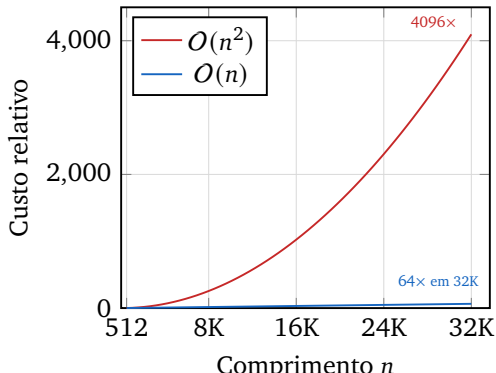
Redução do KV-cache é fundamental para servir muitos usuários simultaneamente, o cache de todos os requests cabe na GPU.

O Gargalo Quadrático da Atenção

Custo da atenção padrão [21]:

- Memória: $O(n^2)$, matriz de scores
- Compute: $O(n^2 d)$, produto $QK^\top$

Com $n = 32\,768$ tokens e $H = 32$ cabeças:
Matriz de atenção: $32768^2 \times 32 \approx 34$ bilhões de valores ≈ 69 GB (float16)



Motivação: reduzir custo ao escalar o contexto [20].

Sliding Window Attention [11]

Motivação: em sequências longas, a maioria das informações úteis está **próxima** do token atual.

Cada posição i atende apenas às W posições anteriores:

$$\text{Attn}(i) \leftarrow \text{tokens em } [i - W,\ i]$$

Custo: $O(n \cdot W)$ em vez de $O(n^2)$.

Campo receptivo em L camadas: $L \cdot W$.

Informação distante propaga-se por composição: camada l agrega o que camada $l-1$ já resumiu da janela anterior.

Configuração Mistral 7B

- Janela local: $W = 4096$
- 32 camadas
- Campo receptivo: $32 \times 4096 = 131\text{K}$
- Contexto declarado: 32K tokens

Resultado: Mistral 7B supera Llama 2 13B em benchmarks com 2× menos parâmetros.

FlashAttention: Atenção Exata e Eficiente [7]

MQA/GQA/SWA **alteram** a atenção. FlashAttention ataca outro gargalo: a **movimentação de memória** na GPU.

- GPUs computam rápido, mas ler e escrever na memória principal (HBM) é lento
- A implementação padrão **materializa** a matriz $n \times n$ de scores na HBM
- FlashAttention computa a atenção **por blocos** na SRAM (cache rápido), com *online softmax*, sem nunca materializar a matriz completa

Online softmax: acumula o softmax bloco a bloco, guardando apenas o máximo e a soma parciais dos scores, corrigidos a cada novo bloco.

Propriedades

- **Exata**: mesmo resultado numérico, sem aproximação
- Memória extra $O(n)$ em vez de $O(n^2)$
- Speedup de 2–4× no treino
- FlashAttention-2 [6]: padrão de fato atual

Lição

Parte do custo da atenção estava na **implementação**, não na matemática: otimizar antes de aproximar.

Resumo: Variantes de Atenção e Trade-offs

Variante	Compute	Mem. KV	Contexto longo	Qualidade	Exemplos
MHA (padrão)	$O(n^2 d)$	$H \cdot d_k$	Limitado	Referência	GPT-2/3
MQA	$O(n^2 d)$	d_k	Moderado	$\downarrow$ levemente	PaLM, Falcon
GQA	$O(n^2 d)$	$G \cdot d_k$	Moderado	$\approx$ MHA	Llama 2 70B, Llama 3, Mistral
SWA	$O(nWd)$	$W \cdot d_k$	Alto	Boa (local)	Mistral

Baseado em [20]. SWA = Sliding Window Attention.

Visão Geral

Motivação: Mais Parâmetros sem Mais Compute

Dilema:

- Modelos maiores → melhor qualidade
- Modelos maiores → mais compute por token

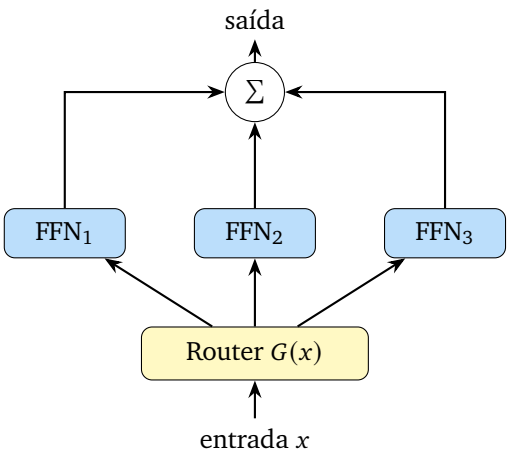
Ideia da Mistura de Especialistas (MoE):
Substitua a FFN densa por E redes menores (*especialistas*).
Cada token ativa apenas k especialistas ($k \ll E$).

Exemplo: Mixtral 8×7B

8 especialistas FFN
Apenas 2 ativados por token
→ 46.7B parâmetros totais
→ compute ≈ 12.9B por token

Origem: **Sparsely-Gated MoE** [19] (2017), revivido em LLMs a partir de 2022.

Arquitetura MoE: Router + Especialistas



Em cada bloco MoE:

1. Router calcula scores para cada especialista
2. Seleciona top-k especialistas
3. Processa x nos k especialistas
4. Combina saídas ponderadas pelos scores

Nota importante

As camadas de **atenção** permanecem densas.
Somente a FFN vira MoE.

Top-k Routing: Mecanismo de Gating

Dado token $x \in \mathbb{R}^d$ e pesos $W_g \in \mathbb{R}^{d \times E}$:

$$g(x) = \text{softmax}\big(\text{TopK}\big(xW_g,\ k\big)\big)$$

onde TopK mantém os k maiores scores e zera o resto.

Saída:

$$y = \sum_{i \in \text{Top-}k} g_i(x)\ \text{FFN}_i(x)$$

- Mixtral usa $k = 2$, $E = 8$
- ~25% dos parâmetros ativos por token
- Backward: apenas pelos k especialistas ativos

Expert collapse

Causa: o router aprende a enviar tokens ao especialista que já é o melhor, que melhora ainda mais, atraindo mais tokens, em ciclo de reforço positivo.

Resultado: 1–2 especialistas processam quase tudo; os demais recebem gradiente zero e nunca aprendem.

Solução: *Loss* auxiliar de balanceamento (próximo slide).

Top-k Routing: Exemplo Numérico

Token x, $E = 4$ especialistas, $k = 2$:

$$s = xW_g = [\,1.2,\ \mathbf{3.1},\ 0.4,\ \mathbf{2.6}\,]$$

$$\mathrm{TopK}(s, 2) = [\,-\infty,\ 3.1,\ -\infty,\ 2.6\,]$$

$$g = \mathrm{softmax} = [\,0,\ \mathbf{0.62},\ 0,\ \mathbf{0.38}\,]$$

$$g_2 = \frac{e^{3.1}}{e^{3.1} + e^{2.6}} = \frac{22.2}{35.7} = 0.62$$

$$y = 0.62\,\mathrm{FFN}_2(x) + 0.38\,\mathrm{FFN}_4(x)$$

O que foi economizado

FFN_1 e FFN_3 **nem são computadas** para este token: com $E = 4$ e $k = 2$, metade do compute da camada é evitado.

- Outro token da mesma sequência pode ativar especialistas diferentes
- O gradiente flui apenas pelos k especialistas ativos e pelo router

Problemas de MoE: Balanceamento de Carga

Perda auxiliar de balanceamento:

$$\mathcal{L}_{\text{aux}} = \alpha \cdot E \cdot \sum_{i=1}^{E} f_i \cdot P_i$$

onde f_i = fração de tokens roteados ao especialista i,
P_i = score médio do router para o especialista i.

Minimizar $\mathcal{L}_{\text{aux}}$ encoraja distribuição uniforme.

Outros desafios:

- **Instabilidade**: gradientes esparsas podem causar divergência
- **Comunicação**: especialistas distribuídos em GPUs diferentes (*model parallelism*)
- **Capacidade máxima**: tokens descartados se um especialista ficar sobrecarregado

Mixtral

Usa $\alpha = 0.01$, roteamento token a token.

Mixtral 8×7B[1]

Mistral AI

Arquitetura:

- 32 camadas transformer
- Cada FFN → 8 especialistas de 14336 neurônios
- Top-2 routing por token
- GQA ($G = 8$), RMSNorm, SWA
- Contexto: 32 768 tokens

Modelo	Params ativos	Params totais
Llama 2 13B	13B	13B
Llama 2 70B	70B	70B
Mistral 7B	7B	7B
Mixtral 8×7B	12.9B	46.7B

Resultado

Mixtral 8×7B supera Llama 2 70B em benchmarks com 5.4× menos parâmetros ativos.

[1] Albert Q. Jiang et al. "Mixtral of Experts". Em: arXiv preprint arXiv:2401.04088 (2024)

MoE Sparse vs. Modelos Densos

Característica	Denso (Llama 2 70B)	Sparse MoE (Mixtral 8×7B)
Parâmetros totais	70B	46.7B
Parâmetros ativos	70B	12.9B
Compute por token	Alto	~ 5× menor
Memória total (fp16)	~140 GB	~93 GB
Throughput (tokens/s)	Menor	Maior
Qualidade (MMLU)	68.9%	70.6%

MoE: mais capacidade com menor custo de inferência,
ao custo de maior memória total e complexidade de treino.

Visão Geral

O que Significa “Escalar” um LLM?

Três eixos de escala:

1. **Parâmetros** N: tamanho do modelo
2. **Dados** D: tokens de treinamento
3. **Compute** C: FLOPs totais de treino

(FLOP = *Floating Point Operation*, unidade de custo aritmético)

Relação aproximada:

$$C \approx 6ND$$

Cada token exige ≈6 FLOPs por parâmetro: 2 no *forward* + 4 no *backward* [13].

Pergunta central

Dado um orçamento de compute fixo, como distribuir entre N (modelo maior) e D (mais dados)?

Exemplo: Llama 2 7B (2T tokens)

$C \approx 6 \times (7{\times}10^9) \times (2{\times}10^{12}) \approx 8.4{\times}10^{22}$ FLOPs

A $\sim 1.5{\times}10^{14}$ FLOP/s úteis por A100, são ≈6500 GPU-dias: 3 semanas em 300 GPUs.

Isso é exatamente o que as **leis de escala** respondem.

Leis de Escala de Kaplan et al. [13]

Descoberta: a perda de teste segue **leis de potência**:

$$L(N) \propto N^{-\alpha_N}, \quad L(D) \propto D^{-\alpha_D}$$

com $\alpha_N \approx 0.076$, $\alpha_D \approx 0.095$.

Recomendação: dado compute fixo, priorize **escalar** N e use menos dados.

- Válido para modelos de 768 a 1.5B parâmetros (não-embedding)
- Curvas de perda suaves e previsíveis
- Arquitetura (profundidade vs. largura) tem impacto menor

Limitação

Kaplan et al. não otimizavam N e D simultaneamente; fixavam N e mediam L ótima em C.

Chinchilla: Treinamento Compute-Ótimo [9]

Hoffmann et al. (2022), DeepMind

Treinaram 400+ modelos variando N e D com mesmo C.

Resultado central

Para compute ótimo:

$$N^* \propto C^{0.5}, \quad D^* \propto C^{0.5}$$

Dobrar compute → dobrar N E dobrar D.

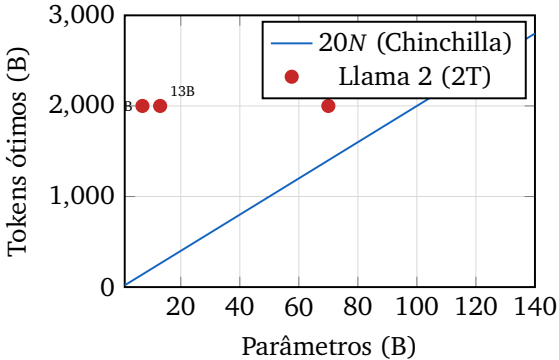
Chinchilla (70B) ≈ Gopher (280B) em qualidade, com 4× menos compute de inferência.

Regra prática

Para um modelo de N parâmetros, treine com pelo menos $20N$ **tokens**.

Exemplo: modelo de 7B → ≥140B tokens.

Relação Ótima Tokens/Parâmetros



- Linha azul: regra dos $20N$ tokens (compute-ótimo, Chinchilla)
- **Llama 2** usou 2T tokens em todos os tamanhos, bem acima do $20N$ (sobretreino proposital)
- Tendência atual: ainda mais dados. Llama 3 8B usou 15T, $\sim 93\times$ o $20N$ (= 160B)

Compute-ótimo ≠ inferência-ótimo

Modelos menores treinados por mais tempo custam menos na inferência, preferíveis quando há muitas chamadas.

Implicações Práticas das Leis de Escala

- **Llama 1** (Meta, 2023): 7B–65B treinados com 1T–1.4T tokens
- **Llama 2** (Meta, 2023): até 2T tokens, contexto 4096
- **Llama 3** (Meta, 2024): 8B e 70B com 15T tokens
- **Mistral 7B** (2023): supera Llama 2 13B com 7B parâmetros

Lição aprendida

Treinar modelos **menores por mais tempo** é mais custo-efetivo para implantação do que modelos gigantes subtreinados.

A Chinchilla reorientou o campo: de corrida por parâmetros (GPT-3, Gopher, Megatron) para equilíbrio $N \times D$.

Modelos Recentes: Parâmetros, Dados e Contexto

Modelo	Params	Tokens treino	Contexto	Norm	Atenção
GPT-2 XL	1.5B	~40B	1 024	Pre-LN	MHA
GPT-3	175B	300B	2 048	Pre-LN	MHA
Llama 2 7B	7B	2T	4 096	RMSNorm	MHA
Llama 2 70B	70B	2T	4 096	RMSNorm	GQA
Llama 3 8B	8B	15T	8 192	RMSNorm	GQA
Mistral 7B	7B	~1T	32 768	RMSNorm	GQA+SWA
Mixtral 8×7B	46.7B	~1T	32 768	RMSNorm	GQA+MoE

SWA = Sliding Window Attention; MoE = Mistura de Especialistas

Visão Geral

Perplexidade: A Métrica Mais Simples

Definição: mede o quão "surpreso" o modelo fica ao ver o texto.

$$\mathrm{PPL}(w_1, \ldots, w_N) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N} \log P(w_i \mid w_1, \ldots, w_{i-1})\right)$$

- Menor PPL $\rightarrow$ modelo estima melhor a distribuição do corpus
- PPL $= 1$: certeza absoluta; PPL $= |V|$: uniforme sobre o vocabulário
- Diretamente ligada à *cross-entropy loss* do treino

Exemplo numérico (3 tokens)

$P(w_1)$=0.5, $P(w_2)$=0.25, $P(w_3)$=0.125

$\mathrm{PPL} = \exp\left(\frac{0.69+1.39+2.08}{3}\right) = e^{1.39} = \mathbf{4.0}$

Intuição

PPL $= 20 \Rightarrow$ o modelo hesita, em média, entre 20 tokens igualmente prováveis a cada passo.

Perplexidade: Por que Não é Suficiente?

Limitações da perplexidade como única métrica

- **Acurácia factual**: um modelo com PPL baixa pode ainda assim alucinar fatos
- **Raciocínio**: baixa surpresa no texto não implica capacidade de resolver problemas
- **Seguimento de instruções**: não captura se o modelo obedece comandos do usuário
- **Desempenho em tarefas reais**: PPL não prediz bem resultados em downstream tasks

$\Rightarrow$ Precisamos de benchmarks específicos por tarefa para avaliar LLMs adequadamente.

Desafios na Avaliação de LLMs

Por que avaliar LLMs é difícil?

- Saída livre → métricas automáticas são imperfeitas
- *Data contamination*: benchmarks no pré-treino
- Capacidades emergentes: falha em instâncias simples, acerta difíceis
- Benchmarks ficam saturados rapidamente

Categorias de avaliação

- **Múltipla escolha**: ARC, MMLU
- **Geração + verificação**: GSM8k
- **Raciocínio**: HellaSwag, WinoGrande
- **Instrução**: MT-Bench, AlpacaEval
- **Arena humana**: Chatbot Arena (Elo)

ARC: AI2 Reasoning Challenge [4]

Clark et al. (2018), Allen Institute for AI

7787 questões de múltipla escolha de ciências (ensino fundamental/médio).

Duas partições:

- **ARC-Easy**: respondidas por sistemas de recuperação simples
- **ARC-Challenge**: exigem raciocínio; falham em sistemas simples

Exemplo (ARC-Challenge):

"Qual propriedade de um asteróide seria afetada de forma diferente pela gravidade de Júpiter e da Terra?"
(A) cor (B) massa (C) volume **(D) peso**

Avaliação: 0-shot ou 25-shot.

Resultados recentes

Llama 3 8B: 79.7% (ARC-C 25-shot)
GPT-4: ≈96%

GSM8k: Raciocínio Matemático [5]

Cobbe et al. (2021), OpenAI

8500 problemas matemáticos de nível fundamental.

Características:

- Resolução em múltiplos passos (2–8 operações)
- Resposta final é um número inteiro
- Avaliado com *chain-of-thought* (CoT), o modelo mostra o raciocínio passo a passo antes da resposta final

Exemplo:
"Natalia vendeu 48 clips em abril e metade disso em maio. Total?" $\Rightarrow 48 + 24 = 72$

Modelo	GSM8k (%)
GPT-2 (1.5B)	≈ 0
GPT-3 (175B)	17.9
Llama 2 7B	14.6
Llama 2 70B	56.8
Mixtral 8×7B	74.4
Llama 3 8B	79.6
Llama 3 70B	93.0
GPT-4	92.0

MMLU: Massive Multitask Language Understanding [8]

Hendrycks et al. (2021), UC Berkeley

15 908 questões em **57 áreas**:

- Matemática, física, química, biologia
- Direito, medicina, economia
- História, filosofia, ciência da computação

Nível: colegial ao profissional avançado.

Avaliação: 5-shot.

Exemplo: Conhecimento Clínico

Qual achado laboratorial é mais típico de anemia ferropriva?
(A) MCV elevado (B) Ferritina elevada

(C) MCV reduzido e RDW elevado (D) B12 reduzida

Modelo	**MMLU (%)**
GPT-3 (175B)	43.9
Llama 2 7B	45.3
Llama 2 70B	68.9
Mixtral 8×7B	70.6
Llama 3 8B	66.6
Llama 3 70B	79.5
GPT-4	86.4
Humano (estimado)	≈89.8

HellaSwag e WinoGrande: Raciocínio de Senso Comum

HellaSwag [23]

Zellers et al. (2019)
Completar a continuação mais plausível.

"Alguém lava um carro. Ele esfrega o capô…"
(A) e joga fora.
(B) por muito tempo.
(C) com esponja verde.
(D) e come um bolo.

GPT-3: 78.9%; Llama 3 8B: 82.0%

WinoGrande

Sakaguchi et al. (2021)
Resolução de co-referência adversarial.

"Mark é mais alto que Paul, então ele agachou na foto."
(Mark ou Paul agachou?)

Requer raciocínio físico/social.

Llama 3 8B: 73.0%; GPT-4: ≈87%

Open LLM Leaderboard: Como Comparar Modelos

Hugging Face mantém *leaderboard* público de LLMs avaliados em condições padronizadas:

- ARC-Challenge (25-shot)
- HellaSwag (10-shot)
- MMLU (5-shot)
- TruthfulQA (0-shot)
- WinoGrande (5-shot)
- GSM8k (5-shot)

Média dos 6 = **score composto**.

Cuidados na interpretação

- Modelos treinados para benchmarks
- Contaminação de dados de teste
- Prompts e few-shot afetam resultados
- Saturação rápida (MMLU-Pro, ARC-AGI)

Comparativo de Modelos nos Principais Benchmarks

Modelo	ARC-C	HSwag	MMLU	TruthQA	WinoG	GSM8k	Média
Llama 2 7B	53.1	78.6	45.3	39.4	74.0	14.6	50.8
Llama 2 13B	59.4	82.1	54.8	37.4	77.0	28.7	56.6
Llama 2 70B	67.3	87.3	68.9	44.9	83.7	56.8	68.2
Mistral 7B	60.0	81.3	60.1	42.2	75.3	52.2	61.8
Mixtral 8×7B	66.4	86.7	70.6	46.8	81.2	74.4	71.0
Llama 3 8B	79.7	82.0	66.6	44.0	73.0	79.6	70.8
Llama 3 70B	93.0	92.2	79.5	52.8	85.3	93.0	82.6

Valores aproximados do Open LLM Leaderboard (2024). HSwag = HellaSwag; TruthQA = TruthfulQA; WinoG = WinoGrande.

Benchmarks Específicos por Tarefa

Benchmarks gerais (MMLU, ARC…) medem conhecimento amplo, mas **não capturam** capacidades especializadas:

Contexto longo

- **RULER**: tarefas sintéticas em janelas de 4K–128K tokens
- **NIAH**: "agulha" escondida em textos de 100K+ tokens

Chamada de ferramentas

- **BFCL**: 2000 pares função/argumento em múltiplas linguagens
- **ToolBench**: uso de APIs reais via ReAct

Código

- **HumanEval**: geração de código Python; métrica *pass@k*
- **SWE-bench**: resolve issues reais do GitHub

Outros domínios

- **MT-Bench**: instrução multi-turno avaliada por GPT-4
- **MATH**: 12500 problemas de competição
- **MMMU**: multimodal (imagem + texto), 57 disciplinas

$\Rightarrow$ Nenhum benchmark único é suficiente: avaliação requer um portfólio de tarefas.

Resumo da Aula

Tópico	Conceito-chave	Inovação / Referência	Modelo exemplo
Decoder-only	Atenção causal, sem encoder; treino paralelo	GPT [15]	GPT, Llama
Normalização	RMSNorm + Pre-Norm	[2, 24]	Llama 2/3
Decodificação	Temperature, top-k/top-p; degeneração	[10]	—
MHA → GQA	Redução de KV-cache	[1]	Llama 2 70B
Sliding Window	Atenção local $O(nW)$	[11]	Mistral 7B
FlashAttention	Atenção exata, IO-eficiente	[7]	todos os atuais
Decod. especulativa	Draft + verify	[14]	Llama 3
MoE	Top-k routing, especialistas	[19, 12]	Mixtral 8×7B
Leis de escala	$20N$ tokens; $N^* \propto C^{0.5}$	Chinchilla [9]	Llama 3
Benchmarks	ARC, GSM8k, MMLU, HellaSwag	[4, 5, 8]	—

Referências I

[1] Joshua Ainslie et al. “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints”. Em: EMNLP. 2023, pp. 4895–4901.

[2] Jimmy Lei Ba, Jamie Ryan Kiros e Geoffrey E. Hinton. “Layer Normalization”. Em: arXiv preprint arXiv:1607.06450 (2016).

[3] Tom B. Brown et al. “Language Models are Few-Shot Learners”. Em: NeurIPS. Vol. 33. 2020, pp. 1877–1901.

[4] Peter Clark et al. “Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge”. Em: arXiv preprint arXiv:1803.05457 (2018).

[5] Karl Cobbe et al. “Training Verifiers to Solve Math Word Problems”. Em: arXiv preprint arXiv:2110.14168 (2021).

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

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

[8] Dan Hendrycks et al. “Measuring Massive Multitask Language Understanding”. Em: ICLR. 2021.

[9] Jordan Hoffmann et al. “Training Compute-Optimal Large Language Models”. Em: NeurIPS. Vol. 35. 2022.

[10] Ari Holtzman et al. “The curious case of neural text degeneration”. Em: ICLR. 2020.

[11] Albert Q. Jiang et al. “Mistral 7B”. Em: arXiv preprint arXiv:2310.06825 (2023).

[12] Albert Q. Jiang et al. “Mixtral of Experts”. Em: arXiv preprint arXiv:2401.04088 (2024).

[13] Jared Kaplan et al. “Scaling Laws for Neural Language Models”. Em: arXiv preprint arXiv:2001.08361 (2020).

[14] Yaniv Leviathan, Matan Kalman e Yossi Matias. “Fast Inference from Transformers via Speculative Decoding”. Em: ICML. 2023, pp. 19274–19286.

[15] Alec Radford et al. Improving Language Understanding by Generative Pre-Training. Rel. técn. OpenAI, 2018. URL: https://openai.com/blog/language-unsupervised.

[16] Alec Radford et al. Language Models are Unsupervised Multitask Learners. Rel. técn. OpenAI, 2019. URL: https://openai.com/blog/better-language-models.

Referências II

[17] Noam Shazeer. “Fast Transformer Decoding: One Write-Head is All You Need”. Em: arXiv preprint arXiv:1911.02150 (2019).

[18] Noam Shazeer. “GLU Variants Improve Transformer”. Em: arXiv preprint arXiv:2002.05202 (2020).

[19] Noam Shazeer et al. “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer”. Em: ICLR. 2017.

[20] Stanford CME295 Teaching Staff. Lecture 2: Modern LLM Architectures. Stanford CME295: Large Language Models in Practice, Fall 2025. 2025. URL: https://cme295.stanford.edu/slides/fall25-cme295-lecture2.pdf.

[21] Ashish Vaswani et al. “Attention is all you need”. Em: NeurIPS. Vol. 30. 2017.

[22] Brandon T. Willard e Rémi Louf. “Efficient Guided Generation for Large Language Models”. Em: arXiv preprint arXiv:2307.09702 (2023).

[23] Rowan Zellers et al. “HellaSwag: Can a Machine Really Finish Your Sentence?” Em: ACL. 2019, pp. 4791–4800.

[24] Biao Zhang e Rico Sennrich. “Root Mean Square Layer Normalization”. Em: NeurIPS. Vol. 32. 2019.