Large Language Models: Da Teoria à Prática

Fundamentos, Inferência, Fine-tuning e Avaliação

Bruno Leonardo Santos Menezes
Fabio André Machado Porto
Daniel Rocha de Senna

LNCC – Laboratório Nacional de Computação Científica

2026

Créditos

Este texto foi desenvolvido a partir do minicurso
"[MC-CD05] Large Language Models e Agentes",
apresentado na Jornada de Ciência de Dados do LNCC (fevereiro de 2026).

Sumário

Prefácio

Este livro apresenta os fundamentos teóricos e práticos de *Large Language Models* (LLMs), com foco em compreensão conceitual e aplicação direta. O material foi desenvolvido originalmente como suporte ao minicurso *"[MC-CD05] Large Language Models e Agentes"*, ministrado na Jornada de Ciência de Dados do LNCC em fevereiro de 2026, e expandido para servir como texto de referência para estudantes, pesquisadores e profissionais interessados em modelagem de linguagem com aprendizado profundo.

Ao longo dos capítulos, buscamos equilibrar a base essencial dos Transformers com tópicos contemporâneos de inferência, ajuste fino e avaliação. Quando relevante, incluímos seções "Na prática" para experimentação guiada, com ênfase em decisões de engenharia e limitações do estado da arte.

Nota de transparência: Este material contou com apoio de ferramentas de IA para revisão e aprimoramento de redação. Todo o conteúdo foi revisado criticamente pelos autores, que assumem total responsabilidade por sua precisão e coerência.

Código e materiais do curso. Os notebooks, scripts e demais recursos utilizados ao longo do minicurso estão disponíveis no repositório oficial: `https://github.com/brunoleomenezes/MC-CD05/`. Para informações sobre a Jornada de Ciência de Dados do LNCC, visite `https://verao.lncc.br/jcd/`.

Como este livro está organizado

O livro segue a estrutura do minicurso e foi pensado para leitura sequencial, mas também funciona como referência para consulta.

Os Capítulos 1–3 constroem o pano de fundo e a base técnica: panorama do ecossistema de LLMs e o núcleo do Transformer (tokenização, embeddings, atenção, FFN e normalização), incluindo variações modernas como RoPE, *attention scaling* e MoE.

Os Capítulos 4–5 entram em **inferência** e uso: diferenças entre arquiteturas causais e bidirecionais, controle de geração (temperatura, top-p), desempenho (KV cache) e práticas de prompting e *in-context learning*.

Os Capítulos 6–8 discutem **treinamento e adaptação**: objetivo de pré-treinamento em alto nível, otimização computacional (quantização) e ajuste fino eficiente (LoRA/QLoRA), seguindo para alinhamento por preferências (DPO, RLHF/GRPO) e avaliação moderna, incluindo limites de *LLM-as-a-judge*.

O Capítulo 9 fecha com tendências recentes e desafios abertos em LLMs abertos, com ênfase em interoperabilidade, governança e impacto científico. Ao longo do texto, as seções "Na prática" propõem experimentos guiados e observações de implementação.

Os autores
Petrópolis, fevereiro de 2026

Capítulo 1

Introdução aos Large Language Models

1.1 Panorama 2023–2025: contexto, escala e mudança de regime

Este capítulo situa os *Large Language Models* (LLMs) no contexto histórico, técnico e científico, enfatizando que eles não surgem como uma ruptura isolada, mas como o resultado de uma evolução contínua na modelagem de linguagem natural.

Entre 2023 e 2025, consolida-se um ponto que costuma ser mal compreendido: o fenômeno dos LLMs não decorre apenas de "modelos maiores". O que caracteriza esse período é uma combinação de mudanças na **escala**, no **regime de uso** e no **impacto sistêmico** dessas tecnologias. Em outras palavras, o salto não está apenas na qualidade média das respostas, mas no fato de que a linguagem natural passa a funcionar como uma interface computacional para uma ampla variedade de tarefas.

Historicamente, a modelagem de linguagem pode ser organizada em gerações, cada uma marcada por avanços conceituais e técnicos. A Figura 1.1 sintetiza essa evolução não apenas cronologicamente, mas sobretudo pela **capacidade crescente de resolver tarefas**.

1.2 As cinco gerações da modelagem de linguagem

1.2.1 Primeira geração: modelos estatísticos

A primeira geração é caracterizada pelos **modelos estatísticos de linguagem**, baseados em n-gramas e hipóteses de Markov. Esses modelos estimam probabilidades locais de sequências de palavras e funcionam bem como componentes auxiliares de tarefas específicas. Suas limitações incluem dependência de contagens esparsas e dificuldade em capturar dependências de longo alcance.

1.2.2 Segunda geração: modelos neurais

A segunda geração corresponde aos **modelos neurais de linguagem**, que introduzem representações distribuídas e aprendizado automático de características. A linguagem passa

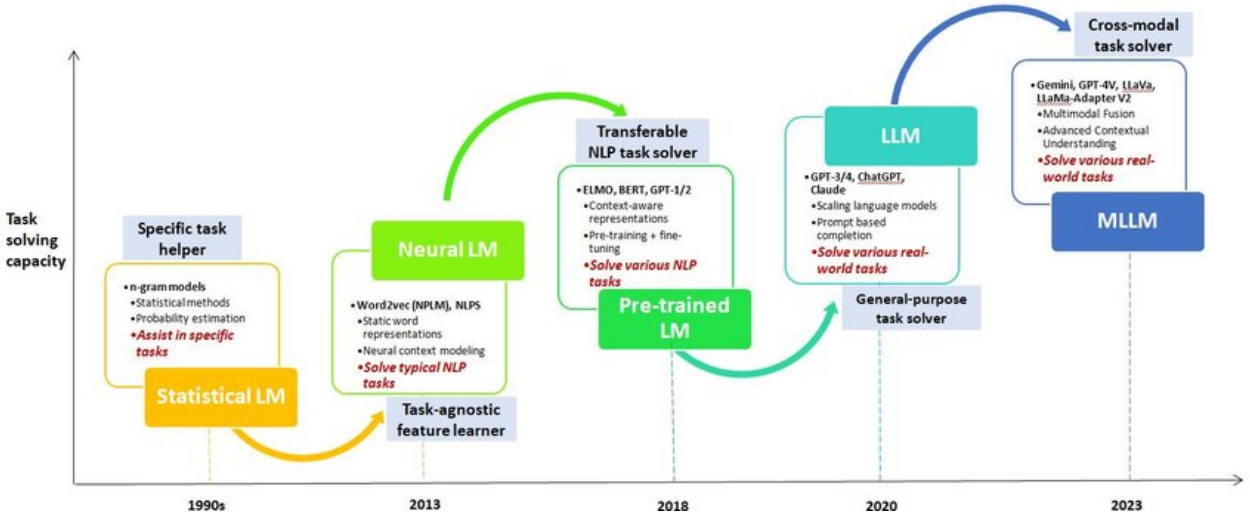

Figura 1.1: Evolução de gerações de modelos de linguagem segundo a capacidade de resolução de tarefas: de modelos estatísticos simples até LLMs modernos e sistemas multimodais (MLLM). Fonte: Mahato et al. [9].

a ser representada em espaços vetoriais contínuos, reduzindo a dependência de engenharia manual de atributos e permitindo melhor generalização a partir dos dados.

1.2.3 Terceira geração: pré-treinamento e adaptação

A terceira geração é marcada pelos **modelos de linguagem pré-treinados** (PLMs), exemplificados por ELMo, BERT e GPT-2. O salto conceitual central é o paradigma de **pré-treinamento em grandes corpora seguido de adaptação por fine-tuning**. Durante o pré-treinamento, o modelo aprende representações contextuais ricas; na etapa de adaptação, essas representações são especializadas para tarefas-alvo com volumes menores de dados supervisionados.

1.2.4 Quarta geração: LLMs e in-context learning

A quarta geração corresponde aos **Large Language Models** propriamente ditos. A partir de certos limiares de escala, emergem capacidades que não são observadas (ou não aparecem de forma robusta) em modelos menores, sendo a mais notável o *in-context learning*. Essa habilidade permite que o modelo execute uma tarefa ao observar exemplos no contexto de entrada, sem atualização explícita de parâmetros.

Nessa transição, o modelo deixa de ser apenas um extrator de representações e passa a operar como um **resolvedor geral de tarefas via linguagem natural**.

1.2.5 Quinta geração: modelos multimodais

A quinta geração, em evolução acelerada, corresponde aos **Multimodal Large Language Models** (MLLMs), que estendem as capacidades dos LLMs para além do texto, integrando imagens, áudio, vídeo e outras modalidades.

Exemplos incluem GPT-4V, Gemini, LLaVA e Qwen2-VL. O ponto central é a integração de múltiplas fontes de informação em um mesmo contexto de decisão, ampliando a gama de tarefas do mundo real que podem ser abordadas por um único sistema.

A Figura 1.2 ilustra a transição fundamental de modelos especializados para resolvedores gerais de tarefas.

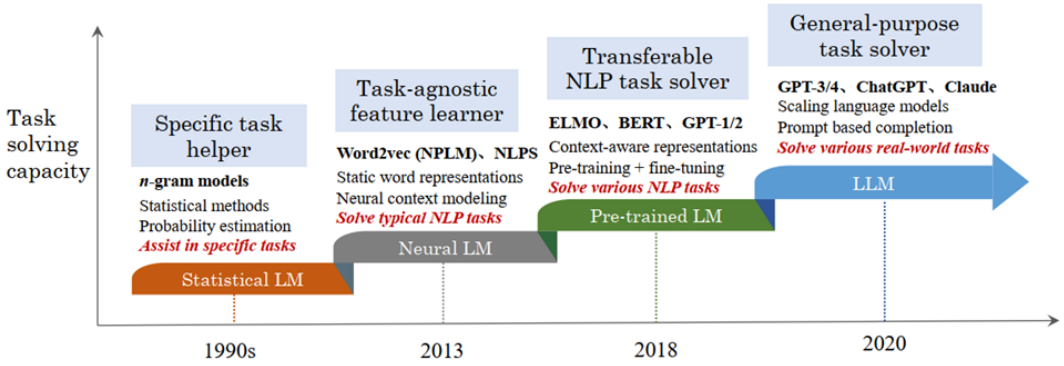

Figura 1.2: Transição de modelos especializados para resolvedores gerais de tarefas que operam via linguagem natural. Fonte: Zhao et al. [20].

1.3 Consolidação e ecossistema open-source)

No período entre 2023 e 2025, consolida-se a visão de que LLMs funcionam como uma **infraestrutura cognitiva de propósito geral**. Seu impacto se estende a áreas como processamento de linguagem natural, recuperação de informação, programação assistida, ciência de dados e aplicações científicas em múltiplos domínios.

Um aspecto particularmente relevante é o fortalecimento do **ecossistema open-source**. Modelos como LLaMA 3 (Meta), Qwen2 (Alibaba), Mixtral (Mistral AI) e DeepSeek indicam que a fronteira tecnológica não se restringe a modelos fechados. Em muitos casos, além dos pesos, há relatórios técnicos detalhados e descrições de pipelines de treinamento, o que facilita reprodutibilidade e acelera a difusão científica.

A Figura 1.3 ilustra o crescimento acelerado de publicações relacionadas a LLMs e técnicas associadas, um sinal da maturidade e da rápida evolução do campo.

1.4 Perspectiva: de modelos isolados a sistemas orquestrados

A literatura recente sugere que o futuro dos LLMs não reside apenas no aumento contínuo de escala, mas em sua atuação como **orquestradores de sistemas**. Nessa visão, o modelo se integra a mecanismos externos especializados, como busca, bases estruturadas e ferramentas computacionais, para aumentar confiabilidade, eficiência e controle sobre o comportamento do sistema.

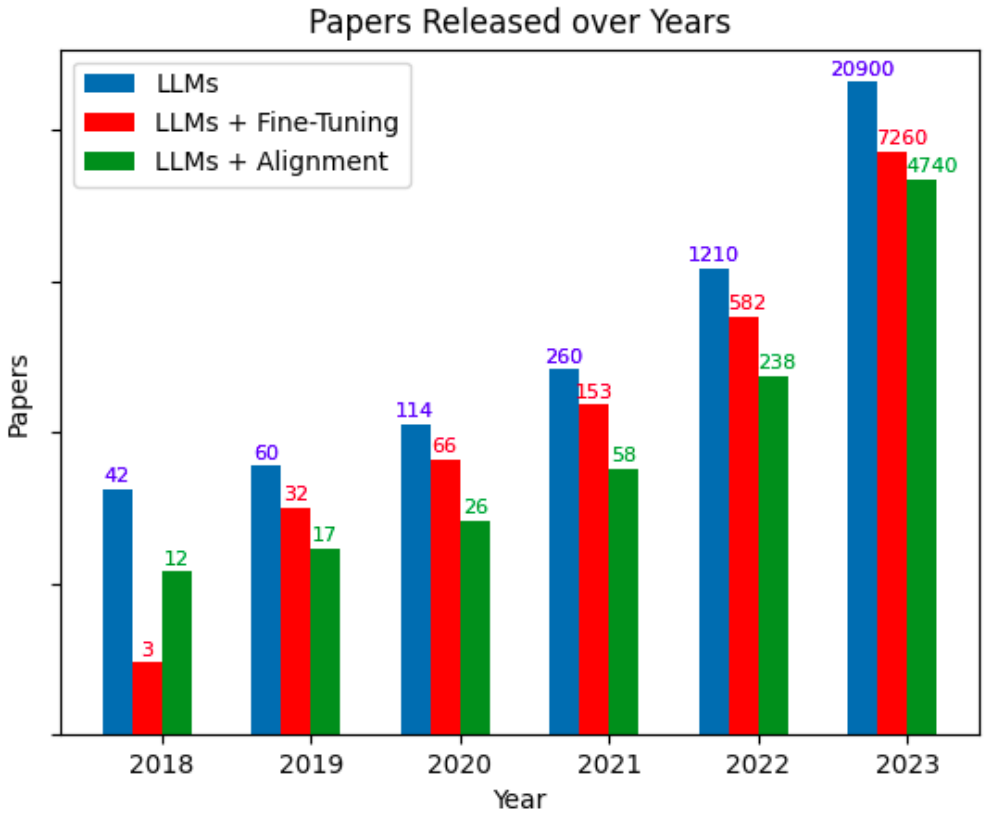

Figura 1.3: Evolução do número de publicações envolvendo LLMs, ajuste fino e alinhamento ao longo dos anos. Fonte: Naveed et al. [11].

Essa integração combina a flexibilidade da interface em linguagem natural com a precisão e verificabilidade de componentes externos, mitigando limitações conhecidas de LLMs isolados, como alucinações factuais e fragilidades em raciocínio matemático.

Pontos para lembrar

- LLMs não "surgem" em 2023: o período marca uma mudança de escala, disponibilidade e regime de uso (linguagem como interface).
- A transição para LLMs se torna evidente quando capacidades como *in-context learning* aparecem de forma robusta com o aumento de escala.
- O ecossistema open-source (modelos, relatórios e pipelines) acelerou a pesquisa e a adoção, aproximando academia e indústria.
- A tendência dominante é a migração de modelos isolados para sistemas orquestrados (LLM + busca + ferramentas + bases estruturadas).

O que vem a seguir

No próximo capítulo, introduzimos o núcleo conceitual do Transformer (atenção, embeddings e normalização) e as intuições que reaparecem mais adiante em inferência, ajuste fino e avaliação.

Capítulo 2

Núcleo do Transformer

2.1 Visão geral: do texto ao espaço vetorial

Este capítulo apresenta o núcleo operacional do Transformer a partir de uma perspectiva **executável e formalizável**: cada etapa é descrita em termos de tensores, operações matriciais e trechos de código que podem ser reproduzidos.

A ideia central é: o Transformer transforma uma sequência de símbolos em uma sequência de vetores. Ao longo de camadas sucessivas, esses vetores deixam de representar apenas a **identidade** de cada token e passam a incorporar também **posição** e **contexto** (isto é, relações de atenção com os demais tokens da sequência).

Nesta primeira parte, cobrimos duas transições fundamentais:

1. **Texto → tokens/IDs**: a entrada simbólica é convertida em índices inteiros;

2. **IDs → vetores**: cada índice passa a apontar para um vetor real via embeddings.

2.2 Tokenização: do texto aos tensores

Até este ponto, o texto existe como linguagem natural: há palavras, frases e pontuação. A tokenização converte essa representação simbólica em uma sequência discreta de tokens que pode ser representada por números inteiros.

Considere a frase de entrada:

$$text = \text{“A cute teddy bear is reading a book.”}$$

Para humanos, trata-se de uma frase com estrutura e significado. Para o modelo, é uma *string* que ainda não pode ser processada numericamente. O primeiro passo do Transformer é converter essa string em índices.

2.2.1 Implementação no BERT

O código abaixo executa a tokenização no BERT:

```
bert_inputs = bert_tokenizer(text, return_tensors="pt")

bert_input_ids = bert_inputs["input_ids"]
bert_attention_mask = bert_inputs["attention_mask"]

bert_tokens = bert_tokenizer.convert_ids_to_tokens(
    bert_input_ids[0].tolist()
)

print("[BERT] Tokens:")
for i, t in enumerate(bert_tokens):
    print(f"{i:02d} {t}")
```

A chamada `bert_tokenizer(text, return_tensors="pt")` realiza simultaneamente:

- normalização do texto (minúsculas, pois o modelo é *uncased*);
- segmentação WordPiece (subunidades frequentes, lidando com palavras raras/desconhecidas);
- inserção de tokens especiais `[CLS]` e `[SEP]`;
- conversão para tensores PyTorch.

2.2.2 Inspeção do objeto retornado

O objeto `bert_inputs` já contém os tensores usados como entrada do modelo:

```
print(bert_inputs)
```

Exemplo de saída:

```
{
 'input_ids': tensor([[ 101, 1037, 10140, 11389, 4562, 2003, 3752,
                       1037, 2338, 1012, 102]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
}
```

2.2.3 Tokens observados

A saída em tokens é:

```
[BERT] Tokens:
00 [CLS]
01 a
02 cute
03 teddy
04 bear
```

```
05 is
06 reading
07 a
08 book
09 .
10 [SEP]
```

Três observações importantes:

- `[CLS]` é inserido no início e `[SEP]` no fim;
- a frase vira uma sequência **ordenada** e **indexada**;
- essas posições serão usadas diretamente nas matrizes de atenção.

2.2.4 O tensor `input_ids`

O tensor `input_ids` é uma matriz de inteiros:

$$\texttt{input\_ids} \in \mathbb{N}^{1\times n}.$$

No exemplo, $n = 11$. A interpretação é:

- 1: tamanho do *batch* (número de sequências processadas em paralelo);
- n: comprimento da sequência após tokenização e tokens especiais;
- cada inteiro é um índice no vocabulário fixo do modelo.

Nota prática: batch size

O *batch size* altera **como** o modelo é executado computacionalmente, mas não altera seus pesos nem cria interação entre frases do batch. Em inferência, batches maiores aumentam paralelismo e consumo de memória; em treinamento, afetam a estimativa do gradiente e a dinâmica de otimização.

2.2.5 A máscara de atenção `attention_mask`

O tensor `attention_mask` é binário:

$$\texttt{attention\_mask} \in \{0,1\}^{1\times n}.$$

Ele indica quais posições são tokens reais (1) e quais são padding (0). Mesmo quando todos os valores são 1, a máscara é relevante para garantir que, em batches com padding, posições artificiais sejam ignoradas.

Máscara não é relevância

A máscara controla **existência estrutural**, não importância semântica: ela responde "este token existe na sequência?" e não "este token é importante?".

Quando sequências têm comprimentos diferentes, o tokenizer adiciona `[PAD]` para igualar o tamanho do batch. Nesses casos, a máscara marca as posições de padding com 0.

Onde a máscara entra na atenção

A atenção começa com uma matriz de compatibilidade:

$$\text{score}_{ij} = \frac{Q_i K_j^\top}{\sqrt{d_k}}.$$

Antes do softmax, posições mascaradas recebem um grande negativo ($-\infty$ conceitualmente), garantindo probabilidade zero após a normalização. Uma forma conceitual de escrever isso é:

$$\text{score}_{ij} \leftarrow \text{score}_{ij} + m_j, \quad m_j = \begin{cases} 0, & \text{se } \texttt{attention\_mask}_j = 1 \\ -\infty, & \text{se } \texttt{attention\_mask}_j = 0 \end{cases}$$

Na prática, implementações usam formas equivalentes (por exemplo, somar -10^9 nas posições mascaradas).

A relevância, por sua vez, surge do próprio mecanismo de atenção:

$$\alpha_{ij} = \text{softmax}\left(\frac{Q_i K_j^\top}{\sqrt{d_k}}\right).$$

em que os coeficientes α_{ij} são contínuos e somam 1 para cada token i.

2.2.6 Entrada formal do modelo

Após tokenização, a entrada do modelo é completamente descrita por:

$$\texttt{input\_ids} \in \mathbb{N}^{1\times n}, \qquad \texttt{attention\_mask} \in \{0,1\}^{1\times n}.$$

Ou seja: a partir daqui não há "texto" no sentido linguístico, apenas índices e tensores.

2.3 De índices a vetores: embeddings

Até aqui, os tokens eram apenas índices inteiros. Nesta etapa ocorre uma transição fundamental: cada índice passa a apontar para um vetor real em um espaço contínuo.

2.3.1 Matriz de embeddings

O BERT possui uma matriz de embeddings:

$$E \in \mathbb{R}^{|V| \times d}, \quad d = 768$$

em que $|V|$ é o tamanho do vocabulário e d é a dimensão do embedding.

2.3.2 Do ID ao vetor

Quando um token aparece na frase, ele indexa uma linha dessa matriz:

$$\mathbf{e}_{\text{teddy}} = E[11389] \in \mathbb{R}^{768}.$$

No código, isso corresponde ao parâmetro:

```
bert_model.embeddings.word_embeddings
```

e ao *lookup* em lote:

```
token_embeds = bert_model.embeddings.word_embeddings(bert_input_ids)
```

que implementa:

$$(i_1, i_2, \ldots, i_n) \longrightarrow (E[i_1], E[i_2], \ldots, E[i_n]).$$

O tensor resultante tem a forma:

$$\texttt{token\_embeds} \in \mathbb{R}^{1 \times n \times 768}.$$

2.3.3 O que embeddings ainda não são

Neste ponto, é importante evitar uma confusão comum: esses embeddings iniciais ainda **não são contextuais**. O vetor associado a `teddy` é o mesmo em qualquer frase; o contexto só aparece quando o modelo aplica atenção e camadas subsequentes.

2.3.4 Intuição: de embeddings de entrada ao surgimento do contexto

Até aqui, o que construímos foi uma representação inicial. Em particular, o embedding **lexical** de um token é recuperado por lookup na matriz E, de modo que o mesmo ID sempre retorna a mesma linha.

Isso implica que, antes das camadas de atenção, tokens com o mesmo ID compartilham o mesmo **componente lexical**. No BERT, porém, a entrada da rede não é apenas esse componente: o vetor que efetivamente entra na primeira camada é a soma

$$\mathbf{X} = \mathbf{E}_{\text{tok}} + \mathbf{E}_{\text{pos}} + \mathbf{E}_{\text{seg}}.$$

Assim, duas ocorrências de um mesmo token (por exemplo, `a` nas posições 1 e 7) podem ter vetores de entrada $\mathbf{X}$ diferentes por causa do termo posicional, mesmo que $\mathbf{E}_{\text{tok}}$ seja idêntico.

O ponto-chave é que, apesar dessa soma com posição/segmento, **ainda não existe interação entre tokens**. O contexto surge apenas quando a self-attention mistura informação entre posições. Por isso, após algumas camadas, **tokens iguais podem terminar diferentes**: cada ocorrência recebe atualizações distintas porque a distribuição de atenção depende do restante da sequência.

2.4 Embeddings de entrada no BERT: do código ao tensor

Nesta seção conectamos diretamente o código do BERT aos tensores observados na execução. Vamos explicitar **o que cada tensor representa** e **por que suas dimensões têm exatamente aquele formato**.

Partimos do ponto em que os tensores `input_ids` e `attention_mask` já existem. A partir deles, o BERT constrói os vetores de entrada que serão processados pelas camadas de atenção.

2.4.1 Comprimento da sequência

O comprimento n da sequência é extraído diretamente de `input_ids`:

```
with torch.no_grad():
    seq_len_bert = bert_input_ids.shape[1]
```

Se

$$\texttt{input\_ids} \in \mathbb{N}^{1\times n},$$

então `seq_len_bert` $= n$. Esse valor será reutilizado na construção dos embeddings subsequentes.

2.4.2 IDs posicionais

O BERT constrói explicitamente um vetor de posições absolutas:

```
position_ids = torch.arange(seq_len_bert).unsqueeze(0)
```

Em termos matemáticos,

$$\text{position\_ids} = [0, 1, 2, \ldots, n-1] \in \mathbb{N}^{1\times n}.$$

Sem um mecanismo posicional, o Transformer trataria a sequência como um conjunto sem ordem.

2.4.3 IDs de segmento (`token_type_ids`)

Para uma única frase, o BERT utiliza segmento zero em todas as posições:

```
token_type_ids = torch.zeros_like(bert_input_ids)
```

Isso produz

$$\text{token\_type\_ids} = \mathbf{0} \in \mathbb{N}^{1\times n}.$$

O embedding de segmento é útil quando duas sentenças são fornecidas na mesma entrada (por exemplo, em tarefas de pares de sentenças, QA ou inferência textual), onde parte dos tokens recebe segmento 0 e parte recebe segmento 1. No nosso caso, ele não introduz distinção adicional, mas permanece por consistência arquitetural.

2.4.4 Três componentes: token, posição e segmento

O BERT combina três tipos de embeddings, todos com dimensão 768 (no BERT-base):

- **embedding lexical** (quem o token é);
- **embedding posicional** (onde o token está);
- **embedding de segmento** (a qual sentença o token pertence).

Embedding lexical (token)

```
token_embeds = bert_model.embeddings.word_embeddings(bert_input_ids)
```

Se $E \in \mathbb{R}^{|V|\times 768}$ é a matriz de embeddings do vocabulário, então:

$$\mathbf{E}_{\text{tok}} = E[\texttt{input\_ids}] \quad \Rightarrow \quad \mathbf{E}_{\text{tok}} \in \mathbb{R}^{1\times n\times 768}.$$

Embedding posicional

```
position_embeds = bert_model.embeddings.position_embeddings(position_ids)
```

Se $P \in \mathbb{R}^{n_{\max}\times 768}$ é a matriz de embeddings posicionais aprendidos:

$$\mathbf{E}_{\text{pos}} = P[\text{position\_ids}] \quad \Rightarrow \quad \mathbf{E}_{\text{pos}} \in \mathbb{R}^{1\times n\times 768}.$$

Embedding de segmento

```
segment_embeds = bert_model.embeddings.token_type_embeddings(token_type_ids)
```

Se $S \in \mathbb{R}^{2\times 768}$ é a matriz de embeddings de segmento:

$$\mathbf{E}_{\text{seg}} = S[\text{token\_type\_ids}] \quad \Rightarrow \quad \mathbf{E}_{\text{seg}} \in \mathbb{R}^{1\times n\times 768}.$$

2.4.5 Composição final da entrada

A entrada vetorial do BERT é a soma componente a componente:

```
bert_input_embeds = token_embeds + position_embeds + segment_embeds
```

Definimos:

$$\mathbf{X} = \mathbf{E}_{\text{tok}} + \mathbf{E}_{\text{pos}} + \mathbf{E}_{\text{seg}} \quad \Rightarrow \quad \mathbf{X} \in \mathbb{R}^{1\times n\times 768}.$$

Para cada token i e dimensão k:

$$x_{i,k} = e^{(\text{tok})}_{i,k} + e^{(\text{pos})}_{i,k} + e^{(\text{seg})}_{i,k}.$$

Neste ponto, o texto já foi completamente convertido em geometria: as camadas seguintes recebem apenas o tensor $\mathbf{X}$.

2.4.6 Interpretação dos tensores impressos

Os quatro tensores frequentemente exibidos em depuração têm leitura direta:

- `token_embeds`: depende apenas do token (não é contextual);
- `position_embeds`: depende apenas da posição (muda com i);
- `segment_embeds`: depende do segmento (constante quando todos são 0);
- `bert_input_embeds`: soma dos três.

2.4.7 Leitura guiada da projeção PCA

Uma forma útil de visualizar o estado inicial é projetar os embeddings de entrada $\mathbf{X} \in \mathbb{R}^{n\times 768}$ em 2D via PCA ($768 \to 2$). Os eixos PC1 e PC2 não têm interpretação semântica direta; o que importa é proximidade relativa, agrupamentos e isolamento de tokens especiais.

A Figura 2.1 ilustra essa projeção para a frase do exemplo.

Tabela de correspondência: índice ↔ token

Índice	Token	Função
0	`[CLS]`	token especial (agregador / início)
1	`a`	artigo (1ª ocorrência)
2	`cute`	adjetivo
3	`teddy`	substantivo
4	`bear`	substantivo
5	`is`	verbo auxiliar
6	`reading`	verbo principal
7	`a`	artigo (2ª ocorrência)
8	`book`	substantivo (objeto)
9	`.`	pontuação
10	`[SEP]`	token especial (fim / separador)

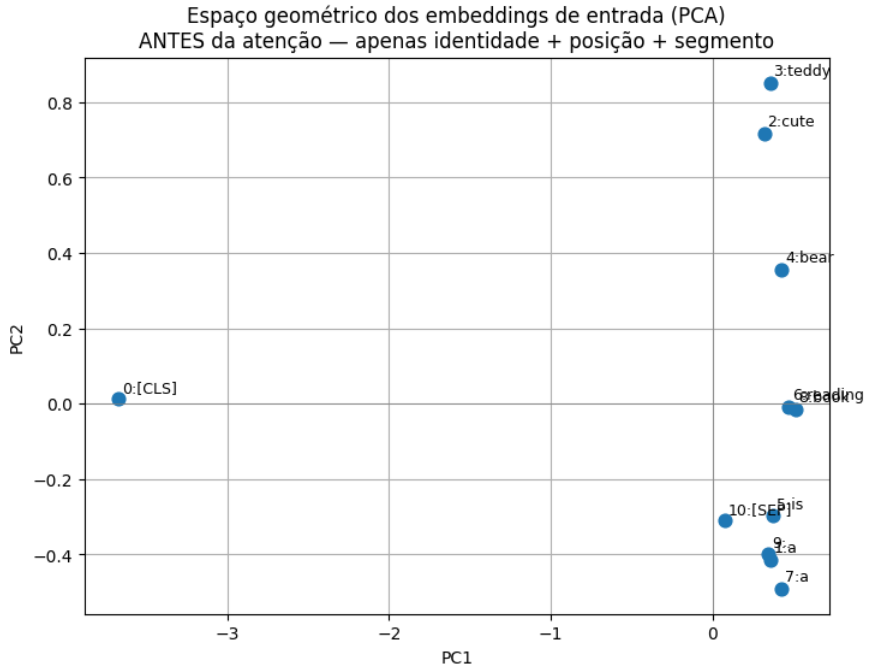

Figura 2.1: Projeção PCA (768→2) dos embeddings de entrada antes da atenção. A figura serve apenas para visualização: eixos PC1/PC2 não têm interpretação direta; o que importa é a posição relativa e o comportamento de tokens como `[CLS]` e `[SEP]`.

Note que `a` aparece duas vezes (índices 1 e 7). Embora o componente lexical seja o mesmo, o componente posicional difere, de modo que os embeddings de entrada não precisam coincidir na projeção.

2.5 Self-attention observada no BERT: do código ao fenômeno

Antes de derivarmos as equações de self-attention, vale observar o fenômeno como ele aparece em um BERT real. Nesta etapa, ainda não discutimos *como* os pesos são calculados; apenas examinamos *o que o modelo produz* ao expor suas atenções internas: quais tokens influenciam quais outros tokens e com que intensidade.

2.5.1 O que o modelo retorna quando pedimos as atenções

Ao executar o forward do BERT com `output_attentions=True`, o modelo retorna, além das representações finais, as matrizes de atenção internas de cada camada e de cada cabeça.

No experimento observado, obtemos:

```
[BERT] Número de camadas: 12
[BERT] Shape da atenção (camada 0): (1, 12, 11, 11)
```

Esse resultado é lido assim:

- BERT-base possui 12 camadas empilhadas;
- cada camada possui 12 cabeças de atenção;
- para cada cabeça, existe uma matriz $n \times n$, onde $n = 11$ é o número de tokens da sequência;
- em self-attention, cada token pode atribuir peso a *todos* os tokens (inclusive a si mesmo).

Portanto, a atenção não é um escalar nem um vetor: é uma **matriz quadrada de relações token–token**.

2.5.2 Leitura conceitual do mapa de atenção

A Figura 2.2 foi gerada a partir do notebook do minicurso (disponível no repositório mencionado no Prefácio) e exibe o *heatmap* de uma matriz de atenção correspondente a:

- uma única camada;
- uma única cabeça;
- um único exemplo do batch (nossa frase).

Denotemos essa matriz por

$$A \in \mathbb{R}^{n \times n}, \qquad n = 11.$$

Cada entrada $A_{i,j}$ representa o **peso de atenção** do token na posição i (*query*) sobre o token na posição j (*key*).

A leitura visual segue da seguinte maneira:

- **linhas** (eixo vertical): "para quem este token olha?";
- **colunas** (eixo horizontal): "quem está sendo observado?";
- cada linha soma aproximadamente 1 salvo erros numéricos de arredondamento), pois é uma distribuição de probabilidade;
- cores mais intensas indicam maior peso de atenção.

No BERT, não há máscara causal: qualquer token pode olhar para qualquer outro. Isso caracteriza a **atenção bidirecional**. Bidirecional, porém, não implica simetria: em geral,

$$A_{i,j} \neq A_{j,i}.$$

2.5.3 Como interpretar o heatmap

1) O que está sendo mostrado

O heatmap não representa "o BERT inteiro", mas uma **amostra interna**: uma cabeça em uma camada aplicada à frase completa. A matriz é $n \times n$ porque há n tokens na sequência (incluindo `[CLS]` e `[SEP]`).

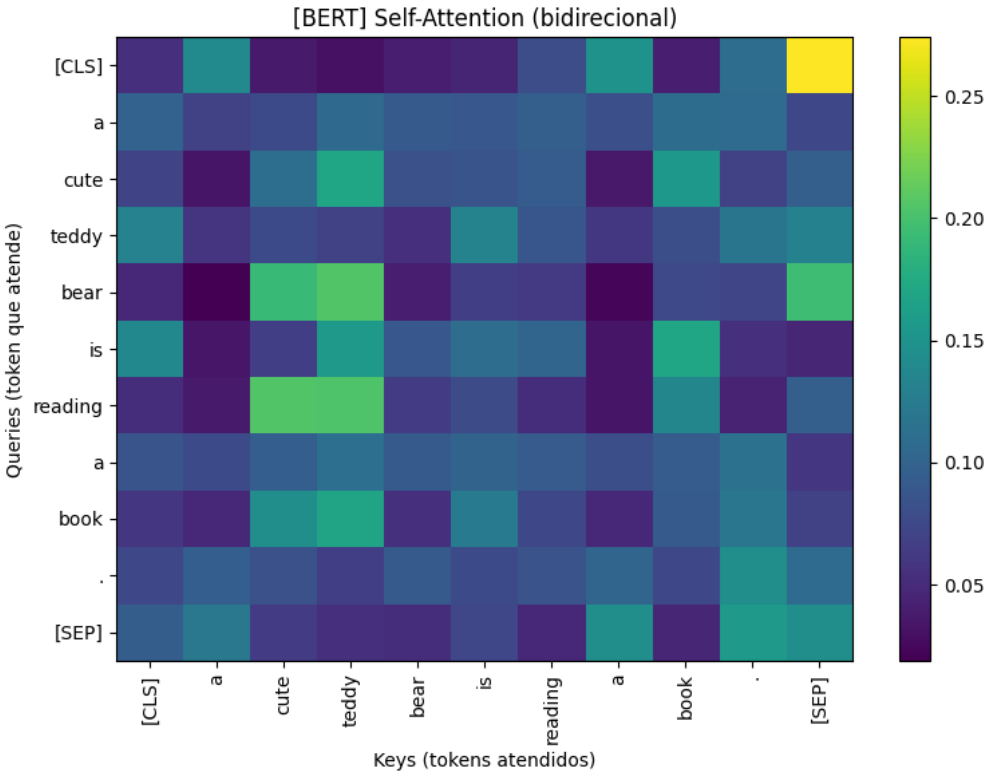

Figura 2.2: Mapa de *self-attention* do BERT (uma cabeça, uma camada) para a frase tokenizada. Linhas são *queries* (quem atende) e colunas são *keys* (quem é atendido). Cada linha é normalizada por *softmax* e soma aproximadamente 1.

2) Rótulos e ordem dos tokens

A sequência tokenizada tem $n = 11$ tokens na ordem abaixo:

pos	token
0	`[CLS]`
1	`a`
2	`cute`
3	`teddy`
4	`bear`
5	`is`
6	`reading`
7	`a`
8	`book`
9	`.`
10	`[SEP]`

Note que `a` aparece duas vezes (posições 1 e 7): o modelo "vê" token *e posição*. Na atenção, isso é crucial.

3) Barra de cores e normalização por linha

Os valores mostrados são pesos de atenção, isto é, probabilidades por linha. Assim, os pesos são não-negativos e cada linha soma aproximadamente 1 (resultado da normalização por softmax). A diagonal $A_{i,i}$ corresponde à autoatenção: é comum observar pesos não triviais porque o modelo preserva parte da informação local enquanto mistura contexto.

4) Leitura linha a linha

Escolha um token i e observe sua linha. Os maiores valores nessa linha indicam quais tokens j mais influenciam a atualização da representação de i *nesta cabeça e nesta camada*. Em termos operacionais, após essa etapa, o vetor do token i será atualizado como uma combinação ponderada de vetores associados às colunas com maior peso.

5) Tokens especiais `[CLS]` e `[SEP]`

`[CLS]` costuma atuar como coletor global: muitas cabeças aprendem a concentrar informação em `[CLS]` ou a fazer `[CLS]` observar tokens relevantes, pois ele é frequentemente usado como representação agregada em tarefas de classificação. `[SEP]` é um marcador estrutural (fim/separação), e também pode receber pesos não triviais por codificar regularidades de formato do input.

2.5.4 Exemplo concreto: para quem o token `teddy` olha

A linha de A correspondente ao token `teddy` (posição 3) pode conter valores como:

```
[0.04, 0.06, 0.09, 0.11, 0.10, 0.18, 0.21, 0.05, 0.07, 0.06, 0.03]
```

Esses valores:

- correspondem às intensidades do heatmap;
- somam aproximadamente 1;
- determinam quanto cada token contribui para a atualização do vetor de `teddy`.

No experimento, os pesos dominantes observados para `teddy` foram:

```
[BERT] Token 'teddy' atende mais para:
  -> pos 05 'is'
  -> pos 00 '[CLS]'
  -> pos 10 '[SEP]'
  -> pos 09 '.'
  -> pos 06 'reading'
```

Essa lista deve ser interpretada de forma operacional: o mecanismo de atenção mistura sinais simultaneamente **semânticos**, **sintáticos** e **estruturais**, dependendo da cabeça e da camada consideradas. Por exemplo, `is` conecta sujeito e predicado, `reading` tende a carregar a ação central, e `[CLS]`, `[SEP]` e `.` marcam aspectos estruturais da sequência.

2.5.5 O que aprendemos antes das equações

Mesmo antes da derivação matemática, já é possível afirmar:

- a atenção produz uma matriz $n \times n$ de probabilidades;
- cada token distribui seu "orçamento" de atenção pelos demais tokens;
- essa distribuição é observável no código e varia com a camada e a cabeça;
- as representações começam a incorporar dependências globais *antes* de qualquer regra linguística explícita.

A questão natural a seguir é: **de onde vêm esses números?** Na próxima seção, derivamos exatamente o cálculo que produz A:

- produtos escalares $QK^\top$ geram escores;
- divisão por $\sqrt{d_k}$ estabiliza as magnitudes;
- softmax transforma escores em probabilidades;
- a multiplicação por V produz a mistura final.

2.6 Self-Attention: definição formal

Na seção anterior, observamos a self-attention em funcionamento: matrizes reais, pesos reais e efeitos concretos sobre os vetores dos tokens. Agora, formalizamos as operações matemáticas que produzem exatamente aquelas matrizes.

2.6.1 Ponto de partida: o tensor de entrada

Após a etapa de embeddings, a entrada de uma camada de Transformer pode ser representada por um tensor

$$\mathbf{X} \in \mathbb{R}^{n \times d},$$

em que n é o número de tokens na sequência e d é a dimensão latente do modelo (por exemplo, $d = 768$ no BERT-base).

Nota de notação (batch): No BERT real, a entrada de uma camada tem três dimensões:

$$\mathbf{X} \in \mathbb{R}^{B \times n \times d},$$

onde B é o tamanho do batch. Ao longo desta derivação, omitimos B e trabalhamos com $\mathbf{X} \in \mathbb{R}^{n \times d}$ para descrever uma única sequência (isto é, um elemento do batch). Na implementação, todas as operações abaixo são aplicadas de forma idêntica e paralela para cada elemento do batch.

Cada linha de $\mathbf{X}$ corresponde a um token e agrega, neste estágio, três componentes já construídos: identidade lexical, posição e (quando aplicável) informação de segmento.

O ponto central é que, até aqui, não houve mistura de informação entre tokens. A self-attention é o mecanismo que introduz essa interação global.

2.6.2 Três projeções para três papéis: Q, K e V

A self-attention opera atribuindo três papéis distintos às representações de entrada: **query** (consulta), **key** (chave) e **value** (valor). Esses papéis não são "interpretações linguísticas" explícitas, mas resultados de *projeções lineares treináveis* aplicadas ao mesmo tensor $\mathbf{X}$:

$$\mathbf{Q} = \mathbf{X}W_Q, \qquad \mathbf{K} = \mathbf{X}W_K, \qquad \mathbf{V} = \mathbf{X}W_V,$$

com

$$W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}.$$

As matrizes W_Q, W_K e W_V são parâmetros aprendidos no treinamento. Elas alteram a geometria do espaço vetorial para que a atenção possa medir compatibilidades e combinar informações de maneira útil para a tarefa.

2.6.3 Compatibilidade entre tokens: produto interno $\mathbf{QK}^\top$

A atenção precisa quantificar o quanto um token deve considerar outro token ao atualizar sua representação. Essa compatibilidade é medida via produto escalar entre $\mathbf{Q}$ e $\mathbf{K}$, gerando uma matriz de escores:

$$\mathbf{S} = \mathbf{QK}^\top.$$

O elemento $S_{i,j}$ expressa o alinhamento entre a *query* do token i e a *key* do token j. Valores maiores indicam maior compatibilidade geométrica; valores menores indicam menor afinidade. Nesse ponto, $\mathbf{S}$ ainda não representa probabilidades: é apenas uma matriz de escores.

2.6.4 Normalização por $\sqrt{d_k}$

Antes de converter escores em probabilidades, aplica-se a normalização conhecida como *scaled dot-product attention*:

$$\mathbf{S}' = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}.$$

Essa divisão tem motivação numérica. À medida que d_k cresce, o produto interno tende a produzir valores com magnitude maior, o que pode fazer o softmax saturar e reduzir a sensibilidade dos gradientes. Ao dividir por $\sqrt{d_k}$, preservamos uma escala mais estável para o softmax, melhorando a dinâmica de treinamento e evitando extremos numéricos.

2.6.5 De escores a probabilidades: softmax

A matriz de atenção propriamente dita é obtida aplicando softmax linha a linha:

$$\mathbf{A} = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right).$$

Com isso, cada linha de $\mathbf{A}$ se torna uma distribuição de probabilidade: os elementos são não negativos e somam 1. Em particular, $\mathbf{A}_{i,j}$ representa o peso com que o token i (query) considera o token j (key) ao atualizar sua representação. É essa matriz $\mathbf{A}$ que aparece visualmente nos heatmaps extraídos do modelo.

2.6.6 Mistura de informação: aplicando a atenção aos valores

Atenção não é apenas "decidir para onde olhar"; ela serve para combinar informação. A atualização efetiva dos vetores ocorre quando os pesos $\mathbf{A}$ são aplicados aos valores $\mathbf{V}$:

$$\mathbf{Z} = \mathbf{A}\mathbf{V}.$$

Cada linha de $\mathbf{Z}$ é uma combinação ponderada das linhas de $\mathbf{V}$. O token não copia literalmente outros tokens; ele mistura sinais de acordo com os pesos de atenção. O resultado é uma representação *contextualizada*: o mesmo token lexical, em contextos diferentes, pode produzir vetores finais diferentes após a propagação pelas camadas do Transformer.

2.7 Conexões Residuais e Multi-Head Attention

Uma camada do Transformer não substitui a representação de entrada do token. Em vez disso, ela aprende uma *correção* (um incremento) sobre o vetor original. Esse princípio aparece nas **conexões residuais**, que preservam um caminho de identidade e estabilizam tanto a propagação do sinal quanto o fluxo de gradiente em redes profundas.

2.7.1 Conexões residuais: somar em vez de reescrever

Dada uma transformação vetorial $F(\mathbf{x})$, a saída residual é definida por

$$\mathbf{y} = \mathbf{x} + F(\mathbf{x}).$$

O termo $\mathbf{x}$ não é um detalhe de implementação: ele cria um atalho direto no grafo computacional, garantindo que a representação original permaneça disponível mesmo quando $F(\cdot)$ é ruidosa ou ainda não foi bem ajustada durante o treinamento.

No bloco de atenção do BERT, as conexões residuais aparecem em dois pontos clássicos da camada:

- após a self-attention (subbloco de atenção);
- após a rede feed-forward (subbloco MLP).

De forma conceitual, o padrão é:

$$\mathbf{X} \leftarrow \mathbf{X} + \mathrm{Attention}(\mathbf{X}), \qquad \mathbf{X} \leftarrow \mathbf{X} + \mathrm{FFN}(\mathbf{X}),$$

isto é, cada subbloco devolve um *incremento* que é somado ao estado corrente.

Correspondência com o código do BERT

No BERT, essa lógica está encapsulada nos módulos de saída do subbloco de atenção e do subbloco feed-forward. Em termos de navegação de módulos, os pontos relevantes são:

- saída do subbloco de atenção: `bert_model.encoder.layer[i].attention.output`
- saída do subbloco feed-forward: `bert_model.encoder.layer[i].output`

Mesmo quando a soma residual não aparece "escrita" no notebook, ela está presente no forward dessas camadas e faz parte da computação interna do modelo.

2.7.2 A equação da atenção, agora no lugar certo

Com as definições da seção anterior, podemos registrar a expressão completa da atenção em um único bloco:

$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}.$$

Neste ponto, essa equação deixa de ser apenas uma forma compacta e passa a ter leitura operacional: ela descreve exatamente (i) a matriz de escores $\mathbf{Q}\mathbf{K}^\top$, (ii) a normalização por $\sqrt{d_k}$, (iii) a conversão em probabilidades via softmax, e (iv) a mistura de informação ao multiplicar pelos valores $\mathbf{V}$. É essa cadeia de operações que produz as matrizes visualizadas nos heatmaps e, em seguida, altera os vetores dos tokens por combinação ponderada.

2.7.3 Multi-Head Attention: projeções paralelas e recombinação

Até aqui, descrevemos a self-attention como uma única projeção $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$. No BERT real, a atenção é computada em **múltiplas cabeças** executadas em paralelo. A ideia aqui é dar ao modelo mais de uma forma de comparar tokens: em vez de ficar preso a um único espaço de compatibilidade, ele aprende várias "perspectivas" independentes de interação entre tokens.

Nota (shapes): Na implementação, $\mathbf{X}$ tem shape (B, n, d) e as projeções produzem tensores (B, n, d_k) por cabeça; a dedução abaixo omite B apenas por clareza.

Para cada cabeça $h \in \{1, \ldots, H\}$, definimos projeções próprias:

$$\mathbf{Q}^{(h)} = \mathbf{X}W_Q^{(h)}, \quad \mathbf{K}^{(h)} = \mathbf{X}W_K^{(h)}, \quad \mathbf{V}^{(h)} = \mathbf{X}W_V^{(h)},$$

onde $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{d \times d_k}$ e, tipicamente,

$$d_k = \frac{d}{H}.$$

Cada cabeça produz uma saída própria:

$$\mathbf{Z}^{(h)} = \text{softmax}\left(\frac{\mathbf{Q}^{(h)}\mathbf{K}^{(h)\top}}{\sqrt{d_k}}\right)\mathbf{V}^{(h)}.$$

As H saídas são então concatenadas ao longo da dimensão de features:

$$\mathbf{Z}_{\text{concat}} = \text{Concat}\big(\mathbf{Z}^{(1)}, \mathbf{Z}^{(2)}, \ldots, \mathbf{Z}^{(H)}\big) \in \mathbb{R}^{n \times d},$$

e finalmente recombinadas por uma projeção linear:

$$\mathbf{Z} = \mathbf{Z}_{\text{concat}}W_O, \qquad W_O \in \mathbb{R}^{d \times d}.$$

Esse $\mathbf{Z}$ é o resultado do subbloco de atenção antes da adição residual e das normalizações tratadas nas próximas seções.

Correspondência com o código do BERT

No BERT, as projeções por cabeça são implementadas dentro do módulo de atenção. Os componentes que correspondem diretamente às matrizes de projeção são:

- projeções $\mathbf{Q}, \mathbf{K}, \mathbf{V}$:
  `bert_model.encoder.layer[i].attention.self.query`,
  `bert_model.encoder.layer[i].attention.self.key`,
  `bert_model.encoder.layer[i].attention.self.value`
- tensores de atenção por camada e cabeça: `outputs.attentions[i]`,
  com shape `[batch, heads, seq_len, seq_len]`
- projeção final W_O:
  `bert_model.encoder.layer[i].attention.output.dense`

Assim, as equações apresentadas nesta seção são exatamente as que o BERT executa internamente, ainda que a implementação esteja distribuída em módulos do framework.

Ideia para guardar: Atenção é um mecanismo de mistura de vetores guiado por probabilidades; as conexões residuais garantem que essa mistura não apague o sinal original.

2.8 Mini-corpus didático: cálculo completo

Para enxergar a self-attention sem "caixa-preta", vamos reduzir o problema a um exemplo mínimo, inteiramente calculável à mão. O objetivo é acompanhar cada soma, produto e normalização até chegar ao vetor contextualizado final.

2.8.1 Vocabulário e mapeamento (token → índice)

Definimos um vocabulário com quatro tokens:

$$V = \{\text{teddy}, \text{bear}, \text{reads}, \text{sleeps}\}.$$

Associamos cada token a um índice inteiro:

$$f(\text{teddy}) = 0, \quad f(\text{bear}) = 1, \quad f(\text{reads}) = 2, \quad f(\text{sleeps}) = 3.$$

A partir daqui, a entrada do modelo é uma sequência de índices.

2.8.2 Embeddings: por que existem e por que aqui são bidimensionais

Em modelos reais (por exemplo, BERT-base), cada token é representado por um vetor em alta dimensão, tipicamente $d = 768$. Essa dimensão alta aumenta a capacidade de representação, mas torna impossível "ver" a matemática passo a passo.

Aqui, o objetivo é transparência, não expressividade. Por isso fixamos deliberadamente:

$$d = 2.$$

Nada essencial sobre o mecanismo muda: apenas reduzimos a escala do espaço vetorial para tornar os cálculos legíveis.

2.8.3 Dimensão latente e matriz de embeddings

Um embedding vive em $\mathbb{R}^d$:

$$\mathbf{e}_t \in \mathbb{R}^d.$$

As dimensões são *latentes*: não têm nomes semânticos fixos (e.g., "animal", "ação"). O modelo aprende essas coordenadas no treinamento.

A matriz de embeddings é uma tabela de consulta (*lookup table*):

$$E \in \mathbb{R}^{|V| \times d}.$$

No nosso caso, $|V| = 4$ e $d = 2$, então $E \in \mathbb{R}^{4\times 2}$. Escolhemos explicitamente:

$$E = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ -1 & 1 \end{bmatrix}.$$

Cada linha é o vetor do token correspondente:

$$\mathbf{e}_{\text{teddy}} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{e}_{\text{bear}} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{e}_{\text{reads}} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{e}_{\text{sleeps}} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Em modelos reais, esses vetores são aprendidos por otimização. Aqui, escolhemos números simples apenas para expor a geometria (produtos escalares e softmax).

2.8.4 Entrada "teddy reads": de tokens a tensor

Considere a sequência:

$$x = \text{"teddy reads"}.$$

Após tokenização:

$$\mathcal{T}(x) = (\text{teddy}, \text{reads}),$$

e após o mapeamento:

$$(\text{teddy}, \text{reads}) \xrightarrow{f} (0, 2).$$

Passo 1: lookup na tabela E

Buscamos as linhas correspondentes em E:

$$\mathbf{e}_{\text{teddy}} = E[0] = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad \mathbf{e}_{\text{reads}} = E[2] = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Passo 2: empilhar para formar X

O Transformer representa a sequência como uma única matriz, com uma linha por token:

$$X = \begin{bmatrix} \mathbf{e}_{\text{teddy}}^\top \\ \mathbf{e}_{\text{reads}}^\top \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \in \mathbb{R}^{n\times d} = \mathbb{R}^{2\times 2}.$$

Aqui, $n = 2$ tokens e $d = 2$ dimensões latentes.

2.8.5 Definindo Q, K e V

Em geral, o Transformer projeta X em três espaços:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V.$$

Para evitar que projeções aprendidas escondam a mecânica básica, usaremos uma única cabeça e definiremos:

$$W_Q = W_K = W_V = I, \qquad I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Logo,

$$Q = X, \qquad K = X, \qquad V = X, \qquad d_k = d = 2.$$

Essa escolha não é "realista", mas é a forma mais direta de expor as contas.

2.8.6 Escores: calculando $QK^\top$ elemento a elemento

Primeiro, calculamos

$$K^\top = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Então,

$$QK^\top = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Cada entrada é "linha $\times$ coluna":

$$(1,1) = 1 \cdot 1 + 0 \cdot 0 = 1, \qquad (1,2) = 1 \cdot 1 + 0 \cdot 1 = 1,$$

$$(2,1) = 1 \cdot 1 + 1 \cdot 0 = 1, \qquad (2,2) = 1 \cdot 1 + 1 \cdot 1 = 2.$$

Logo,

$$QK^\top = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}.$$

Esses escores medem compatibilidade geométrica (produto escalar) entre tokens.

2.8.7 Normalização: dividindo por $\sqrt{d_k}$

Como $d_k = 2$,

$$\sqrt{d_k} = \sqrt{2} \approx 1.414.$$

Dividindo elemento a elemento:

$$S = \frac{QK^\top}{\sqrt{2}} \approx \begin{bmatrix} 0.707 & 0.707 \\ 0.707 & 1.414 \end{bmatrix}.$$

2.8.8 Softmax linha a linha: de escores a probabilidades

A matriz de atenção é:

$$A = \mathrm{softmax}(S),$$

com softmax aplicada *por linha*.

Linha 1: $[0.707,\ 0.707]$**.**

$$e^{0.707} \approx 2.028, \quad e^{0.707} \approx 2.028, \quad \text{soma} = 4.056,$$
$$A_{11} = \frac{2.028}{4.056} = 0.5, \qquad A_{12} = \frac{2.028}{4.056} = 0.5.$$

Linha 2: $[0.707,\ 1.414]$**.**

$$e^{0.707} \approx 2.028, \quad e^{1.414} \approx 4.113, \quad \text{soma} = 6.141,$$
$$A_{21} = \frac{2.028}{6.141} \approx 0.33, \qquad A_{22} = \frac{4.113}{6.141} \approx 0.67.$$

Portanto:

$$A \approx \begin{bmatrix} 0.5 & 0.5 \\ 0.33 & 0.67 \end{bmatrix}.$$

Cada linha soma 1: é uma distribuição de atenção por token.

2.8.9 Mistura final: $Z = AV$

Como $V = X$,

$$V = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Então:

$$Z = AV = \begin{bmatrix} 0.5 & 0.5 \\ 0.33 & 0.67 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Calculando:

$$z_{11} = 0.5 \cdot 1 + 0.5 \cdot 1 = 1.0, \qquad z_{12} = 0.5 \cdot 0 + 0.5 \cdot 1 = 0.5,$$
$$z_{21} = 0.33 \cdot 1 + 0.67 \cdot 1 = 1.0, \qquad z_{22} = 0.33 \cdot 0 + 0.67 \cdot 1 = 0.67.$$

Logo,

$$Z \approx \begin{bmatrix} 1.0 & 0.5 \\ 1.0 & 0.67 \end{bmatrix}.$$

O ponto central é que a saída de cada token já não é o seu embedding isolado, mas uma **combinação ponderada** dos vetores da sequência. Essa mistura é o mecanismo matemático pelo qual o contexto começa a emergir: sem regras linguísticas explícitas, apenas álgebra linear (produtos, exponenciais e normalização).

2.9 Feed-Forward Network: não linearidade por token

Após a self-attention, cada token já incorpora informação do restante da sequência. Mas há um detalhe importante: a atenção é, essencialmente, uma mistura linear (ponderada) de vetores. Para aumentar a capacidade de transformação, o Transformer aplica, em seguida, um bloco **feed-forward posição-wise** (*position-wise FFN*), que atua **independentemente em cada token**.

Em termos práticos: a atenção define **quais tokens** contribuem para cada posição; o FFN define **como** o vetor resultante é transformado (com não linearidade), token a token.

2.9.1 Definição matemática

Para um token com representação $\mathbf{x} \in \mathbb{R}^d$, o FFN é:

$$\mathrm{FFN}(\mathbf{x}) = \phi(\mathbf{x}W_1 + b_1)\, W_2 + b_2,$$

onde

$$W_1 \in \mathbb{R}^{d \times d_{ff}}, \quad W_2 \in \mathbb{R}^{d_{ff} \times d}.$$

No BERT-base, $d = 768$ e $d_{ff} = 3072$. Ou seja, o bloco expande a dimensão ($768 \to 3072$), aplica uma não linearidade (GELU) e comprime de volta ($3072 \to 768$).

2.9.2 Conexão direta com o código do BERT

No modelo do Hugging Face, o FFN aparece como duas projeções por token:

- projeção para o espaço intermediário + ativação (GELU):
  `bert_model.encoder.layer[i].intermediate.dense`
- projeção de volta para d:
  `bert_model.encoder.layer[i].output.dense`

Os tensores têm shape `(batch, seq_len, hidden_size)`. E o ponto-chave aqui é: **não há mistura entre tokens** nesse bloco. Cada posição é transformada separadamente.

2.10 Normalização: por que os vetores não podem crescer sem controle

Até aqui fizemos duas coisas: somamos caminhos residuais e misturamos informação via atenção. O efeito colateral é que os valores internos podem perder escala, variar demais, e tornar o treinamento instável conforme empilhamos camadas.

Esse não é um problema linguístico, mas sim um problema numérico típico de redes profundas. A normalização é o mecanismo que mantém os vetores em uma faixa "estável" sem comprometer a capacidade representacional do modelo.

2.10.1 O que pode dar errado sem normalização

Após atenção (ou FFN), cada token produz um vetor:

$$\mathbf{z}_i \in \mathbb{R}^d.$$

Ao longo de muitas camadas, podem ocorrer:

- crescimento de magnitude (valores explodindo);
- dominância de poucas dimensões;
- amplificação de pequenas variações;

- gradientes instáveis.

O papel da normalização é introduzir um **padrão estável** para a escala interna, tornando viável empilhar muitas camadas.

2.10.2 Layer Normalization: aplicada dentro do token

A LayerNorm é aplicada **por token**, usando as d dimensões daquele vetor (não misturando tokens, nem dependendo do batch).

Dado

$$\mathbf{z}_i = (z_{i,1}, z_{i,2}, \ldots, z_{i,d}),$$

calculamos:

Média

$$\mu_i = \frac{1}{d}\sum_{k=1}^{d} z_{i,k}$$

Variância

$$\sigma_i^2 = \frac{1}{d}\sum_{k=1}^{d}(z_{i,k} - \mu_i)^2$$

Normalização

$$\hat{z}_{i,k} = \frac{z_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

onde ε evita divisão por zero.

Após isso, o vetor fica aproximadamente com média zero e variância unitária, impedindo que uma dimensão "sequestre" a escala do restante.

2.10.3 Parâmetros aprendidos: γ e β

Normalizar demais seria estável, mas poderia limitar expressão. Por isso, o Transformer adiciona dois vetores treináveis:

$$\mathbf{y}_i = \gamma \odot \hat{\mathbf{z}}_i + \beta,$$

com $\gamma, \beta \in \mathbb{R}^d$ e $\odot$ elemento a elemento.

Esses parâmetros permitem ao modelo *reaprender* a escala útil, mantendo a estabilidade numérica sem "aplanar" representações.

2.10.4 Equação compacta

$$\text{LayerNorm}(\mathbf{z}_i) = \gamma \odot \frac{\mathbf{z}_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}} + \beta.$$

2.10.5 Pós-norm no BERT e pré-norm em modelos modernos

O BERT clássico usa **pós-normalização**:

$$\mathbf{y} = \mathrm{LayerNorm}\big(\mathbf{x} + F(\mathbf{x})\big),$$

visível em:

- `bert_model.encoder.layer[i].attention.output.LayerNorm`
- `bert_model.encoder.layer[i].output.LayerNorm`

Em arquiteturas mais recentes e muito profundas, é comum a **pré-normalização**:

$$\mathbf{y} = \mathbf{x} + F(\mathrm{LayerNorm}(\mathbf{x})),$$

que tende a facilitar a estabilidade do fluxo de gradientes em profundidade.

2.10.6 O que a normalização não faz

Para evitar confusão conceitual:

- LayerNorm **não cria contexto**;
- LayerNorm **não adiciona semântica**;
- LayerNorm **não é atenção**.

Ela apenas mantém os vetores numericamente controlados, permitindo que a atenção e o FFN funcionem de forma estável em muitas camadas.

Pontos para lembrar

- O Transformer transforma tokens discretos em vetores e aplica operações contínuas em $\mathbb{R}^d$.
- Self-attention calcula compatibilidades $QK^\top$ e produz uma mistura ponderada de valores via softmax.
- Residuais preservam caminho de sinal; normalização estabiliza escala e facilita treinamento profundo.
- O "contexto" emerge como fenômeno geométrico: a representação de cada token muda ao misturar informação de toda a sequência.

O que vem a seguir

No próximo capítulo, discutimos o que mudou de forma relevante nos Transformers modernos: não o operador de atenção em si, mas (i) a geometria em que ele atua (RoPE), (ii) a escala que controla nitidez e estabilidade numérica (scaling/temperatura) e (iii) o aumento de capacidade efetiva por token sem custo linear (Mixture of Experts).

Capítulo 3

Variações Modernas do Transformer

3.1 O que mudou e o que permaneceu invariável

Nos capítulos anteriores, apresentamos o núcleo conceitual do Transformer: embeddings, autoatenção, conexões residuais e normalização. Esse núcleo permanece essencialmente o mesmo nos modelos recentes. Em particular, a operação central continua sendo o cálculo de compatibilidades via produto escalar, seguido de uma normalização por *softmax* e uma combinação ponderada de vetores.

O que muda, nas arquiteturas modernas, é a resposta a três limitações práticas que se tornaram inevitáveis com o aumento de escala e com a adoção de contextos longos:

- como representar posição sem impor um comprimento máximo rígido para o contexto;
- como manter estabilidade numérica quando a dimensionalidade cresce e os logits variam demais;
- como aumentar capacidade efetiva sem que o custo computacional cresça linearmente com o número total de parâmetros ativos.

As variações discutidas neste capítulo atacam exatamente esses pontos. O mecanismo de atenção permanece. O que se altera é (i) a geometria em que os vetores vivem, (ii) a escala em que os produtos escalares operam e (iii) o número de parâmetros efetivamente usados por token.

3.2 Três refinamentos, um mesmo princípio

Atenção permanece. O que muda é o espaço (e o regime) em que ela opera. As arquiteturas modernas introduzem, de forma recorrente, três refinamentos:

- **Rotary Embeddings (RoPE):** posição representada como rotação geométrica aplicada a $\mathbf{Q}$ e $\mathbf{K}$;
- **Attention Scaling e temperatura:** controle explícito sobre a escala dos logits e, portanto, sobre a entropia do *softmax*;

- **Mixture of Experts (MoE):** aumento de capacidade por roteamento seletivo, ativando apenas parte dos parâmetros a cada token.

Essas ideias não redefinem a atenção, elas a refinam. Podemos resumir seus efeitos em três palavras-chave:

- **geometria** dos vetores (RoPE);
- **escala** dos produtos escalares (scaling / temperatura);
- **densidade de parâmetros por token** (MoE).

3.3 Um experimento-base para visualizar números

Antes de analisar cada variação, fixamos um experimento mínimo (reutilizado ao longo do notebook) com o objetivo de tornar observáveis, numericamente, fenômenos que em modelos grandes ficam ocultos pela escala e pela quantidade de detalhes.

3.3.1 Mini-corpus e embeddings explícitos

```
tokens = ["teddy", "bear", "reads"]

X = torch.tensor([
    [ 1.0, 0.0, 1.0, 0.0],
    [ 1.0, 1.0, 0.0, 1.0],
    [ 0.0, 1.0, -1.0, 1.0],
], device=device)

Q = X.clone()
K = X.clone()
positions = torch.arange(len(tokens), device=device)
```

O objetivo aqui não é simular semântica real, mas controlar a geometria. Esses vetores foram escolhidos para que:

- produtos escalares apresentem valores positivos, nulos e negativos;
- rotações mudem orientação sem alterar norma;
- efeitos de escala possam ser observados diretamente nos logits.

Ao fixar $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{X}$, isolamos o mecanismo de atenção sem introduzir projeções lineares adicionais. Assim, qualquer diferença observada nos logits será consequência direta de RoPE ou de mudanças de escala (quando aplicadas nas seções seguintes).

3.3.2 Leitura do código: o que está sendo controlado

Para evitar que o notebook se torne uma sucessão de resultados "mágicos", vale explicitar o que cada bloco está fixando.

Tokens como rótulos (não como significado)

```
tokens = ["teddy", "bear", "reads"]
```

Esses nomes funcionam como rótulos. O foco é o comportamento geométrico da atenção, não interpretação linguística. Com três tokens, obtemos uma matriz de compatibilidade 3×3, o caso mínimo para visualizar autoatenção (diagonal) e relações cruzadas.

Embeddings manuais em dimensão pequena

```
X = torch.tensor([
    [ 1.0, 0.0, 1.0, 0.0], # teddy
    [ 1.0, 1.0, 0.0, 1.0], # bear
    [ 0.0, 1.0, -1.0, 1.0], # reads
], device=device)
```

Criamos $\mathbf{X} \in \mathbb{R}^{3\times4}$ com $d_{\text{model}} = 4$. Essa escolha é deliberada: RoPE opera em pares de dimensões, portanto d deve ser par, e $d = 4$ permite visualizar duas rotações 2D independentes. Além disso, a dimensionalidade reduzida torna possível conferir contas manualmente.

Queries, keys e values iguais (experimento isolado)

```
Q = X.clone()
K = X.clone()
V = X.clone()
```

Neste experimento, definimos $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{X}$. Com isso, a matriz $\mathbf{QK}^\top$ é simétrica, o que facilita inspeção. Mais importante: eliminamos qualquer efeito de pesos aprendidos, deixando apenas a geometria dos vetores como fonte de variação.

Posições como parâmetro (não como vetor somado)

```
positions = torch.arange(len(tokens), device=device)
```

O vetor de posições é $[0, 1, 2]$. Em RoPE, essa informação não será somada a $\mathbf{X}$. Ela entrará como parâmetro de um operador de rotação aplicado aos vetores de $\mathbf{Q}$ e $\mathbf{K}$.

A partir daqui, todos os números observados são consequência direta dessas escolhas.

3.4 Rotary Embeddings (RoPE)

3.4.1 Por que embeddings posicionais aditivos são limitantes

No Transformer clássico, a posição é incorporada por soma:

$$\mathbf{X} = \mathbf{E}_{\text{token}} + \mathbf{E}_{\text{pos}}.$$

Esse mecanismo tem três limitações frequentes em contextos longos:

- a posição é tratada como **absoluta**, e não como relação entre tokens;
- o comprimento máximo de contexto precisa ser **fixado** durante treinamento;
- conteúdo e posição são misturados por **adição**, o que reduz controle geométrico sobre o espaço.

Quando o modelo precisa extrapolar para posições não observadas no treino, a representação posicional aditiva tende a degradar.

3.4.2 Ideia central: posição como operador geométrico

RoPE substitui a soma por um operador. Em vez de adicionar um vetor posicional, ele aplica uma rotação aos vetores de consulta e chave, transformando $\mathbf{Q}$ e $\mathbf{K}$ em versões rotacionadas dependentes da posição.

A atenção continua sendo:

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

O que muda é que o produto escalar passa a ocorrer entre vetores rotacionados: $\mathbf{Q} \mapsto \mathbf{Q}_r$ e $\mathbf{K} \mapsto \mathbf{K}_r$.

3.4.3 O bloco fundamental: rotação 2D

Para um par (x_1, x_2), a rotação por ângulo θ é:

$$R(\theta)\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1\cos\theta - x_2\sin\theta \\ x_1\sin\theta + x_2\cos\theta \end{bmatrix}.$$

Duas propriedades são essenciais:

- a rotação preserva norma (não "aumenta" nem "diminui" o vetor);
- a rotação altera orientação, permitindo codificar posição como deslocamento angular.

Por que a norma é preservada

Se $\mathbf{y} = R(\theta)\mathbf{x}$, então $\|\mathbf{y}\| = \|\mathbf{x}\|$. A prova decorre da identidade $\cos^2\theta + \sin^2\theta = 1$ e do cancelamento dos termos cruzados. Esse detalhe é importante: RoPE não muda a magnitude de compatibilidade por si só; ele muda como vetores se alinham entre posições diferentes.

3.4.4 RoPE em dimensão real: rotações em pares

No notebook, usamos $d = 4$ para enxergar tudo com clareza. Isso significa que o vetor de cada token é rotacionado em dois planos 2D:

$$(x_1, x_2) \text{ e } (x_3, x_4).$$

Além disso, RoPE define frequências diferentes por par de dimensões. Com `base` $= 10000$, e $d = 4$, obtemos dois pares com frequências aproximadas:

$$\omega_0 = 1, \qquad \omega_1 = 0{,}01.$$

O ângulo do par j na posição p é:

$$\theta_j(p) = p \cdot \omega_j.$$

Isso cria um mecanismo de posição em múltiplas escalas: o primeiro par gira "rápido", enquanto o segundo gira lentamente.

3.4.5 Implementação: RoPE aplicado a Q e K

A função abaixo implementa exatamente a ideia anterior:

```
def rope_apply(q_or_k, positions, base=10000.0):
    d = q_or_k.shape[-1]
    i = torch.arange(0, d, 2, device=q_or_k.device).float()
    inv_freq = 1.0 / (base ** (i / d))
    angles = positions[:, None].float() * inv_freq[None, :]

    cos = torch.cos(angles)
    sin = torch.sin(angles)

    x1 = q_or_k[:, 0::2]
    x2 = q_or_k[:, 1::2]

    y1 = x1 * cos - x2 * sin
    y2 = x1 * sin + x2 * cos

    out = torch.empty_like(q_or_k)
    out[:, 0::2] = y1
    out[:, 1::2] = y2
    return out
```

Observe que não há embedding posicional explícito. A posição entra como parâmetro que determina os ângulos da rotação.

3.5 O efeito de RoPE nos logits

Para tornar o efeito visível, comparamos logits com e sem rotação:

```
def attn_logits(Q, K):
    d = Q.shape[-1]
    return (Q @ K.T) / math.sqrt(d)

logits_plain = attn_logits(Q, K)
logits_rope = attn_logits(rope_apply(Q, positions),
                          rope_apply(K, positions))
```

Com $d = 4$, temos $\sqrt{d} = 2$, e os logits são:

$$\mathbf{L} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}.$$

3.5.1 Sem RoPE: produtos escalares normalizados

Como $\mathbf{Q} = \mathbf{K}$, calculamos $\mathbf{Q}\mathbf{Q}^\top$. Por exemplo, para a entrada $(0, 1)$:

$$\mathbf{q}_0 = [1, 0, 1, 0], \quad \mathbf{q}_1 = [1, 1, 0, 1] \Rightarrow \mathbf{q}_0 \cdot \mathbf{q}_1 = 1 \Rightarrow L_{0,1} = \frac{1}{2} = 0{,}5.$$

E para $(0, 2)$:

$$\mathbf{q}_2 = [0, 1, -1, 1] \Rightarrow \mathbf{q}_0 \cdot \mathbf{q}_2 = -1 \Rightarrow L_{0,2} = \frac{-1}{2} = -0{,}5.$$

O notebook exibe:

```
Logits sem RoPE:
[[ 1.0000, 0.5000, -0.5000],
 [ 0.5000, 1.5000, 1.0000],
 [-0.5000, 1.0000, 1.5000]]
```

3.5.2 Com RoPE: compatibilidade passa a incorporar posição

Ao aplicar RoPE, comparamos $\mathbf{Q}_r$ e $\mathbf{K}_r$. Como $\mathbf{K}_r = \mathbf{Q}_r$ neste experimento, a estrutura permanece simétrica, mas os valores mudam.

A entrada $(0, 1)$, por exemplo, torna-se:

$$L_{0,1}^{(\text{RoPE})} \approx -0{,}1556,$$

enquanto $(0, 2)$ se torna:

$$L_{0,2}^{(\text{RoPE})} \approx -0{,}9645.$$

O notebook exibe:

```
Logits com RoPE:
[[ 1.0000, -0.1556, -0.9645],
 [-0.1556, 1.5000, 0.3444],
 [-0.9645, 0.3444, 1.5000]]
```

A diagonal permanece alta (autoatenção preservada), mas relações entre posições diferentes são moduladas pela diferença de ângulos, isto é, pela diferença de posições.

3.5.3 Por que RoPE produz efeito relativo

No caso 2D, vale a identidade:

$$(R(\alpha)\mathbf{u})^\top (R(\beta)\mathbf{v}) = \mathbf{u}^\top R(\beta - \alpha)\mathbf{v}.$$

Essa expressão captura a intuição central: cada vetor é rotacionado pela sua posição, e o produto escalar passa a depender de $\beta - \alpha$, ou seja, da diferença de posições. Assim, o modelo não precisa "aprender" uma tabela de posições absolutas; a posição relativa emerge naturalmente na compatibilidade geométrica.

3.6 Como interpretar o heatmap dos logits

O heatmap visualiza a matriz de logits

$$\mathbf{L} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}},$$

onde cada célula (i, j) é:

$$L_{i,j} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d}}.$$

Neste estágio, ainda não aplicamos *softmax*. Portanto, o heatmap representa o **campo bruto de similaridade geométrica**.

3.6.1 Leitura estrutural (linhas, colunas e diagonal)

- **linha i:** token que consulta (*query*);
- **coluna j:** token consultado (*key*);
- **diagonal:** $\mathbf{q}_i^\top \mathbf{k}_i = \|\mathbf{q}_i\|^2$, normalmente dominante.

Sem RoPE, o mapa reflete essencialmente a similaridade dos vetores "como são". Com RoPE, o mapa passa a refletir similaridade *após* cada vetor ser rotacionado por sua posição. Isso explica por que certas compatibilidades mudam de sinal ou magnitude: a geometria do espaço foi alterada, mas a atenção continua sendo o mesmo mecanismo.

RoPE não injeta posição como dado. Ele faz a posição emergir no produto escalar.

3.7 Attention Scaling e Temperatura

3.7.1 O problema numérico da atenção em altas dimensões

A atenção começa com a matriz de escores brutos:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top, \qquad S_{i,j} = \mathbf{q}_i^\top \mathbf{k}_j.$$

Cada entrada $S_{i,j}$ é um produto escalar entre vetores de dimensão d_k. Sob hipóteses usuais de inicialização e de componentes aproximadamente independentes, a variância desses produtos escalares cresce com a dimensão:

$$\mathbb{V}[S_{i,j}] \propto d_k.$$

Em termos práticos, quanto maior d_k, mais dispersos tendem a ser os logits.

Essa dispersão tem uma consequência direta: o *softmax* passa a operar em regime saturado. Logits muito separados produzem distribuições quase determinísticas, com gradientes muito pequenos e comportamento numericamente instável.

3.7.2 A correção clássica: normalização por $\sqrt{d_k}$

A estabilização introduzida no Transformer é:

$$\mathbf{S}' = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}.$$

Essa divisão normaliza a escala esperada dos logits. De forma aproximada,

$$\mathbb{V}\left[\frac{S_{i,j}}{\sqrt{d_k}}\right] \approx \mathcal{O}(1),$$

o que torna o regime do *softmax* comparável ao longo de diferentes dimensões latentes.

No notebook, essa afirmação aparece de forma empírica ao comparar o desvio-padrão dos logits antes e depois da normalização. Sem escala, o desvio cresce; com a divisão por $\sqrt{d_k}$, ele permanece controlado:

```
dk= 16 std(raw)=3.661 std(scaled)=0.915
dk= 64 std(raw)=9.061 std(scaled)=1.133
dk= 256 std(raw)=15.276 std(scaled)=0.955
dk=1024 std(raw)=29.405 std(scaled)=0.919
```

A divisão por $\sqrt{d_k}$ não é heurística. Ela é uma normalização estatística necessária para estabilizar o softmax.

3.7.3 Escala, *softmax* e entropia da atenção

A distribuição de atenção (para uma linha de logits) é:

$$p_j = \frac{\exp(S'_j)}{\sum_k \exp(S'_k)}.$$

Uma forma objetiva de medir quão concentrada está essa distribuição é a entropia:

$$H(p) = -\sum_j p_j \log p_j.$$

Entropia alta indica atenção difusa (probabilidades semelhantes). Entropia baixa indica atenção concentrada (poucos tokens dominam).

O efeito de escala aparece diretamente quando multiplicamos os logits por um fator $s > 0$, isto é, quando avaliamos $\text{softmax}(s \cdot \mathbf{s})$ para um vetor fixo $\mathbf{s}$. No notebook, a entropia cai monotonicamente quando s cresce:

```
[scale=0.5] entropy=1.0553
[scale=1.0] entropy=0.9528
[scale=2.0] entropy=0.7139
[scale=4.0] entropy=0.3801
```

Essa evidência é importante por um motivo: a atenção não é apenas "um mecanismo interno", ela é um operador cujo regime pode ser regulado continuamente por escala.

3.7.4 Temperatura como generalização contínua do regime

A temperatura τ introduz um controle explícito sobre a nitidez do *softmax*:

$$\text{softmax}\left(\frac{\mathbf{S}'}{\tau}\right).$$

Dividir por τ equivale a multiplicar os logits por $1/\tau$. Logo:

$$\tau \downarrow \quad \Longleftrightarrow \quad \text{escala} \uparrow$$

e, portanto, τ menor tende a concentrar a distribuição (menor entropia), enquanto τ maior tende a suavizá-la (maior entropia).

Essa interpretação permite ler o gráfico de temperatura do notebook como uma família de regimes da mesma equação:

- τ grande: probabilidades próximas e atenção difusa;
- τ intermediário: regime padrão, com preferência moderada;
- τ pequeno: colapso progressivo para um token dominante.

Temperatura não cria nem destrói informação.
Ela regula a nitidez da decisão do softmax.

3.8 Mixture of Experts (MoE)

3.8.1 Motivação: capacidade sem custo linear por token

Em Transformers densos, uma fração grande dos parâmetros costuma estar no bloco *feed-forward* (FFN). Um FFN típico pode ser escrito como:

$$\text{FFN}(\mathbf{x}) = W_2\, \sigma(W_1 \mathbf{x}),$$

onde $W_1 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ e $W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$.

Aumentar capacidade, nesse regime, significa aumentar d_{ff}, o que aumenta custo computacional para *todo token*. Esse crescimento rapidamente se torna inviável quando se busca aumentar parâmetros totais mantendo custo de inferência controlado.

3.8.2 Definição: especialistas e roteamento top-k

A ideia central de MoE é substituir um FFN denso por um conjunto de E especialistas $\{\text{Expert}_1, \ldots, \text{Expert}_E\}$, cada um sendo, essencialmente, um FFN independente.

A saída do bloco MoE é:

$$\text{FFN}_{\text{MoE}}(\mathbf{x}) = \sum_{e \in \mathcal{E}(\mathbf{x})} g_e(\mathbf{x}) \cdot \text{Expert}_e(\mathbf{x}),$$

onde:

- $\mathcal{E}(\mathbf{x})$ é o conjunto top-k de especialistas selecionados para o token;
- $g_e(\mathbf{x}) \geq 0$ são pesos produzidos pelo roteador;
- $\sum_{e \in \mathcal{E}(\mathbf{x})} g_e(\mathbf{x}) = 1$ (no caso de pesos normalizados sobre os escolhidos).

Apenas $k \ll E$ especialistas são ativados por token. Assim, a capacidade total do modelo pode crescer com E, sem que o custo por token cresça na mesma proporção.

3.8.3 O roteador como operador (e a analogia com atenção)

O roteador define uma distribuição sobre especialistas:

$$\mathbf{p}(\mathbf{x}) = \text{softmax}(W_g \mathbf{x}), \quad W_g \in \mathbb{R}^{E \times d_{\text{model}}}.$$

No caso $k = 1$, a escolha é:

$$e^*(\mathbf{x}) = \arg\max_e p_e(\mathbf{x}).$$

Uma analogia útil é pensar que o roteamento se parece com um passo de atenção, em que o token atua como *query* e os especialistas funcionam como alternativas ("keys") a serem selecionadas. A diferença é que, em MoE, a decisão usualmente é esparsa (top-k), de modo que a computação efetiva é restrita a poucos caminhos.

MoE pode ser visto como uma forma de seleção esparsa (top-k) sobre um conjunto de especialistas computacionais.

3.8.4 Carga (load) e a patologia do desbalanceamento

Uma variável central em MoE é a carga de cada especialista:

$$\text{load}(e) = \sum_{t=1}^{T} \mathbb{I}\big[e = e^*(\mathbf{x}_t)\big],$$

onde T é o número de tokens no batch.

Para $k = 1$, vale a conservação:

$$\sum_{e=1}^{E} \text{load}(e) = T.$$

No experimento toy do notebook, observamos:

$$\text{load} = [0,\ 2,\ 0,\ 1], \quad T = 3,\ E = 4,$$

o que confirma diretamente a identidade $0 + 2 + 0 + 1 = 3$.

Esse exemplo pequeno já ilustra dois efeitos comuns: **colapso de roteamento** e **desbalanceamento de carga**. Quando poucos especialistas recebem a maior parte dos tokens, parte da capacidade fica subutilizada e alguns especialistas podem receber menos sinal de treino.

3.8.5 Ganho de capacidade: total versus ativo

Se cada especialista tem P parâmetros, então:

$$P_{\text{total}} = E \cdot P.$$

Mas, se apenas k especialistas são ativados por token, o custo por token cresce como:

$$P_{\text{ativo}} = k \cdot P.$$

Logo, o MoE desacopla capacidade total de custo por token: é possível aumentar E (capacidade total) sem multiplicar proporcionalmente o custo de inferência.

3.8.6 Balanceamento em modelos reais

Para treinar MoEs em escala, implementações práticas adicionam mecanismos auxiliares para evitar que o roteador colapse em poucos especialistas. Uma estratégia típica é introduzir perdas de balanceamento que penalizam distribuições muito desiguais, por exemplo:

$$\mathcal{L}_{\text{balance}} = \sum_{e} \left(\frac{\text{load}(e)}{T} - \frac{1}{E} \right)^2 .$$

A mensagem prática é que MoE não é apenas "mais parâmetros": ele exige engenharia de treinamento para manter uso efetivo e estável dos especialistas.

3.9 Síntese do capítulo: geometria, escala e capacidade

As variações modernas discutidas neste capítulo operam em níveis distintos do Transformer:

- **RoPE** atua antes do produto escalar, alterando a geometria de **Q** e **K** para que posição relativa emerja na compatibilidade;
- **Scaling e temperatura** atuam sobre os logits, estabilizando o *softmax* e regulando o regime (difuso versus concentrado) da atenção;
- **MoE** atua principalmente no FFN, aumentando capacidade total por roteamento seletivo, com custo controlado por token.

Transformers modernos não reinventam a atenção, eEles refinam o espaço, a escala e a capacidade.

Tabela 3.1: Comparação estrutural das variações modernas do Transformer (com fontes primárias)

Técnica	**Problema resolvido (essência)**	**Modelo(s) emblemático(s)**	**Custo extra**	**Artigo(s) base (científico)**
RoPE	Representar **posição relativa** via rotação (sem soma de embeddings posicionais), favorecendo extrapolação e controle geométrico do produto escalar em atenção.	Família LLaMA Qwen Gemma (e afins)	Praticamente zero (opera como rotação em Q, K).	RoFormer / Rotary Position Embedding (Su et al.)
Scaling / Temperatura	Evitar **saturação do softmax** em altas dimensões: estabiliza a variância esperada de $QK^\top$ com $\frac{1}{\sqrt{d_k}}$; temperatura regula a nitidez (entropia) da distribuição.	Padrão em quase todos os Transformers	Zero (apenas escala dos logits).	Scaled dot-product attention (Vaswani et al.)
MoE	Aumentar **capacidade total** sem custo por-token linear: ativa apenas $k \ll E$ especialistas por token; requer controle de balanceamento de carga para treinar em escala.	Mixtral DeepSeek (MoE) (e MoEs modernos)	Roteador + perdas de balanceamento + overhead de comunicação (em escala).	Sparsely-Gated MoE (Shazeer et al.) Switch / GShard (Fedus et al.; Lepikhin et al.)

Exercício de consolidação

Modifique o mini-corpus do notebook alterando apenas o vetor do token `reads` para:

$$\mathbf{x}_{\text{reads}} = [1,\ -1,\ 0,\ 0].$$

Recalcule:

- os logits sem RoPE;
- os logits com RoPE;
- o heatmap correspondente em ambos os casos.

Em seguida, verifique:

> Como a penalização posicional muda para tokens mais distantes? A diagonal permanece preservada?

Limitações práticas

Apesar do rigor matemático, as técnicas discutidas possuem limitações em modelos reais de grande escala.

Limitações do RoPE: Em contextos muito longos (por exemplo, acima de dezenas de milhares de tokens), pode haver degradação sem ajustes adicionais, como variantes e técnicas de *scaling* do parâmetro θ.

Limitações do MoE: MoEs sofrem com risco de *routing collapse* e subutilização persistente de especialistas sem perdas auxiliares e mecanismos explícitos de balanceamento.

Pontos para lembrar

- **O núcleo do Transformer não mudou:** atenção continua sendo produto escalar $\rightarrow$ softmax $\rightarrow$ mistura vetorial. As variações modernas refinam **como** esse núcleo opera em prática.
- **RoPE (Rotary Position Embeddings)** não soma posição ao embedding; ele aplica rotações em pares de dimensões em Q e K, preservando norma e fazendo **posição relativa** emergir naturalmente no produto escalar.
- **Attention scaling** $(1/\sqrt{d_k})$ é uma normalização **estatística**: sem ela, a variância dos logits cresce com d_k, o softmax satura e os gradientes degradam.
- **Temperatura** τ controla a **nitidez** da distribuição: $\tau \downarrow$ concentra a atenção (entropia menor); $\tau \uparrow$ suaviza (entropia maior). Ela não altera Q, K, V, apenas a escala dos logits.
- **MoE (Mixture of Experts)** desloca capacidade para o bloco FFN: muitos especialistas existem, mas apenas $k \ll E$ são ativados por token, aumentando capacidade total sem custo linear por token.
- **Roteamento em MoE é uma decisão discreta** (top-k) e pode sofrer **colapso de carga**; por isso, modelos reais usam mecanismos de balanceamento para evitar especialistas ociosos/sobrecarregados.

O que vem a seguir

No próximo capítulo, saímos das **mudanças estruturais** do Transformer e entramos no **controle do comportamento em inferência**: como um modelo causal transforma logits em texto, token a token. Vamos formalizar $\Pr(x_{t+1} \mid x_{\leq t})$ e mostrar, com matemática e experimentos, três controles centrais: **temperatura** (concentração da distribuição), **top-p** (recorte por massa de probabilidade) e **KV cache** (redução de custo computacional sem alterar a distribuição-alvo). Também discutimos como **o prompt muda os logits** e por que heurísticas como "step by step" afetam estilo, mas não garantem correção.

Capítulo 4

Inferência em Modelos Causais

4.1 Abertura: o que será controlado na inferência

Até aqui, usamos Transformers principalmente como **modelos de representação**: dado um texto, queremos vetores contextualizados úteis para compreensão (classificação, recuperação, extração, etc.). Agora mudamos de regime: entramos em **inferência gerativa**, isto é, um modelo que produz texto **um token por vez**.

O objetivo deste capítulo é tornar explícitos três controles fundamentais sobre a distribuição do próximo token:

$$\Pr(x_{t+1} \mid x_{\leq t}).$$

Ao final, você deve conseguir enxergar, matematicamente e no código, que:

1. **Temperatura** controla o quão *concentrada* ou *difusa* fica a distribuição do próximo token.

2. **Top-p (nucleus sampling)** controla *quantos candidatos* entram no sorteio, com base em massa de probabilidade acumulada.

3. **KV cache** reduz o custo computacional na geração ao evitar recomputações do prefixo já processado.

Cada parâmetro altera explicitamente uma distribuição probabilística e, portanto, altera o comportamento observável do gerador.

4.2 Transição conceitual: de BERT para GPT

BERT e GPT pertencem à mesma família (Transformers), mas foram otimizados para tarefas diferentes.

Modelo	Tipo de Transformer	Objetivo principal
BERT	Encoder bidirecional	Compreensão (representação)
GPT-2	Decoder causal	Geração autoregressiva

4.2.1 Diferença estrutural essencial: atenção bidirecional vs causal

No **encoder bidirecional** (BERT), cada token pode atender a tokens à esquerda *e* à direita. Em forma canônica, a atenção é:

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

No **decoder causal** (GPT-2), o modelo não pode olhar para o futuro. Para garantir isso, introduzimos uma **máscara causal**:

$$\mathrm{Attn}_{\mathrm{causal}}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top + M}{\sqrt{d}}\right) V,$$

onde

$$M_{ij} = \begin{cases} 0, & j \le i, \\ -\infty, & j > i. \end{cases}$$

Interpretação. O termo M zera a atenção para posições futuras (após o softmax, elas recebem probabilidade 0). Assim, ao prever o token x_{t+1}, o modelo usa apenas o prefixo $x_1, \ldots, x_t$.

BERT entende texto. GPT gera texto.
A diferença não é "ter atenção", mas **restringir** a atenção.

4.2.2 Por que começar com GPT-2

Vamos usar GPT-2 por um motivo didático: ele é uma implementação **transparente** do decoder causal clássico que está na base de muitos geradores modernos. Os princípios são os mesmos em modelos atuais; o que muda é escala, engenharia e pós-treinamento.

4.2.3 O que mudou nos GPTs modernos (sem mudar o princípio)

O princípio fundamental permaneceu:

$$\text{gerar texto} \;\equiv\; \text{amostrar } x_{t+1} \sim \Pr(\cdot \mid x_{\le t}).$$

O que foi refinado na prática inclui:

- **Escala**: mais camadas, parâmetros e dados.
- **Tokenização**: BPEs e vocabulários mais robustos.
- **Estabilidade**: variantes de normalização e inicializações.
- **Atenção otimizada**: kernels fundidos e implementações eficientes (ex.: FlashAttention).
- **KV cache avançado**: inferência em streaming e melhor reutilização do prefixo.
- **Pós-treinamento/alinhamento**: instruções, preferências e feedback humano.

Ponto central. Mesmo com toda a engenharia moderna, um gerador causal continua sendo, no fim das contas, um mecanismo de previsão do próximo token condicionado ao passado.

4.3 Comparação visual: Transformer genérico, BERT e GPT-2

A Figura 4.1 compara, lado a lado, a estrutura-base do Transformer e duas especializações muito comuns: BERT (encoder bidirecional) e GPT-2 (decoder causal). Os blocos principais são os mesmos (atenção + MLP/FFN); o que muda é (1) a máscara de atenção e (2) o objetivo do modelo.

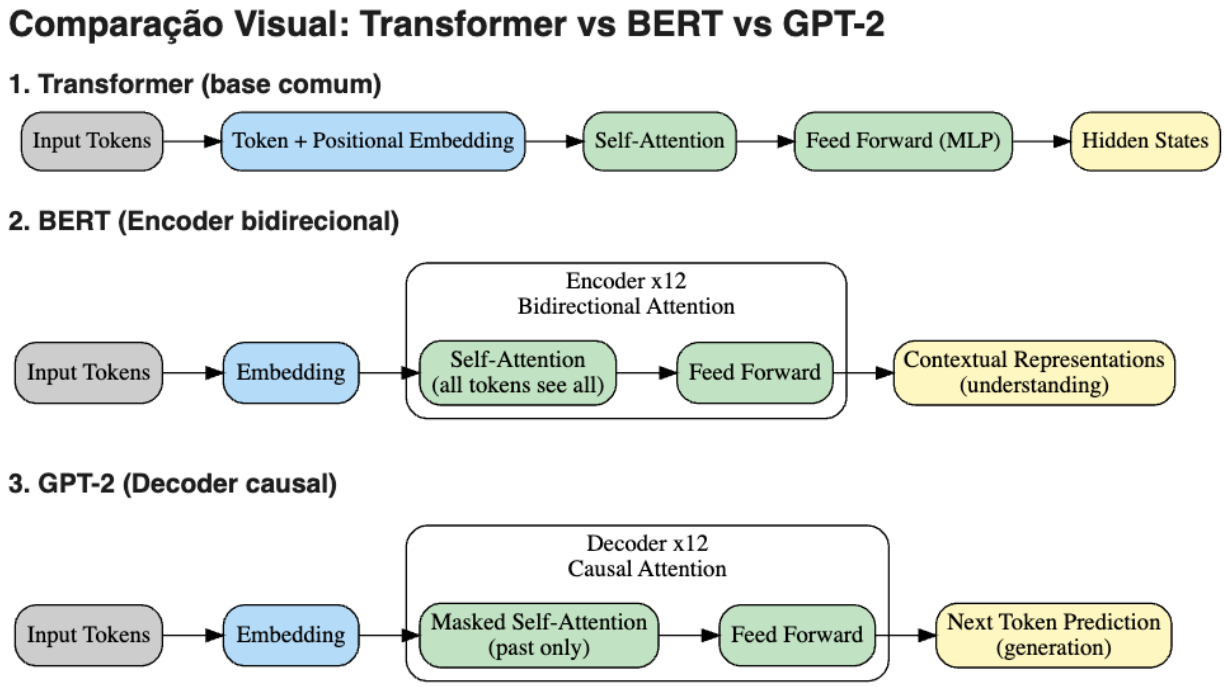

Figura 4.1: Comparação visual entre (i) um Transformer genérico, (ii) BERT (encoder bidirecional) e (iii) GPT-2 (decoder causal). O núcleo computacional é o mesmo; a diferença central está na restrição causal da atenção e no tipo de saída/objetivo.

4.3.1 Como ler os diagramas

Os três diagramas usam a mesma convenção:

- leitura da esquerda para a direita: fluxo do *forward pass*;
- azul: embeddings (token + posição, e às vezes segmento);
- verde: computação principal (atenção e MLP/FFN);
- amarelo: o que o modelo entrega como saída (representação ou logits);

- "x12" indica repetição do mesmo bloco (profundidade).

4.3.2 Transformer genérico: o bloco comum

O caso genérico mostra o padrão básico: embeddings → self-attention → MLP/FFN. A saída aqui é um *hidden state* por token: vetores que vão ficando mais contextualizados camada a camada.

4.3.3 BERT: encoder bidirecional

No BERT, a atenção é **bidirecional**: cada token pode olhar para tokens à esquerda e à direita. Isso é útil para *compreensão*, porque a representação final de cada token incorpora contexto dos dois lados.

4.3.4 GPT-2: decoder causal

No GPT-2, a atenção é **causal**: cada posição só pode olhar para o passado. A saída é conectada a uma projeção para o vocabulário (logits), o que permite geração autoregressiva (token a token).

4.4 Do diagrama ao PyTorch: conectando visual e implementação

A figura dá o mapa em alto nível. Agora a meta é reconhecer esse mesmo mapa na implementação do notebook: embeddings, pilha de blocos (x12), atenção, MLP/FFN e a camada de saída.

4.5 GPT-2: decoder causal na prática

4.5.1 Entrada (cinza): `input_ids`

A caixa "Input Tokens" representa os índices inteiros fornecidos ao *forward*:

```
input_ids: LongTensor [batch_size, seq_len]
```

4.5.2 Embeddings (azul): `wte` e `wpe`

O embedding de tokens aparece como:

```
(wte): Embedding(50257, 768)
```

ou seja, vocabulário de 50,257 tokens e dimensão vetorial 768.

O embedding posicional aparece como:

```
(wpe): Embedding(1024, 768)
```

indicando comprimento máximo 1024 no GPT-2 clássico.

No forward, a composição é:

$$h_0 = \text{wte}(x) + \text{wpe}(p).$$

4.5.3 Profundidade (cluster): `Decoder x12` como `ModuleList`

A repetição do bloco "Decoder x12" corresponde a:

```
(h): ModuleList
  (0-11): 12 x GPT2Block
```

4.5.4 Atenção causal (verde): `GPT2Attention`

Dentro de cada bloco, a atenção aparece como:

```
(attn): GPT2Attention
  (c_attn): Conv1D(2304, 768)
  (c_proj): Conv1D(768, 768)
```

A máscara causal não é listada como uma "camada" separada, mas é aplicada internamente para garantir que

$$x_t \text{ só veja } x_{\leq t}.$$

4.5.5 MLP/FFN (verde): `GPT2MLP`

O feed-forward aparece como:

```
(mlp): GPT2MLP
  (c_fc): Conv1D(3072, 768)
  (c_proj): Conv1D(768, 3072)
```

Isso implementa a expansão típica:

$$768 \to 3072 \to 768.$$

4.5.6 Saída (amarelo): `lm_head` e predição do próximo token

A geração aparece explicitamente na projeção para o vocabulário:

```
(lm_head): Linear(768 -> 50257)
```

Essa camada produz logits $\ell \in^{|\mathcal{V}|}$, que serão convertidos em probabilidades por softmax:

$$p = \text{softmax}(\ell), \qquad x_{t+1} \sim p.$$

4.6 BERT: encoder bidirecional na prática

Ao inspecionar a configuração do modelo (`BertConfig`), é possível recuperar os hiperparâmetros centrais da arquitetura, que se conectam diretamente aos blocos discutidos anteriormente:

- **num_hidden_layers = 12** → Encoder x12
- **num_attention_heads = 12** → multi-head attention
- **hidden_size = 768** → dimensão vetorial
- **intermediate_size = 3072** → FFN/MLP

A diferença central em relação ao GPT-2 não está nos blocos em si, mas na **restrição de atenção**:

- BERT: atenção **bidirecional** (sem máscara causal).
- GPT-2: atenção **causal** (com máscara causal).

Como consequência, o uso típico também muda: BERT prioriza **representações** (compreensão), enquanto GPT-2 prioriza **geração** (logits e amostragem).

A partir daqui, entramos em dois temas práticos: (i) **como controlar a geração** (temperatura, top-p) e (ii) **como acelerar a geração** (KV cache).

4.7 O que você deve levar desta comparação

BERT e GPT-2 reaproveitam o mesmo núcleo (atenção + MLP/FFN). A diferença prática aparece na máscara de atenção (bidirecional vs causal) e no objetivo (representação vs geração).

4.8 O problema matemático da geração

Em um modelo causal, dada uma sequência $x_1, \ldots, x_t$, o modelo produz um vetor de logits $\mathbf{z}_t \in \mathbb{R}^{|V|}$ para o próximo token. A distribuição é:

$$\Pr(x_{t+1} = v \mid x_{\leq t}) = \text{softmax}(\mathbf{z}_t)_v = \frac{\exp(z_{t,v})}{\sum_{u \in V} \exp(z_{t,u})},$$

onde V é o vocabulário e $z_{t,v}$ é o logit associado ao token v.

Ponto de atenção: Logit não é probabilidade. Logits podem ser negativos e arbitrariamente grandes em magnitude. A interpretação probabilística aparece apenas após o softmax.

4.9 Temperatura

4.9.1 Definição formal

A temperatura $\tau > 0$ modifica a distribuição do próximo token por:

$$\Pr_{\tau}(x_{t+1} = v \mid x_{\leq t}) = \text{softmax}\Big(\frac{\mathbf{z}_t}{\tau}\Big)_v = \frac{\exp(z_{t,v}/\tau)}{\sum_{u \in V} \exp(z_{t,u}/\tau)}.$$

4.9.2 Prova: temperatura baixa concentra; temperatura alta espalha

Considere dois tokens a e b com $z_a > z_b$. A razão entre probabilidades é:

$$\frac{p_\tau(a)}{p_\tau(b)} = \frac{\exp(z_a/\tau)}{\exp(z_b/\tau)} = \exp\left(\frac{z_a - z_b}{\tau}\right).$$

Como $z_a - z_b > 0$, obtemos:

- Se τ **diminui**, $\frac{z_a - z_b}{\tau}$ **aumenta** e a razão cresce: o maior logit domina cada vez mais (distribuição mais concentrada).
- Se τ **aumenta**, a razão se aproxima de 1: as probabilidades ficam mais uniformes (distribuição mais difusa).

Em linguagem prática:

$\tau \downarrow \Rightarrow$ amostragem mais determinística, $\qquad \tau \uparrow \Rightarrow$ amostragem mais exploratória.

4.9.3 O que observar nos outputs

Com τ muito baixo, o modelo tende a repetir padrões altamente prováveis e pode "travar" em estruturas recorrentes. Com τ mais alto, surgem continuações mais diversas — mas também cresce o risco de incoerência local.

4.10 Top-p (Nucleus Sampling)

4.10.1 Definição formal

Top-p não seleciona "os k maiores" (isso seria top-k). Ele seleciona o menor conjunto $S_p \subset V$ cuja massa acumulada atinge pelo menos p:

$$S_p = \min\left\{ S \subset V \ : \ \sum_{v \in S} p(v) \geq p \right\}, \qquad p(v) = \text{softmax}(\mathbf{z}_t)_v.$$

Em seguida, renormalizamos dentro de S_p:

$$\tilde{p}(v) = \begin{cases} \frac{p(v)}{\sum_{u \in S_p} p(u)}, & v \in S_p \\ 0, & v \notin S_p. \end{cases}$$

Leitura prática:

- p pequeno (ex.: 0.3) $\Rightarrow$ poucos candidatos $\Rightarrow$ texto mais repetitivo.
- p grande (ex.: 0.9) $\Rightarrow$ mais candidatos $\Rightarrow$ maior diversidade.

4.11 KV cache

4.11.1 O que é recalculado sem cache

Em um Transformer causal, cada novo token exige computar, em cada camada, projeções para Q, K e V:

$$\mathbf{Q}_t = \mathbf{h}_t W_Q, \quad \mathbf{K}_t = \mathbf{h}_t W_K, \quad \mathbf{V}_t = \mathbf{h}_t W_V.$$

Sem cache, ao avançar de t para $t+1$, o modelo recomputa K e V do prefixo inteiro, embora o prefixo não mude.

4.11.2 Ideia do cache

Com KV cache, armazenamos:

$$K_{\leq t},\ V_{\leq t}.$$

No passo seguinte, computamos apenas K_{t+1}, V_{t+1} e concatenamos ao cache. O texto gerado **não muda** por causa do cache; o que muda é a **latência**.

4.11.3 Leitura do experimento

Nos notebooks do material complementar (`https://github.com/brunoleomenezes/MC-CD05/`), medimos o tempo de geração com e sem KV cache. Os valores variam conforme hardware e condições de execução, mas tipicamente observamos uma redução de latência, por exemplo:

```
[KV CACHE  TEMPO]
Com cache : 1.1692 s
Sem cache : 1.2143 s
```

A diferença pode ser pequena em modelos e sequências curtas (e sujeita a ruído de execução), mas a lei geral permanece: quanto maior o modelo e o contexto, maior o ganho.

4.12 Código: geração controlada e interpretação

A proposta experimental é: fixamos um prompt e alteramos apenas os controles. A lição é sempre a mesma, o texto muda porque a distribuição muda.

4.12.1 Carregando o GPT-2

```
gpt_name = "gpt2"
gpt_tokenizer = AutoTokenizer.from_pretrained(gpt_name)
gpt_model = AutoModelForCausalLM.from_pretrained(gpt_name)
gpt_model.to(device)
gpt_model.eval()

print("\n[GPT] Modelo carregado:", gpt_name)
```

4.12.2 Função de geração com temperatura, top-p e cache

```
def generate_text(prompt, max_new_tokens=80, temperature=1.0, top_p=1.0, use_cache=
    True):
    inputs = gpt_tokenizer(prompt, return_tensors="pt").to(device)

    with torch.no_grad():
        output = gpt_model.generate(
            **inputs,
            max_new_tokens=max_new_tokens,
            do_sample=True,
            temperature=temperature,
            top_p=top_p,
            use_cache=use_cache,
            pad_token_id=gpt_tokenizer.eos_token_id
        )

    return gpt_tokenizer.decode(output[0], skip_special_tokens=True)
```

Leitura conceitual dos argumentos:

- `do_sample=True`: amostra da distribuição (não escolhe sempre).
- `temperature`: implementa softmax($\mathbf{z}/\tau$).
- `top_p`: implementa recorte S_p e renormalização $\tilde{p}$.
- `use_cache`: ativa o KV cache (acelera, não muda a distribuição-alvo).

4.12.3 Teste base

```
prompt = "Transformers are models that"
print("\n[PROMPT]")
print(prompt)

print("\n[GERAÇÃO BASE]")
print(generate_text(prompt, temperature=0.7, top_p=0.9))
```

4.12.4 Experimento: temperatura

```
for temp in [0.2, 0.7, 1.0]:
    print("\n==============================")
    print(f"[TEMPERATURA = {temp}]")
    print("==============================")
    print(generate_text(prompt, temperature=temp, top_p=0.9))
```

A leitura correta é probabilística: τ altera a razão relativa entre logits e, portanto, altera quais padrões dominam a amostragem.

4.12.5 Experimento: top-p

```
for p in [0.3, 0.6, 0.9]:
    print("\n==============================")
    print(f"[TOP-P = {p}]")
    print("==============================")
    print(generate_text(prompt, temperature=0.7, top_p=p))
```

Com *top-p* baixo, o conjunto S_p pode ficar tão pequeno que o modelo passa a repetir estruturas muito prováveis. Com *top-p* alto, mais alternativas entram no sorteio, e o texto muda de direção.

4.12.6 Experimento: KV cache (tempo)

```
def timed_generation(use_cache):
    start = time.time()
    _ = generate_text(
        "Large language models rely on attention mechanisms",
        max_new_tokens=120,
        temperature=0.7,
        top_p=0.9,
        use_cache=use_cache
    )
    return time.time() - start

t_cache = timed_generation(True)
t_no_cache = timed_generation(False)

print("\n[KV CACHE  TEMPO]")
print(f"Com cache : {t_cache:.4f} s")
print(f"Sem cache : {t_no_cache:.4f} s")
```

4.13 Demonstração guiada: prompting e variabilidade

Nesta demonstração, fixamos o modelo e observamos como pequenas mudanças no prompt e nos controles afetam o texto gerado. O prompt não muda os pesos do modelo. Mas muda o contexto, e portanto muda os logits $\mathbf{z}_t$ e o tipo de continuação que se torna mais provável.

4.13.1 Direto vs. passo a passo

```
prompt_direct = "What is 27 + 58?"
prompt_step = "Solve step by step: 27 + 58."

print("\n=== Direto ===")
print(generate_text(prompt_direct, temperature=0.7, top_p=0.9, max_new_tokens=40))

print("\n=== Step by step ===")
print(generate_text(prompt_step, temperature=0.7, top_p=0.9, max_new_tokens=80))
```

Uma observação importante: Modelos como GPT-2 podem produzir respostas erradas em aritmética. O experimento serve para enfatizar um ponto: Prompting controla estilo e tendência de resposta, não garante correção.

Para precisão, é comum acoplar ferramentas (calculadora), validação externa ou modelos com *tool use*.

4.14 Regras de bolso antes da prática

Antes de iniciar a atividade prática da próxima seção, vale ter um guia rápido: (i) **o que cada controle faz na distribuição** e (ii) **como ajustar sem se perder**. Use estas regras como "manual de depuração" durante a prática.

Três controles, três efeitos

1. **Temperatura** controla concentração: $\tau \downarrow$ torna a amostragem mais determinística (menos diversidade); $\tau \uparrow$ aumenta diversidade (e também o risco de deriva/ruído).

2. **Top-p** controla o suporte efetivo: $p \downarrow$ reduz candidatos e tende a aumentar repetição/loops; $p \uparrow$ aumenta exploração e pode mudar o texto de direção.

3. **KV cache** controla custo computacional: não muda a distribuição nem o texto, apenas reduz latência ao evitar recomputar K, V do prefixo.

Regras de bolso para ajustar parâmetros

Na prática, τ e top-p são controles de **exploração vs. consistência**: eles não "melhoram" o modelo, apenas definem *como* amostrar $\Pr(x_{t+1} \mid x_{\leq t})$. Ajuste **um parâmetro por vez** e observe o efeito no texto.

- **Ponto de partida robusto:** $\tau \in [0.7, 1.0]$ e top-$p \in [0.9, 0.95]$.
- **Se estiver repetindo/entrando em loop:** aumente primeiro top-p (ex.: $0.90 \to 0.95$); se persistir, aumente levemente τ (ex.: $0.7 \to 0.9$).
- **Se estiver caótico/incoerente:** reduza τ (ex.: $1.0 \to 0.7$) *ou* reduza top-p (ex.: $0.95 \to 0.90$).
- **Se a tarefa exige precisão:** diminua variância (τ mais baixo, top-p moderado) e **valide externamente**; prompting melhora estilo, não garante correção.
- **Cache é engenharia:** use KV cache sempre que possível em geração token-a-token; o ganho cresce com contexto e tamanho do modelo.

4.15 Atividade prática: inferência interativa em modelos de linguagem

Esta prática consolida os controles de inferência discutidos neste capítulo (*temperatura*, *top-p* e *chain-of-thought* como heurística de estilo), por meio de experimentação empírica e análise de outputs reais.

4.15.1 Contexto da atividade

Esta prática integra o minicurso *Large Language Models e Agentes* e tem como objetivo avaliar a compreensão prática do processo de inferência em modelos de linguagem causais, com foco no impacto dos parâmetros de geração.

A atividade é baseada em **experimentação empírica**, observação de saídas reais do modelo e justificativa técnica fundamentada nos resultados obtidos.

Atenção: Esta não é uma atividade conceitual ou de definição teórica. Respostas genéricas, explicativas ou não fundamentadas em evidência empírica não atendem aos objetivos da prática.

4.15.2 Objetivos de aprendizagem

Ao final desta atividade, o estudante deverá ser capaz de:

- Relacionar parâmetros de geração (*temperatura* e *top-p*) com alterações observáveis no texto gerado;
- Interpretar o efeito desses parâmetros como modificações na distribuição de probabilidade do próximo token;
- Identificar os limites do uso de *chain-of-thought* como heurística de estilo, não como garantia de correção;
- Justificar escolhas de parâmetros com base em observação experimental.

4.15.3 Instruções gerais

1. Todas as respostas devem ser baseadas **exclusivamente** nos outputs obtidos no seu ambiente de execução.
2. Sempre que solicitado, copie trechos reais do texto gerado pelo modelo.
3. Não utilize explicações genéricas ou definições teóricas sem conexão direta com os resultados observados.
4. Utilize o mesmo modelo e código-base apresentados no minicurso.

4.15.4 Parte A: efeito da temperatura

Utilize o prompt fixo abaixo:

```
Transformers are models that
```

Execute o modelo com os seguintes parâmetros:

- Execução A1: temperatura = 0.2, top-p = 0.9
- Execução A2: temperatura = 0.7, top-p = 0.9
- Execução A3: temperatura = 1.0, top-p = 0.9

Questões

- **A.1** Copie um trecho representativo (2 a 3 linhas) de cada uma das três execuções.
- **A.2** Em qual execução você observou maior repetição lexical ou estrutural? Justifique com base no texto gerado.
- **A.3** Em qual execução houve maior variação temática ou semântica? Cite evidência textual concreta.

4.15.5 Parte B: efeito do top-p (nucleus sampling)

Fixe a temperatura em 0.7 e execute o mesmo prompt com os valores:

- top-p = 0.3
- top-p = 0.6
- top-p = 0.9

Questões

- **B.1** Em qual valor de *top-p* o modelo apresentou comportamento mais repetitivo?
- **B.2** Copie exatamente uma frase ou padrão que se repetiu ou quase se repetiu.
- **B.3** Relacione esse comportamento com a ideia de massa acumulada de probabilidade.

4.15.6 Parte C: chain-of-thought — estilo vs. correção

Utilize os dois prompts abaixo, mantendo temperatura = 0.7 e top-p = 0.9:

```
P1: What is 27 + 58?
P2: Solve step by step: 27 + 58.
```

Questões

- **C.1** A resposta numérica final está correta em ambos os casos? Responda *sim* ou *não*.
- **C.2** Copie um trecho do output onde o modelo aparenta estar raciocinando.
- **C.3** Com base exclusivamente no output obtido, o raciocínio apresentado garante correção? Justifique.

4.15.7 Parte D: escolha de parâmetros por objetivo

Considere o seguinte comportamento observado em uma geração:

> "Transformers are models that are not part of the game. The following is a list of all the models that are not part of the game..."

Questões

- **D.1** Qual parâmetro você alteraria primeiro para reduzir esse comportamento repetitivo?
- **D.2** Você aumentaria ou diminuiria esse parâmetro?
- **D.3** Justifique sua resposta com base nos experimentos realizados nas partes anteriores.

4.15.8 Critérios de avaliação

Serão considerados na avaliação:

- Uso consistente de evidência empírica (outputs reais);
- Coerência entre observação e justificativa técnica;
- Clareza na relação entre parâmetros e comportamento do modelo;
- Respostas objetivas e fundamentadas.

Respostas genéricas, conceituais ou desvinculadas dos outputs observados não atendem aos objetivos da prática.

Observação final. Esta atividade avalia sua capacidade de analisar empiricamente o comportamento de um modelo de linguagem, e não a reprodução de definições teóricas.

4.15.9 Leitura guiada da arquitetura e interpretação das saídas experimentais

4.15.10 Contexto desta seção

Aqui conectamos três camadas de entendimento que o aluno costuma ver separadas:

1. a **arquitetura real do modelo** impressa pelo PyTorch,
2. os **parâmetros de inferência** que controlamos (temperatura e top-p),
3. e os **comportamentos observados nas saídas textuais**.

O objetivo é mostrar que *nenhuma saída é arbitrária*: ela é consequência direta da arquitetura e da distribuição amostrada.

4.15.11 Confirmação do ambiente e do modelo carregado

O log inicial confirma dois pontos práticos:

- o modelo está sendo executado no dispositivo esperado (CPU ou GPU);
- o modelo carregado é o **GPT-2 base**, um decoder causal clássico.

Essa confirmação importa porque garante que estamos trabalhando com uma arquitetura causal pura, sem adaptações modernas que escondam o comportamento.

4.15.12 Leitura estrutural do GPT-2 impresso

Ao imprimir o modelo, vemos explicitamente sua decomposição interna.

- **Embeddings**:
  - `wte: Embedding(50257, 768)` — vocabulário de 50.257 tokens mapeados para vetores de 768 dimensões;
  - `wpe: Embedding(1024, 768)` — embedding posicional absoluto com contexto máximo de 1024 tokens.
- **Blocos do decoder**:
  - `ModuleList (0-11)` indica 12 blocos empilhados;
  - cada bloco contém atenção causal (máscara implícita), MLP com expansão 4x (768 $\rightarrow$ 3072 $\rightarrow$ 768) e normalizações.
- **Cabeça de linguagem**:
  - `lm_head: Linear(768 → 50257)` projeta o estado oculto final para logits sobre todo o vocabulário.

Conclusão estrutural: O GPT-2 implementa exatamente o modelo matemático:

$$\Pr(x_{t+1} \mid x_{\leq t}) = \text{softmax}(\mathbf{z}_t),$$

onde cada logit $\mathbf{z}_t$ nasce diretamente dessa arquitetura.

Pontos para lembrar

- Em modelos causais, gerar texto é **amostrar** da distribuição $\Pr(x_{t+1} \mid x_{\leq t})$, obtida via softmax sobre logits $\mathbf{z}_t$ (logit não é probabilidade).
- **Temperatura** reescala logits: $\text{softmax}(\mathbf{z}/\tau)$. $\tau \downarrow$ aumenta concentração (menos diversidade, mais "determinístico"); $\tau \uparrow$ achata a distribuição (mais diversidade, maior risco de deriva/ruído).
- **Top-p (nucleus)** recorta o **suporte efetivo** da amostragem: escolhe o menor conjunto com massa acumulada $\geq p$ e renormaliza. $p \downarrow$ reduz candidatos e tende a aumentar repetição/loops; $p \uparrow$ aumenta exploração.
- Na prática, **repetição** costuma ser sinal de **suporte amostral estreito** (top-p baixo) ou **concentração excessiva** (temperatura muito baixa); já **caos/incoerência** costuma aparecer com **temperatura alta** e/ou top-p alto.
- Prompting altera o contexto e, portanto, os logits: muda o **estilo** e a tendência de continuação, mas **não garante correção** (especialmente em aritmética e fatos). "Step by step" é um heurístico de forma, não uma prova.
- **KV cache** é decisão de engenharia: não muda a distribuição nem o texto, apenas reduz latência ao evitar recomputar K, V do prefixo; o ganho cresce com contexto e tamanho do modelo.
- Esta prática é **empírica**: a habilidade avaliada é ligar parâmetros $\rightarrow$ mudanças observáveis no output, sustentando justificativas com evidência textual concreta.

O que vem a seguir

No próximo capítulo, o foco sai dos **parâmetros de amostragem** e entra no **controle por contexto**: como o modelo pode mudar de comportamento *sem* alterar pesos, apenas pela forma como você escreve e estrutura o prompt. Vamos diferenciar **condicionamento** de **aprendizado paramétrico**, observar **zero-shot vs. few-shot** e entender o que realmente acontece em **in-context learning**. Também veremos por que o **template** funciona como regularização do espaço de saída, quando **self-consistency** ajuda (e quando não), e como o **limite de contexto** impõe restrições estruturais em tarefas longas.

Capítulo 5

Prompt Engineering e In-Context Learning

Este capítulo trata de duas ideias centrais no uso de LLMs em aplicações reais: **(i) prompt engineering** como controle do comportamento por texto e **(ii) in-context learning** como indução de padrões a partir de exemplos no próprio contexto. O ponto-chave é separar com clareza *condicionamento por contexto* de *aprendizado paramétrico*: aqui, não alteramos pesos; alteramos apenas o **contexto textual**.

5.1 Contexto da atividade

Esta atividade integra o minicurso *Large Language Models e Agentes* e demonstra, de forma empírica, controlada e reproduzível, como o comportamento de modelos de linguagem pode ser alterado exclusivamente por meio do **contexto textual**, sem qualquer modificação nos pesos internos do modelo.

Diferentemente de abordagens de ajuste fino, as variações observadas nesta prática decorrem apenas de:

- alterações no **prompt**;
- reorganização do **template**;
- exploração da **estocasticidade da inferência** (quando habilitada);
- limites estruturais do **contexto**.

As conclusões devem ser fundamentadas exclusivamente em saídas reais do modelo executado durante a atividade.

Atenção. Esta não é uma atividade conceitual. Definições teóricas desconectadas dos outputs observados não atendem aos objetivos da prática.

5.2 Objetivos de aprendizagem

Ao final desta atividade, o estudante deverá ser capaz de:

- diferenciar claramente treinamento paramétrico de condicionamento por contexto;
- explicar o mecanismo de *in-context learning* a partir de exemplos empíricos;
- analisar o papel do template na redução do espaço de respostas possíveis;
- avaliar criticamente o uso de *self-consistency*;
- reconhecer o limite de contexto como restrição estrutural do modelo.

5.3 Modelo utilizado

Para tornar visível o efeito do condicionamento textual, utiliza-se um modelo pequeno, mas explicitamente *instruction-tuned*. A escolha é didaticamente relevante porque ajuda a reduzir a ambiguidade entre:

- limitações do método (prompting/ICL);
- limitações do modelo específico.

5.3.1 Carregamento do modelo

```
import torch
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)

t5_name = "google/flan-t5-small"
t5_tokenizer = AutoTokenizer.from_pretrained(t5_name)
t5_model = AutoModelForSeq2SeqLM.from_pretrained(t5_name).to(device)
t5_model.eval()

print("[OK] Modelo instruído carregado:", t5_name)
```

5.3.2 Saída observada

```
Device: cuda
[OK] Modelo instruído carregado: google/flan-t5-small
```

Interpretação: O modelo FLAN-T5 foi treinado com exemplos explícitos de instruções no formato *tarefa* → *resposta*. Isso não altera a arquitetura fundamental do Transformer, mas altera a distribuição aprendida durante o treinamento, tornando o modelo mais sensível à estrutura do prompt.

5.4 Funções de geração

Para facilitar comparação entre prompts, começamos com geração determinística (sem amostragem). Quando o objetivo for observar variabilidade (como em *self-consistency*), habilitamos amostragem explicitamente.

5.4.1 Geração determinística (reprodutível)

```
def generate_t5(prompt):
    inputs = t5_tokenizer(prompt, return_tensors="pt").to(device)
    with torch.no_grad():
        out = t5_model.generate(
            **inputs,
            do_sample=False,
            max_new_tokens=64
        )
    return t5_tokenizer.decode(out[0], skip_special_tokens=True)
```

Observação: Ao desativar a amostragem, isolamos o efeito do **conteúdo do prompt** sobre a resposta do modelo.

5.4.2 Geração com amostragem (para variabilidade)

```
def generate_t5_sampled(prompt):
    inputs = t5_tokenizer(prompt, return_tensors="pt").to(device)
    with torch.no_grad():
        out = t5_model.generate(
            **inputs,
            do_sample=True,
            max_new_tokens=64
        )
    return t5_tokenizer.decode(out[0], skip_special_tokens=True)
```

5.5 Few-shot e In-Context Learning

5.5.1 Execução zero-shot

```
task = "Classifique o sentimento como POSITIVO ou NEGATIVO."

zero_shot = f"""{task}

Texto: "Este filme foi um desperdicio de tempo."
Sentimento:"""

print(generate_t5(zero_shot))
```

5.5.2 Saída observada

```
NEGATIVO
```

Análise: Neste cenário, não há exemplos explícitos no contexto. O modelo responde com base em associações internalizadas durante o treinamento, sem "aprender" um padrão novo no prompt. Isso torna o comportamento potencialmente frágil: pequenas mudanças na redação podem alterar o resultado.

5.5.3 Execução few-shot

```
few_shot = f"""{task}

Texto: "Adorei, foi excelente e emocionante."
Sentimento: POSITIVO

Texto: "Péssimo. Não recomendo para ninguém."
Sentimento: NEGATIVO

Texto: "Este filme foi um desperdício de tempo."
Sentimento:"""

print(generate_t5(few_shot))
```

5.5.4 Saída observada

```
NEGATIVO
```

Interpretação: Aqui, o modelo infere explicitamente o padrão (Texto, Rótulo) a partir dos exemplos fornecidos no próprio prompt. Esse fenômeno é conhecido como *in-context learning*: não há atualização de pesos, mas há condicionamento da distribuição

$$p(y \mid x, \text{contexto}).$$

Além do conteúdo, o modelo tende a reproduzir o **formato** induzido.

5.6 O papel do template

O template controla o quanto o modelo fica "livre" para continuar o texto. Na prática, template é uma forma de **restringir o espaço de saída**.

5.6.1 Template pouco restritivo

```
template_ruim = f"""{task}
"Este filme foi um desperdício de tempo." ->"""

print(generate_t5(template_ruim))
```

5.6.2 Saída observada

```
POSITIVO or NEGATIVO. "This film was a desperdício de tempo."
```

Análise: O problema aqui não é, necessariamente, semântico. O template não restringiu suficientemente o formato, então continuações diferentes continuam plausíveis. Do ponto de vista probabilístico, várias extensões têm probabilidade semelhante, e o modelo não tem "incentivo" textual para escolher apenas uma categoria.

5.6.3 Template restritivo

```
template_bom = f"""{task}

Responda APENAS com: POSITIVO ou NEGATIVO.

Texto: "Este filme foi um desperdício de tempo."
Sentimento:"""

print(generate_t5(template_bom))
```

5.6.4 Saída observada

```
NEGATIVO
```

Conclusão: Prompt engineering atua como uma forma de **regularização do espaço de saída**. Ao restringir o formato, reduz-se a entropia efetiva da distribuição gerada.

5.7 Self-consistency

Self-consistency busca explorar a variabilidade do modelo ao produzir múltiplas amostras e observar padrões de consenso. Aqui, o objetivo é evidenciar que, em cenários ambíguos, podem existir múltiplas trajetórias internas plausíveis.

5.7.1 Execução (amostragem habilitada)

```
prompt = """Classifique o sentimento como POSITIVO ou NEGATIVO.
Responda APENAS com POSITIVO ou NEGATIVO.

Texto: "O atendimento foi ótimo, mas o produto quebrou no dia seguinte."
Sentimento:"""

samples = [generate_t5_sampled(prompt) for _ in range(10)]
samples
```

5.7.2 Saída observada

```
Amostras brutas: ['Adunator "Initive an', 'REGUMINAVING DES', 'An', ...]
VOTO FINAL (self-consistency): INDETERMINADO
```

Análise: A alta variabilidade observada indica que, em cenários ambíguos, o modelo pode seguir múltiplas trajetórias internas plausíveis. Self-consistency tenta explorar esse espaço, mas não garante convergência quando não há estrutura suficiente no prompt.

Do ponto de vista estatístico, trata-se de redução de variância, não de aumento de viés correto. Além disso, o exemplo acima ilustra um risco operacional: quando o template não "ancora" suficientemente a resposta, amostragem pode produzir saídas fora do formato esperado, mesmo com instruções explícitas.

5.8 Limites de contexto

O limite de contexto define a memória de trabalho do modelo: tokens fora dessa janela simplesmente não participam do cálculo de atenção. Isso impõe restrições práticas em tarefas longas ou iterativas.

Observação de ambiente: Nesta parte, usamos o GPT-2 (decoder causal clássico) apenas para inspecionar, de forma direta, a janela máxima de contexto da configuração. Isso evita confusão entre interfaces de modelos diferentes (e não depende do comportamento do FLAN-T5 nesta atividade).

5.8.1 Carregamento mínimo do GPT-2 para inspeção de janela

```
from transformers import AutoModelForCausalLM

gpt2_name = "gpt2"
gpt2_model = AutoModelForCausalLM.from_pretrained(gpt2_name)
ctx = gpt2_model.config.n_positions
print("Janela GPT-2:", ctx)
```

5.8.2 Saída observada

```
Janela GPT-2: 1024
```

Interpretação: O limite de contexto é uma restrição estrutural: tokens fora da janela não influenciam a atenção. Em aplicações, isso aparece como uma limitação prática severa em tarefas longas, conversas extensas ou pipelines iterativos.

5.9 Na prática: conduzindo a atividade e analisando resultados

Checklist do experimento

- Fixe um modelo e um ambiente (CPU/GPU) e registre a identificação do modelo carregado.
- Compare prompts com **geração determinística** (`do_sample=False`) para isolar o efeito do contexto.
- Ao estudar variabilidade (self-consistency), habilite **amostragem** explicitamente.
- Em tarefas de rótulo curto (ex.: POSITIVO/NEGATIVO), torne o template restritivo (*"responda apenas com..."*) para reduzir formatos inesperados.

O que observar

- Mudanças de **formato** (ex.: o modelo explica, traduz, ou devolve múltiplas opções).
- Sensibilidade a pequenas alterações no prompt (fragilidade em zero-shot).
- Em few-shot, reprodução do padrão de pares (entrada, rótulo).
- Em self-consistency, dispersão das amostras e presença (ou ausência) de consenso.

Armadilhas comuns

- Concluir "aprendizado" quando houve apenas **condicionamento por contexto**.
- Usar templates abertos e interpretar como erro semântico o que é, na prática, **saída não restringida**.
- Esperar variabilidade com geração determinística (sem amostragem).
- Ignorar o limite de contexto e assumir que o modelo "lembra" de todo o histórico indefinidamente.

Síntese

Os experimentos realizados demonstram que:

- few-shot learning é condicionamento contextual, não aprendizado paramétrico;
- templates controlam a entropia e o formato da saída;
- self-consistency reduz variância, mas não cria correção;
- o contexto é um recurso finito e estrutural.

Compreender prompt engineering é compreender como distribuições probabilísticas são moldadas, restringidas e exploradas por contexto textual.

Pontos para lembrar

- Prompting altera o comportamento via **contexto textual**, sem mudar pesos: é **condicionamento**, não ajuste fino.
- **In-context learning** emerge quando exemplos no prompt induzem um padrão entrada–saída e moldam $p(y \mid x, \text{contexto})$.
- O **template** regulariza o output: restringir formato reduz a entropia efetiva e evita continuações "livres".
- **Self-consistency** depende de amostragem; pode reduzir variância, mas **não garante correção** sem estrutura suficiente no prompt.
- O **limite de contexto** é estrutural: tokens fora da janela não participam da atenção e afetam tarefas longas.

O que vem a seguir

No próximo capítulo, mudamos do **controle por contexto** para o que acontece quando, de fato, **ajustamos os pesos** do modelo. Vamos formular o pré-treinamento como minimização de entropia cruzada para estimar $\Pr(x_{t+1} \mid x_{\leq t})$ e conectar essa visão estatística às decisões de engenharia que tornam LLMs viáveis na prática. Em seguida, comparamos precisão total (FP32) e **quantização** (INT4), discutindo impacto em tempo e, principalmente, em memória, e fechamos com a ponte conceitual para métodos eficientes como **QLoRA**.

Capítulo 6

Pré-treinamento e Otimização Computacional de LLMs

Este capítulo conecta dois aspectos que frequentemente aparecem separados: **o pré-treinamento** como processo estatístico em larga escala e **a eficiência computacional** como condição prática para treinar e adaptar modelos. O objetivo é tornar explícitos, por meio de experimentos reproduzíveis, os conceitos por trás da minimização de entropia cruzada e do custo de inferência, além de introduzir por que técnicas como quantização viabilizam métodos eficientes de ajuste fino, como QLoRA.

6.1 Contexto e objetivo da atividade

Esta atividade prática integra o minicurso *Large Language Models e Agentes (MC-CD05)* e tem como objetivo explicitar, de forma empírica e reproduzível, os principais conceitos associados ao **pré-treinamento de LLMs** e às técnicas modernas de **otimização computacional**.

Em particular, o foco está em:

- compreender o papel do pré-treinamento como processo estatístico em larga escala;
- observar o impacto da precisão numérica (FP32 vs. INT4) na inferência;
- relacionar quantização com custo, memória e viabilidade de fine-tuning;
- estabelecer a conexão conceitual com métodos eficientes como QLoRA.

Todos os resultados discutidos nesta seção derivam diretamente dos outputs reais obtidos durante a execução do código apresentado.

6.2 Configuração do ambiente computacional

O primeiro bloco do código realiza a instalação das bibliotecas necessárias:

```
!pip -q install transformers accelerate bitsandbytes datasets torch
```

Essas dependências refletem o ecossistema moderno de LLMs:

- `transformers`: modelos e pipelines da Hugging Face;
- `bitsandbytes`: suporte a quantização INT8 e INT4;
- `accelerate`: abstrações de device e eficiência;
- `torch`: backend numérico e autograd.

A seguir, o dispositivo de execução é detectado:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)
```

Output observado:

```
Device:  cuda
```

A presença de GPU não altera o resultado semântico do modelo, mas impacta diretamente latência, paralelismo e custo computacional, um aspecto central quando discutimos eficiência em larga escala.

6.3 Pré-treinamento: formulação estatística

O bloco seguinte simula o núcleo matemático do pré-treinamento de um LLM. Aqui trabalhamos diretamente com logits e alvos (tokens verdadeiros) e medimos a perda por entropia cruzada.

```
logits = torch.randn(batch_size, seq_len, vocab_size)
targets = torch.randint(0, vocab_size, (batch_size, seq_len))

loss_fn = CrossEntropyLoss()
loss = loss_fn(
    logits.view(-1, vocab_size),
    targets.view(-1)
)

print("Loss simulada (pré-treinamento):", loss.item())
```

Durante o pré-treinamento real, esse cálculo ocorre trilhões de vezes sobre corpora massivos. O modelo aprende a estimar:

$$\Pr(x_{t+1} \mid x_{\leq t})$$

por meio da minimização da entropia cruzada entre logits previstos e tokens reais.

Output observado:

```
Loss simulada (pré-treinamento):  10.09
```

Análise: O valor absoluto da perda não é o foco. O objetivo é tornar explícito que pré-treinamento não envolve raciocínio simbólico, mas sim ajuste estatístico de distribuições condicionais em grande escala.

6.4 Modelo base em precisão total (FP32)

A seguir, carregamos o GPT-2 em precisão total (FP32), usando-o como referência para comparação.

```
model_fp32 = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float32
).to(device)
```

FP32 representa um cenário de alta precisão numérica, servindo como referência ao comparar custo e desempenho sob quantização.

Output observado:

```
[OK] Modelo FP32 carregado
```

Durante a geração, obtém-se:

```
Tempo FP32 (s):  0.64
```

Esse valor de tempo reflete, em particular:

- o modelo pequeno (GPT-2);
- contexto curto;
- execução em GPU.

Saída textual observada: O texto gerado apresenta estrutura coerente, mas também deriva temática, característica típica de modelos não instruídos e sem ancoragem em evidência externa.

6.5 Quantização INT4 e eficiência

Agora, configuramos quantização em INT4 via `BitsAndBytesConfig`:

```
quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True
)
```

Neste regime:

- pesos base são armazenados em INT4;
- a computação ocorre em FP16;
- aplica-se quantização em dois estágios para reduzir erro.

Output observado:

```
[OK] Modelo INT4 carregado
```

Tempo de inferência:

```
Tempo INT4 (s):  1.06
```

Neste experimento específico, o tempo INT4 é maior que FP32. Isso não invalida a quantização:

- o modelo é pequeno;
- o overhead de kernels quantizados domina;
- o ganho real surge em modelos grandes (7B+).

O benefício principal aqui é **memória**, não latência.

6.6 Comparação de memória

A estimativa de tamanho do modelo FP32 é:

```
Tamanho aproximado FP32 (MB): 474.7
```

Em INT4, o mesmo modelo teria, aproximadamente, uma redução de até 8× no armazenamento dos pesos base. Essa redução é o fator decisivo que torna viável o fine-tuning em hardware limitado.

6.7 Conexão direta com QLoRA

Os resultados observados conectam-se diretamente ao paradigma QLoRA:

- pesos base congelados e quantizados (INT4);
- adaptadores LoRA treinados em FP16;
- número reduzido de parâmetros treináveis;
- custo computacional drasticamente menor.

Sem quantização, o ajuste fino de modelos modernos seria inviável para a maioria dos grupos acadêmicos.

Síntese

Os experimentos demonstram que:

- pré-treinamento é estatística em escala industrial;
- eficiência é parte integrante do uso de LLMs em larga escala;
- quantização redefine custo, memória e acesso;
- QLoRA depende diretamente de INT4/INT8;
- prompt controla texto, mas eficiência controla quem pode treinar.

Compreender esses elementos é essencial para transitar do uso passivo de LLMs para o desenvolvimento consciente de sistemas especializados.

Pontos para lembrar

- Pré-treinamento otimiza a previsão do próximo token minimizando entropia cruzada sobre grandes corpora.
- A perda (loss) no exemplo é ilustrativa: o objetivo é explicitar o mecanismo estatístico, não o valor absoluto.
- FP32 serve como referência de precisão; o custo observado depende de modelo, contexto e hardware.
- Quantização INT4 reduz principalmente **memória**; em modelos pequenos, pode haver overhead e pior latência.
- A redução de memória (até 8× nos pesos base) é o que habilita ajuste fino em hardware limitado.
- QLoRA combina pesos base quantizados e adaptadores treináveis para tornar fine-tuning viável com baixo custo.

O que vem a seguir

No próximo capítulo, saímos da visão de pré-treinamento e eficiência e entramos no **ajuste fino** propriamente dito: vamos comparar **SFT**, **LoRA** e **QLoRA** pela lente de engenharia (*onde* o gradiente passa, *quanto* o modelo pode se mover e *em que precisão*), e mostrar como essas escolhas determinam custo de memória, estabilidade numérica e risco de sobrescrever conhecimento do modelo base.

Capítulo 7

Treinamento e Otimização de LLMs

7.1 Abertura: o problema real de engenharia

Até este ponto, trabalhamos essencialmente com **inferência**. Inferência é barata, reversível e não altera o modelo. A partir de agora, entramos no domínio em que decisões erradas custam **tempo, dinheiro e estabilidade**. Neste bloco, discutimos como ajustar um LLM grande com custo controlado, buscando preservar o comportamento do modelo base.

A resposta moderna não é "treinar tudo". É escolher:

- **onde** o gradiente passa;
- **quanto** o modelo pode se mover;
- **em que precisão** isso acontece.

7.2 Treinamento, alinhamento e avaliação como uma decisão única

Muitas descrições apresentam *treinamento*, *alinhamento* e *avaliação* como etapas separadas. Do ponto de vista prático, é mais útil enxergá-las como partes do mesmo ciclo: ajustar o comportamento do modelo, monitorar efeitos e manter o custo sob controle.

7.2.1 Treinamento: onde o gradiente passa

Ao falar em treinamento, a questão central é decidir *o que vai ser atualizado*. Em outras palavras: por onde o gradiente passa, e o que fica congelado.

Neste hands-on:

- no **SFT clássico**, o gradiente passa por *todos* os pesos;
- no **LoRA**, ele passa apenas por um subespaço de baixa dimensão;
- no **QLoRA**, ele passa por esse subespaço enquanto a base está congelada em INT4.

Essa decisão define custo de memória, estabilidade numérica e risco de *catastrophic forgetting*. Sendo assim, treinar não é "ensinar". É permitir movimento em partes específicas do espaço de parâmetros.

7.2.2 Alinhamento: qual comportamento é reforçado

Alinhamento não começa com RLHF, começa no **dataset**. Instruções como:

"You are a meteorology expert..."
"You are a casual assistant..."

já definem tom, estilo e prioridades discursivas.

Métodos como SFT, DPO, RLHF e GRPO não criam alinhamento do zero: eles ajustam preferências dentro de um espaço já determinado pelo pré-treinamento e pelo fine-tuning. Em termos práticos, alinhamento aqui significa **engenharia de preferências**.

7.2.3 Avaliação: o que você decide medir

Avaliação não é apenas uma etapa no fim: ela influencia todo o pipeline. Se você mede só fluência (ou métricas como BLEU/ROUGE), pode acabar incentivando respostas longas e que *soam* convincentes, mas erram no conteúdo.

Benchmarks mais recentes, como MMLU-Pro, AIME, GPQA e LiveBench, tentam capturar aspectos de raciocínio, mas ainda têm limitações (por exemplo, contaminação de dados e dependência de *LLM-as-a-judge*). No fim, avaliar é tomar decisões práticas, como:

- que tipo de erro é aceitável no seu contexto;
- quando o modelo deve se abster de responder;
- qual trade-off entre custo, cobertura e confiabilidade.

7.2.4 Integração: a decisão real

Agora juntamos as três faces do mesmo problema:

- **Treinamento** define onde o modelo pode mudar;
- **Alinhamento** define para onde ele tende a ir;
- **Avaliação** define se isso é aceitável.

Essas escolhas não são sequenciais; elas são simultâneas. Treinar, alinhar e avaliar é uma única decisão de engenharia, não três etapas independentes. É com essa lente que analisaremos SFT, LoRA e QLoRA no que segue.

7.3 O que significa treinar um LLM: formulação matemática

A partir daqui, usamos uma formulação explícita. Vamos descrever treinamento em termos de função objetivo, gradientes e parâmetros do modelo.

7.3.1 Modelo causal e função objetivo

Considere um modelo de linguagem causal parametrizado por um vetor de pesos θ, que define a distribuição

$$p_\theta(x_1, \ldots, x_T) \;=\; \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}).$$

Dado um conjunto supervisionado de sequências $\mathcal{D} = \{x^{(i)}\}_{i=1}^{N}$, o treinamento padrão consiste em minimizar a perda de entropia cruzada:

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D}}\left[\sum_t \log p_\theta(x_t \mid x_{<t})\right].$$

Treinar um modelo significa alterar θ de forma a reduzir $\mathcal{L}(\theta)$.

O que diferencia SFT, LoRA e QLoRA é **quais componentes de θ podem variar** e **em que precisão numérica isso ocorre**.

7.3.2 SFT clássico: otimização no espaço completo

No ajuste fino supervisionado clássico (SFT), não há restrição estrutural:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta),$$

onde θ contém *todos* os pesos do modelo e o gradiente é denso. Geometricamente, o SFT permite movimento em todo o espaço $\mathbb{R}^{|\theta|}$, o que implica custo alto de memória, maior instabilidade numérica e risco de sobrescrever conhecimento pré-treinado. SFT otimiza diretamente θ, sem qualquer restrição estrutural.

7.3.3 LoRA: restrição do espaço de atualização

LoRA altera o espaço em que o gradiente pode atuar. Considere uma matriz de pesos $W \in \mathbb{R}^{d\times k}$. Em LoRA, W é mantida congelada e introduzimos uma atualização de baixo rank:

$$W' = W + \Delta W, \quad \Delta W = BA,$$

onde

$$A \in \mathbb{R}^{r\times k}, \quad B \in \mathbb{R}^{d\times r}, \quad r \ll \min(d, k).$$

O vetor de parâmetros passa a ser

$$\theta = \{\theta_{\text{base}}, A, B\} \quad \text{com} \quad \nabla_{\theta_{\text{base}}}\mathcal{L} = 0,$$

isto é, o gradiente atua apenas sobre A e B. Geometricamente, o treinamento ocorre em um subespaço de baixa dimensão imerso no espaço original.

LoRA restringe a otimização a um subespaço linear de rank r.

7.3.4 QLoRA: separação entre armazenamento e otimização

QLoRA adiciona uma restrição no domínio numérico: o modelo base é quantizado,

$$\theta_{\text{base}}^{(q)} = Q(\theta_{\text{base}}),$$

tipicamente em INT4 (NF4). Importante:

- $\theta_{\text{base}}^{(q)}$ é usado apenas no *forward*;
- não há gradiente passando por $Q(\cdot)$.

A otimização ocorre exclusivamente sobre os adaptadores LoRA, armazenados em maior precisão:

$$W' = Q(W) + BA \quad \text{com} \quad \nabla_{A,B}\mathcal{L} \neq 0.$$

QLoRA desacopla armazenamento e otimização: quantiza o que não aprende e preserva precisão onde há gradiente.

7.3.5 Comparação unificada

Os três casos podem ser descritos em uma única expressão:

$$\theta = \begin{cases} \theta_{\text{base}} & \text{(SFT)} \\ \{\theta_{\text{base}}, A, B\},\ \nabla_{\theta_{\text{base}}} = 0 & \text{(LoRA)} \\ \{Q(\theta_{\text{base}}), A, B\},\ \nabla_{\theta_{\text{base}}} = 0 & \text{(QLoRA).} \end{cases}$$

Em prática, o que muda é *quais parâmetros podem ser atualizados* e com que precisão isso acontece.

7.4 Prova numérica: "treinar é escolher espaço, direção e precisão"

A seguir, construímos um microcaso calculável, no qual explicitamos perda, gradiente e atualização, e observamos como SFT, LoRA e QLoRA mudam o que pode e o quanto consegue mudar.

Para manter tudo analisável, usamos um "modelo" mínimo: um token de entrada, um vocabulário com 3 tokens e logits lineares $z = Wx$.

7.4.1 Setup: vocabulário, entrada e pesos

Considere um vocabulário com três classes:

$$\mathcal{V} = \{1, 2, 3\},$$

e um único exemplo supervisionado:

$$x = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \in \mathbb{R}^2, \qquad y = 1.$$

A matriz de pesos (cabeçalho linear) é:

$$W = \begin{bmatrix} 0.10 & 0.20 \\ 0.00 & -0.10 \\ -0.10 & 0.10 \end{bmatrix} \in \mathbb{R}^{3\times 2}.$$

7.4.2 Forward: logits $z = Wx$

Os logits são:

$$z = Wx = \begin{bmatrix} 0.10 \\ 0.00 \\ -0.10 \end{bmatrix}.$$

7.4.3 Softmax: probabilidades $p = \mathrm{softmax}(z)$

Defina:

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{3} e^{z_j}}.$$

Calculamos os expoentes:

$$e^{0.10} \approx 1.105170, \quad e^{0.00} = 1, \quad e^{-0.10} \approx 0.904837,$$

e o somatório:

$$S \approx 1.105170 + 1 + 0.904837 = 3.010007.$$

Então:

$$p_1 \approx \frac{1.105170}{3.010007} = 0.3672, \quad p_2 \approx \frac{1}{3.010007} = 0.3322, \quad p_3 \approx \frac{0.904837}{3.010007} = 0.3006.$$

7.4.4 Perda: cross-entropy para $y = 1$

Com um único exemplo:

$$\mathcal{L} = -\log p_y = -\log p_1.$$

Como $p_1 \approx 0.3672$,

$$\mathcal{L} \approx -\log(0.3672) \approx 1.001.$$

7.4.5 Gradiente: $\nabla_W \mathcal{L}$ (conta a conta)

Para softmax + cross-entropy:

$$\frac{\partial \mathcal{L}}{\partial z_i} = p_i - \mathbb{1}[i = y].$$

Como $y = 1$:

$$\frac{\partial \mathcal{L}}{\partial z_1} \approx -0.6328, \quad \frac{\partial \mathcal{L}}{\partial z_2} \approx 0.3322, \quad \frac{\partial \mathcal{L}}{\partial z_3} \approx 0.3006.$$

Como $z = Wx$, para cada linha i, $z_i = w_i^\top x$ e $\partial z_i / \partial w_i = x$. Logo:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial z_i} x.$$

Com $x = [1, 0]^\top$, a segunda coluna do gradiente zera:

$$\nabla_W \mathcal{L} = \begin{bmatrix} -0.6328 & 0 \\ 0.3322 & 0 \\ 0.3006 & 0 \end{bmatrix}.$$

7.4.6 Um passo de treino SFT: $W \leftarrow W - \eta \nabla_W \mathcal{L}$

Escolha:

$$\eta = 0.10.$$

Então:

$$W_{\text{novo}} = W - 0.10 \nabla_W \mathcal{L}.$$

Como a segunda coluna do gradiente é zero, apenas a primeira coluna muda:

$$w_{1,1}^{\text{novo}} = 0.16328, \quad w_{2,1}^{\text{novo}} = -0.03322, \quad w_{3,1}^{\text{novo}} = -0.13006,$$

e

$$W_{\text{novo}} = \begin{bmatrix} 0.16328 & 0.20 \\ -0.03322 & -0.10 \\ -0.13006 & 0.10 \end{bmatrix}.$$

7.4.7 Verificação numérica: a perda cai?

Recomputando:

$$z_{\text{novo}} = W_{\text{novo}} x = \begin{bmatrix} 0.16328 \\ -0.03322 \\ -0.13006 \end{bmatrix}.$$

Com:

$$e^{0.16328} \approx 1.1774, \quad e^{-0.03322} \approx 0.9673, \quad e^{-0.13006} \approx 0.8780,$$

temos:

$$S_{\text{novo}} \approx 3.0227, \quad p_{1,\text{novo}} \approx 0.3894, \quad \mathcal{L}_{\text{novo}} = -\log(0.3894) \approx 0.944.$$

Logo:

$$\mathcal{L}_{\text{novo}} \approx 0.944 \; < \; \mathcal{L} \approx 1.001.$$

7.4.8 Onde entra a "escolha do espaço"? (LoRA)

LoRA não muda a função de perda; muda o espaço de atualizações possíveis. Com rank $r = 1$:

$$\Delta W = BA, \quad B \in \mathbb{R}^{3\times 1}, \quad A \in \mathbb{R}^{1\times 2}.$$

Assim:

$$\Delta W = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \begin{bmatrix} a_1 & a_2 \end{bmatrix} = \begin{bmatrix} b_1 a_1 & b_1 a_2 \\ b_2 a_1 & b_2 a_2 \\ b_3 a_1 & b_3 a_2 \end{bmatrix}.$$

A atualização SFT com $\eta = 0.1$ foi:

$$\Delta W_{\text{SFT}} = \begin{bmatrix} 0.06328 & 0 \\ -0.03322 & 0 \\ -0.03006 & 0 \end{bmatrix}.$$

Escolha didática: $a_2 = 0$ e $a_1 = 1$, isto é,

$$A = \begin{bmatrix} 1 & 0 \end{bmatrix} \Rightarrow \Delta W = \begin{bmatrix} b_1 & 0 \\ b_2 & 0 \\ b_3 & 0 \end{bmatrix}.$$

Para igualar ΔW_{SFT}, basta definir:

$$b_1 = 0.06328, \quad b_2 = -0.03322, \quad b_3 = -0.03006.$$

Nesta instância toy, LoRA rank-1 representa exatamente o delta, porque o gradiente já era "coluna única". Em problemas reais, atualizações necessárias tendem a ser mais ricas, e ranks pequenos passam a aproximar, não reproduzir exatamente.

7.4.9 Onde entra a "escolha da precisão"? (QLoRA)

Para isolar a ideia, usamos uma quantização uniforme em INT4, com faixa simétrica $[-\alpha, +\alpha]$ e $\alpha = 0.20$. Com níveis $\{-7, \ldots, +7\}$, o passo é:

$$\Delta = \frac{\alpha}{7} \approx 0.028571.$$

Quantização por arredondamento:

$$Q(w) = \Delta \cdot \text{round}\Big(\frac{w}{\Delta}\Big).$$

Para $w = 0.16328$:

$$\frac{w}{\Delta} \approx 5.7148, \quad \text{round}(5.7148) = 6, \quad Q(w) \approx 0.171426.$$

Erro:

$$\varepsilon = Q(w) - w \approx 0.008146.$$

Em QLoRA, o peso efetivo no forward pode ser escrito como:

$$w_{\text{efetivo}} = Q(w_{\text{base}}) + \delta,$$

onde δ vem do LoRA em maior precisão. Se quisermos corrigir exatamente o erro:

$$\delta = -\varepsilon \approx -0.008146 \quad \Rightarrow \quad w_{\text{efetivo}} = w.$$

Isso explicita a ideia: quantiza-se o que não aprende e preserva-se precisão onde há gradiente.

7.5 Visão geral do pipeline moderno

Vamos construir um pipeline em três estágios:

1. **SFT** — ajuste fino supervisionado clássico;
2. **LoRA** — adaptação eficiente de baixo rank;
3. **QLoRA** — LoRA sobre modelo quantizado.

O código é essencialmente o mesmo; o que muda é a **engenharia do gradiente**.

7.6 Preparação do ambiente

```
!pip -q install --upgrade pip
!pip -q install torch transformers peft bitsandbytes datasets accelerate
```

Esse conjunto de bibliotecas representa o stack usado hoje em fine-tuning de LLMs abertos.

7.7 Modelo base e referência comportamental

Antes de treinar qualquer coisa, precisamos observar o comportamento do modelo *antes* da adaptação.

```
import torch
from datasets import Dataset
from transformers import (
    AutoTokenizer,
    AutoModelForCausalLM,
    Trainer,
    TrainingArguments,
    BitsAndBytesConfig
)
from peft import LoraConfig, get_peft_model

torch.manual_seed(42)
device = "cuda" if torch.cuda.is_available() else "cpu"
print("[INFO] device:", device)
```

```
MODEL_ID = "gpt2"
tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
tokenizer.pad_token = tokenizer.eos_token
```

```
base_model = AutoModelForCausalLM.from_pretrained(
    MODEL_ID,
    torch_dtype=torch.float16 if device == "cuda" else torch.float32,
    device_map="auto"
)
print(generate_sample(base_model, "What is El Nio?"))
```

Observação: A função `generate_sample` é assumida como definida no notebook do minicurso. Aqui, ela é usada apenas para obter uma resposta de referência.

Este é o comportamento natural do modelo. Daqui em diante, vamos aplicar ajustes controlados sobre ele.

7.8 Construção do dataset supervisionado

Criamos um dataset mínimo em que a variável principal é o **estilo da resposta**.

```
sft_data = Dataset.from_dict({
    "text": [
        "You are a meteorology expert. Explain El Nio with a scientific tone.",
        "You are a casual assistant. Explain El Nio in a friendly, informal way."
    ]
})
```

```
def tokenize_fn(batch):
    toks = tokenizer(
        batch["text"],
        padding=True,
        truncation=True,
        max_length=256
    )
    toks["labels"] = toks["input_ids"].copy()
    return toks

sft_ds = sft_data.map(tokenize_fn, batched=True, remove_columns=["text"])
```

Aqui ocorre o SFT clássico: o alvo é o próprio texto, a loss é cross-entropy causal, e todos os pesos recebem gradiente.

7.9 Estágio 1: SFT clássico e seus limites

Tentamos inicialmente treinar em FP16 e observamos um erro de escalonamento de gradiente. Treinar via SFT direto em baixa precisão pode ficar instável sem cuidados extras (*loss scaling*, otimização, batch/seq, etc.).

Executamos então o treino em FP32:

```
model_sft = AutoModelForCausalLM.from_pretrained(
    MODEL_ID,
    torch_dtype=torch.float32,
    device_map="auto"
)
model_sft.train()
```

```
args_sft = TrainingArguments(
    output_dir="./_out_sft",
    per_device_train_batch_size=1,
    max_steps=5,
    learning_rate=5e-5,
    fp16=False,
    logging_steps=1,
    report_to="none"
)

trainer_sft = Trainer(
    model=model_sft,
    args=args_sft,
    train_dataset=sft_ds
)

trainer_sft.train()
```

Interpretação do output: A perda cai rapidamente, o que mostra que o modelo consegue se ajustar com facilidade. O outro lado disso é que, ao atualizar muitos pesos de uma vez, também fica fácil alterar (ou degradar) comportamentos já aprendidos.

Observação: O SFT direto costuma funcionar, mas tende a ser mais caro (em memória/-tempo) e menos controlável do que abordagens modernas de ajuste fino eficiente.

7.10 Estágio 2: LoRA como adaptação controlada

Agora congelamos o modelo base e treinamos apenas adaptadores de baixo rank.

```
lora_cfg = LoraConfig(
    r=8,
```

```
    lora_alpha=16,
    lora_dropout=0.1,
    target_modules=["c_attn", "c_proj"],
    bias="none",
    task_type="CAUSAL_LM"
)
```

```
model_lora = get_peft_model(base_model, lora_cfg)
model_lora.print_trainable_parameters()
```

Interpretação do output: Menos de 1% dos parâmetros ficam treináveis, porque mantemos os pesos originais congelados e aprendemos apenas matrizes adicionais de baixa dimensão (os adaptadores). Na prática, isso permite ajustar o comportamento do modelo com bem menos custo, com menor risco de "mexer em tudo" no modelo base.

7.11 Estágio 3: QLoRA com base quantizada

No estágio final, reduzimos drasticamente o custo de memória quantizando a base e mantendo o gradiente apenas nos adaptadores.

```
bnb_cfg = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16
)
```

```
model_q = AutoModelForCausalLM.from_pretrained(
    MODEL_ID,
    quantization_config=bnb_cfg,
    device_map="auto"
)
model_qlora = get_peft_model(model_q, lora_cfg)
```

A base fica congelada (quantizada em INT4) e o gradiente flui apenas pelos adaptadores.

Observação: A ideia do QLoRA é reduzir memória/custo usando quantização no modelo base, sem abrir mão de adaptação: quem "aprende" de fato são os adaptadores, não os pesos quantizados.

Síntese

Técnica	O que muda	Custo
SFT	Todos os pesos	Muito alto
LoRA	Subespaço	Baixo
QLoRA	Subespaço + INT4	Muito baixo

Fine-tuning moderno não é escolher técnica. É decidir onde, quanto e em que precisão mexer nos pesos.

7.12 Na prática: como conduzir o pipeline com transparência

Checklist do experimento

- Defina uma **referência** do comportamento do modelo antes de qualquer treino (mesmo prompt, mesma função de geração).
- Registre a **precisão** utilizada em cada estágio (FP16/FP32/INT4) e o que, de fato, recebe gradiente.
- No SFT, verifique explicitamente que o treino ocorre no **espaço completo** (todos os pesos treináveis).
- No LoRA/QLoRA, confirme a fração de parâmetros treináveis (por exemplo, via `print_trainable_parameters`).

O que observar

- Mudanças de **estilo** no output (o dataset foi construído para isso).
- Estabilidade numérica: sinais de instabilidade ao tentar SFT em baixa precisão.
- Diferenças entre "mexer em tudo" (SFT) e "mexer pouco" (LoRA/QLoRA) no comportamento resultante.

Armadilhas comuns

- Tratar "treinamento" como uma caixa-preta e não explicitar onde o gradiente flui.
- Confundir ganho de memória (INT4) com ganho de latência em qualquer regime.
- Avaliar sem definir antes o que conta como melhoria (o que você mede determina o que você otimiza).

Pontos para lembrar

- O problema central do fine-tuning moderno é de engenharia: adaptar com custo controlado e sem degradar o modelo base.
- Treinar, alinhar e avaliar formam uma decisão única: *onde* o modelo muda, *para onde* tende e *como* você mede aceitabilidade.

- SFT atualiza todos os pesos: é flexível, mas caro e com maior risco de sobrescrever conhecimento prévio.
- LoRA restringe ΔW a uma forma de baixo rank ($\Delta W = BA$), reduzindo drasticamente parâmetros treináveis.
- QLoRA quantiza a base (INT4/NF4) e mantém aprendizado nos adaptadores em maior precisão, desacoplando armazenamento e otimização.
- A mensagem operacional é precisa: treinar é escolher **espaço**, **direção** e **precisão** do movimento no espaço de parâmetros.

O que vem a seguir

No próximo capítulo, passamos de **como adaptar pesos** para **o que otimizar**. Introduzimos alinhamento por preferências (como **DPO**) e a intuição de RLHF/GRPO em regime didático, mostrando como sinais de preferência e recompensa *reorganizam* a distribuição do modelo sem necessariamente adicionar capacidade nova. Em seguida, discutimos por que **avaliar** LLMs virou parte do problema: benchmarks modernos, limites de *LLM-as-a-judge* e riscos como contaminação podem distorcer o que o treinamento parece "melhorar".

Capítulo 8

Alinhamento de Preferências e Avaliação Moderna em LLMs

8.1 Contexto e motivação

LLMs podem gerar texto com alta fluência, mas o treinamento padrão (baseado em maximizar a probabilidade do próximo token) não garante, por si só, respostas *úteis*, *seguras* ou *alinhadas às preferências humanas*.

Isso motivou técnicas de **alinhamento por preferências**, nas quais o objetivo deixa de ser apenas "prever tokens" e passa a favorecer certas respostas em detrimento de outras. Em paralelo, avaliar esses modelos tornou-se mais difícil: benchmarks mais desafiadores (como MMLU-Pro, AIME, GPQA e LiveBench) convivem com abordagens baseadas em julgamento por LLMs (*LLM-as-a-judge*), úteis em análises rápidas, mas com limitações importantes.

Neste capítulo, cobrimos dois blocos: (i) métodos de alinhamento por preferências (DPO e a intuição de RLHF/GRPO) e (ii) avaliação moderna, incluindo riscos de contaminação e fragilidades do uso de juízes.

8.2 Alinhamento por preferências diretas: DPO

8.2.1 Ideia central

O **Direct Preference Optimization (DPO)** substitui o pipeline clássico de RLHF (*reward model* + otimização por reforço) por uma formulação direta baseada em pares de preferência.

Cada exemplo do dataset contém:

- um `prompt`;
- uma resposta `chosen` (preferida);
- uma resposta `rejected` (não preferida).

O objetivo é ajustar a política π_θ para aumentar a probabilidade da resposta `chosen` relativamente à `rejected`, mantendo a política próxima de uma referência π_{ref}. Em termos práticos, o DPO **reorganiza preferências** no espaço de saída do modelo, sem criar capacidade do zero.

8.2.2 A ideia por trás das equações

Sem entrar nos detalhes do derivação, a ideia pode ser entendida assim: para um mesmo `prompt`, queremos que

$$\log \pi_\theta(\texttt{chosen} \mid \texttt{prompt}) \; - \; \log \pi_\theta(\texttt{rejected} \mid \texttt{prompt})$$

aumente, mas sem deixar π_θ "escapar" muito da referência. Esse termo de "ancoragem" é importante porque, sem ele, um modelo pode supervalorizar padrões do dataset de preferências e degradar comportamento geral.

8.2.3 O que o DPO altera na prática

Após o DPO, é comum observar mudanças estruturais, mesmo quando o modelo é pequeno:

- a distribuição de tokens passa a refletir preferências do dataset;
- padrões associados às respostas rejeitadas tendem a ser desincentivados;
- respostas excessivamente longas ou evasivas podem ficar menos prováveis.

Em outras palavras, o DPO não "injeta conhecimento" por si só: ele ajusta **quais respostas** o modelo tende a priorizar, dentro da capacidade que o pré-treinamento já forneceu.

8.3 RLHF e GRPO: intuição operacional

8.3.1 RLHF em uma frase

O RLHF clássico usa aprendizado por reforço para maximizar uma recompensa associada à qualidade de respostas. Em configurações tradicionais, essa recompensa vem de um *reward model* treinado a partir de comparações humanas (ou preferências).

A intuição é simples: se uma resposta recebe recompensa maior, o treino deve aumentar sua probabilidade no futuro; se recebe recompensa menor, o treino deve reduzi-la.

8.3.2 Por que isso fica instável em escala

Em modelos grandes, RL é sensível a:

- alta variância do gradiente (amostragem + recompensas ruidosas);
- mudanças muito agressivas na política;
- custo elevado de gerar muitas amostras e avaliar recompensas.

Por isso, na prática, pipelines modernos usam objetivos com **clipping**, regularização (por exemplo, penalização por divergência em relação a uma referência) e baselines para reduzir variância.

8.3.3 GRPO: o que a ideia tenta melhorar

O **GRPO** (*Group Relative Policy Optimization*) é uma variação moderna no espírito de "otimização por reforço com mais controle". A ideia geral é comparar **grupos** de respostas e usar referências relativas para reduzir sensibilidade a escalas absolutas de recompensa.

Em termos práticos, isso tende a:

- reduzir variância (comparações relativas são mais estáveis);
- melhorar eficiência de amostragem em algumas configurações;
- manter a política melhor condicionada durante o treino.

Observação. Existem diferentes formulações e regimes (on-policy e off-policy); o ponto aqui é a intuição: usar comparações relativas e objetivos "com trava" para estabilizar o aprendizado.

8.3.4 Mini-experimento mental para fixar a intuição

Um jeito de internalizar RL é pensar em uma política discreta muito simples: uma distribuição sobre algumas ações. A cada passo:

- amostramos uma ação;
- recebemos uma recompensa;
- ajustamos a política para favorecer ações com maior recompensa.

Mesmo sendo um cenário reduzido, ele captura o núcleo do que acontece em LLMs: "ações" são escolhas de tokens (ou sequências), e o sinal de recompensa molda a distribuição de saída.

8.4 Avaliação moderna e LLM-as-a-judge

8.4.1 Por que a avaliação ficou difícil

Benchmarks tradicionais saturaram rapidamente, o que motivou conjuntos mais difíceis (como MMLU-Pro, AIME, GPQA e LiveBench). Além disso, tarefas que medem raciocínio e solução de problemas muitas vezes não têm avaliação automática simples, o que incentivou o uso de LLMs como avaliadores.

Esse cenário cria um efeito colateral: **avaliar virou parte do problema** — e não apenas uma etapa de medição.

8.4.2 Falha 1: fluência versus factualidade

Um risco comum em avaliação subjetiva é confundir:

- uma resposta fluente e bem escrita
- com uma resposta correta e verificável.

Em especial, quando a tarefa envolve fatos, matemática ou lógica, a avaliação precisa separar *forma* de *conteúdo*. Caso contrário, o sistema incentiva modelos a "soar bem" mesmo quando erram.

8.4.3 Falha 2: contaminação (leakage)

Contaminação ocorre quando informações do gabarito (ou itens do benchmark) circulam no processo de treinamento, ajuste fino, prompts ou avaliação. Isso pode inflar desempenho sem refletir capacidade real.

Mesmo sem "trapaça intencional", benchmarks públicos tendem a perder valor ao longo do tempo: à medida que itens e respostas circulam, contaminação passa a ser um risco estrutural.

8.4.4 Limitações estruturais do LLM-as-a-judge

Na prática, *LLM-as-a-judge* é:

- útil para análises rápidas e comparações exploratórias;
- inadequado como verificador absoluto;
- sensível ao prompt, ao formato e ao contexto fornecido.

Sempre que possível, avaliações objetivas e verificáveis devem ser priorizadas: execução de código, checagem automática, gabaritos fechados ou critérios computáveis.

8.5 Como interpretar experimentos de alinhamento e avaliação

Nesta seção, o objetivo é aprender a interpretar um experimento típico: quais sinais procurar, o que esperar que mude e quais conclusões **não** tirar. Os detalhes (modelo, dataset, hardware e logs) variam, mas a interpretação geral é estável.

8.5.1 Antes e depois: o que comparar

Sempre compare saídas **antes e depois** do treinamento com o mesmo conjunto de prompts. Em modelos pequenos, ganhos semânticos podem ser limitados; por isso, observe também:

- comprimento e evasividade;
- repetição e padrões degenerados;
- mudanças consistentes no estilo e na estrutura;
- sensibilidade a variações pequenas no prompt.

8.5.2 O que um log de DPO costuma indicar

No DPO, o treino busca aumentar a probabilidade de respostas `chosen` relativamente às `rejected`, mantendo proximidade com uma referência. Um sinal positivo é quando o modelo passa a distinguir preferências de forma consistente (redistribuindo probabilidade entre alternativas).

Isso não implica, automaticamente:

- melhora factual,
- aumento de conhecimento,
- capacidade nova de raciocínio.

O método está ajustando **preferências** no espaço de saída, não reescrevendo o pré-treinamento.

8.5.3 Mini-RL: o que observar

Em simulações simples de RL, é comum ver concentração rápida de probabilidade em ações recompensadas. Interprete isso como:

- recompensa moldando a distribuição;
- baselines reduzindo variância;
- risco de colapso para uma ação quando o sinal é forte e o espaço é pequeno.

Em modelos grandes, o desafio é manter esse processo estável e eficiente.

8.5.4 LLM-as-a-judge: sinais de fragilidade

Ao usar um LLM como avaliador:

- registre o prompt do juiz;
- controle o formato de entrada e saída;
- teste robustez com pequenas variações (prompt e ordem de informações);
- evite qualquer vazamento de gabarito no contexto.

Se uma métrica muda muito com pequenas alterações de prompt, ela é um sinal fraco para comparar modelos.

8.6 Na prática: checklist mínimo para experimentos de alinhamento e avaliação

Checklist do experimento

- Defina claramente o que é `prompt`, `chosen` e `rejected` no dataset de preferências.
- Mantenha um **modelo de referência** congelado quando o método exigir ancoragem.
- Compare saídas **antes e depois** do treinamento com o mesmo conjunto de prompts.
- Se usar LLM-as-a-judge, registre o prompt do avaliador e mantenha o formato controlado.

O que observar

- Mudanças estruturais (comprimento, evasividade, padrões) mesmo sem ganho semântico claro.
- Sinais de colapso de política em mini-RL (concentração excessiva) e o papel do baseline.
- Sensibilidade do juiz: pequenas mudanças no prompt podem alterar a decisão.

Armadilhas comuns

- Interpretar alinhamento como "criação de conhecimento" em vez de redistribuição de preferências.
- Confiar em LLM-as-a-judge como verificador absoluto.
- Permitir contaminação (leakage) e ainda assim comparar scores como se fossem confiáveis.

Pontos para lembrar

- O treino padrão por previsão do próximo token não garante utilidade, segurança ou alinhamento a preferências humanas.
- DPO alinha preferências diretamente com pares `chosen`/`rejected`, sem *reward model* explícito.
- O modelo de referência atua como âncora: o objetivo é ajustar preferências, não "criar capacidade" do zero.
- RL (e ideias como GRPO) molda distribuições por sinais de recompensa; baselines e objetivos com controle ajudam a estabilizar o processo.
- Avaliação moderna é difícil: juízes podem confundir fluência com correção e são sensíveis ao prompt.
- Contaminação (leakage) invalida métricas; quando possível, prefira avaliações objetivas e verificáveis.

O que vem a seguir

No próximo capítulo, ampliamos a lente: saímos do **alinhamento e da avaliação** como técnicas isoladas e olhamos para o **ecossistema** que se forma ao redor de LLMs abertos. Discutimos por que interoperabilidade (interfaces, componentes e execução eficiente), governança (documentação, versionamento e avaliação como instrumento) e impacto científico passam a ser tão importantes quanto o próprio treinamento.

Capítulo 9

Tendências e desafios em LLMs abertos

Este capítulo fecha o livro olhando para além do próximo modelo e do ciclo de novidades. A questão aqui é de ecossistema: como LLMs abertos passam a conviver entre si, como seu uso pode ser governável e qual é o impacto científico real dessa tecnologia.

9.1 Interoperabilidade

Durante muito tempo, modelos de linguagem foram sistemas isolados e monolíticos, difíceis de integrar e de substituir. Esse cenário mudou de forma marcante entre 2023 e 2025. Hoje, parte relevante do estado da arte não está apenas em treinar modelos maiores, mas em fazê-los *conversar* com outros componentes e com outros modelos.

Interoperabilidade, neste contexto, pode ser entendida em três frentes.

9.1.1 Padrões de interface

APIs compatíveis, formatos de prompt mais estáveis e convenções claras de entrada e saída permitem trocar o modelo sem reescrever o sistema. Isso muda a prática: um pipeline pode rodar com um modelo hoje e com outro amanhã, preservando a lógica central do experimento.

9.1.2 Interoperabilidade de componentes

O modelo deixou de ser o sistema inteiro. Em pipelines modernos, ele se conecta a mecanismos de busca, rerankers, verificadores, bases científicas e ferramentas externas. Isso é particularmente visível em sistemas de RAG, agentes e Document AI: o modelo atua como um *módulo cognitivo* dentro de um sistema maior.

9.1.3 Interoperabilidade computacional

Quantização, execução distribuída, *sharding* e suporte a múltiplos backends tornaram o uso mais viável em ambientes com recursos limitados. FP8, INT4 e execução eficiente em GPUs menores reduziram barreiras e ampliaram o acesso, inclusive em laboratórios acadêmicos.

9.2 Governança

Governança, aqui, não significa censura. Significa previsibilidade institucional: saber o que um modelo faz, como ele muda ao longo do tempo e como lidar com efeitos conhecidos.

Com LLMs abertos ganhando escala, surgem perguntas difíceis: quem é responsável pelo modelo, quem responde por usos indevidos, como tratar vieses conhecidos e como versionar modelos que evoluem. Entre 2024 e 2025, a literatura passou a tratar governança também como um problema técnico, não apenas jurídico.

9.2.1 Documentação e transparência

Model cards, data cards, relatórios de treinamento, descrição de datasets e métricas padronizadas são essenciais para uso científico. Não basta afirmar que um modelo funciona: é necessário explicitar *como*, *com quais dados* e *com quais limitações*.

9.2.2 Avaliação como instrumento de governança

Benchmarks modernos (por exemplo, MMLU-Pro, AIME, GPQA e LiveBench) não existem apenas para ranquear modelos. Eles também expõem fragilidades, revelam contaminação e mostram quando um modelo parece "ir bem" por razões espúrias. Se a avaliação não é confiável, a governança colapsa, e é nesse ponto que a crítica ao *LLM-as-a-judge* ganha peso.

9.2.3 Alinhamento como política técnica

DPO, RLHF, GRPO e métodos recentes não são apenas técnicas de otimização. Eles codificam preferências: o que o modelo deve priorizar? Concisão, segurança, verificabilidade, raciocínio explícito? Nesse sentido, parte da governança passa a acontecer *dentro do gradiente*.

9.3 Impacto científico

LLMs abertos já alteraram práticas de pesquisa, não por substituírem cientistas, mas por reduzirem o custo cognitivo de tarefas comuns: explorar literatura em escala, comparar hipóteses mais rapidamente, estruturar dados não tabulares e gerar protótipos conceituais com maior velocidade.

Esse ganho vem com riscos. O principal é a ilusão de compreensão: texto fluente e convincente pode esconder erros conceituais profundos. Por isso, a literatura recente insiste em evidência, proveniência, verificação e abstenção.

Em sistemas científicos, não basta "responder". É necessário mostrar de onde veio a resposta, quando o sistema não sabe e por que confia em um resultado. Isso ajuda a explicar a adoção de arquiteturas híbridas: LLM com RAG, LLM com verificação, LLM com simulação. O modelo deixa de ser uma fonte de verdade e passa a atuar como mediador cognitivo.

9.4 Um desafio aberto

O futuro de LLMs abertos não depende apenas de mais parâmetros. Depende de interoperar bem, ser governável e produzir impacto científico real. O avanço em LLMs não é apenas algorítmico: ele é sistêmico, institucional e científico. É nesse espaço entre modelo, sistema e ciência, que os próximos anos tendem a ser decididos.

Pontos para lembrar

- O progresso em LLMs abertos envolve ecossistema: interoperabilidade, governança e impacto científico.
- Interoperabilidade inclui padrões de interface, integração por componentes e viabilidade computacional (quantização/execução eficiente).
- Governança é previsibilidade institucional: documentação, versionamento, responsabilidades e práticas técnicas.
- Avaliação funciona como mecanismo de pressão e diagnóstico, mas pode falhar; *LLM-as-a-judge* não é verificador absoluto.
- Alinhamento codifica preferências técnicas (o que priorizar) e influencia diretamente como o modelo se comporta.
- Em ciência, fluência não basta: evidência, proveniência, verificação e abstenção são requisitos estruturais.

Encerramento

Encerramos aqui o percurso do minicurso: do núcleo do Transformer às decisões de inferência, fine-tuning, alinhamento e avaliação. As referências ao final reúnem os trabalhos utilizados como base ao longo do texto.

Referências Bibliográficas

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020. doi: 10.48550/arXiv.2005.14165. URL `https://arxiv.org/abs/2005.14165`.

[2] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024. URL `https://arxiv.org/abs/2406.11931`.

[3] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In *Advances in Neural Information Processing Systems*, 2023. URL `https://arxiv.org/abs/2305.14314`.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019. doi: 10.18653/v1/N19-1423. URL `https://aclanthology.org/N19-1423/`.

[5] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[6] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL `https://arxiv.org/abs/2401.04088`.

[7] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

[8] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020. URL https://arxiv.org/abs/2006.16668.

[9] M. Mahato, A. Kumar, K. Singh, B. Kukreja, and J. Nabi. Red teaming for multimodal large language models: A survey. 2024. doi: 10.36227/techrxiv.170629758.87975697/v1.

[10] Youssef Mroueh, Nicolas Dupuis, Brian Belgodere, Apoorva Nitsure, Mattia Rigotti, Kristjan Greenewald, Jiri Navratil, Jerret Ross, and Jesus Rios. Revisiting group relative policy optimization: Insights into on-policy and off-policy training, 2025. URL https://arxiv.org/abs/2505.22257.

[11] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *ACM Transactions on Intelligent Systems and Technology*, 16(5), August 2025. doi: 10.1145/3744746. URL https://doi.org/10.1145/3744746.

[12] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NeurIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[13] Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation, 2022. URL https://arxiv.org/abs/2108.12409.

[14] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. Technical report, OpenAI, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[15] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017. URL https://arxiv.org/abs/1701.06538.

[16] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568: 127063, February 2024. doi: 10.1016/j.neucom.2023.127063. URL https://doi.org/10.1016/j.neucom.2023.127063.

[17] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL `https://arxiv.org/abs/2302.13971`.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[19] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report, 2024. URL `https://arxiv.org/abs/2407.10671`.

[20] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2025. URL `https://arxiv.org/abs/2303.18223`.

[21] Changhai Zhou, Shijie Han, Yuhua Zhou, Shiyang Zhang, Qian Qiao, Andrei Simion, and Weizhong Zhang. Efficient fine-tuning of quantized LLMs via three-stage optimization, 2025. URL `https://openreview.net/forum?id=zcx6rIMbbR`.